**Algorithmics and Programming II**

**Introduction**

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

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

Evaluation items:
- Projects (Proj), Parcial Lab (Plab), Final Theory (Fth), Final (Flab).

Grading:
- $N_1 = 0.2 \times \text{Proj} + 0.25 \times \text{Plab} + 0.25 \times \text{Flab} + 0.3 \times \text{Fth}
- $N_2 = 0.2 \times \text{Proj} + 0.4 \times \text{Flab} + 0.4 \times \text{Fth}
- $N = \max(N_1, N_2)$

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.

**First project: Containers**

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

**Second Project: GPS**

- Navigation: find the shortest path
- How to encrypt messages?

**Problems on polygons**

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

**The Closest-Points problem**

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

**Material**

- 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)

**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.”

**Navigation: find the shortest path**

- [Telegram](https://telegram.org)
- Language: Python

**Telegraph**

- Navigation: find the shortest path
- How to encrypt messages?
Abstract Data Types (ADTs) and Object-Oriented Programming

Jordi Cortadella and Jordi Petit
Department of Computer Science

How many horses can you distinguish?

Abstract Data Types (ADTs)
A set of objects and a set of operations to manipulate them:

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

Data type: Graph

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 compare two strings.
  – We cannot divide two strings!
• Programmers can add new operations to the primitive data types: (e.g., add(), multiply(), string())...
• The programming languages provide primitives to group data items and create structured collections of data:
  – C++: array, struct.
  – python: list, tuple, dictionary.

Hiding details: abstractions
Different types of abstractions
Concept maps are hierarchical: why?

The computer systems stack

The computer systems stack

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

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

ADTs © Dept. CS, UPC

Variables used (4):
A, x, i, j
(only 3 modified)
Variables used (5):
A, x, left, right, i
(only 3 modified)

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 j = i;
  while (j > 0 && A[j-1] > A[j]) {
    A[j] = A[j-1];
    j--;
  }
  A[j] = x;
}

Each level has few items

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

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 (Δx, Δy).

The human brain has limitations: 4 things at once.

Modularity and abstraction are for designing large maintainable systems.
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).
- Why? See the next slides.

Class Point: a new implementation

```cpp
#include <iostream>
#include <cmath>

class Point {
public:
    double x, y;

    Point(double x, double y) : x(x), y(y) {
        // Constructor for (0,0)
    }

    double angle() const {
        return atan2(y, x);
    }

    double distance(const Point& p) const {
        return sqrt((x - p.x) * (x - p.x) + (y - p.y) * (y - p.y));
    }

    double distanceToPoint(const Point& p) const {
        return sqrt((x - p.x) * (x - p.x) + (y - p.y) * (y - p.y));
    }

    double distance(const Point& p) const {
        return sqrt((x - p.x) * (x - p.x) + (y - p.y) * (y - p.y));
    }

    double distance(double x, double y) const {
        return sqrt((x - x) * (x - x) + (y - y) * (y - y));
    }

    double getUR() const {
        return x;
    }

private:
    Point(const Point& p) : x(p.x), y(p.y) {
        // Copy constructor
    }

};
```

Public/private: let’s summarize

```cpp
Rectangle R2(8,4);
Point p(1,2);
Rectangle::Rectangle(double w, double h) :
    ll(Point(0,0)), ur(Point(w, h)), area(w*h) {} // Constructor (LL at the origin)

Rectangle R1 = Rectangle(4,3); // Creates a rectangle 4x3
Rectangle R3 = Rectangle(8,4); // Creates a rectangle 8x4
Rectangle R1.move(2,3); // Moves the rectangle
Rectangle R1.scale(1.2); // Scales the rectangle
```

What is *this?

- *this is a pointer (memory reference) to the object
- (pointers will be explained later)

Let us design the new type for Point

```cpp
Point::Point(double x, double y) :
    x(x), y(y) {
        // Constructor for (0,0)
    }
```

A new class: Rectangle

```cpp
Rectangle::Rectangle(double w, double h) :
    ll(Point(0,0)), ur(Point(w, h)) {
        // Constructor (LL at the origin)
    }
```

Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).

- We can modify the private section of the class without modifying the specification of the public methods.
- The private and public methods may need to be rewritten, but not the programs using the public interface.

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.get_radius() and p.get_angle() would still be valid.

Recommendation (reminder):

- All attributes should be private.
Rectangle R1(Point(2,3), Point(6,8));
double areaR1 = R1.area(); // areaR1 = 20
Rectangle R2(Point(3,5), 2, 4); // (is-of) R3 = R1 * R2;

Rectangle R2(Point(3,5), 2, 4);
R2 *= R1; // Intersection with R1

let us work with rectangles
Exercise: draw a picture of R1 and R2 after the execution of the previous code.

>>> a = Rational(4, ...
>>> a/b # uses the __repr__ method (see later)
Rational(-1/6)

the class rational

/// The denominator is zero
Documentation
# Similar for __sub__, __mul__, __truediv__

ADTs © Dept. CS, UPC 21

ADTs © Dept. CS, UPC 30

ADTs © Dept. CS, UPC 17

ADTs © Dept. CS, UPC 24

ADTs © Dept. CS, UPC 28

ADTs © Dept. CS, UPC 22

ADTs © Dept. CS, UPC 26

ADTs © Dept. CS, UPC 29

ADTs © Dept. CS, UPC 25

ADTs © Dept. CS, UPC 23

ADTs © Dept. CS, UPC 19

ADTs © Dept. CS, UPC 18

ADTs © Dept. CS, UPC 16

ADTs © Dept. CS, UPC 14

ADTs © Dept. CS, UPC 13

ADTs © Dept. CS, UPC 15

ADTs © Dept. CS, UPC 12

ADTs © Dept. CS, UPC 11

ADTs © Dept. CS, UPC 10

ADTs © Dept. CS, UPC 9

ADTs © Dept. CS, UPC 8

ADTs © Dept. CS, UPC 7

ADTs © Dept. CS, UPC 6

ADTs © Dept. CS, UPC 5

ADTs © Dept. CS, UPC 4

ADTs © Dept. CS, UPC 3

ADTs © Dept. CS, UPC 2

ADTs © Dept. CS, UPC 1

ADTs © Dept. CS, UPC 0

Using a module: example

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

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

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

Imports and renames the module.

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

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

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

Imports and renames the classes in the module.

Conclusions

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

EXERCISES

Implement the following methods for the class Rectangle:

```c
// 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. (This method is implemented using the rotate method.)
void flip(bool horizontally);

// Check whether point p is inside the rectangle
bool isPointInside(const Point& p) const;
```

Re-implement a class

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

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

```c
// Pre: n ≥ 0
// Returns the Fibonacci number of order n.
int fib(int n) { // Recursive solution
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2); }
```

Fibonacci: iterative version

```c
int fib(int n) { // ... f = f_i + f_i1;
    f_i = f_i1;
    f_i1 = f;
    return f_i;
}
```

Algorithm analysis

- Analysis based on the size of the input: |x| = n
- Only the best/average/worst cases are analyzed:
  - $C_{\text{worst}}(n) = \max x \in D \left( C(x) : |x| = n \right)$
  - $C_{\text{avg}}(n) = \sum x \in D \left( p(x) \cdot C(x) \right)$
  - $p(x)$: probability of selecting input x among all the inputs of size n.

Algorithm analysis: simplifications

- $C(x)$ must be independent of x for some value of x.
- $C(x)$ must be independent of y whenever $x \neq y$. (symmetry)
- $C(x)$ is non-decreasing function of x.

Asymptotic notation

- $f(n) = \mathcal{O}(g(n))$: $f(n)$ is $O$-dominated by $g(n)$
- $f(n) = \Omega(g(n))$: $g(n)$ is $\Omega$-dominated by $f(n)$
- $f(n) = \Theta(g(n))$: $f(n)$ is $\Theta$-dominated by $g(n)$

Examples

- $\mathcal{O}(n^2) = \{ f(n) : \exists c, n_0 \geq 0 \text{ s.t. } f(n) \leq cn^2 \text{ for } n \geq n_0 \}$
- $\Omega(n^2) = \{ f(n) : \exists c, n_0 \geq 0 \text{ s.t. } f(n) \geq cn^2 \text{ for } n \geq n_0 \}$
- $\Theta(n^2) = \{ f(n) : \exists c_1, c_2, n_0 \geq 0 \text{ s.t. } c_1n^2 \leq f(n) \leq c_2n^2 \text{ for } n \geq n_0 \}$
Let us assume that $L$ exists (may be $\infty$) such that:

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

If $L = 0$, then $f \in O(g)$.
If $0 < L < \infty$, then $f \in \Theta(g)$.
If $L = \infty$, then $f \in \Omega(g)$.

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

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

Algorithm Analysis © Dept. CS, UPC 17
Algorithm Analysis © Dept. CS, UPC 18
Algorithm Analysis © Dept. CS, UPC 19
Algorithm Analysis © Dept. CS, UPC 20
Algorithm Analysis © Dept. CS, UPC 21
Algorithm Analysis © Dept. CS, UPC 22
Algorithm Analysis © Dept. CS, UPC 23
Algorithm Analysis © Dept. CS, UPC 24
Algorithm Analysis © Dept. CS, UPC 25
Algorithm Analysis © Dept. CS, UPC 26
Algorithm Analysis © Dept. CS, UPC 27
Algorithm Analysis © Dept. CS, UPC 28
Algorithm Analysis © Dept. CS, UPC 29
Algorithm Analysis © Dept. CS, UPC 30

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

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

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

### The robot and the door in an infinite wall

#### Algorithm 1:
- Pick one direction and move until the door is found.

**Complexity:**
- If the direction is correct $\Rightarrow O(d)$.
- 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:**

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

#### 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:**

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

### 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 $d = 2^n$):

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

### 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 $S1$ is $O(f)$ and $S2$ is $O(g)$, then $S1;S2$ is $O(\max(f, g))$.

