Université catholique de Louvain

LINMA2111

Discrete Mathematics II : Algorithms and Complexity

Course notes

Students:
Nicolas Boutet, Mathieu Dath, Cyril de Bodt, Floran Hachez, Kathleen Hemmer, Anne-Laure Levieux, Dounia Mulders, Gauthier Rodaro, Félicien Schiltz, Damien Scieur, Mélanie Sedda, Benoît Sluysmans, Kim Van Den Eeckhaut, Quentin Vanderlinden & Hélène Verhaeghe

2015

Contents

A. Reminders : Asymptotic notation

TEMPORARY - CM1

Let $\begin{cases} f: & \mathbb{N} \to \mathbb{R}^+ \\ g: & \mathbb{N} \to \mathbb{R}^+ \end{cases}$

- $f \in \mathcal{O}(g)$, often written $f(n) = \mathcal{O}(g(n))$, means: $\exists n_0, c : \forall n \geq_0, f(n) \leq c \cdot g(n)$.
- $f \in \Theta(g)$ $\begin{array}{l} \Leftrightarrow f \in \mathcal{O}(g) \text{ and } g \in \mathcal{O}(f) \\ \Leftrightarrow \exists n_0, c_1, c_2 : \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \end{array}$ (f and g are equivalent in complexity)
- $f \in o(g) \Leftrightarrow f \in \mathcal{O}(g)$ but $f \notin \Theta(g)$
- $f \in \Omega(g) \Leftrightarrow g \in \mathcal{O}(f)$
- $f \in \omega(g) \Leftrightarrow g \in o(f)$

Example A.1.

$$\begin{aligned} n^2 &= \mathcal{O}(n^2) \\ &= o(n^3) \\ &= \Theta\left(\frac{n(n-1)}{2}\right) \\ &= \Omega(n \log(n)) \end{aligned}$$

B. Sorting Algorithms

We will start with illustrating some ideas on sorting algorithms. The sorting problem may be defined by its input and output. Hence our goal is to find some algorithms such that:

Code B.1: Inputs and outputs of sorting algorithms

```
INPUT  : array T = (T(1), T(2), ..., T(n)) of n numbers
OUTPUT : array T sorted
```

1. Selection Sort

First, we consider a naïve algorithm: **Selection Sort**. This first algorithm can be formulated as follows:

Code B.2: Pseudo-code of the selection sort algorithm

```
for i from 1 to n - 1
    select the smallest element in the sub array from index i to n
    swap it with element at index i
```

What about the time complexity of this algorithm?

$$\begin{aligned}\text{Time} &= \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) \\ &= \Theta(n^2)\end{aligned}$$

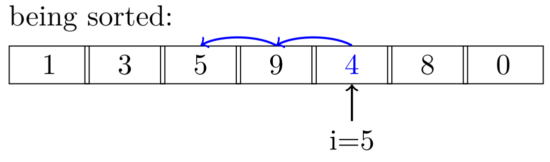
Remark. The complexity of this algorithm is $\Theta(n^2)$ for each instance. We have Worst case = Best case = Average case time complexity = $\Theta(n^2)$.

2. Insertion Sort

Code B.3: Pseudo-code of the insertion sort algorithm

```
for i from 2 to n % loop invarient : T(1) ... T(i-1) are sorted
    shift T(i) to the left by successive swap until well placed
```

Example B.1. We can consider a small exemple, the first four elements of the array being sorted:



Regarding the time complexity, we have:

- in worst case, at every step we swap until the beginning of the array: worst-case complexity $= \Theta(1) + \ldots + \Theta(n) = \Theta(n^2)$. This case corresponds to an instance sorted in decreasing order.
- Best-case corresponds to an array already sorted: $\Theta(n)$
- Average-case: instances come with uniform probability distribution over all possible orders ($= n!$ orders). Then:

$$\begin{aligned}
\mathbb{E}[Time] &= \mathbb{E}[\sum_{i=2}^{n} \Theta(t_i)] && \text{With } t_i \text{ the number of swaps needed for } T(i). \\
&= \Theta\left(\sum_{i=2}^{n} \mathbb{E}[t_i]\right) && \text{By linearity of } \mathbb{E}. \\
&= \Theta\left(\sum_{i=2}^{n} \frac{i}{2}\right) \\
&= \Theta(n^2)
\end{aligned}$$

In *some* applications, the relevant probability distribution is not uniform but biased towards almost-sorted instances.
$\Rightarrow$ Average-case can be $o(n^2)$.

3. Quick Sort

This third algorithm is based on the idea of "Divide-and-Conquer" which solves the sorting problem in the following manner:

- Split into small sorting problems.
- Recombine the outputs, which gives the sorted instance.

In particular, in this case:

Code B.4: Pseudo-code of the quick sort algorithm

```
pivot T(1)
T_low = [T(i) : T(i) < pivot && i > 1]
T_high = [T(i) : T(i) >= pivot && i > 1]
quicksort T_low
quicksort T_high
T = [T_low pivot T_high] (The pivot being in $T_{low}$)
```

About the complexity of this algorithm:

- Worst case: when instance is already sorted !

  In this case: $\left.\begin{array}{ll} T_{low} & = 1 \text{ entry.} \\ T_{high} & = n-1 \text{ entries.} \end{array}\right]$ $\Rightarrow$ maximally unbalanced subproblems. Time $= \Theta(n^2)$

  Indeed: construct $T_{low}/T_{right} = \Theta(n)$, hence we have the worst-case recurrence equation:

$$\begin{aligned} t_n &= t_1 + t_{n-1} + \Theta(n) \\ &= t_1 + (t_1 + t_{n-2} + \Theta(n-1)) + \Theta(n) \\ &= t_1 + (t_1 + (t_1 + t_{n-3} + \Theta(n-2)) + \Theta(n-1)) + \Theta(n) \\ &= \dots \\ &= n \cdot t_1 + \Theta(n^2) = \Theta(n^2) \end{aligned}$$

- Average-case complexity:

$$\begin{aligned} t_n &= t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n) \qquad \text{(B.1)} \\ &= 2 \cdot t_{\frac{n}{2}} + \Theta(n) \\ &= 2\left(2 \cdot t_{\frac{n}{4}} + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= 2\left(2 \cdot \left(2 \cdot t_{\frac{n}{8}} + \Theta(\frac{n}{4})\right) + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= \dots \\ &= 2^{\log_2(n)} \cdot t_1 + \Theta(n \log_2(n)) \\ &= \Theta(n \log_2(n)) \end{aligned}$$

Remark. We will denote $\log_2$ by log, since $\log_a(n) \in \Theta(\log_b(n))$, $\forall a, b > 1$.
Remark. In $t_n = t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n)$,

$$\begin{aligned} \mathbb{E}[t_n] &= \mathbb{E}[t_{|T_{low}|}] + \mathbb{E}[t_{|T_{high}|}] + \Theta(n) \text{ would be more correct.} \\ &\cong t_{\mathbb{E}[|T_{low}|]} + t_{\mathbb{E}[|T_{high}|]} + \Theta(n) \text{ : in fact OK.} \end{aligned}$$

Now, what if you don't know that your instances are uniformly distributed? This leads to the next algorithm.

4. Randomized Quick Sort

Here, the first step of the algorithm is to shuffle entries randomly. It is equivalent to picking a random entry as pivot, instead of $T(1)$.

Now on any instance, Randomized Quick Sort runs in expected time $\Theta(n\log(n)) \Rightarrow$ Worst-case expected time of this random algorithm is also $\Theta(n\log(n))$.

One advantage of randomization: Robin Hood effect, i.e. you take from rich instances and give to the poor. Now, everybody can be unlucky from time to time but lucky most of the time.

Can we reach worst-case complexity $\mathcal{O}(n\log(n))$ deterministically?
The answer is yes, as we will see in the next section (Merge Sort).

5. Merge Sort

This algorithm is also based on Divide-and-Conquer approach.

Code B.5: Pseudo-code of the merge sort algorithm

```
T_left = [T(i) : i <= n/2]
T_right = [T(i) : i > n/2]
mergesort T_left
mergesort T_right
T = [T_left T_right]
```

Time of merge is $\Theta(n)$, so we have the recurrence equation:

$$\begin{aligned} t_n &= 2 \cdot t_{\lceil \frac{n}{2} \rceil} + \Theta(n) \\ &= \Theta(n \log(n)) \text{ (see equation (B.1))} \end{aligned}$$

Which one to use?

- Random Quick Sort:
  - $\ominus$ You need to generate randomness: not so easy for a computer!
  - $\ominus$ You can be unlucky.
  - $\oplus$ Hidden constants are smaller if implemented efficiently.
  - $\oplus$ can be implemented "in place": on T itself $\rightarrow$ no need for extra memory for intermediate variables.

TEMPORARY - CM2

6. Stack v.s. Queue

In one precedent section, we have seen the QuickSort algorithm (Hoare, 1962). This algorithm is often used in practice due to the low constant in its complexity. We have also seen that there is some randomness in this algorithm. One may now wonder if there is a deterministic algorithm in $\mathcal{O}(n \log n)$ (worst case complexity) which sorts "in place" (meaning that it uses no extra memory).

Fortunately for the people who really hate randomness, the answer is yes, but we need to be smart about the way we keep track of the information. In other words, we need a good data structure.

Let us make a little digression about two well studied data structures, stacks and queues. Both are ordered lists of entries with the same static structure but different ways to access/modify it. The difference is graphically explained on Figure B.1.

We define a data structure as a static structured set (like a list, a tree, etc.) with a list of operations (like read, insert, etc.). It can thus be seen as an abstract mathematical concept.

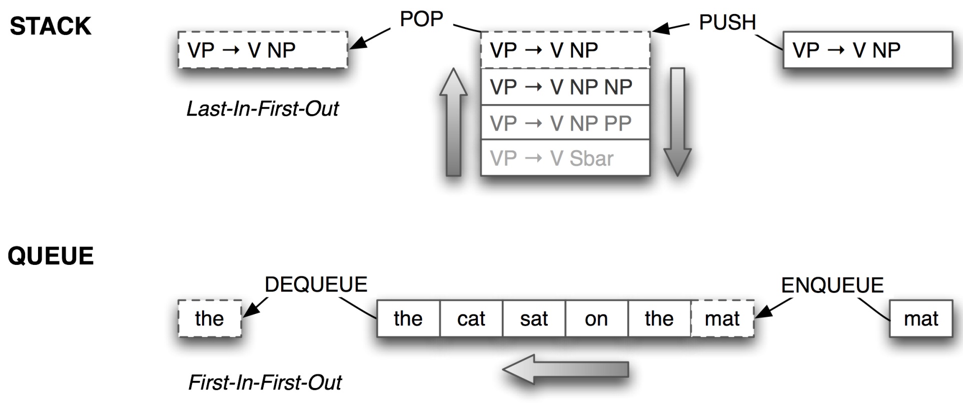


Figure B.1.: Graphic representations of the stack and the queue structures. Elements are inserted using the Push/Enqueue action and removed using the Pop/Dequeue action.

a. A data structure for a deterministic QuickSort

If we get back to our problem, the structure we need is a priority queue: a set of numbers with the following operations

- Insert
- Extract Max

An example of the use of such a structure is a todo list where the elements are tasks and where each task is assigned a priority (which can be interpreted as a number).

If we have an implementation for a priority queue, then we have a sorting algorithm for [T(1) T(2) $\cdots$ T(n)], we just need to:

1. Insert T(1), T(2), ..., T(n) to an empty priority queue.
2. Extract Max n times

The complexity of such an algorithm is $n\,(\text{cost(Insert)} + \text{cost(Extract Max)})$.

b. Potential structures for this sorting algorithm

We need a data structure to implement the sorting algorithm we just described.

i. Unordered table

Using an unordered table, the operations complexity are

- Insert: $\mathcal{O}(1)$
- Extract Max: $\Theta(n)$

The sorting algorithm associated to this implementation has thus the complexity $\Theta(n^2)$.
⇒ This is Selection Sort.

ii. Ordered table

Using an ordered table, the operations complexity are

- Insert: $\mathcal{O}(n)$ [*cost of shifting the entries*]
- Extract Max: $\mathcal{O}(1)$

The sorting algorithm associated to this implementation has thus the complexity $\mathcal{O}(n^2)$.
$\Rightarrow$ This is Insertion Sort.

iii. "Heap"

Using a heap, the operations complexity are

- Insert: $\mathcal{O}(\log n)$
- Extract Max: $\mathcal{O}(\log n)$

The sorting algorithm associated to this implementation has thus the complexity $\mathcal{O}(n \log n)$.

c. Heap structure

A heap is a tree with the following properties:

- essentially binary complete (only the last level may be incomplete - as it is filled from left to right)
- entry$(x) \leq$ entry(father(x)) $\forall$ node $x \neq$ root

An example of a heap is shown in Figure B.2.

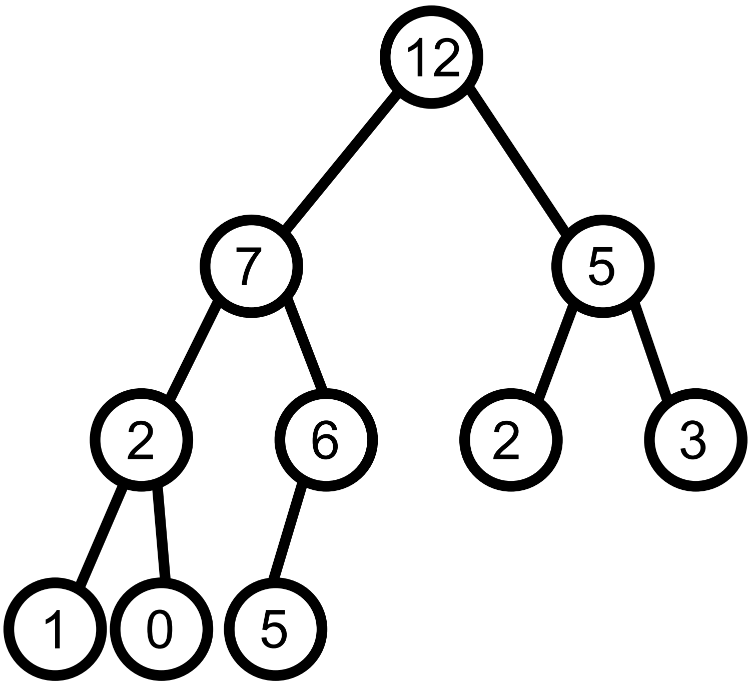


Figure B.2.: Example of a heap

i. Insert

To insert a new node into the heap, add it to the last level in the first free position and swap it with its father until the heap is properly re-established.
$\Rightarrow \mathcal{O}(\log n)$

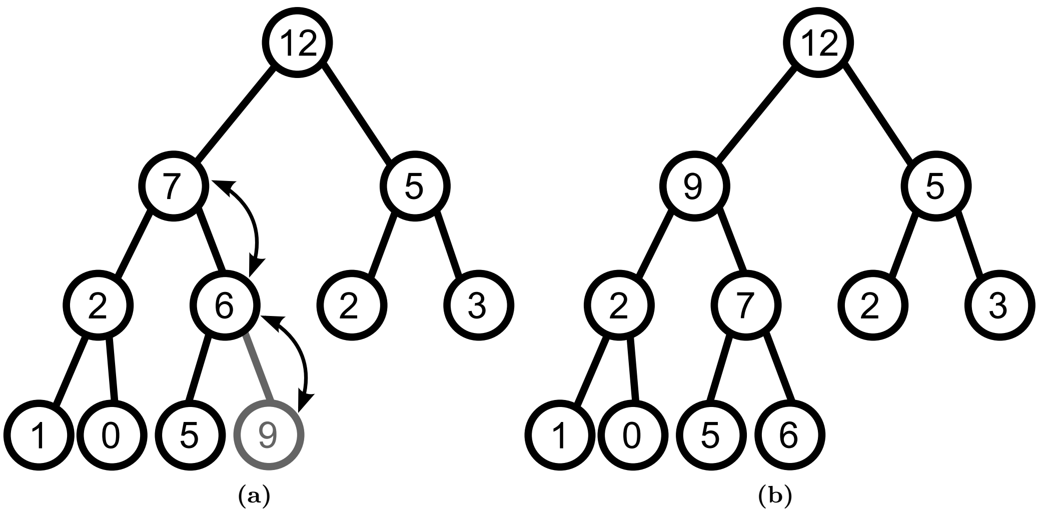


Figure B.3.: A heap before/after adding one element

ii. Extract Max

To extract the maximal node from the heap, one needs to:

1. Swap root (the max entry) with last entry
2. Remove the max entry
3. Swap new root with its *largest* child until the heap properly is restored.

