Monsoon 2021 Graduate Algorithms: Mid-Sem Exam *Max: 75 points*
Due: **Tuesday, 28 September** *Duration: 2 hours*

Write only the answers in the answer sheet. Most questions demand a short justification, and for them it is mandatory to include an explanation using 3-4 sentences (anything more is liable to be ignored). Justifications like "we can use XYZ for doing ABC" is not always acceptable; it is similar to obviously impossible facts like "humans can use their fingers like feathers and arms like wings for flying like a bird". In other words, a statement does not become a possibility just because you say so. Do not forget to include bases for recurrences and recursive algorithms. You can write "I don't know" for any top-level question and get 20% of the marks of that question, subject to a maximum of 8 marks.

Q1 [5 points] Consider the "Tower of Hanoi" problem but with 4 pegs and you have to move n disks from peg-1 to peg-4 with the standard restrictions that no disk can be placed on top of a smaller disk and only one disk can be moved in any step. What is the minimum number of moves to perform this task? Select one from below (assuming $n > 2$) and include a short justification.

(a) Less than $2^n - 2$ moves

(b) Exactly $2^n - 2$ moves

(c) Exactly $2^n - 1$

(d) The number of moves cannot be more than $2^n - 1$ (select this only if you cannot justify any of the above)

(e) Exactly 2^n

(f) More than 2^n

Explanation: Number of steps to move 3 disks with 3 pegs is 7 but with 4 pegs is 5. So fewer moves will be required when 3 disks have to be moved in the recursive calls with n=3. This will lead to fewer moves even for recursive calls with larger n.

Only use 1st, 2nd and 4th pegs

Q2 [5 points] Prove that $T(i,j) = O((i+j)^2)$ is a solution of the following recurrence relation. Show the steps.

$T(i,j) =$

- $\max\{T(i-1,j), T(i,j-1)\} + (i+j)$ for $i > 0, j > 0$,
- $T(i,0) = T(i-1,0) + i$ for $i > 0$,
- $T(0,j) = T(0,j-1) + j$ for $j > 0$,
- $T(0,0) = d$

Satisfied by guessed solution

Let $T(i,j) \le c(i+j)^2 + d$.

$\max\{T(i-1,j), T(i,j-1)\} + (i+j)$

$\le \max\{c(i+j-1)^2 + d, c(i+j-1)^2 + d\} + (i+j)$

$\Rightarrow c(i+j-1)^2 + d + (i+j) = c(i+j)^2 + d - 2c(i+j) + c$

$\le c(i+j)^2 + d$ since $2c(i+j) \ge 1$ for $i > 0, j > 0$

RHS $\le c(i-1)^2 + d + i$

$= ci^2 + d - 2ci + c$

$\le ci^2 + d$ $\because i \ge 1$

Q3 [5 points] Suppose we are solving an n-item Knapsack instance (maximizing the total value possible with itemwise values $v_1 \dots v_n$ and weight limit W) and we choose to first compute the values $B[j,k]$ = smallest weight possible with total value $\ge k$ using items $\{1...j\}$ for all $j = 1 \dots n$ and all $k = 1 \dots \sum_i v_i$. Explain how we can solve the original Knapsack problem (let OPT denote the solution) by stating a mathematical expression (or, a 1-line English statement) for OPT using the $B[\dots]$ values: $OPT =$ _____. Explain your answer.

$= \{v : B(n,v) \le W\}$

Q4 [5 points] Recall the H matrices we discussed for the all-pairs-shortest-path problem, and the algorithm for computing the distance matrix D using the H matrices. Suppose you have obtained $H_1, H_2, H_4, \dots, H_{2^b}, H_{2^{b+1}}$ where b is the next power of 2 greater than equal to $n-1$ (here n denotes the number of vertices). Explain briefly how you can determine if the graph has a negative weight cycle using these matrices. Note that here you have access to $H_{2^{b+1}}$ even though the APSP algorithm discussed in class computed up to H_{2^b}, however, it is not mandatory to use all the H matrices to answer this question.

Two approaches are possible: ① H_{2^b} will have -ve entries on diagonal ② $H_{2^b} \ne H_{2^{b+1}}$

Q5 [5 points] Consider the weighted edit-distance problem over strings $X[1 \dots n]$ and $Y[1 \dots m]$ where the cost of inserting one character is a, that of deletion is b and that of replacement/change is c. The weight of an edit sequence is the sum total of all the insertions, deletions, and replacements. The objective here is to identify a sequence with the smallest weight.

