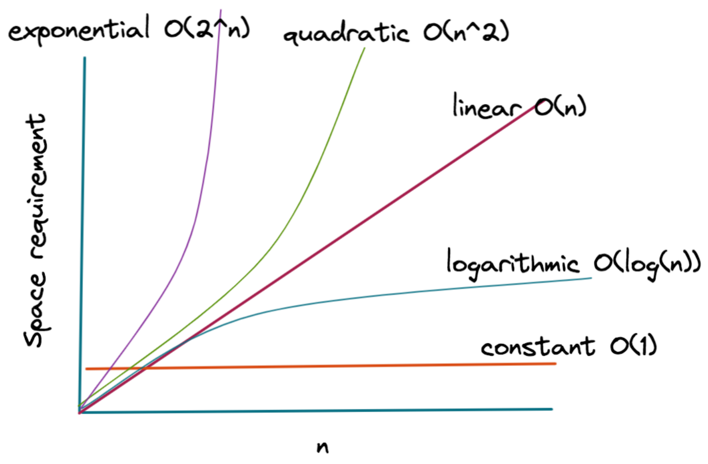


LINMA2111 - Discrete mathematics II

Charles Van Hees
Simon Desmidt
Issambre L'Hermite Dumont

Academic year 2025-2026 - Q1

UCLouvain

Contents

Sorting Algorithms

Let S be a set with a total order $\leq$. Given an array of n elements of S, we owuld like to get a permutation of that input array that respects the order. When analyzing an algorithm, we would like to check its correctness and its complexity (both time and space).

1.1 Bachman-Landau complexity notations

Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We write

$$\begin{aligned} & f \in \mathcal{O}(g) \iff \exists c > 0, \exists n_0, \forall n > n_0,\ f(n) \leq cg(n) \\ & f \in \Omega(g) \iff g \in \mathcal{O}(f) \\ & f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ and } f \in \Omega(g) \\ & f \in o(g) \iff \forall c > 0, \exists n_0, \forall n > n_0,\ f(n) < cg(n) \\ & f \in \omega(g) \iff g \in o(f) \end{aligned} \tag{1.1}$$

For example, $\frac{n(n-1)}{2} \in \mathcal{O}(n^3)$ and $\frac{n(n-1)}{2} \in \Omega(n \log(n))$.

Proposition 1.1.

$$\begin{aligned} \forall f, g, \qquad & \Theta(f) + \Theta(g) = \Theta(f+g) = \Theta(\max(f,g)) \\ & \mathcal{O}(f) + \mathcal{O}(g) = \mathcal{O}(f+g) = \mathcal{O}(\max(f,g)) \end{aligned} \tag{1.2}$$

The time complexity is the number of operations as a function of the input size. The space complexity is the amount of memory (used in addition to the input) as a function of the input size.
The worst-case (respectively best-case) time complexity is the maximum (respectively minimum) running time over all instances of size n. The average-case time complexity is the average time over all instances of size n, for some probability distribution over the instances:

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

where D is the distribution of the input.

1.2 Sorting Algorithms

1.2.1 Selection Sort

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

Algorithm 1 Selection sort algorithm

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

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

1.2.2 Insertion Sort

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

Algorithm 2 Insertion sort algorithm

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

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

1. Check the basic and the induction cases;
2. Check that it is a permutation of the original array;
3. Check that it is sorted.

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

$$\begin{aligned} &\{\text{Algorithm to be used; Precondition; Postcondition}\} \\ &\{IS;\ x = x_0;\ x \text{ is sorted and is a permutation of } x_0\} \end{aligned} \tag{1.4}$$

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

- the worst-case complexity is $\mathcal{O}(n^2)$;
- the best-case complexity is $\mathcal{O}(n)$;
- the average-case complexity is $\mathcal{O}(n^2)$: for a uniform probability distribution over the instances,

$$\mathbb{E}(time) = \mathbb{E}\left[\Theta(\sum_{i=2}^{n} t_i)\right] = \Theta\left[\sum_{i=2}^{n} \mathbb{E}(t_i)\right] = \Theta\left[\sum_{i=2}^{n} \frac{i}{2}\right] = \Theta(n^2) \tag{1.5}$$

1.2.3 Quick sort

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

Algorithm 3 Quick Sort algorithm

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

The worst-case complexity is still $\mathcal{O}(n^2)$:

$$t_n = t_{n-1} + \Theta(n) = t_{n-2} + \Theta(n-1) + \Theta(n) = \Theta(n^2) \tag{1.6}$$

where the $\Theta(n)$ comes from the partioning of the array. The average-case complexity is lower: $\mathcal{O}(n \log(n))$:

$$t_n = 2t_{n/2} + \Theta(n) = 2^{\log_2(n)} t_1 + \Theta(n \log_2(n)) = \Theta(n \log(n)) \tag{1.7}$$

The randomized version of the algorithm consists in shuffling before applying the classical quick sort, in order to change the pivot value. As this is a random algorithm, the worst-case complexity corresponds to the expected complexity, i.e. $\mathcal{O}(n \log(n))$. This is the "Robin Hood effect": we put all instances on an equal footing, and steal from the rich (good) instances to give to the poor ones.

Theorem 1.2. For a non decreasing, nonnegative function $f(\cdot)$, we have the inequalities

$$f(0) + \cdots + f(n-1) \leq \int_0^n f(x)dx \leq f(1) + \cdots + f(n) \tag{1.8}$$

Theorem 1.3. Markov's inequality For a random variable T taking only nonnegative values,

$$Pr[T > a\mathbb{E}[T]] \leq \frac{1}{a} \qquad a > 0 \tag{1.9}$$

1.2.4 Merge Sort

The merge sort algorithm is also based on divide-and-conquer, and its complexity is also $\Theta(n \log(n))$.

Algorithm 4 Merge Sort algorithm

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

The recurrence equation for the complexity is

$$t_n = 2t_{\lceil n/2 \rceil} + \Theta(n) \implies t(n) = \Theta(n \log(n)) \tag{1.10}$$

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

Algorithm	Pros	Cons
Randomized Quick Sort	Small hidden constants if implemented efficiently	Requires randomness generation, which may be nontrivial
	Can be implemented in-place; requires $\Theta(\log n)$ extra memory for recursion	Worst-case behavior possible due to unlucky random choices
Merge Sort	Requires little extra memory when the input is a linked list	Larger hidden constant
	Stable and predictable performance	Requires $\Theta(n)$ extra memory for direct-access arrays

1.2.5 Heap Sort

This algorithm takes the advantages of the two previous ones: being a deterministic algorithm in $\mathcal{O}(n \log(n))$, and sorting in-place (does not need to copy the original array to another structure, hence requiring little extra memory).
We define a data structure as a static structured set, with a set of operations that describe how to access or modify the structure set. What we need is a priority queue: each entry is associated with a number (its priority) and two operations:

- Insert (entry priority): inserts the entry with the given priority;
- ExtractMax: removes and otuputs the entry witht the highest priority number.

The Heap Sorting algorithm consists in inserting the n elements in a priority queue, and ExtractMax n times.
The complexity is

$$\Theta(cost(Insert) + cost(ExtractMax)) \tag{1.11}$$

- For an unordered table, the insert is in $\mathcal{O}(1)$ and the ExtractMax in $\mathcal{O}(n)$, and the algorithm then corresponds to the selection sort ($\mathcal{O}(n^2)$).
- For an ordered table, the insert is in $\mathcal{O}(n)$ and the ExtractMax in $\mathcal{O}(1)$, and the algorithm corresponds to the insertion sort ($\mathcal{O}(n^2)$).

A heap is a tree with the following properties:

- Essentially binary complete (only the last level may be incomplete, it is filled from left to right);
- entry (x) $\leq$ entry(father(x)) for all nodes x different from the root;

The insertion consists in adding the new node to the last level in the first free position, and swap it with its father until the heap is properly re-established ($\mathcal{O}(\log(n))$). The ExtractMax consists in removing the root (max entry) and putting the last entry at the root, then swap the new root with its largest child until the heap is properly restored. The result algorithm is called HeapSort and has complexity $\mathcal{O}(n\log(n))$. 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.

$$\begin{gathered}\text{LeftChild}(T(i)) = T(2i) \qquad \text{RightChild}(T(i)) = T(2i+1) \\ \text{Father}(T(i)) = T(\lfloor i/2 \rfloor)\end{gathered} \tag{1.12}$$

1.2.6 Searching for a better complexity

Assume a comparison-based algorithm, i.e. the only query on the data is "Is $x \leq y$?". Can we sort in $o(n\log(n))$, i.e. in a complexity strictly better than $n\log(n)$.

Theorem 1.4. A binary tree with at most 2 children for each node, wit N leaves, has at least $\lfloor \log_2(N) \rfloor$ levels.

