University of New Mexico
Department of Computer Science

First Midterm Examination

CS 362 Data Structures and Algorithms
Spring, 2006

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) $\Theta(\sqrt{n})$
(d) $\Theta(n)$
(e) $\Theta(n \log n)$
(f) $\Theta(n^2)$
(g) $\Theta(n^3)$
(h) $\Theta(2^n)$

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) $4^{\log n}$ *Solution:* $\Theta(n^2)$
(b) Amount of time required by the dynamic programming algorithm for finding the optimal parenthesization of a sequence of n matrices *Solution:* $\Theta(n^3)$
(c) Worst case cost of n calls to Insert in a Dynamic Table *Solution:* $\Theta(n)$
(d) Solution to the recurrence $T(n) = 4T(n/2) + 3$ *Solution:* $\Theta(n^2)$
(e) Solution to the recurrent $T(n) = 2T(n/4) + n^2$ *Solution:* $\Theta(n^2)$

True or False: Justify your answer briefly (10 points total). **Circle your final answers.**

(a) If an operation takes $O(1)$ worst case time, then it takes $O(1)$ amortized time. *Solution: True*
(b) Greedy algorithms do not always find the correct solutions *Solution: True*
(c) $(\log n)^4$ is $o(\sqrt{n})$ *Solution: True*
(d) $(\log n)^3$ is $\theta(\log n)$ *Solution: False*
(e) A dynamic programming solution is typically faster than a standard recursive solution *Solution: True*

2. **Annihilators**

Consider the following function:

```
int f (int n){
  if (n==0) return 0;
  else if (n==1) return 1;
  else{
    val = 3*f (n-1) -2*f(n-2);
    val += n;
    return val;
  }
}
```

(a) Let $f(n)$ be the value returned by the function f when given input n. Write a recurrence relation for $f(n)$

Solution: $f(n) = 3f(n-1) - 2f(n-2) + n$

(b) Now give the general form for the solution for $f(n)$ using annihilators. *You need not solve for the constants. Solution: First we annihilate the homogeneous part,* $f(n) = 3f(n-1) - 2f(n-2)$. *Let* $F_n = f(n)$, *and* $F = \langle F_n \rangle$. *Then*

$$F = \langle F_n \rangle \quad (1)$$

$$\boldsymbol{L}F = \langle F_{n+1} \rangle \quad (2)$$

$$\boldsymbol{L}^2F = \langle F_{n+2} \rangle \quad (3)$$

Since $\langle F_{n+2} \rangle = \langle 3F_{n+1} - 2F_n \rangle$, *we know that* $\boldsymbol{L}^2F - 3\boldsymbol{L}F + 2F = \langle 0 \rangle$, *and thus* $\boldsymbol{L}^2 - 3\boldsymbol{L} + 2 = (\boldsymbol{L}-1)(\boldsymbol{L}-2)$ *annihilates* F.
Now we must annihilate the nonhomogeneous part $f(n) = n$. *It's not hard to see that* $(\boldsymbol{L}-1)^2$ *annihilates this nonhomogeneous part. So the annihilator for the entire function is* $(\boldsymbol{L}-2)(\boldsymbol{L}-1)^3$. *Looking this up in the lookup table, we see that* $f(n)$ *is of the form:*

$$f(n) = c_0 2^n + c_1 n^2 + c_2 n + c_3 \quad (4)$$

3. **Dynamic Programming**

Consider a new variant of the string alignment problem where the cost of a column is defined to be 1 if the two characters are the same and 2 if the two characters are different. The cost of the alignment is then defined to be the *product* of the costs of all the columns. Assume that the cost of aligning two null strings is 1 (i.e. an alignment must use at least one column). We want to find an alignment with minimal cost. For example in this new variant, the alignment below would have cost 4 and would be an optimal alignment since it *maximizes* the cost.

F	O	O	D
M	O		D

(a) The recurrence relation for the optimal cost of aligning two strings A and B in the original variant of the string alignment problem is given in the formula below. $E(i, j)$ is the value of aligning $A[0..i]$ and $B[0..j]$. Give the modifications needed to get a recurrence relation for the optimal cost in the new variant of the problem. Please cross out the values (or words) to change and write the new values next to the crossed out ones.

$$
\begin{aligned}
E(0, j) &= j \text{ for all } j, \\
E(i, 0) &= i \text{ for all } i \\
E(i, j) &= \min \left\{ \begin{array}{l} E(i-1, j) + 1, \\ \\ E(i, j-1) + 1, \\ \\ E(i-1, j-1) + 0 \text{ if } A[i] = B[j] \\ \text{or} \\ E(i-1, j-1) + 1 \text{ if } A[i] \neq B[j] \\ \\ \end{array} \right\}
\end{aligned}
$$

(b) Now use this new recurrence to find the maximal alignment cost under this new variant for the two strings *ba* and *cb*. Do this by filling in the nine entries in the following dynamic programming table. Also include the arrows used to reconstruct a minimal solution. To the right of the table, give an alignment which achieves the maximal cost.

