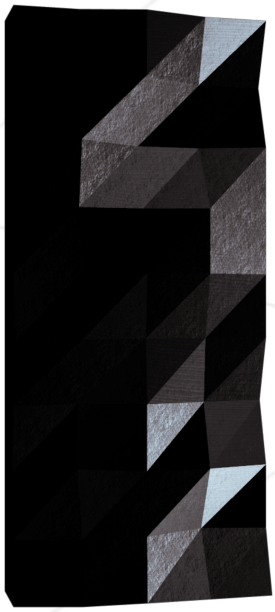


Starfleet - Day 06

Recursive n°2 & Greedy Algorithms

Staff 42 pedago@42.fr

Summary: This document is the day06's subject for the Starfleet Piscine.

Contents

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 explictly 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.

Chapter III

Exercise 00: Distance and wifi hotspots

	Exercise 00
Exercise 00: Distance and wifi hotspots	
Turn-in directory : $ex00/$	
Files to turn in : `probaDistance.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

One day, you decide to go with your friends at `Las Vegas`! The city where everything is possible.

So, as everything is possible, you decide to invest in the community by participating in the `city's improvement` project.

Indeed, the city wishes to add some wifi hotspots on the famous `Las Vegas Strip`.

The city has chosen arbitrarily different locations, for the first tests, they have decided to install 2 wifi hotspots randomly on 2 of the chosen locations.

The engineers in charge of the project are now wondering the probability that these 2 wifi terminals are at 30 feet distance to each other?

Implement a function that, given an array of integers where each elements is one of the arbitrary locations, returns the probability that if we put `randomly` two wifi terminals on two of these positions, the distance between the two is greater than a given `distance` in feet.

Given `locations`, an array of positive integers, and `distance`, an integer. Find the probability that the distance between 2 locations will be above the parameter `distance`.

```
double probaDistance(int dist, int *locations, int n);
```

Comprehension example:

With the locations , what is the probability that the distance between two terminals selected randomly will be above 5 feet?

It will be 1/3. Because only if we create hotspot on pos 10 and pos 20, we will have a distance greater than 5 feet.

Example:

```
$> compile probaDistance.c
$> # with the distance 5 and {10 , 15 , 20} elements of the locations array
$> ./probaDistance 5 10 15 20
0.333333
$> ./probaDistance 1 10 15 20
1.000000
$> ./probaDistance 10 10 15 20
0.000000
$> ./probaDistance 10000 `cat strip.txt`
0.302572
```

No need to gamble or party when you can do some probability algorithms right?

Chapter IV

Exercise 01: Range and wifi hotspots

	Exercise 01
Exercise 01: Range and wifi hotspots	
Turn-in directory : *ex*01/	
Files to turn in : `hotspots.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

The city has now laid all the wifi `hotspots` on the Las Vegas strip, however these hotspots were laid down quite quickly and they did not take into account that some of them have `different ranges`.

So you can have wifi hotspots that `spreads` in the `same place`!

The city wants to get `rid` of all wifi hotspots that are useless.

It thus asks you to implement an algorithm capable of selecting the wifi hotspots that the city will keep.

Given the following structure:

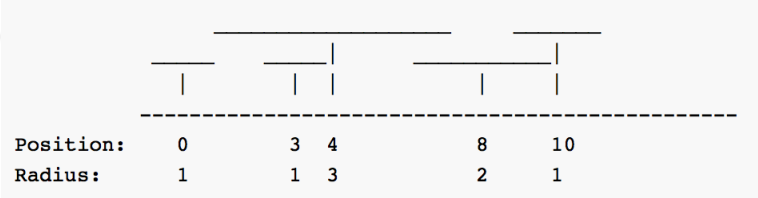
```
struct s_hotspot {
        int pos;      // position of the hotspot
        int radius;   // radius of the range of the hotspot
};
```

Given as parameters the `hotspots` array, create a function able to select the maximum number of wifi hotspots, which ranges are not overlapping.

```
int selectHotspots(struct s_hotspot **hotspots);
```

Your algorithm has to run in O(N) time, where N is the number of hotspots.

