Jenna Ellis

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.
2. An algorithm using $5n^3 + 12n \log n$ operations is a $\Theta(n \log n)$ algorithm. False
3. An array is partially sorted if the number of inversions is linearithmic. False
4. Shellsort is an unstable sorting algorithm. True.
5. Some inputs cause Quicksort to use a quadratic number of compares. True.

2. Match the columns [7 points]

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

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

can be simplified to

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

and further:

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

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

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

$= 8T(n/8) + 3cn \longrightarrow T(n) = 2^k T(n/2^k) + cn$ for all k.

Proof by Induction:

k=1

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

For k-1 → k

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

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

$= 2^k T(n/2^k) + k \cdot cn \longrightarrow$

For $n/2^k = 1$ or $k = \log_2 n$

$T(n) = nT(1) + cn \log n - cn$

$= \sim cn \log n$

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

INCREASING

1) $\log^7 n$
2) n
3) $n \log n$
4) $n^2/201$ AND $n(n-1)+3n$ (both are $\Theta(n^2)$)
5) $2^{n/2}$

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

```
int count = 0;                                  → 1
int N = a.length;                               → 1
Arrays.sort(a);                                 → n log n
for (int i = 0; i < N; i++)                     → n
  for (int j = i+1; j < N; j++)                 → (n-1)
    for (int k = j+1; k < N; k++)               ⇒ (n-1)(n-2)/2 = (n²-3n+2)/2 · n = (n³-3n²+2n)/2
      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**]

$\frac{1}{2}(n^3-3n^2+2n) + n\log n + 2$

$\frac{1}{2}n^3 - \frac{3}{2}n^2 + n + n\log n + 2$

$T(N) = \boxed{\sim \frac{1}{2}n^3}$

$T(N=200) = 500$ seconds

$T(N=10000) = ?$

$\frac{T(N=200)}{\frac{1}{2}(200)^3} = \frac{T(10000)}{\frac{1}{2}(10000)^3}$

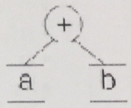
$T(10000) \cdot \frac{1}{2}(200)^3 = 500 \cdot \frac{1}{2}(10000)^3$

$\boxed{T(10,000) \approx 62\,500\,000 \text{ seconds}}$

(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**]

Arrays.sort(Object o) implements mergesort when dealing with any comparable or comparator (in this case) objects. If, instead of Point objects, we were sorting floats, I would use quicksort. Arrays.sort does this when sorting primitive types as well. For objects, a stable sorting algorithm is necessary to appropriately & correctly sort them. Also, for primitive types, using mergesort results in a doubled memory usage (the second array). Quicksort is an unstable sorting algorithm, but this does not affect primitive type sorts, and it manages memory better for this application.

(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)
Prefix: * +ab /cd
Postfix: ab+ cd/ *
(ii) a * (b / c) - d * e
Prefix: - * a/bc *de
Postfix: abc/ * de* -
(iii) a + (b * c) / d - e
Prefix: - +a/ *bc de
Postfix: abc* d/ +e -
(iv) a * b + c * (d / e)
Prefix: + * ab *c/de
Postfix: ab* c de/* +
(v) a * (b / c) + d / e
Prefix: + * a /bc /de
Postfix: abc/ * de/+

• each of these were derived by either drawing a binary tree (alike the one to the right) or by drawing each expression fully parenthesized + moving the operators accordingly.

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]**

Convert to Postfix:
Let values be a string ~~stack~~ queue
Let exp be a INFIX expression
Let ops be a string stack.
for i:=0 to length(exp)
if (exp[i] != "+" or "-" or "/" or "*")
values.enqueue(exp[i]);
else if (exp[i] = "+" or "-" or "/" or "*")
ops.push(exp[i]);
else if (exp[i] = ")")
op = opr.pop()
values.enqueue(op); end for

//printing process
for i:=0 to size(values)
print(values.dequeue());

Evaluate Postfix
Let "values" be an integer stack.
Let exp be a string postfix expression.
for i:=0 to length(exp)
{ if(exp[i]!= "*" or "/" or "-" or "+")
values.push(exp[i]);
else {
int op_1 = values.pop()
int op_2 = values.pop()
switch(exp[i])
case "+": values.push((op_2+op_1))
case "-": values.push((op_2-op_1))
case "*": values.push((op_2*op_1));
case "/": values.push((op_2/op_1)).

return values.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]**

My pseudocode is given on the next page. My method is correct because it follows the same logic of the merge portion of merge sort, but, when it encounters 2 elements of the same value, it simply skips over both values, and continues without inserting them into the merged set.

Solution: Let A and B be sorted arrays of comparable objects.

Let a be an int = 0; let b = 0 as well.
Let i = 0;
Let NEW be an empty array of size (A.length + B.length).
while (a < A.length && b < B.length {
  if (A[a].compareTo(B[b]) < 0) {
    NEW[i] = A[a];
    a++;
    i++;
  }
  else if (A[a].compareTo(B[b]) > 0)
  {
    NEW[i] = B[b];
    b++;
    i++;
  }
  else { // equal
    b++;
    a++;
  }
}
while (a < A.length) { NEW[i] = A[a]; a++; i++; }
while (b < B.length) { NEW[i] = B[b]; b++; i++; }

return NEW;

(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]**

```
private static int sort(Int[] A, Int[] aux, int lo, int hi) {
  if (hi <= lo) return 0;
  int mid = lo + (hi - lo)/2, invert = 0;
  invert += sort(A, aux, lo, mid);
  invert += sort(A, aux, mid+1, hi);
  invert += merge(A, aux, lo, mid, hi);
  return invert;
}
```

- This algorithm returns a counting variable from merge which sorts to maintain inversions during sorting.

```
private static int merge(Int[] A, Int aux, int lo, mid, hi
  assert isSorted(a, lo, mid);
  assert isSorted(a, mid+1, hi);
  for(int k = lo; k <= hi; k++)
    aux[k] = a[k];
  int invert = 0; i = lo; j = mid+1;
  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++];
      invert++; }
    else a[k] = aux[i++];
  }
  return invert; }
```

(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]**

```
Given K,
let r = 2K
i = 0;
j = 0
while (r > K) {
  if (A[i] > B[j])
    j += (b.length - j)/2;
  else
    i += (a.length - i)/2;
  r = i + j;
}
```

```
if (A[i] > B[j])
    return (A[i]);
else
    return B[j];
```

This algorithm partitions the two arrays side by side in order to identify the k^{th} largest integer between them. The values i + j are halved to update partitions, while r = i + j (the number of values still being considered) is still > k. When r = k, then i & j point to indices that are considered to be the k^{th} largest.

This algorithm is O(log n) because i + j are updated in increments of half of their value, efficiently finding the integer in logarithmic time.

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.