CP- EITC

Lecture 1

CP Cell Leads:
Idriss EL ABIDI
Souhail LOUZI

Getting Started

November, 2024

1 Algorithm

2 Time Complexity

3 CP Virtual Judge

4 Expected Knowledge

5 Recursion

1 Algorithm

2 Time Complexity

3 CP Virtual Judge

4 Expected Knowledge

5 Recursion

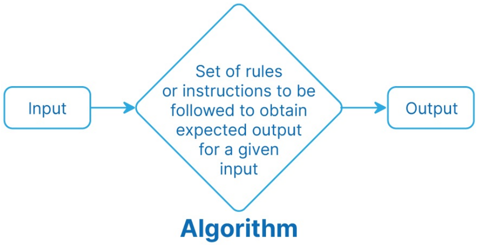
what is an algorithm ?

Definition :

An algorithm is a sequence of ordered instructions that takes some input data in order to get some output data. This abstract notion was first discussed formally by the Muslim and Persian mathematician Muáammad ibn Musa Al-Khwârizmî.

Figure 1: Al-Khwârizmî

Elements of an algorithm :


Input
Set of rules or instructions to be followed to obtain expected output for a given input
Output
Algorithm

Characteristics of an algorithm:

- **is correct (gives the correct answers)**
- **can be automated (programmable)**
- **is finite (has an end state)**

1 Algorithm

2 Time Complexity

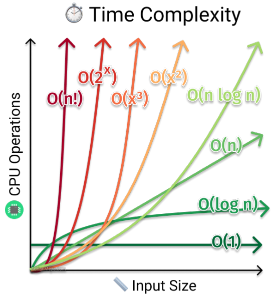
3 CP Virtual Judge

4 Expected Knowledge

5 Recursion

Complexity Calculations

In programming contests, your program needs to finish running within a certain timeframe in order to receive credit.
We want a method to calculate how many operations it takes to run each algorithm, in terms of the input size n. Fortunately, this can be done relatively easily using Big O notation, which expresses worst-case time complexity as a function of n as n gets arbitrarily large.



Complexity Calculations :

Big O notation

In Big O notation, we denote the complexity of a function as $\mathcal{O}(f(n))$, where constant factors and lower-order terms are generally omitted from $f(n)$. We'll see some examples of how this works, as follows.

Complexity classes:

- $\mathcal{O}(1)$: The running time of a constant-time algorithm does not depend on the input size. A typical constant-time algorithm is a direct formula that calculates the answer.

```
int a = 5;
int b = 7;
int c = 4;
int d = a + b + c + 153;
```

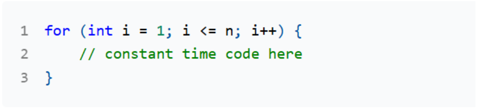
Complexity Calculations :

- $\mathcal{O}(\log n)$: A logarithmic algorithm often halves the input size at each step. The running time of such an algorithm is logarithmic, because log2(n) equals the number of times n must be divided by 2 to get 1.
  Example : Binary search.
- $\mathcal{O}(\sqrt{n})$: A square root algorithm is slower than $\mathcal{O}(\log n)$ but faster than O(n). A special property of square roots is that the square root lies, in some sense, in the middle of the input.
  Example : Prime factorization of an integer, or checking primality or compositeness of an integer naively.

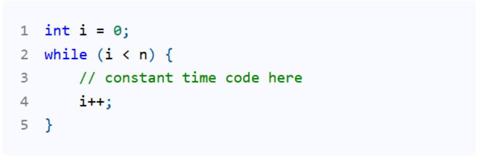
Complexity Calculations :

- $\mathcal{O}(n)$: A linear algorithm goes through the input a constant number of times. This is often the best possible time complexity, because it is usually necessary to access each input element at least once before reporting the answer.

The time complexity of loops is the number of iterations that the loop runs. For example, the following code examples are both $\mathcal{O}(n)$.



```
for (int i = 1; i <= n; i++) {
    // constant time code here
}
```

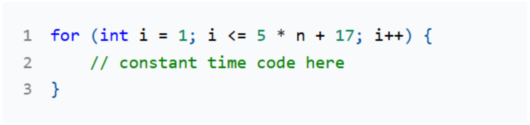


```
int i = 0;
while (i < n) {
    // constant time code here
    i++;
}
```

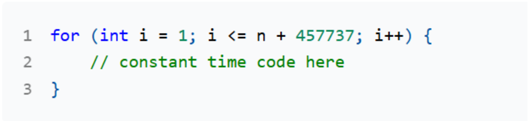
Complexity Calculations :

- Because we ignore constant factors and lower order terms, the following examples are also $\mathcal{O}(n)$:

.



