

# Algorithmics, Spring 2011

Maria Serna

web page: <http://www.lsi.upc.edu/~mjserna>

e-mail: [mjserna@lsi.upc.edu](mailto:mjserna@lsi.upc.edu)

Office: Edifici Omega - 235

Christian Blum

web page: <http://www.lsi.upc.edu/~cblum>

e-mail: [cblum@lsi.upc.edu](mailto:cblum@lsi.upc.edu)

Office: Edifici Omega - 112

# Evaluation

- Final Exam: 70%
- 3-5 problem assignments, at least 3 compulsory: 30%

Junio 2009

# Initial planning: Topics

- Basic algorithmic techniques (review)
- Graph and Network algorithms
- Algorithms for the web
- Approximation algorithms
- Local search and heuristics

# Basic References

## □ Exact algorithms

- ★ Cormen, Leisserson and Rivest. [Introduction to algorithms](#), MIT Press and McGraw Hill 1990.
- ★ Cormen, Leisserson, Rivest and Stein. [Introduction to algorithms](#). Second edition, MIT Press and McGraw Hill 2001.
- ★ Kleinberg, Tardos. [Algorithm Design](#), Pearson Education, 2006.
- ★ Skiena. [The algorithm design manual](#), Springer Verlag, 1998.

□ Approximation algorithms

- ★ Garey and Johnson. [Computers and intractability. A guide to the NP-Completeness](#), W.H. Freeman, 1979.
- ★ Vazirani. [Approximation Algorithm](#). Springer, 2001

## □ Complexity theory

- ★ Sipser. [Introduction to the Theory of Computation](#). PSW, 1997.
- ★ Balcazar, Díaz, Gabarró. [Structural Complexity](#). 2nd edition, Springer, 1995
- ★ Papadimitriou. [Computational Complexity](#). Addison Wesley, 1994

# Already known?

- Growth of functions. Algorithm analysis.
- Problems: types and classes (complexity point of view)
- Basic algorithmic techniques
  - ★ Greedy algorithms
  - ★ Backtracking
  - ★ Divide and Conquer. Master Theorem and recurrence-solving.
  - ★ Dynamic Programming

# Growth of functions: Asymptotic notations

We consider only functions defined on the natural numbers.

$$f, g : \mathbb{N} \rightarrow \mathbb{N}$$

## $\Theta$ -notation

For a given function  $g(n)$

$$\Theta(g(n)) = \{f(n) \mid \text{there are positive constants } c_1, c_2, \text{ and } n_0 \geq 0 \text{ such that} \\ 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$$

Although  $\Theta(g(n))$  is a set we write  $f(n) = \Theta(g(n))$  to indicate that  $f(n)$  is a member of  $\Theta(g(n))$

$$5n^3 + 2n^2 = \Theta(n^3)$$

$$5n^3 + 2n^2 \notin \Theta(n^2)$$

It is used for asymptotic equivalence

## *O*-notation

For a given function  $g(n)$

$$O(g(n)) = \{f(n) \mid \text{there exists a positive constant } c \text{ and } n_0 \geq 0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$5n^3 + 2n^2 = O(2^n)$$

$$5n^3 + 2n^2 = O(n^4)$$

$$5n^3 + 2n^2 = O(n^3)$$

$$5n^3 + 2n^2 = \Theta(n^3)$$

$$2^n = O(2^{2n})$$

$$2^n = O(2^{n \log n})$$

It is used for [asymptotic upper bound](#).

## $\Omega$ -notation

For a given function  $g(n)$

$$\Omega(g(n)) = \{f(n) \mid \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$5n^3 + 2n^2 = \Theta(n^3)$$

$$5n^3 + 2n^2 = \Omega(n^3)$$

$$5n^3 + 2n^2 = \Omega(n^2)$$

$$2^n = \Omega(2^{n/2})$$

It is used for [asymptotic lower bound](#).

## *o*-notation

For a given function  $g(n)$

$$o(g(n)) = \{f(n) \mid \text{for any positive constant } c \text{ there is a positive constant } n_0 \text{ such that} \\ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Note that  $f(n) = O(n)$  implies  $f(n) \leq cg(n)$  asymptotically for some  $c$

but  $f(n) = o(n)$  implies  $f(n) \leq cg(n)$  asymptotically for any  $c$

and when  $f(n) = o(g(n))$  it holds that  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

It is used for **asymptotic upper bounds that are not asymptotically tight**.

## $\omega$ -notation

$$f(n) \in \omega(g(n)) \text{ iff } g(n) \in o(f(n))$$

# Algorithm's analysis

- Time
- Space

Algorithm  $\mathcal{A}$  on input  $x$  takes time  $t(x)$ .

$|x|$  denotes the size of input  $x$ .