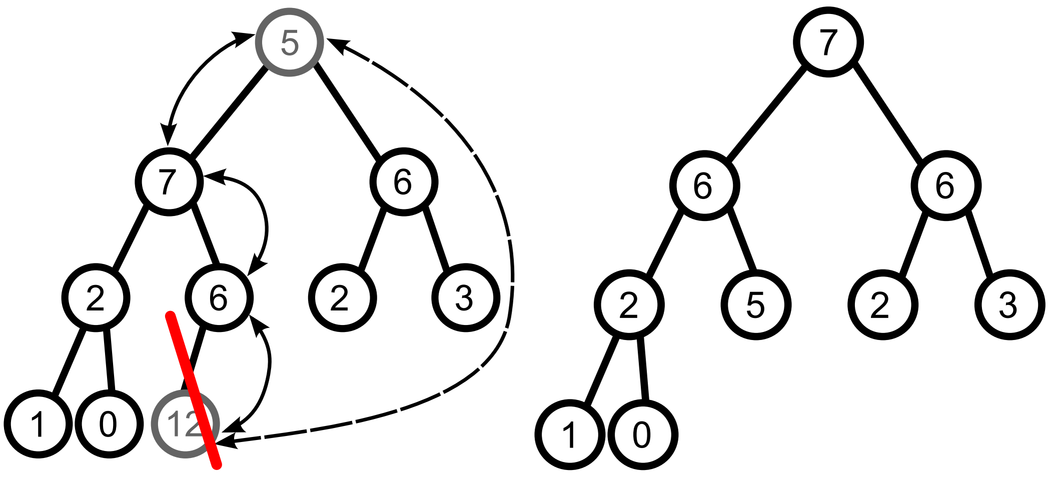
$\Rightarrow \mathcal{O}(\log n)$

The resulting sorting algorithm is called the HeapSort (Williams, 1964) and has a complexity of $\mathcal{O}(n \log n)$.
Remark. The heap may be stored as an array:

Heap (T)	12	7	6	2	6	2	3	1	0	5
Index	1	2	3	4	5	6	7	8	9	10

with the following properties:

$$\begin{aligned} \text{LeftChild}(T(i)) &= T(2i) \\ \text{RightChild}(T(i)) &= T(2i+1) \end{aligned}$$



(a) **(b)**

Figure B.4.: A heap before/after removing the largest entry

$$\text{Father}(T(i)) = T\left(\left\lfloor \frac{i}{2} \right\rfloor\right)$$

⇒ We can thus sort in place!
Remark. Create a heap from an input can be done in $\mathcal{O}(n)$.
Remark. There exist a correspondence between a heap and a priority queue.
Remark. The implementation of a data structure often offer trade-offs: one operation can be cheap if another is costly.

7. Can we do better?

Now that we have found a way to deterministically sort an array in $\mathcal{O}(n \log n)$, we can ask ourselves: can we sort in $o(n \log n)$ (i.e. in a complexity strictly better than $n \log n$) for either the worst case or the average case complexity, if we assume nothing on the entries (i.e. all we can test is "compare two entries", "$T(i) \leq T(j)$")?

Example B.2. 20-questions game With 20 Yes-No questions, you can guess a number between 1 and 1000000, but not an arbitrary number between 1 and 2000000 ($\geq 2^{20}$), no matter the questioning strategy.

In order to answer our question, let us introduce the following theorems.

Theorem B.1. A binary tree (thus with at most 2 children for each node) with N leaves has at least $\lfloor \log N \rfloor$ levels.

Proof Label each edge 0 or 1 (see Figure B.5). Every leaf is uniquely identified by a binary word encoding the path root → leaf.
As we cannot uniquely identify N nodes with binary words all of length $< \lfloor \log N \rfloor$, there is at least one path of length $> \log N$.

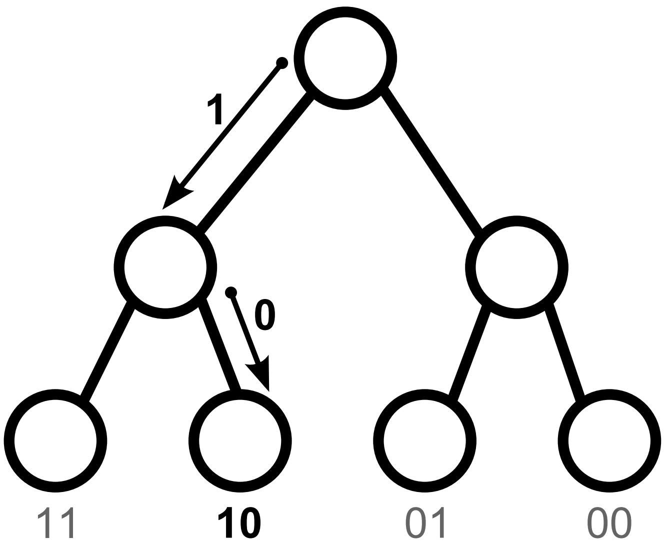


Figure B.5.: Example of a binary tree

QED

Theorem B.2. No deterministic sorting algorithm can sort n entries (with just comparisons) in $o(n \log n)$ worst case scenario.

Proof Build the decision tree (see Figure B.6) from the sequence of questions to be asked to the data (the comparisons). The number of leaves of this tree is $n! = n(n-1)(n-2)...1$.

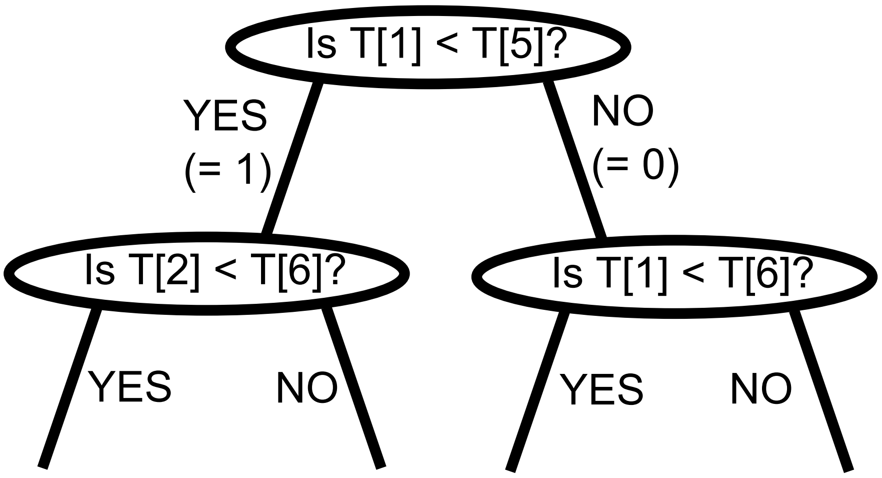


Figure B.6.: Decision tree

This is the number of possible orders of the input array, each of which will require a different sequence of decisions (leading to different answers to the questions, as the algorithm is deterministic: the same answers leads to the same actions on the data).

$\Rightarrow \exists$ one path of length $\log(n!)$ and thus $\log(n!)$ questions.

Using the Stirling approximation $n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$, we obtain $\log(n!) = n \log n$. We have thus $\Omega(n \log n)$ as worst case complexity.

QED

This last theorem answers our question for the worst case scenario, but can the average-case complexity be better? The following theorems will answer that question.

Theorem B.3 (Shannon 1948)**.** Let an N-set be endowed with the probabilistic distribution $\rho_1, \rho_2, ..., \rho_N$. Label every entry with a binary word (in a prefix way: no word is a prefix to another). Then the expected length (weighted by (ρ_i)) is $\geq \sum_{i=1}^{N} -\rho_i \log \rho_i$.

Theorem B.4. The average length of a path root $\rightarrow$ leaf of a N-leaf binary tree, with uniform distribution on leaves, is at least $\sum_{i=1}^{N} -\frac{1}{N} \log \frac{1}{N} = \log N$.

Theorem B.5. The average-case complexity of *any* deterministic sorting algorithm using comparisons is $\Omega(n \log n)$.

Proof Apply previous theorems on decision tree.

QED

Remark. Randomized algorithms cannot be better either.

As we now know, with our current hypothesis, we cannot do better than $\Omega(n \log n)$. But with supplementary knowledge, we can beat the $\Omega(n \log n)$ bound.

If we know that the entries $\in \{1, 2, ..., k\}$, we can define the algorithm Counting Sort on $[\mathrm{T}(1)\ \mathrm{T}(2)\ \cdots\ \mathrm{T}(n)]$.

8. Counting Sort

1. Entries count:
   - Count how many entries $= 1 : U(1)$
   - Count how many entries $= 2 : U(2)$
   - $\vdots$
   - Count how many entries $= k : U(k)$
2. Create $V = [U(1), U(1) + U(2), U(1) + U(2) + U(3), ..., U(1) + U(2) + \cdots + U(k)]$.
3. Entries copy:
   - Copy entries $= 1$ to final array $W(1), ..., W(U(1))$.

- Copy entries $= 2$ to final array $W(U(1)+1), ..., W(U(1)+U(2))$.
- $\vdots$
- Copy entries $= k$ to final array $W(U(1)+\cdots+U(k-1)+1), ..., W(U(1)+\cdots+U(k))$.

We have thus the following relations (for $i = n, ..., 1$):

$$\begin{aligned} W(V(T(i))) &= T(i) \\ V(T(i)) &= V(T(i)) - 1 \end{aligned}$$

a. Complexity

The worst case complexity of this algorithm is $\mathcal{O}(k+n)$, computed as follow:

$$\begin{cases} \text{2 passes on T: } \mathcal{O}(n) \\ \text{Creation of U: } \mathcal{O}(k) \end{cases} \quad \mathcal{O}(k+n)$$

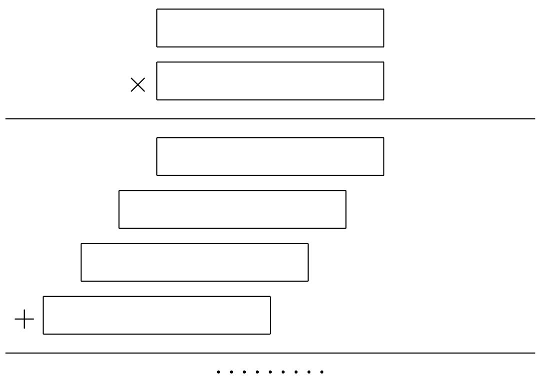
Remark. Often an entry is just a key to satellite data (a file, a picture, a book, etc.). In those cases, Counting Sort is stable: the order of two entries with the same key values is preserved. They are not swapped in the final array W.

C. Divide and Conquer

TEMPORARY - CM3

1. Multiplication of 2 large integers

How to multiply 2 large integers (of n digits each)?
Elementary method : use the grid method for written multiplication :



: $\Theta(n^2)$

We can do better!

$$
\begin{aligned}
x \times y &= (\underbrace{x_0}_{n/2 \text{ digits}} + 10^{\lfloor n/2 \rfloor} \underbrace{x_1}_{n/2}) \times (\underbrace{y_0}_{n/2} + 10^{\lfloor n/2 \rfloor} \underbrace{y_1}_{n/2}) \\
&= x_0 \times y_0 + 10^{\lfloor n/2 \rfloor}(x_1 \times y_0 + x_0 \times y_1) + 10^{2\lfloor n/2 \rfloor}(x_1 \times y_1)
\end{aligned}
$$

Complexity : $t_n = 4t_{\lfloor n/2 \rfloor} + \Theta(n)$
Solution : if we assume $n \approx 2^k$, we have :

$$
\begin{aligned}
t_n &= 4t_{n/2} + \Theta(n) \\
&= 16t_{n/4} + 4\Theta(n/2) + \Theta(n) \\
&= 64t_{n/8} + \underbrace{16\Theta(n/4)}_{\Theta(4n)} + \underbrace{4\Theta(n/2)}_{\Theta(2n)} + \underbrace{\Theta(n)}_{\Theta(n)} \\
&= \ldots \\
&= 2^{2k} \underbrace{t_{n/2^k}}_{\mathcal{O}(1)} + \Theta(\underbrace{(2^{k-1} + 2^{k-2} + \ldots + 1)}_{2^k - 1} n) = \Theta(n^2)
\end{aligned}
$$

This is not better.

a. Master Theorem

This kind of recurrence is solved by the *Master Theorem*.

Theorem C.1 (Master)**.** For the recurrence $t_n = at_{n/b} + f(n)$ (or $t_{\lfloor n/b \rfloor}$ or $t_{\lceil n/b \rceil}$) :

1. If $f(n) \in \mathcal{O}\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $t_n \in \Theta\left(n^{\log_b a}\right)$;
2. If $f(n) \in \Theta\left(n^{\log_b a}\right)$, then $t_n \in \Theta\left(n^{\log_b a} \log n\right)$;
3. If $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$, $af(n/b) \leq cf(n)$ for some $c < 1$ and n is large enough , then $t_n \in \Theta\left(f(n)\right)$.

Example C.1. We can give the following examples

- $t_n = 2t_{n/2} + \Theta(n)$: $a = 2$, $b = 2$ and $\log_2 2 = 1$
  By 2 from Theorem C.1, $f(n) \in \Theta\left(n^{\log_b a}\right) \rightarrow t_n \in \Theta(n \log n)$
- $t_n = 4t_{n/2} + \Theta(n)$: $a = 4$, $b = 2$ and $\log_2 4 = 2$
  By 1, $f(n) = \Theta\left(n^{\log_2 4 - 1}\right) \rightarrow t_n = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^2\right)$

b. Karastuba-Ofman

It is possible to make only 3 multiplications of $\frac{n}{2}$-digits numbers.

Code C.1: `Multiply(x,y)` (Karastuba-Ofman, 1962)

```
x=x_0+(10^floor(n/2))*x_1
y=y_0+(10^floor(n/2))*y_1
M=(x_0+y_0)*(x_1+y_1)
x*y={x_0*y_0}+10^(n/2)*(M-x_0*y_0-x_1*y_1)+10^(2(n/2))*x_1*y_1
```

Time complexity :

$$t_n = 3t_{n/2} + \Theta(n) \Rightarrow t_n = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.585}\right)$$

We can do even better!→ Schönhage-Strassen (1971) : $\mathcal{O}(n \log(n) \log\log(n))$.

2. Multiplication of 2 matrices

How can we multiply two n-by-n matrices?
Elementary algorithm : $\Theta(n^3)$ elementary operations.

a. Block Multiplication

Is block multiplication better?

