HKUST – Department of Computer Science and Engineering

COMP3711: Design and Analysis of Algorithms – Spring 2019

#

Midterm Examination: Solution Key

Date: Monday, March 18, 2019 Time: 19:20–21:20 Venue: LTC/LTD

Name: _____	Student ID: _____
Email: _____	Lecture L1 / L2

Instructions

- This is a closed book exam. It consists of 23 pages and 6 questions.
- Please write your name, student ID and ITSC email and circle your Lecture section (L1 is TTh and L2 is WF) at the top of this page.
- For each subsequent page on which you write a solution, please also write your student ID at the top of the page in the space provided.
- Please sign the honor code statement on page 2.
- Answer all the questions within the space provided on the examination paper. The last 2 pages are for rough work but can be used for real solutions if you CLEARLY MARK THEM as such.
- Most questions provide at least one extra page for writing answers. This is for clarity and is not meant to imply that solutions require all of the blank pages. Many can be answered using much less space.
- All solutions must be written on the front (numbered) side of the pages. Page backs may only be used for rough work. Nothing written on the backs of pages will be marked.

Questions	1	2	3	4	5	6	Total
Points	15	10	18	20	15	22	100
Score							

Student ID: _____

As part of HKUST's introduction of an honor code, the HKUST Senate has recommended that all students be asked to sign a brief declaration printed on examination answer books that their answers are their own work, and that they are aware of the regulations relating to academic integrity. Following this, please read and sign the declaration below.

I declare that the answers submitted for this examination are my own work.

I understand that sanctions will be imposed, if I am found to have violated the University regulations governing academic integrity.

Student's Name: _____

Student's Signature: _____

Student ID: _____

1. **Time Complexity** [15 pts]
a) We have two algorithms, A and B. Let $T_A(n)$ and $T_B(n)$ denote the time complexities of algorithm A and B respectively, with respect to the input size n. Listed below are 9 different cases of time complexities or formulas for each algorithm. Complete the last column of the following table with "A", "B", or "U", where:

   - "A" means that for large enough n, algorithm A is always faster;
   - "B" means that for large enough n, algorithm B is always faster;
   - "U" means that the information provided is not enough to justify stating that, for large enough n, one algorithm is always faster than the other.

   Recall that an algorithm is "faster" if its running time is smaller. To get you started we have provided two example questions and their answers.

   Notational comment: $\log^a b$ is shorthand for $(\log b)^a$. For example $\log^{10} n = (\log n)^{10}$.

Case	$T_A(n) =$	$T_B(n) =$	Faster
1	$\Theta(1)$	$\Theta(n)$	A
2	$\Omega(1)$	$O(n)$	U
3	$\Omega(n/\log n)$	$\Omega(10^{10})$	
4	$\Theta(n^3)$	$O(n^{\log_2 7})$	
5	$4T_A\left(\frac{n}{4}\right) + n$	$\Theta(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n})$	
6	$\Theta(n^{1.5}\log^3 n)$	$\Theta(\log_3(n!))$	
7	$O(\log^{10}(\log n))$	$\Omega(n \log n)$	
8	$O(16^{\log_4 n} + n^{1.5})$	$5T_B\left(\frac{n}{2}\right) + n^2$	
9	$\Omega(\sqrt{5n})$	$\Theta\left(\left(\sqrt{5}\right)^n\right)$	

Continued on next page ....

Solution:

Case	$T_A(n) =$	$T_B(n) =$	*Faster*
1	$\Theta(1)$	$\Theta(n)$	A
2	$\Omega(1)$	$O(n)$	U
3	$\Omega(n/\log n)$	$\Omega(10^{10})$	U
4	$\Theta(n^3)$	$O(n^{\log_2 7})$	B
5	$4T_A\left(\frac{n}{4}\right) + n$	$\Theta(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n})$	B
6	$\Theta(n^{1.5}\log^3 n)$	$\Theta(\log_3(n!))$	B
7	$O(\log^{10}(\log n))$	$\Omega(n \log n)$	A
8	$O(16^{\log_4 n} + n^{1.5})$	$5T_B\left(\frac{n}{2}\right) + n^2$	A
9	$\Omega(\sqrt{5n})$	$\Theta\left(\left(\sqrt{5}\right)^n\right)$	U

Case 5: $T_A(n) = 4T_A\left(\frac{n}{4}\right) + n = \Theta(n \log n)$;
$T_B(n) = \Theta(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^n}) = \Theta(1)$