**Definition:** The **cost function** of algorithm  $\mathcal{A}$  is a function from  $\mathbb{N}$  to  $\mathbb{N}$  defined as

$$C_{\mathcal{A}} = \max_{|x|=n} t(x)$$

If in addition we have a probability distribution on the set of inputs  $Pr(x)$ , we can compute the **average cost** of the cost function seen as a random variable.

# Problem types

## □ Decision

Input  $x$

Property  $P(x)$

**Example:** Given a graph and two vertices, is there a path joining them?

## □ Function

Input  $x$

Compute  $y$  such that  $Q(x, y)$

**Example:** Given a graph and two vertices, compute the minimum distance between them.

# Decision problem classes

- Undecidable

No algorithm can solve the problem.

- Decidable

There is an algorithm solving them.

- ★ P:

- There is an algorithm solving it with polynomial cost.

- ★ EXP

- There is an algorithm solving with exponential cost.

## ★ NP

It is possible to define a **certificate**  $y$  and a **property**  $P(x, y)$  such that

- If  $x$  is an input with answer **yes**, there is  $y$  such that  $P(x, y)$  is true,
- $P(x, y)$  can be decided in polynomial time, given  $x$  and  $y$ .
- $y$  has polynomial size with respect to  $|x|$ .

Problems with a polynomial time verifier

# NP-hardness

It is an open question whether  $P = NP$  or  $NP = EXP$ . Most believed is that  $P \neq NP$

$\Pi$  is NP-hard means that a polynomial time algorithm for  $\Pi$  can be reused to solve in polynomial time any problem in  $P$ .

The NP-hardness of a problem is assessed through reductions

# NP-completeness

Decision problem  $A$  is **NP-complete** iff  $A \in \text{NP}$  and  $A$  is NP-hard.

Look at Garey and Johnson for a big list of NP-hard/complete problems.

# Optimization problems

An **optimization problem** is a structure  $\mathcal{P} = (I, \text{sol}, m, \text{goal})$ , where

- $I$  is the input set to  $\mathcal{P}$ ;
- $\text{sol}(x)$  is the set of feasible solutions for an input  $x$ .
- $m$  is an integer measure defined over pairs  $(x, y)$ ,  $x \in I$  and  $y \in \text{sol}(x)$ .
- $\text{goal}$  is the optimization criterium MAX or MIN.

An optimization problem is a function problem whose goal, with respect to an instance  $x$  is to find an optimum solution, that is, a feasible solution  $y$  such that

$$y = \text{goal}\{(m(x, y') \mid y' \in \text{sol}(x))\}.$$

**Example:** Given a graph and two vertices, obtain a path joining them with minimum length.

# Optimization problem classes

- PO there is a polynomial time algorithm computing an optimal solution for any input.
- EXPO there is an exponential time algorithm computing an optimal solution for any input.
- NPO

An **optimization** is an structure  $\mathcal{P} = (I, \text{sol}, m, \text{goal})$ , where

- ★  $I$  is recognizable in polynomial time.
- ★  $\text{sol}(x)$  is the set of feasible solutions for an input  $x$ . These solutions are short, that is, a polynomial  $p$  exists such that, for any,  $y \in \text{sol}(x)$ ,  $|y| \leq p(|x|)$ . Moreover, it is decidable in polynomial time whether, for any  $x$  and for any  $y$  such that  $|y| \leq p(|x|)$ , whether  $y \in \text{sol}(x)$
- ★  $m$  is an integer measure, defined over pairs  $(x, y)$ ,  $x \in I$  and  $y \in \text{sol}(x)$ . The function  $m$  is computable in polynomial time and is also called the *objective function*.
- ★  $\text{goal}$  is the optimization criterium MAX or MIN.

The **bounded version** of an optimization problem is the decision problem

★ **minimization**  $\mathcal{P} = (I, \text{sol}, m, \text{min})$  is

Given  $x \in I$  and an integer  $k$

Is there a solution  $y \in \text{sol}(x)$  such that  $m(x, y) \leq k$ ?

★ **maximization**  $\mathcal{P} = (I, \text{sol}, m, \text{max})$  is

Given  $x \in I$  and an integer  $k$

Is there a solution  $y \in \text{sol}(x)$  such that  $m(x, y) \geq k$ ?

The bounded version of an NPO problem belongs to NP.

## Other problem types

- **Counting** obtain  $|\text{sol}(x)|$ .
- **Enumeration** obtain all  $y \in \text{sol}(x)$ .
- **Sampling** given a distribution  $\pi$  on  $\text{sol}(x)$  design an algorithm that produces an element  $y \in \text{sol}(x)$  with probability  $\pi(y)$ .

# Problems and algorithms

We are interested in designing efficient algorithms for solving problems.

Efficient?

# Solving recurrences: Master theorem

For completeness we incorporate here the master theorem as presented in [CLRS] as a basic tool to solve some of the recurrences that arise in the analysis of divide and conquer algorithms.

**Theorem:** Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  defined in the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret  $n/b$  to mean either  $\lfloor n/b \rfloor$  or  $\lceil n/b \rceil$ .

