**Problem 1**

1. **Selection sort**

X A T B Q S B (6)

1. A X T B Q S B (5)
2. A B T X Q S B (4)
3. A B B X Q S T (3)
4. A B B Q X S T (2)
5. A B B Q S X T (1)
6. A B B Q S T X
7. **Insertion sort**

X A T B Q S B (1)

1. A X T B Q S B (2)
2. A T X B Q S B (3)
3. A B T X Q S B (3)
4. A B Q T X S B (3)
5. A B Q S T X B (5)

A B B Q S T X

**Problem 2**

1. **2 Arrays**
2. A loop to go through every element of the array a. A loop to go through every element of array b (nested loop)

if(a[i] equals b[j])

Return false

else return true; -this is outside the nested loops

1. Operations affecting the run-time =
2. The number of elements in the array.
3. Array dimension: 3D would take more time than 2D.
4. No same elements in the arrays: the program would compare every element of every array.
5. Number of operations vary for different cases
6. Best case: first element of both arrays is same, number of operation = 1
7. Worst case: No same elements in the arrays. number of operations = n2, or n \* m if arrays of different lengths.
8. Increment of i/j, comparison of i/j with array length to see if it's not going out of bounds, and comparison of the elements of array.

The most occurring and time consuming is the comparison of elements of array a and array b.

1. Best case is when the first element of both arrays is same, so it returns false in the first step. Worst case is when array a and array b have no same elements, in which case it goes to the last element of both arrays and returns true.
2. O(n2)
3. **Duplicate characters**
   1. First, the string would be simplified to remove spaces and punctuations. A loop (with int i) to go through every character of the string a. While another loop (with int j) would go through each character of string to check it with the ith character. i starts at 0 ends at string length -2, j starts at i+1 ends at string length -1.

If(a == null)

Break;

If ( a.charAt( i ) equals a.charAt( j ) )

Count++;

if (count>1){ -this is outside the nested loops

Gcount += count;

Remove that char from the array

}

If(count>1 && j = a.length()-1)

Return Gcount;

* 1. Operations affecting the run-time =
     1. The number of characters in the string.
     2. No same characters in the String: the program would compare every character with every other character.
  2. Number of operations vary for different cases
     1. Best case is if all the characters are equal, wherein the program would increase the count and remove all the characters resulting in a null string which would not proceed further. So the number of operations would be string length (n).
     2. Worst case would be if none of the characters are same: since the program needs to check through the whole string in any case because of the same character count is not greater than 1, the character stays. The number of operations would therefore be n(n-1) + (n-1)(n-2)+(n-3)(n-4)…(2)(1) which is equal to n\*(n2) +constant.
  3. Increment of i/j, comparison of i/j with string length to see if it is going out of bounds, and comparison of the characters of string.

The most occurring and time consuming is the comparison of characters of the string.

* 1. Best case is if all the characters are equal, wherein the program would increase the count and remove all the characters resulting in a null string which would not proceed further.

Worst case would be if none of the characters are same: since the program needs to check through the whole string in any case because of the same character count is not greater than 1, the character stays.

* 1. O(n3)

1. **‘a’ row**
2. One loop to go through every row (1st dimension) of the array a. Another loop to go through every column (2nd dimension) of array a. (nested loop)

If ( a [ i ] [ j ] equals ‘a’)

Count++;

Else{

count = 0;

break;

}

if(count == a[0].length) -this is outside the second loop (one which checks columns)

return i;

else

return -1;

1. Operations affecting the run-time =
   * 1. The number of elements in any or both dimensions of the array.
     2. The number of a’s in a row: more number of rows result in greater number of comparisons
     3. Which row has all a’s: if it’s the last one, there are many comparisons
2. Number of operations vary for different cases
3. Best case: whichever is shorter between row-length and column-length

first row of the array has all a’s, number of operations = a.length.

There are no a’s. Number of operations = a [ 0 ].length

1. Worst case: No row has all a’s however every row has a’s upto the next to last element. For example:

1st row: a a a a a a a a b

2nd row: a a a a a a a a b, and so on.

number of operations = a.length \* a[0].length. Assuming it is a square array to simplify the function expression to n2.

1. Increment of i/j, comparison of i/j with array row-length and column-length to see if they are not going out of bounds, and comparison of a[ i ] [ j ] with ‘a’.

The most occurring and time consuming is the comparison of a[ i ] [ j ] with ‘a’.

1. Best case is when the first row of the array has all a’s, so the number of operations is just the row length. Or when there are no a’s, number of operations is the column-length. Worst case is when every row has a’s upto the next to last element, however no row has all a’s in which case it goes to the last element of all the rows but returns -1 at the end.
2. O(n2)