Case 6: $T_A(n) = \Theta(n^{1.5}\log^3 n) = \Theta(n^{1.5}\log n)$;
$T_B(n) = \Theta(\log_3(n!)) = \Theta(n \log n)$

Case 8: $T_A(n) = O(16^{\log_4 n} + n^{1.5}) = O(n^2)$;
$T_B(n) = 5T_B\left(\frac{n}{2}\right) + n^2 = \Theta(n^{\log_2 5})$

(b) Derive from first principles (you can't use the Master Theorem) the tight asymptotic solution to

$$T(1) = 2; \qquad \forall n > 1, \quad T(n) = 10T(n/3) + n^2.$$

Show your steps. Your final solution should be in the form $T(n) = \Theta(n^c)$ or $T(n) = \Theta(n^c \log n)$ for some constant $c > 0$. You may assume that n is a power of 3.

Solution: Using the expansion method to solve the recurrence.

$T(n)$

$= 10T(n/3) + n^2$, for $n > 1$

$= 10[10T(n/3^2) + (\frac{n}{3})^2] + n^2$

$= 10^2 T(n/3^2) + (\frac{10}{3^2} + 1) \cdot n^2$

$= 10^2[10T(n/3^3) + (\frac{n}{3^2})^2] + (\frac{10}{9} + 1) \cdot n^2$

$= 10^3 T(n/3^3) + (\frac{10^2}{9^2} + \frac{10}{9} + 1) \cdot n^2$

...

$= 10^i T(n/3^i) + [(10/9)^{i-1} + (10/9)^{i-2} + ... + 1] \cdot n^2$

$= 10^h T(n/3^h) + [\frac{(10/9)^h - 1}{10/9 - 1}] \cdot n^2$

$= 10^{\log_3 n} T(1) + (\frac{10^{\log_3 n}}{9^{\log_3 n}} - 1) \cdot 9n^2$, by setting $h = \log_3 n$

$* = 2n^{log_3 10} + 9n^{log_3 10} - 9n^2$, given T(1) = 2

$= \Theta(n^{log_3 10})$

where (*) comes from the fact that

$$10^{\log_3 n} = 10^{\frac{\log_{10} n}{\log_{10} 3}} = 10^{\log_{10} n \cdot \log_3 10} = (10^{\log_{10} n})^{\log_3 10} = n^{\log_3 10}$$

$$9^{\log_3 n} = 9^{\frac{\log_9 n}{\log_9 3}} = 9^{\log_9 n \cdot \log_3 9} = (9^{\log_9 n})^{\log_3 9} = n^2$$

2. **Huffman Coding [10 pts]**
You are given that the following letters appear in a long message with the associated frequencies:

Letter	a	b	c	d	e	f	g	h
Frequency	2	2	2	3	4	5	7	10

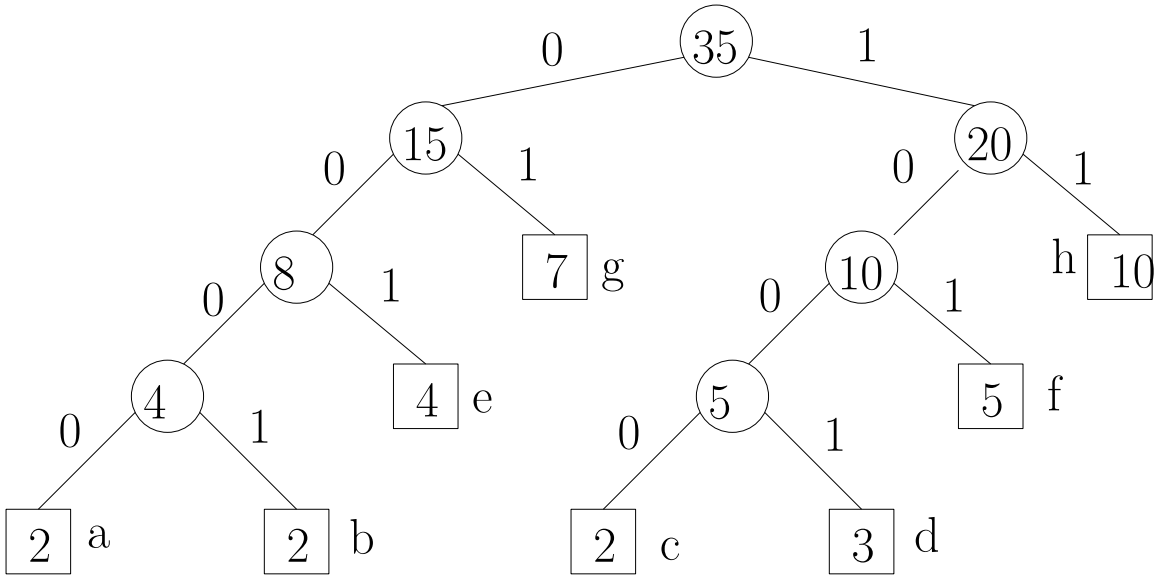
Build a Huffman Tree for these 8 letters with their associated frequencies.

