Course: Basic mathematics and algorithmic

Ali ZAINOUL

for Keyce Academy
December 12, 2022

Table of contents

Table of contents

Introduction

Sets

- $\mathbb{N} := \{1, 2, 3, ...\}$ is the set of **natural numbers**. Sometimes we include zero. In the original definition, we do not include zero.
- $\mathbb{Z} := -\mathbb{N} \cup \{0\} \cup \mathbb{N} = \{-..., -3, -2, -1, 0, 1, 2, 3, ...\}$ is the set of **integers**.
- $\mathbb{D} := \{\frac{z}{10^n}$, such that: $z \in \mathbb{Z}, n \in \mathbb{N}\}$ is the set of **decimal numbers**.
- $\mathbb{Q} := \{\frac{a}{b}$, such that: $a, b \in \mathbb{Z}, b \neq 0\}$ is the set of **rational numbers**.
- $\mathbb{R} :=]-\infty, +\infty[$ is the set of **real numbers**.
- $\mathbb{C} := \{a + bi$, such that: $a, b \in \mathbb{R}$, and: $i^2 = -1\}$ is the set of **complex numbers**.

Remark: we naturally have the inclusion sequence:

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{D} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}.$$

Basic mathematics, reminder

- $\in$: means **in**, e.g.: $x, y \in \mathbb{R}$ means "take x and y in the set of real numbers".
- $\notin$: means **not in**, e.g.: $\pi \notin \mathbb{Q}$ means "the constant π^* is not in the rational set, a.k.a.: π is irrational".
- $\forall$: means **for all**, e.g.: $\forall x \in \mathbb{R}$ "means for all x in the set of real numbers".
- $\exists$: means **it exists**, e.g.: $\exists p \in [0, 1]$ "means it exists some particular number named p in the closed set $[0, 1]$".
- $\exists!$ means **it exists unique**, e.g.: $\exists! n \in \mathbb{N}$ s.t. $n - 1 = 0$ "it exists a unique number named n such that n - 1 equals to zero".

* The constant pi is the ratio between the circumference of a circle and it's diameter.

Basic mathematics, reminder

- $\subset$: means **proper subset**, e.g.: $\mathbb{Q} \subset \mathbb{R}$ means "the set of rational numbers is a subset of the set of real numbers".
- $\subseteq$: means **proper subset or equal**, e.g.: $\mathbb{A} \subseteq \mathbb{B}$ means "the set $\mathbb{A}$ is either a proper subset of $\mathbb{B}$, OR $\mathbb{A}$ is the same set as $\mathbb{B}$".
- $\cup$: means **union**, e.g.: $\mathbb{R} :=]-\infty, +\infty[=]-\infty, 0] \cup [0, +\infty[$ means "that the set of real numbers may be splitted into two half-lines; negative numbers and positive ones".
- $\cap$: means **intersection**, e.g.: $\mathbb{Q} \cap \mathbb{R} = \mathbb{Q}$ means "the intersection of the set of rational numbers with the set of real numbers is the set of rational numbers".

Remark: List of all mathematical symbols and their representation in Latex mode, the same language used to write this document. ;)

Algorithmic definition and historical events

The word algorithm comes from the latin form of the word ”algorismus” named after the arab mathematician ”AL KHAWARIZMI”. He formulates then the first definition of an algorithm:

Definition

An algorithm is a sequence of operations aimed at solving a problem in finite time.

It was in the year 813 that the beginnings of algorithmics were developed, but the origin of the very first algorithm dates back to 300 BC, it is the work of Euclid *via* his algorthime of Euclid which calculates the GCD of two numbers. Link github: Euclide.cpp

Notion of a computer program

Definition

A computer program is a set of instructions and operations intended to be executed by a computer.
A source program is code written by a computer scientist in a programming language.

General structure of an algorithm

An algorithm is usually broken down into three pieces:

- Header / begin: this part is used to give a name to the algorithm, it is often preceded by the keyword: Algorithm <name>.
- Declarative part / declarative part, input processing: in this part we take care of declaring the different inputs / variables / constants that the program needs..
- Body of the algorithm / body and end: this part contains all the instructions of the algorithm, often delimited by the keywords **Begin** and **End**.

