



Starfleet - Day 02

Sort algorithms

Staff 42 pedago@42.fr

Summary: This document is the day02's subject for the Starfleet Piscine.

Contents

I	General rules	2
II	Day-specific rules	3
III	Exercise 00: Stones of the apes	4
IV	Exercise 01: The insertion	6
V	Exercise 02: Make it quick	8
VI	Exercise 03: We need stability	9
VII	Exercise 04: Kitchen utensils	11
VIII	Exercise 05: A monkey stole my watch	13
IX	Exercise 06: The blame machine	16
X	Exercise 07: Youngest prisoner	19
XI	Exercise 08: Granny's game	21
XII	Exercise 09: Which sort algorithm ?	23

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


- For each exercise, 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
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.

Chapter III

Exercise 00: Stones of the apes

	Exercise 00
Exercise 00: Stones of the apes	
Turn-in directory : <i>ex00/</i>	
Files to turn in : sortStone.c main.c header.h bigo	
Allowed functions : all	
Notes : n/a	

As you walk through a zoo, you notice monkeys playing with **pebbles** on the ground.

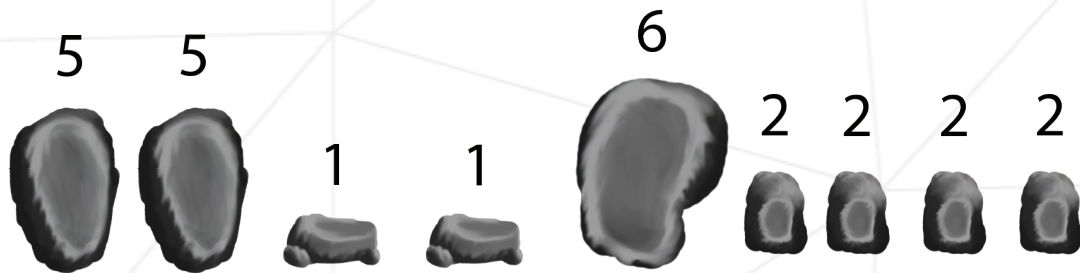
They are putting the pebbles in **line**, grouping them by **size**.

You tell yourself that they are amazing living beings, able to sort stones by their sizes!

However when you approach you noticed they have forgotten a few things:

They have gathered the same size stones **together** but the line of the pebbles is not ordered from the smallest to the biggest. We can see that large pebbles can be placed before medium-sized pebbles.

Representation:



You tell yourself: "How to sort a list in this case in an optimized way?"

You dispose of a singly linked list which represents a stone:

```
struct s_stone {  
    int      size;  
    struct s_stone *next;  
};
```

Your mission is to show these monkeys what you can do by sorting this linked list!

You have to implement the function that can sort a linked list with his first node passed as parameter, in increasing order, by creating an optimized version of the bubble sort.

```
void sortStones(struct s_stone **stone);
```

Note:


- You are authorized to use extra space, and you should.
- You can do tricky things to resolve it, but think simply.
- If a solution is too hard to implement, we advise you to find another method.



If you're thinking of an algorithm which iterate trough each element and doing it in $O(N^2)$ time, as a simple bubble sort, you are wrong!

Chapter IV

Exercise 01: The insertion

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

After this episode with the monkeys (which looked pretty angry when you left them), you end up in these strange times when you are obsessed with a knowledge you want to exploit thoroughly!

You have this need to sort, try out all the sorting algorithms you can and try to learn as much as possible about this notion. (Yes Yes!)

So you tell your loved ones that you are available to help them on any sorting problem.

And so, a **friend** contact you and tell you that she has created a small mobile game and would like you to help her implemented a leaderboard.

There are currently **hundreds of players** on it. The array is not sorted, you think this is the perfect time to implement insertion sort!

Given the following structure:

```
struct s_player {  
    int    score;  
    char   *name;  
};
```

Implement an insertion sort on the null-terminated players array :

```
void insertionSort(struct s_player **players);
```


The resulting array must be in **descending** order (from the greatest score to the lowest score).



For today, take the time to deeply understand how these sorts work, try to find their strength and their weak spot.

Chapter V

Exercise 02: Make it quick

	Exercise 02
Exercise 02: Make it quick	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <code>quickSort.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

After a few weeks, your friend calls you: It's terrible! Her game has had a **great success** these last days! There are **hundreds of thousands** of players who have joined the game. The sorting algorithm takes too much time now every time.

