Midterm Solutions

Problem 1 (25 points). Suppose we are given an array $A[1..n]$ with the special property that $A[1] \leq A[2]$ and $A[n-1] \geq A[n]$.

An element $A[x]$ is a *peak* if $A[x] \geq A[x-1]$ and $A[x] \geq A[x+1]$. For example, there are six peaks in the following array:

5	**6**	**6**	2	3	**7**	5	4	**8**	3	3	4	**10**	6

We can obviously find a peak in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a peak in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array must have at least one peak. Think about why this is.]*

Algorithm

```
FindPeak(A,start,end):
    mid = (start-end)/2
    if A[mid-1] <= A[mid] >= A[mid+1]:
        return A[mid]
    elif A[mid-1] >= A[mid]:
        FindPeak(A,start,mid)
    else:
        FindPeak(A,mid,end)
```

Proof

Base Case If the list is of length 3 then, given the special property, $A[1] <= A[2]$ and $A[2] >= A[3]$ since $n = 3$. Since this is also the definition of a peak, $A[2]$ is the peak and returned.

Induction hypothesis Assume that it finds a peak for all lists of lengths 0 to k.

Induction step We have an input of length $k + 1$.

The proof has three cases:

- Case $A[mid - 1] <= A[mid] >= A[mid + 1]$. Clearly the function works correctly.
- Case $A[mid - 1] >= A[mid]$. Since the start of the list is increasing from the first element to to the second, and that there is an increase from the mid to the element before, there has to be an element that is a "peak" in between them. This is like the special property in the problem statement that means there must be a "peak" within the list. Our list, of size $< k$, still has the same special property, that $A[1] <= A[2]$ and $A[n - 1] >= A[n]$.
- Case $A[mid] <= A[mid + 1]$. The last case has to have this be true since if $A[mid - 1] <= A[mid] >= A[mid+1]$ and $A[mid-1] >= A[mid]$ then, $A[mid] <= A[mid+1]$. This is essentially the same as the proof above.

Problem 2 (25 points). You are given a set of n ropes of different lengths $\{l_1, \ldots, l_n\}$. You are supposed to tie these ropes into a single rope. The cost of joining two ropes equals the sum of the lengths of the two ropes, i.e. $C = l_i + l_j$.

Design an algorithm that will incur the total minimum cost, and prove its optimality.

For the algorithm, think of what happens to the set of ropes after you've made a join. For your proof, think of the greedy stays ahead strategy.

Example: If you have 4 ropes of lengths: $\{4, 3, 2, 6\}$, the optimal strategy is the following:

- Connect 2 and 3 $\rightarrow C = 5$. Now you have three ropes of lengths $\{4, 5, 6\}$.
- Connect 4 and 5 $\rightarrow C = 9$. The remaining ropes are $\{9, 6\}$.
- Connect 9 and 6 $\rightarrow C = 15$.

Total cost: $5 + 9 + 15 = 29$

Algorithm The algorithm is very simple. Repeat the following until you have only one rope: Select the two shortest ropes, tie them, incur the cost, put the resulting rope back in the set.

Proof of correctness: We will prove the optimality via induction, and some structural argument. Notice that the joining procedure for your algorithm can be viewed as a binary tree, where each internal node in the tree corresponds to a single knot. We will call such a tree a Tie_tree We will first prove the following structural lemma.

Lemma 1. *Let x and y be the ropes with the least length (with ties broken arbitrarily). There is an optimal* Tie_tree *in which x and y are siblings and their depth is the maximum.*

Proof. Let T be an optimal Tie_tree with depth d. Because, T has either zero or two children, it must have two leaves at the maximum depth that are siblings. Assume these leaves are not x and y, but rather some other pair a and b.

Let T' be the tree if we swapped x and a. The depth of x increases by some amount (say Δ), and the depth of a decreases by the same amount. Threrefore,

$$\mathsf{cost}(T') = \mathsf{cost}(T) - (\ell_a - \ell_x)\Delta.$$

But by assumption, x is one of the ropes with the least length while a is not. This implies $\ell_a \geq \ell_x$, thus swapping x and a does not increase the total cost of the tree. Thus, T' is an optimal tree. Similarly, swapping y and b. Thus, doing both swaps yeild an optimal tree with x and y as siblings, and at maximum depth.

□

Theorem 2. *The greedy algorithm is the optimal rope tying scheme.*

Proof. **Base case:** If there are at most two ropes. The lemma is trivially true.

Induction hypothesis: For any set of ropes $\ell_1, \cdots, \ell_k$, our greedy algorithm gives the optimal tying stratrgy.

