University of New Mexico
Department of Computer Science

Midterm Examination

CS 362 Data Structures and Algorithms
Spring, 2007

Name:
Email:

- Print your name and email, *neatly* in the space provided above; print your name at the upper right corner of *every* page. Please print legibly.
- This is an *closed book* exam. You are permitted to use *only* two pages of "cheat sheets" that you have brought to the exam and a calculator. *Nothing else is permitted.*
- Do all five problems in this booklet. *Show your work!* You will not get partial credit if we cannot figure out how you arrived at your answer.
- Write your answers in the space provided for the corresponding problem. Let us know if you need more paper.
- Don't spend too much time on any single problem. The questions are weighted equally. If you get stuck, move on to something else and come back later.
- If any question is unclear, ask us for clarification.

Question	Points	Score	Grader
1	20		
2	20		
3	20		
4	20		
5	20		
Total	100		

1. **Short Answer**

Multiple Choice:

The following choices will be used for the multiple choice problems.

(a) $\Theta(1)$
(b) $\Theta(\log^* n)$
(c) $\Theta(\log n)$
(d) $\Theta(\sqrt{n})$
(e) $\Theta(n)$
(f) $\Theta(n \log n)$
(g) $\Theta(n^2)$
(h) $\Theta(n^3)$
(i) $\Theta(2^n)$

For each of the questions below, choose one of the above possible answers. Please write the letter of your chosen answer to the left of the question.

(a) Time to find the optimal way to parenthesize a list of n matrices so that the number of scalar multiplications to compute their produce is minimized *Solution:* $\Theta(n^3)$

(b) Amortized cost of a call to Find-Set using the union-find data structure over n elements with union by tree size (smaller tree under larger tree) but without path compression. *Solution:* $\Theta(\log n)$

(c) Solution to the recurrence $T(n) = 4T(n/2) + \log n$ *Solution:* $\Theta(n^2)$

(d) Solution to the recurrence $T(n) = 2T(n-1) + 1$ *Solution: The annihilator is* $L - 2$ *for the homogeneous part and* $(L-1)$ *for the non-homogeneous part so the solution is* $c_0 2^n + c_1$ *which is* $\Theta(2^n)$

(e) Solution to the recurrence $T(n) = 3T(n-1) - 2T(n-2) + 1$ *Solution: The annihilator is* $(L-2)(L-1)(L-1)$ *so the solution is* $c_0 * 2^n + c_1 n + c_2$ *which is* $\Theta(2^n$

True or False (10 points total). **Circle your final answers.**

(a) If an operation takes $O(1)$ amortized time, then it takes $O(1)$ worst case time. *Solution: False. Amortized time is an average over many operations.*

(b) Any problem that can be solved with a greedy algorithm can also be solved with dynamic programming *Solution: True*

(c) $\log n$ is $o(\sqrt{n})$ *Solution: True. Use L'Hopitals to show this.*

(d) $\log n$ is $\omega(1)$ *Solution: True.* $\log n$ *grows asymptotically faster than any constant.*

(e) A dynamic programming algorithm always uses some type of recurrence relation. *Solution: True*

2. **Amortized Analysis**

Consider a linked list that has the following operations defined on it:

- *AddLast(x)*: Adds the element x to the end of the list
- *RemoveFourths()*: Removes every fourth element in the list i.e. removes the first, fifth, ninth, etc., elements of the list.

Assume these operations have the following costs:

- *AddLast(x)* - cost equals 1
- *RemoveFourths()* - cost equals the number of elements in the list

(a) Assume we perform n operations on the list. What is the worst case run time of a call to RemoveFourths? Justify your answer.

Solution: Worst case is $O(n)$ which happens when we call AddLast() $n-1$ times and then call RemoveFourths()

(b) Now you will show that the amortized cost of these two operations is small using the taxation (accounting) method.

i. First give the amount that you will charge AddLast() and the amount that you will charge RemoveFourths().

ii. Next show how you will use these charges to pay for the actual costs of these operations.

iii. Finally write down the amortized cost per operation.

Solution: AddLast gets charged 5 dollars. RemoveOdds gets charged 0 dollars. When we call AddLast, we spend 1 dollar immediately to pay for the cost of the call, we then store the remaining 4 dollars with the item added to the list. When we call RemoveFourths(), every element in the list has 4 dollars stored with it. We take the 4 dollars from each item that is deleted to pay for the cost of the call to RemoveFourths(). We can do this since $n/4$ items are deleted and so there are n dollars on these deleted items. Thus, at the end of the call to RemoveFourths() all remaining elements in the list still have 4 dollars stored on them. The amortized cost per operation is thus $O(1)$

3. **Knapsack**

Suppose you have a collection of n items $i_1, i_2, ..., i_n$ with weights $w_1, w_2, ..., w_n$ and a bag with capacity W.

