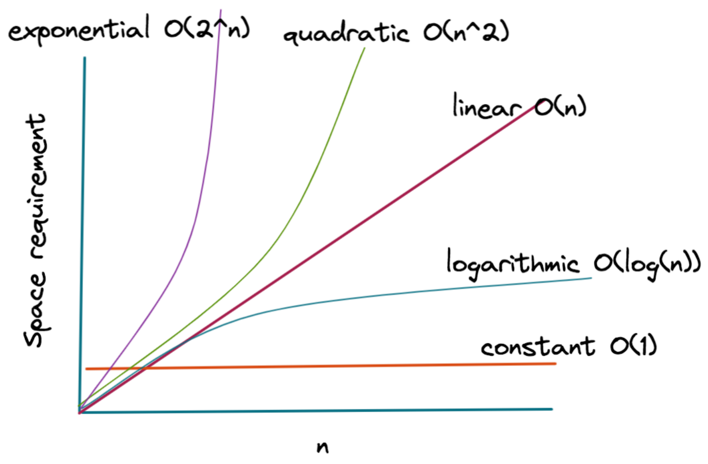


LINMA2111 - Discrete mathematics II

Simon Desmidt
Issambre L'Hermite Dumont

Academic year 2025-2026 - Q1

UCLouvain

Contents

Introduction

1.1 Sorting problems

A sorting problem is a problem that consists of taking a sequence of n objects and putting them in order. This kind of problem is made of three main elements:

- Context: set S with a partial order $<$;
- Input: n elements of S;
- Output: permutation of the input elements respecting the order.

To prove the correctness of an algorithm, we generally use the Hoare triple, i.e. a tuple for any input array x_0:

$$\begin{aligned}&\{\text{Algorithm to be used; Precondition; Postcondition}\}\\&\{IS;\ x = x_0;\ x \text{ is sorted and is a permutation of } x_0\}\end{aligned} \tag{1.1}$$

where IS is the insertion sort algorithm, and x is the sorted array.
In practice, to prove the correctness of an algorithm, we define the invariant and the base case, and do an induction step show that the invariant is preserved. The proof is done in my notes at page 1.

1.2 Definition of complexity

For some functions $f, g : \mathbb{N} \to \mathbb{R}^+$,

$$\begin{aligned}&f \in \mathcal{O}(g) \iff \exists c > 0, \exists n_0, \forall n > n_0,\ f(n) \leq cg(n)\\&f \in \Omega(g) \iff g \in \mathcal{O}(f)\\&f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ and } f \in \Omega(g)\\&f \in o(g) \iff \forall c > 0, \exists n_0, \forall n > n_0,\ f(n) < cg(n)\\&f \in \omega(g) \iff g \in o(f)\end{aligned} \tag{1.2}$$

- The time complexity is the number of operations as a function of the input size;
- The space complexity is the amount of memory used (in addition to the input) as a function of the input size.

We define the average case complexity as an expectation of the time taken by the algorithm on each input possible, with a dependence in the size of the input.

$$t = \mathbb{E}_{x \sim D}[T(x)] \tag{1.3}$$

where D is the distribution of the input.

1.3 Complexity of sorting algorithms

No sorting algorithm can be faster than $\Omega(n \log(n))$. This can be proven through the complexity of comparison-based algorithms.

1.4 Sorting algorithms

1.4.1 Selection Sort

This algorithm is the naive approach in complexity $\mathcal{O}(n^2)$. It goes as follows:

Algorithm 1 Selection sort algorithm

```
1: for $i$ from 1 to $n-1$ do
2:     Select the smallest element in the subarray from index i to n;
3:     Swap it with element at index i;
4: end for
```

The invariant of this algorithm is that the elements 1 to $i-1$ are sorted.

1.4.2 Insertion Sort

The idea of this algorithm is to shift the elements to the left until they are well placed. This also has a complexity $\mathcal{O}(n^2)$, although the best case is in $\mathcal{O}(n)$.

Algorithm 2 Insertion sort algorithm

```
1: for $i$ from 2 to $n$ do
2:     shift element $i$ to the left by successive swaps until it is well placed;
3: end for
```

The invariant of this algorithm is that the elements 1 to $i-1$ are sorted.

1.4.3 Quick Sort

The quick sort algorithm is based on divide and conquer methods (see chapter 2): it splits and solves smaller sorting problems, and recombines the outputs at the end into a sorted instance.

Algorithm 3 Quick Sort algorithm

```
1: pivot = T[1];
2: T_low = [T[i]  :  T[i] < pivot and i >1];
3: T_high = [T[i]  :  T[i] > pivot and i >1];
4: quicksort(T_low);
5: quicksort(T_high);
6: T = [T_low; pivot; T_high];
```

The worst-case complexity is still $\mathcal{O}(n^2)$. However, here, the average-case complexity is lower: $\mathcal{O}(n\log(n))$.
The randomized version of the algorithm consists in shuffling before applying the classical quick sort, in order to change the pivot value. As this is a random algorithm, the worst-case complexity corresponds to the expected complexity, i.e. $\mathcal{O}(n\log(n))$.

1.4.4 Merge Sort

The merge sort algorithm is also based on divide-and-conquer, and its complexity is also $\mathcal{O}(n^2)$.

Algorithm 4 Merge Sort algorithm

```
1: T_left = [T[i] such that i <= n/2]
2: T_right = [T[i] such that i > n/2]
3: mergesort(T_left)
4: mergesort(T_right)
5: T = merge(T_left, T_right)
```

The recurrence equation for the complexity is

$$t_n = 2t_{\lceil n/2 \rceil} + \Theta(n) \Longrightarrow t(n) = \Theta(n\log(n)) \tag{1.4}$$

The disadvatange of quick sort is that this algorithm does not have randomness, as it is complex to simulate. However, the constant in the complexity order is higher than for the quick sort, and the memory usage can be higher too.

