Université catholique de Louvain

LINMA2111
Discrete Mathematics II : Algorithms and Complexity

Partial course notes

Teacher: Jean-Charles Delvenne
Students and assistants:
2015 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
2016 Florentin Goyens, Florimond Houssiau, Adissa Laurent, Guillaume Olikier & Harold Taeter
2019 Maxime Clément de Cléty, Ny Randresihaja , Thomas Saillez
2020 Jehum Cho, Sébastien Colla, Najlae Sahbi, Breno Tiburcio, Igor Teixeira Casatti
2021 Adrien Banse

2020

Contents

1. Sorting Algorithms

This first chapter discusses some classic sorting algorithms. Although interesting on their own, they are meant here to illustrate some of the main concepts developed in the following chapters (e.g. divide and conquer, dynamic programming, random algorithm and complexity).

A sorting algorithm is defined as any algorithm with the following input-output specification.

Code 1.1: Inputs and outputs of sorting algorithms

```
1 INPUT  : array T = (T(1), T(2), ..., T(n)) of n numbers (or elements of a totally ordered set, like words ordered alphabetically)
2 OUTPUT : array T sorted
```

1.1. Selection Sort

First, we consider a naive algorithm: **Selection Sort**. This first algorithm can be formulated in informal pseudo-code as follows:

Code 1.2: Pseudo-code of the selection sort algorithm

```
1 for i from 1 to n-1   % loop invariant : T(1) ... T(i-1) are sorted
2     select the smallest element in the sub array from index i to n
3     swap it with element at index i
```

What about the time complexity of this algorithm? For any instance we have

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

On the use and abuse of the Θ notation, as well as $\mathcal{O}(.)$ and other related notations, see the Appendix.

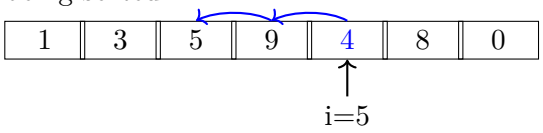
The complexity of this algorithm is $\Theta(n^2)$ for each instance of size n (meaning in this situation, for each possible input of n entries to be sorted). Thus the Worst Case Time Complexity (defined as the maximum running time over all instances of size n), Best Case Time Complexity or Average Case Time Complexity (average time over all instances of size n, for some probability distribution over those instances) are all in $\Theta(n^2)$.

1.2. Insertion Sort

Code 1.3: Pseudo-code of the insertion sort algorithm

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

Example 1.1. We can consider a small example, the first four elements of the array being sorted:



Regarding the time complexity, we have:

- in the 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.
- The best case corresponds to an array which is already sorted: $\Theta(n)$
- Average case: we assume instances come with uniform probability distribution over all possible orders ($= n!$ orders). Then:

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

The average case is defined with respect to a given the probability distribution of the instances of size n, needed to compute $\mathbb{E}[t_i]$). This makes average-case complexity a trickier concept than worst-case complexity, as the relevant probability distribution may be application dependent. For instance, when it comes to sorting, the relevant probability distribution is always uniform over all permutations, but may be for some applications biased towards almost-sorted instances, for instance. In this case the Average Case Complexity fo Insertion Sort may be in $o(n^2)$.

1.3. QuickSort

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

- Split into smaller sorting problems.
- Solve the smaller sorting problems (recursively)

- Recombine the outputs, into a sorted instance.

In particular, in this case:

Code 1.4: Pseudo-code of the quicksort 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]
```

About the complexity of this algorithm:

- Worst case: when the instance is already sorted !
  In this case: $\left.\begin{array}{ll} T_{low} & = 0 \text{ entries.} \\ T_{high} & = n-1 \text{ entries.} \end{array}\right] \Rightarrow$ maximally unbalanced subproblems.

  The worst-case time complexity of the QuickSort algorithm is $\Theta(n^2)$. Let t_n be the running time of QuickSort on a sorted instance of size n. Since the partitioning takes $\Theta(n)$ (one pass over the data), we have the worst-case recurrence equation:

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

- Average-case complexity: call t_n the (random) running time of a uniformly random instance of size n. Then the size of T_{low} and T_{high} are on average $(n-1)/2$, say $n/2$ for simplicity. Therefore we would like to write, heuristically at least:

$$\begin{aligned} t_n &= t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n) \qquad (1.1) \\ &= 2 \cdot t_{\frac{n}{2}} + \Theta(n) \\ &= 2\left(2 \cdot t_{\frac{n}{4}} + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= 2\left(2 \cdot \left(2 \cdot t_{\frac{n}{8}} + \Theta(\frac{n}{4})\right) + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= \dots \\ &= 2^{\log_2(n)} \cdot t_1 + \Theta(n \log_2(n)) \\ &= \Theta(n \log_2(n)) \end{aligned}$$

Remark. We have $\log_a(n) \in \Theta(\log_b(n))$, $\forall a, b > 1$, therefore we may forget the base and speak of $\Theta(n \log n)$ unambiguously.

This reasoning is not quite rigorous! Instead of $t_n = t_{\frac{n}{2}} + t_{\frac{n}{2}} + \Theta(n)$, we should write

$$\mathbb{E}[t_n] = \mathbb{E}[t_{|T_{low}|}] + \mathbb{E}[t_{|T_{high}|}] + \Theta(n)$$

Calling $s_n = \mathbb{E}[t_n]$, and averaging over all possible values of $k = |T_{low}|$, and upper bounding $\Theta(n)$ by cn the true recurrence we have to solve is

$$s_n \leq cn + \frac{1}{n}\sum_{k=0}^{n-1}(s_k + s_{n-k-1}) \tag{1.2}$$

$$\leq cn + \frac{2}{n}\sum_{k=0}^{n-1} s_k. \tag{1.3}$$

From the intuitive reasoning above we suspect that it admits a solution of the form $s_\ell \leq c_1 \ell \ln \ell + c_2$ for all ℓ, and some c_1, c_2 to be found. Let us pick c_2 large enough so that this is true for low values of n, say for all $n \leq n_0$ (where n_0 is arbitrarily chosen). This is the base case. Let us now tackle the values $n > n_0$. Assume $s_\ell \leq c_1 \ell \ln \ell + c_2$ is true for $\ell = 0, 1, \ldots, n-1$. We have to prove that $s_n \leq c_1 n \ln n + c_2$.

Before we prove it, let us anticipate that we will have to evaluate sums $f(0) + f(1) + \ldots + f(n-1)$, for $f(\ell) = c_1 \ell \ln \ell + c_2$. Using a common trick, we replace the sum $\sum_{\ell=0}^{n-1} f(\ell)$ by an integral $\int_0^n f(x)dx$, in the idea that the sum is indeed the Riemann sum approximating the integral, replacing the area under the continuous curve with rectangles of unit width. In fact, if the continuous function s_x is non-decreasing, then we can even state $s_0 + s_1 + \ldots + s_{n-1} \leq \int_0^n s_x dx$ (because all the rectangles are under the curve, thus underestimate the area under the curve).

We are now ready to use the induction hypothesis $s_\ell \leq c_1 \ell \ln \ell + c_2$ for all integers $k = 0, 1, \ldots, n-1$ and check that $s_n \leq c_1 n \ln n + c_2$ is indeed verified whenever () holds. We bound the rhs of ():

$$\begin{aligned}
cn + \frac{2}{n}\sum_{k=0}^{n-1} s_k &\leq cn + \frac{2}{n}\sum_{k=0}^{n-1} c_1 k \ln k + c_2 \\
&\leq cn + \frac{2}{n}\int_0^n c_1 x \ln x + c_2 dx \\
&\leq cn + \frac{2}{n}(c_1 \frac{x^2}{2}\ln x - c_1 \frac{x^2}{4})\Big|_0^n + 2c_2 \\
&= cn + c_1 \frac{n^2}{n}\ln n - c_1 \frac{n^2}{2n} + 2c_2 \\
&= (c - \frac{c_1}{2})n + c_1 n \ln n + 2c_2
\end{aligned}$$

from which it is clear that the statement $s_n \leq c_1 n \log n + c_2$ is true for any choice of c_1 large enough so that $(c - \frac{c_1}{2})n_0 + c_2 \leq 0$.

A similar argument works to prove that $s_n \geq b_1 n \log n + b_2$ for some constants b_1 and b_2, proving that s_n (which is $\mathbb{E}[t_n]$) is in $\Theta(n \log n)$.

From this development we see that the heuristic argument above was useful to guess the right answer, then confirmed by a fully rigorous argument. The formal argument also brings us with a useful observation of independent interest that we extend and summarize here for later use:

Theorem 1.1. For a non-decreasing nonnegative function $f(.)$, we have the inequal-

ities:

$$f(0) + f(1) + \ldots + f(n-1) \leq \int_0^n f(x)dx \leq f(1) + \ldots + f(n-1) + f(n) \quad (1.4)$$

This closes the proof of average complexity for QuickSort.

Now, what if we don't know whether our instances are uniformly distributed? This leads to the next algorithm.

1.4. Randomized QuickSort

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 QuickSort runs in expected time $\Theta(n\log(n))$. Therefore the worst-case expected time of this random algorithm is also $\Theta(n\log n)$. Note here that the expectation is taken over the random choices of the algorithm, and the worst case is the maximum over all instances of size n. So the worst-case expected time is the maximum (over all instances x of size n) of the expected running time (over all random choices of the algorithms) of the algorithm on instance x.

This is to be distinguished clearly from the average case of the deterministic QuickSort, where the expectation was taken over the probability distribution given to the instances.

It may look surprising at first sight that Randomized QuickSort is preferred over its deterministic version, as we ditch any information we might have on the instances on purpose, by a preliminary shuffling (or randomized pivot choice). Intuitively, it makes sure that the instance is not too often close to the worst case. The randomized algorithm puts all instances on an equal footing. This is the so-called *Robin Hood effect*, i.e. we take from 'rich' instances and give to the 'poor'. Now, every instance can be unlucky from time to time but lucky most of the time.

Here we may wonder how unlikely exactly it is that we are unlucky, i.e. have a running time much above the expected value. Given that a random algorithm like Randomized QuickSort taking an expected time $f(n)$ on a worst-case instance of size n, what are the chances that the actually observed running time is much higher than $f(n)$? Markov's inequality offers us a crude but universal answer.

Theorem 1.2 (Markov's Inequality)**.** For a random variable T taking only non-negative values, the probability that $T > a\mathbb{E}[T]$ (for any $a > 0$) is no larger than $1/a$.

In our case, it implies for example that the probability to observe a running time of more than $100f(n)$ is less than one percent.

QuickSort and its randomized version were proposed in 1961 by the British computer scientist Tony Hoare (Turing Prize 1980). It is still one of the most used sorting algorithms in practice.

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

1.5. Merge Sort

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

Code 1.5: Pseudo-code of the merge sort algorithm

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

The time of merging is $\Theta(n)$, so we have the recurrence equation:

$$\begin{aligned} t_n &= 2 \cdot t_{\lceil \frac{n}{2} \rceil} + \Theta(n) \\ &= \Theta(n \log(n)) \text{ (see equation (1.1))} \end{aligned}$$

This is the same complexity class as Quick Sort. Which algorithm to use?

- Randomized QuickSort:
  - $\ominus$ It needs to generate randomness: not so easy for a computer!
  - $\ominus$ We can be unlucky.
  - $\oplus$ Hidden constants are smaller if implemented efficiently.
  - $\oplus$ can be implemented "in place": by swapping entries on T itself, without the need for another array to store intermediate results $\rightarrow$ there is very little extra memory (of the order of $\Theta(\log n)$) for intermediate variables.
- Merge Sort:
  - $\ominus$ Need for more memory ($\Theta(n)$) when operating on direct access arrays; this memory is used by the stack of ongoing recursive computations on subarrays
  - $\oplus$ But little extra memory needed if the instance is given by a linked list instead
  - $\ominus$ Hidden constants in the $\Theta(.)$ notation are typically higher than for QuickSort

1.6. Data structures

One may now wonder if there is a deterministic algorithm in $\mathcal{O}(n \log n)$ (worst case complexity) which sorts "in place" (meaning, as we recall, that just operates swaps on the original array and does not need to copy to another structure, thus requiring little extra memory), thus combining the determinism of Merge Sort with the good performances of QuickSort.

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.

For the sake of the example, let us make a little digression about two well studied data structures, stacks and queues. Both are lists (i.e. totally ordered sets) of entries with the same static structure but different ways to access/modify it. The difference is graphically explained on Figure 1.1.

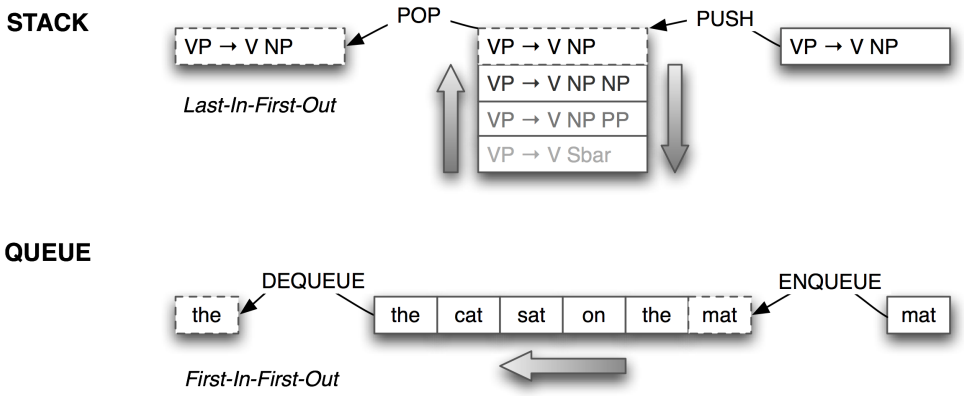


Figure 1.1.: Graphic representations of the stack and the queue structures. Elements are inserted using the Push/Enqueue action and removed using the Pop/Dequeue action.

We define a data structure as a static structured set (like a set with a total order on the elements) with a set of operations that describe how to access or modify the structured set (like read, insert, merge, delete, etc.). It can thus be seen as an abstract mathematical concept, that can be implemented in many ways.

1.6.1. A data structure for sorting

If we get back to our sorting problem, the structure we need is a priority queue: each entry is associated with a number (or an element from a totally ordered set) called the 'priority' of the entry, with the following operations

- Insert(entry, priority), which inserts the entry with the given priority;
- ExtractMax which removes and outputs the entry with highest priority

An example of the use of such a structure is a todo list where the elements are tasks associated with a certain degree of urgency.

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

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

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

1.6.2. Implementations of a priority queue

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

1.6.2.1. Unordered table

Using an unordered table, the operations complexity are

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

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

1.6.2.2. Ordered table

Using an ordered table, the operations complexity are

- Insert: $\mathcal{O}(n)$ [*cost of shifting the entries to insert the entry at the right place*]
- ExtractMax: $\mathcal{O}(1)$

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

1.6.2.3. "Heap"

Using a heap, as defined and developed below, 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 $\mathcal{O}(n \log n)$.

Remark. The different implementations of a data structure often offer trade-offs: one operation can be cheaper if another is costly.

1.6.3. 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)
- $\text{entry}(x) \leq \text{entry}(\text{father}(x)) \; \forall$ node $x \neq$ root

An example of a heap is shown in Figure 1.2.

1.6.3.1. Insert

To insert a new node into the heap (Fig.1.3), add it to the last level in the first free position and swap it with its father until the heap is properly re-established.
$\Rightarrow \mathcal{O}(\log n)$

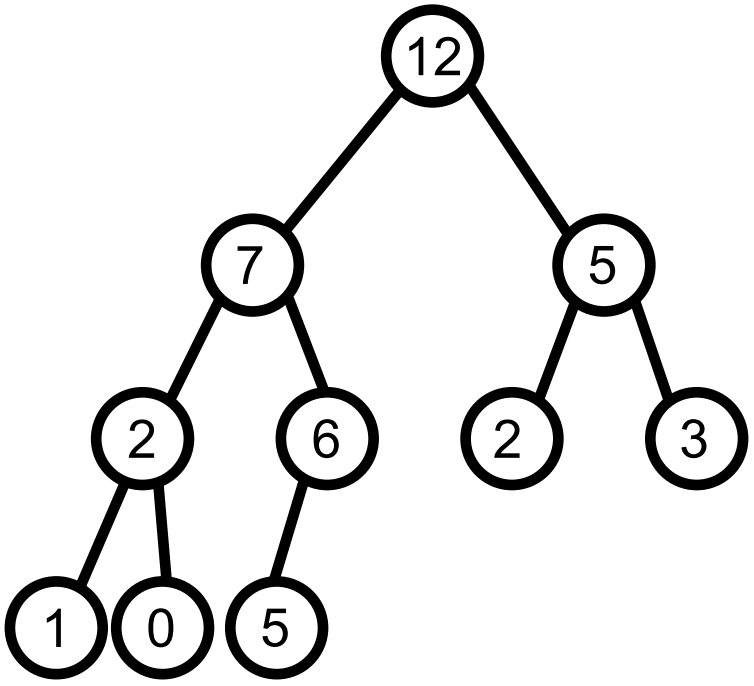


Figure 1.2.: Example of a heap

1.6.3.2. ExtractMax

To extract the maximum-priority node from the heap (Fig.1.4), 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.

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

1.6.3.3. HeapSort

The resulting sorting algorithm is called the HeapSort (proposed along with the heap data structure by John Williams, Welsch-Canadian computer scientist, in 1964) and has a complexity of $\mathcal{O}(n \log n)$.

Importantly, the heap may be stored as an array, if we read the entries from top to bottom and left to right. Because the heap is essentially binary complete, this is an unambiguous representation: every array can be interpreted as representing a certain, unique, heap. For example :

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

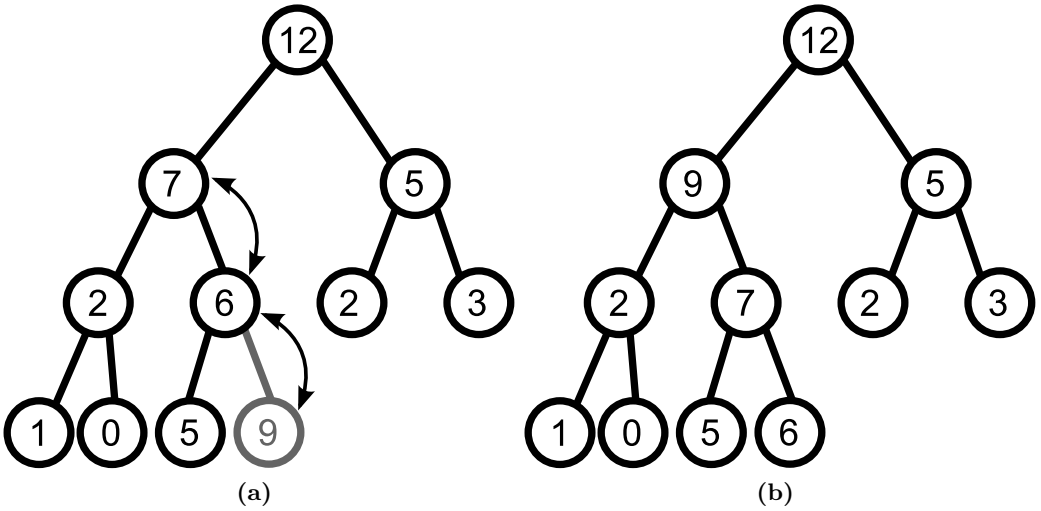


Figure 1.3.: A heap before/after adding one element

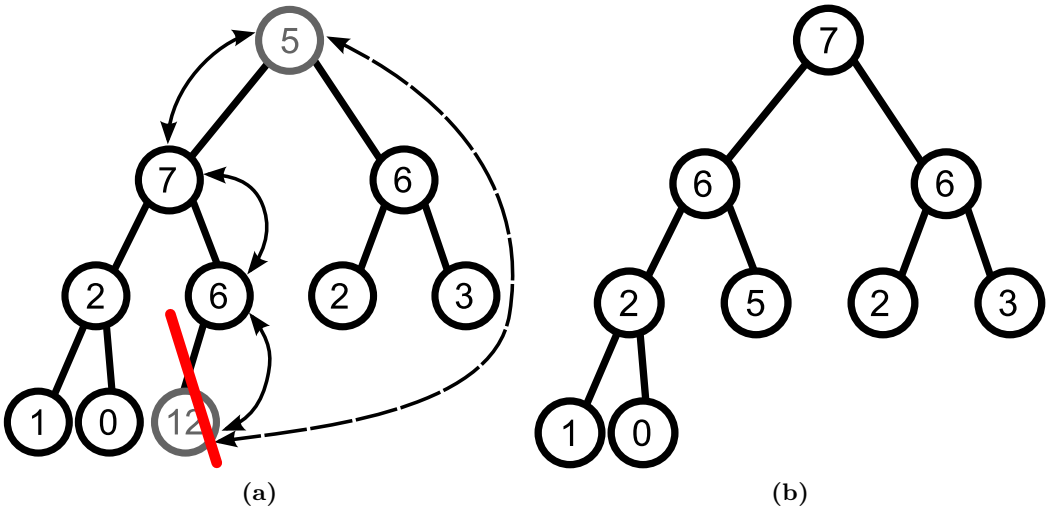


Figure 1.4.: A heap before/after removing the largest entry

with the following properties:

$$\begin{aligned}
&\text{LeftChild}(T(i)) = T(2i)\\
&\text{RightChild}(T(i)) = T(2i+1)\\
&\text{Father}(T(i)) = T\left(\left\lfloor \frac{i}{2} \right\rfloor\right)
\end{aligned}$$

Heapsort can therefore be implemented as an 'in place' sorting method, swapping entries of the initial array !

Remark. Create a heap from an input can be done in $\mathcal{O}(n)$ (admitted here), instead of the rough estimate $\mathcal{O}(n \log n)$ that we give here as the cost of n insertions.

1.7. 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 other than they are totally ordered (i.e. all we can test is "compare two entries", check that "$T(i) \leq T(j)$")?

Example 1.2. 20-questions game With 20 Yes-No questions, you can always guess a number between 1 and 2^{20} (roughly one million), but you're not guaranteed to be able to guess an arbitrary number between 1 and 2000000, no matter the questioning strategy.

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

Theorem 1.3. A binary tree (thus with at most 2 children for each node) with N leaves has at least $\lfloor \log_2 N \rfloor$ levels.

Proof. It is clear that the complete binary tree with ℓ levels will have 2^ℓ leaves, and any subtree will have less leaves than that. This proves the claim.

In order to connect with the arguments below, we give a version of the proof where leaves are first encoded into binary words, in the following way. Label each edge from a node to its (at most) two children 0 and 1 (see Figure 1.5). Every leaf is uniquely identified by a binary word encoding the path root $\to$ leaf. The level of a leaf is the length of the binary word, plus one. With ℓ levels, binary words are all of length $\ell - 1$ at most, one may only identify strictly less than 2^ℓ leaves. This proves again the claim. □

Theorem 1.4. No deterministic sorting algorithm can sort n entries (with just comparisons) in worst case time complexity in $o(n \log n)$.

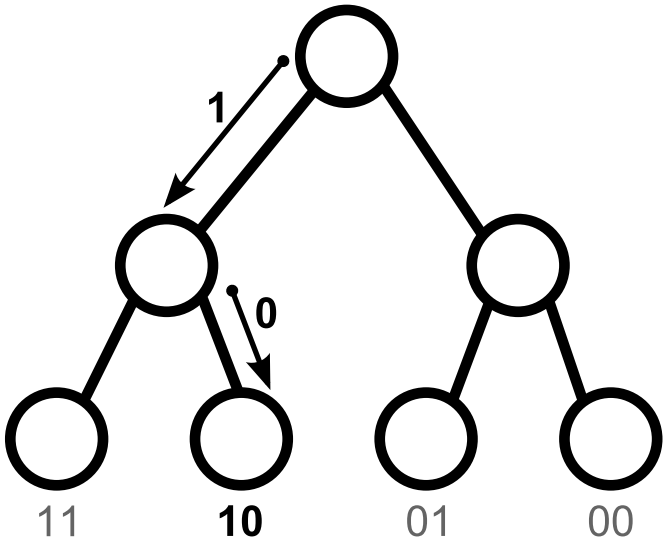


Figure 1.5.: Example of a binary tree

Proof. A sorting algorithm running on an array of n entries proceeds with a sequence of comparisons. For instance the first query to the data may be 'Is $T(1) < T(5)$?'. Depending on the answer, yes or no, it will eventually proceed to another comparison, e.g. 'Is $T(1) < T(5)$?' (following a 'Yes' to the first question) and 'Is $T(1) < T(6)$?' (following a 'No' to the first question). This sequence of comparisons and their answers can be represented by a decision tree (see Figure 1.6). We may number each path from root to a leaf with a binary number (encoding Yes as 1 and No as 0, for instance). In the end we must have

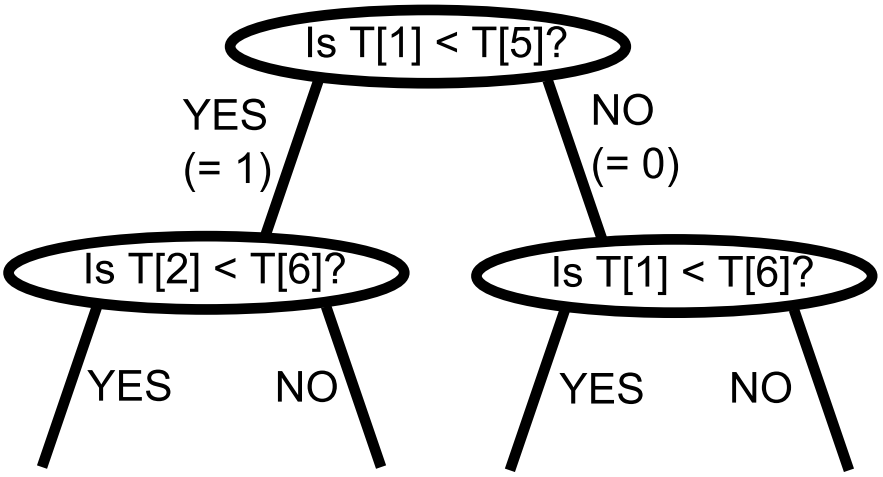


Figure 1.6.: Decision tree

sorted the initial data. The number of leaves of this tree is $n! = n(n-1)(n-2)...1$. Indeed, two distinct initial orders on the data require a different permutation to become correctly sorted, thus require a different sequence of actions from the algorithm, while a (deterministic) algorithm following the same computation path in the tree for two distinct instances will treat these instances in the same exact way. Thus each of the $n!$ possible initial orders must be associated with a different leaf.

This is the number of possible orders of the input array (assuming all different entries), 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).

Therefore this binary tree has at least $\log_2(n!)$ levels, and each path from root to tree is a sequence of binary answers queried from the data, thus a lower bound on the time of the computation.

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

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 1.5 (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 of another). Then the expected length (weighted by (ρ_i)) is at least $-\sum_{i=1}^{N} \rho_i \log_2 \rho_i$.

This theorem is foundation of Shannon's information and communication theory. It has a considerable importance in modern ICT technologies. For example it is the basis of data compression algorithms.

The quantity $-\sum_{i=1}^{N} \rho_i \log_2 \rho_i$ is called the *entropy* of the probability distribution. The entropy of any probability distribution over N elements is at most $\log_2 N$, which is attained for the uniform distribution.

In the case of uniform probability distribution, Shannon's theorem states that the average length of any binary prefix encoding of N elements must be at least $\log N$.

If now the N elements are N leaves of a binary tree, encoded by their path from root to leaf (which form indeed a prefix code), we obtain:

Theorem 1.6. The expected length of a path root $\to$ leaf of an N-leaf binary tree, with probability distribution $\rho_1, \rho_2, ..., \rho_N$ on leaves, is at least the entropy $-\sum_{i=1}^{N} \rho_i \log_2 \rho_i$. In particular, if all leaves occur equally likely, the average length of the path is at least $\log_2 N$.

Theorem 1.7. The average-case complexity, for uniform distribution over all instances, of *any* deterministic sorting algorithm using comparisons is in $\Omega(n \log n)$.

Proof. Apply the previous theorem on the decision tree generated by the computation of the algorithm. □

Notice that if the instances are not uniformly distributed then the theorem leaves hope for a better than $\Omega(n \log n)$ average-case complexity.

Can randomized algorithms (outputting the correct answer after a random time) help us improve these bounds, derived for deterministic algorithms? After all, randomization on Quicksort helped improved the worst-case complexity spectacularly, so why not further?

Here we could revisit the arguments above, leaving the possibility that a single instance may follow several paths. But it is more powerful to invoke a general result comparing worst-case expected complexity with average case deterministic complexity: Yao's minimax principle.

Theorem 1.8 (Yao's minimax principle, 1977)**.** Consider a given probability distribution over instances of a given size of a given problem. Then there is a deterministic algorithm solving the problem for those instances, whose average-case complexity for this distribution is lower than the worst-case expected complexity of any random algorithm solving the same problem.

Proof. Consider all instances $x_1, \ldots, x_N$ of a given size, with probability $\rho_1, \ldots, \rho_N$.

Consider a random algorithm A, drawing random i.i.d. bits $B_0 B_1 B_2 \ldots B_K$ to decide of its random actions. Consider a specific bit sequence $B_1 B_1 B_2 \ldots B_K = b_1 b_1 b_2 \ldots b_K$. Then, knowing this specific sequence of bits, the algorithm is now deterministic, with a fixed running time $t(A, x_i | B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K)$ on each instance x_i. In other words, the kth call of the instruction 'Draw a random bit' is replaced with 'read b_k'.

Then the expected complexity on instance x_i is

$$\sum_{b_1 \ldots b_K} t(A, x_i | B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K) \text{Prob}(B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K)$$

The worst-case expected complexity is thus

$$\max_i \sum_{b_1 \ldots b_K} t(A, x_i | B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K) \text{Prob}(B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K)$$

This is certainly higher than the average-case expected complexity

$$\sum_i \rho_i \sum_{b_1 \ldots b_K} t(A, x_i | B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K) \text{Prob}(B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K)$$

which is itself higher than the

$$\min_{b_1 \ldots b_K} \sum_i \rho_i t(A, x_i | B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K)$$

which is the average-case complexity of the deterministic algorithm A with fixed sequence $B_1 B_2 \ldots B_K = b_1 b_2 \ldots b_K$. □

Applied to the case of comparison-based sorting, Yao's minimax principle tells us that the worst-case expected complexity of any random algorithm is even higher than the average-case deterministic complexity of sorting with respect to any distribution over the instances, in particular for the uniform distribution, which is proved above to be $\Omega(n \log n)$.

With supplementary knowledge on the data however, which allows us to query more than comparison of two items, we can beat the $\Omega(n \log n)$ bound.

1.8. Counting Sort

This sorting technique beats the $\Omega(n \log n)$ bound because it knows more about the entries than just their ordering: it knows that the entries to be sorted are numbers between 1 and k. It proceeds in the following way, on the input table [T(1) T(2) $\cdots$ T(n)].

1. Entries count (one pass on T):
   - 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. Copy entries of T to final array W (a second pass on T):
   - 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 $W(U(1) + \cdots + U(k-1) + 1), ..., W(U(1) + \cdots + U(k))$.

   A efficient way to implement this is to run, for $i = n, ..., 1$:

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

Remark. Often an entry is just a key to satellite data (a file, a picture, a book, etc.). In those cases, Counting Sort as described is *stable*: the order of two entries with the same key values is preserved. They are not swapped in the final array W. This is the reason why we make the last pass backwards, for $i = n, ..., 1$, rather than forward, for $i = 1, ..., n$, which would not respect stability

The worst case complexity of this algorithm is $\Theta(k + n)$, with k the size of the range and n the number of elements to sort, computed as follow:

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

2. Divide and Conquer

Some of the sorting algorithm (quick sort and merge sort) we have seen use the divide-and-conquer approach. Those algorithms proceed in three steps:

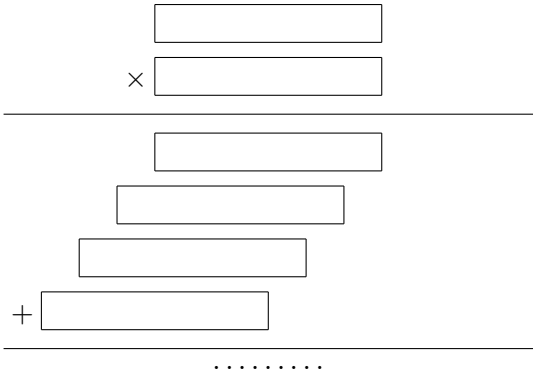
1. Divide the problem in smaller subproblems.
2. Conquer the subproblems by solving them recursively. When the subproblem is sufficiently small, solving it becomes straightforward.
3. Combine the solutions to the subproblems into the solution for the original problem.

Beyond the conceptual simplicity,the divide-and-conquer approach may also offer computational advantage as they are easy to distribute on several processors, in charge of computing different subproblems.

In this chapter we analyse the divide-and-conquer strategy for different specific problems: the multiplication of 2 large integers, the multiplication of 2 matrices, the inversion of a matrix and the selection of the i^{th} smallest element of a table.

2.1. Multiplication of 2 large integers

How to multiply 2 large integers (of n digits each)?
The elementary-school method for written multiplication makes n multiplication of one large integers by a single digit, followed by an addition:



: $\Theta(n^2)$

We can do better! Let's make a first attempt at a divide-and-conquer strategy.

$$\begin{aligned} x \times y &= (\underbrace{x_0}_{n/2 \text{ digits}} + 10^{\lfloor n/2 \rfloor} \underbrace{x_1}_{n/2}) \times (\underbrace{y_0}_{n/2} + 10^{\lfloor n/2 \rfloor} \underbrace{y_1}_{n/2}) \\ &= x_0 \times y_0 + 10^{\lfloor n/2 \rfloor}(x_1 \times y_0 + x_0 \times y_1) + 10^{2\lfloor n/2 \rfloor}(x_1 \times y_1) \end{aligned} \tag{2.1}$$

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

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

It has the same complexity as the elementary method but a larger hidden constant due to the management of recursive calls. This method is thus worse than the previous one. As an aside, let us first generalise the solution of the recursion above to general parameters, into the so-called Master Theorem, which will save us time future examples

2.1.1. Master Theorem for Divide-and-Conquer recurrences

The *Master Theorem* provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, like the one solved above.

Theorem 2.1 (Master)**.** Let t_n be defined on the nonnegative integers by the recurrence $t_n = at_{n/b} + f(n)$ (or $t_{\lfloor n/b \rfloor}$ or $t_{\lceil n/b \rceil}$) with $a \geq 1$ and $b > 1$ and $f(n)$ a given function. Then t_n has the following asymptotic bounds:

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 $f(n) \in \Theta\left(n^{\log_b a} \log^k n\right)$, then $t_n \in \Theta\left(n^{\log_b a} \log^{k+1} n\right)$;
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 2.1. 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 2.1, $f(n) \in \Theta\left(n^{\log_b a}\right) \rightarrow t_n \in \Theta(n \log n)$
- $t_n = 4t_{n/2} + \Theta(n)$: $a = 4$, $b = 2$ and $\log_2 4 = 2$
  By 1, $f(n) = \Theta\left(n^{\log_2 4 - 1}\right) \rightarrow t_n = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^2\right)$

2.1.2. Karatsuba-Ofman's algorithm (1962)

We can improve the previous attempt at a divide-and-conquer strategy. It is in fact possible to make only 3 multiplications of $\frac{n}{2}$-digits numbers, namely $x_0 \times y_0$, $x_1 \times y_1$ and $(x_0 + x_1) \times (y_0 + y_1)$. Indeed the cross products $x_1 \times y_0 + x_0 \times y_1$ (from equation (2.1)) can be expressed as $(x_0 + x_1) \times (y_0 + y_1) - x_0 \times y_0 - x_1 \times y_1$.

Code 2.1: `Multiply(x,y)` (Karatsuba-Ofman, 1962)

