SCHOOL OF COMPUTING
Faculty of Engineering

SEMESTER 1

SESSION 2020/2021

SECJ2013

ASSIGNMENT 5

PREPARED BY :

1. NUR ALEEYA SYAKILA BINTI MUHAMAD SUBIAN (A19EC0127)
2. NUR HADIRAH MUNAWARAH BINTI ROZMIZAN (A19EC0201)

LECTURER'S NAME: DR. RUHAIDAH BINTI SAMSUDIN
SECTION : 02
SUBMITTED ON : 23/11/2020

Exercise 2 :

a) Bubble sort is comparing adjacent elements and swapping if they out of order.
Selection sort is selecting largest elements and swapping with the last element if they out of order.

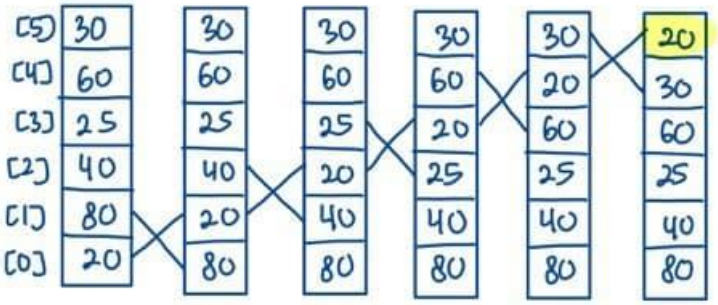
b) Descending order

[0]	[1]	[2]	[3]	[4]	[5]
20	80	40	25	60	30

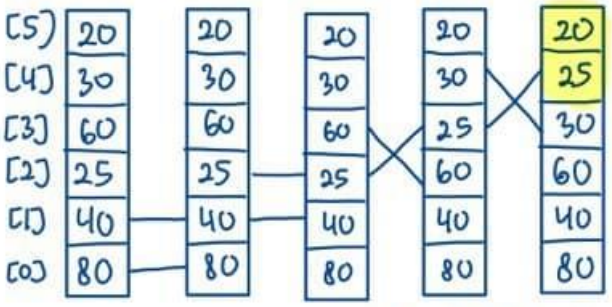
Array D

Bubble Sort

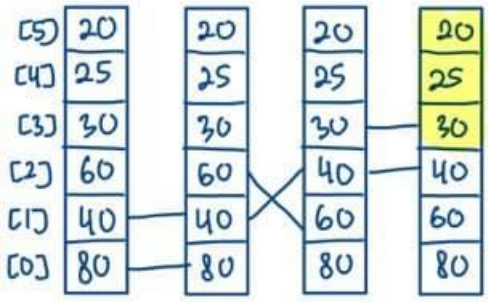
The number of comparisons to sort data in this list
$(6-1)+(6-2)+(6-3)+(6-4)+(6-5) = 15$



