**Algorithmics and Programming II: Introduction**

Jordi Cortadella and Jordi Petit  
Department of Computer Science

---

**Lecturers:**  
– Jordi Cortadella ([jordi.cortadella@upc.edu](mailto:jordi.cortadella@upc.edu))  
– Emma Rollón ([erollon@cs.upc.edu](mailto:erollon@cs.upc.edu))  
– Jordi Petit ([jordi.petit-silvestre@cs.upc.edu](mailto:jordi.petit-silvestre@cs.upc.edu))

**Sessions:**  
– Theory & (Jordi C.)  
– Lab (Emma & Jordi P.)

---

**Evaluation**

**Evaluation items:**  
– Projects (Proj), Parcial Lab (PLab), Final Theory (FTh), Final (FLab).

**Grading:**  
\[ N_1 = 0.2 \text{Proj} + 0.25 \text{PLab} + 0.25 \text{FLab} + 0.3 \text{FTh} \]  
\[ N_2 = 0.2 \text{Proj} + 0.4 \text{FLab} + 0.4 \text{FTh} \]  
\[ N = \max(N_1, N_2) \]

---

**First project: Containers**

**Design a class to manage containers.**  
**Language: Python.**

---

**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."

---

**Second Project: GPS**

**Language: Python**

---

**Slides, exercises:**  
[https://www.cs.upc.edu/~jordicf/Teaching/AP2](https://www.cs.upc.edu/~jordicf/Teaching/AP2)

**Jutge (for lab sessions):**  
[https://jutge.org](https://jutge.org)

**Lliçons (by J. Petit and S. Roura):**  
[https://lliçons.jutge.org](https://lliçons.jutge.org)
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 \( \rightarrow \mathcal{O}(n^2) \)
- We want an \( \mathcal{O}(n \log n) \) solution!

Navigation: find the shortest path

How to encrypt messages?
How many horses can you distinguish?

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

Jordi Cortadella and Jordi Petit
Department of Computer Science

Two examples

// Main loop of binary search
while (left <= right) {
    int i = (left + right)/2;
    if (x < A[i]) right = i - 1;
    else if (x > A[i]) left = i + 1;
    else return i;
}

// Main loop of insertion sort
for (int i = 1; i < A.size(); ++i) {
    int x = A[i];
    int j = i;
    while (j > 0 and A[j - 1] > x) {
        A[j] = A[j - 1];
        --j;
    }
    A[j] = x;
}

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

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

Hiding details: abstractions

Different types of abstractions

Concept maps are hierarchical: why?

The computer systems stack

Image Credit: Christopher Batten, Cornell University

The computer systems stack

Image Credit: Christopher Batten, Cornell University

How data flows through system

Boolean logic gates and functions

Combining devices to do useful work

Transistors and wires

Silicon process technology
Benefits:
We already use a certain level of abstraction: functions.

Modularity: an ADT can be changed without modifying the programs that use it

Abstract reasoning.

Specification: “what does an operation do?”

Software reuse: it can be used by other programs.

We need much more.

The computer systems stack

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

• 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

Data types

• Programming languages have a set of primitive data types (e.g., int, bool, double, char, ...).

• Each data type has a set of associated operations:
  – We can add two integers.
  – We can divide two numbers.
  – 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.

Data type: Polynomial

Operations:
• 𝑃+𝑄
• 𝑃×𝑄
• 𝑇𝑃 𝑄
• gcd(𝑃, 𝑄)
• 𝑃(𝑥)
• degree(𝑃)

𝑃 𝒙 = 𝒙𝟑 −𝟒𝒙𝟐 +𝟓

Data type: Graph

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

Abstract Data Types (ADTs)

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

Operations:
• 𝑃 + 𝑄
• 𝑃 × 𝑄
• 𝑃/𝑄
• gcd(𝑃, 𝑄)
• 𝑃(𝑥)
• degree(𝑃)

User Interface (API)

API: Application Programming Interface
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

### Classes and Objects

- A class represents a data structure, and it can have methods in addition to attributes.
- Objects are instances of classes.
- Objects interact with each other.

Example: a Point

- **File organization:** One file.
- **Implementation of the class Point:**
  - Public:
    - Constructor: different implementations
      - Point::Point(double \(x\_coord\), double \(y\_coord\))
      - Point::Point(const Point&) const
  - Private:
    - double \(x\), \(y\); // Coordinates of the point

```
#ifndef __POINT_H__
#define __POINT_H__
class Point {
public:
  // Constructor: different implementations
  Point(double x_coord, double y_coord);
  // Constructor for (0,0) Point();
  // Gets the x coordinate double getx() const;
  // Gets the y coordinate double gety() const;
  // Returns the distance to the origin double distance(const Point&) const;
  // Returns the distance to point p double distance(const Point& p) 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&) const;
private:
  double x, y; // Coordinates of the point
};
#endif
```

- **Implementation of the class Point:**
  - Operator overloading:
    - double Point::operator+(const Point& p) const
      - return Point(getX() + p.getX(), getY() + p.getY());

```
operator + (const Point& p) const {
  return Point(getX() + p.getX(), getY() + p.getY());
}
```

### OOP 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.
# File organization: two files

```cpp
#pragma once
class Point {
public:
    // Constructor
    Point(double x, double y);
    // Gets the x coordinate
    double getX() const;
private:
    double x, y; // Coordinates of the point
};
```

A header file (.hh) containing the specification and a C++ file (.cc) containing the implementation.

**Advantages:**
- Less compile effort.
- Hidden implementation.

**Disadvantages:**
- Need to distribute a library.
- Data representation still visible.

---

**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 (C++ and python allow it).
  - Recommendation: never define a public attribute. Why? See the next slides.

Public/private: let’s summarize

Class Point: a new implementation

Let us design the new type for Point

double Point::distance() const {
    return _radius;
}
// Notice that no changes are required for the + operator
// with regard to the initial implementation of the class
Point Point::operator + (const Point& p) const {
    return Point(getX() + p.getX(), getY() + p.getY());

Discussion:
- How about having 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.getX() and p.getY() would still be valid.

Recommendation (reminder):
- All attributes should be private.

A new class: Rectangle

- We will only consider orthogonal rectangles (axis-aligned).
- An orthogonal rectangle can be represented in different ways:
  - Two points (extremes of diagonal)
  - One point, width and height

Rectangle: abstract view

class Z {
public:
    void f(const Z & a) {
        // “this” attribute
        x.a ...
    }
    g(); // Ok
    x.g(); // Ok
};
// “this” attribute

int main () {
    Z v1, v2;
    v1.f(v2); // Ok
    v1.g(); // Wrong! (private)
    v1 = v2; // Ok (copy)
};
class Rectangle {
public:
    // Constructor (LL at the origin)
    Rectangle(double width, double height) :
        ll(Point(0,0)), w(width), h(height) {} 

    // LL specified at the origin
    Rectangle(const Point& p, double w, double h) :
        ll(p), w(w), h(h) {} 

    // LL and UR specified at the constructor
    Rectangle(const Point ll, const Point ur) :
        ll(ll), w(ur.getX() - ll.getX()), h(ur.getY() - ll.getY()) {} 

    // Empty rectangle (using another constructor)
    Rectangle() : Rectangle(0, 0) {} 

    // Scales the rectangle
    double scale(double s) { 
        w *= s;
        h *= s;
        return w * h; 
    } 

    // Returns the area of the rectangle
    double area() const {
        return w * h;
    } 

    // Returns the intersection with another rectangle
    Rectangle operator * (const Rectangle& R) const;
};

Rectangle::Rectangle(const Point ll, const Point ur) :
    ll(ll), w(ur.getX() - ll.getX()), h(ur.getY() - ll.getY()) {} 

What is *this?
• this is a pointer (memory reference) to the object
  (pointers will be explained later)
• *this is the object itself

Rectangle: overloaded operators
Rectangle& Rectangle::operator *= (const Rectangle& R) {
    // Calculate the ll and ur coordinates
    Point ll1 = R1.getLL();  
    ll = Point(max(ll1.getX(), ll1.getX()), max(ll1.getY(), ll1.getY()));

    // Calculate width and height (might be negative → empty)
    w = ur.getX() - ll.getX();
    h = ur.getY() - ll.getY();

    return *this;
}

Rectangle& Rectangle::operator * (const Rectangle& R) { 
    // Create a copy of itself
    Rectangle result = *this;

    result *= R;

    return result;
}

rectangle R1(4,5); // Creates a rectangle 4x5
rectangle R2(8,4); // Creates a rectangle 8x4
R1.move(2,3); // Moves the rectangle
R1.scale(1.2); // Scales the rectangle
double Area1 = R1.Area(); // Calculates the area
Rectangle R3 = R1 * R2;
if (R3.empty()) {
    return;
}

Rectangle::area() const {
    return w * h;
} 

// Notice: not a const method
void Rectangle::scale(double s) {
    w *= s;
    h *= s;
}

bool Rectangle::empty() const {
    return w <= 0 or h <= 0;
}

Rectangle: other public methods
Rectangle R1(Point(2,3), Point(6,8));
double areaR1 = R1.area();  // areaR1 = 20
rectangle R2(Point(3,5), 2, 4); // LL=(3,5) UR=(5,9)

// Check whether the point (4,7) is inside the
// intersection of R1 and R2.
bool in = (R1* R2).isPointInside(Point(4,7));
// The object R1 R2 is ”destroyed” after the assignment.
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.

What we would like to do:
Rational R1(3);       // R1 = 3
Rational R2(5, 4);   // R2 = 5/4
Rational R3(8, -10); // R3 = -4/5
R3 += R1 + Rational(-1, 5); // R3 = 2
if (R3 >= Rational(2)) {
    R3 = -R1*R2;       // R3 = -15/4
}
cout << R3.to_str() << endl;

Other private data and methods (if necessary)

class Rational {
public:
    Point Rectangle::getLL() const {
        return ll;
    }
    Point Rectangle::getUR() const {
        return ur;
    }
    Point Rectangle::getXY() const {
        return getLL();
    }
    double Rectangle::getArea() const {
        return w * h;
    }
    double Rectangle::getWidth() const {
        return w;
    }
    double Rectangle::getHeight() const {
        return h;
    }

    // Lower-left corner of the rectangle
    Point Point::getXY() const {
        return Point(0,0);
    }
    // width/height of the rect.
    double Point::getX() const {
        return x;
    }
    double Point::getY() const {
        return y;
    }
}

Rectangle::Rectangle() : Rectangle(0, 0) {}
Rectangle Rectangle::operator * (const Rectangle& R) const {
    // Make a copy of itself
    Rectangle result = *this;

    result *= R;

    return result;
}

Rectangle: a rich set of constructors
Rectangle: using the ADT
Rectangle: ADT
Rectangle: ADT (incomplete)
The class Rational

```cpp
class Rational {
  private:
  int num, den; // Invariant: den > 0 and gcd(num,den)=1
  // Simplifies the fraction (used after each operation)
  void simplify();
  public:
  // Constructor (if some parameter is missing, the default value is taken)
  Rational(int num = 0, int den = 1);
  // Returns the numerator of the fraction
  int getNum() const { return num; }
  // Returns the denominator of the fraction
  int getDen() const { return den; }
  // Returns true if the number is integer and false otherwise.
  bool isInteger() const { return den == 1; }
  ...
};
```

Rational: arithmetic and relational operators

```cpp
Rational::Rational(int num, int den) : num(num), den(den) {
    simplify();
}
void Rational::simplify() {
    assert(den != 0); // We will discuss assertions later
    if (den < 0) {
        num = -num;
        den = -den;
    }
    // Divide by the gcd of num and den
    int d = gcd(abs(num), den);
    num /= d;
    den /= d;
}
```

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
>>> a/b # uses the __repr__ method (see later)
Rational(-1/6)
```

Disclosure: recommended indentation is 4 spaces (here we use only 2 spaces for convenience). Comments are not included, but they should be there!
Implement the following methods for the class Rectangle:

```python
// 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.
void rotate(bool clockwise);

// Flip horizontally (around the left edge) or vertically (around the bottom edge), depending on the value of the parameter.
void flip(bool horizontally);