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \qquad B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

$$\rightarrow AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Time complexity : 8 products of $\frac{n}{2}$-by-$\frac{n}{2}$ matrices.

$$t_n = 8t_{n/2} + \Theta(n^2) = \Theta\left(n^{\log_2 8}\right) = \Theta\left(n^3\right)$$

This is not better.

b. Strassen's Algorithm

We can do it with only 7 products (Strassen's algorithm, 1969) :

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

$$\rightarrow AB = \begin{pmatrix} M_2 + M_3 & M_1 + M_2 + M_5 + M_6 \\ M_1 + M_2 + M_4 - M_7 & M_1 + M_2 + M_4 + M_5 \end{pmatrix}$$

Time complexity : $t_n = 7t_{n/2} + \Theta(n^2) = \Theta\left(n^{\log_2 7}\right) = \Theta\left(n^{2.81}\right)$
We can do even better!$\rightarrow$ Coppersmith-Winograd (1981) : $\Theta(n^{2.376})$
Best one? Unknown but $\Omega(n^2)$.

c. Matrix Inversion

What is the cost of matrix inversion?
Elementary methods : $\Theta(n^3)$

Theorem C.2. Matrix inversion and multiplication have the same complexity

Proof Let $I(n)$ be the (worst-case) complexity of inversion (i.e. of the best possible algorithm) and let $M(n)$ be the (worst-case) complexity of multiplication (i.e. of the best possible algorithm).
We want to show that $I(n) \in \Theta(M(n))$.

1. $M(n) \in \mathcal{O}(I(n))$? We reduce multiplication to inversion.

$$D = \underbrace{\begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}}_{3n*3n} \rightarrow D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}$$

$$\rightarrow M(n) = \mathcal{O}(I(3n)) \qquad\qquad \Rightarrow M(n) \in \mathcal{O}(I(n))$$

Since $I(n) \in \mathcal{O}(n^3)$, $I(3n) \in \mathcal{O}(I(n))$. (Because n is multiplied by 27)

$(*)$ $I(n) \in \mathcal{O}(M(n))$?

We reduce inversion to "a few" multiplications.

$(*)$ First assume $A = A^T \succcurlyeq 0$ (symetric, positive-definite).

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \quad , \quad \begin{matrix} B = B^T \succcurlyeq 0 \\ C = C^T \succcurlyeq 0 \end{matrix}$$

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^TS^{-1}CB^{-1} & -B^{-1}C^TS^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}$$

with S = Schur's Complement $= D - CB^{-1}C^T$
There are:

- 2 inversions: B^{-1}, S^{-1} of size $\frac{n}{2}$
- 4 multiplications: CB^{-1}, $C^T(CB^{-1})$, $S^{-1}(CB^{-1})$, $(CB^{-1})^T\,[S^{-1}(CB^{-1})]$

$$\begin{aligned} \Rightarrow I(n) &\leq 2I(\frac{n}{2}) + 4M(\frac{n}{2}) + \Theta(n^2) \\ &= 2I\left(\frac{n}{2}\right) + \Theta(M(n)) && (\text{because} n^2 \in \mathcal{O}(M(n))) \\ &= \Theta(M(n)) && (\text{Master theorem with } a = b = 2) \end{aligned}$$

$\Rightarrow I(n) \in \mathcal{O}(M(n))$

$(*)$ For general invertible matrices:
$A^{-1} = \underbrace{(A^TA)^{-1}}_{\succcurlyeq 0} A^T$ $\rightarrow$we can apply the trick to (A^TA). There is one more multiplication but it does not change anything.

QED

3. Random Select algorithm

How to find the ith smallest entry of an array?

Example C.2. First possible algorithm:

- $i = 1$ minimum: $\Theta(n)$
- $i = n$ maximum: $\Theta(n)$
- $i = \lfloor \frac{n}{2} \rfloor$ median: $\mathcal{O}(n \log n)$: sort then read $T(\lfloor \frac{n}{2} \rfloor)$

Can we do better?
Idea: Do random Quicksort but save effort by not sorting one of the two subarrays.

If we make the assumption that all entries are different, we have the following algorithm.

Code C.2: Selection (T,i)

```
Selection (T,i) (assumption : all entries are different)
pivot = random entry of T = T(j) for random $j\in \{1.. n\}$
T_{low} = entries < pivot
T_{hight} = entries > pivot
if |T_{low} | = i-1 then Selection (T,i) := pivot
if |T_{low} | < i-1 then Selection (T,i) := Selection (T_{hight}, ↩
   i-|t_{low}| -1)
if |T_{low} | > i-1 then Selection (T,i) := Selection (T_{low}, i)
```

(Worst-case) expected time= t_n

$$\begin{aligned}
t_n &\leq \mathbb{E} t_{\max(|T_{low}|,|T_{high}|)} + \Theta(n) && \Theta(n) \rightarrow \text{finding } T_{low}, T_{high} \text{ and } |T_{low}| \\
&= \sum_{k=0}^{n-1} Prob(|T_{low}| = k) t_{\max(k,n-k-1)} + \Theta(n) && \Theta(n) \leq an \text{ (check following theorem)} \\
&= 2 \sum_{k=\lfloor (n-1)/2 \rfloor}^{n-1} \frac{1}{n} t_k + \Theta(n)
\end{aligned}$$

Clever way to solve it: guess and check if it is correct (many terms otherwise).

Theorem C.3. $t_n = \Theta(n)$

Proof Assume the term $\Theta(n) \leq an$ for n large enough.
Assume $t_n \leq cn$ for some $c > 0$ (to be chosen later), all large enough n.
Proof by induction: assume it's time for $k \leq n-1$: $t_k \leq c_k$.

Show that it is true for n: $t_n \leq c_n$?

$$\begin{aligned}
t_n &\leq 2 \sum_{k=(n-1)/2}^{n-1} \frac{t_k}{n} + an \\
&\leq 2 \sum_{k=(n-1)/2}^{n-1} \frac{ck}{n} + an \\
&= 2\frac{c}{n} \overbrace{\left[\frac{(n-1)(n-2)}{2} - \frac{\frac{n-1}{2}\left(\frac{n-1}{2}-1\right)}{2} \right]}^{\leq \frac{3n^2}{8}} + an \\
&= \frac{3cn}{4} + a_n
\end{aligned}$$

which is $\leq cn$ if we choose c such that $\frac{3c}{4} + a < c \Rightarrow c > 4a$

QED

TEMPORARY - CM4

4. Deterministic Select

In the previous part, we determine the Random Select algorithm in which we can find the i^{th} smallest entry of an array T in $\Theta(n)$ (worst-case expected time). This algorithm was based on a variance of the quick sort.

The question asked is : Can we do it deterministic $\mathcal{O}(n)$ time?

To answer it, we must find a good pivot deterministically. This lead us to the following Deterministic Select (Code C.3).

Code C.3: Pseudo-code of the Determinisic Select algorithm

```
DeterministicSelect(T,i) :
    Split T into ceiling(n/5) arrays of maxsize = 5

    foreach array :
        find median % ceiling(n/5) medians = Theta(ceiling(n/5))

    Pivot = DeterminisiticSelect(medians, 1/2 * ceiling(n/5))
        % find the median of the medians computed

    Tlow = entries < pivot
    Thight = entries > pivot

    if (i <= |Tlow|)
        return DeterministicSelect(Tlow,i)
    if (i == |Tlow| + 1)
        return pivot
    if (i > |Tlow| + 1)
```

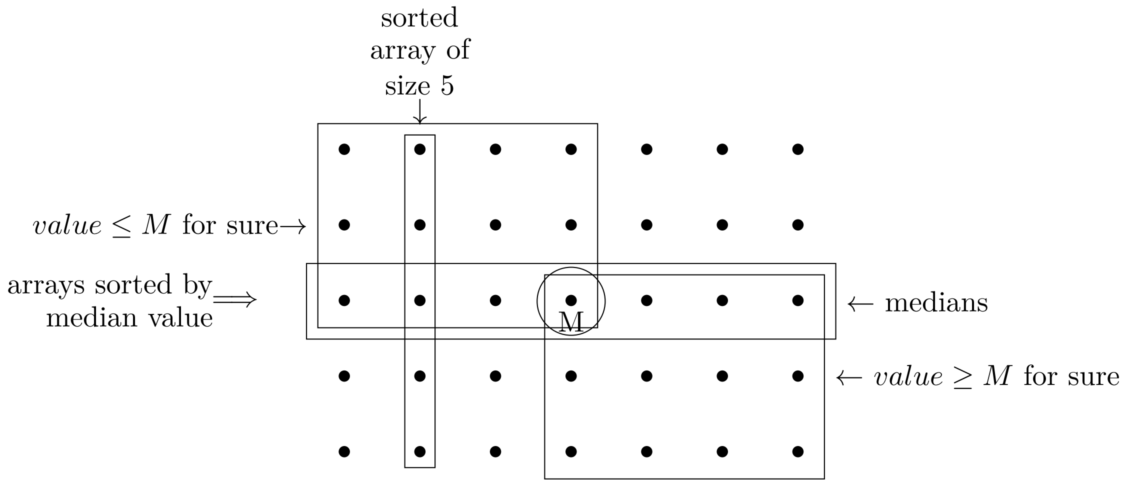


Figure C.1.: Graphical view of the selection of the pivot (point M) in the `DeterministiSelect(T,i)`

```
18          return DeterministicSelect(Thigh,i-|Tlow|-1)
```

Is this a good pivot?

As our pivot is the median of the different medians, we can say for sure that for the half of the array, the pivot is higher than the 3 smallest values (the median and the values smaller than the median). Thus we can affirm that the pivot is higher than at least $3 \times \frac{1}{2} \times \lceil \frac{n}{5} \rceil \approx \frac{3n}{10}$.

We can do the same reasoning for the value higher than the pivot and also affirm that at least $3 \times \frac{1}{2} \times \lceil \frac{n}{5} \rceil \approx \frac{3n}{10}$ values are higher for sure.

A simple drawing (Fig. C.1) can easily illustrate this reasoning.

Knowing that, we can now compute the complexity of this algorithm by first writing down the recurrence equation :

$$\begin{aligned} t_n = & \underbrace{\Theta(n)}_{\text{split \& median finding}} + \underbrace{t_{\lfloor n/5 \rfloor}}_{\text{median of median}} + \underbrace{\Theta(n)}_{\text{split high low}} + \underbrace{\max\{t_{|T_{low}|}, 1, t_{|T_{high}|}\}}_{\text{if cases}} \\ \leq & \; t_{n/5} + t_{7n/10} + \underbrace{\Theta(n)}_{f(n)} \end{aligned}$$

Theorem C.4. $t_n = \mathcal{O}(n)$

Proof. This will be proven by induction. By the hypothesis $f(n) = \Theta(n)$, we know that $f(n) \leq an$ (with $a > 0$) for large enough n.

To be $\mathcal{O}(n)$, there must exist $c > 0$ such as $t_n \leq cn$.

For n small, the theorem is proven.

Now let's assume it is proven for $1, ..., n-1$ and prove it for n :

$$\begin{aligned}
t_n &\leq t_{\frac{n}{5}} + t_{\frac{7n}{10}} + an \\
&\leq c\frac{n}{5} + c\frac{7n}{10} + an \\
&= c\frac{9n}{10} + an \\
&\leq cn && \text{if } a \leq \frac{1}{10}c
\end{aligned}$$

D. Dynamic Programming

1. Computation of combinatorial value

The goal is to compute the following combinatorial value

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

By definition, it can be computed by the following recurrence equation :

$$\begin{aligned} \binom{n}{k} &= \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \\ &= 1 & \text{if } k = 0 or n \\ &= 0 & \text{otherwise} \end{aligned}$$

a. Naive way : simple recursive algorithm

A way to compute $\binom{n}{k}$ would be to follow the recurrence equation by using the following algorithm (List.D.1).

Code D.1: Pseudo-code of the naive algorithm for the combinatorial

```
C(n,k) :
    if (0 < k < n)
        return (C(n-1),k-1) + C(n-1,k))
    else if (k == 1 || k == n)
        return 1
    else
        return 0
```

It can be easily seen that this will lead to an enormous amount of recursive calls. The time complexity is thus a large number. But hopefully for us, there is redundancy in the computation! Some terms are computed many times.

However, as the results are used directly in the computation, the space complexity is $\mathcal{O}(1)$.

b. Better way using Pascal's triangle : "bottom-up" approach

A better way is to store the intermediate result into a table : Pascal's triangle. We can build this table row by row.

As the computation of one term (knowing the recursive values) is done in $\Theta(1)$, the time complexity to fill the whole table is $\Theta(nk)$.

The space complexity is $\Theta(nk)$ if we keep all the intermediate values, but in fact we only need to keep one line at the same time, this reduce the space complexity to $\Theta(k)$.

If we compare the naive way with this one, we can see that the time complexity was improved at the expense of the space complexity. This effect is called the time-space trade-off.

This way of solving all the subproblem from the smallest to the final value (the one needed) is called the "bottom-up" approach.

c. Third way with memoization : "top-down" approach

The "top-down" approach consist of taking the initial recursive algorithm and add a means of storing intermediate results.

We can for example use a table (initially set to $Tab(n,k) = -1 \ \forall n, k$) and take the initial algo but before the computation, check if the result is already in the table.

This is called recursive algorithm with memory function (or with memoization[1]). This technique can be useful for problems where we do not know in advance which subproblems are relevant and need to be computed because it compute exactly what's needed.

The time complexity will be the same as the bottom up approach ($\Theta(nk)$). But as we keep all intermediate result, we have a space complexity of $\Theta(nk)$.

d. Dynamic programing : Chained matrix products

We want to compute the following matrix product : $M = M_1 M_2 M_3 ... M_n$ with M_i : $d_{i-1} \times d_i$ matrix.

There exist several orders to proceed :

$$\begin{aligned} M &= (M_1M_2)(M_3M_4) \\ &= ((M_1M_2)M_3)M_4 \\ &= M_1((M_2M_3)M_4) \\ &= ... \end{aligned}$$

Which order chose to minimise total computation time?

First let's compute the cost of multiplying 2 simple matrix (we neglect STRASSEN and all advance techniques,... we do the simple naive multiplication):
$Cost(AB) = abc$elementary multiplications (if $A = a \times b$, $B = b \times c$ and thus $AB = a \times c$)

[1] if you doubt about the name, check out `http://en.wikipedia.org/wiki/Memoization`

The order can matter: $Cost((M_1M_2)M_3) = abc + acd = ac(b+d)$ and $Cost(M_1(M_2M_3)) = bcd + abd = bd(c+a)$ (if $M_1 = a \times b$, $M_2 = b \times c$ and $M_3 = c \times d$).

How many possible orders for multiplying n matrices? The answer is $C_n = n^{th}$ Catalan number (from Eugène Catalan[2]: 1814-1894 Belgium).