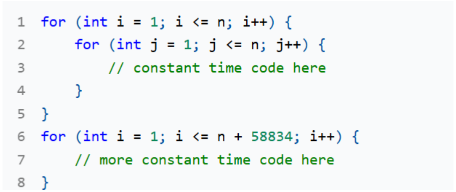
```
for (int i = 1; i <= 5 * n + 17; i++) {
    // constant time code here
}
```



```
for (int i = 1; i <= n + 457737; i++) {
    // constant time code here
}
```

Complexity Calculations :

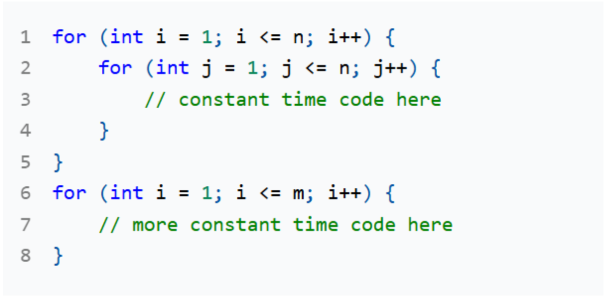
- $\mathcal{O}(n \log n)$: This time complexity often indicates that the algorithm sorts the input, because the time complexity of efficient sorting algorithms is $\mathcal{O}(n \log n)$.
  Example : usually $\mathcal{O}(n \log n)$ for default sorting algorithms (merge sort, Collections.sort, Arrays.sort)
- $\mathcal{O}(n^2)$: A quadratic algorithm often contains two nested loops. It is possible to go through all pairs of the input elements in $\mathcal{O}(n^2)$ time.
  If an algorithm contains multiple blocks, then its time complexity is the worst time complexity out of any block. For example, the following code is $\mathcal{O}(n^2)$.



```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant time code here
    }
}
for (int i = 1; i <= n + 58834; i++) {
    // more constant time code here
}
```

Complexity Calculations :

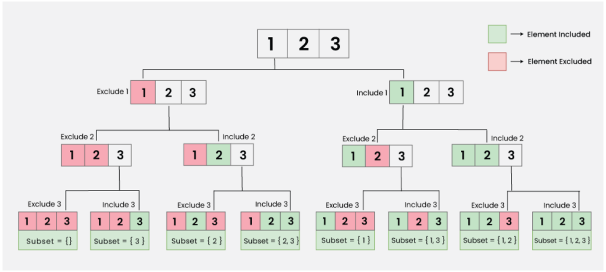
- The following code is $\mathcal{O}(n^2 + m)$, because it consists of two blocks of complexity $\mathcal{O}(n^2)$ and $\mathcal{O}(m)$, and neither of them is a lower order function with respect to the other.



```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // constant time code here
    }
}
for (int i = 1; i <= m; i++) {
    // more constant time code here
}
```

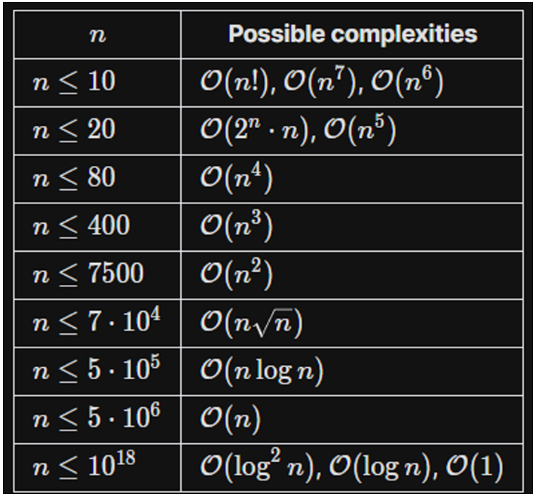
Complexity Calculations :

- $\mathcal{O}(2^n)$: This time complexity often indicates that the algorithm iterates through all subsets of the input elements. For example, the subsets of 1,2,3 are , 1, 2, 3, 1,2, 1,3, 2,3 and 1,2,3.



- $\mathcal{O}(n!)$: This time complexity often indicates that the algorithm iterates through all permutations of the input elements. For example, the permutations of 1,2,3 are (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2) and (3,2,1).