Variables and constants (Types of variables)

Category	Types	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	`byte`	8	-128	127	From +127 to -128	`byte b = 65;`
	`char`	16	0	2^{16}-1	All Unicode characters[1]	`char c = 'A'; char c = 65;`
	`short`	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	`short s = 65;`
	`int`	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	`int i = 65;`
	`long`	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	`long l = 65L;`
Floating-point	`float`	32	2^{-149}	$(2-2^{-23})\cdot2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	`double`	64	2^{-1074}	$(2-2^{-52})\cdot2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	`double d = 65.55;`
Other	`boolean`	--	--	--	false, true	`boolean b = true;`
	`void`	--	--	--	--	--

Figure: Primitive Data Types

- Primitive types are the most basic data types that can be found in programming languages such as C, C++ or Python. We distinguish 8: **boolean**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**. A combination of char**s** results in a **string**.

Alternate and Repeating Conditional Structures & Loops

- In algorithms and programming, we call a **conditional structure** the set of instructions that test whether a condition is true or not. There are three of them:
  - The conditional structure **if**:

```
if (condition) {
my_instructions;
}
```

Alternate and Repeating Conditional Structures & Loops

- **Sequence:**
  - The Conditional Structure **if ...else** :

```
if (condition) {
my_instructions;
}
else {
my_other_instructions
}
```

Alternate and Repeating Conditional Structures & Loops

- **Sequence:**
  - The conditional structure **if ...elif ...else** :

```
if (condition_1) {
my_instructions_1;
}
elif (condition_2) {
my_instructions_2
}
else {
my_other_instructions
}
```

Alternate and Repeating Conditional Structures & Loops

- Furthermore, an **iterative conditional structure** allows the same series of instructions to be executed several times (iterations). The **while** (while) instruction executes the instruction blocks as long as the while condition is true. The same principle for the **for** loop (for), the two structures are detailed below:
  - The iterative Conditional Structure **while**:

```
while (condition)
my_instructions;
```

Alternate and Repeating Conditional Structures & Loops

- **Sequence**:
  - The iterative conditional structure **for**:

```
for (my_initialization; my_condition; my_increment){
my_instructions;
}
```

- **Remark**: The major difference between for loop and the while loop is that for loop is used when the number of iterations is known, whereas execution is done in the while loop until the statement in the program is proved wrong.

Alternate and Repeating Conditional Structures & Loops

- **Suite**:
  - The iterative conditional structure **do ...while**:

```
do {
my_instructions;
while(my_condition);
}
```

- **Note**: a difference lies between the **while** loop and the **do while** loop: in the **while** loop the check condition executes before the first iteration of the loop, while do while, checks the condition **after** the execution of the instructions inside the loop.

Declaration and signature of a fonction

- We recall that the declarative structure of any function takes the following form:

```
return_type my_function_name(my_arguments) {
my_instructions;
return my_value; // (otherwise the function is void)
}
```

Notion of Complexity of an algorithm

- The complexity of an algorithm is the formal study of the amount of time and memory resources needed to run that algorithm.

> **Definition:** The complexity of an algorithm is the measure of the number of fundamental operations it performs on a data set. Complexity is expressed as a function of dataset size.

Notion of Complexity of an algorithm

- The calculation of the complexity of an algorithm is its order of magnitude. To do this, we need asymptotic notations.

$$
\begin{aligned}
\mathrm{O} &: f = O(g) \Leftrightarrow \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n) \\
\Omega &: f = \Omega(g) \Leftrightarrow g = O(f) \\
\mathrm{o} &: f = o(g) \Leftrightarrow \forall c \geq 0, \exists n_0, \forall n \geq n_0, f(n) \leq c \times g(n) \\
\Theta &: f = \Theta(g) \Leftrightarrow f = O(g) \text{ et } g = O(f)
\end{aligned}
$$

Notion of Complexity of an algorithm

- **Classes of complexity:** Algorithms can be sorted in terms of their complexity as below :

  - The **sub-linear** algorithms whose complexity is in general in $O(logn)$.
  - Algorithms **linear** in complexity $O(n)$ and those in complexity $O(nlogn)$ are considered fast.
  - **polynomial** algorithms in $O(n^k)$ for $k > 3$ are considered slow.
  - The **exponential** algorithms in $O(x^n)$ (whose complexity is greater than any polynomial in n) strongly discouraged in practice when the size of the data exceeds a few units.