// Check whether point p is inside the rectangle
bool isPointInside(const Point& p) const;
```
Re-implement the class Rectangle using an internal representation with two Points:

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

// Pre: n \geq 0
// Returns the Fibonacci number of order n.
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}

Fibonacci numbers: iterative version

// Pre: n \geq 0
// Returns the Fibonacci number of order n.
int fib(int n) {
    int f_i = 0;
    int f_i1 = 1;
    for (int i = 0; i < n; ++i) {
        int f = f_i + f_i1;
        f_i = f_i1;
        f_i1 = f;
    }
    return f_i;
}

Algorithm analysis

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

\[
C : D \rightarrow \mathbb{R}^+
\]

\[
x \mapsto C(x)
\]

Runtime \( \approx \log_2 n \) 2x2 matrix multiplications

Algorithm Analysis

© Dept. CS, UPC

2

Algorithm Analysis

© Dept. CS, UPC

3

Algorithm Analysis

© Dept. CS, UPC

4

Algorithm Analysis

© Dept. CS, UPC

5

Algorithm Analysis

© Dept. CS, UPC

6

Algorithm Analysis

© Dept. CS, UPC

7

Algorithm Analysis

© Dept. CS, UPC

8

Algorithm Analysis

© Dept. CS, UPC

9
Algorithm analysis: simplifications

- Analysis based on the size of the input: \( |x| = n \)

- Only the best/average/worst cases are analyzed:
  
  \[
  C_{\text{worst}}(n) = \max \{ C(x) : x \in D, |x| = n \} \\
  C_{\text{best}}(n) = \min \{ C(x) : x \in D, |x| = n \} \\
  C_{\text{avg}}(n) = \sum_{x \in D, |x| = n} p(x) \cdot C(x)
  \]

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

Properties:

\( \forall n \geq 0 : C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n) \)

\( \forall x \in D : C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|) \)

- 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.

Algorithm analysis

- Properties:
  
  \( \forall n \geq 0 : C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n) \)

  \( \forall x \in D : C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|) \)

- We want a notation that characterizes the cost of algorithms independently from the technology (CPU speed, programming language, efficiency of the compiler, etc.).

Asymptotic notation

\( k_1 f_1(n) \) \( g(n) \) \( k_2 f_2(n) \)

\( g(n) \in O(f_1(n)) \)

\( g(n) \in \Omega(f_2(n)) \)

Examples

<table>
<thead>
<tr>
<th>Big-O</th>
<th>Big-Ω</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 13n^3 - 4n + 8 )</td>
<td>( 13n^3 - 4n + 8 )</td>
</tr>
<tr>
<td>( 2n - 5 )</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>( n^3 )</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( 3^n )</td>
</tr>
<tr>
<td>( 3 \log_2 n )</td>
<td>( 3 \log_2 n )</td>
</tr>
<tr>
<td>( n \log_2 n )</td>
<td>( n \log_2 n )</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>( \Omega(n^2) )</td>
</tr>
</tbody>
</table>

Complexity ranking

<table>
<thead>
<tr>
<th>Function</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n! )</td>
<td>factorial</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>exponential</td>
</tr>
<tr>
<td>( n^d, d &gt; 3 )</td>
<td>polynomial</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>cubic</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>quadratic</td>
</tr>
<tr>
<td>( n \sqrt{n} )</td>
<td>quasi-linear</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>linear</td>
</tr>
<tr>
<td>( \sqrt{n} )</td>
<td>root - ( n )</td>
</tr>
<tr>
<td>( \log n )</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( 1 )</td>
<td>constant</td>
</tr>
</tbody>
</table>

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

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

\[ \{ \begin{array}{l} \text{if } L = 0 \text{ then } f \in O(g) \\ \text{if } 0 < L < \infty \text{ then } f \in \Theta(g) \\ \text{if } L = \infty \text{ then } f \in \Omega(g) \end{array} \]

Note: If both limits are \( \infty \) or 0, use L'Hôpital's rule:

\[ \lim_{x \to \infty} \frac{f(x)}{g(x)} = \lim_{x \to \infty} \frac{f'(x)}{g'(x)} \]
The robot and the door in an infinite wall

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

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
</tr>
<tr>
<td>$\log_2 n$</td>
</tr>
<tr>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>$n$</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
</tr>
<tr>
<td>$n^2$</td>
</tr>
<tr>
<td>$n^3$</td>
</tr>
<tr>
<td>$2^n$</td>
</tr>
<tr>
<td>$2^{2n}$</td>
</tr>
</tbody>
</table>

Execution time: example

How about “big data”?

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 $\Rightarrow O(d)$.
- If incorrect $\Rightarrow$ the algorithm does not terminate.

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

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

Asymptotic complexity (small values)

Asymptotic complexity (larger values)

Properties
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 \( d = 2^n \)):

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

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 \( O(1) \).
- Consecutive statements:
  - If \( S_1 \) is \( O(f) \) and \( S_2 \) is \( O(g) \), then \( S_1; S_2 \) is \( O(\max(f, g)) \).
- Conditional statements:
  - If \( S_1 \) is \( O(f) \), \( S_2 \) is \( O(g) \) and \( B \) is \( O(h) \), then \( \text{if} (B) S_1; \text{else} S_2; \) is \( O(\max(f + h, g + h)) \), or also \( O(\max(f, g, h)) \).

### 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: \( O(\log n) \)

Examples:
- Logarithmic time is usually related to divide-and-conquer algorithms
- Binary search
- Calculating \( x^2 \)
- Calculating the \( n \)-th Fibonacci number

- \( T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \cdots \)
- \( T(x^y) \leq 4 + 4 + \cdots + 4 \quad \Rightarrow \quad O(\log y) \)
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

Selection Sort

```c
void selection_sort(vector<elem>& v) {
    int last = v.size() - 1;                  // v.size() = n
    for (int i = 0; i < last; ++i) {          // 0..n-1
        int k = i;                            // try all possible subsequences
        for (int j = i + 1; j <= last; ++j) {  // i+1..n-1
            if (v[j] < v[k]) k = j;            // if (v[j] < v[k]) k = j;
        }
        swap(v[k], v[i]);                     // swap(v[k], v[i]);
    }
    return;
}
```

Insertion Sort

```c
void insertion_sort(vector<elem>& v) {
    for (int i = 0; i < v.size(); ++i) {       // n times
        elem x = v[i];                        // v[i];

        int thisSum = 0;                      // thisSum = 0;
        for (int k = i; k <= last; ++k) {      // k = i; // k <= last; ++k)
            if (thisSum > maxSum) maxSum = thisSum;  // if (thisSum > maxSum) maxSum = thisSum;
            thisSum += a[k];                   // thisSum += a[k];
        }
        return maxSum;                       // return maxSum;
    }
}
```

The Maximum Subsequence Sum Problem

```c
int maxSubSum(const vector<int>& a) {
    int maxSum = 0;                          // int maxSum = 0;
    for (int i = 0; i < a.size(); ++i) {     // try all possible subsequences
        int thisSum = 0;                     // int thisSum = 0;
        for (int j = i; j < a.size(); ++j) {  // for (int j = i; j < a.size(); ++j)
            thisSum += a[k];                 // for (int k = i; k <= j; ++k) thisSum += a[k];
            if (thisSum > maxSum) maxSum = thisSum;  // if (thisSum > maxSum) maxSum = thisSum;
        }
    }
    return maxSum;                          // return maxSum;
}
```

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

The Maximum Subsequence Sum Problem

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

1. Selection sort uses this invariant:
   - Selection sort
   - Insertion sort
   - The Maximum Subsequence Sum Problem
   - Convex Hull

Examples

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

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)

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
int maxSumRec(const vector<int>& a, int left, int right) {
  // base cases
  if (left == right) return maxLCenter = maxSizeRec(a, left, center);
  int maxLeft = maxSumRec(a, left, center);
  int maxRight = maxSumRec(a, center + 1, right);
  // recursive cases: left and right halves
  int thisSum = 0;
  for (int i = left; i < center; ++i) thisSum += a[i];
  for (int j = center + 1; j < right; ++j) thisSum += a[j];
  return maxSumRec(a, left, center); // reuse computation
}
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
int maxSubSum(const vector<int>& a) {
  int maxSum = 0;
  for (int i = 0; i < a.size(); ++i) {
    int thisSum = 0;
    for (int j = i; j < a.size(); ++j) {
      thisSum += a[j];
      if (thisSum > maxSum) maxSum = thisSum;
    }
    return maxSum;
  }
}
```

Max Subsequence Sum: Divide&Conquer

```cpp
int maxSubSum(const vector<int>& a) {
  int maxSum = 0;
  // try all possible subsequences
  for (int i = 0; i < a.size(); ++i) {
    int thisSum = 0;
    for (int j = i; j < a.size(); ++j) {
      thisSum += a[j]; // reuse computation
      if (thisSum > maxSum) maxSum = thisSum;
    }
    return maxSum;
  }
  return maxSubSum;
}
```

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = 2T(n/2) + O(n)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
T(1) = 1
T(n) = 2T(n/2) + O(n)
```

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = O(n^2)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = 2T(n/2) + n = 2(2T(n/4) + n/2) + n
= 4T(n/4) + n + n = 8T(n/8) + n + n + n = ... 
= 2^kT(n/2^k) + n + n + ... + n
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = 2^kT(1) + kn = n + n \log_2 n = O(n \log n)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
But, can we still do it faster?
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• 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 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

Max Subsequence Sum: Divide&Conquer

```cpp
• 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.
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
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)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
// A polygon (an ordered set of vertices)
using Polygon = vector<Point>;
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
Create a polygon from a set of points
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• 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.
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
The max sum can be in one of three places:
1st half
2nd half
Spanning both halves and crossing the middle
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
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)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
MaxSubSumSum of the vector = \sum_{i=0}^{n-1} a[i]
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
maxSubSum = \sum_{i=0}^{n-1} a[i];
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
a:
thisSum:
maxSum:
4 -3 5 -2 -1 2 6 -2
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
The max sum can be in one of three places:
1st half
2nd half
Spanning both halves and crossing the middle
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• Observations:
  – If |a[i]| is positive, it cannot be the start of the optimal subsequence.
  – Any negative subsequence cannot be the prefix of the optimal subsequence.
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• Let us consider the inner loop of the O(n^2) algorithm and assume that all prefixes of a[i..j-1] are positive and a[i..j] is negative:
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
– 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

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = 2^kT(1) + kn = n + n \log_2 n = O(n \log n)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
The max sum can be in one of three places:
1st half
2nd half
Spanning both halves and crossing the middle
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
The max sum can be in one of three places:
1st half
2nd half
Spanning both halves and crossing the middle
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• Observations:
  – If |a[i]| is positive, it cannot be the start of the optimal subsequence.
  – Any negative subsequence cannot be the prefix of the optimal subsequence.
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• Let us consider the inner loop of the O(n^2) algorithm and assume that all prefixes of a[i..j-1] are positive and a[i..j] is negative:
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
T(1) = 1
T(n) = 2T(n/2) + O(n)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
T(n) = 2^kT(1) + kn = n + n \log_2 n = O(n \log n)
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
• Any negative subsequence cannot be the prefix of the optimal subsequence.
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
The max sum can be in one of three places:
1st half
2nd half
Spanning both halves and crossing the middle
```

The Maximum Subsequence Sum Problem

Max Subsequence Sum: Divide&Conquer

```cpp
First half
Second half
```
Simple polygon

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

Select a point $z$ with the largest $x$ coordinate (and smallest $y$ in case of a tie in the $x$ coordinate). Assume $z = p_1$.

For each $p_i \in \{p_2, ..., p_n\}$, calculate the angle $\alpha_i$ between the lines $z - p_i$ and the $x$ axis.

Sort the points $\{p_2, ..., p_n\}$ according to their angles. In case of a tie, use distance to $z$.

**Output:** $P$ (the convex hull of $n$ points $> n$).

$\beta$ is at the left of $\cdot p$.

Assume that a partial path with $\beta$ is at the left of $\cdot p$.

Stop when $P$ is complete (back to point $p_0$).

Convex hull

**Algorithm Analysis**

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

**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).

**Polygon convexHull(const vector<Point>& points) {**

```cpp
int n = points.size();
Polygon hull;

// Pick the leftmost point
int left = 0;
for (int i = 1; i < n; i++)
    if (points[i].x < points[left].x) left = i;

int p = left;

do {
    hull.push_back(points[p]); // Add point to the convex hull
    int q = (p + 1) % n; // Pick a point different from p
    for (int i = 0; i < n; i++)
        if (left == points[p], points[q], points[i])) q = i;
    p = q; // Leftmost point for the convex hull
} while (p != left);  // While not closing polygon
return hull;
```

**Python**

```python
def convex_hull(points):
    n = len(points)
    hull = []

    # Pick the leftmost point
    left = 0
    for i in range(1, n):
        if points[i].x < points[left].x:
            left = i

    p = left

    while p != left:
        hull.append(points[p])
        q = (p + 1) % n
        for i in range(n):
            if points[i] == points[q] or points[i][0] == points[p][0]:
                q = i
        p = q

