Super Study Guide

Algorithms & Data Structures

Afshine Amidi and Shervine Amidi

Cover Design: Afshine Amidi and Shervine Amidi

© 2022 Afshine Amidi and Shervine Amidi
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the authors.

First edition

We would like to dedicate this book to our beloved grandparents,
Dr. Mohammad Sadegh Azimi and Dr. Atieh Azimi,
who will always stay in our hearts.

A lifetime flies as quickly as the pages
of a book flipped through by the wind.
Enjoy it as much as you can.
– Saeb Tabrizi

Contents

SECTION 1

Foundations

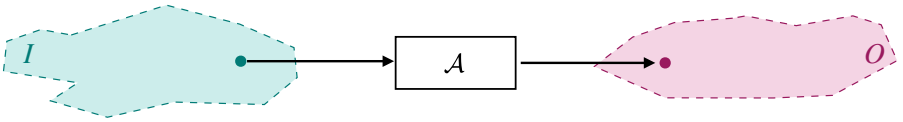
In this first section, we will start with the basic concepts of algorithms, along with mathematics notions that are used in many problems.

1.1 Algorithmic concepts

In this part, we will study the main types of algorithms and learn how to quantify their complexities.

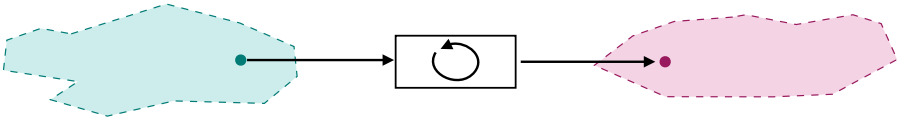
1.1.1 Overview

❒ **Algorithm** – Given a problem, an algorithm $\mathcal{A}$ is a set of well-defined instructions that runs in a finite amount of time and space. It receives an input I and returns an output O that satisfies the constraints of the problem.



As an example, a problem can be to check whether a number is even. In order to do that, an algorithm could be to check whether the number is divisible by 2.

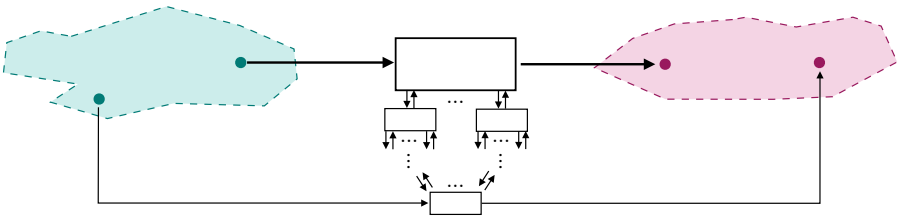
❒ **Iteration** – An iterative algorithm is an algorithm that runs through a sequence of actions. It is characterized by either a `for` or a `while` loop.



Suppose we want to return the sum of all of the elements of a given list. An example of an iterative algorithm would be to sequentially add each element of the list to a variable, and return its final value.

❒ **Recursion** – A recursive algorithm uses a function that calls itself. It is composed of the following components:

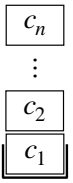
- *Base case*: This is the set of inputs for which the outputs are known.
- *Recursive formula*: The answer of the current step is based on function calls relying on previous steps, eventually using the base case answer.



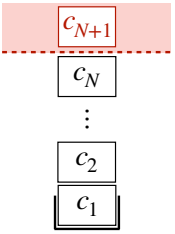
A classic problem is to compute the power of a number x^n without explicitly using the power operation. In order to do that, a recursive solution could rely on the following cases:

$$\begin{cases} x^0 = 1 & \text{is known} \\ x^n = x^{\frac{n}{2}} \times x^{\frac{n}{2}} & \text{when } n \in \mathbb{N}^* \text{ is even} \\ x^n = x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}} & \text{when } n \in \mathbb{N}^* \text{ is odd} \end{cases}$$

❒ **Call stack** – In a recursive algorithm, the space used by function calls c_i is called the stack space.

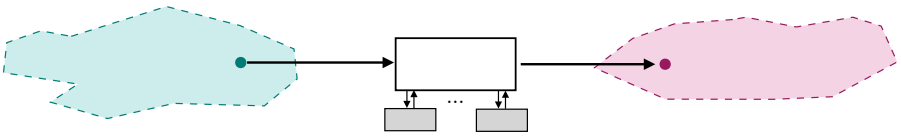


❒ **Stack overflow** – The problem of stack overflow occurs when a recursive algorithm uses more stack space than the maximum allowed N.



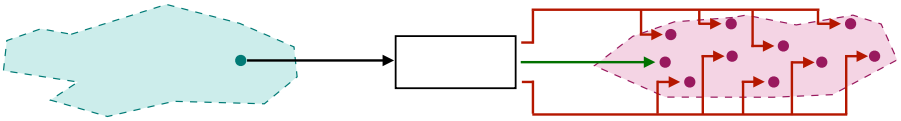
A solution to circumvent this bottleneck is to convert the code from being recursive to being iterative so that it relies on memory space, which is typically bigger than stack space.

❒ **Memoization** – Memoization is an optimization technique aimed at speeding up the runtime by storing results of expensive function calls and returning the cache when the same result is needed.



1.1.2 Types of algorithms

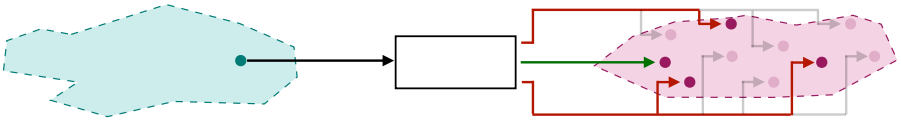
❒ **Brute-force** – A brute-force approach aims at listing all the possible output candidates of a problem and checking whether any of them satisfies the constraints. It is generally the least efficient way of solving a problem.



To illustrate this technique, let's consider the following problem: given a sorted array A, we want to return all pairs of elements that sum up to a given number.

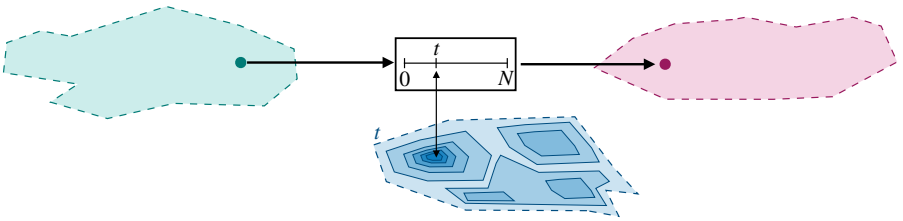
- A brute-force approach would try all possible pairs (a_i, a_j) and return those that sum up to that number. This method produces the desired result but not in minimal time.
- A non-brute-force approach could use the fact that the array is sorted and scan the array using the two-pointer technique.

❒ **Backtracking** – A backtracking algorithm recursively generates potential solutions and prunes those that do not satisfy the problem constraints. It can be seen as a version of brute-force that discards invalid candidates as soon as possible.



As an example, the N-Queens problem aims at finding a configuration of N queens on a $N \times N$ chessboard where no two queens attack each other. A backtracking approach would consist of placing queens one at a time and prune solution branches that involve queens attacking each other.

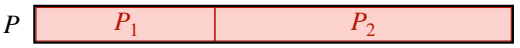
❒ **Greedy** – A greedy algorithm makes choices that are seen as optimal at every given step. However, it is important to note that the resulting solution may not be optimal globally. This technique often leads to relatively low-complexity algorithms that work reasonably well within the constraints of the problem.



To illustrate this concept, let's consider the problem of finding the longest path from a given starting point in a weighted graph. A greedy approach constructs the final path by iteratively selecting the next edge that has the highest weight. The resulting solution may miss a longer path that has large edge weights "hidden" behind a low-weighted edge.

❒ **Divide and conquer** – A divide and conquer (D&C) algorithm computes the final result of a problem by recursively dividing it into independent subproblems:

- *Divide*: The problem is divided into several independent subproblems.



- *Conquer*: Each subproblem is solved independently.

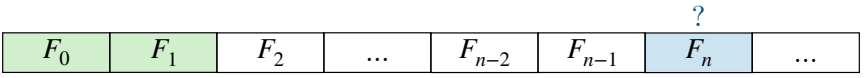
$$P_i \rightarrow S_i$$

- *Combine*: The result of each subproblem is combined to find the final answer.

Algorithms following the D&C principle include sorting algorithms such as merge sort and quick sort.

❒ **Dynamic Programming** – Dynamic programming (DP) is a method of problem resolution that relies on finding answers to overlapping subproblems.

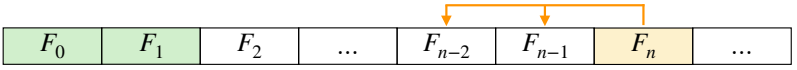
A common example of problem resolution using DP is the computation of Fibonacci numbers.



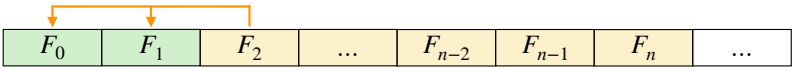
There are two main approaches:

Top-down This approach finds the target value by recursively computing previous values.

- *Step 1*: Try computing the desired value F_n and notice that it is based on previous values.



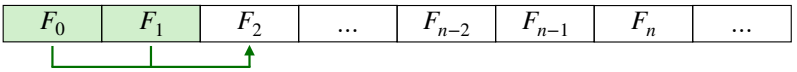
- *Step 2*: Try computing the previous values, which themselves rely on earlier values, some of which may have already been computed. In this case, we can use memoization to avoid duplicate operations.



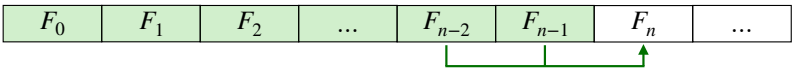
- *Step 3*: Use the newly computed values to deduce F_n.

Bottom-up This approach starts from already-known results and iteratively computes succeeding values until it reaches the target value.

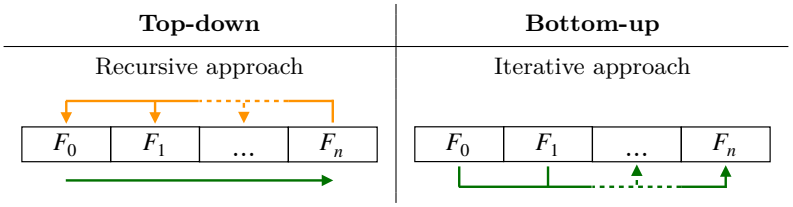
- *Step 1*: Compute F_0, F_1, F_2, etc. in a predetermined way. These values are typically stored in an array.



- *Step 2*: Deduce F_n.



In summary, the two main ways of solving a problem with DP are:



Remark: A key difference between DP and D&C strategies is that DP relies on overlapping subproblems whereas D&C bases itself on independent subproblems.

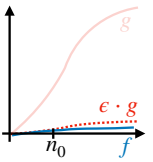
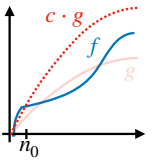
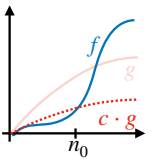
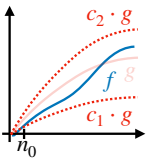
1.1.3 Complexity

❒ **Definition** – The concept of complexity is used to quantify the efficiency of an algorithm. There are two types of complexities:

- **Time**: How many operations are made?
- **Space**: How much extra space is needed?

Both measures are usually given as a function of the input size n, although other parameters can also be used.

❒ **Notations** – The complexity f of an algorithm can be described using a known function g with the notations described in the table below:

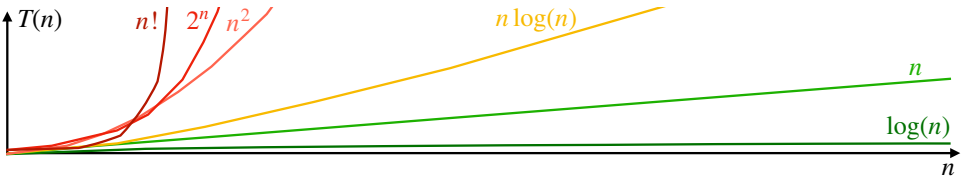
Notation	Definition	Meaning	Illustration
$f = o(g)$ *"little oh of g"*	$\forall \epsilon > 0, \exists n_0, \forall n \geqslant n_0$ $\lvert f(n)\rvert \leqslant \epsilon \lvert g(n)\rvert$	Negligible compared to g $f(n) \underset{n\to+\infty}{\ll} g(n)$	
$f = \mathcal{O}(g)$ *"big oh of g"*	$\exists c > 0, \exists n_0, \forall n \geqslant n_0$ $\lvert f(n)\rvert \leqslant c \lvert g(n)\rvert$	Upper-bounded by g $f(n) \underset{n\to+\infty}{\leqslant} g(n)$	
$f = \Omega(g)$ *"omega of g"*	$\exists c > 0, \exists n_0, \forall n \geqslant n_0$ $\lvert f(n)\rvert \geqslant c \lvert g(n)\rvert$	Lower-bounded by g $f(n) \underset{n\to+\infty}{\geqslant} g(n)$	
$f = \Theta(g)$ *"theta of g"*	$\exists c_1, c_2 > 0, \exists n_0, \forall n \geqslant n_0$ $\lvert f(n)\rvert \geqslant c_1 \lvert g(n)\rvert$ $\lvert f(n)\rvert \leqslant c_2 \lvert g(n)\rvert$	Similar to g $f(n) \underset{n\to+\infty}{\sim} g(n)$	

Remark: The big oh notation is frequently used to describe the time and space complexity of a given algorithm.

❒ **Orders of magnitude** – The table below highlights the main kinds of runtime complexities $T(n)$ as a function of the input size n:

Complexity	$T(n)$ for different n			**Example of application**
	10^1	10^2	10^3	
$\mathcal{O}(1)$ *"constant"*	1			*Hash table lookup*: Accessing the value of a given key does not depend on the size of the hash table.
$\mathcal{O}(\log(n))$ *"logarithmic"*	1	2	3	*Binary search*: The search space is divided by 2 at each iteration.
$\mathcal{O}(n)$ *"linear"*	10^1	10^2	10^3	*Linear search*: All elements of the input are visited.
$\mathcal{O}(n\log(n))$ *"linearithmic"*	10^1	2×10^2	3×10^3	*Merge sort*: The array is broken down by two at every step and each step requires all elements to be checked.
$\mathcal{O}(n^2)$ *"quadratic"*	10^2	10^4	10^6	*Bubble sort*: Each pair of elements of the input array is checked.
$\mathcal{O}(2^n)$ *"exponential"*	$\sim 10^3$	$\sim 10^{30}$	$\sim 10^{301}$	*0/1 knapsack problem*: The naive approach would consist of trying out every combination of items in the final set.
$\mathcal{O}(n!)$ *"factorial"*	$\sim 10^6$	$\sim 10^{158}$	$\sim 10^{2567}$	*Traveling salesman problem*: The naive approach consists of trying all possible permutations of cities to visit.

The following graph shows the difference in their evolutions:



As a rule of thumb, we have:

$$\mathcal{O}(1) < \mathcal{O}(\log(n)) < \mathcal{O}(n) < \mathcal{O}(n\log(n)) < \mathcal{O}(n^2) < \mathcal{O}(2^n) < \mathcal{O}(n!)$$

❒ **Master theorem** – The master theorem gives an explicit solution of a runtime $T(n)$ that satisfies a recursive relationship of the form below:

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)}$$

The solution depends on the relationship between $a \in \mathbb{N}^*$, $b \in \mathbb{N}^* \backslash \{1\}$ and $d \geqslant 0$:

Case	**Solution** $T(n)$	**Example of application**
$d < \log_b(a)$	$\Theta(n^{\log_b(a)})$	*Binary tree traversal*: After a node is visited, we move on to its left and right subtrees. $T(n) = 2T\left(\frac{n}{2}\right) \quad \longrightarrow \quad T(n) = \Theta(n)$
$d = \log_b(a)$	$\Theta(n^d \log(n))$	*Binary search*: The search space is divided by 2 at each pass. $T(n) = T\left(\frac{n}{2}\right) \quad \longrightarrow \quad T(n) = \Theta(\log(n))$
$d > \log_b(a)$	$\Theta(n^d)$	$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3) \quad \longrightarrow \quad T(n) = \Theta(n^3)$

❒ **Problem complexity** – Problems can be divided into classes that quantify how hard it is to solve them and to verify whether a proposed solution works. The table below presents two well-known classes:

Problem class	**Description**	**Example**
P *polynomial time*	The problem can be solved in polynomial time.	*Sorting*: An array of n elements can be sorted in $\mathcal{O}(n^2)$ time with selection sort.
NP *nondeterministic polynomial time*	A solution to the problem can be verified in polynomial time.	*Traveling salesman problem*: Given a path of n cities and a target value, we can determine whether the length of the path is lower or equal than the target value in $\mathcal{O}(n)$ time.

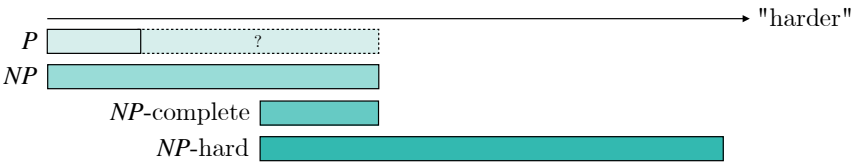
We note the following:

- P $\subseteq$ NP: If it is possible to find a solution in polynomial time, then it is possible to verify a candidate in polynomial time by computing the solution directly.

- P $\overset{?}{\supseteq}$ NP: It is unclear whether being able to verify a solution in polynomial time implies that we can solve the problem in polynomial time. The general consensus is that P $\not\supseteq$ NP (hence P $\neq$ NP) but this has not been formally proven yet.

Any problem in NP can be reduced in polynomial time to the following set of problems:

- NP-hard problems, that are at least as hard as NP problems
- NP-complete problems, that are NP-hard problems in NP



1.2 Mathematical concepts

In this part, we will dig into important mathematical notions and results spanning the fields of combinatorics and mathematical analysis, along with some basics on bit manipulation.

1.2.1 Combinatorics

❒ **Factorial** – The factorial $n!$ of a given integer n is defined as follows:

$$\boxed{n! \triangleq n \times (n-1) \times ... \times 2 \times 1}$$

Remark: By convention, $0! = 1$.

❒ **Binomial coefficient** – For given integers $0 \leqslant k \leqslant n$, the notation $\binom{n}{k}$ is read *"n choose k"* and is called a binomial coefficient. It is defined as follows:

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

For $k, n \in \mathbb{N}^*$, Pascal's rule gives a relationship between neighboring coefficients:

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

The evolution of binomial coefficients with respect to k and n can be visualized using Pascal's triangle:

n \ k						
	1					
	1	1				
	1	2	1			
	1	3	3	1		
	1	4	6	4	1	
	1	5	10	10	5	1

❒ Binomial theorem – The binomial theorem states that for $x, y \in \mathbb{R}$ and $n \in \mathbb{N}$, we have:

$$\boxed{(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k}$$

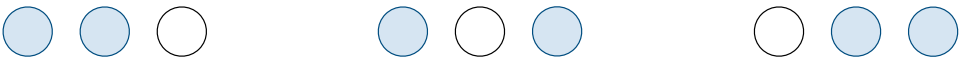
We note that when $x = y = 1$, we have:

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

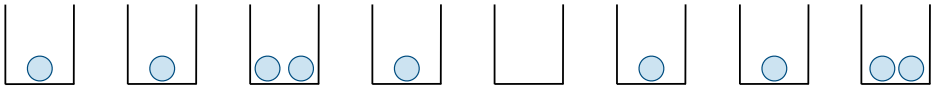
❒ Combination – A combination is an arrangement of k objects from a pool of n objects where the order does not matter. The number of such arrangements is given by $C(n, k)$:

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

For $k = 2$ choices among $n = 3$ elements, there are $C(3, 2) = 3$ possible combinations.



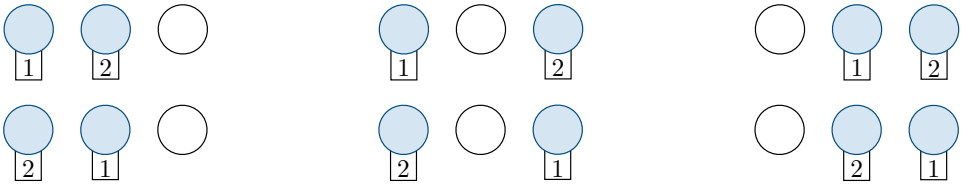
❒ Pigeonhole principle – Suppose n items in m containers, with $n > m$. The pigeonhole principle states that at least one container has more than one item.



❒ **Permutation** – A permutation is an arrangement of k objects from a pool of n objects where the order matters. The number of such arrangements is given by $P(n,k)$:

$$\boxed{P(n,k) = C(n,k) \times k! = \frac{n!}{(n-k)!}}$$

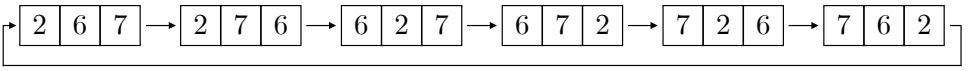
For $k = 2$ choices among $n = 3$ elements, there are $P(3,2) = 6$ possible permutations.



Remark: For $0 \leqslant k \leqslant n$, we have $P(n,k) \geqslant C(n,k)$.

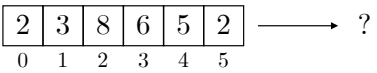
❒ **Lexicographic ordering** – Given a set of n elements, the lexicographic ordering sorts the resulting $P(n,n) = n!$ permutations in an alphabetical way.

To illustrate this, the lexicographic ordering of the 3! permutations of $\{2,6,7\}$ is as follows:



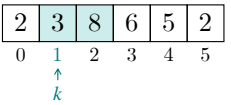
We note that this way of ordering permutations can be applied to sets containing any type of ordered elements, such as integers and characters.

❒ **Next lexicographic permutation** – Given an array $A = [a_0, ..., a_{n-1}]$ representing a permutation of the set of n elements $\{a_0, ..., a_{n-1}\}$, the goal of this problem is to find the next permutation in the lexicographic order.



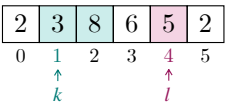
The solution is found in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space as follows:

- *Step 1*: Find the largest index k, such that $a_k < a_{k+1}$.

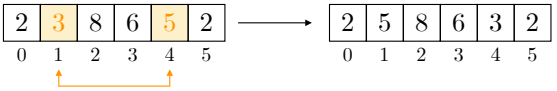


If there is no such k, go directly to *Step 4* and use $k = -1$.

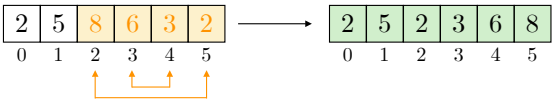
- *Step 2*: Find the largest index $l > k$, such that $a_k < a_l$.



- *Step 3*: Swap a_k and a_l.

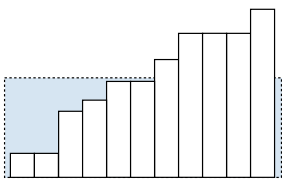
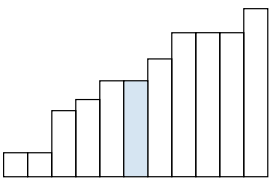
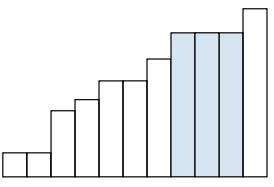


- *Step 4*: Reverse the subarray that starts from index $k + 1$.



1.2.2 Mathematical analysis

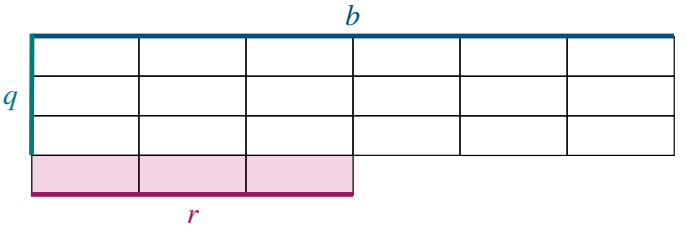
❒ **Statistics** – Given a set of ordered observations $x_1 \leqslant ... \leqslant x_n$, we can compute the following quantities:

Mean	Median		Mode
$\frac{x_1 + ... + x_n}{n}$	*n is odd* $x_{\frac{n+1}{2}}$	*n is even* $\frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2}$	$\underset{x_i}{\text{argmax}}\ \#x_i$
Sum of values divided by the number of values. Also known as average.	Middle value of the sorted set of observations.		Value that occurs the most frequently in the set of observations.
			

❒ **Euclidean division** – The Euclidean division of $a \in \mathbb{N}$ by $b \in \mathbb{N}^*$ finds the unique values of $q \in \mathbb{N}$ and $r \in \{0, ..., b-1\}$ such that:

$$\boxed{a = bq + r}$$

a is called the dividend, b the divisor, q the quotient and r the remainder.



This can be written in a different way using the modulo notation:

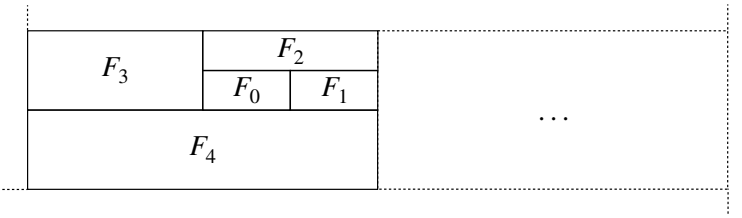
$$\boxed{a \equiv r \pmod{b}}$$

which is read "*a is equal to r modulo b*".

❒ **Fibonacci sequence** – Fibonacci numbers F_n are a sequence defined by:

$$\boxed{F_n = F_{n-1} + F_{n-2}} \quad \text{with } F_0 = 0 \text{ and } F_1 = 1$$

They can be geometrically represented as follows:



Remark: The first ten numbers of the sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

1.2.3 Bit manipulation

❒ **Integer representation** – The decomposition of an integer n in base b is unique and given by:

$$\boxed{n = \sum_{k=0}^{+\infty} \mathtt{n}_k b^k} \quad \text{where } \mathtt{n}_k \in \{0, ..., b-1\}$$

The representation of n in base b can be equivalently written as:

$$\boxed{n = (...\mathtt{n}_k...\mathtt{n}_3\mathtt{n}_2\mathtt{n}_1\mathtt{n}_0)_b}$$

The most commonly used bases are summarized in the table below:

Base b	**Possible values** $\mathtt{n}_k$	**Representation of** $n = 154$	**Application**
2 *"binary"*	$\{0, 1\}$	$(10011010)_2$	Logic
10 *"decimal"*	$\{0, ..., 9\}$	$(154)_{10}$	Day-to-day life
16 *"hexadecimal"*	$\{0, ..., 9, A, ..., F\}$	$(9A)_{16}$	Memory allocation

❒ **Binary number** – A binary number is the representation of an integer n in base 2. It is expressed as follows:

$$\boxed{n = \sum_{k=0}^{+\infty} \mathtt{n}_k 2^k} \quad \text{where } \mathtt{n}_k \in \{0,1\}$$

Each $\mathtt{n}_k$ is called a binary digit, often abbreviated as bit. We note that:

- If $\mathtt{n}_0 = 0$, then n is an even number.
- If $\mathtt{n}_0 = 1$, then n is an odd number.

❒ **Bit notations** – A binary number represented with N bits has the following bit-related notations:

Notation	**Description**	**Illustration with** $N = 10$
Least significant bit	Right-most bit.	2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 0 0 1 1 0 1 1 1 0 0
Lowest set bit	Right-most bit that has a value of 1.	2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 0 0 1 1 0 1 1 1 0 0
Highest set bit	Left-most bit that has a value of 1.	2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 0 0 1 1 0 1 1 1 0 0
Most significant bit	Left-most bit.	2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 0 0 1 1 0 1 1 1 0 0

Remark: The abbreviation "LSB" is sometimes ambiguous as it may either refer to the least significant bit or the lowest set bit.

❒ **Bitwise operators** – The truth table of the bitwise operators `OR`, `XOR`, `AND` between $x \in \{0,1\}$ and $y \in \{0,1\}$ is given below:

		`OR`	`XOR`	`AND`
x	y	$x \mid y$	$x \wedge y$	$x \,\&\, y$
0	0	0	0	0
1	0	1	1	0
0	1	1	1	0
1	1	1	0	1

❒ **Bit shifting** – Shifting operations change the binary representation of a number as follows:

Operation	Description	Illustration
$x \gg y$ *"right shift"*	Binary representation of x moved by y bits to the right.	1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 0 >> 3 0 0 0 1 0 0 0 1 1 0 0 0 1 1 0 1 1
$x \ll y$ *"left shift"*	Binary representation of x moved by y bits to the left.	0 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 0 << 2 0 0 1 1 0 0 0 1 1 0 1 1 1 1 0 0 0

❒ **Tricks** – The table below shows a few bit-related tricks that are useful in establishing non-trivial results with minimal effort:

Operation	Result	Illustration
$x \,\&\, (x-1)$	0 $\iff x$ is a power of 2	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 AND 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$x \,\&\, 1$	0 $\iff x$ is divisible by 2	1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 1 0 AND 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$1 \ll j$	2^j	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 << 5 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

Remark: The first trick stems from the fact that $2^n - 1 = \sum_{k=0}^{n-1} 2^k$.

❒ **Integer overflow** – The problem of integer overflow occurs when the number of bits needed to encode an integer n exceeds the number of available bits N. More precisely, binary numbers encoded on N bits cannot exceed $2^N - 1$.

	2^{N-1}		2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
...	X	X	1	...	0	0	1	1	0	1	1	1	0	0

This issue can sometimes be alleviated by changing the order in which operations are performed. For example, a safe way to average two integers a and b is shown below:

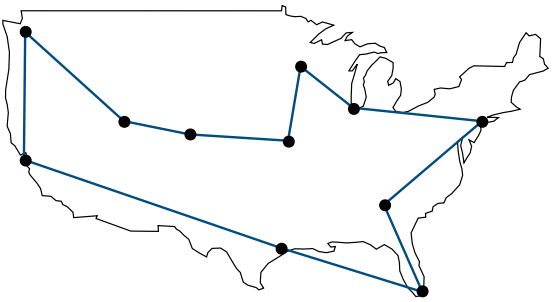
	Naive	**Safe**
Formula	$\frac{a+b}{2}$	$a + \frac{b-a}{2}$
Order of operations	*s_i is the operation done at step i.*	
	$s_1 = a + b$ $s_2 = \frac{s_1}{2}$	$s_1 = b - a$ $s_2 = \frac{s_1}{2}$ $s_3 = a + s_2$
Example	*To illustrate this, we take $a = b = 2^{31}$ and $max = 2^{32} - 1$.*	
	$s_1 = 2^{31} + 2^{31} = 2^{32}$ $s_2 = \frac{2^{32}}{2} = 2^{31}$ Step 1 led to integer overflow because $s_1 > 2^{32} - 1$.	$s_1 = 2^{31} - 2^{31} = 0$ $s_2 = \frac{0}{2} = 0$ $s_3 = 2^{31} + 0 = 2^{31}$ There was no integer overflow.