Theorem D.1. Theorem :

$$\begin{aligned} C_0 &= 0 \\ C_1 &= 1 \\ C_n &= \sum_{i=1}^{n-1} C_i C_{n-i} = \sum_{i=0}^{n} C_i C_{n-i} \end{aligned}$$

Proof A simple proof of it can be made by looking at the following possibilities of decomposition of the product $M_1M_2..M_n$:

$$\underbrace{(M_1)}_{C_1}\underbrace{(M_2..M_n)}_{C_{n-1}} || \underbrace{(M_1M_2)}_{C_2}\underbrace{(M_3..M_n)}_{C_{n-2}} ||...|| \underbrace{(M_1..M_i)}_{C_i}\underbrace{(M_{i+1}..M_n)}_{C_{n-i}} ||...|| \underbrace{(M_1..M_{n-1})}_{C_{n-1}}\underbrace{(M_n)}_{C_1}$$

QED

Theorem D.2. $C_n = 1/n \binom{2n-2}{n-1}$

Proof Generating function: $f(z) = \sum_{n\geq 0} C_n z^n = C_1 z + C_2 z^2 + ...$

$f^2(z) = \sum_{n\geq 0}\sum_{i=0}^{n} C_i C_{n-i} z^n = C_1 z^2 + (C_1C_2 + C_2C_1)z^3 + ... = f(z) - z$

$f^2(z) - f(z) + z = 0$

$f(z) = \frac{1+-\sqrt{1-4z}}{2}$

Taylor : $\sqrt{1-4z} = 1 - 2z - 2z^2 - ... => f(z) = \frac{1}{2}(1 - \sqrt{1-4z})$ to have positive Taylor coeff for f(z)

QED

Theorem D.3. Newton binomial formula works also for $n \notin N$:

$$(x+y)^n = \sum_{i\geq 0} \binom{n}{i} x^i y^{n-1}$$

Proof

$$\begin{aligned} \sqrt{1-4z} &= (1-4z)^{1/2} \\ &= \sum_{i\geq 0} \binom{1/2}{i} (-4z)^i \underbrace{1^{1/2-i}}_{1} \\ &= \sum_{i\geq 0} \frac{1/2(-1/2)(-3/2)...(-(2i-3)/2)}{i} (-4z)^i \end{aligned}$$

[2] http://en.wikipedia.org/wiki/Eug%C3%A8ne_Charles_Catalan

$=> C_i = -1/2\frac{(1/2)(-1/2)(-3/2)...(-(2i-3)/2)}{i!}(-1)^i 4^i = (1/2)(1/2)\frac{\frac{1.3.5....(2i-3)}{2^i}}{i!}\frac{2.4.6...(2i-2)}{2.4.6...(2i-2)}2^i 2^i =$ $\frac{(2i-2)!}{i!}\frac{1}{1.2.3...(i-1)} = \frac{(2i-2)!}{i!(i-1)!} = \frac{1}{i}\frac{(2i-2)!}{(i-1)!(i-1)!} = \frac{1}{i}\binom{2i-2}{i-1}$

$=> C_n = \frac{1}{n}\binom{2n-2}{n-1}$

QED

With Stirling's approximation ($n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$) we can obtain a more practical approximation of the order :

$$\begin{aligned}
C_n &= \frac{1}{n}\binom{2n-2}{n-1} \\
&= \frac{1}{n}\frac{(2n-2)!}{(n-1)!(n-1)!} \\
&\approx \frac{1}{n}\frac{\left(\frac{2n-2}{e}\right)^{2n-2}\sqrt{2\pi(2n-2)}}{\left(\frac{n-1}{e}\right)^{n-1}\sqrt{2\pi(n-1)}\left(\frac{n-1}{e}\right)^{n-1}\sqrt{2\pi(n-1)}} \\
&= \frac{1}{n}\frac{4^{n-1}\left(\frac{n-1}{e}\right)^{2n-2}\sqrt{4\pi}\sqrt{n-1}}{\left(\frac{n-1}{e}\right)^{2n-2}2\pi(n-1)} \\
&= \frac{4^n}{4\sqrt{\pi}n\sqrt{n-1}} \\
&= \Theta\left(\frac{4^n}{n^{3/2}}\right)
\end{aligned}$$

This order is big due to the exponential component. There is no place for brute-force resolution (totally impractical)!

TEMPORARY - CM5

2. Chained matrix multiplication

Let us now see how we can improve chained matrix multiplication.
Let's say we have n matrices $M_1, ..., M_n$ and the dimension of the i-th matrix M_i is $d_{i-1} \times d_i$.
We saw that the naive cost of the product AB (with A being an $a \times b$ matrix and B an $b \times c$ matrix) is abc.

We showed that the number of possible orders is Catalan's number (approximately $\frac{4^n}{n\sqrt{n}}$) and brute force is thus clearly not an option.

a. Divide and conquer approach

Let us define Cost$[i, j]$ as the best cost to multiply $M_i, ..., M_j$. We divide with respect to the top bracketing $(M_i, ..., M_k)(M_{k+1}, ..., M_j)$; then we can further define Cost$[i, j]$

as

$$\text{Cost}[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_k \text{Cost}[i,k] + \text{Cost}[k,j] + d_{i-1}d_kd_j & \text{otherwise} \end{cases} \tag{D.1}$$

By this definition, every leaf of the recursion tree of $\text{Cost}[1,n]$ correspond to one order of multiplication of $M_1, ..., M_n$. This means that the number of leaves is equal to Catalan's number and thus that this approach has a time complexity of $\Omega(\frac{4^n}{n\sqrt{n}})$... This approach gives the same results as the brute force.

This is wasteful because intermediate results are recomputed over and over again in the recursion tree: the subproblems are overlapping.

b. Dynamical programming approach

Within this new approach, $\text{Cost}[i,j]$ is a table, where the entry $C[i,j]$ is the optimal number of operations for multiplying matrices $i, i+1, ...j-1, j$. Indeed, $i \leq j$.

This table is filled diagonal by diagonal (starting from the main diagonal then going up) using Equation (D.1). The time to compute $\text{Cost}[i,j]$ is $\Theta(j-i)$ and the time to fill in the table is thus

$$\begin{aligned}\text{Fill in time} &= \sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(j-i) \\ &= \sum_{i=1}^{n}\sum_{k=0}^{n-i}\Theta(k) = \sum_{i=1}^{n}\Theta((ni)^2) \\ &= \sum_{k}^{n-1}\Theta(k^2) = \Theta(n^3) \leq \Omega(4^n)\end{aligned}$$

This is clealry better than the complexity we had before!

Note that fill in the table is a "bottom-up" approach while a recursive algorithm combined with a memory function (some memorization) is a "top-down" approach.

3. When can dynamical programming be used for optimisation problem?

There are two principles to take into account:

Optimality principle For any optimal structure (like a list, a tree, etc.) answering the problem, the substructures (sublists, etc.) are optimal for a subproblem (a problem with smaller instances).

Subproblems are overlapping Divide and conquer is wasteful because it computes many times the same subproblem.

a. Exemples

i. Shortest path in a graph

We can see that a subpath of a shortest path is also a shortest path (thus the optimality principle is respected). Dijkstra's and Floyd's algorithms are good examples of dynamical programming applied to this problem, altough they are a bit more clever (see next chapter).

ii. Knapsack

The problem here is to fill in a bag of volume V with objets among n, each having a volume v_i and a value c_i. The problem can be written

$$\max \sum_{i\in\text{Bag}} c_i$$
$$\text{s.t.} \sum_{i\in\text{Bag}} N_i \leq V$$

The brute force approach has a time complexity of $\mathcal{O}(2^n)$. But what about the dynamical programming approach?

Let us define $\text{Value}(S, V)$ as the maximal value for the set of object S and a total bag volume V, then we can write

$$\text{Value}(S,V) = \max\big(\underbrace{\text{Value}(S\setminus\{i\}, V - v_i) + c_i}_{\text{take the object } i};\underbrace{\text{Value}(S\setminus\{i\}, V)}_{\text{do not take object } i}\big)$$

with i an arbitrary object of S.

Since we are interessed in:

- A solution with at most a weight V in the knapsack,
- which takes care of all n objects,

we need to compute $\text{Value}(n, V)$. But before computing $\text{Value}(n, V)$ we need to compute $\text{Value}(i, j)$ for all $i \leq n$ and $j \leq V$. Also, we need $\mathcal{O}(1)$ operation to compute one entry. We can thus deduce that the complexity of the algorithm is $\mathcal{O}(nV)$.

As some entries may never be needed to compute Value(Solution), it is a recursive function with memorization (a top-down approach may be useful).

iii. Longuest common subsequence

Let us show the concept with an example: we start with the sequences

$$X = a\ b\ c\ b\ d\ a\ b$$
$$Y = b\ d\ c\ a\ d\ a$$

They have many common subsequences, but the longest is $Z = bcda$:

$$\begin{aligned} X &= a\ \underline{b}\ \underline{c}\ b\ \underline{d}\ \underline{a}\ b \\ Y &= \underline{b}\ d\ \underline{c}\ a\ \underline{d}\ \underline{a} \end{aligned}$$

Let us take the following prefix:

$$\begin{aligned} X_i &= X_1 ... X_i \text{ of } X_n = X_1 ... X_n \\ Y_i &= Y_1 ... Y_i \text{ of } Y_n = Y_1 ... Y_n \end{aligned}$$

and define $\mathrm{LCS}(i,j)$ as the length of the longest common subsequence of X_i and Y_j. We can write

$$\begin{aligned} &\mathrm{LCS}(0,0) = 0 \\ &\mathrm{LCS}(i,j) = \begin{cases} \max\left(\mathrm{LCS}(i-1,j); \mathrm{LCS}(i,j-1)\right) & \text{if } x_i \neq y_j \\ \mathrm{LCS}(i-1,j-1)+1 & \text{if } x_i = y_j \end{cases} \end{aligned}$$

for $i, j > 0$.

The cost for X_n, Y_n is $\Theta(n^2)$, but for all these problems, we do not want only the optimal cost, but also the optimal structure (path, subsequence, etc.). We thus need to create a second table where we will record the optimal choices for each subproblem as it is then easy to reconstruct the optimal structure in the end.

E. Greedy Algorithms

Let us now introduce a new kind of algorithm, which will be seen in more details in this new part: the greedy algorithms. In this part, we look at the activity selection problem: given a set of activities $\{a_1, ..., a_n\}$ and the knowledge that activity a_i takes the time interval $[s_i, f_i[$ (where s_i is the start time and f_i the end time), we want to find the maximum size set of mutually disjoint activities.

This is equivalent to booking an auditorium with as many activities as possible without scheduling conflicts.

TEMPORARY - CM6

1. Activity selection problem

We would like to book as many pairwise disjoint activities as possible for a seminar/lecture room. Each activity is characterized by two features : its starting time and its finishing time. Let us write the i-th activity as $a_i = [s_i \;\; f_i[$ with s_i and f_i its starting and finishing time, respectively. Let us also assume that activities are sorted such that $f_1 \leq f_2 \leq \ldots \leq f_n$.

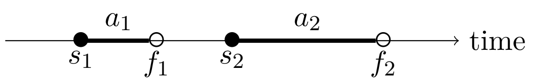


Figure E.1.: Example of two disjoint activities

a. Dynamic programming solution

i. Can we divide this into subproblems ?

Let us define $S_{ij} = \{a_k \,|\, a_k \subseteq [f_i \; s_j[\}$ and c_{ij} as the maximum number of disjoint activities we can book in $[f_i \;\; s_j[$ from activities in S_{ij}.

From Figure E.2, it is pretty easy to see that

$$c_{ij} = \begin{cases} \max\limits_{a_k \in S_{ij}} (c_{ik} + c_{kj} + 1) & \text{if } S_{ij} \neq \emptyset \\ 0 & \text{if } S_{ij} = \emptyset \;. \end{cases}$$

This decomposition in subproblems tells us that there exists a dynamic programming algorithm to solve the activity selection problem.

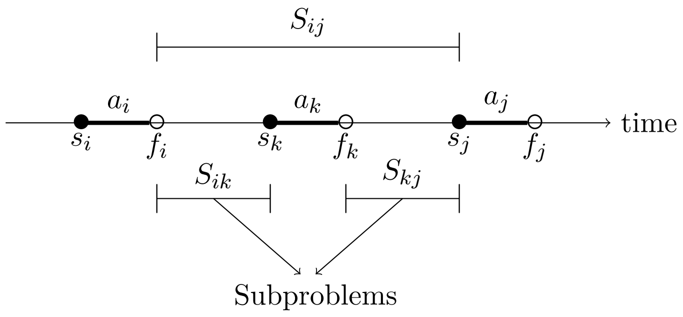


Figure E.2.: Decomposition in subproblems

b. Greedy solution

However, we can be smarter than that ! Indeed, we can know which activity to choose in advance : the one that finishes first among S_{ij}, i.e. the one with smalllest index in S_{ij} (since $f_1 \leq f_2 \leq \ldots \leq f_n$).

i. Why ?

If we have an optimal solution (for the subproblem c_{ij}) that doesn't use $a_{k_{\min}}$ (the activity in S_{ij} with earliest finish) but starts with $a_k \in S_{ij}$ instead, then we can get a solution just as good (same cardinality c_{ij}) by replacing a_k with $a_{k_{\min}}$. This is illustrated on Figure E.3.

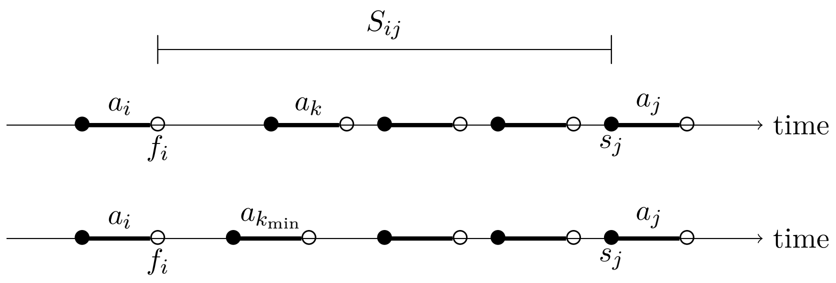


Figure E.3.: Illustration of the greedy algorithm argument

Consequently, a greedy algorithm can be designed to take advantage of that fact. That greedy algorithm is the following :

Code E.1: Pseudo-code of the greedy algorithm for activity selection problem

```
ActivitySelect({ a_1,...,a_n})
        {a_1 } union ActivitySelect({activities disjoint from a_1↩
    })
```

The time complexity of this algorithm is

$$\overbrace{\Theta(n \log n)}^{\text{Sort}} + \Theta(n) = \Theta(n \log n)\,.$$

On the other hand, dynamic programming would cost $\mathcal{O}(n^3)$. We can see a big difference in terms of time complexity between the greedy approach and the dynamic programming approach for the activity selection problem.

"Greedy" : we build up a solution from $\emptyset$ by adding at each step the "best" piece. In a short-sighted way, we don't care for the future.

A greedy approach can sometimes be adopted in dynamic programming situations where the **principle of suboptimality** applies. However, a greedy choice does not always exist.

ii. Other examples of greedy algorithms

- Minimum spanning tree : Kruskal and Prim algorithms
- Shortest path : Dijkstra algorithm

2. Scheduling problem

Let us now consider another famous problem:

- n jobs, each having a one unit of time duration
- t_i: deadline for the job i
- c_i: profit for job i

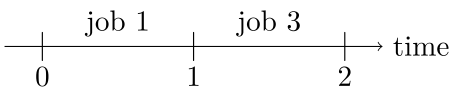
It is important to say that we earn c_i from job i if and only if it is scheduled to finish before its deadline t_i. The aim is to maximize the total profit P_{Total}, being the sum of the individual profits of the jobs finishing before their respective deadline.

a. Example

Let's take the following instance

i	1	2	3	4
c_i	50	10	15	30
t_i	2	1	2	1

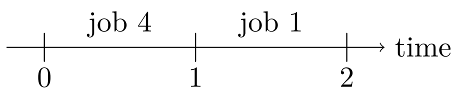
One possible scheduling of the jobs is the following



In that particular case, the total profit will simply be

$$P_{Total} = c_1 + c_3 = 50 + 15 = 65 \, .$$

Another possible scheduling of the jobs is



We have now improved the total profit:

$$P_{Total} = c_1 + c_4 = 50 + 30 = 80\,.$$

This is in fact the optimal solution. Indeed, if we schedule first job 4, and then job 1, we obtain the best solution since they are the jobs with highest individual profits, and furthermore it is impossible to schedule more than 2 jobs, then it must be the best combination.

b. Generalization: Greedy Algorithm

The greedy algorithm is very simple:

Code E.2: Pseudo-code of the greedy algorithm for scheduling problem

```
L = Empty list (will hold selected jobs)
S = stack of jobs ordered by profit with highest on top.
While S not Empty
    a = S.pop()
    Try add a in L at the farthest time index within deadlines
```

The list L has length T where T is the time horizon considered. The i-th element of L is the job being scheduled from $t = i - 1$ to $t = i$, or is empty if no job is scheduled at that time slot. Note that this algorithm works if $t_i \in \mathbb{N}\ \forall i$ but doesn't work if there exist a t_i non-integral. Let's see how it works on our example (in this case $T = 2$):

Code E.3: Example of the greedy algorithm for scheduling problem

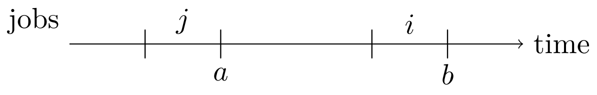
```
Select job with highest profit : job1
    Try add job1 in L -> possible
    $ L ={/, job1} : Profit of 50
Select job with 2nd highest profit : job4
    Try add job4 in L -> possible
    $ L ={job4, job1} : Profit of 80
Select job with 3rd highest profit : job3
    Try add job3 in L -> impossible
    $ Do not add it
Select with 4th highest profit : job2
    Try add job2 in L -> impossible
    $ Do not add it
STOP
```

c. Does the greedy algorithm work? How to check that a set of jobs is feasible?

