Algorithmics and Programming II: Introduction

Jordi Cortadella and Jordi Petit
Department of Computer Science

Algorithmics and Programming II

- Lecturers:
  - Jordi Cortadella (jordi.cortadella@upc.edu)
  - Jordi Petit (jordi.petit-silvestre@cs.upc.edu)
  - Pau Fernández (pau.fernandez@upc.edu)
- Teaching Assistant:
  - Laia Mogas (laia.mogas.pladevall@estudiantat.upc.edu)
- Sessions:
  - Theory (Jordi Cortadella)
  - Lab (1.1-1.2: Jordi Petit; 1.3: Pau Fernández)

Material

- Slides, exercises:

  https://www.cs.upc.edu/~jordicf/Teaching/AP2
- Jutge (for lab sessions):

  https://jutge.org
- Lliçons (by J. Petit and S. Roura):

  https://lliçons.jutge.org

Slides of the lectures

Algorithmics and Programming II

This material is used for the course Algorithmics and Programming II of the Degree Data Science and Engineering.

The lectures of the course are combined with practical programming sessions using a virtual learning environment for computer programming (Jutge.org).

Lectures

1. Introduction	
2. Abstract Data Types (I)	
3. Abstract Data Types (II)	
4. Algorithm Analysis (I)	

1. Introduction	
2. Abstract Data Types (I)	
3. Abstract Data Types (II)	
4. Algorithm Analysis (I)	
5. Algorithm Analysis (II)	
6. Divide and Conquer (I)	
7. Divide and Conquer (II)	
8. Exception handling	
9. Iterating over Data	
10. Linear Containers	
11. Priority Queues	
12. Graphs: Connectivity	
13. Graphs: Shortest Paths	
14. Graphs: A* search	
15. Graphs: Minimum Spanning Trees	
16. Graphs: Maximum Flows	
17. Graph Problem Solving	
18. Trees	
19. Binary Search Trees	
20. Hashing	
21. Cryptography	

Evaluation

- Evaluation items:
  - Projects: P1, P2
  - Exams: Partial Lab (PL), Final Theory (FT), Final Lab (FL)
- Grading:
  - Projects: $P = (P1 + P2)/2$
  - Exams: $X = \max \begin{cases} 0.3\,PL + 0.4\,FT + 0.3\,FL \\ 0.5\,FT + 0.5\,FL \end{cases}$
  - Final grade:

$$N = \begin{cases} \max(0.3\,P + 0.7\,X, 0.15\,P + 0.85\,X) & \text{if } X \geq 4 \\ 0.15\,P + 0.85\,X & \text{if } X < 4 \end{cases}$$

First project: Cities and Roads

Inspired by the game Catan™.

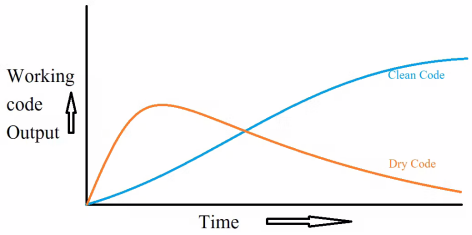
Peer and self assessment

- The project will be evaluated by the students themselves.
- Each project will be evaluated by three students. The grade will be calculated as the average grade given by the students.
- The evaluation will be completely blind.
- Biased evaluations will be detected and penalized.
- Each student will have the right to request the evaluation by the professor (who can upgrade or downgrade the evaluation given by the students).

Donald Knuth (Turing award, 1974)

- "Programming is an art of telling another human what one wants the computer to do."
- "An algorithm must be seen to be believed."
- "The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

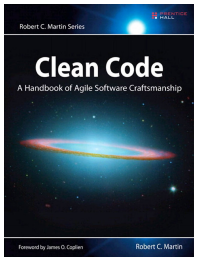
Clean Code Vs Quick Code



Clean Code: Good Practices P1

Must for juniors and seniors to know about "Clean Code"

Rahul Singh · Feb 1, 2023



"Truth can only be found in one place: the code."

"Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...[Therefore,] making it easy to read makes it easier to write."

"It is not enough for code to work."

"So if you want to go fast, if you want to get done quickly, if you want your code to be easy to write, make it easy to read."

"You should name a variable using the same care with which you name a first-born child."

"Clean code always looks like it was written by someone who cares."

"Of course bad code can be cleaned up. But it's very expensive."

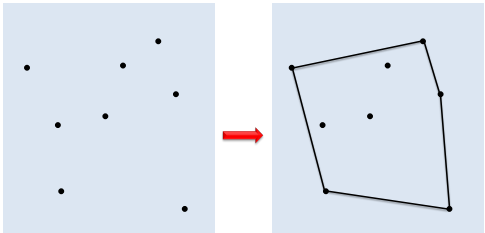
Second project: inferring graphs from GPS routes

Objective of the course

Confronting large and difficult problems. How?

- Skills for abstraction and algorithmic reasoning.
- Design and use of complex data structures.
- Techniques for complexity analysis.
- Methodologies for modular programming.
- High-quality code.

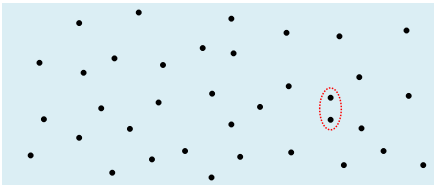
Problems on polygons



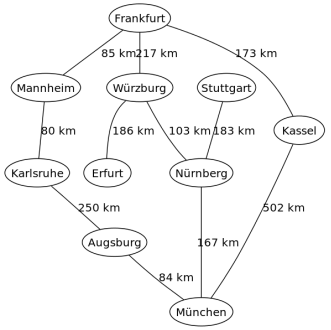
Compute the convex hull of n given points in the plane.

The Closest-Points problem

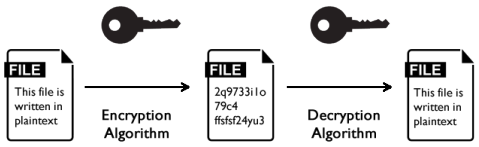
- **Input:** A list of n points in the plane $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs → $O(n^2)$
- We want an $O(n \log n)$ solution !



Navigation: find the shortest path



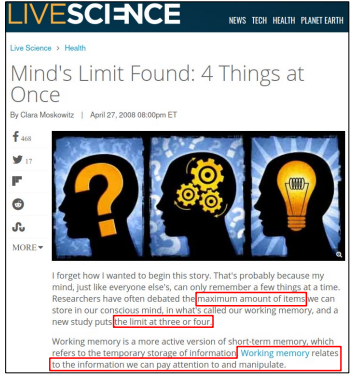
How to encrypt messages?



Abstract Data Types (I)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

How many horses can you distinguish?



LIVESCIENCE NEWS TECH HEALTH PLANET EARTH

Live Science › Health

Mind's Limit Found: 4 Things at Once

By Clara Moskowitz | April 27, 2008 08:00pm ET

I forget how I wanted to begin this story. That's probably because my mind, just like everyone else's, can only remember a few things at a time. Researchers have often debated the maximum amount of items we can store in our conscious mind, in what's called our working memory, and a new study puts the limit at three or four.

Working memory is a more active version of short-term memory, which refers to the temporary storage of information. Working memory relates to the information we can pay attention to and manipulate.

Two examples

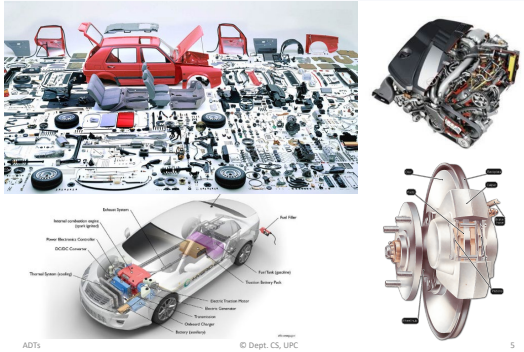
```
# Main loop of binary search
while left <= right:
    i = (left + right)/2
    if x < A[i]: right = i-1
    elif x > A[i]: left = i+1
    else: return i
```

Variables used (5):
A, x, left, right, i
(only 3 modified)

```
# Main loop of insertion sort
for i in range(1, len(A)):
    x = A[i]
    j = i
    while j > 0 and A[j-1] > x:
        A[j] = A[j-1]
        j -= 1
    A[j] = x
```

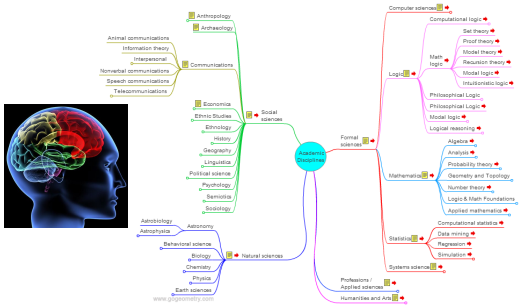
Variables used (4):
A, x, i, j

Hiding details: abstractions



Different types of abstractions

Concept maps are hierarchical: why?



Each level has few items

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

How data flows through system

Boolean logic gates and functions

Combining devices to do useful work

Transistors and wires

Silicon process technology

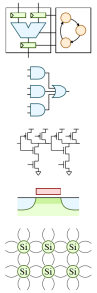


Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Mac OS X, Windows, Linux
Handles low-level hardware management

MIPS32 Instruction Set
Instructions that machine executes

```
blez  $a2, done
move  $a7, $zero
li    $t4, 99
move  $a4, $a1
move  $v1, $zero
li    $a3, 99
lw    $a5, 0($a4)
addiu $a4, $a4, 4
slt   $a6, $a5, $a3
movn  $v0, $v1, $a6
addiu $v1, $v1, 1
movn  $a3, $a5, $a6
```

Image Credit: Christopher Batten, Cornell University

The computer systems stack

- Application
- Algorithm
- Programming Language
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Register-Transfer Level
- Gate Level
- Circuits
- Devices
- Technology

Sort an array of numbers
2,6,3,8,4,5 -> 2,3,4,5,6,8

Insertion sort algorithm
1. Find minimum number in input array
2. Move minimum number into output array
3. Repeat steps 1 and 2 until finished

C implementation of insertion sort

```
void isort( int b[], int a[], int n ) {
  for ( int idx, k = 0; k < n; k++ ) {
    int min = 100;
    for ( int i = 0; i < n; i++ ) {
      if ( a[i] < min ) {
        min = a[i];
        idx = i;
      }
    }
    b[k]   = min;
    a[idx] = 100;
  }
}
```

Image Credit: Christopher Batten, Cornell University

Our challenge

- We need to design large systems and reason about complex algorithms.
- Our working memory can only manipulate 4 things at once.
- We need to interact with computers using programming languages.
- Solution: abstraction
  - Abstract reasoning.
  - Programming languages that support abstraction.
- We already use a certain level of abstraction: functions. But it is not sufficient. We need much more.

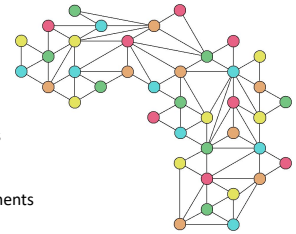
Data types

- Programming languages have a set of primitive data types (e.g., int, bool, float, str, …).
- Each data type has a set of associated operations:
  - We can add two integers.
  - We can concatenate two strings.
  - We can divide two floats.
  - But we cannot divide two strings!
- Programmers can add new operations to the primitive data types:
  - gcd(a,b), match(string1, string2), …
- The programming languages provide primitives to group data items and create structured collections of data:
  - C: array, struct.
  - Python: list, tuple, dictionary.

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them

Operations:
- Number of vertices
- Number of edges
- Shortest path
- Connected components



Data type: Graph

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them:

$$P(x) = x^3 - 4x^2 + 5$$

Data type: Polynomial

Operations:
- $P + Q$
- $P \times Q$
- P/Q
- $\gcd(P, Q)$
- $P(x)$
- $\text{degree}(P)$

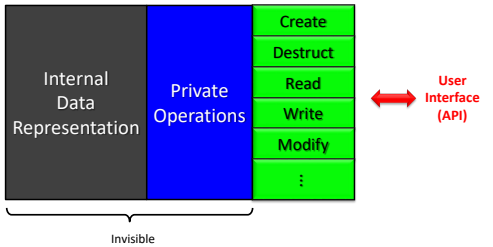
Abstract Data Types (ADTs)

- Separate the notions of specification and implementation:
  - Specification: "what does an operation do?"
  - Implementation: "how is it done?"
- Benefits:
  - Simplicity: code is easier to understand
  - Encapsulation: details are hidden
  - Modularity: an ADT can be changed without modifying the programs that use it
  - Reuse: it can be used by other programs

Abstract Data Types (ADTs)

- An ADT has two parts:
  - Public or external: abstract view of the data and operations (methods) that the user can use.
  - Private or internal: the actual implementation of the data structures and operations.
- Operations:
  - Creation/Destruction
  - Access
  - Modification

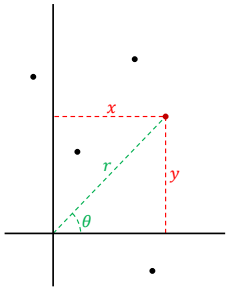
Abstract Data Types (ADTs)



API: Application Programming Interface

Example: a Point

- A point can be represented by two coordinates (x,y).
- Several operations can be envisioned:
  - Get the x and y coordinates.
  - Calculate distance between two points.
  - Calculate polar coordinates.
  - Move the point by $(\Delta x, \Delta y)$.



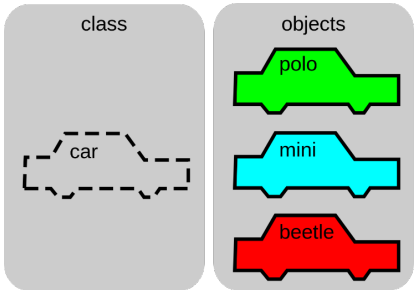
Example: a Point

```
# Things that we can do with points

p1 = Point(5.0, -3.2) # Create a point (a variable)
p2 = Point(2.8, 0)    # Create another point

# We now calculate the distance between p1 and p2
dist12 = p1.distance(p2)

# Distance to the origin
r = p1.distance()

# Create another point by adding coordinates
p3 = p1 + p2

# We get the coordinates of the new point
x = p3.x()  # x = 7.8
y = p3.y()  # y = -3.2
```


ADTs and Object-Oriented Programming

- OOP is a programming paradigm: a program is a set of objects that interact with each other.
- An object has:
  - fields (or attributes) that contain data
  - functions (or methods) that contain code
- Objects (variables) are instances of classes (types). A class is a template for all objects of a certain type.
- In OOP, a class is the natural way of implementing an ADT.

Classes and Objects



Let us design the new type for Point

```
class Point:
    """A class to represent and operate with two-dimensional points"""

    # Declaration of attributes (recommended for type checking)
    _x: float # x coordinate
    _y: float # y coordinate

    def __init__(self, x: float = 0, y: float = 0):
        """Constructor with x and y coordinates"""
        self._x, self._y = x, y

    def x(self) -> float:
        """Returns the x coordinate"""
        return self._x

    def y(self) -> float:
        """Returns the y coordinate"""
        return self._y

    def distance(self, p: Optional['Point']) -> float:
        """Returns the distance to point p
           (or the distance to the origin if p is None)"""
        dx, dy = self.x(), self.y()
        if p is not None:
            dx -= p.x()
            dy -= p.y()
        return math.sqrt(dx*dx + dy*dy)
```


Let us design the new type for Point

```
⋮

    def angle(self) -> float:
        """Returns the angle of the polar coordinate"""
        if self.x() == 0 and self.y() == 0:
            return 0
        return math.atan2(self.y()/self.x())

    def __add__(self, p: 'Point') -> 'Point':
        """Returns a new point by adding the coordinates of two points.
           This is a method associated to the + operator"""
        return Point(self.x() + p.x(), self.y() + p.y())

    def __eq__(self, p: 'Point') -> bool:
        """Checks whether two points are equal.
           This is a method associated to the == operator"""
        return self.x() == p.x() and self.y() == p.y()
```


How the class methods are invoked

```
p1 = Point(5.0, -3.2)  # __init__(p1, 5.0, -3.2)
p2 = Point(2.8)        # __init__(p2, 2.8, 0)

dist12 = p1.distance(p2)  # distance(p1, p2)
                 self

# Distance to the origin
r = p1.distance()         # distance(p1, None)

# Create another point by adding coordinates
p3 = p1 + p2  # Equivalent to p1.__add__(p2)

# We get the coordinates of the new point
x = p3.x()  # x = 7.8
y = p3.y()  # y = -3.2
```


How the class methods are invoked

```
p1 = Point(5.0, -3.2)
p2 = Point(2.8)

dist12 = p1.distance(p2)

r = p1.distance()

p3 = p1 + p2

x = p3.x()
y = p3.y()
```

```
class Point:
    def __init__(self, x: float = 0, y: float = 0):
    def x(self) -> float:
    def y(self) -> float:
    def distance(self, p: Optional['Point']) -> float:
    def angle(self) -> float:
    def __add__(self, p: 'Point') -> 'Point':
    def __eq__(self, p: 'Point') -> bool:
```


Python naming conventions

Type	Examples
Function	distance, dot_product, multiply_by_two
Variable	x, num, num_elements
Class	Point, CityGraph, ParkingLot
Public method	distance, get_angle, shortest_path
Private method	_gcd, _check, _calculate_mean
Magic method	__init__, __add__, __eq__, __str__
Constant	GRAVITY, MIN_DISTANCE, MAX_NUM_PEOPLE
Module	point.py, city_graph.py, parking_lot.py
Package	geometry, citygraph

Recommendation:
- use short names for modules and packages
- no underscores for package names

Comment: PascalCase, camelCase and snake_case

Magic methods

- They are invoked internally to implement certain actions.
- They are not supposed to be invoked by the user.
- Some examples:
  - Arithmetic: __add__, __mul__, __div__, __truediv__, __neg__, …
  - Relational: __eq__, __ne__, __gt__, __ge__, …
  - Representation: __str__, __repr__, …
  - Class initialization: __init__, __new__, __del__
  - and others

Class Point in C++

```
// The declaration of the class Point
class Point {

public:
  // Constructor
  Point(double x, double y);

  // Constructor for (0,0)
  Point();

  // Gets the x coordinate
  double x() const;

  // Gets the y coordinate
  double y() const;

  // Returns the distance to point p
  double distance(const Point& p) const;

  // Returns the distance to the origin
  double distance() const;

  // Returns the angle of the polar coordinate
  double angle() const;

  // Creates a new point by adding the coordinates of two points
  Point operator + (const Point& p) const;

private:
  double _x, _y;  // Coordinates of the point

};
```


Implementation of the class Point

```
// The constructor: different implementations
Point::Point(double x, double y) {
  _x = x; _y = y;
}

// or also
Point::Point(double x, double y) : _x(x), _y(y) {}
```

They are equivalent, but only one of them should be chosen.
We can have different constructors with different *signatures*.

```
// The other constructor
Point::Point() : x(0), y(0) {}
```


Implementation of the class Point

```
double Point::x() const {
  return _x;
}

double Point::y() const {
  return _y;
}

double Point::distance(const Point& p) const {
  double dx = x() - p.x(); // Better getX() than x
  double dy = y() - p.y();
  return sqrt(dx*dx + dy*dy);
}

double Point::distance() const {
  return sqrt(x()*x() + y()*y());
}
```

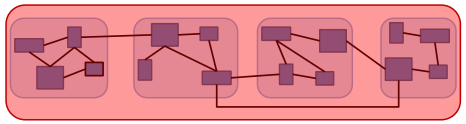
Note: compilers are smart. Small functions are expanded inline.

Implementation of the class Point

```
double Point::angle() const {
  if (x() == 0 and y() == 0) return 0;
  return atan(y()/x());
}

Point Point::operator + (const Point& p) const {
  return Point(x() + p.x(), y() + p.y());
}
```

Conclusions

- The human brain has limitations: 4 things at once.
- Modularity and abstraction are for designing large maintainable systems.



Abstract Data Types (II)
(and Object-Oriented Programming)

Jordi Cortadella and Jordi Petit
Department of Computer Science

Public or private?

- What should be public?
  - Only the methods that need to interact with the external world. Hide as much as possible. Make a method public only if necessary.
- What should be private?
  - All the attributes.
  - The internal methods of the class.
- Can we have public attributes?
  - Theoretically yes (Python and C++ allow it).
  - Recommendation: never define a public attribute.
- Observation: Python does not support public/private attributes and methods. There is no protection to prevents a bad use.
- The naming conventions (underscores) are used to distinguish them.

Class Point: a new implementation

- Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).
- We can modify the private section and the implementation of the class without modifying the specification of the public methods.
- The API (a contract between designers and users) should not be modified.
- Do you know what a *deprecated* method is?
  - Not recommended, possibly superseded by another method
  - Likely to be removed or discontinued in the future

Class Point: a new implementation

```python
class Point:
    """A class to represent and operate with two-dimensional points"""

    # Declaration of attributes (recommended for type checking)
    _radius: float # radius of the polar coordinates
    _angle: float # angle of the polar coordinates

    def __init__(self, x: float = 0, y: float = 0):
        """Constructor with x and y coordinates"""
        self._radius = math.sqrt(x*x + y*y)
        self._angle = 0 if x == 0 and y == 0 else math.atan2(y/x)

    def x(self) -> float:
        """Returns the x coordinate"""
        return self._radius*math.cos(self._angle)

    def y(self) -> float:
        """Returns the y coordinate"""
        return self._radius*math.sin(self._angle)

    def distance(self, p: Optional['Point']) -> float:
        """Returns the distance to point p
           (or the distance to the origin if p is None)"""
        dx, dy = self.x(), self.y()
        if p is not None:
            dx -= p.x()
            dy -= p.y()
        return math.sqrt(dx*dx + dy*dy)
```

Works without any change (but it can be done more efficiently)

Class Point: a new implementation

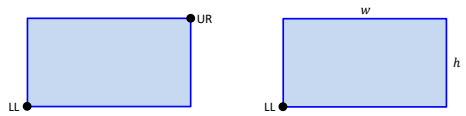
```python
    def angle(self) -> float:
        """Returns the angle of the polar coordinates"""
        return self._angle

    def __add__(self, p: 'Point') -> 'Point':
        """Returns a new point by adding the coordinates of two points.
           This method is associated to the + operator"""
        return Point(self.x() + p.x(), self.y() + p.y())

    def __eq__(self, p: 'Point') -> bool:
        """Checks whether two points are equal.
           This method is associated to the == operator"""
        return self.x() == p.x() and self.y() == p.y()
```

Discussion:
- How about using `_x` and `_y` (or `_radius` and `_angle`) as "public" attributes?
- Programs using `p._x` and `p._y` would not be valid for the new implementation.
- Programs using `p.x()` and `p.y()` would still be valid.

Recommendation:
- All attributes should be ***private***.

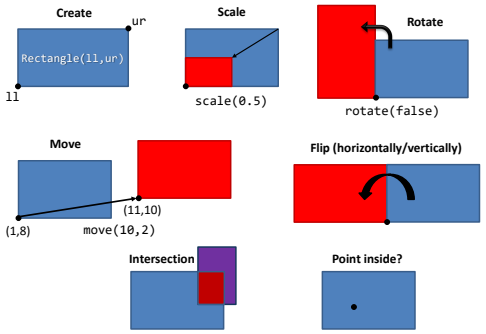
A new class: Rectangle

- We will only consider rectilinear rectangles (axis-aligned).
- A rectilinear rectangle can be represented in different ways:



Two points (extremes of diagonal) — One point, width and height

Rectangle: abstract view



Rectangle: ADT (incomplete)

```python
from point import Point

class Rectangle:
    """Class to operate with rectilinear rectangles"""

    def __init__(self, ll: Point, ur: Point):
        """Constructor using the LL and UR corners"""
        ...

    def area(self) -> float:
        """Returns the area of the rectangle"""
        ...

    def scale(self, s: float) -> None:
        """Scales the rectangle with a factor s > 0"""
        ...

    def __mul__(self, r: 'Rectangle') -> 'Rectangle':
        """Returns the intersection with another rectangle"""
        ...
```


Rectangle: using the ADT

```python
# Creates a rectangle 4x5
r1 = Rectangle(Point(0,0), Point(4,5))

# Creates a rectangle 8x4
r2 = Rectangle(Point(0,0), Point(8,4))

r1.move(2, 3)  # Moves the rectangle
r1.scale(1.2)  # Scales the rectangle
area1 = r1.area()  # Calculates the area

r3 = r1*r2;

if r3.empty(): …
```


Rectangle: ADT

```python
from point import Point

class Rectangle:
    """Class to operate with rectilinear rectangles"""

    # Private attributes to represent a rectangle
    _ll: Point  # Lower-left corner of the rectangle
    _w: float  # width of the rectangle (>= 0)
    _h: float  # height of the rectangle (>= 0)

    def __init__(self, ll: Point, ur: Point):
        """Constructor using the LL and UR corners"""
        assert ll.x() <= ur.x() and ll.y() <= ur.y()
        self._ll = Point(ll.x(), ll.y())
        self._w = ur.x() - ll.x()
        self._h = ur.y() - ll.y()
```

Note: Python does not support function overloading. Classes can only have one constructor.

Rectangle: ADT

```python
class Rectangle:
                    ⋮
    def width(self) -> float:
        """Returns the width of the rectangle"""
        return self._w

    def height(self) -> float:
        """Returns the height of the rectangle"""
        return self._h

    def ll(self) -> Point:
        """Returns the LL corner of the rectangle"""
        return Point(self._ll.x(), self._ll.y())

    def ur(self) -> Point:
        """Returns the UR corner of the rectangle"""
        return self._ll + Point(self.width(), self.height())
```

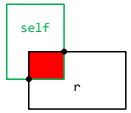
Discussion: what if `ll()` would return `self._ll` (instead of a copy)?

Rectangle: ADT

```python
class Rectangle:
                    ⋮
    def area(self) -> float:
        """Returns the area of the rectangle"""
        return self.width()*self.height()

    def scale(self, s: float) -> None:
        """Scales the rectangle by s,
           keeping the LL corner fixed"""
        self._w *= s
        self._h *= s

    def empty(self) -> bool:
        """Checks whether the rectangle is empty"""
        return self.area() <= 0  # Gives some tolerance
```


Rectangle: ADT

```python
class Rectangle:
                    ⋮
    def __mul__(self, r: 'Rectangle') -> 'Rectangle':
        """Returns the intersection of two rectangles"""

        # Calculate the ll coordinates
        r1ll, r2ll = self.ll(), r.ll()
        ll_x = max(r1ll.x(), r2ll().x())
        ll_y = max(r1ll.y(), r2ll().y())

        # Calculate the ur coordinates
        r1ur, r2ur = self.ur(), r.ur()
        ur_x = min(r1ur.x(), r2ur().x())
        ur_y = min(r1ur.y(), r2ur().y())

        # Check if no intersection: return empty rectangle
        if ur_x <= ll_x or ur_y <= ll_y:
            return Rectangle(Point(0,0), Point(0,0))

        return Rectangle(Point(ll_x, ll_y), Point(ur_x, ur_y))
```



Let us work with rectangles

```python
r1 = Rectangle(Point(2,3), Point(6,8))
area1 = r1.area()  # area1 = 20

r2 = Rectangle(Point(3,5), Point(5, 9))

# Check whether the point (4,7) is inside the
# intersection of r1 and r2.
in = (r1*r2).isPointInside(Point(4,7))

r2.rotate(false)  // r2 is rotated counterclockwise
r2 *= r1;         // Intersection with r1
```

Exercise: draw a picture of R1 and R2 after the execution of the previous code.

A Python session with rational numbers

```python
>>> from rational import Rational  # from file rational.py
>>> a = Rational(4, -6)       # construct with num and den
>>> print(a)
-2/3
>>> b = Rational(4)  # integer value
>>> print(b)
4
>>> print((a+b).num(), (a+b).den())
10 3
>>> c = Rational()  # c = 0
>>> if a < c:
...     print(a, "is negative")
...
-2/3 is negative
>>> print(a*b)  # uses the __str__ method (see later)
-8/3
>>> a/b  # uses the __repr__ method (see later)
Rational(-1/6)
>>>
```


The Rational class in Python

```python
class Rational:
    """Class to represent rational numbers"""

    # Private attributes:
    # Invariant: _den > 0 and gcd(_num, _den) = 1
    _num: int  # numerator
    _den: int  # denominator (invariant: _den > 0)

    def __init__(self, num: int = 0, den: int = 1):
        assert den != 0

        self._num, self._den = num, den
        self._simplify()
```

The Rational class in Python

```python
class Rational:
        ⋮
    def num(self) -> int:
        """Returns the numerator"""
        return self._num

    def den(self) -> int:
        """Returns the denominator"""
        return self._den

    def _simplify(self) -> None:   # Private method
        """Simplifies the representation"""
        if self._den < 0:
            self._num *= -1
            self._den *= -1
        d = math.gcd(abs(self._num), self._den)
        self._num //= d
        self._den //= d
```


Magic representation methods

```python
class Rational:

        ⋮

    def __str__(self) -> str:
        """Returns a user-friendly string with information
        about the value of the object. It is invoked by
        str(x) or print(x)."""
        if self._den == 1:
            return str(self._num)
        return f"{self._num}/{self._den}"

    def __repr__(self) -> str:
        """Returns a string with information about the
        representation of the class. It is invoked by repr(x)
        or simply 'x'."""
        return f"Rational({self})"
```


Magic arithmetic methods

```python
class Rational:

        ⋮

    def __neg__(self) -> 'Rational':
        """Returns -self."""
        return Rational(-self._num, self._den)

    def __add__(self, rhs: 'Rational') -> 'Rational':
        """Returns self + rhs."""
        num = self._num*rhs._den + self._den*rhs._num
        den = self._den*rhs._den
        return Rational(num, den)

    # Similarly for __sub__, __mul__, __truediv__
```


Magic relational methods

```python
class Rational:

        ⋮

    def __eq__(self, rhs: 'Rational') -> bool:
        """Checks whether self == rhs."""
        return self._num == rhs._num and self._den == rhs._den

    def __ne__(self, rhs: 'Rational') -> bool:
        """Checks whether self != rhs."""
        return not self == rhs

    def __lt__(self, rhs: 'Rational') -> bool:
        """Checks whether self < rhs."""
        return self._num*rhs._den < self._den*rhs._num

    def __le__(self, rhs: 'Rational') -> bool:
        """Checks whether self <= rhs."""
        return not rhs < self

    # Similarly for __gt__ and __ge__
```


Python documentation: *docstrings*

```
>>> from rational import Rational
>>> help(Rational.__add__)
Help on function __add__ in module rational:

__add__(self, rhs)
    Returns self + rhs.
>>> help(Rational)
class Rational(builtins.object)
 |  Rational(num=0, den=1)
 |
 |  Class to represent rational numbers.
 |
 |  The class includes the basic arithmetic and relational
 |  operators.
 |
 |  Methods defined here:
 |
 |  __add__(self, rhs)
 |      Returns self + rhs.
 |
 |  __eq__(self, rhs)
 |      Checks whether self == rhs.
```


Python documentation: *docstrings*

- The first line after a module, class or function can be used to insert a string that documents the component.
- Triple quotes (""") are very convenient to insert multi-line strings.
- The ***docstrings*** are stored in a special attribute of the component named __doc__.
- Different ways of print the ***docstrings*** associated to a component:
  - `print(Rational.num.__doc__)`
  - `help(Rational.num)`

Designing a module: example

```python
# geometry.py

"""geometry.py
Provides two classes for representing Polygons and Circles."""

# author: Euclid of Alexandria

from math import pi, sin, cos

class Polygon:

    """Represents polygons and provides methods to
    calculate area, intersection, convex hull, etc."""

    def __init__(self, list_vertices: list[Point]):
        """Creates a polygon from a list of vertices."""
    ...

class Circle:
    ...
```

Documentation of the module

Documentation of the class

Documentation of the method

Using a module: example

```python
import geometry
p = geometry.Poligon(…)
c = geometry.Circle(…)
```

Imports the module. Now all classes can be used with the prefix of the module.

```python
import geometry as geo
p = geo.Poligon(…)
c = geo.Circle(…)
```

Imports and renames the module.

```python
from geometry import *
p = Poligon(…)
c = Circle(…)
```

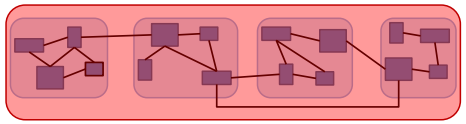
Imports all classes in the module. No need to add the prefix of the module.

```python
from geometry import Poligon as plg, Circle as cir
p = plg(…)
c = cir(…)
```

Imports and renames the classes in the module.

Conclusions

- Finding the appropriate hierarchy is a fundamental step towards the design of a complex system.
- User-friendly documentation is indispensable.



EXERCISES

Implement methods

Implement the following methods for the class Rectangle:

```python
def rotate(self, clockwise: bool) -> None:
    """Rotate the rectangle 90 degrees clockwise or counterclockwise,
       depending on the value of the parameter. The rotation should be
       done around the lower-left corner of the rectangle"""

def flip(self, horizontally: bool) -> None:
    """Flip horizontally (around the left edge) or vertically (around
       the bottom edge), depending on the value of the parameter."""

def isPointInside(self, p: Point) -> bool:
    """ Check whether point p is inside the rectangle"""
```


Re-implement a class

Re-implement the class Rectangle using an internal representation with two Points:

- Lower-Left (LL)
- Upper-Right(UR)

Make the minimum number of changes to **preserve the API** of the class.

Distance between rectangles

Implement the following method for the class Rectangle:

```python
def distance(self, r: 'Rectangle') -> float:
    """Calculates the shortest distance between two rectangles.
       If two rectangles intersect, their distance is zero."""
```

Hint: try to make the code simple and elegant, possibly defining some reusable private method.

Algorithm Analysis (I)

Jordi Cortadella and Jordi Petit
Department of Computer Science

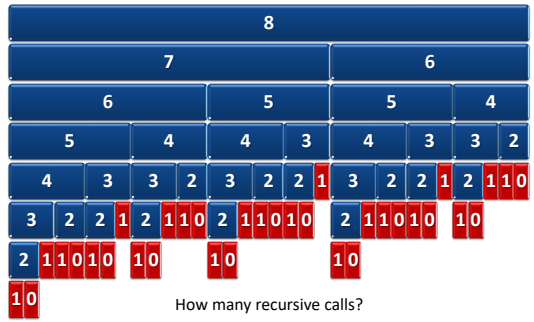
What do we expect from an algorithm?

- Correct
- Easy to understand
- Easy to implement
- Efficient:
  - Every algorithm requires a set of resources
    - Memory
    - CPU time
    - Energy

Fibonacci: recursive version

```
def fib(n: int) -> int:
    """Returns the Fibonacci number of order n
       Pre: n ≥ 0
    """
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
```


Fibonacci



How many recursive calls?

Fibonacci: runtime

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = T(n-1) + T(n-2)$$

Let us assume that $T(n) = a^n$ for some constant a. Then,

$$a^n = a^{n-1} + a^{n-2} \quad \Rightarrow \quad a^2 = a + 1$$

$$a = \frac{1+\sqrt{5}}{2} = \varphi \approx 1.618 \qquad \text{(golden ratio)}$$

Therefore, $T(n) \approx 1.6^n$.

If $T(0) = 1$ ns, then $T(100) \approx 2.6 \cdot 10^{20}$ns > 8000 yrs.

With the age of Universe ($14 \cdot 10^9$ yrs), we could compute up to `fib(128)`.

Fibonacci numbers: iterative version

```
def fib(n: int) -> int:
    """Returns the Fibonacci number of order n
       Pre: n ≥ 0
    """
    f_i, f_i1 = 0, 1
    # Inv: f_i is the Fibonacci number of order i
    #      f_i1 is the Fibonacci number of order i+1
    for i in range(n):
        f_i, f_i1 = f_i1, f_i+f_i1
    return f_i
```

Runtime: **n** iterations

Fibonacci numbers

Algebraic solution: find matrix A such that

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = A^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Fibonacci numbers

$$A^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \qquad A^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \qquad A^8 = \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix}$$

$$A^{16} = \begin{bmatrix} 1597 & 987 \\ 987 & 610 \end{bmatrix} \quad \cdots A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Runtime ≈ *$\log_2 n$ 2x2 matrix multiplications*

Algorithm analysis

Given an algorithm that reads inputs from a domain D, we want to define a cost function C:

$$C : D \to \mathbb{R}^+$$
$$x \mapsto C(x)$$

where $C(x)$ represents the cost of using some resource (CPU time, memory, energy, ...).