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

### For/While loops:
- Running time is at most the running time of the statements inside the loop times the number of iterations.

### Nested loops:
- Analyze inside out: running time of the statements inside the loops multiplied by the product of the sizes of the loops.

### Nested loops: examples
- For (int $i = 0; i < n; ++i$) for (int $j = 0; j < n; ++j$) DoSomething(); // $O(n^2)$
- For (int $i = 0; i < n; ++i$) for (int $j = 0; j < m; ++j$) DoSomething(); // $O(nm)$
- For (int $i = 0; i < n; ++i$) for (int $j = 0; j < m; ++i$) DoSomething(); // $O(n^2)$
- For (int $k = 0; k < p; ++k$) DoSomething(); // $O(n)$
Linear time: $O(n)$

Running time proportional to input size

// Compute the maximum of a vector // with n numbers

int m = a[0];
for (int i = 1; i < a.size(); ++i) {
if (a[i] > m) m = a[i];
}

Algorithm Analysis © Dept. CS, UPC

Linear time: $O(n)$

Other examples:

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

Logarithmic time: $O(\log n)$

• Logarithmic time is usually related to divide-and-conquer algorithms

• Examples:
  - Binary search
  - Calculating $x^n$
  - Calculating the $n$-th Fibonacci number

Example: recursive $x^y$

// Pre: $x \neq 0$, $y \geq 0$
// Returns $x^y$
int power(int x, int y) {
  if (y == 0) return 1;
  if (y%2 == 0) return power(x*x, y/2);
  return x*power(x*x, y/2);
}

// Assumption: each */% takes $O(1)$

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.

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

$T(x^y) \leq 4 + 4 + \cdots + 4 \quad \text{log}_y \text{ times} \quad \Rightarrow \quad O(\log y)$
Algorithm Analysis (II)

Jordi Cortadella and Jordi Petit
Department of Computer Science

**Selection Sort**
- Selection sort uses this invariant:

```java
int selection_sort(vector<elem>& v) {
    int last = v.size() - 1;
    for (int i = 0; i < last; ++i) {
        int k = i;
        for (int j = i + 1; j <= last; ++j) {
            if (v[j] < v[k]) k = j;
        }
        swap(v[i], v[k]);
    }
    return v;
}
```

**Insertion Sort**
- Insertion sort uses this invariant:

```java
void insertion_sort(vector<elem>& v) {
    for (int i = 1; i < v.size(); ++i) {
        elem x = v[i];
        int j = i - 1;
        while (j > 0 and v[j] > x) {
            v[j + 1] = v[j];
            --j;
        }
        v[j + 1] = x;
    }
}
```

**The Maximum Subsequence Sum Problem**
- The maximum subsequence sum is 0 if all integers are negative.
- Example: -2, 11, -3, 5, -2
- Answer: 20 (subsequence 11, 2, 5, 11)

**Convex Hull**
- Given (possibly negative) integers $A_1, A_2, \ldots, A_n$, find the maximum value of $\sum_{i=k}^{j} A_i$.
- The maximum subsequence sum is 0 if all integers are negative.

**Max Subsequence Sum: Divide & Conquer**
- The maximum subsequence sum can be found in one of three cases:
  - First half
  - Second half
  - Spanning both halves and crossing the middle

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

**Examples**
- Selection sort
- Insertion sort
- The Maximum Subsequence Sum Problem

**Observations:**
- If all integers are 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 $A$ are positive and all $A_j$ is negative:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

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

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

```java
int maxSubSumDivideAndConquer(const vector<int>& a, int left, int right) {
    // base cases
    if (left == right) return a[left];
    int maxLeft = maxSubSumDivideAndConquer(a, left, (left + right) / 2);
    int maxRight = maxSubSumDivideAndConquer(a, (left + right) / 2 + 1, right);
    return max(maxLeft, maxRight);
}
```
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.

\[
\text{Vertices: (1,3) (4,1) (7,3) (5,4) (6,7) (2,6) Edges: (1,3)-(4,1)-(7,3)-(5,4)-(6,7)-(2,6)-(1,3)}
\]

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

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

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 \( p_i \in \{p_2, p_3, ..., p_n\} \), calculate the angle \( \alpha \) between the lines \( p_1 \) and \( p_i \) and the \( x \) axis.

Sort the points \( \{p_2, p_3, ..., p_n\} \) according to their angles. In case of a tie, use distance to \( z \).

Clockwise and counter-clockwise

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

- Counter-clockwise (\( p_j \) at the left of \( [p_i, p_k] \))
  \[ q = (p + 1) \mod n; \]

- Clockwise (\( p_j \) at the right of \( [p_i, p_k] \))
  \[ a < b \]

// Returns true if \( p_j \) is at the left of \( [p_i, p_k] \)
// Convex hull: gift wrapping algorithm

\[ \text{Convex hull: Graham Scan} \]

- 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).

Assume that a partial path with \( m \) \( \geq 2 \) points \( \{p_1, p_2, ..., p_m\} \) is the number of points in the convex hull. In the worst case, \( m = n \).

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

For loops: analyze the cost of each code

EXERCISES

For loops: analyze the cost of each code

Prove the following equalities:

1. \[ \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \]
2. \[ \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \]
3. \[ \sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \]
For loops: analyze the cost of each code

O, Ω or Θ?

Primality

The Sieve of Eratosthenes

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 \( n \).

The Sieve of Eratosthenes

Algorithm Analysis © Dept. CS, UPC

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

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

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

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

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

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 \( p \)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

- The work is done in three places:
  - In partitioning the problem into subproblems
  - In solving the basic cases at the tail of the recursion
  - In merging the answers of the subproblems to obtain the solution of the original problem

---

**Product of polynomials: Divide & Conquer**

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

\[
\begin{align*}
P &: n/2 \quad P_R \\
Q &: Q_L \quad Q_R
\end{align*}
\]

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

\[
T(n) = 4 \cdot T(n/2) + O(n)
\]

---

**Product of complex numbers**

- The product of two complex numbers requires four multiplications:
  \[(a + bi)(c + di) = ac - bd + (bc + ad)i\]
  - Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: \(ac, bd\) and \((a+b)(c+d)\)
  - \(bc + ad = (a + b)(c + d) - ac - bd\)
  - A similar observation applies for polynomial multiplication.

---

**Useful reminders**

- **Sum of geometric series with ratio** \( r \):
  \[S = a + ar + ar^2 + ar^3 + \ldots + ar^{n-1} = \frac{a(1 - r^n)}{1 - r} = \frac{a}{1 - r} + \frac{r^n - 1}{r - 1} \]
  - Logarithms:
    \[\log_a n = \frac{\log_b n}{\log_b a}\]
    \[a^\log_a n = n\]

---

**Complexity analysis**

- **Function** \( \text{PolynomialProduct}(P, Q) \)
  - \( P \) and \( Q \) are vectors of coefficients.
  - \( R = P \times Q \)
  - \( \deg(R) = \deg(P)+\deg(Q) \)
  - \( R \times \text{vector with size}(P)\times\text{size}(Q) \) zeros;
  - for each \( P_i \): for each \( Q_j \):
    \[R_{ij} = R_{ij} + P_i \cdot Q_j\]
  - return \( R \)

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

---

**Divide & Conquer (II)**

Jordi Cortadella and Jordi Petit
Department of Computer Science

---

**Examples**

- **Algorithm Branch \( \epsilon \) Runtime equation**
  - Power \( x^y \): \( T(n) = T(n/2) + O(1) \)
  - Binary search: \( T(n) = T(n/2) + O(1) \)
  - Polynomial product: \( T(n) = 4 \cdot T(n/2) + O(n) \)
  - Polynomial product (Gauss): \( T(n) = 3 \cdot T(n/2) + O(n) \)

---

**Master theorem**

- **Typical pattern for Divide & Conquer algorithms:**
  - Split the problem into subproblems of size \( n/b \)
  - Solve each subproblem recursively
  - Combine the answers in \( O(n^\epsilon) \) time

- **Running time:** \( T(n) = a \cdot T(n/b) + O(n^\epsilon) \)

- **Master theorem:**
  \[
  T(n) = \begin{cases} 
  O(n^\epsilon) & \text{if } \epsilon > \log_b a \\
  O(n^{\log_b a}) & \text{if } \epsilon = \log_b a \\
  O(n^{\log_b a}) & \text{if } \epsilon < \log_b a 
  \end{cases}
  \]

---

**Master theorem: recursion tree**

- **Depth** \( \log_b n \)

---

**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:
  \[a^k \times O \left( \frac{n}{b^k} \right) = O(n^\epsilon) \times \left( \frac{n}{b^k} \right) \]
- As \( k \) goes from 0 (the root) to \( \log_b n \) (the leaves), these numbers form a geometric series with ratio \( a/b \).
  - We need to find the sum of such a series.
  \[
  T(n) = O(n^\epsilon) \times \left( \frac{1}{1 - a/b} \right) = a^b \times O(n^\epsilon) \times \frac{1}{b^k} \]

---

**Addition of polynomials of degree \( n \):**

- General pattern:
  \[
  \sum_{k=0}^{n-1} a_k x^k + \sum_{k=0}^{n-1} b_k x^k = \sum_{k=0}^{n-1} (a_k + b_k) x^k
  \]

---

**Convolution of polynomials:**

- 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:
  \[a^k \times O \left( \frac{n}{b^k} \right) = O(n^\epsilon) \times \left( \frac{n}{b^k} \right) \]
- As \( k \) goes from 0 (the root) to \( \log_b n \) (the leaves), these numbers form a geometric series with ratio \( a/b \).
  - We need to find the sum of such a series.
  \[
  T(n) = O(n^\epsilon) \times \left( \frac{1}{1 - a/b} \right) = a^b \times O(n^\epsilon) \times \frac{1}{b^k} \]
Master theorem: proof

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

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

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

Master theorem: visual proof

Master theorem: examples

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

- \( O(n^c) \) if \( a < b^c \)
- \( 0(n^c \log n) \) if \( a = b^c \)
- \( O(n^{\log_a b}) \) if \( a > b^c \)

<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(x) = T(x/2) + O(1) )</td>
<td>( O(\log x) )</td>
</tr>
<tr>
<td>Binary search</td>
<td>1</td>
<td>0</td>
<td>( T(x) = T(x/2) + O(1) )</td>
<td>( O(\log x) )</td>
</tr>
<tr>
<td>Merge sort</td>
<td>2</td>
<td>1</td>
<td>( T(n) = 2T(n/2) + O(1) )</td>
<td>( O(n \log n) )</td>
</tr>
<tr>
<td>Polynomial product</td>
<td>4</td>
<td>1</td>
<td>( T(n) = 4T(n/2) + O(1) )</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td>Polynomial product (Gauss)</td>
<td>3</td>
<td>1</td>
<td>( T(n) = 3T(n/2) + O(1) )</td>
<td>( O(n^{3/2}) )</td>
</tr>
</tbody>
</table>

Note: \( b = 2 \) for all the examples.
Quick sort

- The key step of quick sort is the partitioning algorithm.