Draw the Huffman tree to the left of the table below (labelling each leaf with its associated letter) and then fill in the table by writing down the codeword from the tree associated with each letter. You can show your work on the next page. (It is not necessary to show your work, but if you don't and make an error we will not be able to give you partial credit).

letter	codeword
a	
b	
c	
d	
e	
f	
g	
h	

Student ID: _____

letter	codeword
a	0000
b	0001
c	1000
d	1001
e	001
f	101
g	01
h	11



3. **Interval Scheduling [18pts]**
In this problem you must describe and explain the correctness of the interval scheduling algorithm that was taught in class.

The Interval Scheduling Problem:
Assume you are given a set of n SORTED intervals $I_i = [s_i, f_i]$, $i = 1, \ldots, n$. s_i, f_i are, respectively, the starting and finishing times of interval I_i.

Two intervals I_i and I_j are *compatible* if they do not overlap i.e., if $s_i \geq f_j$ or $s_j \geq f_i$. The *interval scheduling problem* is to select a maximum size set of mutually compatible intervals. That is, to find a largest size set in which no two intervals overlap.

The Greedy Algorithm: ran the following code

1. $A = \{I_1\}$;
2. For $j = 2$ to n
3. if I_j is compatible with the intervals in A
4. $A = A \cup \{I_j\}$; % Add I_j to the set
5. Output set A

The problem definition did not state HOW the intervals were sorted. Consider the following two ways of sorting.

(a) Assume that the intervals are sorted by increasing starting time, i.e, $s_1 \leq s_2 \leq \cdots \leq s_n$.

(b) Assume that the intervals are sorted by increasing finishing time, i.e, $f_1 \leq f_2 \leq \cdots \leq f_n$.

For both (a) and (b) separately answer the following question.

Will the greedy algorithm return the correct result, i.e., a maximum size set of mutually compatible intervals?

If the answer is yes, provide a formal proof of correctness.

If the answer is no, provide a counterexample. A counterexample is a set of intervals (sorted by starting time in (a) and finishing time in (b)) for which the greedy algorithm does not output a correct solution.

Solution:

(a) When the intervals are sorted by increasing start time the solution does not have to be correct. Consider this counterexample:

$$I_1 = (1, 6),\ I_2 = (2, 3),\ I_3 = (4, 5).$$

The optimal solution is $\{I_2, I_3\}$ while the greedy algorithm will return just the one interval $\{I_1\}$.

(b) When the intervals are sorted by increasing finish time the solution us optimal.

This is the analysis from the class notes:

(a) *Assume greedy is different than OPT*

(b) *Let $i_1, i_2, \ldots, i_k$ denote the set of intervals selected by greedy*
We are ordering them so that $i_1 < i_2 < \cdots$

(c) *Let $j_1, j_2, \ldots, j_m$ denote set of intervals selected by OPT*
Again we are ordering them so that $j_1 < j_2 < \cdots$
Since an interval must end before the next one starts this means $f_{j_r} \leq s_{j_{r+1}}$ for all r

(d) *Note that by the definition of OPT, $m \geq k$.*

(e) *Find largest possible value of r such that $i_1 = j_1$, $i_2 = j_2$, …, $i_r = j_r$.*
This means that we can write OPT as

$$OPT = i_1, i_2, \ldots, i_r, \mathbf{j_{r+1}}, j_{r+2}, \ldots, j_m.$$

Note that it's possible that $r = 0$.

(f) *By the definition of the greedy algorithm j_{r+1} can not finish before i_{r+1}, otherwise the greedy algorithm would have chosen j_{r+1}. This means that $f_{i_{r+1}} \leq f_{j_{r+1}}$*

(g) *From the comment at the end of (iii) this automatically implies that $f_{i_{r+1}} \leq f_{j_{r+1}} \leq s_{j_{r+2}}$.*

This simply means that we can replace j_{r+1} with i_{r+1} in OPT and still have a set of compatable coverings. This gives us

$$OPT' = i_1, i_2, \ldots, i_r, \mathbf{i_{r+1}}, j_{r+2}, \ldots, j_m.$$

Now note that $size(OPT') = m = size(OPT)$ so OPT' is also optimal.