Theorem 1.5. There does not exist a deterministic algorithm that can sort n entries (only using comparisons) in worst-case time complexity in $o(n\log(n))$.

Proof. A sorting algorithm running on an array of n entries proceeds with a sequence of comparisons, e.g. "is $T(1) < T(S)$?". Depending on the answer, it will eventually proceed to another comparison. This sequence of comparisons and their answers can be represented by a decision tree. The number of leaves of this tree is $n!$, i.e. the number of permutations of the data.
Indeed, two distinct initial orders on the data require a different permutation to become correctly sorted, and therefore 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. Therefore, the binary tree has at least $\log_2(n!)$ levels, and each path from root to leaf is a sequence of binary answers queried from the data, thus a lower bound on the computation time.
Finally, $\log(n!) = \Theta(n\log(n))$:

$$\begin{gathered}\int_1^n \log(x)dx \leq \log(n!) = \sum_{i=1}^{n} \log(i) \leq \int_1^{n+1} \log(x)dx \\ \implies [x\log(x) - x]_1^n \leq \log(n!) \leq [x\log(x) - x]_1^{n+1} \iff S = \Theta(n\log(n))\end{gathered} \tag{1.13}$$

□

This is the answer to the best worst-case complexity. What about the average-case?

Theorem 1.6. Shannon theorem. Let a N-set be endowed with the probabilistic distribution $\rho_1, \ldots, \rho_N$. If we label every entry with a binary word (called prefix), such that no word is a prefix of another, then the expected length (weighted by the ρ_i) is at least

$$-\sum_{i=1}^{N} \rho_i \log(\rho_i) \tag{1.14}$$

If $S = \{1, \ldots, N\}$ and P is a random variable over S such that $Pr[P = i] = \rho_i$ and $f : S \to \{0, 1\}^*$ is a prefix code, then

$$\sum_{i=1}^{N} \rho_i |f(i)| \geq -\sum_{i=1}^{N} \rho_i \log_2(\rho_i) =: H(P) \tag{1.15}$$

We call $H(P)$ the entropy of the probabilistic distribution. The entropy of any probabilistic distribution over N elements is at most $\log_2(N)$, which is attained for the uniform distribution. If the N elements are N leaves of a binary tree, enco ded by their path from root to leaf (giving the prefix code), we obtain the following theorem:

Theorem 1.7. The expected length of a path from root to leaf of an N-leaf binary tree, with probabilities $\rho_1, \ldots, \rho_N$ on the leaves, is at least the entropy $H(P)$. In particular, if all leaves occur equally likely, the average length of the path is at least $\log_2(N)$.

Theorem 1.8. The average-case complexity, for uniform distributions over all instances, of any deterministic sorting algorithm using comparisons is $\Omega(n \log(n))$.

And now, can we do better with randomized algorithms?

Theorem 1.9. Yao's minimax principle. Consider a probabilistic distribution over the instances of a given size of a given problem. Then, there exists 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 probabilities $\rho_1, \ldots, \rho_N$. Consider a random algorithm A, drawing i.i.d. bit sequences B to decide of its random actions. Consider also a specific bit sequence $B = b$. Then, knowing this specific sequence of bits, the algorithm is now deterministic, with a fixed running time $t(A, x_i | B = b)$ on each instance x_i. Then,

$$\mathbb{E}_B[t(A, x_i)] = \sum_b t(A, x_i | B = b) Pr[B = b] \tag{1.16}$$

The worst-case expected complexity is thus

$$\begin{aligned} \max_{x_i} \mathbb{E}_B[t(A, x_i)] &= \max_{x_i} \sum_b t(A, x_i | B = b) Pr[B = b] \\ &\geq \mathbb{E}_{x_i} \left[\sum_b t(A, x_i | B = b) Pr[B = b] \right] \\ &\geq \min_b \mathbb{E}_{x_i} \left[t(A, x_i | B = b) \right] Pr[B = b] \end{aligned} \tag{1.17}$$

where the first inequality is the average-case expected complexity, and the second is the average-case complexity of the deterministic algorithm A with a fixed sequence $B = b$.

$\square$

Applying this to the comparison-based sorting, 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, this is true for the uniform distribution, which has been proven to be $\Omega(n \log(n))$.

$\to$ Note: with additional knowledge about the data, we can beat the $\Omega(n \log(n))$ bound, e.g. using counting sort.

Divide-and-conquer algorithms

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

- Divide the problem in smaller subproblems;
- Conquer the subproblems by solving them recursively. When it is sufficiently small, solving it is straightforward. Subproblems can be distributed on several processors;
- Combine the solutions of the subproblems into the solution of the original problem.

2.1 Multiplication of large integers

We want to multiply two n-digit numbers. The elementary school method has a complexity $\Theta(n^2)$:

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

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

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

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

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

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

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

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

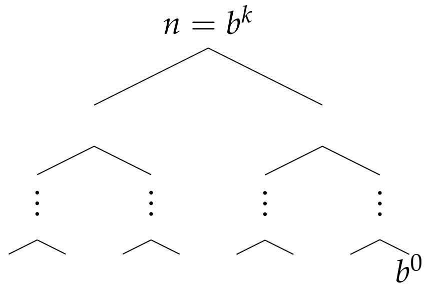
This reduces the complexity to $\Theta(n^{1.58})$.
Following a similar reasoning, we can instead divide a and b into 3 sums instead of 2, and get 5 multiplications. This gives a complexity of $\Theta(n^{\log_3(5)}) = \Theta(n^{1.46})$. Dividing in 4, 5, etc, we converge to a complexity of $\Theta(n)$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

2.1.1 Master Theorem

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

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

2.1.2 Proof of the master theorem



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

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

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

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

Under the assumption that $\alpha \neq \log_b(a)$. As $k = \log_b(n)$, we can replace above and simplify using the formula $a^{\log_b(n)} = n^{\log_b(a)}$. Depending if α is bigger or larger than $\log_b(a)$, the simplifications change. In the case where $a = b^\alpha$, then

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

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

2.2 Discrete Fourier Transform

Let us consider two polynomials $A(z) = a_0 + a_1 z + \cdots + a_{N-1}z^{N-1}$ and $B(z) = b_0 + b_1 z + \cdots + b_{N-1}z^{N-1}$, where $a_i, b_i \in \mathbb{R}$ or $\mathbb{C}$. There are two ways to describe a polynomial: with a list of N coefficients or with its evaluation at N distinct points. Those two representations correspond to two bases of the vector space of polynomials of degree $N-1$, and are therefore related by an invertible matrix. The two representations are computationally convenient for different tasks:

- Evaluation of a polynomial at an arbitrary value z: use coefficients ($\Theta(N)$);
- Multiplication of two polynomials: use the evaluations ($\Theta(N)$).

However, sometimes we only have the coefficients, and we want to multiply the polynomials. The naive multiplication of the polynomials is $\Theta(N^2)$:

$$A(z)B(z) = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} a_j b_k z^{j+k} \tag{2.8}$$

We thus need an algorithm to make it faster.
To take advantage of both representations, we will choose a list of evaluation points $z_0, \ldots, z_{N-1}$ for which the conversion between representations is fast to compute. As the resulting polynomial of the product is of degree $2N-2$, we will use zero-padding to make $A(z)$ and $B(z)$ of degree $2N-2$ as well:

$$\begin{aligned} A(z) &= a_0 + a_1 z + \cdots + a_{N-1}z^{N-1} + 0z^N + \cdots + 0z^{2N-2} \\ B(z) &= b_0 + b_1 z + \cdots + b_{N-1}z^{N-1} + 0z^N + \cdots + 0z^{2N-2} \end{aligned} \tag{2.9}$$

For simplicity of the notations, we will explain the Fourier transform for the first N coefficients, but it can of course be extended to the polynomials using the zero-padding.

As points of evaluation, we can take $z_k = \omega^k$, for $k = 0, \ldots, N-1$ and $\omega = \exp(2\pi i/N)$, where ω is an Nth root of unity, meaning z_k are all the roots of unity, satisfying $z_k^N = 1$. The sequence of evaluations of the polynomial $(A(z_0), \ldots, A(z_{N-1}))$ is called the Discrete Fourier Transform (DFT) of the sequence of coefficients $(a_0, \ldots, a_{N-1})$, denoted $\mathcal{F}(a_0, \ldots, a_{N-1})$. This computation is a matrix-vector product with a Vandermonde matrix:

$$\begin{pmatrix} A(z_0) \\ A(z_1) \\ \vdots \\ A(z_{N-1}) \end{pmatrix} = \begin{pmatrix} A(1) \\ A(\omega) \\ \vdots \\ A(\omega^{N-1}) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{N-1} \end{pmatrix} \tag{2.10}$$

Please also note that computing the coefficients from the evaluation points z_j is quite similar by the definition of the matrix V:

$$\begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{N-1} \end{pmatrix} = \frac{1}{N} V^* \begin{pmatrix} A(z_0) \\ A(z_1) \\ \cdots \\ A(z_{N-1}) \end{pmatrix} \tag{2.11}$$

where V^* is the conjugate transpose of V.
The naive matrix-vector product is still in $\mathcal{O}(N^2)$, so we need to leverage more of the information that we have to accelerate the computations.