Estimating efficiency



n	Possible complexities
$n \leq 10$	$\mathcal{O}(n!)$, $\mathcal{O}(n^7)$, $\mathcal{O}(n^6)$
$n \leq 20$	$\mathcal{O}(2^n \cdot n)$, $\mathcal{O}(n^5)$
$n \leq 80$	$\mathcal{O}(n^4)$
$n \leq 400$	$\mathcal{O}(n^3)$
$n \leq 7500$	$\mathcal{O}(n^2)$
$n \leq 7 \cdot 10^4$	$\mathcal{O}(n\sqrt{n})$
$n \leq 5 \cdot 10^5$	$\mathcal{O}(n \log n)$
$n \leq 5 \cdot 10^6$	$\mathcal{O}(n)$
$n \leq 10^{18}$	$\mathcal{O}(\log^2 n)$, $\mathcal{O}(\log n)$, $\mathcal{O}(1)$

Common Complexities and Constraints

- Mathematical formulas that just calculate an answer: $\mathcal{O}(1)$
- Unordered set/map: $\mathcal{O}(1)$ per operation
- Binary search: $\mathcal{O}(\log n)$
- Ordered set/map or priority queue: $\mathcal{O}(\log n)$ per operation
- Prime factorization of an integer, or checking primality or compositeness of an integer: $\mathcal{O}(\sqrt{n})$
- Reading in n items of input: $\mathcal{O}(n)$
- Iterating through an array or a list of n elements: $\mathcal{O}(n)$
- Sorting: usually $\mathcal{O}(n \log n)$ for default sorting algorithms (mergesort and quicksort used in std::sort())
- Iterating through all subsets of size k of the input elements: $\mathcal{O}(n^k)$. For example, iterating through all triplets is $\mathcal{O}(n^3)$.
- Iterating through all subsets: $\mathcal{O}(2^n)$
- Iterating through all permutations: $\mathcal{O}(n!)$

1 Algorithm

2 Time Complexity

3 CP Virtual Judge

4 Expected Knowledge

5 Recursion

Virtual Judge

Definition :

Its an automated program that evaluates the solutions submitted by the contestants during a contest based on the checkers and verification codes he has beengiven. Its named an online judge when its accessible online. When there is many contestants submitting at the same time, the v-judge may take some to process each submission. However, the input used in the judging test cases are always hidden to public during the contest.

Virtual Judge

- **AC:** Accepted answer.
- **WA:** (Wrong Answer) Your code works fine, but it gives wrong results either in the value or in the format.
- **TLE:** (Time Limit Exceeded) Your code works fine, but it needs optimization in time complexity.
- **MLE:** (Memory Limit Exceeded) Your code works fine, but it needs optimization in space complexity.
- **CE:** (Compilation Error) Your code fails to compile due to a syntax error.
- **RE:** (Runtime Error) Your code crashes at some stage of its execution. (stack overflow, division by zero, index out of range..)

Scoring Modes

In CP, there exists 2 modes of computing score for problems:

ICPC style :

Its a binary approach of evaluating submission : yourcode must work on all the hidden test cases in order to get an AC. Otherwise, you get nothing.

IOI Style :

Its a non-binary approach where each problem is composed of sub-tasks. Each task describes an easier version of the full problem. You get some points for each sub-task achieved and therefore, the more your solution is global the more points youre going to collect.

Penalty

Mainly, all the problems have the same weight. The contestants are ranked by the number of problem they solved. However, to break the tie between contestants who solved the same number of problems, we rank them by submission time also named Penalty that is calculated with the formula :

$$Penalty = \sum_{p \in \text{solved problems}} T(p) + (20 \times W(p))$$

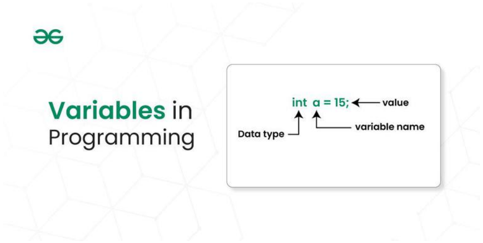
where :

- **W(p):** the number of wrong submissions made in p
- **T(p):** time elapsed from the start when p was solved

The lower is the penalty, the higher is the ranking. Thats why speed and precision count too.