Example 1:



In this example, we would select only three hotspots.

```
$> compile hotspots.c
$> ./hotspots hotspots1.txt
(INFO) Loading the file... finish!
Number of hotspots : 3
$>
```

Example 2:

```
$> ./hotspots hotspots2.txt
(INFO) Loading the file... finish!
Number of hotspots : 6
$>
```

Chapter V

Exercise 02: Roulette probability

	Exercise 02
Exercise 02: Roulette probability	
Turn-in directory : $ex02/$	
Files to turn in : `roulette.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

One day, seized by the temptations of `Las Vegas`, you decide finally to go inside a casino, inside it, you see the pizzeria boss, in front of a `french roulette`, yelling:

-Listen up everyone! I will `win 500$` at the roulette in `less than 3 games`. I will `begin at 100$` and always bet on the `red color`!

Now you wonder what is the actual probability of his prediction?

Before implementing the function, here is some rules about the french roulette that will interest us:

- A wheel has `37` pockets : 18 red, 18 black and 1 green.
- You can bet on color `red`, `black`, `green`.
- If you bet on the `red`, and the ball `falls` into one of the `red` pockets, you win `2 time` what you have bet. if it goes on the `black`, you `loose all` you have bet. If it falls on the `green` pocket, you will loose only `half` of what you have bet.

Here are some considerations to solve the exercise:

- Your friend will always invest `all` he has won. For example if he bets 100$ and wins 200$, the next game he bets 200$. If he has 0$, he stops playing.
- If your friend wins more than what he was actually predicted, he will stop playing. For example, if he wants to win 300$ in 3 game, and in 2 games he has reached 300$ or above, he won't play the third game.

Implement a function that returns a double between 0 and 1 which is the probability of winning more than 'dollarsWanted' dollars in 'nbGame' games with a first bet of 'initDollars':

```
double probabilityWin(double initDollars, int wantedDollars, int nbGame);
```

You have to use recursion to solve this problem. :-)

Examples:

```
$> compile roulette.c
$> ./roulette
Usage: ./roulette initDollars desiredDollars nbGame
$> ./roulette 5 10 1
0.486486
$> ./roulette 5 10 2
0.486486
$> ./roulette 5 10 3
0.492883
$> ./roulette 100 500 3
0.115136
```

Drawing on paper a tree of probability may help you understand the problem.

Be careful, the house always wins!

Chapter VI

Exercise 03: Wheel of fortune

	Exercise 03
Exercise 03: Wheel of fortune	
Turn-in directory : $ex03/$	
Files to turn in : `minPersons.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

One day, you are walking inside the casino,

you just see a room full of people who are `laughing` and `yelling`.

Intrigued, you go inside the room and discover that there is a `wheel` on which they can bet :

There are `50` numbers on the wheel. All of the people around it are betting on a number, then the dealer spins the wheel and the people who bet on the `right` number wins.

Oh the wheel just stopped spinning, and 2 people got the right number this time!

You wonder now: how many `people` do we need inside the room to have, with 70% of probability, two people choosing the same number?

Given as parameters the `number` of elements present on the wheels (here 50), and the probability percentage. Implement a function which returns the `minimum number` of people who are required to bet.

```
int                minPersons(int elements, int minPercentage);
```

Examples:

```
$> compile minPersons.c
$> #the minimum number of people required to make the following statement true:
$> #2 people choosing same the number in a range of 50 numbers, with 70% chance.
$> ./minPersons 50 70
12
$> ./minPersons 10 50
5
$> ./minPersons 20 50
6
```

Chapter VII

Exercise 04: Knight out

	Exercise 04
Exercise 04: Knight out	
Turn-in directory : $ex04/$	
Files to turn in : `knightOut.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Back from Las Vegas, you decide to take a break. Be more `peaceful`. You invite your nephew to play some `chess`.