Notion of Complexity of an algorithm

- **Next:** here is a figure illustrating the previous remarks:

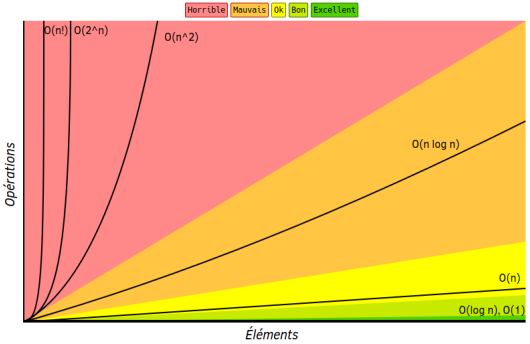

Figure: complexityOrders

Notion de recursivity: recursive functions

Definitions:

- An algorithm is called **recursive** if it is defined by it-self.
- In the same manner, a function is called **recursive** if it references itself.

- The definition seems easy to understand, because in fact it is, below is an example:

Notion of recursivity: recursive functions

- Admettons que l'on veuille calculer un réel $x \in \mathbb{R}$ à la puissance $n \in \mathbb{N}$, **l'algorithme trivial** fait l'affaire, néanmoins comment l'écrire d'une manière élégante ?! On peut ainsi le rendre récursif:

```
double recursive_power(double _x, size_t _n)
{
  if (_x == 0. && _n == 0) return 1.;
  return (_n==0) ? (1.0) :  (_x * recursive_power(_x, _n - 1)) ;
}
```

Notion of recursivity: recursive functions

- Le calcul d'un réel $x \in \mathbb{R}$ à la puissance $n \in \mathbb{N}$ peut donc être calculé d'une manière triviale comme récursive, néanmoins l'algorithme coûte $n - 1$ multiplications dans les deux cas, par conséquent la complexité est de $O(n)$, ainsi la complexité est une fonction affine qui est proportionnelle à la taille de n. Pour n assez large, c'est une catastrophe.

- Comment remédier au problème ?!

Notion de récursivité: Fonctions récursives

- **Solution:** l'exponentiation rapide ! En se basant sur l'écriture binaire d'un nombre

$$n = \sum_{k=0}^{d} b_k 2^k$$

où: $d = \text{floor}(log2(n) + 1)$, $b_k \in \{0, 1\}$, et $k \in [|0, d|]$.
(où: floor(x) $= \lfloor x \rfloor$) En remarquant que:

$$x^n = x^{(\sum_{k=0}^{d} b_k 2^k)} = x^{b_0} \times x^{2b1} \times ... \times x^{2^d b_d}$$

Notion de récursivité: Fonctions récursives

- **Suite:** on peut ainsi décomposer le calcul de x^n sous la forme d'une fonction récursive pour n pair et n impair. La fonction suivante illustre la démarche à suivre:

```
double optimized_recursive_power(double x, size_t n) {
  if (x == 0. && _n == 0)  return 1.;
  if (n == 1)              return x;
  if (n%2 == 0)      return     optimized_recursive_power(x * x,   n  /2);
  else               return x * optimized_recursive_power(x * x, (n-1)/2);
}
// Time complexity  O(log n)
```

Définition: on dit qu'un algorithme est **optimal** si sa complexité est minimale parmi les algorithmes de sa classe.

Notion de récursivité: Fonctions récursives

- **Types de récursivité:** on distingue quatre types de récursivité, on les liste comme suit:
  - Récursivité **simple** (cas de la fonction **recursive_power_fun.cpp**)
  - Récursivité **multiple** (cas de la fonction **optimized_recursive_power_fun.cpp**)
  - Récursivité **mutuelle**, quand une fonction *f* appelle une fonction *g* qui elle-même appelle la fonction *f*. (exemple: une fonction paire est forcément impaire, et réciproquement).
  - Récursivité **imbriquée**, une fonction *f* qui appelle elle-même avec un paramètre d'elle-même. $f(f(f))$. (fonction d'Ackermann par exemple).