1.3 Classic problems

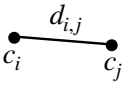
This part covers classic computer science problems and presents detailed approaches to solve them.

1.3.1 Traveling salesman

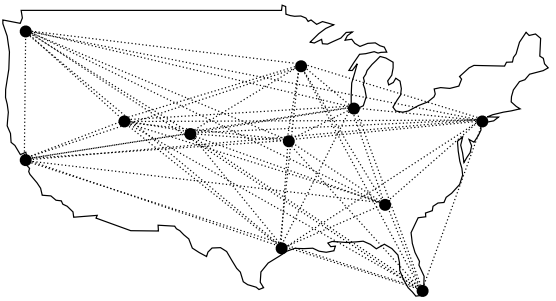
Given n cities $c_1, ..., c_n$, the traveling salesman problem (TSP) is a classic problem that aims at finding the shortest path that visits all cities exactly once and then returns to the starting city.



The distance between each pair of cities (c_i, c_j) is noted $d_{i,j}$ and is known.



A naive approach would consist of enumerating all possible solutions and finding the one that has the minimum cumulative distance. Given a starting city, there are $(n-1)!$ possible paths, so this algorithm would take $\mathcal{O}(n!)$ time. This approach is impracticable even for small values of n.



Luckily, the Held-Karp algorithm provides a bottom-up dynamic programming approach that has a time complexity of $\mathcal{O}(n^2 2^n)$ and a space complexity of $\mathcal{O}(n2^n)$.

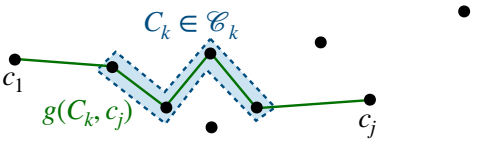
Suppose c_1 is the starting city. This arbitrary choice does not influence the final result since the resulting path is a cycle. We define the following quantities:

- $\mathscr{C}_k$ contains all sets of k distinct cities in $\{c_2, ..., c_n\}$:

$$\text{for } k \in [\![0, n-1]\!], \qquad \mathscr{C}_k = \Big\{ C_k | C_k \subseteq \{c_2, ..., c_n\}, \#C_k = k \Big\}$$

  We note that $\mathscr{C}_k$ has a size of $\binom{n-1}{k}$.

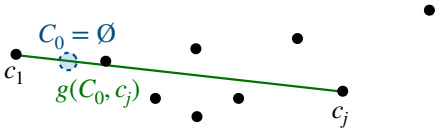
- $g(C_k, c_j)$ is the distance of the shortest path that starts from c_1, goes through each city in $C_k \in \mathscr{C}_k$ exactly once and ends at $c_j \notin C_k$.



The solution is found iteratively:

- *Initialization*: The shortest path between the starting city c_1 and each of the other cities c_j with no intermediary city is directly given by $d_{1,j}$:

$$\forall j \in [\![2, n]\!], \quad \boxed{g(C_0, c_j) = d_{1,j}}$$

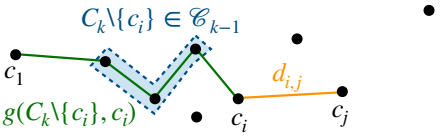


- *Compute step*: Suppose that for all $C_{k-1} \in \mathscr{C}_{k-1}$ and $c_i \notin C_{k-1}$, we know the distance of the shortest path $g(C_{k-1}, c_i)$ between c_1 and c_i via the $k-1$ cities in C_{k-1}.

  Let's take $C_k \in \mathscr{C}_k$. For all $c_i \in C_k$, we note that $C_k \backslash \{c_i\} \in \mathscr{C}_{k-1}$ and that (naturally) $c_i \notin C_k \backslash \{c_i\}$, which means that $g(C_k \backslash \{c_i\}, c_i)$ is known.

  We can deduce the distance of the shortest path between c_1 and $c_j \notin C_k$ via k cities by taking the minimum value over all second-to-last cities $c_i \in C_k$:

$$\forall C_k \in \mathscr{C}_k, \forall c_j \notin C_k, \quad \boxed{g(C_k, c_j) = \min_{c_i \in C_k} \Big\{ g(C_k \backslash \{c_i\}, c_i) + d_{i,j} \Big\}}$$



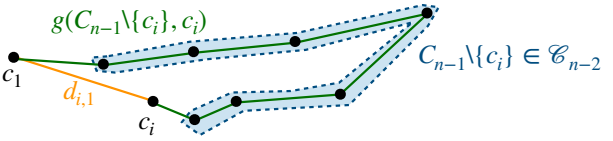
  Each step k has the following complexities:

Time	Space
$k \times (n-1-k) \times \binom{n-1}{k}$	$(n-1-k) \times \binom{n-1}{k}$

- *Final step*: The solution is given by $g(\{c_2, ..., c_n\}, c_1)$.

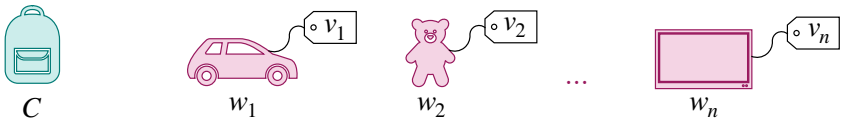
  Since $C_{n-1} = \{c_2, ..., c_n\}$, the last step of the algorithm gives:

$$\boxed{g(C_{n-1}, c_1) = \min_{i \in [\![2,n]\!]} \Big\{ g(C_{n-1} \backslash \{c_i\}, c_i) + d_{i,1} \Big\}}$$



1.3.2 Knapsack

The 0/1 knapsack problem is a classic problem where the goal is to maximize the sum of values of items put in a bag that has a weight limit. Here, "0/1" means that for each item, we want to know whether we should include it (1) or not (0).



More formally, we have a bag of capacity C and n items where each item $i \in [\![1, n]\!]$ has value v_i and weight w_i. We want to find a subset of items $\mathscr{I} \subseteq \{1, ..., n\}$ such that:

$$\boxed{\mathscr{I} = \underset{I \subseteq [\![1,n]\!]}{\operatorname{argmax}} \sum_{i \in I} v_i} \quad \text{with} \quad \sum_{i \in \mathscr{I}} w_i \leqslant C$$

A naive approach would consist of trying out every combination of the n items and take the one with the maximum value which also satisfies the cumulative weight constraint. Such an approach has a time complexity of $\mathcal{O}(2^n)$.

Luckily, this problem can be solved with a bottom-up dynamic programming approach that has a time complexity of $\mathcal{O}(nC)$ and a space complexity of $\mathcal{O}(nC)$.

We note $V_{i,j}$ the maximum value of a bag of capacity $j \in [\![0, C]\!]$ that contains items among $\{1, 2, ..., i\}$. Our objective is to get $V_{n,C}$.

	0	...	j	...	C
0					
...					
i			$V_{i,j}$		
...					
n					$V_{n,C}$

- *Initialization*: The maximum values of the following bags are known:
  - Bags of capacity 0: $V_{i,0} = 0$.
  - Bags with 0 item: $V_{0,j} = 0$.

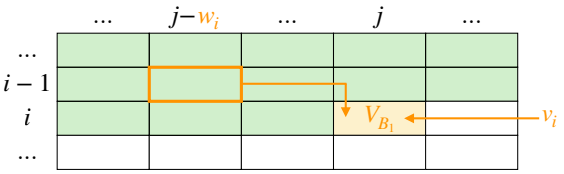
	0	1	...	C
0	0	0	0	0
1	0			
...	0			
n	0			

- *Compute maximum value*: Starting from $i = 1$ and $j = 1$, we iteratively fill the 2D array of maximum values from left to right, top to bottom.

	0	1	...	C
0				
1		⇢	⇢	⇢
...		⇢	⇢	⇢
n		⇢	⇢	⇢

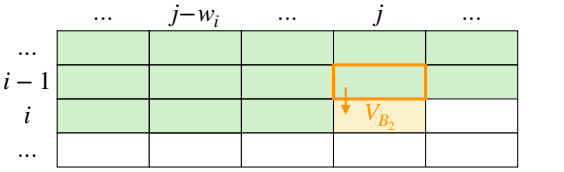
  In order to determine the maximum value $V_{i,j}$ of a bag of capacity j containing items among $\{1, ..., i\}$, we need to choose between two hypothetical bags:
  - *Bag 1: contains item i.* V_{B_1} is found by adding the value v_i of item i to the maximum value of the bag of capacity $j - w_i$ containing items among $\{1, ..., i-1\}$.

$$\boxed{V_{B_1} = V_{i-1,j-w_i} + v_i}$$



– *Bag 2: does not contain item i.* V_{B_2} is already known: it is the maximum value of the bag of capacity j containing items among $\{1, ..., i-1\}$.

$$\boxed{V_{B_2} = V_{i-1,j}}$$



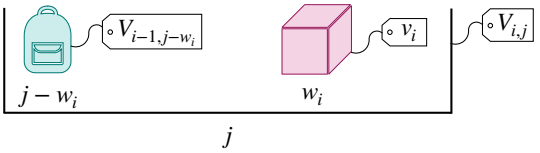
We choose the best potential bag:

$$\boxed{V_{i,j} = \max(V_{B_1}, V_{B_2})}$$

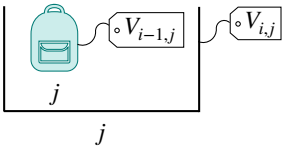
When the 2D array is filled, the desired value is $V_{n,C}$.

- *Get final items*: In order to know which items were selected, we start from position $(i, j) = (n, C)$ of the 2D array and traverse it iteratively:

  – *Case $V_{i,j} \neq V_{i-1,j}$*: This means that item i was included. We go to position $(i-1, j-w_i)$ and include item i in the set of included items.



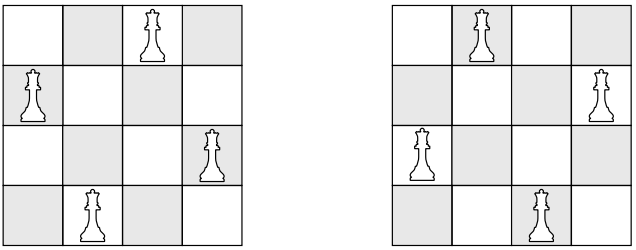
  – *Case $V_{i,j} = V_{i-1,j}$*: This means that item i was not included. We go to position $(i-1, j)$.



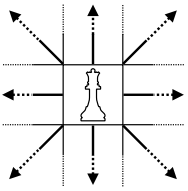
We repeat this process until retrieving all items.

1.3.3 N-Queens

Given a chessboard of size $N \times N$, the N-Queens problem aims at finding a configuration of N queens such that no two queens attack each other.

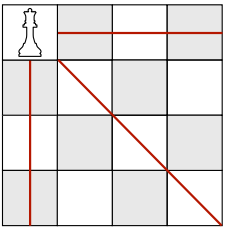


A queen is a chess piece that can move horizontally, vertically or diagonally by any number of steps.

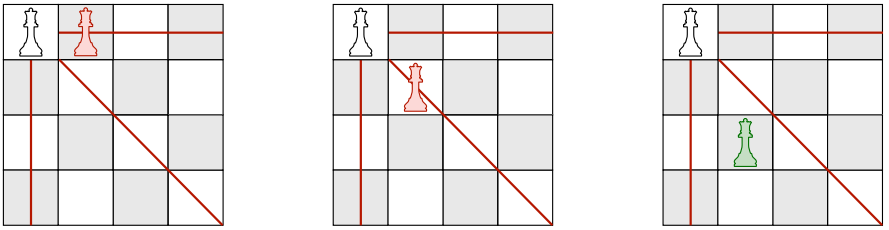


The solution can be found using a backtracking approach:

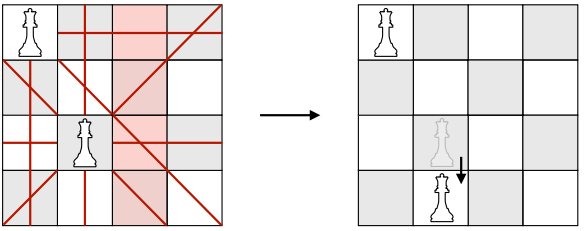
- *Step 1*: Put the 1^{st} queen on the chessboard.



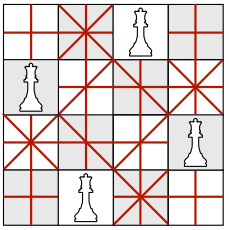
- *Step 2*: Put the 2^{nd} queen in the second column of the chessboard. As long as the partial configuration is invalid, keep changing its position until conditions are satisfied.



- *Step 3*: If there is nowhere to place the next queen, go back one step and try the next partial configuration.

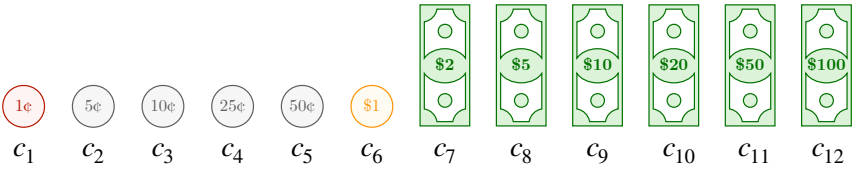


- *Step 4*: Continue this process until finding a valid solution.



1.3.4 Coin change

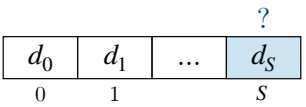
Given an unlimited number of coins of values $\{c_1, .., c_k\}$, the coin change problem aims at finding the minimum number of coins that sum up to a given amount S. By convention, we assume that $c_1 < ... < c_k$.



In other words, we want to find the minimum value of $x = x_1 + ... + x_k$ such that

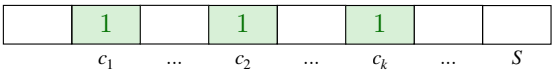
$$S = \sum_{i=1}^{k} x_i c_i$$

The bottom-up dynamic programming solution iteratively computes the minimum amount of coins d_s for each amount $s \in \{1, ..., S\}$.

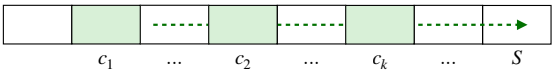


This approach runs in $\mathcal{O}(S)$ time and takes $\mathcal{O}(S)$ space:

- *Initialization*:
  - Initialize array $D = [d_0, ..., d_S]$ with zeros.
  - For each $i \in [\![1, ..., k]\!]$, set d_{c_i} to 1 since we already know that the corresponding amount only needs 1 coin.

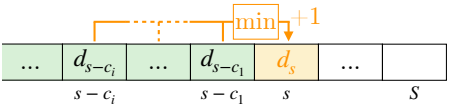


- *Compute step*: Starting from $s = c_1 + 1$, we iteratively fill array D.



  In order to obtain amount s, we distinguish two cases:
  - *Case* $\exists i, d_{s-c_i} > 0$: We look back at valid values d_{s-c_i} and see which coin c_i minimizes the total number of coins needed to obtain amount s.

$$d_s = \min_{\substack{i \in [\![1,k]\!] \\ d_{s-c_i} > 0}} \left\{ d_{s-c_i} + 1 \right\}$$



  - *Case* $\forall i, d_{s-c_i} = 0$: We cannot obtain amount s using coins c_i.

At the end of this step, amounts $s \in \{1, .., S\}$ with $d_s = 0$ cannot be obtained with the given coins.

The answer to this problem is given by d_S.

SECTION 2

Data structures

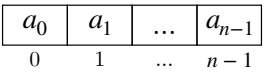
In this second section, we will go through the most common data structures that are used in everyday algorithms, such as arrays, strings, queues, stacks, hash maps, linked lists and more.

2.1 Arrays and strings

In this part, we will learn about arrays and strings along with common tricks such as Kadane's algorithm and the sliding window trick.

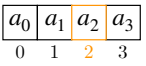
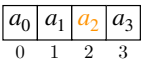
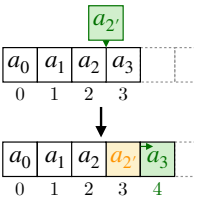
2.1.1 Arrays

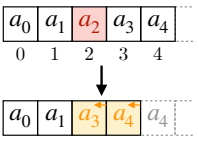
❒ **Definition** – An array A is a data structure of fixed size n that stores elements $a_0, ..., a_{n-1}$ in a sequential way. Each element a_i can be accessed by its index i.



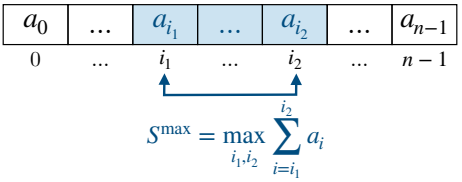
Most programming languages start their indices from 0. The resulting arrays are said to be 0-indexed.

❒ **Operations** – The main operations that can be performed on an array are explained below:

Type	Time	Description	Illustration
Access	$\mathcal{O}(1)$	Using index i, we can directly access the i^{th} element of the array with $A[i] = a_i$.	
Search	$\mathcal{O}(n)$	We need to search the array by checking each element one by one until finding the desired value.	
Insertion	$\mathcal{O}(n)$	1. Elements at indices i and up are moved to the right. 2. The new element is inserted at index i. *Note that if there is no space for the new element to be added in the existing array, we need to create a bigger array and copy existing elements over there.*	

Type	Time	Description	Illustration
Deletion	$\mathcal{O}(n)$	1. Elements at indices $i+1$ and up are moved to the left. 2. The former last element of the array is either ignored or removed.	

❒ Kadane's algorithm – Kadane's algorithm computes the maximum subarray sum of a given array A.

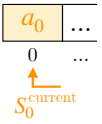


Trick Scan the array and keep track of the following quantities:

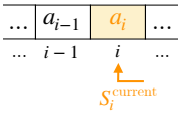
- S_i^{current}: the maximum subarray sum of A up to index i that contains a_i.
- $S_i^{\max}$: the maximum subarray sum of A up to index i.

Algorithm The solution is found in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space:

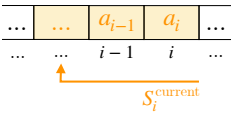
- *Initialization*: Set $S_0^{\text{current}} = S_0^{\max} = a_0$.



- *Update step*: For each index $i \in [\![1, n-1]\!]$, compute S_i^{current} by distinguishing two cases:
  - *Case* $S_{i-1}^{current} < 0$: Appending a_i to the currently tracked subarray sum is worse than starting over. We set $S_i^{\text{current}} = a_i$.



  - *Case* $S_{i-1}^{current} \geqslant 0$: It is worth continuing on the current subarray and appending a_i to it. We set $S_i^{\text{current}} = S_{i-1}^{\text{current}} + a_i$.



Then, the maximum subarray sum seen so far is updated if a new maximum is detected:

$$S_i^{\max} = \max\left(S_{i-1}^{\max}, S_i^{\text{current}}\right)$$

In the end, we output the maximum subarray sum of the array:

$$\boxed{S^{\max} = S_{n-1}^{\max}}$$

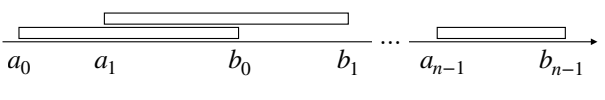
Remark: In order to guarantee a space complexity of $\mathcal{O}(1)$, values of $S^{current}$ and S^{max} are updated in place at each step i.

❒ **Merge intervals** – The merge intervals problem is a classic problem that aims at producing an array of non-overlapping intervals based on an array $I = [I_0, ..., I_{n-1}]$ of intervals that are potentially overlapping.

We note:

- $I = [I_0, ..., I_{n-1}]$ the input array of potentially-overlapping intervals, with:

$$I_k = [a_k, b_k] \quad \text{and} \quad a_k \leqslant b_k \qquad \text{for } k \in [\![0, n-1]\!]$$



a_0	a_1	...	a_{n-1}
b_0	b_1		b_{n-1}
0	1	...	$n-1$

- $I' = [I'_0, ..., I'_{m-1}]$ the output array of non-overlapping intervals, with:

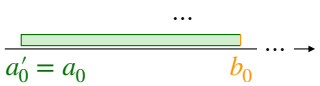
$$I'_k = [a'_k, b'_k] \quad \text{and} \quad a'_k \leqslant b'_k \qquad \text{for } k \in [\![0, m-1]\!] \text{ with } m \leqslant n$$

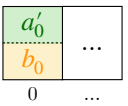
a'_0		a'_{m-1}
b'_0	...	b'_{m-1}
0	...	$m-1$

The output array I' is obtained in $\mathcal{O}(n\log(n))$ time and $\mathcal{O}(n)$ space:

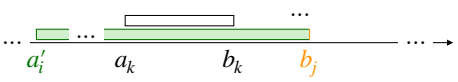
- *Initialization*:

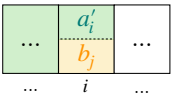
- The input array I is sorted with respect to the lower bound a_k of each interval I_k in $\mathcal{O}(n\log(n))$ time. Without loss of generality, we renumber intervals so that $a_0 \leqslant ... \leqslant a_{n-1}$.
- The output array is initialized with $I'_0 = I_0$.



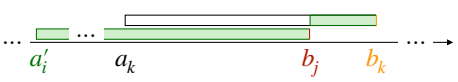


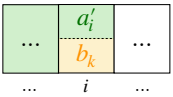
- *Update step*: Starting from $k = 1$, we would like to incorporate $I_k = [a_k, b_k]$ into the output array. By noting $[a'_i, b_j]$ the last added interval of I', we have the following cases:
  - *Case* $b_k < b_j$: I_k does not add any new information since it is entirely overlapping with the last added interval. We discard I_k.



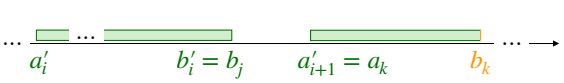


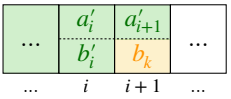
  - *Case* $a_k \leqslant b_j \leqslant b_k$: I_k is partially overlapping with the last added interval, so we update the current upper bound to b_k.





  - *Case* $b_j < a_k$: There is no overlap between I_k and the last added interval, so we add I_k to the final array I' at index $i + 1$.

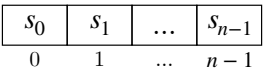




We repeat the *update step* for all intervals I_k in a sequential way to obtain the output array of non-overlapping intervals I'.

2.1.2 Strings

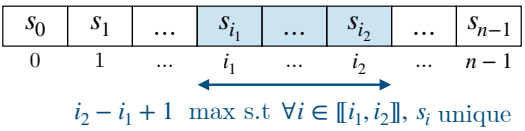
❒ Definition – A string s is a data structure that can be seen as an array of characters $s_0, ..., s_{n-1}$.



The following table summarizes a few categories of strings that have special properties:

Type	Description	Example
Palindrome	Reads the same backward and forward.	`kayak`, `madam`, `radar`
Anagram	Forms another string using a rearrangement of its letters.	`listen` $\longrightarrow$ `silent`
Strobogrammatic	Remains the same when rotated by 180 degrees.	`69`, `818`, `1001`

❒ **Longest substring** – Given a string $s_0 ... s_{n-1}$, the goal of this problem is to find the length of the longest substring that does not contain repeated characters.



A naive approach would check the constraint on every possible substring using a nested `for`-loop in $\mathcal{O}(n^2)$ time. However, there is a more efficient approach that uses the sliding window trick.

Trick Use a left pointer l and a right pointer r to delimit a window which has the constraint to not have repeated characters. We have the following rules:

- *Constraint is not violated*: The current solution may have not reached its full potential yet. The r pointer advances until the constraint is violated.

- *Constraint is violated*: The current solution is not valid anymore, so we need to trim the solution down. The l pointer advances until the constraint is no longer violated.

Algorithm The solution is found in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space:

- *Initialization*:
  - Initialize an empty hash set that aims at storing the characters that are part of the current substring.
  - Set a global counter c to 0, which keeps track of the maximum substring length encountered so far.
  - Set the left pointer l and right pointer r to 0.

- *Compute step*: s_r is the new character proposed to be added to the current substring.

  - If s_r is already in the hash set, then adding it again would make us have s_r twice in the substring.

    Increment l and remove s_l from the hash set until s_r is not there anymore. The substring gets trimmed down.

  - Now, the updated substring is ready to have s_r without violating the constraint. Add s_r to the hash set.

  - Check whether the length $r - l + 1$ of the resulting substring is greater than the counter c. If it is, store the new maximum in c.

– Increment r by 1.

Repeat this process until r reaches the end of the string.

- *Final step*: c is the length of the biggest substring without repeated characters.

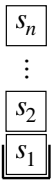
Remark: The sliding window trick is found in many string problems.

2.2 Stacks and queues

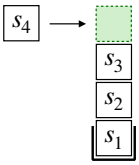
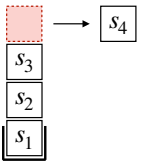
In this part, we will look at stacks and queues, along with their respective operations.

2.2.1 Stacks

❒ **Definition** – A stack s is a data structure that deals with elements $s_1, ..., s_n$ in a *Last In First Out* (LIFO) order.



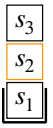
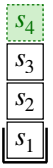
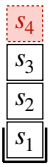
In order to do that, it uses the push and pop operations:

Push	**Pop**
Insert an element on the top of the stack.	Remove the element from the top of the stack and return its value.
	

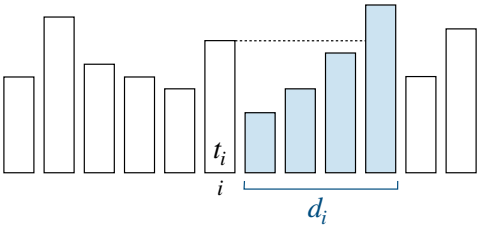
As a result, the last element added to the data structure will be the first one removed.

Remark: Stacks are commonly used for Depth-First Search (DFS) graph traversals.

❒ **Operations** – The main operations that can be performed on a stack are explained below:

Type	Time	Description	Illustration
Access	$\mathcal{O}(1)$	*Top of the stack* Read the top element.	
	$\mathcal{O}(n)$	*Any other position* Depending on the desired position, the stack needs to be popped at most n times.	
Search	$\mathcal{O}(n)$	Depending on whether the desired value gets found right away, the stack needs to be popped at most n times.	
Insertion	$\mathcal{O}(1)$	*Top of the stack* Push the new element.	
	$\mathcal{O}(n)$	*Any other position* 1. Pop elements until accessing the desired position and push them into an auxiliary stack. 2. Push the new element. 3. Pop elements from the auxiliary stack and push them into the initial stack.	
Deletion	$\mathcal{O}(1)$	*Top of the stack* Pop the existing element.	
	$\mathcal{O}(n)$	*Any other position* 1. Pop elements until accessing the desired position and push them into an auxiliary stack. 2. Pop the existing element. 3. Pop elements from the auxiliary stack and push them into the initial stack.	

❒ **Daily temperatures** – Suppose we are given an array $T = [t_0, ..., t_{n-1}]$ of temperatures. For each day i, the goal is to find the number of days d_i until there is a warmer temperature.



In other words, for each day $i \in [\![0, n-1]\!]$, we want to find the number of days d_i

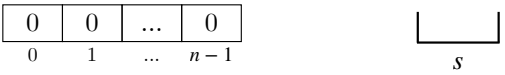
such that:

$$d_i = \min_{j>i, t_j>t_i} (j - i)$$

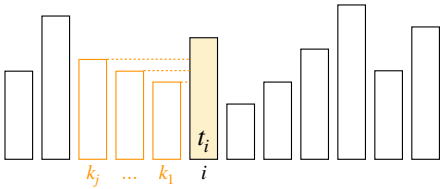
A naive approach would compute the answer for each day in isolation. By scanning up to $\mathcal{O}(n - i)$ elements for each day i, this approach leads to an overall time complexity of $\mathcal{O}(n^2)$.

However, a more efficient approach uses stacks and computes the solution in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space:

- *Initialization*:
  - Initialize an empty stack s.
  - Initialize the output array $D = [d_0, ..., d_{n-1}]$ with zeros.



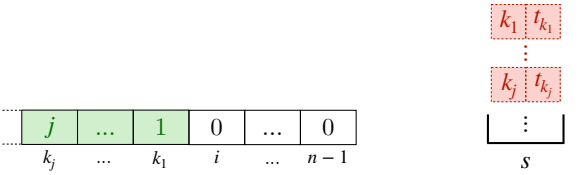
- *Compute step*: We use the monotonic stack trick:
  - The stack s tracks days k and their associated temperatures t_k.
  - Elements added to the stack are days for which we want to find a greater temperature. They are popped as soon as a future warmer day is found.
  - Temperature values are popped in a monotonically non-decreasing manner. This ensures that all relevant values are popped for a given warmer day.



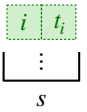
  We start from day 0. For each day i, we have the following cases:
  - If the temperature t_k at the top of the stack is lower than the current temperature t_i, pop (k, t_k) from the stack, and write the corresponding day difference in the final array:

$$\boxed{d_k = i - k}$$

Repeat the popping procedure until the condition is not satisfied anymore.



– Add the current item (i, t_i) to the stack s.

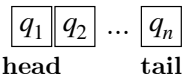


Repeat this process until reaching the end of the array T.

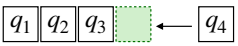
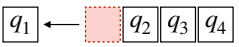
The array D contains the final answer. Entries with zeros correspond to days that do not have future warmer days.

2.2.2 Queues

❒ **Definition** – A queue q is a data structure that deals with elements $q_1, ..., q_n$ in a *First In First Out* (FIFO) order, where its tail has the most recently arrived elements and its head has elements that arrived the earliest.



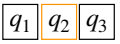
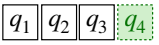
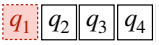
In order to do that, it uses the enqueue and dequeue operations:

Enqueue	**Dequeue**
Insert element at the tail of the queue.	Remove element from the head of the queue.
	

As a result, the first element added to the data structure will be the first one to be removed.

Remark: Queues are commonly used for Breadth-First Search (BFS) graph traversals.

