Luca Sannino 1542194 Homework 1 for Algorithm Design 2020/2021

Exercise 1.1

I decided to implement a dynamic programming algorithm, where ***OPT(i, j)*** contains the length of the longest palindrome sequence inside a substring of the input string that begins at character ***i*** and ends at character ***j***.
The algorithm consists of these different cases:

- **CASE 1**: ***OPT*** returns 0, which is the case when ***i*** is bigger than ***j***, meaning that it's an invalid substring where there can't possibly be any palindrome sequences.
- **CASE 2**: ***OPT*** returns 1, which is the case when ***i*** is the same as ***j***, meaning that we are looking at just one character, which is obviously palindrome to itself.
- **CASE 3**: ***OPT*** returns the length of the substring because it's palindrome, which is the case when ***s[i]*** is equal to ***s[j]***, and ***OPT(i+1, j-1)*** is equal to the size of the substring that starts at ***i***+1 and ends at ***j***-1.
- **CASE 4**: ***OPT*** returns the maximum value between ***OPT(i, j-1)*** and ***OPT(i+1, j)***, which is the case when the substring isn't palindrome, so the optimal value is to be found inside another section of the substring.

The algorithm returns the substring that has the best ***OPT(i, j)***, and can be optimized by implementing caching and unwinding the recursion. It has a time complexity of $O(n^2)$ because of both the initialization of the cache matrix of size ***n*n*** and the unwinding of ***OPT*** that require a cycle inside another cycle, with both cycles that have an iterator that goes through the entire length of the input string of size ***n*** as seen in the pseudocode below.

Exercise 1.2

This is just a variant of the previous algorithm. ***OPT(i, j)*** contains the biggest non-continuous palindrome sequence(and not the length!) inside a substring of the input string that begins at character ***i*** and ends at ***j***.
CASE 1 and **CASE 2** return respectively the empty string and the character inside ***s[i]*** (which is the same as ***s[j]***), while **CASE 3** returns the string formed by ***s[i]+OPT(i+1,j-1)+ s[j]***, and can be triggered regardless of the value of ***OPT(i+1, j-1)***, a consequence of the fact that the sequence can be non-continuous. Lastly, **CASE 4** returns the biggest string between ***OPT(i, j-1)*** and ***OPT(i+1, j)***.
By implementing two types of caching (one for the strings and the other for their lengths) and by unwinding the recursion, the algorithm is $O(n^3)$ for the addition of a new operation of cost ***O(n)*** inside the inner cycle.

Pseudocode

LONGEST-PALINDROME(s, n)

```
Create new caching matrix c[n][n]    // O(n^2)
i_best = 0
j_best = 0
for i = n-1 to 0        // O(n)
  for j = 0 to n-1      // O(n)
    if i > j
      c[i][j] = 0
    else if i == j
      c[i][j] = 1
    else if s[i] == s[j] and c[i+1][j-1] == (j-1)-(i+1)+1
      c[i][j] = 2 + c[i+1][j-1]
      if biggest result so far
        i_best = i;  j_best = j
    else
      c[i][j] = max{c[i][j-1], c[i+1, j]}
Return substring between i_best and j_best
```

LONGEST-NOT-CONTINUOUS-PALINDROME(s, n)

```
Create new caching matrix c[n][n]   // O(n^2)
Create new caching matrix x[n][n]   // O(n^2)
for i = n -1 to 0       // O(n)
  for j = 0 to n-1      // O(n)
    if i > j
      x[i][j] = empty string;    c[i][j] = 0
    else if i == j
      x[i][j] = s[i];   c[i][j] = 1
    else if s[i] == s[j]
      c[i][j] = 2 + c[i+1][j-1]
      x[i][j] = s[i] + x[i+1][j-1] + s[j]
    else
      m = max{ c[i][j-1], c[i+1][j] }
      if x[i][j-1].length == m   // O(n)
        x[i][j] = x[i][j-1];    c[i][j] = c[i][j-1]
      else
        x[i][j] = x[i+1][j];    x[i][j] = x[i+1][j]
Return x[0][n -1]
```