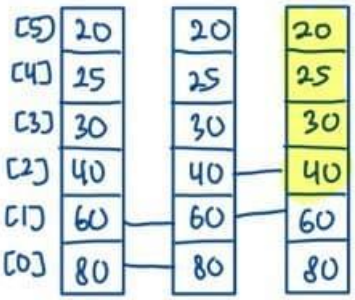
Pass 1 : Comparison $(6-1) = 5$



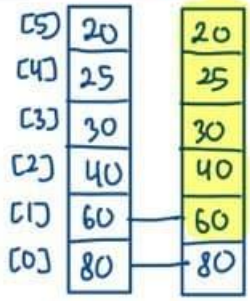
pass 2 : Comparison $(6-2) = 4$



Pass 3 : Comparison $(6-3) = 3$



Pass 4 : Comparison $(6-4) = 2$



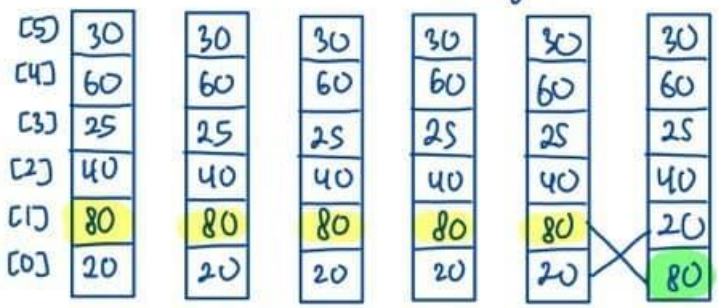
Pass 5 : Comparison $(6-5) = 1$

number of swaps = 8

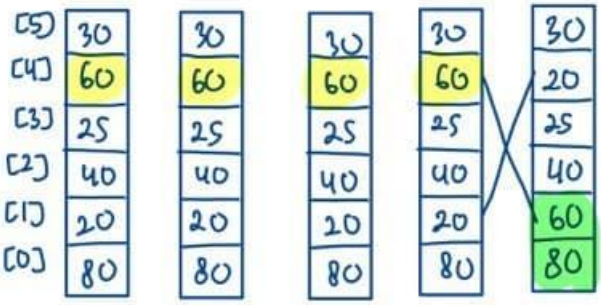
Selection Sort

[0]	[1]	[2]	[3]	[4]	[5]
20	80	40	25	60	30

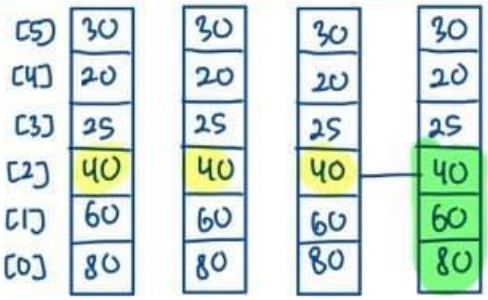
Array D



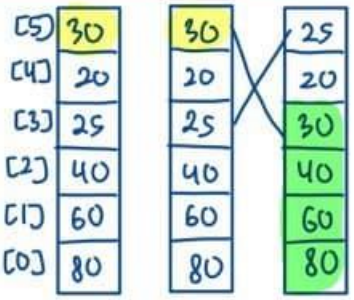
pass = 1 , p = 1, 2, 3, 4, 5



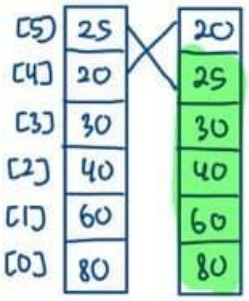
pass = 2 , p = 1, 2, 3, 4



pass 3 , p = 1, 2, 3



pass 4 , p = 1, 2



pass 5 = 1, 2

number of swaps = 4

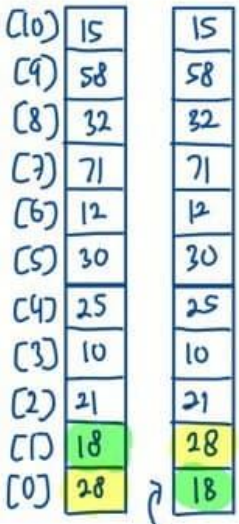
5)

Insertion Sort

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
28	18	21	10	25	30	12	71	32	58	15