Analyzing $C(x)$ for every possible x is impractical.

Algorithm analysis: simplifications

- Analysis based on the size of the input: $|x| = n$
- Only the best/average/worst cases are analyzed:

$$\begin{aligned} C_{\text{worst}}(n) &= \max\{C(x) : x \in D, |x| = n\} \\ C_{\text{best}}(n) &= \min\{C(x) : x \in D, |x| = n\} \\ C_{\text{avg}}(n) &= \sum_{x \in D, |x|=n} p(x) \cdot C(x) \end{aligned}$$

$p(x)$: probability of selecting input x among all the inputs of size n.

Algorithm analysis

- Properties:

$$\forall n \geq 0 : \quad C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n)$$
$$\forall x \in D : \quad C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|)$$

- We want a notation that characterizes the cost of algorithms independently from the technology (CPU speed, programming language, efficiency of the compiler, etc.).
- Runtime is usually the most important resource to analyze.

Asymptotic notation

Let us consider all functions $f : \mathbb{R}^+ \to \mathbb{R}^+$

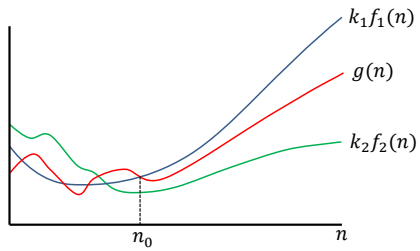
Definitions:

$$\mathrm{O}(f(n)) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \leq k \cdot f(n)\}$$

$$\Omega(f(n)) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \geq k \cdot f(n)\}$$

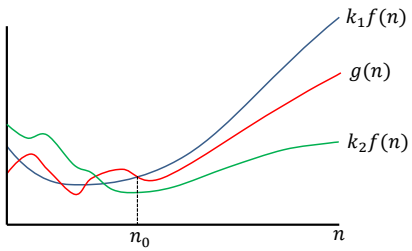
$$\Theta(f(n)) = \mathrm{O}(f(n)) \cap \Omega(f(n))$$

Asymptotic notation



$$g(n) \in \mathrm{O}(f_1(n))$$
$$g(n) \in \Omega(f_2(n))$$

Asymptotic notation



$$g(n) \in \Theta(f(n))$$

Asymptotic notation: example

$$\begin{aligned} \mathrm{O}(n^2) = \{ & 3n^2 - n + 20, \\ & 0.5n^2 - 3, \\ & 4n \log_2 n, \\ & 2n - 5, \\ & n < 20 \ ? \ 2^n : 2n^2 + 1000, \\ & \ldots \\ & \} \end{aligned}$$

$$\begin{aligned} \Omega(n^2) = \{ & 3n^2 - n + 20, \\ & 0.5n^2 - 3, \\ & 3n^3 \log_2 n, \\ & 2^n - 4n^3, \\ & n < 20 \ ? \ 2^n : 2n^2 + 1000, \\ & \ldots \\ & \} \end{aligned}$$

$$\begin{aligned} \Theta(n^2) = \mathrm{O}(n^2) \cap \Omega(n^2) = \{ & 3n^2 - n + 20, \\ & 0.5n^2 - 3, \\ & n < 20 \ ? \ 2^n : 2n^2 + 1000, \\ & \ldots \\ & \} \end{aligned}$$

Examples

Big-O

$13n^3 - 4n + 8$	$\in$	$\mathrm{O}(n^3)$
$2n - 5$	$\in$	$\mathrm{O}(n)$
n^2	$\notin$	$\mathrm{O}(n)$
2^n	$\in$	$\mathrm{O}(n!)$
3^n	$\notin$	$\mathrm{O}(2^n)$
$3 \log_2 n$	$\in$	$\mathrm{O}(\log n)$
$n \log_2 n$	$\in$	$\mathrm{O}(n^2)$
$\mathrm{O}(n^2)$	$\subseteq$	$\mathrm{O}(n^3)$

Big-Ω

$13n^3 - 4n + 8$	$\in$	$\Omega(n^3)$
n^2	$\in$	$\Omega(n)$
n^2	$\notin$	$\Omega(n^3)$
$n!$	$\in$	$\Omega(2^n)$
3^n	$\notin$	$\Omega(2^n)$
$3 \log_2 n$	$\in$	$\Omega(\log n)$
$n \log_2 n$	$\in$	$\Omega(n)$
$\Omega(n^3)$	$\subseteq$	$\Omega(n^2)$

Complexity ranking

Function	Common name
$n!$	factorial
2^n	exponential
$n^d,\ d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	quasi-linear
n	linear
$\sqrt{n}$	root - n
$\log n$	logarithmic
1	constant

The limit rule

Let us assume that L exists (may be ∞) such that:

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$\begin{cases} \text{if } L = 0 & \text{then} \quad f \in \mathrm{O}(g) \\ \text{if } 0 < L < \infty & \text{then} \quad f \in \Theta(g) \\ \text{if } L = \infty & \text{then} \quad f \in \Omega(g) \end{cases}$$

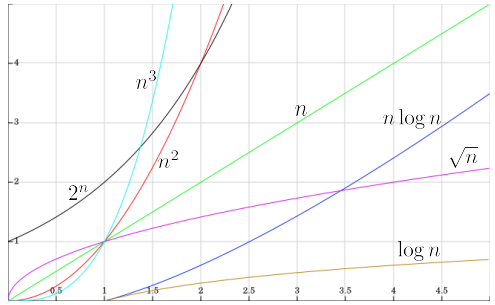
Note: If both limits are ∞ or 0, use L'Hôpital rule:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

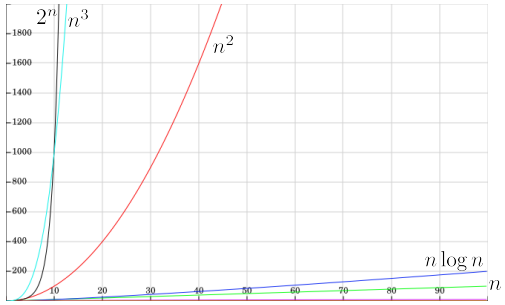
Properties

- $f \in \mathrm{O}(f)$
- $\forall c > 0,\ \ \mathrm{O}(f) = \mathrm{O}(c \cdot f)$
- $f \in \mathrm{O}(g) \wedge g \in \mathrm{O}(h) \Rightarrow f \in \mathrm{O}(h)$
- $f_1 \in \mathrm{O}(g_1) \wedge f_2 \in \mathrm{O}(g_2)$
  $\Rightarrow f_1 + f_2 \in \mathrm{O}(g_1 + g_2) = \mathrm{O}(\max\{g_1, g_2\})$
- $f \in \mathrm{O}(g) \Rightarrow f + g \in \mathrm{O}(g)$
- $f_1 \in \mathrm{O}(g_1) \wedge f_2 \in \mathrm{O}(g_2) \Rightarrow f_1 \cdot f_2 \in \mathrm{O}(g_1 \cdot g_2)$
- $f \in \mathrm{O}(g) \Leftrightarrow g \in \Omega(f)$

Asymptotic complexity (small values)



Asymptotic complexity (larger values)



Execution time: example

Let us consider that every operation can be executed in 1 ns (10^{-9} s).

Function	Time $n = 1,000$	$n = 10,000$	$n = 100,000$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

How about "big data"?

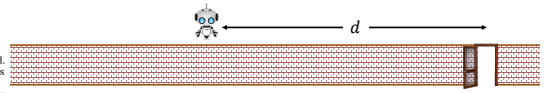
Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

This is often the practical limit for big data

The robot and the door in an infinite wall

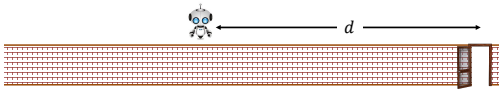


A robot stands in front of a wall that is infinitely long to the right and left side. The wall has a door somewhere and the robot has to find it to reach the other side. Unfortunately, the robot can only see the part of the wall in front of it.

The robot does not know neither how far away the door is nor what direction to take to find it. It can only execute moves to the left or right by a certain number of steps.

Let us assume that the door is at a distance d. How to find the door in a minimum number of steps?

The robot and the door in an infinite wall



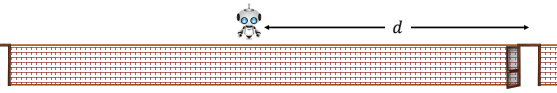
Algorithm 1:

- Pick one direction and move until the door is found.

Complexity:

- If the direction is correct → $\mathrm{O}(d)$.
- If incorrect → the algorithm does not terminate.

The robot and the door in an infinite wall



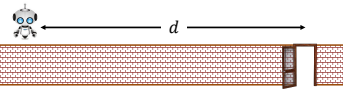
Algorithm 2:

- 1 step to the left,
- 2 steps to the right,
- 3 steps to the left, …
- … increasing by one step in the opposite direction.

Complexity:

$$T(d) = 3d + \sum_{i=1}^{d-1} 4i = 3d + 4\frac{d(d-1)}{2} = 2d^2 + d = \mathrm{O}(d^2)$$

The robot and the door in an infinite wall



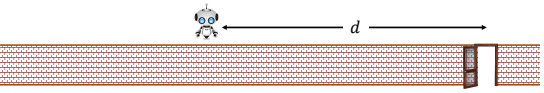
Algorithm 3:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 3 steps to the left and return to origin,…
- … increasing by one step in the opposite direction.

Complexity:

$$T(d) = d + \sum_{i=1}^{d} 2i = d + 2\frac{d(d+1)}{2} = d^2 + 2d = \mathrm{O}(d^2)$$

The robot and the door in an infinite wall



Algorithm 4:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 4 steps to the left and return to origin,…
- … doubling the number of steps in the opposite direction.

Complexity (assume that $d = 2^n$):

$$T(d) = d + 2\sum_{i=0}^{n} 2^i = d + 2(2^{n+1} - 1) = 5d - 2 = \mathrm{O}(d)$$

Runtime analysis rules

- Variable declarations cost no time.
- ***Elementary operations*** are those that can be executed with a ***small number of basic computer steps*** (an assignment, a multiplication, a comparison between two numbers, etc.).
- Vector sorting or matrix multiplication are not elementary operations.
- We consider that the cost of elementary operations is $\mathrm{O}(1)$.

Runtime analysis rules

- Consecutive statements:
  - If S1 is $\mathrm{O}(f)$ and S2 is $\mathrm{O}(g)$, then S1;S2 is $\mathrm{O}(\max\{f, g\})$
- Conditional statements:
  - If S1 is $\mathrm{O}(f)$, S2 is $\mathrm{O}(g)$ and B is $\mathrm{O}(h)$, then if (B) S1; else S2; is $\mathrm{O}(\max\{f + h, g + h\})$, or also $\mathrm{O}(\max\{f, g, h\})$.

Runtime analysis rules

- For/While loops:
  - Running time is at most the running time of the statements inside the loop times the number of iterations
- Nested loops:
  - Analyze inside out: running time of the statements inside the loops multiplied by the product of the sizes of the loops

Nested loops: examples

```
for i in range(n):
    for j in range(n):
        do_something()  # O(1)
```

$\Longrightarrow \mathrm{O}(n^2)$

```
for i in range(n):
    for j in range(i, n):
        do_something()  # O(1)
```

$\Longrightarrow \mathrm{O}(n^2)$

```
for i in range(n):
    for j in range(m):
        for k in range(p):
            do_something()  # O(1)
```

$\Longrightarrow \mathrm{O}(n \cdot m \cdot p)$

Linear time: $O(n)$

Running time proportional to input size

```
# Compute the maximum of a vector with n numbers
m = a[0]
for i in range(1, len(a)):
    m = max(m, a[i])

# Equivalent way in Python (same complexity)
m = max(a)
```


Linear time: $O(n)$

Other examples:

- Reversing a vector
- Merging two sorted vectors
- Finding the largest null segment of a sorted vector: a linear-time algorithm exists (a null segment is a compact sub-vector in which the sum of all the elements is zero)

Logarithmic time: $O(\log n)$

- Logarithmic time is usually related to divide-and-conquer algorithms
- Examples:
  - Binary search
  - Calculating x^n
  - Calculating the n-th Fibonacci number

Example: recursive x^y

```
def power(x: int, y: int) -> int:
    """Returns x^y. Pre: x ≠ 0, y ≥ 0"""
    if y == 0: return 1
    if y%2 == 0: return power(x*x, y//2);
    return x*power(x*x, y//2);

# Assumption: each */% takes O(1)
```

$$T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \quad \cdots$$

$$T(x^y) \leq \underbrace{4 + 4 + \cdots + 4}_{\log_2 y \ \ times} \quad \Longrightarrow \quad O(\log y)$$

Linearithmic time: $O(n \log n)$

- ***Sorting:*** Merge sort and heap sort can be executed in $O(n \log n)$.
- ***Largest empty interval:*** Given n time-stamps $x_1, \cdots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?
  - $O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Algorithm Analysis (II)

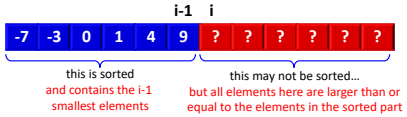
Jordi Cortadella and Jordi Petit
Department of Computer Science

Examples

- Selection sort
- Insertion sort
- The Maximum Subsequence Sum Problem
- Convex Hull

Selection Sort

- Selection sort uses this invariant:



Selection Sort

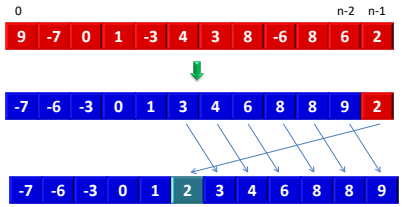
```python
def selection_sort(v: list[T]) -> None:
    """Sorts v in ascending order"""
    for i in range(len(v)-1):
        k = i
        for j in range(i+1, len(v)):
            if v[j] < v[k]:
                k = j
        v[k], v[i] = v[i], v[k]
```

$$T(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} \mathrm{O}(1) = \mathrm{O}(1)\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \mathrm{O}(1)\sum_{i=0}^{n-2}(n-i-1)$$
$$= \mathrm{O}(1)\left(\frac{n}{2}(n-1)\right) = \mathrm{O}(1)\cdot\mathrm{O}(n^2) = \mathrm{O}(n^2)$$

Observation: notice that $T(n) \in \Omega(n^2)$, also. Therefore, $T(n) \in \Theta(n^2)$.

Insertion Sort

- Let us use inductive reasoning:
  - If we know how to sort arrays of size n-1,
  - do we know how to sort arrays of size n?



Insertion Sort

```python
def insertion_sort(v: list[T]) -> None:
    """Sorts v in ascending order"""
    for i in range(1, len(v)):  # n-1 times
        x = v[i]
        j = i
        while j > 0 and v[j - 1] > x:  # 0..i times
            v[j] = v[j - 1]
            j -= 1
        v[j] = x
```

$$\boxed{\begin{aligned} T(n) &= \Omega(n) \\ T(n) &= \mathrm{O}(n^2) \end{aligned}}$$

$$T_{\text{worst}}(n) = \sum_{i=1}^{n-1} i\cdot\mathrm{O}(1) = \mathrm{O}(n^2) \qquad \Rightarrow \text{sorted in reverse order}$$

$$T_{\text{best}}(n) = \sum_{i=1}^{n-1} \mathrm{O}(1) = \mathrm{O}(n) \qquad \Rightarrow \text{already sorted}$$

The Maximum Subsequence Sum Problem

- Given (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum value of $\sum_{k=i}^{j} A_k$.
  (the max subsequence sum is 0 if all integers are negative).
- Example:
  - Input: -2, 11, -4, 13, -5, -2
  - Answer: 20 (subsequence 11, -4, 13)

(extracted from M.A. Weiss, Data Structures and Algorithms in C++, Pearson, 2014, 4th edition)

The Maximum Subsequence Sum Problem

```python
def max_sub_sum(a: list[int]) -> int:
    """Returns the sum of the maximum subsequence of a"""
    n = len(a)
    max_sum = 0
    # try all possible subsequences
    for i in range(n):
        for j in range(i, n):
            this_sum = 0
            for k in range(i, j+1):
                this_sum += a[k]
                max_sum = max(max_sum, this_sum)
    return max_sum
```

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{j}\sum_{k=i}^{j} 1$$

The Maximum Subsequence Sum Problem

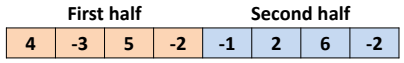
$$T(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1$$
$$= \sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)$$
$$= \sum_{i=0}^{n-1}\frac{(n-i+1)(n-i)}{2} = \cdots$$
$$= \frac{n^3+3n^2+2n}{6} = \Theta(n^3)$$

The Maximum Subsequence Sum Problem

```python
def max_sub_sum(a: list[int]) -> int:
    """Returns the sum of the maximum subsequence of a"""
    n = len(a)
    max_sum = 0
    # try all possible subsequences
    for i in range(n):
        this_sum = 0
        for j in range(i, n):
            this_sum += a[j]  # reuse computation
            max_sum = max(max_sum, this_sum)
    return max_sum
```

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} 1 = \Theta(n^2)$$

Max Subsequence Sum: Divide&Conquer



The max sum can be in one of three places:

- 1st half
- 2nd half
- Spanning both halves and crossing the middle

In the 3rd case, two max subsequences must be found starting from the center of the vector (one to the left and the other to the right)

Max Subsequence Sum: Divide&Conquer

```python
def max_sub_sum_rec(a: list[int], left: int, right: int) -> int:
    """Returns the sum of the maximum subsequence of a[left:right+1]"""
    if left == right:  # base case
        return max(a[left], 0)

    # Recursive cases: left and right halves
    center = (left + right)//2
    max_left = max_sub_sum_rec(a, left, center)
    max_right = max_sub_sum_rec(a, center+1, right)

    # Subsequence in a[center+1:right+1]
    max_rcenter, right_sum = 0, 0
    for i in range(center+1, right+1):
        right_sum += a[i]
        max_rcenter = max(max_rcenter, right_sum)

    # Subsequence in a[left:center+1]
    max_lcenter, left_sum = 0, 0
    for i in range(center, left-1, -1):
        left_sum += a[i]
        max_lcenter = max(max_lcenter, left_sum)

    return max(max_left, max_right, max_lcenter + max_rcenter)
```

Max Subsequence Sum: Divide&Conquer

$$T(1) = 1$$
$$T(n) = 2T(n/2) + \Theta(n)$$

We will see how to solve this equation formally in the next lesson (Master Theorem). Informally:

$$T(n) = 2T(n/2) + n = 2(2T(n/4) + n/2) + n$$
$$= 4T(n/4) + n + n = 8T(n/8) + n + n + n = \cdots$$
$$= 2^k T(n/2^k) + \underbrace{n + n + \cdots + n}_{k}$$

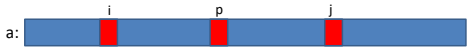
when $n = 2^k$, we have that $k = \log_2 n$, hence

$$T(n) = 2^k T(1) + kn = n + n\log_2 n = \Theta(n\log n)$$

But, can we still do it faster?

The Maximum Subsequence Sum Problem

- Observations:
  - If a[i] is negative, it cannot be the start of the optimal subsequence.
  - Any negative subsequence cannot be the prefix of the optimal subsequence.
- Let us consider the inner loop of the $\mathrm{O}(n^2)$ algorithm and assume that all prefixes of a[i..j-1] are positive and a[i..j] is negative:



  - If p is an index between i+1 and j, then any subsequence from a[p] is not larger than any subsequence from a[i] and including a[p-1].
  - If a[j] makes the current subsequence negative, we can advance i to j+1.

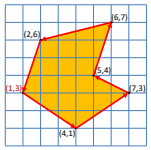
The Maximum Subsequence Sum Problem

```python
def max_sub_sum(a: list[int]) -> int:
    """Returns the sum of the maximum subsequence of a"""
    max_sum, this_sum = 0, 0
    for x in a:
        this_sum += x
        max_sum = max(max_sum, this_sum)
        this_sum = max(this_sum, 0)
    return max_sum
```

$$T(n) = \Theta(n)$$

a:	4	-3	5	-4	-3	-1	5	-2	6	-3	2
this_sum:	4	1	6	2	0	0	5	3	9	6	8
max_sum:	4	4	6	6	6	6	6	6	9	9	9

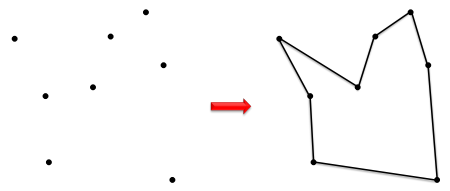
Representation of polygons



- A polygon can be represented by a sequence of vertices.
- Two consecutive vertices represent an edge of the polygon.
- The last edge is represented by the first and last vertices of the sequence.

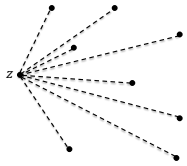
```
Vertices:  (1,3) (4,1) (7,3) (5,4) (6,7) (2,6)

Edges: (1,3)-(4,1)-(7,3)-(5,4)-(6,7)-(2,6)-(1,3)

    # A polygon (an ordered set of vertices)
    Polygon = list[Point]
```

Create a polygon from a set of points



Given a set of n points in the plane, connect them in a simple closed path.

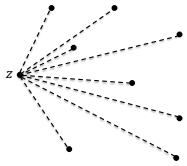
Simple polygon

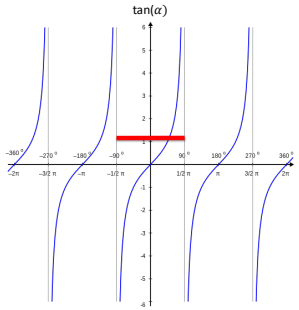


Input: $p_1, p_2, \ldots, p_n$ (points in the plane).
Output: P (a polygon whose vertices are $p_1, p_2, \ldots, p_n$ in some order).

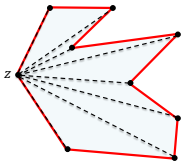
1) Select a point z with the smallest x coordinate (and smallest y in case of a tie in the x coordinate). Assume $z = p_1$.
2) For each $p_i \in \{p_2, \ldots, p_n\}$, calculate the angle α_i between the lines $z - p_i$ and the x axis.
3) Sort the points $\{p_2, \ldots, p_n\}$ according to their angles. In case of a tie, use distance to z.

Simple polygon





Simple polygon

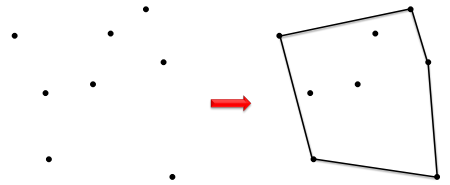


Implementation details:

- There is no need to calculate angles (requires arctan). It is enough to calculate slopes ($\Delta y / \Delta x$).
- There is not need to calculate distances. It is enough to calculate the square of distances (no sqrt required).

Complexity: $O(n \log n)$.
The runtime is dominated by the sorting algorithm.

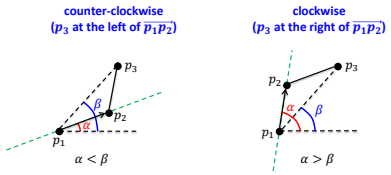
Convex hull



Compute the convex hull of n given points in the plane.

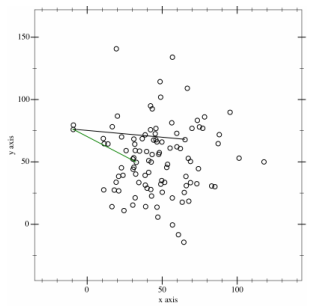
Clockwise and counter-clockwise

How to calculate whether three consecutive vertices are in a **clockwise** or ***counter-clockwise*** turn.



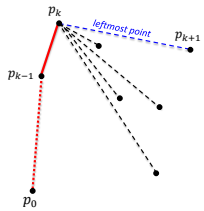

```
def left_of(p1:Point, p2:Point, p3:Point) -> bool:
    """Returns true if p3 is at the left of p1p2"""
    return (p2.x - p1.x) · (p3.y - p1.y) > (p2.y - p1.y) · (p3.x - p1.x)
```


Convex hull: gift wrapping algorithm



https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

Convex hull: gift wrapping algorithm



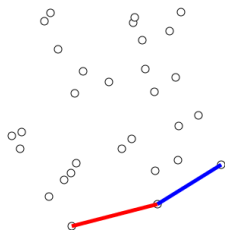
- **Input:** $p_1, p_2, \ldots, p_n$ (points in the plane).
- **Output:** P (the convex hull of $p_1, p_2, \ldots, p_n$).
- **Initial points:** p_0 with the smallest x coordinate.
- **Iteration:** Assume that a partial path with k points has been built (p_k is the last point). Pick some arbitrary $p_{k+1} \neq p_k$. Visit the remaining points. If some point q is at the left of $\overrightarrow{p_k p_{k+1}}$ redefine $p_{k+1} = q$.
- Stop when P is complete (back to point p_0).

Complexity: At each iteration, we calculate n angles. $T(n) = O(hn)$, where h is the number of points in the convex hull. In the worst case, $T(n) = O(n^2)$.

Convex hull: gift wrapping algorithm

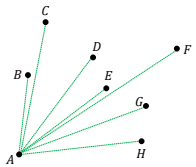
```
def gift_wrapping(pol: Polygon) -> Polygon:
    """Returns the convex-hull of a set of points"""
    hull: Polygon = []

    # Pick the leftmost point
    left = 0
    for i, p in enumerate(pol):
        if pol[i].x < pol[left].x:
            left = i

    p = left
    while True:  # Add points while the polygon is not closed
        hull.append(pol[p])  # Add point to the convex hull
        q = (p+1)%len(pol)  # Pick a point different from p

        for i, new_p in enumerate(pol): # Find leftmost point of p->q
            if left_of(pol[p], pol[q], new_p):
                q = i

        p = q  # This is the leftmost point
        if p == left:  # Stop if the point closes the polygon
            break

    return hull
```

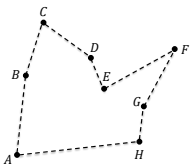

Convex hull: Graham Scan



https://en.wikipedia.org/wiki/Graham_scan

Convex hull: Graham scan

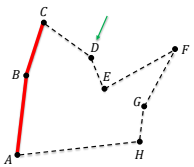




Convex hull: Graham scan

P: A B C D E F G H

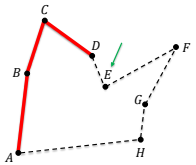
Q: A B C



Convex hull: Graham scan

P: A B C D E F G H

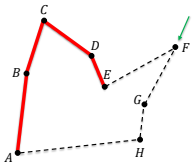
Q: A B C D



Convex hull: Graham scan

P: A B C D E F G H

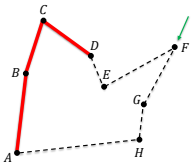
Q: A B C D E



Convex hull: Graham scan

P: A B C D E F G H

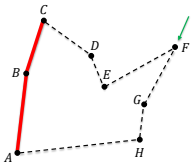
Q: A B C D



Convex hull: Graham scan

P: A B C D E F G H

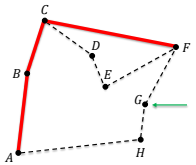
Q: A B C



Convex hull: Graham scan

P: A B C D E F G H

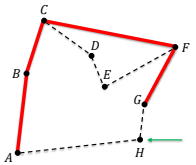
Q: A B C F



Convex hull: Graham scan

P: A B C D E F G H

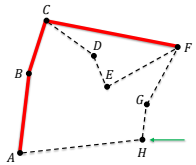
Q: A B C F G



Convex hull: Graham scan

P: A B C D E F G H

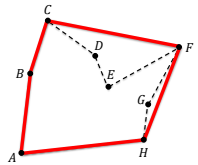
Q: A B C F



Convex hull: Graham scan

P: A B C D E F G H

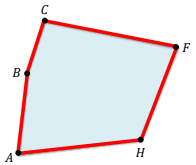
Q: A B C F H



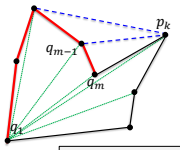
Convex hull: Graham scan

P: A B C D E F G H

Q: A B C F H



Convex hull: Graham scan



Input: $p_1, p_2, \dots, p_n$ (polygon: points in the plane).
Output: $q_1, q_2, \dots, q_m$ (the convex hull).

Initially:
Create a simple polygon P (complexity $O(n \log n)$).
Assume the order of the points is $p_1, p_2, \dots, p_n$.

```
def graham_scan(pol: Polygon) -> Polygon:
    """Returns the convex hull of a non-convex polygon"""
    hull = pol[0:3]
    for k in range(3, len(pol)):
        while left_of(hull[-2], hull[-1], pol[k]):
            hull.pop()
        hull.append(pol[k])
    return hull
```

Observation: each point p_k can be included in Q and deleted at most once.
The main loop of Graham scan has linear cost.
Complexity: dominated by the creation of the simple polygon → $O(n \log n)$.

EXERCISES

Summations

Prove the following equalities:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

For loops: analyze the cost of each code

Calculate the value of variable **s** at the end of each code

```
# Code 1
s = 0
for i in range(n):
    s += 1

# Code 2
s = 0
for i in range(0, n, 2):
    s += 1

# Code 3
s = 0
for i in range(n):
    s += 1
for j in range(n):
    s += 1

# Code 4
s = 0
for i in range(n):
    for j in range(n):
        s += 1

# Code 5
s = 0
for i in range(n):
    for j in range(i):
        s += 1

# Code 6
s = 0
for i in range(n):
    for j in range(i, n):
        s += 1
```


For loops: analyze the cost of each code

```
# Code 7
s = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            s += 1

# Code 8
s = 0
for i in range(n):
    for j in range(i):
        for k in range(j):
            s += 1

# Code 9
s = 0
i = 1
while i <= n:
    s += 1
    i *= 2

# Code 10
s = 0
for i in range(n):
    j = 1
    while j <= n:
        s += 1
        j *= 2

# Code 11
s = 0
for i in range(n):
    for j in range(i*i):
        for k in range(n):
            s += 1

# Code 12
s = 0
for i in range(n):
    for j in range(i*i):
        if j%i == 0:
            for k in range(n):
                s += 1
```


O, Ω or Θ?

The following statements refer to the ***insertion sort*** algorithm and the X's hide an occurrence of O, Ω or Θ. For each statement, find which options for $X \in \{O, \Omega, \Theta\}$ make the statement true or false. Justify your answers.

1. The worst case is $X(n^2)$
2. The worst case is $X(n)$
3. The best case is $X(n^2)$
4. The best case is $X(n)$
5. For every probability distribution, the average case is $X(n^2)$
6. For every probability distribution, the average case is $X(n)$
7. For some probability distribution, the average case is $X(n \log n)$

Primality

The following algorithms try to determine whether $n \geq 0$ is prime.
Find which ones are correct and analyze their cost as a function of n.

```
def is_prime1(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2,n):
        if n%i == 0:
            return False
    return True

def is_prime2(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n))):
        if n%i == 0:
            return False
    return True

def is_prime3(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, round(math.sqrt(n))):
        if n%i == 0:
            return False
    return True
```

```
def is_prime4(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n))+1):
        if n%i == 0:
            return False
    return True

def is_prime5(n: int) -> bool:
    if n <= 1:
        return False
    if n == 2:
        return True
    if n%2 == 0:
        return False
    for i in range(3, int(math.sqrt(n))+1, 2):
        if (n%i == 0):
            return False
    return True
```


The Sieve of Eratosthenes

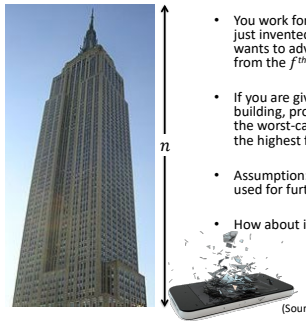
The following program is a version of the Sieve of Eratosthenes. Analyze its complexity.

```
def primes(n: int) -> list[bool]:
    p: list[bool] = [True]*(n+1)
    p[0] = p[1] = False
    for i in range(2, int(math.sqrt(n))+1):
        if p[i]:
            for j in range(i*i, n+1, i):
                p[j] = False
    return p
```

You can use the following equality, where $p \leq x$ refers to all primes $p \leq x$:

$$\sum_{p \leq x} \frac{1}{p} = \log \log x + O(1)$$

The Cell Phone Dropping Problem



- You work for a cell phone company which has just invented a new cell phone protector and wants to advertise that it can be dropped from the fth floor without breaking.
- If you are given 1 or 2 phones and an n story building, propose an algorithm that minimizes the worst-case number of trial drops to know the highest floor it won't break.
- Assumption: a broken cell phone cannot be used for further trials.
- How about if you have p cell phones?

(Source: Wood & Yasskin, Texas A&M University)

Divide & Conquer (I)

Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide-and-conquer algorithms

- Strategy:
  - Divide the problem into smaller subproblems of the same type of problem
  - Solve the subproblems recursively
  - Combine the answers to solve the original problem
- The work is done in three places:
  - In partitioning the problem into subproblems
  - In solving the basic cases at the tail of the recursion
  - In merging the answers of the subproblems to obtain the solution of the original problem

Conventional product of polynomials

Example:

$$P(x) = 2x^3 + x^2 - 4$$

$$Q(x) = x^2 - 2x + 3$$

$$(P \cdot Q)(x) = 2x^5 + (-4+1)x^4 + (6-2)x^3 + 8x - 12$$

$$(P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12$$

Conventional product of polynomials

```python
Polynomial = list[float]

def mul(p: Polynomial, q: Polynomial) -> Polynomial:
    """Returns p×q (product of polynomials)"""

    # degree(p) = len(p)-1, degree(q) = len(q)-1
    # degree(r) = degree(p)+degree(q)
    r: Polynomial = [0]*(len(p) + len(q) - 1)
    for i, pi in enumerate(p):
        for j, qj in enumerate(q):
            r[i+j] += pi*qj
    return r
```

Complexity analysis:
- Multiplication of polynomials of degree n: $O(n^2)$
- Addition of polynomials of degree n: $O(n)$

Product of polynomials: Divide&Conquer

Assume that we have two polynomials with n coefficients (degree $n-1$)

	$n-1$ … $n/2$	$n/2$ … 0
$\boldsymbol{P}$:	$\boldsymbol{P_L}$	$\boldsymbol{P_R}$
$\boldsymbol{Q}$:	$\boldsymbol{Q_L}$	$\boldsymbol{Q_R}$

$$P(x) \cdot Q(x) = P_L(x) \cdot Q_L(x) \cdot x^n + (P_R(x) \cdot Q_L(x) + P_L(x) \cdot Q_R(x)) \cdot x^{n/2} + P_R(x) \cdot Q_R(x)$$

$$T(n) = 4 \cdot T(n/2) + O(n) = O(n^2)$$ ← Shown later

Product of complex numbers

- The product of two complex numbers requires four multiplications:

$$(a+bi)(c+di) = ac - bd + (bc+ad)i$$

- Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: ac, bd and $(a+b)(c+d)$

$$bc + ad = (a+b)(c+d) - ac - bd$$

- A similar observation applies for polynomial multiplication.

