Daniel Bartolo
Introduction to Computer Science
04/24/18

PROBLEM 1
1) Trace selection sort on the following array of letters (sort into alphabetical order).

Initial Array: C Q S A X B T

I) At the first pass, we start at index 0 and find the smallest value lexicographically starting from index 0. After 6 comparisons, it is determined that A is the smallest value. So we swap A with the element at the current index (0).

First Pass: A Q S C X B T

II) At the second pass, we start at index 1 and find the smallest value lexicographically starting from index 1. After 5 comparisons, it is determined that B is the smallest value. So we swap B with the element at the current index (1).

Second Pass: A B S C X Q T

III) At the third pass, we start at index 2 and find the smallest value lexicographically starting from index 2. After 4 comparisons, it is determined that C is the smallest value. So we swap C with the element at the current index (2).

Third Pass: A B C S X Q T

IV) At the fourth pass, we start at index 3 and find the smallest value lexicographically starting from index 3. After 3 comparisons, it is determined that Q is the smallest value. So we swap Q with the element at the current index (3).

Fourth Pass: A B C Q X S T

V) At the fifth pass, we start at index 4 and find the smallest value lexicographically starting from index 4. After 2 comparisons, it is determined that S is the smallest value. So we swap S with the element at the current index(4).

Fifth Pass: A B C Q S X T

VI) At the sixth pass, we start at index 5 and find the smallest value lexicographically starting from index 5. After 1 comparison, it is determined that T is the smallest value. So we swap T with the element at the current index(5).

Sixth Pass: A B C Q S T X

2) Trace insertion sort on the following array of letters (sort into alphabetical order).

Initial Array: C Q S A X B T

To begin with, we have to divided the array into 2 regions, sorted and unsorted.

I) At the first pass, we start at index 1 and compare the element at index 1 with the elements in the sorted region (C) lexicographically. Given that, Q is "greater than" C, Q is not moved. Total Comparisons: 1.

First Pass: C Q (SORTED) S A X B T (UNSORTED)

II) At the second pass, we start at index 2 and compare the element at index 2 with the elements in the sorted region (C Q) lexicographically. Given this, none of the elements are shifted. Total Comparisons: 1

Second Pass: C Q S (SORTED) A X B T (UNSORTED)

III) At the third pass, we start at index 3 and compare the element at index 3 with the elements in the sorted region (C S Q) lexicographically. Given this, C, Q, and S are shifted to the right and A is placed before C. Total Comparisons: 3

Third Pass: A C Q S (SORTED) X B T (UNSORTED)

IV) At the fourth pass, we start at index 4 and compare the element at index 4 with the elements in the sorted region (A C Q S) lexicographically. Given this, none of the elements are shifted. Total Comparisons: 1

Fourth Pass: A C Q S X (SORTED) B T (UNSORTED)

V) At the fifth pass, we start at index 5 and compare the element at index 5 with the elements in the sorted region (A C Q S X) lexicographically. Given this, C, Q, S, and X are shifted to the right and B is placed before C. Total Comparisons: 5

Fifth Pass: A B C Q S X (SORTED) T (UNSORTED)

VI) At the sixth pass, we start at index 6 and compare the element at index 6 with the elements in the sorted region (A B C Q S X) lexicographically. Given this, X is shifted to the right and T is placed before X. Total Comparisons: 2

Sixth Pass: A B C Q S X T

PROBLEM 2

a. DETERMINE IF TWO ARRAYS HAVE NO ELEMENTS IN COMMON

1) For my algorithm, I created two for loops. For the first loop, I had it iterate through the first array. Then I had a second for loop inside the first for loop to iterate through the second array. In the second for loop I had an if statement comparing the current index referenced by the first loop with the indices of the second array. If the current index of the first loop was equal to any indices of the second array, I returned false to indicate that the two arrays had an element in common. However, if both arrays had no elements in common, then I would return true.

