COMPSCI 3AC3

Assignment 2

Nathan Agbomedarho
agbomedn
February 6 2023

1 Part I

Exercises 3, 5, 6, 16, 24 from Chapter 4.

1.1 Q3

Let us assume that n boxes arrive such that: $b_1, \ldots, b_n$ and each box b_i has a positive weight w_i and W is the maximum weight a truck can carry. Boxes will be packed in N trucks, allocating one box to one of the trucks $1, \ldots, N$ such that the weight of boxes on each truck is $\leq W$ and if a box b_i leaves before box b_j, b_i must then have arrived before b_j.

We then prove that this greedy algorithm uses the optimal amount of trucks. If the algorithm can fit boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and $b_1, \ldots, b_i$ of the other solution fits boxes onto the first k trucks, $i \leq j$. By induction on k we will prove this claim. For case $k = 1$, the algorithm puts as many boxes as it can into the first truck. If this holds for $k - 1$: the algorithm fits j' boxes into the first $k - 1$, and $i' \leq j'$ fits in the other solution. For the case of the k^{th} truck, another solution fits $b_{i'+1}, \ldots, b_i$, therefore as $j' \geq i'$ the algorithm can fit all boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and may be able to fit in more.

The proof of the claim is complete, proving the optimal solution of the greedy algorithm.

1.2 Q5

We first define, for any point on the road, its position as miles away from the west end. The base station will be located at the farthest eastern position s_1, such that all houses from 0 to s_1 are covered by s_1. With $\{s_1, \ldots, s_i\}$ placed, the base station $i + 1$ will be put at the largest position s_{i+1} such that all houses from s_i and s_{i+1} are covered by s_i and $s_{i|1}$.

We define $S = \{s_1, \ldots, s_k\}$ to represent the set of the positions of all base stations that the algorithm allocates, and $T = \{t_1, \ldots, t_m\}$ represents the set of all base station positions in an optimal solution, in increasing order from west to east. Now we show that $k = m$. First, we claim that $s_i \geq t_i$ for each i, and we prove this by induction. For the case $i = 1$, as we venture to the easternmost location before we place the first base station, the claim is true. Let us then assume the claim is true for a value $i \geq 1$, which means that the algorithm's first i centers $\{s_1, \ldots, s_i\}$ encompasses every house covered by the first i centers $\{t_1, \ldots, t_i\}$. As such, if we add t_{i+1} to $\{s_1, \ldots, s_i\}$ no houses will be left uncovered between s_i and t_{i+1}. The $(i+1)^{\text{st}}$ step of the algorithm selects s_{i+1} to be the largest possible size within the bounds of coverage for all houses between s_i and s_{i+1}. Thus, $s_{i+1} \geq t_{i+1}$, and by induction the claim is proven. If $k > m$, then $\{s_1, \ldots, s_m\}$ fails the coverage for the houses. If $s_m \geq t_m$, then $\{t_1, \ldots, t_m\} =' T$ doesn't cover every house, therefore we have a contradiction.

1.3 Q6

Contestants are numbered $1, \ldots, n$ and s_i, b_i, r_i to represent the swimming, biking and running times of contestants i. The algorithm is as follows: in decreasing order $b_i + r_i$ the contestants allocated in this order, and this order must minimize the completion time. By an exchange argument, we consider an optimal solution that doesn't use said order. As such, this optimal solution needs two contestants i and j so that j leaves immediately after i, except $b_i + r_i < b_j + r_j$. This pair denoted by (i, j) is an inversion. In a swapped schedule (swapping the order of i and j) j completes first and i leaves the pool as j left the pool before; since $b_i + r_i < b_j + r_j, i$ completes in the swapped schedule before j completes in the other schedule. The swapped schedule has a lower execution time, and as such it is optimal.

We can then remove inversions in the same way, maintaining the same running time. Once the algorithm finishes execution and returns the complete schedule with an optimal running time, thus the produced order of the algorithm is optimal.

1.4 Q16

Here, we have a set of n points and a set of intervals $([t_i - e_i, t_i + e_i])$, with the object of finding a perfect match between points and intervals, in such a way that each point is in its corresponding interval. We assume $x_1 \leq x_2 \leq \ldots \leq x_n$, so a potential algorithm could be as follows:

- for $i = 1, 2, \ldots, n$
- if unmatched intervals with x_i exist
  - pair x_i with the one that ends first
- else
  - no perfect match exists

If there is a perfect match, the algorithm postulates that it will locate it. By contradiction, let us assume that a perfect match exists. We choose a perfect match M, where the first i points $x_1, x_2, \ldots, x_i$ are matched with intervals with x_i. Then, suppose x_{i+1} is matched to a centered interval at t_l in M, however it instead matches x_{i+1} to a different centered interval at t_j. The algorithm stipulates that $t_j + e_j \leq t_l + e_l$, as such let us assume t_j matches x_k $(x_k \geq x_{i+1})$ in M. Therefore, $t_l - e_l \leq x_{i+1} \leq x_k \leq t_j + e_j \leq t_l + e_l$, as such x_k can be matched to t_l in M and also match x_{i+1} to t_j to obtain a new perfect match, M' which adheres to the algorithm. For the first $i|1$ points M' matches the output of the algorithm, which contradicts i.