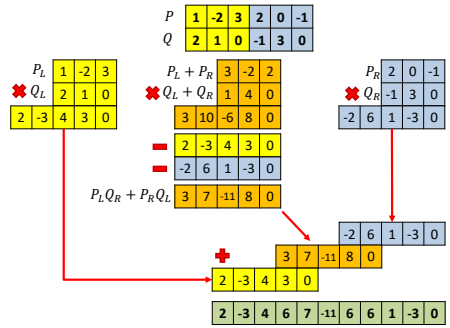
Product of polynomials with Gauss's trick

$$\begin{aligned} R_1 &= P_L Q_L \\ R_2 &= P_R Q_R \\ R_3 &= (P_L + P_R)(Q_L + Q_R) \end{aligned}$$

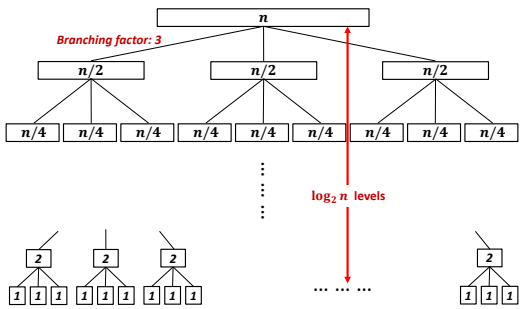
$$PQ = \underbrace{P_L Q_L}_{R_1} x^n + \underbrace{(P_R Q_L + P_L Q_R)}_{R_3 - R_1 - R_2} x^{n/2} + \underbrace{P_R Q_R}_{R_2}$$

$$T(n) = 3T(n/2) + O(n)$$

Polynomial multiplication: recursive step



Pattern of recursive calls



Useful reminders

- Sum of geometric series with ratio r:

$$S = k + kr + kr^2 + kr^3 + \cdots + kr^{n-1} = k\left(\frac{1-r^n}{1-r}\right)$$

For a decreasing series ($r < 1$): $S \le \frac{k}{1-r}$

- Logarithms:

$$\log_b n = \log_b a \cdot \log_a n$$

$$a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}$$

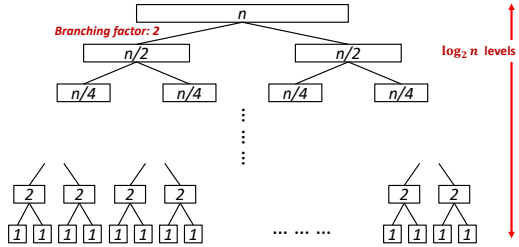
Complexity analysis

- The time spent at level k is

$$3^k \cdot O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \cdot O(n)$$

- For $k = 0$, runtime is $O(n)$.
- For $k = \log_2 n$, runtime is $O(3^{\log_2 n})$, which is equal to $O(n^{\log_2 3})$.
- The runtime per level increases geometrically by a factor of 3/2 per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series.
- Therefore, the complexity is $O(n^{1.59})$.

A popular recursion tree



Example: efficient sorting algorithms.
$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$

Algorithms may differ on the amount of work done at each level: $O(n^c)$

Examples

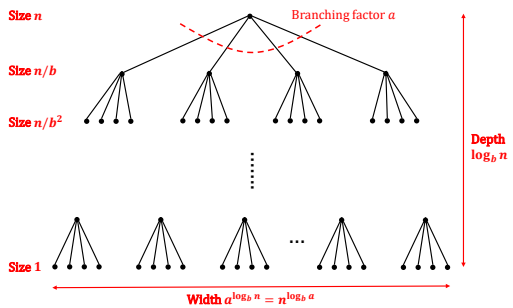
Algorithm	Branch	c	Runtime equation
Power (x^y)	1	0	$T(y) = T(y/2) + O(1)$
Binary search	1	0	$T(n) = T(n/2) + O(1)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + O(n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + O(n)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + O(n)$

Master theorem

- Typical pattern for Divide&Conquer algorithms:
  - Split the problem into a subproblems of size n/b
  - Solve each subproblem recursively
  - Combine the answers in $O(n^c)$ time
- Running time: $T(n) = a \cdot T(n/b) + O(n^c)$
- Master theorem:

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O(n^{\log_b a}) & \text{if } a > b^c \end{cases}$$

Master theorem: recursion tree



Master theorem: proof

- For simplicity, assume n is a power of b.
- The base case is reached after $\log_b n$ levels.
- The kth level of the tree has a^k subproblems of size n/b^k.
- The total work done at level k is:

$$a^k \times O\left(\frac{n}{b^k}\right)^c = O(n^c) \times \left(\frac{a}{b^c}\right)^k$$

- As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^c. We need to find the sum of such a series.

$$T(n) = O(n^c) \cdot \underbrace{\left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \cdots + \frac{a^{\log_b n}}{b^{(\log_b n)c}}\right)}_{\log_b n \text{ terms}}$$

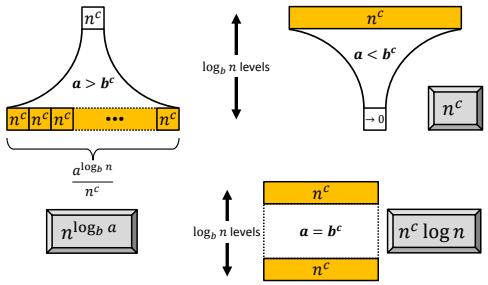
Master theorem: proof

- Case $a/b^c < 1$. Decreasing series. The sum is dominated by the first term ($k = 0$): $O(n^c)$.
- Case $a/b^c > 1$. Increasing series. The sum is dominated by the last term ($k = \log_b n$):

$$n^c \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \left(\frac{a^{\log_b n}}{(b^{\log_b n})^c}\right) = a^{\log_b n} = n^{\log_b a}$$

- Case $a/b^c = 1$. We have $O(\log n)$ terms all equal to $O(n^c)$.

Master theorem: visual proof



Master theorem: examples

Running time: $T(n) = a \cdot T(n/b) + O(n^c)$

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O(n^{\log_b a}) & \text{if } a > b^c \end{cases}$$

Algorithm	a	c	Runtime equation	Complexity
Power (x^y)	1	0	$T(y) = T(y/2) + O(1)$	$O(\log y)$
Binary search	1	0	$T(n) = T(n/2) + O(1)$	$O(\log n)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + O(n)$	$O(n \log n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + O(n)$	$O(n^2)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + O(n)$	$O(n^{\log_2 3})$

$b = 2$ for all the examples

Product multiplication

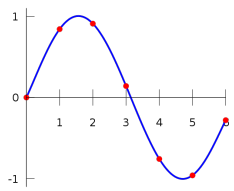
- Fundamental question:

  Can polynomials be multiplied efficiently when the degree is large?

- Answer: yes (FFT: Fast Fourier Transform)
- FFT is an essential algorithm for efficient signal analysis. The algorithm will not be explained in this course.

Polynomials: point-value representation

- **Fundamental Theorem (Gauss)**: A degree-n polynomial with complex coefficients has exactly n complex roots.
- **Corollary**: A degree-n polynomial $A(x)$ is uniquely identified by its evaluation at $n + 1$ distinct values of x.



Polynomial representation

$P(x) = x^3 - 2x^2 - 3x + 1$

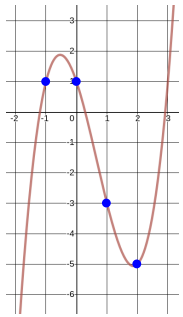
$P(x) = (1, -2, -3, 1)$

Coefficient representation

Evaluation ↓ ↑ Interpolation

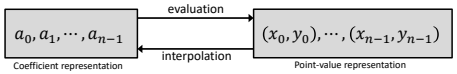
Point-value representation

$P(x) = \{(-1,1), (0,1), (1,-3), (2,-5)\}$



Conversion between both representations

representation	addition	multiplication	evaluation
coefficient	$O(n)$	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n)$	$O(n^2)$



Could we have an ***efficient*** algorithm to move from coefficient to point-value representation and vice versa?

↓

Fast Fourier Transform (FFT): $O(n \log n)$

Fourier series

- Periodic function $f(t)$ of period 1:

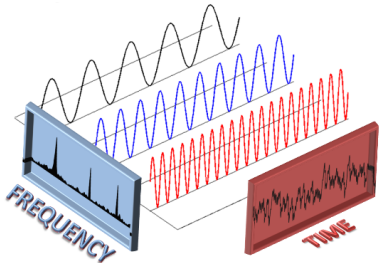
$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(2\pi n t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n t)$$

- Fourier coefficients:

$$a_n = 2\int_0^T f(t)\cos(2\pi n t)\,dt, \qquad b_n = 2\int_0^T f(t)\sin(2\pi n t)\,dt$$

- Fourier series is fundamental for signal analysis (to move from time domain to frequency domain, and vice versa)

Why Fourier Transform?



Divide & Conquer (II)

Jordi Cortadella and Jordi Petit
Department of Computer Science

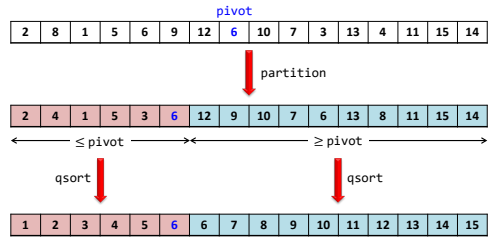
Examples

- Quick sort
- The selection problem
- The closest-points problem

Quick sort (Tony Hoare, 1959)

- Suppose that we know a number x such that one-half of the elements of a vector are greater than or equal to x and one-half of the elements are smaller than x.
  - Partition the vector into two equal parts ($n-1$ comparisons)
  - Sort each part recursively
- Problem: we do not know x.
- The algorithm also works no matter which x we pick for the partition. We call this number the **pivot**.
- **Observation:** the partition may be unbalanced.

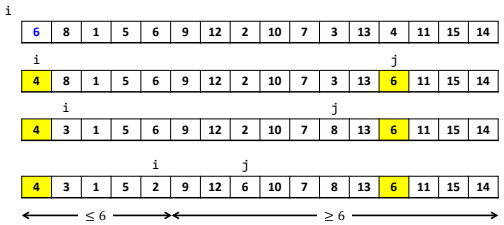
Quick sort: example



The key step of quick sort is the partitioning algorithm.

Question: how to find a good pivot?

Quick sort partition: example



Quick sort: Hoare's partition

```
def partition(a: list[T], left: int, right: int) -> int:
    """a[left..right]: segment to be sorted.
       Output: The left part has elements ≤ than the pivot.
       The right part has elements ≥ than the pivot.
       Returns the index of the last element of the left part
    """
    pivot = a[left]
    i, j = left-1, right+1
    while True:
        while True:  # find a[i] ≥ pivot
            i += 1
            if a[i] >= pivot: break
        while True:  # find a[j] ≤ pivot
            j -= 1
            if a[j] <= pivot: break
        if i >= j:
            return j
        a[i], a[j] = a[j], a[i]  # swap a[i], a[j]
```

The first swap locates the sentinels of the two innermost loops.
No need to check for indices out-of-bounds.

Quick sort: algorithm

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right].
       If right < 0, it sorts the whole list.
       The initial call can be invoked as quick_sort(a)
    """

    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left < right:
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)
```


Quick sort: hybrid approach

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right] partially, leaving unsorted
       chunks of size K (some break-even constant when
       compared to insertion sort).
       If right < 0, it sorts the whole list.
    """
    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left <= right - K: # K: size of the unsorted chunks
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)

def sort(a: list[T]) -> None:
    """Sorts a"""
    quick_sort(a)
    insertion_sort(a)
```

Observation: during `insertion_sort`, elements will never be shifted by more than **K** locations

Quick sort: complexity analysis

- The partition algorithm is $O(n)$.
- Assume that the partition is balanced:

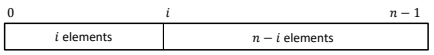
$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$

- Worst case runtime: the pivot is always the smallest element in the vector → $O(n^2)$
- Selecting a good pivot is essential. There are different strategies, e.g.,
  - Take the median of the first, last and middle elements
  - Take the pivot at random

Quick sort: complexity analysis

- Let us assume that x_i is the i-th smallest element in the vector.
- Let us assume that each element has the same probability of being selected as pivot.
- The runtime if x_i is selected as pivot is:

$$T(n) = \underset{\text{partition}}{n} + \underset{\text{qsort}}{T(i)} + \underset{\text{qsort}}{T(n-i)}$$



Possible values for i: $1 \ldots n-1$. Assumption: all values are equiprobable (random pivots)

Quick sort: complexity analysis

$$T(n) = n + \frac{1}{n-1}\sum_{i=1}^{n-1}(T(i) + T(n-i))$$

$$T(n) = n + \frac{1}{n-1}\sum_{i=1}^{n-1}T(i) + \frac{1}{n-1}\sum_{i=1}^{n-1}T(n-i)$$

$$T(n) = n + \frac{2}{n-1}\sum_{i=1}^{n-1}T(i) = 2n \cdot H(n)$$

$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$ is the Harmonic series, that has a simple approximation: $H(n) = \ln n + \gamma + O(1/n)$.
$\gamma = 0.577\ldots$ is Euler's constant. **[see the appendix]**

$$T(n) = 2n(\ln n + \gamma + O(1/n)) \in O(n \log n)$$

Quick sort: complexity analysis summary

- Runtime of quicksort:

$$T(n) = O(n^2)$$
$$T(n) = \Omega(n \log n)$$
$$T_{\text{avg}}(n) = O(n \log n)$$

- Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.
- Possible solution: use random pivots.

The selection problem

- Given a collection of N elements, find the k-th smallest element.
- Options:
  - Sort a vector and select the k-th location: $O(N \log N)$
  - Read k elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only k elements are maintained in the vector. Complexity: $O(kN)$. Why?

Quick sort

```
def quick_sort(a: list[T],
               left: int = 0, right: int = -1) -> None:
    """sorts a[left..right].
       If right < 0, it sorts the whole list.
       The initial call can be invoked as quick_sort(a)
    """

    if right < 0: # initial call (sort the whole list)
        right = len(a)-1

    if left < right:
        mid = partition(a, left, right)
        quick_sort(a, left, mid)
        quick_sort(a, mid+1, right)
```


Quick select

```
def quick_select(a: list[T], k: int,
                 left: int = 0, right: int = -1) -> T:
    """Returns the element at location k assuming
       a[left..right] would be sorted.
       Pre: left ≤ k ≤ right.
       Post: the elements of a have changed their locations.
       The initial call can be invoked as quick_select(a, k)
    """
    if right < 0: # initial call (use the whole list)
        right = len(a)-1

    if left == right:
        return a[left]

    mid = partition(a, left, right)
    if k <= mid:
        return quick_select(a, k, left, mid)
    return quick_select(a, k, mid+1, right)
```


Quick Select: complexity

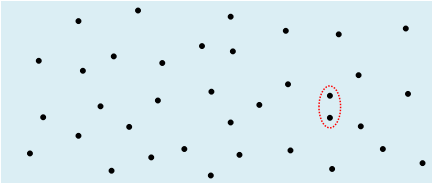
- Master theorem:

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a \quad (a < b^c) \\ O(n^c \log n) & \text{if } c = \log_b a \quad (a = b^c) \\ O(n^{\log_b a}) & \text{if } c < \log_b a \quad (a > b^c) \end{cases}$$

- Assume that the partition is balanced:
  - Quick sort: $T(n) = 2T(n/2) + O(n) = O(n \log n)$
  - Quick select: $T(n) = T(n/2) + O(n) = O(n)$
- The average linear time complexity can be achieved by choosing good pivots (similar strategy and complexity computation to quick_sort).

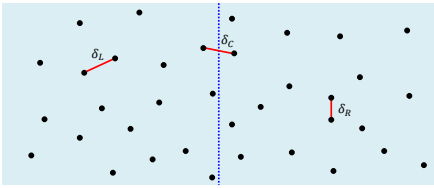
The Closest-Points problem

- **Input:** A list of n points in the plane
  $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs → $O(n^2)$
- We want an $O(n \log n)$ solution !



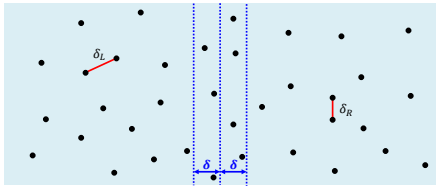
The Closest-Points problem

- We can assume that the points are sorted by the x-coordinate. Sorting the points is free from the complexity standpoint ($O(n \log n)$).
- Split the list into two halves. The closest points can be both at the left, both at the right or one at the left and the other at the right (center).
- The left and right pairs are easy to find (recursively).
  How about the pairs in the center?



The Closest-Points problem

- Let $\delta = min(\delta_L, \delta_R)$. We only need to compute δ_C if it improves δ.
- We can define a strip around de center with distance δ at the left and right. If δ_C improves δ, then the points must be within the strip.
- In the worst case, all points can still reside in the strip.
- But how many points do we really have to consider?



The Closest-Points problem

Let us take all points in the strip and sort them by the y-coordinate. We only need to consider pairs of points with distance smaller than δ.

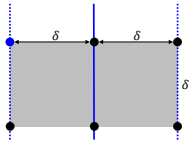
Once we find a pair (p_i, p_j) with y-coordinates that differ by more than δ, we can move to the next p_i.

```
for i in range(NumPointsInStrip):
  for j in range(i+1, NumPointsInStrip):

    if (p_i and p_j's y-coordinate differ by
        more than δ): break // Go to next p_i

    if dist(p_i,p_j) < δ:  δ = dist(p_i,p_j);
```

But, how many pairs (p_i, p_j) do we need to consider?

The Closest-Points problem

- For every point p_i at one side of the strip, we only need to consider points from p_{i+1}.



- The relevant points only reside in the $2\delta \times \delta$ rectangle below point p_i. There can only be 8 points at most in this rectangle (4 at the left and 4 at the right). Some points may have the same coordinates.

The Closest-Points problem: algorithm

- Sort the points according to their x-coordinates.
- Divide the set into two equal-sized parts.
- Compute the min distance at each part (recursively).
  Let δ be the minimal of the two minimal distances.
- Eliminate points that are farther than δ from the separation line.
- Sort the remaining points according to their y-coordinates.
- Scan the remaining points in the y order and compute the distances of each point to its 7 neighbors.

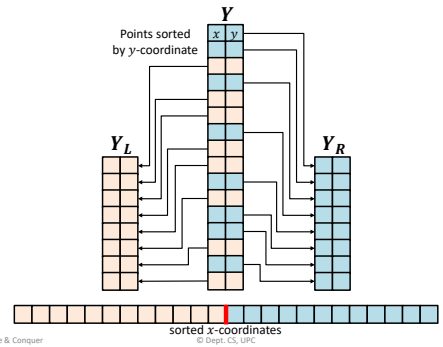
The Closest-Points problem: complexity

- Initial sort using x-coordinates: $O(n \log n)$.
  It comes for free.
- Divide and conquer:
  - Solve for each part recursively: $2T(n/2)$
  - Eliminate points farther than δ: $O(n)$
  - Sort remaining points using y-coordinates: $O(n \log n)$
  - Scan the remaining points in y order: $O(n)$

$$T(n) = 2T(n/2) + O(n) + O(n \log n) = O(n \log^2 n)$$

- Can we do it in $O(n \log n)$? Yes, we need to sort by y in a smart way.

Partitioning and sorting the points (visually)



The Closest-Points problem: complexity

- Let Y a vector with the points sorted by the y-coordinates. This can be done initially for free.
- Each time we partition the set of points by the x-coordinate, we also partition Y into two sorted vectors (using an "*unmerging*" procedure with linear complexity)

```
Y_L = Y_R = ∅   // Initial lists of points
for each p_i ∈ Y in ascending order of y:
    if p_i is at the left part: Y_L.push_back(p_i)
    else: Y_R.push_back(p_i)
```

- Now, sorting the points by the y-coordinate at each iteration can be done in linear time, and the problem can be solved in $O(n \log n)$

Subtract and Conquer

- Sometimes we may find recurrences with the following structure:

$$T(n) = a \cdot T(n-b) + O(n^c)$$

- Examples:

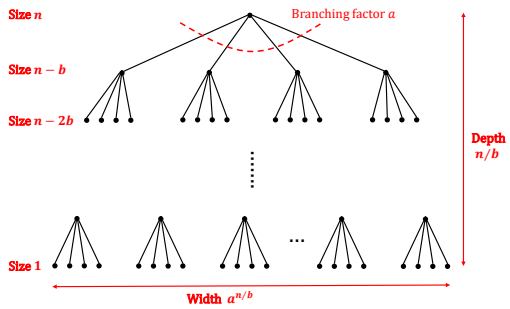
$$\text{Hanoi}(n) = 2 \cdot \text{Hanoi}(n-1) + O(1)$$

$$\text{Sort}(n) = \text{Sort}(n-1) + O(n)$$

- *Muster* theorem:

$$T(n) = \begin{cases} O(n^c) & \text{if } a < 1 \quad \text{(never occurs)} \\ O(n^{c+1}) & \text{if } a = 1 \\ O(n^c a^{n/b}) & \text{if } a > 1 \end{cases}$$

Muster theorem: recursion tree



Muster theorem: examples

- **Hanoi**: $T(n) = 2T(n-1) + O(1)$

  We have $a = 2$ and $c = 0$, thus $T(n) = O(2^n)$.

- **Selection sort** (recursive version):
  - Select the min element and move it to the first location
  - Sort the remaining elements

$$T(n) = T(n-1) + O(n) \qquad (a = c = 1)$$

Thus, $\quad T(n) = O(n^2)$

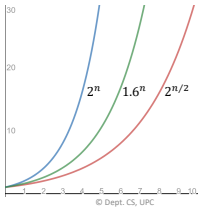
Muster theorem: examples

Fibonacci: $T(n) = T(n-1) + T(n-2) + O(1)$

We can compute bounds:

$$2T(n-2) + O(1) \le T(n) \le 2T(n-1) + O(1)$$

Thus, $\quad O(2^{n/2}) \le T(n) \le O(2^n)$



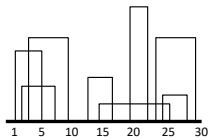
EXERCICES

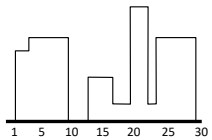
The skyline problem

Given the exact locations and shapes of several rectangular buildings in a city, draw the skyline (in two dimensions) of these buildings, eliminating hidden lines (source: Udi Manber, *Introduction to Algorithms*, Addison-Wesley, 1989).

Input:
(1,**11**,5) (2,**6**,7) (3,**13**,9) (12,**7**,16) (14,**3**,25)
(19,**18**,22) (23,**13**,29) (24,**4**,28)

Output:
(1,**11**,3,**13**,9,**0**,12,**7**,16,**3**,19,**18**,22,**3**,23,**13**,29,**0**)
(numbers in boldface represent heights)





Describe (in natural language) two different algorithms to solve the skyline problem:
- By induction: assume that you know how to solve it for $n-1$ buildings.
- Using Divide&Conquer: solve the problem for $n/2$ buildings and combine.

Analyze the cost of each solution.

A, B or C?

Suppose you are choosing between the following three algorithms:

- Algorithm **A** solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm **B** solves problems of size n by recursively solving two subproblems of size $n-1$ and them combining the solutions in constant time.
- Algorithm **C** solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which one would you choose?

Source: Dasgupta, Papadimitriou and Vazirani, *Algorithms*, McGraw-Hill, 2008.

Crazy sorting

Let $T[i..j]$ be a vector with $n = j - i + 1$ elements. Consider the following sorting algorithm:

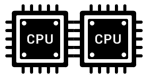
a) If $n \leq 2$ the vector is easily sorted (constant time).

b) If $n \geq 3$, divide the vector into three intervals $T[i..k-1]$, $T[k..l]$ and $T[l+1..j]$, where $k = i + \lfloor n/3 \rfloor$ and $l = j - \lfloor n/3 \rfloor$. The algorithm recursively sorts $T[i..l]$, then it sorts $T[k..j]$, and finally sorts $T[i..l]$.

- Prove the correctness of the algorithm.
- Analyze the asymptotic complexity of the algorithm (give a recurrence of the runtime and solve it).

VLSI chip testing

Professor Diogenes has n supposedly identical VLSI (Very-Large-Scale Integration) chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:



Chip A says	Chip B says	Conclusion
B is good	A is good	Both are good, or both are bad
B is good	A is bad	At least one is bad
B is bad	A is good	At least one is bad
B is bad	A is bad	At least one is bad

a. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

b. Consider the problem of finding a single good chip among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Source: Cormen, Leiserson and Rivest, Introduction to Algorithms, The MIT Press, 1989

Breaking into pieces

Let us assume that f is $\Theta(1)$ and g has a runtime proportional to the size of the vector it has to process, i.e., $\Theta(j - i + 1)$. What is the asymptotic cost of A and B as a function of n? (n is the size of the vector).

If both functions do the same, which one would you choose?

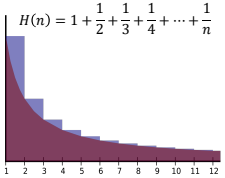
```
def a(v: list[float], i: int, j: int) -> float:
    if i < j:
        x = f(v, i, j)
        m = (i+j)//2
        return a(v, i, m-1) + a(v, m, j) + a(v, i+1, m) + x
    else:
        return v[i]


def b(v: list[float], i: int, j: int) -> float:
    if i < j:
        x = g(v, i, j)
        m1 = i + (j-i+1)//3
        m2 = i + (j-i+1)*2//3
        return b(v, i, m1-1) + b(v, m1, m2-1) + b(v, m2, j) + x
    else:
        return v[i]
```


APPENDIX

Logarithmic identities

$b^{\log_b a} = \log_b b^a = a$
$\log_b(xy) = \log_b x + \log_b y$
$\log_b \frac{x}{y} = \log_b x - \log_b y$
$\log_b x^c = c \log_b x$
$\log_b x = \frac{\log_c x}{\log_c b}$
$x^{\log_b y} = y^{\log_b x}$

$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$



$$\gamma = \lim_{n\to\infty}\left(-\ln n + \sum_{k=1}^{n}\frac{1}{k}\right) \Rightarrow \sum_{k=1}^{n}\frac{1}{k} \in \Theta(\log n)$$

$\gamma = 0.5772\ldots$ (Euler-Mascheroni constant) (Harmonic series)

Full-history recurrence relation

$$\boxed{T(n) = n + \frac{2}{n-1}\sum_{i=1}^{n-1} T(i)}$$

A recurrence that depends on all the previous values of the function.

$$(n-1)T(n) = n(n-1) + 2\sum_{i=1}^{n-1} T(i), \qquad nT(n+1) = (n+1)n + 2\sum_{i=1}^{n} T(i)$$

$$nT(n+1) - (n-1)T(n) = (n+1)n - n(n-1) + 2T(n) = 2n + 2T(n)$$

$$T(n+1) = \frac{n+1}{n}T(n) + 2 \qquad \Rightarrow \qquad T(n) = \frac{n}{n-1}T(n-1) + 2$$

$$T(n) = 2 + \frac{n}{n-1}\left(2 + \frac{n-1}{n-2}\left(2 + \frac{n-2}{n-3}\left(\cdots\frac{2}{1}\right)\right)\right)$$

$$T(n) = 2\left(1 + \frac{n}{n-1} + \frac{n}{n-1}\frac{n-1}{n-2} + \frac{n}{n-1}\frac{n-1}{n-2}\frac{n-2}{n-3} + \cdots + \frac{n}{n-1}\frac{n-1}{n-2}\frac{n-2}{n-3}\cdots\frac{2}{1}\right)$$

$$T(n) = 2\left(1 + \frac{n}{n-1} + \frac{n}{n-2} + \frac{n}{n-3} + \cdots + \frac{n}{1}\right) = 2n\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1\right)$$

$$\boxed{T(n) = 2n \cdot H(n) \in \Theta(n \log n)}$$

Muster theorem: proof

- Expanding the recursion (assume that $f(n)$ is $O(n^c)$)

$$\begin{aligned} T(n) &= aT(n-b) + f(n) \\ &= a\big(aT(n-2b) + f(n-b)\big) + f(n) \\ &= a^2T(n-2b) + af(n-b) + f(n) \\ &= a^3T(n-3b) + a^2f(n-2b) + af(n-b) + f(n) \end{aligned}$$

- Hence:

$$T(n) = \sum_{i=0}^{n/b} a^i \cdot f(n-ib)$$

- Since $f(n-ib)$ is in $O\big((n-ib)^c\big)$, which is in $O(n^c)$, then

$$T(n) = O\left(n^c \sum_{i=0}^{n/b} a^i\right)$$

- The proof is completed by this property:

$$\sum_{i=0}^{n/b} a^i = \begin{cases} O(1), & \text{if } a < 1 \\ O(n), & \text{if } a = 1 \\ O\big(a^{n/b}\big), & \text{if } a > 1 \end{cases}$$

Exception handling

Jordi Cortadella and Jordi Petit
Department of Computer Science

Quadratic equation

```python
import math

def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    a, b, c = [float(x) for x in open(file).readline().split()]
    discrim = b*b - 4*a*c
    if discrim < 0:
        print('The equation has no real roots!')
    elif discrim == 0:
        root = -b / (2*a)
        print(f'There is a double root at {root}')
    else:
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')


quadratic_equation('coef.txt')
```

Source: John Zelle, *Python programming: an introduction to Computer Science*, Franklin, Beedle & Associates, 2004.

Exception handling

- Very often programs are infested with **if** statements to detect errors and special cases that are produced at runtime.
- It is very difficult to predict all the abnormal situations that can occur during the execution of a program.
- Exception handling is a mechanism that allows to catch errors without messing up the code of the algorithm.
- When an error occurs, and exception is raised, the flow of execution is broken and transferred to an exception handler.
- Examples of exceptions: invalid input data, division by zero, square root of a negative number, overflow, file not found, index out-of-bounds, violation of the pre-condition of a function, etc.

Exceptions in Python

```python
try:
    <body>
except <ErrorType>:
    <handler>
```

```python
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    try:
        a, b, c = [float(x) for x in open(file).readline().split()]
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')
    except ValueError:
        print('No real roots')
```


Exceptions in Python

What if the exception is not caught?

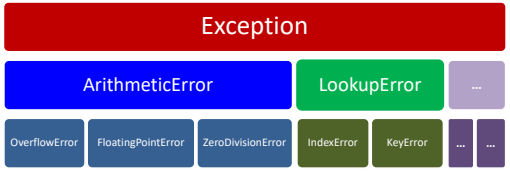
```python
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    a, b, c = [float(x) for x in open(file).readline().split()]
    discRoot = math.sqrt(b*b - 4*a*c)
    root1 = (-b + discRoot) / (2*a)
    root2 = (-b - discRoot) / (2*a)
    print(f'The solutions are: {root1}, {root2}')
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "quadratic.py", line 13, in ?
    discRoot = math.sqrt(b*b - 4*a*c)
ValueError: math domain error
```

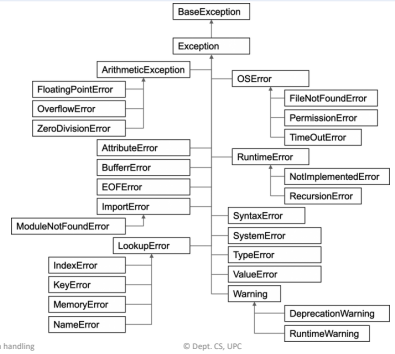
Unhandled exceptions are finally caught by the Python interpreter

Exception hierarchy

- When an exception is raised, the control is transferred to the innermost **try-except** statement than handles the exception.
- There is an exception hierarchy:



Exception hierarchy



Handling multiple exceptions

```python
def quadratic_equation(file: str) -> None:
    """Reads three coefficients from a file and computes
       the real roots of a quadratic equation"""
    try:
        a, b, c = [float(x) for x in open(file).readline().split()]
        discRoot = math.sqrt(b*b - 4*a*c)
        root1 = (-b + discRoot) / (2*a)
        root2 = (-b - discRoot) / (2*a)
        print(f'The solutions are: {root1}, {root2}')
    except ValueError as error:
        if str(error) == 'math domain error':
            print('No real roots.')
        else:
            print('Wrong format for coefficients.')
    except FileNotFoundError:
        print(f'File {file} could not be found.')
    except:  # something else, e.g., division by zero
        print('Something went wrong, sorry!')
```


Raising exceptions

```python
# The program itself can generate exceptions

if x < 0:
    raise ValueError('No negative numbers allowed')

if not isinstance(s, str):
    raise TypeError('A string is expected')
```

Python allows user-defined exceptions using classes (not explained here), e.g.,

```python
class MyError(Exception):
…
```


Assertions

- Assertions are sanity checks used to test the program. They are often used to check the pre-conditions of the functions and detect internal errors of the program, e.g.,

```python
def Kelvin2Fahrenheit(temperature: float) -> float:
    assert temperature >= 0, 'Colder than absolute zero!'
    return (Temperature-273)*1.8+32
```

- Assertions are not exceptions, but they raise an **`AssertionError`** that can be caught by an exception handler.

Conclusions

- Use exceptions to maintain clean code and catch abnormal program executions at runtime
- Use assertions for internal sanity-checks
- Print informative error messages:
  - Non-informative message: *"Unexpected error"*
  - Informative message: *"File abc.txt not found"*

Iterating over data with Python

Jordi Cortadella and Jordi Petit
Department of Computer Science

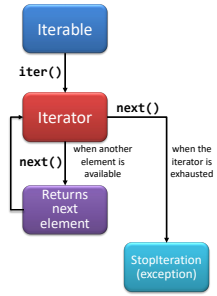
Outline

- Iterables and iterators
- Generators
- Comprehensions
- Enumerate and Zip
- Map, Filter and Reduce
- λ-functions

ITERABLES AND ITERATORS

Iterables

- ***Iterables*** are containers of data in which we can iterate to obtain elements one by one
- Lists, tuples, sets, dictionaries, strings, etc. are iterables
- ***Iterators*** are objects used to iterate over iterables
- Two important functions:
  - `iter()`: creates an iterator from an iterable
  - `next()`: returns the next item



Iterables: example

```
>>> lst = [1, 2, 3]  # lst is an iterable
>>> it = iter(lst)  # it is an iterator
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

`next()` raises a StopIteration exception when no more items are available

`next(it, default)`

```
it = iter(some_iterable)  # Creates an iterable

# next(it, default) does not raise any exception.
# Instead, it returns the default value.
v = next(it, None)
while v is not None:
    do_something(v)
    v = next(it, None)


# Equivalent code
for v in it:
    do_something(v)
```


GENERATORS

Designing data pipelines

- Big data systems often have to process long streams of data with pipelines chaining different operations.
- How to store the data between operations?

Designing data pipelines

- Big data systems often have to process long streams of data with pipelines chaining different operations.
- How to store the data between operations?

Generators

- A mechanism to avoid storage of large amounts of data.
- Generators are lazy iterators that do not store the whole data structures in memory.

Generators: example

```
>>> def natural_numbers():
...     n = 0
...     while True:
...         yield n
...         n += 1
...
>>> gen = natural_numbers()
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
...
```

```
>>> for i in natural_numbers():
...     if is_prime(i):
...         print(i)
...
2
3
5
7
11
...
672131
672137
672143
^CTraceback (...)
...
KeyboardInterrupt
>>>
```


Hamming numbers

- Hamming numbers are those numbers whose only prime divisors are 2, 3, and 5.
- Examples:
  - 20 is a Hamming number ($2^2 \cdot 3^0 \cdot 5^1$)
  - 21 is not a Hamming number ($3 \cdot 7$)
- Exercise: design a program that prints the n smallest Hamming numbers.
- Strategy: for every Hamming number $2^i \cdot 3^j \cdot 5^k$, we can generate three new numbers by increasing each one of the exponents.
- How to generate them in ascending order?

Hamming numbers

- Generating the next Hamming numbers:

$$2^i \cdot 3^j \cdot 5^k$$

$$2^{i+1} \cdot 3^j \cdot 5^k \qquad 2^i \cdot 3^{j+1} \cdot 5^k \qquad 2^i \cdot 3^j \cdot 5^{k+1}$$

- How to generate them in ascending order?
  - Use a priority queue to store the pending numbers
- How to avoid repetitions?
  - Remember the last delivered number

Hamming numbers: simulation

Value	Priority Queue
	1
1	2 3 5
2	3 4 5 6 10
3	4 5 6 6 9 10 15
4	5 6 6 8 9 10 12 15 20
5	6 6 8 9 10 10 12 15 15 20 25
6	6 8 9 10 10 12 12 15 18 20 25 30
6	8 9 10 10 12 12 15 18 20 25 30
8	9 10 10 12 12 15 16 18 20 24 25 30 40
9	10 10 12 12 15 16 18 18 20 24 25 27 30 40 45
10	10 12 12 15 16 18 18 20 20 24 25 27 30 30 40 45 50
⋮	⋮

Hamming numbers

```
import heapq
from typing import Iterator


def hamming_numbers() -> Iterator[int]:
    """Generates all Hamming numbers in ascending order"""
    pq = [1]  # priority queue storing Hamming numbers
    prev = 0  # the last delivered number
    while True:
        value = heapq.heappop(pq) # get the smallest number
        if value != prev:  # avoid repetitions
            yield value  # deliver the value and wait (lazy)
            prev = value
            for x in 2, 3, 5: # generate new numbers
                nxt = x*value
                heapq.heappush(pq, nxt)
```


Hamming numbers

```
def main(n: int) -> None:
    """Test to print first n hamming numbers"""
    hammings = hamming_numbers() # the generator
    for _ in range(n):
        print(next(hammings))


if __name__ == '__main__':
    main(20)
```

```
1
2
3
4
5
6
8
9
10
12
15
16
18
20
24
25
27
30
32
36
```

Merging sequences

- Functions can receive iterators as parameters and generate iterators as results
- Let us design a function that merges two sorted iterators and generated a sorted iterator

Merging sequences: typing

```python
from typing import Iterator, TypeVar, Protocol

# This is an abstract class that contains the __lt__
# operator (<). No need to implement it.
class Comparable(Protocol):
    def __lt__(self: 'T', other: 'T') -> bool: ...