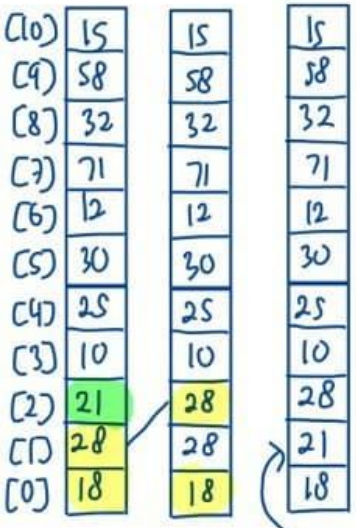
Array E

First 5 phases



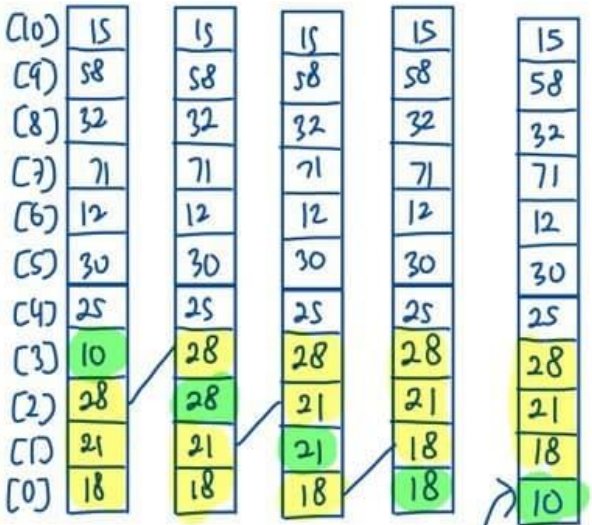
index 1
item 18

pass 1



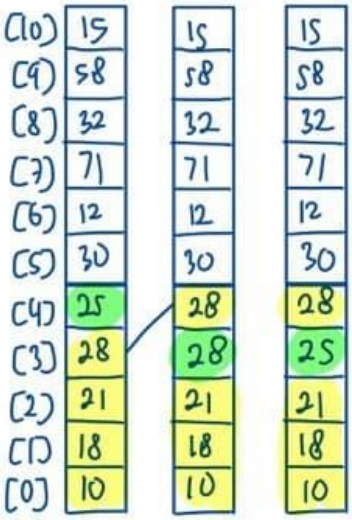
index 2 1 0
item 21

pass 2



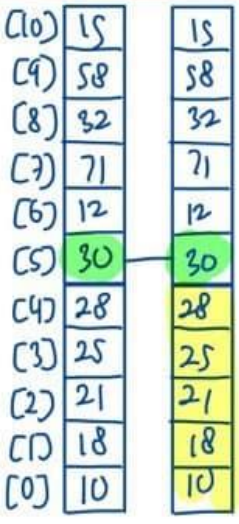
index 3 2 1 0
item 10

pass 3



index 4 3 3
item 25

pass 4



index 5

item 30

pass 5

Exercise 4

a) Ascending order

b) Bubble Sort

[0]	[1]	[2]	[3]
2	4	8	24

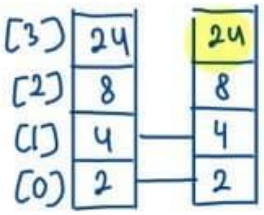
Array L

number of comparison

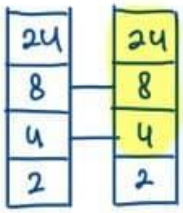
(4-3)+(4-2)+(4-1) = 6

number of pass = 3

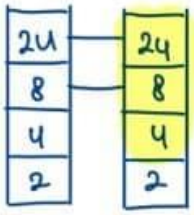
pass = 1



pass = 2



pass = 3



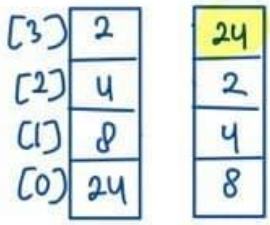
[0]	[1]	[2]	[3]
24	8	4	2

Array M

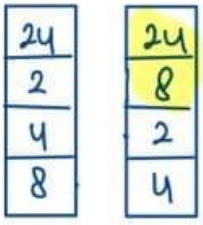
number of comparison
(4-3)+(4-2)+(4-1) = 6