		b	a
c			
b			

Solution:

$$
\begin{aligned}
E(0,j) &= 2^j \text{ for all } j, \\
E(i,0) &= 2^i \text{ for all } i \\
E(i,j) &= \min \left\{ \begin{array}{l} 2 * E(i-1,j), \\ 2 * E(i,j-1), \\ E(i-1,j-1) \text{ if } A[i] = B[j] \\ \quad \text{or} \\ 2 * E(i-1,j-1) \text{ if } A[i] \neq B[j] \end{array} \right\}
\end{aligned}
$$

		b	a
	$1 \rightarrow 2 \rightarrow 4$		
	$\downarrow \searrow \quad \searrow$		
c	$2 \quad 2 \rightarrow 4$		
	$\downarrow \searrow \quad \searrow$		
b	$4 \quad 2 \rightarrow 4$		

Alignment:

-	b	a
c	b	-

4. **Amortized Analysis**

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

- *Append(x)*: Appends the number x to the front of the list
- *Aggregate()*: Removes all numbers from the list, computes their median, and then appends the median back onto the list.

Assume these operations have the following costs:

- *Append(x)* - cost equals 1
- *Aggregate()* - cost equals the number of ints on the list plus one

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

Solution: Worst case is $\theta(n)$ which happens when we call Append() $n-1$ times and then call Aggregate(). The call to Aggregate costs $\theta(n)$.

(b) *Accounting Method.* Now you will show that the amortized cost of these operations are small using the taxation (accounting) method.

i. First give the amount that you will charge Apppend() and the amount that you will charge Aggregate().

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: Append gets charged 2 dollars. Aggregate gets charged 1 dollar. When we do an append, we use one dollar to pay for the append and store the other dollar with the item. When we do an Aggregate, we use the dollars stored with all the items on the stack plus the dollar we charged for the Aggregate() to pay the total cost. This shows that the amortized cost per operation is $O(1)$

(c) *Potential Method.* You will next use the potential method to get the amortized cost per operation. Let L_i be the list after the i-th operation and let $num(L_i)$ be the number of items on L_i. You will use the following potential function:

$$\phi_i = num(L_i)$$

i. First show that this potential function is valid (i.e. $\phi_0 = 0$ and $\phi_i \geq 0$ for all i)

ii. Next use this potential function to calculate the amortized costs of Append and Aggregate (Recall that $a_i = c_i + \phi_i - \phi_{i-1}$ where a_i is the amortized cost of the i-th operation and c_i is the actual cost)

Solution: The number of items on the list is initally 0 *and is always nonnegative so* ϕ *is valid. First we calculate the amortized cost of Append() at time* i*. Note that* $c_i = 1$ *and* $\phi_i - \phi_{i-1} = 1$*. Thus* $a_i = 2$*. Next we calculate the amortized cost of Aggregate(). Note that* $c_i = num(L_{i-1}) + 1$*. Further note that* $\phi_i - \phi_{i-1} = 1 - num(L_{i-1})$*. Thus* $a_i = 2$*. This implies that the amortized cost of both operations is* $O(1)$*.*

5. **Recurrences**

In this problem, you will use recurrence relations to analyze an interesting "magic" trick. The trick is done as follows.

Choose any two integers and write them one after another. Now form a third number by adding the first two numbers you've written down. Form a fourth number by adding the second and third; a fifth number by adding the third and fourth, etc. until you have a sequence of twenty numbers. Now divide the twentieth number by the nineteenth. The value you get will be a good approximation to the "golden ratio", ϕ (recall that $\phi = (1 + \sqrt{5})/2$, which is approximately 1.6180339...).

Can you explain why this trick works? Hint: Write down a recurrence relation $T(n)$ for the n-th number in the sequence. Now get the *general* solution to this recurrence relation using annihilators. Next, figure out what is a good approximation to this solution for large n. Finally compute a good approximation to $T(n)/T(n-1)$ for large n.

Solution: $T(n) = T(n-1)+T(n-2)$. *The annihilator of this sequence is* $(L^2 - L - 1)$. *When we factor this, we get* $L - \phi$ *and* $L - \bar{\phi}$, *where* $\phi = (1+\sqrt{5})/2$ *and* $\bar{\phi} = (1-\sqrt{5})/2$. *Using the lookup table, we can see that the general solution to this recurrence is* $T(n) = C_0\phi^n + C_1\bar{\phi}^n$. *Note that* $|\bar{\phi}| < 1$, *so as* n *grows large,* $T(n) = C_0\phi^n$ *is a good approximation to the recurrence. Thus, a good approximation for* $T(n)/T(n-1)$ *for large* n *is* $C_0\phi^n/C_0\phi^{n-1} = \phi$. *The cool thing is that this is true no matter what the initial two numbers in the sequence are. Test it out!*