# This is a generic type. The bound attribute indicates
# that the type must contain the operators of Comparable.
T = TypeVar('T', bound=Comparable)

# The merge function dealing with sequences of elements
# that are "Comparable" (i.e., the type contains the
# the operator <).
def merge(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator by merging them"""
    ...
```


Merging sequences: code

```python
def merge(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator by merging them"""
    x, y = next(a, None), next(b, None)
    while x is not None and y is not None:
        if x < y:
            yield x
            x = next(a, None)
        else:
            yield y
            y = next(b, None)

    if x is not None:
        yield x
        yield from a  # delivers values from another iterator

    if y is not None:
        yield y
        yield from b
```


COMPREHENSIONS

Comprehensions

- Set builder notation. Example:

$$S = \{x^2 \mid x < 1000, x \text{ is prime}\}$$

- Conventional Python using a `for` loop:

```python
s = {}
for x in range(1000):
    if is_prime(x):
        s.add(x**2)
```

- Using comprehensions:

```python
s = {x**2 for x in range(1000) if is_prime(x)}
```


Nested comprehensions

```python
# Given a list of words, create a dictionary with the key-value
# pairs <word: number of vowels>
words = ['cat', 'kangaroo', 'lion', 'dog', 'hippopotamus']

# We can use s.count(x) to count the number of occurrences
# of x in the string s, e.g., 'kangaroo'.count('o') is 2
vowels = {w: sum(w.count(x) for x in 'aeiou') for w in words}
print(vowels)
{'cat': 1, 'kangaroo': 4, 'lion': 2, 'dog': 1, 'hippopotamus': 5}

# Let us print a list of the words with more than 3 vowels
print([w for w in vowels.keys() if vowels[w] > 3])
['kangaroo', 'hippopotamus']
```


Creating matrices with comprehensions

```python
# Let us create a 4x4 identity matrix
matrix = [[0]*4]*4
for i in range(4):
    matrix[i][i] = 1

# Surprise! What's wrong?
print(matrix)
[[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]

# Let us use comprehensions
matrix = [[1 if i==j else 0 for j in range(4)] for i in range(4)]
print(matrix)
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

# How to create a zero matrix with n rows and m columns
matrix = [[0]*m for _ in range(n)]
```


Generator expressions: example

```python
>>> import sys
>>> # A list comprehension
>>> squares_lc = [i**2 for i in range(10**6)]
>>> # It generates a long list (larger than 8Mb)
>>> sys.getsizeof(squares_lc)
8448728
>>>
>>> # But we can also create a generator using (…)
>>> squares_gc = (i**2 for i in range(10**6))
>>> sys.getsizeof(squares_gc)
104
>>> # and we can iterate over the generator
>>> for n in squares_gc:
...     if is_prime(n+1):
...         print(n+1)
...
```


ENUMERATE AND ZIP

enumerate

```python
# Different ways of printing indices and values
lst = [x**2 for x in range(100)]

for i in range(len(lst)):
    print(i, lst[i])

i = 0
for v in lst:
    print(i, v)
    i += 1

for i, v in enumerate(lst):
    print(i, v)

# It also works for generators!
lst_gen = (x**2 for x in range(100))
for i, v in enumerate(lst_gen):
    print(i, v)
```

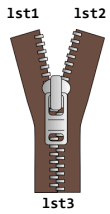

Hamming numbers

```python
def main(n: int) -> None:
    """Test to print first n hamming numbers"""
    for i, x in enumerate(hamming_numbers()):
        if i == n:
            break
        print(x)


if __name__ == '__main__':
    main(20)
```

```
1
2
3
4
5
6
8
9
10
12
15
16
18
20
24
25
27
30
32
36
```


zip

```python
>>> # Zipping lists
>>> lst1 = [x**2 for x in range(100)]
>>> lst2 = [2*x for x in range(100)]
>>> lst3 = zip(lst1, lst2)
>>> # lst3 is an iterator!
>>> print(lst3)
<zip object at 0x7fd8d8beacc0>
>>> for x, y in lst3:
...     print(x, y)
...
0 0
1 2
4 4
9 6
16 8
25 10
36 12
49 14
```



zipping and unzipping

```python
>>> # Let us zip two lists
>>> letters = ['a', 'b', 'c', 'd']
>>> numbers = [1, 2, 3, 4]
>>> ln_zip = zip(letters, numbers)
>>> list_ln = list(ln_zip)
>>> print(list_ln)
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
# Now we can unzip the list of tuples
>>> lett, numb = zip(*list_ln)
>>> print('letters =', lett)
letters = ('a', 'b', 'c', 'd')
>>> print('numbers =', numb)
numbers = (1, 2, 3, 4)
```


MAP, FILTER AND REDUCE

map, filter and reduce

- Loops, comprehensions and generators are techniques used to process data in iterable objects.
- The functions **`map()`**, **`filter()`** and **`reduce()`** provide a functional programming approach to achieve similar goals.
- They can be applied to any iterable object (list, tuple, set, …)
- These functions can provide a very elegant solution to compute expressions like this:

$$\sum_{0 \le i < n,\ \text{is\_prime}(i)} i^2$$

map, filter, reduce: auxiliary functions

```python
def square(x: int) -> int:
    return x*x


def add(x: int, y: int) -> int:
    return x + y


def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    d = 2
    while d*d <= n:
        if n%d == 0:
            return False
        d += 1
    return True
```

map

```
lst = [1, 2, 3, 4, 5, 6]

# map creates an iterator that applies a function
# to all elements of the iterable object
result = map(square, lst)

list(result)

# Output: [1, 4, 9, 16, 25, 36]
```


filter

```
# filter creates an iterator that selects the
# elements that satisfy the filtering condition
result = filter(is_prime, range(30))

list(result)

# Output: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```


reduce

```
from functools import reduce

# reduce visits all elements and executes a
# function that "accumulates" their values
result = reduce(add, range(10))
result
# Output: 45

# An initial value can also be specified
result = reduce(add, range(10), 5)
result
# Output: 50
```


Back to our problem

$$\sum_{0 \le i < n,\ \text{is\_prime}(i)} i^2$$


```
n = 10**7
r = reduce(add, map(square, filter(is_prime, range(n))))
print(r)
# Output: 21113978675102768574
```

Important: no intermediate lists are generated. Very low storage is required (< 1000 bytes). By using lists to store the intermediate results, about 400Mb of storage would be required.

Pythonic Boolean reductions

`any()` and **`all()`** are particular cases of reduce functions with Boolean results.

```
>>> numbers = [2, 3, 7, 11, 13, 23]
>>> all(is_prime(x) for x in numbers)
True
>>> all(x%2 == 1 for x in numbers)
False
>>> any(6 < x < 12 for x in numbers)
True
>>> words = ['cat', 'kangaroo', 'lion', 'dog', 'hippopotamus']
>>> all(len(w) > 10 for w in words)
False
>>> any(len(w) > 10 for w in words)
True
>>> all(len(w) < 15 for w in words)
True
>>> any(w[0] == 'h' for w in words)
True
```


Pythonic numerical reductions

`sum()`, **`min()`** and **`max()`** are particular cases of reduce functions with numerical parameters.

```
>>> numbers = [5, 8, -2, 6, 0]
>>> sum(numbers)
17
>>> sum(numbers, start=10)  # with an initial value of the sum
27
>>> min(numbers)
-2
>>> max(numbers)
8
>>> sum(numbers)/len(numbers)  # mean
3.4
>>> a, b = [3, -4, 2], [1, 3, -1]  # two vectors
>>> sum([x*y for x, y in zip(a, b)])  # dot product of a and b
-11
>>> # a more memory-efficient solution, why?
>>> sum(x*y for x, y in zip(a, b))
-11
```

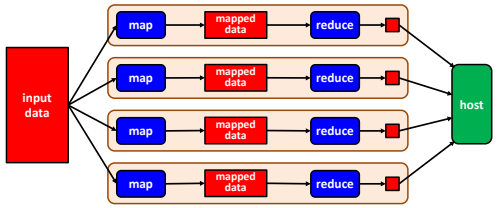

Generate or list?

```
import random, sys, time

n = 200_000_000
a = [random.random() for _ in range(n)]
b = [random.random() for _ in range(n)]

tinit = time.perf_counter()
gen = (x*y for x,y in zip(a,b))
sum_gen = sum(gen)
time_gen = time.perf_counter() - tinit
size_gen = sys.getsizeof(gen)

tinit = time.perf_counter()
lst = [x*y for x,y in zip(a,b)]
sum_lst = sum(lst)
time_lst = time.perf_counter() - tinit
size_lst = sys.getsizeof(lst)

print(f'time: gen={time_gen:.2f} secs, list={time_lst:.2f} secs')
print(f'memory: gen={size_gen} bytes, list={size_lst:,} bytes')
```

```
time: gen=13.51 secs, list=12.32 secs
memory: gen=104 bytes, list=1,693,045,240 bytes
```


MapReduce

A programming model for big data sets using parallel, distributed algorithms



λ-FUNCTIONS

λ-functions

- Lambda functions are anonymous functions that receive parameters and return expressions
- Syntax:

  `lambda parameters: expression`

  is equivalent to:

  ```
  def anonymous(parameters):
      return expression
  ```

- Examples:

  ```
  lambda x: x*x
  lambda x, y: x+y
  ```


Using λ-functions in map/filter/reduce

$$\sum_{0 \le i < n,\ \text{is\_prime}(i)} i^2$$

```
r = reduce(add, map(square, filter(is_prime, (range(n)))))

# Using λ-functions
r = reduce(lambda x, y: x+y,
           map(lambda x: x*x, filter(is_prime, range(n)))
           )

# Using generators
r = sum(i*i for i in range(n) if is_prime(i))
```


Conclusions

- Processing long streams of data is one of the main tasks of big data systems. Memory storage is one of the critical resources
- When designing data pipelines exploit lazy evaluation mechanisms to generate data upon demand and avoid unnecessary data storage

EXERCISES

Comprehensions

- Create a dictionary where the keys are the numbers of a list and the values are the highest one-digit divisor of each number
- Create a list with all positive numbers smaller than n that are divisible by some number included in a list called `divisors`
- Given a rectangular matrix (list of lists), calculate its transpose using list comprehensions

Generating the Fibonacci series

- Design a generator of the Fibonacci series
- Given a list of divisors, design a generator that generates the Fibonacci numbers that are divisible by all divisors of the list. Example:

```
divisors = [3, 5, 7, 11]
gen = (…)  # design the generator
for x in gen:
    print(x)

Output:
0
102334155
23416728348467685
53583592549909666408718 40
12261325953941882930001747 02095995
...
```


Intersection of sequences

Implement the function intersect with the following specification:

```
from typing import Iterator, TypeVar, Protocol

class Comparable(Protocol):
    def __lt__(self: 'T', other: 'T') -> bool: ...

T = TypeVar('T', bound=Comparable)

def intersect(a: Iterator[T], b: Iterator[T]) -> Iterator[T]:
    """reads two sorted iterators and generates a sorted
       iterator with only the common elements"""
    ...

# Example:
#     a = [1, 3, 5, 5, 6, 7, 10, 13, 16, 18]
#     b = [2, 5, 5, 8, 13, 13, 15, 16, 20]
#     output: [5, 13, 16]
```

Farey sequence

- The Farey sequence of order n is the sequence of completely reduced fractions between 0 and 1 with denominators less than or equal to n, arranged in ascending order. Example:

$$F_5 = \left\{\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}\right\}$$

- Design the generator **`farey(n)`** that generates the Farey sequence of order n:

```
def farey(n: int) -> Iterator[tuple[int, int]]:
```

- Write Python expressions to calculate:
  - the sum of the elements of F_n
  - the number of elements of F_n

Hint: The next element of the Farey sequence can be calculated using only the two previous elements (find the rule in Wikipedia!)

Filter/reduce pipeline

Design two versions of the following function using a filter-reduce pipeline:

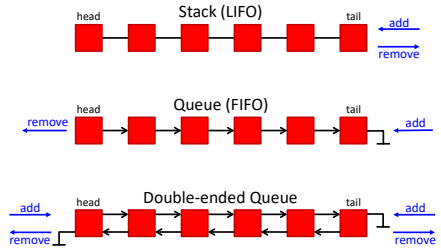
- One version with auxiliary functions
- One version with lambda functions

```
from dataclasses import dataclass
from typing import Iterable
from functools import reduce

@dataclass
class Person:
    name: str
    age: int
    salary: float

def avg_salary(people: Iterable[Person],
               min_age: int, max_age: int) -> float:
    """Returns the average salary of the people with age
       between min_age and max_age"""
```

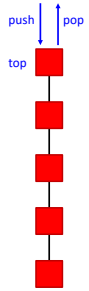
Linear Containers

Jordi Cortadella and Jordi Petit
Department of Computer Science

Linear containers



Stack



- Elements are removed in reversed order of insertion (Last-In-First-Out)
- A stack can be simply implemented with an array, vector or list (adding/removing elements to/from the last location)
- Typical applications:
  - Check balanced parenthesis
  - Backtracking
  - Activation records (function calls)
  - Store actions to "undo" them later

Evaluation of postfix expressions

- This is an infix expression. What's his value? 42 or 144?

  8 * 3 + 10 + 2 * 4

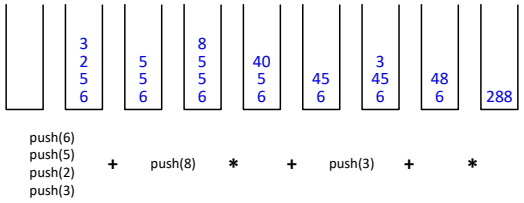
- It depends on the operator precedence. For scientific calculators, * has precedence over +.
- Postfix (reverse Polish notation) has no ambiguity:

  8 3 * 10 + 2 4 * +

- Postfix expressions can be evaluated using a stack:
  - each time an operand is read, it is pushed on the stack
  - each time an operator is read, the two top values are popped and operated. The result is push onto the stack

Evaluation of postfix expressions: example

6 5 2 3 + 8 * + 3 + *



From infix to postfix

a + b * c + (d * e + f) * g

↓

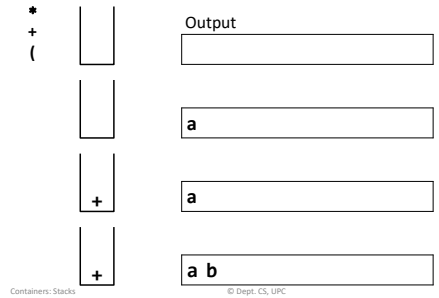
a b c * + d e * f + g * +

Algorithm:

- When an operand is read, write it to the output.
- If we read a right parenthesis, pop the stack writing symbols until we encounter the left parenthesis.
- For any other symbol, i.e., + * (, pop entries and write them until we find an entry with lower priority. After popping, push the symbol onto the stack. Exception: (can only be removed when finding a).
- When the end of the input is reached, all symbols in the stack are popped and written onto the output.

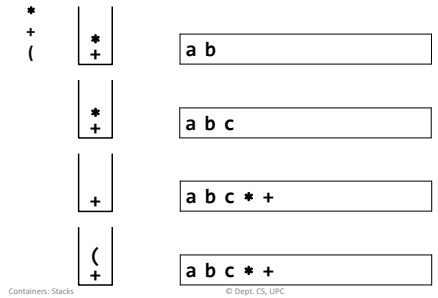
From infix to postfix

a + b * c + (d * e + f) * g



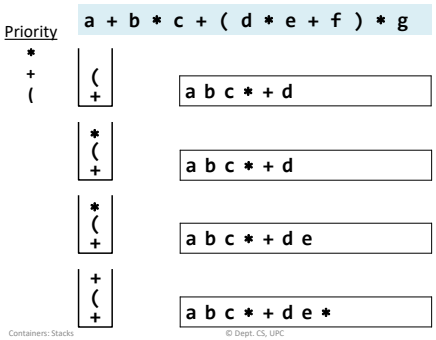
From infix to postfix

a + b * c + (d * e + f) * g



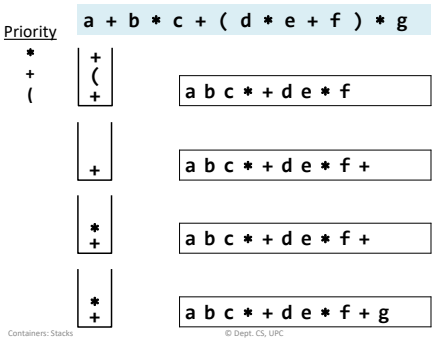
From infix to postfix

a + b * c + (d * e + f) * g



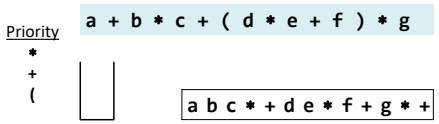
From infix to postfix

a + b * c + (d * e + f) * g



From infix to postfix

a + b * c + (d * e + f) * g

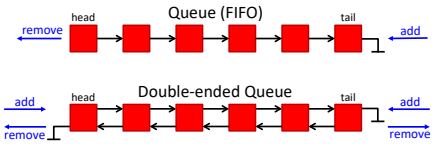


Complexity: $O(n)$

Suggested exercise:

- Add substraction (same priority as addition) and division (same priority as multiplication).

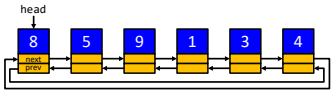
Queues



Queues are usually implemented using references to objects (also called pointers in C/C++). These references allow moving left/right and iterating over the queue.

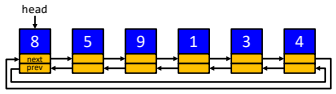
We will explain a toy implementation of a double-ended queue (deque), with the basic functionality to add/remove elements and iterate over them.

Deque



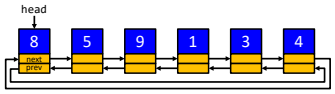
- Implemented as a circular queue with a reference to the head
- Elements can be appended/removed to/from the head or tail
- Operations:
  - `len(q), q.append(x), q.appendleft(x), q.pop(), q.popleft()`
  - Access to the i-th element (`q[0], q[1],q[-1], q[-2],…`)
  - Iterators: `for x in q:`

Deque node



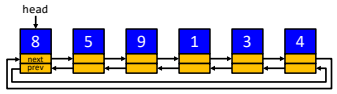

```python
from dataclasses import dataclass, field
from typing import TypeVar, Generic, Iterable

T = TypeVar('T')  # Generic type for the deque

@dataclass
class Node(Generic[T]):
    """Internal node of the deque"""
    data: T  # information stored in the node
    next: 'Node[T]' = field(init=False)  # next in the queue
    prev: 'Node[T]' = field(init=False)  # previous in the queue
```


Deque attributes



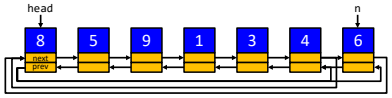

```python
class Deque(Generic[T]):
    """Class to represent a double-ended queue"""

    # Attributes of the class
    _head: Node[T]  # reference to the head of the queue
    _n: int  # number of elements in the queue
```


Deque: __init__ and __len__



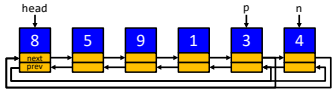

```python
class Deque(Generic[T]):

                    ⋮

    def __init__(self, it: Iterable[T] = list()) -> None:
        """Constructor: initialize from iterable"""
        self._n = 0
        for x in it:
            self.append(x)

    def __len__(self) -> int:
        """Length of the queue"""
        return self._n
```

Deque: append



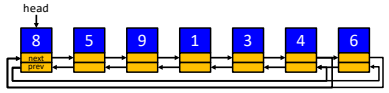

```
class Deque(Generic[T]):
                    ⋮
    def append(self, x: T) -> None:
        """Add element x to the tail of the queue"""
        n = Node(x)
        if self._n == 0:
            self._head = n.next = n.prev = n
        else:
            n.next = self._head
            n.prev = self._head.prev
            n.prev.next = self._head.prev = n
        self._n += 1
```


Deque: pop



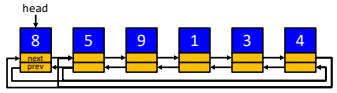

```
class Deque(Generic[T]):
                    ⋮
    def pop(self) -> T:
        """Returns and removes the last element of the queue"""
        if len(self) == 0:
            raise IndexError
        self._n -= 1
        n = self._head.prev
        if self._n > 0:
            p = n.prev
            p.next = self._head
            self._head.prev = p
        return n.data
```


Deque: appendleft



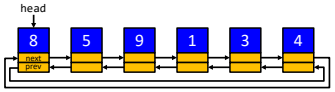

```
class Deque(Generic[T]):
                    ⋮
    def appendleft(self, x: T) -> None:
        """Add element x to the head of the queue"""
        self.append(x)
        self._head = self._head.prev
```


Deque: popleft




```
class Deque(Generic[T]):
                    ⋮
    def popleft(self) -> T:
        """Returns and removes the head of the queue"""
        self._head = self._head.next
        return self.pop()
```


Deque: __getitem__




```
class Deque(Generic[T]):
                    ⋮
    def __getitem__(self, i: int) -> T:
        """Returns the i-th element (i can be negative)"""
        if i < 0:  # Convert the index into positive
            i = len(self) + i
        if i < 0 or i >= len(self):  # Check if out of bounds
            raise IndexError

        p = self._head
        for _ in range(i):
            p = p.next
        return p.data
```

Can we make it more efficient? How about `q[-1]` in a queue with 10^6 elements?

Deque iterator: how to use it?

```
q: Deque[int] = Deque([12, 15, 6, -4])

# Visiting the elements of q with an iterator
q_iter: DequeIter[int] = iter(q)
try:
    while True:
        print('', next(q_iter), end='')
except StopIteration:
    print()

# Equivalent code with the same functionality
for x in q:
    print('', x, end='')
print()

# Important: q may have more than one iterator
# Iterators are independent from each other
q_iter1 = iter(q)
q_iter2 = iter(q)
```


Deque iterator: how to use it?

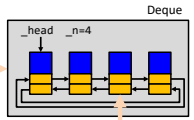
```
q: Deque[int] = Deque([12, 15, 6, -4])

q_iter: DequeIter[int] = iter(q)

try:
    while True:
        print('', next(q_iter), end='')
except StopIteration:
    print()
```

`q.__iter__()`

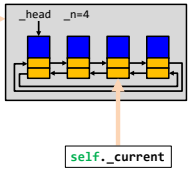
`q_iter.__next__()`

Deque: iterator

```
class Deque(Generic[T]):
                    ⋮
    def __iter__(self) -> 'DequeIter[T]':
        """Returns an iterator"""
        return DequeIter(self)

class DequeIter(Generic[T]):
    """Iterator of a deque"""
    _deque: Deque[T]  # The deque
    _current: Node[T] | None  # The current node

    def __init__(self, q: Deque[T]) -> None:
        """Initializes the iterator"""
        self._deque = q
        self._current = q._head if len(q) > 0 else None
```



Deque: iterator




```
class DequeIter(Generic[T]):
                    ⋮
    def __next__(self) -> T:
        """Returns the next element"""
        if self._current is None:
            raise StopIteration
        data = self._current.data
        self._current = self._current.next
        if self._current == self._deque._head:
            self._current = None
        return data
```


Using a deque

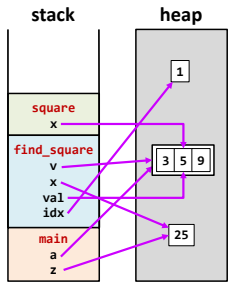
```
q: Deque[int] = Deque(range(1,9))  # Initialize with iterable
q.appendleft(0)
print("Number of elements:", len(q))

# Using q[i] (__getitem__). Quadratic cost!
for i in range(len(q)):
    print("  Element", i, " =", q[i])

# Using the iterators (linear cost: much more efficient!)
for i, x in enumerate(q):
    print("  Element", i, " =", x)

print("Removing the last element:", q.pop())
print("Removing the first element:", q.popleft())

print("Remaining elements in the queue:", end='')
for x in q:
    print(' ', x, sep='', end='')
print()
```


Memory management

- Programming languages have different strategies for memory management
- Typically, there are two structures used for memory management:
  - Stack, for static memory allocation, associated to functions/methods
  - Heap, for dynamic memory allocation

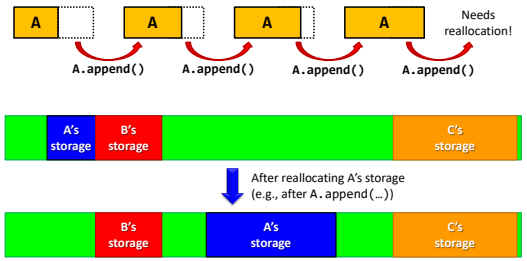
Memory management in Python

```
def square(x):
    return x*x

def find_square(v, x):
    for idx, val in enumerate(v):
        if x == square(val):
            return idx
    return None

a = [3, 5, 9]
z = 25
print(find_square(a, z))
```



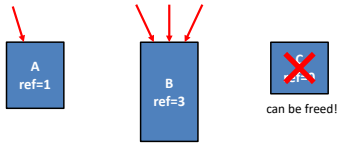
Heap management

Data structures often have some extra allocated memory to avoid frequent reallocations



Garbage collection

- Objects may have multiple references. Each object has a reference count.
- Python runs a garbage collector periodically. All objects with zero references are freed.



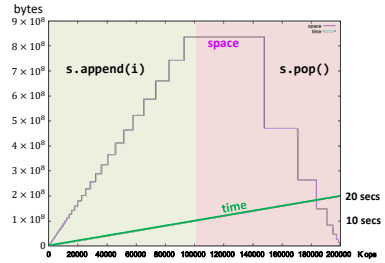
Efficient memory management

```
import sys
import time

t0 = time.time()
n = 100000000
s: list[int] = []

# append n elements
for i in range(n):
    s.append(i)
    if i % 1000 == 0:
        print(i, sys.getsizeof(s), time.time() - t0)

# remove n elements
for i in range(n):
    s.pop()
    if i % 1000 == 0:
        print(n+i, sys.getsizeof(s), time.time() - t0)
```


Efficient memory management

Avoid managing memory at every resizing operation

Use extra memory to amortize the effort in allocating/deallocating memory



EXERCISES

Stack: Interleaved push/pop operations

Suppose that an intermixed sequence of push and pop operations are performed. The pushes push the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?

a) 4 3 2 1 0 9 8 7 6 5
b) 4 6 8 7 5 3 2 9 0 1
c) 2 5 6 7 4 8 9 3 1 0
d) 4 3 2 1 0 5 6 7 8 9

Source: Robert Sedgewick, Computer Science 126, Princeton University.

Middle element of a stack

Design the class **`MidStack`** implementing a stack with the following operations:

- Push/pop: the usual operations on a stack.
- FindMiddle: returns the value of the element in the middle.
- DeleteMiddle: deletes the element in the middle.

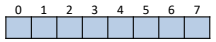
All the operations must be executed in $O(1)$ time.

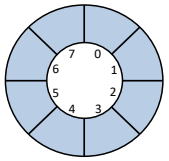
Suggestion: use some of the existing Python containers to implement it.

Note: if the stack has n elements at locations $0..n-1$, where 0 is the location at the bottom, the middle element is the one at location $\lfloor (n-1)/2 \rfloor$.

Queues implemented as circular buffers

- Design the class queue implemented with a circular buffer (using a Python list):
  - The add/remove operations should run in constant time.
  - The class should have a constructor with a parameter n that indicates the maximum number of elements in the queue.
- Consider the design of a variable-size queue using a circular buffer. Discuss how the implementation should be modified.





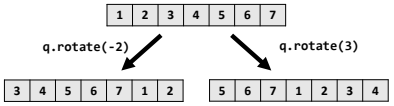
Reverse

Design the method `reverse()` that reverses the contents of a deque:

- No auxiliary data structures should be used.
- No copies of the elements should be performed.

Rotate and Josephus

- Design the method **`q.rotate(n)`** that rotates a deque **`n`** positions to the right (if positive) or **`-n`** positions to the left (if negative). Make the method efficient (e.g., assume that n can be large)



- Solve the Josephus problem, for n people and executing every k-th person, using a deque:

  https://en.wikipedia.org/wiki/Josephus_problem

Merge and quick sort

- Design the method **`q.merge(other: Deque)`** that merges the deque with another deque, assuming that both are sorted. Assume that a pair of elements can be compared with the operator <. After merging, all the elements must have been transferred to **`q`** (**`other`** becomes empty).
- Design the method **`q.sort()`** that sorts the deque according to the < operator. Consider merge sort and quick sort as possible algorithms.

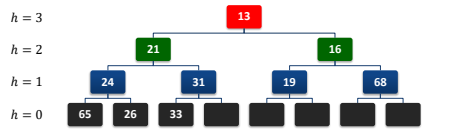
Priority Queues

Jordi Cortadella and Jordi Petit
Department of Computer Science

A priority queue

- A priority queue is a queue in which each element has a priority.
- Elements with higher priority are served before elements with lower priority.
- It can be implemented as a vector or a linked list. For a queue with n elements:
  - Insertion is $O(n)$.
  - Extraction is $O(1)$.
- A more efficient implementation can be proposed in which insertion and extraction are $O(\log n)$: ***binary heap***.

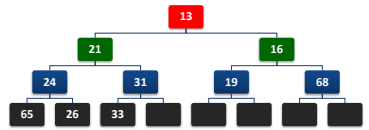
Binary Heap



- Complete binary tree (except at the bottom level).
- Height h: between 2^h and $2^{h+1} - 1$ nodes.
- For N nodes, the height is $O(\log N)$.
- It can be represented in a vector.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	13	21	16	24	31	19	68	65	26	33					

Binary Heap

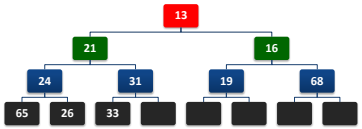


Locations in the vector:

Heap-Order Property: the key of the parent of X is smaller than (or equal to) the key in X.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	13	21	16	24	31	19	68	65	26	33					

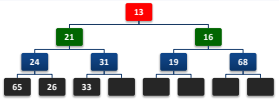
Binary Heap



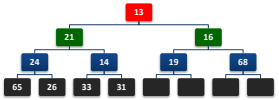
Two main operations on a binary heap:
- Insert a new element
- Remove the min element

Both operations must preserve the properties of the binary heap:
- Completeness
- Heap-Order property

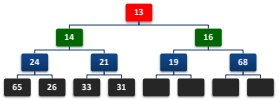
Binary Heap: insert 14



Insert in the last location

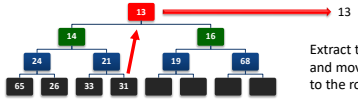


... and bubble up ...

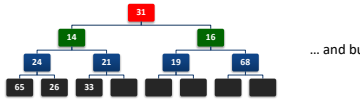


done !

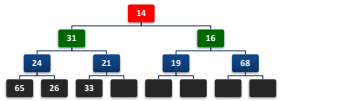
Binary Heap: remove min



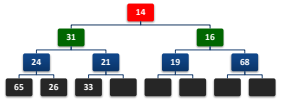
Extract the min element and move the last one to the root of the heap

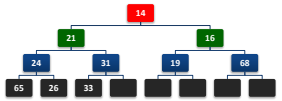


... and bubble down ...



Binary Heap: remove min





done !

Binary Heap: complexity

- Bubble up/down operations do at most h swaps, where h is the height of the tree and

$$h = \lfloor \log_2 N \rfloor$$

- Therefore:
  - Getting the min element is $O(1)$
  - Inserting a new element is $O(\log N)$
  - Removing the min element is $O(\log N)$

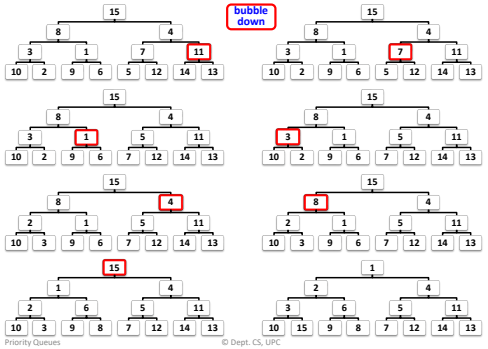
Binary Heap: other operations

- Let us assume that we have a method to know the location of every key in the heap.
- Increase/decrease key:
  - Modify the value of one element in the middle of the heap.
  - If decreased → bubble up.
  - If increased → bubble down.
- Remove one element:
  - Set value to $-\infty$, bubble up and remove min element.

Building a heap from a set of elements

- Heaps are sometimes constructed from an initial collection of N elements. How much does it cost to create the heap?
  - Obvious method: do N insert operations.
  - Complexity: $O(N \log N)$
- Can it be done more efficiently?

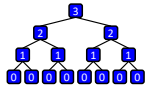
Building a heap from a set of elements



Building a heap: implementation

```
def heapify(L: list[T]) -> None:
    """Converts a list into a heap (assuming L[0] is not used)"""
    for i in range(len(L)//2, 0, -1):
        bubble_down(L, i)
```

Sum of the heights of all nodes:
- 1 node with height h
- 2 nodes with height $h-1$
- 4 nodes with height $h-2$
- 2^i nodes with height $h-i$



$$S = \sum_{i=0}^{h-1} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \cdots + 2^{h-1}(1)$$

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2^h(1)$$

Subtract the two equations:

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h+1) = O(N)$$

A heap can be built from a collection of items in linear time.

EXERCISES

Exercise: insert/remove element

Given the binary heap implemented in the following vector, draw the tree represented by the vector.

6	7	9	10	11	12	13	15	19	14	21	17	16

Execute the following sequence of operations

```
insert(8); remove_min(); insert(6); insert(18); remove_min();
```

and draw the tree after the execution of each operation.

Exercise: guess a and b

Consider the binary heap of integer keys implemented by the following vector:

3	7	a	10	15	18	b	25	13	20	17	22	19

After executing the operations **`insert(8)`** and **`remove_min()`** the contents of the binary heap is:

7	10	8	b	15	18	a	25	13	20	17	22	19

Discuss about the possible values of a and b. Assume there can never be two identical keys in the heap.

Exercise: the k-th element

The k-th element of n sorted vectors.

Let us consider n vectors sorted in ascending order.

Design an algorithm with cost $\Theta(k \log n + n)$ that finds the k-th global smallest element.