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

$$A(z) = A_{\text{even}}(z^2) + zA_{\text{odd}}(z^2) \tag{2.12}$$

where $A_{\text{even}}(z) = a_0 + a_2 z + a_4 z^2 + \dots$ and $A_{\text{odd}}(z) = a_1 + a_3 z + a_5 z^2 + \dots$. Assume N to be even for simplicity. Then,

$$A_{\text{even}}((\omega^k)^2) = A_{\text{even}}((\omega^2)^k) \tag{2.13}$$

where ω^2 is a $N/2$-th root of unity, by definition of ω. Thus, the sequence of $\{A_{\text{even}}((\omega^k)^2)\}_{k=1}^{\frac{N}{2}-1}$ is the DFT of the even coefficients, $\mathcal{F}_{N/2}(a_0, a_2, \dots, a_{N-2})$.
As for $k = \frac{N}{2}, \dots, N-1$, we observe that $(\omega^2)^{N/2+\ell} = (\omega^2)^\ell$ since $\omega^N = 1$. Therefore, the sequence of $A_{\text{even}}((\omega^k)^2)$ for $k = \frac{N}{2}, \dots, N-1$ is once again $\mathcal{F}_{N/2}(a_0, \dots, a_{N-2})$.

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

$$\{(\mathcal{F}_{N/2}(a_1, a_3, \dots, a_{N-1}), \mathcal{F}_{N/2}(a_1, a_3, \dots, a_{N-1}))\}_{k=0}^{N-1}$$

The combination of the two thus gives

$$\begin{aligned} \mathcal{F}_N(a_0, a_1, \dots, a_{N-1}) = &[\mathcal{F}_{N/2}(a_0, a_2, \dots, a_{N-2}), \mathcal{F}_{N/2}(a_0, a_2, \dots, a_{N-2})] \\ &+ \omega\, [\mathcal{F}_{N/2}(a_1, a_3, \dots, a_{N-1}), \mathcal{F}_{N/2}(a_1, a_3, \dots, a_{N-1})] \end{aligned} \tag{2.14}$$

If N is a power of 2 (without loss of generality), we can apply this formula recursively, with the following relation for complexity:

$$t_N = 2t_{N/2} + \Theta(N) \implies t_N = \Theta(N \log(N)) \tag{2.15}$$

This divide-and-conquer strategy to compute the direct and inverse DFT is called the Fast Fourier Transform (FFT). This gives a rigorous method for the multiplication of two polynomials in the coefficient representation: given $A(z)$ and $B(z)$ of degree $N-1$, we want to compute $C(z) = A(z)B(z)$ of degree $2N-2$.

1. Compute the FFT of the coefficients of A (with zero-padding of the last $N-1$ coefficients) and the FFT of the coefficients of B (with zero-padding also), padded with zeros so as to reach $2N-2$ or the next power of 2; $[\Theta(N \log(N))]$
2. Multiply the two resulting Fourier domain representations (each of size $2N-2$) pointwise, which results in the Fourier domain representation of the coefficients of C; $[\Theta(N)]$
3. Compute the inverse DFT in order to obtain the coefficients of C. $[\Theta(N \log(N))]$

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

This idea can be used to compute the product of two large numbers: represent them in polynomials using their base b ($X = x_{N-1} b^{N-1} + \dots$), compute the Fourier domain representations like for polynomials above, and evaluate them at the base value b. Doing it in a finite field, the complexity is $\Theta(N \log(N) \log(\log(N)))$.

2.3 Matrix multiplication

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

2.3.1 Straßen algorithm

Let us divide the matrices in blocks:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{2.16}$$

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

$$\begin{cases} M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 = (A_{21} + A_{22})B_{11} \\ M_3 = A_{11}(B_{12} - B_{22}) \\ M_4 = A_{22}(B_{21} - B_{12}) \\ M_5 = (A_{11} + A_{12})B_{22} \\ M_6 = (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \end{cases} \implies C = \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{pmatrix} \tag{2.17}$$

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

2.3.2 Matrix inversion

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

$$D = \begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix} \implies D^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix} \tag{2.18}$$

This means that the complexity of inversion is an upper bound on the complexity of multiplication: $M(n) \leq I(3n) \leq 3^3 I(n)\ \forall n \geq n_0$.
This kind of inequality is called reduction. Given problems A and B, if A can be transformed in a problem B of size $f(n)$ in time $T_R(n)$, then

$$T_A(n) \leq T_R(n) + T_B(f(n)) \tag{2.19}$$

For example, the problem of finding the median of an array can be transformed into the problem of sorting the array.
Let us assume that $A = A^T$ and $A \succ 0$.

$$A = \begin{pmatrix} B & C \\ C^T & D \end{pmatrix} \implies A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}CS^{-1}C^TB^{-1} & -B^{-1}CS^{-1} \\ -S^{-1}C^TB^{-1} & S^{-1} \end{pmatrix} \tag{2.20}$$

where $S = D - C^T B^{-1} C$ is the Schur complement of A. From Equation 2.19, $f(n) = \Theta(M(n))$ in our case, and so we have the following relation between inversion and multiplication:

$$I(n) = \mathcal{O}(M(n)) \tag{2.21}$$

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

Theorem 2.1. Matrix inversion and multiplication have the same complexity.

Dynamic programming

3.1 Two approaches

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

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

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

3.2 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. Pascal's rule is

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \qquad 0 < k < n \tag{3.1}$$

A naive implementation is the direct translation into a recursive algorithm using the initial condition $\binom{n}{0} = 1$. The time complexity is $\Theta\left(\binom{n}{k}\right)$ and its space complexity is $\Theta(n)$, i.e. the height of the computation tree.
To be more efficient, we can use a divide-and-conquer approach using Pascal's rule, where we store the intermediate results in a table $n \times k$. An efficient implementation of this algorithm uses only one array of size $k+1$, as we only need to store the last column, not the whole table. The space complexity becomes $\Theta(k)$ and the time complexity is $\Theta(nk - k^2)$.
This is a "bottom-up" approach of dynamic programming.
For our initial problem, the time complexity does not change, but the space complexity is worse: $\Theta(nk - k^2)$. The top-down approach 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.

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

3.3 Chained matrix products

Given $n \geq 2$ compatible matrices, we want find the order (associativity) in which perfoming products leads to the smallest cost. The number of possible groupings for computing the product of n matrices is the $(n-1)$-th Catalan number, denoted C_{n-1}. It is also the number of full binary trees[1] with n leaves.

$$((A_1A_2)(A_3A_4))(A_5\dots) = (A_1(A_2A_3)(A_4A_5))\dots \tag{3.2}$$

This problems consists in doing the multiplication of matrices $A_1A_2\dots A_n$ with dimensions $p_0, p_1, \dots, p_n$. The total number of operations depends heavily on the order of the multiplication. The idea for the algorithm is to split the chain into smaller chains until we get to a product of two matrices. The splitting happens at the index i that minimizes the total number of operations, knowing the cost of computing a subchain:

$$Algo(p) = \min_{1\leq i\leq n-1}\{Algo(p[0:i]) + Algo(p[i+1:n]) + p_0p_ip_n\} \tag{3.3}$$

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

$$m[i,j] = \min_{i\leq k<j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} \qquad \forall 1\leq i\leq j\leq n \tag{3.4}$$

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

Theorem 3.1.

$$C_{n+1} = \sum_{k=0}^{n} C_kC_{n-k} \qquad \forall n\geq 0 \tag{3.5}$$

Proof. Let $n\in\mathbb{N}$. Any grouping of $n+2$ factors has the form

$$M_1\dots M_{n+2} = (M_1\dots M_{k+1})(M_{k+2}\dots M_{n+2}) \tag{3.6}$$

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 $n-k+1$ last factors in C_{n-k} possible ways. By the rule of product, there are thus C_kC_{n-k} ways for grouping the product $M_1\dots M_{n+2}$. As it is true for every $k\in\{0,\dots,n\}$, we get the desired result by the rule of sum. $\square$

We can show that[2]

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right) \tag{3.7}$$

To solve the problem, we can use a bottom-up dynamic programming approach:

[1]In a binary tree, every node has between 0 and 2 children.

[2]See section 3.6.

- Create a matrix C of size $n \times n$;
- Fill in the main diagonal with 0;
- Fill the immediate upper diagonal $j = i + 1$;
- Fill all other upper diagonals one by one.

