Starfleet - Day 05

Recursive programming

Staff 42 pedago@42.fr

Summary: This document is the day05'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
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.

- For this day, you must write `less than 21 lines of codes` for each exercise. It has to be clean and easily understandable code (one instruction per line). A line of code is a line inside a function. An empty line does not count. As for the previous days, the norm does not apply.

- For today, you have to use `recursion` for each exercise, even if you don't think you need to :)

Chapter III

Exercise 00: Pizza slices

	Exercise 00
Exercise 00: Pizza slices	
Turn-in directory : *ex*00/	
Files to turn in : `possibleSlices.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Today, the weather is perfect, you are in New York, in the middle of Central Park. Ah ... What a beautiful city New York. It's gigantic, beautiful, cosmopolitan, the food is wonderful ... Oh, speaking of food: you are hungry.

You decide to go to the nearest pizzeria.

Once inside, you notice that there are strange pizzas: these pizzas are rectangular, and on each pizza there is a label that indicates the size of the pizza.

A pizza of size 3 can be cut into 3 slices of size 1. It can also be cut into a slice of size 2 and a slice of size 1, or simply be sold in one slice of size 3.

These combinations are interesting to you, they even obsesses you. Too bad for the pizza, you take out your computer and decide to make an algorithm able to generate all combinations for a given size.

`Using recursion`, create a function able to print all possible slices of a pizza, given an integer `pizzaSize` which is the size of the pizza.

```
void printPossibleSlices(int pizzaSize);
```

Be careful, today you must write less than 21 lines of codes for each exercise and also use recursion. See the day specific rules for more information!

Examples:

```
$> compile possibleSlices.c
$> ./possibleSlices 3
Slices with a pizza of size 3:
[3]
[2, 1]
[1, 2]
[1, 1, 1]
$> ./possibleSlices 2
Slices with a pizza of size 2:
[2]
[1, 1]
```

The printing order is the not important.

For this exercise we provide the data structure `dynamic array` of integers, that you can use if you want.

Here is the definition of the function that we provide:

- `arrayInit()` : return an empty new allocated array.
- `arrayAppend(array, number)` : add the number into the array.
- `arrayClone(array)` : return a copy of the array.
- `arrayPrint(array)` : print the array.

Given the following structure:

```c
struct s_array {
        int *content;
        int length;
        int sum;
        int capacity;
};
```

Here are the prototypes of the functions defined above:

```
struct s_array *arrayInit(void);
int     arrayAppend(struct s_array *arr, int number);
struct s_array *arrayClone(struct s_array *arr);
void    arrayPrint(struct s_array *array);
```

You can get the sum of all elements in the array by accessing to the variable `sum` of the structure.

Example using the `dynamic array`:

```
struct s_array *arr;

arr = arrayInit();

arrayAppend(arr, 15);
arrayAppend(arr, 10);

arrayPrint(arr); //print '[15, 10]\n'

printf("length: %d, sum: %d\n", arr->length, arr->sum); //print 'length: 2, sum: 25\n'
```

The bocal declines all responsibility for any pizza craving.

Chapter IV

Exercise 01: Pizza Tech

Exercise 01
Exercise 01: Pizza Tech
Turn-in directory : *ex*01/
Files to turn in : `bestPrice.c main.c header.h bigo`
Allowed functions : `all`
Notes : `n/a`

The boss of the pizzeria saw you look into the problem, and we can say one thing: he is quite impressed!

He comes kindly to see you, and offers you a part of pizza. You are a little suspiscious, but you are mostly hungry so you accept to spend the meal with him.

During the meal, you are talking about New-York, the weather, anything in fact! Then the boss brings the conversation to what really interests him: he wants you to be his new `Pizza Tech`! In fact, the job consists in giving you some special problems to solve.

You like the challenge so you accept immediatly.

The first problem is that he wants to `make more money`, and for that, he has an idea...