Question: how to find a good pivot?

Function Qsort(A, left, right)

// A[left..right]: segment to be sorted.
// K is a break point such that part of the elements of A are greater than or equal to K and the other part are smaller than K.
// Output: segments A[left..mid) and A[mid+1..right).

\[
\begin{align*}
\text{if } & \text{ left < right then} \\
& \text{mid = Partition(A, left, right);} \\
& \text{Qsort(A, left, mid);} \\
& \text{Qsort(A, mid+1, right);} \\
\end{align*}
\]

Quick sort: hybrid approach

- 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 or the largest element in the vector.
  \[
  T(n) = \begin{cases} 
  n & n \\ 
  2 \cdot T(n/2) + O(n) & n/2
  \end{cases}
  \]

Quick sort: complexities analysis

- Let us assume that \( x_i \) is the \( i \)-th smallest element in the vector.
- Let us assume that each element has the same probability of being selected as pivot.
- The runtime if \( x_i \) is selected as pivot is:
  \[
  T(n) = n - 1 + T(i - 1) + T(n - i)
  \]

Quick sort: example

- 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.

Quick sort partition: example

- Observation: the partition may be unbalanced.

Quick sort partition: example

- Observation: the partition may be unbalanced.
Quick sort: complexity analysis summary

- **Runtime of quicksort:**
  \[ T(n) = O(n^2) \]
  \[ T(n) = \Omega(n \log n) \]
  \[ T_{avg}(n) = O(n \log n) \]

- **Be careful:** Some malicious patterns may increase the probability of the worst case runtime, e.g., when the list is already sorted.

- **Possible solution:** Use random pivots.

### The selection problem

- **Given a collection of \( N \) elements, find the \( k \)-th smallest element.**

- **Options:**
  - Sort a vector and select the \( k \)-th location: \( O(N \log N) \)
  - Read \( k \) elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only \( k \) elements are maintained in the vector. Complexity: \( O(kN) \).

### Quick sort with Hoare’s partition

- **Function:**
  \[ \text{Qsort}(A, \text{left}, \text{right}) \]

- **// A[left..right]: segment to be sorted**

- **if \( \text{left} < \text{right} \) then**
  - \( \text{mid} = \text{HoarePartition}(A, \text{left}, \text{right}) \)
  - \( \text{Qsort}(A, \text{left}, \text{mid}) \)
  - \( \text{Qsort}(A, \text{mid}+1, \text{right}) \)

- **// Returns the element at location \( i \) assuming**

- **// A[left..right] would be sorted in ascending order.**

- **// Post: The elements of \( A \) have changed their locations.**

- **function Qselect(A, left, right, \( k \))**

- **if \( \text{left} \leq k \leq \text{right} \) then return A[\( k \)];**

- **// We only need to sort one half of \( A \)**

- **if \( k \leq \text{mid} \) then return Qselect(A, left, mid, k); else return Qselect(A, mid+1, right, k);**

### Quick select with Hoare’s partition

- **// A\[left..right\]: segment to be sorted**

- **// Returns the element at location \( i \) assuming**

- **// A[left..right] would be sorted in ascending order.**

- **// Post: The elements of \( A \) have changed their locations.**

- **Function:**
  \[ \text{Qselect}(A, \text{left}, \text{right}, \text{mid}, k) \]

- **// We only need to sort one half of \( A \)**

- **if \( k \leq \text{mid} \) then return \( A[\text{mid}, \text{left}, \text{right}, \text{mid}] \);**

### The Closest-Points problem

- **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 \( O(n \log n) \) solution!

### The Closest-Points problem

- **We can assume that the points are sorted by the \( y \)-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 on the left, both on the right or one on the left and the other on the right (center).**

- **The left and right pairs are easy to find (recursively). How about the pairs in the center?**

### The Closest-Points problem

- **Let \( \delta = \min(\delta_L, \delta_R) \). We only need to compute \( \delta_C \) if it improves \( \delta \).**

- **We can define a strip around the center with distance \( \delta \) at the left and right. If \( \delta_C \) improves \( \delta \), then the points must be within the strip.**

- **In the worst case, all points can still reside in the strip.**

- **But how many points do we really have to consider?**

#### Subdivide

- **Sort the points according to their \( x \)-coordinates.**

- **Divide the set into two equal-sized parts.**

- **Compute the min distance at each part (reursively).**

- **Let \( \delta \) be the minimal of the two minimal distances.**

- **Eliminate points that are farther than \( \delta \) from the separation line.**

- **Sort the remaining points according to their \( y \)-coordinates.**

- **Scan the remaining points in the \( y \) order and compute the distances of each point to its 7 neighbors.**

### Muster: recursion tree

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

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

- **Selection sort (recursive version):**

  - Select the min element and move it to the first location

  - Sort the remaining elements

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

  \( (a = c = 1) \)

  Thus, \( T(n) = O(n^2) \)
Let $T[i, j]$ be a vector with $n = j - 1$ 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 $T[i, k - 1]$, $T[k, i]$ and $T[i + 1, j]$, where $k = i + \lfloor n/3 \rfloor$ and $l = j - \lfloor n/3 \rfloor$. The algorithm recursively sorts $T[i, k]$ and $T[i + 1, j]$, and finally sorts $T[i + k]$.

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

### Logarithmic identities

\[
\begin{align*}
\ln a^n &= \ln b^n = n \ln a \\
\ln(ab) &= \ln a + \ln b \\
\ln\left(\frac{a}{b}\right) &= \ln a - \ln b \\
\ln a^x &= x \ln a \\
\ln a &= \frac{\ln a}{\ln 10} \\
\ln e^x &= x \\
\ln\left(\frac{x - y}{x + y}\right) &= \frac{1}{2}\ln\left(\frac{x - e^{-y}}{x + e^{-y}}\right)
\end{align*}
\]

\[
y = \lim_{n \to \infty} -\ln n + \sum_{k=1}^{n} \frac{1}{k} \\
y \approx 0.5772 \ldots \text{ (Euler-Mascheroni constant)}
\]

### Full-history recurrence relation

\[
T(n) = n - 1 + \sum_{k=1}^{n} \frac{1}{k}
\]

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

\[
\begin{align*}
T(0) &= 0 \\
T(n+1) &= (n+1)T(n) + (n+1)
\end{align*}
\]

### The skyline problem

*Input:* (x1, y1, z1, w1), ... (xN, yN, zN, wN) (numbers in boldface represent heights)

*Output:* 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 $n/2$ 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?

### Memory Management

**Jordi Cortadella and Jordi Petit**

**Department of Computer Science**

**Data structures usually have a descriptor (fixed size) and a storage area (variable size).**

- **Aliasing long names**
- **Address of a variable (reference operator &):** In some cases, it might be convenient to obtain the address of a variable during runtime.

---

**References:**

- **auto & r = &*[getIndex(a, b)].val;**
- **... .defineValue();** **// avoids a long name for r**
- **... // avoids a copy of a large data structure**
- **bigVector & V = myMatrix.getMul();**
- **... // an alternative for the following loop:**
- **// for (int i = 0; i < a.size(); ++i) aux[i];**
- **for (auto x: arr) aux;** **// does not work (why?)**
- **for (auto & arr: ary) aux;** **// it works!**

---

**Pointers vs. references:**

- **A pointer holds the memory address of an object. 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**

(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), Chapter 3.4.**

---

**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;`

---

**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

---

**The Vector class**

```cpp
template<typename Object>
class Vector {
public:
private:

};
```

- **What is a class template?**
  - A generic abstract class that can handle various datatypes.
  - The template parameters determine the genericity of the class.
- **Example of declarations:**
  - `Vector<int> V;`
  - `Vector<polygon> vp;`
  - `Vector<Vector<double>> M;`
  - **A Vector may allocate more memory than needed (size vs. capacity).**
  - The memory must be reallocated when there is no enough capacity in the storage area.
  - The pointer stores the base memory address (location of objects). Pointers can be allocated to free memory blocks via new/delete operators.

---

**Access to members through a pointer**

```cpp
int n = vp->size();
```

---

**Allocating/deallocating arrays**

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

---

**The memory**

### Dynamic Object Creation/Destruction

- **The new operator returns a pointer to a newly created object:**
  - `myClass* c = new myClass( );`
  - `myClass* c = new myClass( ); // C++11`
  - `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`
The Vector class

Public:
- declares the size of the vector
- public to get and set the capacity
- public to resize and to add or remove elements

Dangling references and memory leaks

- A and B point at the same object!
- Possible memory leak (unreferenced memory space)

Points and references to dynamic data

- myClass *a = new myClass;
  myClass *b = new myClass;
- We have allocated space for two objects
- 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

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.