    return hull
```

**Source:** https://en.wikipedia.org/wiki/Graham_scan
Convex hull: Graham scan

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

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

// 0 = (q_1, q_2, \ldots) is a vector where the points // of the convex hull will be stored.
q_1 = p_1; q_2 = p_2; q_1 = p_3; m = 3;
while leftof(qm-1, qm, p_0): m = m - 1;
q_m = p_0;

EXERCISES

For loops: analyze the cost of each code

// Code 1
int s = 0;
for (int i = 0; i < n; ++i) ++s;

// Code 2
int s = 0;
for (int i = 0; i < n; i += 2) ++s;

// Code 3
int s = 0;
for (int i = 0; i < n; ++i) ++s;
for (int j = 0; j < n; ++j) ++s;

// Code 4
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) ++s;
}

// Code 5
int s = 0;
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < i; ++j) ++s;
}

O, \Omega \text{ or } \Theta?

The following statements refer to the \textit{insertion sort} algorithm and the X's hide an occurrence of \( O, \Omega \text{ or } \Theta \). 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 \( n \).

\begin{verbatim}
bool isPrime1(int n) {
  if (n <= 1) return false;
  for (int i = 2; i * i <= n; ++i) if (n % i == 0) return false;
  return true;
}

bool isPrime2(int n) {
  if (n <= 1) return false;
  for (int i = 2; i <= n; ++i) if (n % i == 0) return false;
  return true;
}

bool isPrime3(int n) {
  if (n <= 1) return false;
  for (int i = 2; i <= n; ++i) if (n % i == 0) return false;
  return true;
}

bool isPrime4(int n) {
  if (n <= 1) return false;
  for (int i = 2; i * i <= n; ++i) if (n % i == 0) return false;
  return true;
}

bool isPrime5(int n) {
  if (n <= 1) return false;
  for (int i = 2; i <= n; ++i) if (n % i == 0) return false;
  return true;
}

vector<bool> Primes(int n) {
  vector<bool> p(n + 1, true);
  p[0] = p[1] = false;
  for (int i = 2; i * i <= n; ++i) {
    if (p[i]) {
      for (int j = i * i; j <= n; j += i) p[j] = false;
    }
    return p;
}

You can use the following equality, where \( p \leq x \) refers to all primes \( p \leq x \):
\[
\sum_{p \leq x} 1 = \log \log x + O(1)
\]

The Sieve of Eratosthenes

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

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

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

\[
\sum_{i=0}^{2^l} 2^{n+1} - 1
\]
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 $f^{th}$ 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

Conventional product of polynomials

- 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 complex numbers

Polynomial multiplication: recursive step

- Pattern of recursive calls

Product of polynomials: Divide & Conquer

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

\[
P(x) = 2x^3 + x^2 - 4
\]

\[
Q(x) = x^2 - 2x + 3
\]

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

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

Complexity analysis:
- Multiplication of polynomials of degree \(n\): \(O(n^2)\)
- Addition of polynomials of degree \(n\): \(O(n)\)
Useful reminders

- Sum of geometric series with ratio $r$:
  \[ S = a + ar + ar^2 + ar^3 + \ldots + ar^{n-1} \]
  \[ S = a \left( \frac{1 - r^n}{1 - r} \right) = \frac{a(1 - r^n)}{1 - r} \]
- 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 $k = \log_b n$, runtime is $O(3^{\log_b n})$, which is equal to $O(n^{\log_b 3})$.
- 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 $O(n^{1.59})$.

Examples

- Typical pattern for Divide&Conquer algorithms:
  - Split the problem into $a$ subproblems of size $n/b$
  - Solve each subproblem recursively
  - Combine the answers in $O(nc)$ time
- Running time: $T(n) = a \cdot T(n/b) + O(nc)$
- Master theorem:
  \[ T(n) = \begin{cases} 
  O(nc) & \text{if } c > \log_b a \\
  O(n^c \log n) & \text{if } c = \log_b a \\
  O(n^{c\log_b a}) & \text{if } c < \log_b a 
  \end{cases} \]
  \[ a > b^c \]
  \[ a < b^c \]
  \[ a = b^c \]

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: $O(nc)$

Master theorem: proof

- For simplicity, assume $n$ is a power of $b$.
- The base case is reached after $\log_b n$ levels.
- The $k$th level of the tree has $a^k$ subproblems of size $n/b^k$.
- The total work done at level $k$ is:
  \[ a^k \cdot O \left( \frac{n}{b^k} \right) = O(nc) \cdot \left( \frac{a}{b^c} \right)^k \]
  \[ \text{As } k \text{ goes from } 0 \text{ (the root) to } \log_b n \text{ (the leaves), these numbers form a geometric series with ratio } a/b^c. \text{ We need to find the sum of such a series.} \]
  \[ T(n) = O(nc) \cdot \left( 1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \ldots + \frac{a^{\log_b n}}{b^{(\log_b n)c}} \right) \]
  \[ \text{log}_b n \text{ terms} \]
- Case $a/b^c < 1$. Decreasing series. The sum is dominated by the first term ($k = 0$): $O(nc)$.
- Case $a/b^c > 1$. Increasing series. The sum is dominated by the last term ($k = \log_b n$):
  \[ n^c \left( \frac{a}{b^c} \right)^{\log_b n} = n^c \cdot \frac{a^{\log_b n}}{(b^{\log_b n})^c} = a^{\log_b n} = n^{\log_b a} \]
- Case $a/b^c = 1$. We have $O(\log n)$ terms all equal to $O(nc)$.
Master theorem: examples

Running time: \[ T(n) = a \cdot T \left( \frac{n}{b} \right) + O(n^c) \]

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>a</th>
<th>c</th>
<th>Runtime equation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power ((x^c))</td>
<td>1</td>
<td>0</td>
<td>(T(y) = T(y/2) + O(1))</td>
<td>(O(\log y))</td>
</tr>
<tr>
<td>Binary search</td>
<td>1</td>
<td>0</td>
<td>(T(n) = T(n/2) + O(1))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Merge sort</td>
<td>2</td>
<td>1</td>
<td>(T(n) = 2 \cdot T(n/2) + O(n))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>Polynomial product</td>
<td>4</td>
<td>1</td>
<td>(T(n) = 4 \cdot T(n/2) + O(n))</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>Polynomial product (Gauss)</td>
<td>3</td>
<td>1</td>
<td>(T(n) = 3 \cdot T(n/2) + O(n))</td>
<td>(O(n \log n \log n))</td>
</tr>
</tbody>
</table>

\(b = 2\) for all the examples
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: algorithm

```java
function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
  mid = Partition(A, left, right);
  Qsort(A, left, mid-1);
  Qsort(A, mid+1, right);

function Partition(A, left, right)
// A[left..right]: segment to be sorted
// Returns the middle of the partition with
//   A[middle] = pivot
//   A[left..middle-1] ≤ pivot
//   A[middle+1..right] > pivot
x = A[left];  // the pivot
i = left;  j = right;
while i < j do
  while i ≤ right and A[i] ≤ x do i = i+1;
  while j ≥ left and A[j] > x do j = j-1;
  if i < j then swap(A[i], A[j]);
  swap(A[left], A[j]);
return j;
```

Quick sort: partition

```
function Partition(A, left, right)
// A[left..right]: segment to be sorted
// Returns the middle of the partition with
//   A[middle] = pivot
//   A[left..middle-1] ≤ pivot
//   A[middle+1..right] > pivot

x = A[left];  // the pivot
i = left;  j = right;
while i < j do
  while i ≤ right and A[i] ≤ x do i = i+1;
  while j ≥ left and A[j] > x do j = j-1;
  if i < j then swap(A[i], A[j]);
  swap(A[left], A[j]);
return j;
```

Quick sort: example

```
6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14
```

Quick sort partition: example

```
6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14
```

Quick sort: Hungarian, folk dance

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

Quick sort: partition

```
function Partition(A, left, right)
// A[left..right]: segment to be sorted
// Returns the middle of the partition with
//   A[middle] = pivot
//   A[left..middle-1] ≤ pivot
//   A[middle+1..right] > pivot

x = A[left];  // the pivot
i = left;  j = right;
while i < j do
  while i ≤ right and A[i] ≤ x do i = i+1;
  while j ≥ left and A[j] > x do j = j-1;
  if i < j then swap(A[i], A[j]);
  swap(A[left], A[j]);
return j;
```

Quick sort: algorithm

```
function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
  mid = Partition(A, left, right);
  Qsort(A, left, mid-1);
  Qsort(A, mid+1, right);
```

Quick sort: example

```
6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14
```
Quick sort: Hoare’s partition

```c
function HoarePartition(A, left, right)
    // 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.
    x = A[left]; // the pivot
    i = left-1; // the pivot
    j = right+1;
    while true
do i = i+1; while A[i] < x;
    do j = j+1; while A[j] > x;
    if i ≥ j then return j;
    swap(A[i], A[j]);
```

Quick sort partition: example

The first swap creates two sentinels. After that, the algorithm flies.

<table>
<thead>
<tr>
<th>pivot</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pivot</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Quick sort with Hoare’s partition

```c
function Qsort(A, left, right)
    // A[left..right]: segment to be sorted.
    if left < right then
        mid = HoarePartition(A, left, right);
        Qsort(A, left, mid);
        Qsort(A, mid+1, right);
    else
        insert(A, left, right);
    end
```

Quick sort: hybrid approach

```c
function Qsort(A, left, right)
    // A[left..right]: segment to be sorted.
    // K is a break-even size in which insertion sort is
    // more efficient than quick sort.
    if right - left ≥ K then
        mid = HoarePartition(A, left, right);
        Qsort(A, left, mid);
        Qsort(A, mid+1, right);
    else
        insert(A, left, right);
    end
```

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 \( \Rightarrow O(n^2) \)

- 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

- Let us assume that $x_i$ is the $i$-th smallest element in the vector.

- The runtime if $x_i$ is selected as pivot is:

\[ T(n) = n - 1 + T(i - 1) + T(n - i) \]

Quick sort: complexity analysis summary

- Runtime of quicksort:
  - $T(n) = O(n^2)$
  - $T(n) = \Omega(n \log n)$

- $T_{avg}(n) = O(n \log n)$

- 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?

The selection problem

- $H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$ 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]

\[ T(n) \leq 2(n + 1)(\ln n + \gamma - 1.5) + O(1) \in O(n \log n) \]
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 \( \\Theta(n^2) \)
- We want an \( \Theta(n \log n) \) 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?

• **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 qsort).

### The Closest-Points problem: algorithm

- 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.

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 \( \delta \).

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

```
function Qsort(A, left, right)
    // A[left..right]: segment to be sorted
    if left < right then
        mid = HoarePartition(A, left, right);
        Qsort(A, left, mid);
        Qsort(A, mid+1, right);

function Qselect(A, left, right, k)
    if left == right then return A[left];
    mid = HoarePartition(A, left, right);
    if k <= mid then return Qselect(A, left, mid, k);
    else return Qselect(A, mid+1, right, k);
```

```
// Returns the element at location k assuming // A[left..right] would be sorted in ascending order. // Pre: left <= k <= right. // Post: The elements of A have changed their locations.
```

For \( (i=0; i < \text{NumPointsInStrip}; ++i) \)

```
    for (j=i+1; j < \text{NumPointsInStrip}; ++j)
        if \((p_i \text{ and } p_j \text{'s } y \text{-coordinate differ by more than } \delta) \text{ break}; // Go to next } p_i\n```

```
if (dist\((p_i, p_j)\) < \delta) \\delta = dist\((p_i, p_j)\);```

But, how many pairs \((p_i, p_j)\) do we need to consider?
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 $\delta$: $O(n)$
  - Sort remaining points using $y$-coordinates: $O(n \log n)$
  - Scan the remaining points in $y$ order: $O(n)$

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

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

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.


EXERCICES

- **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

  Thus, $T(n) = T(n-1) + O(n)$ (as $c = 1$)

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

  We can compute bounds:

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

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

Subtract and Conquer

- Sometimes we may find recurrences with the following structure:

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

  - Examples:
    - **Hanoi**($n$) = $2 \cdot Hanoi(n-1) + O(1)$
    - **Sort**($n$) = $Sort(n-1) + O(n)$

  - **Muster theorem**:
    - $T(n) =$
      - $O(n^c)$ if $a < 1$ (never occurs)
      - $O(n^{c+1})$ if $a = 1$
      - $O(n^ca^{n/b})$ if $a > 1$

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 then 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?
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 $k = i + \lceil n/3 \rceil$ and $l = j - \lfloor n/3 \rfloor$. The algorithm recursively sorts $T[i..l]$, then it sorts $T[k..j]$, and finally sorts $T[i..l]$.

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

**Logarithmic identities**

$$y = \sum_{k=1}^{n} \frac{1}{k} \in \Theta(\log n)$$

**Crazy sorting**

The majority element

A majority element in a vector, $A$, of size $n$ is an element that appears more than $n/2$ times (thus, there is at most one). For example, the vector $[3,3,4,2,4,4,4]$ has a majority element (4), whereas the vector $[3,3,4,2,4,4,2,2]$ does not. If there is no majority element, your program should indicate this. Here is a sketch of an algorithm to solve the problem:

First, a candidate majority element is found (this is the hardest part). This candidate is the only element that could possibly be the majority element. The second step determines if this candidate is actually the majority. This is just a sequential search through the vector. To find a candidate in the vector, $A$, form a second vector, $B$. Then compare $A_i$ and $A_j$. If they are equal, add one of these to $B$; otherwise do nothing. Then compare $A_k$ and $A_l$. Again if they are equal, add one of these to $B$; otherwise do nothing. Continue in this fashion until the entire vector is read. The recursively find a candidate for $B$; this is the candidate for $A$ (why?).

- How does the recursion terminate?
- What is the running time of the algorithm?
- How can we avoid using an extra array $B$?
- Prove the correctness of the algorithm (hint: prove it for $n$ even)
- How is the case where $n$ is odd handled?

**Full-history recurrence relation**

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

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

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

$$T(n+1) = \frac{n+2}{n+1} T(n) + \frac{2n}{n+1} \leq n+2 \log(n+1) + 2$$

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

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

$$T(n) \leq 2(\log(n+3) - 1.5) \in \Theta(\log n)$$

**Breaking into pieces**

Let us assume that $T$ is $O(1)$. What is the asymptotic cost of $A$ and $B$ as a function of $n$? (if $n$ is the size of the vector).

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

Memory management

Jordi Cortadella and Jordi Petit
Department of Computer Science

Pointers

• Address of a variable (reference operator &):
  
  ```
  int i;
  int* pi = &i;
  // &i means: “the address of i”
  ```

• Access to the variable (dereference operator *)

```
int j = *pi;
// j gets the value of the variable pointed by pi
```  

• Null pointer (points to nowhere; useful for initialization)

  ```
  int* pi = nullptr;
  ```

Dynamic Object Creation/Destruction

• The `new` operator returns a pointer to a newly created object:

```
myClass* c = new myClass();
```  

• The `delete` operator destroys an object through a pointer (deallocates the memory space associated to the object):

```
delete c; // c must be a pointer
```  

Access to members through a pointer

The members of a class can be accessed through a pointer to the class via the `->` operator:

```
vector<int>* vp = new vector<int>(100);
... vp->push_back(5);
... int n = vp->size();
```  

Allocating/deallocating arrays

`new` and `delete` can also create/destroy arrays of objects

```
c = new myClass[10];
```  

References

• A reference defines a new name for an existing value (a synonym).

• References are not pointers, although they are usually implemented as pointers (memory references).

• Typical uses:
  – Avoiding copies (e.g., parameter passing)
  – Aliasing long names
  – Range for loops

References: examples

```
auto r = x[getIndex(a, b)].val;
...
r définieValue(n); // avoids a long name for r
...
```

```// avoids a copy of a large data structure
bigVector & V = myMatrix.getRow(i);
...```  

```// An alternative for the following loop:
for (int i = 0; i < a.size(); ++i) ++a[i];
```  

```for (auto x : arr) ++x; // does not work (why?)
```  

```for (auto & x : arr) ++x; // it works!
```
## Pointers vs. references

- A **pointer** holds the memory address of a variable. A **reference** is an alias (another name) for an existing variable.
- In practice, references are implemented as pointers.
- A pointer can be re-assigned any number of times, whereas a reference can only be assigned at initialization.
- Pointers can be NULL. References always refer to an object.
- You can have pointers to pointers. You cannot have references to references.
- You can use pointer arithmetic (e.g., &object+3). Reference arithmetic does not exist.