(h) We have therefore found an optimal solution that agrees with OPT with at least $r + 1$ places instead of r places.

(i) We can repeat lines (i)-(viii) until we have found an optimal solution OPT that agrees with greedy on all of its first k places

(j) If $m > k$ then greedy could have chosen j_{k+1} after choosing i_k. Since greedy did not choose it, j_{k+1} does not exist so $m = k$.

(k) Since $m = k$, greedy is optimal

Marking Notes:

(a) required a "No" answer + a correct counter example illustrating that the size (number of intervals) of a Greedy solution can be smaller than that of an optimal solution.

(b) required a "Yes" answer. Key steps in the lecture proof were as follows:
Assume that Greedy is different from Optimal and define the two solutions.
Define the distance between Greedy and Optimal, i.e., the first place that the two solution differs.
Define the one-step modification of Optimal into OPT*.
Justify why the modification is legal (compatible with the remaining intervals chosen by optimal).
Justify why the modified solution OPT* is still optimal (number of intervals selected remains the same after the modification).
Repeat the process until OPT* is converted into Greedy. Greedy thus is optimal.

Note that an induction proof was also possible.

Common mistakes encountered were:

(1) **Mistaking the problem for interval partitioning and discussing the classroom allocation problem instead.**

(2) **The problem asks for "a maximum size set of mutually compatible intervals"; size here means the number of intervals, *not* the total length of the intervals.**

(3) **A counterexample was drawn but there was no EXPLANATIOn as to why it showed that greedy is not optimal.**

(4) **Providing pseudo code or algorithm description rather than the formal proof as shown in the lecture note.**

(5) **Not using the ording given in the problem at all in the proof. (Without using the ordering it is impossible to differentiate between (a) and (b)).**

(6) **In the proof by incremental modification, not giving a clear definition of the position (say r) in which the two algorithms differ – only stating that there is "a location" in the solution. Alos, some notational mistakes showing that sizes of Greedy solution and optimal solution are the same.**

(7) **In the proof by incremental modification, providing no explanation as to why that the finishing time of the Greedy choice is smaller or equal to that of the optimal solution. Some got the direction of the proof wrong.**

(8) **In the proof by incremental modification, saying the modification is legal without explanation. Some proved that it is legal for $r-1$, but do not prove it for $r+1$.**

(9) **In the proof by incremental modification, not explaining why after modification OPT* is still optimal.**

(10) **In the proof by incremental modification, saying that Greedy is optimal immediately after the one-step modification. It is necessary to *repeat* the process (not by induction here) until OPT* is completely converted into Greedy.**

(11) **In the proof by induction, not providing the base case.**

(12) **In the proof by induction it was necessary to assume**

that Greedy is optimal for *any* input of a size *smaller* than n (for inductive proof on n), not just a particular problem of size $n-1$. Not making this explicit was an error.

(13) In the proof by induction, understanding that Greedy is optimal means understanding that the *size* of the Greedy output is the same as the size of the optimal output, but their actual choices can be different.

4. Sorting [20 pts]
   This problem has three parts. Each part is on a different page.

   (a)

   i. What does it mean to say that a sorting algorithm is *stable*?

   Solution: A sorting algorithm is STABLE if two items with the same value appear in the same order in the sorted array as they did in the initial array.

   ii. Give the names of two stable sorting algorithms that you learned in class.

   Solution: Mergesort and Insertion Sort are Stable.
   So are Selection Sort and Bubble Sort (as defined in class and the tutorial).
   Counting Sort and Radix sort are stable as well.

   iii. Give the name of a sorting algorithm that you learned in class that is not stable.

   Solution: Quicksort and Heapsort are NOT stable.
   Note. Quicksort COULD be made stable but the variation we learned in class is not stable.

Continued on next page ....

(b) The problem input is n/k lists:
(i) Each list has size k, and
(ii) for $i = 2$ to n/k, the elements in list $i - 1$ are all less than all the elements in list i

The obvious algorithm to fully sort these items is to sort each list separately and then concatenate the sorted lists together. This uses $\frac{n}{k}O(k \log k) = O(n \log k)$ comparisons.

