Fall 2022 Q-exam — CSCE 750 (Algorithms) — Solutions

1. **(Solving a Recurrence)**

   Let $T(n)$ be any positive-valued function defined for all integers $n \geq 1$ by the following recurrence:

   $$T(n) = 2T(n^{2/3}) + T(n-1) + n^2 \ .$$

   Then $T(n) = \Theta(n^k)$ for some real constant $k > 0$. Find k, and justify your choice using the substitution method. You may assume that any implicit floors or ceilings are of no consequence.

   Answer: $k = 3$.

   Justification by the substitution method:

   For the upper bound, assume n is sufficently large and that $T(m) \leq cm^3$ for all $m < n$, where c is a constant to be determined. Then

   $$\begin{aligned} T(n) &= 2T(n^{2/3}) + T(n-1) + n^2 \\ &\leq 2c(n^{2/3})^3 + c(n-1)^3 + n^2 \\ &= 2cn^2 + c(n^3 - 3n^2 + 3n - 1) + n^2 \\ &= cn^3 - c(n^2 + 3n - 1) + n^2 \\ &\leq cn^3 \end{aligned}$$

   provided $-c(n^2 + 3n - 1) + n^2 \leq 0$, or equivalently,

   $$c \geq \frac{n^2}{n^2 + 3n - 1} \ .$$

   This is true for all sufficiently large n if c is chosen so that $c > 1$.

   For the lower bound, assume n is sufficently large and that $T(m) \geq cm^3$ for all $m < n$, where $c > 0$ is a constant to be determined. Then

   $$\begin{aligned} T(n) &= 2T(n^{2/3}) + T(n-1) + n^2 \\ &\geq 2c(n^{2/3})^3 + c(n-1)^3 + n^2 \\ &= 2cn^2 + c(n^3 - 3n^2 + 3n - 1) + n^2 \\ &= cn^3 - c(n^2 + 3n - 1) + n^2 \\ &\leq cn^3 \end{aligned}$$

   provided $-c(n^2 + 3n - 1) + n^2 \geq 0$, or equivalently,

   $$c \geq \frac{n^2}{n^2 + 3n - 1} \ .$$

   This is true for all sufficiently large n if c is chosen so that $c < 1$.

   An alternate shortcut for the lower bound is to immediately drop the first term on the right-hand side of the recurrence. This still gives a tight asymptotic bound as long as $c < 1/3$.

2. **(Maximal Noncontiguous Subsequence)** You are given a sequence $S := \langle a_1, \ldots, a_n \rangle$ of $n > 0$ integers, each of which could be positive, negative, or zero. Say that a subsequence of S is *good* if it does not include any two consecutive elements of S. For example, if $n = 6$, then $\langle a_1, a_3, a_5 \rangle$ and $\langle a_2, a_6 \rangle$ are good, but $\langle a_2, a_4, a_5 \rangle$ and $\langle a_1, a_2, a_6 \rangle$ are not. **Describe** an algorithm that on input S returns a good subsequence of S the sum of whose elements is as large as possible. Your description should include enough detail that an intelligent programmer can implement it. For full credit, your algorithm should run in worst-case time $O(n)$. (As usual, assume each integer arithmetic operation takes $O(1)$ time.)

   Explain why your algorithm works, in enough detail to convince an intelligent but skeptical reader that it is correct.

   Answer: For $0 \leq i \leq n$, let S_i be the length-i prefix $\langle a_1, \ldots, a_i \rangle$ of S (and so $S = S_n$), and let m_i be the largest possible sum of a noncontiguous subsequence of S_i (so our solution subsequence has sum m_n). It is clear, then, that

$$\begin{aligned} m_0 &= 0 && \text{(sum of the empty sequence)} \\ m_1 &= \max\{0, a_1\} \\ m_k &= \max\{m_{k-1}, m_{k-2} + a_k\} && \text{(for } 2 \leq k \leq n\text{).} \end{aligned}$$

   We use dynamic programming to compute $m_1, \ldots, m_n$ while keeping track of the index r_i of the rightmost element of an optimal subsequence for each S_i.

```
On input A[1...n] // A[k] holds a_k
    ALLOCATE integer arrays M[0...n] and R[0...n] // M[k] holds m_k and R[k] holds r_k
    M[0] := 0
    R[0] := 0
    IF S[1] > 0 THEN
        M[1] := A[1]
        R[1] := 1
        ℓ := 1 // ℓ holds the length of the subsequence to be returned
    ELSE
        M[1] := 0
        R[1] := 0
        ℓ := 0
    END-IF
    FOR k := 2 TO n DO
        IF M[k-1] < M[k-2] + A[k] THEN
            M[k] := M[k-2] + A[k]
            R[k] := k // a_k is the rightmost element of an optimal subsequence of S_k.
            ℓ := ℓ + 1
        ELSE
            M[k] := M[k-1]
            R[k] := R[k-1]
        END-IF
    END-FOR
    ALLOCATE an array B[1...ℓ]
```

```
k := R[n]
FOR i := ℓ DOWNTO 1 DO
    B[i] := A[k]
    k := R[k − 1]
END-FOR
RETURN B
```