number of pass = 3

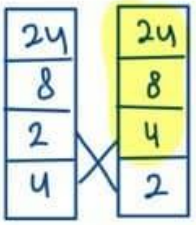
pass = 1



pass = 2



pass = 3



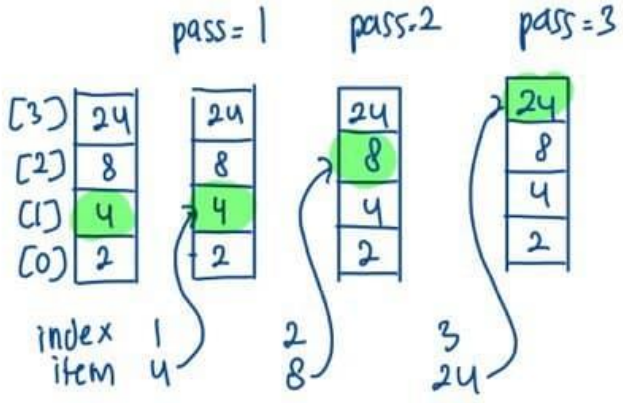
c) **Insertion Sort**

[0]	[1]	[2]	[3]
2	4	8	24

Array L

number of comparison = 1

number of pass = 3

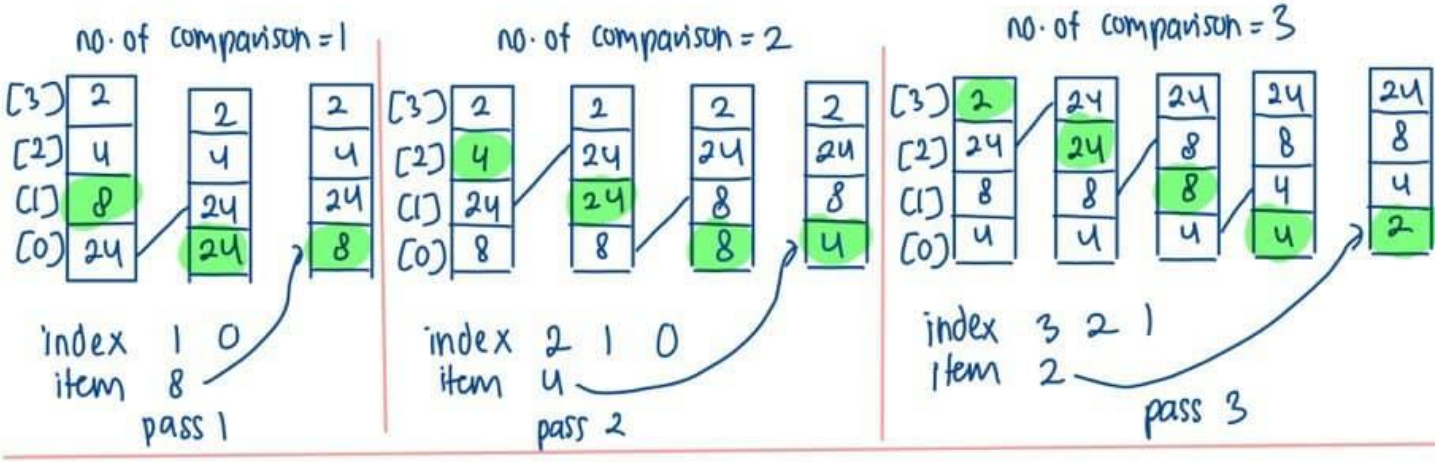


[0]	[1]	[2]	[3]
24	8	4	2

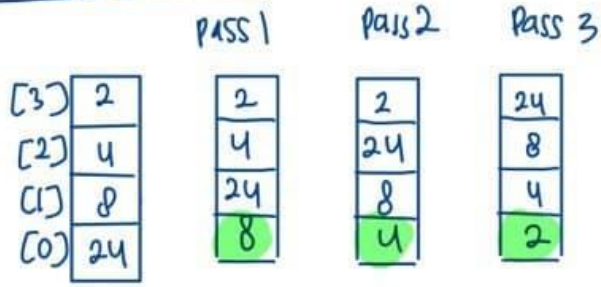
Array M

number of pass = 3

working



Final Answer



d)

Sorting Array L

	Bubblesort C)	insert Sort C)
Number of pass	3	3
Number of comparison	6	1
Type of case	best case	best case
Algorithm complexity number of comparison	$O(n^2)$	$O(n)$
Algorithm complexity number of swaps	0	0

∴ insertion sort is more efficient than bubble sort of Array L because algorithm complexity of number of comparison for insertion sort is $O(n)$ meanwhile bubble sort is $O(n^2)$.

Sorting Array M

	Bubblesort C)	Sort C)
Number of pass	3	3
Number of comparison	6	Pass 1 = 1 pass 2 = 1 Pass 3 = 1
Type of case	worst case	worst case
Algorithm complexity number of comparison	$O(n^2)$	$O(n^2)$
Algorithm complexity number of swaps	$O(n^2)$	$O(n^2)$

∴ Both insertion sort and bubble sort have same Algorithm efficiency $O(n^2)$ in Array M.

Exercise 5

1. | 3 | 6 | 7 | 8 | 9 | 10 |

	Improved Bubble Sort	Total Comparison	Total swap
After pass 1	3 6 7 8 9 10	5	0
After pass 2		-	-
After pass 3		-	-

	Insertion Sort	Total Comparison	Total swap
After pass 1	3 6 7 8 9 10	1	0
After pass 2	3 6 7 8 9 10	1	0
After pass 3	3 6 7 8 9 10	1	0
After pass 4	3 6 7 8 9 10	1	0
After pass 5	3 6 7 8 9 10	1	0

	Selection Sort	Total Comparison	Total swap
After pass 1	3 6 7 8 9 10	5	1
After pass 2	3 6 7 8 9 10	4	1
After pass 3	3 6 7 8 9 10	3	1
After pass 4	3 6 7 8 9 10	2	1
After pass 5	3 6 7 8 9 10	1	1

2.

Best Case	Improved Bubble Sort	Insertion Sort	Selection Sort
Algorithm efficiency on number of comparisons	O(n)	$O(n^2)$	$O(n^2)$
Algorithm efficiency of number of swaps	0	0	O(n)

Algorithm that has the worst performance is **selection sort**

Algorithm that has better performance is **improved bubble sort**

3. Selection sort can be improved in sorting array x by :

    1. Putting a condition statement as follows:

```
If (largetIndex !=last)
      Swap(Data[largestIndex],Data[last]);
```

    2. Follow the step below :

        1. Initialize i to 1
        2. Search the beginning of the unsorted part of the list to the end
        3. Exxx the locations of all values that are the same as the max value
        4. Use the indices on the queue to perform swapping
        5. Repeat step 3-5 until i equals to n

Exercise 6

1.

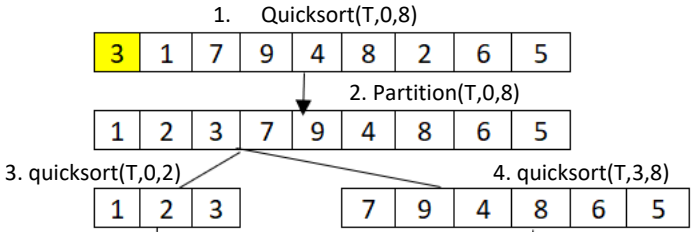
```
While(T[top]>pivot)
        {top-- ;}
```

