Algorithm Design Exercise Sheet 1

Course of Engineering in Computer Science, Academic Year 2020/2021

Riccardo Salvalaggio 1750157 & Lorenzo Scarlino 1761016

December 5, 2020

1 Exercise 1.

In order to guarantee required time complexity, we opted for a Dynamic Programming approach, that consists in the following steps:
1. The main problem is divided into a series of overlapping subproblems;
2. The results of subproblems are saved in an auxiliary data structure (the exercise doesn't require space constraints)
3. Recursively solutions of smaller subproblems ar re-used to build up solutions to larger subproblems until the main one is solved.

1.1 Exercise 1.1

Algorithm 1 Longest Palindromic Substring

```
procedure LONGEST_PALINDROMIC_SUBSTRING($word$)
    $matrix \leftarrow$ word.length x word.length matrix, all elements equals to 0
    Set to 1 main diagonal of matrix                          ▷ every char is palindromic of itself
    $max\_length \leftarrow 1$                                ▷ length of current longest palindromic substring
    $start\_index \leftarrow 0$                               ▷ start index of current longest palindromic substring
    for $elem$ in $matrix$ do
        if $elem == elem.next$ then $elem.next \leftarrow 1$  ▷ pal. substrings of length 2
            $max\_length \leftarrow 2$, $start\_index \leftarrow elem\_index$
    for $k \leftarrow 3$ to word.length do                    ▷ pal. substrings of length k
        for $i \leftarrow 0$ to word.length-$k$+1 do $j \leftarrow i + k - 1$
            if $matrix[i+1][j-1] == 1$ and $word[i] == word[j]$ then
                $matrix[i][j] \leftarrow 1$, $max\_length \leftarrow k$, $start\_index \leftarrow i$
    return word substring from $start\_index$ to $start\_index + max\_length$
```

The most complex operation is the double for loop which takes O(n^2), so the Longest Palindromic Substring algorithm meets the given constraints.

1.2 Exercise 1.2

Algorithm 2 Longest Palindromic Subsequence

```
procedure LONGEST_PALINDROMIC_SUBSEQUENCE(a)
    $b \leftarrow a$.reversed          ▷ looking for longest common subsequence between a and a.reversed
    $matrix \leftarrow$ (a.length+1) x (b.length+1) matrix, all elements equals to 0
    for $i \leftarrow 0$ to a.length+1 do   ▷ in each cell is set length of local longest pal. subsequence
        for $j \leftarrow 0$ to b.length+1 do
            if $i == 0$ or $j == 0$ then $matrix[i][j] \leftarrow 0$
            else if a[$i$-1] == b[$j$-1] then $matrix[i][j] \leftarrow matrix[i-1][j-1] + 1$
            else $matrix[i][j] \leftarrow$ max($matrix[i-1][j]$,$matrix[i][j-1]$)
    $i \leftarrow$ a.length, $j \leftarrow$ b.length, $aux \leftarrow$ empty string
    while $i > 0$ and $j > 0$ do   ▷ starting from the highest value we add characters to final string
        if a[$i-1$] == b[$j-1$] then $aux$.append(a[i]), $i \leftarrow i-1$, $j \leftarrow j-1$
        else if $matrix[i-1][j] > matrix[i][j-1]$ then $i \leftarrow i-1$
        else $j \leftarrow j-1$
     return aux
```

The most complex operation is the double for loop which takes O(n^2), so the Longest Palindromic Subsequence algorithm meets the given constraints.

