Monsoon 2023 — Graduate Algorithms: Mid-sem — *Max: 80 points*

Date: **24 September** — *Duration: 120 mins*

Name: **Roll No.:**

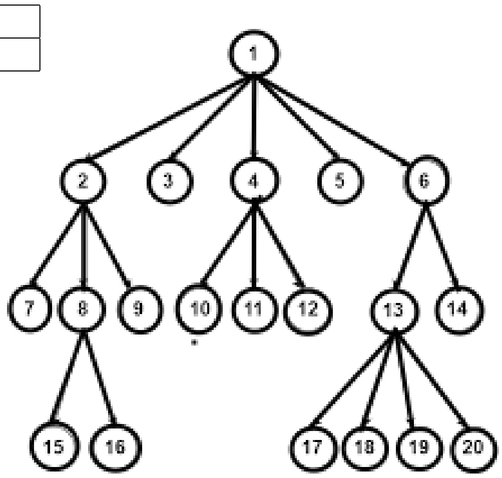
10% if you write **"I don't know"** for any (sub)question; with a cap of total 8 marks.
You will get ~~F straight-away~~ at most **D** (revised policy) if you score less than 20.
Answer Q1 to Q4 on the question-paper itself.

Q1 [5+1+4=10 points] An independent set in a graph is a subset of the vertices with no edges between them. The nodes (1,9,5,10,7,13) form an independent set of size 6 in the tree below; however, (1,3,5,10,7,13,6) is not an independent set due to the edge between 6 and 13.

Consider the following function : $MaxIS(v)$ is defined as the size of the largest independent set in the subtree rooted at v.

(a) Fill these values:

v	15	8	7	2	4
$MaxIS(v) =$	1	2	1	4	3



(b) Now consider an arbitrary tree.

1. Suppose v is a leaf-node. Write the value: $MaxIS(v) =$ 1
2. Suppose v is not a leaf-node. Write a recursive formula (feel free to use $child(node)$ to get the list of children of a node): $MaxIS(v) =$ max of
   - $\text{sum}_{c \in child(v)} MaxIS(c)$
   - $1 + \text{sum}_{c \in child(d), d \in child(v)} MaxIS(c)$

Q2 [4+6=10 points] Assume that n is a power of 2 for this question.

(a) Let $\langle a_0 \dots a_{n-1}\rangle$ denote the coefficients of a degree-$(n-1)$ polynomial $A(x)$. Fill in the blanks to give us an algorithm that runs in $O(n \log n)$ time and returns the discrete Fourier transform of $A(x)$.

def $DFT(A : \langle a_0 \dots a_{n-1}\rangle)$:
    if n=1: return $\langle a_0\rangle$
    else:
        $\langle y_0^{even}, \dots y_{n/2-1}^{even}\rangle \leftarrow DFT(\langle a_0, a_2, \dots a_{n-2}\rangle)$
        $\langle y_0^{odd}, \dots y_{n/2-1}^{odd}\rangle \leftarrow DFT(\langle a_1, a_3, \dots a_{n-1}\rangle)$
        for k=0 to n−1:
            $y_k =$ $w_n^k * y^{odd}_{k \bmod n/2} + y^{even}_{k \bmod n/2}$
        return $\langle y_0, \dots y_{n-1}\rangle$

(b) Next, let $Y = DFT_n(A)$ denote the discrete Fourier transform of a polynomial $A(x)$. Fill the blanks below to design an algorithm that runs in $O(n \log n)$ time and returns the coefficients of $A(x)$.

def $IDFT(Y : \langle y_0 \dots y_{n-1}\rangle)$:
    if n=1: return $\langle y_0\rangle$
    else:
        $\langle a_0^{even}, \dots a_{n/2-1}^{even}\rangle \leftarrow IDFT(\langle y_0, y_2, \dots y_{n-2}\rangle)$
        $\langle a_0^{odd}, \dots a_{n/2-1}^{odd}\rangle \leftarrow IDFT(\langle y_1, y_3, \dots y_{n-1}\rangle)$
        for k=0 to n−1:
            $a_k =$ $\left[w_n^{-k} * a^{odd}_{k \bmod n/2} + a^{even}_{k \bmod n/2}\right] / n$
        return $\langle a_0, \dots a_{n-1}\rangle$

Q3 [6+4=10 points] Suppose we are interested in finding out the optimal edit sequence from $X = GRAD1234$ to $Y = GRAD56$.