To find smaller value than pivot from top array

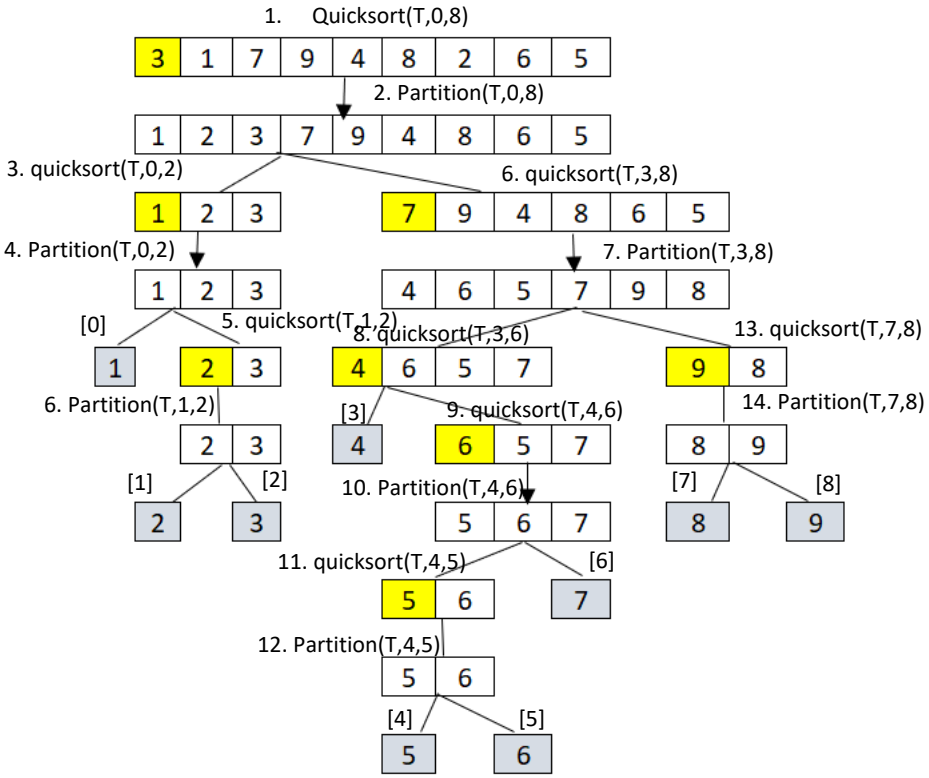
```
While(T[bottom]<pivot)
        {bottom++ ;}
```

To find larger value than pivot from bottom array

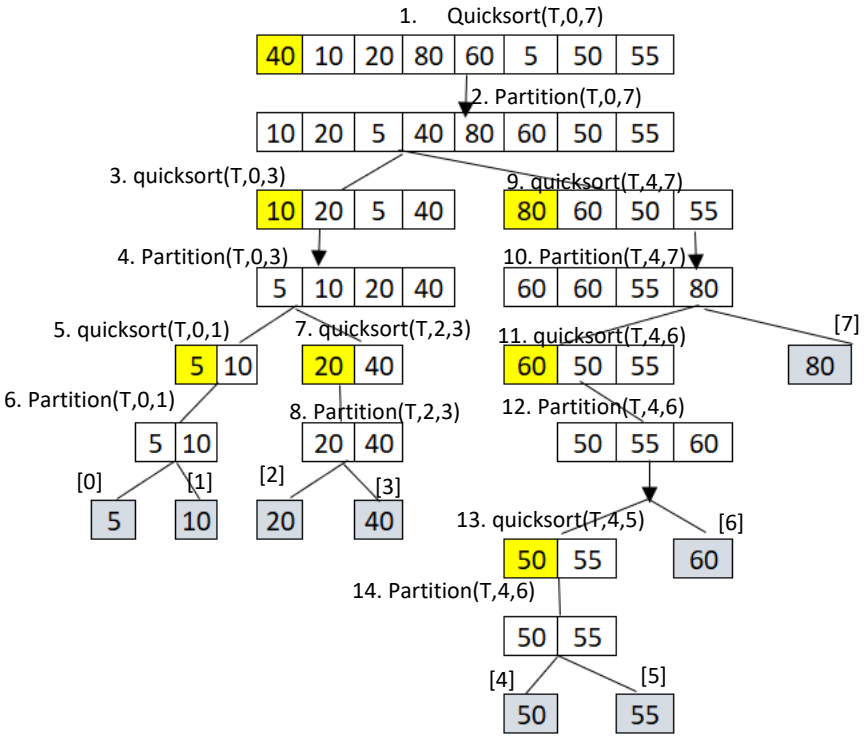
2.



3.



4.

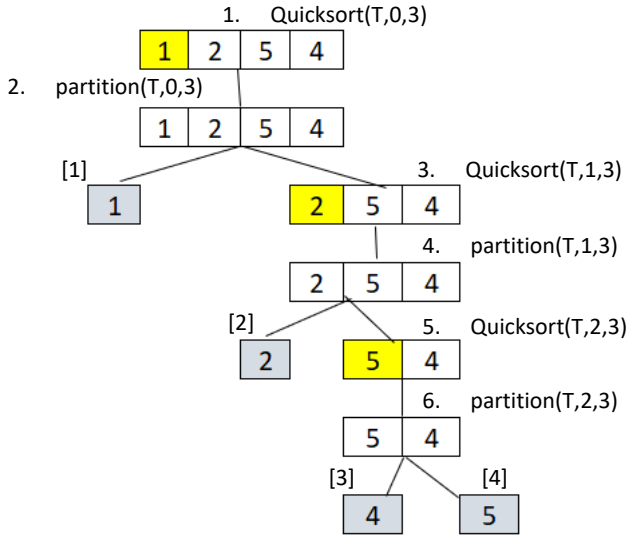


Input data T is a balance case(best case)

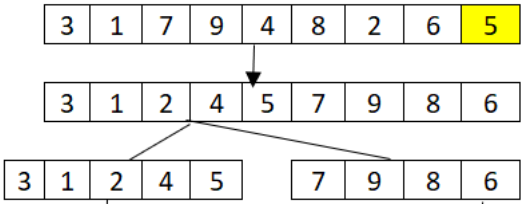
5. This is because the list of input data T is balance segment. The pivot chosen is right which can put other items in balance situation.