2 Exercise 2.

Given the problem of finding a seating arrangement for every participant to the event, it is possible to solve it by defining a flow network N and finding the maximum flow in N. First we have to create the digraph G(V,E), with V = "Participants (Founders or Inverstors)" and E = "Good Pairs". Mathematically, V = I ∪ F and E ⊆ I×F.
In order to meet the constraints on the preferences of every participant, we need to impose some requirements:
1. ∀ v∈V, K = { e(f,i) ∈ E : v=i ∨ v=f }, K ⊆ E, |K| ≥ 2, i.e. every participant must appear at least into two good pairs e1,e2 ∈ E
2. |F| = |I|
3. |V| ≥ 4
Given G, we can model the problem as a circulation one, with demands d(v) and nonnegative capacities.
This function must satisfy the following conditions:
a) For each e ∈ E: 0 ≤ f (e) ≤ c(e) (capacity)
b) For each v ∈ V: $\sum_{in} f(e)$ - $\sum_{out} f(e)$ = d(v) (conservation)
In order to solve the problem we build the graph as it follows:
1. Add source node s with demand -|V| (supply node), i.e. d(s) = -|V|.
2. Add sink node t with demand |V| (demand node), i.e. d(t) = |V|.
3. Set every v ∈ V as a transshipment node, i.e. d(v) = 0 ∀ v∈V.
4. For every Founder node f∈F add an edge from source node s to f with capacity c(s,f) = 2, since each founder cannot seat near to more than two investors.
5. For every Investor node i∈I add an edge from i to sink node t with capacity c(i,t)=2, since each investor cannot sit near to more than two founders.
6. For every Good Pair e(f,i)∈E, add an edge from f to i with capacity c(f,i) = 1.
Since every participant v∈V must find a seat, the maximum flow has to be equal to |V|. Ford-Fulkerson augmenting path algorithm computes the maximum flow and solves the seats arrangement (if there exists). Ford-Fulkerson algorithms starts from s with a random edge, looks for a path to t, calculates the maximum residual edge, tries to add it to an edge and starts with another one from s, until it gets stuck. At the end of the computation, if the result is equal to |V|, then there exists a feasible seating arrangement, otherwise the problem is unfeasible.
In the way the digraph G has been defined, every possible breaking case in which the problem is unfeasable is covered. It is possible to prove the correctness of the construction basing the discussion on two fundamental definitions:
1. The definition of Ford-Fulkerson algorithm in relation with augmenting path theorem tells that if there are no augmenting paths the current flow is the max flow: in our formulation it means that all the edges from investors nodes to t are saturated, so every investor has exactly two neighbours; this also implies that the ingoing flow in t is equal to outgoing flow in s (flow conservation).
2. The definition of circulation with demands and capacity and conservation.
In fact the flow network described above complies with the following theorems:
- "G has circulation iff G' has max flow of value D = $\sum_{d(v)>0} d(v)$ - $\sum_{d(v)<0} -d(v)$".This is satisfied by adding source and destination nodes which respectively provide the global ingoing and outgoing flow of the network.
- Given (V, E, c, d), there does not exist a circulation iff there exists a node partition (A, B) such that $\sum_{v \in B} d(v) > cap(A, B)$. This requirement' is fulfilled by assigning the right value to edge capacities.

3 Exercise 3.

In order to determine whether a list of projects is suitable for a programmer with current score C, we thought to a greedy algorithm that returns an answer in O(n logn). The greedy approach consists in building up the solution incrementally, myopically optimizing some local criteria.

Algorithm 3 Divide And Order

```
1:  procedure DIVIDE_AND_ORDER(proj_list)
2:      pos_list ← []                                   ▷ List of projects with positive bonus
3:      neg_list ← []                                   ▷ List of projects with negative bonus
4:      for i ← 0,i<length of proj_list,i ← i + 1 do
5:          if bonus of proj_list[i] ≥ 0 then
6:              append proj_list[i] to pos_list
7:          else
8:              append proj_list[i] to neg_list
        merge_sort(pos_list,credit_score,asc)           ▷ sort by ascending credit score
9:      merge_sort(neg_list,credit_score + bonus,desc)  ▷ sort by descending credit score + (negative) bonus
10:     ord_list ← pos_list :: neg_list                 ▷ Concatenation between pos_list and neg_list
         return ord_list                                ▷ Returns projects list in suitable order
```

