CS 61B – Summer 2005 – (Porter)

Midterm 2 – July 21, 2005 – SOLUTIONS

Do not open until told to begin

This exam is CLOSED BOOK, but you may use 1 letter-sized page of notes that you have created.

Problem 0: (1 point) Please fill out this information, and when told to begin, please put your name on each page of the exam.

Your name:	Solution
Your cs61b login:	
Lab time:	
Lab TA's name:	
Name of person to your left:	
Name of person to your right:	

Do not write below here:

Problem	**Score**	**Total possible**
0	1	1
1 (10 minutes)	15	15
2 (10 minutes)	15	15
3 (15 minutes)	20	20
4 (35 minutes)	34	34
Total:	85	**85**

Do not open until told to begin

Good Luck!

Problem 1 (15 pts) – Algorithm Analysis – (10 minutes)

Given T(n) = 22n – 37, show that T(n) is O(N^2). Make use of the definition of Big-Oh.

We need to find **c** and **N_0** such that:

22n – 37 <= c N^2. Let's start by setting **c** = 1. Now by moving some terms around, we get:

22n – N^2 <= 37
=
n(22 – N) <= 37.

Now for N > 22, we have that (22 – N) is negative. For N > 22, N is positive. A positive number times a negative number is negative, and all negative numbers are <= 37. Thus we show that T(n) is O(N^2) by setting **c** = 1 and **N_0** = 23.

Problem 2 (15 pts) – Probing – (10 minutes)

Show the result of each insertion sequence using the three insertion methods below. For chaining, use the space to the right of the array for the linked lists.

hash (59, 11) = 4
hash (82, 11) = 5
hash (26, 11) = 4
hash (5, 11) = 5
hash (92, 11) = 4

Linear Probing

0	
1	
2	
3	
4	59
5	82
6	26
7	5
8	92
9	
10	

Quadratic Probing

0	
1	
2	92
3	
4	59
5	82
6	5
7	
8	26
9	
10	

Chaining

0	
1	
2	
3	
4	59, 26, 92
5	82, 5
6	
7	
8	
9	
10	

Problem 3 (20 pts) – Sorting (smaller to larger) – (15 minutes)

3.1) (7 pts) Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Suppose we partition this array using quicksort's partition function (use the Quicksort code from Weiss at the end of the exam) and using 5 for the pivot. Draw the resulting array after the partition finishes.

1) swap 5 with array[high]:	4 3 8 9 1 7 0 2 6 5
2) i stops at 8, j stops at 2; swap:	4 3 2 9 1 7 0 8 6 5
3) i stops at 9, j stops at 0; swap:	4 3 2 0 1 7 9 8 6 5
4) i stops at 7, j < i; swap w/ pivot:	4 3 2 0 1 5 9 8 6 7

* step 4 is optional

3.2) (6 pts) Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Draw this array after the TWO recursive calls of merge sort are completed, and before the final merge step has occurred.

1st recursive call: mergesort of left half:	**1 3 5 8 9** 7 0 2 6 4
2nd recursive call: mergesort of right half:	1 3 5 8 9 **0 2 4 6 7**
we don't do the final merge step, thus:	
	1 3 5 8 9 0 2 4 6 7

3.3) (7 pts) Indicate the worst case and average case performance using Big-Oh notation:

Algorithm	Worst Case	Average Case
Binary Search of a sorted array	log N	log N
Insertion sort	N^2	N^2
Merge sort	N log N	N log N
Quick sort **without** "median of three" pivot selection	N^2	N log N
Quick sort **with** "median of three" pivot selection	N^2	N log N
Shell sort	N^2	N^(3/2) or N^(7/6) or N^(5/4)

Problem 4 (34 pts) – Hashtables vs Sorted Linked Lists – (35 minutes)

Consider a composite data structure to represent an inventory of items, declared as follows. It contains as private data members:

```
class Item {
    public String name;
    public int cost;
};
public class Inventory {
    ...
    private LinkedList sortedByCost;
    private ArrayList hashTable;
        ...
};
```

(The LinkedList class is similar to what you implemented in the homework). Items are identified uniquely by name; i.e. there shouldn't be two items with the same name. Elements of the sortedByCost list appear in decreasing order by cost, except that if there are more than one item with the same cost, those items appear in the list alphabetically by name. The corresponding items should also all be represented in the hash table, which is accessed by hashing a name, then searching the corresponding chain for a pointer to the item with that name.

To aid debugging, one might devise a function that checks that the inventory structure is internally consistent. Such a function might check that the lists are correctly constructed, i.e. that there aren't any circular pointers and that the size is equal to the number of items in each list. It might also check that sortedByCost is correctly sorted. For this problem, you are to describe how to implement two additional consistency checks:

Duplicate check:

no two items represented in sortedByCost should have the same name; no two items represented in hashTable should have the same name.

Correspondence between components:

Every item represented in sortedByCost should also be represented in hashTable, and vice-versa.

Part a (20 points)
Describe an implementation for performing the operations labeled as "Duplicate check" and "Correspondence between components" on the previous page. Each consistency check should be performed as efficiently as possible on the average. You may use additional structures and should not change the existing contents of the inventory. Your description should specify what additional data structures you use, what they contain and, if your data structures include arrays, how big the arrays are. Your description should be in terms suitable for another CS 61B student to understand immediately how it would be translated into code.

1. Hash the contents of the individual chains by name, using another hash function and a table that's of size dependent on M, the size of the chain: If two items with the same name are seen, print an error message

2. Sort the chains by cost (primary key) and name (secondary key) using a fast sort (Quicksort or merge sort).

3. Compare the resulting list with sortedByCost.

or,

Duplicate check:
1) For each chain in your hashtable (M_i), copy it to an array of length size(M_i).
2) Sort M_i using mergesort
3) Check adjacent elements for elements with the same name
you can check sortedByCost in a similar way

Name: ____

Part b (14 points)
Give an estimate of the running time needed for each step involved in your answer to part a. Be specific about the quantities your estimates involve.

1. For each chain, order M for a reasonable hash function. Total this over all chains => order N.
2. Order N log N, where N is the number of items.
3. Order N.

(alternative approach):

building the array: O(size(M_i))
sorting: O(M_i * size(M_i))
checking neighbors: O(size(M_i))

If we sum M_i over all i, we get N. Thus, O(N log N).