Exercise: bubble-up/down

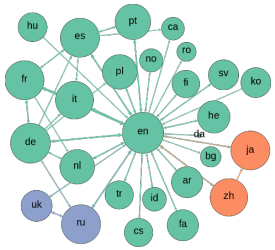
Give an implementation for the methods **`bubble_up`** and **`bubble_down`** of a heap:

```
def bubble_up(L: list[T], int i) -> None:
    """Bubbles up the element at location i"""

def bubble_down(L: list[T], int i) -> None:
    """Bubbles down the element at location i"""
```

Graphs: Connectivity

Jordi Cortadella and Jordi Petit
Department of Computer Science

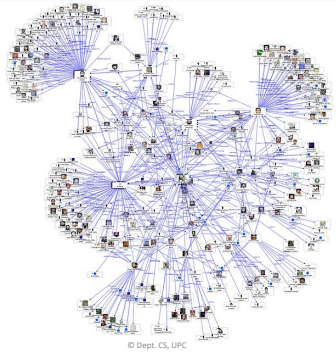
A graph



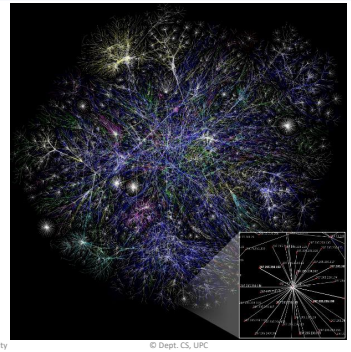
Source: Wikipedia
The network graph formed by Wikipedia editors (edges) contributing to different Wikipedia language versions (vertices) during one month in summer 2013

Transportation systems

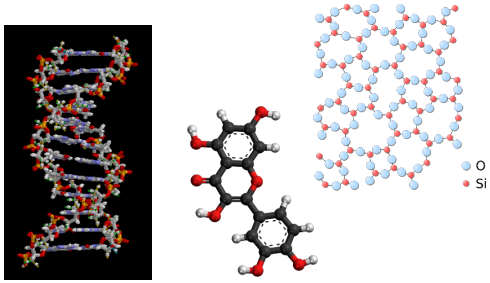
Social networks



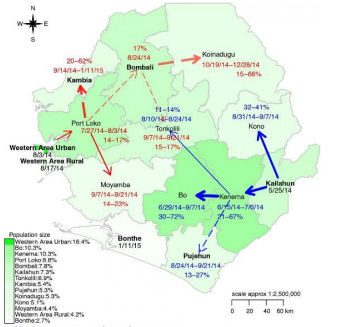
World Wide Web



Biology

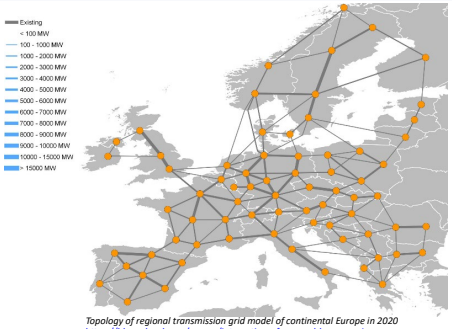


Disease transmission network



https://medicalxpress.com/news/2015-11-reveals-deadly-route-ebola-outbreak.html

Transmission of renewable energy



Topology of regional transmission grid model of continental Europe in 2020
https://blogs.dnvgl.com/energy/integration-of-renewable-energy-in-europe

What would we like to solve on graphs?

- Finding paths: which is the shortest route from home to my workplace?
- Flow problems: what is the maximum amount of people that can be transported in Barcelona at rush hours?
- Constraints: how can we schedule the use of the operating room in a hospital to minimize the length of the waiting list?
- Clustering: can we identify groups of friends by analyzing their activity in twitter?

Credits

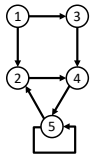
A significant part of the material used in this chapter has been inspired by the book:

Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, *Algorithms*, McGraw-Hill, 2008. [DPV2008]

(several examples, figures and exercises are taken from the book)

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E.



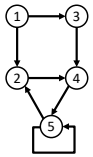
$$V = \{1,2,3,4,5\}$$

$$E = \{(1,2),(1,3),(2,4),(3,4),(4,5),(5,2),(5,5)\}$$

Graphs can be directed or undirected.
Undirected graphs have a symmetric relation.

Graph representation: adjacency matrix

A graph with $n = |V|$ vertices, $v_1, \cdots, v_n$, can be represented by an $n \times n$ matrix with:



$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

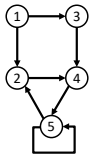
$$a = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

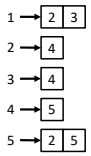
Space: $O(n^2)$

For undirected graphs, the matrix is symmetric.

Graph representation: adjacency list

A graph can be represented by $|V|$ lists, one per vertex. The list for vertex u holds the vertices connected to the outgoing edges from u.





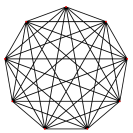
The lists can be implemented in different ways (vectors, linked lists, …)

Space: $O(|E|)$

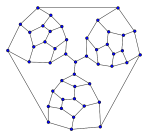
Undirected graphs: use bi-directional edges

Dense and sparse graphs

- A graph with $|V|$ vertices could potentially have up to $|V|^2$ edges (all possible edges are possible).
- We say that a graph is ***dense*** when $|E|$ is close to $|V|^2$. We say that a graph is ***sparse*** when $|E|$ is close to $|V|$.
- How big can a graph be?

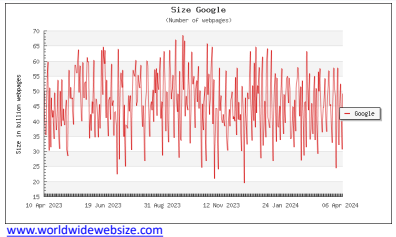


Dense graph



Sparse graph

Size of the World Wide Web



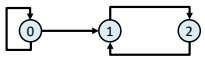
www.worldwidewebsize.com

- April 2024: 50 billion web pages (50×10^9).
- Size of adjacency matrix: 25×10^{20} elements. (not enough computer memory in the world to store it).
- Good news: The web is very sparse. Each web page has about half a dozen hyperlinks to other web pages.

Adjacency matrix vs. adjacency list

- Space:
  - Adjacency matrix is $O(|V|^2)$
  - Adjacency list is $O(|E|)$
- Checking the presence of a particular edge (u, v):
  - Adjacency matrix: constant time
  - Adjacency list: traverse u's adjacency list
- Which one to use?
  - For dense graphs → adjacency matrix
  - For sparse graphs → adjacency list
- For many algorithms, traversing the adjacency list is not a problem, since they require to iterate through all neighbors of each vertex. For sparse graphs, the adjacency lists are usually short (can be traversed in constant time)

Graph usage: example



	succ	pred
0	{0,1}	{0}
1	{2}	{0,2}
2	{1}	{1}

```
Edges:
0 -> 0
0 -> 1
1 -> 2
2 -> 1
```

```python
# Create the graph
g = Graph()

# Create the vertices
g.add_vertex(0)
g.add_vertex(1)
g.add_vertex(2)

# Create the edges
g.addEdge(0, 0)
g.addEdge(0 ,1)
g.addEdge(1, 2)
g.addEdge(2, 1)

# Print all edges of the graph
print("Edges:")
for u, v in g.edges():
    print(u, "->", v)
```


Graph implementation

```python
class Graph:
    """Class to represent a simple graph"""
    _succ: dict[int, set[int]]  # forward adjacency list
    _pred: dict[int, set[int]]  # backward adjacency list

    def __init__(self):
        """Graph constructor"""
        self._succ = {}
        self._pred = {}

    def add_vertex(self, v: int) -> None:
        """Adds vertex v to the graph"""
        assert v not in self._succ
        self._succ[v] = set()
        self._pred[v] = set()

    def vertices(self) -> Iterable[int]:
        """Returns the set of vertices of the graph"""
        return self._succ.keys()
```

	_succ	_pred
0	{0,1}	{0}
1	{2}	{0,2}
2	{1}	{1}

Graph implementation

```python
    def add_edge(self, src: int, dst: int) -> None:
        """Adds edge src -> dst to the graph"""
        assert src in self._succ and dst in self._succ
        self._succ[src].add(dst)
        self._pred[dst].add(src)

    def successors(self, v: int) -> Iterable[int]: # or set[int]
        """Returns the set of successors of vertex v"""
        assert v in self._succ
        return self._succ[v]

    def predecessors(self, v: int) -> Iterable[int]: # or set[int]
        """Returns the set of predecessors of vertex v"""
        assert v in self._pred
        return self._pred[v]
```


Graph implementation

We need a generator to iterate over all edges of the graph, as follows:

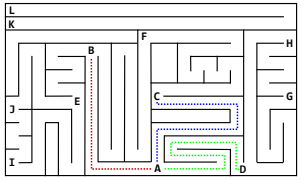
```python
print("Edges:")
for u, v in g.edges():
    print(u, "->", v)
```

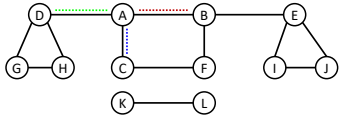
But we only have iterators over successors and predecessors of vertices.

The solution:

```python
    def edges(self) -> Iterator[tuple[int, int]]:
        """Returns a generator for all edges of the graph"""
        for u in self.vertices():
            for v in self.successors(u):
                yield (u,v)
```

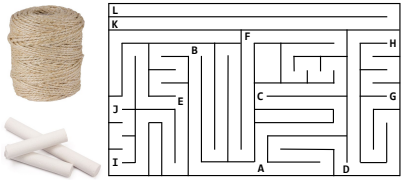

Reachability: exploring a maze





Which vertices of the graph are reachable from a given vertex?

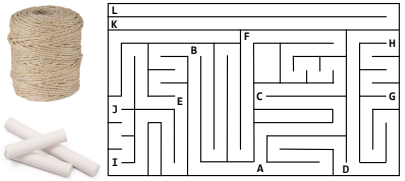
Reachability: exploring a maze



To explore a labyrinth we need a ball of string and a piece of chalk:

- The chalk prevents looping, by marking the visited junctions.
- The string allows you to go back to the starting place and visit routes that were not previously explored.

Reachability: exploring a maze



How to simulate the string and the chalk with an algorithm?

- Chalk: a boolean variable for each vertex (visited).
- String: a stack
  - push vertex to unwind at each junction
  - pop to rewind and return to the previous junction

Note: the stack can be simulated with recursion.

Finding the nodes reachable from another node

```python
def explore(G, v, visited):
    """Input: G = (V,E) is a graph
       Output: visited[u] is true for all the nodes
               reachable from v
    """
    visited[v] = True
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited[u]: explore(G, u, visited)
    postvisit(v)
```

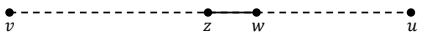
Notes:

- Initially, `visited[v]` is assumed to be *false* for every $v \in V$.
- pre/postvisit functions are not required now.

Finding the nodes reachable from another node

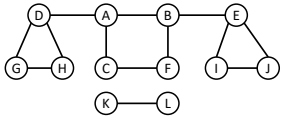
```python
def explore(G, v, visited):
  visited[v] = True
  for each edge (v,u) ∈ E:
    if not visited[u]: explore(G, u, visited)
```

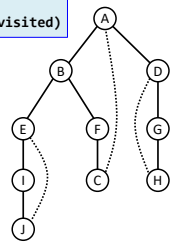
- All visited nodes are reachable because the algorithm only moves to neighbors and cannot jump to an unreachable region.
- Does it miss any reachable vertex? No. Proof by contradiction.
  - Assume that a vertex u is missed.
  - Take any path from v to u and identify the last vertex that was visited on that path (z). Let w be the following node on the same path. Contradiction: w should have also been visited.



Finding the nodes reachable from another node

```python
def explore(G, v, visited):
  visited[v] = True
  for each edge (v,u) ∈ E:
    if not visited[u]: explore(G, u, visited)
```





Dotted edges are ignored (*back edges*): they lead to previously visited vertices.
The solid edges (*tree edges*) form a tree.

Depth-first search

```python
def dfs(G) → visited:
  for all v ∈ V:
    visited[v] = False
  for all v ∈ V:
    if not visited[v]: explore(G, v, visited)
```

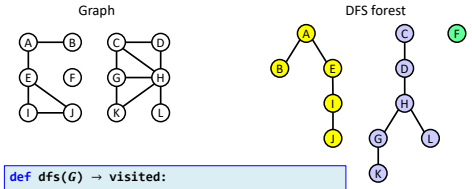
DFS traverses the entire graph.

Complexity:

- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
  - A fixed amount of work (pre/postvisit)
  - All adjacent edges are scanned

Running time is $O(|V| + |E|)$.
Difficult to improve: reading a graph already takes $O(|V| + |E|)$.

DFS example

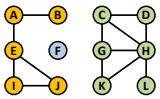



```python
def dfs(G) → visited:
  for all v ∈ V:
    visited[v] = False
  for all v ∈ V:
    if not visited[v]: explore(G, v, visited)
```

- The outer loop of DFS calls ***explore*** three times (for A, C and F)
- Three trees are generated. They constitute a *forest*.

Connectivity

- An undirected graph is connected if there is a path between any pair of vertices.
- A disconnected graph has disjoint *connected components*.
- Example: this graph has 3 connected components:



$\{A, B, E, I, J\}$ $\{C, D, G, H, K, L\}$ $\{F\}$.

Connected Components

```python
def explore(G, v, cc, ccnum):
    """Input: G = (V,E) is a graph, cc is a CC number
       Output: ccnum[u]=cc for each u in the same CC as v"""
    ccnum[v] = cc
    for each edge (v,u) ∈ E:
        if ccnum[u] == 0: explore(G, u, cc, ccnum)

def connected_components(G) → ccnum:
    """Input: G = (V,E) is a graph
       Output: Every vertex v has a CC number in ccnum[v]"""
    for all v ∈ V: ccnum[v] = 0  # Clean cc numbers
    cc = 1  # Identifier of the first CC
    for all v ∈ V:
        if ccnum[v] == 0:  # A new CC starts
            explore(G, v, cc, ccnum)
            cc += 1
```

- Performs a DFS traversal assigning a CC number (starting from 1) to each vertex.
- The outer loop of **connected_components** determines the number of CC's.
- The variable **ccnum[v]** also plays the role of visited[v].

Revisiting the explore function

```python
def explore(G, v, visited):
  visited[v] = True
  previsit(v)
  for each edge (v,u) ∈ E:
    if not visited[u]:
      explore(G, u, visited)
  postvisit(v)
```

Let us consider a global variable **clock** that can determine the occurrence times of previsit and postvisit.

```python
def previsit(v):
  pre[v] = clock
  clock = clock + 1

def postvisit(v):
  post[v] = clock
  clock = clock + 1
```

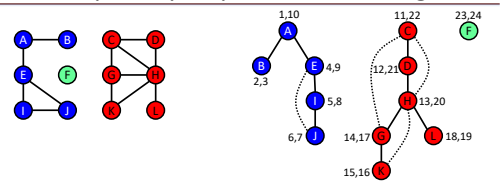
Note: we use clock, pre and post as global variables for convenience. Global variables should *never* be used.

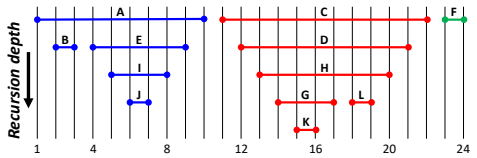
Every node v will have an interval (pre[v], post[v]) that will indicate the time the node was first visited (pre) and the time of departure from the exploration (post).

Property: Given two nodes u and v, the intervals (pre[u], post[u]) and (pre[v], post[v]) are either disjoint or one is contained within the other.

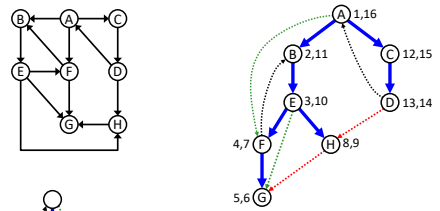
The pre/post interval of u is the lifetime of explore(u) in the stack (LIFO).

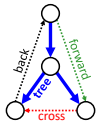
Example of pre/postvisit orderings





DFS in directed graphs: types of edges

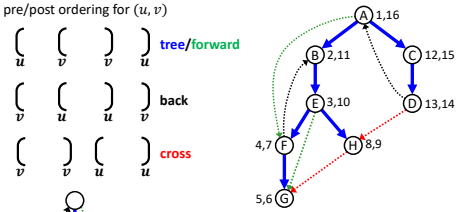


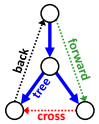


- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

DFS in directed graphs: types of edges

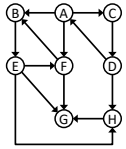
pre/post ordering for (u, v)





- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

Cycles in graphs



A ***cycle*** is a circular path:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0.$$

Examples:

$$B \to E \to F \to B$$
$$C \to D \to A \to C$$

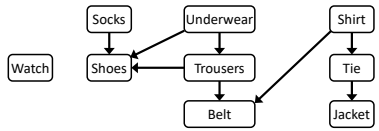
Property: A directed graph has a cycle *iff* its DFS reveals a back edge.

Proof:

⇐ If (u, v) is a back edge, there is a cycle with (u, v) and the path from v to u in the search tree.

⇒ Let us consider a cycle $v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$. Let us assume that v_i is the first discovered vertex (lowest pre number). All the other v_j on the cycle are reachable from v_i and will be its descendants in the DFS tree. The edge $v_{i-1} \to v_i$ leads from a vertex to its ancestor and is thus a back edge.

Getting dressed: DAG representation

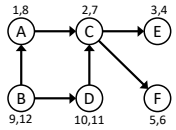


A list of tasks that must be executed in a certain order (cannot be executed if it has cycles).

Legal task *linearizations* (or *topological sorts*):

Underwear	Socks	Trousers	Shoes	Watch	Shirt	Belt	Tie	Jacket
Watch	Socks	Shirt	Tie	Jacket	Underwear	Trousers	Belt	Shoes

Directed Acyclic Graphs (DAGs)



A **DAG** is a directed graph without cycles.

DAGs are often used to represent causalities or temporal dependencies, e.g., task A must be completed before task C.

- Cyclic graphs cannot be linearized.
- All DAGs can be linearized. How?
  - Decreasing order of the post numbers.
  - The only edges (u, v) with post$[u]$ < post$[v]$ are back edges (do not exist in DAGs).
- **Property:** In a DAG, every edge leads to a vertex with a lower post number.
- **Property:** Every DAG has at least one source and at least one sink. (source: highest post number, sink: lowest post number).

Topological sort

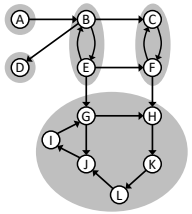
```
def explore(G, v, visited):
  visited[v] = True
  previsit(v)
  for each edge (v,u) ∈ E:
    if not visited[u]:
      explore(G, u, visited)
  postvisit(v)
```

```
Initially: top_sort = ∅
(top_sort is a linked list)

def postvisit(v):
  top_sort.appendleft(v)

# After DFS, top_sort contains
# a topological sort
```

Another algorithm:
- Find a source vertex, write it, and delete it (mark) from the graph.
- Repeat until the graph is empty.

It can be executed in linear time. How?

Strongly Connected Components



This graph is connected (undirected view), but there is no path between any pair of nodes.

For example, there is no path $K \to \cdots \to C$ or $E \to \cdots \to A$.

The graph is not *strongly connected*.

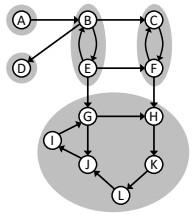
Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u.

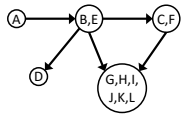
The *connected* relation is an equivalence relation and partitions V into disjoint sets of *strongly connected components*.

Strongly Connected Components

$\{A\}$
$\{B, E\}$
$\{C, F\}$
$\{D\}$
$\{G, H, I, J, K, L\}$

Strongly Connected Components





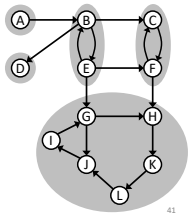
Every directed graph can be represented by a ***meta-graph***, where each meta-node represents a strongly connected component.

Property: every directed graph is a DAG of its strongly connected components. (Exercise: prove it!)

A directed graph can be seen as a 2-level structure. At the top we have a DAG of SCCs. At the bottom we have the details of the SCCs.

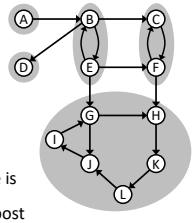
Properties of DFS and SCCs

- **Property:** If the *explore* function starts at u, it will terminate when all vertices reachable from u have been visited.
  - If we start from a vertex in a sink SCC, it will retrieve exactly that component.
  - If we start from a non-sink SCC, it will retrieve the vertices of several components.
- **Examples:**
  - If we start at K it will retrieve the component $\{G, H, I, J, K, L\}$.
  - If we start at B it will retrieve all vertices except A.

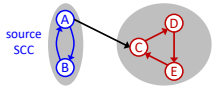


Properties of DFS and SCCs

- **Intuition for the algorithm:**
  - Find a vertex located in a sink SCC
  - Extract the SCC
- **To be solved:**
  - How to find a vertex in a sink SCC?
  - What to do after extracting the SCC?
- **Property:** If C and C' are SCCs and there is an edge $C \to C'$, then the highest post number in C is bigger than the highest post number in C'.
- **Property:** The vertex with the highest DFS post number lies in a source SCC.

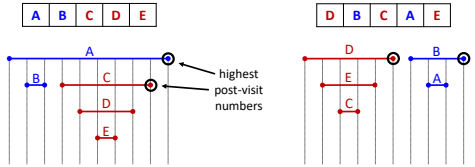


Properties of DFS and SCCs

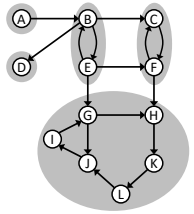


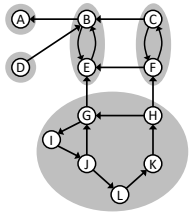
But we would like executing DFS starting from the sink nodes!

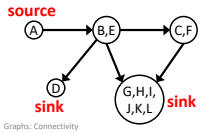
How can we do that?

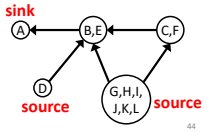


Reverse graph (G^R)









SCC algorithm

```
def scc(G) → ccnum:
    """Input: G(V,E) a directed graph
       Output: each v has an SCC number in ccnum[v]"""
    G^R= reverse(G)
    dfs(G^R, post)  # calculates post numbers
    sort(V, post)  # decreasing order of post number
    ccnum = connected_components(G)
```

Runtime complexity:
- DFS and connected_components Comp run in linear time $O(|V| + |E|)$.
- Can we reverse G in linear time?
- Can we sort V by post number in linear time?

Reversing G in linear time

```
def scc(G) → ccnum:
    """Input: G(V,E) a directed graph
       Output: each v has an SCC number in ccnum[v]"""
    G^R= reverse(G)
    dfs(G^R, post)  # calculates post numbers
    sort(V, post)  # decreasing order of post number
    ccnum = connected_components(G)
```

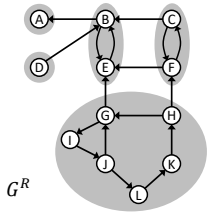
```
def reverse(G) → G^R:
    """Input: G(V,E) graph represented by an adjacency list
              edges[v] for each vertex v.
       Output: G(V,E^R) the reversed graph of G, with the
               adjacency list edgesR[v]."""
    for each u ∈ V:
        for each v ∈ edges[u]:
            edgesR[v].add(u)
    return G(V, edgesR)
```

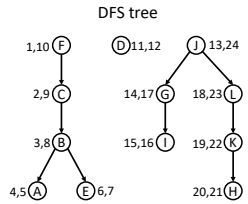

Sorting V in linear time

```
def scc(G) → ccnum:
    """Input: G(V,E) a directed graph
       Output: each v has an SCC number in ccnum[v]"""
    G^R= reverse(G)
    dfs(G^R, post)  # calculates post numbers
    sort(V, post)  # decreasing order of post number
    ccnum = connected_components(G, ccnum)
```

Use the explore function for topological sort:
- Each time a vertex is post-visited, it is inserted at the top of the list.
- The list is ordered by decreasing order of post number.
- It is executed in linear time.

Sorting V in linear time





Assume the initial order:
$F, A, B, C, D, E, J, G, H, I, K, L$

Vertex:	J	L	K	H	G	I	D	F	C	B	E	A
Post:	24	23	22	21	17	16	12	10	9	8	7	5

Crawling the Web

- Crawling the Web is done using depth-first search strategies.
- The graph is unknown and no recursion is used. A stack is used instead containing the nodes that have already been visited.
- The stack is not exactly a LIFO. Only the most "interesting" nodes are kept (e.g., page rank).
- Crawling is done in parallel (many computers at the same time) but using a central stack.
- How do we know that a page has already been visited? Hashing.

Summary

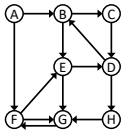
- Big data is often organized in big graphs (objects and relations between objects)
- Big graphs are usually sparse. Adjacency lists is the most common data structure to represent graphs.
- Connectivity can be analyzed in linear time using depth-first search.

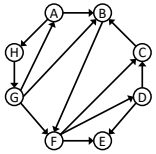
EXERCISES

DFS: stack overflow

- DFS can be implemented with an elegant recursive algorithm, but it may experiment *stack overflow* problems. Explain why.
- Design an iterative version of DFS.
- Challenge: do not to search in stackoverflow

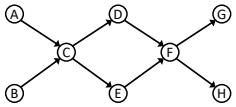
DFS (from [DPV2008])





Perform DFS on the two graphs. Whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge or cross edge, and give the `pre` and `post` number of each vertex.

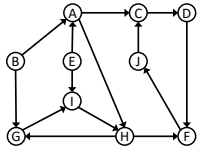
Topological ordering (from [DPV2008])

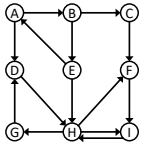


Run the DFS-based topological ordering algorithm on the graph. Whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

1. Indicate the `pre` and `post` numbers of the nodes.
2. What are the sources and sinks of the graph?
3. What topological order is found by the algorithm?
4. How many topological orderings does this graph have?

SCC (from [DPV2008])





Run the SCC algorithm on the two graphs. When doing DFS of G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first. For each graph, answer the following questions:

1. In what order are the SCCs found?
2. Which are source SCCs and which are sink SCCs?
3. Draw the meta-graph (each meta-node is an SCC of G).
4. What is the minimum number of edges you must add to the graph to make it strongly connected?

Reachability of web pages

The CS department would like to analyze the connectivity of their web pages, under the suspicious that some of them are not reachable. For that purpose, the IT staff proposes to launch a DFS exploration on a list that contains all URLs of the web pages. The pages are symbolically represented as $\{A, B, \ldots, J\}$. The DFS visits the pages following the links contained in each page. When there is more than one link in a page, the links are visited in order of appearance inside the page. The DFS exploration reports the pre- and post-visit numbers associated to each page, which are the following:

page	A	B	C	D	E	F	G	H	I	J
pre	14	13	17	4	6	10	2	1	16	3
post	15	20	18	5	7	11	9	12	19	8

The IT staff also knows that the links $C \to A$, $D \to H$ and $E \to D$ are present in the pages because they have explicitly created them.

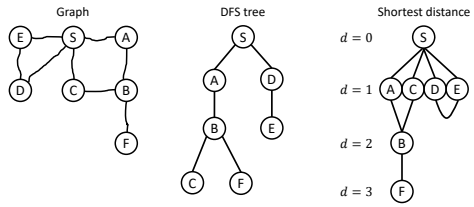
- Draw the DFS forest of the graph. For nodes with a common parent, draw the firstly visited nodes at the left.
- Indicate which pages are reachable from H, unreachable from H and those for which we do not know whether they are reachable from H. Answer the same questions for B.
- Answer where there is a link between the following pairs of nodes (you can answer YES, NO or I DON'T KNOW): $I \to F, F \to I, B \to C, B \to I, G \to F, F \to E$.
- Are D and G in the same SCC? Are A and C in the same SCC?
- The IT staff decides to add the link $F \to B$ *at the end* of page F to improve the connectivity of the pages. Draw the DFS forest after executing the DFS algorithm on the same list of pages.

Graphs: Shortest paths

Jordi Cortadella and Jordi Petit
Department of Computer Science

Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.

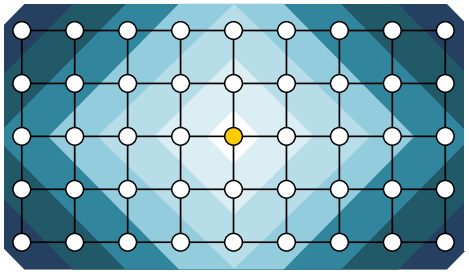


Distance between two nodes: length of the shortest path between them

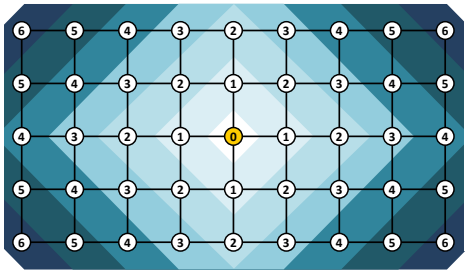
Breadth-first search

Similar to a wave propagation

Breadth-first search



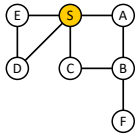
Breadth-first search



BFS algorithm

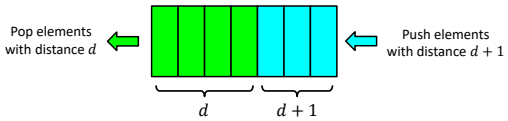
- BFS visits vertices layer by layer: $0,1,2,\ldots,d$.
- Once the vertices at layer d have been visited, start visiting vertices at layer $d+1$.
- Algorithm with two active layers:
  - Vertices at layer d (currently being visited).
  - Vertices at layer $d+1$ (to be visited next).
- Central data structure: a queue.

BFS algorithm: simulation



		S	A	B	C	D	E	F
	S_0	0	∞	∞	∞	∞	∞	∞
S_0	$A_1\ C_1\ D_1\ E_1$	0	1	∞	1	1	1	∞
A_1	$C_1\ D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
C_1	$D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
D_1	$E_1\ B_2$	0	1	2	1	1	1	∞
E_1	B_2	0	1	2	1	1	1	∞
B_2	F_3	0	1	2	1	1	1	3
F_3		0	1	2	1	1	1	3

BFS queue



BFS algorithm

```
def BFS(G, s) → dist:
  """Input: Graph G(V,E), source vertex s.
     Output: For each vertex u, dist[u] is
             the distance from s to u."""

  for all u ∈ V: dist[u]= ∞

  dist[s] = 0
  Q = {s}  # Queue containing just s
  while not Q.empty():
    u = Q.pop_front()
    for all (u,v) ∈ E:
      if dist[v] == ∞:
        dist[v] = dist[u] + 1
        Q.push_back(v)
```

Runtime $O(|V|+|E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

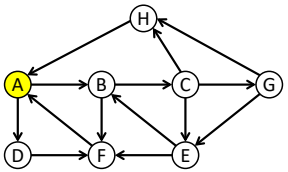
Reachability: BFS vs. DFS

Input: A graph G and a source node s.
Output: $\forall u \in V$: visited[u] $\Leftrightarrow$ u is reachable from s.
The function processes the nodes in BFS/DFS order

```
def BFS(G, s) → visited:
  for all u ∈ V:
    visited[u] = False

  Q = ⇐        # Empty queue
  Q.push_back(s)
  visited[s] = True
  while not Q.empty():
    u = Q.pop_front()
    process(u)
    for all (u,v) ∈ E:
      if not visited[v]:
        visited[v] = True
        Q.push_back(v)
```

```
def DFS(G, s) → visited:
  for all u ∈ V:
    visited[u] = False

  S = ⊔        # Empty stack
  S.push(s)
  while not S.empty():
    u = S.pop()
    process(u)
    if not visited[u]:
      visited[u] = True
      for all (u,v) ∈ E:
        S.push(v)
```


Reachability: BFS vs. DFS



DFS order:	A	B	C	E	F	G	H	D
BFS order:	A	B	D	C	F	E	G	H
Distance:	0	1	1	2	2	3	3	3

Weights on edges

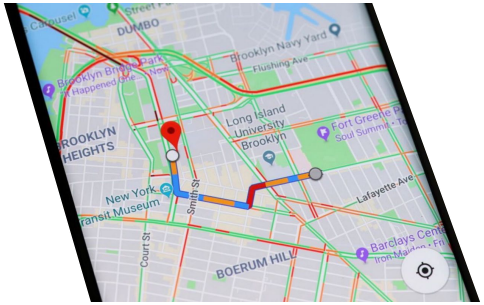
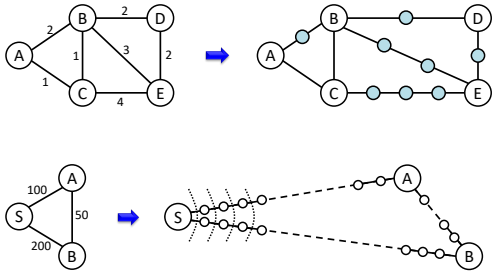


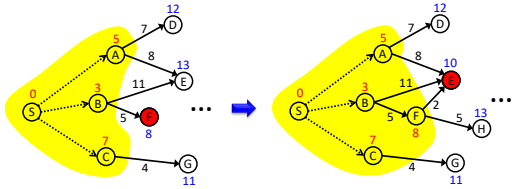
Image credits: https://thegadgetflow.com/blog/google-maps-vs-google-earth/

Reusing BFS



Inefficient: many cycles without any interesting progress. How about real numbers?

Dijkstra's algorithm: invariant

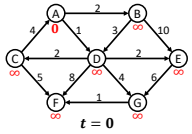


Shortest paths already computed (completed vertices)

Frontier

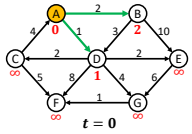
Data structure:
The set of non-completed vertices with their shortest distance from S using only the completed vertices.

Example



$t = 0$

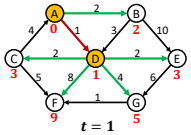
Done	Queue
	A:0
	B:∞
	E:∞
	D:∞
	C:∞
	F:∞
	G:∞



$t = 0$

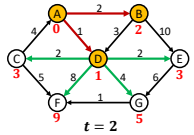
Done	Queue
A:0	D:1
	B:2
	E:∞
	C:∞
	F:∞
	G:∞

Example



$t = 1$

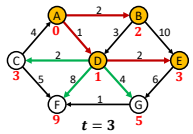
Done	Queue
A:0	B:2
D:1	E:3
	C:3
	G:5
	F:9



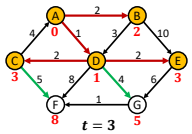
$t = 2$

Done	Queue
A:0	E:3
D:1	C:3
B:2	G:5
	F:9

Example

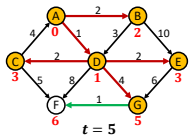


Done	Queue
A:0	C:3
D:1	G:5
B:2	F:9
E:3	

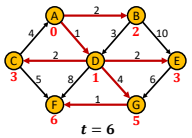


Done	Queue
A:0	G:5
D:1	F:8
B:2	
E:3	
C:3	

Example



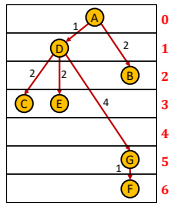
Done	Queue
A:0	F:6
D:1	
B:2	
E:3	
C:3	
G:5	

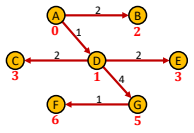


Done	Queue
A:0	
D:1	
B:2	
E:3	
C:3	
G:5	
F:6	

Example

Shortest-path tree





We need to:

- keep a list non-completed vertices and their expected distances.
- select the non-completed vertex with shortest distance.
- update the distances of the neighbouring vertices.