The thing is that your nephew doesn't know how to play chess (he is more a fan of `Snake, Ladders and Mirages`!).

You basically show him how `every pieces work`, then comes a time where you show him how the `knight` works. He asks you: if I move the knight on top just here, the knight goes out, yes?

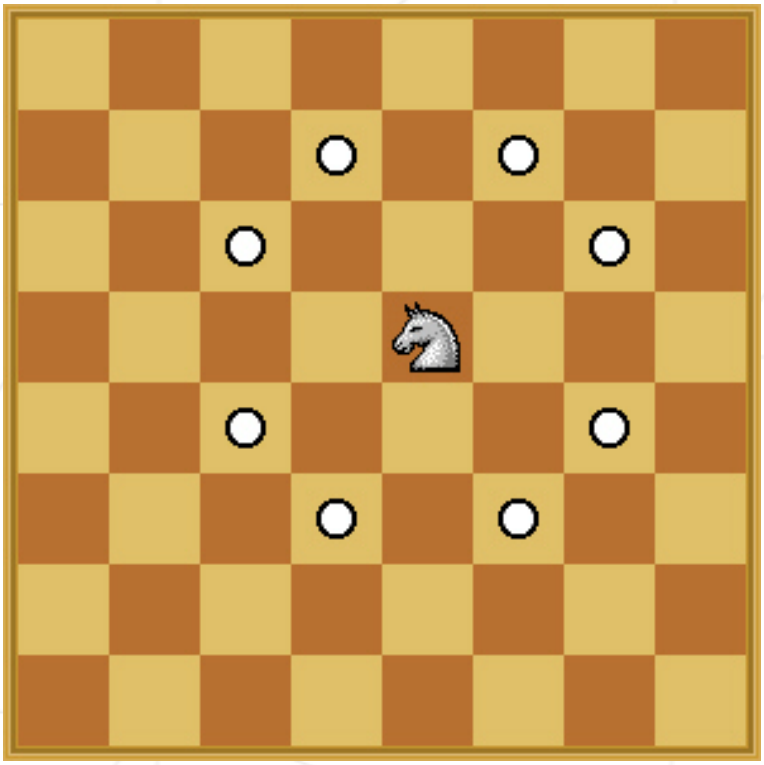
You show him that the knight can't get out of the board, like every pieces.

However this question makes you ask yourself:

If I put a `knight` at a given position, what is the probability that after `n` moves, it gets out?

Here are the information you need to determine that probability:

- At each move, the knight picks one of the eight directions uniformly at random (possibly a direction which makes the knight leave the chess board).
- Once the knight leaves the board, it can't enter it again.



The `8 x 8` chess board is represented by an uint64_t.

An empty chess board with only 1 knight at the position `i = 3` and `j = 4` (like in the picture above) will be represented by an uint64_t where only the bit at position `28` (= i * 8 + j) is set.

You must implement 2 functions:

- `getInitialPos(board)` : return the position of the `only set bit` in an uint64_t `board`. If there is less or more than 1 bit set, the function returns `-1` because the input is invalid.
- `knightOut(board, n)` : return the `probability` that the knight gets out of the `board` after `n` moves. The parameter `board` is an uint64_t representing the chess board with the knight at its initial position. If at least one of the 2 parameters is invalid, the function returns `-1`.

These functions must be declared as follows :

```
int getInitialPos(uint64_t board);
double knightOut(uint64_t board, int n);
```

Examples :

```
$> compile knightOut.c
$> ./knightOut
usage: ./knightOut i j n
$> ./knightOut 3 4 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Probability knight out in 1 move(s) : 0.0000000000
$> ./knightOut 3 4 5
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Probability knight out in 5 move(s) : 0.6443481445
$> ./knightOut 0 0 1
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Probability knight out in 1 move(s) : 0.7500000000
$> ./knightOut 1 2 10
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Probability knight out in 10 move(s) : 0.9477851503
```

Your algorithm has to run in less than a second for any n < 10^4, you must use dynamic programming !

