Minor Exam (COL 351)

Name: **Entry Number:**

Read the instructions carefully:

- Do NOT use examples/figures to explain your algorithm. We will NOT read any such explanation. If giving a dynamic programming algorithm, clearly state the recurrence relation and the meaning of table entries.
- The proofs should be brief and statements in the proof should follow a logical sequence. If using induction, you must clearly state the induction hypothesis. Do NOT use examples or special cases to explain the proof. We will NOT read any such part of the proof.
- There are SIX pages in this question paper. You can write on both sides of each page. No extra page shall be provided.

1. You are given a set S of $n+1$ distinct numbers (which may not be integers). You can assume that S is given as an array (need not be sorted). You are also given an unsorted array A of size n containing exactly n out of the $n+1$ numbers in S. Give an $O(n)$ time divide and conquer algorithm to find the number from S which is not in A. The only operation allowed on numbers in S (or A) is comparison (you are NOT allowed to perform addition, subtraction, multiplication, etc. on these numbers). If needed, you can use new array variables (and copy elements of S or A into it).

   Solution: This is by divide and conquer. We first find the median m of A and partition A based on the median element m Let the two sides of A be A_L and A_R. Now partition S based on m . If m is not present in S, we are done Let the two sides be S_L and S_R. Since $|A| = |S| - 1$, either $|S_L| = |A_L|$ or $|S_R| = |A_R|$ If $|S_L| = |A_L|$, recurse on A_R, S_R (the other case is similar) Since partitioning takes $O(n)$ time, the recurrence $T(n) = T(n/2) + O(n)$ gives $O(n)$ time .

   If done by randomized algorithm instead of median finding, 2 points will be deducted. ■

2. A shuffle of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING. Given three strings $A[1..n], B[1..m], C[1..(n+m)]$, give a dynamic programming algorithm to determine whether C is a shuffle of A and B (the algorithm outputs a boolean value only).

   Solution: We create a table T where $T[i, j]$ stores the boolean value denoting whether $C[1, ..., (i+j)]$ is shuffle of $A[1, ..., i]$ and $B[1, ..., j]$ Base case: if $i = 0$, we need to check if $C[1...j] = B[1...j]$ and similarly for $j = 0$. Note that the base cases can be checked in $O(m + n)$ time by one scan . The recurrence is as follows:

$$T[i,j] = \begin{cases} T[i,j-1] & \text{if } C[i+j] \neq A[i], C[i+j] = B[j] \\ T[i-1,j] & \text{if } C[i+j] \neq B[j], C[i+j] = A[i] \\ T[i-1,j] \vee T[i,j-1] & \text{if } C[i+j] = A[i] = B[j]. \\ \text{false} & \text{otherwise} \end{cases}$$

   The reasoning is as follows: if $C[i + j] \neq A[i]$, then $C[i + j]$ must match with $B[i]$, in which case $C[1, ..., (i + j - 1)]$ must be a shuffle of $A[1...i]$ and $B[1...(j - 1)]$. The case when $C[i + j] \neq B[j]$ is similar. When $C[i + j] = A[i] = B[j]$, then we cannot tell if $C[i + j]$ will match with $A[i]$ or $B[j]$, so we have to look at both cases. n alternate way of writing the above recurrence is

$$T[i, j] = (T[i - 1, j] \wedge C[i + j] = A[i]) \vee (T[i, j - 1] \wedge C[i + j] = B[j]).$$

However the following is wrong (-3 marks):

$$T[i,j] = \begin{cases} T[i,j-1] & \text{if } C[i+j] = B[j] \\ T[i-1,j] & \text{if } C[i+j] = A[i] \\ \text{false} & \text{otherwise} \end{cases}$$

The for loop is as follows :

```
for i = 1 to n
        for j = 1 to m
                T[i,j] = as above
```

The answer is given by $T[n,m]$ and the running time is $O(nm)$.

■

3. You are given a bag of capacity S. The input also consists of n items, where for each item i, you are given its size s_i (assume that the sizes are given in an array of length n, and sizes can be rational numbers). Give an $O(n)$ time algorithm to find the maximum number of items that can be packed in the bag (a set of items can be packed in the bag if their total size does not exceed S). The algorithm just outputs a single number. **Note**: You cannot assume that the parameter S is constant (e.g., it can be as big as n or even more) and sorting the sizes will take $O(n \log n)$ time.