Part 1: Describe a simple, efficient algorithm to select as many items as possible to fit inside the bag e.g. the maximum cardinality set of items that have weights that sum to at most W

Solution: First sort the items by weight from lightest to heaviest. Next greedily select items in this sorted list until no more items can be placed in the knapsack

Part 2: Give a concise and rigorous argument that your algorithm is correct.

Solution: The proof is very similar to the proof we saw in class for the activity selection problem. Let L be the set of items selected by this greedy algorithm. Let L' be some other set of items that has total weight no more than W. We will show that $|L'| \leq |L|$. Sort the list L' so the items are in increasing order of weight. Let j be the first index where the j-th item in L is different than the j-th item in L'. By the way that L was constructed (greedily), we know that the weight of the j-th item in L is no more than the weight of the j-th item in L'. Thus we can replace the j-th item in L' with the j-th item in L without increasing the total weight of all the items in L'. We can continue this process for larger j until all of the items in L' are replaced with items in L. This implies that $|L| \geq |L'|$

4. **Dynamic Programming**

Recall that in the 0-1 Knapsack problem, we are given two things. First a list of n items $(v_1, w_1), (v_2, w_2)...(v_n, w_n)$, where item i has value v_i and weight w_i. Second a weight W which is the maximum weight that can be carried in the knapsack. The items are indivisible so that we must either place the entire item in the knapsack or not place it in the knapsack. Our goal is to maximize the sum of the values of all the items that are placed in the knapsack.

Part 1: Professor Clyde claims that a greedy algorithm will solve this problem. Is he correct? (Just answer yes or no, you do not need to justify your answer)

Solution: No. As we talked about in class, a greedy algorithm will not always give the correct answer.

Consider the following Dynamic Programming approach to the problem. For integers i and j, let $f(i, j)$ be the maximum value that can be achieved if only items from the set $(v_1, w_1), (v_2, w_2)...(v_i, w_i)$ are allowed and the backpack can only carry a total weight of j. Note that for all $j > 0$, $f(0, j) = 0$ and for all $i > 0$, $f(i, 0) = 0$.

Part 2: Let $n = 2$, $W = 3$ and the list of items be $(2, 2), (1, 1))$. Your first job is to fill in the three missing entries in this table.

	i= 0	i=1	i=2
j=0	0	0	0
j=1	0	0	
j=2	0	2	
j=3	0	2	

Solution:

	i= 0	*i=1*	*i=2*
j=0	*0*	*0*	*0*
j=1	*0*	*0*	*1*
j=2	*0*	*2*	*2*
j=3	*0*	*2*	*3*

Part 3: Note that if $w_i > j$ then $f(i, j) = f(i-1, j)$. Now write down the recurrence relation for $f(i, w)$ for the case where $w_i \leq j$. Hint: $f(i, j)$ will be the maximum over two quantities - it will depend on w_i, v_i and on previously computed values of the function f.

Solution: $f(i, j) = max(f(i-1, j-w_i) + v_i), f(i-1, j))$

Part 4: Briefly explain how you would use the above recurrence relation to write a dynamic program to solve the 0-1 knapsack problem. How large would your table be? What value of f would you return to as the maximum value that can be fit in the knapsack? What is the run time of your dynamic program?

Solution: You would create a n by W table to store the values of f. The first row and column of this table would be filled in using the two base cases. The remainder of the table would be filled in going top-down, left-to-right. The value of f that is the maximum value that can be fit in the knapsack is $f(n, W)$. The run time of the algorithm is $O(n * W)$

5. **Recurrences**

Consider the following puzzle. There is a row of n chairs and two types of people: M for mathematicians and P for poets. You want to assign one person to each seat but you can never seat two mathematicians together or they will start talking about mathematics and everyone else in the room will get bored. For example, if $n = 3$, the following are some valid seatings: PPP, MPM, and PPM. However, the following is an invalid seating: MMP.

In this problem, your goal is as follows. Let $f(n)$ be the number of valid seatings when there are n chairs in a row. Write and solve a recurrence relation for $f(n)$. Please show your work.

Solution: The trick to solving this problem is to realize that you can get a valid seating of n people in two ways. First, you can take a valid seating of $n-1$ *players and then put a P in the n-th seat. This ensures there won't be two M's together. Second, you can take a valid seating of* $n-2$ *players, put a P in the (n-1)-st seat and then put a M in the n-th seat. This exhausts all the ways of getting a valid seating. Translating this into a recurrence relation gives us that* $f(n) = f(n-1) + f(n-2)$*. When we solve this recurrence relation (with annihilators), we get that* $f(n)$ *is simply the n-th Fibonacci number*