❒ **Operations** – The main operations that can be performed on a queue are explained below:

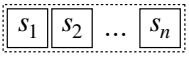
Type	Time	Description	Illustration
Access	$\mathcal{O}(1)$	*Head of the queue* Read the front element.	
	$\mathcal{O}(n)$	*Any other position* Depending on the desired position, the queue needs be dequeued at most n times.	
Search	$\mathcal{O}(n)$	Depending on whether the desired value gets found right away, the queue needs to be dequeued at most n times.	
Insertion	$\mathcal{O}(1)$	*Tail of the queue* Enqueue the new element.	
	$\mathcal{O}(n)$	*Any other position* 1. Initialize a new empty queue. 2. Dequeue elements from the current queue and enqueue them to the new queue up until right before the desired position of insertion. 3. Enqueue the new element in the new queue. 4. Dequeue the remaining elements from the current queue and enqueue them to the new queue.	
Deletion	$\mathcal{O}(1)$	*Head of the queue* Dequeue the existing element.	
	$\mathcal{O}(n)$	*Any other position* 1. Initialize a new empty queue. 2. Dequeue elements from the current queue and enqueue them to the new queue up until right before the desired position of deletion. 3. Dequeue the element to be removed from the current queue and ignore it. 4. Dequeue the remaining elements from the current queue and enqueue them to the new queue.	

2.3 Hash tables

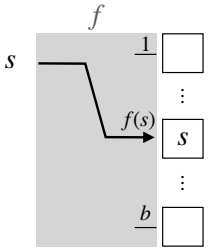
In this part, we will go through the basics of hash tables along with methods to resolve collisions.

2.3.1 General concepts

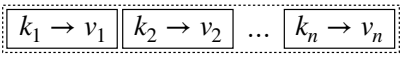
❒ **Hash set** – A hash set is a data structure that stores a collection of unique values $\{s_1, ..., s_n\}$ with fast search times.



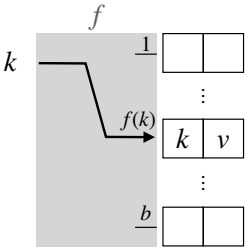
A hash function f is used to link an element s to the address $f(s)$ of the bucket that contains it. $f(s)$ is called hash value of s.



❒ **Hash table** – A hash table, also called hash map, is a data structure used to index large amounts of data with fast key search times. It consists of an unordered collection of key-value pairs $\{(k_1, v_1), ..., (k_n, v_n)\}$.



A hash function f is used to link a key k to the address $f(k)$ of the bucket containing the associated value v. $f(k)$ is called hash value of k.



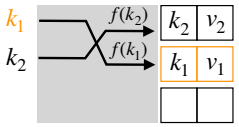
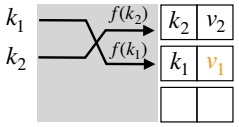
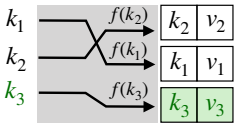
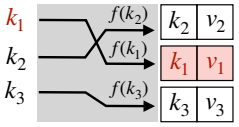
An ideal hash function fulfills the following properties:

- *Be easy to compute*, to minimize runtimes
- *Have a resolution method* if two distinct keys have the same hash value, i.e. form a hash collision

- *Have a uniform distribution of hash values* to ensure a minimal amount of collisions

Remark: Basic hash functions include summing the ASCII representation of characters, whereas more sophisticated ones use advanced algebraic properties.

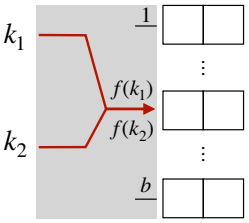
❒ **Operations** – The following operations can be performed on a hash table with a reasonably good hash function:

Type	Time	Description	Illustration
Access	$\mathcal{O}(1)$	Using the key k, compute the associated bucket $f(k)$ and access the corresponding value v.	
Search	$\mathcal{O}(n)$	For each key k: 1. Compute the associated hash value $f(k)$. 2. Check if the value v is in the bucket associated with $f(k)$.	
Insertion	$\mathcal{O}(1)$	Given the key k, compute the corresponding bucket $f(k)$ and add the value v there.	
Deletion	$\mathcal{O}(1)$	Given the key k, compute the corresponding bucket $f(k)$ and remove the value v from there.	

Remark: It is crucial to have a good hash function to ensure the $\mathcal{O}(1)$ runtimes.

2.3.2 Collisions

❒ **Definition** – A hash collision happens when two different keys have the same hash value. When this is the case, we need a way to resolve it.



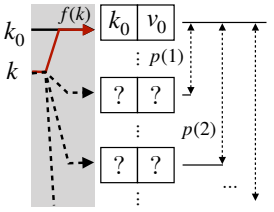
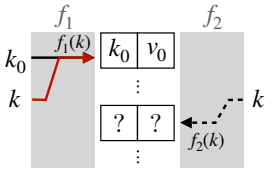
❒ **Load factor** – The load factor ρ of a hash table is defined based on the total number of items stored $n = \#\{(k_i, v_i)\}$ and the number of buckets b as follows:

$$\boxed{\rho \triangleq \frac{n}{b}}$$

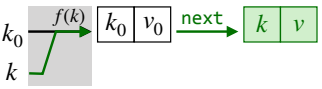
Remark: If there are more items than there are buckets, then we have $\rho > 1$. In this situation, the pigeonhole principle guarantees the presence of hash collisions.

❒ **Collision resolution** – The most common resolution methods can be put into two categories:

- *Open addressing*: We try to find an open spot by looking at other buckets. This works best when ρ is close to 0.

Method	Description	Illustration
Linear probing $p(x) = x$	Tries to put the value in the next bucket until there is no more collision.	
Quadratic probing $p(x) = x^2$	Tries to put the value in buckets that are further and further away in a quadratic fashion, until there is no more collision.	
Double hashing f_1, f_2	Uses a secondary hash function if the first one leads to a collision.	

- *Closed addressing*: We append the value to the existing bucket. This works best when when ρ is close to 1.

Method	Description	Illustration
Chaining	Uses linked lists to sequentially add values assigned to the same bucket.	

The risk of collisions highlight the importance of choosing a hash function that leads to a uniform distribution of hash values.

2.4 Advanced hash tables

In this part, we will focus on space-efficient data structures such as bloom filters and count-min sketches, which use tricks based on hash functions.

2.4.1 Bloom filters

❒ **Definition** – A bloom filter $\mathcal{B}$ is a space-efficient data structure that can check whether a given element x is:

- either potentially in the set

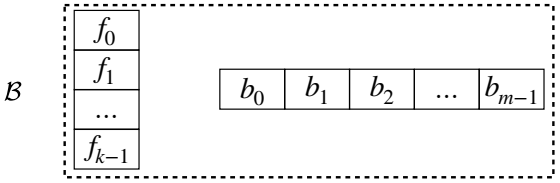
$$x \rightarrow \boxed{\mathcal{B}} \rightarrow \checkmark \implies x \in_? \mathcal{B}$$

- or definitely not in the set

$$x \rightarrow \boxed{\mathcal{B}} \rightarrow \times \implies x \notin \mathcal{B}$$

It is a data structure composed of:

- k independent hash functions $f_0, f_1, ..., f_{k-1}$ with uniform distributions over integer values between 0 and $m-1$
- an array B of size m of elements $b_0, b_1, ..., b_{m-1}$ initially all equal to 0



Element b_j of the array is set to 1 when an element x is inserted and verifies $f_i(x) = j$ for a given $i \in [\![0, k-1]\!]$, with $j \in [\![0, m-1]\!]$.

The false positive rate ϵ quantifies how unreliable a positive prediction given by the bloom filter is:

$$\epsilon = \frac{\#\text{ elements } \textbf{falsely} \text{ predicted to be in the set}}{\#\text{ elements predicted to be in the set}}$$

It is approximated using the number of bits m, the number of hash functions k and the number of elements n that we want to insert as follows:

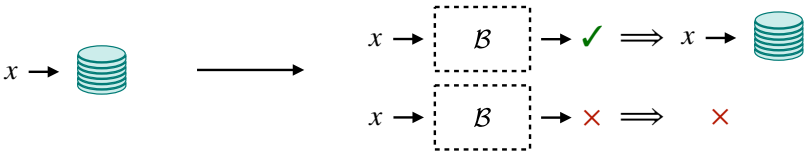
$$\boxed{\epsilon \approx \left(1 - e^{-\frac{kn}{m}}\right)^k}$$

We want ϵ to be as small as possible. Given m and n, the optimal value of the number of hash functions k is such that:

$$\boxed{k = \frac{m}{n} \log_e(2)}$$

Application Suppose we would like to check whether an element belongs to a set of data stored on a disk that is expensive to call.

- A naive approach would consist of calling the disk every time we want to check whether an element is there. This process would be very slow.
- Now, let's assume we have a bloom filter that checks whether the element is in the set (fast operation) and only lets us call the disk if it is predicted to be there. We would be cutting the number of unnecessary call requests, thus making the process faster.

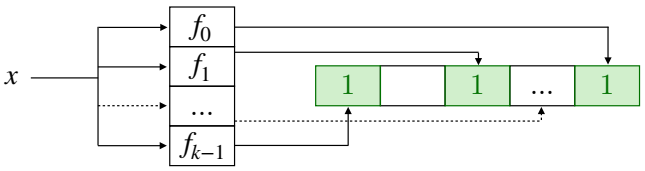


❒ **Operations** – An initialized bloom filter has all the bits of its array B set to 0.

Insertion In order to insert an element x to the bloom filter, we follow the steps below:

- *Compute step*: Compute hash values $f_0(x), ..., f_{k-1}(x)$.
- *Update step*: Update the m-bit array B by turning on the corresponding bits:

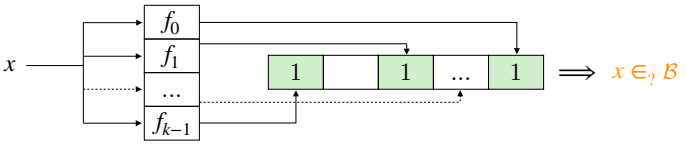
$$\text{for } i \in [\![0, k-1]\!], \quad \boxed{b_{f_i(x)} \longleftarrow 1}$$



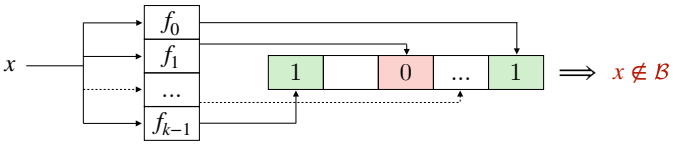
Deletion Deleting an element from the bloom filter is not possible, since removing associated bits may inadvertently remove other elements.

Search In order to check whether an element x is potentially in the bloom filter, we follow the steps below:

- *Compute step*: Compute hash values $f_0(x), ..., f_{k-1}(x)$.
- *Check step*: We have two possible outcomes:
  - *All corresponding bits are on*: This means that the element x is *possibly* in the set. It can also not be, in which case the prediction is a false positive.

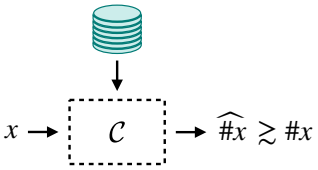


  - *One of the bits is off*: This means that the element is definitely not in the set, and we are sure of it.



2.4.2 Count-min sketches

❒ **Definition** – A count-min sketch $\mathcal{C}$ is a space-efficient data structure that provides an upper bound $\widehat{\#x}$ for the number of times $\#x$ a given element x appeared.



It is a data structure composed of:

- k independent hash functions $f_0, ..., f_{k-1}$ with uniform distributions over values between 0 and $m-1$
- a 2D array C with $k \times m$ elements $c_{i,j}$ initially all equal to 0, with:

- k rows, corresponding to the output of each of the k hash functions
- m columns, corresponding to the m different values that each hash function can take

$\mathcal{C}$

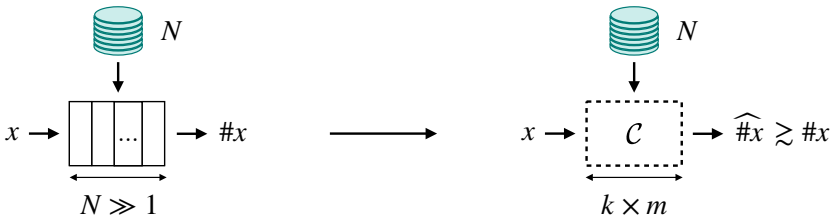
f_0	$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	...	$c_{0,m-1}$
f_1	$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	...	$c_{1,m-1}$
...	...				
f_{k-1}	$c_{k-1,0}$	$c_{k-1,1}$	$c_{k-1,2}$	...	$c_{k-1,m-1}$

Each element $c_{i,j}$ is an integer that corresponds to the number of times an element x was inserted and verified $f_i(x) = j$ for a given $i \in [\![0, k-1]\!]$, with $j \in [\![0, m-1]\!]$.

k and m are hyperparameters that are chosen when initializing the data structure. The bigger they are, the more accurate the approximation will be, but also the more space the data structure will take and the more time each operation will take.

Application Suppose we would like to check how many times an element appeared in a large stream of data.

- A naive approach would consist of keeping track of the stream of data and counting the number of times each element appeared in a large entry table. The space taken by this approach is as big as the number of distinct elements seen. This could be very big.
- Now let's assume we add a count-min sketch that approximately counts each element. We would keep a fixed-size data structure that could handle large streams of data and provide reasonable approximations.

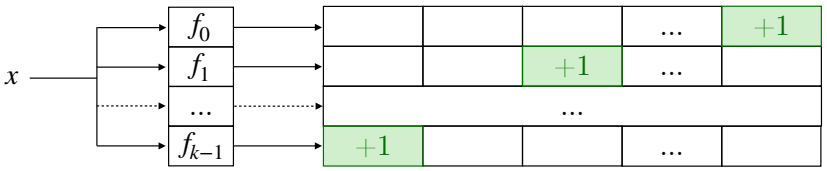


❒ **Operations** – An initialized count-min sketch has its 2D array C filled with zeros.

Insertion In order to insert an element x to the count-min sketch, we follow the steps below:

- *Compute step*: Compute hash values $f_0(x), ..., f_{k-1}(x)$.
- *Update step*: Update the 2D array C by incrementing the corresponding counts:

$$\text{for } i \in [\![0, k-1]\!], \quad \boxed{c_{i,f_i(x)} \longleftarrow c_{i,f_i(x)} + 1}$$



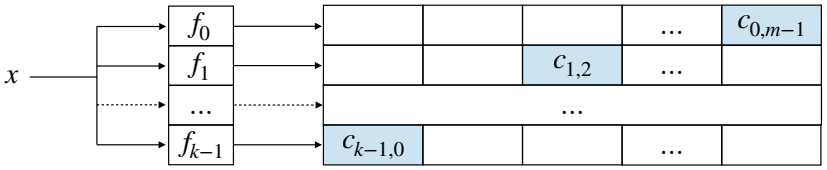
Deletion Deleting an element from the count-min sketch is not possible, since removing associated counts may inadvertently remove other elements.

Search In order to compute an estimate of the number of times a given element x was previously inserted into the count-min sketch, we follow the steps below:

- *Compute step*: Compute hash values $f_0(x), ..., f_{k-1}(x)$.
- *Check step*: Take the minimum across all the corresponding counts:

$$\boxed{\widehat{\#x} = \min\left(c_{0,f_0(x)}, ..., c_{k-1,f_{k-1}(x)}\right)}$$

This estimate is an upper bound because it might have been inflated by hash collisions.



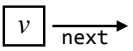
2.5 Linked lists

In this part, we will learn about singly and doubly linked lists, along with tricks to identify cycles and related applications.

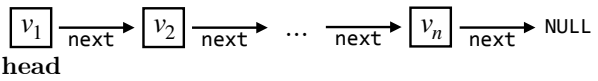
2.5.1 Singly linked lists

❒ **Definition** – A singly linked list is a data structure composed of nodes, where each node carries the information of:

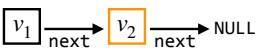
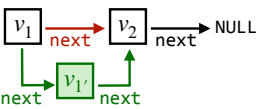
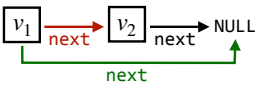
- a value v
- a `next` field, that points to the next node



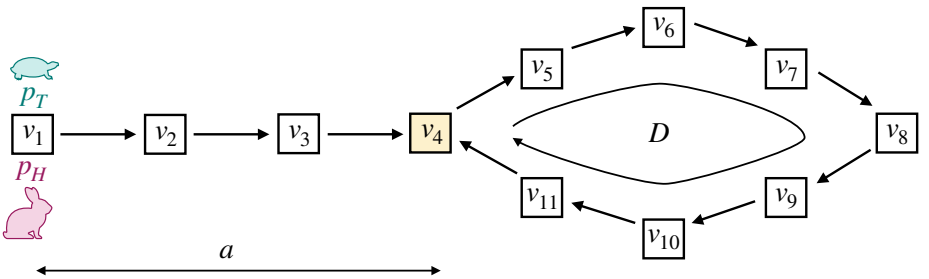
The first node of a singly linked list is called the head. The overall data structure is represented as follows:



❒ **Operations** – The main operations that can be performed on a singly linked list are explained below:

Type	Time	Description	Illustration
Access	$\mathcal{O}(n)$	Starting from the head, the linked list is traversed in a sequential way to access the desired node. If it is the last element, the whole linked list is traversed.	
Search	$\mathcal{O}(n)$	Starting from the head, the linked list is traversed in a sequential way to search for the desired node. If the node is not found, the whole linked list is traversed.	
Insertion	$\mathcal{O}(1)$	1. Assign the `next` pointer of the previous-to-be element to the newly-inserted node. 2. Assign the `next` pointer of the new element to the following element.	
Deletion	$\mathcal{O}(1)$	Assign the `next` pointer of the previous node to the following node of the soon-to-be-removed node.	

❒ **Floyd's algorithm** – Floyd's algorithm, also known as the tortoise and hare algorithm, is able to detect the starting point of a cycle in a linked list.



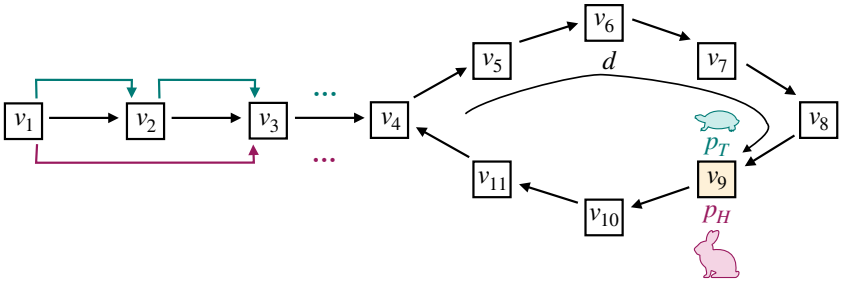
It finds the start of the cycle using the two-pointer method in 3 steps, with a time complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(1)$:

- *Meeting step*: We define two pointers:
  - A slow pointer (tortoise) p_T that goes 1 step at a time.
  - A fast pointer (hare) p_H that goes 2 steps at a time.

  The tortoise and the hare both start from the head of the linked list and respectively travel distances d_T and d_H until they meet. At the meeting point, these quantities verify $d_H = 2d_T$.

  We note Δ_1 the difference between the distances traveled by the hare and the tortoise:

$$\Delta_1 \triangleq d_H - d_T = d_T$$



Given that the two animals necessarily meet somewhere in the cycle, we can also write:

$$\begin{cases} d_T = a + d + n_1 D \\ d_H = a + d + n_2 D \end{cases} \quad \text{where } n_2 > n_1$$

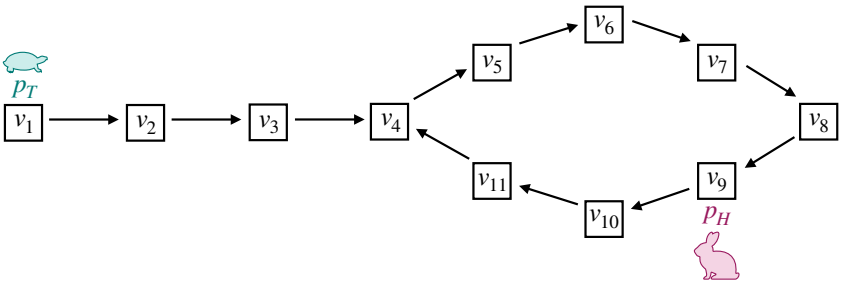
This means that we can rewrite Δ_1 as:

$$\Delta_1 \triangleq d_H - d_T = (n_2 - n_1)D$$

Hence, we have:

$$\boxed{d_T = (n_2 - n_1)D}$$

- *Restart step*: Now, we keep the hare at the meeting point and place the tortoise back to the head of the linked list.



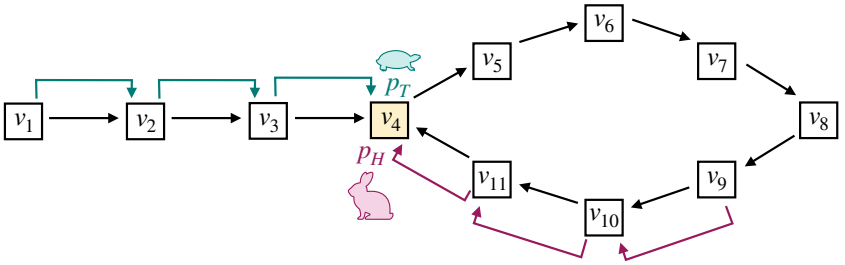
Since the tortoise has moved back by a distance d_T, the new distance Δ_2 between them is such that:

$$\Delta_2 \triangleq \Delta_1 + d_T = 2d_T = 2(n_2 - n_1)D$$

Hence, we have:

$$\boxed{\Delta_2 = kD} \quad \text{with } k \in \mathbb{N}^*$$

- *Detection step*: We make each animal move 1 step at a time, thus keeping the distance Δ_2 between them constant.

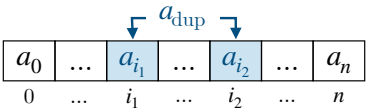


Since Δ_2 is a multiple of the length D of the cycle, the new meeting point between the two animals will precisely be the start of the cycle.

Remark: This algorithm can be used to detect cycles, length of portions of linked lists, among other use cases.

❒ **Duplicate number** – Given an array $A = [a_0, ..., a_n]$ of size $n + 1$ where each element a_i is in the range $[\![1, n]\!]$, suppose there exists exactly two elements a_{i_1}, a_{i_2}

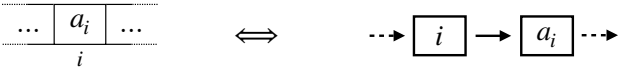
such that $a_{i_1} = a_{i_2} \triangleq a_{\text{dup}}$. All other elements are assumed to be distinct. The goal of the problem is to find the duplicate value a_{dup}.



A naive approach would sort the array and scan the result to find the two consecutive elements that are equal. This would take $\mathcal{O}(n \log(n))$ time.

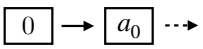
However, a more efficient approach takes $\mathcal{O}(n)$ time and leverages Floyd's algorithm.

Trick Suppose we represent the array by $n + 1$ distinct nodes that compose one or more linked lists. Each element a_i of the array A is seen as a node i that points to node a_i.

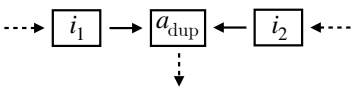


We make the following observations:

- For $i \in [\![0, n]\!]$, a_i covers all values between 1 and n. As a result:
  - The $n + 1$ nodes point to exactly n of its nodes, which guarantees that each resulting linked list has a cycle.
  - No node points to node 0, which means that the cycle in the linked list starting from node 0 is not circular. Linked lists that do not contain node 0 (if any) are necessarily circular.



- Both the nodes i_1 and i_2 point to node a_{dup}. As a result:
  - The desired duplicate number is at the start of a cycle.

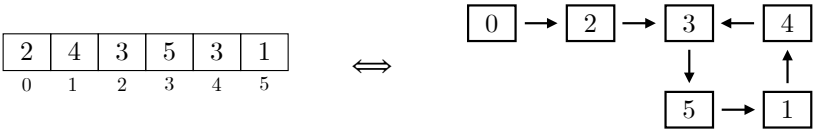


  - The cycle containing the duplicate number is necessarily non-circular.

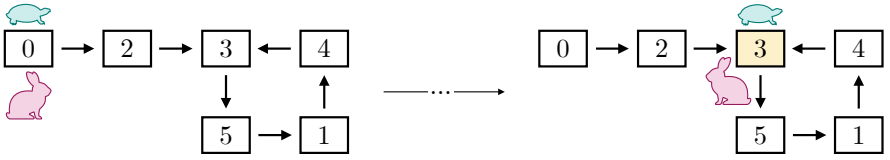
From the two points above, we deduce that the linked list starting from node 0 contains a_{dup} at the start of its cycle.

Algorithm The duplicate number is found in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space:

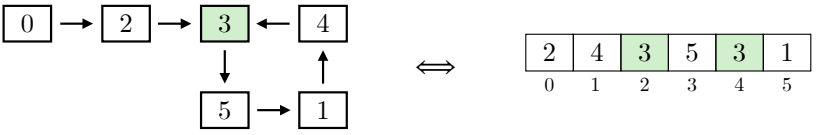
- *Initialization*: Think of array A in terms of linked list(s) using the trick above. Note that no extra-space is needed. For a given element a_i, the next element can be accessed via a_{a_i}.



- *Compute step*: Apply Floyd's algorithm from node 0, which takes $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space.



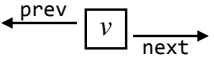
- *Final step*: The start of the cycle is the duplicate number a_{dup}.



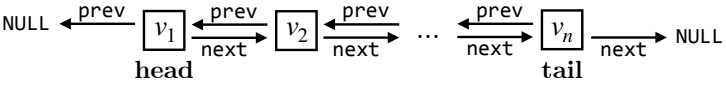
2.5.2 Doubly linked lists

❒ **Definition** – A doubly linked list is a data structure composed of nodes, where each node carries the information of:

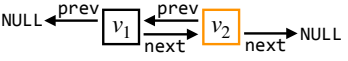
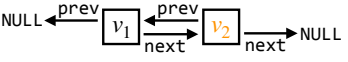
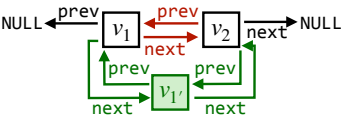
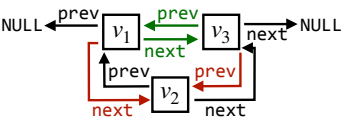
- a value v
- a `next` pointer that points to the next node
- a `prev` pointer that points to the previous node



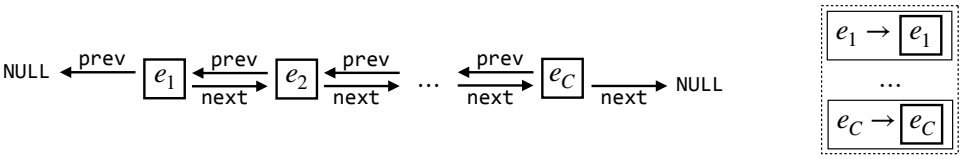
The first and last nodes of a doubly linked list are called the head and tail respectively. The overall data structure is represented as follows:



❒ **Operations** – The main operations that can be performed on a doubly linked list are explained below:

Type	Time	Description	Illustration
Access	$\mathcal{O}(n)$	Starting from the head, the linked list is traversed in a sequential way to access the desired node. If it is the last element, the whole linked list is traversed.	
Search	$\mathcal{O}(n)$	Starting from the head, the linked list is traversed in a sequential way to search for the desired node. If the node is not found, the whole linked list is traversed.	
Insertion	$\mathcal{O}(1)$	1. Assign the `next` pointer of: a. The previous-to-be element to the newly-inserted node. b. The new element to the following element. 2. Assign the `prev` pointer of: a. The new element to the previous-to-be element. b. The following element to the new element.	
Deletion	$\mathcal{O}(1)$	1. Assign the `next` pointer of the previous node to the following node. 2. Assign the `prev` pointer of the following node to the previous node.	

❒ **LRU cache** – A *Least Recently Used* (LRU) cache of capacity C is a bounded-size structure that keeps track of the C most recently used items. An item is said to be "used" if it is either accessed or inserted.

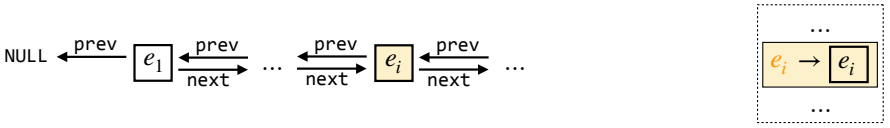


It is a classic example of an efficient combination of doubly linked lists with hash tables, where:

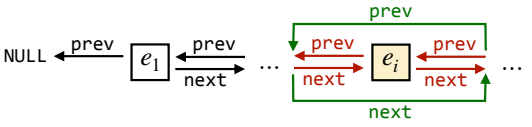
- The doubly linked list provides an ordering of elements with respect to their last use, with the most recently used ones being closer to the head.
- The hash table maps values to nodes and enables access times to existing nodes in $\mathcal{O}(1)$.

Access The retrieval operation is done in $\mathcal{O}(1)$ time as follows:

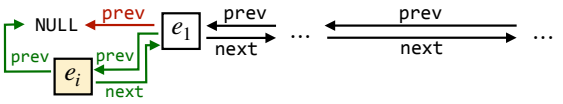
- *Identification step*: Identify the node using the hash table.



- *Update step*: Convey the fact that the node is now the most recently used element.
  - Connect the neighbors of the element between them.

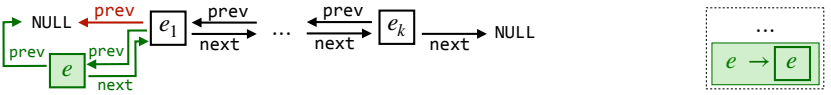


  - Bring the element to the front of the doubly linked list.

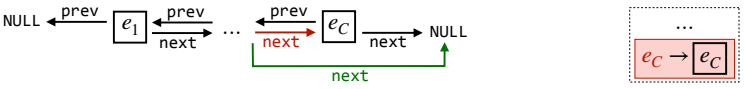


Insertion This operation is achieved in $\mathcal{O}(1)$ time and depends on the situation:

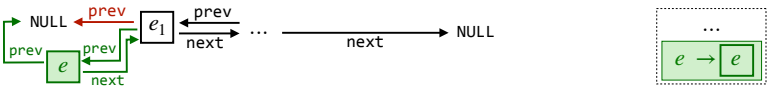
- *Inserted element already in cache.* When a cached element is inserted, that element is taken to the front of the linked list in the same fashion as when the element is accessed.
- *Inserted element not in cache.* When a new element e is inserted, we have the following cases:
  - *If the cache is below capacity*: Insert the new element at the front of the doubly linked list and add it to the hash table.



  - *If the cache is at capacity*: Remove the least recently used element to welcome the new node.
    - *Removal step*: Remove from the hash table the entry corresponding to the element that was the least used and update the tail of the doubly linked list.



