Course: Basic mathematics and algorithmic

Ali ZAINOUL

for Keyce Academy
December 22, 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

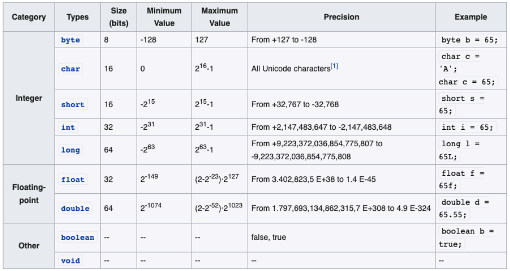
A computer program is a set of instructions and operations intended to be executed by a computer.
A source program is code written by a computer scientist in a programming language.

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, input processing**: in this part we take care of declaring the different inputs / variables / constants that the program needs..
- **Body of the algorithm 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}	$(2-2^{-23})\cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	`float f = 65f;`
	`double`	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	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 chars 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

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

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

Alternate and Repeating Conditional Structures & Loops

- **Continuation:**
  - 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

- **Continuation**:
  - 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.

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

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:

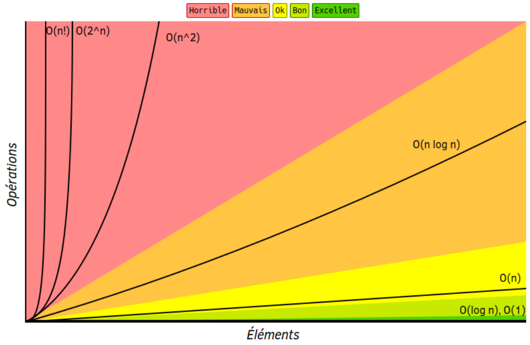


Figure: complexityOrders

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

- Admit that we want to calculate $x \in \mathbb{R}$ to the power of $n \in \mathbb{N}$, **the trivial algorithm** gives the correct answer, but how to write it more elegantly ?! We can then make it recursive:

```
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

- The calculation of $x \in \mathbb{R}$ to the power $n \in \mathbb{N}$ may be calculated in a recusrive manner, however the algorithm cost $n - 1$ multiplications/products in the two cases, therefore the complexity is $O(n)$, hence the complexity is a affine function proportional to the size of inputs n. For n big enough, it is a disaster.
- How to fix the problem?!

Notion of recursivity: recursive functions

- **Solution:** Rapid exponentiation or square-and-multiply method ! Based on the binary representation of a number

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

Where: $d = \text{floor}(log2(n) + 1)$, $b_k \in \{0, 1\}$, and $k \in [|0, d|]$. (Where: floor(x) $= \lfloor x \rfloor$) By remarking that:

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

Notion of recursivity: recursive functions

- **Next:** we may then decompose the calculation of x^n in the form of a recursive function for n even and n odd. The following function illustrates the way to follow:

```
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)
```

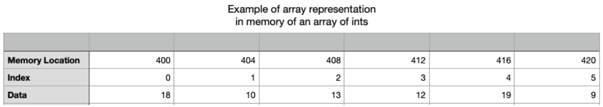
Definition: an algorithm is said to be **optimal** if its complexity is minimal among the algorithms of its class.

Notion of recursivity: recursive functions

- **Types of recursivity:** we distinguish four types of recursivity, we list them as follow:
  - **Simple** recursivity (the case of the function **recursive_power_fun.cpp**)
  - **Multiple** recursivity (the case of the function **optimized_recursive_power_fun.cpp**)
  - **Mutual** recursivity, when a function f calls a function g which itself calls the function f. (exemple: we may define two functions: is_Odd(n) and is_Even(n), the first is calling the latter, and the latter calls the first one too).
  - **Nested** recursivity, a function f that calls itself with a parameter of itself. $f(f(f)))$. (Ackermann function for instance).

Arrays

- An array is a collection of data elements of the same type (generally) that satisfies the following conditions:
    - A collection of data that have the **same type** (bool, int, double, string...)
    - Items are stored **contiguously** in memory (image 3)
    - The **size of the array** must be known when declaring the array.
    - Considering an array of size N, the **index of the first element** of the array is 0, while the **index of the last element** is $N-1$. (in most programming languages: C/C++, Python etc.)
    - The elements of an array are **accessible** from their **position**, hence their **index**.


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

Arrays

- Advantages of arrays:
  - **Code optimization:** the complexity to add or delete an element at the end of the array, access to an element *via* its index, add an element or delete it is in $O(1)$. The **sequential** (linear) search for an element in an array has a complexity of $O(n)$.
  - **Ease of access:** direct access to an element *via* its index (e.g: myArray[3]), iterating over the elements of an array using a loop, simplified sorting.
- Drawbacks of arrays:
  - The constraint of having to declare its size prior to **compile time** (what is called a **static array**, we will come back to this), and that the size of the array does not grow during **runtime**.
  - Inserting and deleting items can be expensive as items have to be managed according to the newly allocated memory area (Example: wanting to add an item in the middle of an array).

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n]$: creates an array of size n (n elements) of type textbftypename with random values. **Example: int a**$[3]$ produces:

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

  - **typename myArray**[n]=$\{v_1, ..., v_n\}$: creates an array of size n of type **typename** with values initialized to v_1 until v_n. **Example: int a**$[3] = \{1, 4, 9\}$ creates an array of 3 elements initialized to 1, 4 and 9 respectively.

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{v, ..., v\}$: creates an array of size n of type **typename** with all values initialized to v. **Example: int a**$[5] = \{1, 1, 1, 1, 1\}$ creates an array of 5 elements, all initialized to 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
```

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{\}$: creates an array of size *n* of type **typename** with all values initialized to 0. **Example: int a**$[4] = \{\}$ creates an array of 4 elements, all initialized to 0.

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

- **typename myArray**$[n] = \{v\}$: creates an array of size *n* of type **typename** with the first value initialized to *v*, the rest to 0 .

Notion of pointers in C and C++

- A **pointer** stores the address of a variable or a memory area.
- A pointer can be related to an array, in this case, the pointer points to the first memory cell of the array.
- General syntax: **typename *my_variable_name**
- **Examples:**
  - **Example 1:** int x = 10; int *ptr; ptr = &x; (or equivalently:)
  - **Example 2:** int x = 12; int *ptr = &x;

Introduction to Data Structures

- Data processing such as sorting, insertion or even deletion poses various problems regarding the management of time and memory space (notion of temporal and spatial complexities respectively).
- In computer science, the mathematical notion of set can be represented in several ways.
- For each given problem, the best solution will be the one that will allow to design the best algorithm (the most aesthetic, and the most efficient: of less complexity).
- Each element of a set may include several fields that can be examined when we have a pointer (include reference) on this element.

Data structures

- There are a variety of data structures, they potentially support a whole range of operations:
    - FIND(S, k): given a set S and a key k, the result of this query is a pointer to an item of S with key k, if one exists, and the value NIL otherwise —NIL being a pointer or reference to ”nothing”.
    - INSERT(S, x): adds to the set S the element pointed to by x.
    - DELETE(S, x): deletes from the set S its element pointed to by x (if we want to delete an element of which we only know the key k, we just need to recover a pointer to this element *via* a call to FIND(S, k)).

Data structures

- And if the set of keys, or the set itself, is totally ordered (sorted), other operations are possible:
  - MINIMUM(S): returns the item of S with minimum key.
  - MAXIMUM(S): returns the maximum key item of S.
  - SUCCESSOR(S, x): returns, if it exists, the element of S immediately greater than the element of S pointed to by x, and NIL otherwise.
  - PREDECESSOR(S, x): returns, if it exists, the element of S immediately smaller than the element of S pointed to by x, and NIL otherwise.

