Q1

(a)

Given below is an algorithm:

1. procedure PROC1(array 𝐴, int 𝑛)

2. if 𝑛 > 0 then

3. PROC1(𝐴, 𝑛 − 1)

4. 𝑥 ← 𝐴[𝑛]

5. 𝑖 ← 𝑛 − 1

6. while 𝑖 ≥ 0 and 𝐴[𝑖] > 𝑥 do

7. 𝐴[𝑖 + 1] ← 𝐴[𝑖]

8. 𝑖 ← 𝑖 – 1

9. end while

10. A[i+1] ← x

11. end if

12. end procedure

*Write down the* ***recurrence******relation*** *𝑇(𝑛) for the* ***worst******case******number******of******operations*** *(you may employ the big O notation in your answer) and give its* ***worst******case******time******complexity****.*

***What*** *is the* ***algorithm******computing****?*

Actual code

public int[] proc1(int[] A, int n) {  
 if(n > 0) {  
 proc1(A, n-1);  
 int x = A[n];  
 int i = n - 1;  
 while(i >= 0 && A[i] > x) {  
 A[i + 1] = A[i];  
 i = i - 1;  
 }  
 A[i+1] = x;  
 }  
 return A;  
}

* The algorithm is known as *Insertion sort.*
* Recurrence relation T(n) for the worst case number of operations (may use big O notation):

The recurrence relation of recursive selection sort is T(n) = T(n-1) + n. It can be solved by the method of substitution and is found to be equal to n^2.

* The worst case time complexity of the algorithm is big Theta of (n^2). (Θ(n^2 ))) and it occurs when the input array is sorted in reverse order. (e.g. A = {5, 4, 3, 2, 1}).
* What is the algorithm computing: The algorithm is given as input an array and its size. Then the input array is sorted in ascending order (e.g. inputting (5, 4, 3, 2, 1) will output (1, 2, 3, 4, 5)). The algorithm sorts the array by keeping a subsequence of elements on the left in the correctly sorted order. This subsequence is increased by inserting the next element into its relatively correct position in the sorted subsequence. Adding to the subsequence stops when the last element of the array has been added and sorted.

(b)

(i)

A comparison-based sorting algorithm is a sorting algorithm that can only gain information about items in the input sequence (a1, a2, …, an) by performing pairwise-comparisons. The only requirement for applying a comparison-based sorting algorithm is that the operator forms a total preorder over the data. Meaning:

* If ai <= aj and aj <= az then ai <= az (transitivity)
* For all ai and aj, ai <= aj or aj <= ai (connexity)

In other words, the algorithm can be applied only when the input data can be ordered.

(ii) The translation of the pseudocode:

public int[] proc2(int[] array) {

for(int j = 0; j < array.length - 1; j++) {

int min = j;

for(int i = j + 1; i < array.length; i++) {

if(array[i] < array[min]) {

min = i;

}

}

int temp = array[j];

array[j] = array[min];

array[min] = temp;

}

return array;

}

This is a selection sort.

Selection sort is in-place and **not** stable. Its worst case time complexity is the same as for insertion sort – big Theta of n^2 (Θ(n^2 )).

For an algorithm to be in-place, the memory used has to be O(1). Selection sort in in-place because it does not require extra space in order to sort the input data. To prove that the algorithm is in-place we need to go through how the algorithm sorts the input. First, the algorithm divides the input into two parts – sorted and unsorted. Initially, the sorted sublist is empty and the unsorted sublist is the whole input. Then the algorithm repeatedly finds the smallest element from the unsorted part and swaps it with the leftmost unsorted element. That element is now “added” to the sorted sublist and “removed” from the unsorted sublist. The process is repeated until there are no elements left in the unsorted sublist and all the elements are part of the sorted sublist. Of course, both sublists do not physically exist. The whole input is divided into sublists with the use of pointers which indicate where they start and end. This means that using the Selection sort algorithm to sort an input does not require extra space in order to correctly sort the input data.

For an algorithm to be stable, the order of the elements that have the same value should not be changed. Selection sort is **not** stable. As described above, the way that selection sort works is by swapping element from the front of the array into the spot vacated by the minimum element. Because of this, the sorted order is not correct, making selection sort **not** stable. An example showing selection sort is **not** stable:

Input array: {4, 4, 2}. Let the two 4’s be shown as “a” and “b” to be able to distinguish between them. ({a, b, 2}).