```
1  Write x as x_0+(10^(floor(n/2))) x_1
2  Write y as y_0+(10^(floor(n/2))) y_1
3  M1=(x_0+x_1)*(y_0+y_1)
4  M2=x_0*y_0
5  M3=x_1*y_1
6  x*y=M2+10^(floor(n/2))*(M1-M2-M3)+10^(2(floor(n/2)))*M3
```

Time complexity : Applying Theorem 2.1, we find

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

We can do even better! In order to do so, we first review the a classic, 'best-seller' Divide and Conquer algorithm: the Fast Fourier Transform.

2.1.3. The Discrete Fourier Transform

Let us consider a sequence of (real or complex) numbers $a_0, \ldots, a_{N-1}$. We can encode it as a polynomial $A(z) = a_0 + a_1 z + \ldots + a_{N-1} z^{N-1}$, where z is an indeterminate variable.

We now use the key equivalence between two descriptions of a polynomial: the list of N coefficients, or the evaluation of the polynomial in N pairwise-distinct values $z_0, z_1, \ldots, z_{n-1}$ of the variable z. The polynomials of degree $N-1$ form a vector space of dimension N, and those two representations correspond to two basis of the vector spaces. They are related by an invertible linear transformation, i.e. an invertible N-by-N matrix.

The two representations are computationally convenient for different tasks. For instance, evaluating a polynomial in an arbitrary value for z (distinct from the values z_0, z_{N-1}) is easy in the coefficient representation, from the expression $A(z) = (\ldots(a_{N-1} + a_{N-2}z)z + a_{N-3})z + \ldots)$ which requires $\Theta(N)$ elementary operations (additions and multiplications).

On the other hand multiplying two polynomials $A(z)$ and $B(z)$ of degree $N-1$ is difficult in the coefficient representation (where it corresponds to a convolution, involving $\Theta(N^2)$ elementary operations in a straightforward way), but easy in the evaluation representation.

Indeed consider $A(z) = a_0 + a_1 z + \ldots + a_{N-1} z^{N-1} + 0z^N + \ldots + 0z^{2N-2}$ as a polynomial of degree $2N-2$, and similarly $B(z)$ as a polynomial of degree $2N-2$. Then we evaluate A and B in $2N-1$ values $z_0, \ldots, z_{2N-2}$. The product $C(z) = A(z)B(z)$ is a polynomial of degree $2N-1$, and its evaluation representation is easily computed as $C(z_0) = A(z_0)B(z_0), \ldots, C(z_{2N-2}) = A(z_{2N-2})B(z_{2N-2})$, with $\Theta(N)$ elementary operations.

If we want to be able to take advantage of both representations, we must choose a list of evaluation points $z_0, \ldots, z_{N-1}$ (for degree $N-1$) for which the conversion between representations is fast to compute. The straightforward cost of evaluating the polynomial in N different values is $\Theta(N^2)$, since each evaluation is done in $\Theta(N)$. This conversion cost is too high to take advantage of convolution by passing into the evaluation representation, and then coming back into the coefficient representation.

One interesting choice of evaluation points is $z_k = \omega^i$ for $k = 0, \ldots, N-1$, where $\omega = \exp(2\pi i/N)$ is an Nth root of unity. In particular, $z_0 = 1$ and all z_k are all roots of

unity, satisfying $z_k^N = 1$. In this case, the sequence of evaluations $(A(z_0), \ldots, A(z_{N-1}))$ is called the *Discrete Fourier Transform* (DFT) of the sequence of coefficients $(a_0, \ldots, a_{N-1})$, denoted $\mathcal{F}_N(a_0, \ldots, a_{N-1})$. The coefficient representation is often called the *time domain* representation, and the sequence of evaluations the *frequency domain*, or *Fourier domain* representation in signal processing or physics.

The evaluation of $A(\omega^k)$ is facilitated by the following observation:

$$A(z) = A_{\text{even}}(z^2) + zA_{\text{odd}}(z^2), \tag{2.2}$$

where $A_{\text{even}}(z) = a_0 + a_2 z + a_4 z^2 + \ldots$ is the polynomial collecting the even terms and $A_{\text{odd}}(z) = a_1 + a_3 z + a_5 z^2 + \ldots$ collects the odd terms. Assume that N is even for simplicity. Then we obtain $A_{\text{even}}((\omega^k)^2) = A_{\text{even}}((\omega^2)^k)$, where $\omega^2 = \exp(2\pi i/(N/2))$ is none but the $(N/2)$th root of unity. Thus the sequence of $A_{\text{even}}((\omega^k)^2)$ for $k = 0, \ldots, N/2 - 1$ is the DFT of even coefficients, $\mathcal{F}_{N/2}(a_0, a_2, \ldots, a_{N-2})$. As for $k = N/2, \ldots, N-1$, we observe that $(\omega^2)^{N/2+\ell} = (\omega^2)^\ell$ (since $\omega^N = 1$). Thus the sequence of $A_{\text{even}}((\omega^k)^2)$ for $k = N/2, \ldots, N-1$ is once again $\mathcal{F}_{N/2}(a_0, a_2 \ldots, a_{N-2})$.

The same reasoning can be made with $A_{\text{odd}}((\omega^k)^2)$, which creates the sequence

$$(\mathcal{F}_{N/2}(a_1, a_3, \ldots, a_{N-1}), \mathcal{F}_{N/2}(a_1, a_3, \ldots, a_{N-1})) \quad \text{as } k = 0, 1, \ldots, N-1.$$

In total, running (2.2) for $z = \omega^k$, with $k = 0, 1, \ldots, N-1$, we finally find

$$\begin{aligned}\mathcal{F}_N(a_0, a_1, a_2, \ldots, a_{N-1})) = (&\mathcal{F}_{N/2}(a_0, a_2, \ldots, a_{N-2}), \mathcal{F}_{N/2}(a_0, a_2, \ldots, a_{N-2})) \\ &+ \omega(\mathcal{F}_{N/2}(a_1, a_3, \ldots, a_{N-1}), \mathcal{F}_{N/2}(a_1, a_3, \ldots, a_{N-1})).\end{aligned}$$

This leads to the conclusion that if N is a power of 2, we can apply this formula recursively, with a complexity to compute the DFT of N numbers:

$$t_N = 2t_{N/2} + \Theta(N).$$

The solution of this is, as we know:

$$t_N = \Theta(N \log N).$$

The inverse DFT (going from the Fourier domain to the time domain, ie coefficient representation) is just as easy. We sketch a quick argument (beyond the scope of this course) for the sake of completeness. Indeed let us comptue the square matrix F representing the DFT, i.e. the matrix F such that if I read the coefficients $(a_0, \ldots, a_{N-1})$ as the column vector a, then the Fa lists the entries of the Fourier domain representation $\mathcal{F}_N(a)$. It is easy to check that $F = (\omega^{k\ell})_{0 \le k,\ell, N-1}$. It can be checked (through explicit computation) that is an orthogonal matrix, with $F^{-1} = \frac{1}{N} F^*$ (where F^* denotes the conjugate transpose), so is 'essentially' the same matrix as F. Thus the inverse DFT is essentially the DFT itself, and can be computed in essentially the same way, with the same complexity.

This Divide and Conquer algorithm to compute the direct and inverse DFT is the celebrated Fast Fourier Transform, or FFT (Cooley-Tukey 1965, Gauss 1805). It is of utmost importance in digital communication technologies.

One application of the FFT is the multiplication of two polynomials in the coefficient representation, as we mentioned above. To recap, in order to compute $C(z) = A(z)B(z)$, of total degree $2N - 2$ (given that A and B are of degree $N - 1$):

- I compute the FFT of the coefficients of A (padded with zeros so as to reach $2N-2$ or the next power of two) and of the coefficients of B in the same way;
- I multiply the two resulting Fourier domain representation pointwise, which results in the Fourier domain representation of the coefficients of C;
- then I compute the reverse DFT in order to obtain the coefficients of C.

The total cost is in $\Theta(N \log N)$ elementary operations.

2.1.4. Schönhage-Strassen's algorithm (1971)

We return to the problem of large number multiplications.

We now write the integer x as a sequence of base-10 digits $x_{n-1}10^{n-1} + x_{n-2}10^{n-2} + \ldots + x_2 10^2 + x_1 10 + x_0$ and similarly for y. We now encode this sequence of digits into a polynomial $X(z) = x_{n-1}z^{n-1} + x_{n-2}z^{n-2} + \ldots + x_2 z^2 + x_1 z + x_0$, and similarly for the polynomial $Y(z)$. Note that $x = X(10)$ and $y = Y(10)$. We now have to compute the product polynomial XY and evaluate it at $z = 10$. The coefficients of XY are a convolution of the coefficients of X and Y, and thus costly ($\Theta(n^2)$) to compute directly, as mentioned in the previous section. Instead, we can compute the Fourier domain representations of X and Y, make the product XY in the Fourier domain, and come back in the coefficient representation for C, then evaluate for $z = 10$.

From this description, the complexity seems to sum up to a total complexity of $\Theta(n \log n)$, as a straight application of the previous section. However the Devil is in the details, of which we provide a brief account (beyond the scope of the course). The fact is that we counted the complexity of the FFT in terms of 'elementary operations'. These elementary operations involve the additions and multiplications of complex numbers, thus are not as 'elementary' as we would like. In signal processing applications, we are usually happy with a floating point representation of the complex numbers, resulting in a negligible loss of quality. In the present application, we want absolute precision on the result. Representing the large integers x and y by a floating point complex numbers is inappropriate. The required floating point precision would be exceedingly large. Therefore a variant of the FFT, the Fast Number Theoretic Transform, performing the operations in a finite field (such as integers modulo p) instead of the field of the complex numbers, is used. In other words, instead as seeing the digits $0, 1, \ldots, 9$ coefficients of X and Y as real numbers, we see them as integers modulo p, for some large enough prime p. The roots of unity are then to be found, not in the complex plane, but in some extension of the field of integers modulo p, which is itself a finite field.

This keeps finite representation of the Fourier coefficients at all times. However, even in this finite representation the coefficients are not in size $\mathcal{O}(1)$, and efficient multiplication of them calls for an application of FFT or Karatsuba-Ofman scheme once again. In the end, an optimal choice of the parameters of the algorithm leads to an $\Theta(n \log n \log\log n)$ complexity.

Schönhague and Strassen's algorithm is used for huge computations beyond 10 000 decimal digits, e.g. when computing many digits of π. In 2019, a Google team computed 31 trillion of digits of π. See a later chapter for insights on how to compute π.

2.2. Multiplication of 2 matrices

How can we multiply two n-by-n matrices? The naive algorithm suggested by the definition takes $\Theta(n^3)$ elementary operations.

2.2.1. Block Multiplication

We attempt a divide-and-conquer algorithm based on block multiplication. Assuming that n is an exact power of 2 in each of the n-by-n matrices (this ensures that while $n \geq 2$, the dimension $n/2$ is an integer) and partitioning A and B into four $n/2$-by-$n/2$ matrices, we get

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

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

Time complexity : we need to perform 8 products of $\frac{n}{2}$-by-$\frac{n}{2}$ matrices. Applying Theorem 2.1, we find:

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

This is not better than the naive algorithm! Nevertheless, following Karatsuba-Ofman's example, we can hope to express the eight products as linear combinations of less than eight intermediate products, as we now see.

2.2.2. Strassen's Algorithm

It turns out that we can do express AB with only 7 $\frac{n}{2}$-by-$\frac{n}{2}$ products (Strassen's algorithm, 1969):

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

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

Time complexity : Applying Theorem 2.1, we find:

$$t_n = 7t_{n/2} + \Theta(n^2) = \Theta\left(n^{\log_2 7}\right) = \Theta\left(n^{2.81}\right)$$

Pushing this line of argument further, we can do even better! Coppersmith-Winograd's algorithm (1981, 1990) is in $\Theta(n^{2.375477})$ but with a huge hidden constant, thus of no practical interest, unlike Strassen's algorithm. Subsequent improvement brought this exponent

down to 2.3728639 (Le Gall 2014) and more recently to 2.3728596 (Alman, Williams 2020) !

Many conjecture that there is an algorithm for any exponent > 2. In any case the complexity is in $\Omega(n^2)$, if only to read the input and write the answer. A slightly better (nontrivial) lower bound is $\Omega(n^2) \log n$ elementary operations (Ran Raz 2002).

2.2.3. Matrix Inversion

What is the cost of matrix inversion? Elementary methods such as LU factorisation cost $\Theta(n^3)$.

Theorem 2.2. 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.

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

   Thus $M(n) = \mathcal{O}(I(3n))$

   Since we know $I(n) \in \mathcal{O}(n^3)$, we have $I(3n) \leq 27I(n) \in \mathcal{O}(I(n))$.

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

   We reduce inversion to "a few" multiplications.

   - First assume $A = A^T \succ 0$ (symmetric, positive-definite).

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix} \quad , \quad \begin{matrix} B = B^T \succ 0 \\ D = D^T \succ 0 \end{matrix} \quad \text{then we can check that}$$

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

     with so-called Schur's complement $S = D - CB^{-1}C^T$
     There are:
     - 2 inversions: B^{-1}, S^{-1} of size $\frac{n}{2}$

– 4 multiplications: CB^{-1}, $C^T(CB^{-1})$, $S^{-1}(CB^{-1})$, $(CB^{-1})^T$ $[S^{-1}(CB^{-1})]$

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

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

- For general invertible matrices:

$$A^{-1} = \underbrace{(A^T A)^{-1}}_{\succcurlyeq 0} A^T$$

Thus we can apply the result above to $(A^T A)$. There is one more multiplication, which does not change the conclusion.

□

2.3. Random Select algorithm

How to find the i-th smallest entry of an array T of n elements?

Example 2.2. 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 simplifying assumption that all entries are different, we have the following algorithm.

Code 2.2: Selection (T,i)

```
1 Selection (T,i) (assumption : all entries are different)
2 pivot = random entry of T = T(j) for random j in (1,2,...,n)
3 T_{low} = entries < pivot
4 T_{hight} = entries > pivot
5 if |T_{low}| = i-1 then Selection (T,i) := pivot
6 if |T_{low}| < i-1 then Selection (T,i) := Selection (T_{hight}, i↩
    -|T_{low}|-1)
7 if |T_{low}| > i-1 then Selection (T,i) := Selection (T_{low}, i)
```

(Worst-case) expected time= t_n

$t_n \leq \mathbb{E}[t_{\max(|T_{low}|,|T_{high}|)}] + an$ where cost of finding T_{low}, T_{high} and $|T_{low}|$ is $\leq an$

$$= \sum_{k=0}^{n-1} P[|T_{low}| = k] t_{\max(k,n-k-1)} + \Theta(n)$$

$$= 2 \sum_{k=\lfloor \frac{n-1}{2} \rfloor}^{n-1} \frac{1}{n} t_k + an$$

The Master theorem is not directly applicable here since the coefficient , thus we need to tailor the proof. The simplest way here is to guess the solution and check that it is correct using induction.

Theorem 2.3. $t_n = \Theta(n)$

Proof. Assume $t_n \leq cn$ for some $c > 0$ (to be chosen later) and for all sufficiently large n. Proof by induction: assume it is true for $k \leq n-1$: $t_k \leq ck$.
We must show that $t_n \leq cn$:

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

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

2.4. Deterministic Select

In the previous part, we described the Random Selection algorithm which can find the i^{th} smallest entry of an array T in $\Theta(n)$ (worst-case expected time). This algorithm was based on a modified version of Quick Sort.

The question asked is : Can we find the i^{th} smallest entry with a deterministic algorithm in $\mathcal{O}(n)$ time?

To answer it, we must find a good pivot deterministically. The idea here is to do so in two steps, first by splitting the array in groups of five, taking the median of each, and computing the median of these medians using Deterministic Selection recursively.

This lead us to the following Deterministic Selection algorithm (Code 2.3).

Code 2.3: Pseudo-code of Determinisic Selection algorithm