6. Worst case for quick sort is when smallest item or the largest items always be chosen as pivot and causing left position and right position not balance.

   Example : T [1 2 5 4]



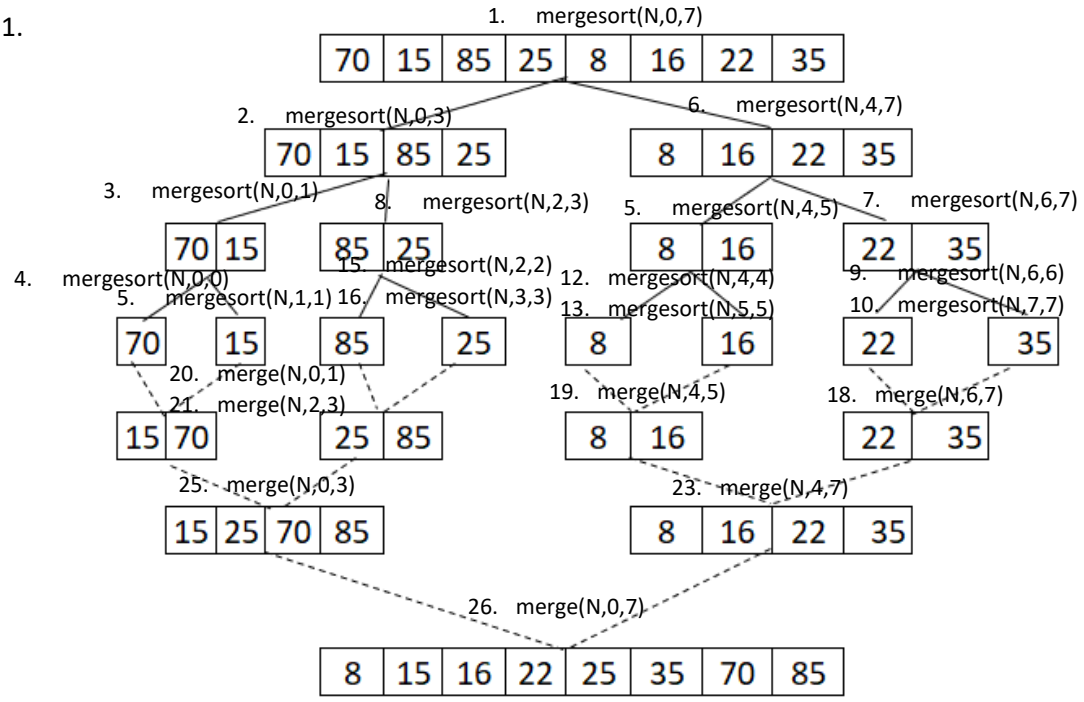
7. Pivot=T[8]



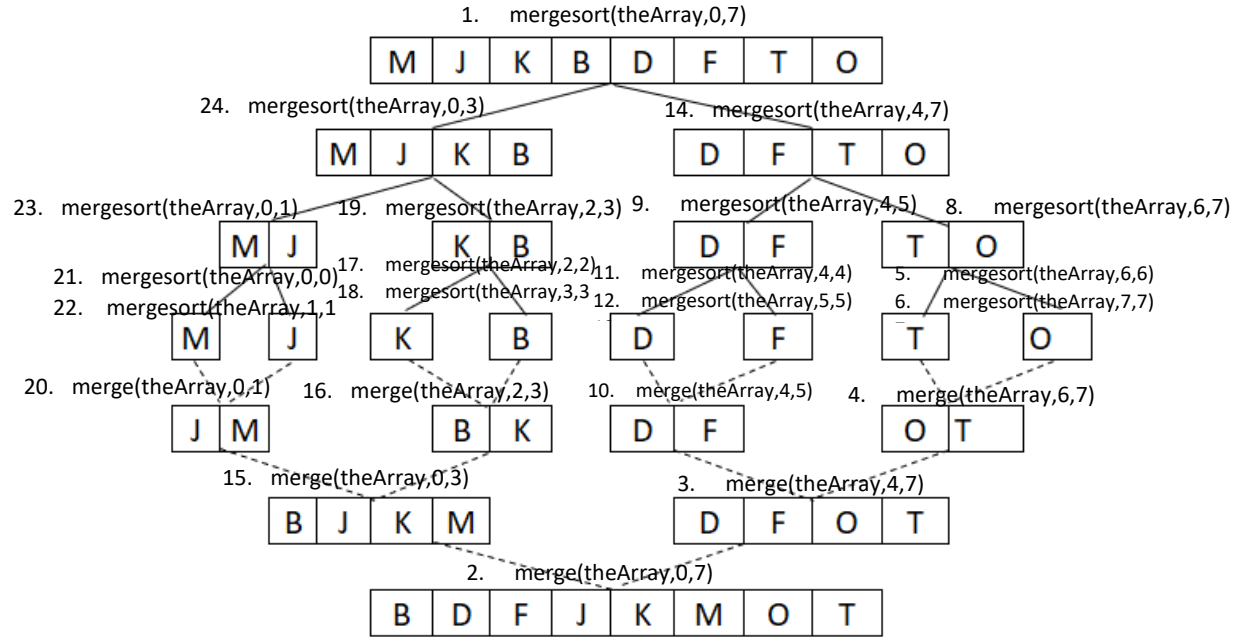
8. When use the value 5 (index 8) as pivot, it balances the left and right partition. The value of 5 is the best choice as a pivot since it is the middle element of the array. The efficiency o quick sort depends on pivot value.