Show that this is the best possible. That is, any comparison-based sorting algorithm to sort the n/k lists into one sorted list with n elements will need to make at least $\Omega(n \log k)$ comparisons.

You may use any theorem or fact that we learned in class as long as you explicitly write out the statement of that theorem or fact. You may also use any fact from the Math sheet distributed with the exam. If you use a fact from the sheet you must explicitly identify that fact and state that it is from the sheet.

Continued on next page ....

Solution:

This is directly from the tutorial.

Let L_i, $i = 1, \ldots, n/k$ be list i.
The final list is the concatenation $L = L_1 L_2 \ldots L_{n/k}$.

Each list L_i has $k!$ possible orderings.

Since elements in L_{i-1} are less than the elements in L_i, a final ordering can be any ordering of L_1, followed by any ordering of L_2, etc..

Then the total *number of possible output orderings of L is*

$$(k!)^{n/k}$$

In other words, the decision tree for sorting these n elements contains at least $(k!)^{n/k}$ leaves.

The fact from class that we will use is that

An algorithm in the decision tree model that has m different outputs has running time $\Omega(\log m)$..

Note that

$$\begin{aligned} \log((k!)^{n/k}) &= \frac{n}{k} \cdot \log(k!) \\ &= \frac{n}{k}\Theta(k \log k) \\ &= \Theta(n \log(k)) \end{aligned}$$

Therefore, any comparison-based sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst-case for solving this problem.

The fact that $k! = \Theta(k \log k)$ is Stirling's approximation and is from the Math Sheet.

Marking Note: You could NOT use the theorem proven in class that a sorting algorithm requires $\Omega(n \log n)$ time to solve this problem. It doesn't apply to the scenario described here.

Instead you either needed to use the specific facts stated above OR prove the fact from scratch, e.g., using the facts about binary tree height from the math sheet.

Also, as explicitly noted in the tutorial from which this problem was taken, it would NOT be correct to state that

- **it requires $\Omega(k \log k)$ time to sort each list and**
- **there are n/k lists**
- **thus the total time required to sort all of them is $\Omega((n/k))k \log k) = \Omega(n \log k)$.**

The theorems don't work that way.

More specifically, let P_i be the problem of sorting list i and P the problem of sorting the union of all of the lists. Let $T(P_i)$ and $T(P)$ be the number of comparisons needed to solve each of those problems. The best that you can know when starting is that

$$T(P) \geq \max_i T(P_i) \quad \textbf{and} \quad T(P) \leq \sum_i T(P_i).$$

You can NOT assume that

$$\sum_i T(P_i) \leq T(P_i).$$

This is because, perhaps, there is some clever way of interleaving the sorts of the different lists that permits saving comparisons. Part (b) is about PROVING that no such clever way exists.

(c) The leftmost array gives 8 3-digit numbers. Run Radix-Sort on these numbers. Show the result after sorting the array on each digit. The rightmost array should be your final result.

3	2	2
8	1	3
1	2	3
4	3	2
3	5	2
4	6	2
7	1	8
4	9	8

Solution:

3	2	2
8	1	3
1	2	3
4	3	2
3	5	2
4	6	2
7	1	8
4	9	8

3	2	2
4	3	2
3	5	2
4	6	2
8	1	3
1	2	3
7	1	8
4	9	8

8	1	3
7	1	8
3	2	2
1	2	3
4	3	2
3	5	2
4	6	2
4	9	8

1	2	3
3	2	2
3	5	2
4	3	2
4	6	2
4	9	8
7	1	8
8	1	3

5. **Indicator Random Variables** [15 pts]

(a) Recall the Hat-Check problem from the tutorial that was analyzed using the indicator random variable technique. In that problem, each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order.

What is the expected number of customers who get back their own hat? Prove the correctness of your answer.

Solution:

The answer is 1.

