



# Starfleet - Day 07

## Algorithms

Staff 42 [pedago@42.fr](mailto:pedago@42.fr)

*Summary: This document is the day07's subject for the Starfleet Piscine.*

# Contents

<b>I</b>	<b>General rules</b>	<b>2</b>
<b>II</b>	<b>Day-specific rules</b>	<b>3</b>
<b>III</b>	<b>Exercise 00: Heap sort</b>	<b>4</b>
<b>IV</b>	<b>Exercise 01: Sliding window</b>	<b>6</b>
<b>V</b>	<b>Exercise 02: It's a secret</b>	<b>9</b>
<b>VI</b>	<b>Exercise 03: Junk Food Search</b>	<b>11</b>
<b>VII</b>	<b>Exercise 04: Bubble path finder</b>	<b>15</b>
<b>VIII</b>	<b>Exercise 05: Memory loss</b>	<b>20</b>

# Chapter I

## General rules

- Every instructions goes here regarding your piscine
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- The exercises must be done in order. The evaluation will stop at the first failed exercise. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The subject can be modified up to 4 hours before the final turn-in time.
- You will NOT be graded by a program, unless explicitly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.
- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

# Chapter II

## Day-specific rules


- If asked, you must turn-in a file named `bigo` describing the time and space complexity of your algorithm as below. You can add to it any additional explanations that you will find useful.

```
$> cat bigo
O(n) time , with n the number of elements in the array.
O(1) space
$>
```

- Your work must be written in C. You are allowed to use all functions from standard libraries.
- For each exercise, you must provide a file named `main.c` with all the tests required to attest that your functions are working as expected.
- This last day is dedicated to some notions that might have been missed during the piscine. So you might recognize some exercises from previous days !

# Chapter III

## Exercise 00: Heap sort

	Exercise 00
Exercise 00: Heap sort	
Turn-in directory : <i>ex00/</i>	
Files to turn in : <code>heapSort.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

Two weeks ago, you helped an auction website.

The database of the auction website was broken.

You managed to save the day, using a large file which contained, for each auction object, its name and its price.

Do you remember how fun it was to sort it ?

Since that day, you can't help but think that you could have sorted it in place.

That's why you decide to sort it again, but this time using a `heap sort` !

Given an array of the following structure:

```
struct s_art {  
    char *name;  
    int price;  
};
```

Implement a `heap sort` to sort the array based on the name of the paintings :

```
void heapSort(struct s_art **masterpiece, int n);
```


The sort must be case-sensitive and in ascending order.



Your function must sort the array in less than 2 minutes.

# Chapter IV

## Exercise 01: Sliding window

	Exercise 01
Exercise 01: Sliding window	
Turn-in directory : <i>ex01/</i>	
Files to turn in : <i>maxSW.c main.c header.h bigo</i>	
Allowed functions : <i>all</i>	
Notes : <i>n/a</i>	

The sales person of the auction website requires your help on a delicate subject.

He wants to plot the curve of his sales for his annual report, but he wonders :

-Wouldn't the curve look a lot smoother if, for each day, I plot the maximum number of sales in the past 4 days? (instead of the actual number of sales for each day)

You have never used sliding windows before, so you gladly write a little algorithm for him.

Given the following structure which contains an array of integers and its size :

```
struct s_max { // just a container for the max sliding window array and its length
    int *max;
    int length;
};
```

And given an array of integers *arr*, its length *n*, and a sliding window of size *k*, implement a function which returns an array containing the maximum in the sliding window for each position from left to right :

```
struct s_max *maxSW(int *arr, int n, int k);
```

Example :

```
Input = {10, -2, 2, 21, -5, 42, 3, -6, 17, 11}, k = 4
```

Sliding window	Max
10 -2 2 21	21
-2 2 21 -5	21
2 21 -5 42	42
21 -5 42 3	42
-5 42 3 -6	42
42 3 -6 17	42
3 -6 17 11	17

```
Output = {21, 21, 42, 42, 42, 42, 17}
```



Your algorithm has to run in  $O(n)$  time, where  $n$  is the number of elements in the array.