Les tableaux

- Un tableau est une collection de données qui satisfait les conditions suivantes:
  - Une collection de données qui ont le **même type** (bool, int, double, string...)
  - Les éléments sont stockés d'une manière **contiguë** dans la mémoire (image 3)
  - La **taille du tableau** doit être connue dès la déclaration du tableau.
  - En considérant un tableau de taille *N*, l'**indice du premier élément** du tableau est 0, tandis que l'**indice du dernier élément** est $N - 1$. (dans la majorité des langages de programmation: C/C++, Python etc.)
  - Les éléments d'un tableau sont **accessibles** à partir de leur **position**, par conséquent leur **indice**.

Example of array representation
in memory of an array of ints

Memory Location	400	404	408	412	416	420
Index	0	1	2	3	4	5
Data	18	10	13	12	19	9

Figure: arrayExample

Les tableaux

- Avantages des tableaux:
  - **Optimisation du code:** la complexité pour ajouter ou supprimer un élément à la fin du tableau, l'accès à un élément *via* son indice, rajouter un élément ou le supprimer est en $O(1)$. La **recherche séquentielle** (linéaire) d'un élément dans un tableau a une complexité de $O(n)$.
  - **Facilité d'accès:** l'accès direct d'un élément *via* son indice (e.g: myArray[3]), itérer sur les éléments d'un tableau grâce à une boucle, tri simplifié.
- Inconvénients des tableaux:
  - La contrainte de devoir déclarer sa taille au préalable du **compile time** (ce qu'on appelle un **tableau statique**, on y reviendra), et que la taille du tableau ne grandit pas lors du **run time**.
  - L'insertion et la suppression des éléments peuvent être coûteux comme les éléments doivent être gérés en fonction de la nouvelle zone mémoire allouée (Exemple: vouloir ajouter un élément au milieu d'un tableau).

Les tableaux

- Déclaration d'un tableau en C/C++:
  - **typename myArray**$[n]$: crée un tableau de taille n (n éléments) de type **typename** avec valeurs au hasard. **Exemple: int a**$[3]$ produit:

```
Index: 0 | address: 0x7ffee253c4c0 | value: -497826512
Index: 1 | address: 0x7ffee253c4c4 | value: 32766
Index: 2 | address: 0x7ffee253c4c8 | value: 225226960
```

  - **typename myArray**$[n] = \{v_1, ..., v_n\}$: crée un tableau de taille n de type **typename** avec valeurs initialisées à v_1 jusqu'à v_n. **Exemple: int a**$[3] = \{1, 4, 9\}$ crée un tableau de 3 éléments initialisés à 1, 4 et 9 respectivement.

Les tableaux

- Déclaration d'un tableau en C/C++:
  - **typename myArray**$[n] = \{v, ..., v\}$: crée un tableau de taille *n* de type **typename** avec toutes les valeurs initialisées à *v*. **Exemple:** **int a**$[5] = \{1, 1, 1, 1, 1\}$ crée un tableau de 5 éléments, tous initialisés à 1.

```
Index: 0| address: 0x7ffeea95b510| value: 1
Index: 1| address: 0x7ffeea95b514| value: 1
Index: 2| address: 0x7ffeea95b518| value: 1
Index: 3| address: 0x7ffeea95b51c| value: 1
Index: 4| address: 0x7ffeea95b520| value: 1
```

Les tableaux

- Déclaration d'un tableau en C/C++:
  - **typename myArray**$[n] = \{\}$: crée un tableau de taille *n* de type **typename** avec toutes les valeurs initialisées à 0. **Exemple: int a**$[4] = \{\}$ crée un tableau de 4 éléments, tous initialisés à 0.

```
Index: 0| address: 0x7ffee3257510| value: 0
Index: 1| address: 0x7ffee3257514| value: 0
Index: 2| address: 0x7ffee3257518| value: 0
Index: 3| address: 0x7ffee325751c| value: 0
```

- **typename myArray**$[n] = \{v\}$: crée un tableau de taille *n* de type **typename** avec la première valeur initialisée à *v*, le reste à 0.