You think that it would require an algorithm that would fit for a **very large unsorted table** ...

Hmmm ... Would it be possible to sort it quickly? Yes with a **quicksort**!

Given the following structure:


```
struct s_player {  
    int    score;  
    char   *name;  
};
```

You have to implement a sort algorithm named **quicksort**:

```
void quickSort(struct s_player **players);
```

Chapter VI

Exercise 03: We need stability

	Exercise 03
Exercise 03: We need stability	
Turn-in directory : <code>ex03/</code>	
Files to turn in : <code>mergeSort.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

You are quite satisfied with this new sorted table, however your friend is not at all!

Indeed, she noticed that, at the end, the time order was lost in the resulting array. That is, if there were 2 players who had exactly the same score, the one who had it first would not necessarily be in before of the one who had it in second.

The quickSort is not a stable algorithm (same for insertion sort). It did not maintain index associations.

You think it is time to make a fast algorithm, which is capable of sorting without losing the order of identical elements:

It's a great time to code a merge sort!

Given the following structure:


```
struct s_player {  
    int    score;  
    char   *name;  
    char   *timeStamp;  
};
```

You have to implement the comparison-sort algorithm named `merge sort`.

```
struct s_player **mergeSort(struct s_player **players);
```

Chapter VII

Exercise 04: Kitchen utensils

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

This sorting obsession is not ready to go away, you know, in your heart, that sorting algorithms are **important!**

On day, while you are in a kitchen, you realize that there are **forks, knives, spoons** and a **dozen different utensils** in your drawer that are not sorted.



You have fun organizing the utensils by number: fork = 1, spoon = 2, ... whip = 15.

You are asking yourself the following question:

What is the most suitable sorting algorithm for such a small range?

You think that this is the perfect time to implement `count sort`!

Given as parameter an array of integer named `utensils`, and his associated `length`,


Implement the non-comparison sort named count sort :

```
void countSort(unsigned char *utensils, int n);
```



Chapter VIII

Exercise 05: A monkey stole my watch

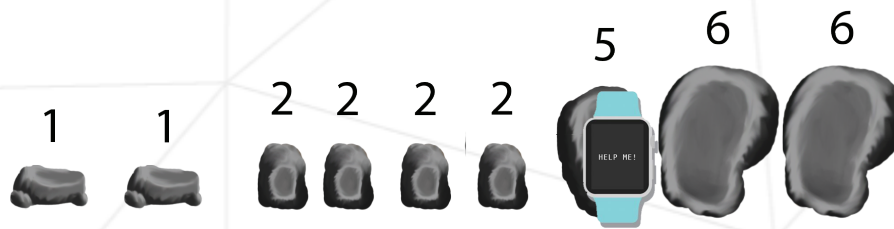
	Exercise 05
Exercise 05: A monkey stole my watch	
Turn-in directory : <code>ex05/</code>	
Files to turn in : <code>findShifted.c main.c header.h bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

One day, you decide to go back to the ZOO. You especially want to see how the **monkeys**, that you helped last time, are going now.

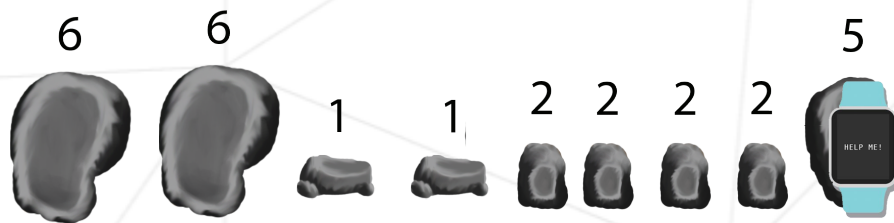
At their spot, you notice that they have learned what you taught them: **the rocks** are completely ordered!

At the moment they see you, they begin to become crazy! They come at you and **steal your watch**. They put it **near** one of the ordered rocks, then they decide to shift their rocks twice to the right!

Reconstitution:



Then they shift their rocks **twice** to the right:



You take back the watch and **run out** of the ZOO as fast as you can, leaving behind you, these terrible monkeys...

At your home, you think about your last interaction with the monkeys,

you wonder: how would I implement the '**watch on the shifted rocks**' thing?

Giving an array of **sorted** integers named '**arr**' that has been shifted an unknown number of times, **length** the length of the array, and an integer **value**, implement the following function that searches in the array for the **value** and returns the **index** of the first value encountered.

```
int searchShifted(int *rocks, int length, int value);
```

If the value could not be found in the array, the function returns -1.