Your function has to use a data structure called **deque**. A **deque** is basically a queue where you can push and pop items to the front or to the back.

Here is the definition of the **deque** functions that you must implement:

- **dequeInit()** : Initialize the deque. The first and last pointers are set to NULL.
- **pushFront(deque, item)** : Add an item to the beginning of the deque.
- **pushBack(deque, item)** : Add an item to the end of the deque.
- **popFront(deque)** : Remove the first item from the deque and returns it. If the deque is empty, the function returns INT\_MIN.
- **popBack(deque)** : Remove the last item from the deque and returns it. If the deque is empty, the function returns INT\_MIN.

Given the following structures:

```
struct s_dequeNode {
    int value;
    struct s_dequeNode *next;
    struct s_dequeNode *prev;
};

struct s_deque {
    struct s_dequeNode *first;
    struct s_dequeNode *last;
};
```




Here are the prototypes of the functions defined above:

```
struct s_deque *dequeInit(void);  
void pushFront(struct s_deque *deque, int value);  
void pushBack(struct s_deque *deque, int value);  
int popFront(struct s_deque *deque);  
int popBack(struct s_deque *deque);
```

# Chapter V

## Exercise 02: It's a secret

	Exercise 02
Exercise 02: It's a secret	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <code>createBST.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

Do you remember when you worked in a toy factory?

The boss just called you and needs one last thing from you.

He has on a paper, an array of **ordered** integer, and he would like that from this array, you create a **balanced Binary Search Tree** Toy.

You immediatly wonder why he would want that, he tells you:

**That's a secret!**

Hum ok... Common he has been nice with you, you can do it!

Given the following structure which represents a node of a tree:

```
struct s_node {  
    int value;  
    struct s_node *right;  
    struct s_node *left;  
};
```

And given as parameters an array of integers ordered in ascending order and its size  $n$ , implement a function which returns a **balanced Binary Search Tree**:

```
struct s_node *createBST(int *arr, int n);
```

**Note:** the array will always be **sorted** in ascending order.

Example:


```
input: arr = [1, 2, 3, 4, 5], n = 5
```

```
output:
```

```
    3  
   /\   
  1  4  
   \  \   
   2  5
```

# Chapter VI

## Exercise 03: Junk Food Search

	Exercise 03
Exercise 03: Junk Food Search	
Turn-in directory : <i>ex03/</i>	
Files to turn in : <code>junkFood.c main.c header.h bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

Do you remember when you went to **Paris**? It was like this ...

Once you arrived in **Paris**, you meet with your friend. He looks troubled ... He explains his problem :

Three days ago he was walking quietly in the **Tuileries Garden**, taking advantage of the Parisian life. The weather was good, perfect to eat one of his favorite **cereal bars**! When all of a sudden, while he was enjoying his **cereal bar**, a thief pops up and steals it from him.

Absolutely furious, our friend is about to set up a giant espionage network in all the city and counts on your help to find the culprit ...

You first tell yourself that it is not the end of the world and that instead of having a "**Big Brother**" in Paris it might be wiser to buy a new cereal bar, offer it to your friend and get this over with !

You have an undirected graph that contains the squares of Paris (or Places), some squares contain sellers of cereal bars.

Your goal is to find the closest sellers.

The undirected graph uses the following structures :

```
struct s_node {
    char      *name;
    int       hasCerealBar; //0 = FALSE, 1 = TRUE
    int       visited;      //0 = FALSE, 1 = TRUE
    struct s_node **connectedPlaces;
};

struct s_graph {
    struct s_node **places; //places of Paris
};
```

Where each node represent a Place, and connectedPlaces is a null-terminated array containing pointers to the places it is connected to.

Given the following structure:

```
struct s_sellers {
    int      distance;
    char     **places;
};
```

Implement a function to find the closest squares with a seller of cereal bars from where you are (Place de la Concorde) :

```
struct s_sellers *closestSellers(struct s_graph *parisPlaces, char *youAreHere);
```

The function will return a pointer to a structure with :

- A null-terminated array containing the names of the closest squares (of equal distance from where you are).
- The distance from where you are to the closest squares with a cereal bar seller. The distance between 2 connected squares is 1.

