CSC 120 Algorithms and Data Structures **Spring Semester, 2002**

Review Solutions Final

1. True or False

$20n^3 + 10n\log n + 5$	is $O(n \log n)$ FALSE
2^{100}	is $\Theta(1)$ TRUE
$\log(n^x)$	is $O(\log n)$ *where* $x > 0$ *is const* TRUE
$500\log^5 n + n + 10$	is $O(n)$ TRUE
$0.5\log n$	is $\Theta(n)$ FALSE

2. The functions are in increasing complexity order (i.e., lowest to highest):

$$\log\log n,\ 6\log n,\ \ n\sqrt{n},\ n^2,\ 2^{\log n}.$$

3. Express as a function of the input size n the worst-case running time $T(n)$ of the following algorithm

```
int Me(int n, int A[], int k)
{
    int tot = 0;                //   1
    for (int i=n; i>=0; i--) { //3*(n+1)
       if (i==k)  {              // 3 , counting the test condition
          cout << i << endl;     // 1
          tot++;                 // 1
       }
    }
    return tot;                 // 1
 }
```

The worst case run time and complexity are:

$$T(n) = 3*(n+1) + 2 = \Theta(n)$$

4. Give the worst-case and the best-case recurrences expressing the running time of the following algorithm manipulating a BST rooted at p.

```
int You( int *p, int a, int b)
{
    int tot = 0;                //  1
    if (p==0) return 0;         //  base case T(0)=2
```

```
        if ( *p > a && *p < b) {   // 3 , counting the test condition
            cout << *p << endl;     // 1
            tot++;                  // 1
        }
        tot += You( Left(p), a, b);  // T(left subtree)
        tot += You( Right(p), a, b); // T(right subtree)
        return tot;                  // 1
    }
```

What are the best- and the worst-case time complexities of this algorithm?

The running time of the recursive procedure on a tree with n nodes can be expressed as:

$$\begin{aligned} T(0) &= 3 \\ T(n) &= T(left\ subtree) + T(right\ subtree) + 5 \end{aligned}$$

The analysis is similar to that for the tree traversal algorithms.

The number of statements executed at each call (excluding the recursive calls themselves) is constant (5 to be precise). The procedure will be called for each node (for a subtree rooted in each node) exactly once. Since there are n nodes, and for each one the work done is constant, the total runt time is $5n$. Independent of the exact tree structure, all nodes are visited, i.e. the best-and worst- case complexities are the same $\Theta(n)$.

5. Using the master method estimate the complexity of the recursive algorithms which run times are expressed by the recurrences below.

   We use the notation introduced in class.

   (a) $T(n) = 21T(\frac{2n}{3}) + \Theta(n^2 \log^3 n)$

   $$a = 21, b = 3/2, k = 2, p = 3$$

   $$21 = a > b^k = (3/2)^2, \text{ thus Case 1, } \Theta(n^{\log_{3/2} 21})$$

   (b) $T(n) = 9T(\frac{n}{3}) + \Theta(n^2)$

   $$a = 9, b = 3, k = 2, p = 0$$

   $$9a = b^k = 3^2, \text{ thus Case 2, } \Theta(n^2 \log n)$$

   (c) $T(n) = 10T(\frac{n}{2}) + \Theta(n^4 \log n)$

   $$a = 10, b = 2, k = 4, p = 1$$

   $$10 = a < b^k = 2^4, \text{ thus Case 3, } \Theta(n^4 \log n)$$

6. Fill in the following table with the asymptotic time complexities (use expected time complexities when appropriate). Be sure to indicate which time complexities are worst-case and which are expected-case. Let n be the number of elements in the ADT. Let m be the number of slots in the hash table.

	BST	heap	hashing with chaining	doubly linked sorted list
SEARCH (for key k)	$\Theta(h)$ worst case, $h = n$	$\Theta(n)$ worst	$\Theta(1 + n/m)$ average	$\Theta(n)$ worst
DELETE(k) (when given the location of k)	$\Theta(h)$ worst, $h = n$	$\Theta(\log n)$ worst, del root	$\Theta(1)$ worst	$\Theta(1)$ worst
MAXIMUM	$\Theta(h)$ worst, $h = n$	$\Theta(n)$ worst, in min heap	$\Theta(n)$ worst	$\Theta(n)$ worst

7. In a directed graph, the *in-degree* of a vertex is the number of edges entering it. Let n be the number of vertices, and m the number of edges in the graph, and let $d_I(v)$ be the in-degree of vertex v.

   Given an **adjacency matrix** representation of a directed graph, and a specified vertex v, how would you best compute the in-degree of v? Write your algorithm in pseudo code, analyze the time complexity of your algorithm.

   Here you need just to count the number of 1's in the column of the adjacency matrix corresponding to v. Since there are n entries in the column, clearly the time complexity is $\Theta(n)$.

```
// M is the adjacency matrix, of size nxn, M[u][v]=1 if (u,v) is an edge
InDegree(M, v)
   int deg=0;                  // 1
   for (int u=0; u<n; u++)     // n
      deg += M[u][v];              // 1
   return deg;                 // 1
```

   Let $G = (V, E)$ be graph with n vertices and m edges.

$$T(n, m) = n + 3 = \Theta(n)$$

8. An undirected graph is *bipartite* if its vertex set V can be partitioned into two sunsets S and T (i.e. $V = S \cup T, S \cap T = \emptyset$) so that every edge in E connects a vertex in S to a vertex in T.