PSEUDOCODE:

```
a) For i to array1.length
        For j to array2.length
            if array[i] == b[j]
                return false
   return true
```

2) One factor that could influence the running time would be if the size of either array (n) is very large. This would increase the running time because the loop would then have to iterate through more elements of the array, also adding to the number of comparisons performed. Another factor that could influence the running time would be if the "duplicate" element is at or near the end of the first array. This would result in more iterations and more comparisons because the first loop would have to iterate until it is at the end or near the end of the first array. The first factor could be known before the code is executed.

3) Operations that must be counted:

a) initialization of i in the first for loop, and comparison of i to the length of the first array

b) initialization of j in the second for loop nested inside the first for loop, and comparison of j to the length of the second array

c) the comparison of every element in the second array with the element in the first array referenced by i

d) incrementation of i and j when necessary

e) returning true if both arrays had no elements in common or false if both arrays had an element in common

4) Operations performed by the algorithm or code:

a) initialization of both i and j = 2

b) incrementation of both i and j = n (size of array)

c) comparisons of i and j with the lengths of the first and second array = n (size of array)

d) comparison with elements of both arrays = n (size of array)

e) returning true or false = 1

Extra: Bounds on the best case is 1

Bounds on the worst case is n*m

5) Determine best case inputs, worst case inputs, and efficiency of your implementation

a) Best Case Input: If both arrays contain no elements thus, it would return true (both arrays have no elements in common). Also if both arrays have a length of 1 and each contain a different element thus, returning true (both arrays have no elements in common).

b) Worst Case Input: If the length of either one array or both arrays are very long, it would take a large amount of operations to return an answer. Another worst case would be if the duplicate element is at or near the end of the first array because then the first for loop would have to iterate until the end of the first array, but first it would have to go to through second loop each time.

c) Efficiency: n ∗ m

6) Transform your count formula into big-O notation:

The big-O is O(n ∗ m). I came to this conclusion because I had two for loops for my algorithm, one which iterated through one array and the other through another array. As a result, this algorithm has a big-O of n ∗ m

b. COUNTING TOTAL NUMBER OF CHARACTERS THAT HAVE DUPLICATE WITHIN A STRING (i.e. "gigi the gato" would result in 7 (g x 3 + i x 2 + t x 2)

1) For my algorithm, I created two for loops. For the first for loop, I had it iterate through the length of the string. Then I had the second for loop inside the first for loop to iterate through the length of the string. However, the second for loop would start at i + 1. I did this because inside the second for loop, I had an if statement, that read "if the character at i equals the character at i + 1, increment count by one. However, after the first for loop, we need to check if we have come across a character that has already been used, if so, then we would stop and continue on to the next character that we haven't come across yet. Lastly, I would return the number of duplicates in the string provided.

PSEUDOCODE:

```
a) FOR i to s.length()
        int count = 1
        if current character has already been accounted for
continue on to the next iteration
        FOR j = i + 1 to s.length()
                if (s.charAt(i) == s.charAt(j))
                        ADD COUNT BY 1
    return count
```

2) One factor that could influence the running time would be if the size of the string (n) is vey large. This would increase the running time because the first and second for loop would have to iterate through more characters of the string. Thus, adding to the amount of comparison and fundamental operations that need to be performed. Another factor that could influence the running time would be if one

or more characters has many duplicates throughout the whole string. This would cause in increase in comparisons and incrementation of the counter.

3) Operations that must be counted:
a) initialization of i in the first for loop, and comparison of i to the length of the string
b) initialization of the count variable
c) check to see if the current character has been "used" already
d) initialization of j in the second for loop nested inside the first for loop, and comparison of j to the length of the string
e) the comparison of the current character with the rest of the characters in the string to see if they are equal, if so increment count by one
f) incrementation of i, j, and count when necessary
returning the number of duplicates or -1 if there are no duplicates