The time to compute $C[i, j]$ is $\Theta(j - i)$, so the total time is

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(j - i) = \sum_{i=1}^{n} \Theta((n - i)^2) = \sum_{k=0}^{n-1} \Theta(k^2) = \Theta(n^3) \tag{3.8}$$

Notice that we only need $\Theta(n)$ memory space as we only need the diagonal we are currently working on.

3.4 Rod cutting

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

Algorithm 5 Rod Cutting Algorithm

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

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

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + T(n-1) + \sum_{j=0}^{n-2} T(j) = 2T(n-1) \implies T(n) = \mathcal{O}(e^n) \tag{3.9}$$

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

Algorithm 6 Memoized rod cutting algorithm

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

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

Algorithm 7 Bottom-up efficient rod cutting algorithm

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

3.5 Principles of dynamic programming

- Principle of optimality: for any optimal structure (e.g. list, tree) answering the problem for an instance, the substructures are also optimal for the subinstances.
- Subproblems are overlapping: divide-and-conquer is wasteful because it solves many times the same subinstance.

3.6 Generating functions

This is a powerful tool for solving recurrence equations. The idea is to associate a power series to a sequence $\{a_n\}_{n\in\mathbb{N}}$:

$$f(z) = \sum_{n=0}^{\infty} a_n z^n \qquad z \in \mathbb{C} \tag{3.10}$$

3.6.1 Fibonacci

$$F_n = F_{n-1} + F_{n-2} \qquad \forall n \geq 2 \qquad F_0 = 0; F_1 = 1 \tag{3.11}$$

We define the power series:

$$f(z) = \sum_{k\geq 0} F_k z^k \tag{3.12}$$

From the recurrence equation, we deduce that

$$\begin{aligned} \sum_{k\geq 0} F_{k+2} z^{k+2} &= \sum_{k\geq 0} F_{k+1} z^{k+2} + F_k z^{k+2} \\ \iff f(z) - F_0 - F_1 z &= z(f(z) - F_0) + z^2 f(z) \\ \iff f(z) = \frac{F_0 + (F_1 - F_0)z}{1 - z - z^2} &= \frac{z}{(z-\varphi)(z-\bar{\varphi})} \end{aligned} \tag{3.13}$$

where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\bar{\varphi} = \frac{1-\sqrt{5}}{2}$. Using partial fraction decomposition,

$$\begin{aligned} f(z) &= \frac{1}{\bar{\varphi} - \varphi} \left(\frac{\phi}{z - \phi} - \frac{\bar{\varphi}}{z - \bar{\varphi}} \right) = \sum_{n\geq 0} \left(\frac{\bar{\varphi}^{-n} - \varphi^{-n}}{\bar{\varphi} - \varphi} \right) z^n \\ \implies F_n &= \frac{\varphi^n - \bar{\varphi}^n}{\varphi - \bar{\varphi}} \end{aligned} \tag{3.14}$$

where the expression of F_n comes from $\varphi = -\bar{\varphi}^{-1}$.

3.6.2 Catalan numbers

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k} \qquad \forall n \geq 0 \qquad C_0 = 1 \tag{3.15}$$

We define the power series:

$$\begin{aligned} f(z) &= \sum_{n\geq 0} C_n z^n = 1 + z \sum_{n\geq 0} C_{n+1} z^n = 1 + z \sum_{n\geq 0} \sum_{k=0}^{n} C_k C_{n-k} z^n \\ &= 1 + z \sum_{n\geq 0} \sum_{k=0}^{n} (C_k z^k)(C_{n-k} z^{n-k}) \\ &= 1 + z \left(\sum_{n\geq 0} C_n z^n \right) \left(\sum_{k\geq 0} C_k z^k \right) = 1 + z f(z)^2 \end{aligned} \tag{3.16}$$

where the last result uses theorem 3.5. This gives $f(z) = \frac{1-\sqrt{1-4z}}{2z}$, rejecting the positive sign since $f(0) = 1$. By the generalized binomial formula (3.4),

$$\sqrt{1-4z} = \sum_{n=0}^{\infty} \binom{1/2}{n} (-4z)^n = 1 + \sum_{n=1}^{\infty} \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)\dots\left(\frac{3}{2}-n\right)}{n!}(-1)^n 2^{2n} z^n \tag{3.17}$$

Which gives

$$f(z) = -\frac{1}{2z}\sum_{n=1}^{\infty} \frac{\left(\frac{1}{2}\right)\left(-\frac{1}{2}\right)\dots\left(\frac{3}{2}-n\right)}{n!}(-1)^n 2^{2n} z^n \tag{3.18}$$

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

$$\begin{aligned} C_{n-1} &= -\frac{1}{2}\frac{\left(\frac{1}{2}\right)\dots\left(\frac{3}{2}-n\right)}{n!}(-1)^n 2^{2n} \\ &= \frac{1}{2}\frac{\left(\frac{1}{2}\right)\left(\frac{1}{2}\right)\dots\left(n-\frac{3}{2}\right)}{n!}2^{2n} = \frac{1}{2}\frac{1\cdot 1\dots(2n-3)}{n!}2^n \\ &= \frac{1}{2}\frac{(2n-2)!}{n!(2\cdot 4\dots(2n-2))}2^n = \frac{1}{2}\frac{(2n-2)!}{n!2^{n-1}(n-1)!}2^n = \frac{(2n-2)!}{n!(n-1)!} = \frac{1}{n}\binom{2n-2}{n-1} \end{aligned} \tag{3.19}$$

Then,

$$C_n = \frac{1}{n+1}\binom{2n}{n} \approx \frac{\sqrt{4\pi n}\left(2\frac{n}{e}\right)^{2n}}{(n+1)(2\pi n)\left(\frac{n}{e}\right)^{2n}} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right) \tag{3.20}$$

using Stirling's approximation: $\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^{2n}} = 1$.

3.6.3 Useful results for the proves

Definition 3.2. For all $r \in \mathbb{C}, n \in \mathbb{N}_0$, we define

$$\binom{r}{n} = \frac{r(r-1)\dots(r-n+1)}{n!} = \frac{1}{n!}\prod_{i=0}^{n-1}(r-i) \tag{3.21}$$

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

Theorem 3.3. For all $z \in \mathbb{C}$ such that $|z| < 1$, $\forall r \in \mathbb{C}$,

$$\sum_{n=0}^{\infty}\binom{r}{n} z^n = (1+z)^r \tag{3.22}$$

Theorem 3.4. Generalized binomial formula.

$$\forall r, x, y \in \mathbb{C} \text{ such that } |x| > |y|, \qquad (x+y)^r = \sum_{n=0}^{\infty}\binom{r}{n} x^{r-n} y^n \tag{3.23}$$

$\rightarrow$ Note: in the real case with r integer, all terms of index higher than r are zero.

Theorem 3.5. Cauchy product formula. Let $\{u_n\}_{n\in\mathbb{N}}$ and $\{v_n\}_{n\in\mathbb{N}}$ be two $\mathbb{C}$ sequences. If the series $\sum_{n=0}^{\infty} u_n$ and $\sum_{n=0}^{\infty} v_n$ converge absolutely, then the series

$$\sum_{n=0}^{\infty}\sum_{k=0}^{n} u_k v_{n-k} \tag{3.24}$$

converges to

$$\left(\sum_{n=0}^{\infty} u_n\right)\left(\sum_{n=0}^{\infty} v_n\right) \tag{3.25}$$

Greedy algorithms

4.1 Activity selection problem

Given a set of activities $\{a_1, \dots, a_n\}$ and the knowledge that activity a_i takes the 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. A dynamic programming approach could be used, as the optimality principle applies: if a set of activities is optimal for a certain time interval, then the subset of consecutive activities taking place in a subinterval is optimal for this subinterval, among all activities taking place in this subinterval.

$\rightarrow$ Note: the subinterval should start and finish at the end times of activities in the optimal set.

This dynamic programming algorithm can be done in $\mathcal{O}(n^2)$. However, there is a better approach, using a greedy algorithm: if we sort the activities with respect to their ending time, we take the first activity that finishes, then the first activity that starts and finishes after the ending time of the first one, and so on. This approach guarantees optimality. If we have an optimal solution that starts with an activity that is not the first to finish, then we can replace that first activity by the one that finishes first, keeping the same total number of activities.
The time complexity is $\Theta(n \log(n)) + \Theta(n)$, as we first need to sort the activities in order of their finishing time, and then go once through the array.

