CS570 Summer 2022: Analysis of Algorithms Exam I

	Points		Points
Problem 1	20	Problem 5	12
Problem 2	10	Problem 6	16
Problem 3	14	Problem 7	16
Problem 4	12		
	Total	**100**	

Instructions:

1. This is a 2-hr exam. Open book and notes. No electronic devices or internet access.
2. If a description to an algorithm or a proof is required, please limit your description or proof to within 150 words, preferably not exceeding the space allotted for that question.
3. No space other than the pages in the exam booklet will be scanned for grading.
4. If you require an additional page for a question, you can use the extra page provided within this booklet. However please indicate clearly that you are continuing the solution on the additional page.
5. Do not detach any sheets from the booklet. Detached sheets will not be scanned.
6. If using a pencil to write the answers, make sure you apply enough pressure, so your answers are readable in the scanned copy of your exam.
7. Do not write your answers in cursive scripts.
8. This exam is printed double sided. Check and use the back of each page.

1) 20 pts
Mark the following statements as **TRUE** or **FALSE** by circling the correct answer. No need to provide any justification.

[TRUE/FALSE]
If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.

[TRUE/FALSE]
In the Gale-Shapley algorithm, it is possible for men to end up with their best valid partners when women propose.

[TRUE/FALSE]
When following Kruskal's algorithm, the greedy choice at each step is to add the edge of least cost to the forest so long as its addition does not create a cycle.

[TRUE/FALSE]
Suppose that a depth first search on a DAG G starting from node s succeeds in finding all nodes in G. Then s must be the first node in every topological ordering of G.

[TRUE/FALSE]
The best-case time complexity of insertion sort is $O(n^2 + logn)$.

[TRUE/FALSE]
The Extract_Max operation on the max-heap array: [16, 3, 14, 1, 2, 9, 7] will require only two comparisons and one swap.

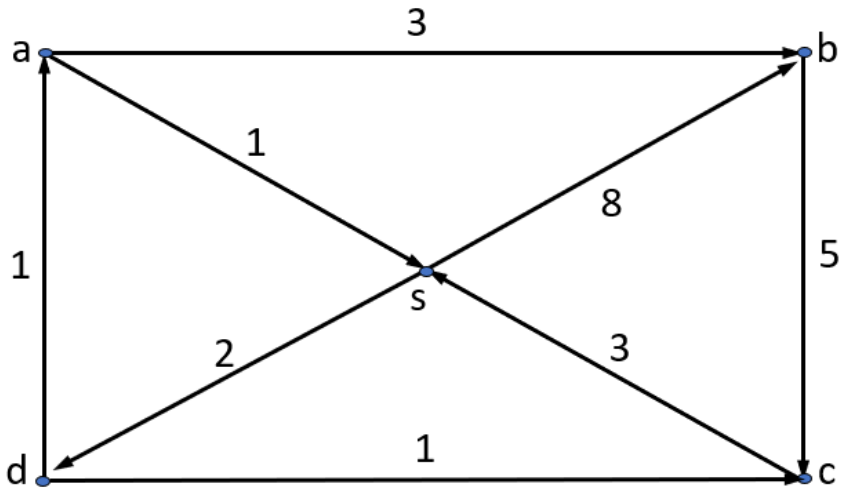
[TRUE/FALSE]
In a divide-and-conquer algorithm, the divide step can be slower than the combine step

[TRUE/FALSE]
A binary max heap can be converted to a binary min heap in $O(n)$ time.

[TRUE/FALSE]
Divide-and-conquer and dynamic programming both require two passes to find the optimal solution—a bottom up pass to find the value of the optimal solution and a top down pass to find the optimal solution.

[TRUE/FALSE]
The recursive implementation of a dynamic programming algorithm has a higher time complexity than the equivalent iterative implementation of that same algorithm.

2) 10 pts
Consider the directed graph given below.



Fill out the following table with the shortest distances from s to each node in the graph after each iteration of Dijkstra's algorithm. Don't forget the initial values.

Iteration	s	a	b	c	d
Initial values	0	∞	∞	∞	∞
1	0	∞	8	∞	2
2	0	3	8	3	2
3	0	3	6/8	3	2
4	0	3	6	3	2

Rubrics:
Initial Values - 2pts
Correct values for all nodes at each iteration - 2pts

3) 14 pts

For each of the following, indicate whether $f = O(g)$ or $f = \Omega(g)$ or $f = \Theta(g)$ (i.e. both $f = O(g)$ and $f = \Omega(g)$)

	$f(n)$	$g(n)$
1	$n^{1/3}$	$n^{3/5}$
2	$nlog(n)$	$n^2log(n^2)$
3	$2^{log(n^2)}$	n^2
4	$log(n)$	$log(log(5^n))$
5	$n^{1/3}$	$(log(n))^3$
6	2^n	2^{3n}
7	$n^4/log(n)$	$n(log(n))^4$

Solution:

1. $f = O(g)$
2. $f = O(g)$
3. $f = \Theta(g)$
4. $f = \Theta(g)$
5. $f = \Omega(g)$
6. $f = O(g)$
7. $f = \Omega(g)$

Rubrics:

1. +2 for each correct answer
2. +1 if solution is $\Theta(g)$ (viz, 3rd and 4th subproblem), but $\Theta(g)$ is mentioned along with either $O(g)$ or $\Omega(g)$ (but not both)

4) 12 pts (3 for each correct)
For the given recurrence equations, solve for *T(n)* if it can be found using the Master Method. Else, indicate that the Master Method does not apply.

a) $T(n) = 4T(n/2) + n^{\sqrt{2}}$

O(n^2)

b) $T(n) = 4T(n/2) + n^{2.5}$

O(n^2.5)

c) $T(n) = 4T(n/2) - n^2 \log^2 n$

Master theorem does not apply

d) $T(n) = 4T(n-2) + n^2$

Master theorem does not apply

5) 12 pts

Imagine you are in a car, and you want to travel on a road. You need to go up and down several hills on this road. Fortunately, your car is specially equipped with a nitrogen booster in addition to a regular gasoline burning engine. You can only use the booster for a limited number of times though (*k*), and you have a certain amount of gas (*m* gallons) in your tank at the start of your journey.

The car does not need to burn any gas or use the booster when you are coming down the hills or going straight on a flat road. But for each hill (with elevation gain of "*h*") that you want to go up, you need to use one of these two options:
1) You can use the engine and burn '*h*' gallons of gas, or
2) You can spend one of your *k* special boosters (if you have any left). The booster can take the car from any height to any height.

You are given an array of heights for the hills you need to pass *hills[1..n],* in the order in which you encounter the hills on your journey. Your job is to find the maximum number of hills that you can pass with the given amount of gas (*m* gallons) and the given number of boosters (*k*). In other words, you need to design an algorithm so that you can reach the furthest on this road by optimally using either your boosters or gas to climb up each hill. Your algorithm should run in O(*n logk*). Explain your algorithm. How much space does it take?

2 possible interpretations of the hill elevations:

1) we come down from each hill after going up from it. So each element i of the hill requires h[i] amount of gas (or 1 booster) .
2) we should go from h[i] to h+[i+1] directly. So we require h[i+1] - h[i] units of gas (or 1 booster)

Both interpretations are valid. for 2) we should consider only h[i+1]-h[i] which are positive. then it will reduce to 1). you should also keep track of the indexes as well. (Thre rest of the solution uses the term 'up-hill' which can map to either interpretation.

Intuition: We should allocate the boosters for the largest 'up-hills' and use the gas tank for the smaller ones.

Algorithm: start from the beginning of the array and use the boosters for the first K 'up-hills' . We will keep track of the 'up-hills' in a min-heap. After the k-th' up-hill, if we encounter an 'up-hill' that

1) is larger than the root of the heap, we should replace it in the heap (extract the old min and insert the new hill). then we should allocate gas for the extracted hill.
2)is less than the root : we should cover it with gas.

In either case, if we don't have enough gasoline left, it means that we have reached the maximum that we can achieve. the algorithm should return the index here.

Time complexity: Each exchange from the heap (extracting and inserting) takes O(logK) time (because the maximum number of elements in the heap is K). we are doing these operations for a maximum of 2*N time (maximum one time for putting into the heap and one time pulling it out of the heap) So the time complexity is O(NlogK).

Space complexity: we are storing a heap of size K. So the space complexity is O(K)

Rubric:
Intuition (using boosters for largest hills) → 2
Using heap (or priority queue) to store the values → 2
Implementation correctness → 4
Which is the maximum hill you can reach? → 1
Time complexity (being O(nlogk) and not O(nlogn)) → 2
space complexity (based on the given algorithm) → 1

6) 16 pts

You are a weapons specialist and are tasked with creating a weapons arsenal by choosing from a set of *n* weapons. You are given the cost as well as the power associated with each weapon. Your goal is to maximize the total power level of the weapons you select. For this, you need to create a proposal to your bosses, but the catch is that the proposal will be rejected if there are any two weapons *i* and *j* in your proposal where $Cost[i] \geq Cost[j]$ but $Power[i] < Power[j]$.

Given the cost of each weapon (*Cost[1....n]*) and the power for each (*Power[1....n]*), choose a group of weapons so that you maximize the power in such a way that the proposal is not rejected. Develop an efficient dynamic programming solution to return the maximum total power achievable by your arsenal.

Note: the proposal does NOT need to meet a maximum total cost limit.

I. Define (in plain English) the subproblems to be solved. (4 pts)

Sort weapons in non-decreasing order of cost, i.e. the Cost array will be in non-decreasing order of cost and the Power array will correspond to that order.

OPT(i) is the maximum power achieved by selecting weapons from 1 to i with the i^{th} weapon selected last, i.e. the i^{th} weapon is always included in OPT(i).