1.5 Decision tree

A decision tree focuses on strategies and outcomes. It captures the main operations and the leaves are the possible outcomes ($\mathcal{O}(n!)$ in sorting problems, for n the size of the data). This helps us compute the complexity of a sorting algorithm: the number of steps is the depth S of the tree. Then, we have

$$2^S \geq n! \Longrightarrow S = \mathcal{O}(log(n!)) \tag{1.5}$$

This is equivalent to $\mathcal{O}(n\log(n))$:

$$S = \log(n!) = \sum_{i=1}^{n} \log(i) \leq \int_1^n log(x)dx \tag{1.6}$$

and we know that $S \geq \int_1^{n+1} \log(x)dx$[1]. Then,

$$\begin{aligned} [x\log(x) - x]_1^n \leq S \leq [x\log(x) - x]_1^{n+1} \\ S = \mathcal{O}(n\log(n)) \end{aligned} \tag{1.7}$$

1.5.1 Shannon coding theorem

Let $\mathcal{S} = \{1, \ldots, N\}$ and P a random variable over $\mathcal{S}$ such that $Pr[P = i] = p_i$. Let $f : \mathcal{S} \to \{0,1\}^*$ maps a number to its binary representation, under the constraint that no number is a prefix of another one. Then,

$$\sum_{i=1}^{N} p_i|f(i)| \geq H(P) = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{1.8}$$

where $H(P)$ is called the entropy of the probability distribution P. This theorem states that the average length of the prefix of the encoding of N elements must be at least $\log(N)$ characters long.

1.5.2 Yao's minimax principle

Consider a probability distribution over instances of a given size of a given problem. There, there exists a deterministic algorithm solving the problem for thoses instances, whose average-case complexity for the given distribution is lower than the worst-case expected complexity of any random algorithm solving the same problem.
This means that a random algorithm cannot be better than a deterministic algorithm in every case of a problem.

1.5.3 What about the part on thermodynamics in CM2?

[1]Why?

Divide-and-conquer algorithms

The divide-and-conquer method consists in three steps:

1. Divide: create smaller subproblems;
2. Recurse: solve them;
3. Combine: merge the solutions.

In sorting problems, a divide-and-conquer algorithm is the merge sort algorithm. It consists in dividing the array in two, sorting each half recursively, and merging the two sorted halves. The merge operation is done in linear time.

2.1 Complexity of an integer multiplication

Given two n-digit numbers, we want to compute their product:

$$\begin{cases} a = a_{n-1}a_{n-2}...a_1a_0 \\ b = b_{n-1}b_{n-2}...b_1b_0 \end{cases} \implies c = a \cdot b = c_{2n-1}...c_0 \tag{2.1}$$

Let us define B as the basis (e.g. 10) and decompose a and b in two parts:

$$\begin{cases} a = \alpha_0 + B\alpha_1 \\ b = \beta_0 + B\beta_1 \end{cases} \implies a \cdot b = \alpha_0\beta_0 + B(\alpha_1\beta_0 + \alpha_0\beta_1) + B^2\alpha_1\beta_1 \tag{2.2}$$

In that case, we find a recurrence relation for the computation time:

$$T(n) = 4T(n/2) + \Theta(n) \tag{2.3}$$

The factor 4 comes from the fact that we need 4 products, and the $\Theta(n)$ is the complexity of the sum of the products.

We introduce the Master theorem to solve this equation, see section 2.1.1. It gives a complexity of $T(n) = \Theta(n^{\log_2(4)}) = \Theta(n^2)$.
To reduce the complexity, we can change the value of the coefficient before $T(n/2)$ from 4 to 3 by calculating only 3 products:

$$\begin{cases} \gamma_0 = \alpha_0\beta_0 \\ \gamma_2 = \alpha_1\beta_1 \\ \gamma_1 = (\alpha_0 + \alpha_1)(\beta_0 + \beta_1) - \gamma_0 - \gamma_2 \end{cases} \tag{2.4}$$

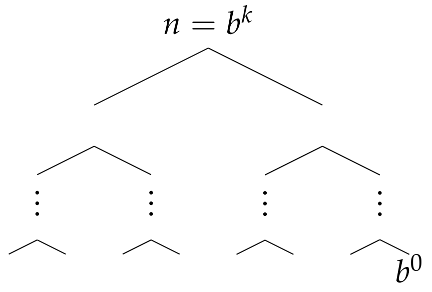
This reduces the complexity to $\Theta(n^{1.58})$.
Following a similar reasoning, we can instead divide a and b into 3 sums instead of 2, and get 5 multiplications. This gives a complexity of $\Theta(n^{\log_3(5)}) = \Theta(n^{1.46})$. Dividing in 4, 5, etc, we converge to a complexity of $\Theta(n)$ and this is the optimal complexity for the multiplication of two numbers of n digits. The problem is that the constant in front of the n starts to grow as we divide into more and more limbs, and so a bigger exponent can be enough for most arrays[1].

2.1.1 Master Theorem

Let $a \geq 1$ and $b > 1$ be constants and $f(n)$ a positive function, and let $T(n)$ be defined by $T(0) > 0$ and $T(n) = aT(\lfloor n/b \rfloor) + f(n)$. Then,

- If $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b(a)})$;
- If $f(n) = \Theta(n^{\log_b(a)})$, then, $T(n) = \Theta(n^{\log_b(a)} \log(n))$;
- If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and if, for some $c > 1$ and n_0 such that $af(\lfloor n/b \rfloor) \leq cf(n)$ for all $n \geq n_0$, then $T(n) = \Theta(f(n))$;

2.1.2 Proof of the master theorem



In that tree, for a level i, the size of the problem is b^i and the number of subproblems is a^{k-i}. By summing up,

$$T(b^k) = \sum_{i=0}^{k} a^{k-i} f(b^i) \tag{2.5}$$

For a function $f(n) = n^\alpha$,

$$T(n) = a^k \sum_{i=0}^{k} \left(\frac{b^\alpha}{a}\right)^i = a^k \frac{1 - (\frac{b^\alpha}{a})^{k+1}}{1 - b^\alpha/a} \tag{2.6}$$

Under the assumption that $\alpha \neq \log_b(a)$. As $k = \log_b(n)$, we can replace above and simplify using the formula $a^{\log_b(n)} = n^{\log_b(a)}$. In the case where $a = b^\alpha$, then

$$T(n) = a^k(k+1) = a^{\log_b(n)}(\log_b(n) + 1) = \Theta(n^{\log_b(a)} \log(n)) \tag{2.7}$$

[1]We call galactical algorithm an algorithm that is asymptotically good but the constant grows so large that it is only useful for huge arrays (e.g. $\sim 10^{80}$).

2.2 Multiplying polynomials

Consider $a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$. It can either be represented by its coefficients, or some values $(a(x_0), \ldots, a(x_{n-1}))$ for n distinct values. To compute those values, we can use a matrix-vector product:

$$\begin{bmatrix} 1 & x_0 & \ldots & x_0^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ \vdots \\ a_{n-1} \end{bmatrix} = \underline{\underline{V}}\mathbf{a} \tag{2.8}$$

where the matrix $\underline{\underline{V}}$ is called the Vandermonde matrix.