Randomized algorithms

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

5.1 The secretary problem

Consider that n candidates of different quality apply for an open position. Every time you interview a candidate who proves better than any other previous one, you hire them and fire the previous one. They are sent in a random order. How many people will you hire on average?
Let $X_i = 1$ if person i is hired and 0 otherwise. Then, $\mathbb{E}[X_i] = Pr[\text{person } i \text{ is hired}] = \frac{1}{i}$ because the best among the i first candidates is i with probability $1/i$.
Let X be the number of people hired.

$$\mathbb{E}[X] = \mathbb{E}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{i} \approx \int_1^n \frac{1}{i} di = \ln(n) \tag{5.1}$$

What if we can only hire one candidate? First, we proceed to an observation phase of k candidates. Next, we hire the first candidate that is better than the k candidates of the observation phase. What is the optimal k?
Let S_k be the event that we hire the best candidate, with an observation phase of k candidates, and let B_i be the event that i is the best candidate: $Pr[B_i] = \frac{1}{n}$, with H_i the event that we hire the i-th candidate.

$$Pr[S_k] = \sum_{i=k+1}^{n} Pr[H_i|B_i]Pr[B_i] \tag{5.2}$$

$H_i|B_i$ occurs iff the best among the first $i-1$ candidates is in the learning phase, which happens with probability $\frac{k}{i-1}$. Then,

$$Pr[S_k] = \sum_{i=k+1}^{n} \frac{k}{i-1}\frac{1}{n} = \frac{k}{n}\sum_{i=k}^{n-1}\frac{1}{i} \tag{5.3}$$

Since $\int_k^n \frac{1}{x}dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x}dx$,

$$\frac{k}{n}(\ln(n) - \ln(k)) \leq Pr[S_k] \leq \frac{k}{n}(\ln(n-1) - \ln(k-1)) \tag{5.4}$$

The lower bound is maximized for $k^* = n/e$ and has maximal value $1/e \approx 37\%$.

5.2 Birthday paradox

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

$$Pr[b_i = b_j] = \sum_{r=1}^{n} Pr[b_i = r \text{ and } b_j = r] = \sum_{r=1}^{n} Pr[b_i = r] Pr[b_j = r] = \sum_{r=1}^{n} \frac{1}{n^2} = \frac{1}{n} \tag{5.5}$$

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

$$\begin{aligned} Pr[B_k] &= 1 \cdot \frac{n-1}{n} \frac{n-2}{n} \dots \frac{n-k+1}{n} = 1 \cdot \left(1 - \frac{1}{n}\right) \cdot \left(1 - \frac{2}{n}\right) \dots \left(1 - \frac{k-1}{n}\right) \\ &\leq e^{-1/n} e^{-2/n} \dots e^{-(k-1)/n} = e^{-k(k-1)/2n} \end{aligned} \tag{5.6}$$

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

$$k(k-1) \leq 2n \ln 2 \iff k \leq 23 \tag{5.7}$$

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

$$\mathbb{E}[X] = \sum_{i=1}^{k} \sum_{j=i+1}^{k} X_{ij} = \frac{k(k-1)}{2n} \tag{5.8}$$

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

5.3 Hash table

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

Definition 5.1. A family of hash functions $\mathcal{H}$ is universal if for all $k \neq l \in U$,

$$Pr_{h \in \mathcal{H}}[h(k) = h(l)] \leq \frac{1}{m} \tag{5.9}$$

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

$$\mathcal{H} = \{h_{a,b}(k) = ((ak+b) \mod p) \mod m \mid a \in \{1,\ldots,p-1\}, b \in \{0,\ldots,p-1\}\} \tag{5.10}$$

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

5.4 Monte Carlo algorithm to compute an integral

Assume that we would like to compute $\int_a^b f(x)dx$, given that $f(x) \geq 0$, for f bounded in $[0,c]$. We can use the following randomized algorithm:

Algorithm 8 Random algorithm to compute an integral

```
1: $k = 0$;
2: for $n$ times do
3:     Pick $(x,y)$ in $[a,b] \times [0,c]$;
4:     If $y \leq f(x) : k++$;
5: end for
6: return $(b-a) \cdot c \cdot \frac{k}{n}$
```

Or a deterministic algorithm, where the upper bound on f is not needed:

Algorithm 9 Deterministic algorithm to compute an integral

```
1: $s = 0$; $x = a$; $D = (b-a)/n$;
2: for $n$ times do
3:     $x += D$;
4:     $s += f(x)$;
5: end for
6: return $s \cdot D$
```

This may be used to approximate π, since

$$\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx \tag{5.11}$$

One-dimensional integrals are usually better evaluated by deterministic methods, e.g. rectangles, Simpson, etc.

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 $\sum_{x\in X} F(x)p(x)$, where p is a probability distribution over a large discrete set X.
The interests of the Monte Carlo integration are:

- Robin Hood effect: for every deterministic integration method, there are functions for which the method will fail, e.g. if the function is zero at all the integration points. Those methods have a very poor worst-case behaviour, while the choice of random points makes all instances behave equally well.
- For higher dimensional integrals, the error of the Monte Carlo approximation still decreases in $\mathcal{O}(1/\sqrt{n})$, where n is the number of evaluation points, while the multidimensional rectangle method's error decreases in $\mathcal{O}(1/\sqrt[d]{n})$, meaning that when the dimension increases, we need exponentially more points to fill the space and preserve the accuracy.
- For domains with complicated geometry, or no explicit description, it may not be clear how to apply a deterministic method, while if we have a method that can generate points within X with probability $p(x)$, then we can still apply the MC method.

5.5 Markov Chain Monte Carlo methods and Metropolis-Hastings algorithm

Sometimes, the generation of samples according to a given probability distribution is not straightforward, e.g. when the probability distribution is only known up to a factor, or when we do not know the set X in which we sample uniformly (or its size).
The following technique allows to generate samples to any probability distribution $p(x)$ proportional to $f(x)$ (known) in a discrete or continuous space X. The idea is to perform a Markov chain on X. Starting from an arbitrary x_0, we jump randomly (with a certain probability law) to another point x_1, then x_2, etc. in a way that the stationary distribution, i..e the probability distribution for x_t for large t, is precisely $p(x)$.
Starting from an arbitrary x_0, one simply has to simulate the Markov chain for sufficiently many steps and retrieve the subsequence $x_T, x_{2T}, \dots$ for T large enough, as an approximately i.i.d. sequence of samples with the correct distribution.
The most common MCMC strategy for jumping from x_t to x_{t+1} is the Metropolis-Hastings random walk: from $x_t = x$, we generate a candidate y with probability $g(y|x)$, for some well-chosen distribution $g(\cdot|x)$. We then compute an acceptance probability

$$\alpha(y|x) = \min\left\{1, \frac{f(y)g(x|y)}{f(x)g(y|x)}\right\} \tag{5.12}$$

We set $x_{t+1} = y$ with probability α and x otherwise. The resulting transition probability for the Metropolis-Hastings random walk from x towards any $y \neq x$ are thus

$$m(y|x) = g(y|x)\alpha(y|x) = \min\left\{\frac{f(y)g(x|y)}{f(x)}, g(y|x)\right\} \tag{5.13}$$

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

$$p(x) = \frac{f(x)}{\sum_{x \in X} f(x)} \tag{5.14}$$

provided that any configuration x can be reached from any configuration x_0 in finitely many jumps with nonzero probability.

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

$$P(x|z) = \frac{P(z|x)P(x)}{\sum_{x \in X} P(z|x)P(x)} \tag{5.15}$$

$P(x)$ is sometimes called the prior distribution, 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. We can hope to sample the posterior thanks to MCMC techniques such as the Metropolis-Hastings random walk.

5.6 MC algorithms for decision problems

We would like to determine whether a polynomial P is identically zero or not.

Lemma 5.3. Schwartz-Zippel. If $P(x_1, \dots, x_n)$ is a polynomial of degree d, then the probability that $P(x_1, \dots, x_n) = 0$ for x_i randomly chosen in a finite set S, for all i, is at most $d/|S|$.