Dijkstra's algorithm for shortest paths

```
def ShortestPaths(G, s, len) → dist, prev:
    """Input: Graph G(V,E), source vertex s,
              positive edge lengths {len(e): e ∈ E}
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    Q = makequeue(V)  # priority queue (dist as value)

    while not Q.empty():
        u = Q.deletemin()
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
                Q.decreasekey(v)
```


Dijkstra's algorithm: complexity

```
Q = makequeue(V)
while not Q.empty():
  u = Q.deletemin()
  for all (u,v) ∈ E:
    if dist[v] > dist[u] + len(u,v):
      dist[v] = dist[u] + len(u,v)
      prev[v] = u
      Q.decreasekey(v)
```

$|V|$ times

$|E|$ times

- The skeleton of Dijkstra's algorithm is based on BFS, which is $O(|V| + |E|)$
- We need to account for the cost of:
  - **makequeue**: insert $|V|$ vertices to a list.
  - **deletemin**: find the vertex with min dist in the list ($|V|$ times)
  - **decreasekey**: update dist for a vertex ($|E|$ times)
- Let us consider two implementations for the list: **vector** and **binary heap**

Dijkstra's algorithm: complexity

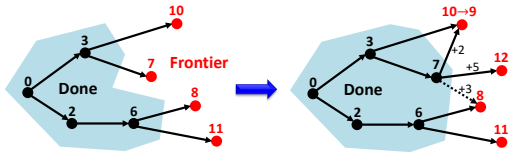
Implementation	deletemin	insert/ decreasekey	Dijkstra's complexity
Vector	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$

Binary heap:

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it *bubble up* swapping the location with the parent (at most $\log_2 |V|$ levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it *bubble down* (at most $\log_2 |V|$ levels).
- **Decreasekey:** decrease the key in the tree and let it *bubble up* (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).

For connected graphs: $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$

Why Dijkstra's works

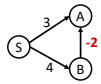


- A tree of open paths with distances is maintained at each iteration.
- The shortest paths for the internal nodes have already been calculated.
- The node in the frontier with shortest distance is "frozen" and expanded. Why? Because no other shorter path can reach the node.

Disclaimer: this is only true if the **distances are non-negative!**

Graphs with negative edges

- Dijkstra's algorithm does not work:



Dijkstra would say that the shortest path S→A has length=3.

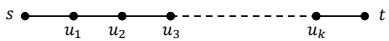
- Dijkstra is based on a safe update each time an edge (u, v) is treated:

$$\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}$$

- Problem: shortest paths are consolidated too early.
- Possible solution: add a constant weight to all edges, make them positive, and apply Dijkstra.
  - It does not work, prove it!

Graphs with negative edges

- The shortest path from s to t can have at most $|V| - 1$ edges:



- If the sequence of updates includes

$$(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_k, t),$$

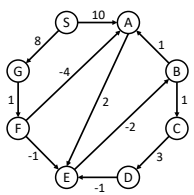
in that order, the shortest distance from s to t will be computed correctly (updates are always safe). Note that the sequence of updates does not need to be consecutive.

- Solution: update all edges $|V| - 1$ times !
- Complexity: $O(|V| \cdot |E|)$.

Bellman-Ford algorithm

```
def ShortestPaths(G, s, len) → dist, prev:
    """Input: Graph G(V,E), source vertex s,
              edge lengths {len(e): e ∈ E}, no negative cycles
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    repeat |V| − 1 times:
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
```


Bellman-Ford: example



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Negative cycles

- What is the shortest distance between S and A?

Bellman-Ford does not work as it assumes that the shortest path will not have more than $|V| - 1$ edges.

- A negative cycle produces $-\infty$ distances by endlessly applying rounds to the cycle.
- How to detect negative cycles?
  - Apply Bellman-Ford (update edges $|V| - 1$ times)
  - Perform an extra round and check whether some distance decreases.

Shortest paths in DAGs

- DAG's property:

  In any path of a DAG, the vertices appear in increasing topological order.

- Any sequence of updates that preserves the topological order will compute distances correctly.
- Only one round visiting the edges in topological order is sufficient: $O(|V| + |E|)$.
- How to calculate the longest paths?
  - Negate the edge lengths and compute the shortest paths.
  - Alternative: update with max (instead of min).

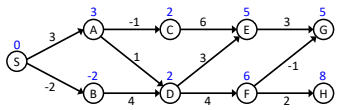
DAG shortest paths algorithm

```
def DagShortestPaths(G, s, len) → dist, prev:
    """Input: DAG G(V,E), source vertex s,
              edge lengths {len(e): e∈E}
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    Linearize G
    for all u ∈ V in linearized order:
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
```

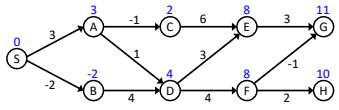

DAG shortest/longest paths: example

Linearization: S A B C D E F G H

Shortest paths



Longest paths



Shortest paths: summary

Single-source shortest paths

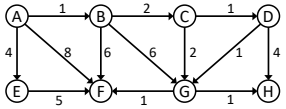
Graph	Algorithm	Complexity
Unit edge-length	BFS	$O(\|V\| + \|E\|)$
Non-negative edges	Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$
Negative edges	Bellman-Ford	$O(\|V\| \cdot \|E\|)$
DAG	Topological sort	$O(\|V\| + \|E\|)$

A related problem: All-pairs shortest paths

- Floyd-Warshall algorithm ($O(|V|^3)$), based on dynamic programming.
- Other algorithms exist.

EXERCISES

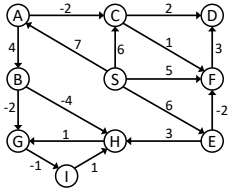
Dijkstra (from [DPV2008])



Run Dijkstra's algorithm starting at node A:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

Bellman-Ford (from [DPV2008])



Run Bellman-Ford algorithm starting at node S:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

Graphs: A* search

Jordi Cortadella and Jordi Petit
Department of Computer Science

Shortest path between two nodes

How to find the shortest path from Barcelona to Girona?

Easy: run Dijkstra from Barcelona

Dijkstra will find ALL shortest paths from Barcelona

Do we really need to waste computations exploring roads that go to Lleida, Amposta or Vielha?

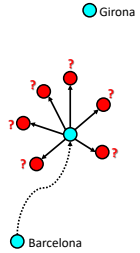
A* search algorithm

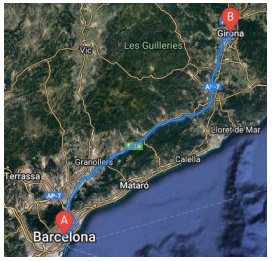
- Original paper:

  P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968.

- A* is a class of graph searching strategies using ad hoc heuristic information. A* guarantees optimal solutions when the heuristic information meets certain properties.

A* search: heuristic guidance

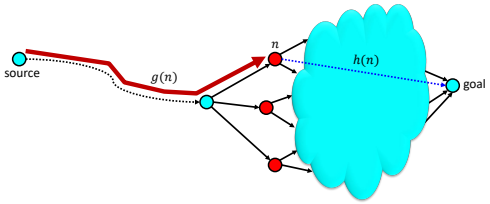




What is the most promising node to explore?

Heuristic: select nodes that reduce the straight-line distance to the target

A* search: intuition



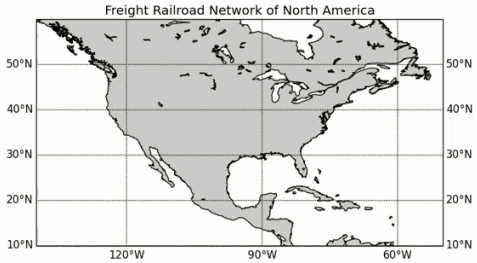
How to extend the path and find the next node n?

For each node n calculate $f(n) = g(n) + h(n)$

- $g(n)$ is the cost of the path from the source to n
- $h(n)$ is the estimated cost of the cheapest path from n to the goal

Select the node with minimum $f(n)$

Example: train network



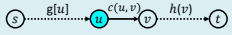
Finding the optimum path from Washington, D.C. and Los Angeles

$h(x)$ is the great-circle distance (the shortest possible distance on a sphere) to the target

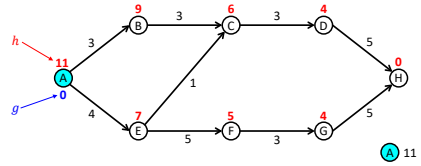
Source: https://en.wikipedia.org/wiki/A*_search_algorithm

A* algorithm for shortest paths

```
def Astar_search(G, s, t, c, h, pred):
  """Input:  Graph G(V,E), source node s, target node t,
             positive edge costs {c(e): e ∈ E},
             function to estimate the cost to the target {h(v): v ∈ V}
    Output: pred[u] has the predecessor in the shortest path from s to t,
             if t ∉ pred, no path exists from s to t
  """
  f = {}  # dictionary for the f value (∞ if not present)
  g = {}  # dictionary for the g value (∞ if not present)
  pred = {}  # dictionary of predecessors

  g[s] = 0
  f[s] = h(s)
  Q = {s}  # open nodes: priority queue sorted by f

  while not Q.empty():
    u = Q.deletemin()  # get open node with min cost
    if u == t: return
    for all (u,v) ∈ E:
      gv = g[u] + c(u,v)
      if gv < g[v]:  # g[v]=∞ if v ∉ g
        g[v] = gv
        f[v] = gv + h(v)
        pred[v] = u
        Q.add(v)  # new open node (or update if already in Q)
```

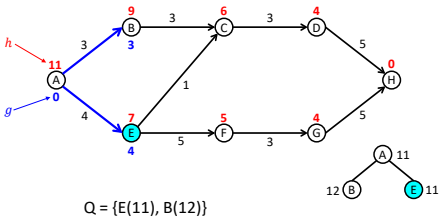


Running A*: example



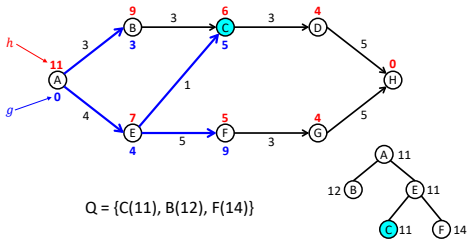
Q = {A(11)}

Running A*: example



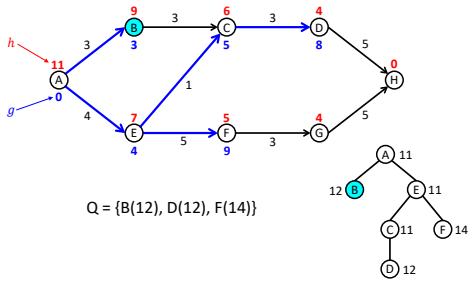
Q = {E(11), B(12)}

Running A*: example



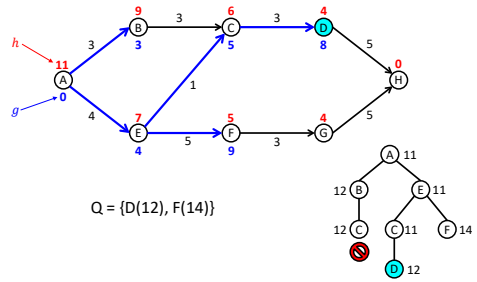
Q = {C(11), B(12), F(14)}

Running A*: example



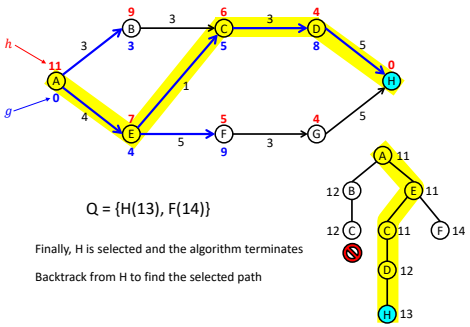
Q = {B(12), D(12), F(14)}

Running A*: example



Q = {D(12), F(14)}

Running A*: example

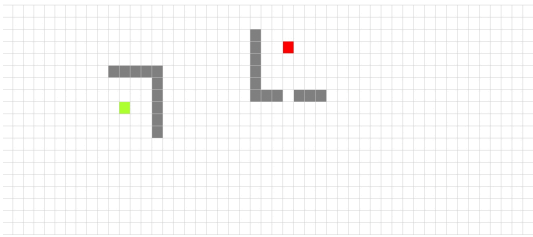


Q = {H(13), F(14)}

Finally, H is selected and the algorithm terminates

Backtrack from H to find the selected path

Avoiding obstacles



Source: https://github.com/vittin/A-Star/

The heuristic function: $h(x)$

- A* relies on a good heuristic to estimate the cost to reach the goal
- How about using a "bad" heuristic function?
  - Does the algorithm find the optimum path?
  - Does it run efficiently?
- Let us study the concepts of *admissible* and *consistent* heuristic function

Admissibility

- A heuristic function is said to be *admissible* if it never overestimates the cost of reaching the goal
- Example: the straight-line distance in a map is an *admissible* function (no path can be shorter than the straight line)

Admissibility

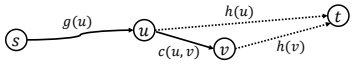
- Important result:
  - If $h(x)$ is admissible, A* will find the optimum path
- Proof (informal):
  - A* will never overlook a path with lower cost, since a node v with lower $f(v)$ than the goal will exist in the set of open nodes before the goal is reached.

Consistency

- A heuristic function $h(x)$ is said to be *consistent* (or monotone) if

$$h(u) \leq c(u,v) + h(v)$$

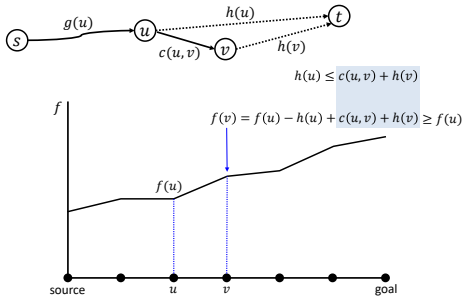
for every edge (u,v) with cost $c(u,v)$



- Important result:
  - If $h(x)$ is consistent, A* is guaranteed to find an optimal path without processing any node more than once

Consistency

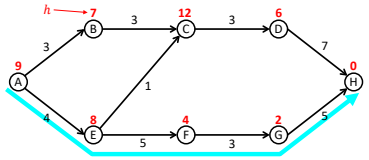
If $h(x)$ is consistent then $f(x)$ is an increasing function



Example

Goal: find the shortest path from A to H

- The h function is not admissible (it overestimates the cost to the goal)
- Example: $h(\mathrm{C}) = 12$. The solution may not be optimal

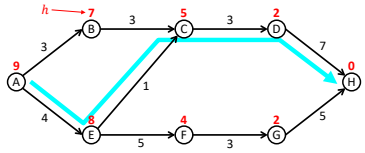


Visited nodes during the A* search: A B E F G H

Example

Goal: find the shortest path from A to H.

- The h function is admissible (it does not overestimate the cost to the goal)
- The h function is not consistent, e.g., $h(\mathrm{E}) > d(\mathrm{E},\mathrm{C}) + h(\mathrm{C})$
- The solution will be optimal. Some nodes may be visited more than once

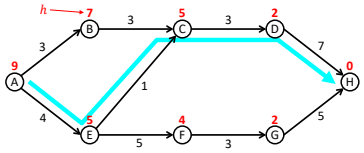


Visited nodes during the A* search: A B **C D** E **C D** F G H

Example

Goal: find the shortest path from A to H.

- The h function is admissible (it does not overestimate the cost to the goal)
- The h function is consistent
- The solution will be optimal and the nodes will be visited once at most



Visited nodes during the A* search: A E B C D F G H

Complexity

- The time complexity of A* highly depends on the heuristic function. The worst-case complexity is $\mathrm{O}(b^d)$, where
  - d is the depth of the shortest path
  - b is the average branching factor (number of successor nodes of each node)
- Each heuristic has an *effective* branching factor b^*, and the number of visited nodes is:

$$1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

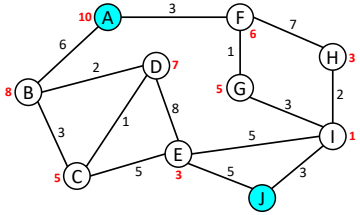
- Good heuristics have small values for b^* (the optimal heuristic has $b^* = 1$)

Complexity

- A* may not terminate if the graph is infinite and no path exists to the target
- A* keeps all generated nodes in memory. There are memory-bounded heuristic searches:
  - Iterative deepening A*: guided DFS, no priority queue, nodes may be visited multiple times
  - Simplified Memory-Bounded A* (SMA*): nodes with highest f-cost pruned from the queue
  - and others ...

EXERCISES

Shortest path



Find the shortest path from A to J using the A* algorithm.
The red numbers next to the nodes represent the heuristic value.
Questions:

- Is the heuristic admissible?
- Is the heuristic consistent?
- Report the visited node at each step of the algorithm

8-puzzle problem

Initial state

2	8	3
1	6	4
7		5

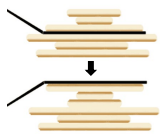
Goal

1	2	3
8		4
7	6	5

- Use the A* algorithm to find the smallest sequence of shifts to reach the goal. Depict the search tree
- Consider:
  - $g(n)$ = depth of the node (number of shifts)
  - $h(n)$ = number of misplaced tiles

Pancake sorting

- You have a disordered stack of pancakes of different sizes. You want to sort this pile (smallest pancake on top, largest one at the bottom) using a spatula by flipping parts of the stack

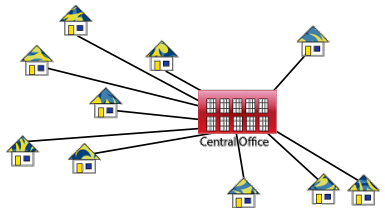


- Describe how you would use A* to find the shortest sequence of flips that sort the pile
  - Define the initial state, successor states, and the functions $g(n)$ and $h(n)$. Make sure $h(n)$ is admissible

Graphs: Minimum Spanning Trees

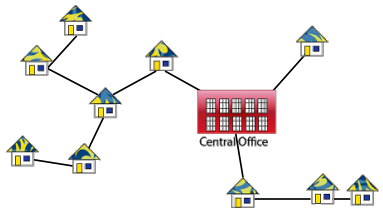
Jordi Cortadella and Jordi Petit
Department of Computer Science

Laying a communication network



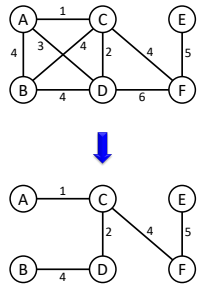
Source: https://www.javatpoint.com/applications-of-minimum-spanning-tree

Laying a communication network



Source: https://www.javatpoint.com/applications-of-minimum-spanning-tree

Minimum Spanning Trees



- Nodes are computers
- Edges are links
- Weights are maintenance cost
- Goal: pick a subset of edges such that
  - the nodes are connected
  - the maintenance cost is minimum

The solution is not unique.
Find another one !

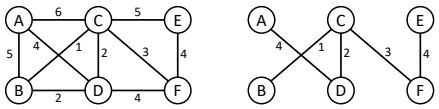
Property:
An optimal solution cannot contain a cycle.

Minimum Spanning Tree

- Given un undirected graph $G = (V, E)$ with edge weights w_e, find a tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e \,.$$

- Greedy algorithm: repeatedly add the next lightest edge that does not produce a cycle.

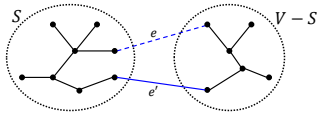


Note: We will now see that this strategy guarantees an MST.

Properties of trees

- **Definition:** A tree is an undirected graph that is connected and acyclic.
- **Property:** Any connected, undirected graph $G = (V, E)$ has $|E| \geq |V| - 1$ edges.
- **Property:** A tree on n nodes has $n - 1$ edges.
  - Start from an empty graph. Add one edge at a time making sure that it connects two disconnected components. After having added $n - 1$ edges, a tree has been formed.
- **Property:** Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
  - It is sufficient to prove that G is acyclic. If not, we can always remove edges from cycles until the graph becomes acyclic.
- **Property:** Any undirected graph is a tree iff there is a unique path between any pair of nodes.
  - If there would be two paths between two nodes, the union of the paths would contain a cycle.

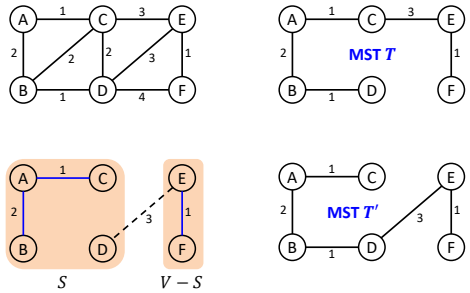
The cut property



Suppose edges X are part of an MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof (sketch): Let T be an MST and assume e is not in T. If we add e to T, a cycle will be created with another edge e' across the cut $(S, V - S)$. We can now remove e' and obtain another tree T' with $\text{weight}(T') \leq \text{weight}(T)$. Since T is an MST, then the weights must be equal.

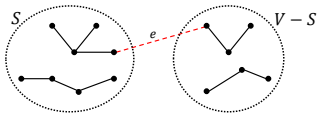
The cut property: example



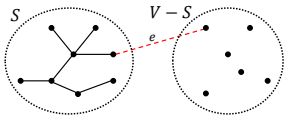
Minimum Spanning Tree

Any scheme like this works (because of the properties of trees):

```
X = {}   # The set of edges of the MST
repeat |V| − 1 times:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```



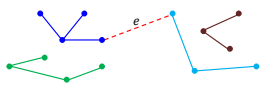
MST: two strategies



Prim's algorithm

Invariant:
- A set of nodes (S) is in the tree.

Progress:
- The lightest edge with exactly one endpoint in S is added.



Kruskal's algorithm

Invariant:
- A set of trees (forest) has been constructed.

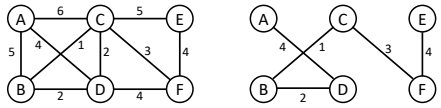
Progress:
- The lightest edge between two trees is added.

Prim's algorithm

```
def Prim(G, w) → prev:
  """Input: A connected undirected Graph G(V,E)
            with edge weights w(e).
     Output: An MST defined by the vector prev."""
  for all u ∈ V:
    visited[u] = False
    prev[u] = nil
  pick any initial node u0
  visited[u0] = True
  n = 1

  # Q: priority queue of edges using w(e) as priority
  Q = makequeue()
  for each (u0,v) ∈ E: Q.insert(u0,v)

  while n < |V|:
    (u,v) = deletemin(Q) # Edge with smallest weight
    if not visited[v]:
      visited[v] = True
      prev[v] = u
      n = n + 1
      for each (v,x) ∈ E:
        if not visited[x]: Q.insert(v,x)
```

Complexity: $O(|E| \log |V|)$

Prim's algorithm



Q:
(AD,4) (AB,5) (AC,6)
(DB,2) (DC,2) (DF,4) (AB,5) (AC,6)
(BC,1) (DC,2) (DF,4) (AB,5) (AC,6)
(DC,2) (CF,3) (DF,4) (AB,5) (CE,5) (AC,6)
(CF,3) (DF,4) (AB,5) (CE,5) (AC,6)
(DF,4) (FE,4) (AB,5) (CE,5) (AC,6)
(FE,4) (AB,5) (CE,5) (AC,6)

Kruskal's algorithm

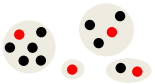
Informal algorithm:
- Sort edges by weight.
- Visit edges in ascending order of weight and add them as long as they do not create a cycle.

How do we know whether a new edge will create a cycle?

```
def Kruskal(G, w) → MST:
  """Input: A connected undirected Graph G(V,E)
            with edge weights we.
     Output: An MST defined by the edges in MST."""

  MST = {}
  sort the edges in E by weight
  for all (u,v) ∈ E, in ascending order of weight:
      if (MST has no path connecting u and v):
          MST = MST ∪ {(u,v)}
```


Disjoint sets

- A data structure to store a collection of disjoint sets.



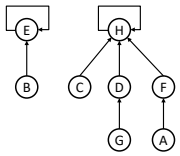
- Operations:
  - makeset(x): creates a singleton set containing just x.
  - find(x): returns the identifier of the set containing x.
  - union(x, y): merges the sets containing x and y.
- Kruskal's algorithm uses disjoint sets and calls
  - makeset: $|V|$ times
  - find: $2 \cdot |E|$ times
  - union: $|V| - 1$ times

Kruskal's algorithm

```
def Kruskal(G, w) → MST:
  """Input: A connected undirected Graph G(V,E)
            with edge weights we.
     Output: An MST defined by the edges in MST."""

  for all u ∈ V: makeset(u)

  MST = {}
  sort the edges in E by weight
  for all (u,v) ∈ E, in ascending order of weight:
      if (find(u) ≠ find(v)):
          MST = MST ∪ {(u,v)}
          union(u,v)
```


Disjoint sets

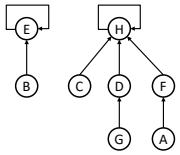
- The nodes are organized as a set of trees. Each tree represents a set.
- Each node has two attributes:
  - parent (π): ancestor in the tree
  - rank: height of the subtree
- The root element is the representative for the set: its parent pointer is itself (self-loop).
- The efficiency of the operations depends on the height of the trees.



```
def makeset(x):
  π(x) = x
  rank(x) = 0

def find(x):
  while x ≠ π(x): x = π(x)
  return x
```

Disjoint sets

```
def union(x, y):
  r_x = find(x)
  r_y = find(y)
  if r_x = r_y: return

  if rank(r_x) > rank(r_y):
    π(r_y) = r_x
  else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y):
      rank(r_y) = rank(r_y) + 1
```

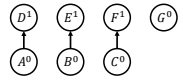



```
def makeset(x):
  π(x) = x
  rank(x) = 0

def find(x):
  while x ≠ π(x): x = π(x)
  return x
```

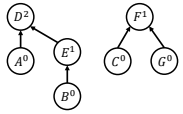

Disjoint sets

After makeset(A),…,makeset(G):

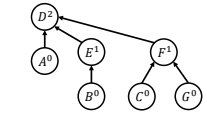
After union(A,D), union(B,E), union(C,F):



After union(C,G), union(E,A):



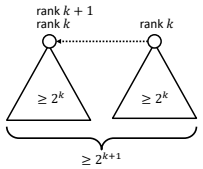
After union(B,G):



Property: Any root node of rank k has at least 2^k nodes in its tree.
Property: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k. Therefore, all trees have height $\leq \log n$.

Disjoint sets

Property 1: proof by induction



Property 2:

For n nodes, the tallest possible tree could have rank k, such that:

$$n \geq 2^k$$

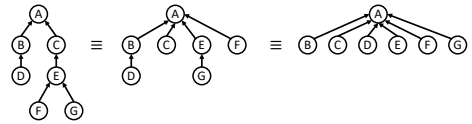
$$k \leq \log_2 n$$

Therefore, find(x) is $O(\log n)$

Property 1: Any root node of rank k has at least 2^k nodes in its tree.
Property 2: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k. Therefore, all trees have height $\leq \log n$.

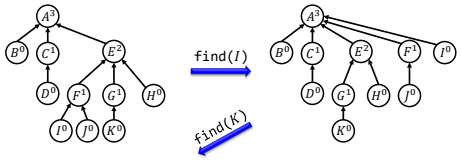
Disjoint sets: path compression

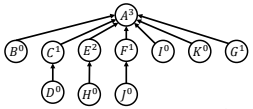
- Complexity of Kruskal's algorithm: $O(|E| \log |V|)$.
  - Sorting edges: $O(|E| \log |E|) = O(|E| \log |V|)$.
  - Find + union ($2 \cdot |E|$ times): $O(|E| \log |V|)$.
- How about if the edges are already sorted or sorting can be done in linear time (weights are integer and small)?
- Path compression:



Disjoint sets: path compression

```
def find(x):
  if x ≠ π(x): π(x)= find(π(x))
  return π(x)
```

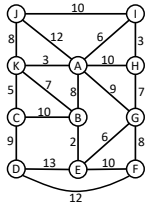




Amortized cost of find: $O(1)$
Kruskal's cost: $O(|E|)$
(if sorting has linear cost)

EXERCISES

Minimum Spanning Trees

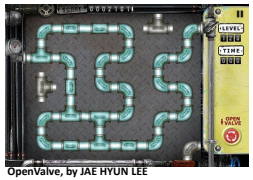


- Calculate the shortest path tree from node A using Dijkstra's algorithm.
- Calculate the MST using Prim's algorithm. Indicate the sequence of edges added to the tree and the evolution of the priority queue.
- Calculate the MST using Kruskal's algorithm. Indicate the sequence of edges added to the tree and the evolution of the disjoint sets. In case of a tie between two edges, try to select the one that is not in Prim's tree.

Graphs: Maximum Flows

Jordi Cortadella and Jordi Petit
Department of Computer Science

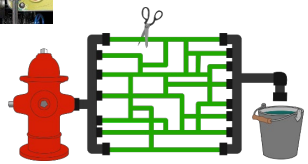
Max-flow/min-cut problems



OpenValve, by JAE HYUN LEE

How much water can you pump from source to target?

What is the fewest number of green tubes that need to be cut so that no water will be able to flow from the hydrant to the bucket?

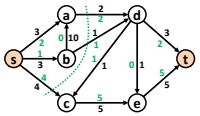


Max-flow/Min-cut algorithm. Brilliant.org.
https://brilliant.org/wiki/max-flow-min-cut-algorithm/

Max-flow/min-cut problems: applications

- Networks that carry data, water, oil, electricity, cars, etc.
  - How to maximize usage?
  - How to minimize cost?
  - How to maximize reliability?
- Multiple application domains:
  - Computer networks
  - Image processing
  - Computational biology
  - Airline scheduling
  - Data mining
  - Distributed computing
  - …

Max-flow problem



Model:
- A directed graph $G = (V, E)$.
- Two special nodes $s, t \in V$.
- Capacities $c_e > 0$ on the edges.

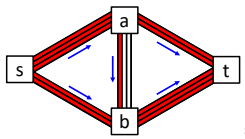
Goal: assign a flow f_e to each edge e of the network satisfying:
- $0 \le f_e \le c_e$ for all $e \in E$ (edge capacity not exceeded)
- For all nodes u (except s and t), the flow entering the node is equal to the flow exiting the node:

$$\sum_{(w,u)\in E} f_{wu} = \sum_{(u,x)\in E} f_{ux}\,.$$

Size of a flow: total quantity sent from s to t (equal to the quantity leaving s):

$$\text{size}(f) = \sum_{(s,u)\in E} f_{su}$$

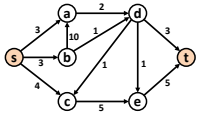
Max-flow problem: intuition

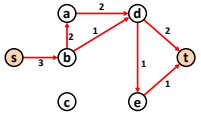


Find an augmenting path

An augmenting path may reverse some of the flow previously assigned

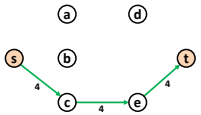
Augmenting paths



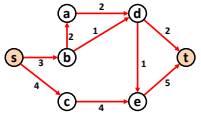


Flow

Given a flow, an ***augmenting path*** represents a feasible additional flow from s to t.

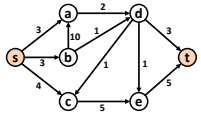


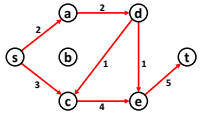
Augmenting path



New flow

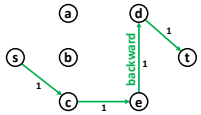
Augmenting paths



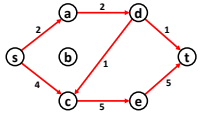


Flow

Augmenting paths can have *forward* and *backward* edges.



Augmenting path



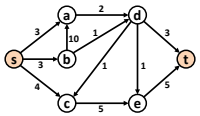
New flow

Augmenting paths

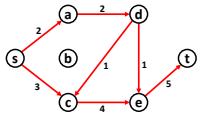
Given a flow f, an augmenting path is a directed path from s to t, which consists of edges from E, but not necessarily in the same direction. Each of these edges e satisfies exactly one of the following two conditions:

- e is in the same direction as in E (forward) and $f_e < c_e$. The difference $c_e - f_e$ is called the *slack* of the edge.
- e is in the opposite direction (backward) and $f_e > 0$. It represents the fact that some flow can be borrowed from the current flow.

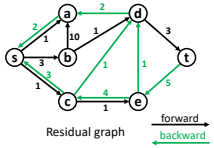
Residual graph



Graph

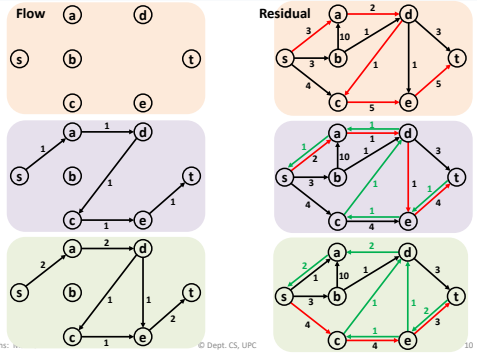


Flow

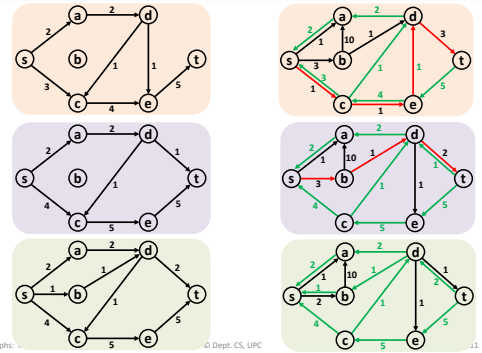


Residual graph

Ford-Fulkerson algorithm: example



Ford-Fulkerson algorithm: example



Ford-Fulkerson algorithm