Chapter VIII

Exercise 05: I'm tired of these monkeys

	Exercise 05
Exercise 05: I'm tired of these monkeys	
Turn-in directory : $ex05/$	
Files to turn in : `helpNephew.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

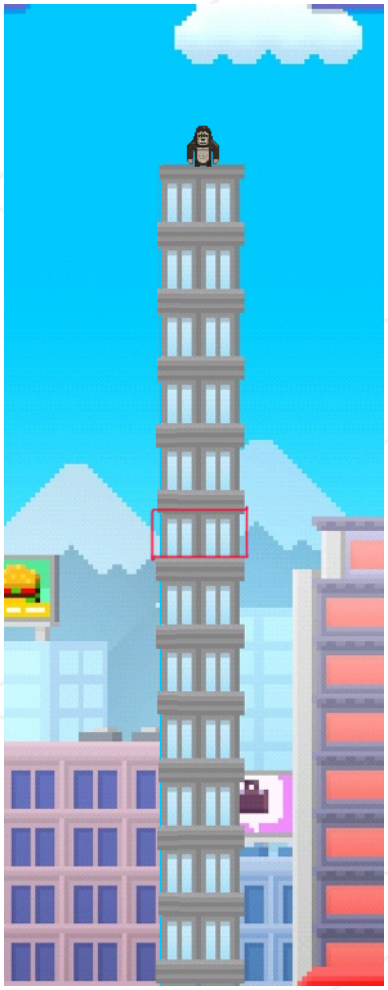
Oh no! You went again to the `zoo`, and as always, you tried to impress some monkeys.

One thing leading to another, the bigger of these monkeys, called 'King Kong', has escaped from the zoo and is now on the top of the building where your nephew lives.

The problem is that this 'King Kong', just by giving a punch on the floor, can break this floor, and in his fall he can randomly break the floor of some other stairs. Everyone has escaped from the building, but remains only your `nephew` who is on the 5th floor and doesn't want to go out because he is playing some `chess` that you taught him.

`King Kong` tells you that he is going to give `2 punches` (yes, you speak the language of monkeys).

You want to save your nephew, but you are a `little lazy` so you tell yourself that if the probability that "the monkey do 3 punches and doesn't reach your nephew's stair is above 50%, it will be find!



Given the number of `stairs`, the number of punches `King Kong` is going to do, and the stair where is your nephew, implement a function able to return the probability that king kong doesn't destroy the floor where your nephew is.

```
double        probaNephewWillLive(int nStairs, int nPunch, int nephewStair);
```

Comprehension example:

Let's says there are 5 stairs, `King Kong` is on the roof and your nephew is hiding in stair 2. `King Kong` will do one punch, there is equally 1/5 chance that every stairs get touched.

- if `King Kong` will go on floor 5 or 4 or 3, the nephew floor isn't reach, so your nephew is safe.
- But if he go on floor 2 or floor 1, your nephew floor will be destructed.

So, for one punch, there is 60% chance that the nephew will survive.

Now what if `King Kong` is doing 2 punches?

- on the first punch, if `King Kong` reach floor 2, 1, then for the second punch there is no chance that your nephew survive (because his floor has already been destructed).
- but on the first punch, if `King Kong` reached floor 5, 4 or 3, then for the `second` punch, there is respectively 2/4, 1/3 and 0/2 chance that your nephew will survive.

So, for 2 punch, there is a around 16.6% of chance that the nephew stairs isn't reached.

Examples:

```
$> compile helpNephew.c
$> ./helpNephew
Usage: ./helpNephew nStairs nPunch nephewStair
$> ./helpNephew 5 1 1
0.6
$> ./helpNephew 5 2 1
0.17
$> ./helpNephew 30 3 5
0.20
```

Chapter IX

Exercise 06: Spanning monkey

	Exercise 06
Exercise 06: Spanning monkey	
Turn-in directory : $ex06/$	
Files to turn in : `printMST.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Now that this terrible 'King Kong' `came down`, he is now going to `visit` every place of the city...

