Algorithm Design 2020/2021

Exercise Sheet 1

Sultan Umarbaev, Matricola: 1954544

Deadline: 5/12/2020

1 Exercise 1

1.1 Part 1. Algorithm

The suggested algorithm is based on bottom-up Dynamic Programming (DP) approach by solving all the related subproblems first and storing their results. Define boolean matrix $P[n, n]$, where $P[i, j] = true$ if substring $w(i, .., j)$ is a palindrome; otherwise $P[i, j] = false$, for $0 \leq i \leq j \leq n$ with $n = \text{length}(w)$ and $left$ starting index of palindrome. First, consider 2 base cases, substrings of length $l = 1$ and $l = 2$:

1. For $0 \leq i \leq n$, $P[i, i] = true$, since every single letter in the string by itself is a palindrome
2. For $0 \leq i \leq n - 1$, $P[i, i + 1] = true$, $max = 2$, $left = i$ if first and last letters are the same: $w[i] = w[i + 1]$

For substrings of length $l \geq 3$ apply the following rule:

$$P[i, j] = (P[i + 1, j - 1] \text{ AND } w[i] = w[j]) \quad (1)$$

Note that $P[i+1, j-1]$ is already known at this point.

Algorithm *for substrings of length* $l \geq 3$

```
▷ ...
for l ← 3 do n
    for i ← 0 do n − l + 1
        j ← i + l − 1
        if P[i + 1, j − 1] and w[i] = w[j] then
            P[i, j] ← true
            if l > max then
                max ← l
                left ← i
return w.subString(left, left + max − 1)
```

1.2 Part 1. Running time

The running time is $O(n^2)$, since it has 2 nested loops. It runs for at most n^2 iterations (because the total number of substrings in a string with size n is $(n * (n + 1))/2$ excluding empty substring) and spends constant time in each iteration checking every substring based on the stored results of the solved subproblems, which allow us to avoid unnecessary recomputations.

1.3 Part 2. Algorithm

The proposed algorithm designed using bottom-up DP approach and Longest Common Subsequence (LCS) algorithm, as in *exercise 1.1* solving subproblems and storing their results in advance. Let $L[n + 1, n + 1]$ be an integer matrix (extra store for 0), where $L[i, j]$ contains length of LCS of substrings $w(0, ..., i - 1)$ and $rev(0, ..., j - 1)$ (reverse of w), for $0 \leq i \leq j \leq n + 1$ with $n = \text{length}(w)$. Each $L[i, j]$ can be computed as follows:

$$L[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ OR } j = 0, \\ L[i-1, j-1] + 1, & \text{if } w[i-1] = rev[j-1] \text{ AND } i, j > 0 \\ \max\{L[i-1, j], L[i, j-1]\}, & \text{if } w[i-1] \neq rev[j-1] \text{ AND } i, j > 0 \end{cases} \quad (2)$$

Algorithm *solving* $L[n + 1, n + 1]$

```
for i ← 0 do n+1
    for j ← 0 do n+1
        if i = 0 OR j = 0 then L[i, j] ← 0
        else if w[i − 1] = rev[j − 1] then
            L[i, j] ← L[i − 1, j − 1] + 1
        else
            L[i, j] ← max(L[i − 1, j], L[i, j − 1])
▷ Append one by one char-s from last L[n + 1, n + 1]
▷ ...
```

Algorithm *recovering the actual palindrome*

```
▷ ...
i ← n + 1, j ← n + 1
while i > 0 AND j > 0 do
    if w[i − 1] = rev[j − 1] then
        append w[i − 1] to result
        i ← i − 1, j ← j − 1
    else if L[i − 1, j] > L[i, j − 1] then i ← i − 1
    else j ← j − 1
return result
```

1.4 Part 2. Running time

This algorithm runs in $O((n + 1)^2)$ time, as double-nested loop takes quadratic time and with constant time in each iteration. The exercise requires an algorithm with the running time $O(n^3)$ and the proposed algorithm runs in $O((n + 1)^2)$. Since $(n + 1)^2 < n^3$ it's also correct to say that its running time is equivalent to $O(n^3)$.