C: parameters by reference

- "a" is received by value and "b" by reference
- int(int a, int b) { "b = a; return a; }"Int main() { ...}
- Use vguard (valgrind.org) to detect memory leaks.
- Remember: no pointers = no memory leaks.

C: arrays

- int a[100]; // an array of 100 int's
- double w[double*][int] = 0; // a matrix of size 30x40
- int c[4][]; // also int c: an array of unknown size
- "a" is a string (array of unknown size)

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.

(continued on next page...)
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).
int main() {
    vector<double> a;
    double s = sum(a.data(), a.size());
    return 0;
}

They should always hold in a bug

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.

The program should not halt unexpectedly.

• 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.

Constructors

```cpp
class Point {
    int x, y;

    public:
    Point(const Point &p) {
        x = p.x; y = p.y;
    }

    int getX() { return x; }
    int getY() { return y; }
};
```

```cpp
class A {
    int id;

    public:
    A() : id(++c) { id++; cout << "A\n"; }

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

They should give meaningful error messages to the users.

Languages are designed to make memory management transparent

Errors handling:
• Useful to detect internal errors.

Containers: Stacks

EXERCISES

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.

```cpp
// using the 'data()' method (C++11).
// a
// Any C++ vector is an object that
// contains
// this private array. This array can be accessed
// using the 'data()' method (C++11).
```

Languages are designed to make memory management transparent

Memory management © Dept. CS, UPC 36

• 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
to 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.

EXERCISES

```cpp
double y = sqrt(x);
```

```cpp
f(const A &x, A y) { ...
```

The internal array
of the vector

Functions using the internal array of a vector should NEVER resize the array!

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

Memory management © Dept. CS, UPC 37

• Memory management
• Containers: Stacks
• Memory management
• Memory management
• Containers: Stacks

```cpp
int c = 2; // Global variable
```

```cpp
int main() {
    Point p1(10);
    Point p2 = p1;
    cout << p1.getX() << p2.getX() << endl;
    return 0;
}
```

int main() {
    Point p1(10);
    Point p2 = p1;
    cout << "\n" << p2.getX() << endl;
    return 0;
}

What is the output of this program?

```
void f(const A &x, A y) {
    A x = x; A w;
} int main() {
    A v1, v2, v3; A w = v1;
    v2 = new A();
    v2 = "\n";
    delete v2;
    A v5;
}
```

```cpp
class A {
    int id;

    public:
    A() : id(++c) { id++; cout << "A\n"; }

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

```cpp
int c = 2; // Global variable
```

```cpp
int main() {
    Point p1(10);
    Point p2 = p1;
    cout << p1.getX() << p2.getX() << endl;
    return 0;
}
```

List with pointers

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

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

```cpp
int c = 2; // Global variable
```

```cpp
int main() {
    Point p1(10);
    Point p2 = p1;
    cout << p1.getX() << p2.getX() << endl;
    return 0;
}
```

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

Design the function `BestStudent` with the following specification:

L points at the first student of the list. `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 "No student,
their memory management (new/delete) inside the class in such a way
that the user of the class does not need to “see” the pointers.

EXERCISES

```cpp
int c = 2; // Global variable
```

```cpp
int main() {
    Point p1(10);
    Point p2 = p1;
    cout << p1.getX() << p2.getX() << endl;
    return 0;
}
```
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
- Balancing symbols: check for syntax errors when expressions have opening/closing symbols, e.g., [ ] {}.
  - Correct: [{}[]{}[]{}{}
  - Incorrect: [{}[]]{
- Algorithm (linear): read all chars until end of file. For each char, do the following:
  - If the char is a closing char or stack is empty, push error.
  - If no closing char and stack is not empty, pop the top symbol from the stack and check they match. If not, push error.
  - At the end of the file, the stack is empty.
- Exercise: implement and try the above examples.

Evaluation of postfix expressions: example
- This is an infix expression. What’s its value? 42 or 144?
  - $8 \times 3 + 10 + 2 + 4$.
- It depends on the operator precedence. For scientific calculators, $\times$ has precedence over $+$. 
- Postfix (reverse Polish notation) has no ambiguity:
  - $8 \times 3 + 10 + 2 + 4$.
- Postfix expressions can be evaluated using a stack:
  - Each time an operand is read, it is pushed onto the stack
  - Each time an operator is read, the two top values are popped and operated. The result is pushed onto the stack

Interleaved push/pop operations
Suppose that an intermixed sequence of push and pop operations are performed. The push pushes the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?
- $a + b + c + (d + e + f) + g$
- $a \times b \times c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$

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

EXERCISES
- Exercise: 
  a) $4321098765$
  b) $4687532901$
  c) $2567489310$
  d) $4321056789$

Middle element of a stack
Suppose that an intermixed sequence of push and pop operations are performed. The push pushes the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$
- $a + b + c + (d + e + f) + g$

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

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

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

Queue

Queue\text{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()) {
    int elem = Q.front();
    cout << elem « endl; // Get the first element
    Q.pop(); // Delete the element
}

Queue usage

Queue: copy (private)

/** Copies a queue. */
void copy(const Queue& Q) {
    n = Q.n;
    if (n == 0) {
        first = last = nullptr;
    }
    p2->next = nullptr;
    last = p2;
}

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(T x) {
    Node* p = new Node(x, nullptr);
    if (n == n) first = last = p;
    else last = last->next = p;
}

/** Removes the first element. */
T front() const {
    if (Q.empty()) assert(false); // Pre: the queue is not empty.
    T x = first->elem;
    first = first->next;
    return x;
}

/** Removes the first element. */
T pop() {
    if (Q.empty()) assert(false); // Pre: the queue is not empty.
    T x = Q.front();
    first = Q.first;
    return x;
}

Queue: constructors and destructor

/** Default constructor: an empty queue. */
Queue() : first(nullptr), last(nullptr), n(0) { }

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

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

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

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.
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.

A list can be considered as a sequence of elements with one or several cursors (iterators) pointing at internal elements.

Consider the design of a variable-size queue using a circular buffer. Discuss how the implementation should be modified.

**EXERCISES**

**Reverse and Josephus**
- 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.

- Design the reverse() method 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:

**Merge sort**
- Design the method 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 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... 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?

Graph representation: adjacency matrix

A graph can be represented by an $|V| \times |V|$ matrix with:

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

For undirected graphs, the matrix is symmetric.

Graph definition

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

A significant part of the material used in this chapter has been inspired by the book: Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani, Algorithms, McGraw-Hill, 2008. [DPV2008]

Several examples, figures, and exercises are taken from the book.

Dense and sparse graphs

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

Adjacency matrix vs. adjacency list

Space:

- Adjacency matrix is $O(|V|^2)$
- Adjacency list is $O(|E|)$

Checking the presence of a particular edge $(u, v)$:

- Adjacency matrix: constant time
- Adjacency list: traverse $v$’s adjacency list

Which one to use?

- For dense graphs $\rightarrow$ adjacency matrix
- For sparse graphs $\rightarrow$ adjacency list

For many algorithms, traversing the adjacency list is not a problem, since they require to iterate through all neighbors of each vertex. For sparse graphs, the adjacency lists are usually short (can be traversed in constant time)
// Declaration of a graph that stores
// a string (name) for each vertex
Graph<string> G;
// ... Connectivity © Dept. CS, UPC

A B C D
E F G H
I J K L

FA
B E
I
J
C
D
H
G L
K

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

Graph DFS forest

Depth-first search

DFS example

Connectivity

Finding the nodes reachable from another node

Function explore(G, v):
visited(v) = true
for each edge (v, u) ∈ E:
if not visited(u): explore(G, u)

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

An undirected graph is connected if there is a path between any pair of vertices.

A disconnected graph has disjoint connected components.

Example: this graph has 3 connected components:

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

Function explore(G, v, cc):
visited(v, cc) = true
for each edge (v, u) ∈ E:
if not visited(u, cc): explore(G, u, cc)

Function ConComp(G):
Input: G = (V, E) is a graph
Output: ccnum = number of CCs in G
for all v ∈ V:
ccnum[v] = 0; CC cleaned
for all v ∈ V:
if ccnum[v] = 0:
new CC starts
exploreG(v, v, cc); cc = cc + 1

The outer loop of ConComp determines the number of CCs.
The variable ccnum also plays the role of visited[cc].

Graph Connectivity © Dept. CS, UPC

Graph DFS in directed graphs: types of edges

All adjacent edges are scanned
The stack can be rewound and return to the previous junction.
Note: the stack can be simulated with recursion.

Graph DFS forest

Find the nodes reachable from another node

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

Function explore(G, v, cc):
visited(v, cc) = true
previsit(v) = clock
postvisit(v) = clock + 1
for each edge (v, u) ∈ E:
if not visited(u, cc):
explore(G, u, cc)

Example of pre/postvisit orderings

Reachability: exploring a maze

To explore a labyrinth we need a ball of string and a piece of chalk:
• The chalk prevents looping, by marking the visited junctions.
• The string allows you to go back to the starting place and visit routes that were not previously explored.

Finding the nodes reachable from another node

Revisiting the explore function

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 v is the lifetime of explore(v) in the stack (LEFD).

Function explore(G, v):
visited(v) = true
previsit(v) = clock
postvisit(v) = clock + 1
for each edge (v, u) ∈ E:
if not visited(u): explore(G, u)

Example of pre/postvisit orderings

DFS example

Connectivity

DFS in directed graphs: types of edges

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

Reachability: exploring a maze

Running time

Between any pair of vertices.