Your algorithm must have an **average** runtime of $O(\log(n))$, with n the number of elements in the array (even though worst case is still $O(n)$).

Examples:

```
$> compile findShifted.c
$> ./findShifted
Rocks : 11 11 11 4 4 4 4 5 7 11 11 11 11
Value 4 at index 3
$> ./findShifted
Rocks : 8 8 8 8 8 8 8 9 9 1 1 1 1 1 1 5 5 5 5 5 5 5 5 5 5 5
Value 6 at index -1
$>
```


Chapter IX

Exercise 06: The blame machine

	Exercise 06
Exercise 06: The blame machine	
Turn-in directory : <code>ex06/</code>	
Files to turn in : <code>findPotentialCriminels.c main.c header.h bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

By a series of unfortunate events, you find yourself stuck in the past in 1947 (ah... Why have you listened to this `old mad scientist?`).

You have to get out of there. You know that the only way is to `find this mad scientist`.

You know that at that time he was a `young and frisky Frenchman`, but you do not know his name and only have a photo portrait of the time.

You also know that he spent some time in a prison in France.

You say to yourself: "Yes! I will use `the blame machine!`"

This machine is in fact a large perforated card sorter. Each perforated card contains information about a former / current prisoner. After the war, the French used it to identify those who during the war, committed a crime whose names are unknown but whose physical descriptions has been reported.

Once arrived on site, the machine is `out of order!`

You absolutely must find that person. You take out your laptop and decide to scan all the perforated cards.

You know that this machine uses a `radix sort`, you tell yourself that this is the perfect time to implement this sort!

The physical description includes 7 criteria :

```
gender:      height (cm):  hairColor:  eyeColor:
0 - male    0 - 140-149    0 - black  0 - amber
1 - female  1 - 150-159    1 - brown  1 - blue
            2 - 160-169    2 - blond  2 - brown
            3 - 170-179    3 - auburn 3 - gray
            4 - 180-189    4 - red    4 - green
            5 - 190-199    5 - gray   5 - hazel
            6 - 200-209    6 - white  6 - red
            7 - 210-219
            8 - 220-229

glasses:    tattoo:      piercing:
0 - false   0 - false   0 - false
1 - true    1 - true    1 - true
```

Based on these criteria, a description integer is calculated. It has at most 7 digits :

```
description =
gender * 10^6
+ height * 10^5
+ hairColor * 10^4
+ eyeColor * 10^3
+ glasses * 10^2
+ tattoo * 10^1
+ piercing * 10^0
```

Given a null-terminated array of criminals with their name et integer description, implement 3 functions :

- `getDescription(info)`, which returns the integer description given the complete description contained in the `s_info` structure.
- `sortCriminals(criminals)`, which sorts the array based on the integer description in an increasing order. You have to implement a **radix sort**.
- `findPotentialCriminals(criminals, info)`, which search through the sorted array for criminal corresponding to the given description. This must be done using binary search (average runtime in $O(\log n)$).

These functions use to following structures :

```
struct s_info {
    int gender;
    int height;
    int hairColor;
    int eyeColor;
    int glasses;
    int tattoo;
    int piercing;
};

struct s_criminal {
    char *name;
    int description;
};
```

And they must be declared as follows :

```
int getDescription(struct s_info *info);  
void sortCriminals(struct s_criminal **criminals);  
struct s_criminal **findPotentialCriminals(struct s_criminal **criminals, struct s_info *info);
```

Example:

If you are looking for a man with a height of approximately 175cm, gray hair, brown eyes, glasses and no tattoo/percing, the description integer would be 352100.


```
$> gcc -o findPotentialCriminals findPotentialCriminals.c main.c  
$> ./findPotentialCriminals 352100  
352100 Bearl Brown  
352100 Millard Rochet  
352100 Clemmie Berre  
352100 Lorn Pena  
$>
```



The most important thing here is to implement a radix sort.

Chapter X

Exercise 07: Youngest prisoner

	Exercise 07
Exercise 07: Youngest prisoner	
Turn-in directory : <code>ex07/</code>	
Files to turn in : <code>printYoungest.c</code> <code>main.c</code> <code>header.h</code> <code>bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

You finally managed to come back to the present!

Thanks to the Dolorea... Hum I mean the time machine that the scientist that you have found has given to you.

