CS251 Homework 1

Handed out: Feb 20, 2017
Due date: Feb 27, 2017 at 11:59pm (This is a **FIRM** deadline, solutions will be released immediately after the deadline)

Question	Topic	Point Value	Score
1	True / False	5	
2	Match the Columns	7	
3	Short Answers	22	
4	Programming Questions	22	
5	Symbol Tables	4	
Total		60	

1. True/False [5 points]

1. Amortized analysis is used to determine the worst case running time of an algorithm. T
2. An algorithm using $5n^3 + 12n\log n$ operations is a $\Theta(n\log n)$ algorithm. F
3. An array is partially sorted if the number of inversions is linearithmic. T
4. Shellsort is an unstable sorting algorithm. T
5. Some inputs cause Quicksort to use a quadratic number of compares. T

2. Match the columns [7 points]

A. Mergesort	1. Works well with duplicates
B. Quicksort	2. Optimal time and space
C. Shellsort	3. Works well with order
D. Insertion sort	4. Not analyzed
E. Selection sort	5. Stable and fast
F. 3-way quicksort	6. Optimal data movement
G. Heapsort	7. Fast general-purpose sort

3. Short Answers [22 points]

(a) Suppose that the running time T(n) of an algorithm on an input of size n satisfies $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + cn$ for all n > 2, where c is a positive constant. Prove that $T(n) \sim cn\log_2 n$. [**4 points**]

T(n) >= T(celi(n/2))+T(floor(n/2))+cn

Say $n=2^k$, floor(n/2) = ceiling(n/2) = 2^{k-1}

$T(2^k)/2^k = 2T(2^{k-1})/2^{k-1} + c2^k/2^k$

$T(2^k)/2^k = T(2^{K-1})/2^{K-1} + c = T(2^{k-2})/2^{k-2} + 1 + c$

$T(2^k)/2^k = (2^0)/(2^0) + ck$

$T(n) = T(2^k) = kc2^k = cn\log_2 n$

$$T(n) \sim cn\log_2 n$$

(b) Rank the following functions in increasing order of their asymptotic complexity class. If some are in the same class indicate so. [**4 points**]

- $n \log n$
- $n^2/201$
- n
- $\log^7 n$
- $2^{n/2}$
- $n(n-1) + 3n$

$\log^7 n < n < n \log n < n^2/201 < n(n-1) + 3n < 2^{n/2}$

(c) Consider the following code fragment for an array of integers:

```
int count = 0;
int N = a.length;
Arrays.sort(a);
for (int i = 0; i < N; i++)
  for (int j = i+1; j < N; j++)
    for (int k = j+1; k < N; k++)
      if (a[i] + a[j] + a[k] == 0)
        count++;
```

Give a formula in tilde notation that expresses its running time as a function of N. If you observe that it takes 500 seconds to run the code for N=200, predict what the running time will be for N=10000. [**5 points**]

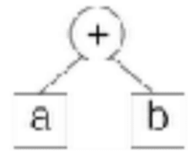
n(n-1)(n-2)/6+nlogn ~ N^3

500 = a*(200(200-1)(200-2)/6 + nlogn) = 500/(200(200-1)(200-2)/6) = a
a * (10000*9999*9998/6+10000log10000) =
$6.33777*10^7$ seconds

(d) In Project 2 you were asked to use `Arrays.sort(Object o)` because this sort is stable. What sorting algorithm seen in class is used in this case? What sorting algorithm would you use if instead of dealing with `Point` objects you were handling `float` values? Justify your answer. [**4 points**]

ANSWER : Merge sort because it is efficient to use it on object. I will use quicksort because float is primitive type, so stability of sort is meaningless. Also, it has faster average case, and even if it has n^2 worst case, it is very easy to avoid the worst case. And it works as in-place.

(e) Convert the following (*Infix)* expressions to *Postfix* and *Prefix* expressions (To answer this question you may find helpful to think of an expression "a + b" as the tree below.) [**5 points**]



(i) (a + b) * (c / d)

(ii) a * (b / c) - d * e

(iii) a + (b * c) / d - e

(iv) a * b + c * (d / e)

(v) a * (b / c) + d / e

i) post : ab+cd/* pre : *+ab/cd

ii) post : abc/*de*- pre : -*/abc*de

iii) post : abc*d/+e- pre : -+a/*bcde

iv) post : ab*cde/*+ pre : +*abc*c/de

v) post : abc/*de/+ pre : +*a/bc/de

4. Programming Questions [**22 points**]

(a) Give the pseudocode to convert a fully parenthesized expression (*i.e.*, an INFIX expression) to a POSTFIX expression and then evaluate the POSTFIX expression. [**5 points**]

```
Program infixToPost:
    s = stack initialize
    while (i = 0 to i < length of infix as char[]) // start of while
        switch(infix[i])
            case number:
                result  += infix[i]
                break;
            case '(' :
                s.push(infix[i])
                break;
            case ')' :
                while(s.top() is not '(')
                    result = s.pop();
                s.pop()
                break;
            case operator(+,-,*,/) :
                while ( s is not empty,
                s.top() is not '(', and
                infix[i] has less or
                equal precedence than
                s.top())
                    result += s.pop();
                s.push(infix[i])
                break;
    //end of while
        while(s is not empty) //start of while
            result += s.pop()
    //end of while
    //end
```

