Problem 3
Let $x_1, x_2, \ldots, x_n$ be a sequence of integers (not necessarily positive and not sorted).

A. (16 points) Design an O(n) algorithm to find the subsequence $x_i, \ldots, x_j$ (of consecutive elements) such that the product of the numbers in it is maximum over all consecutive subsequences. The product of the empty subsequence is defined as 1. You need to prove its correctness as well.

B. (4 points) analyze the time complexity of your algorithm.
Example: If the original sequence is -4 -3 4 1 -5 the answer is (-3)×4×1×(-5) or +60.

Solution

Some examples:

- If all the elements are positive: return the whole sequence.
- If number of negative elements is even: whole sequence.
- If there is only one zero: 1, 3, 4, **0**, 6, -1, -2

A. Algorithm
Show the sequence with $a_1, a_2, a_3, \ldots, a_n$

If we define S as:
$S[0] = 1$
$S[1] = a_1$
$S[2] = a_1 \times a_2$
…
$S[k] = a_1 \times a_2 \times a_3 \times \ldots \times a_k$
the product of a subsequence $a_i, \ldots, a_j$ would be $\frac{s[j]}{s[i]}$

- Starting with a_0, calculate s[i] values as long as a_i is not zero
- In the meantime, keep track of these four values:
  - max_positive, min_positive
  - max_negative, min_negative
    - comparison is done based on absolute value

- when encountered zero:
  - calculate $\frac{\max\_positive}{\min\_positive}$ and $\frac{\max\_negative}{\min\_negative}$ if they exist
  - for example: -1, 1, 2 does not have any positive values
  - maintain a tuple (value, start, end) that always refers to the subsequence with maximum desired value.
    - Update it (Only after seeing zero)
  - Start calculating s[i] from the index after 0, forgetting about previous values.

- Return the desired subsequence at the end

Proof
We need to prove if the subsequence a_i, …, a_j is optimal, the algorithm outputs it.

Based on our modeling, s[j]/s[i] will be considered in the comparisons, and will be found if it has the maximum cumulative product, unless these cases happen:

- There is a zero among a_i, …, a_j
    - Contradiction: the max cannot be zero

B. Time Complexity
For each step of algorithm, we analyze the taken actions.

- For each index:
    - If non-zero:
        - calculating s[i]: O(1)
        - updating the four max and min variables: O(1)
    - If it is zero:
        - Division of max/min values + : O(1)
- So, we have O(1) number of comparisons per index.

Overall: n steps of O(1) operations: O(n)

Problem 5:
Let G = (V, E) be a connected undirected weighted graph with **n** vertices. Assume for simplicity that the weights are positive and distinct. Let **e** be an edge of G. Denote by T(**e**) the minimum spanning tree MST of G that has minimum cost among all spanning trees of G that contain **e**.

a. (8 points) Design an $O(n^2)$ algorithm that finds T(**e**) given a graph G and an edge **e**.
b. (4 points) Prove its correctness.
c. (8 points) Design an algorithm to find T(**e**) for all edges **e** in E. The algorithm should run in time $O(n^2)$.

(You do **not** need to re-prove that finding an MST on a graph can be done in $O(n^2)$ time – we already proved that in class).

Solution

a.
Algorithm
Similar to Prim's algorithm, start with S={u, v}, where e=(u,v) and expand it over time.

- Repeat this process
  - At each step, consider all the edges connecting nodes in S and V-S
  - Pick the vertex that is connected with min weight edge
  - Add the node to set S

It is also possible to pack u and v and create another edge representing them: proof will be simplified

Time complexity is $O(n^2)$: proved previously in class

b.
Proof
Edge e must be in the Tree anyways. Except for that, we prove all the added edges are essential.

- We slightly modify **Cut-Property**: if e'=(w,t) is the min edge between a set of nodes S and V-S, it must be in any **T(e)**.
- Proof
  - Assume there is a T(e) that does not contain e'=(w, t)
  - Add e' to that graph
    - There was already a path between u and v on that tree
    - That path plus e' forms a new cycle
  - Traverse this cycle. In addition to e', there must be another edge connecting S and V-S.
    - Its weight must be greater than the weight of e'
    - **That edge is definitely not e**
      - recall: e was the edge we wanted to be in the resulting tree
      - if it was e, we could not remove it from the tree
  - Remove that edge
    - The result is still connected
      - Because removing an edge from a cycle does not disconnect any nodes
  - The weight of the tree is decreased, which contradicts with the fact that it was a **Minimum** Spanning Tree.

The edges added by us have the modified cut-property: Thus we have an MST

c.

Algorithm

- Get the MST of G
- For each node e:
  - If e is in the MST: T(e) = weight of MST
  - If not:
    - add e to the MST, it forms a cycle
    - remove the edge with highest weight on that cycle
      - if e=(u,v): max edge on the path from u to v

On the MST, we want to compute the edge with maximum weight on the path between each pair of nodes (u, v) for further use in the algorithm.

- Run BFS with **s** as the root.

Time Complexity

- Getting the MST: $O(n^2)$
- Computing max edge on u-v paths: n times O(n): $O(n^2)$

Proof

<u>Side note: forcing edge e to be in MST is like changing the weight of e to a small value ε and run MST algorithm.</u>

- Because the minimum edge is always in the MST
  - Use Cut-Property for proof

We want to prove that running Kruskal's algorithm on modified graph gives the same output as our algorithm (just removing the max_edge in the cycle).

- Consider the operations of running Kruskal on the original graph (weight of e is not modified)
  - select e_1, discard e_2, …, select e_{n-1}
    - select: add to MST
    - discard: adding the edge causes a cycle
- Places where edge e can affect the runtime
  - removal of edge e_k
    - e_k: highest weight on cycle created after adding e to MST
  - An edge was previously discarded due to existence of a path, but that path is broken because if e
    - Broken path happens if an edge (v_1, v_2) was previously selected but discarded this time, because the existence of e has already made a path from v_1 to v_2
      - There is a path from v_1 to v_2
      - Hence, the edges that were discarded because they formed a path including (v_1, v_2) will be discarded in the new run of algorithm too