Data structures

- Here is a non-exhaustive list of the most frequently used data structures:
    - array (arrays)
    - vector (the vectors)
    - forward_list (linked list a.k.a linked list)
    - list (doubly linked list a.k.a doubly linked list)
    - map (dictionary a.k.a dictionary)
    - unordered_map (unordered dictionary)
    - set (ordered set)
    - unordered_set (unordered set)

Data structures

- **Continued:**
  - stack (stacks)
  - queue (files a.k.a queues)
  - deque (double ended tails a.k.a double ended tails)
  - heap
  - pair (a.k.a 2-tuple value pair)
  - tuple (n values a.k.a n-tuple)

Array

- We've seen arrays before, one thing that's been omitted so far:
  - An array is said to be **static** if its size N is known from the declaration. (in C/C++ the allocation is done on the stack)
  - An array is said to be **dynamic** if its size N varies. (in C/C++ the allocation is done at the heap).
- Arrays as we knew them in the previous section are static arrays!
- Dynamic arrays are represented in C++ by what is called a **vector**.

Vector

Remark: In the following, and without abuse of notation, we will call **Container** a given ”data structure”.

- Vectors have exactly the same properties as dynamic arrays with the ability to change size automatically if an item has been inserted or deleted, with the memory managed automatically by the container itself (vector here).
- Elements of a vector are placed contiguously in memory, so they can be accessed *via* their index or *via* **iterables**. (iterators)

Vector

Next:

- Inserting an item into a vector is done at the end with push_back(x) (C++11) or emplace_back(x) (C++17) (takes differential time because sometimes the vector must be extended in terms of size).
- Removing the last item takes $O(1)$ as there is no change in vector size.
- The insertion or deletion at the beginning or in the middle is done in linear time $O(n)$.

Linked list

- A linked list is a linear data structure, in which **items are not** stored in a contiguous manner.
- Items in a linked list are linked by pointers like the image below:

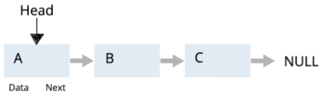


Figure: linked_list

Doubly linked list

- A doubly linked list is a list each item of which can be accessed using pointers to the items positioned immediately before and after it in the list.
- A doubly linked list (ldd) contains an additional pointer, typically called **previous**, the latter together with the pointer **next** allow navigation between the elements of the ldd. (see picture below:)

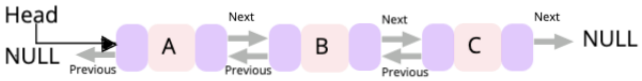


Figure: double_linked_list

Map

- Maps are associative containers that store items like a dictionary.
- Each item has a value **key** (key) and a value associated with it **value** (value).
- Two different values cannot have the same key!

Unordered map

- unordered_map is an associative container that stores items formed from a combination of value and key.
- The key uniquely identifies the item, and the value is the content associated with that key.
- Key and value can be of any type.
- unordered_map is a dictionary-like data structure.
- unordered_map contains a successive list of pairs (key, value) which allows access to an item thanks to its unique key.

Set (set)

- Sets are a type of associative container in which each item must be unique.
- Values are stored in a specific order, either ascending or descending.
- Methods (operations) in $O(logn)$.

Unordered set (unordered_set)

- An unordered_set is implemented using a hash table where keys are hashed into hash table indices, so insertion is done at random.
- The operations (methods) of unordered_set take constant time $O(1)$ on average, and linear time $O(n)$ in the worst case, depending on the hash function used.

Stacks

- Stacks are a type of container adapters following the LIFO (Last In First Out) principle, where a new item can only be added to the top (beginning) and an item can only be removed from it. same top.

Queues

- Queues are a type of container adapter that operate on the FIFO (First in First out) model.
- Items are added at the end (end) and removed from the beginning (front).
- Queues use an encapsulated deque or list object (sequential class container) as the implicit container, entitling all inherited members and methods.

Sorting algorithms

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

Divide to conquer algorithms

- TP Strassen method.

Dynamic Programming

- Dynamic programming is an algorithmic method to solve optimization problems.