

Data Structures and Algorithms – CS2402

Midterm 1 – Solution

50 points - 50 minutes

Make sure that you justify all your answers. Two points will be given for the clarity of presentation of your answers.

Exercise 1 (7 points) Determine the running time function and the corresponding big-Oh notation of the following fragment of code:

```
int sum=0;
for (int i = 0; i < n; i += 2)
    for (int j = 0; j < i; j += 1)
        sum++;
```

Solution.

Let us first rewrite the fragment of code as follows:

```
// Part 1:
for (int i = 0; i < n; i += 2)
    my_procedure;
```

where my_procedure is defined as follows:

```
// Part 2:
for (int j = 0, j < i, j = ++)
    sum++;
```

The time complexity of my_procedure / Part 2 is $2 + 3i$. Therefore, p , the time complexity of my_procedure, depends on i , not on n : $p(i) = 2 + 3i$.

Let's now consider Part 1.

int i=0; **1 step**

per valid loop:

$i < n$ 1 step

$i += 2$ 1 step

my_procedure; $p(i)$ step

= $2 + p(i)$ steps per valid loop

$n/2$ valid values for i

we consider all values of i

= $(2 + p(0)) + (2 + p(2)) + (2 + p(4)) + \dots + (2 + p(2 \cdot (\lceil n/2 \rceil - 1)))$

= $\sum_{i=0}^{\lceil (n/2) \rceil - 1} (2 + p(2i))$

exit condition, when $i \not< n$ **1 step**

TOTAL: $2 + \sum_{i=0}^{\lceil (n/2) \rceil - 1} (2 + p(2i))$ steps

Now let us replace $p(i)$ by its value $2 + 3i$ inside the time complexity of Part 1.

$$f(n) = 2 + \sum_{i=0}^{\lceil n/2 \rceil - 1} (2 + p(2i)) = 2 + \sum_{i=0}^{\lceil n/2 \rceil - 1} (2 + 2 + 3 * 2.i) = 2 + \sum_{i=0}^{\lceil n/2 \rceil - 1} (4 + 6i)$$

$$\begin{aligned} \sum_{i=0}^{\lceil n/2 \rceil - 1} (4 + 6i) &= 4 + 6 \times 0 + \\ &4 + 6 \times 1 + \\ &4 + 6 \times 2 + \\ &\dots \end{aligned}$$

and since we have:

$$\begin{aligned} &4 + 6 \times (\lceil n/2 \rceil - 1) \\ &= \frac{4 \times \lceil n/2 \rceil + 6 \times (0 + 1 + 2 + \dots + (\lceil n/2 \rceil - 1))}{2} \\ &= 4 \times \lceil n/2 \rceil + 6 \times \sum_{i=0}^{\lceil n/2 \rceil - 1} i \\ &= 4 \times \lceil n/2 \rceil + 6 \times \frac{(\lceil n/2 \rceil - 1) \lceil n/2 \rceil}{2} \end{aligned}$$

And the big-Oh of this function is $\mathcal{O}(n^2)$.

Exercise 2 (5 points) Determine the running time function and the corresponding big-Oh notation of the following fragment of code:

```
for (int i = 1; i ≤ n2; i* = 3)
    my_procedure;
// where my_procedure is known to run in running time p(n)
```

Solution.

int i=1;	1 step
per valid loop:	
$i \leq n^2$	1 step
$i* = 3$	1 step = $2 + p(n)$ steps per valid loop
my_procedure;	$p(n)$ step × $\log_3(n^2)$ valid values for i = $\log_3(n)(2 + p(n))$ steps
exit condition, when $i \not\leq n^2$	1 step

TOTAL: **$2 + \log_3(n)(2 + p(n))$ steps**

And the big-oh of this running time function is $\mathcal{O}(p(n) \times \log(n))$.

Exercise 3 (12 points)

- Describe the Josephus problem (2 points).
- Explain why you can solve it using a linked-list, and also using an array (2 points).
- Discuss the pros and cons of each (2 points).

- Give and justify the complexity of the method using an array, and then using a linked-list, when the number of people is N , and you eliminate the n^{th} person each time (6 points).

Solution.

- The Josephus problems consists in considering N persons, placed in a circle. Starting at one (randomly chosen) person of the circle, you traverse the circle until you reach the n^{th} person, and you eliminate this person. You keep on repeating the same traversal and removal of people from the circle, until there is only one person left in the circle. This person wins.

So N and n are the two parameters of this problem.