    - *Insertion step*: Insert the new element at the front of the doubly linked list and add it to the hash table.



SECTION 3

Graphs and trees

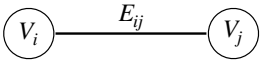
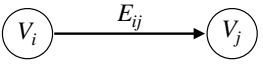
In this third section, we will go through the basics of graphs along with useful graph traversal algorithms. Then, we will focus on both standard trees along with trees that have special properties, such as binary search trees and tries.

3.1 Graphs

In this part, we will study the main notions of graphs and graph traversal algorithms such as BFS, DFS and topological sorting. We will learn about the main shortest paths algorithms such as Dijkstra's, A*, Bellman-Ford, and Floyd-Warshall algorithms.

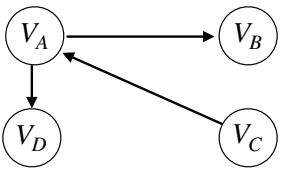
3.1.1 General concepts

❒ **Definition** – A graph G is defined by its vertices V and edges E and is often noted $G = (V, E)$. The following table summarizes the two main types of graph:

Undirected graph	**Directed graph**
Edges do not have a direction	Edges have a direction
	

Remark: The terms "node" and "vertex" can be used interchangeably.

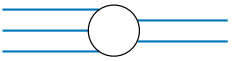
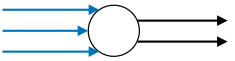
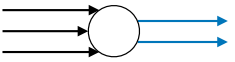
❒ **Graph representation** – Let's consider the following graph:



It can be represented in two different ways:

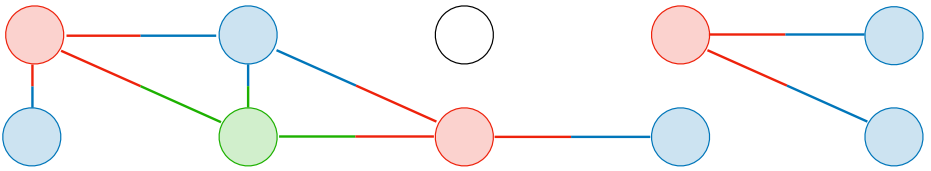
Type	Description	Illustration
Adjacency list	Collection of unordered lists where each entry V_i maps to all the nodes V_j such that E_{ij} exists in the graph.	$V_A \to \{V_B, V_D\}$ $V_B \to \emptyset$ $V_C \to \{V_A\}$ $V_D \to \emptyset$
Adjacency matrix	Matrix of boolean values where the entry (i, j) indicates whether E_{ij} is present in the graph. *In an undirected graph, this matrix is always symmetric.*	$\begin{array}{c\|cccc} & V_A & V_B & V_C & V_D \\ \hline V_A & 0 & 1 & 0 & 1 \\ V_B & 0 & 0 & 0 & 0 \\ V_C & 1 & 0 & 0 & 0 \\ V_D & 0 & 0 & 0 & 0 \end{array}$

❒ **Degree** – A node can have the following characteristics based on its adjacent edges:

Graph	Notation	Definition	Illustration
Undirected	Degree	Number of connected edges	
Directed	In-Degree	Number of connected inbound edges	
	Out-Degree	Number of connected outbound edges	

Remark: Nodes of even (resp. odd) degree are called even (resp. odd) nodes.

❒ **Handshaking lemma** – In an undirected graph, the sum of node degrees is an even number.

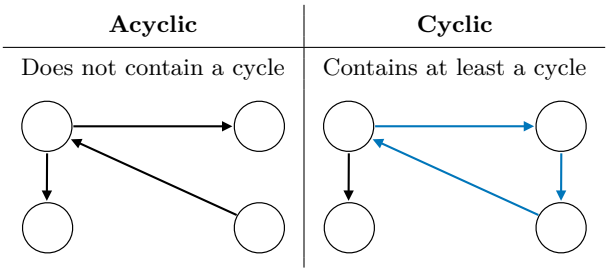


Indeed, every edge links two nodes, therefore responsible for increasing the sum of degrees by 2 (one per node). We have the following formula:

$$\boxed{\sum_{v \in V} \deg(v) = 2|E|}$$

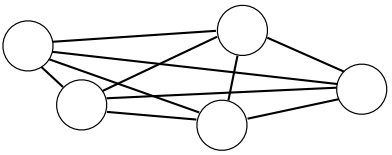
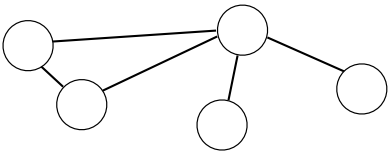
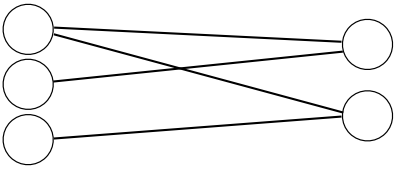
Remark: A consequence of the above formula is that there is an even number of odd nodes.

❒ **Cyclicity** – The cyclicity of a graph is a property that depends on whether the graph has a cycle:



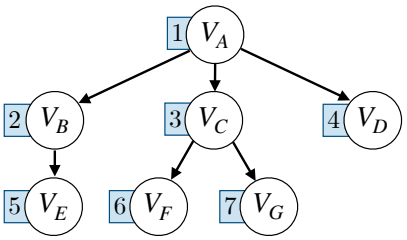
Remark: A directed acyclic graph is commonly abbreviated as DAG.

❒ **Properties** – A graph can have any of the following properties:

Type	Description	Illustration
Complete	Every pair of vertices is connected by an edge.	
Connected	There exists a path between every pair of nodes.	
Bipartite	Vertices can be split into two disjoint sets such that every edge of the graph links a vertex from one set to one in the other.	

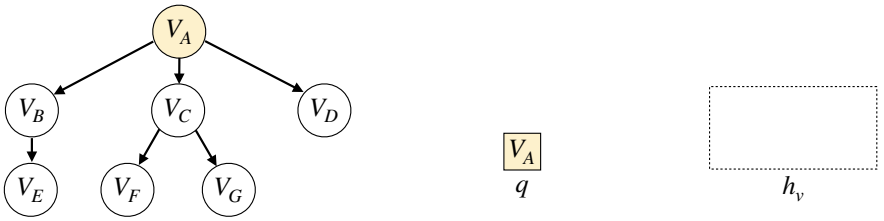
3.1.2 Graph traversal

❒ **Breadth-First Search** – Breadth-First Search (BFS) is an algorithm that explores a graph level by level.

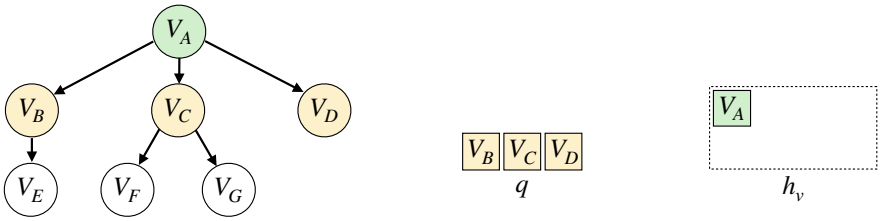


The graph traversal is performed in $\mathcal{O}(|V| + |E|)$ time and $\mathcal{O}(|V|)$ space:

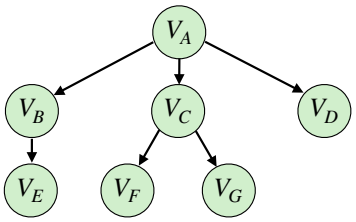
- *Initialization*: The following quantities are tracked:
  - Queue q that keeps track of the next nodes to potentially visit. In the beginning, the first node is the only element inside.
  - Hash set h_v that keeps track of visited nodes. It is initially empty.



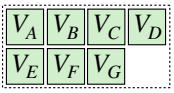
- *Update step*: Dequeue element from q. We have the following cases:
  - *Node already visited*: Skip it.
  - *Node not visited yet*: Add it to the set h_v of visited nodes, and look at its neighbors: if they were not visited before, enqueue them in q.



- *Final step*: Repeat the *update step* until the queue gets empty. The set h_v represents the set of visited nodes.

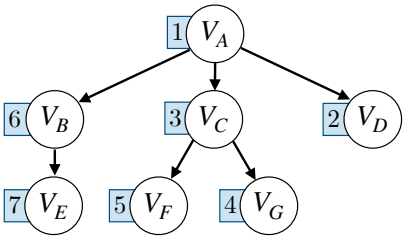


q



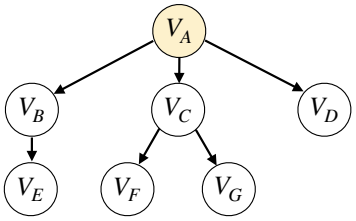
h_v

❒ **Depth-First Search** – Depth-First Search (DFS) is an algorithm that explores a graph by prioritizing depth.



The graph traversal is performed in $\mathcal{O}(|V| + |E|)$ time and $\mathcal{O}(|V|)$ space:

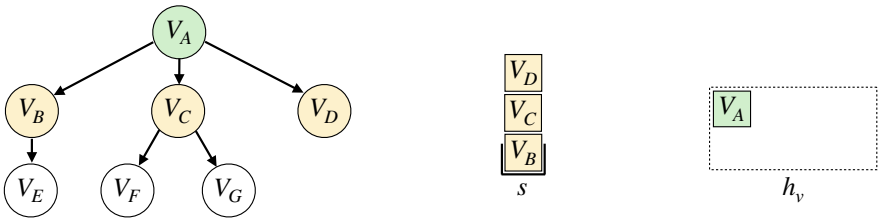
- *Initialization*: The following quantities are tracked:
  - Stack s that keeps track of the next nodes to potentially visit. In the beginning, the first node is the only element inside.
  - Hash set h_v that keeps track of visited nodes. It is initially empty.



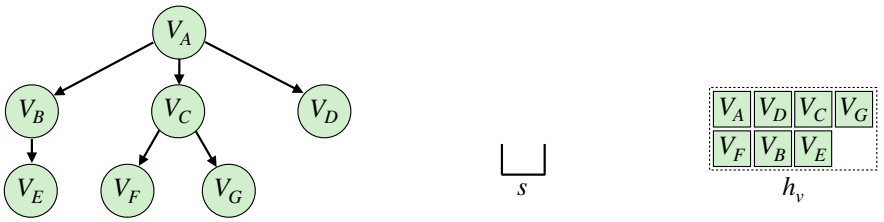
s

h_v

- *Update step*: Pop element from s. We have the following cases:
  - *Node already visited*: Skip it.
  - *Node not visited yet*: Add it to the set h_v of visited nodes, and look at its neighbors: if they were not visited before, push them to s.

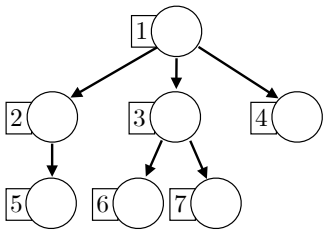
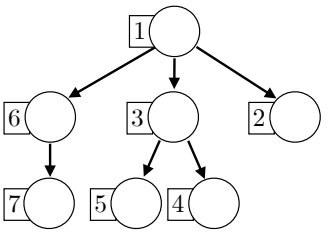


- *Final step*: Repeat the *update step* until the stack gets empty. The set h_v represents the set of visited nodes.



Remark: DFS can also be implemented recursively.

❒ **Graph traversal summary** – The table below highlights the main differences between BFS and DFS:

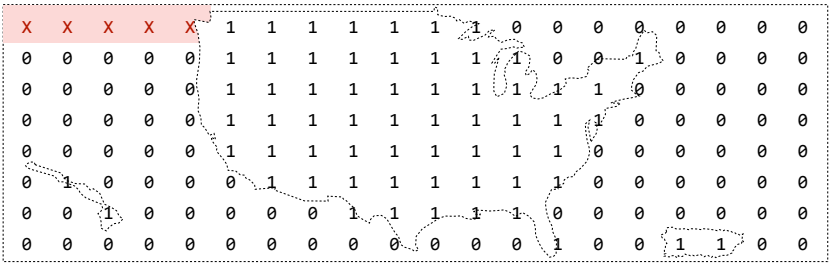
	Breadth-First Search (BFS)	**Depth-First Search** (DFS)
Mindset	Level by level	Depth first
Possible approaches	Iteration using a queue	- Iteration using a stack - Recursion
Illustration		

❒ **Number of islands** – A classic application of graph traversal algorithms is to count the number of islands in an $m \times n$ grid, where each cell is either marked as `1` (land) or `0` (water).

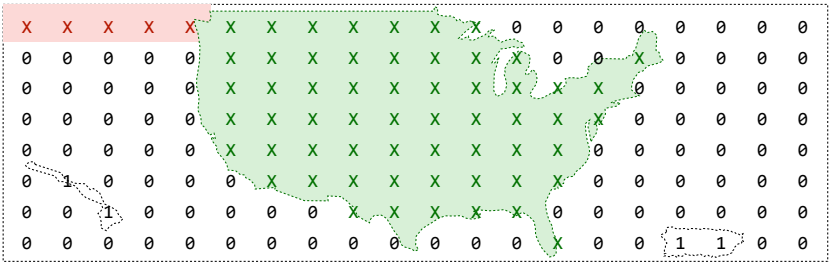
Trick Perform a BFS starting from each unvisited *land* cell of the grid to explore the associated island and skip cells that have already been visited.

Algorithm The entire grid is explored in $\mathcal{O}(m \times n)$ time and $\mathcal{O}(m \times n)$ space as follows:

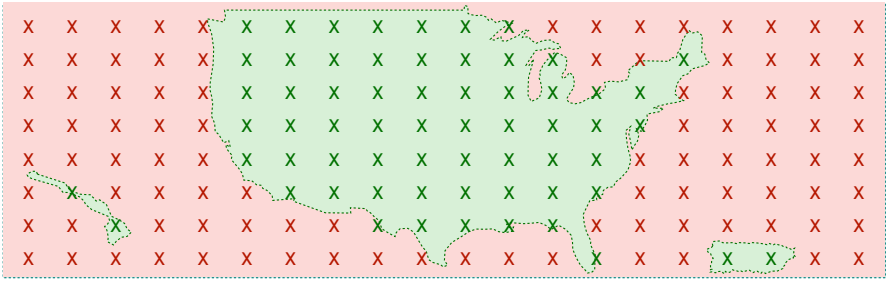
- *Initialization*: Set the counter c that tracks the number of islands to 0.
- *Explore step*: For each cell of the grid, we have the following situations:
  - *Water not visited*: Skip this cell as it is not part of any island.



  - *Land not visited*: Perform a BFS that starts from that cell and visits all neighboring land cells. This process uses a temporary queue q to track nodes to visit, and marks visited nodes by changing their values to `X`.



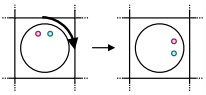
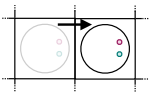
    At the end of the exploration, add 1 to the number of islands c.
  - *Water/Land already visited*: Skip this cell as it was already explored.
- *Final step*: The number of islands in the grid is equal to c.



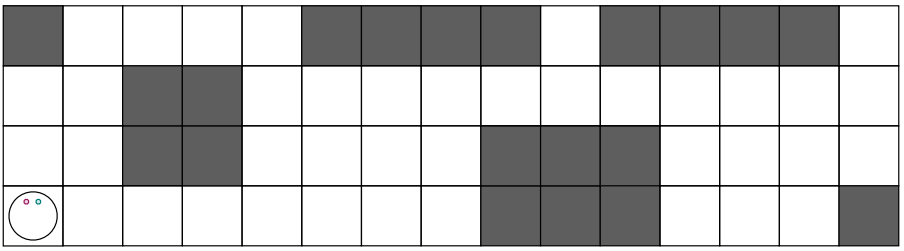
We note that:

- Using an iterative graph traversal algorithm is helpful in preventing stack overflow in case the exploration of an island leads to too many stack calls.
- It would have been equally possible to make island explorations using DFS.

❒ **Robot room cleaner** – The goal of this classic problem is to clean a room composed of n cells using a robot cleaner that can only perform the 3 following actions:

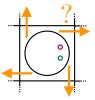
Turn right	**Move cell**	**Clean cell**
Rotate 90° to the right while staying in the same cell. 	Move forward by one cell in the current direction, provided that there is no obstacle. 	Clean current cell 

The configuration of the room is not known by the robot but is assumed to be of finite size. It has obstacles along the way that the robot needs to avoid. We assume that the robot initially starts in a cell with no obstacle.

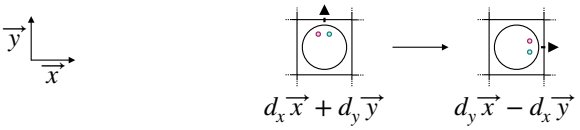


Trick

- Perform a DFS using an exploration strategy that consists of trying all possible directions and that backtracks if an obstacle is found.



- Rotating to the right can be done by keeping track of the direction (d_x, d_y) that the robot is pointing towards.

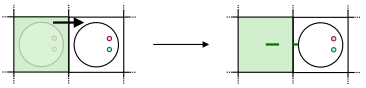


Algorithm The robot cleans the room in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space as follows:

- *Initialization*: We take the convention that:
  - The initial cell of the robot is $(x, y) = (0, 0)$.
  - The robot is initially facing up: $(d_x, d_y) = (0, 1)$.

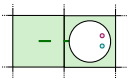
  A hash set is used to keep track of the coordinates of the cells visited by the robot.

- *Explore step*: This is the main step where the robot cleans the cell and moves to an unvisited cell.

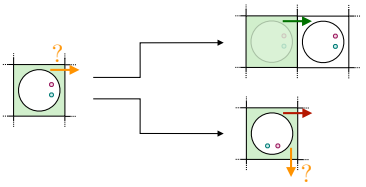


  The robot performs the following actions:

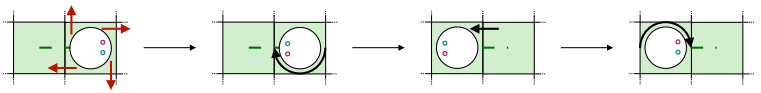
  - It cleans the cell.



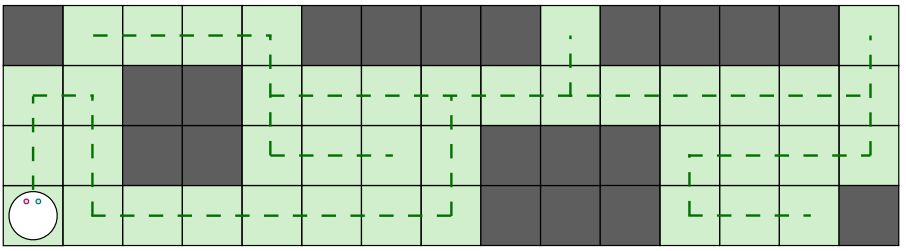
  - It tries to visit each of the 4 neighboring cells by right-rotating and trying to move there. If the neighboring cell is an obstacle or has already been visited, the robot skips it.



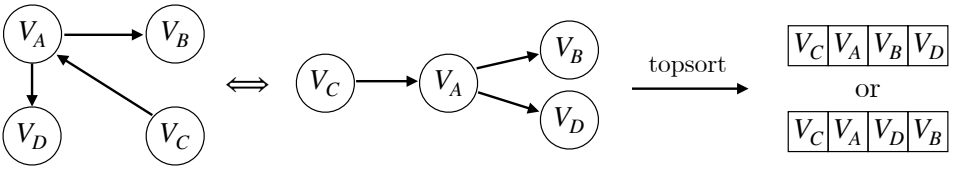
  - When it does not have any more new cells to visit, the robot backtracks by turning right twice, moving and turning right twice again to be pointing back to the initial direction.



- *Final step*: When the robot cleaned all cells, it backtracks to its original cell.



❒ **Topological sort** – In a given directed graph, a topological sort (topsort) is a linear ordering of the nodes such that for all directed edges $E_{ij} = (V_i, V_j)$, V_i appears before V_j.

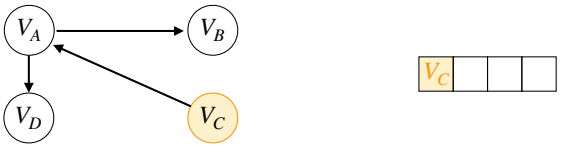


We note that:

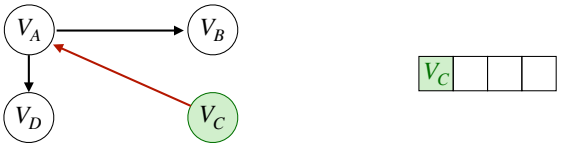
- A graph can have multiple topological orderings.
- A DAG always has a topological sort.
- If a graph contains a cycle, then it cannot have a valid ordering.

❒ **Kahn's algorithm** – Kahn's algorithm aims at finding a topological ordering of a given DAG. It finds a solution in $\mathcal{O}(|V| + |E|)$ time and $\mathcal{O}(|V|)$ space:

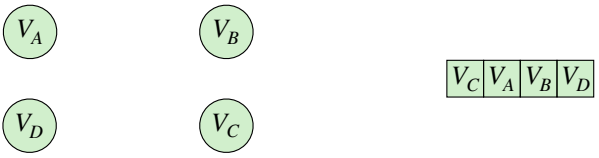
- *Initialization*: Initialize an empty array that keeps track of the final ordering of vertices.
- *Compute step*: Repeat the following procedure until all nodes are visited:
  - Pick a random node of in-degree 0 and visit it.



– Decrement by 1 the in-degrees of the nodes that the newly-visited node is connected to.



- *Final step*: When all nodes are visited, the array contains a valid ordering of vertices.



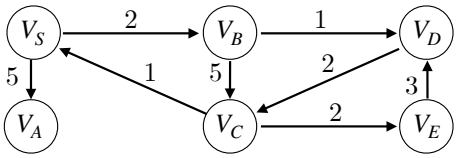
3.1.3 Shortest path

❒ **Weighted graph** – A weighted graph is a graph where each edge E_{ij} has an associated weight w_{ij}.
Remark: When the weight is interpreted as a distance, it is often noted $d_{i,j}$.

❒ **Dijkstra's algorithm** – Dijkstra's algorithm is a greedy algorithm that aims at finding the shortest paths between a source node V_S and all other nodes $j \in V$ of a weighted graph that has no negative edge weights.

The algorithm finds a solution in $\mathcal{O}(|E|\log(|V|))$ time and $\mathcal{O}(|V|)$ space:

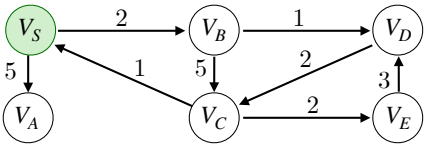
- *Initialization*:
  – Set the distances $d_{V_S,\,j}$ from V_S to each of the other nodes $j \in V$ to $+\infty$ to indicate that they are initially unreachable.
  – Set the distance $d_{V_S,\,V_S}$ from V_S to itself to 0.
  – Initialize a hash table h_p that keeps track of the chosen predecessor for each node. By convention, the predecessor of V_S is itself.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	$+\infty$				

h_p			
	$V_S \to V_S$	$V_A \to ?$	$V_B \to ?$
	$V_C \to ?$	$V_D \to ?$	$V_E \to ?$

- *Compute step*: Repeat the following procedure until all nodes are visited:
  - Pick the unvisited node i with the lowest distance $d_{V_S,\,i}$ and visit it.



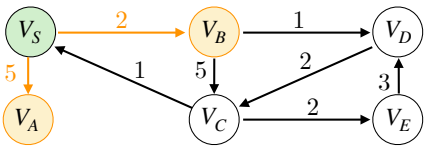
$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	$+\infty$				

h_p			
	$V_S \to V_S$	$V_A \to ?$	$V_B \to ?$
	$V_C \to ?$	$V_D \to ?$	$V_E \to ?$

  - Look at the unvisited nodes j that have an edge coming from the newly-visited node i.
    - *Case* $d_{V_S,\,j} > d_{V_S,\,i} + d_{i,j}$: This means that the distance of the path from node V_S to j via node i is smaller than the current distance. We update the corresponding distance:

$$\boxed{d_{V_S,\,j} \longleftarrow d_{V_S,\,i} + d_{i,j}}$$

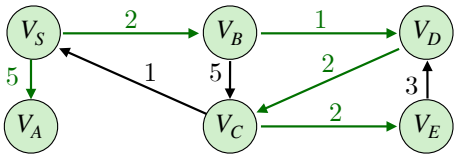
      Also, we update h_p to indicate that the predecessor of j is now i.
    - *Case* $d_{V_S,\,j} \leqslant d_{V_S,\,i} + d_{i,j}$: This means that the proposed path does not improve the current distance. We do not make any updates.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	$+\infty$		

h_p			
	$V_S \to V_S$	$V_A \to V_S$	$V_B \to V_S$
	$V_C \to ?$	$V_D \to ?$	$V_E \to ?$

- *Final step*:
  - The final distances are those of the shortest paths between V_S and each of the other nodes $j \in V$ of the graph.
  - The associated shortest paths can be reconstructed using h_p.



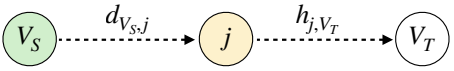
$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	5	3	7

h_p			
$V_S \to V_S$	$V_A \to V_S$	$V_B \to V_S$	
$V_C \to V_D$	$V_D \to V_B$	$V_E \to V_C$	

Remark: Given that choices are based on local optima, Dijkstra's algorithm does not guarantee a correct solution when there are negative edge weights. Indeed, if such an edge were to be found "too late" in the exploration process, the algorithm may visit a node earlier than desired and produce a suboptimal path.

❒ **A* algorithm** – The A* algorithm is an extension of Dijkstra's algorithm that is used when the goal is to find the shortest path between a source node V_S and a fixed target node V_T.

To better direct the search towards the target node, it modifies the distance used to select the next node by using a heuristic $h_{j,\,V_T}$ that approximates the remaining distance between unvisited nodes j and the target node V_T.



The distance used to determine the next node j to visit is given by $d^{A^\star}$:

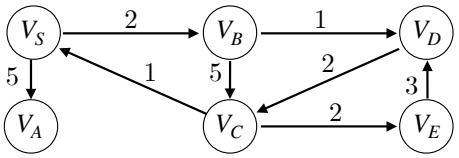
$$\boxed{d^{A^\star}_{V_S,\,j} = d_{V_S,\,j} + h_{j,\,V_T}}$$

Remark: The A* algorithm heavily relies on a good choice of heuristic h.

❒ **Bellman-Ford algorithm** – The Bellman-Ford algorithm aims at finding the shortest paths between a source node V_S and all other nodes $j \in V$ of a weighted graph.

A solution is found in $\mathcal{O}(|V| \times |E|)$ time and $\mathcal{O}(|V|)$ space:

- *Initialization*:
  - Set the distances $d_{V_S,\,j}$ from V_S to each of the other nodes $j \in V$ to $+\infty$ to indicate that they are initially unreachable.
  - Set the distance $d_{V_S,\,V_S}$ from V_S to itself to 0.
  - Initialize a hash table h_p that keeps track of the chosen predecessor for each node. By convention, the predecessor of V_S is itself.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	$+\infty$				

h_p			
	$V_S \to V_S$	$V_A \to ?$	$V_B \to ?$
	$V_C \to ?$	$V_D \to ?$	$V_E \to ?$

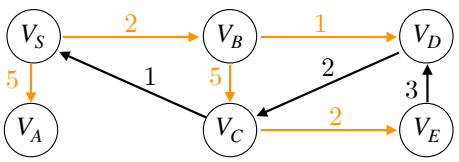
- *Compute step*: Repeat the following procedure until there is no more updates, which is at most $|V| - 1$ times:

  For each node i of the graph, look at its neighbors j and update the distance $d_{V_S,\,j}$ between the source node V_S and the node j depending on the following situations:

  - *Case* $d_{V_S,\,j} > d_{V_S,\,i} + d_{i,j}$: This means that the distance of the path from node V_S to j via node i is smaller than the current distance. We update the corresponding distance:

    $$\boxed{d_{V_S,\,j} \longleftarrow d_{V_S,\,i} + d_{i,j}}$$

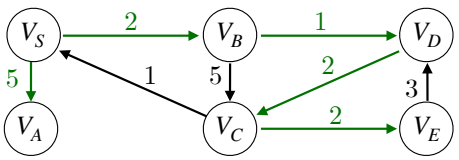
    Also, we update h_p to indicate that the predecessor of j is now i.
  - *Case* $d_{V_S,\,j} \leqslant d_{V_S,\,i} + d_{i,j}$: This means that the proposed path does not improve the current distance. We do not make any updates.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	7	3	9

h_p			
	$V_S \to V_S$	$V_A \to V_S$	$V_B \to V_S$
	$V_C \to V_B$	$V_D \to V_B$	$V_E \to V_C$

- *Final step*:
  - The final distances are those of the shortest paths between V_S and each of the other nodes $j \in V$ of the graph.
  - The associated shortest paths can be reconstructed using h_p.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	5	3	7

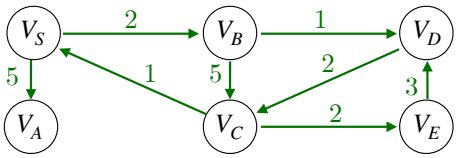
h_p			
	$V_S \to V_S$	$V_A \to V_S$	$V_B \to V_S$
	$V_C \to V_D$	$V_D \to V_B$	$V_E \to V_C$

Compared to Dijkstra's algorithm, Bellman-Ford's algorithm has the advantage of supporting negative edge weights. However, it cannot run on a graph having negative cycles, i.e. cycles with a negative sum of edge weights.

❒ Floyd-Warshall algorithm – The Floyd-Warshall algorithm aims at finding the shortest paths between all pairs of nodes of a weighted graph.