(b) Given two sets A and B represented as sorted sequences, give Java code or pseudocode of an efficient algorithm for computing A ⊕ B, which is the set of elements that are in A or B, but not in both. Explain why your method is correct. [**5 points**]

ANSWER TO (b)

```
)//assume that a and b has same size

import java.util.HashMap;
public class computeXor {
   HashMap<Integer, Integer> a;
   int b[];
   boolean xor[];
   public computeXor(int a[], int b[]) {
       this.a = new HashMap<Integer, Integer>();
       this.b = b.clone();
       for(int i = 0; i < a.length; i++) {
          this.a.put(a[i],a[i]);
      }
      xor = new boolean[b.length];
   }

   public void compute(HashMap a,int b[]) {
      for(int i = 0 ; i < b.length; i++) {
         if(a.containsKey(b[i])) {
            xor[i] = false;
         }else {
            xor[i] = true;
         }
      }
   }

   public HashMap<Integer, Integer> getA() {
      return a;
   }

   public int[] getB() {
      return b;
   }

   public boolean[] getXor() {
      return xor;
   }

   public static void main(String[] args) {
      int[] a = {0,0,1};
      int[] b = {0,1,2};
      computeXor x = new computeXor(a,b);
      x.compute(x.getA(),x.getB());
      for(int i = 0; i < x.getXor().length;i++) {
         System.out.println(x.getXor()[i]);
      }
   }
}
```

I put all a value into hashmap, therefore If there is key(b) contained in hashmap, then it means a has same value with b, so it is false. But if nothing found with key, then it is true because it means it has no same value.

(c) Let A be an unsorted array of integers $a_0, a_1, a_2, \ldots, a_{n-1}$. An inversion in A is a pair of indices (i, j) with i < j and $a_i > a_j$. Modify the merge sort algorithm so as to count the total number of inversions in A in time $\mathcal{O}(n \log n)$. [**5 points**]

```
public class Merge {
   public static int sort(int[] a) {
      return sort(a,0,a.length-1);
   }

   public static int sort(int[] a, int lo, int hi) {
      if(hi <= lo) return 0;
      int mid = lo + (hi-lo)/2;
      int inversion = sort(a, lo, mid);
      inversion += sort(a, mid+1, hi);
      inversion += merge(a,lo,mid,hi);
      return inversion;
   }

   static int merge(int[] arr,int lo,int mid,int hi){
      int n1=mid-lo+1;
      int n2=hi-mid;
      int i=0,j=0,k=lo,invCount=0;
      int[] t1=new int[n1];
      int[] t2=new int[n2];

      for( i=0;i<n1;i++)
         t1[i]=arr[lo+i];
      for( j=0;j<n2;j++)
         t2[j]=arr[mid+j+1];
      i=0;
      j=0;
      while (i<n1 && j<n2){
         if(t1[i]<=t2[j])
            arr[k++]=t1[i++];
         else {
            arr[k++] = t2[j++];
            invCount+=mid-i+1-lo;
         }
      }
      while(j<n2)
         arr[k++]=t2[j++];
      while(i<n1)
         arr[k++]=t1[i++];
      return invCount;
   }

   public static void main(String args[]){
      int[] a = {2, 4, 1, 3, 5};
      System.out.println("inversion :"+ sort(a));

   }
}
```

(d) Let A [1 . . . n] , B[1 . . .n] be two arrays, each containing *n* numbers in sorted order. Devise an $\mathcal{O}(\log n)$ algorithm that computes the *k*-th largest number of the 2*n* numbers in the union of the two arrays. Do not just give pseudocode — explain your algorithm and analyze its running time.

For full credit propose a solution using constant space. [**7 points**]

DOUBLE CLICK

A[1,2,3,…,N], B[1,2,3,…,N]

Program kthLargest(int kth) {

let s1 = A's length, s2 = B's length

k = s1+s2-kth+1 //convert kth to k so we can find k'th smallest, which //is same as kth'th largest!

indexA, indexB, step = 0

while(indexA + indexB < k-1)

step = (k – indexA –indexB) /2

stepA = indexA + step

stepB = index2 + step

if(s1 > stepA -1 and (s2 <= stepB -1 or A[stepA-1] < B[stepB-1])) indexA = stepA

else

indexB = stepB

//while end

if(s1 > indexA and (s2 <= indexB or A[indexA] < B[indexB]))

return A[indexA]

else

return B[index2]

}

It is actually implementation of finding kth smallest, but we know that it is ordered and (length-k)th smallest is kth largest. And because it is modified binary search, so it is O(logk)

5. Symbol Tables [**4 points**]

Draw the Red-Black LL BST obtained by inserting following keys in the given order:

H O M E W O R K S.