Assignment 6 - Searching, Sorting, and Efficiency Analysis

[100 points] - due May 6, 11PM

For this assignment, you will complete searching/sorting tasks and efficiency analysis. No code is to be written for this assignment.

Problem 1 - Binary Search

Search for the character **S** using the binary search algorithm on the following array of characters: **A E G K M O R S Z**

For each iteration of binary search use a table similar to the table below to list: (a) the left index and (b) the right index of the array that denote the region of the array that is still being searched, (c) the middle point of the array, and (d) the character-to-character number of comparisons made during the search (line 09 and line 11 of the Binary Search algorithm at the end of the assignment).

Iteration	Left	Right	Middle	Number of Comparisons

Problem 2 - Selection Sort

List the resulting array after each iteration of the outer loop of the selection sort algorithm. Indicate the number of character-to-character comparisons made for each iteration (line 10 of the Selection Sort algorithm at the end of the assignment). Sort the following array of characters (sort into alphabetical order): **C Q S A X B T**

Problem 3 - Insertion Sort

List the resulting array after each iteration of the outer loop of the insertion sort algorithm. Indicate the number of character-to-character comparisons made for each iteration (line 09 of the Insertion Sort algorithm at the end of the assignment). Sort the following array of characters (sort into alphabetical order): **C Q S A X B T**

Problem 4 - Mergesort

List the resulting array after each iteration of the mergesort algorithm. Indicate the number of character-to-character comparisons made for each call to merge (line 22 of the Merge algorithm at the end of the assignment). Sort the following array of characters (sort into alphabetical order): **C Q S A X B T**

Problem 5 - Big-oh

For each problem given below, do the following:

1. Create an algorithm in pseudocode to solve the problem.
2. Identify the factors that would influence the running time of your algorithm. For example, if your algorithm is to search an array the factor that influences the running time is the array size. Assign names (such as n) to each factor.
3. Count the operations performed by the algorithm. Express the count as a function of the factors you identified in Step 2. To do that, identify the basic operations of the algorithm. There is no need count every statement separately only the ones that will influence the running time.
4. Describe the best case scenario for the algorithm and derive the big O.
5. Describe the worst case scenario for the algorithm and derive the big O.

The problems are:

a. Determine if two arrays have no elements in common.

b. Counting the total number of characters that have a duplicate within a string, including spaces. (i.e. "gigi the gato" would result in 7 (g x 3 + i x 2 + t x 2 + ' ' x 2)

c. Finding a row where every entry is 'x' in a 2-D array.

Submission

Use your preferred text editor to type your answers, and then save the file in **PDF format**. We will only grade typed assignments and in PDF format. You will use Gradescope to submit your assignment.

`https://www.gradescope.com/`

If you have not yet used Gradescope, go to your scarletmail inbox and look for an email from Gradescope with instructions on how to access your account.

Read **Help** $->$ **Student Workflow** $->$ **Submitting a PDF** for instructions on how to submit your **PDF file**.

Binary Search

```
01 public static int binarySearch (int[] a, int target) {
02
03   int left = 0;
04   int right = a.length - 1;
05
06   while ( left <= right ) {
07
08     int middle = ( left + right ) / 2;
09     if ( a[middle] == target ) { // basic operation
10       return middle;
11     } else if ( target < a[middle] ) {
12       right = middle - 1;
13     } else {
14       left = middle + 1;
15     }
16   }
17   return -1; // target not found
18 }
```

Selection Sort

```
01 public static void selectionSort (int[] a) {
02
03   for ( int i = a.length - 1 ; i > 0; i-- ) {
04     // i marks the last item of the unsorted region
05
06     // find the largest item in the unsorted region
07     int maxLoc = 0; // assume the first item is the largest
08
09     for ( int j = 1; j <= i; j++ ) {
10       if ( a[maxLoc] < a[j] ) {  // basic operation
11         maxLoc = j;
12       }
13     }
14
15     // swap largest item with last item of the unsorted region
16     int temp = a[maxLoc];
17     [maxLoc] = a[i];
18     a[i] = temp;
19   }
20 }
```

Insertion Sort

```
01 public static void insertionSort (int[] a) {
02
03   for ( int i = 1; i < a.length; i++ ) {
04     // i marks the beginning of the unsorted region
05     int itemToInsert = a[i]; // first item of the unsorted region
06     int loc = i - 1;    // loc marks the end of the sorted region
07
08     while ( loc >= 0 ) {
09       if ( a[loc] > itemToInsert ) { // basic operation
10         a[loc+1] = a[loc];
11         loc -= 1;
12       } else {
13         break;
14       }
15     }
16     a[loc+1] = itemToInsert;
17   }
18 }
```

Mergesort

```
01 public static void mergesort (int[] a, int l, int r){
02
03   if (l >= r) {
04     return;
05   }
06   int middle = (l+r)/2;
07   mergesort(a, l, middle);
08   mergesort(a, middle+1, r);
09   merge(a, l, middle, r);
10 }
11
12 public static void merge (int[] a, int l, int m, int r){
13
14   //copy lower half of the array into b
15   int[] b = new int[m-l+1];
16   for ( int i = 0; i <= m-l; i++ ) {
17     b[i] = a[l+i];
18   }
19
```

```
20   int i = 0, j = m+1, k = l;
21   while ( i <= m-l && j <= r ) {
22     if (a[j] < b[i]) {
23       a[k] = a[j];
24       k += 1;
25       j += 1;
26     } else {
27       a[k] = b[i];
28       k += 1;
29       i += 1;
30     }
31   }
32   while ( i <= m-l ) {
33     a[k] = b[i];
34     k += 1;
35     i += 1;
36   }
37   while ( j <= r ) {
38     a[k] = a[j];
39     k += 1;
40     j += 1;
41    }
42 }
```