A solution is found in $\mathcal{O}(|V|^3)$ time and $\mathcal{O}(|V|^2)$ space:

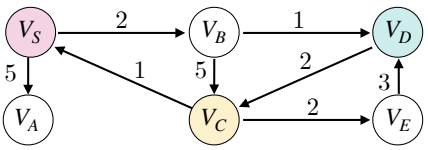
- *Initialization*: Initialize a matrix of distances of size $|V| \times |V|$, where each cell (i, j) represents the distance of the shortest path from node i to node j. We know that:
  - The distance of a node to itself is 0, so each element of the diagonal is equal to 0.
  - The distances of nodes linked by edges are given.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	$+\infty$		
V_A	$+\infty$	0	$+\infty$			
V_B	$+\infty$		0	5	1	$+\infty$
V_C	1	$+\infty$		0	$+\infty$	2
V_D	$+\infty$			2	0	$+\infty$
V_E	$+\infty$				3	0

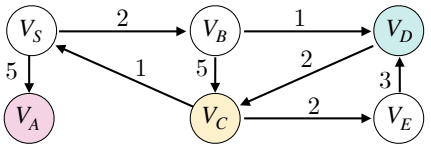
- *Update step*: For each intermediary node $k \in V$, loop through start nodes $i \in V$ and end nodes $j \in V$:
  - *Case* $d_{i,j} > d_{i,k} + d_{k,j}$: This means that the distance from node i to j via node k is smaller than the current distance. We make the following update:

$$d_{i,j} \longleftarrow d_{i,k} + d_{k,j}$$



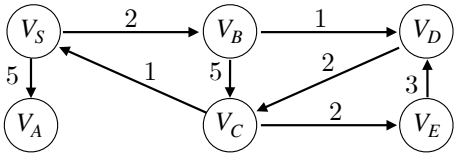
$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	$+\infty$		
V_A	$+\infty$	0	$+\infty$			
V_B	$+\infty$		0	5	1	$+\infty$
V_C	1	$+\infty$		0	$+\infty$	2
V_D	3	$+\infty$		2	0	$+\infty$
V_E	$+\infty$				3	0

– *Case* $d_{i,j} \leqslant d_{i,k} + d_{k,j}$: This means that the proposed path does not improve the current distance. We do not make any updates.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	$+\infty$		
V_A	$+\infty$	0	$+\infty$			
V_B	$+\infty$		0	5	1	$+\infty$
V_C	1	$+\infty$	$+\infty$	0	$+\infty$	2
V_D	3	$+\infty$	$+\infty$	2	0	$+\infty$
V_E	$+\infty$				3	0

- *Final step*: The resulting matrix gives the shortest path between each pair of nodes i and j.



$d_{i,j}$	V_S	V_A	V_B	V_C	V_D	V_E
V_S	0	5	2	5	3	7
V_A	$+\infty$	0	$+\infty$			
V_B	4	9	0	3	1	5
V_C	1	6	3	0	4	2
V_D	3	8	5	2	0	4
V_E	6	11	8	5	3	0

This algorithm allows for negatively weighted edges but it does not allow for negative cycles like the other shortest path algorithms.

Remark: In the same fashion as in Dijkstra's and Bellman-Ford's algorithms, we can reconstruct the resulting shortest paths by keeping track of nodes during updates.

❒ **Shortest path summary** – The differences between the main shortest paths algorithms are summarized below:

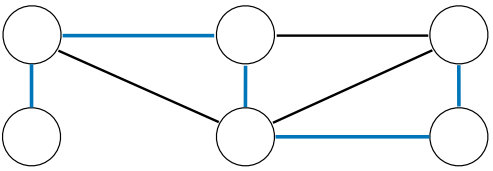
Algorithm	Shortest path between	Negative weights	Complexity	
			Time	Space
Dijkstra	V_S and all other nodes	No	$\mathcal{O}(\lvert E\rvert \log(\lvert V\rvert))$	$\mathcal{O}(\lvert V\rvert)$
A*	V_S and V_T	No	$\mathcal{O}(\lvert E\rvert \log(\lvert V\rvert))$	$\mathcal{O}(\lvert V\rvert)$
Bellman-Ford	V_S and all other nodes	Yes	$\mathcal{O}(\lvert V\rvert \times \lvert E\rvert))$	$\mathcal{O}(\lvert V\rvert)$
Floyd-Warshall	All pairs of nodes	Yes	$\mathcal{O}(\lvert V\rvert^3)$	$\mathcal{O}(\lvert V\rvert^2)$

3.2 Advanced graph algorithms

In this part, we will learn about more advanced graph concepts such as minimum spanning trees and connected components.

3.2.1 Spanning trees

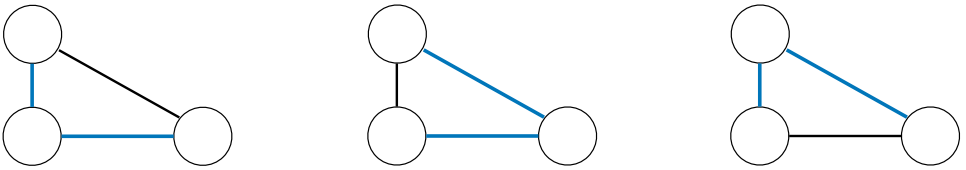
❒ **Definition** – A spanning tree of an undirected graph $G = (V, E)$ is defined as a subgraph that has the minimum number of edges $E' \subseteq E$ required for all vertices V to be connected.



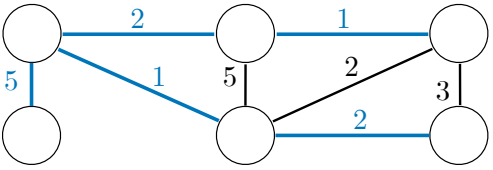
Remark: In general, a connected graph can have more than one spanning tree.

❒ **Cayley's formula** – A complete undirected graph with N vertices has N^{N-2} different spanning trees.

For instance, when $N = 3$, there are $3^{3-2} = 3$ different spanning trees:



❒ **Minimum spanning tree** – A minimum spanning tree (MST) of a weighted connected undirected graph is a spanning tree that minimizes the total edge weight.



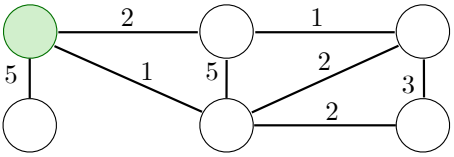
For example, the MST shown above has a total edge weight of 11.

Remark: A graph can have more than one MST.

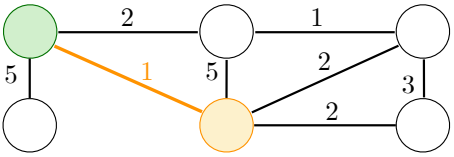
❒ **Prim's algorithm** – Prim's algorithm aims at finding an MST of a weighted connected undirected graph.

A solution is found in $\mathcal{O}(|E|\log(|V|))$ time and $\mathcal{O}(|V|+|E|)$ space with an implementation that uses a min-heap:

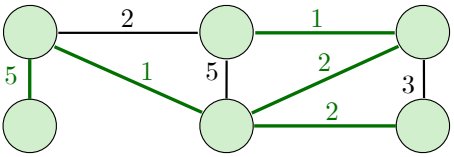
- *Initialization*: Pick an arbitrary node in the graph and visit it.



- *Compute step*: Repeat the following procedure until all nodes are visited:
  - Pick an unvisited node that connects one of the visited notes with the smallest edge weight.
  - Visit it.



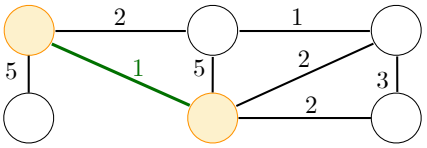
- *Final step*: The resulting MST is composed of the edges that were selected by the algorithm.



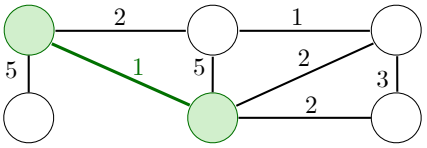
❒ **Kruskal's algorithm** – The goal of Kruskal's algorithm is to find an MST of a weighted connected undirected graph.

A solution is found in $\mathcal{O}(|E|\log(|E|))$ time and $\mathcal{O}(|V|+|E|)$ space:

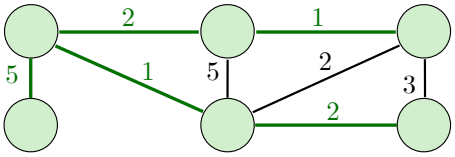
- *Compute step*: Repeat the following procedure until all nodes are visited:
  - Pick an edge that has the smallest weight such that it does not connect two already-visited nodes.



– Visit the nodes at the extremities of the edge.

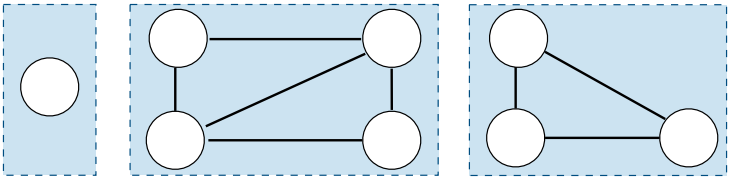


- *Final step*: The resulting MST is composed of the edges that were selected by the algorithm.

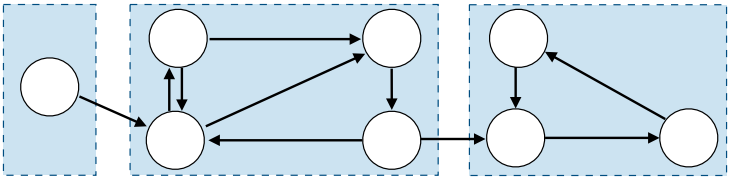


3.2.2 Components

❒ **Connected components** – In a given undirected graph, a connected component is a maximal connected subgraph.



❒ **Strongly connected components** – In a given directed graph, a strongly connected component is a maximal subgraph for which a path exists between each pair of vertices.

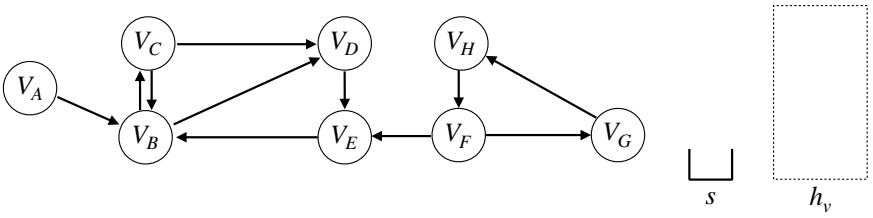


❒ **Kosaraju's algorithm** – Kosaraju's algorithm aims at finding the strongly connected components of a directed graph.

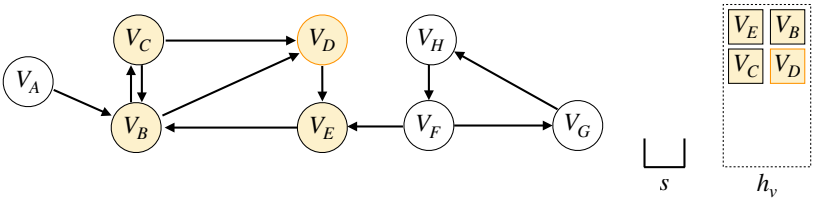
The solution is found in $\mathcal{O}(|V| + |E|)$ time and $\mathcal{O}(|V|)$ space:

First-pass DFS On the original graph, perform a DFS.

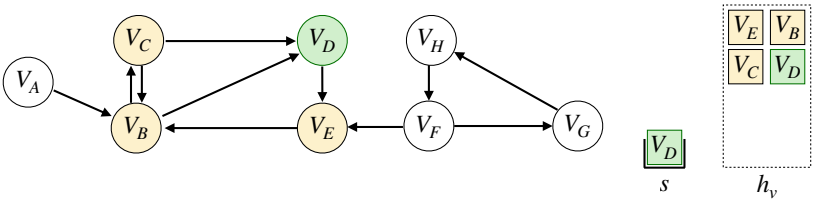
- *Initialization*: The following quantities are used:
  - An initially empty hash set h_v that keeps track of the visited nodes.
  - An initially empty stack s that keeps track of the order at which the nodes have had their neighbors visited.



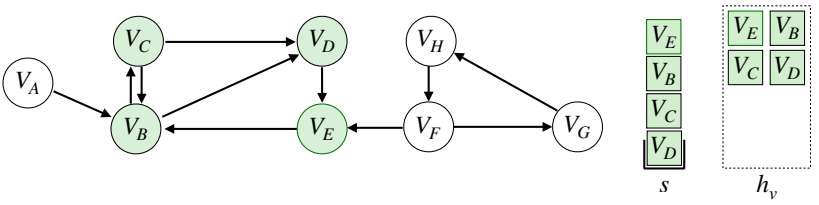
- *Compute step*: Perform the following actions:
  - Pick an arbitrary node and visit it by adding it to h_v. Recursively visit all its neighboring nodes.



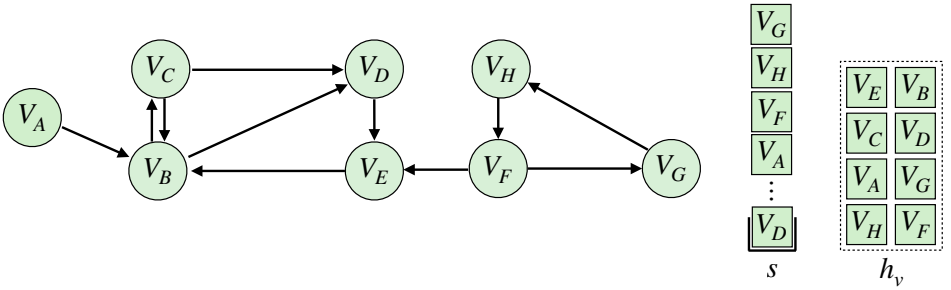
  - Once a visited node has all its neighbors visited, push the visited node into the stack s.



The recursive procedure adds the remaining nodes to the stack s.

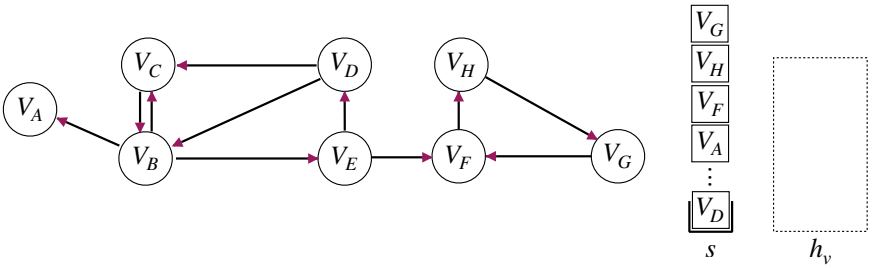


The same process is successively initiated on the remaining unvisited nodes of the graph until all nodes are visited.

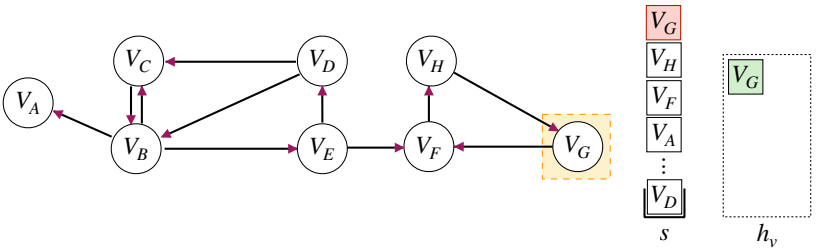


Second-pass DFS On the reverted graph, perform a DFS to determine the strongly connected components.

- *Initialization*: The following quantities are used:
  - The stack s obtained from the first-pass DFS.
  - An initially empty hash set h_v that keeps track of the visited nodes.



- *Compute step*: Pop a node from the stack.
  - *Node is not visited*: This node is the representative of a new strongly connected component. Add it to h_v.

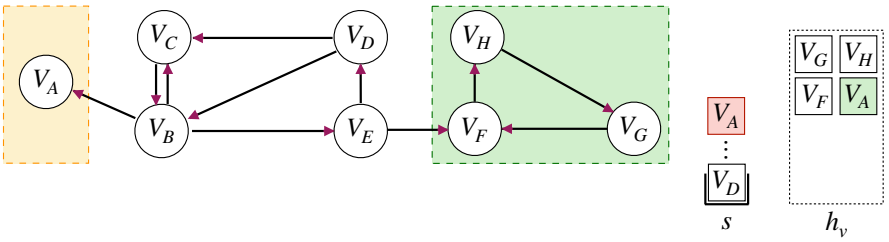


Perform a DFS starting from that node. For the nodes that are being explored:

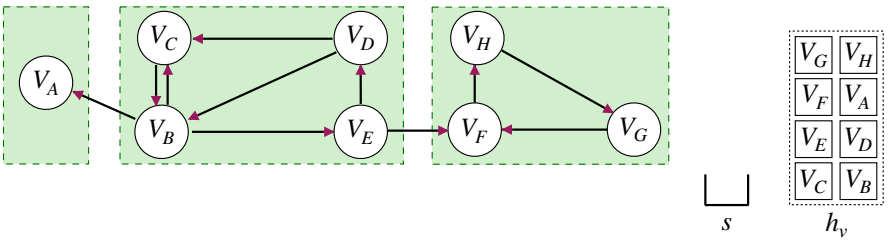
  * *DFS node is not visited*: Add it to h_v and add the node to the hash set identifying the strongly connected component.
  * *DFS node is visited*: Skip it.

– *Node is visited*: Skip it.

Repeat this process again...



- *Final step*: ...until the stack is empty.



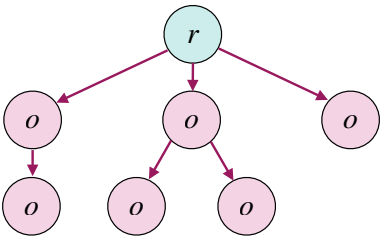
3.3 Trees

In this part, we will learn about trees and focus on various types of binary trees, including heaps and binary search trees. Then, we will focus on applications of generalized types of trees such as tries.

3.3.1 General concepts

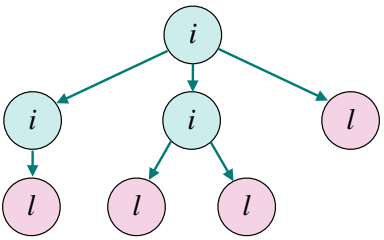
❒ **Definition** – A tree is a DAG with the following properties:

- *Incoming edge*:
  - There is exactly one node that has no incoming edge, and that node is called the root.
  - Each of the other nodes has exactly one incoming edge.
- *Outgoing edge*: A node cannot have an outgoing edge that points to itself.

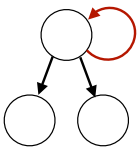
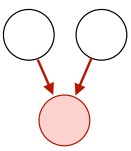
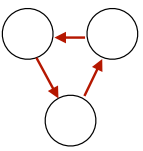
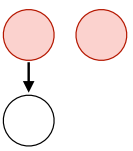


There are two types of nodes in a tree:

- *Internal node*: Node that has one or more children.
- *Leaf*: Node that does not have any children.

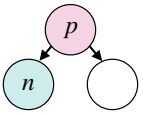
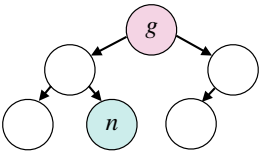
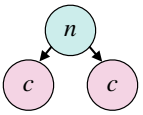
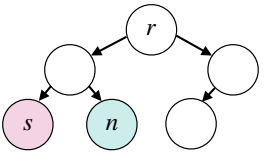
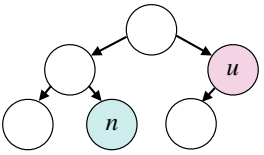


Here are some examples of graphs that are not trees, along with the associated reason:

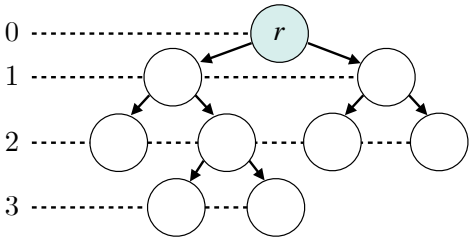
Self-loop	Multiple parents	Cycle	Multiple roots
			

Remark: By construction, a tree with N nodes has $N-1$ edges.

❒ **Notations** – Names given to nodes follow a family-like vocabulary. The most common ones are presented in the table below:

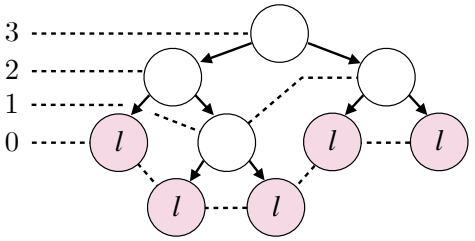
Concept	Description	Illustration
Parent	The parent p of a node n is the predecessor of that node. n has at most one parent.	
Grandparent	The grandparent g of a node n is the predecessor of the predecessor of that node. n has at most one grandparent.	
Child	A child c of a node n is a successor of that node. n can have 0, 1 or multiple children.	
Sibling	A sibling s of a node n is a different node with the same parent as n. n can have 0, 1 or multiple siblings.	
Uncle	An uncle u of a node n is a child of n's grandparent that is not its parent. n can have 0, 1 or multiple uncles.	

❒ **Depth** – The depth of a given node n, noted depth(n), is the number of edges from the root r of the tree to node n.



Remark: The root node has a depth of 0.

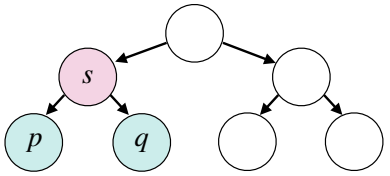
❒ **Height** – The height of a given node n, noted height(n), is the number of edges of the longest path from node n to the deepest leaf.

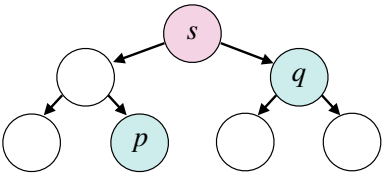


The height of a tree is the height of its root node.

Remark: Leaf nodes have a height of 0.

❒ **Lowest Common Ancestor** – In a given tree, the lowest common ancestor of two nodes p and q, noted LCA(p, q), is defined as being the deepest node that has both p and q as descendants. The examples below use $s \triangleq \text{LCA}(p, q)$.

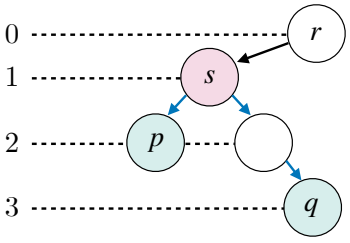




Remark: For the purposes of this definition, we may consider a node to be a descendant of itself.

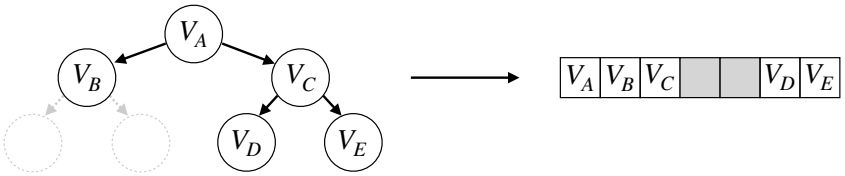
❒ **Node distance** – The distance $d(p,q)$ between two nodes p and q is the minimum number of edges between these two nodes. If we note $s \triangleq \text{LCA}(p,q)$, we have the following formula:

$$\boxed{d(p,q) = \text{depth}(p) + \text{depth}(q) - 2 \times \text{depth}(s)}$$



For instance, nodes p and q have a distance of 3 in the tree above.

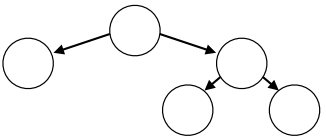
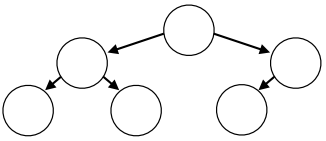
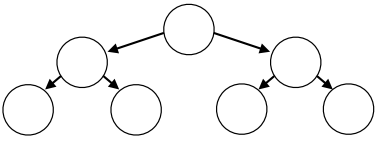
❒ **Serialization** – Tree serialization is a method to encode a tree into a generic format (e.g. string) without losing information.



Remark: There can be more than one way to serialize a tree. The figure above illustrates a level-by-level approach in the case of binary trees.

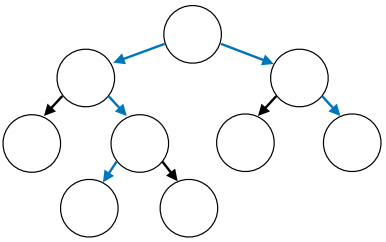
3.3.2 Binary trees

❒ **Definition** – A binary tree is a tree where each node has at most 2 children. The table below sums up possible properties of binary trees:

Type	Description	Illustration
Full	Every node has either 0 or 2 children.	
Complete	All levels are completely filled, except possibly the last one where all nodes are to the left.	
Perfect	All internal nodes have 2 children and all leaves are at the same level.	

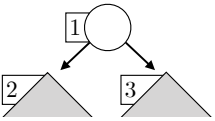
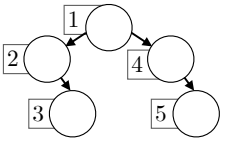
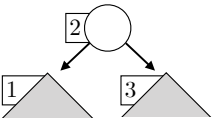
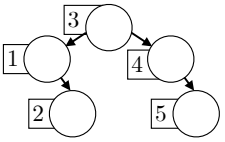
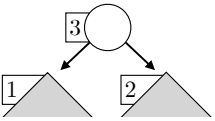
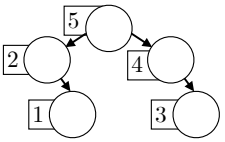
Remark: A perfect tree has exactly $2^{h+1} - 1$ nodes with h the height of the tree.

❒ **Diameter** – The diameter of a binary tree is the longest distance between any of its two nodes.



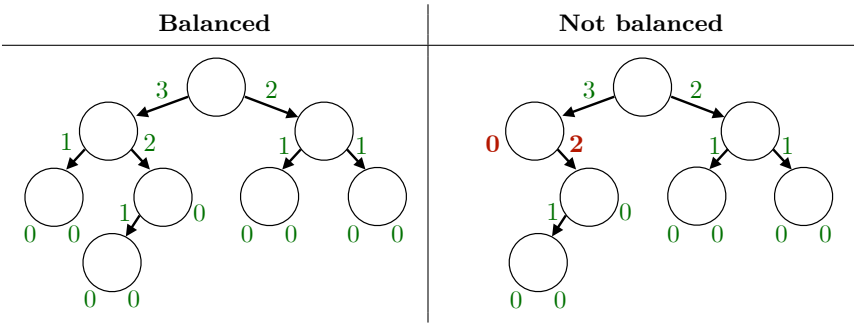
For example, the binary tree above has a diameter of 5.

❒ **Main tree traversals** – The table below summarizes the 3 main ways to recursively traverse a binary tree:

Traversal	Algorithm	Illustration	Example
Pre-order	1. Visit the `root` 2. Visit the `left` subtree 3. Visit the `right` subtree		
In-order	1. Visit the `left` subtree 2. Visit the `root` 3. Visit the `right` subtree		
Post-order	1. Visit the `left` subtree 2. Visit the `right` subtree 3. Visit the `root`		

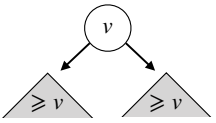
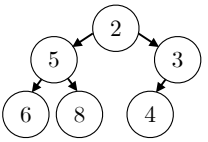
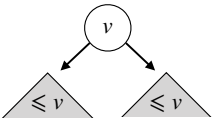
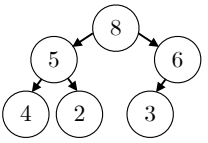
Remark: Pre-order, in-order and post-order traversals are all variations of DFS.

❒ **Balanced tree** – A binary tree is said to be balanced if the difference in height of each node restricted to its left and right subtrees is at most 1.

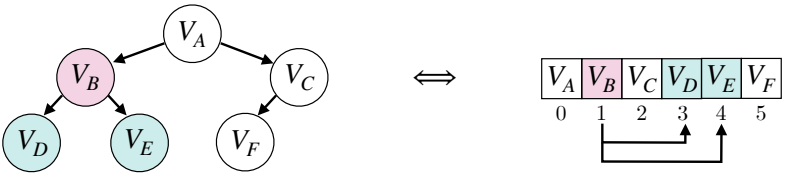


3.3.3 Heaps

❒ **Definition** – A heap is a complete binary tree with an additional property that makes it either a min-heap or a max-heap:

Type	Description	Illustration	Example
Min-heap	The value of each node is lower than its children's, if any. By definition, the root has the lowest value.		
Max-heap	The value of each node is higher than its children's, if any. By definition, the root has the highest value.		

Since a heap is a complete binary tree, it is convenient to represent it with an array of size n, where n is the number of nodes.

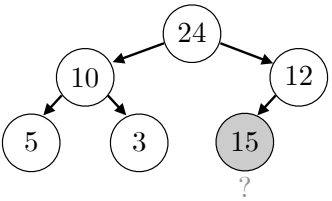


For a given node of index $i \in [\![0, n-1]\!]$,

- its parent has an index of $\left\lfloor \frac{i-1}{2} \right\rfloor$
- its left child has an index of $2i+1$ and its right child has an index of $2i+2$

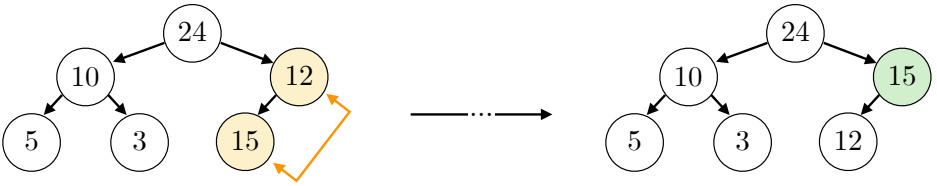
The following parts will use max-heaps for consistency. However, one could also use min-heaps to obtain the same results.

❒ **Heapify up** – Let's suppose that the last child is potentially not fulfilling the properties of the max-heap. The heapify up operation, also called *bubble up*, aims at finding the correct place for this node.

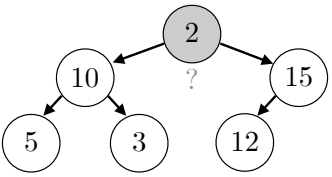


This operation is done in $\mathcal{O}(\log(n))$ time:

- *Update step*: While the node's parent has a lower value than the node's, swap the two.
- *Final step*: We now have a valid max-heap.

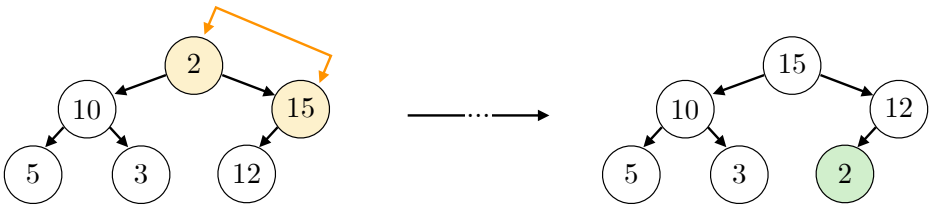


❒ **Heapify down** – Let's suppose that the root is potentially not fulfilling the properties of the max-heap. The heapify down operation, also called *bubble down*, aims at finding the correct place for this node.



This operation is done in $\mathcal{O}(\log(n))$ time:

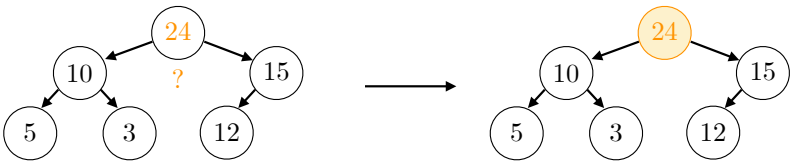
- *Update step*: While the highest-value child of the node has a higher value than the node's, swap the two.
- *Final step*: We now have a valid max-heap.