If your current square or a seller could not be found in the graph, the function returns NULL.

Examples, using the file 'main.c', and searching the closest sellers for a place given in the argument:

```
$> compile junkFood.c
$> ./junkFood squares.txt "Place de la Concorde"
At a distance of 1, these places have cereal bars:
Place Maurice-Chevalier
Place Joachim-du-Bellay
$> ./junkFood squares.txt "Place El Salvador"
At a distance of 3, these places have cereal bars:
```

```
Place Constantin-Pecqueur
Place Violet
$> ./junkFood squares.txt "Place de la Fontaine-aux-Lions"
At a distance of 5, these places have cereal bars:
Place Violet
```



You can use the visited field to search through the graph. Don't forget to reset the field to 0 after.

For this exercise, we provide an implementation of a queue that you can use with the following functions :

- `queueInit()` : Initialize the queue. The first and last pointers are set to NULL.
- `enqueue(queue, node)` : Add a node to the end of the queue.
- `dequeue(queue)` : Remove the first item from the queue and return it.

And the following structures:

```
struct s_queueItem {
    struct s_node *place;
    struct s_queueItem *next;
};

struct s_queue {
    struct s_queueItem *first;
    struct s_queueItem *last;
};
```

The functions defined above are declared as follows:

```
struct s_queue *queueInit(void);

void enqueue(struct s_queue *queue, struct s_node *place);

struct s_node *dequeue(struct s_queue *queue);
```

Here is an example of code on how to use it:

```
struct s_graph *graph = getSquares("squares.txt"); // get some Paris places

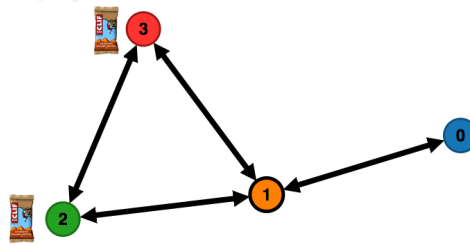
struct s_queue *queue = queueInit(); // queue content: NULL

enqueue(queue, graph->places[0]); // queue content: 'Place de la Contrescarpe'
enqueue(queue, graph->places[1]); // queue content:
                                // 'Place de la Contrescarpe' => 'Place de la Porte-Molitor'

dequeue(queue); //queue content: 'Place de la Porte-Molitor'
```

Examples on smaller graphs :

- Example 1

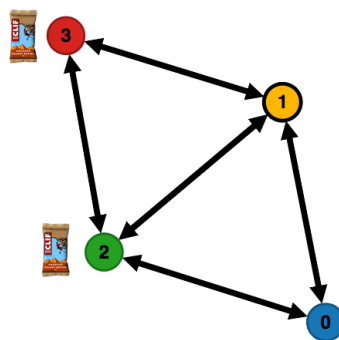


On the above graph, you are at node 0 and there are cereal bar sellers only at nodes 2 and 3.

The output would be :

```
$> ./junkFood example1.txt 'Node 0'
At a distance of 2, these places have cereal bars:
Node 2
Node 3
$>
```

- Example 2




On the above graph, you are at node 0 and there are cereal bar sellers only at nodes 2 and 3.

The output would be :

```
$> ./junkFood example2.txt 'Node 0'
At a distance of 1, these places have cereal bars:
Node 2
$>
```

# Chapter VII

## Exercise 04: Bubble path finder

	Exercise 04
Exercise 04: Bubble path finder	
Turn-in directory : <i>ex04/</i>	
Files to turn in : <code>maxTraffic.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

Now that you have found a **cereal bar**, you can finally enjoy your well deserved vacation.

Unfortunately, your friend has another request before you leave France. A huge **outdoor reggae concert** will be organized in Paris, he sees an opportunity to make some money.