Graph usage: example

Graph implementation

Graph implementation

Graphs: Connectivity © Dept. CS, UPC

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

 DFS forest

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

Function explore(G, v, cc):
Input: G = (V, E) is a graph
Input: cc is a CC number
Output: conncomp[v] = number of CCs in V
for all v ∈ V:
conncomp[v] = 0; CC cleaned
cc = 1; Identifier of the first CC
for all v ∈ V:
if ccnum[v] = 0:
new CC starts
exploreG(v, v, cc); cc = cc + 1

A fixed amount of work (pre/postvisit) is missed.
All adjacent edges are scanned.
The stack can be rewound and return to the previous junction.
The variable ccnum also plays the role of visited[cc].

Reversing the depth

DFS traverses the entire graph.

DFS example

Connectivity

DFS in directed graphs: types of edges

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

Graph Connectivity © Dept. CS, UPC

Graph DFS in directed graphs: types of edges

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

Reachability: exploring a maze

To explore a labyrinth we need a ball of string and a piece of chalk:
• The chalk prevents looping, by marking the visited junctions.
• The string allows you to go back to the starting place and visit routes that were not previously explored.

Finding the nodes reachable from another node

Revisiting the explore function

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 v is the lifetime of explore(v) in the stack (LEFD).

Function explore(G, v, cc):
visited(v, cc) = true
previsit(v) = clock
postvisit(v) = clock + 1
for each edge (v, u) ∈ E:
if not visited(u, cc):
explore(G, u, cc)

Example of pre/postvisit orderings

DFS example

Connectivity

DFS in directed graphs: types of edges

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

Reachability: exploring a maze

To explore a labyrinth we need a ball of string and a piece of chalk:
• The chalk prevents looping, by marking the visited junctions.
• The string allows you to go back to the starting place and visit routes that were not previously explored.

Finding the nodes reachable from another node

Revisiting the explore function

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 v is the lifetime of explore(v) in the stack (LEFD).

Function explore(G, v, cc):
visited(v, cc) = true
previsit(v) = clock
postvisit(v) = clock + 1
for each edge (v, u) ∈ E:
if not visited(u, cc):
explore(G, u, cc)

Example of pre/postvisit orderings

DFS example

Connectivity

DFS in directed graphs: types of edges

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

Reachability: exploring a maze

To explore a labyrinth we need a ball of string and a piece of chalk:
• The chalk prevents looping, by marking the visited junctions.
• The string allows you to go back to the starting place and visit routes that were not previously explored.

Finding the nodes reachable from another node

Revisiting the explore function

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 v is the lifetime of explore(v) in the stack (LEFD).

Function explore(G, v, cc):
visited(v, cc) = true
previsit(v) = clock
postvisit(v) = clock + 1
for each edge (v, u) ∈ E:
if not visited(u, cc):
explore(G, u, cc)

Example of pre/postvisit orderings

DFS example

Connectivity

DFS in directed graphs: types of edges

• Tree edges: those in the DFS forest.
• Forward edges: lead to a reachable descendant in the DFS tree.
• Back edges: lead to an ancestor in the DFS tree.
• Cross edges: lead to neither descendant nor ancestor.
Cycles in graphs

A cycle is a circular path:

- Example 1: $B \rightarrow E \rightarrow F \rightarrow B$
- Example 2: $C \rightarrow D \rightarrow C$

Properties: A directed graph has a cycle if 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 $u$ to $v$ in the search tree.
- Let us consider a cycle $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$. Let us assume that $v_0$ is the first discovered vertex (lowest pre number). All the other $v_i$ vertices on the cycle are reachable from $v_0$, and will be its descendants in the DFS tree. The edge $v_{k-1} \rightarrow v_0$ leads from a vertex to its ancestor and is thus a back edge.

Crawling the Web

- Cyclic graphs cannot be linearized.
- All DAGs can be linearized. How? Decreasing order of the post numbers.
- The only edges: $(u,v)$ with post($u$) < post($v$) are back edges (do not exist in DAG).

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.

Crawling is done in parallel (many computers at the same time).

Topological sort

- Function: explore($G, v$)
  - visit(v) = true
  - postvisit(v) = true
  - for each (v, w) ∈ E:
    - if not visit(w):
      - explore($G, w$)

Another algorithm:
- Find a source vertex, write it, and delete it (mark) from the graph.
- Repeat until the graph is empty.

It can be executed in linear time. How?

Topological sort

1. Intuition for the algorithm:
   - Find a vertex located in a sink SCC
   - Extract the SCC
2. To be solved:
   - How to find a vertex in a sink SCC?
   - What to do after extracting the SCC?
3. 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 $C'$.
4. Property: The vertex with the highest DFS post number lies in a source SCC.
5. Property: Every directed graph can be represented as a meta graph, where each meta node represents a strongly connected component.

Properties of DFS and SCCs

- DFS and Comps 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

<table>
<thead>
<tr>
<th>Streets in Computopia (from [DPV2008])</th>
<th>Topological ordering (from [DPV2008])</th>
<th>SCC (from [DPV2008])</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>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. For each graph, answer the following questions:</td>
<td>Run the SCC algorithm on the two graphs. When doing DFS of 𝐺, whenever there is a choice of vertices to explore, always pick the one that is alphabetically first. For each graph, answer the following questions:</td>
</tr>
</tbody>
</table>
| 1. What are the sources and sinks of the graph? | 1. In what order are the SCCs found? | 2. Which are source SCCs and which are sink SCCs?
2. What are the sources and sinks of the graph? | 2. Which are source SCCs and which are sink SCCs? | 3. Draw the meta-graph (each meta-node is an SCC of 𝐺).
3. What topological order is found by the algorithm? | 3. Draw the meta-graph (each meta-node is an SCC of 𝐺).
4. How many topological orderings does this graph have? | 4. What is the minimum number of edges you must add to the graph to make it strongly connected? |

DFS: stack overflow

- DFS can be implemented with an elegant recursive algorithm, but it may experiment stack overflow problems. Explain why.
- Design an iterative version of DFS.
- Challenge: do not to search in stack overflow.

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.

Pouring water (from [DPV2008])

We have three containers whose sizes are 10 pints, 7 pints and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. We are allowed one type of operation: pouring the contents of one container into another, stopping only when the source container is empty or the destination container is full. We want to know if there is a sequence of pourings that leaves exactly 2 pints in the 4-pint container.

a) Model this as a graph problem: give a precise definition of the graph involved and state the specific question about this graph that needs to be answered.

b) What algorithm should be applied to solve the problem?

c) Give a sequence of pourings, if it exists, or prove that it does not exist any sequence.

Hint: A vertex of the graph can be represented by a triple of integers.
Containers: Priority Queues
Jordi Cortadella and Jordi Petit
Department of Computer Science

A priority queue is a queue in which each element has a priority.
Elements with higher priority are served before elements with lower priority.

It can be implemented as a vector or a linked list. For a
queue with n elements:
- Insertion is \(O(n)\).
- Extraction is \(O(1)\).

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

A binary heap is a complete binary tree.

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

It can be implemented as a vector or a linked list. For a
heap sorted tree:
- Modify the value of one element in the middle of the heap.
- Remove one element:
  - Set value to \(-\infty\), bubble up and remove min element.

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

### Binary Heap: complexity

- Bubble up/down operations do at most \(h\) swaps, where \(h\) is the height of the tree and
  \[ h = \lfloor \log_2 N \rfloor \]

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

- 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 \(\Rightarrow\) bubble up.
  - If increased \(\Rightarrow\) bubble down.

- Remove one element:
  - Set value to \(-\infty\), bubble up and remove min element.

### Binary Heap: insert \(14\)

1. Insert in the last location
2. \(\ldots\) and bubble up ...
3. done !

### Binary Heap: remove min

1. Insert in the last location
2. Extract the min element and move the last one
to the root of the heap
3. \(\ldots\) and bubble down ...
4. done !

### Building a heap from a set of elements

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

- Can it be done more efficiently?

### Binary Heap: other operations

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

### Containers: Priority Queues

- \(O(\log N)\) complexity:
  - Insertion is \(O(\log N)\)
  - Removing the min element is \(O(\log N)\)

### Binary Heap: implementation

```cpp
// Constructor from a collection of items
BinaryHeap(const std::vector<int>& items)
    .push_back(items); // v is the vector holding the elements
for (auto& v : items)
    BinaryHeap(v); // do heapify()
for (int i = size()/2; i > 0; i--) bubble_down();
```

### Exercise: insert/remove element

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

![Binary Heap Diagram](image)

Execute the following sequence of operations

- insert(8); remove_min(); insert(9); insert(8); remove_min();

and draw the tree after the execution of each operation.
Consider the binary heap of integer keys implemented by the following vector:

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

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

The $k$-th element of $n$ sorted vectors.

Let us consider $n$ vectors sorted in ascending order.

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

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

- BFS visits vertices layer by layer: 0, 1, 2, ..., 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

**Input:** A graph G and a source vertex s.
**Output:** A vertex v ∈ V: reached[v] = u is reachable from s.

```
Function BFS(G, s):
    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)
    return
```

Reachability: BFS vs. DFS

- BFS visits vertices layer by layer: 0, 1, 2, ..., d.
- DFS visits vertices in a depth-first order, exploring deeper paths first.
- Central data structure: a stack.

Dijkstra's algorithm: invariant

**Input:** A graph G and a source vertex s.
**Output:** The shortest path from s to every other vertex in G.

```
Function Dijkstra(G, s):
    for all u ∈ V: dist[u] = ∞
    dist[s] = 0
    Q = V // Queue containing all vertices
    S = ∅ // Set of completed vertices
    while not Q.empty():
        u = Q.min()
        S.push_back(u)
        for all (u, v) ∈ E:
            if dist[v] > dist[u] + G[u][v]:
                dist[v] = dist[u] + G[u][v]
                Q.push(v)
    return
```

Graphs: Shortest paths

- Distance in a graph: Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.
- Breadth-first search: Similar to a wave propagation.
- Reusing BFS: Graphs: Shortest paths © Dept. CS, UPC

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

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

Graphs: Shortest paths

- Distance between two nodes: length of the shortest path between them.
- Reachability: BFS vs. DFS
- Weights on edges

Example

- Data: Queue
- Queue: A0 A1 A2 B1 B2 C1 C2 C3 D1 D2
- Data: Queue
- Queue: A0 A1 A2 A3 B1 B2 B3 C1 C2 D1 D2
**Dijkstra’s algorithm: complexity**

- \( |V| \) times: \( Q \) makequeue, \( Q \) while loop; \( |V| \) Decreasekey steps.
- \( |E| \) times: \( Q \) deletemin and relevant for-loops.