```
def Ford-Fulkerson(G, s, t) → f:
  """Input: A directed Graph G(V,E) with edge capacities c_e.
           s and t and the source and target of the flow.
     Output: A flow f that maximizes the size of the flow.
             For each (u,v) ∈ E, f(v,u) represents its flow.
  """
  for all (u,v) ∈ E:
    f(u,v) = c(u,v)   # Forward edges
    f(v,u) = 0        # Backward edges

  while there exists a path p = s ⇝ t in the residual graph:
    f(p) = min{f(u,v): (u,v) ∈ p}
    for all (u,v) ∈ p:
      f(u,v) = f(u,v) − f(p)
      f(v,u) = f(v,u) + f(p)
```


Ford-Fulkerson algorithm: complexity

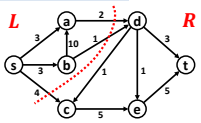
- Finding a path in the residual graph requires $O(|E|)$ time (using BFS or DFS).
- How many iterations (augmenting paths) are required?
  - The worst case is really bad: $O(C \cdot |E|)$, with C being the largest capacity of an edge (if only integral values are used).
  - By selecting the path with fewest edges (using BFS) the maximum number of iterations is $O(|V| \cdot |E|)$.
  - By carefully selecting *fat* augmenting paths (using some variant of Dijkstra's algorithm), the number of iterations can be reduced.
- Ford-Fulkerson algorithm is $O(|V| \cdot |E|^2)$ if BFS is used to select the path with fewest edges (Edmonds-Karp algorithm).

Max-flow problem

Cut: An (s,t)-cut partitions the nodes into two disjoint groups, L and R, such that $s \in L$ and $t \in R$.

For any flow f and any (s,t)-cut (L,R):

$$\text{size}(f) \le \text{capacity}(L,R).$$



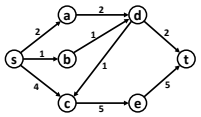
The max-flow min-cut theorem:

The size of the maximum flow equals the capacity of the smallest (s,t)-cut.

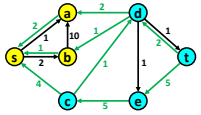
The augmenting-path theorem:

A flow is maximum iff it admits no augmenting path.

Min-cut algorithm



Flow

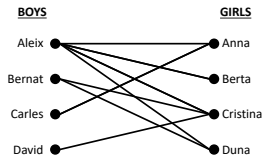


Residual graph

Finding a cut with minimum capacity:

1. Solve the max-flow problem with Ford-Fulkerson.
2. Compute L as the set of nodes reachable from s in the residual graph.
3. Define $R = V - L$.
4. The cut (L,R) is a min-cut.

Bipartite matching



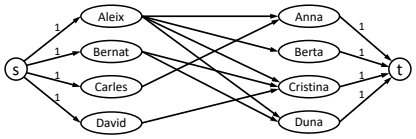
There is an edge between a boy and a girl if they like each other.

Can we pick couples so that everyone has exactly one partner that he/she likes?

Bad matching: if we pick (Aleix, Anna) and (Bernat, Cristina), then we cannot find couples for Berta, Duna, Carles and David.

A ***perfect matching*** would be: (Aleix, Berta), (Bernat, Duna), (Carles, Anna) and (David, Cristina).

Bipartite matching



Reduced to a max-flow problem with $c_e = 1$.

Question: can we always guarantee an integer-valued flow?

Property: if all edge capacities are integer, then the optimal flow found by Ford-Fulkerson's algorithm is integral. It is easy to see that the flow of the augmenting path found at each iteration is integral.

Extensions of Max-Flow

- **Max-Flow with Edge Demands**
  - Each edge e has a demand $d(e)$. The flow f must satisfy $d(e) \leq f(e) \leq c(e)$.
- **Node Supplies and Demands**
  - An extra flow $x(v)$ can be injected (positive) or extracted (negative) at every vertex v. The flow must satisfy:

$$\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) = x(v).$$

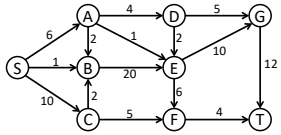
- **Min-cost Max-Flow**
  - Each edge e has a weight w_e. Compute a max-flow of minimum cost:

$$\text{cost}(f) = \sum_{e \in E} w_e \cdot f(e)$$

- **Max-Weight Bipartite Matching**
  - Each edge e has a weight w_e. Find a maximum cardinality matching with maximum total weight.

EXERCISES

Flow network (from [DVP2008])



- Find the maximum flow from S to T. Give a sequence of augmenting paths that lead to the maximum flow.
- Draw the residual graph after finding the maximum flow.
- Find a minimum cut between S and T.

Graph Problem Solving

Jordi Cortadella and Jordi Petit
Department of Computer Science

Motivation

- Here is a compendium of problems that can be solved using graph algorithms
- No hints are given about the algorithm that must be used, although the statement of the problem makes it evident in many cases
- Strategy:
  - Model the problem as a graph (vertices, edges, weights, etc.)
  - Choose an algorithm to solve a problem on the model
  - Find an interpretation of the solution in terms of the original problem

New road (from [DPV2008])

There is a network of roads $G = (V, E)$ connecting a set of cities V. Each road in E has an associated length l_e. There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length.

As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G would result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

Streets in Computopia (from [DPV2008])

The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

a) Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

b) Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

Blood transfusion

Enthusiastic celebration of a sunny day at a prominent northeastern university has resulted in the arrival at the university's medical clinic of 169 students in need of emergency treatment. Each of the 169 students requires a transfusion of one unit of whole blood. The clinic has supplies of 170 units of whole blood. The number of units of blood available in each of the four major blood groups and the distribution of patients among the groups is summarized below.

Blood type	A	B	O	AB
Supply	46	34	45	45
Demand	39	38	42	50

Type A patients can only receive type A or O; type B patients can receive only type B or O; type O patients can receive only type O; and type AB patients can receive any of the four types.

Give a graph formulation of the problem that determines a distribution that satisfies the demands of a maximum number of patients.

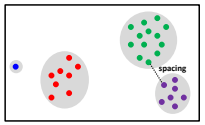
Can we have enough blood units for all the students?

Source: Sedgewick and Wayne, Algorithms, 4th edition, 2011.

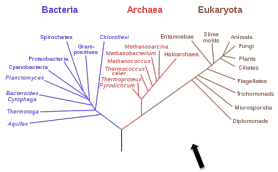
k-clustering of maximum spacing

We want to classify a set of points into k clusters. We define the distance between two points as the Euclidean distance. We define the *spacing* of the clustering as the minimum distance between any pair of points in different clusters.

Describe an algorithm such that, given an integer k, finds a k-clustering such that *spacing* is maximized. Argue about the complexity of the algorithm.



$k = 4$



Note: k-clustering of maximum spacing is the basis for the construction of dendograms.

Nesting boxes

A d-dimensional box with dimensions $(x_1, x_2, \dots, x_d)$ nests within another box with dimensions $(y_1, y_2, \dots, y_d)$ if there exists a permutation π on $\{1,2, \dots, d\}$ such that:

$$x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \dots, x_{\pi(d)} < y_d.$$

a. Argue that the nesting relation is transitive.

b. Describe an efficient method to determine whether or not one d-dimensional box nests inside another.

c. Suppose that you are given a set of n d-dimensional boxes $\{B_1, B_2, \dots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \dots, B_{i_k}\rangle$ of boxes such that B_{i_j} nests within $B_{i_{j+1}}$ for $j = 1,2, \dots, k-1$. Express the running time of your algorithm in terms of n and d.

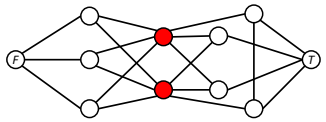
Source: Cormen, Leiserson and Rivest, Introduction to Algorithms, The MIT Press.

Contagious disease

The island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (Covid 19) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices *F* and *T* represent the stations in Ffarquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.



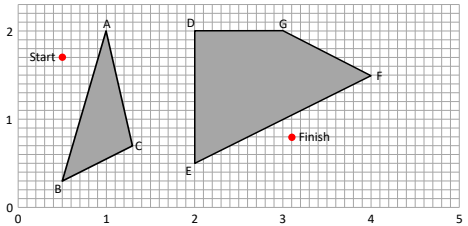
Source: Jeff Erickson, Algorithms, UIUC, 2015.

Edge-disjoint paths

Given a digraph $G = (V, E)$ and vertices $s, t \in V$, describe an algorithm that finds the maximum number of edge-disjoint paths from s to t.

Note: two paths are edge-disjoint if they do not share any edge.

Robot Path Planning



- Find the shortest path from Start to Finish without crossing any gray region
- Show all the steps of the selected algorithm
- Hint: moving to a point in the middle of an open area does not help. The shortest path can be found by visiting vertices of the gray areas

Pouring water (from [DPV2008])

We have three containers whose sizes are 10 pints, 7 pints and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 4-pint container.

a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

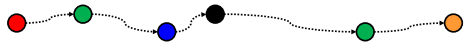
b) What algorithm should be applied to solve the problem?

c) Give a sequence of pourings, if it exists, or prove that it does not exist any sequence.

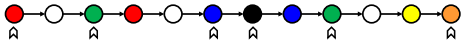
Hint: A vertex of the graph can be represented by a triple of integers.

Colored path

- Let us consider a directed graph in which each node is assigned a color and each edge is assigned a positive distance. Let us also consider a sequence of k colors, e.g.,



- Problem: find the shortest path that visits nodes with the given sequence of colors (and possibly other nodes in the middle), e.g., for the previous sequence, a valid path could be:



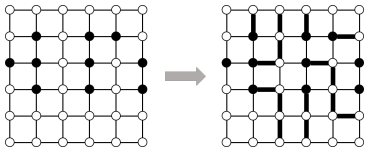
- Discuss the computational complexity of your solution.

The escape problem [CLR]

An $n \times m$ grid is an undirected graph with $n \times m$ vertices organized in n rows and m columns. We denote the vertex in the i-th row and the j-th column by (i, j). Every vertex (i, j) has exactly four neighbors $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$, except the boundary vertices, that have fewer neighbors.

Let (x_1, y_1), (x_2, y_2), …, (x_k, y_k) be distinct vertices, called terminals, in the $n \times m$ grid. The escape problem is to determine whether there are k vertex-disjoint paths in the grid that connect the terminals to any k distinct boundary vertices.

Hint: maxflow (the graph model is not obvious)



Critical edge

Let us consider a strongly connected graph $G(V, E)$ with positive weights $w_e > 0$ on the edges. We would like to answer the following question: given $s, t \in V$, and $e \in E$, do all shortest paths from s to t contain e?

Find an efficient algorithm to answer this question and discuss the complexity of the algorithm.

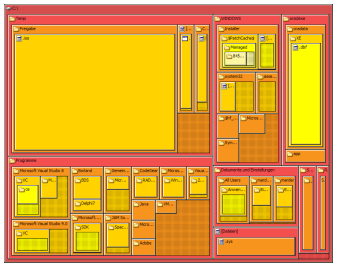
You do not need to give the code of the algorithm. It is sufficient if you describe a precise strategy to solve the problem.

Trees

Jordi Cortadella and Jordi Petit
Department of Computer Science

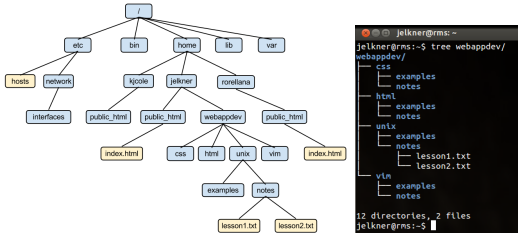
Trees

Data are often organized hierarchically

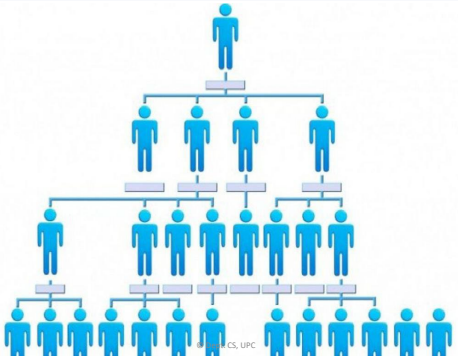


source: https://en.wikipedia.org/wiki/Tree_structure

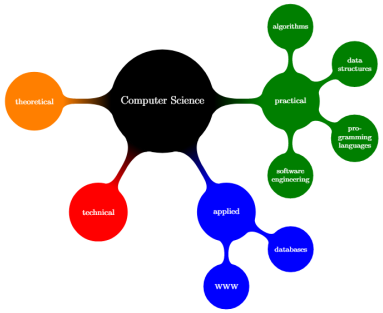
Filesystems



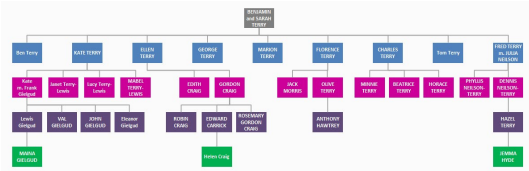
Company structure



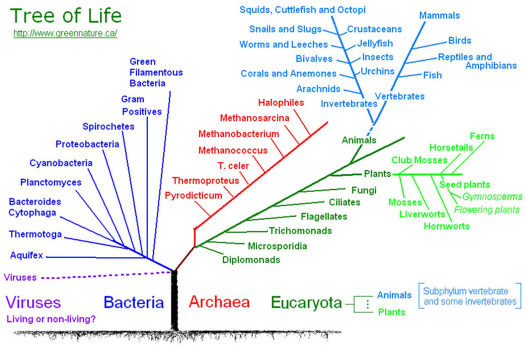
Mind maps



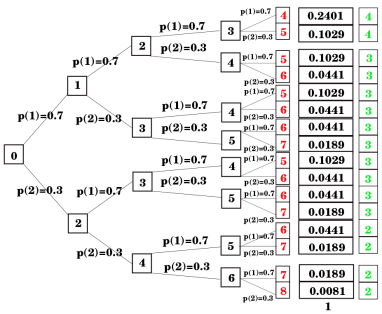
Genealogical trees



Tree of Life



Probability trees



Parse trees

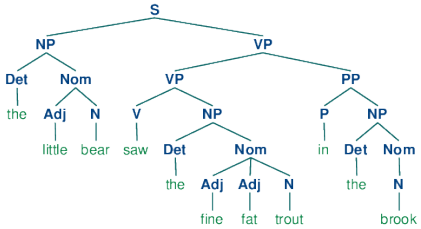
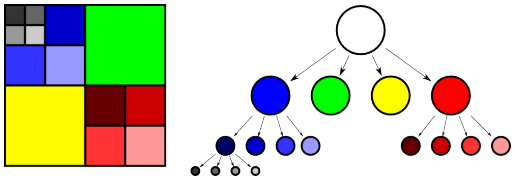
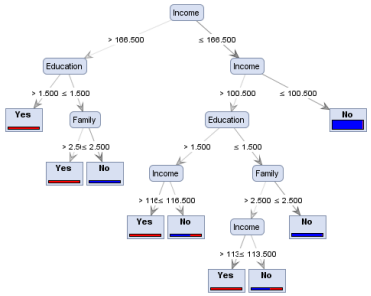


Image representation (quad-trees)



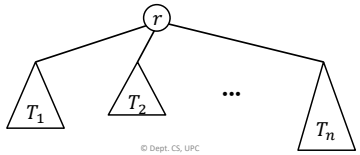
Decision trees



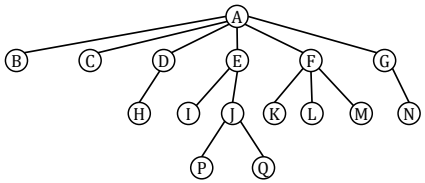
source: http://www.simafore.com/blog/bid/94454/A-simple-explanation-of-how-entropy-fuels-a-decision-tree-model

Tree: definition

- Graph theory: a tree is an undirected graph in which any two vertices are connected by exactly one path.
- Recursive definition (CS). A non-empty tree T consists of:
  - a root node r
  - a list of non-empty trees $T_1, T_2, \ldots, T_n$ that hierarchically depend on r. The list can be possibly empty ($n \geq 0$).



Tree: nomenclature



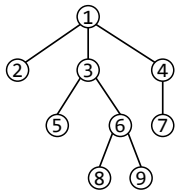
- A is the **root** node.
- Nodes with no children are **leaves** (e.g., B and P).
- Nodes with the same parent are **siblings** (e.g., K, L and M).
- The **depth** of a node is the length of the path from the root to the node. Examples: depth(A)=0, depth(L)=2, depth(Q)=3.

Tree: representation

There is a plethora of data structures that can be used to represent a tree, e.g., a hierarchical list.

`[root, child₁, child₂, …, childₙ]`

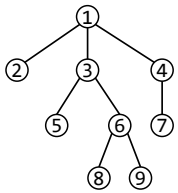
another tree



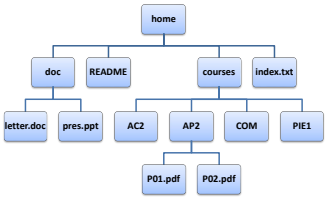

```
tree = [1, 2,
        [3, 5, [6, 8, 9]],
        [4, 7]
       ]
```


Tree: Abstract Data Type

```
from dataclasses import dataclass
from typing import TypeVar, Generic

T = TypeVar('T')

@dataclass
class Tree(Generic[T]):
    """Class to represent a generic tree"""
    data: T
    children: list[Tree[T]]


def size(t: Tree) -> int:
    return 1 + sum(c.size() for c in t.children)


def num_levels(t: Tree) -> int:
    # implement it!
```



Write a tree




```
home
  doc
    letter.doc
    pres.ppt
  README
  courses
    AC2
    AP2
      P01.pdf
      P02.pdf
    COM
    PIE1
  index.txt
```

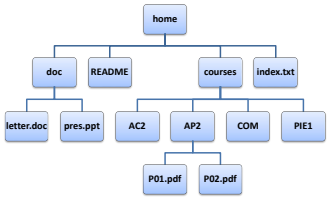
```
def write(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree indented according to the depth"""
```

Write a tree

```python
def write(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree indented according to the depth"""

    # print the root
    print(' '*2*depth, t.data, sep='')

    # print the children with depth + 1
    for c in t.children:
        write(c, depth + 1)
```

This function executes a ***preorder*** traversal of the tree: each node is processed *before* the children.

Write a tree (postorder traversal)




```
    letter.doc
    pres.ppt
  doc
  README
    AC2
      P01.pdf
      P02.pdf
    AP2
    COM
    PIE1
  courses
  index.txt
home
```

Postorder traversal: each node is processed after the children.

Write a tree (postordre traversal)

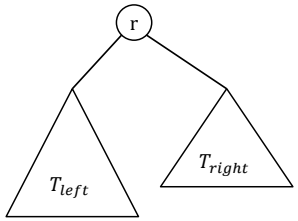
```python
def write_postorder(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree (in postorder) indented according
       to the depth"""

    # print the children with depth + 1
    for c in t.children:
        write_postorder(c, depth + 1)

    # print the root
    print(' '*2*depth, t.data, sep='')
```

This function executes a ***postorder*** traversal of the tree: each node is processed *after* the children.

Binary tree: definition

A binary tree is a finite set of nodes that either

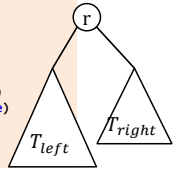
- is empty, or
- is comprised of three disjoint sets of nodes: a root node and two binary trees called its left and right subtrees



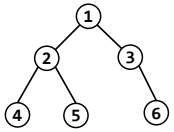
Binary tree: representation

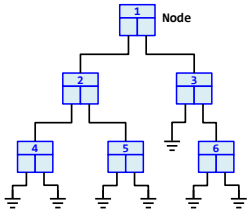
Data structures to represent binary trees are typically based on the definition of a node.

```python
from dataclasses import dataclass, field
from typing import TypeVar, Generic, Optional, Iterator

T = TypeVar('T')

@dataclass
class Node(Generic[T]):
    """Node of a bin tree"""
    data: T
    left: 'BinTree[T]' = field(default = None)
    right: 'BinTree[T]' = field(default = None)

BinTree = Optional[Node[T]]
NodeIter = Iterator[Node[T]]
```

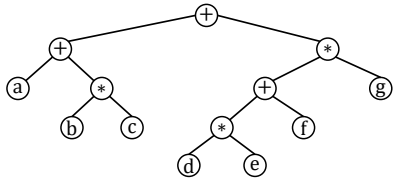


Binary tree: representation





Example: expression trees



Expression tree for: **a + b*c + (d*e + f) * g**

Postfix representation: **a b c * + d e * f + g * +**

How can the postfix representation be obtained?

Example: expression trees

Expressions are represented by strings in postfix notation in which **'a'…'z'** represent operands and **'+'** and **'*'** represent operators.

```python
Exprtree: TypeAlias = BinTree[str]

def build_expr(expr: str) -> Exprtree:
    """Builds an expression tree from a correct
       expression represented in postfix notation"""

def infix_expr(t: Exprtree) -> str:
    """Generates a string with the expression in
       infix notation"""

def eval_expr(t: Exprtree, v: dict[str, int]) -> int:
    """Evaluates an expression taking v as the value of the
       variables (e.g., v['a'] contains the value of a)"""
```

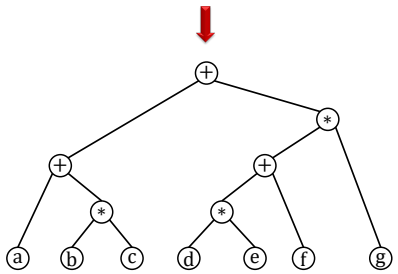

Example: expression trees

```python
def main():
    t = build_expr('a b c * + d e * f + g * +')
    print(infix_expr(t))
    print(eval_expr(t, {'a':3, 'b':1, 'c':0, 'd':5,
                        'e':2, 'f':1, 'g':6}))
```

```
Output:

((a+(b*c))+(((d*e)+f)*g))
69
```

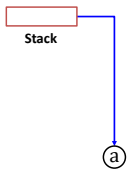

How to build an expression tree

a b c * + d e * f + g * +



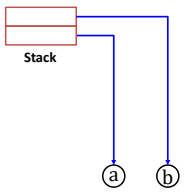
How to build an expression tree

a **b c * + d e * f + g * +**



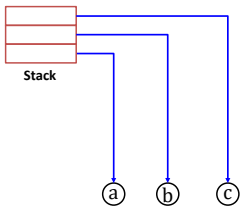
How to build an expression tree

a b **c * + d e * f + g * +**



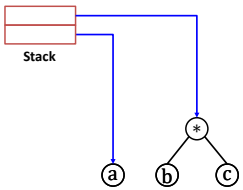
How to build an expression tree

a b c *** + d e * f + g * +**



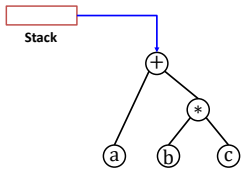
How to build an expression tree

a b c * **+ d e * f + g * +**



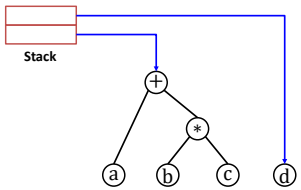
How to build an expression tree

a b c * + **d e * f + g * +**



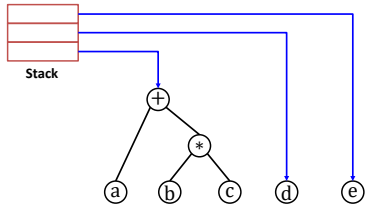
How to build an expression tree

a b c * + d **e * f + g * +**



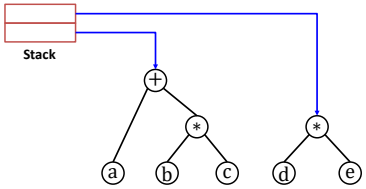
How to build an expression tree

a b c * + d e * f + g * +



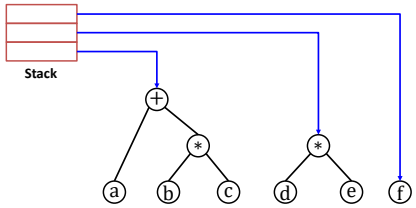
How to build an expression tree

a b c * + d e * f + g * +



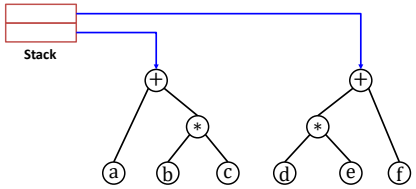
How to build an expression tree

a b c * + d e * f + g * +



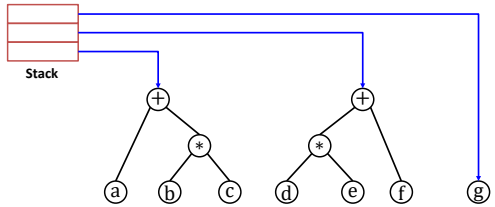
How to build an expression tree

a b c * + d e * f + g * +



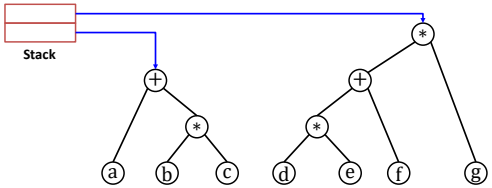
How to build an expression tree

a b c * + d e * f + g * +



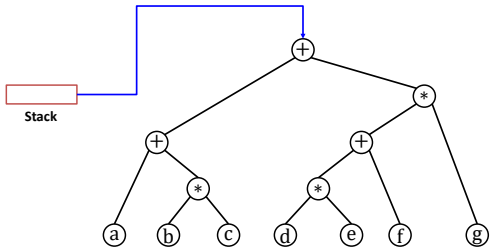
How to build an expression tree

a b c * + d e * f + g * +



How to build an expression tree

a b c * + d e * f + g * +



Example: expression trees

```python
def build_expr(expr: str) -> Exprtree:
    """Builds an expression tree from a correct
       expression represented in postfix notation"""
    # Create a list of all characters (without spaces)
    expr_char = [x for x in expr if not x.isspace()]
    stack: list[Node[str]] = []
    for c in expr_char:
        if c.isalpha():
            # We have an operand. Create a leaf node
            stack.append(Node(c))
        else:
            # We have an operator (+ or *)
            right = stack.pop()
            left = stack.pop()
            stack.append(Node(c, left, right))
    # The stack has only one element: the root of the expression
    return stack.pop()
```


Example: expression trees

```python
def infix_expr(t: Exprtree) -> str:
    """Generates a string with the expression in
       infix notation"""

    if not t.left:  # it is a leaf node (operand)
        return t.data

    # We have an operator. Add enclosing parenthesis (for safety)
    return '(' + infix_expr(t.left) + t.data +
           infix_expr(t.right) + ')'
```

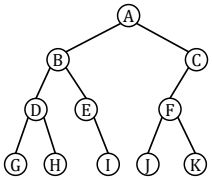
Inorder traversal: node is visited *between* the left and right children.

Exercise: redesign infix_expr to minimize the number of parenthesis.

Example: expression trees

```python
def eval_expr(t: Exprtree, v: dict[str, int]) -> int:
    """Evaluates an expression taking v as the value of the
       variables (e.g., v['a'] contains the value of a)"""

    if not t.left:  # it is a leaf node: return the value
        return v[t.data]

    # We have an operator: evaluate subtrees and operate
    left = eval_expr(t.left, v)
    right = eval_expr(t.right, v)
    return left + right if t.data == '+' else left * right
```


Tree traversals



Let us consider generators to visit the nodes of the tree in some specific order.

```python
t: BinTree[str] = … # some tree constructor

Lpreorder  = [n.data for n in preorder(t)]
Lpostorder = [n.data for n in postorder(t)]
Linorder   = [n.data for n in inorder(t)]
Llevels    = [n.data for n in level_order(t)]
```

```
Lpreorder:  ['A', 'B', 'D', 'G', 'H', 'E', 'I', 'C', 'F', 'J', 'K']

Lpostorder: ['G', 'H', 'D', 'I', 'E', 'B', 'J', 'K', 'F', 'C', 'A']

Linorder:   ['G', 'D', 'H', 'B', 'E', 'I', 'A', 'J', 'F', 'K', 'C']

Llevels:    ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
```

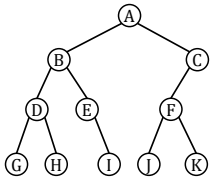

Tree traversals

```python
# Remember:
#     BinTree = Optional[Node[T]]
#     NodeIter = Iterator[Node[T]]

def preorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in preorder"""
    if t:
        yield t.data
        yield from preorder(t.left)
        yield from preorder(t.right)


def postorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in postorder"""
    if t:
        yield from postorder(t.left)
        yield from postorder(t.right)
        yield t.data
```


Tree traversals

```python
def inorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in inorder"""
    if t:
        yield from inorder(t.left)
        yield t.data
        yield from inorder(t.right)


def level_order(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes by levels"""
    if not t:
        return
    q: deque[Node] = deque([t])
    while q:
        n = q.popleft()
        yield n
        if n.left:
            q.append(n.left)
        if n.right:
            q.append(n.right)
```


Tree visitors



A visitor is a function that is applied to all nodes of a tree.

Similar to the **map** function applied to iterables (e.g., lists)

```python
def visit_preorder(t: BinTree[T], f: Callable[[T], T]) -> None:
    """Visits all the nodes of the tree in preorder and applies
       f() to the data. The result is reassigned to the data"""
```

Type: `Callable[[T1,…Tn], Tr]`.
A function with parameters `[T1,…,Tn]` and result `Tr`.

Tree visitors

```python
def visit_preorder(t: BinTree[T], f: Callable[[T], T]) -> None:
    """Applies f to all data in preorder"""
    if t:
        t.data = f(t.data)
        visit_preorder(t.left, f)
        visit_preorder(t.right, f)

# Example
def square(x: int) -> int:
    return x*x

t: Bintree[int] = …  # some tree constructor
visit_preorder(t, square) # squares all data in the tree

# equivalent with lambda: visit_preorder(t, lambda x: x*x)
```


EXERCISES

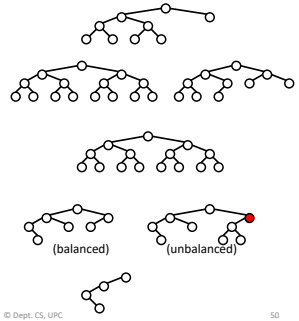
Expression tree

- Modify `infixExpr` for a nicer printing:
  - Minimize number of parenthesis.
  - Add spaces around + (but not around ∗).
- Extend the functions to support other operands, including the unary – (e.g., –a/b).

Binary tree types

Design the function "`def check_type(t: BinTree) -> bool:`" for each type tree.

- **Full Binary Tree**: each node has 0 or 2 children.
- **Complete Binary Tree:** all levels are filled entirely with nodes, except the lowest level. In the lowest level, all nodes reside on the left side.
- **Perfect Binary Tree:** all the internal nodes have exactly two children and all leaves are at the same level.
- **Balanced Binary Tree:** the tree height is $O(\log n)$, where n is the number of nodes. The height of the left and right subtrees of each node should vary by at most one.
- **Degenerated Binary Tree:** every internal node has a single child.



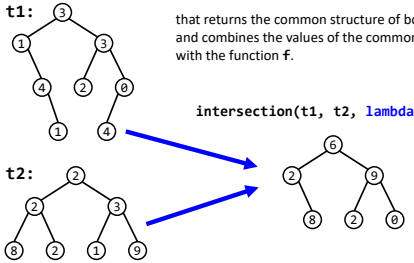
Intersection of binary trees

Design the function

```
def intersection(t1: BinTree[T], t2: BinTree[T],
                 f: Callable[[T, T], T]) -> BinTree[T]:
```

that returns the common structure of both trees and combines the values of the common nodes with the function `f`.

```
intersection(t1, t2, lambda x, y: x*y)
```



Traversals: Full Binary Trees

- A Full Binary Tree is a binary tree where each node has 0 or 2 children.
- Draw the full binary trees corresponding to the following tree traversals:
  - Preorder: 2 7 3 6 1 4 5; Postorder: 3 6 7 4 5 1 2
  - Preorder: 3 1 7 4 9 5 2 6 8; Postorder: 1 9 5 4 6 8 2 7 3
- Given the pre- and post-order traversals of a binary tree (not necessarily full), can we uniquely determine the tree?
  - If yes, prove it.
  - If not, show a counterexample.

Traversals: Binary Trees

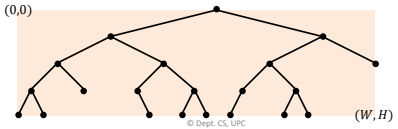
- Draw the binary trees corresponding the following traversals:
  - Preorder: 3 6 1 8 5 2 4 7 9; Inorder: 1 6 3 5 2 8 7 4 9
  - Level-order: 4 8 3 1 2 7 5 6 9; Inorder: 1 8 5 2 4 6 7 9 3
  - Postorder: 4 3 2 5 9 6 8 7 1; Inorder: 4 3 9 2 5 1 7 8 6
- Describe an algorithm that builds a binary tree from the preorder and inorder traversals.

Drawing binary trees

We want to draw the skeleton of a binary tree as it is shown in the figure. For that, we need to assign (x, y) coordinates to each tree node. The layout must fit in a pre-defined bounding box of size $W \times H$, with the origin located in the top-left corner. Design the function:

```
T = TypeVar('T')
Coordinate = tuple[float, float]
Coordinates = dict[Bintree, Coordinate]

def draw(t: Bintree, w: float, h: float) -> Coordinates:
```

that returns a dictionary with the coordinates of all tree nodes in such a way that the lines that connect the nodes do not cross.

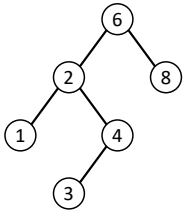


Containers: Binary Search Trees

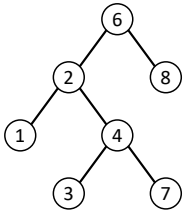
Jordi Cortadella and Jordi Petit
Department of Computer Science

Binary Search Trees

BST property: for every node in the tree with value V:
- All values in the left subtree are smaller than V.
- All values in the right subtree are larger than V.

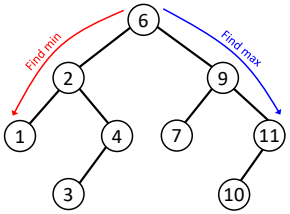


This is a binary search tree



This is ***not*** a binary search tree

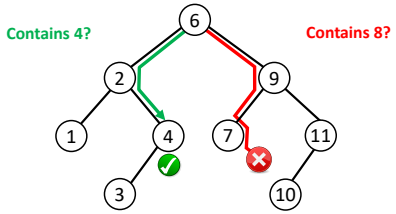
BST: find min/max



Find min: Go to the leftmost element.

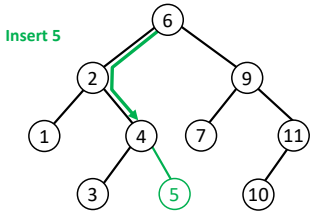
Find max: Go to the rightmost element.

BST: find an element



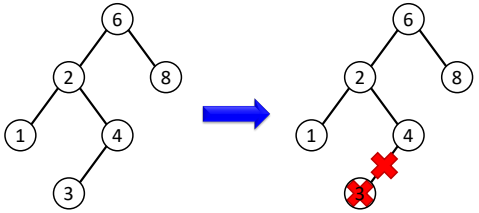
Find an element:
- Move to left/right depending on the value.
- Stop when:
  - The value is found (contained)
  - No more elements exist (not contained)

BST: insert an element



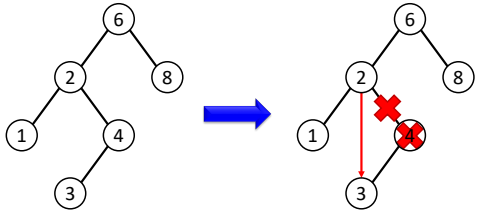
Insert:
- Move to left/right depending on the value.
- Stop when the element is found (nothing to do) or a null is found.
- If not found, substitute null by the new element.

remove: simple case (no children)



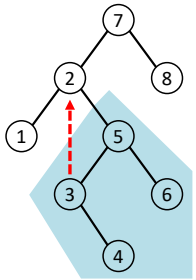
`remove(3)`

remove: simple case (one child)



`remove(4)`

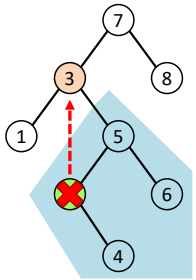
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.

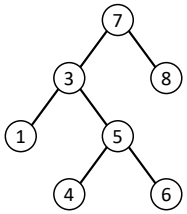
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

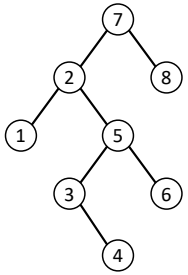
remove: complex case (two children)



`remove(2)`

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

Visiting the items in ascending order



Question:

How can we visit the items of a BST in ascending order?

Answer:

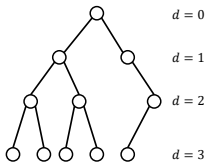
Using an **<u>in-order</u>** traversal

BST: runtime analysis

- Copying and deleting the full tree takes $O(n)$.
- We are mostly interested in the runtime of the `insert/remove/contains` methods.
  - The complexity is $O(d)$, where d is the depth of the node containing the required element.
- But, how large is d?

BST: runtime analysis

- Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.



$\text{ILP} = 0 \times 1 + 1 \times 2 + 2 \times 3 + 3 \times 5 = 23$

$\text{Avg. IPL} = \frac{23}{11} \approx 2.09$

BST: runtime analysis

- Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.
- $D(n)$ is the IPL of a tree with n nodes. $D(1) = 0$. The left subtree has i nodes and the right subtree has $n - i - 1$ nodes. Thus,

$$D(n) = D(i) + D(n - i - 1) + (n - 1)$$

- If all subtree sizes are equally likely, then the average value for $D(i)$ and $D(n - i - 1)$ is

$$\frac{1}{n}\sum_{j=0}^{n-1} D(j)$$

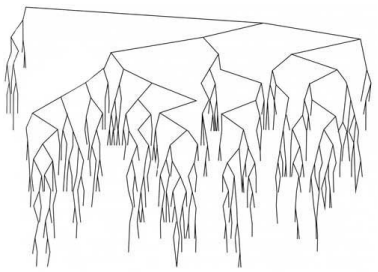
BST: runtime analysis

- Therefore,

$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1$$

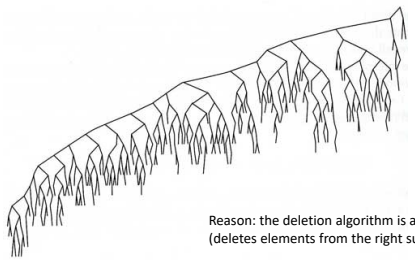
- The previous recurrence gives: $D(n) = O(n \log n)$
- The average height of nodes after n random insertions is $O(\log n)$.
- However, the $O(\log n)$ average height is not preserved when doing deletions.

Random BST



Source: Fig 4.29 of Weiss textbook

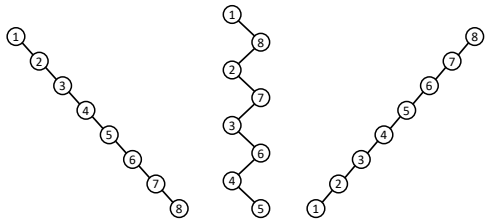
Random BST after n^2 insert/removes



Reason: the deletion algorithm is asymmetric (deletes elements from the right subtree)

Source: Fig 4.30 of Weiss textbook

Worst-case runtime: $O(n)$



Self-balancing BST

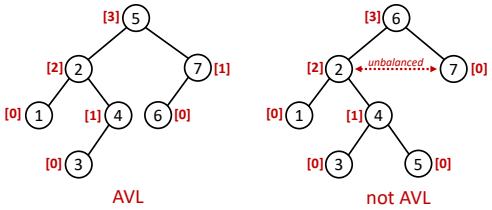
- The worst-case complexity for insert, remove and search operations in a BST is $O(n)$, where n is the number of elements.
- Various representations have been proposed to keep the height of the tree as $O(\log n)$:
  - AVL trees
  - Red-Black trees

AVL trees

- Named after Adelson-Velsky and Landis (1962).
- Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.
- Properties:
  - The height of the tree is always $\Theta(\log n)$.
  - The time devoted to balancing is $O(\log n)$.

AVL tree: definition

- An AVL tree is a BST such that, for every node, the difference between the heights of the left and right subtrees is at most 1.



AVL tree in action

https://en.wikipedia.org/wiki/AVL_tree

Complexity

- Single and double rotations only need the manipulation of few pointers and the height of the nodes ($O(1)$).
- Insertion: the height of the subtree after a rotation is the same as the height before the insertion. Therefore, at most only one rotation must be applied for each insertion.
- Deletion: more complicated. More than one rotation might be required.
- Worst case for deletion: $O(\log n)$ rotations (a chain effect from leaves to root).

EXERCISES

BST

- Starting from an empty BST, depict the BST after inserting the values 32, 15, 47, 67, 78, 39, 63, 21, 12, 27.
- Depict the previous BST after removing the values 63, 21, 15 and 32.

Methods of a BST

Use the class **BinTree** from the previous chapter and implement the following methods for a BST:

