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]

T 1. Amortized analysis is used to determine the worst case running time of an algorithm.

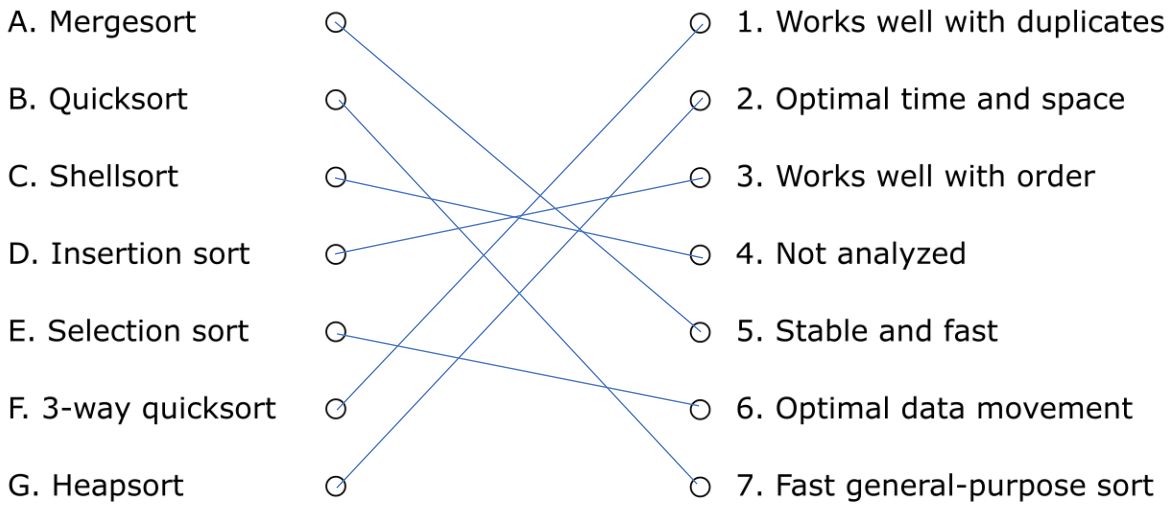
F 2. An algorithm using $5n^3 + 12n \log n$ operations is a $\Theta(n \log n)$ algorithm.

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.

2. Match the columns [7 points]



3. Short Answers [22 points]

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

Base Step: $T(n < 2) = T(\lfloor n < 2 \rfloor) + T(\lceil n < 2 \rceil) + cn = T1 + T2 + cn$

Induction Step: $Assuming\ T(k)\ holds\ true, prove\ T(k+1) \sim cn * \log_2 n$

$Let\ n = 2^k\ for\ some\ k. We\ get\ 2n = 2^{k+1}$

$T(2n) = T(\lfloor n \rfloor) + T(\lceil n \rceil) + cn = T(n) + T(n) + 2cn$

$T(n) \sim cn \log_2 n\ \ so\ \ldots\ cn \log_2 n + cn \log_2 n + 2cn = 2cn(\log_2 n + 1)$

Therefore…

c * (2^{k+1}) ($\log_2 2^{k+1}$) = c * 2^{k+1} * (k+1)

Proving…

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

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

- $n \log n$ — 3.
- $n^2/201$ — 4.
- n — 2.
- $\log^7 n$ — 1.
- $2^{n/2}$ — 6.
- $n(n-1) + 3n$ — 5.

(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**]

The tilde approximation for the inner loop is: $T(N) \sim \frac{N^3}{6}$

To determine the time interval, let's set a variable 't' such that…

$T(N) \sim t * \frac{N^3}{6}$ using the given information we get $T(200) \sim t * \frac{200^3}{6} = 500\ seconds$

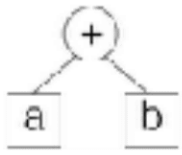
This gives us t = 3/8000. Plugging in we now can solve for N = 10,000:

$$T(10000) \sim \frac{3}{8000} * \frac{10000^3}{6} = 62{,}500{,}000\ seconds$$

(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**]

In the case of the project, Arrays.sort utilized *merge sort* as it was sorting an array of objects. If it were to sort an array of primitive types (such as an array of floats, in the case of this example) Arrays.sort would instead utilize quicksort. The reason merge sort is used for an array of objects is because it is stable as well as faster in the case of dealing with objects (since there is a possibility of duplicate object references in a 'to be sorted' array). Quick Sort can be used for primitive types because they have no identity and thus their relative position in the order does not matter. In an object array the relative position might also matter, making merge sort viable over quick sort for Objects. Overall-- merge sort is faster and more accurate for objects while quick sort is simply more practical for primitive data types.

