Algorithm Design - Homework 1

Gianluca Pulicati - 1708686

Exercise 1. **1)**To approach this exercise first i started thinking about how to improve a "Brute Force" solution. Analyzing a palindrome string brought me to the idea of capitalizing the solution around the "symmetry condition": searching for the center. The main problem of this intuitive solution was to deal with strings which haven't a center with single char, i.e "*tennet* $\neq$ *tenet*". Basically the algorithm initialize **maxSub = 0**, **startSub = 0** then starts scanning the string: in this first cycle the current value **mid** is considered the center of the palindrome (**left = mid, right = mid**); then from this value **mid** we check (with a while) if there exist consecutive same characters, the algorithm skips them. (if **s[right] = s[right + 1] then right++;**) At this point the algorithm found out if the center has a single char (left = right) or not. Now while in the boundaries of the string the solution checks (with a while) if **s[left-1] == s[right+1]** and continues (left– and right++) until the string is over (out of boundaries conditions). Then at the end of this third while the algorithm update **maxSub** and **startSub** only if **right - left** $>$ **maxSub**. Here the algorithm start a new iteration of the first cycle (now **mid = mid +1**) and repeats all. When this first cycle ends, the output of the solution is substring(s,startSub,maxSub). The cost of this method is $O(n^2)$, even if there are multiple cycles, what this algorithm does is to use every character as the center (first $O(n)$) and from it "expand from the center" considering again all characters, maybe skipping the ones which are equals (second $O(n)) = O(n) * O(n) = O(n^2)$ in the worst case.

2) This is a dynamic programming problem: is necessary to test every subsequence but is important to keep track of operations already done: scan the string with two index **i** and **j** for each value of i we have a row on a matrix **M**[len(s),len(s)], for each j a column instead. Keep i and j "at the same distance" and check if the character covered are the same, if yes then: **M**[i,j] = **M**[i+1][j-1] + 2 (increase the length of longest subsquence), otherwise **M** = max(**M**[i][j-1], **M**[i+1][j]) (keep the max length found until now). To keep track of the longest subsequence simply save in a variable the current highest length found and the value of **i** and **j** of it (and update it accordingly while filling the matrix), the relative string at the end of the algorithm will be substring(s,i,j). The cost of this solution is again $O(n^2)$ basically it is creating a matrix and filling it with values while scanning it, only once. ($O(n) * O(n) = O(n^2)$ in the worst case).

Exercise 2. This exercise gives a list of *good pairs: (i-f)* into a list $P \subseteq IXF$ which are used to build a flow network N that respects the constraints. The formal definition of N is the following: Model the system as a bipartite matching problem (with some differences) and add the constraint about having the same number of founders and investors (otherwise will always be false cause someone will not have a neighbor); create a digraph that has a source **s** and a sink **t**. If we suppose that n_f is the number of different founders according to P then the "left" nodes of the bipartite graph are all n_f elements of P. "Right" nodes are instead the n_i elements; in this way we have the source connected with a directed edge to all n_f nodes and all n_i nodes connected with a directed edge each to the sink. To model the pair between founder f_i and investor i_i an edge is added only if $\exists$ *(f_i,i_i)* $\in P$, thus the graph defines only the existence of good pairings. Then we set all the edges from **s** to left nodes with a max capacity C of 2, the same is done for all the edges from right nodes to sink; this condition models the fact every founder and investor may have only 2 different neighbors (left and right on the round table). Edges modeling pair condition have instead a max capacity C_p of 1 (founder and investor are neighbors "only once"). To find the solution the algorithm starts searching the max flow $val(fl)$ through the graph. Output will be *True* only if $val(fl)$ is equal to the sum of all capacities C between right nodes and sink edges.
The prove of correctness comes from the way the graph is designed: first of all $C = 2$ laws the fact of both having two neighbors of different "nature" (so no founder sited next to another one and the same for investors) and tables being filled with at least four people. When the flow starts with the search of an augmenting path (Ford-Fulkerson), it is basically checking if the pairs modeled by the graph are admissible. Suppose that the algorithm returns *False* even if a "good pairing" exists; this means that an investor is somehow not reached by two founders, but if a good pairing exist every left and right node is reached by at least two edges (pairing) otherwise even the "at least three in a table" condition is violated. Now suppose that the algorithm returns *True* even if a "good pairing" doesn't exist; this means every investors is reached by at least two edges (pairing), but if a good pairing doesn't exist then neither these edges should. What happens if a founder has multiple preferences of investors (multiple tuples (f_i,i_i))? Thanks to residual graph built by Ford-Fulkerson algorithm (and the theorem which confirms that a max flow of a graph built in this way is equal to the max cardinality of a matching in a bipartite graph) we can be sure that every founder will eventually find 2 investors as good neighbors (if at least two pairs (f_i,i_i) exists otherwise will always return false).