In class we showed how to solve this using the indicator random variable technique

Let

$$X_i = \begin{cases} 1 & \textit{if person } i \textit{ gets their own hat back} \\ 0 & \textit{if person } i \textit{ does not get their own hat back} \end{cases}$$

Let Y be the total number of people who get their own hat back. Then

$$Y = \sum_{i=1}^{n} X_i$$

so, by **Linearity of Expectation**

$$E(Y) = E\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} E(X_i)$$

Each person gets a random hat so each person has probability $1/n$ of getting their own hat back. Thus

$$E(X_i) = 1 \cdot \Pr(X_i = 1) + 0 \cdot \Pr(X_i = 0) = \Pr(X_i = 1) = \frac{1}{n}.$$

Then

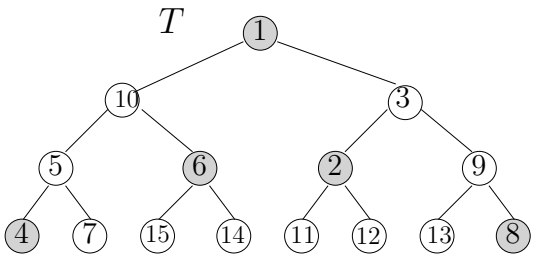
$$E(Y) = \sum_{i=1}^{n} E(X_i) = n\frac{1}{n} = 1$$

(b) For the purpose of this problem assume $n = 2^k - 1$ where k is a positive integer. Let $A[1 \ldots n]$ be an array. Recall the correspondence between arrays A and trees T that we learned in the Heapsort lecture.

The root is in array position 1. For any element in array position i, its left child is in position $2i$ and its right child is in position $2i + 1$.

Define a *local minimum* of the tree T to be a node in the tree whose value is less than all of its neighbors (nodes that it connects to).

In the example below, the shaded nodes are the local minima.



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$A[i]$	1	10	3	5	6	2	9	4	7	15	14	11	12	13	8

Now suppose that the elements of A form a uniform random permutation of $< 1, 2, \ldots, n >$. That is, each of the $n!$ possible permutations is equally likely to occur.

Use the indicator random variable technique to calculate the expected number of local minima in T if the elements of A form a uniform random permutation of $< 1, 2, \ldots, n >$. Show your work.

Hint: Let $X_i = 1$ if $A[i]$ is a local minimum and $X_i = 0$ if it is not. Calculate the expected value of $\sum_{i=1}^{n} X_i$.

Solution:

We solve this problem using the indicator random variable technique.

Let

$$X_i = \begin{cases} 1 & \text{if node } i \text{ is a local minimum} \\ 0 & \text{if node } i \text{ is not a local minimum} \end{cases}$$

Given that $n = 2^k - 1$ and the tree T is constructed out of the array A layer by layer starting from the root, T is a full binary tree of height $k - 1$.

As shown in the figure, the tree T contains three types of nodes:

(1) 1 root node with no parent and two children;

(2) $2^k - 1 - 1 - 2^{k-1} = 2^{k-1} - 2 = \frac{n-3}{2}$ internal nodes that links to its parent and two children; and

(3) $2^{k-1} = \frac{n+1}{2}$ leaf nodes connecting to the parent but with no children.

We examine the three cases separately.

(1) Root Node: The root node has two neighbors that collectively form a local neighborhood of three $(A[1], A[2], A[3])$. Consider the three items thrown into these three locations. Since A is a random permutation, meaning that each item is equally likely to end up in any of the three locations, the probability that the smallest one is in $A[1]$ is just $\frac{1}{3}$.

But $A[1]$ the root node is a local mimimum if and only if $A[1]$ is the smallest of the three. This shows that,

$$E(X_1) = \Pr(X_1 = 1) = \frac{1}{3}.$$

(2) Internal Nodes: Internal node $A[i]$ and its three neighbors form a local neighborhood of four $(A[\lfloor i/2 \rfloor], A[i], A[2i], A[2i + 1])$. Consider the four items thrown into these locations. Each one is equally likely to be the smallest of the four, i.e., the probability that $A[i]$ is a local minimum is $\frac{1}{4}$. Thus for any internal node $A[i], 1 < i < n$,

$$E(X_i) = \Pr(X_i = 1) = \frac{1}{4}.$$

(3) Leaf Nodes: Smilarly, a leaf node $A[i]$ that forms a local neighborhood of two with its parent $(A[\lfloor i/2 \rfloor], A[i])$, so

$$E(X_i) = \Pr(X_i = 1) = \frac{1}{2}.$$

Combining pieces and using linearity of expectation gives

$$E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) = 1 \cdot \frac{1}{3} + (2^{k-1} - 2) \cdot \frac{1}{4} + (2^{k-1}) \cdot \frac{1}{2} = \frac{3}{4} \cdot 2^{k-1} - \frac{1}{6}$$

or, equivalently,

$$E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) = 1 \cdot \frac{1}{3} + \left(\frac{n-3}{2}\right) \cdot \frac{1}{4} + \left(\frac{n+1}{2}\right) \cdot \frac{1}{2} = \frac{9n+5}{24}$$

Marking Note: By definition, the total number of local minima is at most the number of items, i.e., n. It is therefore impossible that $E(\sum_{i=1}^{n} X_i) > n$. Solutions that stated this result, e.g., by reporting the answer as $\Theta(n^2)$, were therefore totally incorrect.

