CS 577: Introduction to Algorithms — Solutions to Midterm 1

Problem 1 [30 points]

Assume that you are given a fully parenthesized numerical expression with positive and negative numbers and like the following

$$\Big([+3] + \big([+1] + [-7]\big) \Big) + \Big(\big([-8] + [-3]\big) + \big([+1] - [+4]\big) \Big)$$

where the only possible operations between two numbers are addition $+$ and subtraction $-$. Notice that to avoid confusion we use brackets, e.g., $[-7]$ to denote a number together with its sign. Now consider the same expression with the operators $+$ and $-$ missing, i.e, replaced by $?$

$$\Big([+3] \; ? \; \big([+1] \; ? \; [-7]\big) \Big) \; ? \; \Big(\big([-8] \; ? \; [-3]\big) \; ? \; \big([+1] \; ? \; [+4]\big) \Big)$$

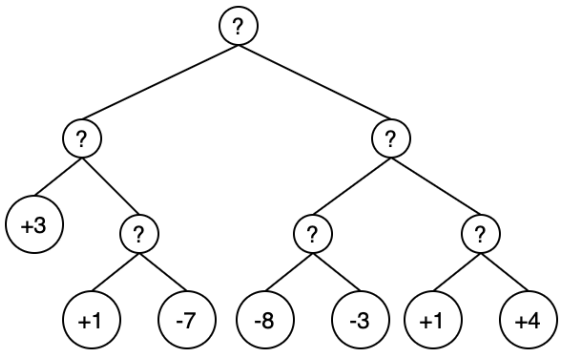
Your goal is figure out the **maximum and minimum** possible value of the expression when $?$ are replaced with either $+$ or $-$. For example, to maximize the above expression we have replace the $?$ as follows

$$\Big([+3] + \big([+1] - [-7]\big) \Big) - \Big(\big([-8] + [-3]\big) - \big([+1] + [+4]\big) \Big) = 27\,.$$

On the other hand to minimize it we have to replace them as

$$\Big([+3] - \big([+1] - [-7]\big) \Big) + \Big(\big([-8] + [-3]\big) - \big([+1] + [+4]\big) \Big) = -21\,.$$

In general you will be given an arithmetic expression with n numbers structured as a **binary tree** with root r where the leaves correspond to the numbers. For example, the input may be the following tree corresponding to the expression above:



In this case, the output of your algorithm should be $(27, -21)$.

(a) The following is an outline of a recursive algorithm for this problem. The main function call is to $\text{FindMaxAndMin}(r)$ where r is the root of the tree. Fill in the missing parts.

Algorithm 1: FindMaxAndMin(v)
/* Returns the maximum and minimum possible values of the expression. */

if *v is a leaf node* **then**
  Return (value of v, value of v).
else
  Let l (resp. r) be the left (resp. right) child of v.
  $(L_{\min}, L_{\max}) = \text{FindMaxAndMin}(l)$
  $(R_{\min}, R_{\max}) = \text{FindMaxAndMin}(r)$
  $\text{maxValue} = \max(L_{\max} + R_{\max}, L_{\max} - R_{\min})$
  $\text{minValue} = \min(L_{\min} + R_{\min}, L_{\min} - R_{\max})$
  Return $(\text{maxValue}, \text{minValue})$

(b) State the running time of the algorithm in terms of the number n of numbers in the expression.

Since we spawn one recursive call for each node of the tree the runtime is linear in the total number of nodes of the tree representation of the expression and therefore the total runtime is $O(n)$, linear in the number of numbers n in the expression.

(c) Prove the correctness for your algorithm.

We will use induction on the height of the tree representation of the expression.
Base Case: Consider an expression consisting of a single number (in that case the tree only consists of a single node). The algorithm returns the value of that node which is the correct answer.
Inductive Hypothesis: Assume that $\text{FindMaxAndMin}(v)$ correctly returns the minimum and maximum value of any expression whose tree representation has height at most k.
Step: Consider an expression whose tree representation has height $k+1$. By the inductive hypothesis we have that $(L_{\min}, L_{\max}) = \text{FindMaxAndMin}(l)$ and $(R_{\min}, R_{\max}) = \text{FindMaxAndMin}(r)$ are correct since the corresponding trees rooted at l, r have heigh at most k. We have that the maximum possible value of the whole expression is

$$\max(L_{\max}+R_{\max}, L_{\max}+R_{\min}, L_{\min}+R_{\max}, L_{\min}+R_{\min}, L_{\max}-R_{\max}, L_{\max}-R_{\min}, L_{\min}-R_{\max}, L_{\min}-R_{\min})$$

However, we have that $L_{\max}+R_{\max}$ is larger than or equal to $L_{\max}+R_{\min}, L_{\min}+R_{\max}, L_{\min}+R_{\min}$ and also that $L_{\max}-R_{\min}$ is larger than or equal to $L_{\max}-R_{\max}, L_{\min}-R_{\max}, L_{\min}-R_{\min}$. Therefore, our algorithm correctly computes the maximum value. The proof for the minimum value is similar.