Algorithm 4 Execution

```
1:  procedure EXECUTION(proj_list,C)
2:      ord_list← Divide_And_Order(proj_list)
3:      for i ← 0,i<length of ord_list,i ← i + 1 do
4:          if C < credit score of ord_list[i] then
5:              return False                            ▷ List of projects is not suitable
6:          else
7:              C ← C + credit score of ord_list[i]
        return True                                     ▷ List of projects is suitable
```

It is possible to calculate computational time complexity for every piece of code: - dividing the list of projects takes O(n) time;
- merge sort algorithm to sort the arrays takes O(n log n) time;
- concatenation of neg_list with pos_list takes O(n);
- execution of ordered projects takes O(n) time.
Global computational time complexity of the algorithm is O(n log n).
The fundamental thought behind the logic of the algorithm is the following: by sorting projects with positive bonus by ascending credit score it's obvious that the first project in the list is the easiest one, so if the programmer is not able to execute it, he won't be able to execute any other project. After the execution of positive bonus projects, the programmer will have the highest credit score possible and he will be in the best condition to perform projects with negative bonus, which are sorted by descending credit score + (negative) bonus, so he won't get stuck with them (it's the opposite logic with respect to the case of projects with positive bonus).

4 Exercise 4.

4.1 Exercise 4.1

Every candidate has a different impact on Sars-CoV-2 virus, and in mathematical terms this can be defined by assigning a different value to each cure (the higher is the value, the lower is the minimum number of doses which kill the virus) and by choosing a numeric value as the objective (i.e. reaching objective implies that the virus has been defeated).

Algorithm 5 Best Candidate

```
1: procedure BEST_CANDIDATE($array$,$candidates\_list$, $objective$, $d$)          ▷ array ← 1 to d
2:     $units$ ← empty array with $candidates\_list$ length
3:     for $i \leftarrow 0$,$i$< candidates_list length,$i \leftarrow i+1$ do
4:         $units[i]$ ← Binary_Search($array$, $candidates\_list[i]$, $objective$, $d$)  ▷ it returns the minimum
   dose of cure i which kills the virus
5:     $minimum\_dose \leftarrow units[0]$
6:     for $j \leftarrow 1$,$j$< candidates_list length,$j \leftarrow j+1$ do
7:         if units[i]<minimum_dose then
8:             $minimum\_dose \leftarrow units[i]$
9:             $winner\_candidate \leftarrow i$
10:    return minimum_dose,winner
```

Binary_Search is an algorithm based on the 'divide-et-impera' logic. Starting from given d value and an array composed by natural numbers from 1 to d, the algorithm search the minimum value (based on candidate value) by recursively halving the array until it finds it (if it does exist) in O(log d) time (in his worst case).
Best_Candidate algorithm instead executes Binary_Search for every candidate (n times), then compares computed outcomes to each other in order to return the best global cure, so it has O(n*log d) complexity.

4.2 Exercise 4.2

Algorithm 6 Randomized Best Candidate

```
1: procedure RANDOMIZED_BEST_CANDIDATE($array$, $candidates\_list$, $objective$, $d$) ▷ array ← 1 to
   d
2:     $best\_vax$, $min\_doses$ ←Randomized_ Binary_Search($array$, $candidates\_list[i]$, $objective$, $d$)   ▷
   candidate and min dose used as current minimum.
3:     while |candidate_list| > 1 do
4:         if min_doses - 1 applyed at last vaccine of the list kill the virus then
5:             $last\_vaccine\_cure$ ←Randomized_Binary_Search(array,candidates_list[last],$objective$, $d$)
6:             $best\_vax \leftarrow last\_one$
7:         else
8:             remove last item from the candidates_list
9:     return best_vax
```

The algorithm is based on a randomized version of binary search (it randomly selects the recursive partition of the array).
Starting from a random vaccine, Random_Best_Candidate finds its a_i and looks for a better candidate by periodically testing other cures with a_i-1 vials. Candidates which are not effective with that amount of vaccine are discarded, while an eventual more efficient one become the current best solution.
This procedure on average discards n/2 vaccines for every iteration, so it can be said that the algorithm finds the best possible cure in $\Theta(n + logd * logn)$ tests (on average).