- Using a linked-list, each node of the linked-list will contain the information of one person in the circle. Adjacent nodes stand for adjacent persons in the circle. Traversing the circle translates, in a linked-list, into traversing a list, which comes at a linear cost. Removing a person will be easy because you just have to reassign links so that you “skip” one node of the list (and then, through garbage collection, the disregarded node will eventually be deleted).

Using an array, you have to make an array of size N . Each cell contains the information of a person in the circle. Adjacent cells stand for adjacent persons in the circle. Traversing the circle to reach the n^{th} person after the starting one (at position $\text{array}[\text{index}]$), will come down, in an array, to access the $\text{array}[\text{index} + n - 1]$ or, if $\text{index} + n \geq N$, $[(\text{index} + n - 1) \bmod N - 1]$. This is done in constant time (the nice thing about arrays :-). Removing a person from the circle (say person: $\text{array}[\text{index}]$) amounts to shifting all persons after index in the array ($N - \text{index} - 1$ persons, and in the worst case N) (this is the bad thing about arrays... :-).

- As mentioned before,
 - pros of a list: easy removal of a person (constant time $\mathcal{O}(1)$)
 - cons of a list: expensive access to a person (linear time $\mathcal{O}(n)$)
 - pros of an array: easy access to the n^{th} person (constant time $\mathcal{O}(1)$)
 - cons of an array: expensive removal of a person (linear time $\mathcal{O}(N)$)
- Complexity using a linked-list: You have to access and remove successively ($N - 1$) person. As explained before, (access + removal) costs $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$. Therefore adding up to: $(N - 1)\mathcal{O}(n) = \mathcal{O}(nN)$.

Complexity using an array: You have to access and remove successively ($N - 1$) person. As explained before, (access + removal) costs $\mathcal{O}(1) + \mathcal{O}(N) = \mathcal{O}(N)$. Therefore adding up to: $(N - 1)\mathcal{O}(N) = \mathcal{O}(N^2)$.

Exercise 4 (24 points)

1. What is a stack? describe the data structure. (4 points)
2. Describe Depth-First Search. (3 points) You can use examples to help you describe the method.
3. Give the algorithm (pseudo-code) for DFS. And explain how/why you use a stack to implement DFS. (6 points)
4. Suppose you implement DFS to find a solution to a constraint problem. Does the method you've just described (in question 4.3) allow to return all solutions? (6 points)
 - (a) If so, 1. explain why, and 2. explain how to modify your algorithm to return only the first solution found.
 - (b) If not, 1. explain how to modify your algorithm so that it returns all solutions, and 2. explain why this modified algorithm actually returns all solutions.
5. What is the time complexity of DFS, if d_{max} is the size of the longest branch in your search tree, and b the maximum number of alternatives given at each choice point? (5 points)

Solution.

- A stack is a list-based ADT, with the specific property that the only of its elements that can be accessed is the last element that was stored. Such a data structure is also called: Last-In / First-Out.

This data structure has the following operations:

- push()
- pop()
- peek() (or top())
- isEmpty()
- isFull() (in case of a limited capacity stack)
- clear()

as described in class.

- DFS (Depth-First Search) is a searching algorithm whose strategy is to go in depth first.

Consider a problem where you have several parameters x_1, \dots, x_n , and alternative assignments/instantiations for each x_i of them $a_1^i, \dots, a_{n_i}^i$. Then when an alternative a_j^i is considered for a parameter x_i , instantiations of the instantiation of the other parameters will be considered before to consider the alternative assignments for x_i .

This comes down to make a decision and follow it until either evidence is that you've found a solution (in which case you can stop), or that the complete assignment is not a solution (in which case you go back to the last parameter you assigned and look for other alternative assignments: this process is also called backtracking).

- The pseudocode for DFS is as follows:

```
stack S = new stack();
push(R,S); // where R is the root of your problem
while (S is not empty){
    P=top(S);
    pop(S);
    if (P is a solution)
        then return P and stop
    else if (P has children) // i.e., if there are still decisions to make
        then push(all children of P, S);
}
```

- The above-described algorithm returns the first solution found and stops. In order to return all solutions, you can:

- either store them in some container and output them at the end of the process;
- or output them each time you find a solution and yet not return anything so that the process continues.

- The time complexity of DFS is as follows: at worst you will traverse all the nodes of your tree. Therefore, you will have to traverse $b^{d_{max}}$.

The space complexity is the most space you need at each time of your process. The most space you need is when you are at a leaf and that you have all alternatives stored:

- to reach a leaf, you've had to make at most d_{max} decisions;
- each time you've made a decision, you've had to store at most $b - 1$ alternatives;

As a result, the maximum space you need is $(b-1)d_{max}$, whose big-oh is $\mathcal{O}(b \cdot d_{max})$.