(a) Fill the table below in which the entry in the i-th column and j-row represents some number h such that there is an optimal edit sequence from $X[1 \dots i]$ to $Y[1 \dots j]$ which can be divided into two optimal edit sequences: one from $GRAD$ to $Y[1 \dots h]$ and another from $\cancel{ALGO}$ 1234 to $Y[h+1 \dots 6]$ (this was the definition for the $Half(i,j)$ function used in class).

	1	2	3	4	5	6	7	8
1	∞	∞	∞	1	1	1	1	1
2				2	2	2	2	2
3				3	3	3	3	3
4				4	4	4	4	4
5				5	4	4	4	4
6	∞	∞	∞	6	4	4	4	4

(b) Give recursive expressions to compute $T(m,n)$ and $S(m,n)$ that denote the running time and the space complexity of the edit-sequence dynamic programming algorithm using Hirshberg's space optimization. Then, write the solutions of those recurrences (no need to show derivation). Here, m denotes the size of the source string, and n denotes the size of target string, and if needed, you can assume that $m < n$.

$T(m,n) = O(mn) + T(\frac{m}{2}, h) + T(\frac{m}{2}, n-h) = O(mn)$

$S(m,n) = O(m) + \max(S(\frac{m}{2}, h), S(\frac{m}{2}, n-h)) = O(m+n)$

Q4 [5 points] Consider the following algorithm.

```
def Sort(A[0...n-1]):
    if n=2 and A[0] > A[1]
        swap(A[0] ↔ A[1])
    else if n > 2:
        m = ⌊2n/3⌋
        Sort(A[0...m-1])
        Sort(A[n-m...n-1])
        Sort(A[0...m-1])
```

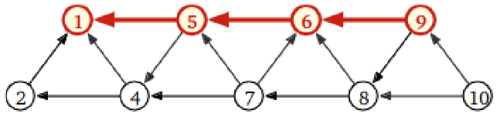
(a) State if this algorithm correctly sorts any array with distinct elements in increasing order. No

(b) If the answer to the above is "yes", then write the recurrence for its time-complexity, and its solution. If the answer is "no", present a counter-example array A of at most 6 elements, and trace the execution of this algorithm on A (basically, show all recursive calls and the state of the array when they return).

Write the answers to these questions in the answer booklet.

Q5 [15 points] Let G be a directed graph, where every vertex v has an associated height $h(v)$, and for every edge $u \to v$ we have the inequality $h(u) > h(v)$. Assume all heights are distinct. The **span** of a path from u to v is the height difference $h(u) - h(v)$.

Describe and analyze an algorithm using dynamic programming to find the **maximum span** of a path in G with at most k edges. Your input consists of the graph G, the vertex heights $h(\cdot)$, and the integer k. Report the running time as a function of n (number of nodes), m (number of edges), and k. Do **not** write a pseudocode, instead answer using the format explained in class. Note that only the maximum span, an integer, has to be returned.



For example, given the following labeled graph and the integer k = 3 as input, your algorithm should return the integer 8, which is the span of the downward path $9 \to 6 \to 5 \to 1$.

Q6 [15 points] An element x in an array A (in which elements may be occurring multiple times) is said to be a *semi-majority* if its frequency in A is **greater than** $|A|/3$. Observe that not every array would have a semi-majority element, and that there may more than one such element. Design a divide-and-conquer algorithm to return any semi-majority of an input array A, or return `null` if there is no such element. Write its pseudocode, add enough comments or a separate explanation to make it clear, and then derive its time-complexity. An $O(n \log n)$ algorithm is required for full-credit; however, a slow but correct approach will fetch more marks than a fast but incorrect approach.

Q7 [15 points] A **zigzag walk** in a directed graph G is a sequence of vertices (not necessarily distinct) connected by edges in G, but the edges alternately point forward and backward along the sequence. Specifically, the first edge points forward, the second edge points backward, and so on. An example of a zigzag walk in the graph above is $9, 6, 7, 5, 7, 4, 5, 1, 2$. The length of a zigzag walk is the number edges, both forward and backward.