Proof. For $n = 1$, it is true by the fundamental theorem of algebra: univariate polynomials of degree d have at most d roots.
For $n \geq 1$, we write $P(x_1, \dots, x_n) = \sum_{i=1}^{d} x_1^i P_i(x_2, \dots, x_n)$ for some polynomials P_i of $n-1$ variables and of degree at most $d-i$. Let j be the highest index for which $P_j \not\equiv 0$. By induction hypothesis, if you pick $r_2, \dots, r_n \in S$ uniformly, then

$$Pr[P_j(r_2, \dots, r_n) = 0] \leq \frac{d-j}{|S|} \tag{5.16}$$

If $P_j(r_2, \dots, r_n) \neq 0$, then $P(x_1, r_2, \dots, r_n)$ is a univariate polynomial in x_1, of degree j, and

$$Pr[P(r_1, r_2, \dots, r_n) = 0 | P_j(r_2, \dots, r_n) \neq 0] \leq \frac{j}{|S|} \tag{5.17}$$

for a uniform r_1 in S. Let us denote event $A = "P_j(r_2, \dots, r_n) = 0"$ and $B = "P(x_1, r_2, \dots, r_n) = 0"$. Then,

$$Pr[B] = Pr[B|\bar{A}]Pr[\bar{A}] + Pr[B|A]Pr[A] \tag{5.18}$$

We already know that $Pr[B|\bar{A}] = \frac{j}{|S|}$ and $Pr[A] = \frac{d-j}{|S|}$. This gives

$$Pr[P(x_1, r_2, \dots, r_n) = 0] = Pr[B] = \frac{j}{|S|} Pr[\bar{A}] + \frac{d-j}{|S|} Pr[B|A] \le \frac{d-j}{|S|} + \frac{j}{|S|} = \frac{d}{|S|} \tag{5.19}$$

□

An algorithm using this idea has a probability of success

$$\begin{aligned} Pr[correct] &= Pr[\text{return } P \equiv 0 \text{ and } P \equiv 0] + Pr[\text{return } P \not\equiv 0 \text{ and } P \not\equiv 0] \\ &= Pr[\text{return } P \equiv 0 | P \equiv 0] Pr[P \equiv 0] + Pr[\text{return } P \not\equiv 0 | P \not\equiv 0] Pr[P \not\equiv 0] \\ &\ge Pr[P \equiv 0] + \left(1 - \frac{d}{|S|}\right) Pr[P \not\equiv 0] \ge 1 - \frac{d}{|S|} \end{aligned} \tag{5.20}$$

We may improve this using amplification of stochastic advantage: repeating k times improves the probability lower bound to $\left(1 - \frac{d}{|S|}\right)^k$.
Amplification of stochastic advantage is very powerful when errors on a decision problem are one-sided: there may be false negatives, but not false positives, or the contrary.

If we have a two-sided error, amplification of stochastic advantage still works: if $Pr[\text{True}|\text{it is true}] \ge 1/2 + \varepsilon$ and $Pr[\text{False}|\text{it is false}] \ge 1/2 + \varepsilon$. Then, we may proceed to a majority vote after n trials.
Let $X_i = 1$ if the i-th run is correct and 0 otherwise. Then, $\mathbb{E}[X_i] \ge 1/2 + \varepsilon$. Let us take the worst case: $\mathbb{E}[X_i] = 1/2 + \varepsilon$. The repeated algorithm is wrong if $\sum_i X_i < n/2$. But

$$\begin{cases} \mathbb{E}[\sum_i X_i] = n\left(\frac{1}{2} + \varepsilon\right) \\ Var[X_i] = \mathbb{E}[X_i^2] - \mathbb{E}[X_i]^2 = \mathbb{E}[X_i] - \mathbb{E}[X_i]^2 = \frac{1}{4} - \varepsilon^2 \\ \implies Var[\sum_i X_i] = n\left(\frac{1}{4} - \varepsilon^2\right) \end{cases} \tag{5.21}$$

Using the Central Limit Theorem, $\sum_i X_i \sim \mathcal{N}\left(n\left(\frac{1}{2} + \varepsilon\right), n\left(\frac{1}{4} - \varepsilon^2\right)\right)$ if n is large. Then, for example, if $n > 1/\varepsilon^2$ implies a probability of error smaller than 2.5%.

5.7 Las Vegas Algorithms

For a class of problems X, A is a

- Las Vegas algorithm if, for all input $x \in X$, if A returns a solution s, then s is a valid solution to x, but the runtime of A is a random variable;
- Monte Carlo algorithm if its output is randomized, and is not always a valid solution to the problem, but gives an approximate solution, or is correct with some probability.

Example with hash functions: the main problem is collisions. Ignoring them is a Monte Carlo approach, and storing the values elsewhere (e.g. linear probing) is a Las Vegas approach.

5.8 Random number generation

In a computer, randomness is simulated deterministically by generating pseudo-random numbers.
Let $f : X \to X$, $g : X \to Y$ and $s \in X$. A general algorithm is

Algorithm 10 Typical random number generator

```
1: $x_0 = s$;
2: for i do
3:     $x_{i+1} = f(x_i)$;
4:     $y_i = g(x_i)$;
5: end for
```

The output of the pseudo-random generator (PRG) is then $y = (y_0, \dots, y_n)$. f, g is a PRG if y "looks like" randomness when s is uniformly chosen in X. Another example is Blum-Blum-Schub:

Algorithm 11 BBS

```
1: Select random prime numbers $p$ and $q$ such that $p \equiv q \equiv 3 \mod 4$;
2: Let $N = pq$;
3: Let $s \in \mathbb{Z}_N^*$;
4: $f_N(x) = x^2 \mod N$ and $g(x) = x \mod 2$.
```

The same seed s will generate the same sequence of bits. Assuming that factoring N is difficult, this is a good PRG, i.e. any algorithm that distinguishes the output from true randomness can factor N.

5.9 Derandomization

Generating truly i.i.d. random bits is far from obvious. Therefore, the number of random bits required by an algorithm is a costly resource. 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.
There are two generic derandomization techniques:

- Use a PRG: cheap but still needs some randomness;
- Try all possible random values: remove all randomness but can be a polynomial time slowdown.

For example, consider the MaxCut problem for a graph $G = (V, E)$, with $|V| = n$ and $|E| = m$. A random algorithm consists in assigning each node either to the partition S or T, by drawing n i.i.d. random bits $r_1, \dots, r_n$.

$$\mathbb{E}(Cut(S,T)) = \sum_{e \in E} Pr[e \text{ is in the cut}] = \sum_{e \in E} \frac{1}{2} = \frac{m}{2} \geq \frac{MaxCut}{2} \tag{5.22}$$

This comes from the fact that

$$Pr[e \text{ is in the cut}] = Pr[r_u \in S]Pr[r_v \in T] + Pr[r_u \in T]Pr[r_v \in S] \tag{5.23}$$

Hence we only need pairwise independence between the random bits, and not full independence. This allows us to use less bits. Indeed, consider for example the variables $X = Y \oplus Z$, with $X, Y, Z \in \{0, 1\}$. If Y and Z are pairwise independent, then so are X and Y, and X and Z; but X, Y, Z are not jointly independent.
Suppose now that we can generate $r_1, \dots, r_k$ i.i.d. random bits. For any subset $A \subseteq \{1, \dots, k\}$ such that $A \neq \emptyset$, we define $s_A = \sum_{i \in A} r_i \mod 2$. There are $2^k - 1$ such subsets A with associated uniformly distributed bits s_A, which are all pairwise independent. Indeed, $\forall A, B \subseteq \{1, \dots, k\}$ such that $A \neq B, A, B \neq \emptyset$, s_A and s_B are independent since $s_A = s_B \oplus s_{A \setminus B} \oplus s_{B \setminus A}$ and if $A \setminus B \neq \emptyset$, then $s_{A \setminus B}$ is independent from s_B and that implies independence of s_A and s_B. The same argument applies when $B \setminus A \neq \emptyset$.
For the MaxCut problem, since we need n pairwise independent pairs of random bits, it is sufficient to draw $k = \lceil \log_2(n+1) \rceil$ i.i.d. random bits. To derandomize completely, we need to try all of the 2^k different possibilities for $r_1, \dots, r_k$, which is then linear in n since $2^k = \Theta(n)$.

Computability and decidability

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 A): $f : A^* \to A^* : w \to f(w)$ or undefined. A decision problem P is given by a total function of the subset: $A^* \to \{\text{yes}, \text{no}\}$. A Python machine is given a syntatically correct, deterministic Python code that takes a string in A^* as input, and that outputs a string in A. The machine has infinite memory, meaning that the code can run without ever overflowing the memory. In doing so, the machine computes a function that associates to any given input string the corresponding output string. As the code may never stop, this function is possibly partial. This function is the problem computed by the Python machine. If the function is total with output in $\{\text{yes}, \text{no}\}$, we say that the corresponding decision problem is **decided** by the Python machine. A problem is **computable** if it is computed by some Python machine.

6.1 Definition of a Turing machine

A Turing machine has the following elements:

- An infinite tape as an unlimited memory;
- A head that reads and writes symbols on the cells and moves around the tape;
- A specific blank symbol $\sqcup$ proper to the machine;
- Initially, the tape contains only the input string and is blank elsewhere;
- Storing information can be done by writing on the tape through the head;
- Reading information can be done by moving left or right the head over it;
- The machine continues to compute until it decides to produce an output;
- The outputs "ACCEPT" and "REJECT" are obtained by entering designated states: the accepting state q_{accept} and the rejecting state is q_{reject};
- If the machine does not enter one of these states, it loops, never halting;
- The number of states must be finite.

6.1.1 Formal definition

Definition 6.1. A Turing machine is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are finite sets and

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

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

The Turing machine (TM) M receives its input $w = w_1 \ldots w_n \in \Sigma^*$ on the leftmost n squares of the tape. Furthermore, M never tries to move its head to the left of the left-hand end of the tape (it stays in the same place even though δ indicates L).
As a TM computes, changes occur in the current state, in the tape content and in the head location. A setting of these 3 items is called a configuration of the TM.
For a state q and strings u, v over Γ, we note uqv for the configuration where the current state is q, the tape content is uv, and the current head location is the first symbol of v.