**Why Dijkstra’s works**

- A tree of open paths with distances 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.

**Graphs with negative edges**

- Dijkstra’s algorithm does not work:
  - Dijkstra would say that the shortest path \( S \rightarrow A \) has length 5.
  - Dijkstra is based on a safe update each time an edge \((u, v)\) is treated:
    \( \text{dist}(v) = \min(\text{dist}(v), \text{dist}(u) + l(u, v)) \)
  - Possible solution: add a constant weight to all edges, make them positive, and apply Dijkstra.

**Negative cycles**

- What is the shortest distance between \( S \) and \( A \)?

**Bellman-Ford algorithm**

- Bellman-Ford does not work as it assumes that the shortest path will not have more than \( |V| - 1 \) edges.

**Shortest paths in DAGs**

- DAG’s property:
  - In any path of a DAG, the vertices appear in increasing topological order.
  - Any sequence of updates that preserves the topological order will compute distances correctly.
  - Only one round visiting the edges in topological order is sufficient: \( O(|V| + |E|) \).
- How to calculate the longest paths?
  - Negate the edge lengths and compute the shortest paths.
  - Alternatively: update with max (instead of min).

**DAG shortest paths algorithm**

- **Input**: DAG \( G(V,E) \), source vertex \( s \), edge lengths \( l(u,v) \).
- **Output**: \( \text{dist}(v) \) has the distance from \( s \), and \( \text{prev}(v) \) has the predecessor in the tree.

**Single-source shortest paths**

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

Run Dijkstra’s algorithm starting at node A:
- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

Run Bellman-Ford algorithm starting at node S:
- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

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

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

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 \( \{B_1, B_2, \ldots, B_n\} \).

Describe an efficient algorithm to determine the longest sequence \( \{B_1, B_2, \ldots, B_k\} \) of boxes such that \( B_i \) nests within \( B_{i+1} \) for \( j = 1, 2, \ldots, k-1 \). Express the running time of your algorithm in terms of \( n \) and \( d \).

### Minimum Spanning Trees

**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 the total weight \( \sum w_e \).

**Properties of trees**
- **Definition**: A tree is an undirected graph that is connected and acyclic.
- **Property**: Any connected, undirected graph \( G = (V, E) \) has \( |E| = |V| - 1 \) edges.
- **Property**: A tree on \( n \) nodes has \( n - 1 \) edges.
- **Property**: Any connected, undirected graph \( G = (V, E) \) with \( |E| = |V| - 1 \) is a tree.
- **Property**: Any undirected graph is a tree if there is a unique path between any pair of nodes.
- **Property**: If there would be two paths between two nodes, the union of the paths would contain a cycle.

**Greedy algorithm**
- Repeatedly add the next lightest edge that does not produce a cycle.

**Prim’s algorithm**
- **Property**: Any minimum weight edge \( e \) of \( G \) has \( e \in E' \).
- **Property**: If \( e = (u, v) \) is the edge added, \( u \in S \) and \( v \notin S \).
- **Property**: The lightest edge with exactly one endpoint in \( S \) is added.

**Kruskal’s algorithm**
- **Property**: The lightest edge between two trees is added.

**Prim’s algorithm**
- **Function**: Prim(G, w)
  - **Input**: A connected undirected graph \( G(V, E) \) with edge weights \( w_e \).
  - **Output**: An MST defined by the vector \( prev \).
  - **Invariant**: A set of nodes \( \{\} \) is in the tree.
  - **Progress**: The lightest edge with exactly one endpoint in \( S \) is added.

**Kruskal’s algorithm**
- **Function**: Kruskal(G, w)
  - **Input**: A connected undirected graph \( G(V, E) \) with edge weights \( w_e \).
  - **Output**: An MST defined by the edges in \( X \).
  - **X = \{\}
    - sort the edges in \( E \) by weight for all \( (u, v) \in E \), in ascending order of weight:
      - if \( (u, v) \in E \) and \( u \in S \) but \( v \notin S \):
        - \( X = X \cup (u, v) \)

**Disjoint sets**
- A data structure to store a collection of disjoint sets.

**Operations**
- **function makeSet(x)**: creates a singleton set containing just \( x \).
- **function find(x)**: returns the identifier of the set containing \( x \).
- **function union(x, y)**: merges the sets containing \( x \) and \( y \).

**Kruskal’s algorithm uses disjoint sets and calls**
- **makeSet**:
  - \( |V| \)
- **find**:
  - \( |E'| \)
- **union**:
  - \( |V| - 1 \) times

---

**Properties of Minimum Spanning Trees**
- A tree is an undirected graph that is connected and acyclic.
- Any connected, undirected graph \( G = (V, E) \) with edge weights \( w_e \), find a tree \( T = (V, E') \), with \( E' \subseteq E \), such that:
  - The 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

**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 \( (S, V-S) \). Since \( T \) is an MST, then the weights must be equal.

**Minimum Spanning Trees and Maximum Flows**
- A tree on \( n \) nodes has \( n - 1 \) edges.
- Any scheme like this works (because of the properties of trees):
  - **Invariant**: A set of nodes \( \{\} \) is in the tree.
  - **Progress**: The lightest edge with exactly one endpoint in \( S \) is added.

**Kruskal’s algorithm**
- **Function**: Kruskal(G, w)
  - **Input**: A connected undirected graph \( G(V, E) \) with edge weights \( w_e \).
  - **Output**: An MST defined by the edges in \( X \).
  - **X = \{\}
    - sort the edges in \( E \) by weight for all \( (u, v) \in E \), in ascending order of weight:
      - if \( (u, v) \in E \) and \( u \in S \) but \( v \notin S \):
        - \( X = X \cup (u, v) \)

**Disjoint sets**
- The nodes are organized as a set of trees. Each tree represents a set.
- Each node has two attributes:
  - parent \( p \): 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 tree.

**Prim's algorithm**
- **Function**: Prim(G, w)
  - **Input**: A connected undirected graph \( G(V, E) \) with edge weights \( w_e \).
  - **Output**: An MST defined by the vector \( prev \).
  - **Invariant**: A set of nodes \( \{\} \) is in the tree.
  - **Progress**: The lightest edge with exactly one endpoint in \( S \) is added.
In the residual graph:

- Directed graph $H_1, J$
- For all nodes $3/28$
- The slack of $e = 27$
- Rank $(30)$
- $D_2 = \Rightarrow$
- Some $= 1$
- $V = \pi 2$
- One Image processing $E_1$
- Ford $1$
- $r = x$
- Makeset $s$
- $x = t$
- $E$ direction $+ c$ for all $p$
- $B = 2$
- Two special nodes $slack = cost: 0$
- $f = x$
- has at least $\log 17$
- $23$
- nodes in its tree.
- Therefore, all trees have height $\log n$
- Distributed computing $D$
- is $r D_1 F$
- Computer networks $C$
- Any root node of rank $D$
- $r D_1 F$
- $\Rightarrow$
- Ford-Fulkerson algorithm

Function Ford-Fulkerson($G, s, t$):
// Input: A directed graph $G = (V, E)$ with edge capacities $c_e$.
// $a$ and $t$ are the source and target of the flow.
// Output: A flow $f$ that maximizes the size of the flow.
// For each $(v, u) \in E$, $f(v, u)$ represents its flow.
// 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.

```
function Ford-Fulkerson(G, s, t):
  // Input: A directed graph G=(V,E) with edge capacities c_e.
  // a and t are the source and target of the flow.
  // Output: A flow f that maximizes the size of the flow.
  // For each (v, u) in E, f(v, u) represents its flow.
  // 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.
```
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: $O(|V|^2)$ with $|C| = 1$ being the largest capacity of an edge (if only integral values are used).
- By selecting the path with lowest edge-capacity (using BFS), the number of iterations can be reduced.

Ford-Fulkerson algorithm: complexity

Bipartite matching

Max-flow problem

Min-cut algorithm

Bipartite matching

Minimum Spanning Trees

EXERCISES

Contagious disease

Blood transfusion

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.
Print a tree (postorder traversal)

```cpp
/** Prints a tree (postorder) indented according to depth.
* Pre: The tree is not empty. */
void print(const Tree& T, int depth) {
    cout << string(2*depth, ' ') << T.name << endl;
    print(T.left, depth + 1);
    print(T.right, depth + 1);
}
```

Postorder traversal: each node is processed after the children.

Print the root indented by 2*depth

```cpp
/** Prints a tree indented according to depth.
* Pre: The tree is not empty. */
void print(const Tree& T, int depth) {
    cout << string(2*depth, ' ') << T.name << endl;
    print(T.left, depth + 1);
    print(T.right, depth + 1);
}
```

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

Print the children with depth + 1

```cpp
/** Builds an expression tree from a correct
* infix expression. */
ExprTree* buildExpr(const string& expr) {
    // ... (implementation code)
    return PIE1; // return a new expression tree
}
```

Example: expression trees

Expression tree for: \(a \ast \left( b \ast c \right) \ast \left( d \ast \left( e \ast f \right) \right) \ast g\)

Postfix representation: \(a b c \ast \ast \left( d e f \ast \ast \right) \ast \ast g \ast\)

How can the postfix representation be obtained?

How to build an expression tree

```cpp
struct ExprTree {
    char op; // operand or operator
    ExprTree* left; // left child
    ExprTree* right; // right child
};
```

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

How can the postfix representation be obtained?