Suppose you are given a directed unweighted graph G, along with two vertices s and t. Design a graph-reduction based approach to find the shortest zigzag walk from s to t in G. Do **not** write a pseudocode, instead, (a) first describe the vertices and edge of another graph H, (b) then, explain what problem will you solve on H and what algorithm will you choose (a problem is difference from an algorithm), and (c) finally, analyse the complexity of your approach in terms of m and n, the number of edges and vertices of G, respectively (the complexity should include the time to construct H and the complexity of the algorithm on H).

---

5. MaxSpan(v,t) = maximum span of any path starting at v and using at most t edges

MaxSpan(v,0) = 0 no more edges are allowed

MaxSpan(v,t) = max_{u : v -> u} [h(v) - h(u)] + MaxSpan(u,t-1), any path must begin with $v \to u$ for some u

0 if no such u

Memo: 2D table. (v,t) depends on (u,t-1). So, fill in the increasing order of t. Order of v does not matter.

(if stored in tree, need to traverse the tree multiple times which is ok).

Time: O(n^2 k). K is also an input. (even if the DP is correct, 3 is penalty for slower DP)

Space: O(n)

Final problem: Find max_v MaxSpan(v,k).

(Topological sorting is not required.)

Initial

7. $H = \langle W, F \rangle$ $W = \{ V \times \{\text{forward}, \text{backward}\} \}$

(v, forward) means last edge selected is in forward direction / another logic: forward denotes direction of next edge to be added
(v, backward) mean ... backward direction

Add $(u, \text{forward}) \xrightarrow[H]{} (v, \text{backward})$ if $u \xleftarrow[G]{} v$

Add $(u, \text{back}) \xrightarrow[H]{} (v, \text{fwd})$ if $u \xrightarrow[G]{} v$

Find shortest path from (s, backward) to (t, fwd)
(s, backward) to (t, back) & return the shorter of both

Use BFS for shortest path

$|W| = 2n$ $|F| = 2m$

Constructing H: $O(n+m)$
BFS : $O(n+m)$ } Total: $O(n+m)$

6.

```
// returns all the semimajority elements of A;
// there can be at most two, so, for the sake
// of consistency, this function returns an
// array which is filled with NULL
def semimajority(A):
    if |A| = 1: return [NULL, A[0]]

    // recursive calls, some of the return values could be NULL
    (l1, l2) = semi-majority(left half of A)
    (r1, r2) = semi-majority(right half of A)
    // Any semimajority of A _must_ be a semi-majority
    // of either left half of A, or of right half of A
    // Check the frequency of all the returned elements
    // in A to check which can be semimajority of A
    answer = [ ]
    For x in [l1, l2, r1, r2]:

      if x is not NULL:
          count = 0
          // compute frequency of x
          for a in A: if a = x, count ++
          if count > |A|/3:
              answer.append(x)

   // fill answer with NULL
   if |answer| = 0: answer.append(NULL); answer.append(NULL);
   if |answer| = 1: answer.append(NULL);
   return answer
```

Complexity: T(n) denotes complexity of the function on A with |A|=n
$T(n) = 2T(n/2) + O(n) = O(n \log n)$

There should be no global variable other than the array.
For full credit, the entire algorithm should be a divide and conquer type.
Use of a divide-and-conquer algorithm (like computing the frequencies of all elements) and some pre/post-processing (like filtering based on the frequencies) is discouraged, and will get partial marks.

Two approaches are possible. Right one gets partial credit since the entire algorithm is not a divide-and-conq type.

```
// returns a frequency table of the elements of A
def freqtable(A):
    if |A| = 1:  return {A[0] : 1}

    lefttable = freqtable(left half of A)
    righttable = freqtable(right half of A)

    // combine these two table to compute
    // frequency table of A
    answertable = { } // empty dictionary
    for x in A:
        answertable[x] = lefttable[x] + righttable[x]

    return answertable

def semimajority(A):
    table = freqtable(A)
    for x in table:
        if table[x] > |A|/3 : return x
    return NULL
```

Complexity:
T(n) denotes complexity of freqtable on A with |A|=n
$T(n) = 2T(n/2) + O(n) = O(n) = O(n \log n)$