### The Vector class

**The Vector class** (an approximation to the STL vector class)

- The natural replacement of C-style arrays.
- Main advantages:
  - It can dynamically grow and shrink.
  - It is a **template** class (can handle any type T)
  - No need to take care of the allocated memory.
  - Data is allocated in a contiguous memory area.
- We will implement a **Vector** class, a simplified version of STL's vector.
- Based on Weiss' book (4th edition), see Chapter 3.4.

```cpp
#include <vector>

int main() {
    std::vector<int> v;
    v.push_back(5);
    v.push_back(10);
    v.push_back(15);
    return 0;
}
```

### Memory management

- Data structures usually have a descriptor (fixed size) and a storage area (variable size).
- Programmers do not have to take care of memory allocation, but it is convenient to know the basics of memory management.
- Beware of memory leaks, fragmentation, ...

#### Example of declarations:

- `Vector<int> V;`
- `Vector<polygon> Vp;`
- `Vector<Vector<double>> M;`

#### Example of usage:

```cpp
int main() {
    std::vector<int> v;
    // ... allocate memory
    v.push_back(5);
    v.push_back(10);
    v.push_back(15);
    return 0;
}
```
The Vector class

Constructors, copies, assignments

The Vector class

```
// Reserve space (to increase capacity)
void reserve(int newCapacity) {
    if (newCapacity <= theCapacity) return;

    // Allocate the new memory block
    Object* newArray = new Object[newCapacity];

    // Copy the old memory block
    for (int k = 0; k < theSize; ++k)
        newArray[k] = objects[k];

    theCapacity = newCapacity;
    // Swap pointers and free old memory block
    std::swap(objects, newArray);
    delete [] newArray;
}
```

```
void foo(Myclass x);
foo(c); // Constructor is used
```

```
// Default constructor with initial size
Vector(int initSize = 0) : theSize(initSize),
    theCapacity(initSize + MAX_SPARE_CAPACITY) {
    objects = new Object[theCapacity];
}
```

```
// Copy constructor
Vector(const Vector& rhs) :
    theSize(rhs.theSize),
    theCapacity(rhs.theCapacity),
    objects(rhs.object) {
    for (int k = 0; k < theSize; ++k) objects[k] = rhs.object[k];
}
```

```
// Assignment operator
Vector& operator=(const Vector& rhs) {
    if (theSize == rhs.theSize) {
        theSize = rhs.theSize;
        theCapacity = rhs.theCapacity;
        for (int k = 0; k < theSize; ++k)
            objects[k] = rhs.objects[k];
    } else {
        return *this;
    }
    return *this;
}
```

```
// Destructor
Vector::~Vector() {
    // Destructor
}
```

```
Vector& operator[](int i) {
    return objects[i];
}
```

```
void main() {
    int i;
    vector<double> Y;
    …
}
```

```
int g(vector<int>& X) {
    int i;
    vector<double> Y;
    …
}
```

```
int f() {
    int z;
    vector<int> …
}
```

```
int main() {
    double r;
    …
    return 0;
}
```

```
// (points at free space)
delete B; // Error
```

```
// Do not use pointers, unless you are very desperate.
// If you have to use pointers, hide their usage inside a class
// and define consistent constructors/destructors.
// Use valgrind (valgrind.org) to detect memory leaks.
// Remember: no pointers ➔ no memory leaks.
```

Dangling references and memory leaks

```
myClass* A = new myClass;
myClass* B = new myClass;
// We have allocated space for two objects
A = B;
// A and B point at the same object!
// Possible memory leak (unreferenced memory space)
delete A;
// Now B is a dangling reference
// (points at free space)
delete B; // Error
```

```
for (i = 0; i < 1000000; ++i) {
    myClass* A = new MyClass; // 1Kbyte
    …
    doSomething(A, i);
    …
    // forgets to delete A
}
// This loop produces a 1-Gbyte memory leak!
```

```
Recommendations:
• Do not use pointers, unless you are very desperate.
• If you have to use pointers, hide their usage inside a class
  and define consistent constructors/destructors.
• Use valgrind (valgrind.org) to detect memory leaks.
• Remember: no pointers ➔ no memory leaks.
```

Pointers and memory leaks

```
vector<Myclass> a;
…
// do some push_back’s to a
…
const myClass& x = a[0]; // ref to a[0]
// x contains a memory address pointing at a[0]
a.push_back(something);
// the address of a[0] might have changed!
// x might be pointing at garbage data
```

```
Recommendation: don’t trust on pointers/references to dynamic data.
```

```
Possible solution: use indices/keys to access data in containers.
```

Pointers and references to dynamic data

Memory layout of a program

```
+-----------------+-----------------+-----------------+
| Static          | Local           | Dynamic         |
| Static data     | Local variables | Dynamic data    |
| (binary file)   | of a function   | Since created   |
| Static          | Lifetime        | (until destroyed |
| (fragmented)    | Lifetime of the | (delete)        |
| Stack           | program         | )               |
| Local variables | Lifetime of the  |                 |
| of a function   | function        |                 |
| Stack           | Lifetime        |                 |
| Dynamic data    | Lifetime        |                 |
| Heap            | of the program  |                 |
| Dynamic data    | Lifetime        |                 |
| Heap            | Since created   |                 |
| Dynamic data    | until destroyed |
| Heap            | (delete)        | )               |
|                    |                 |                 |
```

Dangling references and memory leaks

```
myClass* A = new myClass;
myClass* B = new myClass;
// We have allocated space for two objects
A = B;
// A and B point at the same object!
// Possible memory leak (unreferenced memory space)
delete A;
// Now B is a dangling reference
// (points at free space)
delete B; // Error
```

Memory management models

```
• Programmer-controlled management:
  – The programmer decides when to allocate (new) and deallocate (delete) blocks of memory.
  – Example: C++.
  – Pros: efficiency, memory management can be optimized.
  – Cons: error-prone (dangling references and memory leaks)

• Automatic management:
  – The program periodically launches a garbage collector that frees all non-referenced blocks of memory.
  – Examples: Java, Python, R.
  – Pros: the programmer does not need to worry about memory management.
  – Cons: cannot optimize memory management, less control over runtime.

Memory layout of a program

```
+-----------------+-----------------+-----------------+
| Static          | Local           | Dynamic         |
| Static data     | Local variables | Dynamic data    |
| (binary file)   | of a function   | Since created   |
| Static          | Lifetime        | (until destroyed |
| (fragmented)    | Lifetime of the  | (delete)        |
| Stack           | program         | )               |
| Local variables | Lifetime        |                 |
| of a function   | Lifetime        |                 |
| Stack           | Lifetime        |                 |
| Dynamic data    | Lifetime        |                 |
| Heap            | Lifetime        |                 |
| Dynamic data    | Lifetime        |                 |
| Heap            | Lifetime        |                 |
| Dynamic data    | Since created   |                 |
| Heap            | until destroyed |
|                    | (delete)        | )               |
```

Pointers and memory leaks

```
for (i = 0; i < 1000000; ++i) {
    myClass* A = new MyClass; // 1Kbyte
    …
    doSomething(A, i);
    …
    // forgets to delete A
}
// This loop produces a 1-Gbyte memory leak!
```

```
Recommendations:
• Do not use pointers, unless you are very desperate.
• If you have to use pointers, hide their usage inside a class
  and define consistent constructors/destructors.
• Use valgrind (valgrind.org) to detect memory leaks.
• Remember: no pointers ➔ no memory leaks.
```

Pointers and references to dynamic data

```
vector<Myclass> a;
…
// do some push_back’s to a
…
const myClass& x = a[0]; // ref to a[0]
// x contains a memory address pointing at a[0]
a.push_back(something);
// the address of a[0] might have changed!
// x might be pointing at garbage data
```

```
Recommendation: don’t trust on pointers/references to dynamic data.
```

```
Possible solution: use indices/keys to access data in containers.
```
Some aspects that must be known to interface with C libraries:

- Developed by Dennis Ritchie at Bell Labs (1972) and used to re-implement Unix.
- It was designed to be easily mappable to machine instructions and provide low-level memory access.
- Today, it is still necessary to use some C libraries, designed by skilled experts, that have not been rewritten in other languages (the same happens with some FORTRAN libraries).
- Some aspects that must be known to interface with C libraries:
  - No references (only pointers).
  - No object-oriented support (no STL, no vectors, no maps, etc).
  - Vectors must be implemented as arrays.

Languages are designed to make memory management transparent (no STL, no vectors, no maps, etc).

Pointers imply the use of the heap and all the problems associated with it (memory management, memory leaks, fragmentation). Pointers imply the use of the heap and all the problems associated with it (memory management, memory leaks, fragmentation).

Memory management © Dept. CS, UPC 31

About C (the predecessor of C++)

- Developed by Dennis Ritchie at Bell Labs (1972) and used to re-implement Unix.
- It was designed to be easily mappable to machine instructions and provide low-level memory access.
- Today, it is still necessary to use some C libraries, designed by skilled experts, that have not been rewritten in other languages (the same happens with some FORTRAN libraries).
- Some aspects that must be known to interface with C libraries:
  - No references (only pointers).
  - No object-oriented support (no STL, no vectors, no maps, etc).
  - Vectors must be implemented as arrays.

Languages are designed to make memory management transparent (no STL, no vectors, no maps, etc).

Pointers imply the use of the heap and all the problems associated with it (memory management, memory leaks, fragmentation). Pointers imply the use of the heap and all the problems associated with it (memory management, memory leaks, fragmentation).

Memory management © Dept. CS, UPC 31

C: parameters by reference

- Memory is a valuable resource in computing devices. It must be used efficiently.
- Languages are designed to make memory management transparent to the user, but a lot of inefficiencies may arise unintentionally (e.g., copy by value).
- Pointers imply the use of the heap and all the problems associated with memory management (memory leaks, fragmentation).
- Recommendation: do not use pointers unless you have no other choice. Not using pointers will save a lot of debugging time.
- In case of using pointers, try to hide the pointer manipulation and memory management (new/delete) inside the class in such a way that the user of the class does not need to “see” the pointers.

Memory management © Dept. CS, UPC 31

Summary

- Memory is a valuable resource in computing devices. It must be used efficiently.
- Languages are designed to make memory management transparent to the user, but a lot of inefficiencies may arise unintentionally (e.g., copy by value).
- Pointers imply the use of the heap and all the problems associated with memory management (memory leaks, fragmentation).
- Recommendation: do not use pointers unless you have no other choice. Not using pointers will save a lot of debugging time.
- In case of using pointers, try to hide the pointer manipulation and memory management (new/delete) inside the class in such a way that the user of the class does not need to “see” the pointers.

Memory management © Dept. CS, UPC 31

Assertions vs. error handling

- Assertions:
  - Runtime checks of properties (e.g., invariants, pre-/post-conditions).
  - Useful to detect internal errors.
  - They should always hold in a bug-free program.
  - They should give meaningful error messages to the programmers.

- Error handling:
  - Detect improper use of the program (e.g., incorrect input data).
  - The program should not halt unexpectedly.
  - They should give meaningful error messages to the users.

Memory management © Dept. CS, UPC 31

C: arrays

```c
int a[100]; // an array of 100 int's
double m[30][40]; // a matrix of size 30x40
char * c; // an array of unknown size
```

- Memory allocation for n integers
  ```
c = malloc(n* sizeof(int));
```
- Memory allocation for m chars
  ```
s = malloc(m* sizeof(char));
```
- Anything that is allocated must be freed
  ```
free(c);
free(s);
```

Memory management © Dept. CS, UPC 31

C: arrays and C++ vectors

```c
double sum(double v[], int n) {
    double s = 0;
    for (int i = 0; i < n; ++i) s += v[i];
    return s;
}
```

- Any C++ vector is an object that contains a private array. This array can be accessed using the ‘data’ method (C++11).
- Memory management © Dept. CS, UPC 31

C arrays

```c
int f(int a, int* b) {
    int x, y, z;
    z = f(x, &y);
    return a * a;
}
```

- A is received by value and b by reference
  ```
  // using a pointer
  int main() {
    int x, y, z;
    // pointers used to pass by reference
    z = f(x, &y);
  }
```

Memory management © Dept. CS, UPC 31
Consider two versions of the program below, each one using a different definition of the class Point. Comment on the behavior of the program at compile time and runtime.

```
class Point {
  int x, y;
  public:
  Point(const Point &p) {
    x = p.x; y = p.y;
  }
  int getX() { return x; }
  int getY() { return y; }
};
```

```
class Point {
  int x, y;
  public:
  Point(int i=0, int j=0) {
    x = i; y = j;
  }
  int getX() { return x; }
  int getY() { return y; }
};
```

```
int main() {
  Point p1(10);
  Point p2 = p1;
  cout << "x = " << p2.getX() << endl;
  cout << "y = " << p2.getY() << endl;
}
```

Consider two versions of the program below, each one using a different definition of the class Point. Comment on the behavior of the program at compile time and runtime.

```
int main() {
  Point p2 = p1;
  cout << "x = " << p2.getX() << endl;
  cout << "y = " << p2.getY() << endl;
}
```

Consider the following definition of a list of students organized as shown in the picture.

