Name: _____

CS343: Analysis of Algorithms, Sp 04
Exam 2

Score:	1.	(max 15)	4.	(max 25)
	2.	(max 17)	5.	(max 31)
	3.	(max 12)	6.	(max 0)
	Total:		(max 100)	

This exam is closed book. Calculators are not allowed.

1. (5 pts each, 15 pts total) A sort algorithm is called *natural* if it does not move any elements when the algorithm is applied to an already sorted array. For each of the sorts below, say if the sort is natural and give a brief explanation for your answer.

   (a) bubble sort

   (b) shellsort

   (c) quicksort

2. (17 pts total) Assume that you have a hash table of size 10 and you insert the following numbers:

   23 8 10 18 9 5 13 28

   Show the resulting table and determine the total number of collisions that occur when the following collision detection strategies are used.

   (a) (6 pts) linear probing with hash function $h(x) = x$ % tablesize

   (b) (6 pts) quadratic probing with $h_i(x) = (h(x) + i * i)$ % tablesize

   (c) (5 pts) Suppose you want to place a list of people's ages (in years) into a hash table using the strategy given in part a). There are 50 numbers to hash. You want to make sure that there is plenty of room in the table, so you decide to use a table of size 100, i.e twice the number of items to hash. What is wrong with this scheme? How could you fix the problem?

3. (12 pts total) Heaps:

   (a) (6 pts) Show the resulting heap (in tree form) that is obtained when the algorithm *heapify* is applied to the array given below. You need not show every step but show as many as are required for me to follow your process:

   3 7 6 24 10 11 4

   (b) (2 pts) What is the complexity of creating a heap of size n?

   (c) (2 pts) What is the complexity of inserting a new item to an existing heap?

   (d) (2 pts) What is the complexity of sorting the items in the heap using heap sort?

4. (25 pts total) Define a *dictionary* to be any data structure that allows the following operations (Assume that no deletions occur):

   - insert(i): adds the integer i into the dictionary.
   - find(i): checks to see if the integer i is already in the dictionary.
   - largest(): returns the largest integer in the dictionary. Note, the value is returned but the item itself is not removed from the data structure.

   Each of the following columns below gives the time complexities associated with a particular data structure (DS).

Operation	DS 1	DS 2	DS 3	DS 4
insert	$O(\lg n)$	$O(n)$	$O(1)$	$O(\lg n)$
find	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(n)$
largest	$O(\lg n)$	$O(1)$	$O(n)$	$O(1)$

   Give the name of a well known data structure that achieves those complexities on a dictionary of up to n elements and briefly describe how to implement each of the 3 operations.

   For example, DS 3 could correspond to an unsorted list. *Insert* consists of adding an item to the end of the list which is $O(1)$. However, *find* and *largest* would require searching through the entire list which is $O(n)$.

   (a) (5 pts) DS 1

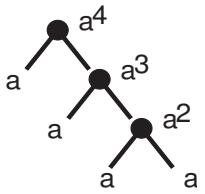
   (b) (5 pts) DS 2

   (c) (5 pts) DS 4

(d) (10 pts) Suppose instead of largest(), you wanted to determine the largest(k), i.e. the k^{th} largest. For example, largest(1) is the same as largest(), largest(2) is the second largest, etc. For any two of the three data structures (1,2, or 4), explain how you would calculate the largest(k) and what is the complexity?

5. (31 pts total) Throughout this problem, assume that *n is always a power of 2.*

   Suppose you want to calculate a^n, where n is an integer and a is some real number. One way to do this (called Method 1) is to first compute $a * a$ (1 multiply), then to take this result and compute $(a * (a * a))$ (another multiply) and so forth. This could be done in a simple loop. When you are done you will have done $n - 1$ total multiples. The calculations for $n = 4$ are shown in the binary tree shown below. Each inner node represents a multiplication (in this case, 3).



   Being a clever computer scientist, you believe that you can reduce the number of multiples by computing the product recursively based on the formula $a^n = a^{n/2}a^{n/2}$. We'll call this Method 2.

   (a) (5 pts) Write a brute force recursive method in Java using Method 2, i.e. based on the formula $a^n = a^{n/2}a^{n/2}$.

```
public double powerM2(double a, int n) { // n is power of 2




}
```

   (b) (4 pts) For $n = 8$, draw the binary tree showing all of the recursive calls made by your Method 2 algorithm. How many multiplies are required?

   (c) (5 pts) Using what you know about binary trees, give the exact number of multiples that are needed in Method 2 for general n. Is your recursive algorithm better or worse than Method 1?

(d) (5 pts) You notice that Method 2 seems to suffer from overlapping subproblems so you decide to apply dynamic programming (Method 3). Show what the dynamic programming table would look for $n = 8$. Describe the table for general n.

(e) (8 pts) Write a method in Java (or pseudo code) that uses dynamic programming (Method 3).

```
public double powerM3(double a, int n) { // n is power of 2
     // determine size and create table




     // fill in table




     // return the final result obtained from table


}
```

(f) (4 pts) For general n, how many multiplies does your dynamic program (Method 3) use to compute a^n?

6. (0 pts) What sort algorithm would you use to sort out your ideas?