2.2.1 Convolution theorem

$$\begin{tikzcd}
\boxed{\begin{array}{l} a(x) = \sum\limits_i a_i x^i \\ b(x) = \sum\limits_i b_i x^i \end{array}} \arrow[d, "\text{Evaluate } \underline{\underline{V}}"'] & \boxed{c(x) = \sum_i c_i x^i} \\
\boxed{\begin{array}{lll} a(x_0) & \ldots & a(x_{2n-1}) \\ b(x_0) & \ldots & b(x_{2n-1}) \end{array}} \arrow[r, "\text{pointwise mult}"'] & \boxed{c(x_0) \ldots c(x_{2m-1})} \arrow[u, "\text{Interpolate } \underline{\underline{V}}^{-1}"]
\end{tikzcd}$$

This method gives bad conditioning, but this is not a problem here as we consider integer matrices.

2.3 Discrete Fourier Transform

From the previous section, if we take the points as the n complex roots of 1, i.e. $x_0 = e^{2\pi i/n}$ and $x_j = e^{2\pi i j/n}$, then we get a Discrete Fourier Transform matrix. Defining $\omega_n = x_0$,

$$\underline{\underline{V}} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \vdots & \omega_n & \ldots & \omega_n^{n-1} \\ \vdots & \ddots & \omega_n^{(i-1)(j-1)} & \vdots \\ 1 & \omega_n^{n-1} & \ldots & \omega_n^{(n-1)^2} \end{bmatrix} \tag{2.9}$$

This gives us

$$\left(\underline{\underline{V}}^{-1}\right)_{ij} = \frac{1}{n}\left[\omega_n^{-(i-1)(j-1)}\right] \tag{2.10}$$

and so $\underline{\underline{V}} = \frac{1}{n}\underline{\underline{V}}^*$. We conclude on

$$DFT(\mathbf{x}) = \underline{\underline{V}}\mathbf{x} \tag{2.11}$$

2.3.1 Fast Fourier Transform

The Fast Fourier Transform (FFT) algorithm uses the divide-and-conquer approach from above to compute the Fourier transform of a vector x. The exact algorithm is the following.
Assume that $n = 2^k$, $k \geq 1$, $\mathbf{x} = (x_0, \dots, x_{n-1})$ and $(y_0, \dots, y_{n-1}) = \underline{\underline{V}}_n\mathbf{x}$. The explicit formula to calculate y_i, $i = 0, \dots, n/2 - 1$ is

$$\begin{aligned} y_i &= \sum_{j=0}^{n-1} x_j\omega_n^{ji} = \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2ji} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{(2j+1)i} \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{ji} + \omega_n^i \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{ji} \end{aligned} \tag{2.12}$$

And in matrix form:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n/2-1} \end{pmatrix} = \underline{\underline{V}}_{n/2} \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{n-2} \end{pmatrix} + \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \underline{\underline{V}}_{n/2} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix} \tag{2.13}$$

And for the next half of y, we have a similar expression:

$$\begin{aligned} y_{i+n/2} &= \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2ji} \left(\omega_{n/2}^{n/2}\right)^j + \omega_n^{i+n/2} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{ji} \left(\omega_{n/2}^{n/2}\right)^j \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{ji} + (-1)\cdot\omega_n^i \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{ji} \end{aligned} \tag{2.14}$$

And, once again, in matrix form:

$$\begin{pmatrix} y_{n/2} \\ y_{n/2+1} \\ \vdots \\ y_{n-1} \end{pmatrix} = \underline{\underline{V}}_{n/2} \begin{pmatrix} x_0 \\ x_2 \\ \vdots \\ x_{n-2} \end{pmatrix} - \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \underline{\underline{V}}_{n/2} \begin{pmatrix} x_1 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix} \tag{2.15}$$

Let us define three notations:

$$\begin{aligned} \mathbf{x}^{[0]} &= (x_0, \dots, x_{n/2-1}) \\ \mathbf{x}^{[1]} &= (x_{n/2}, \dots, x_{n-1}) \\ \underline{\underline{T}}_n &= \begin{pmatrix} \omega_n^0 & & \\ & \ddots & \\ & & \omega_n^{n/2-1} \end{pmatrix} \end{aligned} \tag{2.16}$$

Then,

$$\underline{\underline{DFT}}(\mathbf{x}) = \begin{pmatrix} \underline{\underline{DFT}}(\mathbf{x}^{[0]} + \underline{\underline{T}}_n \underline{\underline{DFT}}(\mathbf{x}^{[1]})) \\ \underline{\underline{DFT}}(\mathbf{x}^{[0]} - \underline{\underline{T}}_n \underline{\underline{DFT}}(\mathbf{x}^{[1]})) \end{pmatrix} \tag{2.17}$$

The time complexity of this algorithm is $\Theta(n\log(n))$.

→ Note: there exists other algorithm to compute the FFT, such as the non power-of-two n alorithm.

2.4 Matrix multiplication

Given $A, B \in \mathbb{Z}^{n\times n}$, we want to compute $C = A \cdot B$. The basic algorithm is in $\Theta(n^3)$, but we want a better one.

2.4.1 Straßen algorithm

Let us divide the matrices in blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{2.18}$$

The Straßen algorithm uses the same idea as in section section 2.1: we define 7 block matrices to reduce the number of products done.

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22})B_{11} \\ M_3 = A_{11}(B_{12} - B_{22}) \\ M_4 = A_{22}(B_{21} - B_{12}) \\ M_5 = (A_{11} + A_{12})B_{22} \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \end{cases} \implies C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix} \tag{2.19}$$

Which has a complexity of $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$. The lower bound for the complexity is $\Omega(n^2)$, as we need to make at least one operation per element of C, and the current best algorithm (galactic) is in $\Theta(n^{2.371339})$.

2.4.2 Matrix inversion

As we can assume intuitively, matrix multiplication and inversion are closely linked. Therefore, the complexity to compute both should be linked too. Let us call $M(n)$ the time complexity of multiplication of matrices of size n, and $I(n)$ the time complexity of inversion of a matrix of size n. Then,

$$D = \begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix} \implies D^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix} \tag{2.20}$$

This means that the complexity of inversion is an upper bound on the complexity of multiplication: $M(n) \leq I(3n) \leq 3^3 I(n)\ \forall n \geq n_0$.