```
struct Student {
  string name;
  vector<double> marks;
  Student* next;
};
```

```
string BestStudent(Student* L);
```

Consider the following definition of a list of students organized as shown in the picture.

You can assume that the vector of marks is never empty.

The last student in the list points at "null" (nullptr).

Design the function BestStudent with the following specification:

BestStudent returns the name of the student with the best average mark. In case no student has an average mark greater than or equal to 5, the function must return the string "Bad Teacher".
Containers: Stack

Jordi Cortadella and Jordi Petit
Department of Computer Science

Containers: Stacks © Dept. CS, UPC

The Stack ADT

- A stack is a list of objects in which insertions and deletions can only be performed at the top of the list.
- Also known as LIFO (Last In, First Out)

**Balancing symbols**: check for syntax errors when expressions have opening/closing symbols, e.g., [ ] {}{

- **Algorithm** (linear): read all chars until end of file. For each char, do the following:
  - If the char is opening, push it onto the stack.
  - If the char is closing and stack is empty, 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 pushed onto the stack.
  - This is an infix expression. What’s his value? 42 or 144?

```
8 * 3 + 10 + 2 * 4
```

Evaluation of postfix expressions: example

```
6 5 2 3 + 8 * + 3 + *
```

### Evaluation of postfix expressions: example

<table>
<thead>
<tr>
<th>8</th>
<th>6</th>
<th>5</th>
<th>2</th>
<th>3</th>
<th>48</th>
<th>45</th>
<th>3</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push(6)</td>
<td>Push(5)</td>
<td>Push(3)</td>
<td>Push(2)</td>
<td>Push(8)</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Output: 288

### From infix to postfix

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

```
abc + de + fg*
```

#### 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 (+, -, *), 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.

---

Jordi Cortadella and Jordi Petit
Department of Computer Science

Containers: Stacks © Dept. CS, UPC

### From infix to postfix

```
<table>
<thead>
<tr>
<th>Priority</th>
<th>a + b * c + ( d * e + f ) * g</th>
</tr>
</thead>
<tbody>
<tr>
<td>/ \</td>
<td>Output</td>
</tr>
<tr>
<td>+ \</td>
<td>a</td>
</tr>
<tr>
<td>(</td>
<td>a</td>
</tr>
<tr>
<td>/ \</td>
<td>b</td>
</tr>
<tr>
<td>+ \</td>
<td>a</td>
</tr>
<tr>
<td>\ \</td>
<td>b</td>
</tr>
</tbody>
</table>
```

---

Jordi Cortadella and Jordi Petit
Department of Computer Science

Containers: Stacks © Dept. CS, UPC
From infix to postfix

<table>
<thead>
<tr>
<th>Priority</th>
<th>( a + b \cdot c + (d \cdot e + f) \cdot g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>( a )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( b )</td>
</tr>
<tr>
<td>( + )</td>
<td>( c )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( d )</td>
</tr>
<tr>
<td>( + )</td>
<td>( e )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( f )</td>
</tr>
<tr>
<td>( + )</td>
<td>( g )</td>
</tr>
</tbody>
</table>

From infix to postfix

<table>
<thead>
<tr>
<th>Priority</th>
<th>( a + b \cdot c + (d \cdot e + f) \cdot g )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>( a )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( b )</td>
</tr>
<tr>
<td>( + )</td>
<td>( c )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( d )</td>
</tr>
<tr>
<td>( + )</td>
<td>( e )</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>( f )</td>
</tr>
<tr>
<td>( + )</td>
<td>( g )</td>
</tr>
</tbody>
</table>

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

Complexity: \( O(n) \)

EXERCISES

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) 4321098765
b) 4687532901
c) 2567489310
d) 4321056789

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

Middle element of a stack

Design the class \texttt{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 container of the STL 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 \( \lceil (n - 1)/2 \rceil \).
Containers: Queue and List

Jordi Cortadella and Jordi Petit
Department of Computer Science

Queue

• A container in which insertion is done at one end and deletion is done at the other end (the head).

• Also called FIFO (First-In, First-Out)

Queue usage

```
Queue<int> Q; // Constructor
Q.push(5);    // Inserting few elements
Q.push(8);
Q.push(6);

int n = Q.size(); // n = 3
while (not Q.empty()) {  // Get the first element
    int elem = Q.front();
    cout << elem << endl;
    Q.pop();  // Delete the element
}
```

The class Queue

```
template<typename T>
class Queue {
    // Constructor
    Queue();
    ~Queue();

    // Copy constructor
    Queue(const T& Q);

    // Assignment operator
    Queue& operator=(const Queue& Q);

    // Insert a new element at the tail of the queue. */
    void push(const T& x);

    // Returns the number of elements. */
    int size() const {
        return n;
    }

    // Checks whether the queue is empty. */
    bool empty() const {
        return size() == 0;
    }

    // Insert a new element at the end of the queue. */
    void push(const T& x) {
        // Node* p = new Node (x, nullptr);
        Node* p = new Node (x, nullptr);
        if (n++ == 0) first = last = p;
        else last->next = p;
        last = p;
        n++;
    }

    // Removes the first element. */
    void pop() {
        assert(not empty());
        Node* old = first;
        first = first->next;
        delete old;
        if (--n == 0) last = nullptr;
    }

    // Returns the first element. */
    // Pre: the queue is not empty. */
    T front() const {
        assert(not empty());
        return first->elem;
    }

    // Pointer to the first element */
    Node* *first;  // Pointer to the last element
    Node* last;
    int n;  // Number of elements
};
```

Implementation with linked lists

```
template<typename T>
class Queue {
    Node* first;
    Node* last;
    int n;

    // Constructor
    Queue();
    ~Queue();

    // Copy constructor
    Queue(const T& Q);

    // Assignment operator
    Queue& operator=(const Queue& Q);

    // Insert a new element at the end of the queue. */
    void push(const T& x) {
        // Node* p = new Node (x, nullptr);
        Node* p = new Node (x, nullptr);
        if (n++ == 0) first = last = p;
        else last->next = p;
        last = p;
        n++;
    }

    // Removes the first element. */
    void pop() {
        assert(not empty());
        Node* old = first;
        first = first->next;
        delete old;
        if (--n == 0) last = nullptr;
    }

    // Returns the first element. */
    // Pre: the queue is not empty. */
    T front() const {
        assert(not empty());
        return first->elem;
    }

    // Pointer to the first element */
    Node* *first;  // Pointer to the last element
    Node* last;
    int n;  // Number of elements
};
```

Queue: some methods

```
/** Returns the number of elements. */
int size() const {
    return n;
}

/** Checks whether the queue is empty. */
bool empty() const {
    return size() == 0;
}

/** Inserts a new element at the end of the queue. */
void push(const T& x) {
    // Node* p = new Node (x, nullptr);
    Node* p = new Node (x, nullptr);
    if (n++ == 0) first = last = p;
    else last->next = p;
    last = p;
    n++;
}

/** Removes the first element. */
void pop() {
    assert(not empty());
    Node* old = first;
    first = first->next;
    delete old;
    if (--n == 0) last = nullptr;
}

/** Returns the first element. */
// Pre: the queue is not empty. */
T front() const {
    assert(not empty());
    return first->elem;
}

/** Copy constructor. */
// Queue(const Queue& Q) { 
    copy(Q);
}

/** Assignment operator. */
// Queue& operator=(const Queue& Q) { 
    if (Q.empty()) {
        free();
        return *this;
    }

    /** Destructor. */
    ~Queue() {
        free();
    }

    private:
        /** Frees the linked list of nodes in the queue. */
        void free();

        // Node* p = new Node (x, nullptr);
        Node* p = new Node (x, nullptr);
    }  
}```
A queue can also be implemented with an array (vector) of elements.

It is a more efficient representation if the maximum number of elements in the queue is known in advance.

Queue: Complexity
- All operations in queues can run in constant time, except for:
  - Copy: linear in the size of the list.
  - Delete: linear in the size of the list.
- Queues do not allow to access/insert/delete elements in the middle of the queue.

List
- List: a container with sequential access.
- It allows to insert/erase elements in the middle of the list in constant time.
- A list can be considered as a sequence of elements with one or several cursors (iterators) pointing at internal elements.
- For simplicity, we will only consider lists with one iterator.
- Check the STL list: it can be visited by any number of iterators.
### Higher-order functions

- A higher-order function is a function that can receive other functions as parameters or return a function as a result.

- Most languages support higher-order functions (C++, python, R, Haskell, Java, JavaScript, ...).

- The have different applications:
  - `sort` in STL is a higher-order function (the compare function is a parameter).
  - Functions to visit the elements of containers (lists, trees, etc.) can be passed as parameters.
  - Mathematics: functions for composition and integration receive a function as parameter. 
    - etc...

```cpp
template <typename T>
class List {
    // Doubly linked node of the list.
    struct Node {
        Node* prev;  // Pointer to the previous node.
        T elem;      // The element of the list.
        Node* next;  // Pointer to the next element.
    };

    Node* sentinel;  // Sentinel of the list.
    Node* cursor;    // Node after the cursor.
    int n;           // Number of elements (without sentinel).

    /** Constructor of an empty list. */
    List(): sentinel(new Node), cursor(sentinel), n(0) {
        sentinel->next = sentinel->prev = sentinel;
    }

    /** Destructor. */
    ~List() {
        free();
    }

    /** Copy constructor. */
    List(const List& l) {
        copy(l);
    }

    /** Assignment operator. */
    List& operator= (const List& l) {
        if (this == &l) {
            free();
            copy(l);
            return *this;
        }
        return *this;
    }

    /** Returns the number of elements in the list. */
    int size() const {
        return n;
    }

    /** Checks whether the list is empty. */
    bool empty() const {
        return size() == 0;
    }
}
```

### Higher-order functions: example

```cpp
template <typename T>
class List {
    /** Transforms every element of the list using f. 
    It returns a reference to the list. */
    List<T>& transform(T f(T&));

    /** Returns a list with the elements for which f is true */
    List<T> filter(bool f(const T&));

    /** Applies f sequentially to the list and returns a 
    single value. For the list \[x_1, x_2, \ldots, x_n\] it returns 
    f(\ldots f(f(\text{init}, x_1), x_2), \ldots, x_n). 
    If the list is empty, it returns init. */
    T reduce(T f(const T&, const T&), T init) const;
}
```

```cpp
/** The following code computes: 
\[
\sum_{x \in \mathbb{Z}} x^2 \quad \text{if isPrime(x)}.
\]
int n = L.filter(isPrime).transform(square).reduce(add, 0);
```
Higher-order functions: example

```cpp
List<T>& transform(void f(T&)) {
    Node* p = sentinel->next;
    while (p != sentinel) { // Visit all elements and apply f to each one
        f(p->elem);
        p = p->next;
    }
    return *this;
}
```

```cpp
List<T> filter(bool f(const T&)) const {
    List<T> L;
    Node* p = sentinel->next;
    while (p != sentinel) {
        // Pick elements only if f is asserted
        if (f(p->elem)) L.insert(p->elem);
        p = p->next;
    }
    return L;
}
```

```cpp
T reduce(T f(const T&, const T&), T init) const {
    T x = init; // Initial value
    Node* p = sentinel->next; // First element (if any)
    while (p != sentinel) {
        x = f(x, p->elem); // Composition with next element
        p = p->next;
    }
    return x;
}
```

EXERCISES

Queues implemented as circular buffers

• Design the class queue implemented with a circular buffer (using a vector):
  – The push/pop/front operations should run in constant time.
  – The copy and delete operations should run in linear time.
  – The class should have a constructor with a parameter \( n \) that should indicate 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 and Josephus

• Design the method \texttt{reverse()} that reverses the contents of the list:
  – No auxiliary lists should be used.
  – No copies of the elements should be performed.

• Solve the Josephus problem, for \( n \) people and executing every \( k \)-th person, using a circular list:

  \url{https://en.wikipedia.org/wiki/Josephus_problem}

Merge sort

• Design the method \texttt{merge(const List& L)} that merges the list with another list \( L \), assuming that both lists are sorted. Assume that a pair of elements can be compared with the operator \(<\).

• Design the method \texttt{sort()} that sorts the list according to the \(<\) operator. Consider merge sort and quick sort as possible algorithms.
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?
A significant part of the material used in this chapter has been inspired by the book:


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).

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

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

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

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

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

Space: $O(n^2)$

For undirected graphs, the matrix is symmetric.

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

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

Space: $O(n^2)$

For undirected graphs, the matrix is symmetric.

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

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?

Size of the World Wide Web

- December 2017: 50 billion web pages ($50 \times 10^9$).
- Size of adjacency matrix: $25 \times 10^{10}$ 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.

Credits

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


(several examples, figures and exercises are taken from the book)
Reachability: exploring a maze

Finding the nodes reachable from another node

**Function explore(G, v):**

// 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)  
postvisit(v)

**Notes:**
- Initially, visited(v) is assumed to be false for every v ∈ V.  
- pre/postvisit functions are not required now.

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.

**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 $O(|V| + |E|)$.

**Function DFS(G):**

for all v ∈ V:  
visited(v) = false  
for all v ∈ V:  
if not visited(v): explore(G, v)

**DFS example**

The solid edges (tree edges) form a tree.  