Exercise 2

Before showing the design of the flow network, I wanted to illustrate the driving principles behind its ideation:

● **Principle 1**: Given ***n*** investors (***I***) and ***n*** founders (***f***) that need to sit in round tables, where $n \geq 2$, if each investor sits in a table where their two neighbors are founders they form good pairs with, and if each founder sits in a table where their neighbors are investors they form good pairs with, then it's possible to have a seating composition of at least one table where each founder sits in a table where half of the participants are investors, and vice versa, such that if I take any pair of neighbors (***a***, ***b***), we have that $a \in I$ and $b \in F$ (or vice versa), and that $(a, b) \in P$, which is the list of good pairs.

● **Principle 2**: If principle 1 holds, then the smallest table returned by the seating composition with only good pairs has size $x*2$, with $x \leq n$ and where ***x*** is the size of the smallest subset of investors whose sum of distinct founders they form good pairs with has a value of ***x***. The reason is that since they have no relation with the rest of the founders, they can only sit in a table of size $x*2$ with the founders they form pairs with, otherwise were they to sit in a larger table one of the investors would be forced to have a neighbor which doesn't make a good pair with. If $x == n$, then we have only one large table of size $n*2$, while if $x == 1$ we have one table of two, which is forbidden by the exercise text and impossible according to principle 1, since the min value of ***x*** can be only 2. The same reasoning also applies to the founders.

Now it's possible to introduce the graph:

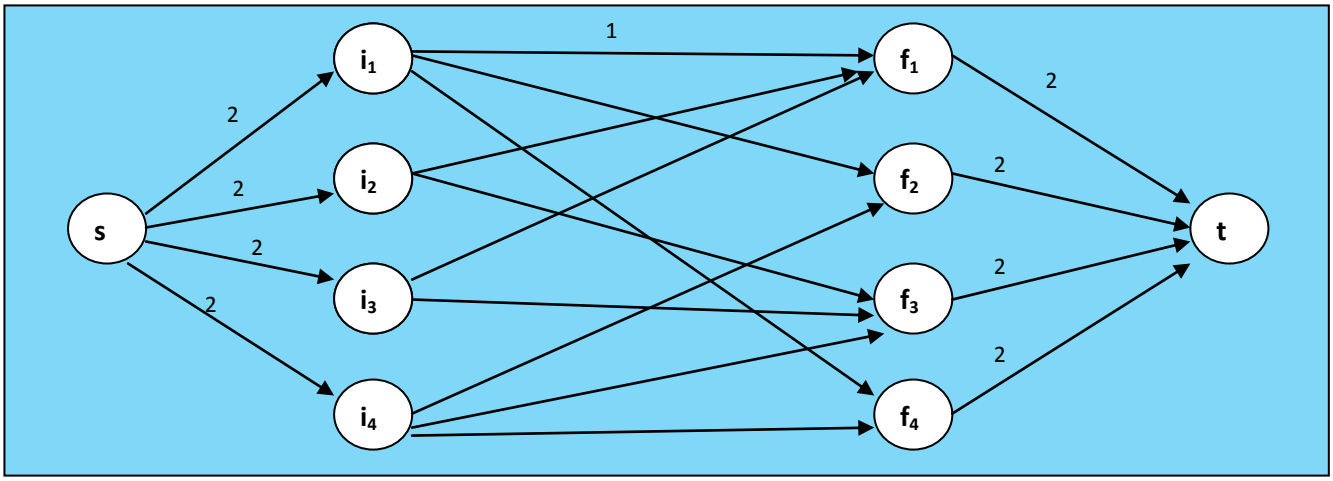
● We create a node ***i*** for all ***n*** investors in ***I***, and we add an edge (***s***, ***i***) with capacity 2: the flow will indicate the number of founders of interest sitting near ***i*** .

● We create a node ***f*** for all ***n*** founders in ***F***, and we add an edge (***f***, ***t***) with capacity 2: the flow will indicate the number of investors of interest sitting near ***f***.