```
1  DeterministicSelect(T,i) :
2      if (|T| == 1)
3          return T
4
5      Split T into ⌈n/5⌉ arrays of maxsize <= 5
6          call these arrays G_1, G_2, ..., G_⌈n/5⌉
7
8      foreach array G_i :
9          find median m_i
10         % ⌈n/5⌉ medians = Θ(⌈n/5⌉)
11     medians = {m_1, ..., m_⌈n/5⌉}
12
13     Pivot = DeterministicSelect(medians, 1/2⌈n/5⌉)
14         % find the median of the medians computed
15
16     Tlow   = {x ∈ T : x < pivot}
17     Thight = {x ∈ T : x > pivot}
18
19     if (i <= |Tlow|)
20         return DeterministicSelect(Tlow,i)
21     if (i == |Tlow| + 1)
22         return pivot
23     if (i > |Tlow| + 1)
24         return DeterministicSelect(Thigh,i-|Tlow|-1)
```

Is this a good pivot?

As our pivot is the median of the medians, we can say for sure that for 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 $3 \times \frac{1}{2} \times \lceil \frac{n}{5} \rceil \approx \frac{3n}{10}$ values of the array.

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. 2.1) can easily illustrate this reasoning.

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

$$\begin{aligned} t_n &= \underbrace{\Theta(n)}_{\text{split \& median finding}} + \underbrace{t_{\lfloor n/5 \rfloor}}_{\text{median of medians}} + \underbrace{\Theta(n)}_{\text{split high/low}} + \underbrace{\max\{t_{|T_{low}|}, 1, t_{|T_{high}|}\}}_{\text{'if' cases}} \\ &\leq t_{n/5} + t_{7n/10} + \underbrace{\Theta(n)}_{f(n)} \end{aligned} \tag{2.3}$$

Theorem 2.4. $t_n = \mathcal{O}(n)$

Proof. Let us proceed by induction. We want to prove that $t_n \leq cn$ for some $c > 0$ and for n sufficiently large. For $n = 1$, the problem is trivial (since i can only be 1) and $t_1 = 1$.

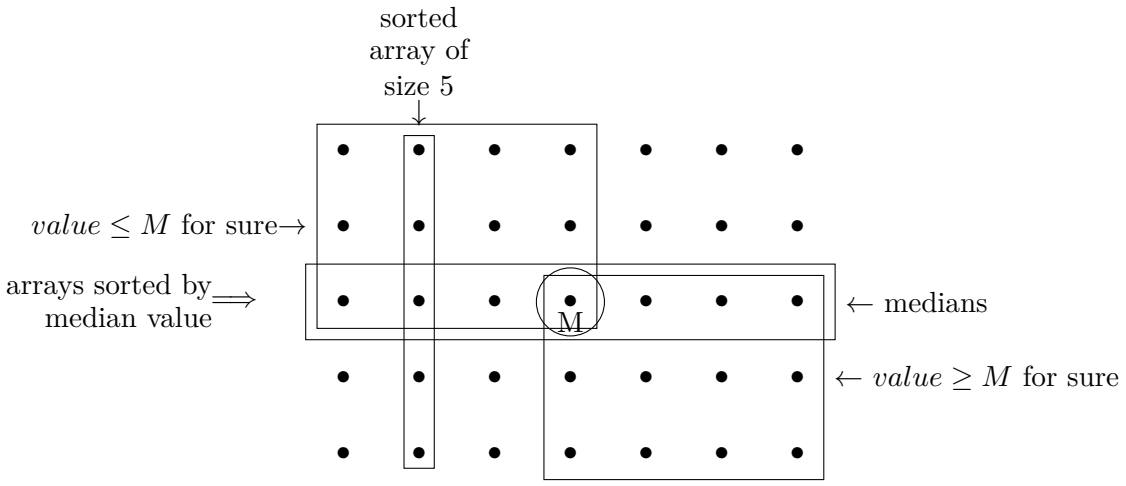


Figure 2.1.: Graphical view of the selection of the pivot (point M) in the `DeterministiSelect(T,i)`

Let us assume that the inequality holds for $t_1, \ldots, t_{n-1}$. Let us prove that it also holds for t_n.

Using equation (2.3), and knowing that the term $f(n) \in \Theta(n)$ can be bounded by $f(n) \leq an$ (with $a > 0$) for n sufficiently large, we have:

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

□

3. Dynamic Programming

3.1. Computing binomial coefficients

Given integers $0 \leq k \leq n$, we seek to compute the number $\binom{n}{k}$ of subsets of size k of a set of size n. The number $\binom{n}{k}$, read "n choose k", is called a *binomial coefficient*. The simplest way to compute $\binom{n}{k}$ is to use the explicit formula.

Theorem 3.1 (Explicit formula for the binomial coefficients)**.** For all integers $0 \leq k \leq n$,

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}.$$

Proof. Let us construct a k-tuple from the n elements of the set. We have n possibilities for the choice of the first element, $n-1$ for the second, and so on until the k^{th} element for which we have $n-k+1$ possibilities. By the rule of product, we can construct $n(n-1)\dots(n-k+1)$ distinct k-tuples.

Now observe that a set of size k defines $k!$ distinct k-tuples obtained by performing all the possible permutations of the set elements. (It is clear by induction. Let us show that if the set $\{a_1, \dots, a_k\}$ can be ordered into $k!$ k-tuples, then $\{a_1, \dots, a_k, a_{k+1}\}$ defines $(k+1)!$ ones. In any $(k+1)$-tuple, a_{k+1} can hold $k+1$ positions and for each of them, there exist $k!$ possible positions for the other elements by inductive hypothesis.) Consequently, the number of subsets of size k is the number of k-tuples divided by $k!$, which gives the above formula. □

3.1.1. Divide and conquer

The explicit formula is not computationally efficient unless k is close to 1 or n. So we are looking for another way to compute $\binom{n}{k}$. One possibility is Pascal's rule, a recursive formula.

Theorem 3.2 (Pascal's rule)**.** For all integers $0 < k < n$,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Proof. Observe that any subset of size k of $\{a_1, \dots, a_n\}$ either contains a_n, either does not. The number of subsets of size k that do not contain a_n is $\binom{n-1}{k}$ while the number of subsets of size k that contain a_n is $\binom{n-1}{k-1}$. The formula follows from the rule of sum. □

This formula, used with $\binom{n}{0} = \binom{n}{n} = 1$, allows computation of $\binom{n}{k}$. A naive implementation is the direct translation into a recursive algorithm based as below.

Pascal
in: $n, k \in \mathbb{N}$ **out:** $\binom{n}{k}$

if $k > n$
  return 0
else if $k = 0$ or $k = n$
  return 1
else
  return Pascal$(n-1, k)$ + Pascal$(n-1, k-1)$
end

The time complexity of this algorithm is given by the following recurrence equation:

$$t_{n,k} = t_{n-1,k} + t_{n-1,k-1} + \Theta(1)$$

where $\Theta(1)$ represents the cost of an addition (assumed to be independent of the size of numbers to add). The initial condition is $t_{n,0} = t_{n,n} = \Theta(1)$. A solution for this is in $\Theta(\binom{n}{k})$. We see that the recursive approach is in fact equivalent to a complete enumeration of all subsets of k elements among n. This is clearly inefficient.

The space complexity (memory used) is $\Theta(n)$ (height of the computation tree: longest chain of embedded computations when resolving the recursive calls).

3.1.2. Dynamic programming: bottom-up approach

The divide-and-conquer algorithm above computes many times the same subproblems. We can prevent this issue by storing intermediate results in an array. So we construct the following table, called "Pascal's triangle", column by column using Pascal's formula and starting from the initialization below.

	0	1	2	3	$\cdots$	n
0	1	1	1	1	$\cdots$	1
1		1				
2			1			
3				1		
$\vdots$					$\ddots$	
k						1

An efficient implementation of this algorithm uses only one array of size $k + 1$. (We just need to store the last computed column, not the whole table.) The spatial complexity is thus in $\Theta(k)$. Each entry is obtained by summing two known numbers which is performed in $\Theta(1)$. The time complexity to fill the whole table, i.e. n columns of size k, is therefore $\Theta(nk)$ (in fact $\Theta((n-k)k)$ in a more refined count).

Compared to the naive divide-and-conquer approach, this algorithm has a far better time complexity (linear versus exponential, for a given value of k).

This last algorithm uses the "bottom-up" approach of dynamic programming: it solves subproblems from the smallest to the largest.

3.1.3. Dynamic programming: top-down approach

The other approach is the so-called "top-down". This consists in applying the divide-and-conquer strategy while storing subproblems solutions in an array sometimes called *memory function*. Each time we meet a problem, we first check whether it has already been solved. Should it be the case, we go on; otherwise, we recursively solve it. This technique is called *memoization*.

In this case, this approach would take the following form.

in: $n, k \in \mathbb{N}$
out: $\binom{n}{k}$
$T(n,k) := -1$
if $k = 0$ or $k = n$
$\quad \binom{n}{k} = 1$
else if $T(n,k) \neq -1$
$\quad \binom{n}{k} = T(n,k)$
else
$\quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$
$\quad T(n,k) := \binom{n}{k}$
end

The time complexity is again $\Theta((n-k)k)$. However the space complexity is $\Theta((n-k)k)$ since we work in the whole table. So this approach is not efficient in this case. In fact memoization works well when we do not know in advance what small instances will have to be treated because it then computes exactly what is needed.

3.1.4. Conclusion

Like the divide-and-conquer approach, dynamic programming solves a problem by combining solutions to subproblems. Dynamic programming differs from divide-and-conquer by storing the solutions of subproblems: A subproblem appearing many times is solved only once.

3.2. Chained matrix products

We are interested in the product of n matrices. One knows that the matrix product is an associative, albeit not commutative, operation. We can therefore wonder how to parenthesise the product (and thus decide in which order to perform the products) to obtain the fastest computation.

The first step is evaluating the cost of the product of a $m \times n$ matrix A with a $n \times p$ matrix B, using the definition i.e. without using special algorithms such as Strassen's one. The result AB is a $m \times p$ matrix whose entries are the sum of n terms, each of them being the product of two scalars. By cost we mean the number of elementary operations which are here scalar multiplication and addition. Computing each of the mp entries of AB requires n scalar products and $n-1$ scalar additions. Hence the cost of computing AB is $\Theta(mnp)$.

Now let us convince ourself on a simple example that the order in which we compute the matrix products effectively impacts the computation efficiency. Let us consider 3 matrices

M_1, M_2 and M_3 of size $a \times b$, $b \times c$ and $c \times d$ respectively. The product $M_1M_2M_3$ can be computed in 2 distinct ways leading to different costs. Indeed, computing $(M_1M_2)M_3$ costs $abc + acd = ac(b + d)$ while computing $M_1(M_2M_3)$ costs $bcd + abd = bd(a + c)$. The difference is high when e.g. $a = c \gg b = d = 1$.

Let us now precisely state our problem: given $n \geq 2$ compatible matrices, find the order in which performing products leading to the smallest cost.

3.2.1. Brute force

A first idea is to compute the cost of each possible grouping. This is the "brute force" method. To evaluate this method cost we need to know how many ways to bracket a n-factor product are possible. The number of possible groupings for computing the product of n matrices is the $(n-1)^{\text{th}}$ *Catalan*⟨1⟩ *number*, denoted C_{n-1}. It is also the number of full binary trees (every node has zero to two children) with n leaves. It is easy to compute by enumeration that $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$ but the following terms rapidly become tedious to count. The following result offers a recursive characterization.

Theorem 3.3. For every integer $n \geq 0$,

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

Proof. Let $n \geq 0$ be an integer. Any grouping of $n + 2$ factors has the form

$$M_1 \dots M_{n+2} = (M_1 \dots M_{k+1})(M_{k+2} \dots M_{n+2})$$

for some $k \in \{0, \dots, n\}$, where the first $k + 1$ terms are further parenthesised (in C_k possible ways) and so are the $n - k + 1$ factors (in C_{n-k} possible ways). By the rule of product, there are thus C_nC_{n-k} ways for grouping the product $M_1 \dots M_{n+2}$. As it is true for every $k \in \{0, 1, \dots, n\}$, we get the desired relation by the rule of sum. □

This recurrence equation together with the initial condition $C_0 = 1$ can be solved for instance by the generating function method as we will see in the last section of this chapter. As we result we can show

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$$

which proves that the brute force approach is inefficient.

3.2.2. Divide and conquer

Let us see if a recursive strategy could be more efficient. The following algorithm is based on the recurrence reasoning used in the proof of Theorem 3.3.

⟨1⟩ Eugène Catalan (1814-1894) was a Belgian mathematician.

Cost(i,j)
in: $M_i, \ldots, M_j$ with M_k of size $d_{k-1} \times d_k$ for all $k \in \{i, \ldots, j\}$ **out:** cost of computing $M_i \ldots M_j$ in the optimal way

if $i = j$
  Cost$(i,j) = 0$
else
  Cost$(i,j) = \min_{i \leq k \leq j-1} \left[\text{Cost}(i,k) + \text{Cost}(k+1,j) + d_{i-1}d_k d_j\right]$
end

This algorithm is nothing else but the recursive version of the brute force algorithm seen in the previous subsection. In order to find the optimal cost the algorithm will try all the possible groupings and the time to do so is of the order $\text{Cost}(1,n) = C_n$. Indeed, every leaf of the recursion tree of $\text{Cost}(1,n)$ corresponds to one order for the product $M_1 \ldots M_n$. This means that the number of leaves is equal to C_n.

In conclusion, this approach is no better than the brute force. However, we observe that this algorithm performs the same computations many times. We then wonder if we could improve it by storing intermediate results in an array.

3.2.3. Dynamic programming

Within this new approach, $\text{Cost}(i,j)$ is a table which is filled by the main diagonal ($i = j$) then the immediate upper diagonal ($j = i + 1$) then all the upper diagonals one by one, using the equation given in the preceding algorithm. The time to compute $\text{Cost}(i,j)$ is $\Theta(j - i)$ and thus the time to fill in the table is

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(j-i) = \sum_{i=1}^{n}\sum_{k=0}^{n-i}\Theta(k) = \sum_{i=1}^{n}\Theta((n-i)^2) = \sum_{k=0}^{n-1}\Theta(k^2) = \Theta(n^3).$$

This is clearly 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 memoization) is a "top-down" approach.

3.3. Using dynamic programming to solve optimization problems

3.3.1. Principles

Dynamic programming typically applies to optimization problems when the two following conditions are satisfied.

Principle of Optimality For any optimal structure (like a list, a tree, etc.) answering the problem for an instance, the substructures (sublists, etc.) are optimal for the subinstances (smaller instances).

Subproblems are overlapping Divide and conquer is wasteful because it solves many times the same subinstance.

3.3.2. Examples

Many algorithms used in a broad range of domains use dynamic programming. Here are a few examples.

3.3.2.1. Shortest path in a graph

Observe that any 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, although they are a bit more clever (see next chapter on Greedy Algorithms).

3.3.2.2. Knapsack problem

The problem here is to fill in a bag of volume V with objects among $\{1, 2, \ldots, n\}$, each object i having a volume v_i and a value c_i. Assume all volumes $V, v1, \ldots, v_n$ are integers. The problem can be written as follows:

$$\max_{\text{Bag}\subseteq\{1,2,\ldots,n\}} \sum_{i\in\text{Bag}} c_i \qquad \text{s.t.} \qquad \sum_{i\in\text{Bag}} v_i \leq V.$$

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

Defining $\text{Value}(\{1, \ldots, k\}, v)$ as the maximal value for the set of objects $\{1, \ldots, k\}$ (for $1 \leq k \leq n$) and a total bag volume v, we can write

$$\text{Value}(\{1,\ldots,k\},v) = \max\big(\underbrace{\text{Value}(\{1,\ldots,k-1\},v-v_k)+c_k}_{\text{take the object } k};\underbrace{\text{Value}(\{1,\ldots,k-1\},v)}_{\text{do not take object } k}\big).$$

Since we are interested in

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

we need to compute $\text{Value}(\{1, \ldots, n\}, V)$. In order to compute $\text{Value}(\{1, \ldots, n\}, V)$ we may need to compute the table of entries $\text{Value}(\{1, \ldots, k\}, v)$ for all $1 \leq k \leq n$ and $0 \leq v \leq V$. Also, we need $\mathcal{O}(1)$ operation to compute one entry. We can thus deduce that the complexity of the algorithm is in $\mathcal{O}(nV)$.

As many entries may never be needed to compute the final value $\text{Value}(\{1, \ldots, n\}, V)$, a top-down approach (recursive function with memoization) may be advisable here.

3.3.2.3. Longest common subsequence

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

$$X = a\ b\ c\ b\ d\ a\ b, \qquad\qquad Y = b\ d\ c\ a\ d\ a.$$

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

$$X = a\ \underline{b\ c}\ b\ \underline{d\ a}\ b, \qquad\qquad Y = \underline{b}\ d\ \underline{c}\ a\ \underline{d\ a}.$$

Let us take the following prefix:

$$X_i = X_1 \dots X_i \text{ of } X_n = X_1 \dots X_n, \qquad Y_i = Y_1 \dots Y_i \text{ of } Y_n = Y_1 \dots Y_n$$

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 $\Theta(n^2)$.

3.3.2.4. Optimal binary search tree

A dictionary is an abstract data structure where entries are composed of a key and satellite data. The keys are totally ordered. One can SEARCH for a key (with the satellite data as output, or a message that the key is not in the dictionary), or INSERT a new entry, or DELETE an existing one.

One implementation is a binary search tree, structured as follows. Every node contains a key k, has possibly a left child and possibly a right child. The keys of left child and its descendants are all $\leq k$, and the keys of the right child and its descendants $\geq k$. DELETE, INSERT and SEARCH (by dichotomy) therefore take a worst case time in Θ(height of the tree). It is therefore important that the tree remains balanced, i.e. the height is logarithmic in the number of nodes. A randomly constructed tree verifies this with high probabilities.

As an aside, note that specific techniques allow to keep balanced trees, such as red-black trees, where every node is red or black, the root and leaves are black, as well as children of a red node. Moreover, every path from root to leaves contains the same number of black nodes. This ensures balancedness. The difficulty is to maintain the properties at logarithmic cost in the INSERT and DELETE operations, which is fortunately possible. Note that EXTRACT-MIN is also available for the same cost, which means that red-black trees can also be used to implement priority queues. Other structures for keeping balanced trees (binary or not) are B-trees or van Emde Boas trees.

In this section, we assume we have a fixed dictionary, with only the SEARCH operation, and a probability of query p_i associated to every entry i in the dictionary. For simplicity of the exposition, we assume that all entries $1, 2, \dots, n$ are ordered as their keys: entry 1 has the smallest key, etc. and that only existing keys are queried, although this is not essential. The problem is now to build the binary search tree with the minimum expect cost of query $\sum_i p_i \mathrm{depth}(k_i)$.

The number of possibly trees with n nodes is again the Catalan number C_n, thus exponential in n: we can prove it by showing it satisfies the recurrence and the initial condition satisfied by Catalan's numbers. As an aside, note that C_n is also the number of proper binary trees (i.e., binary tree where every node has exactly zero or two children) with $n+1$ leaves, since these are in bijection with all possible bracketings of a product of $n+1$ matrices.

A recursive expression for the cost would go through scanning every node r as a possible root, then optimally arranging the entries $< r$ into a left-tree and the entries $> r$ as the right-tree. We name

$$\text{Cost}([i,j]) = \min \sum_{i \leq \ell \leq j} p_\ell \text{depth}(\ell)$$

where the minimum is taken over all possible binary search trees for entries $i, \ldots, j$. The cost breaks down recursively as

$$\text{Cost}([i,j]) = \min_{i \leq r \leq j} (p_r.1 + \sum_{i \leq \ell \leq r-1} p_\ell.1 + \text{Cost}([i,r-1]) + \sum_{r+1 \leq \ell \leq j} p_\ell.1 + \text{Cost}([r+1,j]),$$

in other words:

$$\text{Cost}([i,j]) = \sum_{i \leq \ell \leq j} p_\ell + \min_{i \leq r \leq j} \text{Cost}([i,r-1]) + \text{Cost}([r+1,j])$$

whenever $i \neq j$, with trivial $\text{Cost}([i,i]) = p_i$ as base case.

Similarly the example of optimal product bracketing, this can be resolved with dynamic programming on a table registering all $\text{Cost}([i,j])$ with total time $\Theta(n^3)$.

3.3.3. Computing the optimal solution vs the optimal cost

Note that for all these problems, we usually 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.

3.4. Generating functions

Many discrete mathematics problems can be solved using analytical methods i.e. continuum mathematics. One of the most famous examples is the Prime Number Theorem.

The generating functions method, which is a powerful tool for solving recurrence equations, is another one. The idea is to associate a power series to a sequence $(a_n)_{n \in \mathbb{N}}$:

$$f(z) = \sum_{n=0}^{\infty} a_n z^n$$

where $z \in \mathbb{C}$. If the sequence $(a_n)_{n \in \mathbb{N}}$ satisfies some recurrence equation one can sometimes transform this recurrence into a functional equation of the form $F(f(z), z) = 0$, F being a known function. If one can get an expression for $f(z)$ as a function of z and expand this expression in series one gets the terms of the sequence by identifying coefficients in both power series.

We shall illustrate this technique by computing Fibonacci and Catalan numbers, the former being solution of a linear recurrence equation while the second obey a nonlinear recurrence.

3.4.1. Fibonacci numbers

The first terms of the sequence are $F_0 = 0$ and $F_1 = 1$. The following terms are given by the following recurrence equation:

$$F_n = F_{n-1} + F_{n-2}.$$

It is a linear homogeneous recurrence equation which can be easily solved by the characteristic equation method.

Characteristic equation method One postulates a solution of the form x^n for the Fibonacci recurrence (or a linear combination of such solutions, since the recurrence is linear), which leads to solve the so-called characteristic equation:

$$x^2 - x - 1 = 0$$

which admits $\varphi = \frac{1+\sqrt{5}}{2}$ and $\bar{\varphi} = \frac{1-\sqrt{5}}{2}$. The solution of the recurrence equation has therefore the following form:

$$F_n = c_1\varphi^n + c_2\bar{\varphi}^n$$

for some $c_1, c_2 \in \mathbb{R}$. Taking initial conditions into account yields

$$F_n = \frac{\varphi^n - \bar{\varphi}^n}{\sqrt{5}} \approx \frac{\varphi^n}{\sqrt{5}}$$

since $|\bar{\varphi}| < 1$.

Generating functions method One defines the power series

$$f(z) = \sum_{k\geq 0} F_k z^k.$$

From the recurrence equation $F_{k+2} = F_{k+1} + F_k$, we deduce that

$$\begin{aligned}
\sum_{k\geq 0} F_{k+2} z^{k+2} &= \sum_{k\geq 0} F_{k+1} z^{k+2} + \sum_{k\geq 0} F_k z^{k+2} \\
f(z) - F_0 - F_1 z &= z(f(z) - F_0) + z^2 f(z) \\
f(z) - z &= z f(z) + z^2 f(z).
\end{aligned}$$

From the last equation we deduce that

$$f(z) = -\frac{z}{z^2 + z - 1} = -\frac{z}{(z+\varphi)(z+\bar{\varphi})} \overset{\langle 2\rangle}{=} \frac{1}{\sqrt{5}}\left(\frac{\bar{\varphi}}{z+\bar{\varphi}} - \frac{\varphi}{z+\varphi}\right) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\frac{z}{-\bar{\varphi}}} - \frac{1}{1-\frac{z}{-\varphi}}\right).$$

$\langle 2\rangle$ Partial fraction decomposition:

$$-\frac{z}{(z+\varphi)(z+\bar{\varphi})} = \frac{A}{z+\varphi} + \frac{B}{z+\bar{\varphi}} = \frac{(A+B)z + A\bar{\varphi} + B\varphi}{(z+\varphi)(z+\bar{\varphi})} \iff \begin{pmatrix} 1 & 1 \\ \bar{\varphi} & \varphi \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$
$$\iff \begin{pmatrix} A \\ B \end{pmatrix} = \frac{1}{\sqrt{5}}\begin{pmatrix} -\varphi \\ \bar{\varphi} \end{pmatrix}.$$

Developping in series the right hand side[3],

$$\begin{aligned}
\frac{1}{1-\frac{z}{-\bar{\varphi}}} - \frac{1}{1-\frac{z}{-\varphi}} &= \sum_{k\geq 0}\left(\frac{z}{-\bar{\varphi}}\right)^k - \sum_{k\geq 0}\left(\frac{z}{-\varphi}\right)^k \\
&= \sum_{k\geq 0}\left(\frac{1}{(-\bar{\varphi})^k} - \frac{1}{(-\varphi)^k}\right) z^k \\
&= \sum_{k\geq 0}\left(\frac{(-\varphi)^k - (-\bar{\varphi})^k}{(\bar{\varphi}\varphi)^k}\right) z^k \\
&= \sum_{k\geq 0}\left(\frac{(-\varphi)^k - (-\bar{\varphi})^k}{(-1)^k}\right) z^k \\
&= \sum_{k\geq 0}\left(\varphi^k - \bar{\varphi}^k\right) z^k,
\end{aligned}$$

we conclude that

$$F_n = \frac{\varphi^n - \bar{\varphi}^n}{\sqrt{5}}.$$

Conclusion The advantage of the second method (although minor in the case of linear recurrences) is that we need not guess in advance the form of the solution, which emerges naturally from the solution. It also applies to some nonlinear equations such as the Catalan's, where the full power of the method is revealed.

3.4.2. Catalan numbers

The first terms are $C_0 = 1$ and $C_1 = 1$. The following terms are given by the nonlinear recurrence equation

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

Let us define the power series

$$f(z) = \sum_{n\geq 0} C_n z^n = 1 + z\sum_{n\geq 0} C_{n+1} z^n.$$

From the recurrence equation we deduce successively that

$$\sum_{n\geq 0} C_{n+1} z^n = \sum_{n\geq 0}\left(\sum_{k=0}^{n} C_k C_{n-k}\right) z^n = \sum_{n\geq 0}\sum_{k=0}^{n} (C_k z^k)(C_{n-k} z^{n-k}).$$

[3] Geometric series: if $z \in \mathbb{C}$ and $|z| < 1$ then

$$\frac{1}{1-z} = \sum_{n=0}^{\infty} z^n.$$

By the Cauchy product formula (see next subsection) we have[4]

$$\sum_{n\geq 0}\sum_{k=0}^{n}(C_k z^k)(C_{n-k}z^{n-k}) = \left(\sum_{n\geq 0} C_n z^n\right)\left(\sum_{k\geq 0} C_k z^k\right) = f(z)^2.$$

So $f(z)$ satisfies the quadratic equation

$$f(z) = 1 + zf(z)^2$$

and so

$$f(z) = \frac{1-\sqrt{1-4z}}{2z}, \tag{3.1}$$

rejecting positive sign since $f(0) = 1$. By the generalized binomial formula (see next subsection)

$$\sqrt{1-4z} = \sum_{n=0}^{\infty}\binom{\frac{1}{2}}{n}(-4z)^n = 1 + \sum_{n=1}^{\infty}\frac{\frac{1}{2}(-\frac{1}{2})\dots(\frac{3}{2}-n)}{n!}(-1)^n 2^{2n} z^n$$

and so

$$f(z) = -\frac{1}{2z}\sum_{n=1}^{\infty}\frac{\frac{1}{2}(-\frac{1}{2})\dots(\frac{3}{2}-n)}{n!}(-1)^n 2^{2n} z^n$$

from which we deduce that for every $n \in \mathbb{N}_0$,

$$\begin{aligned}
C_{n-1} &= -\frac{1}{2}\frac{\frac{1}{2}(-\frac{1}{2})\dots(\frac{3}{2}-n)}{n!}(-1)^n 2^{2n}\\
&= \frac{1}{2}\frac{(-\frac{1}{2})(-\frac{1}{2})\dots(\frac{3}{2}-n)}{n!}(-1)^n 2^{2n}\\
&= \frac{1}{2}\frac{\frac{1}{2}\frac{1}{2}\dots(n-\frac{3}{2})}{n!}2^{2n}\\
&= \frac{1}{2}\frac{1\cdot 1\dots(2n-3)}{n!}2^n\\
&= \frac{1}{2}\frac{(2n-2)!}{n!(2\dots(2n-2))}2^n\\
&= \frac{1}{2}\frac{(2n-2)!}{n!2^{n-1}(n-1)!}2^n\\
&= \frac{(2n-2)!}{n!(n-1)!}.
\end{aligned}$$

In conclusion, for every $n \in \mathbb{N}_0$,

$$C_{n-1} = \frac{(2n-2)!}{n!(n-1)!} = \frac{1}{n}\binom{2n-2}{n-1}.$$

or said otherwise

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

[4] As we shall see later, we do not have to be so careful about convergence of series. We could just compute algebraically term-by-term that

$$f(z)^2 = (C_1 z + C_2 z^2 + \dots)(C_1 z + C_2 z^2 + \dots) = \underbrace{C_1 C_1}_{C_2} z^2 + \underbrace{(C_1 C_2 + C_2 C_1)}_{C_3} z^3 + \dots = f(z) - z.$$

One can get an asymptotic expression using Stirling's approximation:

$$C_n \approx \frac{\sqrt{4\pi n}(2\frac{n}{e})^{2n}}{(n+1)(2\pi n)(\frac{n}{e})^{2n}} = \frac{4^n}{(n+1)\sqrt{\pi}\sqrt{n}} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right).$$

3.4.3. Some useful results

This subsection collects some useful results used in the previous calculation.

Theorem 3.4. Stirling approximation:

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(\frac{n}{\mathrm{e}})^n} = 1.$$

A coarser but easy to remember approximation is $n! \approx (n/e)^n$. A quick argument to show it is $\ln n! = \ln 1 + \ln 2 + \ldots + \ln n \approx \int_1^n \ln x \approx n\ln n - n$ and then exponentiate. A refinement of this proves the full theorem.

Definition 3.1. For all $r \in \mathbb{C}$ and $n \in \mathbb{N}_0$, we define

$$\binom{r}{n} = \frac{r(r-1)\ldots(r-n+1)}{n!} = \frac{1}{n!}\prod_{i=0}^{n-1}(r-i)$$

and $\binom{r}{0} = 1$.

Theorem 3.5. For all complex z such that $|z| < 1$ and an arbitrary complex r,

$$\sum_{n=0}^{\infty}\binom{r}{n}z^n = (1+z)^r.$$

A sketch of proof goes as follows: we notice that both sides satisfy the (complex) differential equation $(1+z)f'(z) = rf(z)$ with initial condition $f(z) = 0$.

Equivalently:

Theorem 3.6 (Generalized binomial formula, Newton 1665)**.** For all $r, x, y \in \mathbb{C}$ such that $|x| > |y|$,

$$(x+y)^r = \sum_{n=0}^{\infty}\binom{r}{n}x^{r-n}y^n.$$

A useful criterion for discrete convolutions, which arise naturally when multiplying power series:

Theorem 3.7 (Cauchy product formula)**.** Let $(u_n)_{n\in\mathbb{N}}$ and $(v_n)_{n\in\mathbb{N}}$ be two complex sequences. If the series $\sum_{n=0}^{\infty} u_n$ and $\sum_{n=0}^{\infty} v_n$ converge absolutely, then the series

$\sum_{n=0}^{\infty}\sum_{k=0}^{n} u_k v_{n-k}$ converges to $\left(\sum_{n=0}^{\infty} u_n\right)\left(\sum_{n=0}^{\infty} v_n\right)$.

3.4.4. Link with Taylor series and holomorphic functions

A generating function can be seen as a purely algebraic object, i.e. a formal power series obeying the usual rules of addition and multiplication for (infinite-degree) polynomials—not caring about the 'numerical value' of the variable z. Thus using Cauchy's product theorem above was useless in that view.

One may however also consider generating functions as analytical objects, i.e. functions $\mathbb{C} \to \mathbb{C}$ in order to squeeze out more information from the coefficients. In some cases indeed it is not possible to obtain an exact formula for the coefficients a_k of the generating function, in which case we would like at least to estimate how a_k grows asymptotically for large k. This is possible through the study of the radius of convergence of the generating function.

Remember that the radius of convergence of a power series $f(z) = a_0 + a_1 z + a_2 z^2 + \dots$ is the number $r \in [0, +\infty]$ such that $a_0 + a_1 z + a_2 z^2 + \dots$ converges towards a finite complex number for any $|z| < r$ and diverges (i.e. the limit does not exist or goes to infinity) for any $|z| > r$. Such number always exists. It can be computed from Cauchy's criterion :

$$1/r = \limsup_{n\infty} |a_n|^{1/n}$$

(in other words it is the infimum r such that $|a_n| \in \mathcal{O}(r^{-n})$).

Another criterion is the ratio criterion:

$$r = \lim_{n\to\infty} |a_n|/|a_{n+1}|$$

whenever that limit exists. Another characterisation stems from the scrutiny of $f(z)$. The radius of convergence r of the Taylor series $f(z) = a_0 + a_1 z + a_2 z^2 + \dots$ is the absolute value $|z_0|$ such that f is not differentiable (or 'holomorphic' in the complex analysis linguo) at z_0.

A simple, typical example is $f(z) = \frac{1}{1-az} = 1 + az + a^2 z^2 + a^3 z^3 + \dots$ where we can see that all characterisations coincide, with $r = 1/|a|$. A more powerful application is precisely the application to Catalan's numbers. We know that Catalan's numbers are encoded by (3.1). It is clear that f is differentiable at $z = 0$ (through L'Hospital's rule) et for any $z : |z| < 1/4$ but not at $z = 1/4$. Indeed the square root function is not differentiable when the argument is zero. Thus the radius of convergence is $r = 1/4$, and we deduce that C_n grows as 4^n, up to subexponential factors. We have spared the need to find an explicit expression for C_n.

In summary we have proved here the complexity of an algorithm (the brute force search for optimal matrix product) with the help of complex analysis!

This method is at the heart of de la Vallée Poussin and Hadamard's (independent) proofs of the Prime Number Theorem, stating that the nth prime number grows roughly as $n \ln n$ (the full statement offers a better estimate) based on detailed analysis of Riemann's

zeta function $\zeta(z) = \sum_{n \in \mathbb{N}_0} n^{-z}$, which can be seen (nontrivially) as a kind of generating function for prime numbers.

3.4.5. Link with the z-transform

Another use of analytic generating functions is in signal processing and control theory. A discrete-time temporal signal $a_0, a_1, \ldots, a_t, \ldots$ is represented through the so-called z-transform as $A(z) = a_0 + a_1 z^{-1} + \ldots + a_t z^{-t} + \ldots$. If $(a_t)_t$ is the response of a linear system to an impulse, then $A(z)$ is called the transfer function of the system. It is then of interest to check that the impulse response does not grow exponentially, which would be the mark of an unstable (explosive) behaviour; this is checked by locating the singularities of the transfer function $A(z)$. We see therefore that z-transform and generating functions are two names for the same tool, save for the arbitrary change of variables $z \to z^{-1}$.

Note that the z-transform is also the discrete-time equivalent of Laplace (or Fourier) transform. This makes the Laplace transform a continuous equivalent of generating functions. Indeed Laplace transforms can be used to solve linear and some nonlinear differential equations, the continuous-time equivalent of recurrence equations.

4. Greedy algorithms

Let us now introduce a new kind of algorithm: the greedy algorithms. In this part, we first 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.

4.1. Activity selection problem

We would like to book as many non-overlapping activities as possible for a seminar/lecture room. Each activity is characterised by two features : its starting time and its finishing time. Let us write the i-th activity as $a_i = [s_i \ \ f_i[$ (for $i = 1, \ldots, n$)with s_i and f_i its starting and finishing time, respectively. We define $f_0 = 0$ (start of the day). Let us also assume that activities are sorted such that $f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n$.

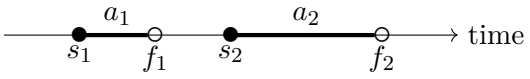


Figure 4.1.: Example of two disjoint activities

4.1.1. Dynamic programming solution

Let us check that a principle of optimality applies. Clearly, if a set of activities $\{a_{i_1}, \ldots, a_{i_k}\}$ (written in chronological order) is optimal for a certain time interval, then the subset of consecutive activities taking place in the subinterval $[f_{i_\ell} \ \ f_{i_{\ell'}}[$ is optimal for this subinterval, among all activities taking place in this subinterval. It is therefore natural to apply a Divide and Conquer algorithm or Dynamic Programming solution.

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

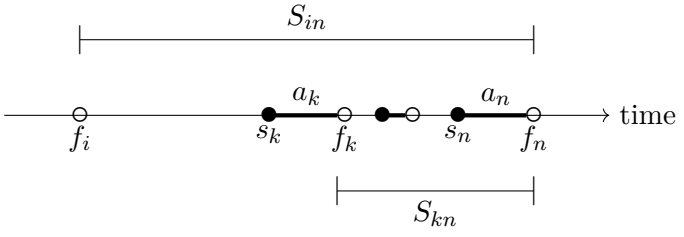


Figure 4.2.: Decomposition into subproblems

We can scan through all activities a_i as candidates for the first activity to start in the day. If a_i is the first activity then we can solve again the same from problem on the interval $[f_i \;\; f_n[$, with activities in S_{in}. Therefore c_{in}, defined as the maximum number of non-overlapping activities in $[f_i \;\; f_n[$, obeys

$$c_{in} = \begin{cases} 1 + \max_k(c_{kn}) \\ 0 \qquad\qquad \text{if } S_{in} = \emptyset \ . \end{cases}$$

This decomposition in subproblems gives us a dynamic programming algorithm to solve the activity selection problem. Filling the entire array takes $\mathcal{O}(n^2)$ operations. There are n entries that each take $\mathcal{O}(n)$ to be computed.

4.1.2. Greedy solution

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

Why is that so? If we have an optimal solution that starts with $a_k \in S_{ij}$ for $k > \ell$, we also have an optimal solution that starts with a_ℓ, as we can remove a_k and replace it with a_ℓ (which finishes earlier than a_k) to obtain a solution with no overlaps and of same quality (thus optimal as well). This argument can be repeated for each of the subproblems, therefore it is never a suboptimal choice to pick the activity in S_{in} with earliest finish time.

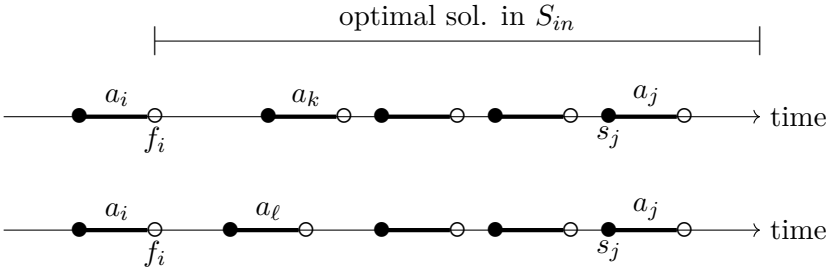


Figure 4.3.: Illustration of the greedy algorithm argument

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

Code 4.1: Pseudo-code of the greedy algorithm for activity selection problem

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

The time complexity of this algorithm is

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

In general, a "greedy" approach to solve an optimality problem refers to building a solution from $\emptyset$ by adding at each step the "best" piece, chosen in a short-sighted way, i.e. that brings the best immediate reward, without care for the future steps.

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

4.2. Other examples of greedy algorithms

Other well-known examples of greedy algorithms include:

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

4.3. Scheduling problem

Let us now consider another famous problem:

- n jobs, all of same (unit) duration
- t_i: deadline for the job i
- c_i: profit for job i

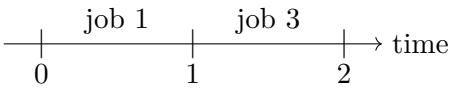
The profit c_i is earned from job i if and only if it is scheduled to finish before its deadline t_i. The aim is to maximize the total profit P_{Total}, which is the sum of the individual profits of the jobs finishing before their respective deadline.

4.3.1. Example

Let us take the following instance

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

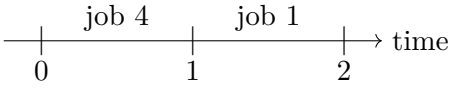
One possible scheduling of the jobs is the following



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.

4.3.2. Generalization: Greedy Algorithm

The greedy algorithm simply proceeds by repeatedly adding highest profit feasible job first:

Code 4.2: Pseudo-code of the greedy algorithm for scheduling problem

```
1 L = Empty list % will hold selected jobs
2 S = priority queue of jobs ordered by profit.
3 While S not Empty
4     a = highest profit job in S
5     If L+a is feasible then insert a in L
```

Let us see how it works on our example (in this case $T = 2$):

Code 4.3: Example of the greedy algorithm for scheduling problem