It is well know, during **reggae concert**, reggae fan love to ... **chew bubble gum** (of course !). He wants to set a little bubble gum shop (more of a stand in fact) near the event. In order to sell a maximum number of bubble gums, he wants to know which street it should set it on.

Your mission will be to create a function which will return the **maximum traffic** on one street, depending on the location of the concert.

You have an **undirected graph** which represents the city. Each node is a **square** of this city, each node has an associated **population** (people living in this neighborhood). Each **edge** (arc between two nodes) is considered as a street.



The following structures are used for the graph of Paris places:

```
struct s_node {
    char    *name;
    int     population;
    int     visited; //0 = FALSE, 1 = TRUE
    struct s_node **connectedPlaces;
};

struct s_graph {
    struct s_node **places;
};
```

Implement a function that finds the street with the most important traffic and returns that traffic, given as parameters :

- The undirected graph of places of Paris
- The name of the place where the concert will take place

This function will be declared as follows :

```
float maxTraffic(struct s_graph *parisPlaces, char *eventHere);
```

**Note:** If the place passed as parameter could not be found in the graph, the function returns -1.

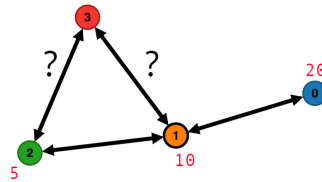
For this exercise you can assume that:

- All the inhabitants of Paris will attend the concert.
- They will take the shortest path.
- If there is more than one shortest path, the traffic will be evenly distributed between the different shortest paths.

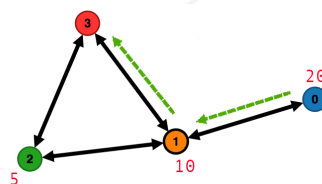
For this exercise, we also provide an implementation of a queue in the file 'main.c'.

## Example 1 :

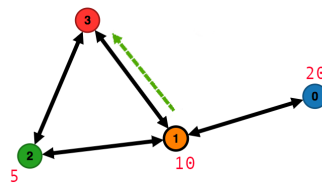
You know that the concert will be organized at the square 3, this square has 2 streets around it. You want to know which one of these streets will have the most traffic.



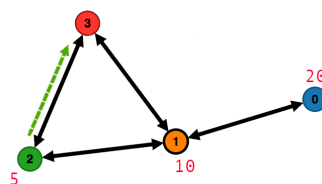
You first find the shortest path from node '0' to '3', which is going through node '1', then node '3':



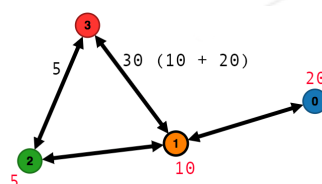
Then you find the shortest path from node '1' to '3':



And then for node '2':



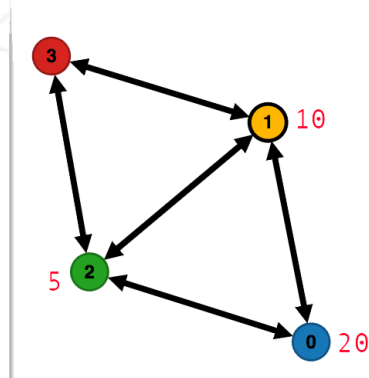
Now, you know which street the population of each node will use:



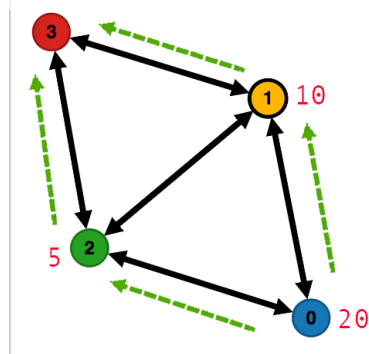
In this case, the function returns 30.

Example 2 :

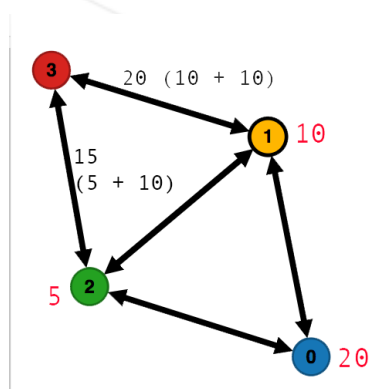
The concert will be organized at node '3':



This time, node '0' has 2 equal paths to go to the node '3'.