Example 6.2. Assume we are in the configuration uaq_ibv for $u, v \in \Gamma^*$, $a, b \in \Gamma$ and $q_i \in Q$. Then, if $\delta(q_i, b) = (q_j, c, L)$, the next configuration will be uq_jacv.

A Turing machine M accepts input w if a sequence of configurations $c_1, \ldots, c_k$ exists where

- $c_1 = q_0w$ is the start configuration of M on input w;
- each c_i yields c_{i+1};
- c_k is an accepting configuration.

The collection of strings that M accepts is the language of M, or the language recognized by M, denoted $\mathcal{L}(M)$. A language is **Turing-recognizable** if some Turing machine recognizes it.
A decider is a Turing machine that halts on all inputs. A decider that recognizes some language decides it. A language is **Turing-decidable** if some TM decides it.

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

6.1.2 Example

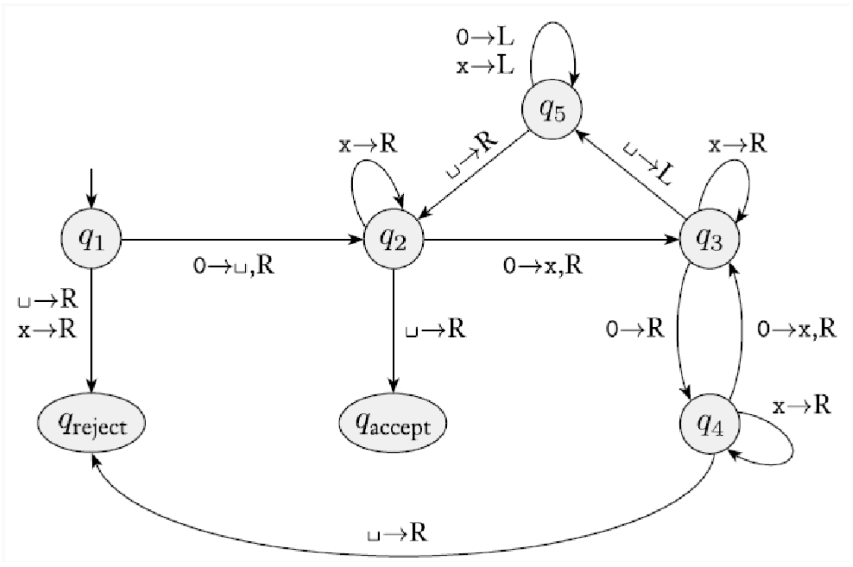


Figure 6.1: Example of a Turing machine

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

- $Q = \{q_1, \ldots, q_5, q_{accept}, q_{reject}\}$ with q_1 the initial state;
- $\Sigma = \{0\}$ and $\Gamma = \{0, x, ␣\}$;
- δ is the Figure 6.1.

Here is the breakdown of the steps:

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

What if we allowed the head to stay put? This is equivalent to the previous machine, as one model can simulate the other, and conversely: they are equivalent as they recognize the same languages.

6.2 Multitape Turing Machines

Definition 6.3. A multitape Turing machine is a TM with several tapes, each with its own head. For k tapes, the transition function then becomes

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k \tag{6.1}$$

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

6.3 Enumerator

Definition 6.4. An **enumerator** is a Turing machine that always starts with a blank input on its tape, and that has an attached printer to print strings. Every time it wants to print a string to the list, it sends it to the printer. An enumerator does not have to halt, and may print an infinite list of strings. The language it enumerates is the collection of all strings that is printed out, and it can generate the strings in any order, possibly with repetitions.

Theorem 6.5. A language is Turing-recognizable iff some enumerator enumerates it.

Proof. $\Longrightarrow$: Let E be an enumerator that enumerates A. The Turing machine that we define is M = "on input w, run E; every time that E outputs a string, compare it with w. If it is the same, accept; else continue."
$\Longleftarrow$: Let M be a TM that recognizes A. Say $s_1, \ldots$ is a list of all strings in Σ^*. We take as enumerator E : "ignore the input. Repeat the following for $i = 1, 2, \ldots$: run M for i steps on each input $s_1, \ldots, s_i$. If any computation accepts, then print out the corresponding s_j."
If M accepts a particular string s, eventually it will appear on the list generated by E[2]. □

6.4 Non-deterministic Turing Machines

A non-deterministic TM proceeds according to several possibilities. The transition function returns a probability distribution over $Q \times \Gamma \times \{L, R\}$. The computation of a non-deterministic TM is a tree whose branches correspond to different possibilities for the machine. If some branch of the computation leads to the ACCEPT state, the machine accepts.

Theorem 6.6. Every non-deterministic TM has an equivalent deterministic TM.

Proof. Let N be a non-deterministic TM. We design D, a TM inspecting the branches of N's tree by using BFS. Let b be the size of the largest set of possible choices given N's transition function. To every node in the tree, we assign an address as a string over $\Gamma_b = \{1, \ldots, b\}$.
D has three tapes: the first correspond to the input tape, the second to the simulation tape, and the last to the address tape.

[2] In fact, it will apear on the list infinitely many times because M runs from the beginning on each string for each repetition.

1. Initially, tape 1 contains the input w, and tapes 2 and 3 are empty;
2. Then, we copy tape 1 on tape 2 and initialize the string on tape 3 to be Γ_b;
3. We use tape 2 to simulate N with input w on one branch of its non-deterministic computation. Before each step of N, we consult the next symbol o, tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this non-deterministic choice is invalid, abort this branch by going to the next step. Also go to next step if a REJECT configuration is encountered. If an ACCEPT configuration is encountered, accept the input.
4. Replace the string on tape 3 with the next string in the string ordering. Simulate the next branch of N's computation by going back to step 2.

□

Corollary 6.7. A language is Turing-recognizable iff a non-deterministic TM recognizes it.

The Church-Turing thesis is that our intuitive notion of algorithm is captured by a TM. Notice that TMs can always be designed to first check the validity of the encoding.

6.4.1 10th Hilbert's problem

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

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

$$D_n = \{P \in \mathbb{Z}[x_1, \dots, x_n] \mid \exists a \in \mathbb{Z}^n \ : \ P(a) = 0\} \tag{6.2}$$

6.5 Universal Turing Machine

Definition 6.8. A universal TM U is a TM that solves the following *halting problem*:

- Input: a description of a TM M and a description of a finite word on initial tape w. We write $\langle M, w\rangle$.
- Output: If M stops when the initial tape is w, U outputs the finite content of the tape when M stops, i.e. 6.3. If T never stops, undefined.

$$U(\langle M, w\rangle) = M(w) \tag{6.3}$$

Hence, the language $A_{TM} := \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}$ is Turing-recognizable, but it is not decidable, since the above constructed TM will loop on input $\langle M, w\rangle$ if M loops on w. A TM can always be described as a finite word in an alphabet and can thus be used as an input to a TM.

The notation $\langle \cdot, \cdot \rangle$ simply means input, it can contains several arguments. A Universal Turing Machine outputs the same thing as the TM it has as input for the given string w. It is a bit like a translator that extracts the meaning of a sentence independently from the language.

6.6 Decidability

The set of all TM is countable due to their finite encoding. The set $\mathcal{L}$ of all languages over Σ has the size of B, the set of infinite binary sequences, and is therefore uncountable.

Theorem 6.9.

$$A_{TM} := \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\} \tag{6.4}$$

is undecidable.

Proof. By contradiction, let H be a decider for A_{TM}. Then

$$H(\langle M, w\rangle) = \begin{cases} \text{ACCEPT if } M \text{ accepts } w \\ \text{REJECT otherwise} \end{cases} \tag{6.5}$$

Let us build a TM D that, on input $\langle M\rangle$, runs H on $\langle M, \langle M\rangle\rangle$ and halts as $H(\langle M, w\rangle)$ is false, i.e. D accepts if H yields a rejecting configuration, and conversely. D is a TM such that

$$\begin{aligned} D(\langle M\rangle) = \overline{H(\langle M, \langle M\rangle\rangle)} &= \begin{cases} \text{ACCEPT if } M \text{ does not accept } \langle M\rangle \\ \text{REJECT otherwise} \end{cases} \\ &= \begin{cases} \text{ACCEPT if } H(\langle M, \langle M\rangle\rangle) = \text{ REJECT} \\ \text{REJECT if } H(\langle M, \langle M\rangle\rangle) = \text{ ACCEPT} \end{cases} \end{aligned} \tag{6.6}$$

But then, $D(\langle D\rangle)$ is in contradiction with itself:

$$\begin{aligned} D(\langle D\rangle) &= \begin{cases} \text{ACCEPT if } H(\langle D, \langle D\rangle\rangle) = \text{ REJECT} \\ \text{REJECT if } H(\langle D, \langle D\rangle\rangle) = \text{ ACCEPT} \end{cases} \\ &= \begin{cases} \text{ACCEPT if } D(\langle D\rangle) = \text{ REJECT} \\ \text{REJECT if } D(\langle D\rangle) = \text{ ACCEPT} \end{cases} \end{aligned} \tag{6.7}$$

We thus show that the decider cannot decide itself. □

Theorem 6.10. A language is **decidable** iff it is Turing-recognizable and co-Turing recognizable[3].