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

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

Connected Components

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

function ConnComp(G):
    // Input: G = (V,E) is a graph
    // Output: every vertex v in G has a unique CC number ccnum[v]
    for all v in V:
        if ccnum[v] == 0:
            // Clean cc numbers
            cc = 1
            // Identifier of the first CC
            for all v in V:
                if ccnum[v] == 0:
                    // A new CC starts
                    explore(G, v, cc);
                    cc = cc + 1;
```

• Performs a DFS traversal assigning a CC number to each vertex.
• The outer loop of ConnComp determines the number of CC’s.
• The variable ccnum[v] also plays the role of visited[v].

Revisiting the explore function

```python
function explore(G, v):
    visited[v] = true
    previsit(v)
    for each edge v,u in E:
        if not visited[u]:
            explore(G, u)
    postvisit(v)
```

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

Let the clock be initially set to 0. Then 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).

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.

Example: pre/post order of (u,v)

Directed Acyclic Graphs (DAGs)

• 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).

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.

Cycles in graphs

A cycle is a circular path:

\[ v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_0. \]

Examples:

\[ B \rightarrow E \rightarrow F \rightarrow B \quad C \rightarrow D \rightarrow A \rightarrow C \]

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

Cycles can never 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).

A DAG is a directed graph without cycles.

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

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.
Topological sort

Initially: TSort = Ø
function explore(G, v):
    visited(v) = true
    previsit(v)
    for each edge (u, v) ∈ E:
        if not visited(u):
            explore(G, u)
    postvisit(v)
return TSort

Strongly Connected Components

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.

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

• Property: If C and \( C' \) are SCCs and there is an edge \( C \rightarrow C' \), 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

• 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 \( C \rightarrow C' \), then the highest post number in C is bigger than the highest post number in \( C' \).

Graphs: Connectivity
© Dept. CS, UPC
37

Strongly Connected Components

Every directed graph can be represented by a meta-graph, where each meta-node represents a strongly connected component.

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.

Reverse graph \( (G^R) \)

function SCC(G):
// Input: G(V,E) a directed graph
// Output: each vertex v has an SCC number in ccnum[v]
G^R= reverse(G)
DFS(G^R) // calculates post numbers
sort V // decreasing order of post number
ConnComp(G)

SCC algorithm

function SCC(G):
// Input: G(V,E) a directed graph
// Output: each vertex v has an SCC number in ccnum[v]
G^R= reverse(G)
DFS(G^R) // calculates post numbers
sort V // decreasing order of post number
ConnComp(G)

Reversing G in linear time

function reverse(G):
// Input: G(V,E) graph represented by an adjacency list
// edges[v] for each vertex v.
// Output: G'(V,E') the reversed graph of G, with the adjacency list edgesR[
for each u ∈ V:
  for each v ∈ edges[u]:
    edgesR[v].insert(u)
return (V, edgesR)

Runtime complexity:
• DFS and ConnComp run in linear time \( O(|V| + |E|) \).
• Can we reverse \( G \) in linear time?
• Can we sort \( V' \) by post number in linear time?
**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 (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 SCC's found?
2. Which are source SCC's and which are sink SCC's?
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?
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.

Hints: A vertex of the graph can be represented by a triple of integers.

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.
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 complete binary tree (except at the bottom level).

Insert a new element

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

\[
 h = \lceil \log_2 N \rceil
\]

Therefore:

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

A more efficient implementation can be proposed in which insertion and extraction are \( O(\log n) \): binary heap.

Containers: Priority Queues

Jordi Cortadella and Jordi Petit
Department of Computer Science
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 -∞, bubble up and remove min element.

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?
Consider the following declaration for a Binary Heap:

```cpp
template <typename T> // T must be a comparable type
class BinaryHeap {
private:
    vector<T> v; // Table for the heap (location 0 not used)
    // Bubbles up the element at location i
    void bubble_up(int i);
    // Bubbles down the element at location i
    void bubble_down(int i);
}
```

Give an implementation for the methods `bubble_up` and `bubble_down`.
**Breadth-first search**

- **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**

```cpp
Function BFS(G, s)
    // 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: $\forall u \in V$: reached[$u$] $\iff$ $u$ is reachable from $s$.

function $\text{BFS}(G, s)$
for all $u \in V$:
    reached[$u$] = false
Q = $\square$ // Empty queue
Q.push_front($s$)
reached[$s$] = true
while not Q.empty():
    u = Q.pop_front()
    for all $(u, v) \in E$:
        if not reached[v]:
            reached[v] = true
            Q.push_back(v)

function $\text{DFS}(G, s)$
for all $u \in V$:
    reached[$u$] = false
S = $\square$ // Empty stack
S.push($s$)
reached[$s$] = true
while not S.empty():
    u = S.pop()
    for all $(u, v) \in E$:
        if not reached[u]:
            reached[v] = true
            S.push(v)

Graphs: Shortest paths

Example

Reachability: BFS vs. DFS

$G$: A B C D E F G H
$S$: A 0 B 2 C 3 D 1 E 3 F 5 G 9 H 10

BFS order: A B D C F E G H
Distance: 0 1 2 2 3 3 3 3

DFS order: A B C E F G H D

Dijkstra’s algorithm: invariant

Example

Example

Example

Example

Example

Inefficient: many cycles without any interesting progress. How about real numbers?

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example
Dijkstra’s algorithm for shortest paths

Function ShortestPaths(G, l, s)
// Input: Graph G(V, E), source vertex s, positive edge lengths \( l(e) \in E \)
// Output: dist[v] has the distance from s, prev[v] has the predecessor in the tree
for all \( u \in V \):
    dist[u] = \( \infty \)
    prev[u] = nil
dist[s] = 0
Q = makequeue(V) // using dist as keys
while not Q.empty():
    u = Q.deleteMin()
    for all \( (u, v) \in E \):
        if dist[v] > dist[u] + \( l(u, v) \):
            dist[v] = dist[u] + \( l(u, v) \)
            prev[v] = u
            Q.decreasekey(v)

Complexity:

\( O(V + E) \) for connected graphs:

\[ O((|V| + |E|) \log |V|) \]

Dijkstra’s algorithm: complexity

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!

Bellman-Ford algorithm

Function ShortestPaths(G, l, s)
// Input: Graph G(V, E), source vertex s, edge lengths \( l(e) \in E \), no negative cycles.
// Output: dist[v] has the distance from s, prev[v] has the predecessor in the tree
for all \( u \in V \):
    dist[u] = \( \infty \)
    prev[u] = nil
dist[s] = 0
repeat |V|−1 times:
    for all \( (u, v) \in E \):
        if dist[v] > dist[u] + \( l(u, v) \):
            dist[v] = dist[u] + \( l(u, v) \)
            prev[v] = u

Complexity: \( O(|V| \cdot |E|) \).

Bellman-Ford: example

Example

Dijkstra’s algorithm does not work:

Dijkstra would say that the shortest path \( S \rightarrow A \) 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:

\[ s \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \ldots \rightarrow u_k \rightarrow t \]

• 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: \( O(|V| \cdot |E|) \).

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!

Bellman-Ford: example

Graphs with negative edges

• The shortest path from \( s \) to \( t \) can have at most \(|V| − 1\) edges:

\[ s \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow \ldots \rightarrow u_k \rightarrow t \]

• 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: \( O(|V| \cdot |E|) \).
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.

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's property:

- Non-negative edges: $O(|V| + |E|)$
- Negative edges: $O(|V|^2)$
- DAG: $O(|V| + |E|)$

A related problem: All-pairs shortest paths
- Floyd-Warshall algorithm ($O(|V|^3)$), based on dynamic programming.
- Other algorithms exist.

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.
Nesting boxes

A $d$-dimensional box with dimensions $(x_1, x_2, ..., x_d)$ nests within another box with dimensions $(y_1, y_2, ..., y_d)$ if there exists a permutation $\pi$ on $\{1, 2, ..., d\}$ such that:

$$x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, ..., x_{\pi(d)} < y_d.$$  

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 $\{B_1, B_2, ..., B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, ..., B_{i_k} \rangle$ of boxes such that $B_{i_j}$ nests within $B_{i_{j+1}}$, for $j = 1, 2, ..., k - 1$. Express the running time of your algorithm in terms of $n$ and $d$.

Graphs: Minimum Spanning Trees and Maximum Flows

Jordi Cortadella and Jordi Petit
Department of Computer Science

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!

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 weight $w(T') \leq w(T)$. Since $T$ is an MST, the weights must be equal.

The cut property: example

Minimum Spanning Tree

Given an 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.

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 does not produce a cycle.
  - If there would be two paths between two nodes, the union of the paths would contain a cycle.
- 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.
- 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.

Minimum Spanning Tree

Graphs: MST and Max Flow

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.
  - 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 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.

Any scheme like this works (because of the properties of trees):

```plaintext
X = \{1\} // The set of edges of the MST
repeat |V| - 1 times:
pick a set $S \subseteq V$ for which $X$ has no edges between $S$ and $V - S$
let $e \notin E$ be the minimum-weight edge between $S$ and $V - S$
$X = X \cup \{e\}$
```
**MST: two strategies**

**Prim’s algorithm**

- 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 |E| times
  - union: |V| − 1 times

**Disjoint sets**

- The nodes are organized as a set of trees. Each tree represents a set.
- Each node has two attributes:
  - parent (n): 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.

**Kruskal’s algorithm**

- A set of nodes (\(S\)) is in the tree.
- Progress: The lightest edge with exactly one endpoint in \(S\) is added.
- Invariant: A set of trees (forest) has been constructed.
- Progress: The lightest edge between two trees is added.

**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?**

**Function makeset(x):**

- \(\pi(x) = x\)
- \(\text{rank}(x) = 0\)

**Function find(x):**

- while \(x \neq \pi(x)\):
  - \(x = \pi(x)\)
- return \(x\)

**Function Kruskal(G, w)**

- // Input: A connected undirected Graph G(V,E)
- // with edge weights \(w_e\).
- // Output: An MST defined by the vector prev.
- \(\pi = \text{makequeue}()\)
- for each \((u,v) \in E:\)
  - \(Q\).insert(\(u, v\))

**Function Prim(G, w)**

- // Input: A connected undirected Graph G(V,E)
- // with edge weights \(w_e\).
- // Output: An MST defined by the vector prev.
- \(\pi = \text{makequeue}()\)
- for each \((u,v) \in E:\)
  - \(Q\).insert(\(u, v\))
  - \(\text{visited}(u) = \text{false}\)
  - \(\text{prev}(u) = \text{nil}\)
  - \(\text{rank}(u) = 0\)

**Complexity:** \(O(|E| \log |V|)\)
Disjoint sets

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 $\leq \log_2 n$.

Disjoint sets: path compression

Function $\text{find}(x)$:
- if $x \neq \pi(x), \pi(x) = \text{find}(\pi(x))$
- return $\pi(x)$

Amortized cost of find: $O(1)$
Kruskal's cost: $O(E \log V)$ (if sorting has linear cost)

Max-flow/min-cut problems

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 problems: applications

Model:
- A directed graph $G = (V, E)$
- Two special nodes $s, t \in V$. $s$ is the source and $t$ is the sink.
- Capacities $c_e > 0$ on the edges.

Goal: assign a flow $f_e$ to each edge $e$ of the network satisfying:
- $0 \leq f_e \leq c_e$ for all $e \in E$ (edge capacity not exceeded)
- For all nodes $u$ (except $s$ and $t$), the flow entering the node $u$ is equal to the flow exiting the node $u$.

Size of a flow: total quantity sent from $s$ to $t$ (equal to the quantity leaving $s$):

$$\sum_{(u,v) \in E} f_{uv} = \sum_{(v,u) \in E} f_{vu}$$

Augmenting paths

Find an augmenting path

An augmenting path may reverse some of the flow previously assigned.

Augmenting paths can have forward and backward edges.
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.

Ford-Fulkerson algorithm: example

Function Ford-Fulkerson \((G,s,t)\)

// 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 \((u,v) \in E\):
// \( f(u,v) = c(u,v) \) // Forward edges
// \( f(v,u) = 0 \) // Backward edges

while there exists a path \( p = s \rightarrow t \) in the residual graph:

\[
\begin{align*}
  f(p) &= \min(f(u,v), c(u,v)) \\
  f(u,v) &= f(u,v) - f(p) \\
  f(v,u) &= f(v,u) + f(p)
\end{align*}
\]

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 \((L, R)\):

\[ \text{size}(f) \leq \text{capacity}(L, R) \]

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 if it admits no augmenting path.

Residual graph

Min-cut algorithm

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**

- 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.

**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 \( s(v) \) can be injected (positive) or extracted (negative) at every vertex \( v \). The flow must satisfy:
  \[
  \sum_{u: u \to v} f(u \to v) - \sum_{w: v \to w} f(v \to w) = x(v).
  \]
- **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**

1. Find the maximum flow from S to T. Give a sequence of augmenting paths that lead to the maximum flow.
2. Draw the residual graph after finding the maximum flow.
3. Find a minimum cut between S and T.

**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 Flaranquhar. 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 Flaranquhar 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 Flaranquhar 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 Flaranquhar and Tidmouth.

For example, given the following input graph, your algorithm should return the number 2.

**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 blood units. 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.

**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.
Data are often organized hierarchically

source: https://en.wikipedia.org/wiki/Tree_structure

Company structure

Mind maps

Genealogical trees

Probability trees

Parse trees
**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, I, 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 with linked lists**

```cpp
struct TreeNode {
    Type element;
    list<TreeNode> children; // Linked list of children
};
```

**Tree: representation with vectors**

```cpp
struct TreeNode {
    Type element;
    vector<TreeNode> children; // Vector of children
};
```

**Print a tree**

```cpp
/** Prints a tree indented according to depth. */
void print(const Tree& T, int depth = 0) {
    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T.name << endl;
    // Print the children with depth + 1
    for (const TreeNode& child : T.children)
        print(child, depth + 1);
}
```

This function executes a **preorder** traversal of the tree: each node is processed before the children.