   Describe a brute force algorithm for checking if an undirected graph is bipartite. What's the time complexity of your algorithm.

ANSWER:

The brute force algorithm is an exhaustive search of all two-set partitions of V, i.e., for each partition of V into two subsets S and T, check if the partition satusfies the "bipartite coindition". If you find that a partition satisfies the condidtion, return TRUE, otherwise return FALSE.

Given a set on n vertices, there are n ways we can partition V into two sets one of which has exactly one element, there are $C_2^n = \frac{n(n-1)}{2!}$ ways to partition V into 2 subsets one of which has exactly 2 elements, ..., there are $C_n^k = \frac{n(n-1)...(n-k+1)}{k!}$ ways of partitioning V into two subsets such that one has exactly k elements. Here C_k^n is the binomial coefficient "choose k of n". Thus, the total number of ways we can partition V into two subsets is the sum of all binomial coefficients for $k = 1, \ldots, n/2$. Since the sum of the binomial coefficients, for $k = 0, \ldots, n$ is 2^n, and since they are symmetric, i.e. $C_k^n = C_{n-k}^n$, the sum of the first half of the coefficients (minus 1), will be on the order of $\Theta(2^{n-1})$, and that will be the number of the partitions that have to be checked in worst case. For each partition, in worst case we have to check all edges to see if they connect vertices in the same subset of the partition or not. Thus the worst case complexity for the whole algorithms will be $\Theta(2^{n-1}m)$ – exponential!!!

9. Show the hash table obtained when inserting the keys 26, 18, 19, 32, 4, and 65 into a hash table (in the given order) with collisions resolved by chaining. Let the table have 7 slots and let the hash function h be such that $h(26) = 3$, $h(18) = 6$, $h(19) = 0$, $h(32) = 0$, $h(4) = 3$, $h(65) = 3$.)

   You need just show the final result. Be careful to correctly show the order that would result within the chains/lists.

   ANSWER: X marks the null pointer

```
table
slots

0: --> 32 --> 19
2:X
3: --> 65 ---> 4 --> 3
4:X
5:X
6: --> 18
```

10. Give pseudo-code for a procedure DECREASE-KEY(i, k) that modifies the given binary heap A by decreasing the value of $A[i]$ to k. (If $A[i] \leq k$ then no change should be made.) You can use (without describing) any of the standard binary heap procedures that we've studied. You should give the most efficient implementation you can.

    Now analyze the asymptotic time complexity of your algorithm. Be sure to explain!

    ANSWER:

Note that if $A[i] > k$, we set $A[i] = k$, and there is a chance that k is smaller than the key values of the ancestors of i, and the heap PORD property might be violated. As far as descendents of i are concerned, things can't go wrong. Thus we have to restore the heap order by proragating the key k up the path to the root till it falls in place (similar to insert procedure).

```
Decrease-Key(i,k) {
   if (i >= 1 && A[i] > k) {
      while (i > 1 && k < Key(Parent(i)) )
         A[i] = Key(Parent(i)) // shift key of parent down, to child
         i = Parent(i)
      A[i] = k
   }
}
```

The complexity in worst case is the distance from i to the root, same as in insert, i.e., $O(\log n)$ in worst-case.

11. You are to implement a caller-id system which supports the operations given in b)-d) below. Pick a data structure that is best suited for the problem (i.e. the required operation will run as efficiently as possible). You should very clearly describe how the data structure is to be applied (e.g. what is used as the key, what the associated data is,...). Also be sure to give the complexity for each of the operations.

   (a) Describe your data structure choice first.

   Answer:

   Operations needed are the one supported by ADT dictionary, and search in particular is of interest. We will use a hash table with collision resolution by chaining, which will guarantee efficient searches.

   Each individual record will have the following fields: the key will be a phone number, the data are name and address.

   (b) Give the time complexity analysis for inserting into the system of new item that consists of a phone number, address, and name.

   Answer:

   Insertion is insertion into a hash table, we insert in the front of the corresponding list to which the value hashes. Complexity is $\Theta(1)$.

   (c) Give the time complexity analysis for searching: given a phone number, return the address and name.

   Answer:

   In worst-worst case, with a bad hash function, when all keys are hashed to the same slot, is $\Theta(n)$, but we are not going to use such a hash function. A good hash function distributes the keys uniformly, and a the list have close to average length (n/m), where m is the number of slots, so in worst and average case the search takes $\Theta(1 + n/m)$ time.

(d) Give the time complexity analysis for deletion: given a phone number, remove the corresponding item from the system.

Answer:

Given a phone number first we have to search for it, and then remove it. Searching is $\Theta(1+n/m)$ on average, and for good hash function in worst-case too. The actual removing, since we keep the linked lists unsorted takes a constant time, thus the time for delete *given* the phone number is $\Theta(1 + n/m)$ on average, and in worst case with a good has function.

12. Write an algorithm to print all keys in BST between two given keys, k_1 and k_2.

```
//Print keys between k1 and k2 (included) for a BST rooted in p
PrintKeys(int *p, int k1, int k2) {
   if (p==0) return  // base case

   if (Key(p) > k2 )  // search only left subtree
      PrintKeys(Left(p), k1, k2)
      return

   if (Key(p) < k1) // search only right subtree
      PrintKeys(Right(p), k1, k2)
      return

   // otherwise, print key and continue search
   print Key(p)
   PrintKeys(Left(p),k1,k2)
   PrintKeys(Right(p),k1,k2)
   return
}
```