Grading comments

Part (a)

Many solutions had minor errors on their base case. For most algorithms for problem 1, the base case of $n = 1$ is necessary. Consider an tree of three values, i.e. $n = 3$. The algorithm will recurse on the left and right child and with one of the children having only one node, it is necessary for the algorithm to deal with a single node base case. This is what algorithms that started with a base case of $n = 2$ missed.

Part (b)

The most common mistake was a hasty analysis of the runtime. If one has a completely balanced binary tree in mind, it may be easy to think that the runtime is similar to binary search or mergesort. However, even with a balanced tree, the runtime still remains to be $O(n)$.

Part (c)

- A base case of $n = 1$ was necessary. Writing $n = 1$ case is trivial without any explanation did not merit any credit.
- Most approaches to prove the correctness required strong induction. For example, if one is doing induction on trees of height h, you must assume the algorithm works for trees of height 1 to $h - 1$ for your inductive hypothesis. This is necessary because a tree of height h may have a left child of height $h - 1$ and have a right child of height 1. To guarantee the algorithm's correctness on these children, one must use strong induction.
- There are cases where strong induction may not be necessary. One may induct on the level of the nodes l and prove that the algorithms works for all nodes on each level. Although this does not need strong induction, it's necessary to mention the case where the node in level l is a leaf since this node may not have children.
- An incorrect way to use induction is to go from k values to $k + 1$ values by adding a node in specific locations. This argument can work by using structural induction and carefully constructing trees of $k + 1$ values from k values, but most approaches with this argument were not justified sufficiently. An easier way would be to consider a tree with $k + 1$ values and try to reduce it to k values by combining two leaves together to one value.

Problem 2 [30 points]

You are playing a videogame in which your character is attacked by a fleet of n enemy monsters. Each monster i inflicts damage d_i to your character every second it remains alive. To fight back, you start attacking monsters one by one until they are all defeated. It takes t_i seconds to defeat monster i and, during that time, both that monster as well as every other monster alive is constantly attacking you. Your goal is to find the order of attacks that minimizes the total damage you receive until all monsters are defeated.

Suppose there are $n = 2$ monsters with $(d_1, t_1) = (1, 2)$ and $(d_2, t_2) = (1, 1)$. It takes 3 seconds to defeat both monsters.

- Consider attacking monster 1 before monster 2. In the first 2 seconds until monster 1 is defeated, you receive 4 points of damage (2 from each monster). In the last second, only monster 2 attacks inflicting 1 additional damage point. In total you receive 5 points of damage.
- Consider attacking monster 2 before monster 1. In the first second until monster 2 is defeated, you receive 2 points of damage (1 from each monster). In the last 2 seconds, only monster 1 attacks inflicting 2 additional damage points. In total you receive 4 points of damage.

Thus, in this example, the optimal order is $[2, 1]$. For another example, consider again two monsters with $(d_1, t_1) = (3, 2)$ and $(d_2, t_2) = (1, 1)$. The optimal order in this case is $[1, 2]$ which leads to $2 \cdot 4 + 1 = 9$ points of damage.

(a) Design a greedy algorithm that finds the optimal order to attack the monsters.

The optimal order to attack the monsters is to sort the by decreasing ratio of damage d_i over "hitpoints" t_i, i.e., attack the monster with largest ratio d_i/t_i first.

(b) State the running time of your algorithm. No explanation is required.

The runtime of this approach is equal to the runtime to sort the tuples (t_i, d_i) according to d_i/t_i and this takes $O(n \log n)$ time using Mergesort.

(c) Prove the correctness of your algorithm.