● We add an edge (***i***, ***f***) with capacity 1 if an investor and a founder belong to the same good pair in ***P***: if the flow is 1 they sit near in the same table, if the flow is 0 they do not.

If the max flow returned by the Ford-Fulkerson algorithm is equal to $n*2$, and $n = |I| = |F|$, then it means that each of the ***n*** investors has two founders of interest sitting near them, and vice versa, and since investors and founder have the same size ***n***, principle 1 is respected and a seating combination is possible.

Here's an example where $n = 4$ and P=[(i_1,f_1), (i_1, f_2), (i_1, f_4), (i_2, f_1), (i_2, f_3), (i_3, f_1), (i_3, f_3), (i_4, f_2), (i_4,f_3), (i_4, f_4)].



The max flow here is 8=$n*2$, so first principle is respected. But since i_2 and i_3 form good pairs with only f_1 and f_3, we have the same situation described in principle 2 with value $x = 2$, so the smallest table has to be of size $x*2=4$. The possible seating is: t_1=(i_2, f_1, i_3, f_3) and t_2=(i_1, f_2, i_4, f_4), since both i_2 and i_3 can't sit near f_2 or f_4.

Exercise 3

I have decided to solve this problem through a greedy algorithm:

- The first step is to separate all projects in two different lists: the first one will contain all projects that net a non-negative credit reward, while the second will contain all the rest.
- We order the first list by smallest credit requirement first.
- We then cycle through all the ordered projects in the first list: if for current project ***i*** I have that $C \geq c_i$, then I am able to take it and add its credit reward to my ***C***. Otherwise I return false: the reason is that since I already took all previous projects in the non-negative credit reward list that have as requirement $c_j < c_i$, and since the other list contains all jobs with negative credit reward, I have no way to increase my ***C*** so that I am able to take project ***i***, meaning that a sequence where I take all ***n*** projects is impossible.
- If I am able to make it through the first list, then it means I have taken all the jobs that have not decreased my credit score, meaning that ***C*** is in the best possible scenario to tackle all the other jobs that will decrease it.
- We then order the second list by greatest $(c_i + b_i)$ first. If we have projects ***i*** and ***j*** such that $(c_i + b_i) \geq (c_j + b_j)$, then it means that $c_i \geq c_j$ and/or $b_i \geq b_j$. Both cases help us, because in the first case we want to give priority to projects that have bigger credit requirements, so that I can exploit ***C*** while it's still big enough (since ***C*** will inevitably decrease the further I go in the list), while in the second one we want to give priority to projects that do the smallest damage to ***C***, so that I always have the best chance to take the next one (remember that all rewards are negative). Since project ***j*** can satisfy both cases only when it has the same values as project ***i***, while there are no such constraints for ***i***, there are more valid reasons to pick project ***i*** over ***j*** than vice versa.
- We then cycle through all the ordered projects in the second list: if for current project ***i*** I have that $C \geq c_i$, then I am able to take it and add its credit reward to my ***C***, decreasing it in the process. Otherwise I return false, since I don't have a way to increase ***C*** while being in the optimal path because I already went through the non-negative reward list, meaning that a sequence where I take all ***n*** projects is impossible.
- If I am able to reach the end of both lists, then I can return true because taking all ***n*** projects is possible, and the right order is the one which I just went through. The time complexity is ***O(n log n)*** because I can order the lists in ***O(n log n)*** and iterate through them in ***O(n)***.

Pseudocode

```
DO-ALL-PROJECTS(n, C, c_1, ..., c_n, b_1, ..., b_n)
Create two empty lists l_1 and l_2
for i = 1 to n // O(n)
  if p_i ≥ 0    Add job i to l_1
  else          Add job i to l_2
Sort l_1 so that c_1 ≤ c_2 ≤ ... ≤ c_x  // O(n log n)
for i = 1 to x    // O(n) since x ≤ n
  if C ≥ c_i
    C += b_i
  else
    Return false
Sort l_2 so that c_1+b_1 ≥ ... ≥ c_y+b_y  // O(n log n)
for i = 1 to y    // O(n) since y ≤ n
  if C ≥ c_i
    C += b_i
  else
    Return false
Return true
```