   Solution: Let m be the item with median size. It can be found in linear time. Let I_L be the items of size at most m and I_R be the items of size at least m. This partition can be done in $O(n)$ time If all the items in I_L do not fit in the bag, we recurse on the sub-problem given by I_L and bag capaicty S If all the items in I_L fit in the bag, we recurse on the sub-problem given by I_R and bag capacity $S - A$, where A is the total size of all the items in I_L Deciding which of these two cases apply takes $O(n)$ time . The base case is when $S = 0$ or negative, in which case the answer is 0 . Since the recurrence is $T(n) = T(n/2) + O(n)$, running time is $O(n)$

   If using randomization instead of median, -2 marks. ■

4. You are organizing a sports event, where each contestant has to swim 20 rounds in a pool, and then run 3 kilometers. However, the pool can be used by only one person at a time. In other words, the first contestant swims 20 rounds, gets out and then starts running. As soon as this first person is out of the pool, a second contestant begins swimming the 20 rounds; as soon as he/she is out and starts running, a third contestant begins swimming... and so on.)

   You are given a list of n contestants, and for each contestant you are given the time it will take him/her to complete swimming 20 rounds of the pool, and the time it will take him/her to run 3 kilometers. Your job is to decide on a schedule for the event, i.e., an order in which to sequence the starts of the contestants. The completion time of a schedule is the earliest time at which all contestants will be finished with swimming and running. Give an efficient algorithm that produces a schedule whose completion time is as small as possible.

   Example: Suppose there are two contestants C_1, C_2 with swimming and running times being $(10, 5)$ for C_1 and $(2, 8)$ for C_2. If we schedule them as C_1, C_2, then C_1 will finish swimming and running by time $10 + 5 = 15$. C_2 can start swimming at time 10, and so, will finish by time $10 + 2 + 8 = 20$. Note that both contestants will finish by time 20, and so the completion time is 20. If we order them as C_2, C_1, then C_2 will finish by time 10 and C_1 by time 17. Therefore, the completion time is 17. Thus, the best ordering is C_2, C_1.

   Solution: The algorithm is: sort the contestants in decreasing order of running times.

Let the sorted sequence be $C_1, \ldots, C_n$. Induction Hypothesis: let $C'_1, \ldots, C'_n$ be any ordering of the contestants. Then for all orderings with at most K inversions, the completion time is at least the completion time of the greedy algorithm. Clearly when $K = 0$, this is true.

Consider an optimal solution S which does not order the contestants in the order $C_1, \ldots, C_n$ and has K inversions. Then there must be two consecutive contestants C_i, C_j in the optimal solution S where $i > j$. We now produce a new solution S' where we keep the same ordering as the solution S but switch C_i, C_j to C_j, C_i.

Note that in both the solutions, the finish time of all contestants except for C_i, C_j remains unchanged. Let T be the total swimming time of all contestants appearing before C_i in the solution S– then C_i finishes at time $T + s_i + r_i$ and C_j at time $T + s_i + s_j + r_j$ (where s_k and r_k denote the swimming and the running times respectively). Now when we interchange the ordering in S', C_j finishes at $T + s_j + r_j$ and C_i at $T + s_i + s_j + r_i$. Now observe that

$$\max(T + s_j + r_j, T + s_i + s_j + r_i) \leq \max(T + s_i + s_j + r_j, T + s_i + r_i),$$

because $r_i \leq r_j$ Therefore, S' is better than S, i.e., it is also an optimal solution. Now S' has $K - 1$ inversions with the greedy ordering, and so its completion time is at least that of the greedy algorithm. Hence the greedy algorithm is optimal.

Common Mistakes: 1. Wrong greedy approach 2. The proof is not mathematical i.e. only contains intuitions such as "we can't parallelize swimming" etc. 3. The proof is incomplete: i.e. the student was unable to conclude that the new schedule isn't worse that the greedy one. 4. The exchange argument was proven however induction was not mentioned/used in the proof.

■

5. You are given a set of n jobs. Each job j is specified by two parameters: a size s_j and a deadline d_j. A schedule needs to decide on an ordering of the jobs. Given such a schedule, the penalty of a job j, is defined as $\max(0, C_j - d_j)$, where C_j is the time by which the job finishes. In other words, if the job finishes before its deadline, then the penalty is 0, otherwise it is the extent to which it exceeds the deadline.