```
1  Select job with highest profit : job1
2      Try add job1 in L -> possible
3      $ L ={job1} : Profit of 50
4  Select job with 2nd highest profit : job4
5      Try add job4 in L -> possible
6      $ L ={job4, job1} : Profit of 80
7  Select job with 3rd highest profit : job3
8      Try add job3 in L -> impossible
9      $ Do not add it
10 Select with 4th highest profit : job2
11     Try add job2 in L -> impossible
12     $ Do not add it
13 STOP
```

4.3.3. The greedy algorithm is correct

We start with a criterion to check that a given set of jobs is optimal.

Theorem 4.1. A set of jobs with deadlines $t_1 \leq t_2 \leq \ldots \leq t_n$ is feasible, if and only if it is feasible in the order $1, 2, \ldots, n$.

Proof.

- ($\Rightarrow$) If a schedule where job i (finishing at time b) is placed after job j (finishing at time a), as in the picture, with $t_i \leq t_j$, then we can swap i and j, and it

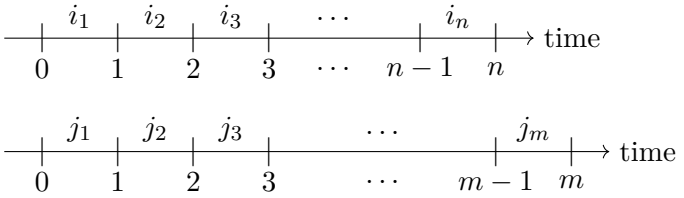
is still a feasible schedule. Indeed, since this schedule is feasible, we have $t_i \geq b$ and $t_j \geq a$. Furthermore, we have that $b \geq a$ and $t_j \geq t_i$. It follows that $t_i \geq b \geq a$ and $t_j \geq t_i \geq b$, which make the schedule where i and j are swapped feasible. We can repeat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

- ($\Leftarrow$) Trivial.

□

Theorem 4.2. 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. As they are both feasible, they can be both ordered at unit time slots at times 0, 1, etc., as in the picture.



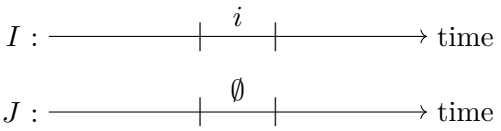
We now want to "align" I and J as much as possible to compare them: schedule same jobs at the same time. We modify I and J in order to have every job present in both solutions to be scheduled at the same time, while maitaining feasibility. We illustrate how to this in the following example.

Example 4.1. 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.

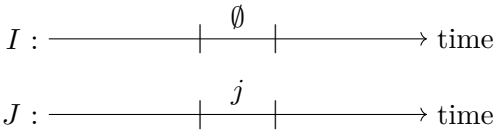
To that end, we will swap i_4 and i_8 in I. If i_8 is empty, then after the swap is performed, i_4 will be empty and i_8 will contain job k. If $i_8 = \tilde{k}$ was not empty, then after the swap has been performed, job $\tilde{k}$ is now scheduled on the 4-th time slot and job k on the 8-th. Moving 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 : 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 ! But if $k = i_7 = j_2$, then swap j_2 and j_7 in J, to keep feasibility.

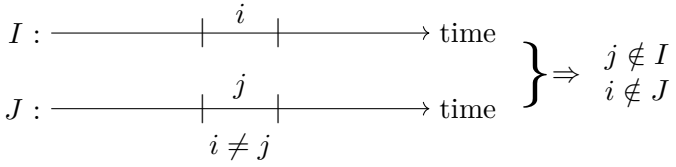
Repeat all swaps until all common jobs are aligned. Now, let us take a look at all the possible situations where the jobs are different in I and in J. The first situation has a job $i \in I$, with a corresponding empty slot in J.



This situation is impossible because since $i \notin J$, we can schedule it in the empty slot and improve the total profit of the sequence J strictly. A second situation is to have a job $j \in J$, with a corresponding empty slot in I.



Once again, this situation is impossible because since $j \notin I$, then adding a to I is feasible, thus the greedy algorithm would have chosen it. Finally, the last situation that could potentially occur is two different jobs $i \in I$ and $j \in J$ scheduled at the same time.



If $c_j > c_i$: then the greedy algorithm would have chosen j before i thus j would be in I.
If $c_i > c_j$: then J is strictly improved by replacing j with i, which impossible as J is optimal.
It follows that $c_i = c_j$, and consequently we have that $C_I = C_J$. □

Time complexity: from the feasibility criteria above, it is enough to maintain a table N with N_t the number of selected jobs of deadline $\leq t$, for $t = 1, 2 \ldots$. The feasibility criterion amounts to checking that $N_t \leq t$ for all t, which can be tested in $\mathcal{O}(n)$ for each candidate job.

$$\underbrace{\Theta(n \log n)}_{\text{Sort by profit}} + \underbrace{\Theta(n^2)}_{\text{Check feasibility } n \text{ times}}$$

In fact one can do better : $\Theta(n \log n)$ (accepted).

5. Random algorithms

We first introduce or recall a few classic problems in probability.

5.1. The secretary problem

This is also called the best choice problem, or the marriage problem.

In a first version of the problem, consider that 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 and fire the previous one.

5.1.1. How many persons will you hire on average?

Say you interview them

- in increasing order of quality $q_1 < q_2 < \ldots < q_n$ ($1, 2, \ldots, n$ 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 variables*. An indicator variable is a random variable with value 0 or 1. To each event we can associate an indicator variable, taking value 1 on the event and 0 otherwise. Let

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

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

We express X, the number of persons hired, as a sum of indicator variables:

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

5.1.2. Optimal strategy to hire the best candidate

This time, we can only hire 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). We look for a strategy with an observation phase of $k-1$ candidates we interview with no intent to hire them, followed by a hiring phase where we hire the first best-ever candidate. One can prove that the optimal strategy is necessarily of this form, which we accept here. We focus on finding the optimal k.

Code 5.1: Pseudo-code of the strategy for the Secretary Problem algorithm

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

5.1.3. What is the best k?

$$
\begin{aligned}
\text{Probability of success} &= \sum_{i=k}^{n} \mathrm{P}[i \text{ is hired and is the best overall}] \\
&= \sum_{i=k}^{n} [i \text{ is the best overall}] \times \mathrm{P}[i \text{ is hired} \mid i \text{ is the best}] \\
&= \sum_{i=k}^{n} [i \text{ is the best overall}] \\
&\qquad \times \mathrm{P}[\text{the second best in } 1, 2, \ldots, i-1 \text{ was among } 1, \ldots, k-1 \mid i \text{ is the best}] \\
&= \sum_{i=k}^{n} \frac{1}{n} \frac{k-1}{i-1} \\
&\approxeq \frac{k}{n} \int_{k-1}^{n-1} \frac{1}{x} \, \mathrm{d}x. \\
&= \frac{k}{n} (\ln n - \ln k) \\
&= \frac{\ln \frac{n}{k}}{\frac{n}{k}} \\
&= \frac{\ln y}{y} \quad \text{for } y = \frac{n}{k} \geq 1
\end{aligned}
$$

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, we have $P[i \text{ is hired } | i \text{ is the best}] = \frac{k-1}{i-1}$.

Then, the probability is zero for $y = 0$ and for $y = 1$. The probability is optimal for y

cancelling the derivative:

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

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

In summary one must interview 37% of the candidates in the observation phase, and then pick the first best candidate ever seen in the hiring phase. The probability to find the best candidate in this way is 37%.

5.2. 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.

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

So, if $n = \sqrt{2t}$, then $\mathbb{E}[X] \approxeq 1$, i.e. there exist a fair chance to observe a coincidence. If $t = 365$, $n = \sqrt{2 \times 365} = 25$. 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.

5.2.1. Application to hash functions and hash tables

The birthday paradox can be reformulated in the following way: imagine that I want to find ID numbers to refer to n people (instead of their full name), and I randomly choose these IDs independently and uniformly among $\{1, 2, \ldots, t\}$. Then the birthday paradox states that we must choose at least $t \gg n^2$ to avoid collisions with reasonable probability (two persons identified by the same ID).

Let us get back to dictionaries and their implementations. We see in a previous chapter that n elements can be stored in a tree with typically $\log n$ costs for the operations SEARCH, INSERT, DELETE. Let us look for another implementation. If we know that the keys used to access the entries of the dictionary are numbers between 1 and N, we can simply store them in a table with direct access indexed from 1 to N. The dictionary operations then become trivial. The problem is that if N is very large, much larger than the number n of keys actually encountered, then most cells of the table are empty and the waste of memory can be huge. The idea is therefore to create a so-called hash function h from $\{1, 2, \ldots, N\}$ to a smaller set of keys $\{1, 2, \ldots, t\}$, where each key x is represented with a new (shorter) identifier $h(x)$. Although h is necessarily non-injective, one may hope that on the n keys actually encountered while using the dictionary, h is indeed an injection. In this case we have the benefit of saving memory while retaining the simplicity and speed of direct access in a table. As the birthday paradox shows, one expects to have no or few collisions if $t \gg n^2$. That holds under the Simple Uniform Hashing assumption, i.e. under the assumptions that keys are hashed to any entry of the table with probability $1/t$, independently of the keys already placed in the table. Since h is deterministic, this is really an assumption on the random distribution of the inputs. Formally, it says that the sequence of keys $x_1, x_2, \ldots$ (eliminating repeated keys) that are being sent to the hash function h is a stochastic process such that $h(x_1), h(x_2), h(x_3), \ldots$ is an i.i.d. uniform process.

One however has to anticipate how to manage collisions if they occur. The simplest way is to fill a cell of the table, not with a single entry, but with a chained list of all entries hashed to the same cell. One can show that in general, the SEARCH, INSERT and DELETE are performed in expected time $\Theta(1 + n/t)$, under the Simple Uniform Hashing assumption.

Note that hash functions have applications beyond the use of hash tables. For instance they can be used to produce a 'fingerprint' or 'checksum' of a large object. For instance a large file maybe hashed to a relatively small hash value, so that if we suspect that the file has been corrupted (e.g. after sending over a noisy communication channel, or after a virus alert), we recompute the hash value of the file and verify that the hash value is still unchanged.

Examples of 'good' hash functions are given later in this chapter.

5.3. The coupon collector's problem

Assume there exist N different stickers to be collected, distributed randomly with food items to customers of a food store. How many items will you need to buy on average in order to collect at least one copy of each of the N stickers ?

Call t_i the number of items to buy before obtaining a new sticker, knowing that $i - 1$ different stickers have already been collected. Clearly $t_1 = 1$ for sure, while $t_2, t_3, \ldots$ are random variables. In fact with $i - 1$ stickers already collected, the probability of every sticker to be different from those is $\frac{n-i+1}{n}$. Therefore t_i is the time for the number of Bernoulli trials before obtaining a success, whose expected value is here $\frac{n}{n-i+1}$ trials. For instance $\mathbb{E}t_n = n$: it takes n trials to find the last sticker of the collection.

Therefore the total number of items to buy, or total number of trials is expected to be:

$$\begin{aligned}\mathbb{E}\sum_i t_i &= \sum_{i=1}^{n} \mathbb{E}t_i \\ &= \sum_{i=1}^{n} \frac{n}{n-i+1} \\ &= n\left(1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}\right) \\ &\approxeq n\ln n\end{aligned}$$

For example to collect 100 stickers one has to buy 460 items on average.

5.3.1. Application to random brute force

Suppose one has to check if an element with a specific property exists among N by scanning through all of them, for instance in a brute force solution of a combinatorial problem. A deterministic systematic search will take N trials in the worst case (that is, when the searched element does not exist), while a random, memoryless search will take an expected time of $N\ln N$ trials in virtue of the coupon collector's problem. It is a relatively low increment in time in exchange for the simplicity of the scheme. This can be relevant in a distributed context, when many agents look for solutions randomly independently of one another, without any central entity to organise the search. The 'price of anarchy' is a $\ln n$ factor.

5.4. Monte Carlo algorithms in numerical analysis: the example of π

5.4.1. Buffon's theorem for computing π

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

Theorem 5.1. (Buffon): If I throw randomly a needle of length ℓ on a floor with planks of width 2ℓ, the probability of crossing a crack (i.e. overlap two planks) is $\frac{1}{\pi}$ (see figure 5.1).

Proof. 1. The direct (and somewhat boring) proof using trigonometry and calculus: the needle falls with an angle with respect to the horizontal (perpendicular to the planks), uniformly distributed in the interval $[0,\pi]$ (in radians).

The interval $[0,\pi]$ can therefore be divided into N smaller intervals of the same size: $[0,\pi] = [\varphi_1,\varphi_2]\cup[\varphi_2,\varphi_3]\cup\ldots\cup[\varphi_{N-1},\varphi_N]$ where $\varphi_1 = 0$ and $\varphi_i = \varphi_{i-1} + d\varphi$. We

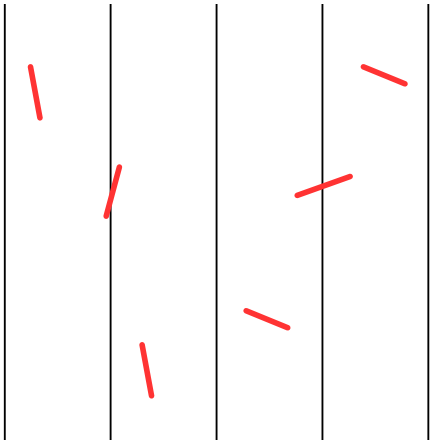


Figure 5.1.: Needles of length ℓ crossing cracks of width 2ℓ

have

$$\begin{aligned} P(\text{needle overlaps two planks}) &= \lim_{N\to\infty} \sum_{i=1}^{N} P(\text{overlap}|\theta \in [\varphi_i, \varphi_i + d\varphi])P(\theta \in [\varphi_i, \varphi_i + d\varphi]) \\ &= \lim_{d\varphi\to 0} \sum_{i=1}^{N} \frac{\ell|\cos(\varphi_i)|}{2\ell} \frac{d\varphi}{\pi} \\ &= \int_0^{\pi} \frac{|\cos(\varphi)|}{2\pi} d\varphi \\ &= \frac{1}{\pi}. \end{aligned}$$

2. Another, beautiful and powerful, proof is the following. We consider a much more general problem and discover that it is in fact easier to solve, in that it avoids the need to solve an integral. Consider a needle of length ℓ and planks of arbitrary width d. Define $E_\ell :=$ expected number of cracks crossed by the needle $= \mathbb{E}(X_\ell)$, where X_ℓ is the number of cracks crossed. We look at how E_ℓ depends on ℓ. Assume a needle of length $\ell = \ell' + \ell''$, so that the needle can be thought of composed of two segments of lengths ℓ' and ℓ''. Call $X'_{\ell'}$ the number of crossing by the segment of length ℓ' and $X''_{\ell''}$ the number of crossings of the segment of the needle of length ℓ''.

$$E_{\ell'+\ell''} = \mathbb{E}(X'_{\ell'} + X''_{\ell''}) = \mathbb{E}(X'_{\ell'}) + \mathbb{E}(X''_{\ell''}) = E_{\ell'} + E_{\ell''} =$$

From this property (and continuity of E_ℓ) we deduce that E_ℓ has a linear form: $E_\ell = \alpha\ell$, for some α.
If $\ell < d$, as the needle cannot cross two cracks, then X_ℓ turns out to be an indicator variable:

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

So E_ℓ is the probability to cross a crack. Now we have to find the value of α.

For this purpose, let us be even more general and bend the needle with an angle between two segments of lengths ℓ' and ℓ''. We have again, from linearity of $\mathbb{E}(.)$:

$$\mathbb{E}(\text{number of cracks crossed}) = E_{\ell'} + E_{\ell''} = \alpha(\ell' + \ell'') = \alpha\ell$$

where $\ell = \ell' + \ell''$ is the total length of the needle. We can bend it several times to any polygonal shape with still no impact on the expected number of crossings:

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

Pushing this technique to the limit, we can build a needle of any shape of well-defined length ℓ and the relation above will always be verified.

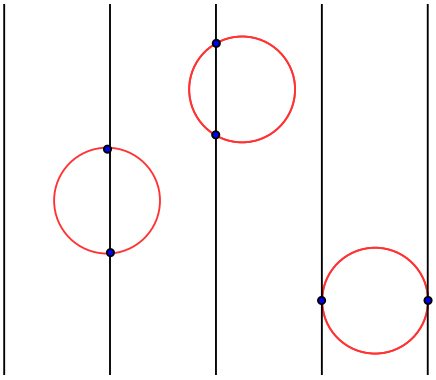


Figure 5.2.: Circular needles of radius 1 crossing cracks of width 2

To fix α, let us look at a circular needle of diameter d, which crosses the cracks twice, always (see figure 5.2)! Therefore $E_{\pi d} = 2$. On the other hand $E_{\pi d} = \alpha\pi d$, and we conclude with $\alpha = \frac{2}{\pi d}$. We therefore have

$$E_\ell = \frac{2\ell}{\pi d}.$$

By taking $d = 2\ell$, we find $E_\ell = \frac{1}{\pi}$.

□

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)
- $\pi = \frac{n}{k}$

What is the 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 $\text{Var}(X_i) = \frac{1}{\pi}(1 - \frac{1}{\pi})$ and the X_i are independent, it follows that

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

We can now compute the standard deviation $\sigma = \frac{1}{\sqrt{n}}\sqrt{\frac{1}{\pi}(1-\frac{1}{\pi})}$, which is interpreted as the typical order of magnitude of the error when evaluating $\frac{1}{\pi}$. Using the Central Limit

Theorem for a large number of draws, we know that the number of successful draws follows a Gaussian, and we deduce a confidence interval:

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

with 95% probability.

This is a probabilistic algorithm, where the answer is within the confidence interval only with some probability. Such a random algorithm is named a *Monte Carlo* algorithm.

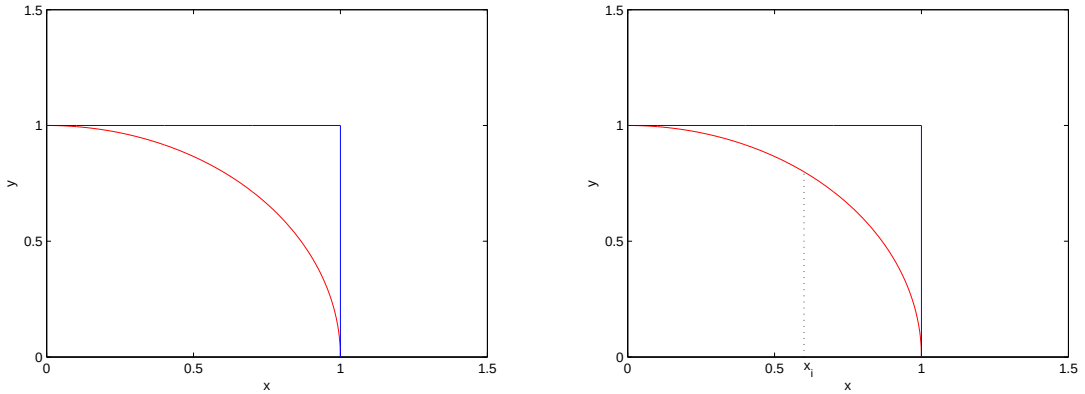
This specific Monte Carlo algorithm converges slowly, as the error (standard deviation) is in $1/\sqrt{n}$. To gain one digit of accuracy, one needs 100 times more throws, which is costly.

5.4.2. 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 5.3a. 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).



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

Figure 5.3.

5.4.3. A slightly smarter method

We can easily improve the last naive π estimation:

- Pick x_i randomly between 0 and 1 (see figure 5.3b);

- Since, 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 5.3b, is $\sqrt{1-x_i^2}$, under uniform distribution. Since we know this probability, we can use it directly instead of picking a random y_i! We then compute

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

Again the error of this algorithm decreases like $\sim \frac{1}{\sqrt{n}}$. The procedure above corresponds to the numerical approximation of $\int_0^1 \sqrt{1-x^2}dx = \pi/4$ using the rectangle method with random integration points x_i. One-dimensional integrals are usually better evaluated by deterministic methods (rectangles, Simpson, ...). Indeed, the error of the rectangle method (the simplest deterministic method) is in $\mathcal{O}(\frac{1}{n})$, far better than $\frac{1}{\sqrt{n}}$. The hidden constant in the $\mathcal{O}(.)$ depends on the function however.

Remark: at the end of this Chapter we describe, for the record, the top-notch methods actually used to compute π.

5.5. Why and when should we use a Monte Carlo algorithm for integration?

A Monte Carlo method can be used to evaluate any integral $\int_X F(x)p(x)dx$ (where p is a probability density function), or sums $\sum_X F(x)p(x)$ (where p is a probability distribution over a large discrete set X).

The interest of the Monte Carlo integration is at least threefold:

Robin Hood effect: For every deterministic integration method, there are functions for which the method will fail. For example, for high frequency sine (see figure 5.4a), 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 poor worst-case behaviour. The choice of random points makes all instances behave equally well.

For higher dimensional integrals: $\underbrace{\int\int\ldots\int}_{d} F(x_1,\ldots,x_d)dx_1\ldots dx_d$

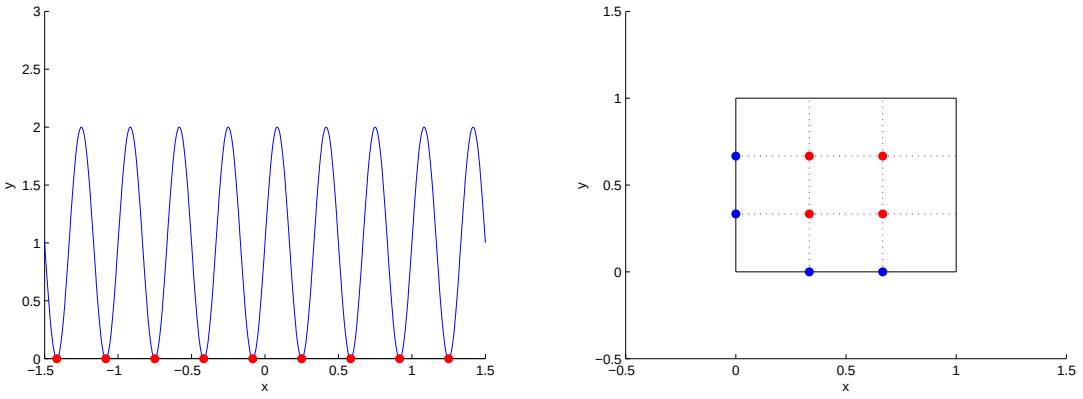
For deterministic methods, n points regularly spaced in $[0,1]^d \to \sqrt[d]{n}$ points in each dimension (see figure 5.4b: by choosing the 4 red points, we have 2 blue points in each dimension). Typically, the error will behave as $\frac{1}{\sqrt[d]{n}}$ (when the dimension d increases, we need exponentially more points to fill the space and preserve the accuracy)—this is a form of the famous "curse of dimensionality".

To see this otherwise, assume F decomposes as a product $F_1(x_1)F_2(x_2)\ldots F_d(x_d)$, so that $\int_{[0,1]^d} F = \int F_1 \int F_2 \ldots \int F_d$. Of course if we know this then we will evaluate each $\int_{[0,1]^d} F_i$ separately, which is much easier. Let us evaluate $\int_{[0,1]^d} F$ numerically as an arbitrary multidimensional integral (ignoring the factorization) with the following

multidimensional rectangle method: consider the grid of all points in $[0,1]^d$ with coordinates $1/3$ or $2/3$, we evaluate F in these $n = 2^d$ points and take the average of these evaluations (up to a normalising constant) as the estimate of $\int_{[0,1]^d} F$. Given the factorisation $F = F_1 F_2 \dots F_d$, this method amounts to estimating each $\int_{[0,1]} F_i$ as $\frac{F_i(1/3)+F_i(2/3)}{2}$ (error in $\mathcal{O}(1)$), and multiplying the outcome (error in $d\mathcal{O}(1)$) — a very crude estimate indeed. So even an exponential number of points placed in a regular grid cannot evaluate a high-dimensional integral accurately.

The random Monte-Carlo algorithm, which evaluates F in n randomly chosen points in $[0,1]^d$, still behaves in $\mathcal{O}(\frac{1}{\sqrt{n}})$ error (because it depends on the properties of variance of i.i.d. random variables, regardless of k) and is typically a better choice for integrals in medium or high dimension.

For domains with complicated geometry or no explicit description If X is not simply described, e.g. as an hypercube, but has no simple explicit description, it is not even clear how to apply a deterministic method such as the rectangle method. If instead we have a method to generate points within X with probability $p(x)$ (possibly with the help of a Markov Chain Monte Carlo method as described below), then we can still apply the Monte Carlo method.



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

Figure 5.4.

A frequent problem in engineering and science is the estimation of expected quantities defined on a stochastic model on a large number of variables. For example one may want to compute the expected speed of traffic where events, drivers' decisions, etc. are modelled stochastically. Or one may want to estimate the viscosity of macromolecule materials. Etc. Measuring the expected quantity according to a certain probability distribution is essentially an integration problem. The large number of degrees of freedom to integrate make it a typical example where a Monte Carlo method is preferred to deterministic integration methods: one generates N random scenarios/trajectories/configurations for the system in order to estimate the quantity of interest.

5.6. Markov Chain Monte Carlo (MCMC) methods and Metropolis-Hastings algorithm

Sometimes the generation of samples according to a given probability distribution is not straightforward, unlike the simple examples above where uniform sampling on a well-defined piece of $\mathbb{R}^n$ is sufficient.

A frequent occurrence is when the probability distribution is only known up to a factor. This occurs e.g. in physical systems where we know that a configuration of energy E is observed with probability $\propto e^{-E/kT}$ where k is a physical constant (Boltzmann's constant) and T is the absolute temperature, but the normalising constant, itself the result of an integration problem (summing $e^{-E/kT}$ over all possible physical configurations), is difficult to compute.

An even simpler case is when we want to sample a set with uniform distribution but the total volume or cardinality of the set X is unknown. Assume for instance that we want to sample nodes in the connected component of a node x_0 in a large graph. Here X is the set of nodes in the connected component, and f is a node feature, e.g. the age in a social network. If the graph is truly large, and can only be explored through adjacency lists (hopping from neighbour to neighbour), then we do not want to e.g. make a depth-first search from node x_0 to explore exhaustively the connected component, and want to resort to a uniform sampling instead.

We indicate here a key technique to generate sample to any probability distribution $p(x)$ proportional to (known) $f(x)$ on configuration x in a discrete or continuous space X. The idea is to perform a *random walk* (also called *Markov chain*) on X. Starting from an arbitrary x_0, we jump randomly (with a certain probability law) to another point x_1, then to x_2, etc., in a way that the stationary distribution, i.e. the probability distribution for x_t in the limit of large t, is precisely $p(x)$. The practical number of steps T for which the distribution of x_T is close to stationary distribution $p(x_T)$ is called the *mixing time* of the random walk.

Starting from an arbitrary x_0, one simply has to simulate the random walk for sufficiently many steps and retrieve the subsequence $x_T, x_{2T}, x_{3T}, \ldots$ as an approximately i.i.d. sequence of samples with the correct distribution. A sampling technique obeying this general scheme, or variants of it, is called a *Markov Chain Monte Carlo* (MCMC) method.

The most common MCMC strategy for jumping from x_t to x_{t+1} is the *Metropolis-Hastings random walk* (Metropolis-Ulam 1949, Hastings 1970). From $x_t = x$ we generate a *candidate* y with probability $g(y|x)$, for some well-chosen distribution $g(.|x)$. We then compute an *acceptance probability*

$$\alpha(y|x) = \min(\frac{f(y)g(x|y)}{f(x)g(y|x)}, 1).$$

We now set

$$\begin{cases} x_{t+1} = y & \text{with probability } \alpha \\ x_{t+1} = x & \text{with probability } 1 - \alpha \end{cases}$$

In the first case, the randomly generated candidate y is accepted as the next iterate x_{t+1}, while in the last it is rejected.

Note that each $g(.|x)$ is a probability distribution over X that also defines valid transition probabilities over X, thus defined a random walk. Thus the Metropolis-Hastings random walk acts as a 'correction' of the random walk g, by aborting each jump with a probability $1-\alpha$. The resulting transitions probabilities for the Metropolis-Hastings random walk from x towards any $y \neq x$ are thus:

$$m(y|x) = g(y|x)\alpha(y|x) = \min(\frac{f(y)g(x|y)}{f(x)}, g(y|x)).$$

Theorem 5.2. Let f be a real valued function over a set X, and for every $x \in X$, let $g(.|x)$ define a probability distribution over X. The stationary distribution of the Metropolis-Hastings random walk on X, as defined above, is unique and equal to

$$p(x) = \frac{f(x)}{\sum_{x \in X} f(x)},$$

provided that every configuration x can be reached from any configuration x_0 in finitely many jumps with non-zero probability.

Proof. The second condition asks that the Markov chain is ergodic, i.e. the possible jumps on X form a strongly connected graph, and is a well-known condition on the uniqueness of the probability distribution.

We just need to check that $p(x)$ is indeed the stationary distribution, i.e. that if x_t is distributed randomly according to $p(x_t)$ then x_{t+1} (the next state as chosen from Metropolis-Hastings rule) is again distributed according to $p(x_{t+1})$.

For any $y \neq x$, recall that $m(y|x) = g(y|x)\alpha(y|x)$ is the probability that y is the next iterate x_{t+1}, knowing that $x_t = x$. Then it is easy to check that

$$f(x)m(y|x) = f(y)m(x|y).$$

as both sides are equal to $\min(f(x)g(y|x), f(y)g(x|y))$. Defining $p(x) = f(x)/\sum_X f(x)$, we deduce

$$p(x)m(y|x) = p(y)m(x|y).$$

in other words:

$$P(x_t = x, x_{t+1} = y) = P(x_t = y, x_{t+1} = x)$$

This condition, called 'detailed balance', or 'reversibility', implies that $p(x)$ is stationary. Indeed

$$P(x_t = x) = \sum_y P(x_t = x, x_{t+1} = y) \tag{5.1}$$

$$= \sum_y P(x_t = y, x_{t+1} = x) \tag{5.2}$$

$$= P(x_{t+1} = x). \tag{5.3}$$

which concludes the proof. □

The notations above are for a discrete set X. In the continuous case, e.g. X being part of $\mathbb{R}^d$, then $g(.|x)$ is understood as a probability density function, and the normalising constant is $\int_X f(x)dx$. A typical example in this situation for $g(.|x)$ is a Gaussian centred around x.

Note that there is a large freedom in the choice of $g(.|.)$, whether in discrete or continuous time. In practice the difficulty is to obtain a reasonably small mixing time T for the random walk.

The example of sampling the nodes of a connected graph uniformly, mentioned above, is solved for instance with $g(y|x) = 1/\text{degree}(x)$ (for any neighbour y of x) and $f = 1$ (we want uniform distribution). The probability of jumping to neighbour y starting from x is then

$$m(y|x) = \min(1/\text{degree}(x), 1/\text{degree}(y)).$$

As already mentioned, a typical application of the Metropolis-Hastings random walk is the sampling of physical microscopic configurations following Boltzmann's distribution.

The simulated annealing method in combinatorial optimisation follows the same principle: here X is the set of feasible solutions for a combinatorial problem $\min\{c(x) : x \in X\}$ with objective function $E(.)$, e.g. the Travelling Salesman Problem, and one wants to explore X so as to reach feasible solution x with probability $\propto e^{-E(x)/T}$, where T is a 'temperature' that is slowly decreased, thus converging to a situation where the global minimum is reached with probability one. The exploration of the feasible set X is performed with Metropolis-Hastings's algorithm.

Another application is in the field of statistical inference. Imagine that x is a 'hidden cause' or 'hidden parameter' generating the observations z according to well-known distribution $P(z|x)$, sometimes called the 'likelihood function'. We're interested in $P(x|z)$: given the observations, what are the probable causes explaining these observations? From Bayes' formula, we have:

$$P(x|z) = \frac{P(z|x)P(x)}{\sum_x P(z|x)P(x)}.$$

Here $P(x)$ is sometimes called the *prior* distribution, in that it describes the probability of hidden causes prior to any observation, and $P(x|z)$ is the *posterior* distribution, i.e. the distribution on x as adjusted after observing z.

In this formula, the denominator is often difficult to compute explicitly. Thus we can hope to sample the posterior thanks to MCMC techniques such as the Metropolis-Hastings method.

5.7. Ex-cursus: Deterministic methods for computing π

We exemplified numerical Monte Carlo algorithms on the computation of π. It is only fair to complete with an ex-cursus on efficient algorithms for π.

Those algorithms are exclusively deterministic. Of course evaluation the area of the circle with deterministic methods such as Simpson's method offer fairly efficient methods to evaluate π. However the most performant methods involve expansion of π as an infinite sum or product.

For instance one may express

$$\frac{\pi}{4} = \arctan(1)$$

and develop the Taylor series $\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots$, leading to Madhava-Leibniz formula (Madhava 1400, Leibniz 1682) :

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \ldots$$

This formula is not competitive, with an error in $\mathcal{O}(1/n)$ for n terms, as Taylor's development around zero is more efficient for values closer to zero, while here it is evaluated in 1.

Much more efficient is therefore the Taylor expansion of John Machin's formula (1706):

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

It is proved from the usual trigonometric identities, or from the observation $(5+i)^4(-239+i) = -4(13^4)(1+i)$ in the complex plane, and expressing this equality in the exponential notation.

We now evaluate $\arctan(1/5)$ and $\arctan(1/239)$ with the same Taylor series. The slower converging sequence of the two is $\arctan(1/5)$, which still converges in $\mathcal{O}(1/5^n)$ for n terms, much faster than Madhava-Leibniz's formula. Many Machin-like formulae have been developed for modern computations of π. Other methods exist based e.g. on Ramanujan's formulae for π, for instance:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

The following variant was developed by the Chudnovsky brothers (1988):

$$\frac{1}{\pi} = \frac{\sqrt{1005}}{4270934400}\sum_{k\geq 0}(-1)^k\frac{(6k)!}{(k!)^3(3k)!}\frac{13591409+545140134k}{640320^{3k}}.$$

It is the basis of all recent records of computation of π, for instance the 50 trillion digits computed by Timothy Mullican in 10 months, until Jan 2020, on a single computer (with a lot of memory).

Computation of π to trillions of digits involves huge multiplications where Schönhague-Strassen algorithm is useful.

Finally, we note that we can compute a specific binary digit of high rank in a cheap way, without having to compute all previous digits. This is the Bailey-Borwein-Plouffe formula (1995):

$$\pi = \sum_{k\geq 0}\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right)$$

The expression as sums of powers of 2 can be exploited (nontrivially) to find the kth binary digit directly, in time $\Theta(k\log^3 k)$ and space $\Theta(\log k)$. It is used to check calculations made

with Chudnovsky's formula, or to compute that 'The Two Quadrillionth Bit of Pi is 0 !' (as Tsz Wo Sze wrote in 2010).

5.8. Monte Carlo algorithms for decision problems

We have described Monte Carlo algorithms for algorithms where the output is a numerical value. Monte Carlo algorithms are also useful in number theory and cryptography, such as generating random 'probably prime' numbers, or checking primality of a given number with a low probability of error.

The latter is a decision problem, i.e. a problem where the output is YES or NO. In this section we describe and analyse a simple decision problem where a Monte Carlo method is spectacularly useful to obtain "probably correct" answer at cheap cost. It is a particular case of the powerful Schwartz-Zippel algorithm for testing polynomial identities, see Section 7.4.2.

5.8.1. Checking matrix products

Suppose one wants to check that $AB = C$, with A, B and C $n \times n$ matrices.

- Obvious method: compute AB. 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 $(AB)_{i,j}$ ($\mathcal{O}(n)$ 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 localized to a few entries.
- If $AB = C$ then $\forall v \in \mathbb{R}^{n\times 1}$, $\underbrace{ABv}_{\mathcal{O}(n^2)} = \underbrace{Cv}_{\mathcal{O}(n^2)}$ (such checksum avoids localization by involving typically all or many columns of AB and C, and checking if $AB = C$ in the direction of v). How to pick v? Let us choose it randomly in $\{0,1\}^n$. Then ABv (resp. Cv) amounts to summing a random subset of columns of AB (resp. C)
  Claim :
  1. If $AB = C$, then $ABv = Cv$ with probability one (trivial).
  2. If $AB \neq C$, then $ABv \neq Cv$ with probability $\geq 1/2$.

  We prove the claim. Say there is an error in column j of C (and possibly elsewhere). For any $v \in \{0,1\}^n$ and any index j, call $v_{\bar{j}}$ as v where entry j is flipped ($0 \leftrightarrow 1$).
  - Either $ABv \neq Cv$ and $ABv_{\bar{j}} \neq Cv_{\bar{j}}$;
  - or $ABv = Cv$ and $ABv_{\bar{j}} \neq Cv_{\bar{j}}$;
  - or $ABv \neq Cv$ and $ABv_{\bar{j}} = Cv_{\bar{j}}$;
  - but $ABv = Cv$ and $ABv_{\bar{j}} = Cv_{\bar{j}}$ is impossible because then

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

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

The fact that only the three first cases are possible implies that 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's algorithm:

Code 5.2: Freivalds Algorithm (A,B,C)

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

However with this algorithm, false negatives (i.e. $AB \neq C$ but undetected, with probability $\leq \frac{1}{2}$) are possible. To improve Freivalds's algorithm, we can use amplification of stochastic advantage: We can try again with other v! We will never be sure that $AB = C$, but the probability of error will decrease a lot if we test a lot of v:

Code 5.3: Repeat(A,B,C,k)

```
1         Repeat Freivalds k times;
2         If k conclusions "A*B==C"
3             then print "A*B=C" [maybe];
4         Elseif at least one conclusion "A*B~=C"
5             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 $\mathcal{O}(n^{2.3\ldots})$ or $\mathcal{O}(n^3)$.

5.8.2. Amplification of stochastic advantage

Amplification of stochastic advantage is very powerful when errors on a decision problem are one-sided: there may false negatives, but no false positives, or the contrary (in Freivalds's case, if we conclude "$AB \neq C$", it is with certainty). For instance if

- answer is YES and we conclude YES with probability 1;
- answer is NO but we conclude NO with probability $\geq \epsilon$;

(high probability $1 - \epsilon$ for a false positive) then repeating the process k times as above reduces the probability of a false positive to $(1 - \epsilon)^k$, which reaches $\frac{1}{2}$ with for $k = -1/\log_2(1 - \epsilon) = \Theta(\frac{1}{\epsilon})$. As above, the error is reduced to 0.01% for $k \approx \frac{10}{\epsilon}$.

What now if a Monte Carlo algorithm produces possible false negatives and false positives ? For example:

- 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 is only guaranteed to be correct slightly more often than a coin flip)

Amplification of stochastic advantage still works! We simple 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

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

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. Therefore, we can say for example that $\sum_{i=1}^{n} X_i$ has 2.5% of probability to be 2 standard deviations below its mean, and 0.15% of probability to be 3 standard deviations below its mean.

This means that if we want probability of error limited to 2.5%, we must have that $\frac{1}{2}n$ is at least 2 standard deviations below the mean $(\frac{1}{2} + \epsilon)n$, which is the case as soon as $\epsilon n \geq 2\sqrt{n}\sqrt{\frac{1}{4} - \epsilon^2} \Rightarrow n \geq \frac{4}{\epsilon^2}\left(\frac{1}{4} - \epsilon^2\right) \sim \frac{1}{\epsilon^2}$ for small ϵ.

Similarly, if $n \gtrapprox \frac{9}{4\epsilon^2}$, then we reach a wrong conclusion 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 $n \sim \frac{9}{4\epsilon^2}$), the probability of error decreases much.

5.9. Monte-Carlo solutions for optimisation problems

For some optimisation problems it so happens that a randomly computed solution can be proved to be relatively close to optimality.

For example, the MaxCut problem consists in cutting the nodes of an undirected graph in two classes so the number of edges between the sides of the cut is maximal. Taking a random cut (every node being randomly assigned to set S or T) can be seen to place on average half of the edges between S and T.

Indeed, every given edge e is in the random cut (linking a node from S with a node from T) with probability $1/2$. Thus, considering X_e the indicator variable of whether e is in the random cut, the cut size is $\sum_e X_e$, and $\mathbb{E}\sum_e X_e = \sum_e \mathbb{E}X_e = \frac{1}{2}|\text{Edges}|$. This is clearly at most a factor away from the optimal maxcut.

This algorithm is derandomised, i.e. transformed into a deterministic algorithm with similar performances, later in this chapter.

