Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Practice Sheet #12

Topic: Sorting Techniques

Date: 28-03-2017

1. (a) Assuming that in a recursive quick sort, each recursive call partitions the input array into two roughly equal halves, give the recurrence relation depicting the time complexity; hence, obtain a close form of the time complexity.

   (b) Write a *recursive* C function **`findMed`** which returns the median of a one-dimensional array of integers, that is, the element which is larger than half of the elements and smaller than half of the elements. Suppose $a_1 <= a_2 <= \cdot \cdot \cdot <= a_n$ is the sorted version of the input array A. If $n = 2k + 1$, then the element a_{k+1} is the median of A. On the other hand, if $n = 2k$, we take a_k as the median of A. Your function should use a partitioning technique as in quick sort. Use the following function prototype: **`int findMed ( int A[], int startidx, int endidx, const int medidx );`**
   Here the second and the third parameters are the start and end indices of the current partition; the fourth parameter is the index of the median in the sorted version of A and is kept constant across the calls.

2. The absolute distance between two integers *x1* and *x2* is given by | *x2 – x1* |. Write a function which sorts an array *x[ ]* of n integers in ascending order of their absolute distances with a given number *z*. For example, given *x[ ] = {9, 1, 12, 4, 2}* and *z = 6*, the sorted array will be *x[ ] = {4, 9, 2, 1, 12}*. Note that 4 is closest to 6, and 12 is farthest from 6, in terms of absolute distances. The function will have the following prototype: void dist_sort(int x[ ], int n, int z) ;

3. You are given an $m \times n$ array A of integers, each row of which is a sorted list of size n. Your task is to merge the m sorted lists and store the merged list in a one-dimensional array B. It is given that each row does not contain repetition(s) of integers, that is, the n integers in each sorted list are distinct from one another. However, integers may be repeated in different rows. During the merging step, you must remove all these repetitions.
   Complete the following function to achieve this task. The function uses an array of m indices, where the i-th index is used for reading from the i-th row ($0 <= i <= m - 1$). The function starts by initializing each of these read indices to point to the beginning of the corresponding row. Subsequently, inside a loop, it computes the minimum of the m elements pointed to by these indices. The minimum is then written to the output array. Note, however, that during the computation of this minimum, we do not need to consider those rows all of whose elements have already been written to the output array B. Finally, for all rows containing this minimum element at the current read index positions, the index values are incremented. The function is supposed to return the total number of elements written to the output array B.

```
#define INFINITY 123456789
int merge ( int B[], int A[][MAX], int m, int n )
/* A is the input two-dimensional array of size m× n.
B is the output array whose size is to be returned. */
{
int index[MAX], i, k, min; /* Do not use other variables */
for (i=0; i<_______________ ; ++i) index[i] =________________ ;
k = ________________; /* k is for writing to B[] */
while (1) { /* Let us decide to return inside this loop */
min = INFINITY; /* Initialize min to a suitably large value */
/* Write a loop for computing the minimum */
for (________________ ) {
if_____________________
min =___________________________ ;
}
/* If all input arrays are fully processed, return the size of B */
_______________________________________________________
/* Otherwise, write the computed minimum to B */
_______________________________________
/* Advance all relevant indices */
_______________________________________________
for (________________________ ) {
if_____________________________________
__________________________________________;
}
}
}
```

4. Consider two integer arrays A and B of the same size n > 2. We define a relation A _ B if there exists an index j in the range 0 6 j 6 n□1 such that A[j] < B[j], and for all i in the range 0 6 i < j, we have A[i] = B[i]. Note that for any two arrays A and B of size n, exactly one of the three cases hold: (a) A _ B, (b) B _ A, or (c) A = B (that is, A and B are identical for all elements). As examples, take n = 3, and observe that (1;2;3) <(1;2;4) < (1;4;2) < (3;0;0).
You are given an m_n two-dimensional array M. Treat each row of M as a one-dimensional array of size n. The relation _ just defined apply to the rows of M. Your task is to bubble sort the rows of M with respect to this relation. This means that if R1 and R2 are two rows of M with R1 _ R2, then R1 should appear before R2 in the sorted output.
(a) Complete the following function which takes two arrays **A** and **B** of size **n** as inputs, and returns -1;1;0 according as whether A < B, B < A, or A = B, respectively.

```
int compare ( int A[], int B[], int n )
{
int i;
for ( ______________________ ) { /* loop on i */
if ( ___________________ ) return -1;
if ( ____________________ ) return 1;
}
return ________________ ;
}
```