Exercise 7

1.



2.



3.

					tempArray							
First1	Last1	First2	Last2	index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	3	4	7	0	B							
1	3	4	7	1	B	D						
1	3	5	7	2	B	D	F					
1	3	6	7	3	B	D	F	J				
2	3	6	7	4	B	D	F	J	K			
3	3	6	7	5	B	D	F	J	K	M		
3	3	6	7	6	B	D	F	J	K	M	O	
3	3	7	7	7	B	D	F	J	K	M	O	T

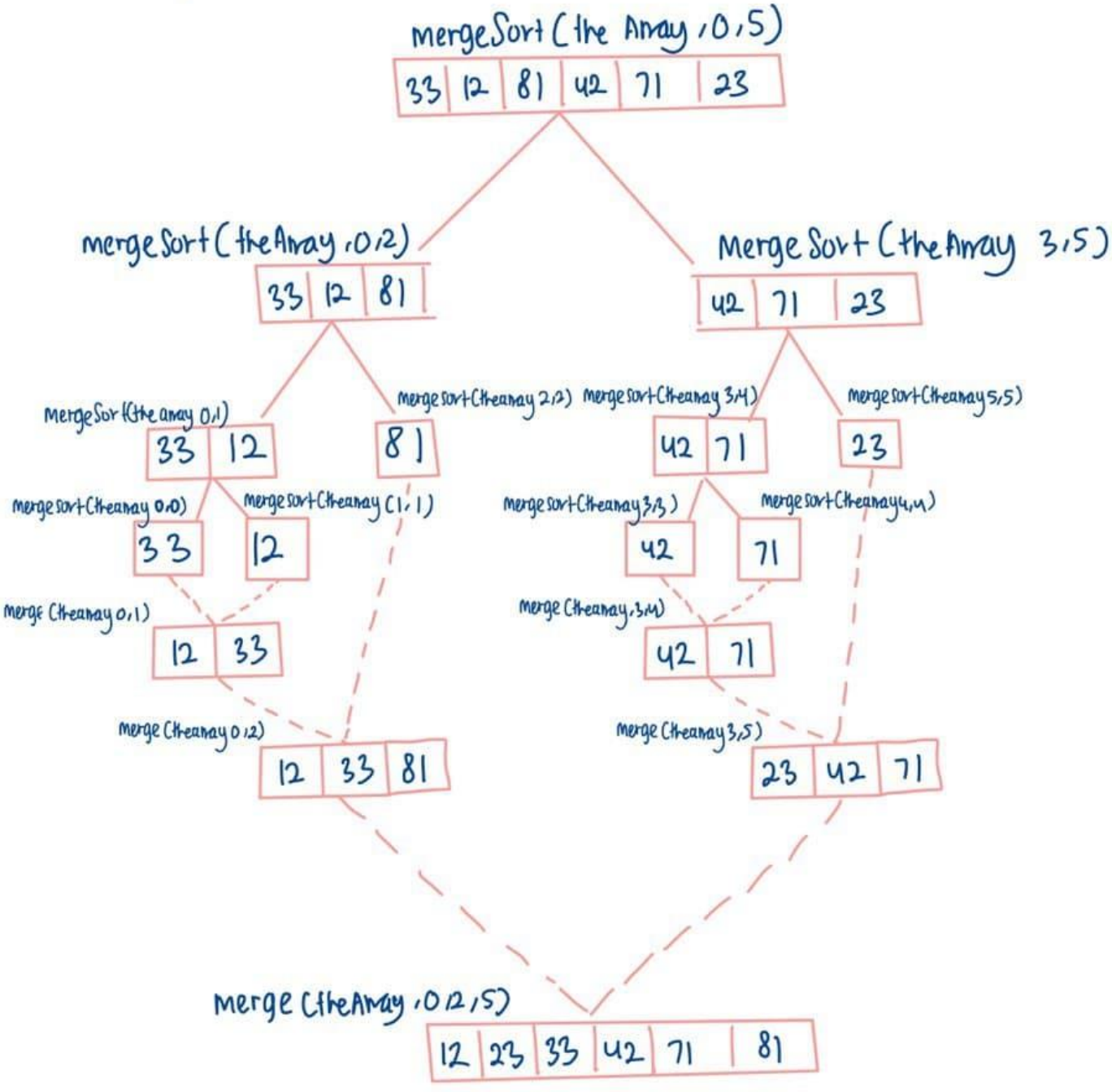
Exercise 8

[0]	[1]	[2]	[3]	[4]	[5]