The police has shot him with a `tranquillizer` gun. It should have put him to sleep, but they just `swapped` the `sleeping` serum with another `strange` serum...

Now 'King Kong' is just visiting the city in a strange manner :

It visits every place only once, by taking always the `shortest path`.

You noticed that his `movement` are `familliar` to you:

`King Kong` is creating a `minimum spanning tree (MST)`!

The Police now wants your help to give them the futur place where `King Kong` will go.

Given a `connected and undirected graph` of the City places, using the following structures :

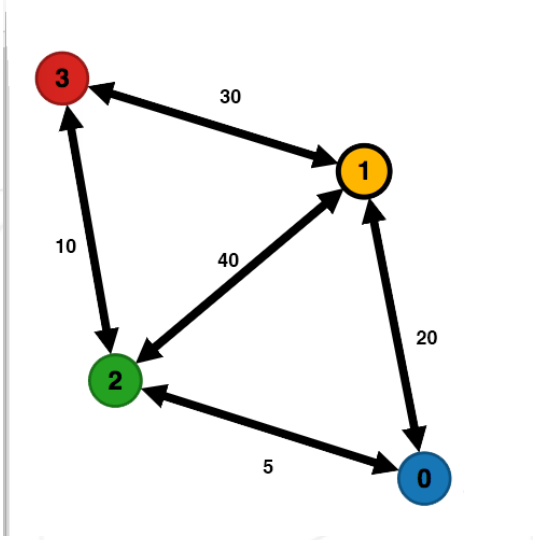
```
struct s_node {
        int id;                 // index of the node in the graph
        char *name;             // name of the node
        struct s_edge **edges; // null-terminated array of edges
};

struct s_edge {
        int dist;               // distance to the destination node
        struct s_node *dest;    // destination node
};

struct s_graph {
        struct s_node **nodes; // null-terminated array of
};                          // all the nodes in the graph
```

Implement a function which find a `minimum spanning tree` using a `greedy approach`, and prints the edges of the MST and their distance :

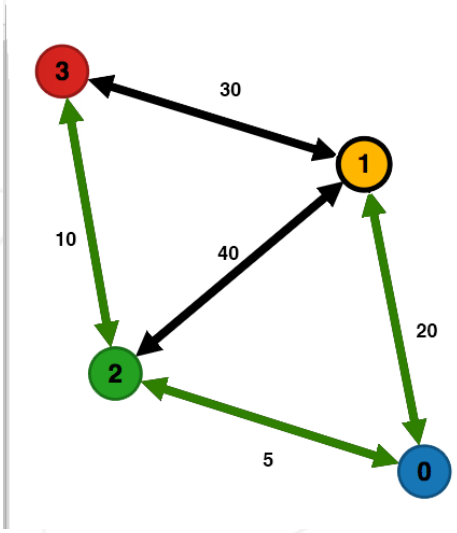
```
void printMST(struct s_graph *graph);
```

Comprehension examples:

`Example 2` :



The MST would be :




```
$> compile printMST.c
$> ./printMST example2.txt
Distance : Edges
      20 : Node 0 - Node 1
       5 : Node 0 - Node 2
      10 : Node 2 - Node 3
$>
```

Other examples :

```
$> ./printMST example1.txt
Distance : Edges
      20 : Node 0 - Node 1
      10 : Node 3 - Node 2
      30 : Node 1 - Node 3
$> ./printMST example3.txt
Distance : Edges
       4 : Node 0 - Node 1
       8 : Node 1 - Node 2
       7 : Node 2 - Node 3
       9 : Node 3 - Node 4
       4 : Node 2 - Node 5
       2 : Node 5 - Node 6
       1 : Node 6 - Node 7
       2 : Node 2 - Node 8
$>
```

The display is up to you, as long as you display the edges and their distance.