If we try to sort the example input array using selection sort we get as an output:

{2, 4, 4} or {2, b, a}.

From this example we can clearly see that, while the output is valid (the array is sorted correctly), that the order of the numbers with the same value is changed. This example proves that selection sort is **not** stable.

(iii)

We can modify Selection sort by using an implicit heap data structure. Using the heap data structure in conjunction with Selection sort, we greatly improve the base algorithm. This “new” algorithm is called Heapsort.

Heapsort, like selection sort, divides its input into a sorted and unsorted part. The difference is that Heapsort does not linearly scan the unsorted part. Instead, it holds the unsorted part in a heap data structure which allows the algorithm to more quickly find the sought-after element in each step.

Implementing the heap data structure in the Selection sort algorithm improves the worst-case time complexity of the algorithm from Θ(n^2 ) to Θ(n log n).

Implementing the heap data structure does not change the space complexity of Selection sort (which is O(1)). This is because the heap data structure is not physically created but merely the data in the unsorted part of the input array is rearranged into a heap, in place.

(iv)

In order to improve the performance of the original Quicksort algorithm we could:

* Use multiple pivots to partition its input into some *x* number of subarrays using *x – 1* pivots. In the original Quicksort algorithm, only a single pivot is used to partition the input into two subarrays. Using multiple (more than one) pivots improves the performance of the algorithm because it takes advantage of modern caching, and therefore demonstrates better practical performance.
* Choose the median of the first, middle and last element of the input as a pivot. In the original Quicksort algorithm, the last element of the input sequence in selected as a pivot. But this causes worst-case behaviour on already sorted arrays.

When no information about the ordering of the input sequence is known, then setting the median of the first, middle and last element of the input as a pivot:

* + Prevents the worst-case performance of having a sorted array as an input.
  + Increases the likelihood of the pivot being close to the median of the whole array. The closer the selected pivot is to the median of the input array, the better performance we get. This is because when we divide the input into two parts, the closer to being equal the parts are, the faster the input array can be sorted.

Q2:

(a)

Kruskal’s algorithm is a minimum spanning-tree (MST) algorithm. It works by creating a forest in which edges are added by:

* Finding an edge with the least possible weight.
* Checking that the edge does not form a cycle with the already added edges.

How Kruskal’s algorithm works (step-by-step):

**Sorted edges by weight**

1: (H, I)

2: (D, B)

3: (A, C)

4: (H, B)

5: (J, C)

6: (G, H)

7: (C, D)

9: (J, E)

10: (C, B)

11: (E, F)

12: (A, B)

…

**Subtrees**

(H, I), (A), (B), (C), (D), (E), (F), (G), (J)

(H, I), (D, B), (A), (C), (E), (F), (G), (J)

(H, I), (D, B), (A, C), (E), (F), (G), (J)

(H, I, D, B), (A, C), (E), (F), (G), (J)

(H, I, D, B), (A, C, J), (E), (F), (G)

(H, I, D, B, G), (A, C, J), (E), (F)

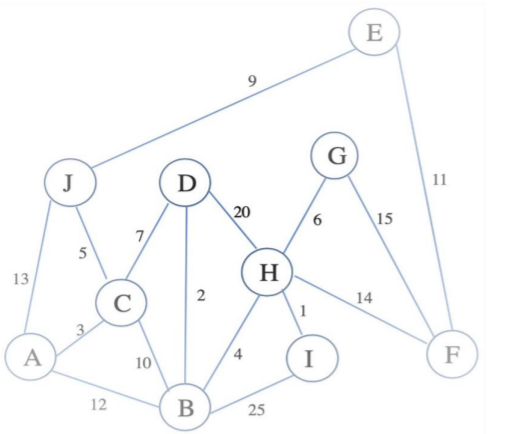
(H, I, D, B, G, A, C, J), (E), (F)

(H, I, D, B, G, A, C, J, E), (F)

REJECTED

(H, I, D, B, G, A, C, J, E, F) DONE

This is how the MST formed by Kruskal’s algorithm will look:



(b)

The problem given is known as 0-1 Knapsack problem.