Back in your era, you realize that you still have the datas of all the prisoners that you got from the blame machine.

You now wonder : "Who was the youngest prisoner ?"

Given an unordered array of integers which represents the age of the prisoners, find the ages of the youngest one, and print it !

You could do it using a modified count sort, but you want do it without creating another array.

As a personal challenge,

you decide to search through the array using the divide and conquer approach!

Here is the prototype of the function:

```
void printYoungest(int *ages, int length);
```

Example:


```
$> compile printYoungest.c
$> ./printYoungest
Ages : 103 33 104 14 17 103 70 104 71 60
Youngest : 14
$> ./printYoungest
Ages : 83 54 100 79 16 10 21 70 100
Youngest : 10
$>
```



This function has to be implemented using a divide and conquer approach, don't forget it!

Chapter XI

Exercise 08: Granny's game

	Exercise 08
Exercise 08: Granny's game	
Turn-in directory : <code>ex08/</code>	
Files to turn in : <code>externalSort.c main.c header.h bigo</code>	
Allowed functions : <code>all</code>	
Notes : <code>n/a</code>	

One Sunday afternoon, Granny offers you a tea break. Sitting in her living room, a green tea in hand, you noticed that this invitation hides something: **She still needs your help!**

You learn that she is **addicted** to the **pinball game** on her computer and that she has done tens of thousands of games.

There is thus a file that stores all her personal scores, however this file has never been sorted.

She would like you to **sort this file by score**.

With all that you have just done recently, you think that it will be easy!

However, there is far too much to sort out so that you can sort them at once on her computer (which has a ram of **8kb**), you will have to find another way. Why not an **external sort**?

Implementing an **external sort** is not an easy task.

It should be done in 2 steps :

- Reading the file by **blocks** of a specific size. For each block, the elements are sorted and written to a **temporary file**.

- Merging all of the temporary files into one final sorted file.

For both steps, the space used mustn't exceed the capacity of the ram. For this exercise, the ram limitation only applies to the number of elements you can read at once (for instance, 8kb => 2000 integers). The algorithm can use additional space as long as it is constant space.

Implement the function `externalSort` declared as follows :

```
void externalSort(char *scoreFile, char *sortedFile, int ram);
```

Where :

- `scoreFile` is the name of the file to be sorted (here, "grannyScores.txt").
- `sortedFile` is the name of the complete sorted file to be created (here, "grannyScores-Sorted.txt").
- `ram` is the ram in bytes.

In the file "main.c" provided in the resources, some utility functions are implemented. We recommend that you use them :

```
FILE *openFile(char *filename, char *flag);  
int readBlock(FILE *f, int *arr, int size);  
void writeFile(char *filename, int *arr, int n);  
int fileIsSorted(char *filename);
```


- `openFile(filename, flag)` : open the file which name is passed as parameter. You can specify if you want to read ("r") or write ("w") with the `flag` parameter. A `FILE` pointer is returned.
- `readBlock(f, arr, size)` : read up to `size` integers in the file pointed by `f`, and stock them in the array `arr`. The number of read integers is returned.
- `writeFile(filename, arr, n)` : write `n` intergers from the array `arr` to the file named `filename`.
- `fileIsSorted(filename)` : check if the file named `filename` is sorted. 1 is returned if the file is sorted, 0 otherwise.



The choice to use the utility functions or not is up to you.
However, we recommend that you do !

Chapter XII

Exercise 09: Which sort algorithm ?

	Exercise 09
Exercise 09: Which sort algorithm ?	
Turn-in directory : <i>ex09/</i>	
Files to turn in : answers.txt	
Allowed functions : all	
Notes : n/a	

We think, if you have come to this point, you have done enough sort for today!

This exercise is then a little different:

We are going to present you some situations, you will have to select which sort algorithm you would choose for each situation.

- Given a large file with millions of data (name and phone number), what is the most efficient way to sort the array?
- Given an array of Ascii characters, what's the most efficient way to sort it?
- Let's say you have a very large array of strings and you just have enough RAM to put it on the RAM. Which sort algorithm would you choose?
- Given an almost sorted array of 10k elements. Which sort algorithm would you choose?
- Given an array which is almost sorted, and only one element is not at his good place, Which sorting algorithm would you choose?



Turn in your answers in a file named "answers.txt". You only need to give the name of the algorithm, you will explain why during the evaluation.