Note that another famous random approximationg algorithm for MaxCut is a factor 0.878 away from optimality, a much more impressive perfomance due to Goemans and Williamson (1995), using semidefinite relaxation techniques.

5.10. Las Vegas Algorithms

Another kind of algorithms are those that always provide a correct answer (unlike the Monte Carlo algorithms), but with random execution time. They are called Las Vegas algorithms. These include Random QuickSort, Random selection/Median, etc.

5.10.1. The 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;

This problem (and its obvious generalisation to n queens on an n-by$n$ chessboard) is a classic mathematical puzzle, and a classic benchmark for algorithms searching solutions in a large space. On the mathematical side, it has been proved that there are $(0.143n)^n$ solutions to the n-queen version (Simkin 2021). On the algorithm see a basic deterministic algorithm, and it is improved by randomisation.

5.10.1.1. Brute force algorithm

Explore systematically the tree of possibilities for each row and **backtrack** when reaching a dead-end:

- place a queen on the first row;
- place a queen on the second row (but different column, and different diagonal from previous queen);
- place a queen on the third row (but different column and diagonal from previous queens);
- $\vdots$
- if no position available : backtrack

This is a depth-first search in the tree of possibilities. If at each step, we choose the column with the smallest index available, the algorithm finds the solution in 114 queen placements.

5.10.1.2. Placing queens randomly

The random algorithm proceeds as the brute force algorithm, with two differences:

- on each row we place the queen randomly
- when no position is available on some row, we start the process **from scratch**, no backtracking and no memory kept of previous attempts.

In summary:

- place a queen randomly on the first row;
- place a queen randomly on the second row (but different column, and different diagonal from previous queen);
- place a queen randomly on the third row (but different column and diagonal from previous queens);
- $\vdots$
- if no position available : start again from first row

The expected time (measured in queen placements) is:

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

$$\text{because } \left[\frac{1}{P[Success]} - 1\right] = \mathbb{E}\left[\text{number of failures beofre getting a success}\right]$$

Empirically we find that the probability of success in around 0.129, and failures place on average 5 or 6 queens before failing. Thus:

$$\begin{aligned}\mathbb{E}\left[\text{time success}\right] &= 8 + (\frac{1}{0.129} - 1) \cdot 6 \text{ queen placements} \\ &\approx 50\text{queen placements}\end{aligned}$$

whereas the backtracking solution will get a success after 114 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". Systematic exploration introduces wrong assumptions that are kept for a long time in the exploration. It turns out that the best strategy to queen's problem is to choose 3 queens randomly and then explore the rest deterministically.

5.10.2. Hash functions and universal hashing

We mentioned hashing tables earlier in this chapter, but left in the shade the choice of the hash function. It should be chosen so as to satisfy as much as possible the Simple Uniform Hashing assumption: for a large class of probability distributions on the keys $x_1, x_2, \ldots$, the hashed keys $h(x_1), h(x_2), \ldots$ should behave as an i.i.d. uniformly distributed sequence. In other words, the hash function should be as 'random'-looking as possible.

A practical hash function with value in $\{0, 1, \ldots, t-1\}$ takes for example the form $h(x) = \lfloor t(rx \mod 1) \rfloor$ (indeed an integer between 0 and $t-1$) for some irrational number r. A good choice turns out to be $r = \frac{\sqrt{5}-1}{2}$, the inverse of the golden ratio.

Another simple choice is of the form $h(x) = x \mod t$, for some well-chosen t (typically a prime number). This is used for example on Belgian account numbers (IBAN), where the last two digits is the hashed key of the previous digits modulo 97 (the largest 2-digit prime number), in order to detect an error in the number with probability $\frac{96}{97}$, if the Simple Uniform Hashing assumption is indeed verified for the probability distribution followed by error patterns.

As a side note, in the latter example, we see that random errors may be detected efficiently, but an adversarial interference may easily fool the hash function by creating errors that will go undetected. Cryptographic hash functions (e.g. SHA-1, SHA-256 MD5) make it very hard to produce a key hashing to a particular number.

Another choice for a hash function consists in choosing *randomly* a function among a family of hash functions. We call such a family *universal* if for any two keys x, y, a hash function h chosen uniformly at random in the family creates a collision $h(x) = h(y)$ with probability at most $1/t$.

An obvious choice for such a family is the family of *all* t^N functions from the N possible input keys to the t hashed keys. Indeed only a fraction $1/t$ of them create a collision between x and y. However we would like a smaller family of functions, easy to describe and compute. Such an example is given by

$$h_{ij}(x) = ((ix + j) \mod p) \mod t$$

where p is a prime number larger than N (thus larger than all keys x), and $1 \leq i \leq p$ and $0 \leq j \leq p$ are integers. Choosing a random i and a random j, uniformly in their range, guarantees universal hashing. Why? The crucial point, admitted here, is that the arithmetics modulo p, for a prime number p, in $\{0, 1, \ldots, p-1\}$ is a *field*, i.e. the multiplication by an integer not multiple of p is invertible: for i not multiple of p, the equation $ix = a$ has a unique solution x modulo p, which can be denoted a/i.

Thus the map $(x, y) \mapsto (ix + j, iy + j) \mod p$ is a bijection on $\{0, 1, \ldots, p-1\}^2$ for all $i \neq 0$ and j. In particular if $x \neq y$ then $ix + j \neq iy + j$.

Moreover, while taking random i, j uniformly on their range, and fixing distinct $x \neq y$, one reaches all distinct pairs of $\{0, 1, \ldots, p-1\}^2$ with equal probability. Indeed given $x \neq y$ and $a \neq b$ all in $\{0, 1, \ldots, p-1\}$, one can solve the equation $(a, b) = (h_{ij}(x), h_{ij}(y))$ for i, j as $i = (a-b)/(x-y) \mod p$ and $j = a - ix \mod p$. Clearly the solution exists and is unique modulo p regardless of a and b. Thus the (i, j) pairs are in bijection with the distinct pairs (a, b). In conclusion, every distinct pair (a, b) is equally likely as the outcome of $(h_{ij}(x), h_{ij}(y))$ for a random choice of h_{ij}.

We now have to check the probability that the distinct pair $a \neq b$ will collide to the same cell of the hash table, i.e. the probability that $a = b \mod t$. This happens if $b = a \pm t$ or $b = a \pm 2t$, etc. This makes at most $\lceil p/t \rceil - 1$ values of b that collide with a, among all $p-1$ possible values for b. Therefore the probability of a collision $h_{ij}(x) = h_{ij}(y)$ is at most

$$\frac{\lceil p/t \rceil - 1}{p-1} \leq \frac{(p+t-1)/t - 1}{p-1} = \frac{1}{t}.$$

This proves universality of the h_{ij} family.

We would also like that the keys hash uniformly to $0, \ldots, t-1$. This is not strictly verified here. Indeed $z = ix + j \mod p$ is uniformly distributed on $0, \ldots, p-1$ (for any fixed x and random i, j) but once we reduce this number in $0, \ldots, p-1$ modulo t, we don't get uniform distribution over $0, \ldots, t-1$, because p (a prime number) is not an exact multiple of p. For instance, if $p = 13$ and $t = 5$, we see that a random number in $0, 1, \ldots, 12$ mapped into $0, 1, 2, 3, 4$ will fall on $0, 1, 2$ more frequently than on $3, 4$. Nevertheless this discrepancy disappears asymptotically for $p \gg t$, thus hashed keys are approximately uniformly distributed in practice.

This is an example of Las Vegas algorithm for hashing, relying on a Robin Hood effect. Indeed if the hash function h is fixed once for all, one may always fear applications where the hash function will behave terribly, sending many the n keys to the same slot, thereby degrading the performance of the dictionary operations, or failing to detect corruption of data. By contrast, with a random hash function as above, any given list of n keys used in an application will present the same expected number of collisions regardless of the application, performing sometimes slowly (if, unluckily, one has to manage many collisions) and most of the time fast.

5.11. Random number generation

How to generate random numbers ? Truly random numbers come from external processes such as a user's behaviour (e.g. timing of interactions with keyboard) or physical processes such as radioactivity, or thermal noise in the transistors. In a deterministic computer, randomness is simulated deterministically by generating *pseudo-random* numbers. A example of generator is the following (Blum-Blum-Schub, 1986):

1. Take p, q prime, congruent to $3 \mod 4$.
2. $n = pq$.
3. Choose $x_0 = \{2, 3, 4, \ldots, n-1\}$ not multiple of p or q.
4. $x_{t+1} = x_t^2 \mod n$ for $t = 0, 1, \ldots$
5. $y_t = x_t \mod 2$.

which outputs a pseudo-random sequence of bits y_t. The initial condition x_0, called the seed, determines the process: the same seed will generate the same sequence of bits. Moreover, because there are only $< n$ possible values for x_t, the sequence of bits is eventually periodic, thus obviously non-random. However the period, of the order of n, is exponential in the size of x_0. Thus a truly random x_0 can be 'expanded' into a comparatively large

number or random-looking bits. This specific generator is used in cryptography, because it has been proved that it is computationally hard to prove non- randomness and extrapolate the sequence.

Other, faster if less secure, algorithms exist, which are able to fool extensive statistical tests for a long time for a given seed. Most programming languages such as Python or Matlab use so-called Mersenne Twister's algorithm (Matsumoto-Nishimura 1997) with a seed defined by the user or derived from the computer clock.

5.12. Derandomisation

One conclusion one may draw from the previous section is that generating truly i.i.d. random bits is far from obvious. One may therefore see the number of random bits required by an algorithm as a costly resource, like time and space (memory).

In some circumstances it is possible to derandomise random algorithms, i.e. make them use less random bits, or no random bits at all while keeping the same or similar guarantees on accuracy and time/space complexity.

We illustrate two derandomisation techniques on the random algorithm for the MaxCut problem explained in a previous section.

5.12.0.1. Method of Pairwise independant bits

Suppose we are interested in the MaxCut problem for some graph with a set of nodes V and a set of edges E, with $|V| = n$ and $|E| = m$. Let us recall that the random algorithm for the MaxCut problem consists in assigning each node either to the partition S or T, by drawing n i.i.d. random bits $r_1, \dots, r_n$.

An obvious way to derandomize the algorithm is to try the 2^n possibilities for the random bits and take the maximal cut, at the expense of exponential slowdown. But can we derandomize it by simply using *less* random bits, without degrading (too much) the running time?

Remember that

$$\mathbb{E}(\text{Cut}(S,T)) = \sum_{e \in E} \mathbb{P}(e \text{ is in the cut}) = \sum_{e \in E} \frac{1}{2} = \frac{m}{2}. \tag{5.4}$$

The latter is obtained since, for e linking u and v, $\mathbb{P}(e \text{ is in the cut}) = \mathbb{P}(r_u \in S)\mathbb{P}(r_v \in T) + \mathbb{P}(r_u \in T)\mathbb{P}(r_v \in S) = \frac{1}{2}$, thanks to the pairwise independance of r_u and r_v. Thus, in order to keep equation (5.4) true, one only needs *pairwise* independance between the random bits. By drawing n i.i.d. random bits $r_1, \dots, r_n$, we get much more than pairwise between any pair r_i, r_j. In fact, we get full independance.

Take the following example to perceive the difference between pairwise independance and full independance. Suppose $X = Y \oplus Z$ with $X, Y, Z \in \{0, 1\}$ (here $\oplus$ is addition modulo 2, also known as XOR, i.e. eXclusive OR). Suppose Y and Z are pairwise independant, it follows that it is also the case for the pairs X, Y and X, Z. But X, Y and Z are not jointly independant since knowing Y and Z, one can recover X.

Suppose that now, we generate $r_1, \ldots, r_k$ i.i.d. random bits with $k < n$. For any subset $A \subseteq \{1, \ldots, k\}$ such that $A \neq \emptyset$, define

$$s_A = \sum_{i \in A} r_i \mod 2. \tag{5.5}$$

There are $2^k - 1$ such subsets A with associated (uniformly distributed) bit s_A. These bits are all pairwise independant. Indeed, for any $A, B \subseteq \{1, \ldots, k\}$ such that $A \neq B$, $A \neq \emptyset$, $B \neq \emptyset$, s_A is independant from s_B. Indeed we see that $s_A = s_B \oplus s_{A \setminus B} \oplus s_{B \setminus A}$. If $A \setminus B$ is nonempty, then $s_{A \setminus B}$ is independent from s_B, thus s_A is independent from s_B. The same argument can be written with $B \setminus A$ non-empty.

Coming back to the MaxCut problem, one only needs n pairwise independant random bits. Since the technique explained above gives $2^k - 1$ pairwise independant bits, it is sufficient to draw $k = \lceil \log_2(n+1) \rceil$ i.i.d. random bits. To derandomize completely, we only need to try all the 2^k different possibilities for $r_1, \ldots, r_k$, which is then linear in n since $2^k = \Theta(n)$.

5.12.0.2. Method of Conditional Expectation

For any Monte-Carlo algorithm A and input x, suppose we have a guarantee of the form $\mathbb{E}(A(x)) \geq q$ for some q. For instance, say R is the MaxCut random algorithm, one has guarantee $\mathbb{E}(R(G)) \geq \frac{m}{2}$ for G a graph with m edges.

This algorithm being determined by the realisations of the random bits $r_1, \ldots, r_n$, for some random output $A(x)$, it yields

$$\begin{aligned} \mathbb{E}(A(x)) &= \mathbb{P}(r_1 = 0)\mathbb{E}(A(x)|r_1 = 0) + \mathbb{P}(r_1 = 1)\mathbb{E}(A(x)|r_1 = 1) \\ &= \frac{1}{2}\mathbb{E}(A(x)|r_1 = 0) + \frac{1}{2}\mathbb{E}(A(x)|r_1 = 1). \end{aligned} \tag{5.6}$$

The guarantee implies that either $\mathbb{E}(A(x)|r_1 = 0) \geq q$, or $\mathbb{E}(A(x)|r_1 = 1) \geq q$, or both. So by fixing r_1, one can only improve the expectation for $A(x)$.

Suppose we can compute $\mathbb{E}(A(x)|r_1 = 0)$ and $\mathbb{E}(A(x)|r_1 = 1)$ efficiently, then we can choose $r_1 = r$ such that $\mathbb{E}(A(x)|r_1 = r)$ is the maximum of both choices. And we can repeat this step until we have completely derandomized the algorithm, by choosing $r_{i+1} = r$ that maximises $\mathbb{E}(A(x)|r_1, \ldots, r_i, r_{i+1} = r)$.

Take the MaxCut random algorithm example, noted R, for graph G. Suppose we already have assigned nodes $1, \ldots, i$ to sets S or T, and we want to assign node $i+1$. Let S_i and T_i containing the nodes assigned respectively to S and T up to node i, and $U_i = V \setminus (S_i \cup T_i)$, we can compute the expected value

$$\mathbb{E}(R(G)|S_i, T_i) = \mathrm{Cut}(S_i, T_i) + \frac{1}{2}\mathrm{Cut}(S_i, U_i) + \frac{1}{2}\mathrm{Cut}(T_i, U_i) + \frac{1}{2}\mathrm{Edges}(U_i, U_i) \tag{5.7}$$

where $\mathrm{Edges}(U_i, U_i)$ is the number of edges in U_i.

Since adding node $i+1$ to S and adding it to T are respectively equivalent to $r_i = 1$ and $r_i = 0$, following the strategy explained above, we assign $i+1$ to S if $\mathbb{E}(R(G)|S_i \cup \{i+1\}, T_i) > \mathbb{E}(R(G)|S_i, T_i \cup \{i+1\})$, and assign it to T otherwise.

Note that equation (5.7) yields

$$\begin{aligned}\mathbb{E}(R(G)|S_i \cup \{i+1\}, T_i) - \mathbb{E}(R(G)|S_i, T_i \cup \{i+1\}) \\ = \frac{1}{2}\mathrm{Cut}(T_i, \{i+1\}) - \frac{1}{2}\mathrm{Cut}(S_i, \{i+1\})\end{aligned} \tag{5.8}$$

which shows that this derandomisation is simply a greedy criterion: we assign $i+1$ to S if there are more links between node $i+1$ and T_i than with S_i. In this way we have obtained a deterministic greedy algorithm with a simple random algorithm.

6. Computability and decidability

In this chapter, we address the question of *computability*: what can computers do? Are there problems they cannot solve or functions they cannot compute, even in principle?

To answer this question rigorously, we must define precisely what is meant with 'computer', 'problem' or 'function'. In doing so, we will be able to build a theory of *computability*.

6.1. Decision problems, partial functions and machines

With our modern experience of computers and algorithmics, it is not hard to accept the following definitions as relevant.

Definition 6.1. A *problem* f is a (possibly partial) mapping from a set of instances to a set of outputs (both defined as words of a finite alphabet):

$$
\begin{aligned}
&f : A^* \to A^* \\
&\quad w \to f(w) \text{ or } \mathsf{undefined} \\
&\quad \text{with:} \\
&\qquad A = \text{ a finite alphabet, e.g.} A = \{0,1\} \\
&\qquad A^* = \text{the set of all finite words (or strings) with symbols of A,} \\
&\qquad\qquad \text{e.g.} \quad A^* = \{\emptyset, 0, 1, 00, 01, 11, 10, 000, 001, \ldots\} \\
&\quad \text{We call:} \\
&\qquad w = \text{ "input" or "instance"} \\
&\qquad f(w) = \text{"output"}
\end{aligned}
$$

The set A^* can always be identified to $\mathbb{N}$, through the natural ordering of A^*. Thus we may also see a problem as a partial function from $\mathbb{N}$ to $\mathbb{N}$. For instance the squaring problem is the function: $\mathbb{N} \to \mathbb{N} : n \mapsto n^2$.

Definition 6.2. A *decision problem* P is given by a total function of the subset: $A^* \to \{\text{YES,NO}\}$. Equivalently, it is given by a subset of A^* (the set of YES-instances, also called the positive instances).

A decision problem may informally be described as a yes/no question, e.g. 'Is n a prime number?'.

A computer may be trickier to define, because this can be done as several levels of abstraction. Strictly speaking, a computer is a physical object, usually a huge nonlinear electronic circuit described by a large system of stochastic (or even quantum-mechanical) differential equations describing the voltages, charges, currents in each device. We may abstract by a large but finite boolean circuit, with memory cells and logical gates. Nevertheless in this

chapter we abstract it in the way that is most familiar to the algorithmic/programming level. We consider that a computer is for example a *Python machine*:

Definition 6.3. A *Python machine* is given a syntactically correct, deterministic Python code that takes a string in A^* (for some alphabet A) as input, and that outputs a string in A^*. The machine has infinite memory, which means that the code can run without ever overflowing memory. In doing so, the machine computes a function, which associates to any given input string the corresponding output string. As the code may never stop (e.g. caught in an infinite loop), this function is possibly partial. This function is the *problem computed by the Python machine*. If the function is total (i.e. defined on every word in A^*), with output in $\{\mathrm{YES}, \mathrm{NO}\}$, we say that the corresponding decision problem is *decided* by the Python machine.

This gives us a precise notion of which problems can be solved by computers:

Definition 6.4. We say that this problem is *computable* if it is computed by some Python machine. If the problem is a decision problem, we say it is *decidable*.

We may wonder why we restrict inputs and outputs as single strings (or integers), instead of admitting more diverse types, including lists of strings, or pairs of floats, etc. For instance we may want to define a Python machine that solves the addition problem, which is the function $(m, n) \to m + n$.

The reason is that all types can eventually be recoded as strings. Even a pair or finite sequence of strings can be recoded without ambiguity as a single string. The same goes if we work with integers. An explicit bijection $\mathbb{N}\times\mathbb{N} \to \mathbb{N}$ is for example $(m, n) \to 2^m(2n+1)$. Since we want to build a mathematical theory of computers, it is convenient to restrict as much as possible, without loss of generality, the kind of objects we have to consider, and bring them under a 'canonical' form.

Of course we may define similarly Java machines, Matlab machines, Julia machines, etc. We will see however that this choice has no impact on the notion of computability or decidability. Before showing this, we turn to much simpler notion of machines, corresponding to a much simpler programming language: Turing machines, proposed by Turing in the 1936.

6.2. Historical context for Turing: the Entscheidungsproblem

One must realise the challenge for Turing to propose a mathematical definition of computing was made much harder by the fact that computers did not exist at all! Yet, definitions of 'algorithm' or 'algorithmic procedure' started to appear in the US from 1934, by Kleene and Church.

These works were motivated by Hilbert's and Gödel's work in mathematical logic. Hilbert and Ackermann asked the question in 1928: 'Is there a mechanical procedure to prove or disprove every mathematical theorem in finite time?' which obviously calls for a definition of 'mechanical procedure'. This is the *Entscheidungsproblem*. Hilbert was convinced that such a mechanical procedure would exist.

Gödel made a breakthrough in 1931 with his *incompleteness theorem*. This theorem shows that there are logical sentences about the integers (i.e. formal sentences such as $\forall n \exists m_1, m_2, m_3, m4 : n = m_1^2 + m_2^2 + m_3^2 + m_4^2$ where all the variables are integers) that

cannot be proved from the elementary axioms of arithmetics, and whose negation cannot be proved either. This showed that a *specific* mechanical procedure to prove every theorem, namely writing all possible proofs resulting from the axioms and elementary rules of logic until finding either the theorem or its negation, didn't actually work!

Although this came as quite a shock to many mathematicians, this did not exactly solve the Entscheidungsproblem: perhaps another mechanical procedure would work? (Hilbert himself said very little about Gödel's theorem, only to say that it didn't affect his conjecture) Hence the need to generalize Gödel's ideas. The ideas of Kleene (recursive functions theory) and Church (lambda calculus) were rooted in arithmetics and mathematical logic, and relatively unintuitive as a general definition of 'algorithm'.

Instead, Turing, who was following these American advances while doing a PhD thesis at University of Cambridge, wanted to take the problem from scratch and find a definition that would undeniably capture the notion of 'algorithm' or 'mechanical/effective procedure', whether in arithmetics, logic or otherwise. This lead him to invent a mathematical concept of 'computing machine', that will be later on enriched and converted into actual computers. On a theoretical level, Turing's definition will be proved to be equivalent to Kleene and Church's definitions, thus mutually reinforcing the legitimacy of all these definitions as 'the' definition of algorithms.

6.3. Turing machines

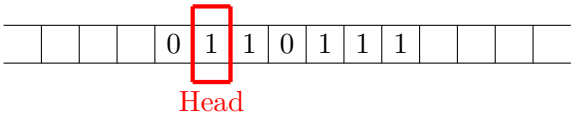
6.3.1. Definition of a Turing machine

Turing motivates his definitions of Turing machine and problem with a thought experiment, trying to capture without loss of generality the process of a human being executing an algorithm, i.e. a mechanical procedure to solve a problem that leaves no room to creativity or randomness.

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 scrutiny by the computer on the paper.
There is a specific halting criterion (e.g. 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* (abbreviated T.M.) is composed of:

- a bi-infinite *tape*, divided into cells, each cell either 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, and shift the tape to the left or to the right.



- a finite set of instructions, each numbered $(1, 2, 3, \ldots)$, telling what the head must do. They take the form:
  - either `STOP` (special stopping instruction)
  - `"if CurrentCell=blank/0/1/...`
    `then Write(blank/0/1/...)`
    `and ShiftLeft/ShiftRight/NoShift`
    `and goto Instruction k."`

Observe that this formal definition can be used to represent both humans and computers as we know them today:

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

Turing Machines offer a formal definition of what is computing. It fits well our intuitive understanding of computing.

From there, the input-output partial function from A^* to A^* computed by Turing machine is naturally called the *problem computed by the Turing Machine*.

A problem f is called *computable* if it is solved by at least one Turing machine.

This gives us a formal definition of *computability* for a problem: every problem with a Turing Machine solving it for every possible instance can thus be computed.

At this stage one must wonder how this relates to the more modern (and for us, more palatable) notion of computability by Python machines.

Theorem 6.1. A problem is computed by a Turing machine if and only if it is computed by a Python machine.

Proof. As the full proof is quite technical, this is only a sketch.

$\Rightarrow$ Equivalent to stating that it is possible to simulate a T.M. in Python: as Python is much more expressive than Turing Machines, this seems obviously possible $\rightarrow$ admitted.

$\Leftarrow$ Equivalent to stating that it is possible to convert any Python function into a T.M., hence that it is possible to write a Python-to-TM compiler. Even though quite tedious, this is actually feasible $\rightarrow$ admitted.

□

We say that Python is *Turing-equivalent*, or *Turing-complete*. So are Matlab, C, C^{++}, Java, Julia and most programming languages[1].

Theorem 6.1 offers us better insight into what a T.M. can do: whatever can be done in Python can be done with a T.M. The Church-Turing thesis, below, pushes this even further.

Thesis 6.1 (**Church-Turing**)**.** Any "obviously algorithmic" procedure (in the intuitive sense, described in natural language) can be converted to an equivalent T.M./Python machine.

This is *not* a theorem, because the notion of *obviously algorithmic* is not formally defined. However, 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 (sometimes with sweat and pain).
- all models 'obviously algorithmic procedures', including Kleene's recursive functions, Church's lambda calculus, Post machines (defined by Post more or less simultaneously and independently from Truing), and post-1948 programming languages proved to be all equivalent to one another.

The Church-Turing thesis implies that, for instance, the Euclidean division algorithm, "obviously algorithmic", can be solved by a T.M./a Python 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. Note also that with the Church-Turing thesis, the theorem that Python is Turing-equivalent is trivial: Python instructions are clearly algorithmic. Believing in Church-Turing thesis is therefore very convenient to shorten proofs dramatically!

One should not mistake thesis 6.1 with the *strong Church-Turing thesis*, which states that anything the brain can do (writing poetry, proving theorems, answering an exam, understanding a joke,...) can be simulated by a T.M. This strong version is highly controversial! It basically states that a strong Artificial Intelligence, not limited to very specific, well-defined tasks like like today's A.I., is possible.

Having formally defined what computing is, we can now prove theorems about computability.

6.3.2. Universal Turing Machine

The theorem states that there exists a universal Turing machine that can simulate an arbitrary Turing machine on an arbitrary input.

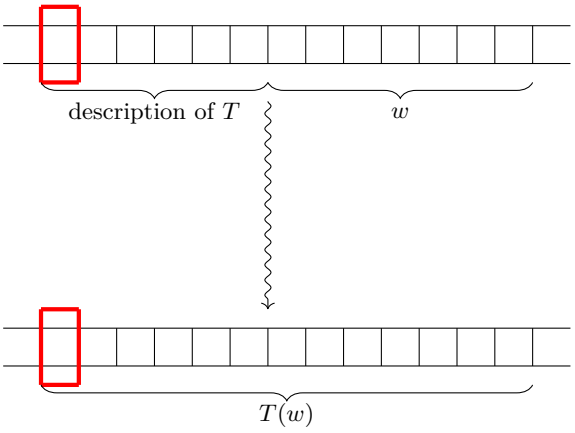
Theorem 6.2 (Turing, 1936)**.** The following problem is computable:

- *Input*: description of a T.M. (T) + description of finite word on initial tape ("w");

[1] Possibly, the simplest Turing-complete programming language beyond Turing machines is Brainf*ck: `https://en.wikipedia.org/wiki/Brainfuck`.

- *Output*:
  - If T stops when the initial tape is w: *the finite content of the tape when T stops*, which is equal to the output $T(w)$ of T on w;
  - If T never stops: undefined.

A T.M. solving this problem is called a "*universal Turing Machine*".



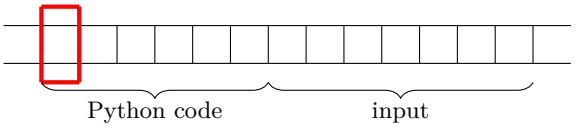
Proof (sketch). Any of the following is sufficient to prove this theorem:

- Either construct explicitely a universal Turing Machine (Cfr. Alan Turing's PhD thesis).
- Or construct explicitely a universal Python machine (same problem, yet slightly easier).
- Or by the Church-Turing thesis, simulating T step-by-step is obviously algorithmic (in the intuitive sense) thus there is a Turing Machine doing it.

□

Equivalent variants of this theorem:

- There is a Python machine simulating any other Python machine on any input.
- There is a Python machine simulating any other Turing Machine on any input. (this is a TM-to-Python interpreter)
- There is a Turing Machine simulating any Python machine on any input.



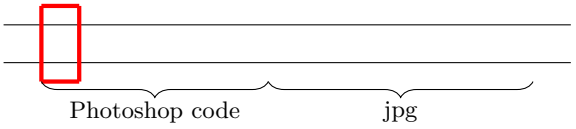
=Python-to-TM interpreter

The concept of Universal Turing Machine is essential, and reveals Turing's fundamental (genius) insight: there is no fundamental difference between the programme (i.e. description of a Turing Machine) and data (input to a Turing machine). A

T.M. can always be described as a finite word in an alphabet, and can thus be used as the input of a Turing machine.

Another insight of Turing is that his machines, although modelling the process of a human brain when applying an algorithm, was simple enough to be actually built. He quickly had the intuition that electrical, rather than mechanical processes were ideal to encode binary words. He took part after the war in building and programming some of the early universal computers in the UK. Our modern programmable computers are the direct descendants of universal Turing Machines, while the 4-operation pocket calculator, for example, is not universal.

e.g.



6.4. Computability and decidability

Using the definitions from the previous section, we now attempt to build some non-computable problems.

6.4.1. Existence of undecidable problems

Firstly, let us recall the concept of decision problem:

Definition 6.5. A *decision problem* is a total mapping: $A^* \to \{\text{YES,NO}\}$

A decision problem is *decidable* iff it is computable, i.e. iff there exist an algorithm (coded as Python machine, Turing Machine or whatever Turing-equivalent language) that can decide this problem.

Since there are many more decision problems that possible T.M. (or Python code), we can already claim that many decision problems are undecidable. On the one hand indeed, the decision problems (subset of $\mathbb{N}$, or mappings $\mathbb{N} \to \{\text{YES,NO}\}$) are uncountable (a proof of this is recalled later in the next section). On the other hand, the number of T.M. (or Python machines) are equivalent to words of a finite alphabet, and are therefore countable.

However, this raises another question: are there any *interesting* problems that are undecidable? This is somewhat similar to the notion of *transcendantal* number: any random real number is most certainly transcendantal, but it is more interesting to prove that specific numbers are transcendental (among which π).

6.4.2. The HALTING problem and diagonal argument

The following decision problem, called the HALTING problem, describes the question: *Does the T.M. T stop, when given w as initial tape?*

The HALTING problem:

Input:

$$\begin{cases} \text{Description of a T.M. (T);} \\ \text{a description of input (w) for T.} \end{cases} \tag{6.1}$$

Output:

- YES if T eventually stops on input w
- NO if T never stops ("infinite loop")

This problem was proven to be undecidable by Turing. In the following proof, we will manipulate Python machines instead of T.M. (which are equivalent) for readability purposes.

Theorem 6.3. (**Turing, 1936**): The HALTING problem for Python machines is undecidable.

Proof. By contradiction: we assume that there is a Python machine "Halt.py" deciding the HALTING problem. We build "Diagonal.py", which takes as argument another Python machine "P.py" as input.

Code 6.1: Diagonal(P)

```
1 INPUT : a Python machine P
2 def Diagonal(P):
3     if Halt(P,P) == YES: # P halts on input "P.py"
4         while True: # infinite loop: will not halt
5             continue
6     else:
7         return # if P does not halt on P, then stop
```

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

⇒ This is a paradox, hence a contradiction
⇒ "Halt.py" does not exist. □

This is called a *diagonal argument*. Why?

We create this table:

	x_1	x_2	x_3	x_4	...
P1.py	**Halt**	NotHalt	NotHalt	Halt	...
P2.py	Halt	**Halt**	NotHalt	Halt	...
P3.py	NotHalt	NotHalt	**NotHalt**	Halt	...
⋮	⋮	⋮	⋮	⋮	

Here x_j represents all words (inputs), and "Pi" represents all Python machines. Both are countable, so it is possible to number them and order them in a table. We have entry(i,j) = Halt iff Pi(x_j) halts.

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

In our case, we obtain (Diagonal(1),Diagonal(2),Diagonal(3),...) = ("NotHalt","NotHalt","Halt",...). This Diagonal function can be computed using the Halt function (supposing it exists), therefore it should appear as a row of our table. However this row is not in the table, by construction of the function, i.e. "Diagonal.py"≠"Pi.py", $\forall i$. This is a contradiction with the definition of the table that should enumerate all possible programs with all possible inputs. Consequently, our initial hypothesis that suppose the existence of the Halt function is false.

Cantor (1891) invented the diagonal argument to prove:

Theorem 6.4. The set of real numbers between 0 and 1 is uncountable, i.e there exists no surjection $\mathbb{N} \to [0,1[$. It follows that the set of all real numbers is also uncountable.

Proof. If there exists a surjection $c : \mathbb{N} \to [0,1[$, the real numbers between 0 and 1 can be numbered: $[0,1[= \{c_0, c_1, c_2, ...\}$. Let us suppose these numbers are written in a table :

c_0	0.	**1**	0	2	...
c_1	0.	3	**9**	1	...
c_2	0.	0	0	**1**	...
⋮	⋮	⋮	⋮	⋮	

From that table, we can create a new number $x = 0.202...$ (by flipping entries of the diagonal as: $0 \to 1, 1 \to 2, 2 \to 3, ..., 9 \to 0$), such that its i^{th} decimal of x is different from the i^{th} decimal of c_i. We now see that x is not in the array, by construction $\Rightarrow x \neq c_i \ \forall i$, but belongs to $[0,1[$, which is a contradiction. □

The same reasoning can be extended to the decision problems on integers, to prove that there are uncountably many such problems. We can build the table:

Input integers):	0	1	2	3	...
Problem P:	YES	YES	NO	YES	...
(in binary):	1	1	0	1	...