```
def find_min(t: BinTree[T]) -> BinTree[T]:
    """Returns the tree (node) containing the min element"""

def find_max(t: BinTree[T]) -> BinTree[T]:
    """Returns the tree (node) containing the max element"""

def find(t: BinTree[T], data: T) -> BinTree[T]:
    """Returns the tree (node) where data is located,
       or None if not found"""

def insert(t: BinTree[T], data: T) -> BinTree[T]:
    """Inserts data on the BST. It returns the new
       BinTree after insertion (t if it was not empty)"""

def remove(t: BinTree[T], data: T) -> BinTree[T]:
    """Removes data from the BST. It returns the new
       BinTree (None if it becomes empty)"""
```


Merging BSTs

- Describe an algorithm to generate a sorted list from a BST. What is its cost?
- Describe an algorithm to create a balanced BST from a sorted list. What is its cost?
- Describe an algorithm to create a balanced BST that contains the union of the elements of two BSTs. What is its cost?

Hashing

Jordi Cortadella and Jordi Petit
Department of Computer Science

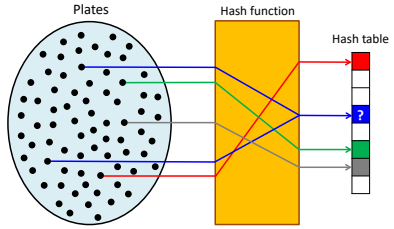
The parking lot

- We want to keep a database of the cars inside a parking lot. The database is automatically updated each time the cameras at the entry and exit points of the parking read the plate of a car.
- Each plate is represented by a free-format short string of alphanumeric characters (each country has a different system).
- The following operations are needed:
  - Add a plate to the database (when a car enters).
  - Remove a plate from the database (when a car exits).
  - Check whether a car is in the parking.
- **Constraint**: we want the previous operations to be very efficient, i.e., executed in ***constant time***.
  (This constraint is overly artificial, since the activity in a parking lot is extremely slow compared to the speed of a computer.)

Naïve implementation options

- Lists, vectors or binary search trees are not valid options, since the operations take too long:
  - Unsorted lists: adding takes $O(1)$. Removing/checking takes $O(n)$.
  - Sorted vector: adding/removing takes $O(n)$. Checking takes $O(\log n)$.
  - AVL trees: adding/removing/checking takes $O(\log n)$.
- A (Boolean) vector with one location for each possible plate:
  - The operations could be done in constant time!, but …
  - The vector would be extremely large (e.g., only the Spanish system can have 80,000,000 different plates).
  - We may not even know the size of the domain (all plates in the world).
  - Most of the vector locations would be "empty" (e.g. assume that the parking has 1,000 places).
- Can we use a data structure with size $O(n)$, where n is the size of the parking?

Hashing



A hash function maps data of arbitrary size to a table of fixed size.
Important questions:
- How to design a good hash function?
- The hash function is not injective. How to handle collisions?

Hash function

- We can calculate the location for item x as

$$h(x) \mod S$$

where h is the hash function and S is the size of the hash table.

- A good hash function must scatter items *uniformly* (to minimize the impact of collisions).
- A hash function must also be *consistent*, i.e., give the same result each time it is applied to the same item.

Hashing the plates: some attempts

- Add the last three characters (e.g., ASCII codes) of plate:

$$h(x) = x_{n-1} + x_{n-2} + x_{n-3}$$

Bad choice: For the Spanish system, this would concentrate the values between 198 (BBB) and 270 (ZZZ).

- Multiply the last three characters:

$$h(x) = x_{n-1} \cdot x_{n-2} \cdot x_{n-3}$$

The values are distributed between 287,496 and 729,000. However the distribution is not uniform. The last three characters denote the age of the car. The population of new cars is larger than the one of old cars (e.g., about 15% of the cars are less than 1-year old).

Moreover: consecutive plates would fall into the same slot. Some companies (e.g., car renting) have cars with consecutive plates and they could be located in the neighbourhood of the parking lot.

Hashing the plates: some attempts

- Multiply all characters of the plate:

$$h(x) = x_0 \cdot x_1 \cdots x_{n-1}$$

Better choice, but not fully random and uniform. Two plates with permutations of characters would fall into the same slot, e.g., 3812 DXF and 8321 FDX.

- The perfect hash function does not exist, but using ***prime numbers*** is a good option since most data have no structure related to prime numbers.
- Where can we use prime numbers?
  - In the size of the hash table
  - In the coefficients of the hash function

Example of hash function for strings

A usual hash function for a string with size n is the *polynomial rolling hash function*:

$$h(x) = \sum_{i=0}^{n-1} x_i \cdot p^i \mod m$$

where
- p is a prime number (e.g., 29791, 11111, …)
- m is a large prime (to reduce the probability of collision) $10^9 + 7$ and $10^9 + 9$ are widely used.
- and x_i is the character at location i

This function can be efficiently implemented using Horner's rule for the evaluation of a polynomial.

Polynomial rolling hash function

```python
def hash(s: str)-> int:
    """Hash function for strings"""
    p, m = 31, 10**9 + 7
    h = 0
    p_pow = 1
    for c in s:
        h = (h + ord(c) * p_pow) % m
        p_pow = (p_pow * p) % m
    return h

print(hash('hello'), hash('bye'), hash('hash'))
105835282 100910 3387231
```

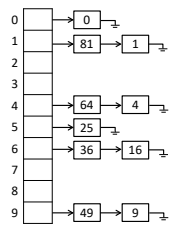

Handling collisions

- A collision is produced when

$$h(x_1) \equiv h(x_2) \mod S$$

- There are two main strategies to handle collisions:
  - Using lists of items with the same hash value (separate chaining)
  - Using alternative cells in the same hash table (linear probing, double hashing, …)

Handling collisions: separate chaining



(perfect squares mod 10)

Each slot is a list of the items that have the same hash value.

Load factor: $\lambda = \frac{\text{number of items}}{\text{table size}}$

λ is the average length of a list.

A successful search takes about $\lambda/2$ links to be traversed, on average.

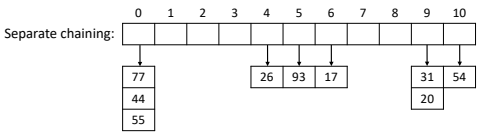
Table size: make it similar to the number of expected items.
Common strategy: when $\lambda > 1$, do rehashing.

Handling collisions: using the same hash table

- If the slot is occupied, find alternative cells in the same table. To avoid long trips finding empty slots, the load factor should be below $\lambda = 0.5$.
- Deletions must be "lazy" (slots must be invalidated but not deleted, thus avoiding truncated searches).
- **Linear probing:** if the slot is occupied, use the next empty slot in the table.
- **Double hashing:** if the slot is occupied using the first hash function h_1, use a second hash function h_2. The sequence of slots that is visited is $h_1(x)$, $h_1(x) + h_2(x)$, $h_1(x) + 2h_2(x)$, etc.

An example

Insertion of the elements 54, 26, 93, 17, 77, 31, 44, 55, 20.
Hash function: $h(x) = x \mod 11$.



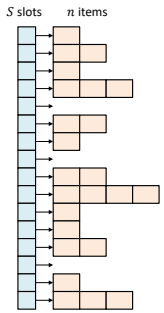
	0	1	2	3	4	5	6	7	8	9	10
Linear probing:	77	44	55	20	26	93	17			31	54

What if we remove 55? Use lazy deletion!

Rehashing

- When the table gets too full, the probability of collision increases (and the cost of each operation).
- Rehashing requires building another table with a larger size and rehash all the elements to the new table. Running time: $O(n)$.
- New size: $2n$ (or a prime number close to it). Rehashing occurs very infrequently and the cost is amortized by all the insertions. The average cost remains constant.

Complexity analysis



The hash table occupies $O(S + n)$ space.
Each slot has n/S items, on average.
The runtime to find an item is $O(n/S)$, on average.

Cases	Space: $O(n + S)$	Time: $O(n/S)$
$S \gg n$	$O(S)$	$O(1)$
$n \gg S$	$O(n)$	$O(n)$
$S = O(n)$	$O(n)$	$O(1)$

The best strategy is to have $S = O(n)$ that allows to maintain a constant-time access without wasting too much memory.

Rehashing should be applied to maintain $S = O(n)$.

Sets and Dictionaries

- A set: a collection of items. The typical operations are:
  - Add/remove one element
  - Does it contain an element?
  - Size?, Is it empty?
  - Visit all items
- A dictionary (map): a collection of key-value pairs. The typical operations are:
  - Put a new key-value pair
  - Remove a key-value pair with a specific key
  - Get the value associated to a key
  - Does it contain a key?
  - Visit all key-value pairs

Sets and Dictionaries

- A dictionary can be treated as a set of keys, each key having an associated value.
- We will focus on the implementation of sets.

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

key value

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
python.org	18.21.3.144
sourceforge.net	51.98.23.53

set

dictionary

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Source: Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper

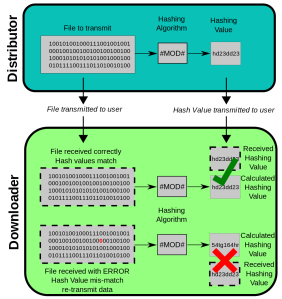
Binary Search Trees vs. Hash Tables

Operation	Binary Search Tree	Hash Table
Insertion/Deletion/Lookup	$O(\log n)$	$O(1)$
Sorted Iteration	In-order traversal: $O(n)$	Needs an extra sorted vector: $O(n \log n)$
Hash function	Not required	Required
Total order	Required	Not required
Min/max search	$O(\log n)$	$O(n)$
Range search	$O(\log n)$	$O(n)$

Python:
- `set` and `dict` are implemented with hash tables

C++ STL:
- `set` and `map` are implemented with BSTs
- `unordered_set` and `unordered_map` are implemented with hash tables

Application: data integrity check

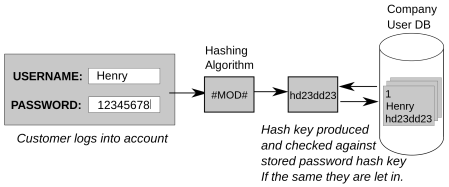


Hash functions are used to guarantee the integrity of data (files, messages, etc) when distributed between different locations.

Different hashing algorithms exist: SHA1, SHA255, ...

The probability of collision is extremely low.

Application: password verification



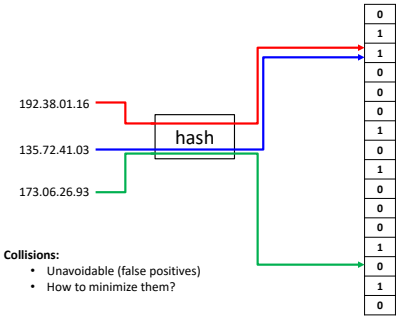
Security is based on the fact that hashing functions are cryptographic (not reversible).

Be careful: there are databases of hash values for "popular" passwords (e.g., 123456, qwerty, password, barcelona, android,...).

Bloom filters

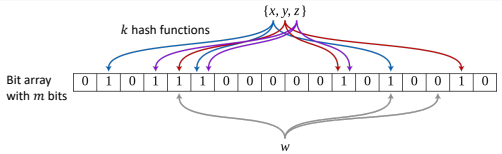
- Let us assume that we have an internet router that wants to filter the packets from a set of blacklisted IPs. The router has an speed of 100 Gbits/s. New IPs can be added to the blacklist dynamically.
- IPv4 addresses have 32 bits (4,294,967,296 possible addresses).
- Each time the router receives a packet, it must perform a search to check whether the IP is blocked. In most cases, the IP will not be blocked.
- We would like to have a *fast* and *small* data structure to avoid the packets waiting for the search of their IP.
- Bloom filters give a space-efficient solution to the membership testing problem. They do not store the keys of the set. But the answer is probabilistic (small probability of false positives).

Simple Bloom filter



Collisions:
- Unavoidable (false positives)
- How to minimize them?

How a Bloom filter works

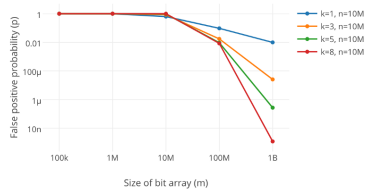


- Initially, all array positions are at 0.
- To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
- To test whether an element is in the set, feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is not in the set. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a *false positive*.
- Note: "basic" Bloom filters do not allow to delete elements.

Bloom filter: probabilistic analysis

- Goal: given a set of elements, what is the size of the bit array and the number of hash functions required to achieve a false positive rate ε?
- Assumptions for the probabilistic analysis:
  - Hash functions select each position of the array with equal probability
  - Different hash functions are independent

Bloom filter probabilistic analysis



Source: Abishek Bhat, "Use the bloom filter, Luke!"
https://medium.com/engineering-semantics3/use-the-bloom-filter-luke-b59fd0839fc4

Intuition:
- As we increase the size of the array (m), false positives decrease
- As we increase the number of hash functions (k), ... (not clear)

Bloom filter: probabilistic analysis

- Probability that a bit is not set to 1 by any of the hash functions:

$$\left(1-\frac{1}{m}\right)^k$$

- A well-known identity:

$$\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^m=\frac{1}{e}$$

- Thus, for large m:

$$\left(1-\frac{1}{m}\right)^k=\left(\left(1-\frac{1}{m}\right)^m\right)^{\frac{k}{m}}\approx e^{-\frac{k}{m}}$$

- After inserting n elements, the probability that a certain bit is still 0 is

$$e^{-\frac{kn}{m}}$$

- And the probability that is 1 is

$$1-e^{-\frac{kn}{m}}$$

Bloom filter: probabilistic analysis

- The probability of a false positive (ε) is the probability that each of the k positions computed by the hash functions are at 1, i.e.,

$$\varepsilon=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k\approx\left(1-e^{-\frac{kn}{m}}\right)^k$$

- For a given n and m, the optimal number of hash funcions to minimize ε is:

$$k=\frac{m}{n}\ln 2\approx 0.693\frac{m}{n}$$

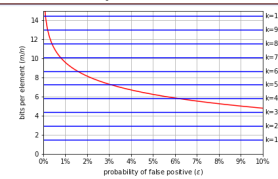
- Given ε and n, the optimal size of the bit array and number of hash funcions is:

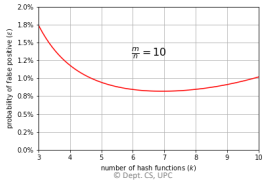
$$m=-\frac{n\log_2\varepsilon}{\ln 2}\qquad k=-\log_2\varepsilon$$

- Equivalently, the number of bits per element must be

$$\frac{m}{n}\approx -1.44\log_2\varepsilon$$

Bloom filter: probabilistic analysis





EXERCISES

Hash function

Given the values {2341, 4234, 2839, 430, 22, 397, 3920}, a hash table of size 7, and hash function $h(x) = x \bmod 7$, show the resulting tables after inserting the values in the given order with each of these collision strategies:

- Separate chaining
- Linear probing

All elements different

Let us assume that we have a list with n elements. Design an algorithm that can check that all elements are different. Analyze the complexity of the algorithm considering different data structures:

- Checking the elements without any additional data structure, i.e., using the same list
- Using balanced BSTs
- Using hash tables

String matching

- Let us assume that we have two strings, a pattern s and a text t. Propose an algorithm to find all occurrences of the pattern in the text.
- Constraint: we want to do it in $O(|s| + |t|)$ time.
- Hint: Use the polynomial rolling hash function. Propose an efficient (incremental) method to calculate $\text{hash}(t[i \dots j])$ from $\text{hash}(t[i-1 \dots j-1])$.

Bloom filter parameters

- Let us consider a Bloom filter with $n = 10^6$ elements.
  - Give the size of the bit array and the number of hash functions to achieve a 1% false positive rate
- Let us consider a Bloom filter with $m = 10^7$.
  - What is the maximum number of elements it must contain to guarantee a false positive rate smaller than 2%?
  - How many hash functions would be required?

Bloom filter

Let us consider a Bloom filter with $m = 17$ and three hash functions:

$$h_1(x) = 19x, \quad h_2(x) = 23x, \quad h_3(x) = 31x$$

Insert the values 1 and 2, and give the smallest positive integer than produces a false positive

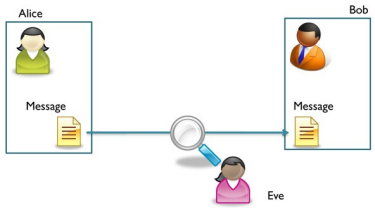
Cryptography

Jordi Cortadella and Jordi Petit
Department of Computer Science

Where do we need cryptography?

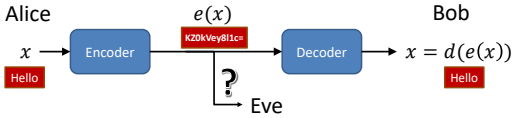
- Communication (e.g., sending private emails).
- Digital signatures, i.e., guarantee that digital documents are authentic.
- Network services over unsecure networks (e.g., secure shell (ssh) for remote login, file transfers, remote command execution, etc.).
- HyperText Transfer Protocol Secure (HTTPS): secure communication on Internet.
- Cryptocurrencies (e.g., bitcoin)

Cryptography



- How can we avoid an eavesdropper (Eve) to overhear a message sent from Alice to Bob?
- Solution: encrypt the message!

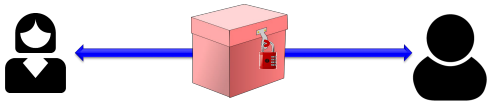
Cryptosystem



- The encryption function $e(x)$ must be invertible. The inverse is $d(\cdot)$.
- Two schemes:
  - The traditional: Secret-key protocols (symmetric).
  - The modern: Public-key protocols (asymmetric).

Secret-key protocols

- Alice and Bob have to meet privately and chose a secret key.
- They can use the secret key to mutually exchange messages.



- There are many secret-key protocols. We will explain two of them:
  - XOR encoding.
  - Advanced Encryption Standard (AES).

Secret-key protocol: XOR encoding

- A secret key r is chosen (a binary string).
- The encoding and decoding functions are identical: $e_r(x) = d_r(x) = x \oplus r$.
- Example: $r = 11011100$.

- It is convenient that the bits of r are randomly generated.
- Still, this is not a very robust scheme since Eve can figure out important information by listening several messages.

Secret-key protocol: XOR encoding

- If the key is too short, it needs to be applied many times (once for each block). Messages often show similarities and repeated patterns (same header, same tail, long sequences of zeros, ...).
- If we send two messages, $e_r(x)$ and $e_r(z)$, then $e_r(x) \oplus e_r(z)$ may reveal important information. It is convenient to change the key at every message (one-time pad).

$r = 0101$

$x = 1001\ 0000\ 0000\ 0000\ 0000\ 1011\ 1010$
$z = 1001\ 1001\ 0000\ 0000\ 0000\ 0000\ 0111$

$e_r(x) = 1100\ 0101\ 0101\ 0101\ 0101\ 1110\ 1111$
$e_r(z) = 1100\ 1100\ 0101\ 0101\ 0101\ 0101\ 0010$

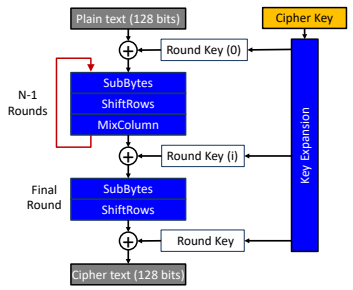
$e_r(x) \oplus e_r(z) = 0000\ 1001\ 0000\ 0000\ 0000\ 1011\ 1101$

Same header. Same sender?
The key !!!
Long identical sequence. Maybe zeros?

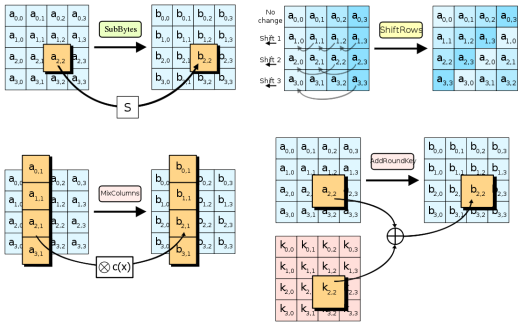
Secret-key protocol: AES

- AES: Advanced Encryption Standard.
- Established as a standard by the U.S. National Institute of Standards and Technology (NIST) in 2001.
- Very robust and used worldwide.
- A family of ciphers with different key and block sizes (key sizes: 128, 196 and 256 bits).

AES scheme

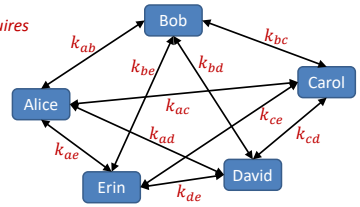


AES steps



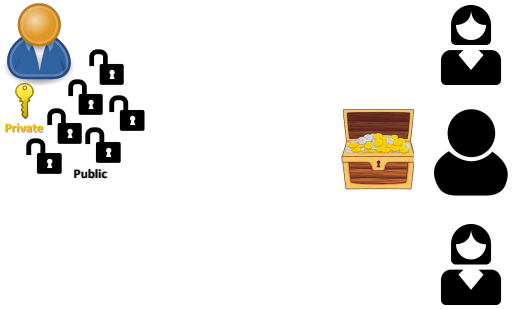
Secret-key protocols: problems

Every channel requires a different key

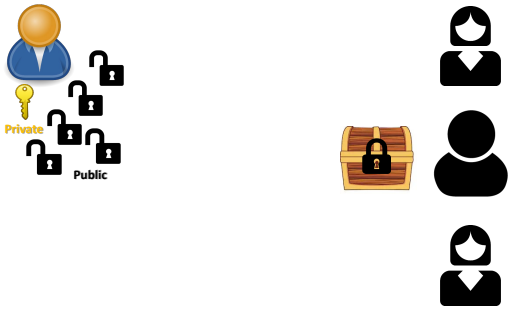


The key cannot be transmitted through the communication channel !

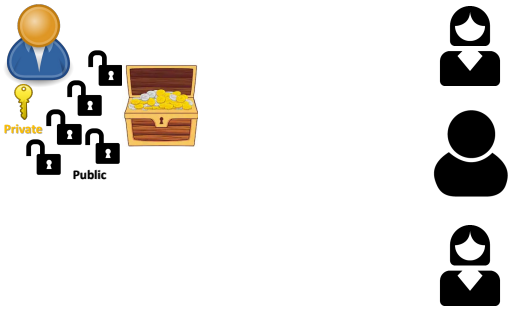
Public-key protocols



Public-key protocols



Public-key protocols



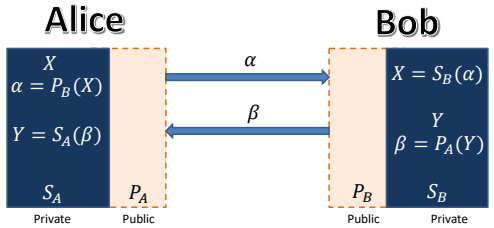
Public-key protocols

- Each participant generates a *public key* (P) and a (private) *secret key* (S). Public keys are revealed to everybody.
- The public/secret keys are a matched pair, i.e.,

$$M = S(P(M)) = P(S(M))$$

- If Alice has the pair (P_A, S_A), anybody can compute $P_A(X)$, but only Alice can compute $S_A(X)$.
- If Bob wants to send a secret message M to Alice, Bob will compute $X = P_A(M)$ and send it to Alice. Only Alice will be able to decipher the message: $M = S_A(X)$.

Public-key protocols



But, how to create a cryptosystem like this?
Using number theory.

RSA cryptosystem

- Public-key cryptosystem (Rivest-Shamir-Adleman, 1977).
- Based upon number theory: modular arithmetic and prime numbers.
- Security: based on the fact that factoring a large number (product of two large primes) is hard.

Bézout's identity

- **Lemma:** If d divides both a and b, and $d = ax + by$ for some integers x and y, then necessarily $d = \gcd(a, b)$.
- **Proof:**
  - Clearly, $d \le \gcd(a, b)$, since d is a divisor of a and b.
  - Since $\gcd(a, b)$ is a divisor of a and b, it must also be a divisor of $ax + by = d$. This implies that $\gcd(a, b) \le d$.
  - Therefore, $d = \gcd(a, b)$.

Extended Euclid's algorithm

```
function extendedGcd(a,b)
// Input: two positive integers a and b, with a ≥ b ≥ 0
// Output: Integers x,y,d such that d = gcd(a,b) and ax + by = d
  if b = 0: return (1,0,a)
  (x',y',d) = extendedGcd(b, a mod b)
  return (y', x' − ⌊a/b⌋y', d)
```

Proof by induction using the identity $a \bmod b = a - \lfloor a/b \rfloor b$

extgcd(25,11)	(4,-9,1)	$4 \cdot 25 - 9 \cdot 11 = 1$
extgcd(11,3)	(-1,4,1)	$-1 \cdot 11 + 4 \cdot 3 = 1$
extgcd(3,2)	(1,-1,1)	$1 \cdot 3 - 1 \cdot 2 = 1$
extgcd(2,1)	(0,1,1)	$0 \cdot 2 + 1 \cdot 1 = 1$
extgcd(1,0)	(1,0,1)	$1 \cdot 1 + 0 \cdot 0 = 1$

Modular arithmetic: properties

- $x \bmod N$ is the remainder when x is divided by N.

$$x \equiv y \pmod N \quad \Leftrightarrow \quad N \text{ divides } (x - y).$$

- Addition and multiplication. Let us assume $a_1 \equiv b_1 \pmod N$ and $a_2 \equiv b_2 \pmod N$, then

$$a_1 + a_2 \equiv b_1 + b_2 \pmod N$$
$$a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod N$$

- Exponentiation. Let us assume $a \equiv b \pmod N$, then

$$a^k \equiv b^k \pmod N$$

Modular arithmetic: properties

- Given a and N, we say that x is the multiplicative inverse of $a \pmod N$ if

$$ax \equiv 1 \pmod N.$$

Example. The multiplicative inverse of 4 (mod 7) is 2:

$$4 \cdot 2 \equiv 1 \pmod 7$$

- When $\gcd(a, N) = 1$, the multiplicative inverse of a always exists and can be calculated by the extended Euclid's algorithm:

$$ax + Ny = 1 \quad \Rightarrow \quad x \text{ is } a\text{'s inverse } (\bmod N)$$

Fundamental property

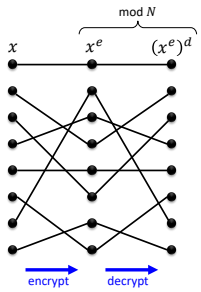
Let p and q be any two primes and $N = pq$. $\phi(N) = (p-1)(q-1)$ is the totient of N, i.e., the number of positive integers smaller than N which are co-prime to N.

For any e co-prime to $\phi(N)$:

1. The mapping $x \mapsto x^e \bmod N$ is a bijection on $\{0, 1, \ldots, N-1\}$.
2. The inverse mapping can be obtained as follows. Let d be the inverse of e modulo $\phi(N)$. Then for all $x \in \{0, \ldots, N-1\}$,

$$(x^e)^d \equiv x \bmod N.$$

The RSA cryptosystem



Bob chooses public and secret keys:

- Bob picks two large random primes, p and q.
- The public key is (N, e), where $N = pq$ and e is a small number co-prime to $(p-1)(q-1)$.
- The secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the extended Euclid's algorithm.

Alice sends a message x to Bob:

- Alice takes Bob's public key (N, e) and sends $y = (x^e \bmod N)$.
- Bob decodes the message by computing $y^d \bmod N$.

The RSA cryptosystem: example

- Let $p = 5$ and $q = 17$, thus $N = 85$ and $\phi(N) = 64$.
- Let $e = 3$. It satisfies: $\gcd(e, \phi(N)) = \gcd(3, 64) = 1$.
- We calculate $d = 3^{-1} \bmod 64 = 43$ using extended Euclid's algorithm:

$$43 \cdot 3 - 2 \cdot 64 = 1$$

Note: the algorithm gives $1 \cdot 64 - 21 \cdot 3 = 1$, but $-21 = 43 \pmod{64}$

- Let us consider the message $x = 12$.
  - The sender must encrypt x as $y = 12^3 \bmod 85 = 28$.
  - The receiver must decrypt y by computing $x = 28^{43} \bmod 85 = 12$.
- **Remember:** x^k can be efficiently computed with $\log_2 k$ multiplications.
  Note: Multiplication and division of "long" numbers is required (similar to multiplication of polynomials).

Why is RSA secure?

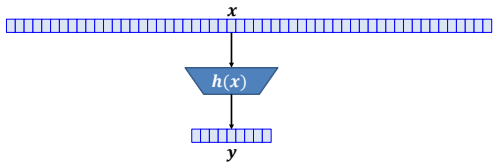
- Typical sizes for p and q are 1024-bit numbers with values larger than $2^{1023.5} \approx 1.8 \times 10^{308}$.
- Eve knows the public key (N, e) and the message y. How can she guess x? There are two options:
  1. Try all possible values of x and check whether $y = x^e \bmod N$. But x is a large n-bit number and checking all values would take exponential time (impractical).
  2. Try to guess d and calculate $x^d \bmod N$. This would require to calculate the inverse of e modulo $(p-1)(q-1)$. But p and q are not known unless the factors of N are calculated. Factoring is still a hard problem.

Hybrid cryptosystems

- Public-key cryptosystems (e.g., RSA) are convenient (no need to share keys) but computationally expensive. Secret-key (symmetric) cryptosystems (e.g. AES) are more efficient. Both can be combined.
- Bob wants to send an encrypted message to Alice:
  - Bob generates a new symmetric key k and encrypts the data with this key (using AES).
  - Bob encrypts k using Alice's public key (using RSA).
  - Bob sends both encryptions to Alice.
- Alice wants to decrypt Bob's message:
  - Alice uses her private key to decrypt the encrypted symmetric key k.
  - Alice uses the symmetric key k to decrypt the data (using AES).

Cryptographic hash function (CHF)

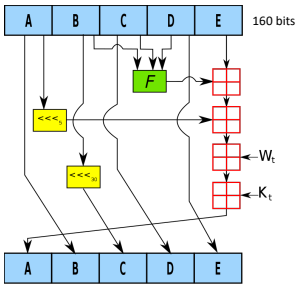
A CHF maps data of arbitrary size to a fixed-size bit string.



Cryptographic hash function (CHF)

- Properties:
  - **Easy** to compute.
  - **Pre-image resistance**: if $y = h(x)$, it is difficult to find x from y.
  - **Collision resistance**: It is difficult to find two inputs, x_1 and x_2, such that $h(x_1) = h(x_2)$.
- Popular CHFs:
  - Message Digest: MD2, MD4, MD5 and MD6. It is a 128-bit hash function.
  - Secure Hash Function: SHA-0, SHA-1, SHA-2, SHA-3. They produce hash values with 160 bits (SHA-1) or 256 bits (SHA-2).
  - And some others …

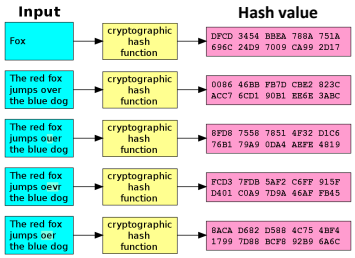
Example: SHA-1



One step of SHA-1

The result is "accumulated" to the result of previous steps.

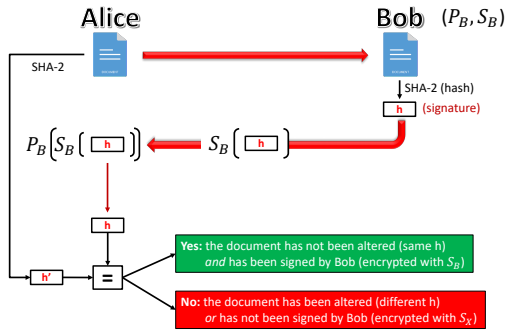
Example: SHA-1



Digital signatures

- A scheme to guarantee that a message is authentic.
- Consider the following case:
  - Alice sends a document (possibly unencrypted) to Bob and wants Bob to electronically sign the document.
  - Bob "signs" the document and sends it back to Alice.
- Questions:
  - How does Alice know that the document has not been altered? → integrity.
  - How does Alice know that Bob has signed the document (and not somebody else)? → authentication.

Digital signatures



The pending challenge

How to generate large prime numbers?

(not explained in this lecture)

Internet + Privacy → Cryptography

EXERCISES

Simple cryptographic hash

We want to use the XOR operator $\oplus$ for cryptographic hashing as follows. We split every message M into blocks B_i of 5 bits, e.g., $M = 11101 \cdot 00011 \cdot 10100 \cdot 110$. In case the length is not a multiple of 5, additional zeroes are added at the end of the message.

For a message M with k blocks, we define the cryptographic hash h as follows:

$$h(M) = B_1 \oplus B_2 \oplus \cdots \oplus B_k.$$

where $\oplus$ means the bitwise application of XOR. For example, $01110 \oplus 11010 = 10100$.

- What would be the output $h(M)$ for the previous message M?
- If we change one bit of a message, does the output change a lot?
- Assume that we know $h(M)$ and the length of M. Is it easy to find another M' with the same length such that $h(M) = h(M')$? Justify your answer.

Simple RSA

Assume you have $p = 5$ and $q = 7$.

- Which is the smallest value for e?
- What is the corresponding value for d?
- Encrypt the message $M = 3$.
- Find all possible pairs (e, d) valid for this cryptosystem.

Implement an RSA cryptosystem

- Given two primes, p and q, design an RSA cryptosystem (in Python or C++) as follows:
  - Let $N = p \cdot q$. Find the smallest $e \geq 3$, such that (N, e) can be used as public key. Use the extended gcd algorithm.
  - Find d that can be used for secret key.
  - Implement the function $\text{encode}(x, e, N)$ that computes $x^e \bmod N$. This function must be efficient. Note: assume that N^2 can be represented as an int.
  - Implement a function to double check, for $0 \leq x < N$, that $\text{encode}(\text{encode}(x, e, N), d, N) = x$.
- Example: $p = 79$, $q = 491$.

  Public key: $(38789, 11)$, Secret key: 31271.

  $e(2) = 2048$, $e(19) = 23855$, $e(32757) = 4$, $e(38788) = 38788$, $e(10) = 18550$.