Derive a recursive formula (along with base cases) to compute $Edit(i,j)$ = weight of the optimal edit sequence from $X[1 \dots i]$ to $Y[1 \dots j]$.

$$Edit(i,j) = \begin{cases} 0 & \text{if } i=0, j=0 \\ \min \begin{cases} Edit(i-1,j) + b \\ E(i,j-1) + a \\ Edit(i-1,j-1) + c & \text{if } x[i] \ne y[j] \\ Edit(i-1,j-1) & \text{if } x[i] = y[j] \end{cases} \end{cases}$$

if $x[i] = y[j]$, 1st two options can be ignored.

Q6 [5 points] Read the previous question on weighted edit-distance. Define $W^i[j,k]$ as "length of the prefix of $Y[1\ldots k]$ that $X[1\ldots i]$ maps to in an optimal edit sequence from $X[1\ldots j]$ to $Y[1\ldots k]$" for suitable values of i,j,k. $W^i[j,k]$ is set to ∞ for the other values of i,j,k. Write a recursive formula to compute $W^i[j,k]$ for <u>all</u> values of $i=1\ldots n$, $j=1\ldots n$, $k=1\ldots m$. Explicitly state for which values $W^i[j,k]$ should be set to ∞. You may use the $Edit(i,j)$ values from the previous question. Exact same as in the unweighted case (refer to lecture slides/JE).

Q7 [10 points] This question is about the median-of-median algorithm for finding median. (a) Suppose we modified the algorithm to split the input array into $\lceil \frac{n}{3} \rceil$ blocks of 3 elements each. There is no need to show steps in this question but be mindful of floors and ceilings. $T(n) = T(\lceil n/3 \rceil) + T(\frac{2n}{3}) + O(n) = O(n \lg n)$

1. State the recurrence for the time complexity. Explain in 1 sentence. State its solution.
2. State the recurrence for the space complexity. Explain in 1 sentence. State its solution. $S(n) = \max\{S(\lceil \frac{n}{3} \rceil), S(\frac{2n}{3})\} + c = S(\frac{2n}{3}) + c = O(\lg n)$
3. Now, consider the case when we split the input array into $\lceil \frac{n}{11} \rceil$ blocks of 11 elements each. State the recurrence for the time complexity. Explain in 1 sentence. State its solution.
   T(n) = T(ceil(n/11)) + T(8n/11) + O(n) = O(n) since for each block at least 6 elements are <= mom or >= mom.

Q8 [5 points] Consider the following algorithm to determine if a string X is a subsequence of another string Y.

```
def IsSubsequence(X,Y):
    return IsSubseqRecursive(len(X), len(Y))

// determine if X[1...i] is a subsequence of Y[1...j]
def IsSubseqRecursive(i,j):
    if j == 0: return True   if i>0 return True else return False (correction was announced during exam)

    if X[i] not= Y[j]:
        return ISS(i,j-1)_____________________ (i)
    else:
        return ISS(i-1,j-1) OR ISS(i,j-1)_____ (ii)
```

(a) Write the return statements (i) and (ii) in your answer sheet. Both of these should be one liners.
(b) Write a recurrence (along with base cases) for the time-complexity of `IsSubseqRecursive(i,j)`.

1 T(i,j) = O(1) if j=0 T(i,j) = O(1) + max{T(i-1,j-1) + T(i,j-1), T(i,j-1)} = O(1) + T(i-1,j-1) + T(i,j-1) 2

Q9 [10 points] We are given an array $A[1\ldots n]$ of distinct integers which we want to sort in increasing order by calling `ReverseSort(A,1)`. Design a <u>recursive</u> subroutine `ReverseSort(A,s)` to sort the subarray $A[s\ldots n]$ <u>in-place</u> (without using any additional space). `ReverseSort` can modify the array by <u>only</u> calling `Reverse(...)` (maybe, multiple times). A call to `Reverse(i)` reverses the subarray $A[i\ldots n]$ in $O(1)$ time and $O(1)$ space, and $O(n^2)$ should be the complexity of the sorting algorithm.

For example, if $A=[5,4,3,2,1]$, then $Reverse(2)$ changes A to $[5,1,2,3,4]$. The running time of $ReverseSort(A,1)$ should be $O(n^2)$ and the number of calls to $Reverse()$ should be at most $2n$.