We are given:

* A maximum allowed weight of **7**.
* Four items that we can choose from (we can choose only one item from each):
  + (weight, value) -> (1, 1), (3, 4), (4, 5), (5, 7)

We need to choose such items that will *maximise* the total value, while not going over the weight limit. (the weight of our items needs to be *less or equal* to 7)

To solve this problem, we can either use the recursive approach or the bottom-up approach. For this problem I will use the bottom-up approach.

The Java code for solving this problem using Dynamic Programming (DP):

public class Solution {  
 //Returns the bigger of the two input integers.  
 static int getMax(int a, int b) {  
 return (a > b) ? a : b;  
 }  
  
 // Returns the maximum value  
 static int calculateMaxValue(int maxWeight, int weights[], int values[], int length)  
 {  
 int table[][] = new int[length + 1][maxWeight + 1];  
  
 // Build the table in a bottom up manner  
 for (int i = 0; i <= length; i++) {  
 for (int j = 0; j <= maxWeight; j++) {  
  
 if (i == 0 || j == 0)  
 table[i][j] = 0;  
  
 else if (weights[i - 1] <= j)  
 table[i][j] = *getMax*(  
 values[i - 1] + table[i - 1][j - weights[i - 1]],  
 table[i - 1][j]);  
 else  
 table[i][j] = table[i - 1][j];  
 }  
 }  
  
 return table[length][maxWeight];  
 }  
  
 public static void main(String args[])  
 {  
 // Gives the input for solving the problem  
 // The input number must be sorted (ascending order)  
 int values[] = new int[] { 1, 4, 5, 7 };  
 int weights[] = new int[] { 1, 3, 4, 5 };  
 int maxWeight = 7;  
  
 // Calculates the maximum value  
 int maxValue = *calculateMaxValue*(maxWeight, weights, values, values.length);  
  
 // Displays the maximum value  
 System.*out*.println("Max value: " + maxValue);  
 }  
}

The output of the following code is: **9**

Clearly the output is correct because we can get the maximum value by choosing the second (3, 4) and the third (4, 5) element.

How the solution works:

We have two arrays – *values*, containing the values, and *weights*, containing the weights. The first element of the *weights* array has its value stored in the first element of the *values* array. This is valid for all the other elements in the arrays. (the second element of the *weights* array corresponds to the second element of the *values* array and so on).

In order to solve this problem using Dynamic Programming we will create a temporary array which will hold all possible *maximum values*. This is done so we can avoid recomputations of the same subproblems, thus making our algorithm better timewise. (rather than using a basic recursive function).

We create a two-dimensional array in which the *columns* are all the possible weights and the rows are the weights given to us in our problem.

The table containing our solution looks like this:

Total weight

0 1 2 3 4 5 6 7

0

1

3

4

5

0 0 0 0 0 0 0 0

0 1 1 1 *1* 1 1 1

Given weights

0 1 1 4 **5** 5 5 5

0 1 1 4 5 6 6 9

0 1 1 4 5 7 8 9

For a chosen *value* (table[i][j]), it will denote the maximum valuefor a given weight (j) using only the values until the i-th element. (i-th element included).

To explain this we will use an example: table[2][4] -> the element that we have chosen is element “**5**” (it is denoted in BOLD in the example). What this element shows is the *maximum value* that we can get using the first two given weights - (1, 1), (3, 4) – while maintaining a maximum weight of no more than 4. The *maximum value* is calculated by choosing the bigger value of the two:

* table[i-1][j] which is equal to 1
* *values*[i] + table[i-1, j-i] which is equal to 4 + 1 = 5

So, the formula for populating the table from the “bottom-up” can be condensed to the following:

**Max (table[i-1][j] OR *values*[i] + table[i-1, j-i])**

Note: if, when calculating the bigger of the two elements, we get an *undefined* value, we just choose the defined value. It is guaranteed that at least one of the value will be defined.

Note: when constructing the table, if *i* or *j* are equal to zero, we set the corresponding value in the table to zero. This is done so the formula can function properly.

When the table is constructed and populated, we just return the final element. This element is the maximum value that we can obtain, given our maximum weight and items that we can choose from. Using this “bottom-up” method we get a time complexity of O(N\*W) where ‘N’ is the number of items we can choose from and ‘W’ is the maximum weight. We also use some space for constructing the array, meaning that our algorithm is not in-place. (O(N\*W) auxiliary space used for construing the two-dimensional array).