Ans.

```
int compare ( int A[], int B[], int n )
{
int i;
for ( i=0; i<n; ++i ) { /* loop on i */
if ( A[i] < B[i] ) return -1;
if ( A[i] > B[i] ) return 1;
}
return 0 ;
}
```

(b) Complete the following function that bubble sorts the rows of an m_n two-dimensional array **M** with respect to the relation <. Assume that **m** and **n** are not larger than a suitably defined **MAX_SIZE**.

```
void rowsort ( int M[][MAX_SIZE], int m, int n )
{
int i, j, k, t;
for (i = ___________ ; i >= 0; i--) {
for ( j = ______________________ ) {
if ( compare( ___________,___________,___________ ) > ____________ ) {
/* Swap rows */
for ( ________________ ) {
__________________________________
}
}
}
}
}
```

5. Consider a polynomial with real (floating-point) coefficients:
$f(X)=c_0X^{d0}+c_1X^{d1}+c_2X^{d2}+\_\ \_\ \_+c_{t-1}X^{dt-1}$ with integer degrees $0 <= d_0 <= d_1 <= d_2 <= \_\ \_\ \_ < =d_{t-1}$ and with coefficients $c_i \neq 0$. We call each c_iX^{di} a nonzero term in f (X). We store f as the sequence $(c_0;d_0)$; $(c_1;d_1)$; $(c_2;d_2)$; : : : ; $(c_{t-1};d_{t-1})$ which is sorted with respect to the degrees (the second components in the pairs). We first define a term as follows:

typedef struct {
double coeff; /* The coefficient in a non-zero term */
int degree; /* The degree of X in the term */
} term;

A polynomial is then stored as a structure instance of the following type:

typedef struct {
int nterms; /* The number of non-zero terms */
term *termlist; /* The list of terms, that is, (coefficient,degree) pairs */
} poly;

We assume that the term-list contains a sequence of terms sorted in the increasing order of the degrees, and that no two terms have the same degree. All coefficients are assumed to be non-zero. Moreover, the term-list should be allocated memory just sufficient to store all the non-zero terms in the polynomial.

Let us have two polynomials f and g in the above representation. We plan to compute their sum h = f +g again in the same representation. Before the sum is computed, the number of non-zero terms in h is not known, so we prepare a local array **sum[]** to store the maximum possible number of terms that can appear in the sum. The intermediate addition result is stored in **sum[]**. Finally, the term-list of h is allocated the exact amount of memory as needed, and the local array **sum[]** is copied to the term-list of h.

As an example, let $f(X) = 1-2X^3-3X^7+4X^9-5X^{15}$ and $g(X) = 4+3X +2X^3+X^9$. Their sum can have a maximum of nine non-zero terms. The sum $h(X) = f(X)+g(X) = 5+3X -3X^7 +5X^9 -5X^{15}$ actually contains only five non-zero terms. This happens because each of the sums 1+4 and $4X^9 +X^9$ introduces only one new term. Moreover, the sum $-2X^3+2X^3$ does not add to the sum any term involving X^3.

(a) We first write a recursive helper function to generate the intermediate array **sum[]**. This function uses a merging procedure as in merge sort (notice that the term-lists of **f** and **g** are sorted with respect to the degrees of the terms). The term-lists of **f** and **g** are indexed by **i** and **j**, respectively. The result is stored in the intermediate array **sum[]**, and the index **k** is used for writing to this array. Thus **k** stores the number of non-zero terms. In the recursive calls, the indices **i**, **j** and **k** are changed appropriately. The outermost call in Part (b) gets the exact number of non-zero terms in the final sum. Complete the helper function.

```
int addhelper ( poly f, poly g, int i, int j, term *sum, int k )
{
/* If both f and g are completely read, return the number of non-zero terms */
if ((i == f.nterms) && (j == g.nterms)) return k;
/* If g is completely read (but not f), or the current term in f has lower degree
than that in g, then copy the current term in f to sum */
if ( ( j == g.nterms ) ||
( (_____________________________________________) ) ) {
sum[k] = _______________________________ ;
return addhelper( ________________________ );
}
/* If f is completely read (but not g), or the current term in g has lower degree
than that in f, then copy the current term in g to sum */
if ( (i == f.nterms) || (_________________________________ ) {
sum[k] = __________________________ ;
return addhelper( ________________________ );
}
/* Here the current terms in both f and g have the same degree */
sum[k].degree = _________________________ ;
sum[k].coeff = _____________________________ ;
if (sum[k].coeff) ++k; /* Update k if needed */
return addhelper( __________________________ );
}
```