He gives you a price list of every sizes of pizza slice he's currently selling:

```
$> cat list.txt
1: 1.5$
2: 3.4$
3: 4.8$
4: 6.2$
5: 8.2$
...
```

He would like that, with this list, you create an algorithm able to return the best price he can get for a pizza of any size (best price == make the most money).

For example for a pizza of size 3, the best combination would be to have a slice of size 2 and a slice of size 1 (4.9$ in total).

Implement a function that takes the size of the whole pizza and an array of double `prices`, which contains the prices for each slice size:

```
double bestPrice(int pizzaSize, double *prices);
```

`Note`:

You have access to the price of a slice size using the size as the `index`, example:

```
// get the price for a slice of size 1:
printf("%.2f\n", prices[1]); // print '1.50\n'
```

The length of `prices` will always be >= `pizzaSize`!

Chapter V

Exercise 02: Pizza Festival

	Exercise 02
Exercise 02: Pizza Festival	
Turn-in directory : *ex*02/	
Files to turn in : `bestPriceV2.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

The program is working very well. However the boss of the pizzeria will soon participate in the New York City Pizza Festival and he plans to cook pizzas of gigantic size.

Unfortunately your program is taking hours to be able to handle these large sizes.

You need to modify your program to be able to get the best price in a better delay.

Here is the prototype:

```
double optimizedBestPrice(int pizzaSize, double *prices);
```

Example:

```
$> compile bestPriceV2.c
$> ./bestPriceV2 99
(INFO) Loading the file... finish!
99 : 187.87
```

The above example returns the best price of a `pizzaSize` of `99`, in less than a second!

Chapter VI

Exercise 03: Sims Island

	Exercise 03
Exercise 03: Sims Island	
Turn-in directory : *ex*03/	
Files to turn in : `sinkIsland.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

After this episode at the pizzeria, you come home to rest, you then turn on your computer to play a cool simulation game in which you can control a set of islands.

If you're tired of an island, you have the power to sink the island by just clicking on a point on the island.

You find this feature fabulous and now you wonder: how would I programatically do that?

Given as parameters a matrix of integer `map`, an integer `row` and an integer `column`,

implement the following function able to sink an island:

```
void sinkIsland(int **map, int row, int col);
```

`Note` to solve the exercise:

- On the matrix, to know if you are on the last row, the next row is `NULL`.
- On the matrix, to know if you have reached the end of a column, the next element is `-1`.
- An island on the map is a group of touching 1s.
- Two points that touches on a diagonal are not considered as an island.

`Graphical` example:

```
map =  0 0 0 0
       1 1 0 0
       1 1 1 0
       0 1 1 0
       0 1 0 0
       0 0 1 1

row = 1
col = 1
       col=1
         |
         v
       0 0 0 0
       1 1 0 0 <-- row=1
       1 1 1 0
       0 1 1 0
       0 1 0 0
       0 0 1 1

sinkIsland(map, row, col)

print map

       0 0 0 0
       0 0 0 0
       0 0 0 0
       0 0 0 0
       0 0 0 0
       0 0 1 1
```

Chapter VII

Exercise 04: Scrabble

	Exercise 04
Exercise 04: Scrabble	
Turn-in directory : *ex*04/	
Files to turn in : `permutation.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

While playing a game of Scrabble in Bryant Park with an elderly person,

you have a list of characters, but you have no idea what word you can form with it.

This game is going to be hard!

Happily, your opponent, taken by a brutal sleepy feeling, falls asleep.

It's a perfect time to find the word that you weren't able to find!

You turn on your laptop, and begin to make a little program that generates all combinations of a string...

Implement a function that print all `permutations` of a string given as parameter.

```
void        printPermutations(char *str);
```

Examples:

```
$> compile permutation.c
$> permutation ab
ab
ba
$> permutation abc
abc
bac
bca
acb
cab
cba
$> permutation aba
aba
baa
baa
aab
aab
aba
```

Chapter VIII

Exercise 05: I wasn't cheating !