So, its population will be evenly distributed (half go to the first path, the second half go to the second path):




In this case, the function return 20.

Expected output:

```
$> compile maxTraffic.c
$> ./maxTraffic
Place du Louvre : 497301.2
Place Pigalle : 449797.3
Place des Invalides : 512979.1
I do not exist : -1.0
$>
```

# Chapter VIII

## Exercise 05: Memory loss

	Exercise 05
Exercise 05: Memory loss	
Turn-in directory : <i>ex05/</i>	
Files to turn in : <code>neverForget.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

Your Grandpa calls and need your help !

Turns out he is really cautious.

He encrypts all his documents with a mono-alphabetic substitution technique. Unfortunately, his memory is very tricky: he can't remember which substitution!

You know what it's like with old people, so you decide to create a function that will decrypt all his documents using only a dictionary of english words.

Implement a function that takes a `text` and a `dictionary` as parameters, and returns the decrypted text.

```
char *neverForget(char *words, char** dictionary);
```

Example:

```
$> cat text1.txt
fhhkqnv zxn whvi fhwwhytd azknj uyiqt fxqvc hx buvi thyo nyhuog igzi ignd xnwzqy vphi aui vhnv kqyvjv
hp fhhkqnv zxn yhi azknj zi ztt fhhkqnv zxn wzjn qy z sqjn mzxqnid hp vidtnv uvqyo zy zxxzd hp
qyoxnjqnyiv qyftujqyo vuoxyz vcqfnv fghfhtzin auuinx cnzyui auuinx yuiv hx jxqnj pxuqiv ign vphiynnv
hp ign fhhkqn wzd jncnyj hy ghs thyo qi qv azknj z onynxzt ignhxd hp fhhkqnv wzd an phxwutzinj igqv
szd jnvcqin qiv jnvfny pphw fzknv zyj hignx vsnninynj axnzjv ign fhhkqn qy ztwhvi ztt qiv phxwv gzv
zazyjhynj szinx zv z wnjquw phx fhgnvqhy szinx qy fzknv vnxmnv ih wzkn ign azvn zv igqy zv chvqvqatn
sgqfg ztthsv ign auaatnv xnvchyvqatn phx z fzknv ptuppqynnv ih auuinx phxw qy ign fhhkqn ign zonyi hp
fhgnvqhy gzv anfhwn vhnv phxw hp hqt hqtv sgnignx ignd an qy ign phxw hp auuinx mnonizatn hqtv hx
tzxj zxn wufg whxn mqvfhuu igzy szinx zyj nmzchxzjn pxnntd zi z wufg gqognx inwcnxziuxn igzy szinx
iguv z fzkn wzjn sqig auuinx hx noov qyvinzj hp szinx qv pzx jnyvnx zpinx xnwzmzt pphw ign hmny
$> compile neverForget.c
$> ./neverForget text1.txt
cookies are most commonly baked until crisp or just long enough that they remain soft but some kinds
of cookies are not baked at all cookies are made in a wide variety of styles using an array of
ingredients including sugars spices chocolate butter peanut butter nuts or dried fruits the softness
of the cookie may depend on how long it is baked a general theory of cookies may be formulated this
way despite its descent from cakes and other sweetened breads the cookie in almost all its forms has
abandoned water as a medium for cohesion water in cakes serves to make the base as thin as possible
which allows the bubbles responsible for a cakes fluffiness to better form in the cookie the agent of
cohesion has become some form of oil oils whether they be in the form of butter vegetable oils or
lard are much more viscous than water and evaporate freely at a much higher temperature than water
thus a cake made with butter or eggs instead of water is far denser after removal from the oven
```

The function returns NULL if:

- The input text contains anything other than letters from 'a' to 'z' or spaces.
- A substitution pattern could not be found with the dictionary provided.