(b)Complete the following function that adds **f** and **g** by invoking the helper function of Part (a).

```
poly add ( poly f, poly g )
{
poly h;
term *sum;
int i;
/* Allocate the maximum possible amount of memory that may be needed for sum
*/
sum = __________________________________ ;
h.nterms = addhelper(f,g,0,0,sum,0); /* Call the helper function */
/* Allocate the exact amount of memory to the term-list of h */
h.termlist = ________________________ ;
/* Copy the intermediate array sum[] to the term-list of h */
________________________________________
____________________/* Clean locally used dynamic memory */
return h;
}
```

6. (a) Complete the SortedMerge() function below that takes two non-empty lists, each of which is sorted in increasing order, and merges the them into one list which is in increasing order. SortedMerge() should return the new list. Assume that the elements of the lists are

distinct and there are no common elements among the lists. For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20. The linked list node structure is defined as: struct node {int data; struct node *next;}.

```
struct node *SortedMerge(struct node *a, struct node*b){
struct node *mergedList, *head;
if(a->data < b->data)____________________; else________________;
head = mergedList;
while(a!= NULL && b != NULL){
if(a->data < b->data){mergedList->next = a; ____________;}
else{ mergedList->next = b ;_____________;}
mergedList = mergedList->next;
} /* if a list is exhausted before the other */
if(a == NULL)___________________;
else_____________________;
return________________;
}
```

(b) Now, complete the recursive function, RecursiveSortedMerge(),to merge two sorted lists.

```
struct node* RecursiveSortedMerge(struct node* a, struct node* b){
struct node* result = NULL;
if (a==NULL)_______________; /* base cases */
else if (b==NULL)________________;
if (a->data < b->data) { /* pick either a or b, and recur */
result = a;
result->next =______________________; }
else { result = b;
result->next =__________________________; }
return(result); }
```

7. Which one of the following in place sorting algorithms needs the minimum number of swaps?
   (a) Quick sort
   (b) Insertion sort
   (c) Selection sort
   (d) Heap sort.

8. Suppose we have a O(n) time algorithm that finds median of an unsorted array. Now consider a QuickSort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst case time complexity of this modified QuickSort.
   (a) O(n^2 Logn)
   (b) O(n^2)
   (c) O(n Logn Logn)
   (d) O(nLogn)

9. In quick sort, for sorting n elements, the (n/4)th smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the quick sort?
   (a) \theta(n)
   (b) \theta(nLogn)
   (c) \theta(n^2)
   (d) \theta(n^2 log n)

10. (a) Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

(a) 25,12,16,13,10,8,14
(b) 25,14,13,16,10,8,12
(c) 25,14,16,13,10,8,12
(d) 25,14,12,13,10,8,16

(b) Consider the data given in above question. What is the content of the array after two delete operations on the correct answer to the previous question?
(a) 14,13,12,10,8
(b) 14,12,13,8,10
(c) 14,13,8,12,10
(d) 14,13,12,8,10

11. Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort?
(a) O(log n)
(b) O(n)
(c) O(nLogn)
(d) O(n^2)

12. Let P be a quick sort program to sort numbers in ascending order using the first element as the pivot. Let t1 and t2 and t2 be the number of comparisons made by P for the inputs [1 2 3 4 5] and [4 1 5 3 2] respectively. Which one of the following holds?
(a) $t_1 = 5$
(b) $t_1 < t_2$
(c) $t_1 > t_2$
(d) $t_1 = t_2$

13. Given max heap with level order elements as 10, 8, 5, 3, 2 in order. Insert 1 and 7 into the heap tree and find the BFS of the resultant tree.
(a) #
(b) #
(c) 10, 8, 7, 3, 2, 1, 5
(d) #

14. In an array middle element is choosen as pivot element. What is the worst case time complexity of quick sort?
(a) $O(n^2)$
(b) O(nlogn)
(c) O(logn)
(d) $O(n^3)$

15. Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is
(a) 40, 30, 20, 10, 15, 16, 17, 8, 4, 35
(b) 40, 35, 20, 10, 30, 16, 17, 8, 4, 15
(c) 40, 30, 20, 10, 35, 16, 17, 8, 4, 15
(d) 40, 35, 20, 10, 15, 16, 17, 8, 4, 30

--*--