Exercise 05
Exercise 05: I wasn't cheating !
Turn-in directory : *ex*05/
Files to turn in : `permutationV2.c main.c header.h bigo`
Allowed functions : `all`
Notes : `n/a`

-Not at all sir! I wasn't cheating, I was creating an algorithm!

The gentleman noticed your algorithm, and decide to encourage you in the process.

Actually, he would like to use your program for his next games!

However, he noted that you displayed multiple times the same combination when a character repeats.

He wonder if you can do it without `duplicates`!

Rewrite an algorithm, able to print all the `permutations` of a string given as parameter, without duplicates:

```
void        printUniquePermutations(char *str);
```

Example:

```
$> compile permutationV2.c
$> ./permutationV2 aba
aba
aab
baa
```

For this exercise, we provide an `hash table`, that you can use if you want.

Here is the definition of the hash table functions that we provide:

- `dictInit(capacity)` : `Initialize` the hash table given the capacity of the array.
- `dictInsert(dict, key, value)` : `Insert` an item in the hash table given its key and value.
- `dictSearch(dict, key)` : `Search` an element in the hash table given the key. If not found, return `-1`.

Given the following structures:

```
struct s_item {
  char          *key;
  int                value;
  struct s_elem *next;
};

struct s_dict {
  struct s_item **items;
  int           capacity; //the capacity of the array 'items'
};
```

Here are the prototypes of the functions defined above:

```
struct s_dict *dictInit(int capacity);
int        dictInsert(struct s_dict *dict, char *key, int value);
int        dictSearch(struct s_dict *dict, char *key);
```

Chapter IX

Exercise 06: Make them eat, please PizzaTech !

	Exercise 06
Exercise 06: Make them eat, please PizzaTech !	
Turn-in directory : *ex*06/	
Files to turn in : `makeThemEat.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

The boss of the pizzeria, still needs his favorite PizzaTech: indeed he has a mission for you!

In a week, a dancing evening is organized in his restaurant.

He sent a form to all the groups of participants asking them how long they would take to eat.

He has a dozen tables. He would like to know if, with these tables, he can theoretically receive all groups in a total of 3 hours.

Given the following structure:

```c
struct s_people
{
        char *name;
        int time; //in minute
};
```

Implement a function that take as parameters an `array of people`, the number of tables available, and the total time in `minutes`,

This function returns `1` if it's possible to make all of them eat, or `0` if it's not possible:

```
int isPossible(struct s_people **people, int nbTable, int totalTime);
```

For example, if there are 3 groups of people:

- `Alain`, eats in 30min.
- `Patrick and his buddy`, eats in 60min.
- `Phillipe`, eats in 30min.

There are 2 tables,

Is it possible in 60 minutes to make them eat all?

Yes, by doing the following steps:

- minute 00 : put `Alain` on the first table, `Patrick's group` on the second table.
- minute 30 : Alain leave the first table and Phillipe takes it, `Patrick's group` is still on the second table.
- minute 60 : Phillipe leaves the first table, `Patrick's group` leaves the second table. All the groups have eaten!

Using the main, and its `parser` function that parses a file passed as parameter (here `guestList.txt`), here is an example:

```
$> cat guestList.txt
Alain: 30min
Patrick and his buddy: 60min
Phillipe: 30min
$> compile makeThemEat.c
$> ./makeThemEat 2 60
(INFO) Loading the file... finish!
It is possible !
$>
```

Chapter X

Exercise 07: Palindrome

Exercise 07
Exercise 07: Palindrome
Turn-in directory : *ex*07/
Files to turn in : `findPalindrome.c main.c header.h bigo`
Allowed functions : `all`
Notes : `n/a`

As you stroll in the streets in the middle of New York. A man a little badly dressed and smelly approaches you. This gentleman has a rather strange question:

-I have a word which is originally a `palindrome`. I then add some characters at a random position in the word, you have to find the original word. If you do not find this word you must give me a penny, what do you say boy?

You accept with enthousiasm.