Notion de pointeurs en C et en C++

- Un **pointeur** stocke l'adresse d'une variable ou une zone mémoire.
- Un pointeur peut être apparenté à un tableau, dans ce cas, le pointeur pointe sur la première case mémoire du tableau.
- Syntaxe générale: **typename *my_variable_name**
- **Exemples:**
  - **Exemple 1:** int x = 10; int *ptr; ptr = &x; (ou de manière équivalente:)
  - **Exemple 2:** int x = 12; int *ptr = &x;

Introduction aux structures de données

- Le traitement de données comme le tri, l'insertion ou encore la suppression posent divers problèmes quant à la gestion du temps et de l'espace mémoire (notion de complexités temporelle et spatiale respectivement).
- En informatique, on peut représenter la notion mathématique d'ensemble de plusieurs manières.
- Pour chaque problème donné, la meilleure solution sera celle qui permettra de concevoir le meilleur algorithme (le plus esthétique, et le plus performant: de moindre complexité).
- Chaque élément d'un ensemble pourra comporter plusieurs champs qui peuvent être examinés dès lors que l'on possède un pointeur (comprendre référence) sur cet élément.

Structures de données

- Il existe une variété de structures de données, elles supportent potentiellement tout une série d'opérations :
  - FIND(S, k) : étant donné un ensemble S et une clé k, le résultat de cette requête est un pointeur sur un élément de S de clé k, s'il en existe un, et la valeur NIL sinon —NIL étant un pointeur ou une référence sur «rien».
  - INSERT(S, x) : ajoute à l'ensemble S l'élément pointé par x.
  - DELETE(S, x): supprime de l'ensemble S son élément pointé par x (si l'on souhaite supprimer un élément dont on ne connaît que la clé k, il suffit de récupérer un pointeur sur cet élément *via* un appel à FIND(S, k)).

Structures de données

- Et si l'ensemble des clés, ou l'ensemble lui-même, est totalement ordonné (trié), d'autres opérations sont possibles :
  - MINIMUM(S) : renvoie l'élément de S de clé minimale.
  - MAXIMUM(S) : renvoie l'élément de S de clé maximale.
  - SUCCESSOR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S pointé par x, et NIL dans le cas contraire.
  - PREDECESSOR(S, x) : renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S pointé par x, et NIL dans le cas contraire.

Structures de données

- Voici une liste non exhaustive des structures des données les plus fréquemment utilisées:
  - array (les tableaux)
  - vector (les vecteurs)
  - forward_list (liste chaînée a.k.a linked list)
  - list (liste doublement chaînée a.k.a doubly linked list)
  - map (dictionnaire a.k.a dictionary)
  - unordered_map (dictionnaire non ordonné)
  - set (ensemble ordonné)
  - unordered_set (ensemble non ordonné)

Structures de données

- **Suite**:
  - stack (piles)
  - queue (files a.k.a queues)
  - deque (queue à deux bouts a.k.a double ended queues)
  - heap
  - pair (paire de valeurs a.k.a 2-uplet)
  - tuple (n valeurs a.k.a n-uplet)

Array

- On a déjà vu les tableaux précédemment, une chose qui a été omise jusqu'à présent:
  - Un tableau est dit **statique** si sa taille N est connue dès la déclaration. (en C/C++ l'allocation se fait au stack)
  - Un tableau est dit **dynamique** si sa taille N varie. (en C/C++ l'allocation se fait au heap).
- Les tableaux comme on les a connus dans la section précédente sont des tableaux statiques!
- Les tableaux dynamiques sont représentés en C++ par ce qu'on appelle un **vector** (vecteur).

Vector

Remarque: Dans la suite, et sans abus de notation, on appellera **Container** une ”structure de donnée” donnée.