To prove correctness we will use an exchange argument. Assume that the optimal ordering of the monsters is the following:

$$(t_1, d_1), (t_2, d_2), \ldots, (t_j, d_j), (t_{j+1}, d_{j+1}), \ldots, (t_n, d_n)$$

The total damage that we take assuming this order is

$$D = t_1 \sum_{i=1}^{n} d_i + t_2 \sum_{i=2}^{n} d_i + \ldots + t_j \sum_{i=j}^{n} d_i + t_{j+1} \left(-d_j + \sum_{i=j}^{n} d_i \right) + \ldots + t_n d_n$$

Assume now that $d_{j+1}/t_{j+1} \geq d_j/t_j$, i.e., the damage over hitpoints of monster $j+1$ is higher (or equal) than that of monster j. We will show that we can swap monsters j and $j+1$ without increasing the total damage that we take. The ordering after we swap monsters j and $j+1$ is

$$(t_1, d_1), (t_2, d_2), \ldots, (t_{j+1}, d_{j+1}), (t_j, d_j), \ldots, (t_n, d_n)$$

The total damage of the new ordering is

$$D' = t_1 \sum_{i=1}^{n} d_i + t_2 \sum_{i=2}^{n} d_i + \ldots + t_{j+1} \sum_{i=j}^{n} d_i + t_j \left(-d_{j+1} + \sum_{i=j}^{n} d_j \right) + \ldots + t_n d_n$$

We have

$$D - D' = -d_j t_{j+1} + d_{j+1} t_j \geq 0\,,$$

where for the last inequality we used the fact that $d_{j+1}/t_{j+1} \geq d_j/t_j$. Therefore, we can perform such swaps of consecutive monsters to the optimal solution without increasing the damage taken. Thus, we can perform all the required swaps until the optimal solution is sorted in the same way as our greedy solution, i.e., by decreasing ratio of d_i/t_i. This means that the damage achieved by the greedy solution is equal to that of the optimal solution.

Grading comments

Part (a)

One of this problem's challenges was figuring out the optimal order in which to sort the monsters. The most common mistake here was sorting by some criterion other than decreasing d_i/t_1. We present some counterexamples for some of the most common criteria:

- Decreasing d_i. For monsters $(d_1, t_1) = (3, 1), (d_2, t_2) = (6, 3)$, the optimal order is to defeat monsters 1 and 2 in this order, but this criterion puts monster 2 first.
- Decreasing $d_i - t_i$. For monsters $(d_1, t_1) = (10, 20), (d_2, t_2) = (5, 12)$, the optimal order is to defeat monsters 1 and 2 in this order, but this criterion puts monster 2 first.
- Decreasing $\sum_{j \neq i} t_i \cdot d_i$. For monsters $(d_1, t_1) = (1, 1), (d_2, t_2) = (2, 3), (d_3, t_3) = (3, 5)$, the optimal order is to defeat monsters 1, 2 and 3 in this order, but this criterion puts monster 2 first.

Part (b)

The most common mistake here was to claim an $O(n^2)$ algorithm (usually a variation of BubbleSort or InsertionSort) executed in time $O(n \log n)$ because it is sorting the input. Remember that not all sorting algorithms run in time $O(n \log n)$.

Part (c)

Here, the challenge was to come up with a correct exchange argument to prove optimality of the decreasing d_i/t_i ordering. Some common mistakes here were the following:

- Making an exchange involving two monsters that are not adjacent without accounting for the change in damage for the monsters that are in between. When considering two adjacent monsters that form an inversion in relation to the greedy order, we the algebra involved in the proof that shows this exchange does not make the solution worse is much simpler than in the case where they are further apart. The reason for this is that there is no change in the damage received from other monsters, and we can focus on the change in damage received by just the two involved in the exchange. If we swap two monsters that are further apart, then the damage received by monsters that are in between the two changes, and the analysis gets much more complicated.
- Assuming that the optimal solution is somehow close to the greedy solution. Note that we cannot assume any structure of an optimal solution. This means, for example, that we cannot assume that an optimal solution is equal to the greedy solution modulo a single inversion. If we do this, our proof is no longer able to account for optimal solutions that are not as close to the greedy solution (and in principle these could exist).
- Missing explanation for why performing an exchange leaves the solution no worse than before. The best way to do this is by presenting some algebra comparing the costs before and after the exchange. Note that if we do not present an explanation for this, then we could convince ourselves of the correctness of any algorithm for the problem, since with enough exchanges we are always able to transform any solution into any other.
- Focusing only on the case $n = 2$. Many solutions derived the greedy order by analyzing the case with two monsters but did not extend this to the case $n > 2$. While this analysis contains many of the elements that are present in a full exchange argument, we still need to argue that the order for general n is optimal using an exchange argument involving (for example) two adjacent monsters that are out of order in an optimal solution.
- Proving correctness by presenting examples. Remember that a proof of correctness should guarantee that the algorithm produces an optimal solution for any given input. In this case, it is not enough to argue correctness by presenting some examples where the algorithm performs well.