Then  $T(n)$  can be bounded asymptotically as follows.

- If  $f(n) = O(n^{\log_b a - \epsilon})$  for some constant  $\epsilon > 0$ , then  $T(n) = \Theta(n^{\log_b a})$ .
- If  $f(n) = \Theta(n^{\log_b a - \epsilon})$ , then  $T(n) = \Theta(n^{\log_b a} \log n)$ .
- If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for some constant  $\epsilon > 0$ , and if  $af(n/b) \leq cf(n)$  for some constant  $c < 1$ , then  $T(n) = \Theta(f(n))$ .

# The first problem

## Maximum Fractional Knapsack (MAX-F-KNAPSACK)

We have  $n$  objects and a knapsack.

The  $i$ -th object has positive **weight**  $w_i$  and positive unit **value**  $v_i$ .

The knapsack **capacity** is  $C$ .

We wish to select a set of proportions of objects to put in the knapsack so that the total values is maximum and without breaking the knapsack.

**function** FracknapsackGreedy( $w[1 .. n], v[1 .. n], M$ ) : **array**  $[1 .. n]$  **of** rational

The objects are sorted in increasing order of (\*)...

**for**  $i \leftarrow 1$  **to**  $n$  **do**

$x[i] \leftarrow 0$

**end for**;

$weight \leftarrow 0; i \leftarrow 1;$

**while**  $weight < M \wedge i \leq n$  **do**

**if**  $weight + w[i] \leq M$

**then**  $x[i] \leftarrow 1; weight \leftarrow weight + w[i];$

**else**  $x[i] \leftarrow (M - weight)/w[i]; weight \leftarrow M$

**end if**;

$i \leftarrow i + 1$

**end while**;

**return**  $(x)$

**end**

(\*) Different selections will provide different greedy algorithms.

In any reasonable sorting criteria the algorithm's cost is polynomial.

# Fractional Knapsack an example

Input: 5 objects,  $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

## Select always the most valuable object

object	1	2	3	4	5
selected	0	0	1	0.5	1

Total selected weight **100** and total value **146**.

Input: 5 objects,  $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

## Select always the lighter object

object	1	2	3	4	5
selected	1	1	1	1	0

Total selected weight **100** and total value **156**.

Input: 5 objects,  $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

Select always the object with highest ratio value/weight

object	1	2	3	4	5
ratio	2.0	1.5	2.2	1.0	1.2
selected	1	1	1	0	0.8

Total selected weight **100** and total value **164**.

Input: 5 objects,  $C = 100$

w	10	20	30	40	50
v	20	30	66	40	60

**Theorem:** The greedy algorithm that always selects the most valuable object does not always find an optimal solution to the Fractional Knapsack problem.

**Theorem:** The greedy algorithm that always selects the lighter object does not always find an optimal solution to the Fractional Knapsack problem.

**Theorem:** The greedy algorithm that always selects the object with better ratio value/weight always finds an optimal solution to the Fractional Knapsack problem.

**Proof.** Assume that the objects are  $\{1, \dots, n\}$  and that

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}.$$

Let  $X = (x_1, \dots, x_n)$  be the solution computed by the greedy algorithm.

If  $x_i = 1$  for all  $i$ , the solution is optimal.

Otherwise, let  $j$  be the smallest value for which  $x_j < 1$ .

According with the algorithm we have:

If  $i < j$  then  $x_i = 1$ , and if  $i > j$  then  $x_i = 0$ .

Furthermore  $\sum_{i=1}^n x_i w_i = W$

Let  $Y = (y_1, \dots, y_n)$  be any feasible solution, we have

$$\sum_{i=1}^n y_i w_i \leq W = \sum_{i=1}^n x_i w_i$$

therefore

$$\sum_{i=1}^n x_i w_i - \sum_{i=1}^n y_i w_i \geq 0.$$

Let  $V(Z)$  denote the total value of a feasible solution.

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) v_i = \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i}.$$

If  $i < j$ ,  $x_i = 1$  then  $x_i - y_i \geq 0$  and  $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$  and we have

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

If  $i > j$ ,  $x_i = 0$  then  $x_i - y_i \leq 0$  but  $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$  therefore

$$(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j}$$

Plugging the inequality we have

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \geq \sum_{i=1}^n (x_i - y_i) w_i \frac{v_j}{w_j} \geq \frac{v_j}{w_j} \sum_{i=1}^n (x_i - y_i) w_i \geq 0$$

Therefore  $X$  is an optimal solution.



## Next problems

Given a sequence  $a_1, \dots, a_n$  of elements from a well ordered set, the indices  $i < j$  form an **inversion** if  $a_i > a_j$ .

### **Inversions** (INS)

Given a sequence without repeated elements compute the number of inversions.

Number of inversion is one measure used when comparing two different rankings.

### **Closest pair of points** (CP)

Given  $n$  points in the plane find the closest pair of points, according to the euclidian distance.