This kind of inequality is called reduction. Given problems A and B, if A can be transformed in a problem B of size $f(n)$ in time $T_R(n)$, then

$$T_A(n) \leq T_R(n) + T_B(f(n)) \tag{2.21}$$

For example, the problem of finding the median of an array can be transformed into the problem of sorting the array.

2.4.3 Matrix inversion upper bound

Let us assume that $A = A^T$ and $A \succ 0$.

$$A = \begin{pmatrix} B & C \\ C^T & D \end{pmatrix} \implies A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}CS^{-1}C^TB^{-1} & -B^{-1}CS^{-1} \\ -S^{-1}C^TB^{-1} & S^{-1} \end{pmatrix} \tag{2.22}$$

where $S = D - C^TB^{-1}C$ is the Schur complement of A. From Equation 2.21, $f(n) = \Theta(M(n))$ in our case, and so we have the following relation between inversion and multiplication:

$$I(n) = \mathcal{O}(M(n)) \tag{2.23}$$

$\rightarrow$ Note: the hypotheses that $A = A^T$ and $A \succ 0$ are not binding. If A is invertible but does not verify those conditions, we can work with AA^T that does, and we find the inverse of A with $A^{-1} = (A^TA)^{-1}A^T$.

Dynamic Programming

3.1 Two approaches

There are two approaches for a dynamic programming algorithm: bottom-up and top-down. In the bottom-up approach, we solve the subproblems first, and then use their solutions to solve bigger problems. In the top-down approach, we start from the main problem, and recursively solve the subproblems as needed.

- The main idea of bottom-up is to use memoization, i.e. storing the solutions of subproblems to avoid recomputing them.
- The main idea of top-down is recursion.

Dynamic programming is used to improve time complexity by trading time for space.

3.2 Examples

3.2.1 Rod cutting

The problem of rod cutting consists in wanting to cut a beam of length n, given commands of clients.Finding the optimal solution can be done by computing a solution on sub-beams and merging them.

Algorithm 5 Rod Cutting Algorithm

```
function CutRod($p, n$)
    if $n = 0$ then
        return 0
    end if
    $q \leftarrow -\infty$
    for $i = 1$ to $n$ do
        $q \leftarrow \max\{q,\ p_i + \text{CutRod}(p, n-i)\}$
    end for
    return $q$
end function
```

The complexity of this algorithm is given by a recurrence relation:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + T(n-1) + \sum_{j=0}^{n-2} T(j) = 2T(n-1) \Longrightarrow T(n) = \mathcal{O}(e^n) \quad (3.1)$$

This basic algorithm has a very bad complexity, but memoization can improve it.

Algorithm 6 Memoized rod cutting algorithm

```
function CUTROD_MEMOIZED($p, n$)
    $r[0:n] = -\infty$
    function CRM_AUX($p, n, r$)
        if $r[n] \geq 0$ then
            return $r[n]$
        end if
        if $n = 0$ then
            return 0
        end if
        $q = -\infty$
        for i=1 to n do
            $q = \max\{q, p_i + CRM\_Aux(p, n-i, r)\}$
        end for
        $r[n] = q$
        return $q$
    end function
    return CRM_Aux(p,n,r)
end function
```

Finally, another algorithm exists, using a bottom-up approach in dynamic programming. It has a complexity $\Theta(n^2)$.

Algorithm 7 Bottom-up efficient rod cutting algorithm

```
function BOTTOM-UP-CR($p, n$)
    $r[0:n]$
    $r[0] = 0$
    for j=1 to n do
        $q = -\infty$
        for i=1 to j do
            $q = \max\{q, p_i + r_{j-1}\}$
        end for
        $r[j] = q$
    end for
    return $r[n]$
end function
```

3.2.2 Matrix chain multiplication

This problems consists in doing the multiplication of matrices $A_1 A_2 \dots A_n$ with dimensions $p_0, p_1, \dots, p_n$. The total number of operations depends heavily on the order of the multiplication. The idea for the algorithm is to split the chain into smaller chains until we get to a product of two matrices. The splitting happens at the index i that minimizes the total number of operations, knowing the cost of computing a subchain:

$$Algo(p) = \min_{1 \leq i \leq n-1} \{Algo(p[0:i]) + Algo(p[i+1:n]) + p_0 p_i p_n\} \tag{3.2}$$

Let us define $m[i,j]$ the optimal cost of the product $A_i \dots A_j$. Our goal is to compute $m[1,n]$, with the initial condition $m[i,i] = 0$ for all i. The recurrence relation is

$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} \qquad \forall 1 \le i \le j \le n \tag{3.3}$$

This gives a complexity $\Theta(n^2)$, as we need to compute the entries of a triangular matrix $n \times n$.

3.3 Generating functions

Let $\{a_k\}_{k=0}^{\infty}$ be a sequence. We associate the function $\sum_{k=0}^{\infty} a_k x^k$ to the sequence, on which we have addition, multiplication, differentiation, etc.
Example:

$$a_k = 1 \; \forall k \in \mathbb{N} \implies f(x) = \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} = x \sum_{k=0}^{\infty} x^k + 1 = xf(x) + 1 \tag{3.4}$$

Another example is the Fibonacci sequence:

$$f_{n+2} = f_{n+1} + f_n \qquad f_0 = 0 \qquad f_1 = 1 \tag{3.5}$$

From the recurrence equation, we can get the generative function:

$$\begin{aligned} f(x) &= f_0 x^0 + f_1 x^1 + \sum_{k \ge 2} f_k x^k = f_0 + f_1 x + \sum_{k \ge 0} f_{k+2} x^{k+2} \\ &= f_0 + f_1 x + x^2 \sum_{k \ge 0} f_{k+1} x^k + x^2 \sum_{k \ge 0} f_k x^k \\ &= f_0 + f_1 x + x(f(x) - f_0) + x^2 f(x) \\ &= \frac{f_0 + (f_1 - f_0)x}{1 - x - x^2} \end{aligned} \tag{3.6}$$

Knowing that $\sum_{k \ge 0} f_{k+1} x^k = \frac{f(x) - f_0}{x}$

Randomized Algorithms

4.1 Probabilistic analysis and randomized algorithm

In probabilistic computing, we need a distribution of the inputs to be able to sample. Based on prior knowledge or assumptions, we can determine the average-case complexity. The goal is for it to be lower than the deterministic algorithm, although it comes at the price of the accuracy of the solution.