1. Algorithm
2. Time Complexity
3. CP Virtual Judge
4. Expected Knowledge
5. Recursion

Variables



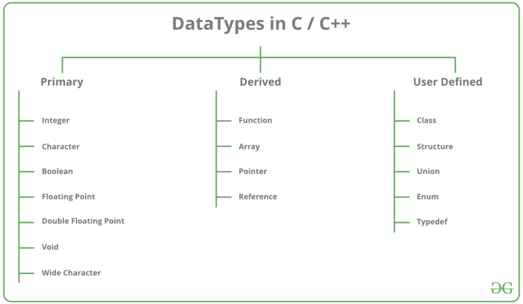
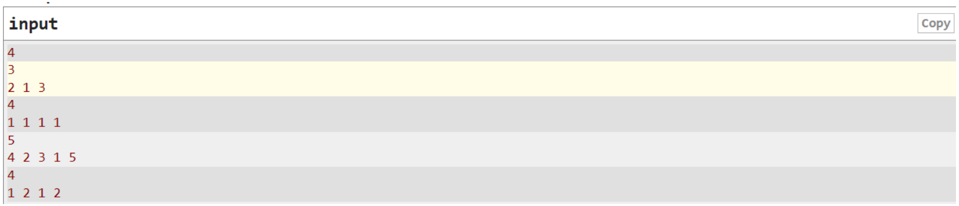


Figure 2: Variables & Data types

Data Types

Data Types			
Data Type	Size	Range	Macro Constants
int	4	-2,147,483,648 to 2,147,483,647	INT_MIN, INT_MAX
unsigned int	4	0 to 4,294,967,295	
long long int	8	$-(2^{63})$ to $(2^{63})-1$	LLONG_MIN, LLONG_MAX
unsigned long long	8	0 to 18,446,744,073,709,551,615	
signed char	1	-128 to 127	
unsigned char	1	0 to 255	
float	4	-3.4×10^{38} to 3.4×10^{38}	
double	8	-1.7×10^{308} to 1.7×10^{308}	

Reading Input



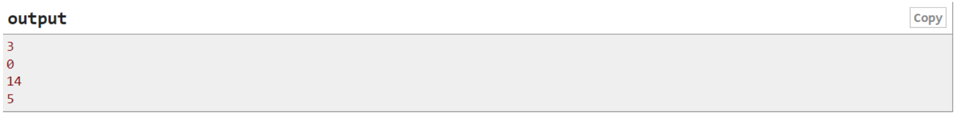
input

Copy

```
4
3
2 1 3
4
1 1 1 1
5
4 2 3 1 5
4
1 2 1 2
```

Figure 3: Input

Writing Output



```
3
0
14
5
```

Figure 4: Output

Loops

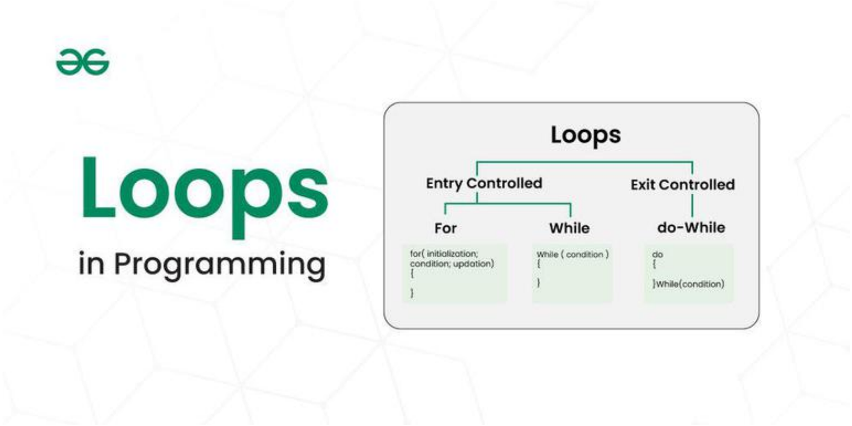


Figure 5: Loops

If / Else

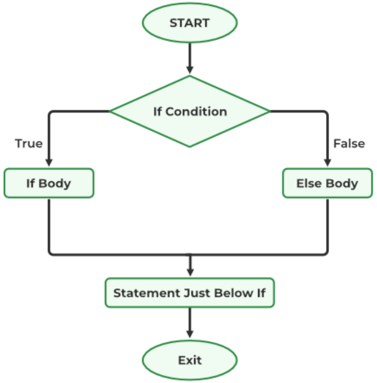
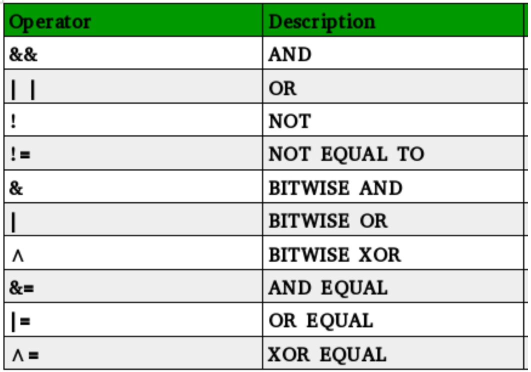