2 Exercise 2

2.1 Algorithm

The problem of finding seating arrangement can be solved using *bipartite graph* and finding *maximum bipartite matching*. Given an undirected graph $N = (V, E)$ where $V = I \cup F$ and set of edges E representing mutual interests of $i \in I$ and $f \in F$ based on a list of *good* pairs $P \subseteq I \times F$, for all $e = (i, f) \in E$.
For N with $|V| > 3$ and $|I| = |F|$, and when $|P| \geq |V|$:

1. Construct a *directed graph* $N' = (I \cup F \cup \{s, t\}, E')$
2. Direct all edges from I to F and assign unit capacity
3. Add source s and edges with capacity equal to 2 from s to each node in I
4. Add sink t and edges with capacity equal to 2 to t from each node in F
5. Use any max flow algorithm to solve max flow problem on the new graph N', like Ford-Fulkerson algorithm.
6. If value of max flow x from s to t is equal to total sum of capacities from s to t, then seating arrangement exists; otherwise no.

2.2 Proof of correctness

Capacity of 2 from s and to t ensures that each investor/founder can be matched with up to 2 founders/investors respectively. Specifically, because the edges from s to I have capacity 2, it allows to have flow on **at most 2 edges leaving the same vertex** in I. And similarly, the edges from F to t have capacity 2 means that the flow uses **at most 2 edges to enter the same vertex**. So, each vertex in I and F participates in at most 2 edges in any matching, allowing 1 investor to be matched with no more than 2 founders and vice versa. Suppose some $i' \in I$ participated in 3 edges. This means that 3 units of flow leave from i', thus by conservation of flow, 3 units of flow enters i'. This is impossible, since at most 2 units of flow come into i'. Same reasoning applies to any $f \in F$.
Because one vertex can participate in at most 2 edges, any matching in N has cardinality of at most $min(I, F)*2$, but knowing that $|I| = |F|$ for seating arrangement to exist, any matching has cardinality $\leq |I| + |F|$. From which derives that the largest possible matching in N equals number of guests, which means that all guests have seats and were arranged according to their interests. Maximum cardinality of a matching in N equals the value of max flow in N', therefore finding max flow in N' determines the existence of seating arrangement. Define M is a maximum matching in N and the corresponding flow x in N'. Suppose that x is not maximum. Then there is a maximum flow x' in N' such that $|x'| > |x|$ and x' corresponds to a matching M' in N with cardinality $|M'| > |M|$, because $|x'| > |x|$, which contradicts the maximality of M. Hence, maximum flow in N' corresponds to a maximum matching on N.

3 Exercise 3

3.1 Algorithm

The algorithm can be achieved using greedy approach. Define your current score C, array of projects $p = \{p_0, p_1, ..., p_{n-1}\}$, such that each p_i is defined by its credit scores $c(p_i)$ and positive or negative value $b(p_i)$, which influences your score C, as a pair $(c(p_i), b(p_i))$, with assumption $\forall p,\ c_p + b_p \geq 0$.
Problem is divided into subproblems and solved in similar way but applying different ordering rule. First is to consider projects with **positive** $b(p)$ and sort them in **ascending order** by $c(p)$. For the second case of **negative** $b(p)$ order projects in **descending order** by $c(p) + b(p)$:

1. Consider p in **increasing order** of $c(p)$
2. Iterate through the projects in p and consider only **positive** $b(p)$
   (a) if project p_i is feasible, then complete project p_i
3. Consider p in **decreasing order** of $c(p) + b(p)$
4. Iterate through the projects in p and consider only **negative** $b(p)$
   (a) if project p_i is feasible, then complete project p_i

Algorithm Project completion algorithm

```
▷ SORT p in ascending order by c(p)
for i ← 0 do n
    if b(p_i) ≥ 0 then
        if C ≥ c(p_i) then C ← C + b(p_i)
        else return FALSE
▷ SORT p in descending order by c(p) + b(p)
for i ← 0 do n
    if b(p_i) < 0 then
        if C ≥ c(p_i) then C ← C + b(p_i)
        else return FALSE
return TRUE
```

3.2 Running time

The algorithm performs 2 sorting operations and 2 for-loop iterations. Sorting n elements can be performed with running time $O(nlogn)$. The running time of iteration over n elements is $O(n)$ with the constant time operation in each iteration $O(1)$. In total the running time of the algorithm is $O(2nlogn+2n)$ which is equivalent to $O(nlogn)$.