Let us use as example the hiring problem. We interview n candidates, one at a time, and we need to decide wether we hire the candidate or not right after the interview. Our goal is of course to hire the best candidate.
One approach would be to interview the $n/2$ first candidates and only observe their skills. Then, hire the first candidate that is more skilled than the average (or all) of the first half of candidates.

4.1.1 Indicator Random Variable

Given a sample space S and an event A, we define the indicator random variable as

$$X_A = \begin{cases} 1 \text{ if } A \text{ occurs} \\ 0 \text{ otherwise} \end{cases} \tag{4.1}$$

where $Pr[X_A = 1] = Pr[A]$. This is useful because $\mathbb{E}[X_A] = Pr[A]$.

4.2 Random Sampling and Applications

4.2.1 Birthday paradox

The birthday paradox is not stricly speaking a paradox, but is called that way because it goes against the first intuition. What is the number of people that must be in a room so that the probability that two people have the same birthday is higher than 1/2, assuming that the birthdays are uniformly distributed? Let $r = 1, \ldots, n$ and $Pr[b_i = r] = 1/n$. Then, for two people,

$$Pr[b_i = b_j] = \sum_{r=1}^{n} Pr[b_i = r \text{ and } b_j = r] = \sum_{r=1}^{n} Pr[b_i = r]Pr[b_j = r] = \sum_{r=1}^{n} \frac{1}{n^2} = \frac{1}{n} \tag{4.2}$$

And for k people, denoting B_k the event that k people have distinct birthdays, we have

$$\begin{aligned} Pr[B_k] &= 1 \cdot \frac{n-1}{n}\frac{n-2}{n}\dots\frac{n-k+1}{n} = 1 \cdot \left(1-\frac{1}{n}\right)\cdot(1-\frac{2}{n})\dots\left(1-\frac{k-1}{n}\right) \\ &\leq e^{-1/n}e^{-2/n}\dots e^{-(k-1)/n} = e^{-k(k-1)/2n} \end{aligned} \tag{4.3}$$

Then, for $Pr[B_k] \geq 1/2$, we get

$$k(k-1) \geq 2n \ln 2 \iff k \geq 23 \tag{4.4}$$

We can also compute the expectation: let X_{ij} be the 1 if the people i and j have the same birthday and 0 otherwise. Then $\mathbb{E}[X_{ij}] = 1/n$ and for $X = \sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij}$,

$$\mathbb{E}[X] = \sum_{i=1}^{k}\sum_{j=i+1}^{k} X_{ij} = \frac{k(k-1)}{2n} \tag{4.5}$$

This means that for at least $\sqrt{2n}+1$ people in a room, we can EXPECT at least one collision. For $n = 365$, this is $k = 28$ people.

4.2.2 Hash table

A hash table is a data structure in which inserting, searching and deleting is done in $\mathcal{O}(1)$ in average. The idea of a hash table is to store elements in an array of size m using a hash function $h : U \to \{0,\dots,m-1\}$, where U is the set of possible keys. So instead of using a direct addressing table, where k is stored at $T[k]$ ($\mathcal{O}(n)$ space), we store k at $T[h(k)]$ ($\mathcal{O}(m)$ space). This idea save space but introduces collision. We can deal with collisions using multiples way, this will be investigated later. The simplest hash function is the modulo function $h(k) = k \mod m$.
Using deterministic hash functions can lead to bad performance. If the input is well chosen, we could have a worst-case complexity of $\mathcal{O}(n)$. To overcome this, we randomize the choice of the hash function.

Definition 4.1. A family of hash functions $\mathcal{H}$ is universal if for all $k \neq l \in U$,

$$Pr_{h\in\mathcal{H}}[h(k) = h(l)] \leq \frac{1}{m} \tag{4.6}$$

This definition means that even with well-chosen inputs, the probability of collision is garanteed to be low. An example of universal hash function family is the following:

$$\mathcal{H} = \{h_{a,b}(k) = ((ak+b) \mod p) \mod m \mid a \in \{1,\dots,p-1\}, b \in \{0,\dots,p-1\}\} \tag{4.7}$$

where p is a prime number $p > |U|$. The randomness comes from the random choice of a and b.
There exists multiples ways to manage collisions. It is possible to ignore collisions, in that case you lose data. Another way is to use buckets with linked lists to store multiple elements at the same index. We can also use probing. Linear probing consists into putting the element at the next available index after a collision, so we try $h(k)$ then $h(k)+1$ then $h(k)+2$ and so on. Quadratic probing uses $h(k)+i^2$ instead of $h(k)+i$. Double hashing uses a second hash function h_2 to compute the next index: $h(k)+i\cdot h_2(k)$. And finally, Cuckoo hashing that consists in using two hashing function ($h_1(k)$ and $h_2(k)$) and allowing an item k to be either in the place given by $h_1(k)$ or by h_2.

4.2.3 Randomized algorithm for computing integrals

To approximate π , we can put n random points on a square of size 1. For each point, we define the associated random variable $X_i = 1$ if $x_i^2 + y_i^2 \leq 1$ and 0 otherwise. If we observe k points in the orthant, we have

$$\frac{k}{n} \approx \frac{\pi}{4} \tag{4.8}$$

And we can even compute the error using the variance of $X_n = \sum_{i=1}^n X_i$:

$$\mathbb{V}(X_n) = \frac{1}{n}\mathbb{V}(X_i) = \frac{1}{n}(\mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2) = \frac{1}{n}(\mathbb{E}[X_i] - \mathbb{E}[X_i]^2) = \frac{1}{n}\frac{\pi}{4}\left(1 - \frac{\pi}{4}\right) \tag{4.9}$$

This is can be done for any curve, not just the quarter of the circle, and we can thus approximate any integral with this method, with an absolutr error (standard deviation) of order $\mathcal{O}(1/\sqrt{n})$. This bound is valid for any dimension of the integration.

4.2.4 Randomized algorithm for decision problems

Polynomials

An example of a decision problem is to determine wether a polynomial is identically 0 or not. In one dimension, it is easy: use $d + 1$ points for a polynomial of degree d, and if all are 0, then $P \equiv 0$. In more than one dimension, the number of roots of the polynomial is infinite and we need something more. We can use the Schwartz-Zippel lemma:

Given a finite set S of an integral domain, let P be a polynomial of n variables $(x_1, \ldots, x_n)$ and of degree at most $d \geq 0$. Let us pick a random point $(r_1, \ldots, r_n) \in S$ uniformly:

$$Pr[P(r_1, \ldots, r_n) = 0] \leq \frac{d}{|S|} \tag{4.10}$$

We can prove it by induction:

$$P(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_1^i P_i(x_2, \ldots, x_n) \tag{4.11}$$

for some polynomials P_i of degree at most $d - i$ and $n - 1$ variables. We can go all the way to polynomials of one single variable and use the fundamental theorem of algebra.