Where the third row characterizes the decision problem P, and we can create $x_P \in [0,1]$ from the last row: $x_P = 0.1101...$ (binary representation of a real number). This means that $\forall x \in [0,1[$, we can create a decision problem. So decision problems are uncountable because in bijection with $[0,1[$. As we said, this alone proves that many decision must be undecidable. One may see Turing's argument as a refinement of Cantor's argument to produce a specific, interesting, undecidable problem.

6.5. Undecidable problems

Other interesting undecidable problems exist, in the field of computing science but also in subjects belonging to pure mathematics.

6.5.1. Comparison between two programs

Input : Two Python codes `P1.py` and `P2.py`
Ouput : YES if $P1(x) = P2(x)\ \forall x$ (they compute the same function)
How do we prove the comparison problem is undecidable?

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

- $P1(y) = P(x)\ \forall y$ (constant function)
- $P2(y)$ does not halt $\forall y$ (e.g. an empty function with a `while True`)

If P does not halt on x, then $P1(y) = P2(y)\ \forall y$. If P halts on x, then $P1(y) \neq P2(y)$ for some y (in fact, all of them but we don't even need it). This implies that, if the comparison problem is computable, so is the HALTING problem. But HALTING is undecidable, and hence the comparison problem as well. □

The proof technique is typical: We constructed a **reduction** of the HALTING problem to the comparison problem, i.e. a total computable function converting each instance of the HALTING problem into an instance of the comparison problem of the same answer: positive (resp., negative) instances of the HALTING problem are mapped to positive (resp., negative) instances comparison problem.

It shows the HALTING problem is a particular case of the comparison problem, up to encoding. If we could solve the comparison problem, then we could solve the HALTING problem too.

All the problems in this subsection can be proven to be undecidable by reduction from the HALTING problem, or reduction to any other problem already proved undecidable.

6.5.2. Deciding Arithmetic formulae (1936)

Input : Logical formula of arithmetic (containing symbols like $\forall x, \exists y \cdots +, \times, = \cdots$, being understood that all variables $x, y, \cdots$ are in $\mathbb{N}$)
Output : YES if it is a theorem, i.e. it can be proved from the 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.

This problem is undecidable, as showed by Turing, and independently by Post, in 1936. This can be proved once again by reduction from the HALTING problem. This is done in showing that the statement 'Turing Machine T halts on input w' can be formulated with the language of arithmetics.

This proves that the Entscheidungsproblem is unsolvable for arithmetics. This is in turn a stronger theorem than Gödel's incompleteness theorem, which 'only' proved that a specific algorithm failed to prove or disprove all arithmetic formulae. This result was Turing's goal all along, even his work had consequences far beyond mathematical logic.

One could ask if the Entscheidungsproblem is solvable for logical formulae involving only *real* variables: for instance $\exists x : x^2 + 1 = 0$ is false but $\forall x, \exists y : x^2 + y = 0$ is a correct

theorem. This problem is actually decidable! The algorithm is due to Tarski, and generalises earlier works such as Sturm's theorem which can find whether a polynomial has a root located in any given interval.

This is a situation where real numbers behave more 'gently' than integers.

6.5.3. Hilbert's 10^{th} problem: Diophantine equations (Matiyasevich, 1970)

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

Note that this is a particular case of the decision problem for all arithmetic formulae above, which we already know is undecidable. Thus proving that this small fragment of arithmetics is *already* undecidable is a stronger result.

6.5.4. Matrix mortality problem (Paterson, 1970)

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

6.5.5. Tiling problem (Berger, 1966)

Input : A finite set of polyominoes (finite tetris-like shapes)
Output : YES if you can tile the entire plan with copies of these shapes (e.g. Figure 6.1)

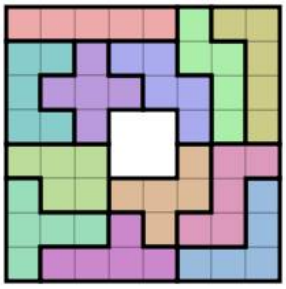


Figure 6.1.: Tiling problem

7. Complexity classes

7.1. Deterministic complexity classes

In this section we assume—like in the previous chapter— a deterministic computation model, i.e. the standard Turing Machines, or Python Machines without random instructions. We look at classes of decision problems which are decidable with an amount of resources (time or space, i.e. memory) that is bounded as a function of the input size.

7.1.1. Time

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

We generally consider that problems in P are those that use a reasonable amount of time, where

$$\mathsf{P} = \left\{ \begin{array}{l} \text{decision problems that can be solved by an algorithm running} \\ \text{in polynomial time, i.e. with worst-case time-complexity } \mathcal{O}(n^d) \\ \text{(where } n \text{ is the input size), for some } d. \end{array} \right\}.$$

This class is the same for all reasonable languages we may use to code the algorithm (*Python*, *Java*, *Matlab*, *Turing machine*...), as for instance coding in Turing Machines involve only a polynomial slowdown with respect to higher level languages.

Theorem 7.1. There is a decidable problem not in P.

Proof. We can use the diagonal argument !

Idea : we enumerate a list of *Python* programs that solve all problems in P. So, we enumerate all *Python* programs of the sort:

Code 7.1: Enumeration of all P problems

```
1     while flop <= (input_size)^d + c:
2         (any_block_of_code)
3     return Yes # forced stop
```

We know that **every** problem in P is solved by a Python machine in this list, and every Python machine in this list runs in polynomial time.

Let us 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 outputs the opposite of the diagonal of the table $problems \times inputs$!

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 $A \notin \mathsf{P}$. □

An analysis of the time complexity of the diagonal problem created in the proof shows that it runs at most in exponential time, for any reasonable way to list the programs and the inputs. Therefore, we have proved that the complexity class EXPTIME strictly includes P. The class EXPTIME is defined as follows:

$$\mathsf{EXPTIME} = \left\{ \begin{array}{l} \text{decision problems solved by an algorithm running in exponential} \\ \text{time, i.e. with worst-case time-complexity } \mathcal{O}(2^{(n^d)}) \text{ (where } n \text{ is} \\ \text{the input size), for some } d. \end{array} \right\}.$$

In fact we can repeat the argument for any time-complexity class, and create a hierarchy of larger and larger time-complexity class.

The theorem and its proof can be ultimately refined version of the so-called Time Hierarchy Theorem (Stearns-Hartmanis 1965):

Theorem 7.2. Let two functions $f(n)$ and $g(n)$ such that there is a Turing machine running exactly in time $f(n)$, and another running in time $g(n)$ exactly, and such that $f(n) \in o(g(n)/\log g(n))$. Then there is a problem solvable by a Turing machine in time $g((n)$ but not in time $f(n)$.

Such a fine version may depend more specifically on the computation model. This version was proved for the multitape Turing machine (where several tapes are available to the head to store information) and hold a fortiori for stronger computation models (Python, Matlab, etc.) but a one-tape Turing Machine model would lead to a larger required gap between f and g (due to the overhead to simulate one Turing machine while keeping track of its computation time, which is non-trivial with a single tape).

7.1.2. Space

We are also interested by the memory complexity. Generally we consider that an algorithm using a reasonable amount of memory is in PSPACE:

$$\mathsf{PSPACE} = \left\{ \begin{array}{l} \text{decision problems that can be solved by an algorithm using a polyno-} \\ \text{mial amount of space, i.e. with space-complexity in } \mathcal{O}(n^d), \text{ for some} \\ d \in \mathbb{N} \text{ (where } n \text{ is the input size).} \end{array} \right\}.$$

There is a Space Hierarchy Theorem similar to the Time Hierarchy Theorem, and proved in a similar way. Moreover space complexity classes are related to the time complexity classes, for instance in the following ways:

Theorem 7.3. $\mathsf{P} \subseteq \mathsf{PSPACE}$

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

Theorem 7.4. $\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$

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

Since a Turing machine deciding a problem in PSPACE must always stop (it cannot loop since it must always return a solution), it must do so in time $O(2^{n^d})$. □

7.1.3. Relation between P, PSPACE and EXPTIME

There are 3 possibilities compatible with the fact that $P \subsetneq \mathsf{EXPTIME}$:

$$\mathsf{P} = \mathsf{PSPACE} \subsetneq \mathsf{EXPTIME} \tag{7.1}$$
$$\text{or } \mathsf{P} \subsetneq \mathsf{PSPACE} = \mathsf{EXPTIME} \tag{7.2}$$
$$\text{or } \mathsf{P} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPTIME} \tag{7.3}$$

Which one of these three relations is true is still an open question. Most experts tend to believe that the third one, $\mathsf{P} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPTIME}$, is most likely to hold.

7.2. The NP class: problems with a nondeterministic polynomial time algorithm

7.2.1. First definition: with nondeterministic machines

Let us slightly enrich our standard deterministic computational model (Turing machines, Python machines or else) with a new instruction available in the programming language. It is a instruction which when called output a bit 0 or 1 in a nondeterministic fashion. Nondeterministic means that it will sometimes output a 0, sometimes a 1. The only difference with the more familiar 'random bit' instruction is that we do not care to assign a probability for each of these output: we just care to know that every time we query a new nondeterministic bit, we may possibly get a 0, and possibly a 1.

Thus such a machine, when called twice on the same input, may deliver two different outputs, depending on the 'computation path' followed by the machine, i.e. depending on the sequence of all nondeterministic choices made during the computation.

What does it mean for such a nondeterministic machine that it decides a certain decision problem?

Definition 7.1. A nondeterministic machine decides a decision problem if on all YES-instances, the machine may output a YES, while on all NO-instances, the machine always outputs a NO.

We find an analogy with the one-sided random algorithms. Recall for instance Freivalds's algorithm for deciding, given three square matrices A, B, C, whether or not $AB \neq C$. Here a YES-instance is a triple A, B, C such that $AB \neq C$, and Freivalds's algorithm *may* deliver a 'YES, $AB \neq C$' indeed. On a negative instance (A, B, C such that $AB = C$) the algorithm will always deliver a NO.

We may consider the random choices of Freivalds's algorithm as nondeterministic choices. Then we find that Freivalds's algorithm decides indeed (in the nondeterministic sense given in the definition above) the decision problem of checking whether three matrices A, B, C satisfy $AB \neq C$. In saying so, we just do not mention the fact (proved in an earlier chapter) that at least half of the computation paths (not just one) followed by Freivalds's algorithm on a YES-instance as input will end up with a YES.

Thus we can see the nondeterministic paradigm as a relaxation of the random computation paradigm. The connection between non-deterministic and random algorithms will be further explored later in this chapter.

$$\mathsf{NP} = \left\{ \begin{array}{l} \text{decision problems that are decided by some nondeterministic machine} \\ \text{running in time } \mathcal{O}(n^d) \text{ for any input of size } n \text{ and any computation} \\ \text{path, for some } d \in \mathbb{N} \end{array} \right\}$$

The definition is as usual insensitive to the choice of the computational model among the usual choices: Turing machine, Python machine, etc.

7.2.2. Second definition: with certificates

We propose a second definition of the class NP, based on the usual deterministic machines.

$$\mathsf{NP} = \left\{ \begin{array}{l} \text{decision problems for which there exists some } d, k \in \mathbb{N} \text{ and a deter-} \\ \text{ministic machine } M(x, y) \text{ running in time in } \mathcal{O}((|x| + |y|)^d) \text{ such that} \\ x \text{ is a YES-instance } \textit{iff } \exists y : |y| \in \mathcal{O}(|x|^k) \text{ for which } M(x, y) \text{ returns} \\ \text{YES} \end{array} \right\}$$

This y is called a certificate (or witness, or proof) for x.

Example 7.1. HAMILTONIAN, the Hamiltonian Graph Problem
Input : A graph G
Output : Yes *iff* G is Hamiltonian (i.e. $\exists$ 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 a Hamiltonian cycle of x.

Since $|y| \in \mathcal{O}(|x|^k)$ and $M(x, y)$ is a polynomial-time algorithm, HAMILTONIAN $\in$ NP.

7.2.3. Comparing the two definitions

Let us prove that the two definitions are equivalent.

Consider a problem A that belongs to NP according to the second definition: there is a two-input polynomial-time deterministic machine M such that an instance x of A is positive iff there is a certificate y of length $f(|x|) \in \mathcal{O}(n^k)$ so that $M(x, y) = YES$. Then A is decided by a polynomial-time non-deterministic machine, which first writes a string y of $f(|x|)$ bits nondeterministically, then runs $M(x, y)$.

Conversely, assume A belongs to NP according to the first definition: it is decided by some non-deterministic machine, running in time $f(|x|) \in \mathcal{O})(|x|^d)$ on input x. Then it makes at most $f(n)$ non-deterministic binary choices. Now simulate this non-deterministic machine with a two-input machine $M(x, y)$, replacing each instruction querying a nondeterministic bit by reading the next bit of y. Thus each y allows to explore a computation path of the nondeterministic machine.

This shows the formal equivalence of the two definitions.

Intuitively, those two definitions translate the intuition that for positive instances, one may prove they are positive by making guesses (under the form of some certificate or non-deterministic queries), and then checking that those guesses are 'right'. E.g. for HAMILTONIAN, one may 'guess' a cycle in the graph and then check it is Hamiltonian indeed. Said otherwise, it easy to *convince* someone that an instance of a problem in NP is indeed a Yes-instance: we just need to provide the certificate and check that it is a certificate indeed. Of course, it does not mean that the certificate is itself easy to find, when we do not know it. Finally, one may say that positive instances of a problem in NP have a *short proof* of their positivity: one just have to write down all the steps performed by $M(x, y)$, for a suitable certificate y.

Many natural and practical relevant combinatorial problems are in NP.

7.2.4. The coNP class

Unlike the class P or PSPACE for instance, the class NP is not symmetric in the way it considers positive and negative instances.

For a decision problem A we can define the *complementary* problem $\overline{A}$, defined as follows: positive instances of $\overline{A}$ are the negative instances of A and conversely.

Clearly, $A \in \mathsf{P}$ iff $\overline{A} \in \mathsf{P}$, and $A \in \mathsf{PSPACE}$ iff $\overline{A} \in \mathsf{PSPACE}$: we say that P and PSPACE are *closed under complementation.*

We define the class coNP as

$\mathsf{coNP} = \{\text{problems whose complementary is in NP }\}$

7.2.5. Comparing with other complexity classes

Theorem 7.5. $\mathsf{P} \subseteq \mathsf{NP}$

Proof. It is trivial from the first definition of NP as those problems which are decided by a nondeterministic machine in polynomial time. Indeed a deterministic machine maybe con-

sidered as a special sort of nondeterministic machine, that nevers uses the nondeterministic bit instruction. □

For the same reason:

Theorem 7.6. $\mathsf{P} \subseteq \mathsf{coNP}$

and thus:

Theorem 7.7. $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$

We do not know if $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \subsetneq \mathsf{NP}$. Experts believe that $\mathsf{P} \subsetneq \mathsf{NP}$. It is one of the most important conjectures, not just in Theoretical Computer Science, but in mathematics!

We don't even know if $\mathsf{P} = \mathsf{NP} \cap \mathsf{coNP}$.

We can also relate NP to space-complexity classes:

Theorem 7.8. $\mathsf{NP} \subseteq \mathsf{PSPACE}$

Proof. For a problem in NP s.t. x is a Yes-Instance iff $\exists y : |y| \leq f(|x|) \in \mathcal{O}(|x|^k)$ such that $M(x, y)$ outputs Yes, where $M(x, y)$ runs in $\mathcal{O}(|x| + |y|^d)$ time.

Consider the algorithm

Code 7.2: algorithm checking all candidates for a certificate

```
1    for y in range(1,2,3...,2^{f(|x|)}):
2        if M(x,y) == YES:
3          return YES
4    Return NO #We tried all possible y and found no certificate
```

This solves the problem in polynomial space, since for each y we can wipe out the memory used by each computation of $M(x, y) = NO$ to make room for the next value of y, while $M(x, y)$ uses polynomial-space $\forall |y| \leq |x|^d$. □

Remark. For example, the Hamiltonian Graph problem is in PSPACE because we can simply write a program that generates all the possible cycles and verify if one of them is a Hamiltonian cycle. This program just keep the current candidate cycle and graph in memory, so the memory required is polynomial indeed.

Since PSPACE is closed under complementation we also have

Theorem 7.9. $\mathsf{coNP} \subseteq \mathsf{PSPACE}$

In summary we have proved:

Theorem 7.10. $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$

While we know $\mathsf{P} \neq \mathsf{EXPTIME}$, we don't know if any of the other inclusions are proper, although it is widely believed so.

7.3. Reducibility and NP-complete problems

For two problems S and T, we want to say if S is *"easier"* than T. We say a decision problem S is *reducible* to another decision problem T, written $S \leq_p T$, if $\exists f : \mathbb{N} \leftarrow \mathbb{N}$, 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. Thus any algorithm known to decide T can also be used to solve S.

If two problems are mutually reducible, $S \leq_p T$ and $T \leq_p S$, then those problems are considered equivalent (up to polynomial-time reduction), and we write $S \equiv_p T$

Example 7.2. 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 $\leq_p$ INDEPENDENT-SET $\leq_p$ CLIQUE. 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($\bar{G}$).

Overall we can write CLIQUE $\equiv_p$ INDEPENDENT-SET. In particular, if one is in P then so is the other.

The polynomial time reduction $\leq_p$ induces a partial order on decision problems. We may now look for the properties of each class of interest, for example NP, as a partially ordered set.

Theorem 7.11. There is a problem $S \in \mathsf{NP}$ such that $\forall T \in \mathsf{NP} : T \leq_p S$

This is not trivial, we could have thought a priori that several incomparable maximal elements exist in NP (a maximal element in NP being an element not smaller than any other in NP). More importantly, there are many such problems, which can be explictly described, and are of great importance both as practical and theoretical problems, as we now see.

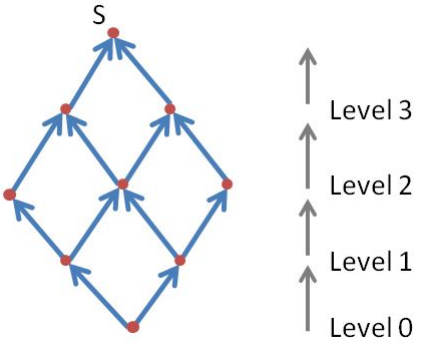


Figure 7.1.: A partial order with a unique maximal element.

7.3.1. The SAT Problem

Let us define SAT, the problem of satisfiability of a Boolean formula.

Definition 7.2. SAT :

- 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 the formula is satisfiable, i.e. there exist truth value for the variables so that the formula evaluates to true, for instance
  - $(P \vee \neg\, Q) \rightarrow$ Yes, for P= true
  - $Q \wedge \neg\, Q \rightarrow$ No

A Boolean formula can also be equivalently represented by a Boolean circuit, composed of OR, AND and NOT gates. The inputs of the circuit correspond to the truth value of the variables $P, Q, \ldots$ and there is a single output, which is the truth value of the formula. So SAT can also be described as the problem of checking whether a given Boolean circuit (the instance of the problem) can deliver a TRUE as output (positive instance), or always delivers a FALSE for whatever inputs (negative instance).

Theorem 7.12. (Cook-Levin 1971) $\underbrace{\text{SAT} \in \mathsf{NP} \;\; \textit{and} \;\; \underbrace{\forall T \in \mathsf{NP} : T \leq_p \text{SAT}}_{\text{"SAT is NP-hard"}}}_{\text{"SAT is NP-complete"}}$

Proof. (Sketch)

- SAT $\in$ NP: The certificate for a Boolean formula that is satisfiable list of TRUE/-FALSE values for all variables that make a formula evaluated to YES: the certificate is of polynomial size and can clearly be checked in polynomial time.
- $\forall T \in NP : T \leq_p$ SAT:
  Let T be in NP, then it exists a Turing Machine M such that $M(x, y) = YES$ for some poly-length certificate y, running in poly-time, if and only if x is a YES-instance of T.

  We detail the Turing Machine in Fig. 7.2.

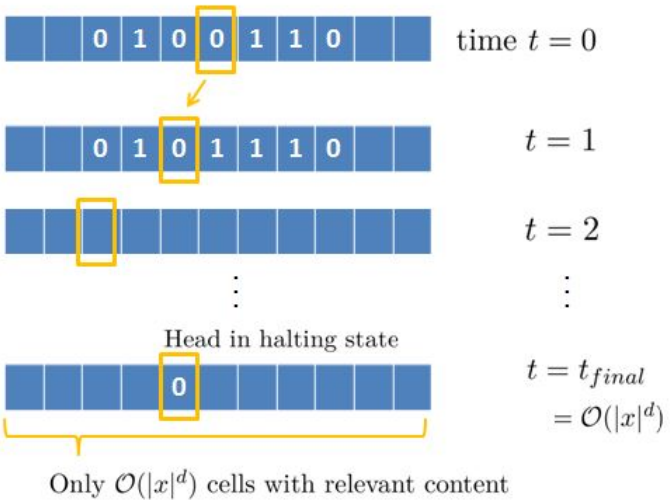


Figure 7.2.: Turing Machine

We create

- $\mathcal{O}\left(|x|^{2d}\right)$ variables $P_{t,m,b}$=TRUE if cell m at time t is b (for b a symbol 0 or 1);
- $\mathcal{O}\left(|x|^{d}\right)$ Boolean variables describing the head state $Q_{t,s}$=TRUE if and only if the head is in state s at time t (remember the state of a Turing machine head is akin to the code line number it's currently reading in its list of instructions);
- $\mathcal{O}\left(|x|^{d}\right)$ variables $R_{t,m}$=TRUE if and only if the head is reading cell m at time t.

.

We can describe the rules of the Turing Machine by a Boolean formula: for example assume the head in state s_1 transitions to s_2 every time it reads a 0, then shifts to the left. Then we can express this with the sentence " if Q_{t,s_1} and $P_{t,m,0}$ and $R_{t,m}$ then P_{t+1,s_2} and $R_{t,m-1}$", repeated for every m and t, which creates a formula of length $\mathcal{O}(|x|^{2d})$. Remember that a sentence "If A then B" is equivalent to $\neg A \vee B$ We can also add consistency constraints such as "$\neg(P_{t,m,0} \wedge P_{t,m,1}$" (one cannot have a zero and a one at the same time at the same place).

We can also describe the input as the content of the tape at $t = 0$, with the variables $P_{0,m,b}$.

The result of the whole computation is encoded into a large, but poly-size (and computed in poly-time), Boolean formula $\phi_{x,y,M}$ meaning "$M(x, y)$ halts and outputs YES".

The question "$\exists ?y : M(x, y)$ halts and outputs YES" (for a given x) amounts to: "Are there truth values for the $P_{0,m,b}$ variables encoding y such that $\phi_{x,y,M}$ is true ?", with x and M which are given and y is to be found. This is a SAT-instance, derived in polynomial time from the instance x of problem T, and satisfiable iff x is a YES-instance of T.

Therefore $T \leq_p SAT$.

□

This theorem proves intuitively that SAT is "the hardest" problem in NP. For instance, if we could prove that SAT is in P then we know that every problem in NP would be in P as well, therefore P = NP.

7.3.2. NP-Complete Problems

To this day, thousands of interesting problems have been proved NP-complete, thus equivalent to SAT. The usual way to prove that a given problem S is NP-complete does not resemble the proof of Cook-Levin's theorem but rather obeys the following strategy:

- prove that $S \in \mathsf{NP}$
- for some other NP-complete problem T, e.g. $T = \text{SAT}$, prove that $T \leq_p S$

One can thus conclude:

- S is NP-complete
- $S \equiv_p T \equiv_p \text{SAT}$

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

One key example is 3-SAT, a restriction of SAT to a subset of instances, hence apparently simpler than SAT, which turns to be equivalent to SAT, thus NP-complete.

Definition 7.3. 3-SAT:

- Instance: A Boolean formula such as $\underbrace{(P \vee \neg Q \vee R)}_{\text{"}clause\text{"}} \wedge (\neg P \vee \neg R \vee S) \wedge (P \vee P \vee Q)$.
  The clauses are formed of 3 (possibly negated) variables linked with $\vee$ (OR), and all the clauses (in arbitrary number) are linked by $\wedge$ (AND).
- Output: YES if satisfiable (i.e. all clauses are true simultaneously for some value of $P, Q, R, S, \cdots$)

Theorem 7.13. 3-SAT is NP-complete

Proof. (Sketch)

- 3-SAT $\in$ NP: clear
- To show that 3-SAT is NP-hard, we prove SAT $\leq_p$ 3-SAT

The reduction proceeds by transforming in poly-time every Boolean formula into the normal form $(\vee \ \ \vee) \wedge (\vee \ \ \vee) \wedge \cdots$ 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
$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$ (de Morgan's law) and more.
Doing so transforms the original formula into a formula of the correct form, except that the clauses are of arbitrary size, instead of comprising 3 variables. We may then add variables to split the clauses into several smaller clauses. For instance $P \vee Q \vee R \vee S$ would be split into $(P \vee Q \vee T) \wedge (\neg T \vee R \vee S) \wedge (\neg R \vee T) \wedge (\neg S \vee T)$. The second clause encodes that "If T then $R \vee S$", the third that "If R then T" and the fourth "If S then T". Overall the last three clauses prove "T iff $R \vee S$". □

3-SAT, because of its simple structure, can help prove that even more problems are NP-complete.

7.3.3. The CLIQUE Problem

We recall that the CLIQUE problem asks whether there is a clique of a given size k in a given graph G.

Theorem 7.14. (Karp, 1972) CLIQUE is NP-complete

Proof.

- CLIQUE $\in$ NP: certificate of a YES-instance (G, k)= list of k nodes of G forming clique
- To show that 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 G and an integer k.

  We illustrate the reduction on an example: For the $3-SAT$ instance $\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4)$ of 4 clauses, the reduction builds the graph of Fig. 7.3, whose nodes are each occurrence of the (possibly negated) variables in the clauses, and the instance of CLIQUE is this graph together with the number $k = 4$ (in general the number of clauses in ϕ). In other words one must decide whether there is a clique of $k = 4$ nodes in the graph.

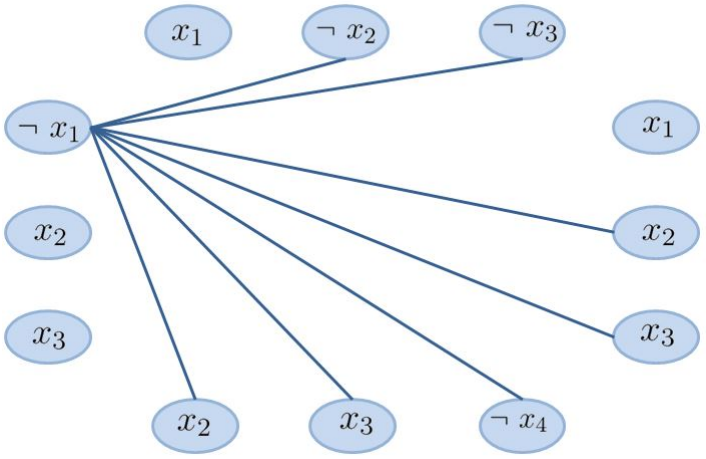


Figure 7.3.: example of graph (not all edges represented) built by the reduction $3-SAT \leq_p CLIQUE$.

All pair of nodes are linked except:

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

We check that if ϕ is satisfiable then it exists a k-clique in the graph. Indeed, assume ϕ satisfiable by for instance:

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

(in general at least one is true in every clause to make the formula true), then the corresponding nodes in the graph form a clique of k nodes, see Fig. .

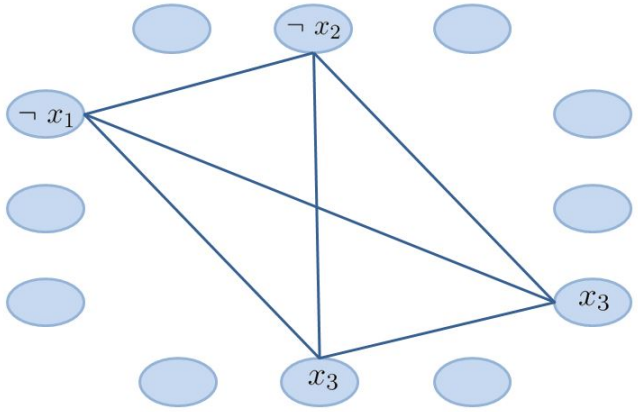


Figure 7.4.: example of clique proving satisfiability of the formula

If there exists a k-CLIQUE in the graph then ϕ is satifiable. Indeed, if we find a k-clique then we can see that it identifies a (possibly negated) variable to be evaluated to TRUE in each clause, forming a consistent truth assignment of the variables (as one cannot have x_i and $\neg x_i$ true simultaneously), thus satisfying ϕ. To take again the example of Fig. :

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

makes ϕ true.

□

Often problems encountered in practical life are optimization problems.

Definition 7.4. MAXCLIQUE:

- Instance: A graph
- Output: Size of the maximum clique

We convert this into the decision problem CLIQUE and we can run CLIQUE for several k (e.g.by dichotomy). This is implicitly what we mean when we say that an optimization problem (Knapsack, Travelling Salesman, etc.) is NP-complete: that the corresponding decision problem of whether a solution at least as good as a certain threshold exists is.

7.3.4. NPC

As we said, thousands of problems have been found NP-complete. All those problems are typically proved NP-complete with the same strategy: First we show that the problem is in NP (by explaining how to prove in a polynomial time that a Yes-Instance is a Yes-Instance indeed) ; then we reduce a known NP-complete problem (such as SAT, CLIQUE, etc.) to it, to show that it is at least as hard as SAT, CLIQUE, etc, thus is NP-hard (i.e. at least as hard as every problem in NP).

The class of all NP-complete problems is denoted NPC.

7.4. Randomized complexity classes

Let us now characterize the classes of problems that are solvable efficiently by randomised algorithms. We thus now adopt a computational model of deterministic (Turing, Python, etc.) machines which are enriched with a 'random bit' instruction. We assume that the sequence of random bits delivered by successive calls of this instruction is guaranteed to be uniformly i.i.d., like a sequence of Heads or Tail obtained by perfectly random coin tossing. As underlined already, this can be seen as an enrichment of the 'nondeterministic bit' instruction with a probabilistic structure.

7.4.1. The RP class and coRP class: problems with an efficient one-sided Monte-Carlo decision algorithm

We start by defining the RP (these letters standing for 'randomised polynomial time') and coRP complexity classes. These classes contain problems efficiently solved by a one-sided Monte Carlo algorithm. Formally the RP class is defined as:

$$\mathsf{RP} = \left\{ \begin{array}{c} \text{decision problems such that } \exists \text{ Python machine } M(x,r): \\ M(x,r) \text{ runs in } O(|x|^d),\ \forall x, \\ x \text{ is a No-instance } \implies M(x,r) \text{ returns always No and} \\ x \text{ is a Yes-instance } \implies \mathbb{P}[M(x,r) \text{ returns Yes}] \geq 1/2 \end{array} \right\}$$

And symmetrically the coRP complexity class is formally defined as the class of problems whose complement is in RP, or more explicitly:

$$\mathsf{coRP} = \left\{ \begin{array}{c} \text{decision problems such that } \exists \text{ Python machine } M(x,r): \\ M(x,r) \text{ runs in } O(|x|^d),\ \forall x, \\ x \text{ is a Yes-instance } \implies M(x,r) \text{ returns always Yes and} \\ x \text{ is a No-instance } \implies \mathbb{P}[M(x,r) \text{ returns No}] \geq 1/2 \end{array} \right\}$$

From the above definitions we infer the following inclusions:

Theorem 7.15.

$$\mathsf{P} \subseteq \mathsf{RP} \subseteq \mathsf{NP}$$

$$\mathsf{P} \subseteq \mathsf{coRP} \subseteq \mathsf{coNP}$$

That P ⊆ RP is clear from the definitions.

That RP ⊆ NP comes from the fact that a random bit instruction can serve as a non-deterministic bit instruction: in that case the (first) definition of NP is identical to the

definition of RP except that the probability of a YES output on a positive instance is only required to be > 0 instead of $\geq 1/2$.

This strictly positive probability may be exponentially small in the input size, e.g. smaller than 2^{-n^k}, thus too small in general to make advantage of stochastic amplification here. For instance if a graph is Hamiltonian then just visiting nodes in a random order in a graph delivers a Hamiltonian cycle with non-zero probability but this is certainly not an efficient procedure as this probability may be very slim.

Equivalently, one may express RP with nondeterministic machines only, as we now develop.

Remember that from a same input, a nondeterministic machine may follow different computation paths, i.e. different sequences of bits obtained from the successive calls nondeterministic bit instructions. Each computation path can be labelled with the final output of the algorithm, YES or NO, when the machine follows that path. Of course we may organise these paths into a binary tree, so that each computation path is now a path from root to a leaf, and label the leaf with the output YES or NO.