II. Write a recurrence relation for the subproblems. (4 pts)

OPT(i) = Power[i] + { Max(OPT(j)) for j=0 to i-1 where Power[i]≥Power[j] }

Continued on the next page

III. Using the recurrence formula in part II, write pseudocode using iteration to compute the maximum total power that can be achieved. (6 pts total)
Make sure you specify:

i. the base cases and their values (1 pts)

OPT(0) = 0

ii. rest of the pseudocode (4 pt)

The pseudo-code should show the pre-sorting, the recurrence and the base cases as aforementioned. Additionally and most importantly, it should show the order of computation (i.e. loop variable order): for (i=0 to n).

iii. where the final answer can be found (e.g. *OPT(n)*, or *OPT(0,n)*, etc.) (1 pt)

Max(OPT(i)) for i=1 to n

IV. What is the run time complexity of your solution? (2 pts)

$O(n^2)$

7) 16 pts

Suppose you are the professor teaching CS570. After exam I, you see that all students get fantastic grades, so you decide to gift them with algorithm books. Now there are n students sitting in a line and their grades can be seen in the integer array g[1..n].
You will give rewards to the students based on the following criteria:

1. Each student receives at least one book
2. For any two students i and j sitting next to each other, if g(i) > g(j) then number of books given to i must be greater than that given to j
3. For any two students i and j sitting next to each other, if g(i) = g(j) then the same number of books should be given to i and j

You need to find the minimum number of books you must distribute.

Example: g = [80,100,100,90]
Solution: Minimum number is 6
You can give 1 book to the first student, give 2 books to the second and third students, and finally one book to the fourth student.

B) Find the minimum number of books you must distribute when
g = [91,92,92,95,94,93,92] (2 pts)

C) Provide a divide and conquer based algorithm to solve this problem and analyze its worst-case running time complexity. (14 pts)

Solution/rubric 7:

A) (0 points)

Part A does not exist. (a typo in numbering)

B) (2 points)

15 books, [1,2,2,4,3,2,1].

Wrong number (anything but 15): 0 points
Correct number 15 but work missing/incorrect: 1.5 points
Correct number and correct work: 2 points

C) (14 points)

Let's call the two arrays g[] for grades and b[] for books.

Divide-and-conquer algorithm:

Divide:

Recursively divide the arrays into 2 halves.
(2 points, no partial credit.)

Conquer:

At the base case of arrays of size 1, each student gets 1 book.
(2 points, no partial credit. Calling it "initialization" of the books array or equivalent language is also acceptable.)

Combine:

For each two arrays that need to be joined, compare the edge elements to each other (last element of the first array to the first element of the second array).

We call the elements of the two arrays that are compared b1/g1 (rightmost element of the left array) and b2/g2 (leftmost element of the right array), where b=book and g=grade. In the following, x and y can stand for 1 or 2, respectively, with x being unequal to y.

Case 1 (grades equal, books unequal):

gx=gy and bx<by:
Set bx to by and preserve consistency(*)

Case 2 (higher grade does not have more books than neighbor):

gx<gy and bx>=by:
Set by to bx+1 and preserve consistency(*)

Default: Do nothing and combine the arrays without any changes.

Finding these 2 cases (can be expressed as more than 2): 2 points
Describing only 1 case: 1 point

Solving both cases (independent of consistency preservation): 2 points

Solving one case (independent of consistency preservation): 1 point
Flaws in reasoning (e.g. just incrementing a value of an element by 1 instead of setting it to a higher value than the other one): Given points for a faulty case are halved

(*) Consistency preservation:

This is based on the consideration that after we increment the edge element of an array, we need to make sure that all equalities or inequalities between adjacent elements in the subarray whose edge element was changed stay preserved as required by the rules. Think of it as a potential “ripple effect” that goes on up to the first pair of elements whose equality or inequality is unaffected by the increment. We need to stop at that element because we want our algorithm to assign the minimum number of books that fulfills the rules.

1. Set the number of books of the next element in the array to be joined as high as necessary to preserve the equality or inequality that existed before the change prescribed in case 1 or 2.
2. Continue with the next element until you reach one that does not need to be changed.

Preserving consistency: 3 points (no partial credit)

Runtime: O (n log n), either by case 2 of Master Theorem or other reasoning.

We divide the arrays recursively log n times. At each level, we perform at most n operations.

Correct runtime with correct reasoning: 2 points
Correct runtime with no/faulty reasoning: 1 point

General:

- Different divide-and-conquer algorithms are evaluated on their merits. Algorithms of any other type are NOT considered since we are asking for a divide-and-conquer algorithm.
- Since we are not asking for pseudocode, any pseudocode will only be considered to fill gaps of the other text. Where the two are in conflict, only the text is considered.
- Step descriptions that do not give more instructions than to “make this array/subarray/pair of elements comply with the rules” cannot be considered, since otherwise the entire questions could be solved by “make the books array comply with the rules and add up the numbers”.