Also, a correct solution required understanding that there are THREE different types of nodes and each one requires a slightly different analysis.

6. **Dynamic Programming** [22pts]
You are given an input array $A[1 \dots n]$. Recall that the *maximum-contiguous subarray (MCS)* is a subarray $A[i \dots j]$ such that $\sum_{k=i}^{j} A[k]$ is maximum among all subarrays. As an example, $A[4 \dots 7]$ is a MCS of the array below

k	1	2	3	4	5	6	7	8	9	10
A[k]	−3	3	−5	18	−1	2	8	−50	−30	5

.

and has value $18 - 1 + 2 + 8 = 27$.

The problem is now modified so that for given $x > 0$ you need to find the *x-discounted MCS*. The x-discounted *cost* of $A[i \dots j]$ is defined as

$$C(i, j : x) = A[j] + xA[j-1] + x^2 A[j-2] + \cdots + x^{j-i} A[i] = \sum_{k=0}^{j-i} A[j-k]x^k.$$

In the example array above,

$$C\,(2, 7 : 2) = 8 + 2 \cdot 2 - 2^2 \cdot 1 + 2^3 \cdot 18 - 2^4 \cdot 5 + 2^5 \cdot 3 = 168$$

Given $x > 0$, the *x-discounted* MCS is the subarray $A[i \dots j]$ such that $C(i, j : x)$ is maximum among all subarrays.

By definition, if $x = 1$, the *x-discounted MCS* is exactly the MCS. If $x \neq 1$ it might be different.

In the array above, for example, if $x = 2$, then $A[2 \dots 7]$ is the *x-discounted* MCS of the full array.

The full problem is, given array $A[1 \dots n]$ and $x > 0$, to design an $O(n)$ time dynamic programming algorithm that calculates the cost of the x-discounted MCS.

Do this by completing parts (A),(B), (C) and (D) listed on the following page. Solutions (except for part (B) should be written on pages 20 and 21.

Continued on next page ....

Student ID: _____

(A) Prove that, for every i, j with $1 \leq i < j \leq n$ and $x > 0$,

$$C(i, j : x) = A[j] + xC(i, j - 1 : x).$$

Note: For clarity, we emphasize that the definition implies $C(j, j : x) = A[j]$. Also, if you are not able to prove part (A) you may still assume its correctness later in the problem.

(B) Define

$$V_j = \max_{1 \leq i \leq j} C(i, j : x).$$

Give a recurrence relation (write it in the space below) for V_j in terms of V_i with $i < j$ and the values in the array.

$$V_j = \begin{cases} \_\_\_\_\_ & \text{if } j = 1 \\ \_\_\_\_\_ & \text{if } j > 1 \end{cases}$$

Justify the correctness of your recurrence relation

(C) Give documented pseudocode for your DP algorithm to calculate the cost of the x-discounted MCS (based on the recurrence from part (B)), and explain why it is correct.

(D) Analyze the running time of your algorithm. Full credit will be given for $O(n)$ algorithms. Algorithms slower than that will only receive partial credit.

Solution.
(A) This follows directly from the definition

$$\begin{aligned} C(i,j:x) &= \sum_{k=0}^{j} A[j-k]x^k \\ &= A[j] + \sum_{k=1}^{j-i} A[j-k]x^k \\ &= A[j] + \sum_{t=0}^{j-1-i} A[j-(t+1))]x^{t+1} \\ &= A[j] + x\sum_{t=0}^{j-1-i} A[j-1-t]x^t \\ &= A[j] + xC(i,j-1:x) \end{aligned}$$

(B)

$$V_j = \begin{cases} \underline{A[1]} & \text{if } j = 1 \\ \underline{\max\left(A[j], A[j] + xV_{j-1}\right)} & \text{if } j > 1 \end{cases}$$