If we match all unmatched intervals in every iteration, the algorithm is bounded by $O(n)$, and with n iterations the algorithm is bounded by $O(n)^2$.

1.5 Q24

We will design a greedy algorithm to solve this problem, following the Deferred Merge Embedding technique, which is as follows: Let v represent the root, with v' and v'' be its two children. D' denotes the maximum root-to-leaf distance that covers all leaves that descend from v', and then d'' represents the maximum root-to-leaf distance the covers all leaves that descend from v''. Knowing this then, If $d' > d''$, $d' - d''$ is added to the length of the v-to-v" edge and the length of the v-to- v' edge remains the same. If $d'' > d'$, $d'' - d'$ is added to the length of the v-to- v' edge and the length of the v-to-v" edge remains the same. If $d' = d''$ the length of either edge below v is not modified.

This procedure is applied recursively to rooted subtrees at v' and v''. We must establish some facts pertinent to the optimal solution to prove that the DME algorithm is optimal. T will denote a complete binary tree.

1. w denotes an internal node inside T, e and e'' denote two edges located immediately below w. As such, if non-zero lengths are added to e' and e'', the solution is not optimal. Proof: Let's assume that $\delta' > 0$ and $\delta'' > 0$ are added to $\ell_{e'}$ and $\ell_{e''}$ respectively; then let $\min(\delta', \delta'')$. A solution that adds $\delta' - \delta$ and $\delta'' - \delta$ to these edges must have no skew and less total length.
2. w denotes a node in T that is not a root or a leaf. A suboptimal solution must not increase the length of every path from w to a leaf located below w. Proof: Let us suppose that $x_1, \ldots, x_k$ are leaves below w. Edges e below w in the subtree will have the property: the length of e is increased, however the length of edges between w and e are not increased. F denotes the set of said edges and in F, for every leaf x_i the first edge on the $w - x_i$ path with an increased length is in F, as such there is exactly only one edge from F on each $w - x_i$ path. Furthermore, $|F| \geq 2$, considering that a path in the left subtree located below w has no edges in common with a path in the right tree respectively below w; each path exists on an edge of F. e_w denotes the edge that comes in w from its parent. The minimum length added to any edge in F will be denoted by δ. If by subtracting δ from the added length of edges in F, and adding δ to the edge above w, we then find that the length of all root-to-leaf paths are unchanged, and the tree is sill zero-skew. A length $|F|\delta \geq 2\delta$ has been subtracted from the total length of the tree and only δ has been added, thus, we then have a zero-skew tree with a combined less total length.
3. If we wish to produce a DME algorithm that results in a unique optimal

solution, then first, let us consider infinite solutions, and v will denote nodes in T where the solution does not add lengths produced by DME. We assume that $d' \geq d''$ and the solution adds δ' to the edge (v, v') and δ'' to the edge (v, v''). In such a way, if $\delta'' - \delta' = d' - d''$, it follows that $\delta' > 0$ or the DME would reflect the same as the solution, and by (1) it is not optimal. If $\delta'' - \delta' < d' - d''$, it follows that the solution must increase the length of the path outlined from v'' to each leaf, so that the tree is zero-skewed; as such according to (2) it is not optimal. Finally, if $\delta''\delta' > d'd''$, it follows that the solution must increase the length of paths from v' to its leaves so that the tree is made zero-skew; according to (2) this is also proven not optimal.

2 Part II

Exercises 3, 5, 7, 10, 17 from Chapter 6.

2.1 Q3

(a) A graph that could illustrate this is one wherein nodes $v_1, \ldots, v_5$ with edges $(v_1, v_2), (v_1, v_3), (v_2, v_5), (v_3, v_4)$ will return 2 paths belonging to edges (v_1, v_2) and (v_2, v_5); the optimal solution returns 3 corresponding to the paths $(v_1, v_3), (v_3, v_4)$ and (v_4, v_5).