Theorem E.1. A set of jobs with deadlines $t_1 \leq t_2 \leq \ldots \leq t_n$ is feasible, if and only if, the order $1, 2, \ldots, n$ is feasible.

Proof

- ($\Rightarrow$) If the schedule



  is feasible, with $t_i \leq t_j$, then we can swap i and j, and it's still a feasible schedule. Indeed, since this schedule is feasible, we have $t_i \geq b$ and $t_j \geq a$. Furthermore, we have that $b \geq a$ and $t_j \geq t_i$. It follows that $t_i \geq b \geq a$ and $t_j \geq t_i \geq b$, which make the schedule where i and j are swapped feasible. We can repeat swaps as many times as necessary $\Rightarrow$ order $1, 2, \ldots, n$ feasible.

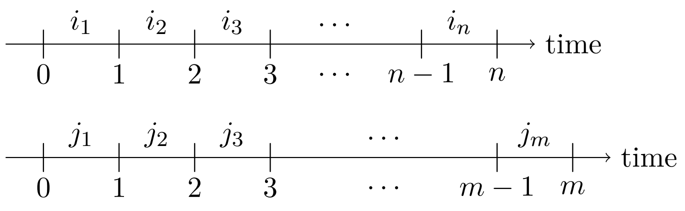
- ($\Leftarrow$) Trivial.

QED

Note that if the jobs are already ordered by increasing deadlines, then checking feasibility of the schedule is done in $\Theta(n)$.

Theorem E.2. The greedy algorithm provides the optimal scheduling.

Proof Suppose that $I = \{i_1, i_2, \ldots, i_n\}$ is the greedy solution and $J = \{j_1, j_2, \ldots, j_m\}$ is the optimal solution. $\Rightarrow$ Both feasible $\Rightarrow$ can be ordered at unit time slots from the left



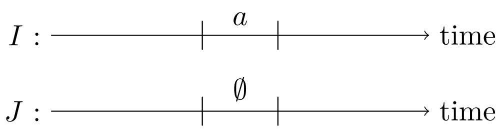
We want to "align" I and J as much as possible to compare them: schedule same jobs at the same time.

Example E.1. Suppose that job $k \in I \bigcap J$, e.g. $k = i_4 = j_8$, then job k is scheduled at the 4-th time slot in I and at the 8-th time slot in J. We want to swap jobs either in I or in J in order to have job k scheduled at the same time slot in both schedules.

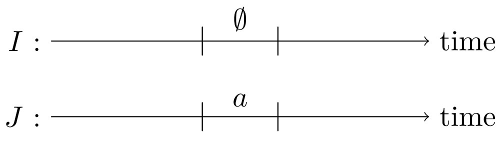
To that end, we will swap i_4 and i_8 in I. If i_8 is empty, then after the swap is performed, i_4 will be empty and i_8 will contain job k. If $i_8 = \tilde{k}$ was not empty, then after the swap has been performed, job $\tilde{k}$ is now scheduled on the 4-th time slot and job k on the 8-th. Moving job $\tilde{k}$ from the 8-th time slot to the 4-th does not put the feasibility of the new schedule in jeopardy, since it is now scheduled earlier. Furthermore, scheduling the job k at the 8-th time slot instead of the 4-th is also feasible since job k is scheduled at that 8-th time slot in J, and J is assumed to be feasible.

> Could we have swapped j_8 and j_4 in J instead of swapping i_4 and i_8 in I? The answer is : it depends, but in general **NO** ! Indeed, although scheduling job k at the 4-th time slot instead of at the 8-th in J would always be feasible, there is no guarantee that the job that was initially in j_4 would still be feasible at the 8-th time slot, therefore preventing the "swapped" schedule to be feasible ! e.g. : $k = i_7 = j_2$: then swap j_2 and j_7. Job k will never be rescheduled

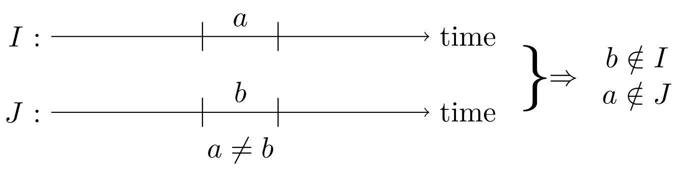
Repeat all swaps until no more is possible, then all common jobs are aligned. Now, let us take a look at all the situations one could think of where the jobs are different in I and in J. The first situation looks like



This first situation is impossible because since $a \notin J$, we can schedule it in the empty slot and improve the total profit of the sequence J strictly. A second situtation is



Once again, this situation is impossible because since $a \notin I$, then adding a to I is feasible $\Rightarrow$ the greedy algorithm would have chosen it. Finally, the last situation that could potentially occur is



If $c_b > c_a$: then the greedy algorithm would have chosen b before $a \Rightarrow b$ would be in I.
If $c_a > c_b$: then J is strictly improved by replacing b with a, then impossible case.
It follows that $c_a = c_b$, and consequently we have that $C_I = C_J$.

QED

Time complexity $\underbrace{\Theta(n \log n)}_{\text{Sort by profit}} + \underbrace{\Theta(n^2 \log n)}_{\text{Sort by deadline } n \text{ times}}$

But we can do better : $\Theta(n \log n)$

F. Random Algorithms

TEMPORARY - CM7

First, let's do a few exercices in probability.

1. The marriage/secretary/harem/... problem

n candidates of different quality apply for an open position. They are sent in random order. Every time you interview a candidate who proves better than any other previous one, you hire him/her.

a. How many persons will you hire on average?

Say you interview them

- in increasing order $q_1 < q_2 < \ldots < q_n$ ($1, 2, \ldots, n$ denote the order of the interviews) $\Rightarrow n$ hires;
- in decreasing order $q_1 > q_2 > \ldots > q_n \Rightarrow 1$ hire;
- in random order $\Rightarrow$ something in between.

To find the number of persons we will hire on average, we will use the powerful technique of *indicator variable*. Let

$$X_i = \begin{cases} 1 \text{ if person } i \text{ is hired} \\ 0 \text{ it not} \end{cases}$$

$$\mathbf{E}[X_i] = \mathrm{P}[\text{person } i \text{ is hired}] \qquad \text{fondamental property of indicator variable}$$

We want to find

$$\begin{aligned}
\mathbf{E}[X] &= \mathbf{E}[X_1 + X_2 + \ldots + X_n] \\
&= \mathbf{E}[X_1] + \mathbf{E}[X_2] + \ldots + \mathbf{E}[X_n] && \text{since the expectation is linear} \\
&= \sum_{i=1}^{n} \mathrm{P}[i \text{ is hired}] \\
&= \sum_{i=1}^{n} \frac{1}{i} && \text{because the best among } 1, 2, \ldots, i \text{ is } i \text{ with probability } \frac{1}{i} \\
&\approx \int_1^n \frac{1}{i}\,\mathrm{d}i. && \text{continuous problem} \\
&= \ln n.
\end{aligned}$$

b. Variant

This time, we hire only 1 person and we want to maximize the probability to pick the best among the n candidates (as soon as we interview someone, we need to say yes or no).

Code F.1: Pseudo-code of the Variant of Hiring Problem algorithm

```
Don't hire the first k-1 candidates (observation phase).
Hire the first among the candidates k, k+1,..., n to be the best ↩
    interviewed so far.
```

c. What is the best k?

$$\begin{aligned}
\text{Probability of success} &= \sum_{i=k}^{n} \mathrm{P}[i \text{ is hired and is the best}] \\
&= \sum_{i=k}^{n} [i \text{ is the best}] \times \mathrm{P}[i \text{ is hired} \mid i \text{ is the best}] \\
&= \sum_{i=k}^{n} \frac{1}{n}\frac{k-1}{i-1} \\
&\approxeq \frac{k}{n}\int_{k-1}^{n-1} \frac{1}{x}\,\mathrm{d}x. \\
&= \frac{k}{n}(\ln n - \ln k) \\
&= \frac{\ln \frac{n}{k}}{\frac{n}{k}} \\
&= \frac{\ln y}{y} \text{for } y = \frac{n}{k} \geq 1
\end{aligned}$$

To find this, we need to notice that if the second best among $1, \ldots, i$ is in

- $1, \ldots, k-1$ then i is picked;
- $k, \ldots, i-1$ then i is not picked.

So, i is picked with probability $\frac{k-1}{i-1}$.

Then, y is optimal for

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}y}\left(\frac{\ln y}{y}\right) &= 0 \\
\frac{1-\ln y}{y^2} &= 0 \\
\Rightarrow \ln y &= 1 \\
y &= e
\end{aligned}$$

So $k = \frac{n}{e} \approxeq 0.37n$ is optimal and the corresponding probability of success is $\frac{\ln \frac{n}{k}}{\frac{n}{k}} = \frac{1}{e} \approxeq 0.37$.

2. The birthday paradox

How many people do you need to gather so that two of them will have the same birthday with fair chances? We will solve this problem with indicator variables.

$$\begin{aligned}
X_{ij} &= \begin{cases} 1 \text{ if } i \text{ and } j \text{ have same birthday} \\ 0 \text{ it not} \end{cases} \\
\mathbf{E}[X_{ij}] &= \frac{1}{t} \text{ where } t = 365 \\
\sum_{pairs\{i,j\}} X_{i,j} &= X \\
&= \# \text{ of coincidences} \\
&= \# \text{ pairs of people with same birthday} \\
\mathbf{E}[X] &= \mathbf{E}[\sum_{i,j} X_{i,j}] \\
&= \sum_{i,j} \mathbf{E}[X_{i,j}] \\
&= \frac{n(n-1)}{2}\frac{1}{t} \approxeq \frac{n^2}{2t} \text{ for } n \text{ persons}
\end{aligned}$$

So, if $n = \sqrt{2t}$, then $\mathbf{E}[X] \approxeq 1$, i.e. there exist a fair chance to observe a coincidence. If $t = 365$, $n = \sqrt{2 * 365} = 25$. Therefore already with 25 people there is a very good chance that two people were born on the same day, whereas we would expect that more people would be needed.

3. Compute π

a. Buffon theorem

Let's see an (inefficient) method to compute π:

Theorem F.1. (Buffon): If I throw randomly a needle of length 1 on a floor with planks of width 2, the probability of crossing a crack (i.e. overlap two planks) is $\frac{1}{\pi}$ (see figure F.1).

Proof.

1. Trigonometry + calculus: integrate over all angles and horizontal coordinates of needles.

2. Consider a needle of length l. $E_l :=$ expected number of cracks crossed by the needle $= \mathbb{E}(X_l)$, where X_l is the number of cracks crossed. We have

$$E_{2l} = \mathbb{E}(X_{2l}) = \mathbb{E}(X_l^1 + X_l^2) = \mathbb{E}(X_l^1) + \mathbb{E}(X_l^2) = E_l + E_l = 2E_l$$

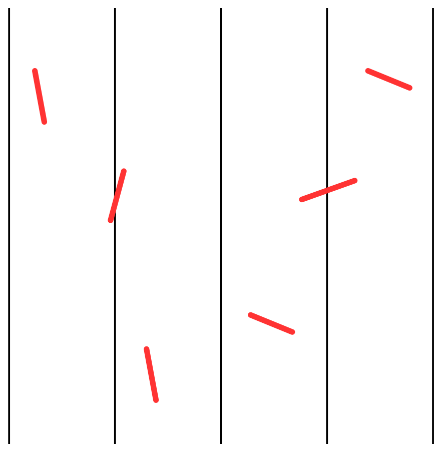


Figure F.1.: Needles of length 1 crossing cracks of width 2

More generally, $E_{l_1+l_2} = E_{l_1} + E_{l_2}$. So we see it follows a linear form: $E_l = \alpha l$, for some α.
If $l = 1$, as it never crosses twice, we can write

$$X_l = \begin{cases} 1 \text{ if the needle crosses a crack} \\ 0 \text{ otherwise} \end{cases}$$

So E_l is the probability to cross a crack. Now we have to proof that $\alpha = \frac{1}{\pi}$.
Let be even more general: let's bend the needle. We have

$$\mathbb{E}(\text{number of cracks crossed}) = E_{l_1} + E_{l_2} = \alpha(l_1 + l_2) = \alpha l$$

with l the total length of the needle. We can bend it several times...

$$\mathbb{E}(\text{number of cracks crossed}) = \alpha \sum_i l_i = \alpha l$$

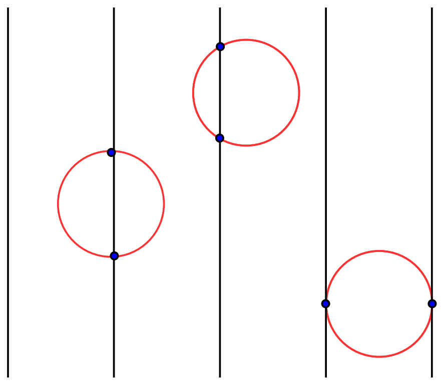


Figure F.2.: Circular needles of radius 1 crossing cracks of width 2

In the limit, for a curved needle of length l, we always have $\mathbb{E}(\text{number of cracks crossed}) = \alpha l$. To fix α, let's look at a circular needle of radius 1, such that $E_l = 2$, always (see figure F.2)! We can write $E_{2\pi} = 2$, but $E_{2\pi} = \alpha 2\pi$, and we conclude with $\alpha = \frac{1}{\pi}$.

We can then develop an algorithm to compute π:

- Throw a needle n times on the floor (as in Buffon's theorem)

- Count how many times a crack is crossed (define this number as k)
- $\pi = \frac{n}{k}$

Accuracy of this algorithm? Let's look at its variance. Let

$$X_i = \begin{cases} 1 \text{ if the } i^{th} \text{ needle crosses a crack} \\ 0 \text{ otherwise} \end{cases}$$

As we have $\mathbb{E}(X_i) = \frac{1}{\pi}$ and $Var(X_i) = \frac{1}{\pi}(1 - \frac{1}{\pi})$ and the X_i are independant, it follows that

$$\begin{aligned} \mathbb{E}(\frac{\sum_i X_i}{n}) &= \frac{1}{\pi} \\ Var(\frac{\sum_i X_i}{n}) &= \frac{n\frac{1}{\pi}(1-\frac{1}{\pi})}{n^2} = \frac{1}{n}\frac{1}{\pi}(1-\frac{1}{\pi}) \end{aligned}$$

We can now compute the standard deviation $\sigma = \frac{1}{\sqrt{n}}\sqrt{\frac{1}{\pi}(1-\frac{1}{\pi})}$ = typical error in $\frac{1}{\pi}$:

$$\frac{k}{n} \in [\frac{1}{\pi} - \frac{2}{\sqrt{n}}\sqrt{\frac{1}{\pi}(1-\frac{1}{\pi})}; \frac{1}{\pi} + \frac{2}{\sqrt{n}}\sqrt{\frac{1}{\pi}(1-\frac{1}{\pi})}]$$

with 95% probability.
Remarks:

- To gain one digit of accuracy, you need 100 times more throws: not good.
- The result is possibly wrong, with some probability (here 5%).