To see that this is correct note that

$$V_1 = \max_{1\le i\le 1} C(i,j:x) = C(1,1:x) = A[1].$$

For $j > 1$, from part (A) and the fact that $C(j,j:x) = A[j]$, we find

$$\begin{aligned} V_j &= \max_{1\le i\le j} C(i,j:x) \\ &= \max\left(A[j], \max_{1\le i\le j-1} C(i,j:x)\right) \\ &= \max\left(A[j], \max_{1\le i\le j-1} (A[j] + xC(i,j-1:x))\right) \\ &= \max\left(A[j], A[j] + x\max_{1\le i\le j-1} C(i,j-1:x)\right) \\ &= \max\left(A[j], A[j] + xV_{j-1}\right) \end{aligned}$$

(C)
The problem asks you to calculate

$$XDMCS = \max_{1 \le i \le j} C(i, j : x).$$

But this is just

$$XDMCS = \max_{1 \le i \le j} C(i, j : x) = \max_{1 \le j \le n} \left(\max_{1 \le i \le j} C(i, j : x) \right) = \max_{1 \le j \le n} V_j.$$

So, to solve the problem, it suffices to calculate all of the V_j and then return the max value. From the recurrence relation, this can be done as below:

% Initialize
1. $V[1] = A[1]$;

% Calculate the V_j
2. For $j = 1$ *to* n *do*
3. $\quad V[j] = \max(A[j], A[j] + xV_{j-1})$

% Return $\max_{1 \le j \le n} V_j$.
4. $XDMCS = V[1]$
2. For $j = 2$ *to* n *do*
3. $\quad$ *If* $V[j] > XDMCS$ *then*
3. $\quad\quad XDMCS = V[j]$

(D) The algorithm has two for loops that are implemented $O(n)$ *times. Each for loop does* $O(1)$ *work each time its called so the entire algorithm is* $O(n)$.

Marking Notes:
Part (A) was provided to give you the tools to prove part B.

Our solutions for Part (A) and the justification for Part (B) were written out in length. It was not necessary to provide ALL of the detail to get full marks.

The intent for this problem was that that students would use logic similar to that given in the $O(n)$ time DP tutorial solution for the maximum contiguous subarray problem. The solution to this problem only required minor modification to that.

For Part B you at least needed, though, to explicitly explain why the recurrence relation for V_j was correct. The easiest way would be through some direct application of Part (A).

Many students who wrote the correct recurrence did not provide any justification and had a few points deducted because of that.
In particular, many students understood that the two cases for V_j were either $V_j = A[j]$ or $V_j = A[j] + xV_{j-1}$. The problem was that they never explained why those were the only two cases. WHY, couldn't $V_j = C(i', j : x)$ for some other i'? For full marks it was necessary to explain why this could not happen.

One major issue was that some students wrote their recurrences in terms of i. A common example was

$$V_j = \begin{cases} A[1] & \text{if } j = 1 \\ \\ \max\left(A[j], A[j] + xC(i, j1 : x)\right) & \text{if } j > 1 \end{cases}$$

This recurrence was incorrect because i is a non-defined

variable so the statement is not well-defined. More generally some students answered part (B) in terms of $C(i, j : x)$ and NOT in terms of V_{j-1} (or some other V_i).
There was NO way to fix this problem. If your solution was stated in terms of an undefined i is was wrong because it could not be justified.
The only way that this could be well-defined is if the max was then explicitly taken over all i, but in this case the algorithm would no longer be $O(n)$.

Another issue that frequently occurred is that students confused V_j with the solution to the $XDMCS$ problem. It was not. As described in part (C), the final solution was

$$XDMCS = \max_{1 \le j \le n} V_j.$$

Trying to fold the final solution into the definition of V_j let to incorrect recurrences.
An example of this was the incorrect recurrence

$$V_j = \begin{cases} A[1] & \textbf{if } j = 1 \\ \\ \max\left(V_{j-1}, A[j] + xV_{j-1}\right) & \textbf{if } j > 1 \end{cases}$$

Consider setting $x = 2$ and the V_j values generated by the recurrence above on the input below:

j	1	2	3	4	5	6
A[j]	5	-50	-50	-50	3	1
V_j	5	5	5	5	13	27

.

The recurrence would return the value 27 but the actual correct solution is $C(5, 6) = 1 + 3 * 2 = 7$. Another incorrect recurrence was

$$V_j = \begin{cases} A[1] & \textbf{if } j = 1 \\ \\ \max\left(A[j], V_{j-1}, A[j] + xV_{j-1}\right) & \textbf{if } j > 1 \end{cases}$$

This recurrence would exhibit exactly the same problem as the previous one on the input given.

Finally, full points were only awarded for parts (C) and (D) if the recurrence in part (B) was correct.