Exercise 3. The exercise is solved with the use of greedy approach. In the following lines will always be respected this condition: Given the two arrays $c_p[n]$ and $b_p[n]$, if one of them is sorted respecting some condition the other one will always be adjusted in order to have the corresponding value in the same index (or in other words: if $c_p[h] = c_i$ then $b_p[h] = b_i$). To check if is possible to do all n projects the algorithm will consider only project with a positive b_i, and will try to solve first the ones with lowest cost_min c_i (updating $C = C + b_i$); if some project doesn't meet the $C = c_i$ condition the algorithm return false. Otherwise will be the turn of b_i negative. To solve this part the algorithm calculate $\Delta = c_i + b_i$ for every remeaning project and save this value in a new array. This array will be ordered this time in descending order and will be solved the projects with the corresponding highest Δ value. If some project have the same Δ then will be solved first the ones with the lowest value of (negative) b_i. At the end, if some project is still not resolved, the algorithm returns False.
The algorithm consists in a sequential use of a sorting algorithm (for instance Mergesort). Depending on how many Δ will have the same value, the algorithm cost will be: $O(nlogn) = O(nlogn) + O(nlogn) + O_\Delta 1(nlong) + O_\Delta 2(nlogn) + ... \cong O(nlogn)$. The key for this exercise is to to take care of both b_i and c_i once is the time to resolve the projects with b_i negative; in this way (according to the fact that $c_i + b_i >= 0$) the cost itself of each project will be somehow "normalized" with negative gain it provides (and avoid the situation of resolving project with very low cost and huge gain b_i loss, for example).
This solution works because in the first part (positive b_i) works only with projects which increase the gain, thus by induction is only needed to check the cost c_i. If $c_p[h]$ is ordered in ascending way, then even if any of these project "fails" the algorithm stops cause there will be no way to solve it (even all the projects after him which have a higher cost). Then when b_i becomes negative, as explained above, is needed to check both c_i and b_i values. By induction is possible to prove that to check if is possible to do all projects, Δ need to be calculated: the higher Δ has the higher "distance between c_i and b_i" and this condition is the best one to solve first (Cost higher and gain loss tinier). Suppose that the highest Δ is not resolvable, this means that C is not enough to solve the project, but the subsequent projects will decrease the cost even more, and so it will never be! It's quite similar to the previous step, the only condition modeled is: "this project with higher cost or higher gain b_i loss, is not resolvable and it will never be".

Exercise 4. **1)** In order to find the best cure for the solution of this problem, the only way possible is to test every cure. The key, though, is to try reducing the amount of test needed for each cure. Solving this point requires the use of Binary Search; simply keep track of the current minimal among all cures (C_m) and start a Binary Search for every cure (so first check if $a_i = d/2$ is valid, then proceed to test between $[1, d/2]$ and so on), the result of this Binary Search will update C_m accordingly. The cost of this algorithm is the cost of $n*$BinarySearch $= O(n) * O(logn) = O(nlogn)$ in the worst case.

2) The approach for this randomization solution is not much different than the previous one. Basically pick a random index k and use the first n texts to check if the cure c_i works with a_k; is possible to proof that the number of cures working with a_k is $> log(n)$ with very low probability thanks to Chernoff upper bound. Set Y_ik = Cure i works with a_k (or <k) (1 = yes, 0 = no) and X_k = Total number of cures working with a_k. Thus $X_k = \sum_n Y_ik$; also $\mu = E[X_k] = 1$ and $\delta = log(n) - 1$. It results: $Pr[X_k > log(n)] < (e/log(n))^log(n)$. Now let γ be the number x such that $x^x = n$ and choose $log(n) = e * \gamma(n)$. Is possible to write: $Pr[X_k > log(n)] < 1/n^2$.
So thanks to Chernoff bounds we can assume that random value a_k picked results suitable for more than $log(n)$ cures with a really tiny probability ($< 1/n^2$) which decreases according to the number of cures (n).
Thanks to this is possible to test these $log(n)$ cures found with the Binary Search approach, thus the cost of this second part is: $O(log(n) * O(log(d)) = O(log(n) * log(d))$
Is important to stress the fact that this Chernoff upper bound works only in the average case of "picking a good a_k", is still possible to randomly pick the worst possible value of it and to have the worst run of the algorithm with the highest number of tests needed to find the perfect cure.