TEMPORARY - CM8

However, this approach converges quite slowly: the accuracy, measured by the standard deviation, is $\sim \frac{1}{\sqrt{n}}$. We would need a lot of needles...

b. Another way to compute π

Let us first notice that π is simply the area of a unit disc. Thus $\frac{\pi}{4}$ is the area under the red curve of figure F.3a. Then a new random algorithm to compute π is simply:

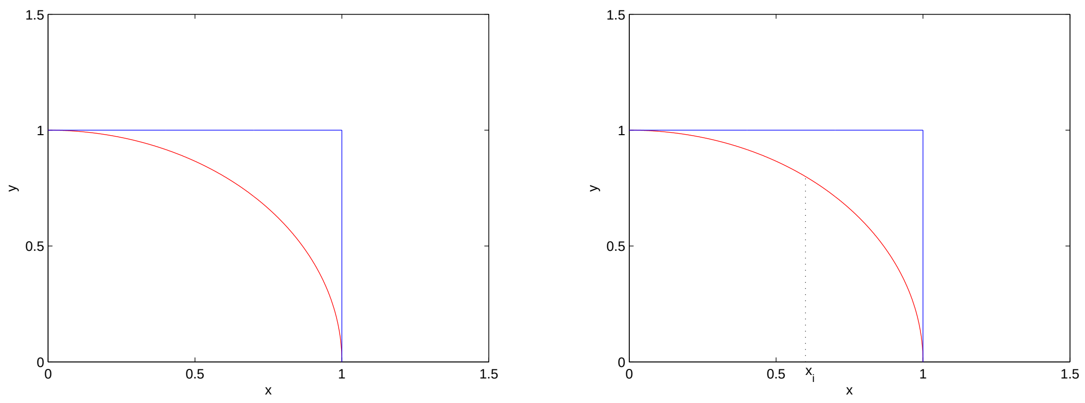
- Pick n random points (x, y) uniformly in $[0, 1]^2$;
- Define $k := |\{(x_i, y_i) : x_i^2 + y_i^2 < 1\}|$;
- $\frac{\pi}{4} \approx \frac{k}{n}$

This algorithm is very similar to the one using the needles, instead that this one can easily be implemented on a computer. Therefore the analysis of convergence remains the same: the accuracy evolve like $\sim \frac{1}{\sqrt{n}}$ (slow convergence).

c. A bit smarter

We can easily improve the last naive π estimation:

- Pick x_i randomly between 0 and 1 (see figure F.3b);



(a) $\frac{\pi}{4}$ = area under the red curve. **(b)**

Figure F.3.

- Compute[1]

$$\begin{aligned}\frac{\sum_{i=1}^{n}\sqrt{1-x_i^2}}{n} &\approx \mathbb{E}\left[\sqrt{1-x^2}\right] \text{ under uniform distribution of } x\\ &:= \int_0^1 \sqrt{1-x^2}dx \text{ by definition of } \mathbb{E}\\ &= \frac{\pi}{4}\end{aligned}$$

Again the error of this algorithm decreases like $\sim \frac{1}{\sqrt{n}}$.

This method generalizes into a random computation of integrals. 1D integrals are usually better evaluated by deterministic methods (rectangles, simpson, ...). Indeed, the error of the rectangles method (the simplest deterministic method) is in $\frac{1}{n}$, far better than $\frac{1}{\sqrt{n}}$.

4. Why use this random algorithm for integration?

For at least two reasons:

Robin hood effect: For every deterministic method, there are functions for which the method will fail. For example, for high frequency sine (see figure F.4a), if the red points are chosen as equally spaced quadrature points, the deterministic method will give a very poor estimate of the integral.

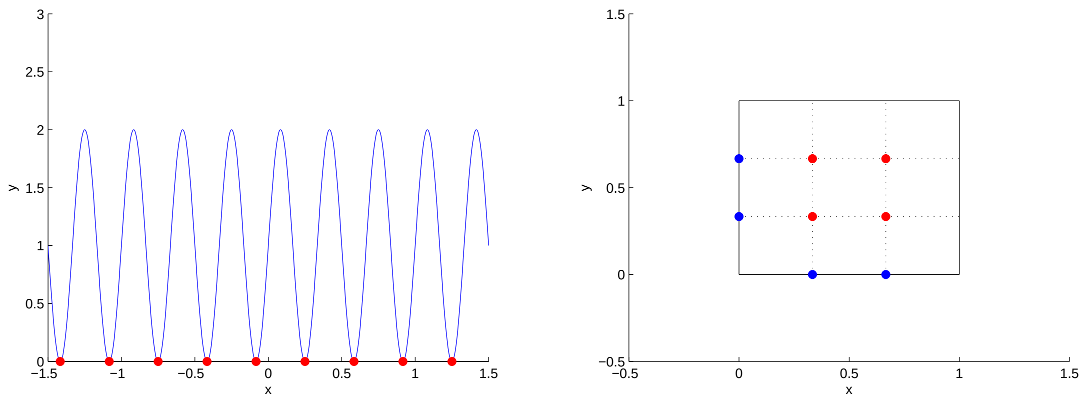
Thus deterministic methods have very bad worst case behaviour. The choice of random points makes all instances "equal".

For kD integrals: $\underbrace{\int\int\dots\int}_{k} f(x_1,\dots,x_k)dx_1\dots dx_k$

[1] For a fixed x_i, the probability for the associated y_i to be such that (x_i, y_i) lies under the red curve of figure F.3b, is $\frac{\sqrt{1-x_i^2}}{1}$, under uniform distribution. Since we know this probability, we can use it directly instead of picking a random y_i!

For deterministic methods, n points regularly spaced in $[0,1]^k \to \sqrt[k]{n}$ points in each dimension (see figure F.4b: by choosing the 4 red points, we have 2 blue points in each dimension). Typically, the error will be $\sim \frac{1}{\sqrt[k]{n}}$ (when k increases, we need more points to fill the space⟨2⟩).

The random algorithm still behaves in $\frac{1}{\sqrt{n}}$ error (because it depends on the properties of the gaussian distribution) and could possibly be a better choice for triple integrals or more (a hybrid method is even better).



(a) High frequency sine. **(b)**

Figure F.4.

5. Monte-Carlo Algorithms

a. An example where randomness is very useful

Suppose you want to check that $A \cdot B = C$, with A, B and C $n \times n$ matrices.

- Obvious method: compute $A \cdot B$. It costs $\mathcal{O}(n^{2,3\cdots})$ (using divide and conquer methods). If n is large, then this method might take a lot of time.
- Compute $(A \cdot B)_{i,j}$ ($\mathcal{O}(n)$ time) and compare with $C_{i,j}$ for random i, j. However if there is only one wrong $C_{i,j}$, it is hard to capture it using this method. We would like a method which finds errors with good probability, even if they are really localized.
- If $A \cdot B = C$ then $\forall v \in \mathbb{R}^{n\times 1}$, $\underbrace{A \cdot B \cdot v}_{\mathcal{O}(n^2)} = \underbrace{C \cdot v}_{\mathcal{O}(n^2)}$ (this solution is not localized but spread on the columns, by checking if $A \cdot B = C$ in the direction of v). How to pick v? Randomly in $\{0,1\}^n \to$ summing a subset of columns of $A \cdot B$ (resp. C) $\Rightarrow A \cdot B \cdot v$ (resp. $C \cdot v$).

Say there is an error in column j of C (and possibly elsewhere). $\forall v \in \{0,1\}^n$, call $v_{\bar{j}} = v$ except that entry j is flipped ($0 \leftrightarrow 1$).

⟨2⟩Thus with fixed n, we have less information along each dimension and thus the accuracy decreases.

- Either $A \cdot B \cdot v \neq C \cdot v$ and $A \cdot B \cdot v_{\bar{j}} \neq C \cdot v_{\bar{j}}$;
- or $A \cdot B \cdot v = C \cdot v$ and $A \cdot B \cdot v_{\bar{j}} \neq C \cdot v_{\bar{j}}$;
- or $A \cdot B \cdot v \neq C \cdot v$ and $A \cdot B \cdot v_{\bar{j}} = C \cdot v_{\bar{j}}$;
- but $A \cdot B \cdot v = C \cdot v$ and $A \cdot B \cdot v_{\bar{j}} = C \cdot v_{\bar{j}}$ is impossible because then

$$\underbrace{A \cdot B \cdot (v - v_{\bar{j}})}_{\pm j^{th} \text{ column of } A \cdot B} = \underbrace{C \cdot (v - v_{\bar{j}})}_{\pm j^{th} \text{ column of } A \cdot B}$$

which is impossible since there is a mistake in j^{th} column of C!

Therefore at least half of possible $v \in \{0,1\}^n$ will show an error $\Rightarrow$ Error detected with probability $\geq \frac{1}{2}$ (if there is at least one error).

This leads us to Freivalds algorithm:

Code F.2: Freivalds Algorithm (A,B,C)

```
Pick  v in {0,1}^n randomly (using uniform distribution);
If A*B*v==C*v
    then print "A*B=C" [maybe];
Else A*B*v~=C*v
    then print "A*B~=C" [for sure].
```

However with this algorithm, false negatives (i.e. $A \cdot B \neq C$ but undected, with probability $\leq \frac{1}{2}$) are possible... To improve Freivalds, we can use amplification of stochastic advantage⟨3⟩:

Code F.3: Repeat(A,B,C,k)

```
    Repeat Freivalds k times;
    If k conclusions "A*B==C"
        then print "A*B=C" [maybe];
    Elseif at least one conclusion "A*B~=C"
        then print "A*B~=C" [for sure].
```

A mistake will go undetected with probability $\leq \frac{1}{2^k}$. For example, with $k = 10 \Rightarrow \frac{1}{2^k} \approx 10^{-3} = 0.1\%$, with time $\mathcal{O}(kn^2)$ which is better than $\mathcal{O}(n^{2,3\cdots})$ or $\mathcal{O}(n^3)$.

Amplification of stochastic advantage is very powerful when errors on a decision problem (YES/NO problem⟨4⟩) are one-sided: $\exists$ false negative, $\nexists$ false positives, or the contrary (in our case, if we print "$A \cdot B \neq C$", it is for sure).

⟨3⟩ We can try again with other v! We will never be sure that $A \cdot B = C$, but the probability of error will decrease a lot if we test a lot of v.

⟨4⟩ Answer to the problem is binary; $\neq$ compute π.

b. Monte-Carlo Algorithms

What now if you get possible false negatives and false positives (for other problems)? E.g.:

- answer is YES but we conclude YES with probability $\geq \frac{1}{2} + \epsilon$;
- answer is NO but we conclude NO with probability $\geq \frac{1}{2} + \epsilon$;

(the algorithm gives us slightly more often the good answer that when we throw a coin)

Amplification of stochastic advantage still works!

Repeat n times and vote among the answers:

- If $> 50\%$ of YES $\Rightarrow$ output YES;
- if $> 50\%$ of NO $\Rightarrow$ output NO.

What is the probability of error? We can use indicator variables. Let

$$X_i = \begin{cases} 1 & \text{if } i^{th} \text{ run is correct} \\ 0 & \text{if } i^{th} \text{ run is wrong} \end{cases}$$

Then we have $\mathbb{E}[X_i] \geq \frac{1}{2} + \epsilon$. Let us say that $\mathbb{E}[X_i] = \frac{1}{2} + \epsilon$, to study the worst case.

The repeated algorithm is wrong if $\sum_{i=1} X_i < \frac{n}{2}$. But

$$\begin{aligned} \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] &= \left(\frac{1}{2} + \epsilon\right) n \\ \operatorname{Var}[X_i] &= \mathbb{E}\left[X_i^2\right] - (\mathbb{E}[X_i])^2 \\ &= 1 \cdot \left(\frac{1}{2} + \epsilon\right) + 0 \cdot \left(\frac{1}{2} - \epsilon\right) - \left(\frac{1}{2} + \epsilon\right)^2 \\ &= \frac{1}{4} - \epsilon^2 \\ \operatorname{Var}\left[\sum_{i=1}^{n} X_i\right] &= n \cdot \left(\frac{1}{4} - \epsilon^2\right) \quad \text{using independence of } X_i\text{'s} \end{aligned}$$

Using the Central Limit Theorem, $\sum_{i=1}^{n} X_i \sim \text{Gaussian}\left(\left(\frac{1}{2} + \epsilon\right) n, \left(\frac{1}{4} - \epsilon^2\right) n\right)$ if n is large. Using the tables, $\sum_{i=1}^{n} X_i$ has 2.5% of probability to be 2 standard deviations above its mean.

E.g. $\epsilon \cdot n = 2 \cdot$ (standard deviation) $= 2\sqrt{n}\sqrt{\frac{1}{4} - \epsilon^2} \Rightarrow n = \frac{4}{\epsilon^2}\left(\frac{1}{4} - \epsilon^2\right) \sim \frac{1}{\epsilon^2}$ for small ϵ. Then we have the wrong answer with probability $\leq 2.5\%$.

If $\epsilon \cdot n = 3 \cdot$(standard deviation) $\Rightarrow n \sim \frac{9}{4\epsilon^2}$, then we get the wrong answer with probability $\leq 0.15\%$.

Thus amplification of stochastic still works, but we need to perform more trials then when only false negatives (or false positive) are appearing. Also we can notice that if $n \sim \frac{1}{\epsilon^2}$ is approximately doubled (so that $n \sim \frac{9}{4 \cdot \epsilon^2}$), the probability of error decreases much.

These algorithms provide an uncertain answer (correct with some probability). They are called Monte-Carlo algorithms (they provide possibly wrong answer):

- estimate π;
- estimate kD integrals;
- Freivalds;
- checking that a number is prime (like Freivalds, error is one-sided);
- generating a large prime number[5].

6. Las Vegas Algorithms

Another sort of algorithms are those that provide a correct answer always, but with random execution time. They are called Las Vegas algorithm. E.g., Random QuickSort, Random selection/Median, ...

a. Eight Queens problem

Let us see a Las Vegas algorithm for the Eight Queens problem: how to place 8 queens in a chessboard in mutually non-attacking positions, i.e.

- no two queens in the same row;
- no two queens in the same column;
- no two queens in the same diagonal;

i. Brute force solution

Explore systematically the tree of possibilities, **backtracking** when you reach a dead-end:

- place a queen on the first row;
- place a queen on the second row (but different column);
- place a queen on the third row (but different column);
- $\vdots$
- avoiding same diagonal-positions.

ii. The random 8 queens algorithm makes random choices

- random column for the 1st queen (in 1st row);
- random column for the 2nd queen (but $\neq$ from the first one and among the valid squares for diagonals)
- ...

[5] The last two items are useful in cryptography.

If there is no more solution to place a new queen, we repeat **from scratch** = no memory of previous attempt (it is the difference with the backtracking solution). We can compute that

$$\mathbb{E}\left(\text{time success}\right) = \underbrace{\text{Time}_{\text{in case of Success}}}_{\substack{\text{time to fill in the chessboard}\\ \text{for the final solution}}} + \left[\frac{1}{P[Success]} - 1\right] \cdot \ \text{Time}_{\text{in case of failure}}$$

$$\text{because } \frac{1}{P\left[success\right]} = \mathbb{E}\left[\text{number of runs to get a success}\right] = \mathbb{E}\left[\text{number of failure}\right]$$

Thus we have [6]

$$\begin{aligned}\mathbb{E}\left(\text{time success}\right) &= 8 + (\frac{1}{0,129...} - 1) \cdot 6 \text{ queen placements}\\ &\approx 50 \text{ or } 60 \text{ queen placements}\end{aligned}$$

whereas the backtracking solution will get a success after 155 queen placements.