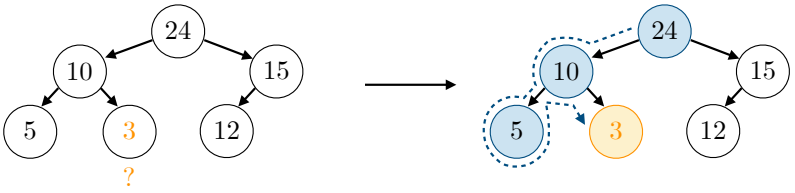
❒ **Operations** – The main operations that can be performed on a max-heap are explained below:

Search We distinguish two cases:

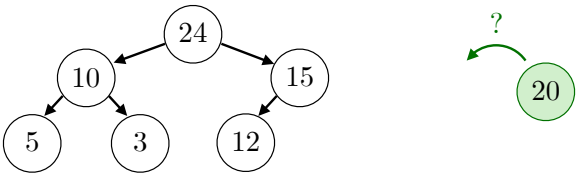
- *Maximum value*: Look at the value corresponding to the root of the heap. It takes $\mathcal{O}(1)$ time.



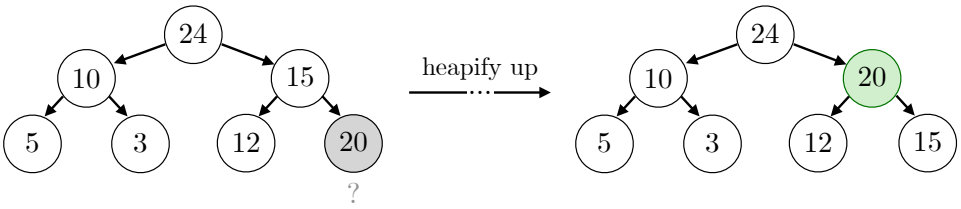
- *Any other value*: Traverse the tree, given that we have no information as to where each node is. It takes $\mathcal{O}(n)$ time.



Insertion It takes $\mathcal{O}(\log(n))$ time.

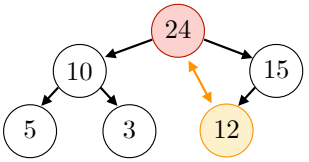


- *Placeholder step*: Add new node as the last child.
- *Heapify up step*: Heapify up the child to its final position.

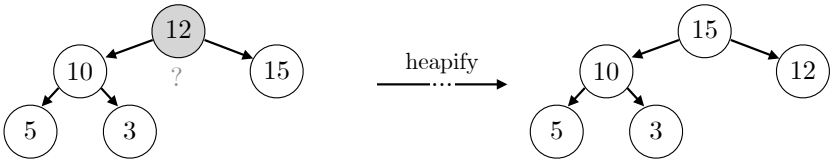


Deletion It takes $\mathcal{O}(\log(n))$ time.

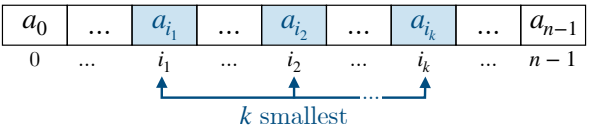
- *Swap step*: Swap node with last child and remove new last child.



- *Heapify step*: Move the newly-placed node to its final position depending on the situation:
  - *Node's value is higher than parent's*: Heapify up to its final position.
  - *Node's value is lower than highest of its children*: Heapify down to its final position.
  - *Node's value is lower than parent's and higher than highest of its children*: There is nothing to do.



❒ k **smallest elements** – Given array $A = [a_0, ..., a_{n-1}]$, the goal is to find the k smallest elements, with $k \leqslant n$.

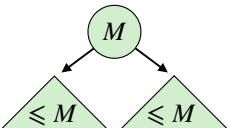


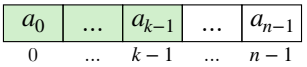
Trick Use a max-heap of size k.

Algorithm The solution is found in $\mathcal{O}(n\log(k))$ time and $\mathcal{O}(k)$ space:

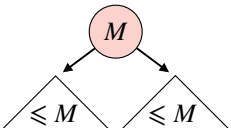
- *Initialization*:
  - Set an empty max-heap that will keep track of the k smallest elements.
  - Add the first k elements into the max-heap.

  At any given point, we note M the value of the root of the max-heap, i.e. its maximum value.

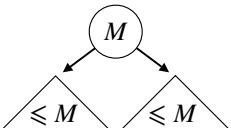




- *Update step*: We need to see whether any of the remaining elements $a_{i\in[\![k,n-1]\!]}$ is potentially part of the k smallest elements:
  - If $a_i < M$, pop the max-heap and insert a_i in $\mathcal{O}(\log(k))$ time.



  - If $a_i \geqslant M$, it means that the element is greater than any of the current k smallest elements. We do not do anything. The check is done in $\mathcal{O}(1)$ time.



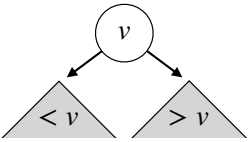
- *Final step*: The max-heap contains the k smallest elements.

Remark: Similarly, the k largest elements can be retrieved using a min-heap.

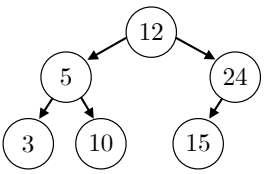
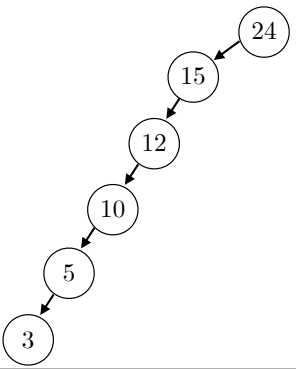
3.3.4 Binary search trees

❒ **Definition** – A binary search tree (BST) is a binary tree where each node has the following properties:

- Its value is greater than any node values in its left subtree
- Its value is less than any node values in its right subtree

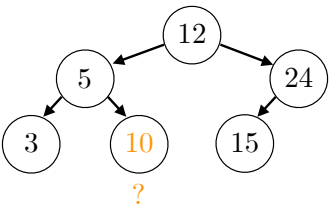


We note h the height of the BST, where h can go anywhere between:

	Best case	**Worst case**
Description	Tree is balanced	Every node has at most 1 child
Height	$h \approx \log(n)$	$h = n$
Illustration		

❒ **Operations** – The main operations that can be performed on a BST each have a time complexity of $\mathcal{O}(h)$ and are explained below:

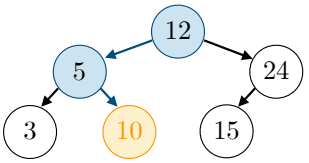
Search Starting from the root, compare the node value with the value v we want to search for.



- *Node value different than v*:

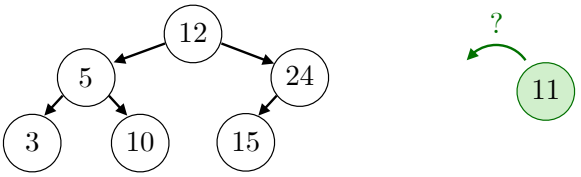
  - If it is higher than v, go to its left child.
  - If it is lower than v, go to its right child.

- *Node value equal to v*: We found the target node.



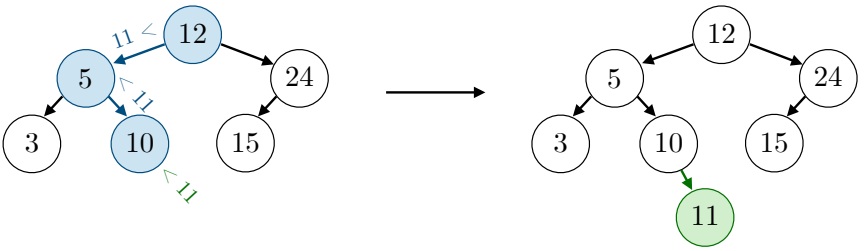
Continue this process recursively until either finding the target node or hitting the end of the tree, in which case v is not present in the BST.

Insertion Let's suppose we want to insert a node of value v in the BST.



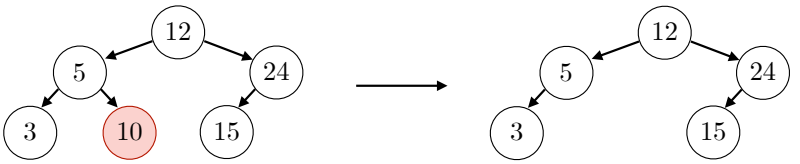
In order to do that, we check if there is already a node with the same value in $\mathcal{O}(h)$ time. At the end of the search, there are two possible situations:

- *Node is found*: There is nothing else to do, since the element that we want to insert is already in the tree.
- *Node is not found*: By construction, the last node seen during the search is a leaf.
  - If v is higher than the value of the last node, add it as its right child.
  - If v is lower than the value of the last node, add it as its left child.

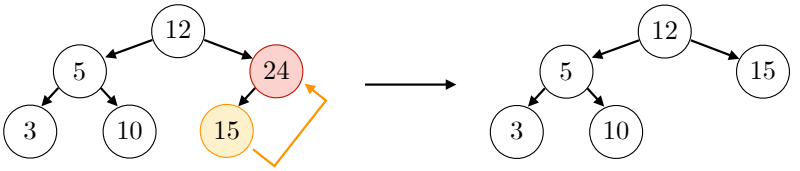


Deletion The process depends on the number of children of the node to be deleted. We have the following situations:

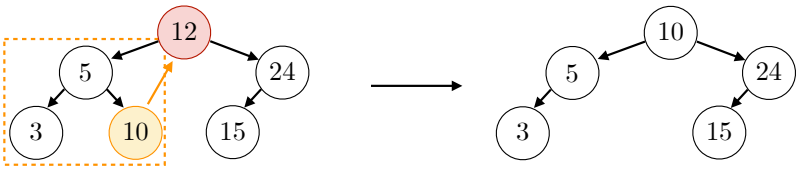
- *0 child*: Delete the node.



- *1 child*: Replace the node with its child.

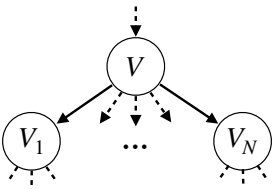


- *2 children*: Replace the node with the max of its left subtree, which is also equal to the in-order predecessor.

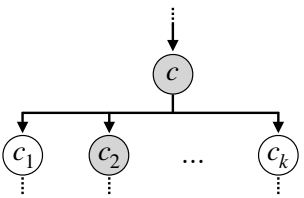


3.3.5 *N*-ary trees

❒ **Definition** – An N-ary tree is a tree where each node has at most N children.



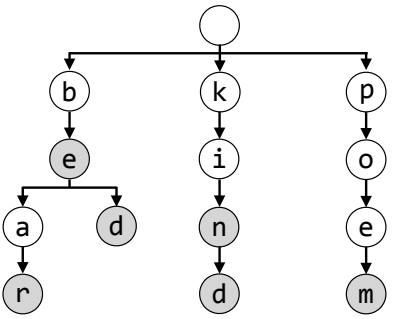
❒ **Trie** – A trie, also called prefix tree, is an N-ary tree that allows for efficient storing and fast retrieval of words. The path from the root node to another given node forms a word that is deduced by concatenating the characters along that path.



Each node n is defined by the following quantities:

- A character c.
- A hash table $h = \{(c_1, n_1), ..., (c_k, n_k)\}$ gathering the children of that node, where c_i is the child's character and n_i its associated node.
- A boolean that indicates whether the word formed by the path leading to that node is in the dictionary ◉ or not ○.

By convention, the root is a node for which c is an empty string.



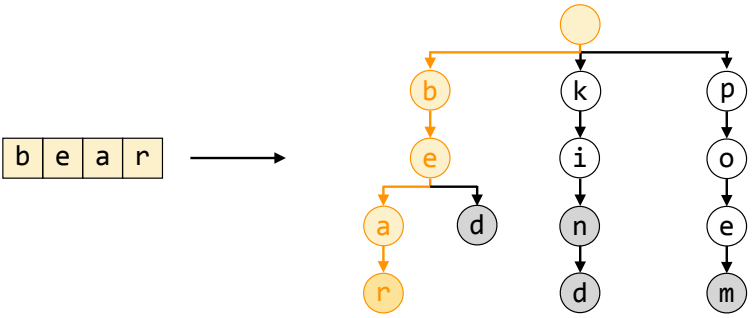
Note that even though all characters of a word are present in the correct order, it does not necessarily mean that the word itself is present in the trie. In order to

ensure that the word is indeed in the trie, an important additional condition is for the boolean of the last character to be set to ◯.

The operations that can be done with a trie are described below:

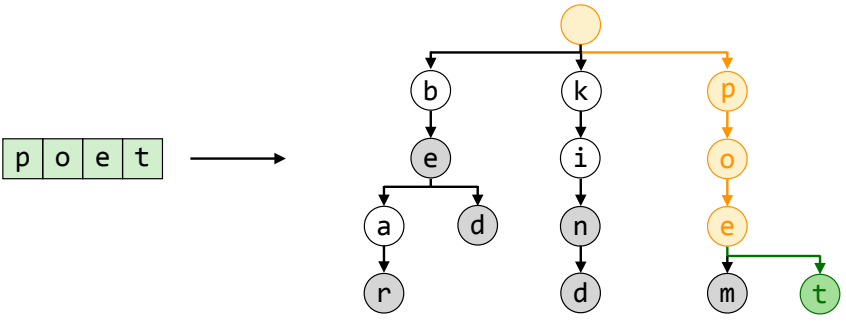
Search We would like to know whether a word w is in the trie.

- *Character step*: Starting from the root, traverse the trie by checking one character of w at a time. If any character is missing, it means that the word is not present.
- *Word step*: Check whether the boolean of the node corresponding to the last character is set to ◯. If it is not, it means that the word is not present.



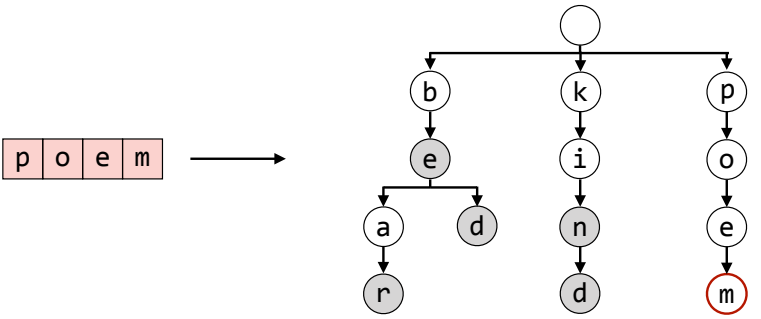
Insertion We would like to insert a word w in the trie.

- *Character step*: Starting from the root, traverse the trie one character of w at a time. Starting from the first missing character (if applicable), add corresponding nodes until completing the word.
- *Word step*: Set the boolean of the last character to ◯.



Deletion We would like to remove a word w from the trie.

- *Character step*: Traverse the trie one character of w at a time.
- *Word step*: Set the boolean of the last character to ◯.



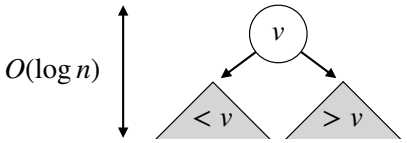
Remark: Use cases of a trie include searching for a word starting with a given prefix.

3.4 Advanced trees

In this part, we will learn about self-balancing trees such as red-black trees and AVLs, along with more advanced trees that can handle range sum queries such as binary indexed trees and segment trees.

3.4.1 Self-balancing trees

❒ **Definition** – A self-balancing tree is a BST that keeps its height in $\mathcal{O}(\log(n))$ by maintaining specific properties. Examples of such trees include red-black trees and AVL trees.



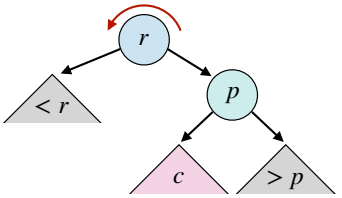
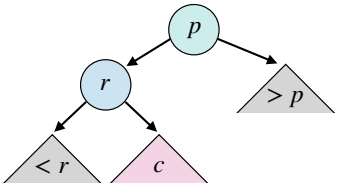
❒ **Tree rotations** – Most self-balancing tree operations involve tree rotations, which are done by changing the shape of the BST while keeping its properties satisfied. There are two kinds of rotations: left and right rotations. We give the recipe to do both below.

We arbitrarily focus on the three following nodes, which are useful during a tree rotation:

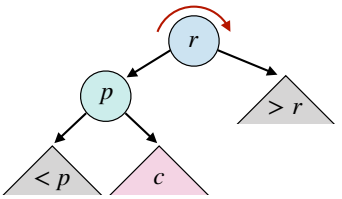
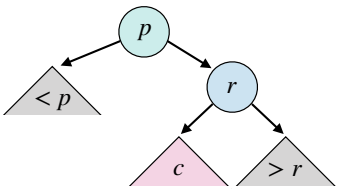
- *Pivot p*: Node that becomes the new root.
- *Original root r*: Root of the original tree.
- *Moving child c*: Node that changes parents.

Tree rotations have a time complexity of $\mathcal{O}(1)$.

Left rotation

	Before	After
Pivot	Right child of root	New root
Original root	Root	Left child of pivot
Moving child	Left child of pivot	Right child of original root
Illustration		

Right rotation

	Before	After
Pivot	Left child of root	New root
Original root	Root	Right child of pivot
Moving child	Right child of pivot	Left child of original root
Illustration		

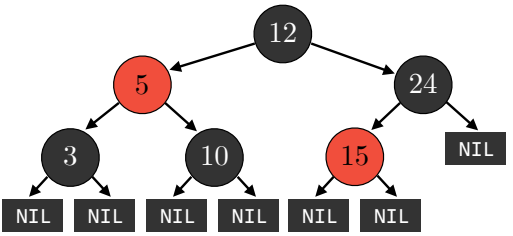
Remark: Performing tree rotations does not change the in-order traversal of the BST.

❒ **Red-black tree** – A red-black tree is a self-balancing tree with height $\mathcal{O}(\log(n))$ that maintains the following properties:

- *Coloring*:

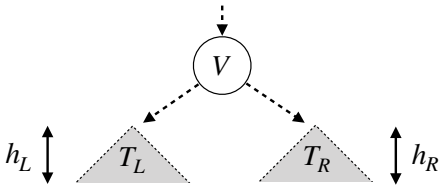
  - Each node is either red or black.
  - The root and leaves (NIL) are black.
  - If a node is red, then its children are black.

- *Count*: All paths from a given node to any of the leaves have the same number of black nodes, often referred to as the black-height.



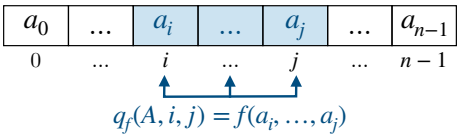
❒ **AVL tree** – An Adelson-Velsky and Landis (AVL) tree is a self-balancing tree with height $\mathcal{O}(\log(n))$. Its key property is that any of its nodes must have the heights h_L, h_R of their child subtrees T_L, T_R differ by at most 1:

$$\boxed{|h_L - h_R| \leqslant 1}$$

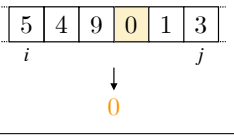
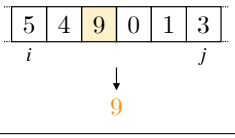
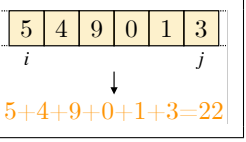


3.4.2 Efficient trees

❒ **Range query** – Given an array $A = [a_0, ..., a_{n-1}]$, a range query $q_f(A, i, j)$ is defined as being an operation f over elements $a_i, ..., a_j$.

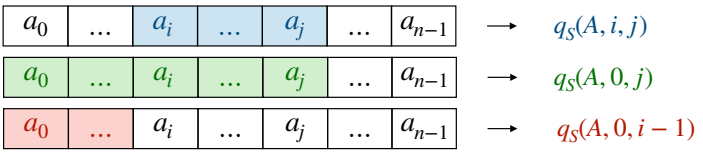


The following are the most common types of range queries:

	Minimum	Maximum	Sum
Operation	$\min(a_i, ..., a_j)$	$\max(a_i, ..., a_j)$	$a_i + ... + a_j$
Illustration			

In the case of a range sum query q_S, we note that for $0 \leqslant i \leqslant j < n$, we have:

$$\boxed{q_S(A, i, j) = q_S(A, 0, j) - q_S(A, 0, i-1)}$$

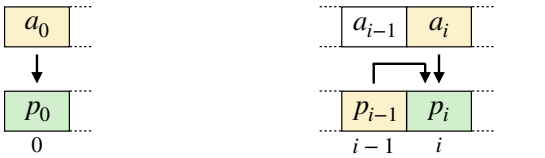


This means that if we can compute $q_S(A, 0, i)$ for all i, then we can handle any range sum query on array A.

❒ **Prefix sum array** – A prefix sum array $P = [p_0, ..., p_{n-1}]$ is an array that computes range sum queries on A in $\mathcal{O}(1)$ time.

Construction It is built in $\mathcal{O}(n)$ time as follows:

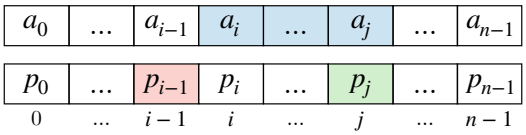
$$\boxed{p_0 = a_0} \quad \text{and} \quad \forall i \in [\![1, n-1]\!], \quad \boxed{p_i = p_{i-1} + a_i}$$



By convention, we set $p_{-1} \triangleq 0$.

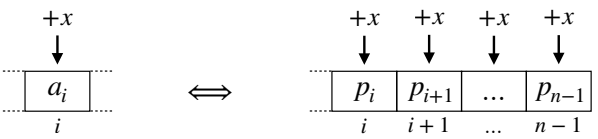
Access The computation of a range sum query takes $\mathcal{O}(1)$ time as follows:

$$\boxed{q_S(A, i, j) = p_j - p_{i-1}}$$

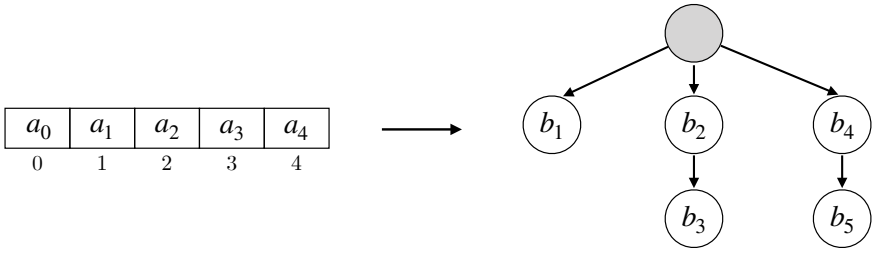


Update If we update $a_i \longleftarrow a_i + x$, it takes $\mathcal{O}(n)$ time to update P:

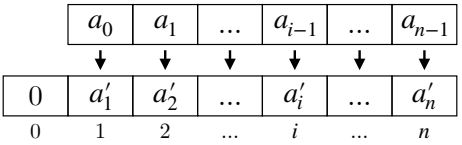
$$\forall k \in [\![i, n-1]\!], \quad \boxed{p_k \longleftarrow p_k + x}$$



❒ Binary indexed tree – A binary indexed tree B, also called Fenwick tree, is a data structure that efficiently handles range sum queries on array $A = [a_0, ..., a_{n-1}]$. Updates in A are reflected in B in $\mathcal{O}(\log(n))$ time.



Rescaling A prerequisite is to map the input array indices from $[\![0, n-1]\!]$ to $[\![1,n]\!]$. We rewrite A as $A' = [0, a'_1, ..., a'_n]$ with $a'_i = a_{i-1}$.



Intuition Elements of the binary indexed tree leverage the binary representations of their indices $i \in [\![1,n]\!]$. In order to do that, it relies on the concept of lowest set bit l. We note l_i the lowest set bit of i.

$$(i)_{10} \longleftrightarrow (\ldots \overset{2^{l_i}}{1} 0 \ldots 0)_2$$

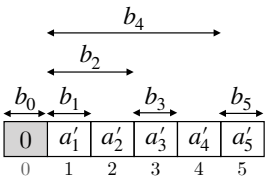
$$\begin{aligned}
(1)_{10} &\longleftrightarrow (0\,0\,1)_2 \\
(2)_{10} &\longleftrightarrow (0\,1\,0)_2 \\
(3)_{10} &\longleftrightarrow (0\,1\,1)_2 \\
(4)_{10} &\longleftrightarrow (1\,0\,0)_2 \\
(5)_{10} &\longleftrightarrow (1\,0\,1)_2
\end{aligned}$$

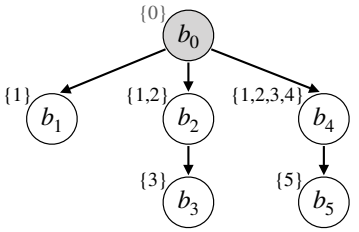
A given element b_i of the binary indexed tree B is said to be *responsible* for the range of indices $[\![i - 2^{l_i} + 1, i]\!]$ and we have:

$$\boxed{b_i = \sum_{j=i-2^{l_i}+1}^{i} a'_j}$$

The data structure can be represented either as an array or as a tree:

Array	**Tree**
Ranges of indices are indicated by horizontal lines	- Ranges of indices are next to nodes - Parent of b_i is $b_{i-2^{l_i}}$ - Tree depth is $\mathcal{O}(\log(n))$

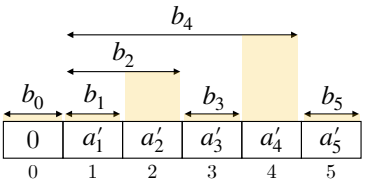




Construction The binary indexed tree is built iteratively in $\mathcal{O}(n)$ time:

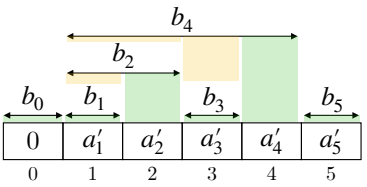
- *Initialization*: Initialize B with the values of A':

$$\forall i \in [\![0, n]\!], \quad \boxed{b_i \longleftarrow a'_i}$$



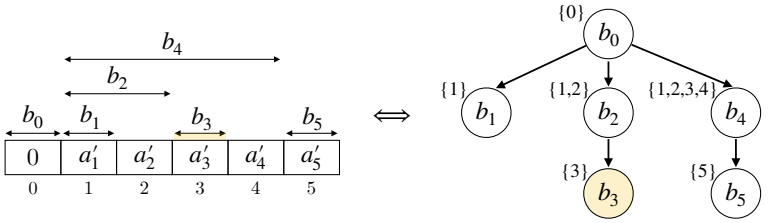
- *Compute step*: For each index $i \in [\![1,n]\!]$, the relevant partial sums are propagated through the overlapping ranges of responsible values:

$$\text{If } i + 2^{l_i} \leqslant n, \quad \boxed{b_{i+2^{l_i}} \longleftarrow b_{i+2^{l_i}} + b_i}$$



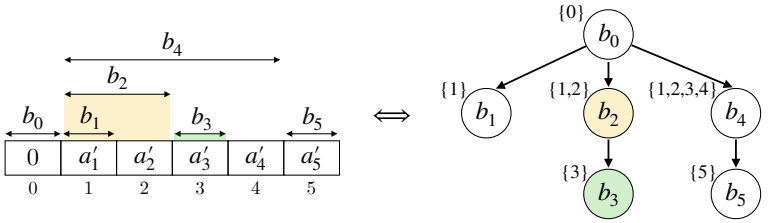
Access In order to compute $q_S(A', 0, i)$, we need to sum relevant elements of B in $\mathcal{O}(\log(n))$ time:

- *Initialization*: We start by initializing the sum S to b_i.



- *Compute step*: Starting from $j \longleftarrow i - 2^{l_i}$ and then by further decrements of 2^{l_j}, we update S until j is equal to 0:

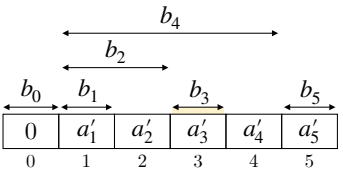
$$\boxed{S \longleftarrow S + b_j}$$



Update If we update $a'_i \longleftarrow a'_i + x$, it takes $\mathcal{O}(\log(n))$ time to update B:

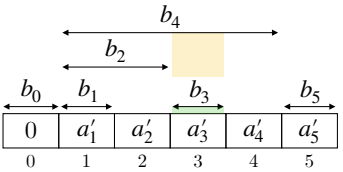
- *Initialization*: We start by updating b_i as follows:

$$\boxed{b_i \longleftarrow b_i + x}$$

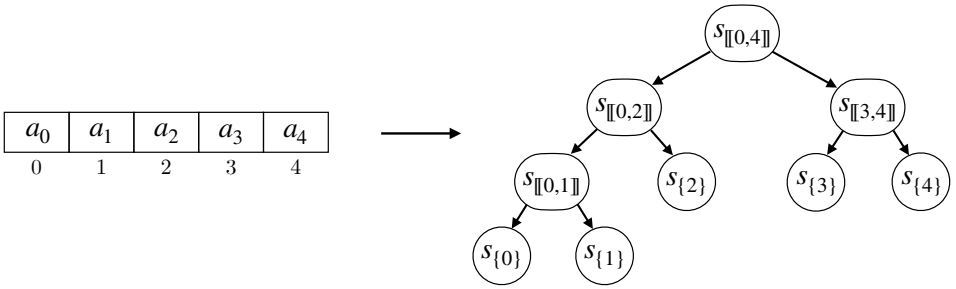


- *Compute step*: We continue the same update process with $j \longleftarrow i + 2^{l_i}$, and then by further increments of 2^{l_j} until reaching the maximum index of the array:

$$\boxed{b_j \longleftarrow b_j + x}$$

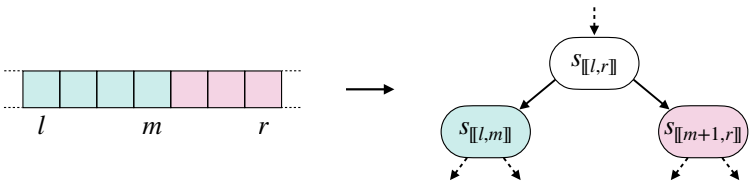


❒ **Segment tree** – A segment tree S is a data structure represented by a binary tree of nodes $s_{[\![i,j]\!]}$ that is designed to support the main types of range queries (minimum, maximum, sum) on an array $A = [a_0, ..., a_{n-1}]$. It has an update time complexity of $\mathcal{O}(\log(n))$.