From this lemma, the probability to predict correctly from a set of points if a polynomial is identically zero or not is

$$Pr[correct] = Pr[TRUE \;\&\; P \equiv 0] + Pr[FALSE \;\&\; P \not\equiv 0] \tag{4.12}$$

with $Pr[TRUE|P \equiv 0] = 1$ and $Pr[FALSE|P \not\equiv 0] \geq 1 - \left(\frac{d}{|S|}\right)^k$. Then,

$$Pr[correct] \geq 1 - (d/|S|)^k \tag{4.13}$$

Matrix product

Given $A, B, C \in \mathbb{R}^{n\times n}$, we want to check if $AB = C$. Doing the product is costly, so we can take random vectors $x \in \mathbb{R}^n$ and check the products:

$$A(Bx) - Cx \stackrel{?}{=} 0 \tag{4.14}$$

This is a n-variate polynomial of degree 1 and we can use the same method as in the previous section.

MaxCut Problem

Given a graph $G = (V, E)$, we want to partition V into two sets S_0 and S_1 such that we maximize edges crossing the cut. If we assign each vertex randomly to one of the two sets with a probability $1/2$, then:

$$\mathbb{E}[|\text{RandomCut}(S_0, S_1)|] \geq \frac{1}{2}\text{MaxCut}(G) \tag{4.15}$$

4.2.5 Amplification of Stochastic Advantage

A randomized algorithm may be wrong but it has a bias towards the correct answer. The idea is to run k times the algorithm to take advantages of the properties of the randomness. For example, for one-sided error, with Schwartz-Zippel, we know that we have no false negative. So if we get at least one TRUE in k runs, we can be sure that it is TRUE. For two-sided error, we can run k times independently and take the majority answer.

4.3 Las Vegas and Monte Carlo

An algorithm is said to be Las Vegas when the output is correct, but the time taken to get to this solution is random. On the other hand, an algorithm is said to be Monte Carlo when the output is correct with some probability, or we only get a bound on the real value.

4.4 Hash tables

TODO

4.5 Randomness generation

Let us take functions $f, g : X \to Y$ and an element $s \in X$. We define the sequence

$$\begin{cases} x_0 = s \\ x_{i+1} = f(x_i) \\ y_i = g(x_i) \end{cases} \tag{4.16}$$

The output of this pseudo-random generator (PRG) is the sequence (usual bits) $y = (y_0, \dots, y_n)$. One famous example of pseudo-random generator is Blum-Blum-Shub (BBS):
select random κ-bit prime integers p, q such that $p = q = 3 \mod 4$, and let $N = pq$. The PRG is

$$\begin{cases} x_0 = s \in \mathbb{Z}_N^* \\ f_N(x) = x^2 \mod N \\ g(x) = x \mod 2 \end{cases} \tag{4.17}$$

4.6 Derandomization

Derandomization transforms a randomized algorithm into a deterministic algorithm, or at least into an algorithm using less randomness, without increasing time and memory costs too much.

4.6.1 Techniques

- Use a PRG: cheap, the randomness goes in the time complexity, and some randomness still exists;
- Try all possible values: trades n random bits for 2^n time, can remove all randomness, but can have a polynomial time slowdown.

Computability

5.1 Turing machine

A Turing machine is made of the following elements:

- An infinite tape serving as unlimited memory;
- A head that reads and write the symbols and moves around the tape;
- A specific blanc symbol ␣or ϵ;
- The number of states must be finite.

Initially, the tape contains only the input string and is blank elsewhere. Storing information can be done by writing on the tape with the head, and reading information can be done by moving the head over the information. The possible outputs are only *accept* and *reject*, obtained when entering the corresponding state q_{accept} or q_{reject}.

5.1.1 Formal definition

Definition 5.1. A Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are finite sets and

- Q is the set of states;
- Σ is the input alphabet (not containing the blank symbol);
- Γ is the tape alphabet, where $␣ \in \Gamma$ and $\Sigma \subseteq \Gamma$;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function[1];
- $q_0 \in Q$ is the initial state;
- $q_{accept} \in Q$ is the accept state and $q_{reject} \in Q$ is the reject state and is different from the accept state.

For $a, b \in \Gamma$, and $q, r \in Q$, $\delta(q, a) = (r, b, L)$ means that if the machine is in state q and the head is over a tape square containing a symbol a, it replaces a with b, the state becomes r, and the head moves one step to the left.

$\to$ Note: the input of length n is on the n leftmost squares of the tape, and the machine never tries to move its head to the left when it is on the first square.

[1]We can add a "stay put" move S: $\{L, R, S\}$.

Definition 5.2. We call the configuration of the Turing Machine the tuple (current state, current tape content, current head location). We write uqv the configuration where the current state is q and the tape content is the string uv, and the head location is the first symbol of v.

A Turing machine M accepts input w if a sequence of configurations $C_1, C_2, \dots, C_k$ exists where

- C_1 is the start configuration of M on input w, i.e. $C_1 = q_0 w$;
- each C_i yields C_{i+1};
- C_k is an accepting configuration.

The collection of string inputs that the machine accepts is called its language, and is denoted $\mathcal{L}(M)$. We say that the language is Turing-recognizable.

Definition 5.3. A decider is a Turing machine that finds a finishing state on all inputs. A decider that recognizes some language is said to decide it, and the language is Turing-decidable is some Turing machine decides it.

A decision problem is decidable iff it is computable, i.e. iff there exists an algorithm that can decide this problem.
We say that two machines are equivalent if they recognize the same languages.