**Print a tree (postorder traversal)**

```cpp
void print(const Tree& T, int depth = 0) {
    // Print the children with depth + 1
    for (const TreeNode& child : T.children)
        print(child, depth + 1);
    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T.name << endl;
}
```

**Postorder traversal**: each node is processed after the children.
/** Prints a tree (in postorder) indented according to depth. */
void printPostOrder(const Tree& T, int depth) {
    // Print the children with depth + 1
    for (const Tree& child: T.children)
        printPostOrder(child, depth + 1);
    // Print the root indented by 2*depth
    cout << string(2*depth, ' ') << T.name << endl;
}

This function executes a postorder traversal of the tree: each node is processed after the children.

Example: expression trees
Expressions are represented by strings in postfix notation in which the characters 'a'...'z' represent operands and the characters '+' and '*' represent operators.

How to build an expression tree
Expressions are represented by strings in postfix notation in which the characters 'a'...'z' represent operands and the characters '+' and '*' represent operators.
How to build an expression tree

1. Start with an empty stack.
2. Read the expression from left to right.
3. For each operand, push it onto the stack.
4. For each operator, pop the top two operands from the stack, create a new node with the operator, and push the new node back onto the stack.
5. Continue until all operators and operands have been used.
6. The final node on the stack is the root of the expression tree.

Example:

Expression: (a * b) + c

Steps:
1. Push 'a' onto the stack.
2. Push 'b' onto the stack.
3. Pop 'b' and 'a', create a new node with '*', push it back onto the stack.
4. Push 'c' onto the stack.
5. Pop 'c' and '*', create a new node with '+', push it back onto the stack.
6. The final node is the root of the expression tree.

Expression Tree:

```
        +
       /|
      / |  
     *  c  
    /   
   a    b
```

Trees © Dept. CS, UPC
Example: expression trees

**Builds an expression tree from a string.**
```cpp
Expr buildExpr(const string& expr) {
    stack<Expr> S;
    // Visit the chars of the string sequentially
    for (char c : expr) {
        if (c == '+' || c == '*') {
            Expr T = new ExprTree(c, nullptr, nullptr);
            S.push(T);
        } else {
            // c is an operator ('-' or '*')
            Expr right = S.top();
            S.pop();
            Expr left = S.top();
            S.pop();
            Expr T = new ExprTree(c, left, right);
            S.push(T);
        }
    }
    // The stack has only one element and is freed after return
    return S.top();
}
```

**Infix representation of an expression tree.**
```cpp
string infixExpr(const Expr T) {
    // Let us first check the base case (an operand)
    if (T->left == nullptr) return string(1, T->elem);
    // We have an operator. Return ( T->left ) op ( T->right )
    return (infixExpr(T->left) + T->op + infixExpr(T->right) + "+");
}
```

**Example: expression trees (no freeExpr yet)**
```
int main() {
    Expr T = buildExpr("abc+def*gh1234567890abc");
    do {
        cout << evalExpr(T, visitor v);
        cout << endl;
        T = evalExpr(T, visitor v);
    } while (not T->isLeaf);
    freeExpr(T); // Not implemented yet
    return 0;
}
```

**Tree traversals**

- **Preorder traversal:** Visit the root first, followed by the subtrees.
- **Inorder traversal:** Visit the left subtree, then the root, then the right subtree.
- **Postorder traversal:** Visit the left subtree, then the right subtree, then the root.

```cpp
void preorder(Tree T, visitor v) {
    if (T != nullptr) {
        v(T->elem);
        preorder(T->left, v);
        preorder(T->right, v);
    }
}
```

**Example: expression trees**
```
TreeNode buildExpr(const string& expr) {
    stack<Expr> S;
    // Visit the chars of the string sequentially
    for (char c : expr) {
        if (c == '+' || c == '*') {
            TreeNode T = new TreeNode(c);
            S.push(T);
        } else {
            // c is an operator ('-' or '*')
            TreeNode right = S.top();
            S.pop();
            TreeNode left = S.top();
            S.pop();
            TreeNode T = new TreeNode(c, left, right);
            S.push(T);
        }
    }
    // The stack has only one element and is freed after return
    return S.top();
}
```

**Example: expression trees**
```
Example: expression trees
/** Evaluates an expression taking V as the value of the variables (e.g., V['a'] contains the value of a). */
/*
 * evalExpr(const Expr T, const map<char,int>& V) {
 *     if (T->left == nullptr) return V.at(T->elem);
 *     int l = evalExpr(T->left, V);
 *     int r = evalExpr(T->right, V);
 *     return T->op == '+' ? l + r : l * r;
 * }
 */
```

**Exercise:**
- Design the function freeExpr.
- 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).

**Remember using Expr = ExprTree*;**
EXERCISES

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 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 to 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 predefined bounding box of size \(W \times H\), with the origin located in the top-left corner.

Design the function

```c
void draw(Tree T, double W, double H)
```

to assign values to the attributes \(x\) and \(y\) of all nodes of the tree in such a way that the lines that connect the nodes do not cross.

Suggestion: calculate the coordinates in two steps. First assign \((x, y)\) coordinates using some arbitrary unit. Next, shift/scale the coordinates to exactly fit in the bounding box.

\((0,0)\) \((W, H)\)
Set and Dictionary

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

Note:
- A dictionary can be treated as a set of keys, each key having an associated value.
- We will focus on the implementation of sets.

Possible implementations of a set

<table>
<thead>
<tr>
<th>Unsorted list or vector</th>
<th>Sorted vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion</td>
<td>0(n), if checking for duplicate keys, O(1) otherwise.</td>
</tr>
<tr>
<td>Deletion</td>
<td>O(n) since it has to find the item along the list.</td>
</tr>
<tr>
<td>Lookup</td>
<td>O(n) since the list must be scanned.</td>
</tr>
<tr>
<td>Good for</td>
<td>Small sets.</td>
</tr>
</tbody>
</table>

Can we have a data structure with efficient insertion/deletion/lookup operations?

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.

Binary Search Trees

This is a binary search tree

This is not a binary search tree

Contents: 4?

Contains: Move to left/right depending on the value.
Stop when:
- The value is found (contained)
- No more elements exist (not contained)
Binary Search Trees: insert

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)
- simple case (one child)
- complex case (two children)

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 in-order 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 $d$?

- 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.

$$d = 0$$
$$d = 1$$
$$d = 2$$
$$d = 3$$

$$\text{ILP} = 0 \times 1 + 1 \times 2 + 2 \times 3 + 3 \times 5 = 23$$

$$\text{Avg. IPL} = \frac{23}{11} = 2.09$$
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.

\[ D(n) \]

\[ D(n) = D(i) + D(n - i - 1) + (n - 1) \]

If all subtree sizes are equally likely, then the average value for \( D(i) \) and \( D(n - i - 1) \) is

\[ \frac{1}{n} \sum_{j=0}^{n-1} D(j) \]

Therefore,

\[ D(n) = \frac{2}{n} \sum_{j=0}^{n-1} D(j) + n - 1 \]

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.

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 \( O(\log n) \).

Random BST after \( n^2 \) insert/removes

Worst-case runtime: \( O(n) \)

Balanced trees

• 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
  – Splay trees
  – B-trees

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
The height is $\Omega(\log n)$

- The size $n$ of a tree with height $h$ is:
  \[ n \leq 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1 \]
  (full binary tree)

- Therefore,
  \[ \log_2(n+1) - 1 \leq h \]
  and $h = \Omega(\log n)$.

Unbalanced insertion: 4 cases

Any newly inserted item may fall into any of the four subtrees (LL, LR, RL or RR).

A new insertion may violate the balancing property. Re-balancing might be required.

The important question: what is the size of an AVL tree with height $h$?

Theorem: the height of an AVL tree with $n$ nodes is $\Theta(\log n)$.

Proof in two steps:
- The height is $\Omega(\log n)$.
- The height is $O(\log n)$.

Smallest AVL tree with $h = 6$.

The recurrence
\[ S(h) = S(h-1) + S(h-2) + 1 \]
resembles the one of the Fibonacci numbers. A tighter bound can be obtained.

- Theorem: the height of an AVL tree with $n$ internal nodes satisfies:
  \[ h < 1.44 \log_2(n+2) - 1.328 \]

Single rotation: the left-left case

Insertion

Single rotation: the right-right case

Rotation

Theorem: The height of an AVL tree with $n$ nodes is $\Theta(\log n)$.

Proof in two steps:
- The height is $\Omega(\log n)$.
- The height is $O(\log n)$.
The height must be stored at each node. Only the unbalancing factor \((-1, 0, 1)\) is strictly required. The insertion/deletion operations are implemented similarly as in BSTs (recursively). The re-balancing of the tree is done when the recursive calls return to the ancestors (check heights and rotate if necessary). 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.

Sets & Dictionaries

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?

AVL

Depict the three AVL trees after sequentially inserting the values 31, 32 and 33 in the following AVL tree:

AVL

• Build an AVL tree by inserting the following values: 15, 21, 23, 11, 13, 8, 32, 33, 27. Show the tree before and after applying each rotation.

• Depict the AVL tree after removing the elements 23 and 21 (in this order). When removing an element, move up the largest element of the left subtree.
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.)

A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

• 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

A usual hash function for a string with size \( n \) is as follows:
\[ h(x) = \sum_{i=0}^{n-1} x_i \cdot p^i \]
where \( p \) is a prime number 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.

• A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

A (Boolean) vector with one location for each possible plate:

Where can we use prime numbers?

– In the size of the hash table
– In the coefficients of the hash function

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?

Where can we use prime numbers?

– In the size of the hash table
– In the coefficients of the hash function

A hash function for a string
\[ h(x) = \prod_{i=0}^{n-1} x_i \]

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 collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

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).

– A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

• Can we use a data structure with size \( O(n) \), where \( n \) is the size of the parking?

A hash function

• We can calculate the location for item \( x \) as
\[ h(x) \mod m \]

where \( h \) is the hash function and \( m \) is the size of the hash table.

• A good hash function must scatter items randomly and 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.

A usual hash function for a string with size \( n \) is as follows:
\[ h(x) = \sum_{i=0}^{n-1} x_i \cdot p^i \]

where \( p \) is a prime number 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.

• A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

A (Boolean) vector with one location for each possible plate:

• Add the last three characters (e.g., ASCII codes) of plate:
\[ h(x) = x_{n-1} + x_{n-2} + x_{n-3} \]

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.

• 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.

A hash function

• 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

A hash function for a string
\[ h(x) = \prod_{i=0}^{n-1} x_i \]

This function can be efficiently implemented using Horner’s rule for the evaluation of a polynomial.

• Here is a slightly different implementation (reversed string):

```c
/** Hash function for strings */
unsigned int hash(const string& key, int int tableSize) {
    unsigned int hval = 0;
    for (char c: key) hval = 37*hval + c;
    return hval%tableSize;
}
```

Handling collisions

• A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

• 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

Example of hash function for strings

• A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

Handling collisions

• A collision is produced when
\[ h(x_1) \equiv h(x_2) \mod m \]

• 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
- Each slot is a list of the items that have the same hash value.
- Load factor: $\lambda = \frac{\text{number of items}}{\text{table size}}$
- $\lambda$ 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.

### 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

<table>
<thead>
<tr>
<th>Cases</th>
<th>Space: $O(n + M)$</th>
<th>Time: $O(n/M)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M \gg n$</td>
<td>$O(M)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$n \gg M$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$M = O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

The best strategy is to have $M = O(n)$ that allows to maintain a constant-time access without wasting too much memory. Rehashing should be applied to maintain $M = O(n)$.

### Binary Search Trees vs. Hash Tables

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Search Tree</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion/Deletion/Lookup</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted Iteration</td>
<td>In-order traversal: $O(n)$</td>
<td>Needs an extra sorted vector: $O(n \log n)$</td>
</tr>
<tr>
<td>Hash function</td>
<td>Not required</td>
<td>Required</td>
</tr>
<tr>
<td>Total order</td>
<td>Required</td>
<td>Not required</td>
</tr>
<tr>
<td>Range search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Not a clear winner

### Application: data integrity check
Hash functions are used to guarantee the integrity of data (files, messages, etc) when distributed between different locations.

### Application: password verification
Different hashing algorithms exist: MD5, SHA1, SHA255, ...

The probability of collision is extremely low.

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, samsung,...).
Given the values \(\{2341, 4234, 2839, 430, 22, 397, 3920\}\), a hash table of size 7, and hash function \(h(x) = x \mod 7\), show the resulting tables after inserting the values in the given order with each of these collision strategies:

- Separate chaining
- Linear probing

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 AVLs.
- Using hash tables.
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!

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.

AES scheme

- 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).

Cryptography

- 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 protocols: problems

Every channel requires a different key

The key cannot be transmitted through the communication channel!

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: \(M = S_A(X)\).

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, \(d \leq \gcd(a, b)\), 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 \(\gcd(a, b) \leq d\).
  – Therefore, \(d = \gcd(a, b)\).

Public-key protocols

• 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.

But, how to create a cryptosystem like this? Using number theory.

RSA cryptosystem
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 \( x \mapsto x^e \mod N \) 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 \( x \in \{0, \ldots, N-1\} \),
\[(x^e)^d \equiv x \mod N.\]

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 \( d = 3^{-1} \mod 64 = 43 \) using extended Euclid's algorithm:
\[ 43 \cdot 3 - 2 \cdot 64 = 1. \]

Note: the algorithm gives \( 1 \cdot 64 - 21 \cdot 3 = 1 \), but \(-21 = 43 \mod 64\).

Let us consider the message \( x = 12 \).
– The sender must encrypt \( x \) as \( y = x^e \mod 85 = 28 \).
– The receiver must decrypt \( y \) by computing \( x = y^{3^{-1}} \mod 85 = 12 \).

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 \( y = x^e \mod N \). But \( x \) is a large \( N \)-bit number and checking all values would take exponential time (impractical).
  2. Try to guess \( d \) and calculate \( x^d \mod N \). 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.