Figure 6: If / Else

Logical operators



Operator	Description
&&	AND
| |	OR
!	NOT
!=	NOT EQUAL TO
&	BITWISE AND
|	BITWISE OR
^	BITWISE XOR
&=	AND EQUAL
|=	OR EQUAL
^=	XOR EQUAL

Figure 7: Logical Operators

Functions

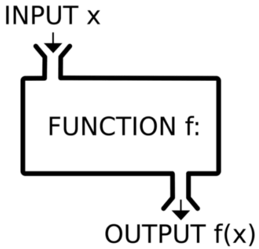


Figure 8: Functions

Arrays

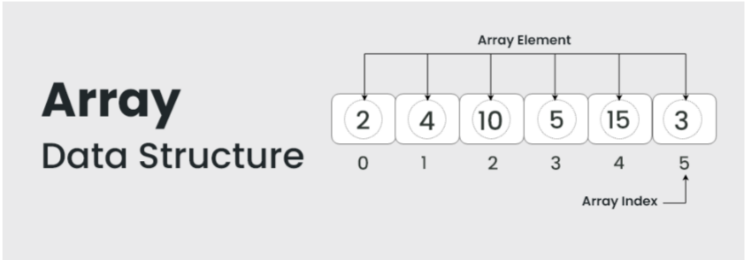


Figure 9: Array

1 Algorithm

2 Time Complexity

3 CP Virtual Judge

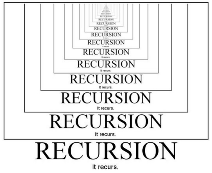
4 Expected Knowledge

5 Recursion

What is recursion ?

Definition :

Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.



The time complexity :

The time complexity of a recursive function depends on the number of times the function is called and the time complexity of a single call. The total time complexity is the product of these values.

What is recursion ?

For example, consider the following function:

Base case

Recursive call

```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

The call f(n) causes n function calls, and the time complexity of each call is O(1). Thus, the total time complexity is O(n).

All recursive algorithms must have the following:

- Base Case (i.e., when to stop)
- Work toward Base Case
- Recursive Call (i.e., call ourselves)

Recursion in the memory: illustration

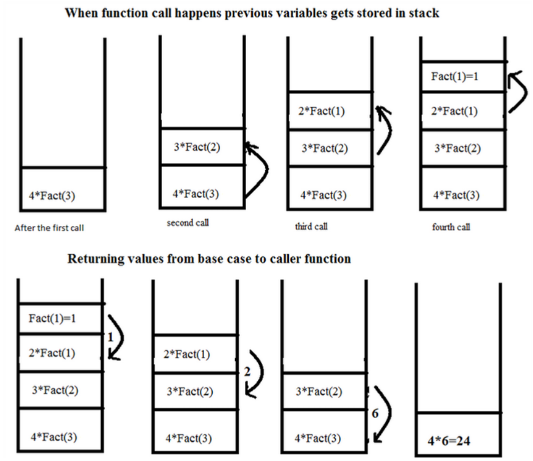


Figure 10: LIFO: Last-In-First-Out

Fibonacci

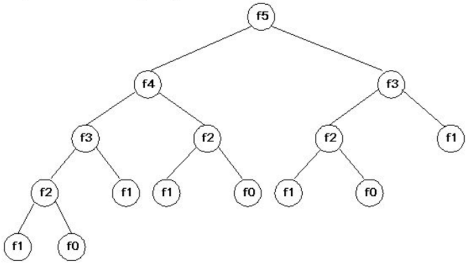
The definition of the Fibonacci number at location X in the sequence :

e.g. 0 , 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 ,.......

$F_i = i$ if $i \leq 1$

$F_i = F_{i-1} + F_{i-2}$ if $i \geq 2$

Solved by a recursive program:



Practice

Thank you for listening !

Idriss El Abidi
Souhail Louzi

Special thanks to the former CP leads:
Hamza BA-MOHAMMED, Hachim Hassani Lahsinui and mohamed ghacham amrani
for their insightful review and precious remarks.