A problem is in NP if there is a nondeterministic polynomial-time algorithm such that at least one computation path (one leaf in the tree) ends with output YES for a YES-instance, and all computation paths (all leaves) end with NO for a NO-instance. A problem is in RP if there is a nondeterministic polynomial-time algorithm such that at least *half* of computation paths (half of leaves in the tree) end with output YES for a YES-instance, and all computation paths (all leaves) end with NO for a NO-instance.

Let us finish our comparison of RP vs NP with a the certificate viewpoint. We saw in an earlier section that NP can be described in terms of deterministic machines with two inputs, the second input y being essentially the computation path of the corresponding nondeterministic machine. In other words, we provide once for all to the algorithm a sufficient stock of nondeterministic bits to the algorithm instead of being called by the nondeterministic bit instruction. An input y leading to a YES output is called a certificate for this instance.

Similarly, we can describe a random algorithm operating on an input x, as a a deterministic machine with two inputs x and y, where y has been generated randomly, and is sufficiently long to provide enough random bits to the algorithm whenever needed. In other words we 'externalise' the random bit generator. In this description, a problem is in RP if for some polynomial time machine of this form and each positive instance, at least half of all possible inputs y of suitable length turn out to be certificates (while none is for a negative instance).

7.4.2. Polynomial Identity testing: a problem in coRP

Let us now consider a decision problem in coRP for which we do not know if it is in P. The problem is the polynomial identity testing,or PIT:

Definition 7.5. PIT:

- Instance : A formula of a polynomial, e.g. $(x+y)^2 - x^2 - y^2$. This formula can be described by a circuit with the variables x and y as inputs and with $+$, $-$, $\times$ or multiplication by a scalar as gates, and a single output as on figure 7.5.

- Output: $\begin{cases} \text{Yes} & \text{If the polynomial is identically 0} \\ \text{No} & \text{If not} \end{cases}$

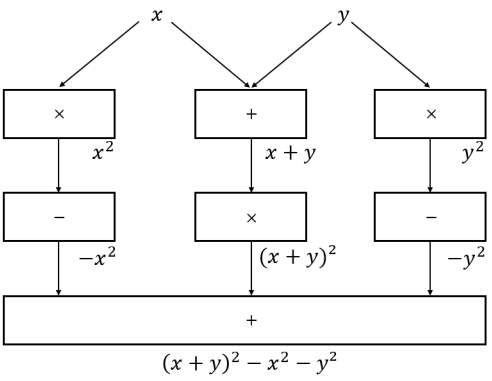


Figure 7.5.: Gate representation of polynomial $(x+y)^2 - x^2 - y^2$.

Let us consider two obvious algorithms for it:

Straightforward algorithm: Develop the sum and check if the coefficients are all 0. Notice that this method will be very inefficient if the number of monomials is exponential. For instance $(x+y)^{1000}$, once developed, has 2^{1000} monomials.

Schwartz-Zippel's random algorithm: For a polynomial P:

- Pick a finite set $S \subseteq \mathbb{R}$ (or $\mathbb{Q}$).
- Assign to every variable a random value $X_i \in S$.
- Evaluate the polynomial at $(X_1, \ldots, X_n)$. If $P(X_1, \ldots, X_n) \neq 0$ then return "NO". If $P(X_1, \ldots, X_n) = 0$ then return "YES (maybe)".

The second one runs in polynomial time as we just have to evaluate the fomula for specific values (see final remarks below however) but we must understand how likely it is that it may produce false positives. The answer is in the following theorem.

Theorem 7.16. (Schwartz-Zippel, 1979-1980) If $P(x_1, x_2, \ldots, x_n)$ is a polynomial of degree d, then the probability that $P(X_1, \ldots, X_n) = 0$ for X_i randomly taken in S $\forall i$, is at most $\frac{d}{|S|}$.

Proof. We proceed by induction on the number of variables of P.

For $n = 1$: An univariate polynomial of degree d has at most d roots. The probability to have $P(X_1) = 0$ is then at most $\frac{d}{|S|}$. Indeed, S contains in the worst case the d roots of P. The probability to select one root at random is then $d/|S|$.

For $n > 1$: We write

$$P(x_1, \ldots, x_n) = P_d(x_1, \ldots, x_{n-1})x_n^0 + \cdots + P_k(x_1, \ldots, x_{n-1})x_n^{d-k} + \cdots + P_0(x_1, \ldots, x_{n-1})x_n^d$$

We observe that each P_k has degree at most k (otherwise $P_k x_n^{d-k}$ would exceed d in degree).

Let i be the lowest k such that P_k is not identically zero. One of these two events will occur:

- $P_i(X_1, \ldots, X_{n-1}) \neq 0$ (i.e. takes a non-zero value at the randomly chosen values $X_1, \ldots, X_{n-1}$). This happens with some probability q. In this case, the polynomial $P(X_1, \ldots, X_{n-1}, x_n)$ is univariate in variable x_n and is of degree $d - i$. Knowing this, the probability for this polynomial to output 0 when evaluated at X_n randomly picked in S is at most $\frac{d-i}{|S|}$ (by the induction hypothesis).
- $P_i(X_1, \ldots, X_{n-1}) = 0$ (i.e. takes the zero values at the randomly chosen values $X_1, \ldots, X_{n-1}$). By the induction hypothesis, this happens with probability at most $\frac{i}{|S|}$ (given that P_i has degree i at most, and $n - 1$ variables). Knowing this, the probability for the whole P to evaluate to zero is r, for some r.

Overall the probability of finding a zero value when evaluating P randomly is bounded as follows:

$$\mathbb{P}[P(X_1, \ldots, X_n) = 0] \leq q\frac{d-i}{|S|} + \frac{i}{|S|}r \leq \frac{d-i}{|S|} + \frac{i}{|S|} = \frac{d}{|S|}.$$

□

If the set S is such that $|S| > d$, the probability that the random algorithm fails a negative instance is less or equal to $d/|S| < 1$. Thus PIT $\in$ coRP. The reasoning can also be adapted in finite fields, e.g. the integers modulo p for some prime number p.

A final remark: evaluating a polynomial at specific integers or rationals will for sure take a polynomial number of elementary operations. But the numbers involved may become very large, so that the cost of these operations cannot be considered constant anymore. For instance $(x + y)^{1000}$ evaluated at $x = 1 = y$ is 1000 bit long. A practical solution is to perform all the operations modulo p for some relatively large random prime p. This caps the cost of elementary operations to $\mathcal{O}(\log^2 p)$. This also increases the possibility of a wrong answer (whenever the value of the polynomial is a nonzero multiple of p, so that it would be correctly assessed as non-zero by the Schartz-Zippel test in $\mathbb{Z}$, but is zero modulo p), but this probability can be controlled to be small enough by the right random choice of p.

Other problems can be recoded into checking a polynomial identity. In section 5.8.1, given matrices A, B, C, we checked whether or not $AB = C$, which is equivalent to $ABx - Cx$ being identically zero, where x is a vector of n variables $x_1, \ldots, x_n$. If $AB \neq C$ and we evaluate each x_i randomly in $S = \{0, 1\}$ and $|S| = 2$, we will find a zero value with probability less or equal to $1/2$ (the maximum degree is 1 and $|S| = 2$).

7.4.3. The BPP class: problems with an efficient Monte Carlo algorithm

Just as RP and coRP are the class of problems that are efficiently solved with a one-sided Monte Carlo algorithm, we can define BPP as the problems that are efficiently solved with a two-sided Monte Carlo algorithm. The name BPP refers to 'bounded error probability polynomial time'. Formally we define:

$$\mathsf{BPP} = \left\{ \begin{array}{c} \text{decision problems such that } \exists \text{ Python machine } M(x, r) : \\ M(x, r) \text{ runs in } O(|x|^d), \text{ and } \forall x, \\ x \text{ is a Yes-instance } \implies \mathbb{P}[M(x, r) \text{ returns Yes}] \geq \frac{2}{3} \text{ and} \\ x \text{ is a No-instance } \implies \mathbb{P}[M(x, r) \text{ returns No}] \geq \frac{2}{3} \end{array} \right\}.$$

Note that the 2/3 could be replaced by any value in $]1/2, 1[$, thanks to the amplification of stochastic advantage.

Trivially, we have $\mathsf{RP} \subseteq \mathsf{BPP}$ and $\mathsf{coRP} \subseteq \mathsf{BPP}$.

As mentioned in the section on RP, we can represent a randomised (Turing, Python, etc.) machine with input x as a deterministic machine $M(x, r)$, where r is a sequence of random bits that is used by the machine every time it needs a random bits. Then a problem in BPP is correctly solved by a polynomial-time algorithm for at least 2/3 of all possible inputs r of appropriate length (polynomial in $|x|$).

The following theorem is proved similarly to $\mathsf{NP} \subseteq \mathsf{PSPACE}$:

Theorem 7.17. $\mathsf{BPP} \subseteq \mathsf{PSPACE}$

Proof. For each instance x of a problem in BPP and for each r $M(x, r)$ runs in polynomial time, therefore it can only access a polynomial-sized memory. We can therefore run $M(x, r)$ for each possible r of appropriate length (polynomial in $|x|$ in any case) and count how many times $M(x, r)$ outputs YES and NO. The algorithm returns the answer that occurs at least 2/3 of the times, which is guaranteed to be the correct answer. This procedure uses polynomial space, as the same memory space can be reused over and over again for every new value of r. □

The big open question is whether $\mathsf{P} = \mathsf{BPP}$ or not. In other words, is randomness useful at all to design efficient algorithm?

The next two theorems give us ways to derandomise problems in BPP.

Theorem 7.18. If $T \in \mathsf{BPP}$ is decidable in the BPP sense using only $\mathcal{O}(\log |x|)$ random bits, then $T \in P$.

Proof. Using the same algorithm as in the proof of $\mathsf{BPP} \subseteq PSPACE$ above works because we now have a polynomial number $2^{\mathcal{O}(\log |x|)}$ of different strings r to try. We have already used this derandomization argument for derandomizing a MAXCUT heuristic in a previous chapter. □

This theorem implies that if there is a perfect pseudo-random generator using a k-bit truly random seed and generating a sequence of $\mathcal{O}(2^k)$ pseudo-random bits that cannot be effectively distinguished from a truly random string of same length (in a sense that can be made precise, which we do not attempt to formalise here) then $\mathsf{P} = \mathsf{BPP}$.

Theorem 7.19. If $M(x, r)$ is a Monte-Carlo algorithm solving instance x of a decision problem with random string r with a probability of failure smaller than $4^{-|x|}$ then for each fixed n we can find some bit string r_n such that $M(x, r_n)$ (now a deterministic algorithm) is correct for all instances x of length n.

Proof. For each x of size n there is at most a fraction 4^{-n} of bit strings r such that $M(x,r)$ returns the incorrect answer. Therefore we can bound the fraction f of all bit strings r which lead to a wrong answer for at least one of the instances of size n: $f \leq 2^n 4^{-n} \leq 2^{-n} < 1$. Therefore there is a string r which gives always a correct result to $A(x,r)$. □

Finding this r_n that brings a correct answer to all instances of size n may be very time consuming of course.

Note that for any problem in BPP, one can reach a probability of error less than 4^{-n} in polynomial time, thanks to stochastic amplification. The theorem thus has the following consequence: If we can afford a costly pre-computation, then we can solve all instances of *fixed size* of a problem in BPP efficiently in a deterministic way. This kind of fixed-length derandomisation can be relevant for instance when building a cryptographic scheme with fixed-length keys or messages.

The concept of fixed-input-length complexity is further explored in a later section on nonuniform complexity classes.

7.4.4. The ZPP class: problems with an efficient Las Vegas algorithm

Now instead of looking the problems that can be solved in polynomial time but with probabilistic result (Monte Carlo algorithms) we take a look at problems that can be solved by an algorithm runnign in polynomial time *in expectation* (possibly much longer in the worst case), returning always the a correct result (Las Vegas algorithms).

We define:

$$\mathsf{ZPP} = \left\{ \begin{array}{l} \text{decision problems such that } \exists \text{ Python machine } M(x): \\ \quad \text{Running time } r(x) \text{ is such that } \mathbb{E}(r(x)) = O(|x|^d) \end{array} \right\}$$

The name ZPP stands for 'zero error probability polynomial time'.

The next theorem provides a satisfactory characterisation of ZPP.

Theorem 7.20. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$.

Proof. First let us prove $\mathsf{ZPP} \subseteq \mathsf{RP}$, it will also prove $\mathsf{ZPP} \subseteq \mathsf{coRP}$. Let us consider a problem in ZPP that runs in expected time $f(x) = \mathbb{E}(t(x))$ with an algorithm **A**.

Markov's inequality states that $\mathbb{P}[t(x) \geq 2f(x)] \leq \frac{1}{2}$. We define the following algorithm:

Code 7.3: RP algorithm

```
1    while running time <= 2f(x):
2    run A
3    return YES # in case of forced stop
```

By the previous assumption, we see that this algorithm runs in polynomial time and the probability of a false positive is smaller than $\frac{1}{2}$.

Now let us consider a problem in $\mathsf{RP} \cap \mathsf{coRP}$ for which we have respective algorithms **A** and **B**. We define the following algorithm **Z**:

Code 7.4: ZPP algorithm

```
1       (y,z)=(A(x), B(x))
2       if (y,z)=(NO,NO):
3       return NO
4       if (y,z)=(YES,YES):
5       return YES
6       if (y,z)=(NO,YES):
7       repeat from step 1
```

Let us observe that (YES, NO) never happens because it would be a instance which is positive and negative at the same time. Since the probability of (NO, YES) is smaller than $\frac{1}{2}$, $\mathbb{E}[\text{number of repeats}] \leq 2$.

Therefore $\mathbb{E}[\text{running time of } \mathbf{Z}] \leq 2(\text{running time of } \mathbf{A} + \text{running time of } \mathbf{B})$. □

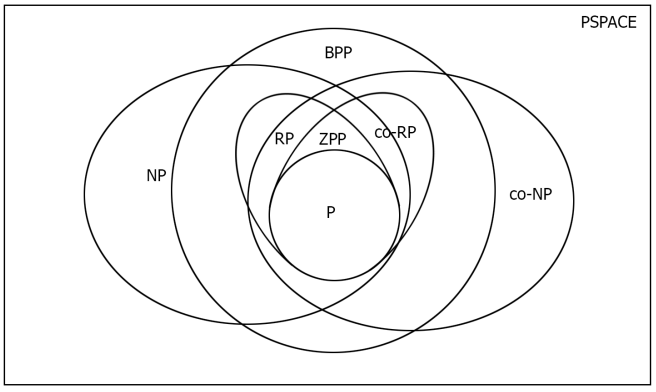


Figure 7.6.: Summary of complexity classes

7.5. Conclusions

We have seen a wild variety of complexity classes. Even though P is often described as the class of 'efficiently solvable' problems, we see that BPP (containing P, RP, coRP and ZPP) is also an excellent candidate for this role, provided that we have a good (pseudo)random generator. It may very well be that BPP = P, and we will see even more reasons to think so in the next chapter.

We have seen also that even though we can separate complexity resources based on the same resource (time or space), e.g. P ≠ EXPTIME thanks to diagonal arguments, most other containments of complexity are not known to be strict (e.g. P ⊆ NP).

8. Nonuniform complexity classes and circuit complexity

In this chapter we look at models of computation that focus on algorithms for each instance size n separately. These are called *nonuniform* computation models because algorithms for different values of n may be very different indeed.

In this context we look at two ways to define 'efficient algorithm' for a given input size. These ways ultimately prove equivalent, and regard the size of the smallest Boolean circuit that solves instances of a given size.

8.1. Machines with advice

In this chapter, we introduce new classes of decision problems:

$$\mathsf{P/f} = \left\{ \begin{array}{l} \text{decision problems that can be decided by the output some machine} \\ M(x, r_n) \text{ whose running time is polynomial in } n, \text{ when receiving an} \\ \text{'advice' } r_n \text{ of length} \leq f(n) \text{ for all instances } x \text{ of size } n. \end{array} \right\}$$

Here $f : \mathbb{N} \to \mathbb{N}$ is a function of n. Such complexity classes are called 'nonuniform' because for each size n we may have a completely different algorithm $x \mapsto M(x, r_n)$ deciding all instances of size n. There is no promise that the advice r_n is easy to find. It may even be a non-computable function of n. This results in complexity classes that contain undecidable problems! As a result, even $\mathsf{P/1}$ (only one bit of advice for every input size) contains undecidable problems. For instance, for any undecidable set $A \subset \mathbb{N}$, the decision problem $x \mapsto YES$ iff $|x| \in A$ is undecidable but in $\mathsf{P/1}$ (here the advice for each n is 1 if $n \in A$, 0 otherwise). The practical and theoretical relevance of such complexity classes that mix very simple and very hard problems is discussed further below.

It is clear that if the advice is too long (more than polynomial in n) then it cannot even be accessed by a Turing machine in time polynomial in n. Thus our main object of interest is the class of problems that can be decided in polynomial time with a polynomial-long advice:

$$\mathsf{P/poly} = \cup_{c,d \geq 0} \mathsf{P}/(cn^d) = \left\{ \begin{array}{l} \text{decision problems decided by Turing machines in polynomial} \\ \text{time with polynomial length advice} \end{array} \right\}$$

Although this definition is sometimes convenient to work with, it is not extremely intuive. An equivalent characterization is given from the viewpoint of Boolean circuit, as we now describe.

8.2. Circuit complexity

A Boolean circuit is composed of Boolean gates interconnected as nodes on an acyclic graph. There are two types of gates:

- 2 bits to 1 bit: "AND", "OR", "=","XOR", ... and all 2-to-1 boolean functions.
- 1 bit to 1 bit: "NOT"

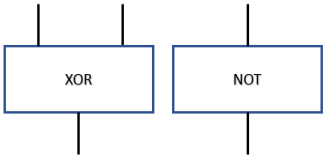


Figure 8.1.: example of 2 bits to 1 bit gate (left) and 1 bit to 1 bit gate (right)

For each fixed instance size n of a decision problem, we may want to design a Boolean circuit with n inputs (reading the n bits of the instance), and one output (0 for a NO-instance, 1 for a YES-instance) solving the decision problem, i.e. computing the Boolean function $\{0,1\}^n \to \{0,1\}$ mapping each instance to its truth value.

For a given decision problem, we may want to ask for the size of the smallest possible circuit deciding all instances of size n. Here we measure the *size* as the number of gates. Another relevant measure of complexity would be the depth, i.e. the length (number of gates traversed) of longest input-to-output path. The smallest size of a circuit solving all n-bit instances, as a function of n, is called the *circuit complexity* of the decision problem.

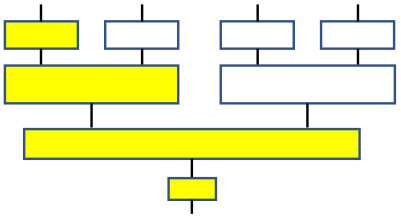


Figure 8.2.: Example: the depth of this circuit is 4, its size is 8.

Note that in our definition, any 2-to-1 gate is considered here as a single gate. We could consider a more restricted family such as the more familiar AND and OR gates. We could even restrict ourselves to the sole NAND gate, which allows to simulate the behaviour of any other gate (even NOT). This is a simple recoding exercice leading to a constant factor in the number of gates of the circuit, inessential to the theory.

One may also simulate unbounded number of inputs, such as multiple-input AND or XOR, with 2-input gates.

Example 8.1. Some unbounded-input gates:

- the standard circuit computing the "AND" of n inputs with 2-input gates has size $n-1$ and depth $\log(n)$;

- the standard circuit computing the "XOR" (sum modulo 2) of n inputs with 2-input gates has size $n-1$ and depth $\log(n)$.

Theorem 8.1. All decision problems (even the undecidable ones) have circuit complexity at most $(2n+1)2^n-1$, i.e. are solvable with $(2n+1)2^n-1$ gates for instances of size n.

Proof. We show that any Boolean function $A : \{0,1\}^n \to \{0,1\}$ can be computed with a circuit of $n2^n$ gates. The idea of the circuit is to build a 'look-up table' by comparing the input x of the circuit in parallel with every possible $y \in \{0,1\}^n$. In case $x = y$, then the circuit outputs $A(y)$. We build a circuit with inputs $x_1, \ldots, x_n$ and $(n+1)2^n$ constant inputs listing $(y, A(y))$ for all y of length n. See Figure for an illustration.

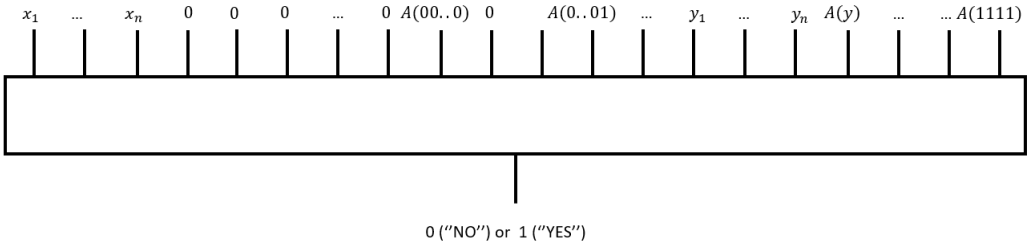


Figure 8.3.: Circuit testing $x = y$ for each y, and reading $A(y)$

The subcircuit that tests $x = y$ and then outputs $A(y)$ if so (and zero otherwise) is drawn at figure 8.4. The n-input "AND" gate, whose output is 1 iff $x = y$, can also be expanded into a binary tree of $n-1$ 2-input gates. With the final AND gate (typo on the figure), this makes $2n$ gates. The circuit comprises 2^n parallel copies of this subcircuit, testing $x = y$ for each y. A final OR gate combines all 2^n outputs to the final output of the ciruits. This 2^n-to-one OR gate can be simulated with $2^n - 1$ gates.

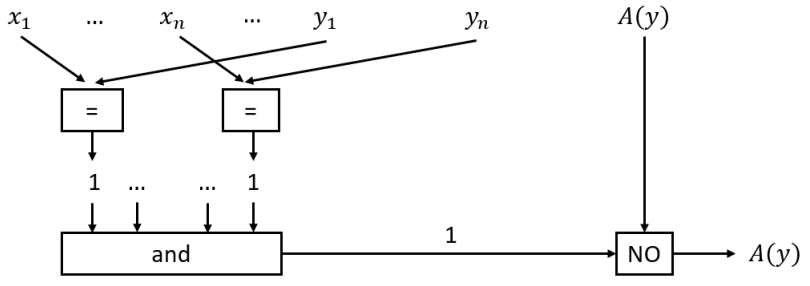


Figure 8.4.: Expanded circuit testing $x = y$ for each y, and reading $A(y)$

We need $n2^n$ gates for the whole circuit. Note that this circuit has $n2^n$ constant inputs (in addition to the inputs $x_1, \ldots, x_n$). The circuit can now be simplified by removing those constant inputs and replace the 2-input gates they enter with 1-input gates or constants

which can be further propagated. For instance x_i AND 1 is just x_i, and x_j XOR 1 is NOT x_j, x_k AND 0 is 0, etc. □

Note that this construction can be improved to show that every Boolean function can be build with $\frac{2^n}{n}(1+o(1))$ gates (Lupanov, 1958).

We now show by a counting argument that there are (many) problems close to this maximal complexity.

Theorem 8.2. (Shannon, 1949) There is at least one problem (and in fact many) of circuit complexity at least $2^n/10n$.

Proof. There exists 2^{2^n} boolean functions $\{0,1\}^n \to \{0,1\}$.

Let $F(t,n)$ denotes the number of different Boolean circuits with s gates and n inputs that can be built.

Our argument is to show that $F(s,n) < 2^{2^n}$ for $s = \frac{2^n}{10n}$, i.e. there are strictly more decision problems (Boolean functions) than Boolean circuits of size $\frac{2^n}{10n}$ and therefore, at least one problem cannot be decided with a such circuit.

Let us estimate $F(s,n)$: Given a s-gate circuit, let us number these gates $1, ..., s$. Each of the 2 inputs of a gate can be:

- an input x_1 or x_2 ... or x_n
- "0" or "1"
- the output of another gate j (except for the last gate)

Therefore the number of possible pair of inputs is at most $(n+s+1)^2$. The number of different 2-to-1 gates is $16 = 2^{2^2}$ (two possible output values for each of the four possible input pair 00, 01, 10 and 11). An upper bound on $F(s,n)$, counting all possible ways to choose s gates and interconnect them, can be written as follows:

$$F(s,n) \leq \left(16(s+n+1)^2\right)^s$$

It is only an upper bound. For example this counts many invalids circuits (e.g. with cycles) and counts $s!$ times the same circuit (with labels of the gates permuted).

Let us check now for which value of s we have $F(s,n) < 2^{2^n}$:

$$\left(16(s+n-1)^2\right)^t < 2^{2^n}$$
$$\Leftrightarrow 4s + 2s\log(s+n-1) < 2^n$$

This latter inequality is true for $s = \frac{2^n}{10n}$, for all n. Indeed, we have:

$$\frac{4 \cdot 2^n}{3n} + \frac{2^n}{5n}\log\left(\frac{2^n}{10n} + n - 1\right) \approx \frac{2^n}{5} < 2^n$$

□

We can improve this bound (essentially by incorporating the $s!$ factor discussed in the proof and using Stirling's approximation to handle it) to show that most Boolean functions need at least $\frac{2^n}{n}(1 - o(1))$ gates. Together with Lupanov's bound mentioned above, this shows that 'most' problems have circuit complexity close to $2^n/n$.

Yet we do not know any *explicit* problem needing exponentially many gates. In fact the only lower bounds we know for specific Boolean functions are linear in n.

In this framework of circuit complexity, we are interested in the class of problems decided by polynomial-size circuits. This happens to coincide with the class $\mathsf{P/poly}$, defined above in term of Turing machines with advice.

Theorem 8.3. A decision problem in $\mathsf{P/poly}$ iff it is decidable with Boolean circuits of polynomial size.

Proof. $\Rightarrow$: Assume the Turing Machine $M(x, r_n)$ decides every instance x of length n, with advice r_n. Then we simulate the computation of the Turing Machine on (x, r_n) with a Boolean circuit in the same way as in the proof of Cook-Levin's theorem ('SAT is NP-complete). From Cook-Levin'proof, we know that the statement '$M(x, r_n)$ outputs YES' can be recoded into a Boolean formula whose size is polynomial in the number of time steps t of the Turing Machine, and whose variables are the successive bits of (x, r_n). This formula can in turn be converted into a Boolean circuit of size polynomial in t (itself polynomial in n) in a systematic way.

This circuit has some fixed constant input r_n, along with the variable input x. Those constant inputs can be simplified, as seen in proof of Theorem 8.1: a two-gate input with one constant input can be reduced to a one gate input (NOT or identity) or a constant, which can be further propagated. In this way we obtain a circuit whose size is polynomial in n, deciding all instances of size n.

For the record (and complement of information) let us mention that the number of gates is in $\mathcal{O}(t^2)$ (quadratic in the computation time t). Recall that the computation of a Turing machine on our instance can be encoded by a Boolean formula with variables x, y:

$$\begin{cases} X_{s,k,a} = \text{TRUE} & \text{if cell } s \text{ at time } k \text{ has symbol a} \\ Y_{i,k} = \text{TRUE} & \text{if the head is reading instruction } i \text{ at time } k \end{cases}$$

where the inputs are the encoded by $X_{s,0,1}$ and $X_{s,0,1}$. Each of these Boolean variables will turn out to be the output of a gate of a circuit.

$\Leftarrow$: If all instances of size n are decided by a circuit C_n of s gates, then we can build a Turing machine (or Python machine) $M(x, r_n)$ where r_n is a description of the circuit C_n, that simulates the action of the circuit C_n on input x of length n. Since there are less than $(16(s + n - 1)^2)^s$ (see proof of Theorem 8.2), the description length of such a circuit is in $\mathcal{O}(s \log s)$. The simulation can clearly be done in time polynomial in s.

In conclusion, if s is polynomial in n, then M is a Turing machine running in time polynomial in n with advice polynomial in n. □

8.3. Links between uniform and nonuniform classes

One has a number of interesting relationships between P/poly and uniform classes.

Problems in P are those problems solved in polynomial time with zero bits of advice, thus:

Theorem 8.4. $\mathsf{P} \subseteq \mathsf{P/poly}$

Problems in BPP can be solved by polynomial-size circuits. This is fact a reformulation of the derandomisation procedure mentioned for BPP problems in the previous chapter:

Theorem 8.5. (Adleman 1992) $\mathsf{BPP} \subseteq \mathsf{P/poly}$

Proof. We repeat here the 'nonuniform' derandomisation argument. For a problem in BPP, there is a deterministic algorithm $M(x, r)$ which solves the problem for any instance x and a random string r (of polynomial length) with a very small probability of error, for example $4^{-|x|}$ (such a low probability can always be obtained thanks to amplification of stochastic advantage). The total fraction of strings leading to some error for some n-bit input is less than $2^{|x|}4^{-|x|} \ll 1$.

Thus, we can choose a fixed string r_n such that $M(x, r_n)$ provides the correctness with probability 1 for all x of the size n: this strings serves as polynomial-length advice. □

What about NP? We don't know if it is included in P/poly or not. If $\mathsf{NP} \subsetneq \mathsf{P/poly}$ then the Venn diagram for NP, P and P/poly set of problems can be seen at Figure 8.5.

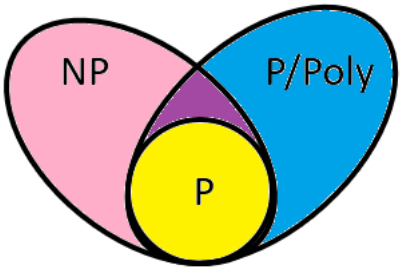


Figure 8.5.: Venn diagram for NP, P and P/poly set of problems (if $\mathsf{NP} \subsetneq \mathsf{P/poly}$)

One way to prove that $\mathsf{P} \neq \mathsf{NP}$ is to find a problem in NP but not in P/poly, in other words a problem in NP with superpolynomial circuit sizes. As previously mentioned, lower bounds on the size of circuits are surprisingly hard to derive, and so far one could only prove $\Omega(n)$ bound on a circuit size for a few problems. On the positive, we know that for instance solving the CLIQUE problem ($\exists$ a clique of k nodes in a graph G?) require exponentially many AND and OR gates (in circuits only allowing AND or OR gates, in particular not allowing the NOT gate).

Finally, let us state a deep and surprising derandomization theorem, relating the question $\mathsf{P} = \mathsf{BPP}$ to the existence of small circuits for E (i.e. the class of decision problems which are decidable in $2^{\mathcal{O}(n)}$ time; it is a proper subclass of $\mathsf{EXPTIME}$) :

Theorem 8.6. (Impagliazzo and Wigderson 1997) If there is a problem in E which requires circuits in $2^{\Omega(n)}$ then $\mathsf{P} = \mathsf{BPP}$.

In other words, this theorem states that $\mathsf{BPP} = P$ unless all problem in E have subexponential circuits. This is a win-win situation: either all problems in E have subexponential circuits (good news!) or $\mathsf{P} = \mathsf{BPP}$ (good news!).

8.4. Conclusion: relevance of uniform vs nonuniform models of computation

A computer, as a physical object, can be modelled in various ways that enlighten us on what we can and cannot hope to achieve with it. The Turing (or Python) machine is one idealisation that assumes unbounded memory and thus ability to treat arbitrarily large inputs. The Boolean circuit model is another idealisation, considering the computer as managing a finite, bounded number of bits. As usual in Science, all models are wrong but some are sometimes useful.

Although in practice a computing device is obviously always finite, and any instance of any problem we want to solve in practice is bounded in size, these bounds can still be relatively large, so the that Turing machine model is more relevant in practice. Especially relevant are the negative conclusions of complexity theory: even with unbounded resources, the Halting problem cannot possibly be decided, for instance. On the other hand, a problem proved to be in P (thus considered to be 'efficiently solvable' in principle) might still have hidden constants so large that it is intractable for even moderate-size instances.

Some problems are always solved on constant-size instances, e.g. in embedded computing, making the Boolean circuit complexity theory potentially relevant for them. E.g. imagine a cryprg

$\mathsf{P/poly}$ is practically relevant if the instances are of fixed (or bounded) size. For instance, in cryptography: if breaking the code (e.g. finding a private key from a public key) is a problem that is NP-hard or even undecidable but in $\mathsf{P/poly}$ then a protocol with fixed key length is potentially weak. In general if it is acceptable to spend—once for all—a possibly large precomputation effort to design the correct circuit that solves each instance of a given size fast, then $\mathsf{P/poly}$ has potential practically relevance.

The relevance is also theoretical, as we have seen nontrivial relationships with the uniform classes. For instance it offers perspectives—unsuccessful so far—to prove that $\mathsf{P} \neq \mathsf{NP}$, by looking for a problem in NP that would require superpolynomial circuit sizes.

9. Quantum computing

9.1. Introduction

We first define what quantum computing is, then we present a few examples and algorithms, that can be run on quantum computers, that are not so easy to run on classical machines. We will also see how quantum computers can beat standard classical computing for some very specific problems, as Turing Machines rely on classical intuitions. One might wonder, after the discovery of the quantum nature of the universe, whether taking this quantum way of thinking into consideration can help quantum computing beat Turing Machines. The answer is: quantum computing does not alter the notion of decidability. For instance we see that a quantum computer we cannot possibly solve the HALTING problem. Why is that? The answer lies in the formulation of the equations (example: Schrodinger's equation). Quantum equations can be simulated on a classical computer (although it might be highly time consuming) and so any quantum computer can be simulated on a classical one. Therefore, there is no new added notion of decidability.