- Providing only a high-level intuition for why the decreasing d_i/t_i order is optimal. While high-level ideas are good for coming up with greedy criteria, we should be careful when using those to prove correctness. For example, it may make sense that killing the monster with the highest value of d_i/t_i first is optimal because we are removing a monster that, at the same time, has high damage and low time-to-kill values. Note, however, that a similar argument applies to ordering the monsters by $d_i - t_i$, and we know this order is not optimal. A good rule of thumb is to ask yourself whether the proof of correctness could be adapted to "work" for other, non-optimal solutions. If this is the case, then you should provide more details for your proof.

Some less common, but repeated mistakes were the following:

- Some answers seemed to incorrectly assume that the solutions we need to consider are subsets of monsters instead of orderings of those, and tried proving correctness in a manner similar to the solution for Problem 1 on Homework 4. When proving correctness using exchange arguments, we should be careful that our exchanges make sense in relation to the types of solutions for the problem.

- Trying to prove correctness of the greedy order directly by induction. The difficulty here is the following: say the monsters are already sorted using the greedy criterion, even if we (by the inductive hypothesis) assume that for the first i monsters the optimal solution is to order then by decreasing values of d_i/t_i, we still need to argue that the $(i+1)$-th monster should be killed last to get an optimal solution for $i+1$ monsters. This is not trivial to do, since there are a total of $i+1$ locations where this monster could be added to the solution, and we need to somehow argue that killing it last is better than all other options.

We talk about greedy-stays-ahead arguments separately. As far as we know, there is no greedy-stays-ahead argument that works for proving correctness for this problem. Remember that greedy-stays-ahead arguments work by identifying a quality measure, proving that this measure implies optimality and arguing using induction that partial solutions produced by the greedy algorithm "stay ahead" in relation to this measure. here are some examples of quality measures that were attempted, and an explanation of why they fail for this problem.

- At every moment $0 \leq t \leq \sum_{i=1}^{n} t_i$ (the time required to defeat all monsters), the greedy solution takes less accumulated damage than any other solution. The problem with this is that this claim is false. Consider the input $(d_1, t_1) = (4, 3), (d_2, t_2) = (1, 1)$: The greedy solution takes $2 \cdot (4+1) = 10$ points of damage in the first 2 seconds, while the solution that kills monster 2 first takes $1 + 2 \cdot 4 = 9$ points of damage in the same interval.

- When defeating each monster, the greedy solution takes less accumulated damage than any other solution. Again, this claim is false, and the same counterexample as before works to show that this is the case.

Problem 3 [40 points]

Suppose you are given an **(undirected, unweighted)** graph $G = (V, E)$. The graph has colored edges, and so you are also given a partition of the edges $E = R \cup B$ into two sets, the red edges and the blue edges.

(a) Give a polynomial time algorithm to compute a spanning tree of G with as many red edges as possible. Prove your algorithm is correct and analyze its running time.

Algorithm: The algorithm is simply to run a minimum spanning tree algorithm (say, Prim's algorithm) with every red edge having weight 0 and every blue edge having weight 1.

Proof of Correctness: The correctness of our algorithm follows closely from that of an MST algorithm.

Suppose our algorithm returns a tree with k red edges. We claim there cannot exist a MST tree with $k+1$ edges. If such a tree did exist, it would have $n-k-2$ blue edges and $k+1$ red edges and hence a weight of $n-k-2$. But the tree returned by the MST algorithm would have k red edges and $n-k-1$ blue edges for a total weight of $n-k-1$. This contradicts the correctness of the MST algorithm. Therefore, whenever our algorithm returns a MST with k red edges, there cannot exist an MST with more than k red edges. Since our algorithm explicitly finds a spanning tree with k red edges (again by the correctlness of the MST algorithm we use as a subroutine), our algorithm correclty returns a spanning tree with the maximum number of red edges. Clearly, the runtime is the same as whichever MST algorithm we use. We'll say we use Prim's algorithm, which we've seen can run in $O(m \log n)$.

