CS251 Homework 1

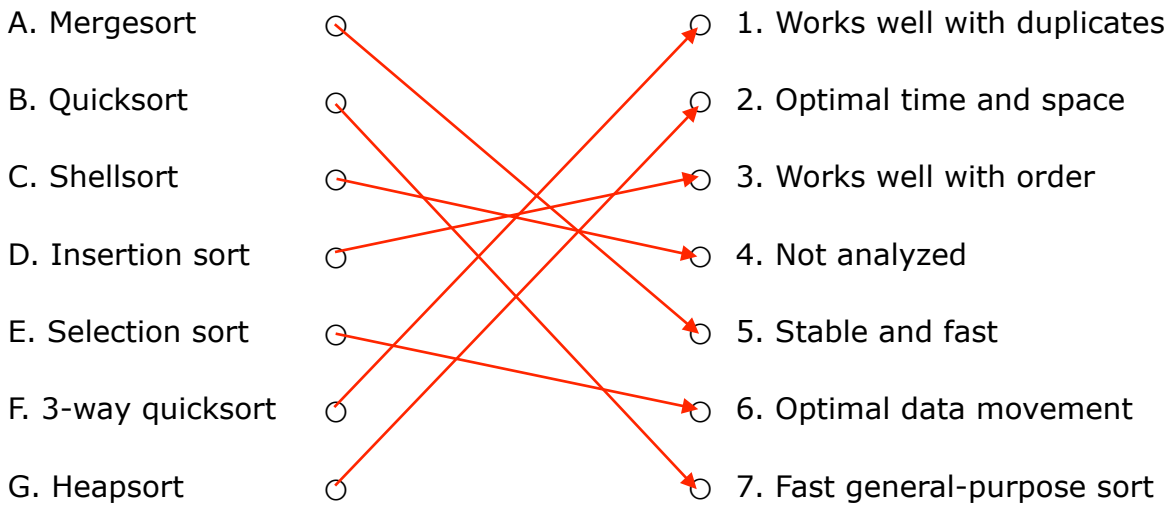
Handed out: Feb 20, 2017
Due date: Feb 27, 2017 at 11:59pm (This is a **FIRM** deadline, solutions will be released immediately after the deadline)

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. (**False**: amortized analysis determines average cost of operations)
2. An algorithm using $5n^3 + 12n \log n$ operations is a $\Theta(n \log n)$ algorithm.
(**False**: Big theta is asymptotic growth rate, i.e. tilde without constant: $\Theta(n^3)$)
3. An array is partially sorted if the number of inversions is linearithmic.
(**False**: partially sorted means number of inversions is linear in the size of the problem)
4. Shellsort is an unstable sorting algorithm. (**True**: long distance exchanges)
5. Some inputs cause Quicksort to use a quadratic number of compares.
(**True**: worst case complexity is quadratic)

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**]

If we assume for simplicity that n is a power of 2, we have:

$$T(n) = 2T(n/2) + cn$$
$$T(n/2) = 2T(n/4) + cn/2$$
$$T(n/4) = 2T(n/8) + cn/4$$

Hence after substitution, we obtain:

$$\begin{aligned} T(n) &= 4T(n/4) + 2cn \\ &= 8T(n/8) + 3cn \\ &\dots \\ &= nT(1) + c\,n \log_2(n) \sim c\,n\,\log_2(n) \end{aligned}$$

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

3 • $n \log n$
4 • $n^2/201 \quad = \Theta(n^2)$
2 • n
1 • $\log^7 n$
6 • $2^{n/2}$
4 • $n(n-1) + 3n \; = \Theta(n^2)$

(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**]

If we name A(n) the number of array accesses in the innermost loop, we have:

$$A(n) = 3\binom{n}{3} = \frac{3\,n!}{(n-3)!\,3!} = \frac{3\,n\,(n-1)\,(n-2)}{6} \sim \frac{n^3}{2}$$

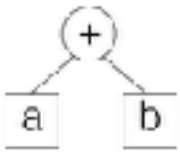
We now name T(n) = T(1) x A(n) the time it takes to perform A(n) array accesses. We know that T(200) = 500s. Hence

$$T(1) = \frac{2 \times 500}{200^3} = 1.25\,10^{-4} \quad \text{and} \quad T(10^4) = \frac{T(1)}{2}\,10^{12} = 6.25\,10^7\,s.$$

(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**]

Arrays.sort() uses **merge sort** for **efficiency** and **stability**. In contrast, if we need to sort primitive types (such as float's) stability becomes irrelevant and **quicksort** is a more efficient solution.