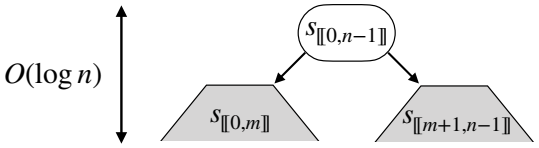
Intuition Each node of the segment tree is responsible for a range $[\![l, r]\!]$ of indices of the original array. In particular, each node $s_{[\![l,r]\!]}$ falls into one of the following categories:

- *Case* $l \neq r$: It has two children. By noting $m = \left\lfloor \dfrac{l+r}{2} \right\rfloor$:
  - Its left child is responsible for indices $[\![l, m]\!]$.
  - Its right child is responsible for indices $[\![m+1, r]\!]$.



- *Case $l = r$*: It has 0 child and it is responsible for index $\{l\}$.

The root node is responsible for indices $[\![0, n-1]\!]$. The segment tree has a height of $\mathcal{O}(\log(n))$.



For illustrative purposes, we are going to focus on range sum queries so that operations of the segment tree can be easily compared to those of the data structures we saw previously.

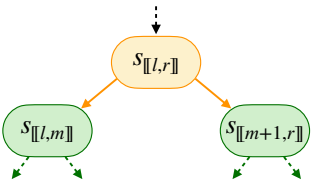
Construction The segment tree is built recursively in $\mathcal{O}(n)$ time, where the value of each node depends on where it is located:

- *Node has no children*: Its value is the corresponding value in the original array:

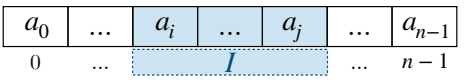
$$\boxed{s_{\{e\}} = a_e}$$

- *Node has two children*: Its value is the sum of the values of its child nodes:

$$\boxed{s_{[\![l,r]\!]} = s_{[\![l,m]\!]} + s_{[\![m+1,r]\!]}}$$



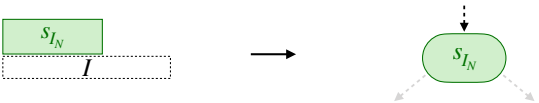
Access The operation $q_S(A, i, j)$ is done with a complexity of $\mathcal{O}(\log(n))$ time.



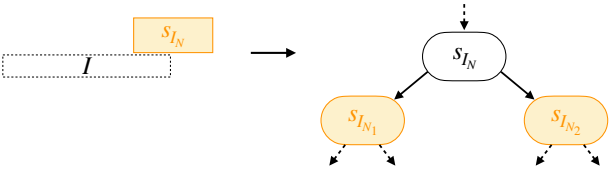
Given an interval $I = [\![i, j]\!]$, we start from the root node of the segment tree and make recursive calls.

At node s_{I_N}, there are 3 possible situations:

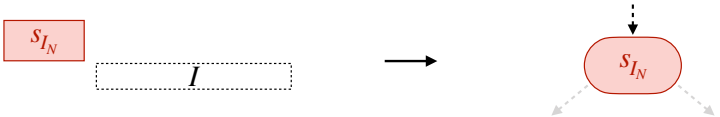
- *Case* $I_N \cap I \neq \varnothing$: There is an overlap.
  - *Sub-case* $I_N \subseteq I$: There is a total overlap, so the information of that node is added to the total sum.



  - *Sub-case* $I_N \not\subseteq I$: There is a partial overlap, so we are going to look at both children of the node.



- *Case* $I_N \cap I = \varnothing$: There is no overlap, the answer is not there. We don't look further.



Update If we update $a_i \longleftarrow a_i + x$, it takes $\mathcal{O}(\log(n))$ time to update S.

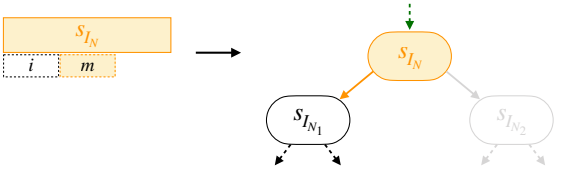
We start at the root node and make recursive calls. We check whether i is in the interval $I_N = [\![l, r]\!]$ associated with the node:

- *Case* $i \in I_N$: The node contains information related to a_i. We update the node by adding x:

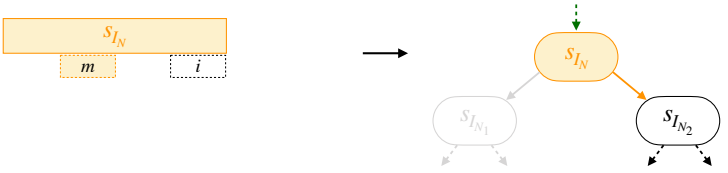
$$\boxed{s_{I_N} \longleftarrow s_{I_N} + x}$$

  By noting $m = \left\lfloor \frac{l+r}{2} \right\rfloor$:

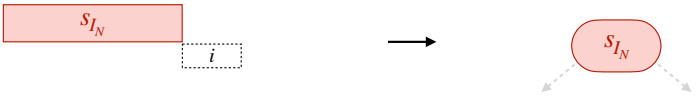
  - *Sub-case* $i \leqslant m$: i is also located in its left child.



  - *Sub-case* $i > m$: i is also located in its right child.



- *Case* $i \notin I_N$: The node does not contain information related to a_i.



SECTION 4

Sorting and search

In this last section, we will go through the most common sorting algorithms, along with useful search techniques that are used to solve many problems.

4.1 Sorting algorithms

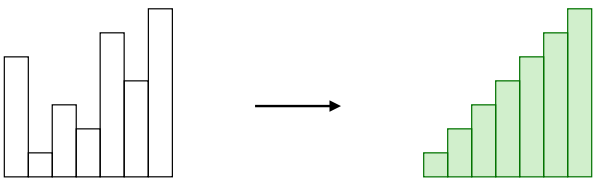
In this part, we will learn about common sorting algorithms such as bubble sort, insertion sort, selection sort, merge sort, heap sort and quick sort. We will also study sorting algorithms such as counting sort and radix sort that are particularly efficient in some special cases.

4.1.1 General concepts

In this part, arrays of n elements are visually represented as histograms. The height of each bar represents the value of the associated element in the array.

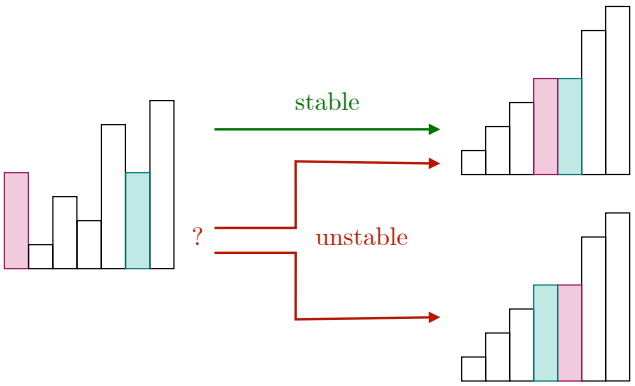
❒ **Sorting algorithm** – A sorting algorithm takes an unsorted array $A = [a_0, ..., a_{n-1}]$ as input and returns a sorted array $A' = [a'_0, ..., a'_{n-1}]$ as output. A' is a permutation of A such that:

$$\boxed{a'_0 \leqslant a'_1 \leqslant ... \leqslant a'_{n-1}}$$



A A'

❒ **Stability** – A sorting algorithm is said to be stable if the order of tied elements is *guaranteed* to remain the same after sorting the array.



Remark: Examples of stable sorting algorithms include merge sort and insertion sort.

The next sections will go through each sorting algorithm into more detail.

4.1.2 Basic sort

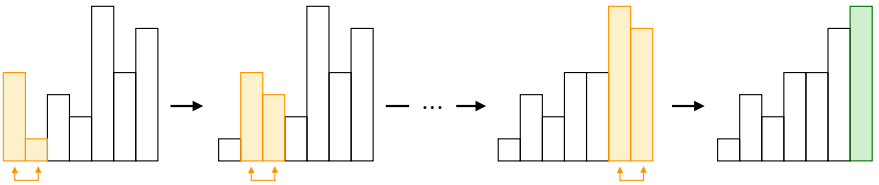
❒ **Bubble sort** – Bubble sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Compare consecutive elements and swap them if they are not in the correct order.

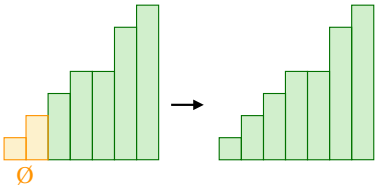
Algorithm

- *Compute step*: Starting from the beginning of the array, compare the element a_l at position i with the element a_r at position $i+1$.
  - *Case* $a_l \leqslant a_r$: They are already in the correct order. There is nothing to do.
  - *Case* $a_l > a_r$: They are not in the correct order. Swap them.

  Repeat this process until reaching the end of the array.



- *Repeat step*: Repeat the *compute step* until no swap can be done.

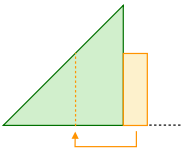


We note that:

- At the end of the k^{th} pass, the last k elements of the array are guaranteed to be in their final positions.
- The algorithm finishes in at most $n-1$ passes. In particular:
  - If the input array is already sorted, then the algorithm finishes in 1 pass.
  - If the input array is reverse sorted, then the algorithm finishes in $n-1$ passes.

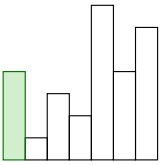
❒ **Insertion sort** – Insertion sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively inserting the next unsorted element at its correct position.



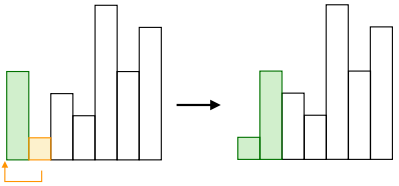
Algorithm

- *Initialization*: The first element of the unsorted array can be interpreted as a subarray of one element that is already sorted.

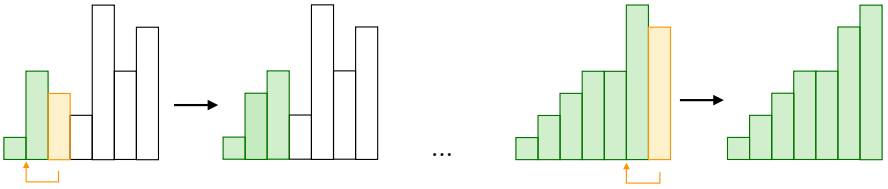


- *Compute step*: Starting from position $i = 1$, we want to insert element a_i into the current sorted subarray of size i. In order to do that, we compare a_i with its preceding element a_p:
  - As long as $a_p > a_i$, we iteratively swap a_p with a_i.
  - This process ends with either a_i verifying $a_p \leqslant a_i$ or a_i being at position 0 of the array.

  At the end of this step, the sorted subarray is of size $i + 1$.

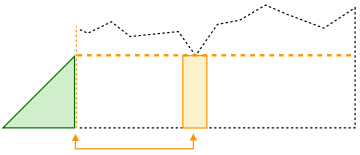


- *Repeat step*: Repeat the *compute step* until the sorted subarray reaches a size of n.



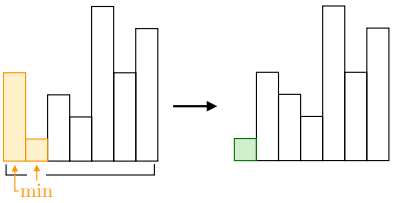
❒ **Selection sort** – Selection sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively inserting the minimum value among the remaining elements.



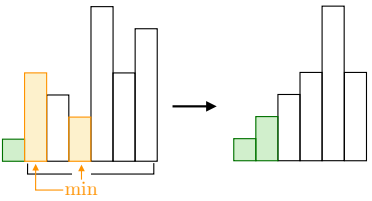
Algorithm

- *Initialization*:
  - Find the minimum value of the unsorted array.
  - Swap it with the element at the beginning of the array. The first element can now be interpreted as a sorted subarray with one element.

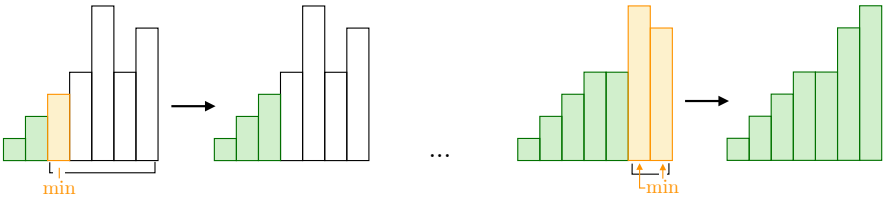


- *Compute step*: Starting from $i = 1$, we want to insert a new element into the sorted subarray of size i.
  - Find the minimum value $a_{\min}$ among the remaining elements at positions $i, ..., n-1$ of the array. By construction, $a_{\min}$ is greater or equal than all elements of the current sorted subarray.
  - Swap $a_{\min}$ with the element at position i of the array.

  At the end of this step, the sorted subarray is of size $i + 1$.



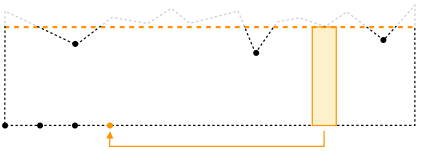
- *Repeat step*: Repeat the *compute step* until the sorted subarray reaches a size of n.



Remark: Insertion and selection sort are very similar in that they build the sorted array from scratch and add elements one at a time.

❒ **Cycle sort** – Cycle sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(1)$.

Intuition Determine the index of the final position of each element in the sorted array.

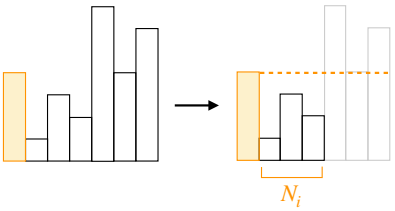


Algorithm

- *Compute step*: Starting from $i = 0$, we want to find the final position of element a_i along with those of the other elements impacted by this move.

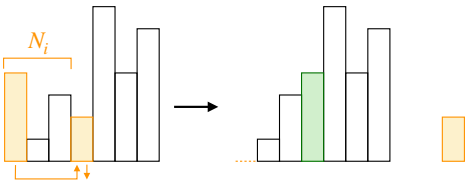
  To do so, we count the number of elements that are smaller than a_i:

  $$\boxed{N_i = \#\{k, a_k < a_i\}}$$

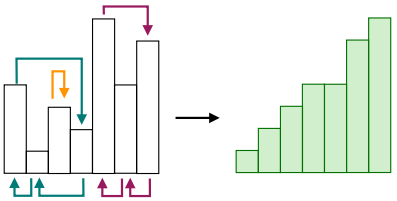
  The final position of a_i is at index N_i, since we know that there are N_i smaller elements than a_i in the final sorted array.

  - *Case* $N_i = i$: This is a self-cycle, meaning that the element is already at its correct position. There is nothing to do.
  - *Case* $N_i \neq i$: This is the start of a cycle of moves. Place a_i at position N_i (or to the right of any duplicates, if applicable) and keep the replaced value in a temporary variable.



Keep moving elements using this logic until getting back to position i.

- *Repeat step*: Repeat the *compute step* until reaching the end of the array. We can see cycles being formed from the way elements are moved.



Remark: This algorithm sorts the array with a minimum amount of rewrites since it only moves elements to their final positions.

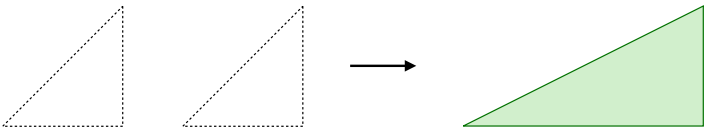
❒ **Basic sorting algorithms summary** – The table below summarizes the main basic sorting algorithms:

Type	Time			Space	Stability
	Best	Average	Worst		
Bubble sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Insertion sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes
Cycle sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	No

4.1.3 Efficient sort

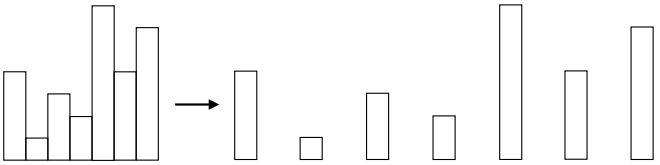
❒ **Merge sort** – Merge sort is a stable sorting algorithm that has a time complexity of $\mathcal{O}(n \log(n))$ and a space complexity of $\mathcal{O}(n)$.

Intuition Build a sorted array by merging two already-sorted arrays.

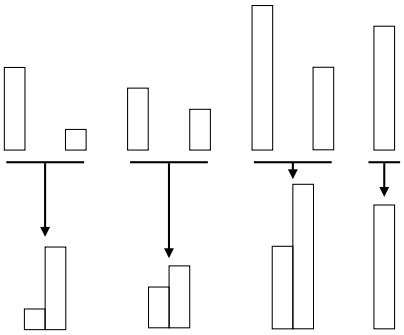


Algorithm

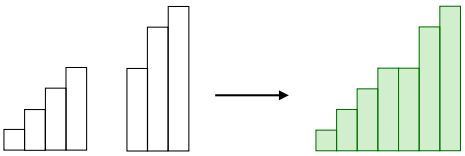
- *Divide step*: Divide the array into as many subarrays as there are elements. Each resulting subarray has only one element and can be considered as sorted.



- *Conquer step*: For each pair of sorted subarrays, build a sorted array by merging them using the two-pointer technique.



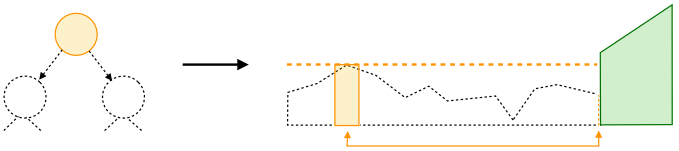
Repeat the process until all subarrays are merged into one final sorted array.



Remark: This algorithm is usually implemented recursively.

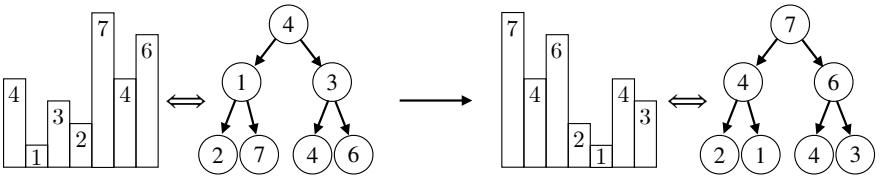
❒ **Heap sort** – Heap sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n \log(n))$ and a space complexity of $\mathcal{O}(1)$.

Intuition Incrementally build a sorted subarray by successively retrieving the maximum of remaining elements using a max-heap.



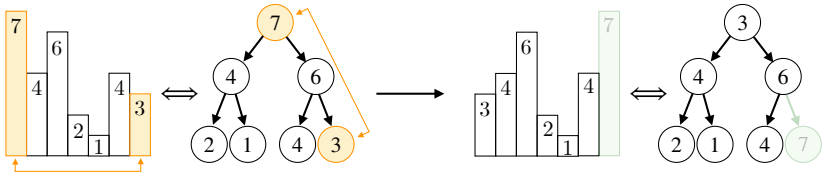
Algorithm

- *Initialization*: Build a max-heap from the unsorted array in $\mathcal{O}(n)$ time. This is done by recursively swapping each parent with their child of highest value.



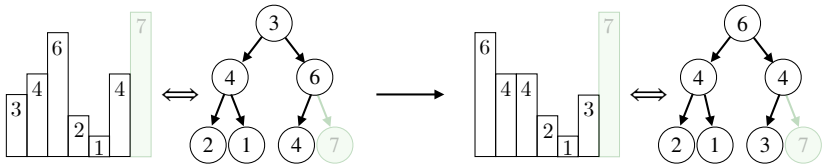
We can use the same array to represent the max-heap.

- *Compute step*: Starting from $i = 0$, incrementally build a sorted subarray of size $i + 1$ that is placed at the end of the array.
  - Pop an element from the max-heap and place it at position $n - i - 1$ of the array.

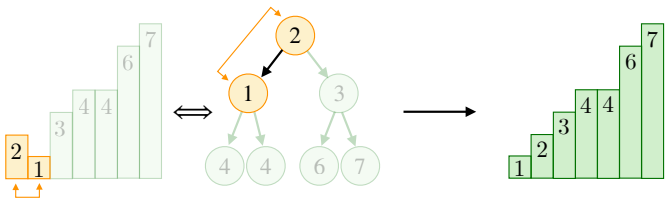


By construction, the popped element is greater or equal than the $n-i-1$ elements of the heap and smaller or equal than the i previously popped elements.

  - Heapify the resulting max-heap of size $n - i - 1$ in $\mathcal{O}(\log(n))$ time.



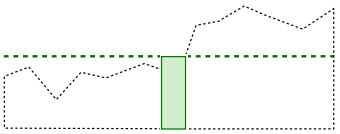
- *Repeat step*: Repeat the *compute step* until the subarray reaches a size of n.



❒ **Quick sort** – Quick sort is an unstable sorting algorithm that has a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(n)$.

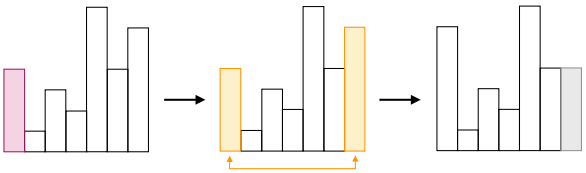
Intuition Recursively choose an element of the array to be the pivot, and put:

- Smaller elements to the left of the pivot.
- Bigger elements to the right of the pivot.



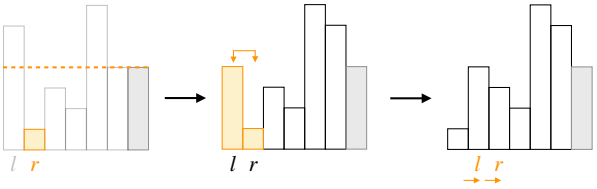
Algorithm

- *Compute step*:
  - *Pivot isolation step*: Choose an element of the array to be the pivot p and swap it with the last element of the array.

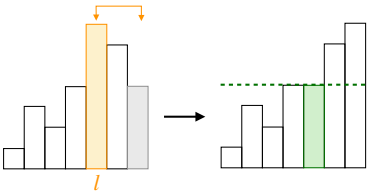


  - *Partition step*: Starting from indices $l = 0$ and $r = 0$, we make sure that elements a_r that are smaller or equal than the pivot p are sent to the left of the array:
    - *Case* $a_r \leqslant p$: Swap element a_r at index r with element a_l at index l. Increment l by 1.
    - *Case* $a_r > p$: There is nothing to do.

    This process is successively repeated for all indices r until the second-to-last position of the array.

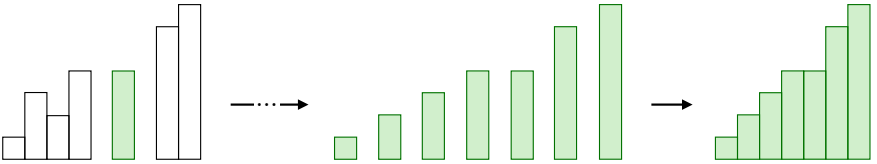


– *Pivot inclusion step*: The final position of the left pointer l represents the position in the array where its left elements are all array elements that are smaller than or equal to the pivot. We swap the pivot with the element from position l.



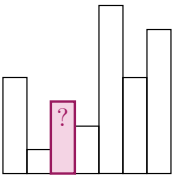
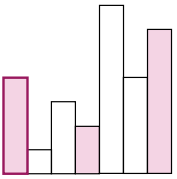
At the end of this step, the pivot is at its correct and final position.

- *Recursion step*: The *compute step* is run recursively on the resulting subarrays on the left and right of the pivot. They are then merged back together to form the final sorted array.



We note that the runtime of the algorithm is sensitive to the choice of the pivot and the patterns found in the input array. In general, a time complexity of $\mathcal{O}(n\log(n))$ and a space complexity of $\mathcal{O}(\log(n))$ can be obtained when the pivot is a good approximation of the median of the array.

The two most common methods to choose the pivot are:

Random choice	**Median of three**
Pivot is chosen at random.	Pivot is the median of the first, middle and last elements of the array.
	

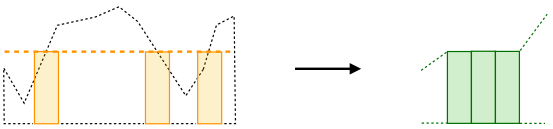
❒ **Efficient sorting algorithms summary** – The table below summarizes the main efficient sorting algorithms:

Type	Time			Space	Stability
	Best	Average	Worst		
Merge sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n)$	Yes
Heap sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(1)$	No
Quick sort	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	No

4.1.4 Special sort

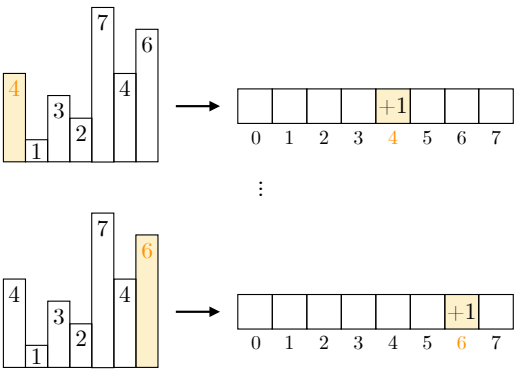
❒ **Counting sort** – Counting sort is a stable sorting algorithm that is efficient for integer arrays with values within a small range $[\![0,k]\!]$. It has a time complexity of $\mathcal{O}(n+k)$ and a space complexity of $\mathcal{O}(n+k)$.

Intuition Determine the final position of each element by counting the number of times its associated value appears in the array.



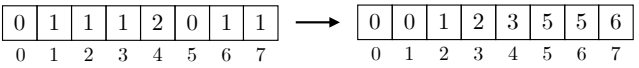
Algorithm

- *Number of occurrences*: This step takes $\mathcal{O}(n+k)$ time and $\mathcal{O}(k)$ space.
  - *Initialization*: Initialize an array C of length $k+1$.
  - *Count step*: Scan array A and increment the counter c_v of each encountered value $v \in \{0,...,k\}$ by 1.



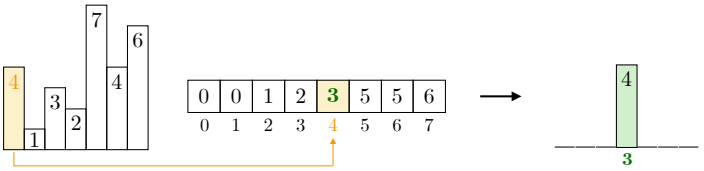
Each count c_v represents the number of times the value $v \in \{0, ..., k\}$ appears in array A.

– *Cumulative step*: Compute the cumulative sum of array $C = [c_0, .., c_k]$ and move each resulting element to the right. This operation is done in-place and takes $\mathcal{O}(k)$ time and $\mathcal{O}(1)$ space.

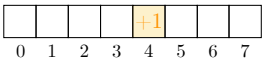


For each value v present in A, the associated element c_v in the resulting array C indicates its starting index i in the final sorted array A'.

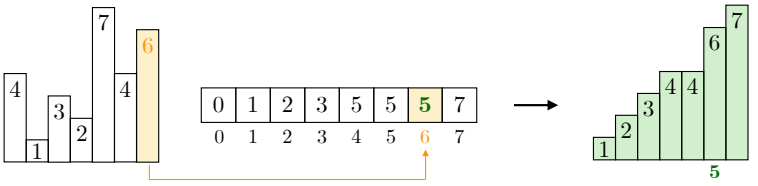
- *Construction of the sorted array*: This step takes $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.
  – *Initialization*: Initialize an array A' of length n that will contain the final sorted array.
  – *Main step*: For each value v of the unsorted array A:
    * Write v to index c_v of A'.



    * Increment c_v by 1 so that any later duplicates can be handled.

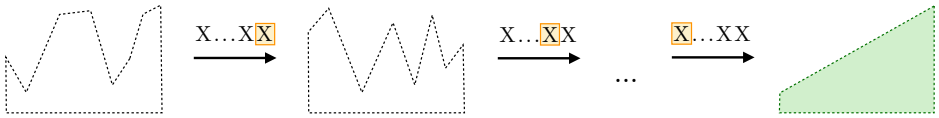


At the end of this process, we obtain the final sorted array A'.



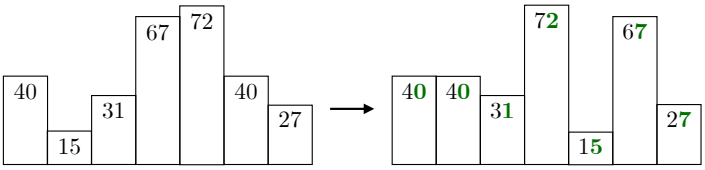
❒ **Radix sort** – Radix sort is a stable sorting algorithm that is well suited for integer arrays where elements are written with a limited number of digits d, each digit being in the range $[\![0, k]\!]$. This algorithm has a time complexity of $\mathcal{O}(d(n+k))$ and a space complexity of $\mathcal{O}(n+k)$.

Intuition Successively sort elements based on their digits using counting sort.

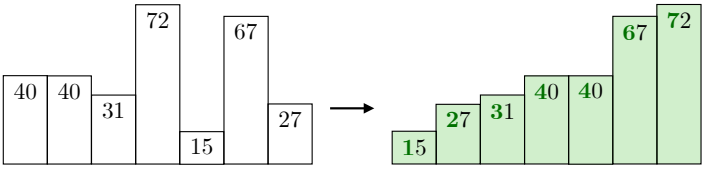


Algorithm

- *Compute step*: Perform counting sort based on the rightmost digit. This step takes $\mathcal{O}(n+k)$ time and $\mathcal{O}(n+k)$ space.



- *Repeat step*: Repeat the *compute step* on the remaining digits until reaching the leftmost digit.



At the end of this process, the array is sorted.

The trick of this algorithm lies in the stability property of counting sort: the relative ordering based on a given (weak) digit helps in breaking ties of later (stronger) digits.

❒ **Special sorting algorithms summary** – The table below summarizes the main special sorting algorithms:

Type	Time			Space	Stability
	Best	Average	Worst		
Counting sort	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$	$\mathcal{O}(n+k)$	Yes
Radix sort	$\mathcal{O}(d(n+k))$	$\mathcal{O}(d(n+k))$	$\mathcal{O}(d(n+k))$	$\mathcal{O}(n+k)$	Yes

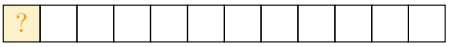
4.2 Search algorithms

In this part, we will focus on the most important search algorithms such as linear search and binary search. We will also study the main string pattern matching algorithms.