How to build an expression tree

```cpp
/** Generates a string with the expression in
* infix notation. */
int infixExpr(const ExprTree* T, const map<char, int>& V) {
    // ... (implementation code)
    return PIE1; // return a new expression tree
}
```

How to build an expression tree

```cpp
/** Evaluates an expression taking V as the value of the
* variables (e.g., V['a'] contains the value of a). */
int evalExpr(const ExprTree* T, const map<char, int>& V) {
    // ... (implementation code)
    return PIE1; // return a new expression tree
}
```

Binary trees

Nodes with at most two children.
How to build an expression tree

Example: expression trees

```cpp
Expr buildExpr(const string& expr) {
    stack<Expr> S;
    for (char c : expr) {
        if (c == '<' && c == '>') {
            // We have an opening ('<' or '>'.
            Expr left = S.top();
            S.pop();
            Expr right = S.top();
            S.pop();
            S.push(nu ExprTraverser(left, right));
        } else if (c < 'a' || c > 'z') {
            // The stack has only one element and is freed after return.
            return S.top();
        } else if (c == 'a' && c == 'b') { 
            // We have an operator ('\-' or '\*').
            Expr left = S.top();
            S.pop();
            Expr right = S.top();
            S.pop();
            S.push(nu ExprTraverser(left, right, c, c, c, c));
        } else if (c == 'a' && c == 'b') { 
            // We have an operator. Return (T\-left T\-op T\-right).
            return "<" + infixExpr(T\-left) + T\-op + infixExpr(T\-right) + ");"
        } // The stack has only one element and is freed after return.
    }
    return S.top();
}
```

### Exercises

- **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).**

---

**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: 2736145; Postorder: 3674512
  - Preorder: 71495268; Postorder: 195468273
- 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: 361852479; Inorder: 163528749
  - Level-order: 483127569; Inorder: 185246793
  - Postorder: 432596871; Inorder: 439251786
- Describe an algorithm that builds a binary tree from the preorder and inorder traversals.
We want to draw the skeleton of a binary tree as it is shown in the figure. For that, we need to assign \((x, y)\) coordinates to each tree node. The layout must fit in a pre-defined bounding box of size \(W \times H\), with the origin located in the top-left corner.