33	12	81	42	71	23

Array T

1. Merge-Sort

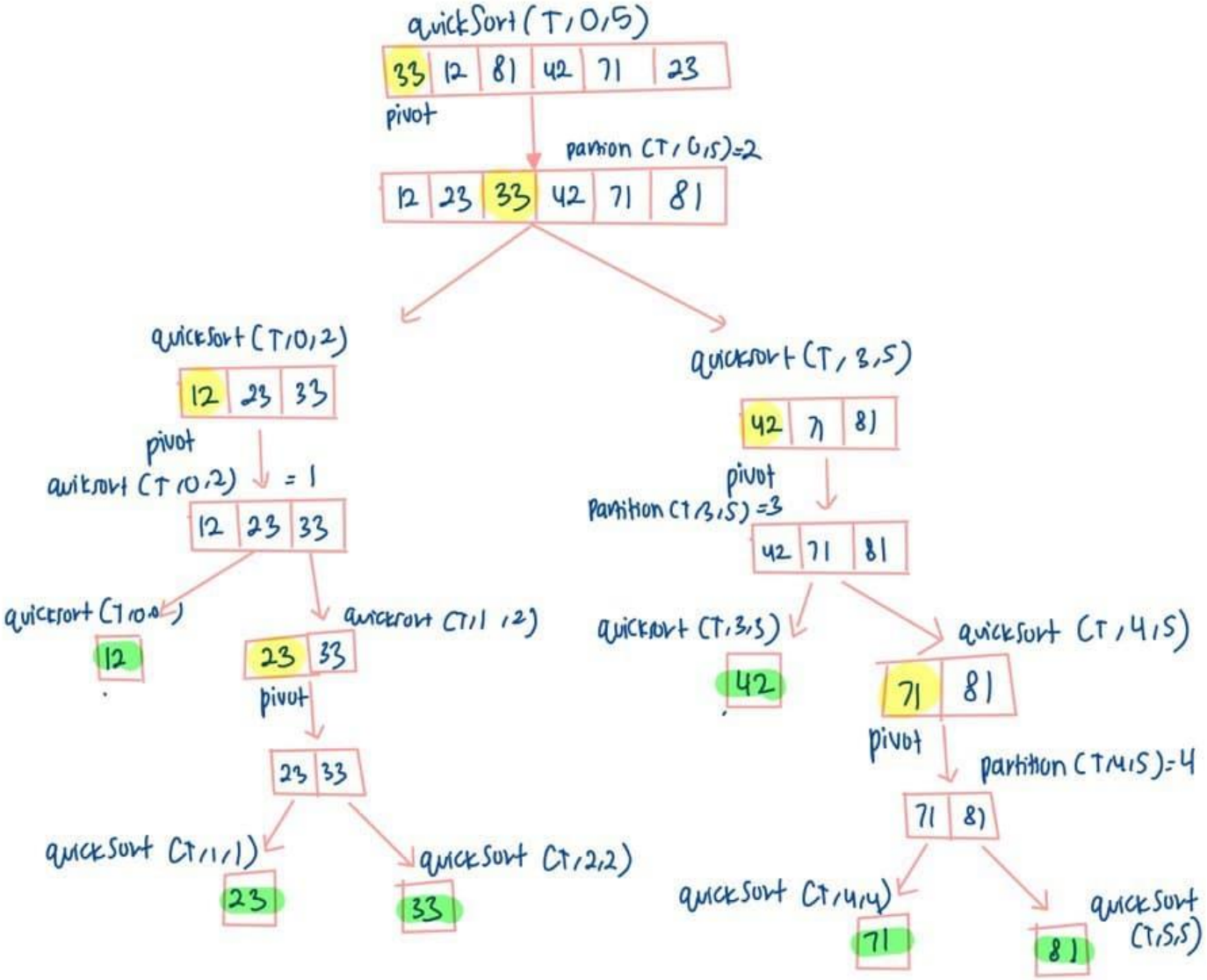
Ascending



2. Quick sort

[0]	[1]	[2]	[3]	[4]	[5]
33	12	81	42	71	23

Array T



3. Efficiency of quick sort for the average case : $O(n \log_2 n)$
Efficiency of merge sort for the average case : $O(n \log_2 n)$

However, quick sort is more efficient due to it works well on smaller array