Induction step: Let $\ell_1, \cdots, \ell_n$ be the lengths of the original n ropes. Without loss of generality, let ℓ_1 and ℓ_2 be the smallest two lengths. By Lemma 1, we know that there is a optimal Tie_tree that has ℓ_1 and ℓ_2 as siblings. Let T be such a tree. In T, clearly ℓ_1 and ℓ_2 are tied to form $\ell_{n+1} = \ell_1 + \ell_2$. Now consider, the set of ropes $\ell_3, \cdots, \ell_{n+1}$. Notice, that number of ropes in this new set is $n-1$. By our Induction hypothesis, our greedy strategy is optimal for this set of ropes. Call the tree generated by our greedy strategy as T'. Now, we can write the cost of the tree T in terms of the cost of T'. Let depth define the depth of a node in the tree T, and let d be the maximum depth in the tree T.

$$\begin{aligned}
\mathsf{cost}(\mathsf{T}) &= \sum_{i=1}^{n} \mathsf{depth}(\ell_i) \cdot \ell_i \\
&= \left(\sum_{i=3}^{n+1} \mathsf{depth}(\ell_i) \cdot \ell_i \right) + \ell_1 \cdot \mathsf{depth}(\ell_1) + \ell_2 \cdot \mathsf{depth}(\ell_2) - \ell_{n+1} \cdot \mathsf{depth}(\ell_{n+1}) \\
&= \left(\sum_{i=3}^{n+1} \mathsf{depth}(\ell_i) \cdot \ell_i \right) + (\ell_1 + \ell_2) \cdot d + \ell_{n+1}(d-1) \\
&= \left(\sum_{i=3}^{n+1} \mathsf{depth}(\ell_i) \cdot \ell_i \right) + \ell_1 + \ell_2
\end{aligned}$$

Since, the cost of our greedy strategy T' is optimal for $\ell_3, \cdots, \ell_{n+1}$ (by the induction hyptothesis), therefore the following is true:

$$\mathsf{cost}(\mathsf{T}) \geq \mathsf{cost}(\mathsf{T'}) + \ell_1 + \ell_2.$$

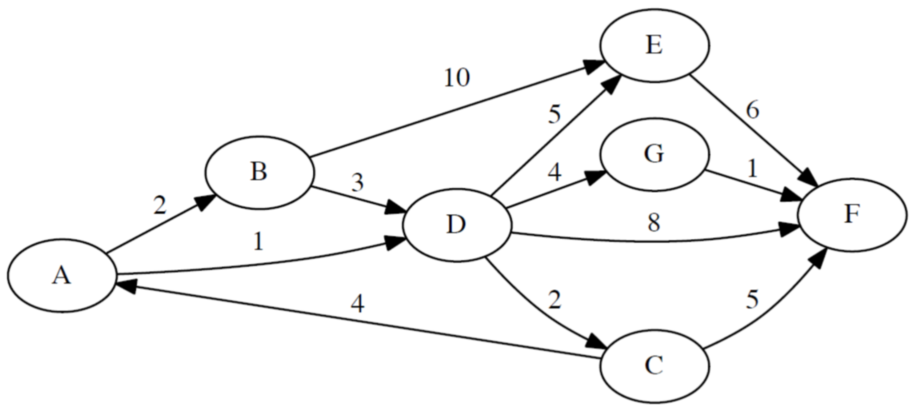
Furthermore, notice that our greedy strategy incurs a cost of $\mathsf{cost}(T') + \ell_1 + \ell_2$ on the original set of ropes $\ell_1, \cdots, \ell_n$. Hence, our greedy strategy is optimal. □

A wrong approach In a lot of cases, there was this argument using greedy stays ahead/exchange argument. The problem here is that if you replace a solution from the optimal set with one from the greedy set, you lose the optimality guarantee of the following tieings. It might be possible to use a similar argument to the one from Lemma 1, but the tree structure must be built first. I have not seen such an argument.

Problem 3 (25 points). Use Dijkstra's algorithm to find the shortest path from node A to every other node.

- (12.5 points) Complete the table showing how Dijkstra's algorithm computes the shortest paths.
- (12.5 points) Give the shortest paths from node A to every other node. List all vertices along each of the paths.

Complete the table below so that it shows each iteration of Dijkstra's Algorithm.



Iteration	Set of vertices S	Current Distance D						
1		A	B	C	D	E	F	G
	$\{A\}$	0	2	∞	1	∞	∞	∞

Table 1: Part a)

Iteration	Set of Vertexes	A	B	C	D	E	F	G
1	{A}	0	2	∞	1	∞	∞	∞
2	{A,D}	0	2	3	1	6	9	5
3	{A,D,B}	0	2	3	1	6	9	5
4	{A,D,B,C}	0	2	3	1	6	8	5
5	{A,D,B,C,G}	0	2	3	1	6	6	5
6	{A,D,B,C,G,E}	0	2	3	1	6	6	5
7	{A,D,B,C,G,E,F}	0	2	3	1	6	6	5

Table 2: Part b)

Nodes	S.Path	Length
B	A->B	2
D	A->D	1
C	A->D->C	3
E	A->D->E	6
G	A->D->G	5
F	A->D->G->F	6