5.1.2 Example

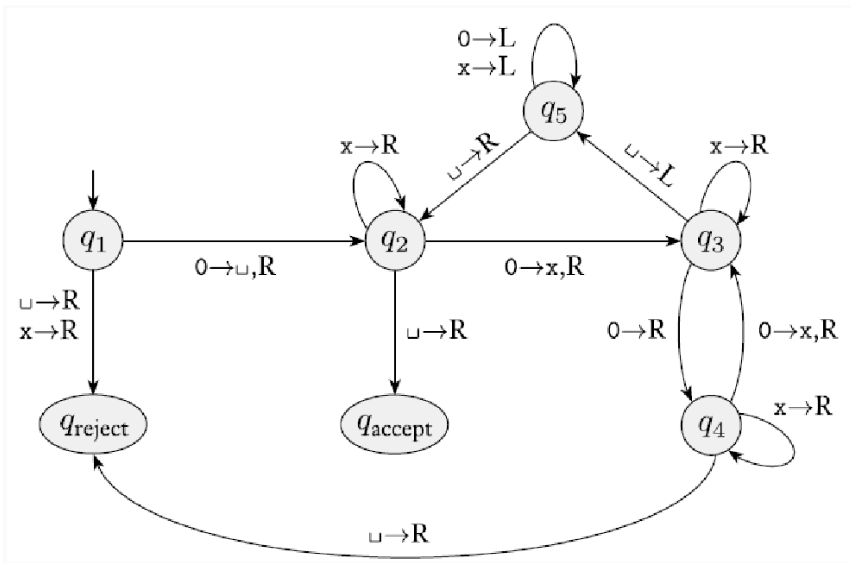


Figure 5.1: Example of a Turing machine

The language $A = \{0^{2^n} | n \geq 0\}$ is decidable. Let $M_2 = (Q, \Sigma, \Gamma, \delta, q_1, q_{accept}, q_{reject})$ such that

- $Q = \{q_1, \dots, q_5, q_{accept}, q_{reject}\}$ with q_1 the initial state;

- $\Sigma = \{0\}$ and $\Gamma = \{0, x, \textvisiblespace\}$;
- δ is the Figure 5.1.

Here is the breakdown of the steps:

- q_1: reads the first symbol and if it is 0, replaces it with $\textvisiblespace$and moves right to state q_2. If it is something else (x or $\textvisiblespace$), then the input is invalid and reject.
- q_2: moves to the right whatever the bit is. If the bit is 0, then replaces it with x and moves to q_3. If it is $\textvisiblespace$, then there is no more unmarked 0 and goes to accept.
- q_3: If the bit is 0, then moves to the right and to state q_4 and does nothing. If the bit is $\textvisiblespace$, then goes back to the left and moves to state q_5. If the bit is x, goes to the right and no change of state.
- q_4: If the bit is 0, then marks it x and goes to the right, back to state q_3. If it is x, goes to the right with no change of state. If it is $\textvisiblespace$, goes to the right in a reject configuration.
- q_5: if the bit is 0 or x, goes back to the left and no change of state. If the bit is $\textvisiblespace$, goes to the right and back to state q_2.

5.2 Multitape Turing Machines

Definition 5.4. A multitape Turing machine is a TM with several tapes, each with its own head. For k tapes, the transition function then becomes

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k \tag{5.1}$$

Each multitape Turing Machine has an equivalent single-tape Turing machine.

5.3 Enumerator

Definition 5.5. An enumerator is a Turing machine that always starts with a blank input on its tape, and that has an attached printer to print strings. Every time it wants to print a string to the list, it sends it to the printer. An enumerator does not have to halt, and may print an infinite list of strings. The language it enumerates is the collection of all strings that is printed out, and it can generate the strings in any order, possibly with repetitions.

Theorem 5.6. A language is Turing recognizable iff some enumerator enumerates it.

5.4 Non deterministic Turing Machine

A Turing machine is said to be non deterministic when its transition function is a probability distribution:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}) \tag{5.2}$$

Theorem 5.7. As a non deterministic TM can be represented as a tree, every non deterministic TM has an equivalent deterministic TM that goes through every possible state of the tree.

5.4.1 10th Hilbert's problem

"Devise a process according to which it can be determined by a finite number of operation that a given multivariate polynomial has an integer root."

For an univariate polynomial, we must check if there exists $a \in \mathbb{Z}$ such that $P(a) = 0$. We know also that the roots are in the interval $\left[-n\frac{c_{\max}}{c_1}, n\frac{c_{\max}}{c_1}\right]$. Therefore, there is a finite number of integer values to check, and we can evaluate the polynom $P(x)$ at each of those values. However, for a multivariate polynomial, it is impossible.
The rigorous formulation of the 10th Hilbert problem is to find a decider for $D = \cup_n D_n$ with

$$D_n = \{P \in \mathbb{Z}[x_1, \ldots, x_n] \mid \exists a \in \mathbb{Z}^n \ : \ P(a) = 0\} \tag{5.3}$$

5.5 Universal Turing Machine

Definition 5.8. A universal Turing machine is a TM that solves the following problem:

- Input: a description of the TM (T) and a description of a finite word on an initial tape (w);
- Output: $T(w)$, i.e. the finite content of the tape when T stops; and undefined if T never stops.

Decidability

6.1 Countable sets

- Two sets have the same number of elements iff they are in bijective relation;
- A set S is countable if it either is finite or has the same size as $\mathbb{N}$, e.g. $\mathbb{Z}, \mathbb{Q}, \mathbb{N}^d, \dots$;

6.2 Undecidable language

Theorem 6.1. $A_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$ (called halting problem) is undecidable.

This notation is the way to write a universal Turing machine with inputs $\langle M, w\rangle$, M being a Turing machine and w a string. Here is the proof of the theorem, by contradiction:
Let H be a decider for A_{TM}. Then

$$H(\langle H, w\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ accepts } w \\ \text{reject} & \text{if } M \text{ does not accept } w \end{cases} \tag{6.1}$$

Build a Turing machine D that, on input $\langle M\rangle$, runs H on $\langle M, \langle M\rangle\rangle$ and halts with $\overline{H(\langle M, w\rangle)}$, i.e. D returns the opposite of H:

$$D(\langle M\rangle) = \begin{cases} \text{accept} & \text{if } M \text{ does not accept } \langle M\rangle \\ \text{reject} & \text{if } M \text{ accepts } \langle M\rangle \end{cases} \tag{6.2}$$

But then, $D(\langle D\rangle)$ is a contradiction with itself. Define notation $\langle M\rangle$!

6.3 Unrecognizable language

Theorem 6.2. A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

Proof:

- $\Rightarrow$ is trivial;
- $\Leftarrow$

Let M_1 be a recognizer for A, and M_1 a recognizer for $\overline{A}$. We build M that runs M_1 and M_2 in parallel, one step at a time, on input w. If M_1 accepts, ACCEPT, and if M_2 accepts, REJECT.