- Les vecteurs ont exactement les mêmes propriétés que les tableaux dynamiques avec la capacité de pouvoir changer de taille automatiquement si un élément a été inséré ou supprimé, avec la mémoire gérée automatiquement par le container lui-même (vector ici).
- Les éléments d’un vecteur sont placés d’une manière contiguë dans la mémoire, ils peuvent donc être accessibles *via* leur index ou *via* des **itérables**. (iterators)

Vector

Suite:

- L'insertion d'un élément dans un vecteur se fait à la fin avec push_back(x) (C++11) ou emplace_back(x) (C++17) (prend un temps différentiel car parfois le vecteur doit être étendu en terme de taille).
- La suppression du dernier élément prend $O(1)$ comme il n'y a pas de changement de taille du vecteur.
- L'insertion ou la suppression au début ou au milieu se fait en temps linéaire $O(n)$.

Liste chaînée (linked list)

- Une liste chaînée est une structure de données linéaire, dans laquelle les éléments **ne sont pas** stockés d'une manière contiguë.
- Les éléments d'une liste chaînée sont liés par des pointeurs comme l'image ci-dessous:

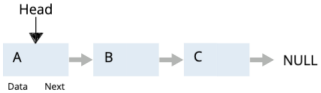

Figure: linked_list

Liste doublement chaînée (doubly linked list)

- Une liste doublement chaînée est une liste dont chaque élément peut être accédé à l'aide de pointeurs aux éléments positionnés immédiatement avant et après lui dans la liste.
- Une liste doublement chaînée (ldd) contient un pointeur en plus, appelé typiquement **previous**, ce dernier avec le pointeur **next** permettent de naviguer entre les éléments de la ldd. (cf figure ci-dessous:)

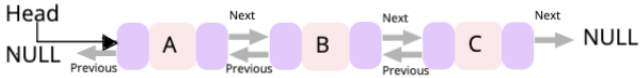

Figure: double_linked_list

Map

- Les maps sont des containers associatifs qui stockent les éléments comme à l'image d'un dictionnaire.
- Chaque élément a une valeur **key** (clé) et une valeur qui lui est associée **value** (valeur).
- Deux valeurs différentes ne peuvent pas avoir la même clé!

Map non ordonnée

- unordered_map est un container associatif qui stocke les éléments formés d'une combination de la valeur et de la clé.
- La clé permet d'identifier d'une manière unique l'élément, et la valeur est le contenu associé à cette clé.
- La clé et la valeur peuvent être de n'importe quel type.
- unordered_map est une data structure ressemblant à un dictionnaire.
- unordered_map contient une liste successive de paires (key, value) qui permet l'accès à un élément grâce à sa clé unique.

Ensemble (set)

- Les sets sont un type de containers associatifs dans lesquels chaque élément doit être unique.
- Les valeurs sont stockées dans un ordre spécifique, ou ascendant ou descendant.
- Méthodes (opérations) en $O(logn)$.

Ensemble non ordonné (unordered_set)

- Un unordered_set est implémenté en utilisant une table de hachage où les clés sont hachés dans des indices d'une table de hachage, ainsi l'insertion est faite au hasard.
- Les opérations (méthodes) de unordered_set prennent un temps constant $O(1)$ en moyenne, et un temps linéaire $O(n)$ dans le pire des cas, cela dépendra de la fonction de hachage utilisée.

Piles

- Les piles (stack) sont un type d'adapteurs de containers suivant le principe LIFO (Last In First Out), où un nouveau élément peut être uniquement ajouté au top (début) et qu'un élément peut être uniquement supprimé de ce même top.

Files

- Les Queues sont un type d'adapteur de container qui opère selon le modèle FIFO (First in First out).
- Les éléments sont ajoutés à la fin (end) et sont supprimés du début (front).
- Les Queues utilisent un objet encapsulé de deque ou list (container de class séquentiel) comme container implicite, donnant droit à tous les membres et méthodes hérités.

Les algorithmes de tri

- Selection sort
- Bubble sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort
- Counting sort
- Radix sort
- Bucket sort

Divide to conquer algorithms

- TP Méthode de Strassen.

Programmation dynamique

- La programmation dynamique est une méthode algorithmique afin de résoudre des problèmes d'optimisation.