The first sequence he tells you is:

-"krayaok", there is 2 extra characters.

You answer him easily: `kayak`!

The man, without any reaction, looks at you and then tells you:

- All right! This time I give you a series of 15.001 characters, there is a palindrome inside if you delete 4242 characters, it's up to you to find it!

His look, malicious, clearly indicates `'give me a penny, it will be faster!'`.

However you do not let yourself get fooled, you take out your laptop and start making a small program, able to find the answer!

Given a string `sequence` and an integer `nDelete` which is the number of characters to delete,

find in `sequence` the palindrome by deleting `nDelete` characters.

```
char *findPalindrome(char *sequence, int nDelete);
```

Example :

```
$> compile findPalindrome.c
$> ./findPalindrome krayaok 2
kayak
$>
```

Chapter XI

Exercise 08: The SML game

	Exercise 08
Exercise 08: The SML game	
Turn-in directory : *ex*08/	
Files to turn in : `minimumMoves.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

One day, as you are babysitting your nephew, you ask him to play your `Snakes And Ladders` game!

However, he laughs at you, your game is out of date.

In fact, a new board game has just been released, it's called `"Snakes and Ladders and Mirages"`.

It's the same as `Snake and Ladders`, but it has a special feature: `Mirages`.

A `Mirage` is in fact another `board`, close to the current `Board`, which can be accessed by `snakes` or `ladders`.

When you have a simple `Snakes and Ladders` game, the `'random'` pointer can bring you to any cell on the board.

Here is a simple example, a snake goes from node `'4'` to node `'2'`:

```
1() -> 2( ) -> 3() -> 4(:) -> 5()FINISH
        ^                |
        |                |
        |________________|
```

But with `SLM`, a snake can now go to a cell on one of the `Mirages`!

Here, a snake goes from node '4' of the current board to node '3' of a mirage:

```
1() -> 2( ) -> 3() -> 4(:) -> 5()FINISH
                        |
                        |
                   v____|
   1() -> 2() -> 3( ) -> 4()
```

On a `SLM` game, there are multiple `Mirages` and each cell of a `Mirage` can also have `Snakes` and `Ladders`!

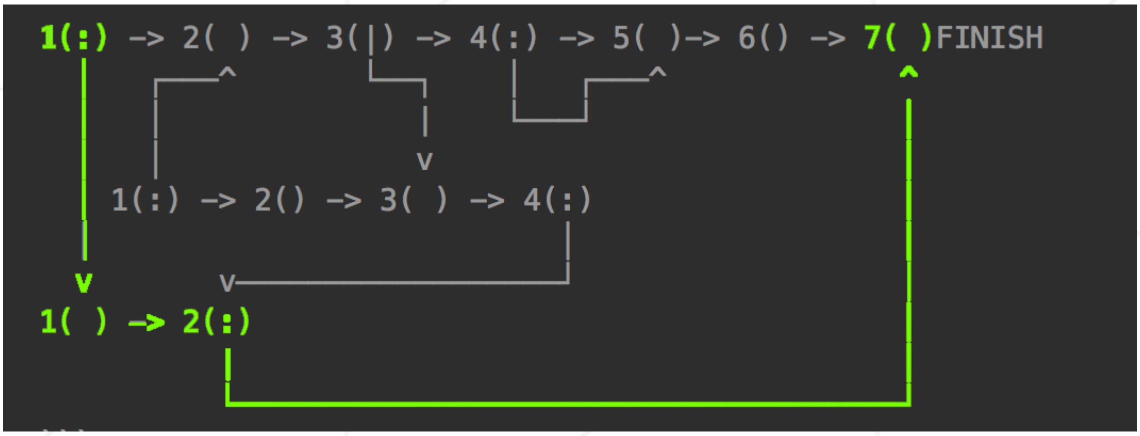
The goal of this game is to get as fast as possible to the end of the first board.