(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**]

```
// Grammar: EXPR -> ( EXPR OP EXPR )
//          EXPR -> VAL
void In2Post(String expr, Stack<String> post)
{ In2Post_(expr, 0, post); }
int In2Post_(String expr, int i, Stack<String>
post) {
  if (expr[i] == '(') { // ( EXPR1 OP EXPR2 )
    i = In2Post_(expr, i+1, post); // EXPR1
    String op = expr[i]; // OP
    i = In2Post_(expr, i+1, post); // EXPR2
    post.push(oper); // postfix
    return ++i; // skip closing parenthesis
  }
  else { // VAL
    post.push(expr[i]);
    return ++i;
  }
}
```

```
Value PostEval(Stack<String> post) {
  Stack<Value> vals;
  while (!post.empty()) {
    String term = post.pop();
    if (isOperator(term)) {
      Value a(vals.pop());
      Value b(vals.pop());
      vals.push(compute(a, b, Operator(term));
    }
    else vals.push(Value(term));
  }
  return vals.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**]

The solution to this problem is a simple variation on the merge function of merge sort.

```
void merge(Comparable[] a, Comparable[] b, Comparable[] a_plus_b) {
 int i=0, j=0, k=0;
 while (i<n || j<n)
  {
    if      (i==n)              a_plus_b[k++] = b[j++];
    else if (j==n)              a_plus_b[k++] = a[i++];
    else if (less(a[i], b[j]))  a_plus_b[k++] = a[i++];
    else if (less(b[j], a[i]))  a_plus_b[k++] = b[j++];
    else { i++; j++; } // skip equal values
}
```

(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**]

Similarly, the merge function can be modified to count the number of inversions: any value from the upper half of the array inserted before a value from the lower half requires a series of inversions to reach its correct position. This number of inversions corresponds to the number of elements left to be processed in the lower half: [i, i+1, …, mid]: mid-i+1.The complexity of merge sort (n log n) is not affected.

```
void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];

   int i = lo, j = mid+1, inversions=0;
   for (int k = lo; k <= hi; k++)
   {
      if (i > mid) a[k] = aux[j++];
      else if (j > hi) {
        a[k] = aux[i++];
      }
      else if (less(aux[j], aux[i])) {
        a[k] = aux[j++];
        inversions += mid-i+1;
      }
      else a[k] = aux[i++];
   }
}
```

(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**]

This problem can be solved with a **binary search**. Assume for convenience that the arrays are sorted in *decreasing* order. The result is based on following observation: A[i] is the k-th largest number of A U B iff A[i] is larger than k-i values in B and not strictly larger than k-i+1 values in B. Indeed, as i-th entry in A, A[i] is larger or equal than i values in A, hence it must be larger than k-i values and not strictly larger than k-i+1 values in B to be the k-largest number. If we call j=k-i we have a similar requirement on B[j] to be the k-largest number.

```
int k_largest(int A[], int B[], int k) {
  int lo=1, hi=k; // range of possible values for i, assuming k<n
  while (lo <= hi) {
    int i = (lo+hi)/2;
    int j = k-i;
    if (B[j]) <= A[i] <= B[j+1]) return A[i];
    else if (A[i] <= B[j] <= A[i+1]) return B[j];
    else if (A[i] > B[j+1]) // i is too large
      hi = i-1;
    else /* if (B[j] > A[i+1]) */ // i is too small
      lo = i+1;
   }
}
```

The complexity result follows directly from the recursive halving of the [lo, hi] interval at each iteration. It is in fact O(log k) and constant space since we did not use a recursion.

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.