I also believe a divide-and-conquer approach might work for this problem: divide the sequence in half at the middle, recurse on the left half and the right half, then combine results. It seems that to do this properly, each call to the procedure on a subarray returns four optimal subsequences: one that uses both ends of the subarray; one that uses the left end but not the right; one that uses the right end but not the left; and one that uses neither end. Combining these sequences from the two recursive calls then hopefully takes $O(1)$ time, so the time $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(1)$, which implies $T(n) = O(n)$ by the Master Theorem. The combination step is tricky, though, since one does not have time to make copies of subsequences. I'm not sure if $O(1)$ time is possible for this—maybe organize the optimal subsequence elements in some kind of tree structure? (One can certainly compute the optimal *sum* in $O(n)$ time with this approach.)

3. **(Dynamic Minimum Spanning Tree)** Let $G := (V, E)$ be a connected graph with vertex set $V := \{v_1, v_2, \ldots, v_n\}$ and with edge weight function $w : E \to \mathbb{R}$. Let (V, T) be a minimum spanning tree (MST) of G with respect to w. For any edge $e := \{v_i, v_j\} \in E$ and real number δ, define an altered weight function w' obtained by adding δ to $w(e)$, that is, for any $e' \in E$,

$$w'(e') = \begin{cases} w(e') + \delta & \text{if } e' = e, \\ w(e') & \text{if } e' \neq e. \end{cases}$$

You may assume that all edge weights are pairwise distinct (with respect to both w and w').

Given G, (V, T), e, and δ as input,

(a) **describe** an algorithm that finds an MST of G with respect to w', assuming $e \in T$ and $\delta > 0$.

(b) **describe** an algorithm that finds an MST of G with respect to w', assuming $e \notin T$ and $\delta < 0$.

You'll get 80% credit for either one and 100% credit for both. High-level descriptions are enough, provided they are precise enough for an intelligent programmer to implement them without guesswork. You may assume that both G and (V, T) are given in adjacency list representation, and that (V, T) really is an MST with respect to w (so you don't need to check this).

Both algorithms must run in time $O(n+m)$, where $m := |E|$, assuming real number operations take $O(1)$ time each. This means that simply recomputing a minimum spanning tree with respect to w' from scratch takes too much time and will earn zero credit. You need not justify the correctness of your algorithm.

Answer: Let $e := \{u, v\}$ for some $u, v \in V$ with $u \neq v$.

(a) For part (a):

i. Remove e from T. (This makes (V, T) a forest with two connected components, i.e., a unique cut $(S, V - S)$, where S contains the vertices in the component of u and $V - S$ contains the vertices in the component of v.)

ii. Find the cut $(S, V - S)$ as above using Breadth-First Search from u to determine S.

iii. Find the w'-lightest edge e' crossing the cut $(S, V - S)$. (It is possible that $e' = e$.)

iv. Add e' to T. (T is now a minimum spanning tree with respect to w'. The proof of this is nontrivial.)

(b) For part (b):

i. Using Breadth-First Search, find the unique path in T connecting u with v. (This path together with e forms a cycle C.)

ii. Find the heaviest edge e' in C (heaviest with respect to w').

iii. If $e' = e$, then there is nothing to be done. (T is already a minimum spanning tree with respect to w'.)

iv. Otherwise, replace e' in T with e, that is, $T := (T - \{e'\}) \cup \{e\}$. ($T$ is now a minimum spanning tree with respect to w'. Again, the proof of this is nontrivial.)

Some remarks:

- The correctness of each algorithm can be shown, e.g., by comparing the behaviors of Kruskal's algorithm running with w versus w' and noting when discrepancies occur.
- The run times of each can be kept to $O(n + m)$ with a little care taken with the internal data structures.
- For part (a), one could instead look at the whole cycle formed by adding edge e' to T and then replace e (which is in this cycle) with the lightest edge along the cycle. This is acceptable as it does not increase the asymptotic run time, but it is unnecessary because the lightest edge on the cycle is provably e'.