Therefore, $\overline{A_{TM}} = \{\langle M, w\rangle | M$ is a TM and M does not accept $w\}$ is not Turing-recognizable.

Complexity Theory

7.1 Definitions

Definition 7.1. Let M be a deterministic TM that halts on all inputs. The running time, or time complexity, of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of length n.

Definition 7.2. Let $t : \mathbb{N} \to \mathbb{R}^+$ be the time complexit class TIME$(t(n))$, i.e. the collection of languages that are decidable by an $\mathcal{O}(t(n))$ time TM.

7.1.1 Example

Let $\{0^k1^k | k \geq 0\}$ be a string composed of k 0 followed by k 1. The goal is to check if the number of 0 is the same as the number of 1.
One method would be to start at the first element. If it is 0, cross it and go right until we find a 1. Then, go left until we find the first 0, and go again. Once we have no more 0 or 1, we check that it is also the case for the other character. If not, return FALSE. The complexity is $\mathcal{O}(k^2)$.
Another method is to cross every other 0 and do the same with the 1. With what is left, do it again (so one every four characters), and so on. At the end, compare if there is one character left in one of the strings. The complexity of this is $O(k\log(k))$.

- $\to$ Note: in the case where a counter variable is available, the overall complexity is not better, as the complexity to increment it is $\mathcal{O}(\log(k))$.

7.2 Complexity of multitape

Theorem 7.3. *Let $t(n) \geq n$, as we can reasonably assume that we need to go through the whole input to find the solution. Every $t(n)$ time multitape TM has an equivalent $\mathcal{O}(t^2(n))$ time single-tape TM.*

7.3 Complexity of non deterministic TM

Let N be a nondeterministic TM that is a decider. The running time of N is the function $f : \mathcal{N} \to \mathcal{N}$ where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.

Theorem 7.4. Let $t(n) \geq n$. Every $t(n)$ time nondeterministic single-tape TM has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape TM.

This is because the number of nodes is in $\mathcal{O}(b^{f(n)})$, and the number of operations for one node is $\mathcal{O}(n + f(n))$. b is the number of possibilities at each node. This gives a total complexity of

$$\mathcal{O}\left((n + f(n))b^{f(n)}\right) = \mathcal{O}\left(2^{\log_2(b)(\log_b(f(n))+f(n))}\right) = 2^{\mathcal{O}(f(n))} \tag{7.1}$$

The result is the same for a multitape or single-tape TM, as (from 7.3):

$$2^{\mathcal{O}(f(n))} \to \left(2^{\mathcal{O}(f(n))}\right)^2 = 2^{\mathcal{O}(f(n))} \tag{7.2}$$

7.4 The class P

Definition 7.5. **P** is the class of languages that are decidable in polynomial time on a deterministic single-tape TM.

$$\mathbf{P} = \cup_k TIME(n^k) \tag{7.3}$$

Definition 7.6. **coP** is the class of languages such that their complement is in **P**.

$$\mathbf{coP} = \{L | \overline{L} \in \mathbf{P}\} \tag{7.4}$$

If $L \subset \mathbf{P}(\Gamma^*)$, then $\overline{L} = \mathbf{P}(\Gamma^*) \setminus L$, where L is a language, and $\mathbf{P}(\Gamma^*)$ is the class **P** of argument Γ^*, Γ is the alphabet and the star means the set of words of this alphabet.

$\rightarrow$ Note: any problem in **coP** is in **P**, as it is only the inverse problem.

$$\implies \mathbf{P} = \mathbf{coP}$$

7.5 The class NP

Definition 7.7. A verifier for a language L is an algorithm V where

$$L = \{w | V \text{ accepts } \langle w, c \rangle \text{ for some string } c\} \tag{7.5}$$

Definition 7.8. The class **NP** is the class of languages that have a polynomial time verifier.

Theorem 7.9. A language is in **NP** iff it is decided by some nondeterministic polynomial time TM.

We write $NTIME(t(n))$ the set of languages that are decided by a $\mathcal{O}(t(n))$ time nondeterministic TM, and so

$$\mathbf{NP} = \cup_k NTIME(n^k) \tag{7.6}$$

Theorem 7.10.

$$\mathbf{P} \subset \mathbf{NP} \tag{7.7}$$

This is obvious: any problem whose solution can be found in polynomial time has a verifier in polynomial time.

7.5.1 Example – SAT

The SAT problem consists in finding boolean variables $\{x_n\}_{n\geq 0}$ such that a combination of boolean operators applied to them returns 1. For example,

$$\exists?\{x_1, x_2, x_3\} \text{ such that } \Big((x_1 \wedge x_2) \vee \overline{(x_2 \vee x_3)}\Big) \wedge x_3 = 1 \tag{7.8}$$

It is very hard to find a solution, but easy to check.

Conjunctive normal form

We define the conjunctive normal form as

$$\phi = \bigwedge_i c_i \qquad c_i = \bigvee_j l_j \qquad l_j = \begin{cases} x_k \\ \overline{x_k} \end{cases} \tag{7.9}$$

We call c_i the clauses and l_j the literals.
The 3SAT problem add the constraint that there are at most 3 l_j per clause.

The SAT problem is NP-complete (see section 7.7).

7.6 The EXPTIME class

EXPTIME is the class of languages that are decidable in exponential time on a deterministic single-tape TM.

$$\textbf{EXPTIME} = \cup_k TIME(2^{n^k}) \tag{7.10}$$

Theorem 7.11. **P** $\subset$ **NP** $\subset$ **EXPTIME**

7.7 Polynomial time reduction

Definition 7.12. A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial time TM exists that halts with $f(w)$ on its tape when started on any input w.

Definition 7.13. A language A is polynomial time reducible to a language B, written $A \leq_P B$, if there exists a polynomial time computable function f such that, for every w, $w \in A \Leftrightarrow f(w) \in B$. We call f the reduction of A to B.

$$A \leq_P B \text{ and } B \in \mathbf{P} \Longrightarrow A \in \mathbf{P} \tag{7.11}$$

Definition 7.14. A language B is NP-complete if $B \in$ **NP** and every $A \in$ **NP** is reducible to B in polynomial time.
If B is NP-complete and $B \in$ **P**, then **P** $=$ **NP**.

$$\forall B \text{ NP-complete and } C \in \mathbf{NP},\ B \leq_P C \Longrightarrow C \text{ is NP-complete} \tag{7.12}$$

Je sais pas trop quoi faire des slides 21 à 26 du dernier cours.