### The RSA cryptosystem: example

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 \( y = x^e \mod N \).
- Bob decodes the message by computing \( x^{ed} \mod N \).

## 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 \( y = x^e \mod N \). But \( x \) is a large \( N \)-bit number and checking all values would take exponential time (impractical).
  2. Try to guess \( d \) and calculate \( x^d \mod N \). 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.

### 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 \( y = x^e \mod N \). But \( x \) is a large \( N \)-bit number and checking all values would take exponential time (impractical).
  2. Try to guess \( d \) and calculate \( x^d \mod N \). 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.
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](image)

The result is "accumulated" to the result of previous steps.

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? \(\rightarrow\) integrity.
- How does Alice know that Bob has signed the document (and not somebody else)? \(\rightarrow\) authentication.

Example: SHA-1

![Digital signatures](image)

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., $M = 11101 \cdot 00011 \cdot 01000 \cdot 110$. 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, $01110 \oplus 11010 = 10100$.

**EXERCISES**

- 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.
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 C++ or python) 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 \( encode(x, e, N) \) that computes \( x^e \mod N \). 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 \( encode(encode(x, e, N), d, N) = x \).

#### Example:

\( p = 79 \), \( q = 491 \).

Public key: \((38789, 11)\), Secret key: 31271.

\( e(2) = 2048 \), \( e(19) = 23855 \), \( e(32757) = 4 \),
\( e(38788) = 38788 \), \( e(10) = 18550 \).
Why Fourier Transform?

**Fourier series**

- Periodic function $f(t)$ of period 1:
  $$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(2\pi nt) + \sum_{n=1}^{\infty} b_n \sin(2\pi nt)$$

- Fourier coefficients:
  $$a_n = 2 \int_0^T f(t) \cos(2\pi nt) \, dt, \quad b_n = 2 \int_0^T f(t) \sin(2\pi nt) \, dt$$

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

**Polynomials: coefficient representation**

- A polynomial is represented as a vector of coefficients $(a_0, a_1, \ldots, a_{n-1})$:
  $$A(x) = 2x^3 + x^2 - 4x + 3$$
  $$A = (3, -4, 1, 0, 2)$$

- Addition: $O(n)$
  $$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_{n-1} + b_{n-1})x^{n-1}$$

- Evaluation: $O(n)$ using Horner’s method
  $$A(x) = a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1})\cdots)))$$

- Multiplication: $O(n^2)$ using brute force
  $$A(x) \cdot B(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

**Interpolation: Lagrange polynomials**

- A polynomial is represented as a set of pairs $(x_i, y_i)$:
  
  $$A(x) = \sum_{k=0}^{n-1} y_k \prod_{j \neq k} (x - x_j) \prod_{k=0}^{n-1} (x - x_k)$$

- A polynomial is uniquely identified by its evaluation at $n + 1$ distinct values of $x$. 

**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 $x$. 

**Discrete-time signals**

- Periodic function $f(t)$ of period 1:
  $$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(2\pi nt) + \sum_{n=1}^{\infty} b_n \sin(2\pi nt)$$

- Fourier coefficients:
  $$a_n = 2 \int_0^T f(t) \cos(2\pi nt) \, dt, \quad b_n = 2 \int_0^T f(t) \sin(2\pi nt) \, dt$$

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

**Polynomials: point-value representation**

- Evaluation

- Interpolation

- $A(x) = \frac{1}{3} x(x - 2) + (x + 1)(x - 2) + \frac{x + 1}{3}$

- $A(x) = x^2 - 2$
Interpolation: Lagrange polynomials

Conversion between both representations

From coefficients to point-values

Given a polynomial \( a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^{n-1} \), evaluate it at \( n \) different points \( x_0, x_1, \ldots, x_{n-1} \).

Selection of the evaluation points

Fast Fourier Transform © Dept. CS, UPC 11

Fast Fourier Transform © Dept. CS, UPC 12

Fast Fourier Transform © Dept. CS, UPC 13

Fast Fourier Transform © Dept. CS, UPC 14

Fast Fourier Transform © Dept. CS, UPC 15

Fast Fourier Transform © Dept. CS, UPC 16

Fast Fourier Transform © Dept. CS, UPC 17

Fast Fourier Transform © Dept. CS, UPC 18


We want to evaluate \( A(x) \) at \( n \) different points. Let us choose them to be positive-negative pairs: \( \pm x_0, \pm x_1, \ldots, \pm x_{n/2-1} \).

The computations for \( A(x_i) \) and \( A(-x_i) \) overlap a lot.

Split the polynomial into odd and even powers

\[
3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)
\]

The terms in parenthesis are polynomials in \( x^2 \):

\[
A(x) = A_e(x^2) + xA_o(x^2)
\]

The calculations needed for \( A(x_i) \) can be reused for computing \( A(-x_i) \).

\[
A(x_i) = A_e(x_i^2) + x_iA_o(x_i^2)
\]

\[
A(-x_i) = A_e(x_i^2) - x_iA_o(x_i^2)
\]

Evaluating \( A(x) \) at \( n \) paired points

\[
\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}
\]

reduces to evaluating \( A_e(x) \) and \( A_o(x) \) at just \( n/2 \) points: \( x_0^2, \ldots, x_{n/2-1}^2 \)

Evaluating \( A(x) \) at \( n \) different points

\[
\left[ y_0 \ y_1 \ y_2 \ \ldots \ y_{n-1} \right]
\]

Runtime: \( O(n^3) \) matrix-vector multiplication (apply Horner \( n \) times).

Horner’s rule:

\[
p(x) = a_0 + x \left( a_1 + x (a_2 + x (a_3 + \cdots + x (a_{n-1} + x_{n0}) \cdots)) \right)
\]

Selection of the evaluation points

We need \( x_0^2 \) and \( x_1^2 \) to be a plus-minus pair. But a square cannot be negative!

Not if we use real numbers. How about complex numbers?

\[
\sqrt{-i} = \pm \frac{1}{\sqrt{2}} (1 + i)
\]

\[
\sqrt{-1} = \pm \frac{1}{\sqrt{2}} (1 - i)
\]

Note:
Complex numbers: review

- Polar coordinates: \( r \cos \theta + i \sin \theta = r e^{i\theta} \)
- Length: \( r = \sqrt{a^2 + b^2} \)
- Angle \( \theta \in [0, 2\pi) \): \( \cos \theta = \frac{a}{r}, \sin \theta = \frac{b}{r} \)

Some examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>( i )</th>
<th>( 5 + 5i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polar coords</td>
<td>( (1, \pi) )</td>
<td>( (5\sqrt{2}, \pi/4) )</td>
</tr>
</tbody>
</table>

Complex numbers: multiplication

\[(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)\]

For any \( z = (r, \theta) \):

\[-z = (r, \theta + \pi), \text{ since } -1 = 1, \pi\]

If \( z \) is on the unit circle, then \( z^n = (1, n\theta) \)

Recursive divide-and-conquer

Evaluate \( A(x) \) at \( n \)th roots of unity Evaluate \( A_e(x^2) \) and \( A_o(x^2) \) at \( \omega \)th roots of unity

Divide-and-conquer step

Evaluate \( A(x) \) at \( \omega^n \)th roots of unity

Divide-and-conquer steps

FFT algorithm

```
function FFT(A, \omega)
    Inputs: \( A = (a_0, a_1, ..., a_{n-1}) \), for \( n \) a power of 2
    \( \omega \): A primitive \( n \)th root of unity
    Output: \( (A(1), A(\omega), A(\omega^2), ..., A(\omega^{n-1})) \)

    if \( \omega = 1 \): return \( A \)  // Only 1 coef. (constant)
    for \( k = 0 \) to \( n-1 \):
        \( A(\omega^k) = A_0(\omega^{2k}) + \omega^k A_0(\omega^{2k}) \)
        \( A_0(\omega^k), A_0(\omega^{2k}), ..., A_0(\omega^{(n-1)k}) \) = FFT(A_0, \omega^2)
        \( A_0(\omega^k), A_0(\omega^{2k}), ..., A_0(\omega^{(n-1)k}) \) = FFT(A_0, \omega^2)
    return \( (A(1), A(\omega), A(\omega^2), ..., A(\omega^{n-1})) \)
```
**Fast Fourier Transform (FFT) Algorithm**

For $k = 0$ to $n-1$: $A(\omega^k) = A_e(\omega^{2k}) + \omega^k A_o(\omega^{2k})$

**Example ($n = 8$):**

$A(\omega^0) = A_e(\omega^0) + \omega^0 A_o(\omega^0)$
$A(\omega^1) = A_e(\omega^2) + \omega A_o(\omega^2)$
$A(\omega^2) = A_e(\omega^4) + \omega^2 A_o(\omega^4)$
$A(\omega^3) = A_e(\omega^6) + \omega^3 A_o(\omega^6)$
$A(\omega^4) = A_e(\omega^8) + \omega^4 A_o(\omega^8)$
$A(\omega^5) = A_e(\omega^{10}) + \omega^5 A_o(\omega^{10})$
$A(\omega^6) = A_e(\omega^{12}) + \omega^6 A_o(\omega^{12})$
$A(\omega^7) = A_e(\omega^{14}) + \omega^7 A_o(\omega^{14})$

**Unfolding the FFT (butterfly diagram)**

**Example ($n = 8$):**

$A(\omega^0) = A_e(\omega^0) + \omega^0 A_o(\omega^0)$
$A(\omega^1) = A_e(\omega^2) + \omega A_o(\omega^2)$
$A(\omega^2) = A_e(\omega^4) + \omega^2 A_o(\omega^4)$
$A(\omega^3) = A_e(\omega^6) + \omega^3 A_o(\omega^6)$
$A(\omega^4) = A_e(\omega^8) + \omega^4 A_o(\omega^8)$
$A(\omega^5) = A_e(\omega^{10}) + \omega^5 A_o(\omega^{10})$
$A(\omega^6) = A_e(\omega^{12}) + \omega^6 A_o(\omega^{12})$
$A(\omega^7) = A_e(\omega^{14}) + \omega^7 A_o(\omega^{14})$

**FFT: asymptotic complexity**

- The runtime of the FFT can be expressed as:
  $$T(n) = 2 \cdot T(n/2) + O(n)$$

- Using the Master Theorem we conclude:
  Runtime FFT $(n) = O(n \log n)$

**Why is it called a butterfly diagram?**

- FFT algorithm
  function FFT(a, $\omega$)
    Inputs: $a = (a_0, a_1, \ldots, a_{n-1})$, for $n$ a power of 2
    $\omega$: A primitive $n$th root of unity
    Output: $(a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{n-1}))$
    if $\omega = 1$: return $a$ // $a$ has only one element
    $(s_0, s_1, \ldots, s_{n/2-1}) = FFT(a_0, a_2, \ldots, a_{n-2}, \omega^2)$
    $(s'_0, s'_1, \ldots, s'_{n/2-1}) = FFT(a_1, a_3, \ldots, a_{n-1}, \omega^2)$
    for $k = 0$ to $n/2 - 1$: // FFT shuffling
      $r_k = s_k + \omega^k s'_k$
      $r_{k+n/2} = s_k - \omega^k s'_k$
    return $(r_0, r_1, \ldots, r_{n-1})$
The Fast Fourier Transform computes:
\[
\begin{bmatrix}
    y_0 \\
    y_1 \\
    \vdots \\
    y_{n-1}
\end{bmatrix} = \begin{bmatrix}
    1 & 1 & \cdots & 1 \\
    1 & \omega & \cdots & \omega^{n-1} \\
    \vdots & \vdots & \ddots & \vdots \\
    1 & \omega^{n-2} & \cdots & \omega^{2(n-1)}
\end{bmatrix} \begin{bmatrix}
    a_0 \\
    a_1 \\
    \vdots \\
    a_{n-1}
\end{bmatrix}
\]
where \( \omega = e^{2\pi i / n} \).

Let us call \( F_n(\omega) \) the Fourier matrix. Thus,
\[
y = F_n(\omega) \cdot a
\]
How about if we know \( y \) and we want to obtain \( a \)?

Note: The inverse of unitary matrix is its conjugate transpose.

Example: from values to coefficients
Let us consider a polynomial:
\[ P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]
We have \( n = 4 \) and \( \omega = i \). Let us assume that the evaluation at four different points is:
\[ \begin{align*}
    P(1) &= 2 \\
    P(i) &= 1 - i \\
    P(-1) &= 4 \\
    P(-i) &= 1 + i
\end{align*} \]
We want to calculate the coefficients \( (a_0, a_1, a_2, a_3) \) using the inverse FFT, i.e.,
\[ [a_0, a_1, a_2, a_3] = \frac{1}{4} \text{FFT}[2, 1 - i, 4, 1 + i, \omega^{-1}] \]
Gilbert Strang (MIT, 1994): “the most important numerical algorithm of our lifetime.”

EXERCISES

Multiplication

Consider the polynomials $1 + x - 2x^2 + x^3$ and $-1 + x^2$:

– Choose an appropriate power of two to execute the FFT for the polynomial multiplication. Find the value of $\omega$.

– Give the result of the FFT for $x^2 - 1$ using the value of $\omega$ required for the multiplication (no need to execute the FFT).

Polynomial evaluation

Consider the FFT of the polynomial $x^2 + 2x + 1$:

– Find the value of $\omega$ to execute the FFT.

– In which points the polynomial must be evaluated?

– Execute the FFT and give the point-value representation of the polynomial.

Multiplication using FFT

Consider the polynomials $-1 + 2x + x^2$ and $1 + 2x$:

– Choose an appropriate power of two to execute the FFT. Find the value of $\omega$.

– Calculate their point-value representation using the FFT (execute the FFT algorithm manually).

– Calculate the product of the point-value representations.

– Execute the inverse FFT to obtain the coefficients of the product.