Design the function

```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.
Sets and Dictionaries

Containers:
Set and Dictionary

Jordi Cortadella and Jordi Petit
Department of Computer Science

Sets and Dictionaries

• A set: a collection of items. The typical operations are:
  – Add/remove one element
  – Does it contain an element?
  – Size?, is it empty?
  – Visit all items

• A dictionary (map): a collection of key-value pairs. The typical operations are:
  – Put a new key-value pair
  – Remove a key-value pair with a specific key
  – Get the value associated to a key
  – Does it contain a key?
  – Visit all key-value pairs

We will focus on the implementation of sets.

Possible implementations of a set

- Unsorted list or vector
  - Insertion: $O(n)$, if checking for duplicate keys, $O(1)$ otherwise.
  - Deletion: $O(n)$ since it has to find the item along the list.
  - Lookup: $O(n)$ since the list must be scanned
  - Good for: Small sets.

- Sorted vector
  - Insertion: $O(n)$ in the worst case (similar to insertion sort)
  - Deletion: $O(n)$ since it has to shift the elements after deletion.
  - Lookup: $O(log n)$ with binary search
  - Good for: Read-only collections (only lookups) or very few updates.

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

BST in ascending order

Sets and Dictionaries

Sets and Dictionaries

Binary Search Trees: find min/max

• A dictionary can be treated as a set of keys, each key having an associated value.

• We will focus on the implementation of sets.

BST: public methods

template<typename T>
class Set {
public:
  // Constructors, assignment and destructor
  Set();
  Set(const Set& S);
  Set& operator=(const Set& S);
  ~Set();

  // Finding elements
  const T& findMin() const;
  const T& findMax() const;
  bool contains(const T& x) const;
  int size() const;
  bool isEmpty() const;

  // Insert/remove methods
  void insert(const T& x);
  void remove(const T& x);
};

Binary Search Trees: contains

Contains?:
• Move to left/right depending on the value.
• Stop when:
  – The value is found (contained).
  – No more elements left (not contained)

Binary Search Trees: insert

Insert:
• Move to left/right depending on the value.
• Stop when:
  – The value is found (nothing to do).
  – A node is reached where the value is not found.

Binary Search Trees: remove

remove: simple case (no children)
remove: simple case (one child)
remove: complex case (two children)

Visiting the items in ascending order

Answer:

Using an in-order traversal
• 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 $d$?

Height of an AVL tree

• 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 time devoted to balancing is $O(\log n)$.

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

• Therefore, $h = \Omega(\log n)$.

AVL trees

• Named after Adelson-Velsky and Landis (1962).
  – Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.
  – Properties:
    – The height of the tree is always $\Theta(\log n)$.
    – The time devoted to balancing is $O(\log n)$.

AVL in action

• 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 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

The important question: what is the size of an AVL tree with height $h$?
The recurrence resembles the one of the Fibonacci numbers. A tighter bound can be obtained.

The height of an AVL tree with \( n \) internal nodes satisfies:

\[
h < 1.44 \log_2(n + 2) - 1.328
\]

Single and double rotations only need the manipulation of few pointers and the height of the nodes (\( O(1) \)).

The height must be stored at each node. Only the unbalancing factor (\( \{-1, 0, 1\} \)) is strictly required for balancing, and the recursive calls return to the ancestors (check heights and rotate if necessary).

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).

**EXERCISES**

- Starting from an empty BST, depict the BST after inserting the values 32, 15, 47, 67, 78, 39, 63, 21, 12, 27.
- 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?
Depict the three AVL trees after sequentially inserting the values 31, 32 and 33 in the following AVL tree:

- 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 a unique alphanumeric string (each country has a different system).

The following operations are needed:

1. **Sorted vector:** adding/removing takes \(O(\log n)\).
2. **Deletions:** must be "lazy" (slots must be invalidated but the slots are not actually removed).

In the parking lot, the cars are not in fixed positions, and the cars change their location frequently. Moreover, in the real world, the parking size is likely to be much smaller than the number of cars; therefore, the typical organization in a parking lot is a lot of cars, but the total number of cars is not large.

- We may not even know the exact size of the domain (all plates in the world).
- Most of the vector locations would be "empty" (e.g., assume that the parking has 1,000 places).

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

### Hashing

A hash function maps data of arbitrary size to a table of fixed size.

**Important questions:**

- How to design a good hash function?
- The hash function is not injective. How to handle collisions?

#### Hash function

- **We can calculate the location for item \(x\) as**
  \[ h(x) \equiv x \mod m \]
  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.

#### Handling collisions

- **A collision is produced when**
  \[ h(x_1) = h(x_2) \mod m \]

- **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.

#### Handling collisions: linear probing

- 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.

#### Handling collisions: 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 \(h_1(x), h_1(x) + h_2(x), h_1(x) + 2h_2(x), \ldots\)

#### Complexity analysis

- 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: \(2n\) (a prime number close to \(n\)). Rehashing occurs very infrequently and the cost is amortized by all the insertions. The average cost remains constant.

#### Binary Search Trees vs. Hash Tables

- 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 \(O(\log n)\). Checking takes \(O(\log n)\).
  - AVL trees: adding/removing/checking takes \(O(\log n)\).

- A (Boolean) vector with one location for each possible plate:
  - The operations could be done in constant time, but...
  - The vector would be extremely large (e.g., only the Spanish system has 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).

#### Hashing

By using a hash function, the insertion and removal operations can be done in constant time when the table is not full. However, the cost of rehashing should be taken into account, as it involves rebuilding the table with a larger size.

**Naive implementation options**

- **Hash Table**
  - \(O(1)\) for insertion, deletion, and search
  - \(O(N)\) for rehashing if the table is full

- **Binary Search Tree**
  - \(O(\log n)\) for insertion, deletion, and search
  - Space \(O(n)\)

- **Sorted Vector**
  - \(O(n)\) for insertion, deletion, and search
  - Space \(O(n)\)

**Operation**

- **Insertion/Deletion/Lookup**
  - Hash Table: \(O(1)\)
  - Binary Search Tree: \(O(n)\)
  - Sorted Vector: \(O(n)\)

- **Sorted Iteration**
  - In-order traversal: \(O(n)\)
  - Requires an extra sorted vector: \(O(n \log n)\)

- **Hash Function**
  - Not required

- **Total Order**
  - Required

- **Range Search**
  - Required

**Application:** data integrity check

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

**Different hashing algorithms exist:** SHA1, SHA256, ...

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, ...).

EXERCISES

Hash function

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

- Separate chaining
- Linear probing

All elements different

Let us assume that we have a list with \( n \) elements. Design an algorithm that can check that all elements are different. Analyze the complexity of the algorithm considering different data structures:

- Checking the elements without any additional data structure, i.e., using the same list.
- Using AVLs.
- Using hash tables.
Secret-key protocols

- Alice and Bob have to meet privately and choose a secret key.
- They can use the secret key to mutually exchange messages.
- There are many secret-key protocols. We will explain two of them:
  - XOR encoding.
  - Advanced Encryption Standard (AES).

Secret-key protocol: XOR encoding

- A secret key $r$ is chosen (a binary string).
- The encoding and decoding functions are identical:
  \[ \sigma_r(x) = d_r(x) = x \oplus r \]
- Example: $r = 11011100$.

\[ x = 00111010 \]
\[ r: 11011100 \]
\[ m: 11100110 \]
\[ m: 11100110 \]
\[ r: 11011100 \]
\[ x: 00111010 \]
\[ m \oplus r = 00001001 \]

It is convenient that the bits of $r$ are randomly generated.

Still, this is not a very robust scheme since Eve can figure out important information by listening several messages.

Secret-key protocol: AES encoding

- 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).

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.,
\[ M = \sigma_P(P(M)) = \sigma_S(S(M)) \]
- 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 $X = P_B(M)$ and send it to Alice. Only Alice will be able to decipher the message: $M = S_A(X)$.

Public-key protocols

But, how to create a cryptosystem like this? Using number theory.
Public-key cryptosystem (Rivest-Shamir-Adleman, 1977).

- Based upon number theory: modular arithmetic and prime numbers.
- Security: based on the fact that factoring a large number (product of two large primes) is hard.

**Bézout’s identity**

- Lemma: If \( d \) divides both \( a \) and \( b \), and \( d = ax + by \) for some integers \( x \) and \( y \), then necessarily \( d = \gcd(a, b) \).
- Proof:
  - Clearly, \( d \leq \gcd(a, b) \), since \( d \) is a divisor of both \( a \) and \( b \).
  - Since \( \gcd(a, b) \) is a divisor of both \( a \) and \( b \), it must also be a divisor of \( ax + by = d \). This implies that \( \gcd(a, b) \leq d \).
  - Therefore, \( d = \gcd(a, b) \).

**Extended Euclid’s algorithm**

- We calculate \( \gcd(q - 2, 1) \) and \( 1 \) modulo both \( q - 2 \) and 1.

**Modular arithmetic: properties**

- Given \( a \) and \( N \), we say that \( x \) is the multiplicative inverse of \( a \) (mod \( N \)) if:
  \[ ax \equiv 1 \pmod{N} \]

- Example. The multiplicative inverse of 4 (mod 7) is 2:
  \[ 4 \cdot 2 \equiv 1 \pmod{7} \]

- When \( \gcd(a, N) = 1 \), the multiplicative inverse of \( a \) always exists and can be calculated by the extended Euclid’s algorithm:
  \[ ax + Ny = 1 \rightarrow x \text{ is } a's \text{ inverse (mod } N) \]

- Public-key cryptosystems (e.g., RSA) are convenient (no need to share keys) but computationally expensive. Secret-key (symmetric) cryptosystems (e.g. AES) are more efficient. Both can be combined.

- Bob wants to send an encrypted message to Alice:
  - Bob 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 \( N - 1 \).
    - The secret key is \( d \), the inverse of \( e \) modulo \( N - 1 \), computed using the extended Euler’s algorithm.

- Alice sends a message \( m \) to Bob:
  - Alice encrypts the message using Bob’s public key (using RSA).

- Alice sends the encrypted message to Bob.

- Alice decrypts the message by computing \( m^d \mod N \).

**Cryptographic hash function (CHF)**

- A CHF maps data of arbitrary size to a fixed-size bit string.

- 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, SHA 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).

**Example: SHA-1**

- A scheme to guarantee that a message is authentic.

- Consider the following case:
  - Alice sends a document (possibly unencrypted) to Bob and wants Bob to electronically sign the document.
  - Bob “signs” the document and sends it back to Alice.

- Questions:
  - How does Alice know that the document has not been altered? → integrity.
  - How does Alice know that Bob has signed the document (and not somebody else)? → authentication.
How to generate large prime numbers?
(not explained in this lecture)

Simple RSA

Assume you have \( p = 5 \) and \( q = 7 \).

- Which is the smallest value for \( e \)?
- What is the corresponding value for \( d \)?
- Encrypt the message \( M = 3 \).
- Find all possible pairs \((e, d)\) valid for this cryptosystem.

Implement an RSA cryptosystem

\begin{itemize}
  \item Given two primes, \( p \) and \( q \), design an RSA cryptosystem (in C++ or python) as follows:
    \begin{itemize}
      \item 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.
      \item Find \( d \) that can be used for secret key.
      \item Implement a function \( \text{encode}(x, e, N) \) that computes \( x^e \mod N \). This function must be efficient. Note: assume that \( N^2 \) can be represented as an int.
      \item Implement a function to double check, for \( 0 \leq x < N \), that \( \text{encode}(\text{encode}(x, e, N), d, N) = x \).
    \end{itemize}
  \item Example: \( p = 79, q = 491 \).
    \begin{itemize}
      \item Public key: \((38789, 11)\), Secret key: \(31271\).
      \item \( e(2) = 2048, e(19) = 23855, e(32757) = 4, e(38788) = 38788, e(10) = 18550 \).
    \end{itemize}
\end{itemize}

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., \( M = 11101 \oplus 00011 \oplus 10100 \oplus 110 \). In case the length is not a multiple of 5, additional zeroes are added at the end of the message.

For a message \( M \) with \( k \) blocks, we define the cryptographic hash \( h \) as follows:
\[
h(M) = B_1 \oplus B_2 \oplus \cdots \oplus B_k.
\]
where \( \oplus \) means the bitwise application of XOR. For example, \( 01110 \oplus 11010 = 10100 \).

- What would be the output \( h(M) \) for the previous message \( M \)?
- If we change one bit of a message, does the output change a lot?
- Assume that we know \( h(M) \) and the length of \( M \). Is it easy to find another \( M' \) with the same length such that \( h(M) = h(M') \)? Justify your answer.
**Fast Fourier Transform**

**Polynomial representation**

- A polynomial is represented as a vector of coefficients \( a_0, a_1, \ldots, a_n \).
- For example, \( P(x) = x^3 - 2x^2 - 3x + 1 \) can be represented as \( (1, -2, -3, 1) \).

**Polynomials: coefficient representation**

- Coefficient representation of \( P(x) = x^3 - 2x^2 - 3x + 1 \):
  - Evaluation at the points \( \{x_0, x_1, \ldots, x_n\} \): \( \{y_0, y_1, \ldots, y_n\} \).

**Polynomials: point-value representation**

- Point-value representation of \( P(x) = x^3 - 2x^2 - 3x + 1 \):
  - Evaluation at \( x = 1 \): \( y_1 = 1 \).

**Interpolation: Lagrange polynomials**

- Lagrange polynomials for \( P(x) = x^3 - 2x^2 - 3x + 1 \):
  - \( A(x) = \sum_{i=0}^{n} \frac{P(x_i)}{x - x_i} \) at \( x_0, x_1, \ldots, x_n \).

**Evaluation by divide-and-conquer**

- The calculations needed for \( A(x) \) can be reused for computing \( A(-x) \).
- \( A(x) = A_r(x^2) + xA_s(x^2) \), \( A(-x) = A_r(x^2) - xA_s(x^2) \).

**Evaluation of \( A(x) \) at \( n \) paired points**

- Reduces to evaluating \( A_r(x) \) and \( A_s(x) \) at just \( n/2 \) points: \( x_0, \ldots, x_{n/2-1} \).

**Why Fourier Transform?**

- Fourier series of a periodic function \( f(t) \) of period 1:
  \[ f(t) = \sum_{m=0}^{\infty} a_m \cos(2\pi mt) + \sum_{m=1}^{\infty} b_m \sin(2\pi mt) \]

- Fourier coefficients:
  \[ a_n = \frac{1}{T} \int_0^T f(t) \cos(2\pi nt) \, dt \]
  \[ b_n = \frac{1}{T} \int_0^T f(t) \sin(2\pi nt) \, dt \]

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

**Discrete-time signals**

- Fast Fourier Transform (FFT) for discrete-time signals.

- Given a polynomial \( p(x) = a_0 + a_1 x + \cdots + a_n x^n \), evaluate it at \( n \) points \( x_0, x_1, \ldots, x_n \):
  \[ \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \]

- Horner’s rule:
  \[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- We need \( x_0^2 \) and \( x_1^2 \) to be a plus-minus pair. But a square cannot be negative!
The runtime of the FFT can be expressed as:

\[ T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \]

• Using the Master Theorem we conclude:

Runtime FFT(n) = O(n \log n)

Let us consider a polynomial:

\[ A(\omega) = a_0 + a_1 \omega + a_2 \omega^2 + \cdots + a_n \omega^n \]

where \( \omega = e^{2\pi i/n} = (1, 2i/n) \).

Let us call \( F_n(\omega) \) the Fourier matrix. Thus,

\[ y = F_n(\omega) \cdot a \]

Example: from values to coefficients

\[ F(2, 1-\omega, 4, 1+\omega, \omega = -\omega) \]

\( a_0 = 2, a_1 = 1-\omega, a_2 = 4, a_3 = 1+\omega, a_4 = \omega = -\omega \)

\[ [y_0, y_1 = F(2, 1-\omega, 4, 1+\omega, \omega = -\omega)] \]

\( x = [y_0, y_1] = F^{-1}(a_0, a_1, a_2, a_3) \)

\[ [x_0, x_1, x_2, x_3] = F^{-1}(2, 1-\omega, 4, 1+\omega) = \omega \]

Polynomial multiplication

\[ P(x) = x^2 + 2x + 1 \]

\[ Q(x) = x^2 - 2x + 1 \]

\[ R(x) = P(x) \cdot Q(x) \]

\[ R(x) = (x^2 + 2x + 1) \cdot (x^2 - 2x + 1) \]

\[ R(x) = x^4 - 3x^2 + 1 \]

\[ R(x) = FFT([2, 1-\omega, 4, 1+\omega, \omega = -\omega]) \]

\[ [x_0, x_1, x_2, x_3] = F^{-1}(2, 1-\omega, 4, 1+\omega) = \omega \]

\[ [x_0, x_1, x_2, x_3] = F^{-1}(2, 1-\omega, 4, 1+\omega) = \omega \]

\[ [x_0, x_1, x_2, x_3] = F^{-1}(2, 1-\omega, 4, 1+\omega) = \omega \]

Consider the FFT of the polynomial \( x^2 + 2x + 1 \):

\[ \text{Find the value of } \omega \text{ to execute the FFT.} \]

\[ \text{In which points the polynomial must be evaluated?} \]

\[ \text{Execute the FFT and give the point-value representation of the polynomial.} \]

**EXERCISES**

- 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).
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.