We might still wonder if we can break complexity barriers or compute (fundamentally) faster with quantum machines? We will see that the answer to that question is yes and we show that by demonstrating how we can use quantum physics equations to embed computations.
What is a quantum computer? 2 approaches:

- quantum Turing machine;
- quantum (Boolean) circuit, composed of quantum gates

Both approaches exist in the literature only as concepts but the most developed (common) one is the one based on quantum circuits and gates, it is also the one we consider here.

9.2. A classical example: the NOT gate as a permutation matrix

Let's start with a classical example of gate and see how we can generalize that to the quantum domain.

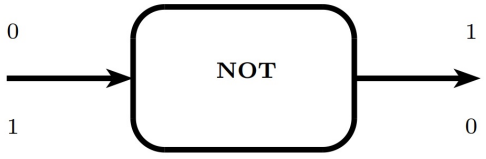


Figure 9.1.: Classical NOT gate.

In quantum computing literature, a 0 bit is denoted as $|0\rangle$ and a 1 bit is denoted as

$|1\rangle$. They can also be seen as binary vectors (essentially a 'one-hot' encoding, to borrow Machine Learning's vocabulary)
Notation:

- $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
- $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

In this vector notation, the NOT gate can be seen as a permutation matrix:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

For example:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

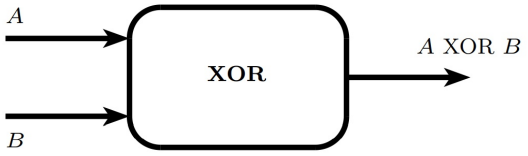


Figure 9.2.: Quantum NOT gate.

9.3. One qubit

If the memory follows the axioms of quantum physics, the bit can be in state 0, 1 or in linear combination, i.e. a quantum state written as:

$$\alpha_0 \, |0\rangle \, + \, \alpha_1 \, |1\rangle \, ,$$

where $\alpha_0, \alpha_1 \in \mathbb{C}$ such that $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

Example: the spin of an electron can take the following values:

- $|\uparrow\rangle$ = "up"
- $|\downarrow\rangle$ = "down"
- $\alpha_0 \, |\uparrow\rangle + \alpha_1 \, |\downarrow\rangle$

Other example: a photon (or electron or any particle)'s can go left, or right or a superposition (i.e. the sign of the momentum along that direction can be positive or negative). Other example arising from Schrödinger's cat's famous thought experiment: if quantum physics can apply to the macroscopic world, we can conceive that a cat can be both alive and dead at the same time.

Vector representation:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \text{ and } |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \text{ such that } \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{pmatrix}\alpha_0\\\alpha_1\end{pmatrix} \in \mathbb{C}^2$$

Explanation of the notation: "$|0\rangle$" is for column vectors and "$\langle 0|$" is for the row vector $\begin{pmatrix}1 & 0\end{pmatrix}$.

Notation for the scalar product:

$$\langle 0|0\rangle = (\langle 0|).(|0\rangle) = \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 1$$

In general:

$$\langle a|b\rangle = \text{ scalar product of } |a\rangle \text{ and } |b\rangle,$$

and $\langle a| = |a\rangle *$ where $|b\rangle *$ is the conjugate transpose (take the transpose and then the conjugate). This is Dirac's "bra" and "ket" notation (the left part is called the "bra" and the right one is the "ket", put together, they form a "bracket").

The definition of measurement in elementary quantum physics leads to a random outcome defined by "projecting" the quantum state onto a classical state. This means that if I measure $\begin{pmatrix}\alpha_0\\\alpha_1\end{pmatrix} = \alpha_0 |0\rangle + \alpha_1 |1\rangle$, I get $|0\rangle$ with probability $|\alpha_0|^2$ or $|1\rangle$ with probability $|\alpha_1|^2$.

So when I measure, I destroy the superposition in a random way, by projecting either a 0 or a 1. The "classical states" can be seen as an orthonormal basis for the set of all possible states.

In summary, we have defined the **qubit** (= quantum bit), as a vector $\begin{pmatrix}\alpha_0\\\alpha_1\end{pmatrix}$ i.e. a superposition $\alpha_0 |0\rangle + \alpha_1 |1\rangle$, with $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

The question now is: how do we compute with a qubit?

9.3.1. Evolution matrices for quantum systems and one-qubit gates

We now study the time evolution of the state of a quantum system, in this case the memory of a quantum computer.

Assume we want to compute the evolution of a quantum system in state $|\Psi\rangle$ between t and $t+\Delta t$. The laws of quantum mechanics teach us that if the system is isolated (i.e. not interacting with the rest of the world), the evolution is always given by a unitary matrix U as follows:

$$|\Psi(t+\Delta t)\rangle = U |\Psi(t)\rangle.$$

A **unitary matrix** is a complex-valued matrix such that:

$$U^*U = I = UU^*$$

where U^* indicates the transpose conjugate of the matrix U. This means that all the columns of U are of unit norm and orthogonal (for the complex-valued scalar product). An n-by-n matrix U, when applied to vector of $\mathbb{C}^n$, preserves the norm and angles of

vectors. In geometric terms, it means that it is essentially a rotation or a symmetry or a combination thereof.

For a single qubit:

$$\begin{pmatrix}\alpha_0(t+\Delta t)\\ \alpha_1(t+\Delta t)\end{pmatrix} = U\begin{pmatrix}\alpha_0(t)\\ \alpha_1(t)\end{pmatrix}$$

U is therefore a 2-by-2 complex-valued matrix.

Example: as seen previously, a NOT gate is a permutation matrix

$$U = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \text{ and } U|0\rangle = U\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}0\\ 1\end{pmatrix} = |1\rangle$$

Although this is a "classical" gate, it also satisfies the requirements of quantum mechanics and we can therefore have a quantum system implementing a NOT gate, which behaves like a classical gate on classical inputs, but can also receive proper quantum inputs:

$$\text{"NOT"}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} = \begin{pmatrix}\alpha_1\\ \alpha_0\end{pmatrix}$$

Example:Hadamard's gate that is used to create superpositions:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \text{ where } H^*H = HH^* = I = H^TH = HH^T$$

H is a real-valued matrix, and it is therefore orthogonal.

$$H\begin{pmatrix}1\\ 0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\ 1\end{pmatrix}$$

$$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Note that $H = H^T$, thus $H^2 = I$: it is a square root of the identity, just like the NOT gate.

9.4. Two qubits and more

We now see how to encode 2-qubits. Classically, a 2-bit memory we can be in 4 states:

$$|00\rangle$$
$$|01\rangle$$
$$|10\rangle$$
$$|11\rangle$$

The state of a 2-qubit memory denoted $|\Psi\rangle$ is a superposition of the 4 possible classical states:

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

with $\alpha_{ij} \in \mathbb{C}$ satisfying:

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1.$$

Vector notations:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$|11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The notation above can be further expanded for 3 qubits (a vector of 8 entries) etc.

More generally, we are interested to see what happens in quantum mechanics when there are two systems (for instance one with 1-qubit and the other with 2-qubits). In other words, we want to describe the **joint evolution of two quantum systems.**
Suppose we have 2 systems A and B with states $|a\rangle_A$ and $|b\rangle_B$ respectively, that have never interacted with one another.
Notation:

$$|\Psi\rangle_{AB} = |a\rangle_A |b\rangle_B = |a\rangle_A \otimes |b\rangle_B = |ab\rangle$$

Vector notation:
If $|a\rangle$ is a vector of the form $\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}$ and $|b\rangle$ is a vector $\begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$ then $|a\rangle_A |b\rangle_B$ is given by the tensor product (also known as the Kronecker product):

$$\begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} \otimes \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} \triangleq \begin{pmatrix} \alpha_0\beta_0 \\ \alpha_0\beta_1 \\ \alpha_1\beta_0 \\ \alpha_1\beta_1 \end{pmatrix}$$

In general if $|a\rangle \in \mathbb{C}^n$ and $|b\rangle \in \mathbb{C}^m$, then $|a\rangle \otimes |b\rangle \in \mathbb{C}^{nm}$.

Example:

$$\begin{aligned}|01\rangle &= |0\rangle \otimes |1\rangle \\ &= \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \\ &= \begin{pmatrix}0\\1\\0\\0\end{pmatrix}\end{aligned}$$

If A and B are not independent then we have may superposition of those tensor products: $|\Psi\rangle$ is any vector of $\mathbb{C}^{nm}$ of unitary norm.
Example of a perfect superposition:

$$\frac{1}{2}\begin{pmatrix}1\\1\\1\\1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \otimes \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$$

In the example above we have two qubits representing the same state: the superposition of 1-qubit between 0 and 1. We have here two bits in a quantum state (not a classical one) that is not a mixed one (not entangled).
However: $\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\\0\\0\end{pmatrix}$ cannot be expressed as the Kronecker product of two vectors, and is called an **entangled state.** Quantum entanglement is analogous to correlation in the classical state.

Joint evolution of AB:

- If the evolutions are separate (the two systems do not interact), then the matrix describing the evolution of A and B is the Kronecker product of the evolution of A and of the evolution of B.

  $$U_{AB} = U_A \otimes U_B$$

  Therefore given separate states:

  $$U_{AB}(|a\rangle \otimes |b\rangle) = (U_A \otimes U_B)(|a\rangle \otimes |b\rangle) = (U_A\,|a\rangle) \otimes (U_B\,|b\rangle)$$

  This does not create entanglement and the two states remain separated.

- If A and B interact then the nm-by-nm evolution operator U is potentially any unitary matrix. Thus even starting from pure states, the evolution might create an entanglement.

Example: can we find an equivalent of the XOR gate ("exclusive or") ?

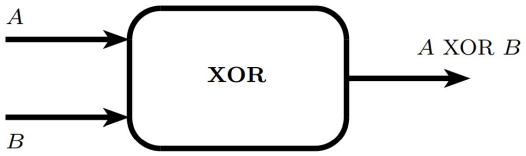


Figure 9.3.: Classical XOR gate.

A XOR gate takes two inputs and gives a single output, for which there is not a quantum equivalent since there is no corresponding square matrix. It is not invertible, and we lose information in the process. Therefore there is no straightforward quantum equivalent to a XOR gate. In order to be able to make a quantum XOR gate, we build a reversible XOR (is also called a CNOT).

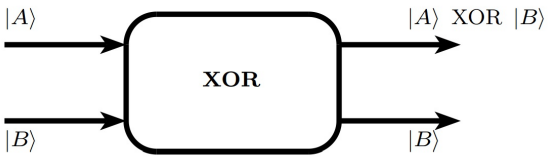


Figure 9.4.: Quantum reversible XOR gate.

While the construct defined above is invertible (we can retrieve the input from the output), can we conclude that the corresponding input-to-output matrix U is unitary? Yes, because a reversible gate will be represented with a permutation matrix U on classical input, and all permutation matrices are unitary. Thus any classical reversible gate can in principle be extended to quantum inputs.

9.5. Extending any (reversible) classical circuit to a quantum circuit

The question is can we transform any classical Boolean circuit into a quantum one? The answer is yes: **Toffoli's gate (1980)** can simulate any classical gate by playing on the values of the inputs. Using this gate we can (non-trivially) embed every classical Boolean circuit into a reversible classical circuit, using only Toffoli's gates, thus can be extended into a quantum circuit. In summary, any bijection $f : \{0,1\}^n \to \{0,1\}^n$ that can be computed by a classical Boolean circuit of a certain size, can be computed by a reversible classical circuit (using Toffoli gates only) of essentially the same size, thus can be extended to a quantum circuit of the same size.

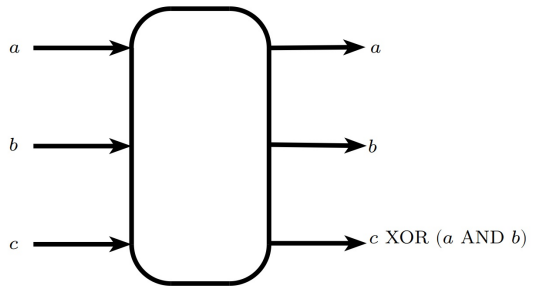


Figure 9.5.: Toffoli's gate.

9.6. The No-Cloning Theorem

Unfortunately not every classical operation has a quantum equivalent. For instance: we cannot copy information, or create duplicates of a quantum state. This is the **no-cloning theorem.**
Let us prove the theorem for a single qubit. Given an initially 'empty' qubit (state $|0\rangle$) , we want to fill it with a copy of qubit $|b\rangle$. In other words we would like to build a 2-qubit system evolving as:

$$|0b\rangle \Longrightarrow |bb\rangle$$

That implies that there would exist a unitary matrix U such that:

$$U\,|0b\rangle = |bb\rangle \quad \forall\ |b\rangle = \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix}$$

Given:

$$|0b\rangle = |0\rangle \otimes |b\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix} = \begin{pmatrix}\beta_0\\ \beta_1\\ 0\\ 0\end{pmatrix}$$ which is the initial state of the 2-qubit memory,

we want to evolve to:

$$|bb\rangle = |b\rangle \otimes |b\rangle = \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix} \otimes \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix} = \begin{pmatrix}\beta_0^2\\ \beta_0\beta_1\\ \beta_1\beta_0\\ \beta_1^2\end{pmatrix}$$

It is immediately clear that this would require a nonlinear transformation, that cannot be delivered by a linear transformation U. We illustrate the requirements on U for several values β:

- if $|b\rangle = \begin{pmatrix}1\\0\end{pmatrix} = |0\rangle$ then $\begin{pmatrix}1\\0\\0\\0\end{pmatrix} \xrightarrow{U} \begin{pmatrix}1\\0\\0\\0\end{pmatrix}$
- if $|b\rangle = \begin{pmatrix}0\\1\end{pmatrix}$ then $\begin{pmatrix}0\\1\\0\\0\end{pmatrix} \xrightarrow{U} \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$

- if $|b\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$ then $\begin{pmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\\0\\0\end{pmatrix} \xrightarrow{U} \begin{pmatrix}\frac{1}{2}\\\frac{1}{2}\\\frac{1}{2}\\\frac{1}{2}\end{pmatrix}$

But the last example is a linear combination of the first two, thus the output must be $\begin{pmatrix}\frac{1}{\sqrt{2}}\\0\\0\\\frac{1}{\sqrt{2}}\end{pmatrix}$, implying a contradiction.

9.7. Grover's algorithm

In this section we see an specifically quantum algorithm that fulfills a task faster than any classical algorithm.

9.7.1. Property checking

Let's suppose that among n-bit strings we are looking for one unique string with some property that can be tested with a quantum circuit. In classical computation, the analogous problem (search algorithm) cannot be solved in less than $\mathcal{O}(N)$ operations. We can show that using quantum computing this problem can be solved with $\mathcal{O}(\sqrt{N})$ operations using the Grover's algorithm.

To define the Grover's search algorithm we must first define how we test the property (i.e. when a string satisfies some condition). If we are looking for a vector $|x\rangle$ such that $|x\rangle = |\omega\rangle$, then we define the unitary operator $U_\omega = |\omega\rangle\langle\omega|$ and the conditions:

- If $U_\omega |x\rangle = |x\rangle$
- If $U_\omega |x\rangle = -|x\rangle$

So, the binary answer is encoded in the sign and we look for the unique classical string $|x\rangle$ such that $U_\omega |x\rangle = -|x\rangle$.

9.7.2. Grover's iterative algorithm

For a set of $N = 2^n$ strings we define:

$$s = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{N-1} |x\rangle = \frac{1}{\sqrt{2^n}} \begin{pmatrix}1\\1\\1\\\vdots\\1\end{pmatrix} = \frac{1}{\sqrt{2^n}}\mathbf{1} \in \mathbb{C}^{2^n},$$

and the unitary operator U_s:

$$U_s = I - 2|s\rangle\langle s|$$

So the final algorithm is:

- We initialize $|\psi_0\rangle = s$;
- we iterate $|\psi_{t+1}\rangle = -U_s U_\omega |\psi_t\rangle$;
- $|\psi_{t_{\text{stop}}}\rangle = |\omega\rangle$ and we measure $|\psi_{t_{max}}\rangle$ and obtain ω with probability approximately 1.

(for where the number of iterations t_{stop} is determined below).

The matrix $-U_s U_\omega$ is also called Grover's diffusion operator, in (imperfect) analogy with classical random walks. It consists in an even number of orthogonal symmetries, thus has determinant one, and amounts to a rotation in the plane spanned by $|s\rangle$ and $|\omega\rangle$. To find the angle of this rotation, we can draw the successive symmetries along $|s\rangle$ and $|\omega\rangle$, or we can compute the scalar product $\langle \psi_{t+1}, \psi(t) | \psi_{t+1}, \psi(t) \rangle$. In any case we find that the rotation is at each step is 2θ, towards $|\omega\rangle$, where $\frac{\pi}{2} - \theta$ is the angle between $|s\rangle$ and $|\omega\rangle$.

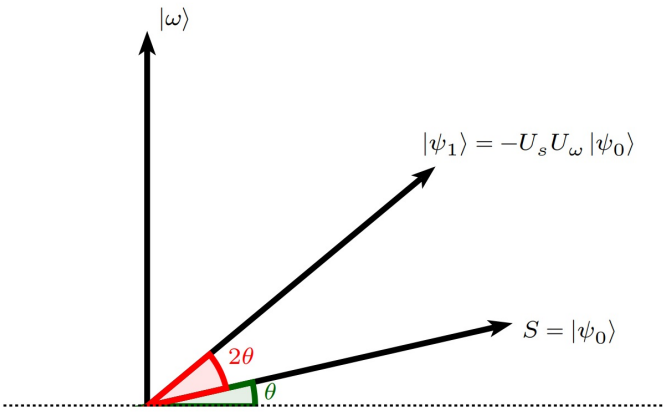


Figure 9.6.: Geometric interpretation of the Grover's algorithm.

Thus the vector $|\psi_t\rangle$ will be (approximately) aligned to $|\omega\rangle$ when $t \approx \frac{\frac{\pi}{2} - \theta}{2\theta}$.

Thus $\cos(\frac{\pi}{2} - \theta) = \sin\theta = \langle \omega, s | \omega, s \rangle = \frac{1}{\sqrt{2^n}}$.

For θ small enough we have $\sin\theta \approx \theta$ and $t_{\text{stop}} = \frac{\frac{\pi}{2}}{2\theta} = \frac{\pi}{4\sqrt{N}} = \mathcal{O}(2^{\frac{n}{2}})$.

This algorithm supposes that there is exactly one input with the desired property, but the algorithm can be adapted if there is more than one string satisfying the property.

9.7.3. Creating the initial state

How do we create the initial state s ? For $n = 1$, one can obtain it from Hadamard's gate:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad \text{and} \quad H \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

More generally, build $H \otimes H$:

$$H \otimes H = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & 1 \end{pmatrix}.$$

which is again called a Hadamard matrix. We can further build the 2^n-by-2^n Hadamard matrix $H \otimes H \otimes \ldots \otimes H$. The first column of this matrix is a column of all identical $1/\sqrt{2^n}$ entries. Applying this operator on the classical state $|00\ldots0\rangle$ (in other words, apply H separately on each qubit$|0\rangle$) thus retrieves $|s\rangle$.

9.7.4. Implementing the property-checking

The way to check the property by applying U_ω, as encoding the truth value of the property as a sign flip, may seem odd at first sight, and not in line with previous considerations on extension of classical computing. Indeed, if we can test a property $x \mapsto f(x) \in \{0,1\}$ on an n-bit string classically (with a classical Boolean circuit), then we can also build the map $g : |x\rangle |b\rangle \mapsto |x\rangle |f(x) \oplus b\rangle$, which is reversible (bijective). Here $\oplus$ is the addition modulo two (i.e. XOR). If $b = 0$ then the last bit of the output is $f(x)$.

Because g is bijective, it can be built from reversible gates, which can be extended to quantum inputs. This is the straightforward way to build a quantum circuit U_g testing a classical property. Fortunately, this quantum circuit U_g can emulate U_ω simply by setting $|b\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$ (which can be done with a Hadamard gate applied to $|1\rangle$). Indeed we see that $|x\rangle |b\rangle$ is then mapped by U_g to $|x\rangle |b\rangle$ if $f(x) = 0$, and to $-|x\rangle |b\rangle$ if $f(x) = 1$ (i.e., if $x = \omega$).

9.8. Shor's algorithm

One of the most important problems that can be solved with a quantum computer is the factorization of an integer. The problem of factoring numbers is an important problem for computer science and the fact that there is no known algorithm in polynomial time is the basis of some cryptographic protocols, such as RSA (Rivest-Shamir-Adleman). Also, the problem of finding an n-bit integer is conjectured to be not in P and not in NP-complete, but "in-between" these classes.

Peter Shor (1994) discovered that Quantum computers can factor numbers efficiently and in this section we will show the main ideas behind this algorithm, which have interest on their own.

9.8.1. Quantum Fourier Transform (QFT)

Remember the matrix form of the Discrete Fourier Transform (DFT):

$$FT_N = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \in \mathbb{C}^{N \times N}$$

where ω is the N$^{\text{th}}$ root of 1, i.e. $\omega = e^{\frac{2\pi i}{N}}$ and $\omega^N = 1$. We can also verify that $\frac{1}{\sqrt{N}} FT_N$ is unitary: all columns are of unit norm and orthogonal to one another.

For these reasons, if $N = 2^n$ then we can interpret $\frac{1}{\sqrt{N}} FT_N$ as a quantum gate acting on n qubits (a vector of 2^n components). So, this transformation is the Quantum Fourier Transform (QFT). And we have that:

$$QFT_N = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \in \mathbb{C}^{N \times N}$$

For example, if we want to apply a QFT on 2 qubits, we will have that $N = 2^2$ and $w = e^{\frac{2\pi i}{4}}$ and the matrix QFT_4 will be:

$$QFT_4 = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

Even simpler is QFT_2, the QFT on a single qubit, which coincides with Hadamard's gate.

$$H = QFT_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

9.8.2. Encoding a set of integers as a superposition state

A natural integer $k \in \{0, ..., 2^n - 1\}$ can be encoded classically as a the sequence of n bits describing its usual binary expansion $|b_0 b_1 \dots b_{n-1}\rangle$, which we denote $|k\rangle$ for short. For example 6 can be represented with the classical state $|110\rangle$, denoted $|6\rangle$.

Starting from there, we can represent a set of integers $S \subseteq \{0, ..., 2^n - 1\}$ as a superposition $|S\rangle = \frac{1}{\sqrt{|S|}} \sum_{s \in S} |s\rangle$.

9.8.3. Finding a period

Suppose a set $S \subseteq \{0, ..., 2^n - 1\}$ is periodic, i.e. of the form $\left\{s, s + r, s + 2r, ..., s + \lfloor \frac{2^n - 1 - s}{r} \rfloor r\right\}$ for some period r and initial shift $s < r$. We can encode as a superposition of all $|s + jr\rangle$

as above. For example:

$$|S\rangle = \sum_{j=0,1,\ldots} |jr+s\rangle = \alpha \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ \vdots \end{pmatrix} \begin{matrix} \\ \\ \text{entry } s \\ \\ \\ \\ \text{entry } s+r \\ \vdots \end{matrix}$$

where $\alpha = 1/\sqrt{|S|}$ is a normalizing factor.

How can we find the period r? With QFT, of course! The transformation $QFT_N |S\rangle$ will have high values for "frequencies" b such that $br \equiv 0 \pmod{2^n}$ i.e. br will be close to a multiple of 2^n.

For example, consider an input of 8-qubits:

$$|S\rangle = \frac{1}{\sqrt{4}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

The $\mathrm{QFT_N}$ on $|S\rangle$ will yield, for $\omega^8 = 1$:

$$QFT_N |S\rangle = \frac{1}{\sqrt{8}} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^{10} & \omega^{12} & \omega^{14} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \omega^{15} & \omega^{18} & \omega^{21} \\ 1 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \omega^{20} & \omega^{24} & \omega^{28} \\ 1 & \omega^5 & \omega^{10} & \omega^{15} & \omega^{20} & \omega^{25} & \omega^{30} & \omega^{35} \\ 1 & \omega^6 & \omega^{12} & \omega^{18} & \omega^{24} & \omega^{30} & \omega^{36} & \omega^{42} \\ 1 & \omega^7 & \omega^{14} & \omega^{21} & \omega^{28} & \omega^{35} & \omega^{42} & \omega^{49} \end{pmatrix} \frac{1}{\sqrt{4}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{32}} \frac{1}{\sqrt{8}} \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Applying the $\mathrm{QFT_N}$ on $|S\rangle$ we will have:

$$QFT_N |S\rangle = \frac{1}{\sqrt{8}} \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{|0\rangle + |4\rangle}{\sqrt{2}}$$

In this case we know that the period r is 2 and after applying the QFT we get that b is 0 or 4 (nonzero entries in 0th and 4th position) and indeed $rb \equiv 0 \pmod 8$.

In general if we measure $QFT_N |S\rangle$ we find one b such that $br \approx 2^n m$ for some m. We can conclude then, that r is a multiple of $\frac{2^n}{b}$.

This way, we can combine several measurements to find r with high probability. We will omit the details of this process.

9.8.4. Finding the order of a number modulo M

Now let's suppose we have a n-bit number $M \in \mathbb{N}$ and we want to find a number y such that $y < M$ coprime with M.

There is r s.t. $y^r \equiv 1$ modulo M, which means that $y^r = 1 + aM$, for some integer a.

Proof. If we take the powers of y: y^1, y^2, y^3, y^4.... At some point $y^k \equiv y^\ell$ modulo M for some k, ℓ, then $y^{k-\ell}(y^\ell - 1) \equiv 0$ modulo M. Thus $y^{k-\ell}(y^\ell - 1)$ is a multiple of M.

Because y is prime with M, then so is $y^{k-\ell}$. Then, $y^\ell - 1$ must be a multiple of M, i.e. $y^\ell \equiv 1$ modulo M. □

Our problem is to find the smallest such r, called the "order of y modulo M". The sequence 1, y^1, y^2, y^3, y^4, ... (modulo M) forms a periodic sequence with period r, so this suggests we may find it using the QFT. How do we proceed?

First we need to compute the powers of y modulo M. Let us consider the reversible map on $\{0,1\}^n \times \{0,1\}^n$, where $n = \lceil \log_2 n \rceil$:

$$(t, s) \mapsto (t, s \oplus y^t \text{ modulo } M).$$

Here $\oplus$ is the bitwise XOR. The argument s (an n-bit string, like t) is only there to make the map reversible.

There is an efficient classical way to compute this reversible map, using the square-and-multiply method to compute y^t modulo M. For instance y^{16} is obtained by squaring y four times, and $y^{18} = y^{16}y^2$. Overall, computing y^t mod N requires $\Theta(\log n)$ multiplications modulo M.

We can thus convert it to an efficient quantum circuit:

$$|t, s\rangle \mapsto \left|t, s \oplus y^t \text{ modulo } M\right\rangle$$

Specifically, we can write:

$$|t, 0\rangle \mapsto \left|t, y^t \text{ modulo } M\right\rangle$$

Like in Grover's algorithm, we start with the quantum state $|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{t=0}^{2^n-1} |t\rangle$, that can be obtained with Hadamard's gates. Then we have that:

$$|x,0\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{t=0}^{2^n-1} \Big|t, y^t \bmod M\Big\rangle = \frac{1}{\sqrt{2^n}} \sum_{m=0}^{r-1} \sum_{j=0}^{\lfloor (2^n-m)/r \rfloor} |jr+m, y^m \bmod M\rangle$$

In the last summation we have regrouped all $t = jr + m \in \{0, ..., 2^n - 1\}$ so that $y^t \equiv y^{mt}$ modulo M.

Consider the subsum $\sum_{j=0}^{\lfloor (2^n-m)/r \rfloor} |jr+m, y^m \bmod M\rangle$ for a given m, it can be written as:

$$\Big(\sum_{j=0}^{\lfloor (2^n-m)/r \rfloor} |jr+m\rangle\Big) \otimes |y^m \bmod M\rangle$$

The second n-qubit string $|y^m \bmod M\rangle$ is constant, while the first is a superposition representing the periodic set $\{m, m+r, m+2r, \ldots\}$ of period r. We can thus apply the Quantum Fourier Transform $\mathrm{QFT_n}$ on the first term and obtain of b such that br is (approximately) a multiple of 2^n This is true for each m separately, so, by linearity, this is true for the sum of $\sum_{j=0}^{2^n-\frac{1}{2}} |jr+m, y^m \bmod M\rangle$ over all m.

So, in order to obtain the order of a number modulo M, we need to apply $\mathrm{QFT_n}$ on the first n qubits of the state above, then measure the whole state (and obtain some classical state $|b\rangle \otimes |y^m \bmod M\rangle$), repeat this several times then deduce from the values of b the period r.

9.8.5. Factoring Large Numbers: Shor's algorithm

Now we will present the full algorithm to factor a number M, that is not prime.

First we pick a random y s.t. $y < M$, if $\gcd(y, M) > 1$ we can compute it easily using the Euclid's algorithm. However, if $\gcd(y, M) = 1$, then y is coprime with M and we compute the order r of y mod M ($y^r \equiv 1$ modulo M) with the algorithm above.

If the calculated r is odd we start the algorithm again, choosing another y. Nonetheless, if r is even we can write $(y^{\frac{r}{2}} - 1)(y^{\frac{r}{2}} + 1) \equiv 0$ modulo M.

The number r is the smallest solution to $y^r \equiv 1$ modulo M, so $y^{\frac{r}{2}} - 1$ is not a multiple M. We will accept that $y^{\frac{r}{2}} + 1$ is not a multiple of M either (with high probability).

Consequently, M has a common factor with $y^{\frac{r}{2}} + 1$, i.e. $\gcd(M, y^{\frac{r}{2}} + 1) > 1$, so we found a non-trivial divisor of M.

For example, if $M = 15$ and Shor's algorithm happens to pick $y = 7$ then we find $r = 4$ (as $7^4 \equiv 1 \bmod 15$),which is even, thus $(7^2 - 1)(7^2 + 1) \equiv 0 \bmod 15$. We see that neither $7^2 - 1$ nor $7^2 + 1$ is a multiple of 15, thus both have a factor in common with 15. Indeed $\gcd(7^2 - 1, 15) = 3$ and $\gcd(7^2 + 1, 15) = 5$. Thus we found two divisors of 15, namely 3 and 5.

Note that the only quantum part in this algorithm is to find the order of y. All the other steps are performed efficiently by classical computers.

9.9. Discussion

9.9.1. Practical realizations

Handling a quantum system is complicated due to unwanted interference with the surroundings, which is akin to 'unwanted measurement'. It must be well controlled in the Lab. In spite of 'quantum error-correcting codes' that can handle partial loss of coherence (superposition), current devices can only handle a few hundreds of qu-bits. For instance IBM boast 433 qu-bits for their best 'Osprey' processor. See Table for some milestones in practical realisations.

Table 9.1.: Timeline

1980	Alain Aspect achieves a quantum EPR pair in the lab
2001	Factoring 15 with 7 qubits using Shor's algorithm (IBM, Stanford)
2012	Factoring 21 (IBM)
2012	The Nobel Physics Prize goes to S.Haroche and D. J. Wineland Quantum System Manipulation
2012	DeltaQBit - Quantum software company
2013	Google launch Quantum Artificial Intelligence Lab
2014	NSA Quantum research program
2019	Google claims "quantum supremacy"
2019	An attempt to factor 35 fails, due to errors
2022	The Nobel Physics Prize goes to Alain Aspect

A. Asymptotic notation

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

- $f \in \mathcal{O}(g)$, often written $f(n) = \mathcal{O}(g(n))$ by abuse of notation, means: $\exists n_0, c : \forall n \geq n_0, f(n) \leq c \cdot g(n)$.
- $f \in \Theta(g) \quad \begin{array}{l} \Leftrightarrow f \in \mathcal{O}(g) \text{ and } g \in \mathcal{O}(f) \\ \Leftrightarrow \exists n_0, c_1, c_2 : \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \end{array}$ (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. We can write, with the usual abuse of notations

$$\frac{n(n-1)}{2} = \Theta(n^2)$$
$$\frac{n(n-1)}{2} = \mathcal{O}(n^2)$$
$$\frac{n(n-1)}{2} = o(n^3)$$
$$\frac{n(n-1)}{2} = \Omega(n \log(n))$$

A key property of the $\Theta(.)$ sets is that, for any functions f, g:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max(f + g)).$$

and similarly for $\mathcal{O}(.)$ sets we may write:

$$\mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f + g) = \mathcal{O}(\max(f + g)).$$

Of course these properties are extended to the sum of three, or four, or any fixed number of functions. One should be wary of unbounded sums however. For example writing

$$\Theta(1) + \ldots + \Theta(n-1) + \Theta(n) = \Theta(n^2)$$

is only correct if we specify that the hidden constants in the $\Theta(.)$ notation can be chosen uniformly over the terms. Sometimes this specification is implicit because obviously satisfied, from the context: this is yet another common (but dangerous) abuse of notations associated to the $\Theta(.)$ notation.