   Example: Suppose you have two jobs: job 1 has size 2 and deadline 4 and job 2 has size 4 and deadline 5. If we process them in order job 1, job 2, then job 1 finishes at time 2 and incurs 0 penalty. Job 2 finishes at time 2+4 = 6 and hence incurs penalty of 1. Thus, the maximum penalty of any job is 1.

   Prove that the following algorithm minimizes the maximum penalty incurred by any job: order the jobs by ascending order of deadlines.

   Solution: The proof is again by induction on the number of inversions. More formally, let $1, 2, \ldots, n$ be the ordering of the jobs by decreasing deadline. Let M denote the maximum penalty of any job in this solution.

   Induction Hypothesis : for any ordering of the jobs with K inversions, the maximum penalty of any job is at least M. Base case is when $K = 0$: in this case, the ordering is same as $1, 2, \ldots, n$ and so this is same as the greedy solution.

   Now consider an optimal ordering S with K inversions: there must be two consecutive jobs i, j in this ordering such that $d_i > d_j$. We produce a new solution S' by interchanging i and j. We now check that the maximum penalty of any job in S' is at most the maximum penalty of any jon in S. Other than i, j all other jobs have the same penalty in the two solutions. Let T be the time at which i starts in the solution S. Since $d_j < d_i$ and j ends after i in S, the penalty of j in S is

more than that of i. Further the penalty of j in S is $\max(0, T + s_i + s_j - d_j)$. In S', penalty of i is $\max(0, T + s_i + s_j - d_i)$ and penalty of j is $\max(0, T + s_j - d_j)$. Both these quantities are at most $\max(0, T + s_i + s_j - d_j)$ Therefore S' is a better solution than S. By induction hypothesis, the maximum penalty of a job in S' is at least M, and so the same applies to S as well.

■

6. You are given an array A containing n numbers (which could be positive, zero or negative **rational** numbers). A sub-array $A[i, j]$ of A, where $i \leq j$, is defined by the sequence $A[i], A[i + 1], \ldots, A[j]$. For each such sub-array, define $P(i, j)$ as the product of the entries in $A[i, j]$. Give an $O(n)$ time algorithm to find the largest value of $P(i, j)$ overall sub-arrays $A[i, j]$ (note that the algorithm just outputs a number). You can assume that arithmetic operations like multiplication on numbers in A take constant time.

   Example: Suppose A is $\{-3, 10, -6, 0.7, 2, -1\}$. Assuming that the first element of A is denoted by $A[1]$, the sub-array $A[1, 4]$ has total product $-3 \times 10 \times -6 \times 0.7 = 126$, whereas sub-array $A[3, 6]$ has total product $-6 \times 0.7 \times 2 \times -1 = 8.4$.

   Solution: We use dynamic programming. We have two tables: $M[i]$ and $N[i]$. The table $M[i]$ denotes the maximum subarray product ending at $A[i]$ and $N[i]$ denotes the minimum subarray product ending at $A[i]$.

   Now the recurrence for $M[i]$ would be: if $A[i] > 0$, then $M[i] = \max(A[i], A[i] \cdot M[i-1])$. If $A[i] < 0$, then $M[i] = \max(A[i], A[i] \cdot N[i-1])$. The reason is that if $A[i] > 0$, then the maximum subarray ending at $A[i]$ is either $A[i]$ or $A[j]...A[i-1] \cdot A[i]$ for some $j < i$. But then $A[j]...A[i-1]$ must be as large as possible, and so equal to $M[i-1]$. Similar argument applies for $N[i]$. We can use the for loop from $i = 0$ to N and output $M[N]$.

   The above can also be replace by the following for loops: note that if $A[i] > 0$ then $A[i] > A[i]M[i-1]$ iff $M[i-1] \leq 1$. Thus, we can keep two variables storing $M[i]$ and $N[i]$ – call these M, N.

   $M = N = 1$
   If $A[i] \geq 0$: If $M < 1$, $M = A[i]$ else $M = M \cdot A[i]$.
   If $N > 1$, $N = A[i]$ else $N = N \cdot A[i]$.
   If $A[i] < 0$: If $M > 1$, $M = A[i]$ else $M = M \cdot A[i]$.
   If $N < 1$, $N = A[i]$ else $N = N \cdot A[i]$.
   Output the largest M seen.

   The above is also ok, but then one needs to explain the loop invariants: i.e., state that after i iterations M is the largest sub-array product ending at $A[i]$ and similarly for N. And then explain why the manner of updating M and N as above maintains this invariant.

■