Luca Sannino 1542194 Homework 1 for Algorithm Design 2020/2021

Exercise 4.1

In the deterministic approach, I execute a variant of the binary search for all ***n*** cures, where for every iteration of the search with values ***i*** and ***j***, and where ***i*** < ***j***, I look if the cure works with a dose ***mid*** of quantity (***i***+***j***)/2. If not, it means that a working dose can only be bigger, so I search in the upper half [***mid***+1, ***j***], otherwise I save it as the smallest dose recorded so far (every search begins between 1 and the previously smallest dose) and I search in the lower half [***i***, ***mid***-1] hoping to find an even smaller one.
This variant of the binary search always reaches the leaves of the tree, and since the trees at worst have height ***O(log d)*** (the search is first inizialized with values [1, ***d***]), the total cost is ***O(n log d)***.

Exercise 4.2

In the randomized approach, I pick at random one cure between all ***n*** available. I then execute the variant of the binary search described above in order to find the smallest dose that make that particular cure work. Finally I check if the other ***n***-1 cures work with the dose I just found, but subtracted by one, by selecting them randomly and permanently removing the ones that don't work and stopping when the first of them turns out to work. If no cure works, then the one I have randomly chosen is the best one; else, I start a new iteration (meaning a new binary search) with the cure that I know is better than the one I had chosen previously: this means that every binary search sans the first one is done as consequence for not picking the best cure, so the effectiveness of the algorithm depends on how many times I expect this to happen.
If we think at the chosen cures as pivots, then this problem is similar to Quicksort, where in order to not have lists ***L*** and ***G*** with more than ¾ of the elements at the end of each iteration, you have a probability of choosing a good pivot of ½, because you don't want to pick one of the bigger elements in the upper quarter or one of the smaller elements in the lower quarter of a set. Here, however, not only we don't care if we pick an element with a small value, but it's our objective, so this means we have a possibility of picking a good cure of ¾, so that we don't have in the next iteration a set of cures that is bigger than ¾****s***, where ***s*** is the size of the one that was given in input in that iteration. The fact that I can instantly find a better cure after a binary search, meaning that I don't get to remove inferior cures, with the consequence that I can possibly reach the next iteration with a set bigger than ¾ ****s***, has no importance, because they will be eventually removed in future iterations by single tests, without being considered for an useless binary search. Being a better case of Quicksort, where on average you have ***k*** good picks for 2****k*** iterations, I expect a tree of size ***O(log n)***, with the worst case at ***O(n)***, where I do just one binary search for each iteration. When summing the single tests after a search, which at max can be ***n***-1 across the whole run time, the total number of tests expected is ***O(n + log d * log n).***

Pseudocode

```
smallest_dose = d        // Global variables
best_cure = null;
FIND-BEST-CURE(c_1, ..., c_n)
for i = 1 to n       // O(n)
  binary-search(c_i, 1, smallest_dose) // O(log d)
Return best_cure and smallest_dose
BINARY-SEARCH(c, i, j)
  if i < j
    mid = (i+j)/2
    if c with mid quantity doesn't work
      binary-search(c, mid+1, j)
    else
      smallest_dose = mid
      best_cure = c
      binary-search(c, i, mid-1)
```

```
smallest_dose = d     // Global variable
FIND-BEST-CURE-RANDOMIZED( c_1, ..., c_n)
Insert all n cures into list l;
c_i = Choose (and remove) a random cure c from l
while true   // O(log n) on average, O(n) at worst
  binary-search(c_i, 1 , smallest_dose) // O(log d)
  single_positive_test = false
  while l is not empty and single_positive_test == false
    c_j = Choose (and remove) a random cure c from l
    if single_positive_test == false  // n-1 tests in total
      if c_j works with quantity (smallest_dose-1)
        single_positive_test = true
        c_i = c_j; smallest_dose += -1
  if single_positive_test == false
    Return c_i and smallest_dose
```