University of New Mexico
Department of Computer Science

Midterm Examination

CS 561 Data Structures and Algorithms
Fall, 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 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. **Asymptotic Analysis and Recurrence Relations** Consider the functions 2^n and 4^n. Is $2^n = \Theta(4^n)$ or is $2^n = o(4^n)$? Prove your answer using definitions of asymptotic notation given in the book and in class and solve for the values required to show the definitions hold. *Solution:* $2^n = o(4^n)$. *To show this, we need to show that for any positive constant* $c > 0$, *there exists a constant* n_0 *such that* $0 \leq f(n) \leq cg(n)$ *for all* $n \geq n_0$. *This is equivalent to saying that*

$$\begin{aligned} 2^n &\leq c4^n \\ (1/2)^n &\leq c \\ 2^n &\geq 1/c \\ n &\geq \log(1/c) \end{aligned}$$

Thus we need to choose $n_0 = \log(1/c)$

Recurrence Relations: Consider the Recurrence $T(n) = T(n-1) + 6T(n-2) + n$. Write down the general form of the solution for this recurrence (i.e. don't solve for the constants). *Solution: The homogeneous part is annihilated by* $\boldsymbol{L}^2 - \boldsymbol{L} - 6$ *which factors into* $(\boldsymbol{L}-3)(\boldsymbol{L}+2)$. *The nonhomogeneous part is annihilated by* $(\boldsymbol{L}-1)^2$. *Looking this up in the lookup table gives us that the general solution is of the form* $T(n) = c_1 3^n + c_2(-2)^n + c_3 n + c_4$.

2. **Sorting**

Consider the following sorting program:

```
SillySort(A,i,j){
  if i+1 > j
       then return;
  Let m be the index of the minimum element in A[i..j];
  exchange A[i] and A[m];
  SillySort(A,i+1,j);
}
```

Assume it takes n operations to find the value m in the third step of the algorithm for a list of size n. Write and solve a recurrence relation for the run time of SillySort when called on the array A[1..n]. *Solution:* $T(n) = T(n-1) + (n+1)$. $(\boldsymbol{L}-1)$ *annihilates the homogeneous part and* $(\boldsymbol{L}-1)^2$ *annihilates the non-homogeneous part. So the complete annihilator is* $(\boldsymbol{L}-1)^3$. *The general form of the solution is thus* $c_1n^2 + c_2n + c_3 = O(n^2)$

Prove *succinctly* that SillySort correctly sorts a list of n elements by induction on n. Don't forget to include the base case, inductive hypothesis and inductive step.

Solution: We will do induction on $n = j - i + 1$ *to show that SillySort sorts the array A[i..j]. B.C. if* $n = 1$*, there is only one element in the list and thus the list is already sorted. Hence SillySort is correct in this case by simply returning without doing anything. I.H. SillySort correctly sorts lists of size less than* n*. I.S. When faced with a list of size* n*, SillySort first moves the minimum element to the front of the list. It then calls it self recursively on the remainder of the list. We can assume by the I.H., that the recursive call correctly sorts the remaining elements. Thus the entire array A[i..j] is in sorted order when the algorithm exits.*

3. **Probability**

Hat check: Each of n people give their hats to a drunken hat check person at a restaurant. The hat check person gives back the hats in a random order. What is the expected number of customers that get back their own hat? *Solution: For each person i, let X_i be an indicator random variable that is 1 iff person i gets back their own hat. Let $X = \sum_{i=1}^{n} X_i$ be a random variable that equals the number of hats returned to the right person. Then $E(X) = E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i)$, where the last step follows by linearity of expectation. REMEMBER: Linearity of Expectation holds even if the random variables aren't independent. Finally note that $E(X_i) = 1/n$ since this is the probability that person i's hat is returned correctly. Thus $E(X) = 1$.*

Minimum Value: Assume that a set of n unique values are inserted in random order into a binary tree. What is the expected number of times that minimum value in the tree changes? Hint: Let X_i be an indicator random variable that is 1 iff the i-th element inserted is smaller than the first $i-1$ elements inserted. *Solution: Let X be a random variable giving the number of times that the minimum value in the tree changes. Note that $X = \sum_{i=1}^{n} X_i$. Thus $E(X) = E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i)$. Finally note that $E(X_i) = 1/i$ since this is the probability that the i-th element is the smallest element among the first i elements (since we're assuming the values are inserted in random order). Thus $E(X) = \sum_{i=1}^{n} 1/i$. This last summation is $\ln n + O(1)$ as we showed in class.*

4. **Sums**

Imagine you are given a list of n integers and a single integer x. Your goal is to determine if there is a pair of numbers in the list that sum to the value x. Note that a naive algorithm for this problem is to just consider all pairs of numbers in the list and see if any of them sum to x. This naive algorithm takes $O(n^2)$ time.

This problem consists of two parts. First, devise an algorithm to solve this problem with good worst case run time. Second devise an algorithm for this problem with good *expected* run time. Note that the word "good" is deliberately ambiguous, part of this problem is to try to get as low as possible.

Hint: Make use of data structures and algorithms discussed in class.

Solution: To get the best worst case bound, first traverse each value in the list and insert it into a balanced BST (such as a red black tree). Next traverse each value y in the list and do a search for the value $x - y$ in the BST. If this value is ever found, then return the pair y, $x - y$, otherwise return that there is no pair. This algorithm takes $O(n \log n)$ time to insert the values in the list and $O(n \log n)$ time to do all the searches. To get the best expected time, perform the same algorithm except insert the values $x - y$ into a hash table with n buckets. Assuming a good hash function, the expected time for all the insertions and all the deletions will be $O(n)$.

5. **Matrix Maximum**

In this problem, you are given a n by n matrix, M of integers. You will create a data structure that will provide the following operations: IncrementRow, IncrementColumn and GetMaximum. IncrementRow takes as input a row number and increases every entry in that row by 1. IncrementColumn takes as input a column number and increases every entry in that column by 1. GetMaximum returns the maximum element in the matrix (breaking ties arbitrarily).

The data structure is initialized with the start matrix M and you can take any amount of time for processing at initialization. After this, every call to GetMaximum must take $O(1)$ time and every call to IncrementRow and IncrementColumn must take $O(n \log n)$ time.

Describe how you would create this data structure and sketch your analysis of its efficiency. Hint: Make use of data structures and algorithms discussed in class.

Solution: On Initialization, create a heap of the n^2 items in the matrix. Keep a pointer from each item in the matrix to its corresponding node in the heap (these pointers will be used for HEAP-INCREASE-KEY Operations). On a call to GetMaximum, just return the element at the root of the heap (which will be the largest element). On a call to IncrementRow(i), traverse each element in row i and call HEAP-INCREASE-KEY on each of these elements in the heap to increase the key by exactly 1. HEAP-INCREASE-KEY takes $O(\log n^2) = O(\log n)$ time per call and it's called n times to give a total cost of $O(n \log n)$. IncrementColumn is similar. Note that the easiest way to keep pointers between the elements of the matrix and the elements of the heap is for the heap to not be implemented as an array but rather with nodes and child and parent pointers. This does not significantly change the heap data structure.