(b) The algorithm will use sub-problems $Opt[i]$ for the longest path from v_1 to v_i. As not all nodes v_i will have a path from v_1 to v_i, this will be denoted by the value "$-\infty$". $Opt(1) = 0$ is the longest path with 0 edges. The algorithm is as follows:

- Declare an array $A[1 \ldots n]$ & $A[1] = 0$
- The first loop: $i = 2, \ldots, n$, $A = -\infty$
  - Second loop: all edges (x, y) - if $A[x] \neq -\infty$ - if $A < A[x] + 1$ - $A = A[j] + 1$ $A[y] = A$

This algorithm is bounded by $O(n)^2$ assuming all edges enter nodes i are in $O(n)$ time.

2.2 Q5

If the segmentation $y_1 y_2 \ldots y_n$ is optimal for the string y, then segmentation $y_1 y_2 \cdots y_{n-1}$ will result in an optimal segmentation of the prefix of y which excludes y_n. With this, we design $OPT(i)$ to represent the most optimal segmentation of the prefix, which contains the first i of characters y. By recurrence:

$$Opt(i) = \min_{j \leq i} \{Opt(j-1) + \text{ Quality } (j \ldots n)\}$$

we obtain the most optimal segmentation. The correctness of this formula is proven by induction on index i. The base case is trivial; for the inductive step, let us assume the Opt function calculates the optimum solution for indices less than i; we must demonstrate that $Opt(i)$ is the optimal cost for any segmentation for the prefix y to the i−th character. Let $j \leq i$ be the first index. From our above claim, the prefix with only the first $j-1$ must be optimal, however our induction hypothesis claims that $Opt(j)$ will return the value for the optimal segmentation. As such, the optimal cost $Opt(i)$ will be equal to $Opt(j)$ in addition to the last word.

The above recurrence finds the computes the possibility of the last word, thus it will calculate the cost of the optimal segmentation. The time complexity of such an algorithm will evaluate the above formula and will result in a quadratic running time.

2.3 Q7

For this question, we will illustrate a way in which it is possible to calculate the optimal numbers i and j in $O(n)$ time. X_j (for $j = 1, \dots, n$) will denote the maximum amount that it is possible for investors to get if their stocks are sold on day j. Here $X_1 = 0$, as such an optimal solution can be found by, selling the stock on day j and deciding if the investors held it on day $j-1$ or not. If not, $X_j = 0$, consequently $X_j = X_{j-1} + (p(j) - p(j-1))$ and thus $X_j = \max\left(0, X_{j-1} + (p(j) - p(j-1))\right)$. We return a maximum value over $j = 1, \dots, n$, of X_j.

2.4 Q10

(a) Here is an example of an instance in which the algorithm is proven to not correctly solve this problem

	Minute 1	Minute 2	Minute 3	Minute 4
A	2	1	1	200
B	1	1	20	100

In this case the algorithm chooses A, moves, then chooses B for the last two steps. The optimal solution would be to select A for all 4 steps.

(b)

Let $\text{Opt}(i, A)$ represent the maximum value of a plan in minutes 1 to i which ends on machine A, and $Opt(i, B)$ is denoted for B. If machine A is used for i minutes, we are either on machine A or moving from machine B. In case 1, $\text{Opt}(i, A) = a_i + \text{Opt}(i-1, A)$. In case 2, we were at B in minute $i-2$, $\text{Opt}(i, \Lambda) = a_i \mid \text{Opt}(i-2, B)$ and overall: $\text{Opt}(i, A) = a_i + \max(Opt(i-1, A), Opt(i-2, B))$. This holds for $\text{Opt}(i, B)$. The algorithm instantiates $\text{Opt}(1, A) - a_1$ and $\text{Opt}(1, B) - b_1$ and for $i - 2, 3, \dots, n$, it calculates $\text{Opt}(i, A)$ and $Opt(i, B)$ with the use of the formula. This is bounded by $O(n)$ as constant time is used for every $n-1$ iteration.

2.5 Q17

(a) A sequence such as: $1, 4, 2, 3$. The algorithm returns a rising trend $1, 4$, even though the optimal solution is $1, 2, 3$.

(b) We denote $OPT(j)$ to be the longest increasing subsequence over the set $P[j], P[j+1], \ldots, P[n]$, with elements $P[j]$. We can instantiate $OPT(n) = 1$ and $OPT(1)$ denotes the longest rising trend. We want to achieve a solution of $OPT(j)$. Consider the first element $P[j]$ and its subsequent element $P[k]$ for some $k > j$ where $P[k] > P[j]$. From k it is the longest increasing subsequence beginning at $P[k]$; as such this sequence is of length $OPT(k)$, with $P[j]$ the complete sequence $1 + OPT(k)$. We then have derived the following recurrence:

$$OPT(j) = 1 + \max_{k>j:P[k]>P[j]} OPT(k$$

OPT values are returned in decreasing order according to j, bounded by $O(n-j)$ resulting in a running time of $O(n^2)$.