3.3 Proof of correctness

Algorithm determines if the n number of projects can be completed, otherwise it is impossible. The correctness of the proposed algorithm can be proven by induction. Since algorithm solves 2 subproblems which only differ in ordering rule, there will be a proof for one which can be applied to the second considering different ordering. First subproblem of positive $b(p)$. Let $S \subseteq P$ with positive $b(p)$ and size r, and S' be an array that will be filled with feasible projects from S, specifically if $C \geq c(p)$. S is sorted in increasing order by $c(p)$ and scanned for feasible projects. The goal is to show that for all r projects, if all projects from S can be completed then $S' = S$.
Base case of $r = 1$, if project is feasible it is certainly added to S', hence $S' = S$.
Induction step, assume that the statement is true for p_i and prove it for p_{i+1}. Since p_{i+1} is the next project to be added to S', it means that $C \geq c(p_{i+1})$, so any feasible project in S is added to S'. Hence, for all feasible projects from S, they are added to S', thus $S' = S$. Consider case where $S' < S$, which means there is at least 1 project in S that is not feasible, but that project p', which was not added to S', satisfies $C \geq c(p_i)$. This is contradiction, since algorithm always chooses to add feasible projects to S', but it ignored p'.
For the second subproblem of negative $b(p)$, define $N \subseteq P$ but with negative $b(p)$ and size t, and N' be an array filled with feasible projects from N. N is sorted in decreasing order by $c(p) + b(p)$. The proof for the first subproblem can also be applied to this subproblem but considering different ordering rule.
In the end, algorithm returns *TRUE* if all projects with positive and negative $b(p)$ were considered feasible, otherwise if at least 1 project from P does not satisfy $C \geq c(p)$ then it returns *FALSE*.

4 Exercise 4

4.1 Part 1. Algorithm

The proposed algorithm is based on the idea of *binary search* algorithm. Given an array C of cures with the size n and dose d of any cure which kills virus, define an array of minimum required doses a for each cure, j as a test dose, x as previous j, u and l as upper and lower bounds for j. For each cure c_i, where $0 \leq i \leq n$, initialize $x = 0$, $l = 0$, $u = d$ and $j = (l + u)/2$, half of d. While $j > 0$, set $x = j$ and test if j dose kills virus:

1. If test is positive, reduce upper bound, $u = j$
2. Else, increase lower bound, $l = j$
3. Given new bounds, set j to the half of new range closer to the minimum a_i, $j = (l + u)/2$
4. If $x = j$, previous j equals current, then minimum a_i is found, thus $a_i = x + 1$

The last step is to scan a for minimum a_i and select corresponding c_i.

Algorithm Find dose a for each cure c algorithm

```
for i ← 0 do n
    x ← 0                                  ▷ Precious j
    l ← 0                                  ▷ Lower bound
    u ← d                                  ▷ Upper bound
    j ← (l + u)/2                          ▷ Note, j is not rounded
    while j > 0 do
        x ← j
        if Test(c_i, j) = TRUE then u ← j
        else l ← j
        j ← (l + u)/2                      ▷ Note, j is not rounded
        if x = j then
            a_i ← x + 1
            break
```

Note, $a_i = x + 1$ since j is not rounded.

4.2 Part 1. Running time

The running time of the algorithm is $O(nlogd)$. The algorithm runs as double-nested loop where outer for-loop performs in $O(n)$ time, iterating over each cure in C with size n; and inner while-loop runs in $O(logd)$ time, iterating until the minimum required dose a_i for c_i is not found, starting from the half of the dose d and repeating the process about halfway through the part of d where the minimum a lies. The last step is iteratation over array a to find minimal a_i. Thus, $O(nlogd + n)$ which is equivalent to $O(nlogd)$.

4.3 Part 2. Algorithm

Define an array C of cures with the size n, dose d of any cure which kills virus, an array of minimum required doses a for each cure.

4.4 Part 2. Running time