Proof. $\Longrightarrow$: Trivial.
$\Longleftarrow$: Let M_1 be a recognizer for A and M_2 be a recognizer for $\bar{A}$. Build M that runs M_1 and M_2 in parallel, one step at a time, on input w. If M_1 accepts, ACCEPT. If M_2 acccepts, REJECT. □

[3] $\forall w \in \mathcal{L}$, M halts on $w \Longrightarrow \mathcal{L}$ is recognizable.
$\forall w \notin \mathcal{L}$, M halts on $w \Longrightarrow \mathcal{L}$ is co-recognizable.

Corollary 6.11. $\overline{A_{TM}} := \{\langle M, w\rangle | M \text{ is a TM and } M \text{ does not accept } w\}$ is not Turing-recognizable.
"M does not accept w" means that it either rejects w, or does not halt on w. If it does not halt, it is not Turing-recognizable by definition.

Theorem 6.12. Comparing two programs (determining if $P_1(x) = P_2(x), \forall x$) is undecidable.

Proof. We will reduce the halting problem to the comparison problem. Suppose that we want to check if a program P halts on x. Then, we create an instance of the comparison problem by defining the two programs: $P_1(y) := P(x)$ for all y (it is a constant function), and $P_2(y)$ such that it does not halt for all y. If P does not halt on x, then $P_1(y) = P_2(y)$ for all y. If P halts on x, then $P_1(y) \neq P_2(y)$ for all y. This implies that if the comparison problem is computable, so is the halting problem. But halting is undecidable, so the comparison problem is as well.
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. □

Complexity classes

Computability theory looks at what it means to compute; the fundamental limits of computation. Complexity theory looks at how efficiently we can solve problems and for which of those problems we can find a solution in "reasonable time".

7.1 Time complexity

Let M be a deterministic TM that halts on all inputs. The running time (or time complexity) of M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M uses on any input of lengt n.
Let $t : \mathbb{N} \to \mathbb{R}^+$. The time complexity class $TIME(t(n))$ is the collection of languages decidable by a $\mathcal{O}(t(n))$ time TM.

Proposition 7.1. *Let $t(n) \geq n$. Every $t(n)$ time multitape TM has an equivalent $\mathcal{O}(t^2(n))$ time single-tape TM.*

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

Proposition 7.2. *Let $t(n) \geq n$. Every $t(n)$ time non-deterministic single-tape TM has an equivalent $2^{\mathcal{O}(t(n))}$ time deterministic single-tape TM, e.g. using BFS on the tree of the non-deterministic TM.*

7.2 The class P

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM. In other words,

$$P = \bigcup_k TIME(n^k) \tag{7.1}$$

This class is the same for all reasonable languages we may use to code the algorithm, as coding on a TM involves only a polynomial slowdown with respect to higher level languages.
We define the complement of P: $coP := \{L | \bar{L} \in P\}$. The complementary of a decision problem A, $\bar{A}$, is defined by the property that the positive instances of $\bar{A}$ are the negative instances of A and conversely.

Theorem 7.3. *$coP = P$.*

Theorem 7.4. There is a decidable problem not in P.

Proof. We enumerate a list of Python programs that solve all problems in P? We know that every problem in P is solved by a Python machine in this list, and every Python machine in thislist runs in polynomial time. Let us call this list $(M_i)_{i\in\mathbb{N}}$ and let us suppose that we enumerate the possible inputs as $1, 2, 3, 4, \dots$
We create a problem A with input n and that outputs YES if $M_n(n) = No$, and NO if $M_n(n) = Yes$. This problem outputs the opposite of the diagonal of the table (problems $\times$ inputs), and is not in P because the diagonal is not a row of this table. This implies that we found a problem A not in P. $\square$

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

$$EXPTIME = \bigcup_k TIME(2^{n^k}) \tag{7.2}$$

→ Note: the diagonal problem constructed in the proof above is in EXPTIME. In fact, we can repear the argument for any time-complexity class, and create a hierarchy of larger and larger time-complexity classes.

PSPACE is the class of languages that are decidable by an algorithm using a polynomial amount of space.

Theorem 7.5. *$P \subseteq PSPACE$.*

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. Hence, if $N \in \mathcal{O}(n^d)$, then $n + N \in \mathcal{O}(n^d)$ for all d. $\square$

Theorem 7.6. *$PSPACE \subseteq EXPTIME$.*

7.3 The class NP

Definition 7.7. A verifier for a language $\mathcal{L}$ is an algorithm V where

$$\mathcal{L} = \{w | V \text{ accepts } \langle w, c\rangle \text{ for some string } c\} \tag{7.3}$$

We measure the time of a verifier only in terms of the length of w, so a polynomial-time verifier runs in polynomial time in the length of w.

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

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

NTIME(t(n)) is the class of languages that are decided by an $\mathcal{O}(t(n))$ time non-deterministic TM.

$$NP = \bigcup_k NTIME(n^k) \tag{7.4}$$

Unlike P and PSPACE, NP is not closed under complementation: $NP \neq coNP :=$ {problems whose complementary is in NP}.

Theorem 7.10. *$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$.*

The only thing that we are sure of is that $P \subsetneq EXPTIME$.

7.3.1 Example – SAT

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

$$\exists?\{x_1, x_2, x_3\} \text{ such that } \Big((x_1 \wedge x_2) \vee \overline{(x_2 \vee x_3)}\Big) \wedge x_3 = 1 \qquad (7.5)$$

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

Conjunctive normal form

We define the conjunctive normal form as

$$\phi = \bigwedge_i c_i \qquad c_i = \bigvee_j l_j \qquad l_j = \begin{cases} x_k \\ \overline{x_k} \end{cases} \qquad (7.6)$$

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

The SAT problem is NP-complete.

7.4 NP-completeness

A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial time TM exists that halts with $f(w)$ on its tape when started on any input w.
A language A is polynomial time reducible to a language B, written $A \leq_P B$ if there exists a polynomial time computable function f such that for every w, $w \in A \iff f(w) \in B$. We call f the reduction of A to B.

Theorem 7.11. If $A \leq_P B$ and $B \in P$, then $A \in P$.

Definition 7.12. A language B is NP-complete if $B \in NP$ and every $A \in NP$ is polynomial-time reducible to B.

Theorem 7.13. If B is NP-complete and $B \leq_P C$ for some $C \in NP$, then C is NP-complete.

We define the problem of satisfiability of a Boolean formula SAT in the following way:

- Instance: a boolean formula;
- Output: yes if the formula is satisfiable, i.e. there exists a truth value for the variables such that the formula evaluates to true.

Theorem 7.14. Cook-Levin. SAT is NP-complete.

Proof.
- $SAT \in NP$. Indeed, the certificate that a boolean formula is satisfiable is a list of TRUE/FALSE values for all variables, and it can be checked in polynomial time that it makes a formula evaluated to YES.
- Any language in NP is polynomial reducible to SAT, details are in CM12.

□

7.5 Reduction

The usual way to prove that a given problem S is NP-complete is to show that $S \in NP$ and that for some other NP-complete problem T, we have $T \leq_P S$. The 3-SAT problem is a restriction of SAT to a subset of instances, where we have conjunctions of clauses, which are disjunctions of 3 (possibly negated) variables.

Theorem 7.15. The 3-SAT problem is NP-complete.

Indeed, it is in NP and SAT is polynomial reducible to 3-SAT, using the Conjunctive Normal Form, possibly adding some useless variables.

Theorem 7.16. The clique problem is NP-complete.

Proof. We may reduce the 3-SAT problem to the clique problem in polynomial time. For this, we construct a graph with a triple of nodes for each clause. There is no edge inside a triple, nor between nodes with contradictory labels. The problem becomes to find a k-clique in the graph, where k is the number of clauses. Indeed, if the 3-SAT instance is satisfiable, all the variables allowing the satisfiability will be connected in the graph, forming a k-clique. Conversely, if there is a k-clique, then we have a solution for the 3-SAT problem.
Finally, the clique problem is in NP since the certificate that there exists a k-clique in the graph G is a list of the k nodes of G forming a clique, which can be checked in polynomial time. □