4) Operations performed by the algorithm or code:
a) initialization of i, j, and count = 3
b) incrementation of i and j = n (length of string)
c) incrementation of count = n (number of duplicates)
d) comparisons of i and j to the length of the string = n (length of string)
e) comparing current character with other characters of the string = n (size of string)
f) returning the number of duplicates = 1
Extra: Bounds on the best case is 1
Bounds on the worst case is n^2

5) Determine best case inputs, worst case inputs, and efficiency of your implementation
a) Best Case Input: If the the string is of length 0 or 1, thus, there would be a little amount of comparisons. Thus, the algorithm would return -1 because there are now duplicates. Another best case could be if their are no duplicates or the length of the string isn't long.
b) Worst Case Input: If the length of the string is very long, then there would be an increase in the number of operations that the algorithm would have to perform. Another worst case would be if the string contains numerous amounts of duplicates or if the duplicates were at the end of the string. This would cause the algorithm to go through the whole string numerous times and iterate until it reaches the end (for case that duplicates are at the end of the string).
c) Efficiency: n^2

6) Transform your count formula into big-O notation:
a) The big-O is O(n^2). I came to this conclusion because I had two for loops that each iterated through the same string. As a

result, the algorithm has a big-O of n^2

c. FINDING A ROW WHERE EVERY ENTRY IS 'X' IN A 2-D ARRAY.

1) For my algorithm, I created two for loops. For the first loop, I had it iterate through the length of the 2-D array. I also had a count variable to count the number of times each element in a row was equal to x. After the count variable, I had a for loop nested inside the first for loop. The purpose of this for loop was to iterate through element in each row of the 2-D array. Inside the second for loop, I had conditional statement that read "if arr[i][j] != 'x'" then break. Then every time that the element didn't equal 'x', I added one to count. Lastly, I had a statement that if the count variable was equal to the row, it would return the index of the row.
PSEUDOCODE:

```
        a) FOR i to arr.length
                int count = 0
                FOR j to arr[i].length
                        IF arr[i][j] != 'x'
                                break
                        count++
                        IF count == arr[I].length
                                return i
           return -1
```

2) One factor that could influence the running time would be if the size of the 2-D array (n) is very large. This would increase the running time because the first for loop would have to iterate through even more rows (unless x is in every entry in either of the first couple rows), adding to more comparisons and fundamental operations. Another factor that could influence the running time would be if the row that contains an 'x' in every entry is located at or near the end of the 2-D array. Thus, there would be more iterations (n) to perform as well as the fundamental operations. The first factor could be known before the code is executed.

3) Operations that must be counted:
   a) initialization of i in the first for loop, and comparison of i to the length of the 2-D array
   b) initialization of the count variable
   c) initialization of j in the second for loop nested inside the first for loop, and comparison of j to the length of the current row
   d) the comparison of every element in each row to see if it not equal to 'x'
   e) incrementation of i, j, and count when necessary
   f) returning i for the row that contains all 'x's or -1 if none of the rows has x in every entry

4) Operations performed by the algorithm or code:
   a) initialization of i, j, and count = 3

b) incrementation of i, and j = n (size of the array)
c) incrementation of count = n (number of 'x's)
d) comparisons of i with the 2-D array length, j with the current row length, and count with the current row length = n
e) returning the row index or -1 = 1
Extra: Bounds on the best case is 1
Bounds on the worst case is n*m

5) Determine best case inputs, worst case inputs, and efficiency of your implementation

a) Best Case Input: If the first row of the 2-D array contains all 'x's thus, it would return the row at which this occurs (in this case 0).

b) Worst Case Input: If the length of the 2-D array is long, then it would take a large amount of time/operations to get to an end result. Another worst case would be if the row that contains an 'x' in every entry is at or near the end of the 2-D array because then the first for loop would have to iterate until near the last index of the 2-D array performing even more operations.

c) Efficiency: n*m

6) Transform your count formula into big-O notation:

a) The big-O is O(n*m). I came to this conclusion because I had two for loops for my algorithm, one which iterated through the length of the array and the other one which iterated through the rows of the same array. As a result, this algorithm has a big-O of n*m.