Furthermore the gap grows quickly when 8 queens $\Rightarrow$ n queens (when we increase the number of queens to place). This is at first sight surprising because the random algorithm and the backtracking solution explore the same possibilities but in different orders.

The random algorithm needs less time because placements have nothing "regular".

[6] The 6 in the equation comes from the fact that on average we have 5 or 6 queen placement in a failure

G. Computability and decidability

TEMPORARY - CM9

1. Computability and Turing machines

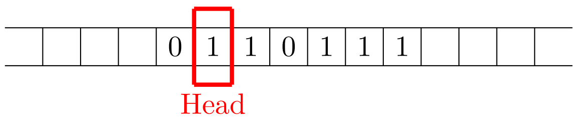
a. Definitions

What computers can and can't do? *Computability* answers this question.
There are problems no computer or algorithm can ever solve. This is a bad news.
To prove such results, we need a more formal definition of computations.

Intuitive definition of computing (Turing, 1936) : a human being is asked a *question* ("input" or "instance", e.g. "123+479=?"), under the form of finitely many symbols from a finite alphabet, written on a sheet of paper.
The (human) computer can write down intermediate results, with unlimited supply of paper, using again symbols from a finite alphabet. The brain follows blindly, non-creatively, a finite list of elementary instructions. The next instruction is decided from the current instruction and the intermediate results currently under scruting by the computer on the paper.
There is an instruction called `STOP`. At this moment, the answer ("output") is written unambiguously on the paper (e.g."602 is the answer").

Formal definition of computing (Turing, 1936): a *Turing machine* (T.M.) is composed of:

- a *tape*, which is bi-infinite, divided into cells, each being blank or containing a symbol from a finite alphabet A (e.g. $A = \{0, 1\}$);
- a *head*, which reads one cell at a time, can write a new symbol on this cell, shift the tape to the left or right.



- a finite set of instructions, each being numbered $(1, 2, 3, \ldots)$ of the form:
  - `STOP`
  - `"if CurrentCell=blank/0/1`
    `then Write(blank/0/1)`

```
        and ShiftLeft/ShiftRight/NoShift
        and goto Instruction k."
```

So:

	1936		>1948	
algorithm performed by a human brain	$\leftrightarrow$	Turing machine	$\leftrightarrow$	modern computer
paper	$\leftrightarrow$	tape	$\leftrightarrow$	memory
(finite) brain	$\leftrightarrow$	head	$\leftrightarrow$	CPU
instructions	$\leftrightarrow$	instructions	$\leftrightarrow$	instructions, software

A *problem* f is a (partial) mapping.

$$\begin{aligned} f : & A^* \to A^* \\ & w \to f(w) \text{ or undefined} \end{aligned}$$

$A =$ finite alphabet

$A^* =$ set of finite words, e.g. $A^* = \{\emptyset, 0, 1, 00, 01, 11, 10, 000, 001, \ldots\}$

$w =$ "input" or "instance"

$f(w) =$ "output"

A problem f is *computed* or *solved* by a Turing machine T if whenever we write w on an otherwise blank tape (and head is in initial instruction, at the beginning of w), T runs and eventually stops if and only if $f(w)$ is defined, in which case the tape contains $f(w)$ only (if $f(w)$ is undefined, T runs forever: infinite loop).

The problem f is *computable* if it is solved by at least a Turing machine.

A more *modern machine*: Matlab machine: a Matlab machine is the code of a Matlab function (without "rand" instruction) running on an ideal infinite-memory computer.

Theorem G.1. A problem is solved by a Turing machine if and only if it is solved by a Matlab machine.

Proof

- $\boxed{\Leftarrow}$ We have to convert any Matlab function into a T.M.=Matlab-to-TM compiler: admitted;
- $\boxed{\Rightarrow}$ We have to write a TM-to-Matlab compiler: easier, admitted.

QED

We say that Matlab is "Turing-equivalent". So are Python, C, C^{++} and Java.

Thesis G.1 (Church-Turing). Any "obviously algorithmic" procedure (in the intuitive sense) corresponds to an equivalent T.M./Matlab program.

This is *not* a theorem, because it is not formally defined. It is widely accepted because:

- of Turing's argument about human computer;
- every time we tried to convert an "obviously algorithmic" procedure into a T.M., we succeeded (with sweat and pain).

It says by instance that the Euclidean division algorithm, "obviously algorithmic", can be solved by a T.M./a Matlab program (although the T.M. is not trivial to find).
Of course in this specific case we can prove formally that this T.M. exists, by constructing it.

With Church-Turing thesis, the theorem that Matlab is Turing-equivalent is trivial: Matlab instructions are clearly algorithmic.

Pay attention not confuse it with "strong Church-Turing thesis": anything the brain can do (writing poetry, proving theorems, ...) can be simulated by a T.M.
This one is highly controversial!

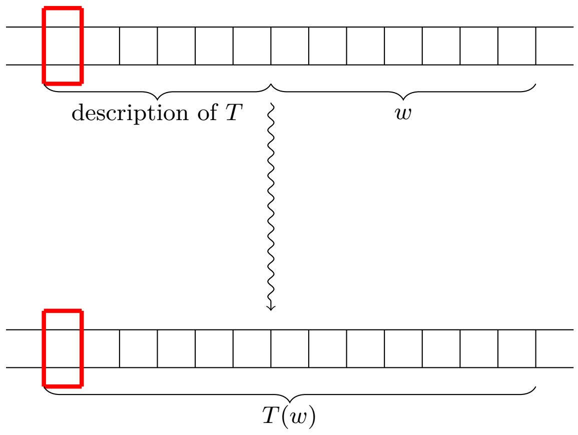
Now that we know what a formal algorithm is, we can prove *theorems* about them.

b. Universal Turing Machine

Theorem G.2 (Turing, 1936)**.** The following problem is computable:

- *Input*: description of a T.M. T + description of finite word on initial tape, "w";
- *Output*:
  - If T stops when initial tape is w, *the finite content of the tape*;
  - If t never stops, *undefined.*

A T.M. solving it is called a "*universal T.M.*".



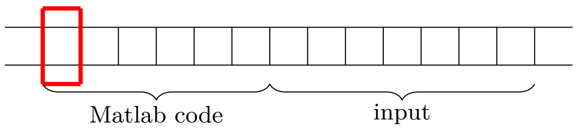
Proof

- construct explicitely universal Turing Machine (thank you Alan)
- OR construct explicitely universal Matlab machine.
- OR by C-T thesis, simulating T is obviously algorithmic (in theintuitive sense) thus there is a Turing Machine doing it.

QED

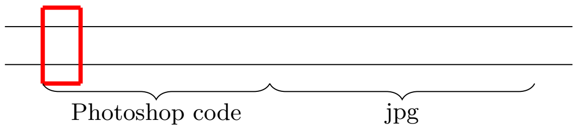
NB: Variante:
Proof

- There's a Matlab program simulating any other Matlab program on any input.
- There's a Turing Machine simulating any Matlab program on any input.



=Matlab-to-TM interpreter

- Universal Turing Machine means that code is just data, written on memory, on the same level as the input.
  e.g.



QED

Modern computer are universal Turing Machines, i.e., they are (re-)programmable.
≡ Turing's fundamental insight.
(Pocket) 4-operation calculator = non universal T.M.

2. Decidability

a. The Harting problem

A decision problem is a total mapping: $A^* \to \{\text{YES,NO}\}$
E.g. "Is number n prime?"
A decision problem is decidable if it's computable.

The Harting problem:

Input:

$$\begin{cases} \text{Matlab code f.m. + input for f} \\ \text{a T.M. description} \end{cases} \tag{G.1}$$

> Output:
>
> - YES if f eventually stops on input x
> - NO if f never stops ("infinite loop")

TEMPORARY - CM10

b. The Halting problem and diagonal argument

Halting problem:

> *Input*: A Matlab code "M.m", an input for this matlab code
> *Output*: YES if M(x) halts after a finite time, NO otherwise

This is a decision problem, i.e. a problem with output YES or NO. A decision problem is decidable iff it is computable.

Theorem G.3. (**Turing, 1936**): The halting problem for Matlab machines is undecidable.

Proof By contradiction: we assume that there is a Matlab code "Halt.m" solving the halting problem. We build "Diagonal.m", a program wich take as argument another Matlab code "M.m" with a string of char as input.

Code G.1: Diagonal(M)

```
INPUT : a Matlab Code M taking a string as input
Diagonal(M)
    if (Halt(M,M) == YES) % M halts on input "M.m"
        while (true) {} % infinite loop
    else
        stop % if M does not halt on M, then stop
end if
```

What happens if we call Diagonal(Diagonal)?

- If it stops then halt(Diagonal,Diagonal) = YES ⇒ infinite loop, does not stop
- If it does not stop, then it stops

⇒ Contradiction
⇒ "Halt.m" does not exist.

QED

This is called a diagonal argument. Why?

We create this array:

	x_1	x_2	x_3	x_4	...
M1.m	**Halt**	NotHalt	NotHalt	Halt	...
M2.m	Halt	**Halt**	NotHalt	Halt	...
M3.m	NotHalt	NotHalt	**NotHalt**	Halt	...
⋮	⋮	⋮	⋮	⋮	

Where x_i represent all strings of char, and "M.i" represent all codes with string of char as inputs. We have entry(i,j) = Halt iff Mi(x_j) halts.

We now change all entries of the diagonal:
If Mi(x_i) = "Halt" then Diagonal(i) = "NotHalt" and conversely.

In our case, we obtain (Diagonal(1),Diagonal(2),Diagonal(3),...) = ("NotHalt","NotHalt","Halt",...). This row is not in the array, by construction. That means that "Diagonal.m"≠"Mi.m, $\forall i$.

Cantor (1891) invented the diagonal argument to prove:

Theorem G.4. $[0,1]$ (or $\mathbb{R}$, etc) is not countable, i.e. it does not exist a surjection $\mathbb{N} \to [0,1]$.

Proof If there exists a surjection $c : \mathbb{N} \to [0,1]$, it follows that $[0,1] = \{c_0, c_1, c_2, ...\}$.

We can now create the array
$$\begin{array}{c|ccccc}
\hline
c_0 & 0. & \mathbf{1} & 0 & 2 & \dots \\
c_1 & 0. & 3 & \mathbf{9} & 1 & \dots \\
c_2 & 0. & 0 & 0 & \mathbf{1} & \dots \\
\vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}$$

From which we create a new number $x = 0.202...$ $(0 \to 1, 1 \to 2, 2 \to 3, ..., 9 \to 0)$. We now see that x is not in the array, by construction $\Rightarrow x \neq c_i \ \forall i$, which is a contradiction.

QED

What about the decision problems (from integer or string of char to YES or NO)?

a	b	...	aa	...
0	1	2	3	...
YES	YES	NO	YES	...
1	1	0	1	...

Where the third row characterizes the decision proble P, and we can create $x_P \in [0,1]$ from the last row: $x_P = 0.1101....$ That means that $\forall x \in [0,1]$, I can create a decision problem. So decision problems are uncountable because in bijection with $[0,1]$.

Matlab codes are countable (e.g. string of char, ASCII codes $\Rightarrow$ integer $\in \mathbb{N}$). And every Matlab code can solve at most one decision problem. So most decision problems are undecidable! But Turing's proof is interesting because it provides a specific, interesting decision problem that is undecidable.

c. Undecidable problems

Other undecidable problems :

i. Malicious problem

Input : A piece of code/program
Ouput : YES if malicious (virus)
The perfect anti-virus does not exist.

ii. Comparison between two programs

Input : Two Matlab codes `M1.m` and `M2.m`
Ouput : YES if $M1(x) = M2(x)\ \forall x$ (they compute the same function)
How do we prove the comparison problem is undecidable?

Proof Suppose I want to check that M halts on x. Then I create an instance of the comparison problem :

- $M1(y) = M(x)\ \forall y$ (constant function)
- $M2(y)$ does not halt $\forall y$ (empty function, while true do 1)

If M does not halt on x, then $M1(y) = M2(y)\ \forall y$. If M halts on x, then $M1(y) \neq M2(y)$ for some y (in fact, all of them but we don't even need it). This is called a reduction of the halting problem to the comparison problem. It shows the halt problem is a particular case of the comparison problem, up to encoding. If we could solve the comparison problem algorithm, then we could solve the halt problem too.

iii. Arithmetic theorem problem

Input : Logical formula of arithmetics ($\forall x, \exists y \cdots +, \times, \cdots$ with $x, y, \cdots \in \mathbb{N}$)
Ouput : YES if it is a theorem, i.e. it can be proved from basic rules of $=, \times$ and logic
For instance, $\forall x, \exists y : 2y = x$ is not a theorem but $\exists x, \exists y : 2y = x$ is one.

Same question with $x, y, \cdots \in \mathbb{R}$: $\exists x : x^2 + 1 = 0$ is false but $\forall x, \exists y : x^2 + y = 0$ is true. This problem is actually decidable. For instance, Sturm's theorem can find the number of distinct real roots of a polynomial located in an interval. Some questions about real numbers are actually easier than questions about integers.

iv. Matrix mortality problem

Input : A finite set of 3×3 matrices with integral entries ($\in \mathbb{Z}^{3\times3}$)
Ouput : YES if there is a product of the matrices equal to 0 (e.g. $M_1M_2M_1M_3M_1^2 = 0_{3\times3}$)

v. Tiling problem (Berger, 1966)

Input : A finite set of tetris-like shapes
Ouput : YES if you can tile the entire plan with copies of these shapes (e.g. Figure G.1)

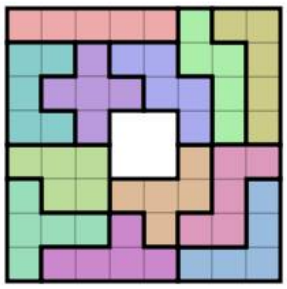


Figure G.1.: Tiling problem

vi. Diophantine equations (Matiyasevich, 1970)

Input : A polynomial $P(x_1, \ldots, x_m)$ with coefficients in $\mathbb{Z}$
Ouput : YES if $\exists x_1, \ldots, x_m \in \mathbb{Z} : P(x_1, \ldots, x_m) = 0$

We can prove all that all these problems are undecidable by reduction from halting problem or other problems already proved undecidable.

H. Complexity classes

TEMPORARY - CM11

1. Complexity classes

a. Time

Among decidable problems, which one can be solved with a reasonable amount of resource?

We generally consider that the problem in P use a reasonable amount of resources. Where P={ decision problems that can be solved with an algorithm running in $O(n^d)$ time in worst case for some d }.
This class does not depend of the language (*Python*, *Matlab*, *Turing machine*) we use to code the algorithm.

Theorem H.1. There is a decidable problem not in P.

Proof We can use the diagonal argument !
Idea : we enumerate a list of *Matlab* code that solves all problems in P. So, we enumerate all *Matlab* codes of the sort :

Code H.1: Enumeration of all P problems

```
    While flop <= (input size)^d + c
        (any block of code)
    end
    return yes % forced stop
```

We know that **every** problem in P is solved by a *Matlab* machine in this list, and every *Matlab* machine in this list runs in polynomial time.
Let's call this list : $M_1, M_2, ...$ and let's suppose you enumerate the possible inputs as $1, 2, 3, 4, ...$

For the diagonal argument, we create a problem A:
Input : n
Output : $\begin{cases} \text{Yes if } M_n(n) = \text{No} \\ \text{No if } M_n(n) = \text{Yes} \end{cases}$

This problem output the inverse of the diagonal of the table *problem* $\times$ *inputs* !

This problem A is not in P. Because **the diagonal is not a row of this table** since whatever the row i, we have that $M_i(i) \neq A(i)$! This implies that we found a problem $A \notin P$.

