University of New Mexico
Department of Computer Science

First Midterm Examination

CS 362 Data Structures and Algorithms
Fall, 2004

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) $8^{\log n}$ *Solution:* $\Theta(n^3)$
(b) Amount of space required by the dynamic programming algorithm for finding the optimal parenthesization of a sequence of n matrices *Solution:* $\Theta(n^2)$
(c) Worst case cost of n calls to Pop, Push and Multipop on a stack *Solution:* $\Theta(n)$
(d) Solution to the recurrence $T(n) = 4T(n/2) + \log n$ *Solution:* $\Theta(n^2)$
(e) Solution to the recurrent $T(n) = 2T(n/4) + n$ *Solution:* $\Theta(n)$

True or False: Justify your answer briefly (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*
(b) Greedy algorithms do not always find the correct solutions *Solution: True*
(c) $\log n$ is $o(\sqrt{n})$ *Solution: True*
(d) $\log n^2$ is $\Omega(3 \log n)$ *Solution: True*
(e) A dynamic programming algorithm is typically faster than a greedy algorithm *Solution: False*

2. **Substitution Method**
   Consider the following recurrence: $T(1) = 1$, $T(2) = 2$ and

$$T(n) = \frac{T(\lfloor n/2 \rfloor) * T(\lfloor n/2 \rfloor)}{n}$$

.

Show that $T(n) \leq n$ by induction. Include the following in your proof: 1)the base case(s) 2)the inductive hypothesis and 3)the inductive step.

Solution: Base Case: $T(1) = 1$ and $T(2) = 2$ which are both in fact no more than n.
Inductive Hypothesis: For all $j < n$, $T(j) \leq j$
Inductive Step: We must show that $T(n) \leq n$, assuming the inductive hypothesis.

$$\begin{aligned} T(n) &= \frac{T(\lfloor n/2 \rfloor) * T(\lfloor n/2 \rfloor)}{n} && (1) \\ &\leq \frac{(\lfloor n/2 \rfloor) * (\lfloor n/2 \rfloor)}{n} && (2) \\ &\leq \frac{(n/2) * (n/2)}{n} && (3) \\ &\leq n/4 && (4) \end{aligned}$$

where the inductive hypothesis allows us to make the replacements in the second step.

3. **Annihilators**

Consider the following function:

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

(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,* $f(n) = 2f(n-1) - f(n-2)$. *Let* $F_n = f(n)$, *and* $F = \langle F_n \rangle$. *Then*

$$F = \langle F_n \rangle \quad (5)$$

$$\boldsymbol{L}F = \langle F_{n+1} \rangle \quad (6)$$

$$\boldsymbol{L}^2 F = \langle F_{n+2} \rangle \quad (7)$$

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

$$f(n) = c_1 n^2 + c_2 n + c_3 \quad (8)$$

4. **Dynamic Programming**

Consider a new variant of the string alignment problem where *1) the cost of an alignment is defined to be the number of columns which contain two characters which are the same and 2) we want to find an alignment of* maximal *cost.* For example in this new variant, the alignment below would have cose 2 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. *To do this, you will need to make seven small changes to the formula below.* Please cross out the values (or words) to change and write the new values next to the crossed out ones.

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

(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:

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

$$\begin{tikzcd}
 & & b & a \\
 & 0 \arrow[r] \arrow[d] & 0 \arrow[r] \arrow[d] & 0 \arrow[d] \\
c & 0 \arrow[r] \arrow[d] \arrow[dr] & 0 \arrow[r] & 0 \\
b & 0 & 1 \arrow[r] & 1
\end{tikzcd}$$

Alignment:

-	b	a
c	b	-

5. **Amortized Analysis**

Consider a stack of ints that has the following operations defined on it:

- *Push(x)*: Pushes the int x onto the stack
- *AddUp()*: Removes all ints from the stack, adds them up and then pushes the sum back on the stack.

Assume these operations have the following costs:

- *Push(x)* - cost equals 1
- *AddUp()* - cost equals the number of ints on the stack plus one

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

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

(b) *Accounting Method.* 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 Push() and the amount that you will charge AddUp().

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: Push gets charged 2 dollars. AddUp gets charged 1 dollar. When we do a push, we use one dollar to pay for the push and store the other dollar with the item. When we do an AddUp, we use the dollars stored with all the items on the stack plus the dollar we charged for the AddUp() to pay the toal 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 S_i be the stack after the i-th operation and let $num(S_i)$ be the number of ints on S_i. You will use the following potential function:

$$\phi_i = num(S_i)$$

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 Push and AddUp (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 stack is initally 0 *and is always nonnegative so* ϕ *is valid. First we calculate the amortized cost of Push() 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 AddUp(). Note that* $c_i = num(S_{i-1}) + 1$*. Further note that* $\phi_i - \phi_{i-1} = 1 - num(S_{i-1})$*. Thus* $a_i = 2$*. This implies that the amortized cost of both operations is* $O(1)$*.*