You can employ the following operations by simply calling them (avoid writing detailed pseudocode for them): $\min(arr)$ returns the minimum of an array, $\max(arr)$ does the same but for maximum, $rank(elem, arr)$ returns the rank of an element $elem$ in an array, $size(arr)$ returns the number of elements in an array (if you do not have it already), $select(k, arr)$ returns the k-th smallest element in an array. But do not forget to account for their complexities.

(a) Write the pseudocode of `ReverseSort(A,s)`, and (b) write a recurrence for its time complexity, and (c) write a recurrence for the number of calls to `Reverse()`, (d) solve the recurrence in (c) to obtain a tight (upper) bound on the number of calls to `Reverse()`, and (e) state the space complexity of your algorithm.

Q10 [20 points] The cost of a stock of some company on each day is given in an array $C=[c_1,c_2,\ldots,c_n]$. On any day you have three options: think about DPs (ignoring the stock market), buy 1 stock only if you are not holding any stock, or sell 1 stock if you are already holding 1 stock (you cannot sell if you do not have any!). Therefore, you can at most hold 1 stock at the end of any day, you can buy-sell multiple number of times, and even choose to not hold any stock on some days. When you sell, the profit you earn is "price on the day you sell a stock - price on the day you bought that stock" (which can be negative). Further, everytime you sell you have to pay a capital gains tax which is a fixed amount denoted X.

Find the max profit that you can make after n days. For example, if the prices are $\{100,300,200,800,400,900\}$ and $X=200$, the maximum profit can earned by buying on day 1, selling on day 4 (earning profit $(800-100)-X=500$), buying on day 5, selling on day 6 (earning profit $(900-400)-X=300$), leading to a total profit of 800.

Q9. def ReverseSort(A,s):
t = position/index of min A[s...n]
Reverse(t)
Reverse(s)
ReverseSort(A,s+1)
} minimum of A[s...n] put at front. other variations are possible.

Also add base case : if s=n: return // do nothing for a 1-sized array

Let T(k) denote the time complexity of ReverseSort(A,n-k+1) i.e., calling ReverseSort on A[n-k+1, ... n].
T(k) <= O(k) + O(k) + T(k-1)
T(1) = O(1)

Let N(k) denote the number of calls to Reverse() made by ReverseSort(A,n-k+1).
N(1) = 0 // if base case is for s=n; if base case is for s > n, N(0)=0 is the base case and then N(k)=2k
N(k) = 2 + N(k-1) = 2(k-1)

Let S(k) denote the space complexity of ReverseSort(A,n-k+1).
S(1) = O(1)
S(k) = S(k-1) + O(1) + O(k) // if space complexity of t = rank(min(A[n-k+1...n])) takes O(k) space.

Q10. Let Profit(k,mode) be defined as : if mode="hold" then it is the largest profit possible after k days such that a stock is held at the end of the k-th day, if mode="sold" then it is the largest profilt possible after k days such that no stock is held at the end of the k-th day.

base case:
Profit(1,hold) = -c1
Profit(1,sold) = 0

recursive formulae:
Profit(k,hold) = max{ Profit(k-1,hold) // no op on the k-th day
Profit(k-1,sold) - ck // buy a stock on the k-th day, so no stock must have been held at the end of the k-1-th day
Profit(k,sold) = max{ Profit(k-1,sold) // no op on the k-th day
Profit(k-1,hold) + ck // sell on the k-th day, so stock must have been held at the end of the k-1-th day

Need a 2D n x 2 array.
Array can be filled in the increasing order of k, and in any order among hold/sold.

The original problem can be solved as: max{ Profit(n,hold), Profit(n,sold) } = Profit(n,sold) since it does not make sense to hold a stock at the end of the n-th day

Time complexity: O(n)
Space complexity: O(n)

To obtain the complete buy-sell schedule, assume that we have first computed the entire memo, denoted M; now call print_schedule(n,sold) which prints the schedule for the first n days.

```
def print_schedule(k,mode)://prints the schedule for the first k days that generates the maximum profit at the end of the
                            // k-th day and stock holding the is in mode "mode"
    if k=1:
        if mode="hold": print "buy on 1st day"
        else          : print "no op on 1st day"

    if mode = "hold":
        if Profit(k,hold) = Profit(k-1,hold): print "no-op on kth day", return print_schedule(k-1,hold)
        else                                : print "buy on k-th day", return print_schedule(k-1,sold)
    else:
        if Profit(k,sold) = Profit(k-1,sold): print "no op on the k-th day", return print_schedule(k-1,sold)
        else                                : print "sell on the k-th day", return print_schedule(k-1,hold)
```