4.2.1 Basic search

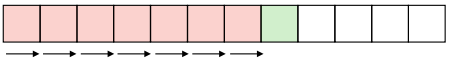
❒ **Linear search** – Linear search is a basic search method often used when the relative ordering of elements is not known, e.g. in unsorted arrays. It has a complexity of $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space:

- *Scan step*: Check whether the constraint of the problem is verified on the first element of the array.



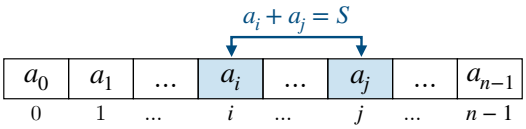
  Repeat the process on the remaining elements in a sequential way.

- *Final step*: Stop the search when the desired condition is satisfied.



❒ **Two-pointer technique** – The two-pointer technique provides an efficient way to scan arrays by leveraging the assumptions of the problem, e.g. the fact that an array is sorted.

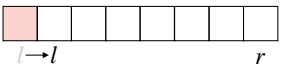
To illustrate this concept, suppose $A = [a_0, ..., a_{n-1}]$ a sorted array. Given S, the goal is to find two elements a_i, a_j such that $a_i + a_j = S$.



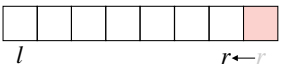
The two-pointer technique finds an answer in $\mathcal{O}(n)$ time:

- *Initialization*: Initialize pointers l and r that are set to be the first and last element of the array respectively.

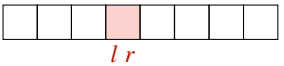
- *Compute step*: Compute $S_{\text{pointers}} \triangleq a_l + a_r$.
  - *Case* $S_{\text{pointers}} < S$: The computed sum needs to increase, so we move the l pointer to the right.



  - *Case* $S_{\text{pointers}} > S$: The computed sum needs to decrease, so we move the r pointer to the left.

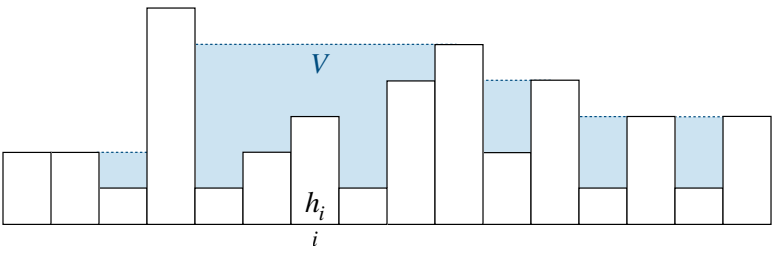


- *Final step*: Repeat the *compute step* until one of the following situations happens:
  - *Case* $l = r$: There is no solution to the problem.



  - *Case* $S_{pointers} = S$: A solution is found with $i = l$ and $j = r$.

❒ **Trapping rain water** – A classic problem that uses the two-pointer technique is the trapping rain water problem. Given a histogram $H = [h_0, ..., h_{n-1}]$ that represents the height $h_i \geqslant 0$ of each building i, the goal is to find the total volume V of water that can be trapped between the buildings.

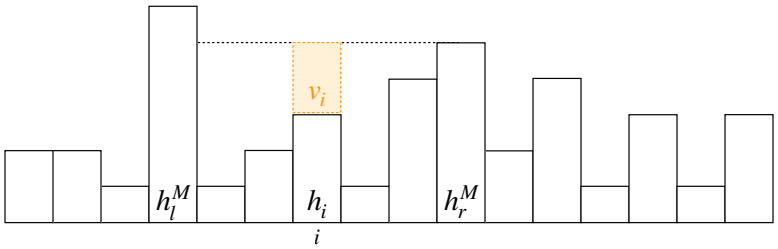


Tricks

- The maximum volume of water v_i trapped at index i is determined by the relative height between h_i and the tallest buildings on the left h_l^M and on the right h_r^M:

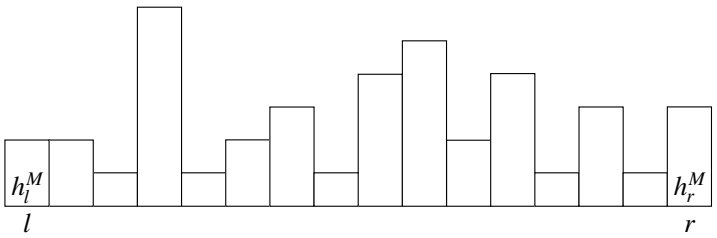
$$\boxed{v_i = \min(h_l^M, h_r^M) - h_i}$$

- Use the two-pointer technique to efficiently update h_l^M and h_r^M at each step.



The solution is found in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space:

- *Initialization*:
  - *Pointers*: Set the left and right pointers to $l = 0$ and $r = n - 1$.
  - *Maximum heights*: Set the maximum height of the left buildings h_l^M and the right buildings h_r^M to h_l and h_r.
  - *Total amount of water*: Set the total amount of water V to 0.



- *Compute step*: Update the values of the maximum heights of left and right buildings with the current values:

$$\boxed{h_l^M = \max(h_l^M, h_l)} \quad \text{and} \quad \boxed{h_r^M = \max(h_r^M, h_r)}$$

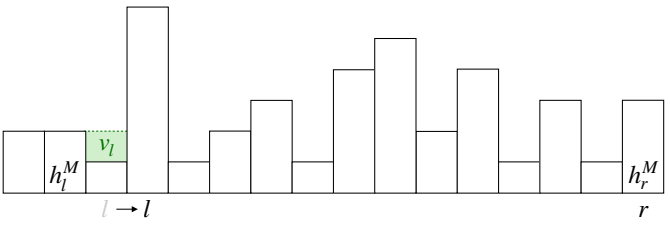
  - If $h_l^M < h_r^M$, then the left side is the limiting side, so:

* Add the available volume $v_l = h_l^M - h_l$ at index l to the total amount of water V:

$$\boxed{V \longleftarrow V + v_l}$$

* Move the left pointer to the right:

$$\boxed{l \longleftarrow l + 1}$$



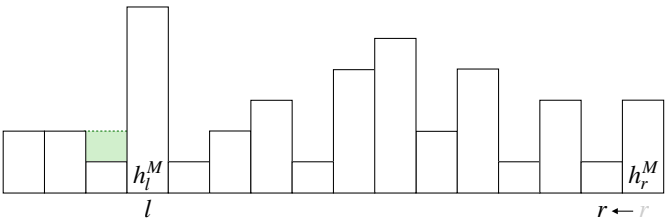
– Conversely, if $h_l^M \geqslant h_r^M$, then:

* Add the available volume $v_r = h_r^M - h_r$ at index r to the total amount of water V:

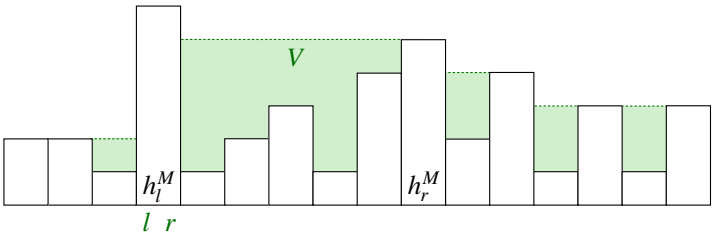
$$\boxed{V \longleftarrow V + v_r}$$

* Move the right pointer to the left:

$$\boxed{r \longleftarrow r - 1}$$



- *Final step*: The algorithm finishes when the two pointers meet, in which case the amount of water is given by V.

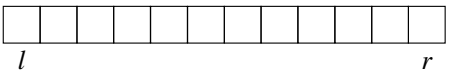


4.2.2 Binary search

❒ **Algorithm** – Binary search aims at finding a given target value t in a sorted array $A = [a_0, ..., a_{n-1}]$.

This algorithm has a complexity of $\mathcal{O}(\log(n))$ time and $\mathcal{O}(1)$ space:

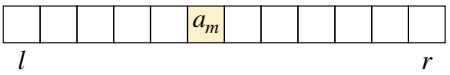
- *Initialization*: Choose the lower bound l and the upper bound r corresponding to the desired search space. Here, $l = 0$ and $r = n - 1$.



- *Compute step*: Compute the middle m of the two bounds:

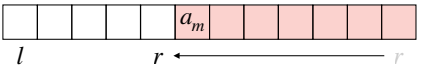
$$\boxed{m \triangleq l + \left\lfloor \frac{r-l}{2} \right\rfloor}$$

  The above formula is written in a way that avoids integer overflow in case l and r are too large.

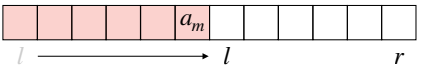


  We have the following cases:

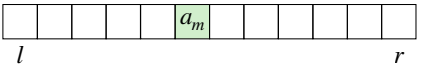
  – *Case* $t < a_m$: Shrink the search space to the left side by updating r to $r = m - 1$.



  – *Case* $t > a_m$: Shrink the search space to the right side by updating l to $l = m + 1$.



  – *Case* $t = a_m$: We found a solution.



- *Final step*: Repeat the *compute step* until one of the following situations happens:

- *Case* $t = a_m$: A solution is found.
- *Case* $r < l$: There is no solution since the pointers intersect after fully covering the search space.

❒ **Median of sorted arrays** – Suppose we want to compute the median of two sorted arrays A, B of respective lengths n_1, n_2, with $n_1 \leqslant n_2$.

a_0	a_1	...	a_{n_1-1}

b_0	b_1	...	b_{n_2-1}

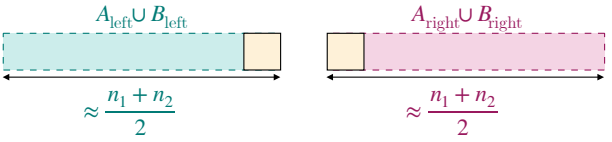
A naive approach would combine the two sorted arrays in $\mathcal{O}(n_1+n_2)$ time, and then select the median of the resulting array. However, a $\mathcal{O}(\log(n_1))$ approach exists that uses binary search in a clever way.

Tricks

- If we categorize all elements of both arrays into:
  - A left partition that has the $\approx \frac{n_1+n_2}{2}$ smallest elements.
  - A right partition that has the $\approx \frac{n_1+n_2}{2}$ biggest elements.

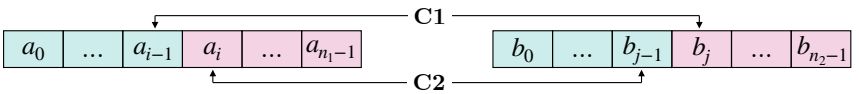
  Then we can deduce the median by looking at the maximum of the left partition and the minimum of the right partition.

- If we know how many elements from A are in the left partition, then we can deduce how many elements from B are in there too.



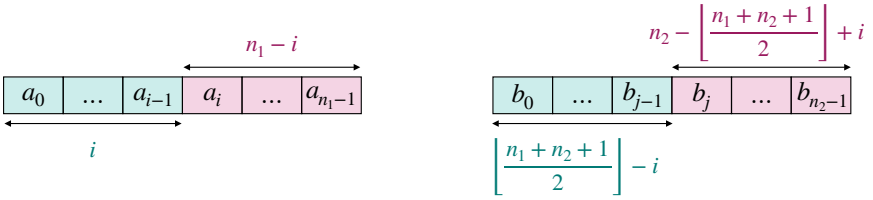
- By noting $A_{\text{left}}, A_{\text{right}}, B_{\text{left}}, B_{\text{right}}$ the left and right partitions from A and B, we know the partitioning is done correctly when the following conditions are satisfied:

(**C1**) $\max(A_{\text{left}}) \leqslant \min(B_{\text{right}})$ and (**C2**) $\max(B_{\text{left}}) \leqslant \min(A_{\text{right}})$



The binary search solution is based on the position of the partitioning within the array A. The solution is found in $\mathcal{O}(\log(n_1))$ time:

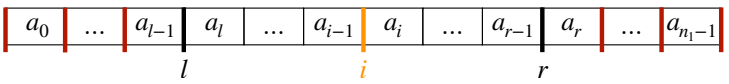
- *Initialization*: We note $i \in [\![0, n_1]\!]$ and $j \in [\![0, n_2]\!]$ the lengths of the left partitions within array A and B respectively. We fix the total size of the left partitions to be $\left\lfloor \frac{n_1+n_2+1}{2} \right\rfloor$.



  For a given i, the following formula determines j:

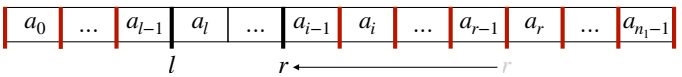
$$\boxed{j = \left\lfloor \frac{n_1 + n_2 + 1}{2} \right\rfloor - i}$$

- *Binary search*: We perform a binary search on i between 0 and n_1.

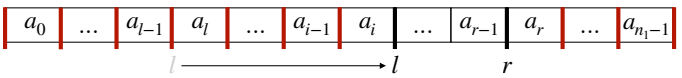


  For each candidate, we have the following cases:

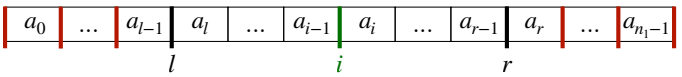
  - *(C1) is not satisfied*: We restrict the search space to the left side of i.



  - *(C2) is not satisfied*: We restrict the search space to the right side of i.



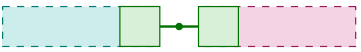
  - *Both (C1) and (C2) are satisfied*: The partitioning is done correctly.



- *Final result*: Once the correct partitioning is found, we deduce the value of the median. We note:

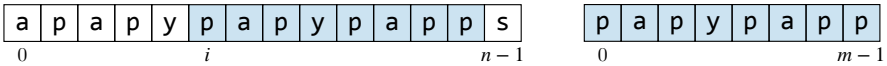
$$\boxed{\max_{\text{left}} \triangleq \max(A_{\text{left}}, B_{\text{left}})} \quad \text{and} \quad \boxed{\min_{\text{right}} \triangleq \min(A_{\text{right}}, B_{\text{right}})}$$

We have:

Case	Result	Illustration
$n_1 + n_2$ even	$\text{median} = \frac{\max_{\text{left}} + \min_{\text{right}}}{2}$	
$n_1 + n_2$ odd	$\text{median} = \max_{\text{left}}$	

4.2.3 Substring search

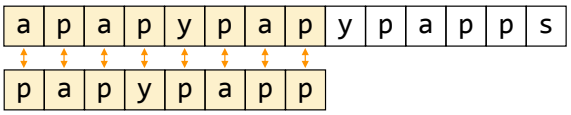
❒ **String pattern matching** – Given a string $s = s_0...s_{n-1}$ of length n and a pattern $p = p_0...p_{m-1}$ of length m, the string matching problem is about finding whether pattern p appears in string s.



We note $s_{i:i+m-1}$ the substring of s of length m that starts from s_i and finishes at s_{i+m-1}.

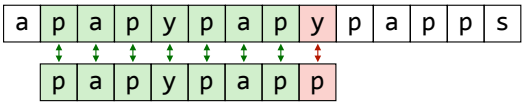
❒ **Brute-force approach** – The brute-force approach consists of checking whether pattern p matches any possible substring of s of size m in $\mathcal{O}(n \times m)$ time and $\mathcal{O}(1)$ space.

Starting from $i = 0$, compare the substring that starts from index i with pattern p.

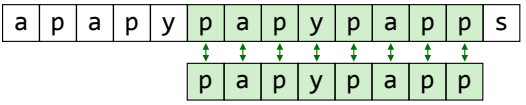


We have the following possible situations:

- *One of the characters is not matching*: The pattern does not match the associated substring.

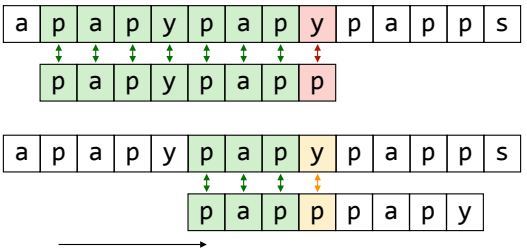


- *All characters are matching*: The pattern is found in the string.



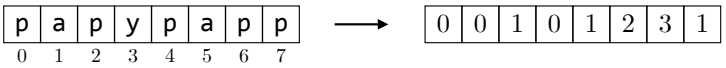
❒ Knuth-Morris-Pratt algorithm – The Knuth-Morris-Pratt (KMP) algorithm is a string pattern matching algorithm that has a complexity of $\mathcal{O}(n+m)$ time and $\mathcal{O}(m)$ space.

Intuition Avoid starting the pattern search from scratch after a failed match.

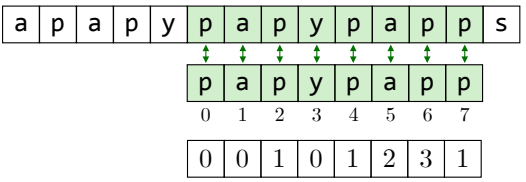


We do so by checking whether the pattern matched so far has a prefix that is also a suffix:

- *Step 1*: Build the prefix-suffix array.

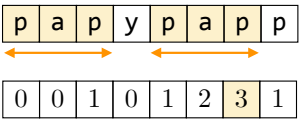


- *Step 2*: Find the pattern in the original string using the prefix-suffix array.

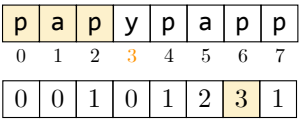


Prefix-suffix array construction We want to build an array $Q = [q_0, ..., q_{m-1}]$ where each value q_i has two interpretations:

- It is the length of the largest prefix that is also a suffix in the substring $p_0...p_i$.

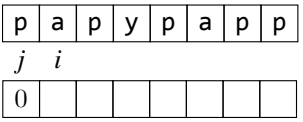


- It is also the index in the pattern that is right after the prefix.



This part has a complexity of $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ space:

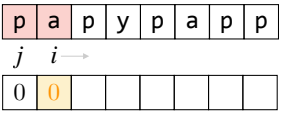
- *Initialization*:
  - By convention, we set q_{-1} to 0.
  - The element q_0 is set to 0 since the substring p_0 has a single character.
  - The pointers i and j find the largest prefix that is also a suffix to deduce the value of q_i. They are initially set to $i = 1$ and $j = 0$.



- *Compute step*: Determine the length of the largest prefix that is also a suffix for substring $p_0...p_i$:
  - *Case $p_i \neq p_j$ and $j = 0$*: The substring does not have a prefix that is also a suffix, so we set q_i to 0:

$$\boxed{q_i \longleftarrow 0}$$

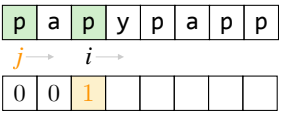
We increment i by 1.



  - *Case $p_i = p_j$*: The last letter p_i of the substring matches with p_j. As a result, the length of the prefix is the length of $p_0...p_j$:

$$\boxed{q_i \longleftarrow j + 1}$$

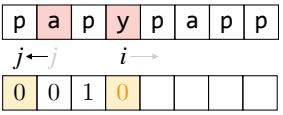
We increment both i and j by 1 respectively.



– *Case* $p_i \neq p_j$ *and* $j > 0$: The substring may have a smaller prefix that is also a suffix:

$$j \longleftarrow q_{j-1}$$

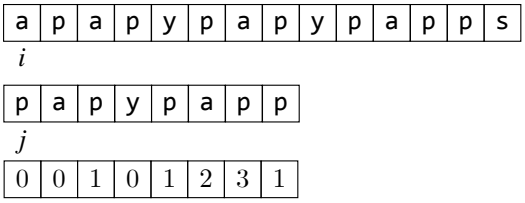
We repeat this process until we fall in one of the two previous cases.



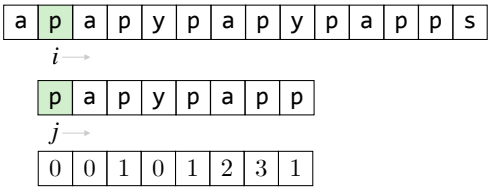
Linear-time string pattern search We want to efficiently search pattern p in string s using the newly-constructed prefix-suffix array Q.

This part has a complexity of $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space:

- *Initialization*: The pointers i and j check whether there is a match between each character of the string and the pattern respectively. They are initially set to 0.



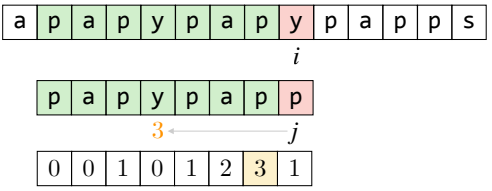
- *Compute step*: We check whether characters s_i and p_j match.
  - *Case* $s_i = p_j$: The characters match. We increment i and j by 1.



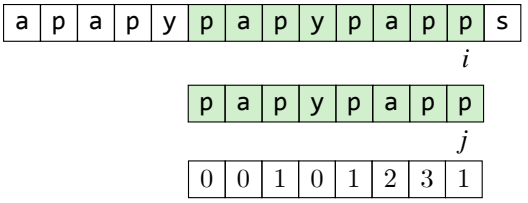
  - *Case* $s_i \neq p_j$: The characters do not match. We determine where we should start our search again by using the information contained in the prefix-suffix array.

  - $q_{j-1} = 0$: We need to restart the pattern match from the beginning since there is no prefix/suffix to leverage.
  - $q_{j-1} > 0$: We can continue our search after the next largest matched prefix of the pattern.

  We set j to q_{j-1}.



- *Final step*: The algorithm stops when one of the following situations happens:
  - *Case* $j = m$: The pattern has been matched.
  - *Case* $i = n$ *and* $j < m$: The whole string has been searched without finding a match.

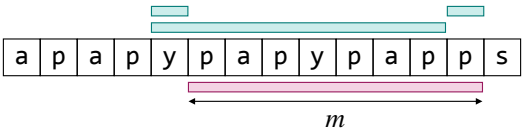


❒ Rabin-Karp algorithm – The Rabin-Karp algorithm is a string pattern matching algorithm that uses a hashing trick. It has an expected time complexity of $\mathcal{O}(n+m)$, but depending on the quality of the hash function, the complexity may increase up to $\mathcal{O}(n \times m)$. The space complexity is $\mathcal{O}(1)$.

Intuition Compute the hash value of the pattern along with those of the substrings of the same length. If the hash values are equal, confirm whether the underlying strings indeed match.

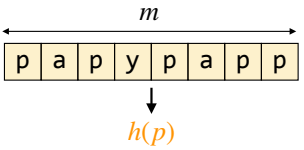
The trick is to choose a hash function h that can deduce $h(s_{i+1:i+m})$ from the hash value of the previous substring $h(s_{i:i+m-1})$ in $\mathcal{O}(1)$ time via a known function f:

$$\boxed{h(s_{i+1:i+m}) = f(h(s_{i:i+m-1}), h(s_i), h(s_{i+m}))}$$

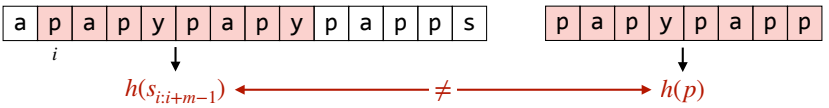


Algorithm

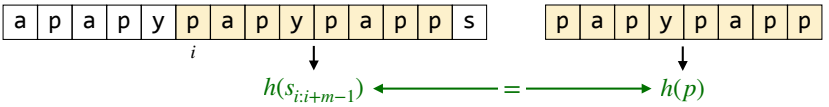
- *Initialization*: Given a hash function h, we compute the hash value $h(p)$ of the pattern p.



- *Compute step*: Starting from index $i = 0$, we use the hashing trick to compute the hash value of substring $s_{i:i+m-1}$ that starts from i and that has the same length as p in $\mathcal{O}(1)$ time.
  - *Case* $h(p) \neq h(s_{i:i+m-1})$: p and $s_{i:i+m-1}$ definitely do not match.



  - *Case* $h(p) = h(s_{i:i+m-1})$: There may be a match between p and $s_{i:i+m-1}$, so we compare p and $s_{i:i+m-1}$.
    - *Sub-case* $p = s_{i:i+m-1}$: The pattern has been matched.
    - *Sub-case* $p \neq s_{i:i+m-1}$: There is a collision between the hash values of p and $s_{i:i+m-1}$. Since there is no match, we continue the search.



Notations

This page summarizes the meaning behind the conventions taken in this book.

Coloring

- ❒ **Blue**: definitions, notations
- ❒ **Red**: theorems, properties, algorithms, techniques, summaries
- ❒ **Green**: operations, applications, classic problems
- *Purple*: remarks

Symbols

Notation	Meaning	Illustration
$\triangleq$	by definition, equal to	$3! \triangleq 3 \times 2 \times 1$
$\longleftarrow$	is assigned the value of	$i \longleftarrow 1$
$\approx$	approximatively equal to	$e \approx 2.72$
$\subseteq$	is a subset of	$\{2\} \subseteq \{2,3\}$
$\varnothing$	empty set	$\{2\} \cap \{3\} = \varnothing$
$\#S$ or $\lvert S \rvert$	number of elements in S	$\#\{4,6,7\} = 3$
$\Longrightarrow$	implies that	$x > 0 \Longrightarrow x^2 > 0$
$\Longleftrightarrow$	equivalent to	$x = 0 \Longleftrightarrow x^2 = 0$
$\widehat{a}$	estimate of a	$\widehat{\mu} = \frac{x_1+\ldots+x_n}{n}$
$\lfloor a \rfloor$	floor of a	$\lfloor 2.3 \rfloor = 2$
$\lceil a \rceil$	ceiling of a	$\lceil 2.3 \rceil = 3$
$\lvert a \rvert$	absolute value of a	$\lvert -1 \rvert = 1$
$[\![a,b]\!]$	integers between a and b included	$[\![1,3]\!] = \{1,2,3\}$
$\mathbb{N}$	set of nonnegative integers	$\{0,1,2,...\}$
$\mathbb{N}^*$	set of strictly positive integers	$\{1,2,...\}$

Indexing

- Indices of arrays are chosen to start from 0 to align with the convention taken by most programming languages.
- The numbering of elements in other data structures, such as stacks, queues, linked lists, hash sets and hash tables, start from 1.

References

Adelson-Velsky G., Landis E. (1962) *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences, Vol. 146, pages 263-266

Bayer R. (1972) *Symmetric binary B-Trees: Data structure and maintenance algorithms*, Acta Informatica, Vol. 1, pages 290-306

Bitner J. R., Reingold E. M. (1975) *Backtracking Programming Techniques*, Communications of the ACM, Vol. 18, No. 11, pages 651-656

Bloom B. H. (1970) *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, Vol. 13, No. 7

Booth A. D., Colin A. J. T. (1960) *On the Efficiency of a New Method of Dictionary Construction*, Information and Control, Vol. 3, pages 327-334

Cayley A. (1889) *A theorem on trees*, Quarterly Journal of Pure and Applied Mathematics, Vol. 23, pages 376-378

Cormode G., Muthukrishnan S. (2003) *An Improved Data Stream Summary: The Count-Min Sketch and its Applications*

Dijkstra E. W. (1959) *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik, pages 269-271

Fenwick P. M. (1994) *A new data structure for cumulative frequency tables*, Software: Practice and Experience, Vol. 24, No. 3, pages 327-336

Floyd R. W. (1962) *Algorithm 97: Shortest path*, Communications of the ACM, Vol. 5, No. 6, page 345

Floyd R.W. (1967) *Nondeterministic Algorithms*, Journal of the ACM, Vol. 14, No. 4, pages 636-644

Fredkin E. (1960) *Trie Memory*, Communications of the ACM, Vol. 3, No. 9, pages 490-499

Goldstine H. H. (1948) Planning and Coding of Problems for an Electronic Computer Instrument, Part 2, Vol. 2

Held M., Karp R. M. (1962) *A Dynamic Programming Approach to Sequencing Problems*, Journal of the Society for Industrial and Applied Mathematics, Vol. 10, No.1, pages 196-210

Hibbard T. N. (1962) *Some Combinatorial Properties of Certain Trees With Ap-*

plications to Searching and Sorting, Journal of the ACM, Vol. 9, No. 1, pages 13-28

Hoare C. A. R. (1961) *Algorithm 64: Quicksort*, Communications of the ACM, Vol. 4, No. 7, page 321

Karp R., Rabin M. (1977) *Efficient Randomized Pattern-Matching Algorithms*, IBM Journal Research and Development, Vol. 31, No. 2, pages 249-260

Knuth D., Morris J. H., Pratt V. (1977) *Fast pattern matching in strings*, SIAM Journal on Computing, Vol. 6, No. 2, pages 323-350

Knuth D. (1998) *Sorting and searching, The Art of Computer Programming, Volume 3*

Kruskal J. B. (1956) *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society, Vol. 7, No. 1, pages 48-50

Merigoux D. (2013) *Pense-bête : analyse*

Prim R. C. (1957) *Shortest Connection Networks And Some Generalizations*, The Bell System Technical Journal, pages 1389-1401

Sharir M. (1981) *A strong-connectivity algorithm and its applications to data flow analysis*, Computers and Mathematics with Applications, Vol. 7, No. 1, pages 67-72

Tarjan R. E. (1972) *Depth-first search and linear graph algorithms*, SIAM Journal on Computing, Vol. 1 No. 2, pages 146-160

Tarjan R. E. (1983) *Data Structures and Network Algorithms*

Williams J. W. J. (1964) *Algorithm 232: Heapsort*, Communications of the ACM, Vol. 7, No. 6, pages 347-348

Windley P. F. (1960) *Trees, Forests and Rearranging*, The Computer Journal, Vol. 3, No. 2, pages 84-88

Index

P

Q

R

S

T

V

About the authors

Afshine and **Shervine** are twin brothers of Persian origin. They were born and raised in Paris, France, came to the US for grad school and are currently working in the tech industry in the San Francisco Bay Area. They have a passion for creating easy-to-digest material on computer science-related topics.

You can find a more detailed individual bio for each of them below.

Afshine Amidi is currently working on solving natural language processing problems at Google. He also teaches the Data Science Tools class to graduate students at MIT as well as a deep learning workshop at Stanford. Previously, he worked in applied machine learning for recommender systems at Uber Eats where he focused on building ranking models that improve the quality of the overall search results by taking into account several objective functions. Also, Afshine published a few papers at the intersection of deep learning and computational biology. He holds a Bachelor's and a Master's Degree from École Centrale Paris and a Master's Degree from MIT.

You can learn more about Afshine at `www.mit.edu/~amidi`

Shervine Amidi is currently working on problems at the intersection of ranking and natural language processing at Google. He also teaches a deep learning workshop at Stanford. Previously, he worked in applied machine learning for recommender systems at Uber Eats where he focused on representation learning to better surface dish recommendations. Also, Shervine published a few papers at the intersection of deep learning and computational biology. He holds a Bachelor's and a Master's Degree from École Centrale Paris and a Master's Degree from Stanford University.

You can learn more about Shervine at `www.stanford.edu/~shervine`