(b) Given two spanning trees T_1 and T_2 of G, one with r_1 red edges and one with r_2 red edges, where $r_1 \leq r_2$, it is always possible to construct a spanning tree with r red edges for any $r \in [r_1, r_2]$. Give a polynomial time algorithm that takes as input the trees T_1 and T_2 and the parameter $r \in [r_1, r_2]$ and constructs such a spanning tree T with r edges. Prove the correctness of your algorithm.
Hint: Try constructing a sequence of trees leading from T_1 to T_2 where consecutive trees differ by one edge.

Following the hint, we'll create a sequence of tree leading from T_1 to T_2 where consecutive trees differ by one edge. Consider the following procedure to do a step of the transformation.

1. Take an edge e that is in tree T_1 and not in tree T_2
2. Add edge e to T_1
3. Since T_1 was a spanning tree, this creates a cycle, and there must be some edge on that cycle that is not in T_2 (since T_2 is also a tree). Remove that edge from T_1.

We will simply need to repeat this procedure until the resulting tree has r red edges, and then return the resulting tree.

Proof of Correctness: First, we want to prove that the subroutine we've defined actually works.

Claim 1. *As long as $T_1 \neq T_2$, we can execute the three steps of the subroutine. When the three steps are complete, T_1 is still a spanning tree.*

Proof. First, if $T_1 \neq T_2$, and both T_1 and T_2 are spanning trees, then there must be an edge in T_1 that is not in T_2, and hence we can execute the first step. Second, since T_1 was a spanning tree before we added the edge, that must create a cycle. Third, some edge on the cycle in T_1 must not be in T_2: assume, for the sake of contradiction that every edge on the cycle is in T_2; then there is a cycle in T_2 which contradicts the fact that T_2 is a tree. Finally, after the edge on the cycle is removed, T_1 is again a tree, since it again has exactly $n-1$ edges and is still connected (since the only edge removed was on a cycle). □

Now, we can consider what would happen if we ran the subtroutine repeatedly.

Claim 2. *If T_1 and T_2 are each spanning trees where $T_1 \neq T_2$, then after at most $n-1$ iterations of the subroutine, $T_1 = T_2$.*

Proof. In every iteration of the interpolation steps, we add an edge to T_1 that is in T_2 but not in T_1, and we never remove an edge from T_1 that is in also T_2; thus the number of edges in T_2 but not in T_1 decreases at every iteration of the subroutine. Therefore, after at most $n-1$ iterations of the interpolation steps, every edge in T_2 is also in T_1. Since T_2 is a spanning tree and T_1 is also a tree, we conclude that $T_2 = T_1$.. □

Now, we've established that we can contruct a sequence of trees from T_1 to T_2, each differing by an edge, as in the hint. The final step of the proof just needs to argue that one of the trees in the sequence will have r red edges.

Claim 3. *If T_1 and T_2 are each spanning trees, and initially T_1 has r_1 red edges and T_2 has r_2 red edges, and if we repeat the subroutine until $T_1 = T_2$, then then after one of these iterations, the tree T_1 exactly r red edges for any $r \in [r_1, r_2]$.*

Proof. The claim holds because the number of red edges in T_1 changes by at most 1 with each run of the subroutine. This is clearly true: either the removed and added edges are different colors and the number of red edges changes by 1, or they are the same and the number of red edges doens't change. But since there are initially $r_1 \leq r$ red edges in T_1 and are ultimately $r_2 \geq r$ red edges in T_1, there must be some intermediate point where T_1 has r red edges. □

Runtime Analysis: We repeat the subroutine at most $n-1$ times. Each iteration requires: (i) finding an edge to add to T_1, which takes at most n time; and (ii) finding an edge on the cycle to remove from T_1, which takes at most n time. Thus, the total runing time is at most $O(n^2)$, which is polynomial.

(c) Suppose $n = |V|$ is odd, meaning that any spanning tree of G will have an even number of edges. Describe how to use the algorithms you developed in the previous parts to efficiently compute a spanning tree of G with an equal number of red and blue edges if one exists.
You do not need to prove correctness of your algorithm or analyze its runtime.

Run the algorithm from a) to compute the spanning tree T_R with the maximum number of red edges and the spanning tree T_B with the maximum number of blue edges. If T_R has fewer than $n/2$ red edges or T_B has fewer than $n/2$ blue edges, return that no tree exists. Otherwise, call the algorithm from b) on T_R and T_B with parameter $r = n/2$, returning the resulting spanning tree.