(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)

Postfix: a b + c d / *
Prefix: * + a b / c d

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

Postfix: a b c / * d e * -
Prefix: - * a / b c * d e

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

Postfix: a b c * d / + e -
Prefix: - + a / * b c d e

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

Postfix: a b * c d e / * +
Prefix: + * a b * c / d e

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

Postfix: a b c / * d e / +
Prefix: + * a / b c / d e

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**]

```
String infixToPostfix(String toTok) {
   StringTokenizer e = new StringTokenizer(toTok);
   String toRet = "";
   Stack<String> stack;
   while (e.hasMoreTokens()) {
      String token = e.nextToken();
      If (token is operand) {
         toRet = toRet + token + " ";
      } else if (token == ")") {
         While (stack.top() != '(')
            toRet = toRet + stack.pop() + " ";
         stack.pop();
      } else {
         While (ISP(stack.top() >= ICP(token))
            toRet = toRet + stack.pop() + " ";
         stack.push(token);
      }
   }
   While (!stack.empty())
      toRet = toRet + stack.pop() + " ";
   Return toRet;
}
```

```
int calcPostfix(String toTok) {
   StringTokenizer e = new StringTokenizer(toTok);
   Stack<Integer> stack;
   While (e.hasMoreTokens()) {
      String token = e.nextToken();
      If (token is operand) {
         stack.push(Integer.parseInt(token));
      } else {
         Remove operands related to operator 'token' from the stack

         Perform the mathematical operation 'token'

         Push the result to the stack
      }
   }

   return stack.pop();

}
```

(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)

```
public static ArrayList<Integer> findIntersection(int[] A, int[] B) {

  ArrayList<Integer> intersection = new ArrayList<Integer>();
  int n1 = A.length;
  int n2 = B.length;
  int i = 0, j = 0;

  while (i < n1 && j < n2) {

    if (A[i] > B[j]) {

      j++;

    } else if (B[j] > A[i]) {

      i++;

    } else {

      intersection.add(A[i]);
      i++;
      j++;

    }

  }

  return intersection;

}
```

JUSTIFICATION:

For the above requested algorithm, I believe I have coded it in the most efficient way possible. The above method results in a O(n) runtime since it iterates two separate counters and compares the array indexes at these counters. This method is very efficient because instead of having to do a binary search there is instead just simple iteration from 0 to n (n being the size of the array). This results in less of a runtime for various sizes of n and thus making this algorithm the most efficient, especially as opposed to a binary search (where it can be up to O(nlogn)) This method only works because the arrays are sorted **beforehand** so we take advantage of that and benefit our runtime (as seen demonstrated in the code above).

(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**]

```
import java.util.Arrays;

   public class countMergeAndSort {

   public static long merge(int[] arr, int[] left, int[] right) {

       int i = 0;
       int j = 0;
       int count = 0;

       while (i < left.length || j < right.length) {
           if (i == left.length) {
               arr[i+j] = right[j];
               j++;
           } else if (j == right.length) {
               arr[i+j] = left[i];
               i++;
           } else if (left[i] <= right[j]) {
               arr[i+j] = left[i];
               i++;
           } else {
               arr[i+j] = right[j];
               count += left.length - i;
               j++;
           }
       }
       return count;
   }

   Public static long sortAndCount(int[] arr) {
       if (arr.length < 2)
           return 0;

       int m = (arr.length + 1) / 2;
       int left[] = Arrays.copyOfRange(arr, 0, m);
       int right[] = Arrays.copyOfRange(arr, m, arr.length);

       return sortAndCount(left) + sortAndCount(right) + merge(arr, left, right);
   }

}
```

(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 $2n$ 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**]

PSEUDOCODE:

```
Function findNthLargest(int[] A, int[] B, int k) {

If (last element in A <= first element in B)
       Return B[B.length - k];

If (last element in B <= first element in A)
       Return A[A.length – k];

Int aIndex = middle index of A;
Int bIndex = middle index of B;

If (A[aIndex] > B[bIndex]) {

       Swap arrays A and B;

}

If (k <= aIndex + bIndex) {
       Int[] firstHalfB = first half of array B;

       Return findNthLargest(A, firstHalfB, k);
}

Int[] secHalfA = first half of array A;

Return findNthLargest(secHalfA, B, k - aIndex);


}
```

EXPLANATION:

This algorithm works in a runtime of log(size of A) + log(size of b) and since and in this case A and B are the same size we can deduce a runtime of log(n).

This works in a way similar to determining the median of an array recursively until you find the desired number. Each recursive iteration determines which half of which array the k-th largest integer lies in and then halves the according array and sends it back into the function until it is either only the desired number or it is at the top of the array. This runtime is very efficient as it ‘divides and conquers’ very much like we learned in class.

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.