An example that represents a `Snakes and Ladders and Mirages` board:

```
1(:) -> 2( ) -> 3(|) -> 4(:) -> 5( )-> 6() -> 7( )FINISH
 |    ┌──^      └──┐    |   ┌──^              ^
 |    |            |    └───┘                 |
 |    |            v                          |
 | 1(:) -> 2() -> 3( ) -> 4(:)                |
 |                         |                  |
 v         v───────────────┘                  |
1( ) -> 2(:)                                  |
         |                                    |
         └────────────────────────────────────┘
```

On the example above, the shortest path, starting from the first cell of the main Board, will be:

```
1() -> Mirage 2 : cell 1() -> 2() -> Main Board : 7() FINISH.
```



You are convinced that an AI can defeat all humans at this game, so you decided to implement it!

Given the following structure:

```
struct s_node {
        int value; //the value of the cell
        int isFinal; //tell if this node is the last node of the first board, 0 = FALSE, 1 = TRUE
        struct s_node *random;
        struct s_node *next;
};
```

Given as parameter only the first cell of the main board, implement a function that returns the minimum number of moves to get to the to `finish cell`!

```
int minimumMoves(struct s_node *node);
```

We also provide a `parse()` function in the file `main.c`, which reads, parses and transforms a file into a `SLM Board game`!

The extension of the file for this game is `'.slm'` Here is a sample of a SLM file:

```
b
1() -> 2() -> 3() -> 4(b.1) -> 5(m1.4) -> 6(m2.1) -> 7() -> 8() -> 9()

m1
1() -> 2(m2.1) -> 3() -> 4(b.9)

m2
1() -> 2() -> 3()
```

With 'b' the main board, 'm1' as mirage 1, 'm2' as mirage 2 etc...

Inside the parenthesis '()', you can tell where the random pointer should point.

Example: if you have '(m1.2)', the random pointer will point to the `Mirage 1` on the node 2.

> There can be a large number of mirages, so be careful!

For this exercise, we provide a `stack`, a `queue` and an `hash table`, that you can use if you want.

Here is the definition of the `stack` functions that we provide:

- `init()` : Initialize the stack. The top pointer is set to `NULL`.
- `pop(stack)` : Remove the top item from the stack and return it. If the stack is empty, the function returns `NULL`.
- `push(stack, item)` : Add an item to the top of the stack.

Given the following structures:

```
struct s_stackNode {
  void *content;
  struct s_stackNode *next;
};

struct s_stack {
  struct s_stackNode *top;
};
```

Here are the prototypes of the functions defined above:

```
struct s_stack *init(void);
void *pop(struct s_stack *stack);
void push(struct s_stack *stack, void *content);
```

Here is the definition of the `queue` functions that we provide:

- `init()` : Initialize the queue. The first and last pointers are set to `NULL`.
- `enqueue(queue, item)` : Add an item to the end of the queue.
- `dequeue(queue)` : Remove the first item from the queue and return it. If the queue is empty, the function returns `NULL`.

Given the following structures:

```
struct s_queueNode {
        void *value;
        struct s_queueNode *next;
};

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

Here are the prototypes of the functions defined above:

```
struct s_queue *queueInit(void);
void *dequeue(struct s_queue *queue);
void enqueue(struct s_queue *queue, void *value);
```

Here is the definition of the hash table functions that we provide:

- `dictInit(capacity)` : `Initialize` the hash table given the capacity of the array.
- `dictInsert(dict, key, value)` : `Insert` an item in the hash table given its key and value.
- `dictSearch(dict, key)` : `Search` an element in the hash table given the key. If not found, return `-1`.

Given the following structures:

```
struct s_item {
  char          *key;
  int                value;
  struct s_elem *next;
};

struct s_dict {
  struct s_item **items;
  int           capacity; //the capacity of the array 'items'
};
```

Here are the prototypes of the functions defined above:

```
struct s_dict *dictInit(int capacity);
int        dictInsert(struct s_dict *dict, char *key, int value);
int        dictSearch(struct s_dict *dict, char *key);
```