QED

We can repeat the argument for other classes of complexity as for example for :
$EXPTIME$={decision problem solved by an algorithm in $O(2^{(inputsize)^d})$ time}.
Thanks to this, we can create a hierarchy of problems

b. Space

We are also interested by the memory complexity. Generally we consider that an algorithm that use a reasonable amount of memory is in $PSPACE$.
$PSPACE$= { decision problems that can be solved by an algorithm consuming memory in $O(n^d)$ for some d }
For the time complexity, we saw that we can create a hierarchy of time complexity via the *diagonal* arguments and we'll see that we can do the same for the space complexity.

Theorem H.2. $P \subset PSPACE$

Proof In N steps of time can can only write N new symbols in memory.
The memory use is $\leq n + N$ where n is the input size.
$\implies$ If $N \in O(n^d)$ then $n + N \in O(n^d)$ $\forall d \in N$

QED

Theorem H.3. $PSPACE \subset EXPTIME$

Proof If the *Matlab* machine is twice in the same memory configuration while reaching the same instruction in the program then it **must loop** and never stop. If N memory cells are used then there are at most (#code line $\times 2^N$) instructions before looping.

$\implies$ Since a *Matlab* machine solving a problem in $PSPACE$ must always stop [1]. It must do so before $O(2^{n^d})$ instructions, where space used is n^d.

QED

c. Relation between P, $PSPACE$ and $EXPTIME$

There are 3 possibilities since $P \neq PSPACE$:

$$\begin{aligned} P &= PSPACE \subsetneq EXPTIME && \text{(H.1)}\\ \text{or } P &\subsetneq PSPACE = EXPTIME && \text{(H.2)}\\ \text{or } P &\subsetneq PSPACE \subsetneq EXPTIME && \text{(H.3)}\\ & && \text{(H.4)} \end{aligned}$$

[1] it can't loop since it always must return a solution

The first one is not possible⟨2⟩. But we don't know if it's the second or the third one. People "believe" that it is the third one, $P \subsetneq PSPACE \subsetneq EXPTIME$.

d. NP

Now let's see another important class of time complexity :

NP={decision problems such that $\exists$ *Matlab* machine $M(x, y)$:

$$\begin{cases} M(x,y) \text{ runs in } O(|x| + |y|^d)\ \forall d \in N \\ x \text{ is a Yes-instance } \textit{iff } \exists y : |y| \in O(|x|^k) \text{ and } M(x,y) \text{ returns Yes} \end{cases}$$

}

This y is called a certificate for x.

Example H.1. Hamiltonian Graph
Input : *A* graph G
Output : Yes *iff* G is Hamiltonian (<=> $\exists$ Hamiltonian cycle in G)
Here we have $x = G$ and y =cycle where x is Hamiltonian iff $\exists$ cycle y s.t. y is an Hamiltonian cycle in x. $M(x, y)$ checks that y is Hamiltonian in x.

Since $M(x, y)$ is a polynomial algorithm => Hamiltonian Graph $\in NP$ and we have that $|y| \in O(|x|^k)$, we have that this problem $\in NP$.

Remark. We can see the NP problems are problems that are *"easy to check"* once solved. Many natural and practical relevant combinatorial problems are in NP.

Theorem H.4. $P \subset NP$

Proof It is trivial for the certificate $y \equiv 0$ e.g. (no certificate needed) *"If it's easy to solve, it's easy to check"*.

QED

Theorem H.5. $PSPACE \subset NP$

Proof For a problem in NP s.t. x is a Yes-Instance iff $\exists y$: $|y| \leq |x|^d$ and that $M(x, y)$ =Yes. Where $M(x, y)$ runs in $(|x| + |y|^d)$.

We can have the algorithm (probably not the best time complexity to solve the problem)

Code H.2: $PSPACE \subset NP$ control algorithm

```
for all y=1,2,3...,2^|x|^d
    if M(x,y) = Yes then return Yes
end
```

⟨2⟩it is possible to show that there is a $PSPACE$ program which use $O(2^{n^d})$ instructions

Which solve the problem and runs in polynomial space (for each new y we can wipe out the memory and use it again). $M(x,y)$ use polynomial-space $\forall |y| \leq |x|^d$.

QED

Remark. The Hamiltonian Graph problem $\in PSPACE$ because we can simply write a program that generate all the possible cycle and verify if one is an Hamiltonian cycle. This program just keep the current cycle and graph in memory, so the memory require $\in PSPACE$.

We don't know if $P = NP$ or $P \subsetneq NP$. People believe that $P \subsetneq NP$.
It is one of the most important conjectures in maths!

Remark. We know: if $P \neq NP$ then $NP \neq PSPACE$ thanks to a diagonal argument.

2. Reducibility

For two problems S and T, we want to say if S is *"easier"* than T. We say S is **reducible** to T, written $S \leq_p T$, if $\exists f : N \leftarrow N$, computable in polynomial-time.
s.t x is a Yes-Instance for S **iff** $f(x)$ is a Yes-instance for T. This is called a reduction.
Intuition: through the function f we show that S is in fact a particular case of T.

Example H.2. Reduction between the Clique problem and the Independent Set problem

1. Clique
   Input : Graph G, integer k
   Output : Yes if $\exists$ k-clique (clique of at least k nodes) in G
2. Independent set
   Input : Graph G, integer k
   Output : Yes if $\exists$ k-independent set (set of at least k independent nodes) in G

We have that Clique $\leq_p$ IndependentSet $\leq_p$ Clique.
Where for the reduction, we use the function $f(G,k) = (\bar{C}, k)$ where the $\bar{C}$, the complement of G : edge $ab \in G$ *iff* $ab \notin \bar{G}$ and nodes(G)=nodes($\bar{G}$)

Remark. Clique $\equiv_p$ IndependentSet (equivalent problems for polynomial-time reduction). If one is in P then so is the other or NP or $PSPACE$.

TEMPORARY - CM12

We have this expression:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

However, we do not know if these they are different or not.
Order on problems: $S \leq_p T$, where $\leq_p$ is the poly-time reduction.

Theorem H.6. There is a problem $S \in NP$ such that $\forall T \in NP : T \leq_p S$

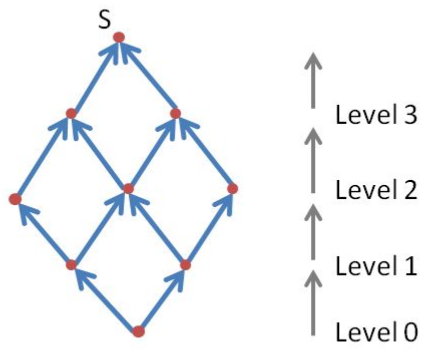


Figure H.1.: Reducibility with multiple roots being in fact only one.

It is not trivial, we could have the problems reducible for other in many way and with multiple roots but, in fact, there is only one (see Fig H.1 with $\nearrow$:$\leq_p$: the element which point to the other is the simplest one).

a. SAT Problem

Definition H.1. SAT problem :

- Instance: A boolean formula, e.g. $(P \vee \neg Q) \wedge Q$, with $\vee$ which corresponds to "or", $\neg$: "not" and $\wedge$: "and".
- Output: Yes, if exists true value for the variables so that the formula evaluates to true, e.g.
  - $(P \vee \neg\ Q) \rightarrow$ Yes, for P= true
  - $Q \wedge \neg\ Q \rightarrow$ No

Theorem H.7. $\underbrace{\text{SAT} \in NP \quad \textit{and} \quad \underbrace{\forall T \in NP : T \leq_p \text{SAT}}_{\text{"SAT is NP-hard"}}}_{\text{"SAT is NP-completed"}}$

Proof (Sketch)

- SAT $\in$ NP: certificate= list of TRUE/FALSE values for all variables that make a formula evaluated to YES $\Rightarrow$ ok!
- $\forall T \in NP : T \leq_p$ SAT:
  Let T be in NP, then it exists a Turing Machine M such that $M(x, c) = YES$ for some poly-length certificate c, running in poly-time, if and only if x=YES instance of T.

  We detail the Turing Machine in Fig H.2.

  $\Rightarrow \mathcal{O}\left(|x|^{2d}\right)$ variables $P_{t,m}$=TRUE if cell m at time t is 1, with t, the time and m, the space. $\mathcal{O}\left(|x|^{d}\right)$ boolean variables describing the head state $P_{t,s}$=TRUE if and

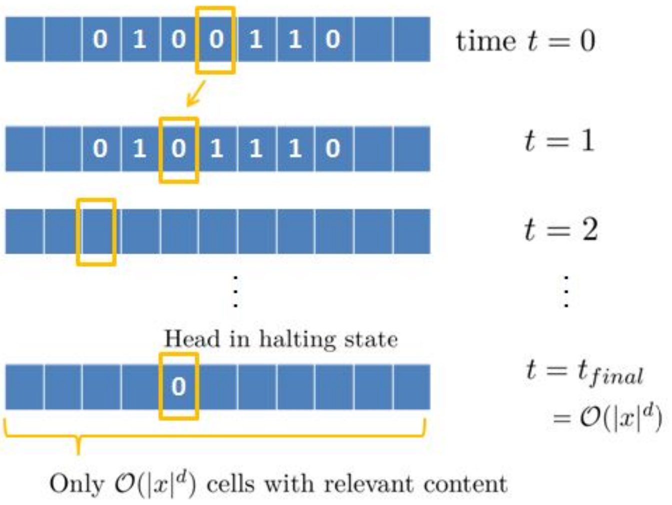


Figure H.2.: Turing Machine

only if Head is in state s at time t. $\mathcal{O}\left(|x|^d\right)$ variables $Q_{t,m}$=TRUE if and only if head is reading cell m at time t.

We can describe the rules of the Turing Machine by a boolean formula: e.g. t, m, the transition $s_1 \longrightarrow s_2$ if we read a 0. (If P_{t,s_1} and $\neg P_{t,m}$ and $Q_{t,m}$ then P_{t+1,s_2}).

(Reminder: " If A then B" $\Leftrightarrow \neg A \vee B$)

The whole computation is encoded into a big, but poly-size (and computed in poly-time), boolean formula $\phi_{x,c,M}$ (described by $P_{t,m}$ for $t = 0 \equiv$ Initial state).

The question "$\exists ?c : T(x, c) = YES$" (for a given x).

Amounts: "Are there truth values for the $P_{0,m}$ encoding c such that $\phi_{x,c,T}$=YES ?", with x and T which are given and c that to be found.

$\Rightarrow$ This is a SAT-instance

$\Rightarrow T \leq_p$ SAT

QED

Intuition: SAT is "the hardest" problem in NP
The main technique to prove that a problem S is NP-complete is:

- prove that S $\in$ NP
- for some other NP-complete problem S_0, e.g. S_0 =SAT, prove that $S_0 \leq_p S$

You can conclude:

- S is NP-complete
- $S \equiv_p S_0 \equiv$ SAT

Since we conjecture that $P \neq NP$, the practical consequence is that we cannot reasonably hope to find a poly-time algorithm for S.

b. NP-Complete Problems

To this day, thousands of interesting problems have been proved NP-complete.

Example H.3. NP-complete problems: 3 (possibly negated) variables, related by 1.
Definition H.2. 3-SAP:

- Instance: A boolean formula in the form $\underbrace{(P \vee \neg Q \vee R)}_{\text{"}clause\text{"}} \wedge (\neg P \vee \neg R \vee S) \wedge (P \vee P \vee Q)$. The clauses are linked by $\neg$.
- Output: YES if satisfiable (i.e. true for some value of $P, Q, R, S, \cdots$)

Theorem H.8. 3-SAT is NP-complete

Proof

- 3-SAT $\in$ NP: clear
- 3-SAT is NP-hard ?, we prove SAT $\leq_p$ 3-SAT

QED

The reduction proceeds by transforming in poly-time every boolean formula into the normal form $(\vee \quad \vee) \wedge (\vee \quad \vee) \wedge \cdots$ using the identities of boolean logic in a systematic way.
e.g. :
$(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$ distribution
$\neg(P \vee Q) \equiv \neg P \vee \neg Q$ (de Morgan) and More.

3-SAT can help prove that even more problems are NP-complete.

c. Clique Problem

Theorem H.9. CLIQUE is NP-complete

Proof

- CLIQUE $\in$ NP: certificate of a YES-instance= list of nodes in the CLIQUE
- CLIQUE is NP-hard ?
  we show 3-SAT $\leq_p$ CLIQUE.
  We have to transform every formula $\phi = \underbrace{(\cdots \vee \cdots \vee \cdots) \wedge (\cdots) \wedge (\cdots)}_{\text{k clauses}}$ into a CLIQUE instance: a graph or an integer.

  We illustrate on an example:
  $\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)$: 4 clauses
  $f(\phi)$ =(Graphe,4)=instance of CLIQUE H.3.

  All pair of nodes are linked except:

  - inside a triplet representing a clause
  - x_i with $\neg x_i$

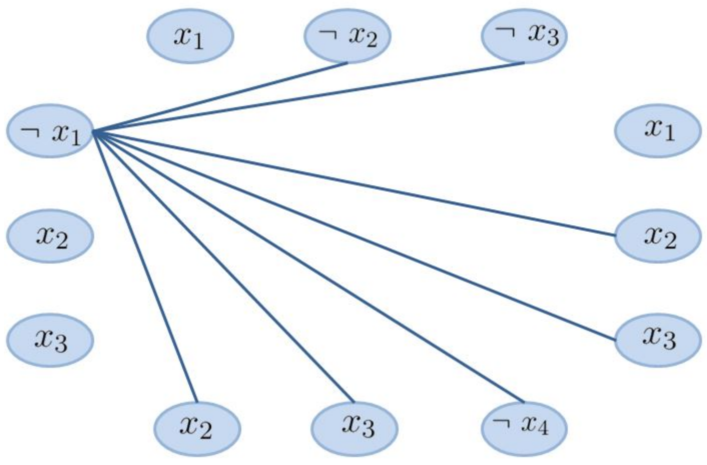


Figure H.3.: example of clique

We check, if ϕ is satisfiable then it exists k-clique in Graph. Indeed, if ϕ satisfiable by:

$$\begin{array}{lll} \neg x_1 & =\text{TRUE in clause} & 1 \\ \neg x_2 & =\text{TRUE in clause} & 2 \\ x_3 & =\text{TRUE in clause} & 3 \\ x_4 & =\text{TRUE in clause} & 4 \end{array}$$

(at least one is true in every clause - see fig H.4)

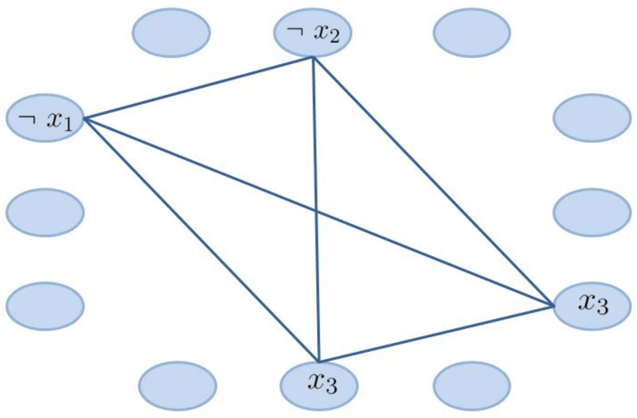


Figure H.4.: example of clique

If exists k-CLIQUE in a graph then ϕ is satifiable. Indeed, we find clique then we can see that the assignment :

$$\begin{array}{rcl} \neg x_1 & = & \text{TRUE} \\ \neg x_2 & = & \text{TRUE} \\ x_3 & = & \text{TRUE} \\ x_3 & = & \text{TRUE} \end{array}$$

(and anything for other variables) makes Φ true (because well defined and makes one possibly negated) variable true in every clause.

QED

Often practical problems are optimization problems.

Definition H.3. MAX clique:

- Instance: A graph
- Output: Clique of max size

We convert this into a decision problem: CLIQUE and we can run CLIQUE for several k (e.g.by dichotomy)