22nd SCSE – Past Year Paper Solution (2022 – 2023 Semester 1)
CE/CZ 2101 – Algorithm Design and Analysis

1 (a) (i) 18 key comparisons

(ii) 14 swaps

Editor's note:

Iteration	Initial Array	No. of Key Comps	No. of Swaps
1	[7, **2**, 5, 6, 3, 1, 4]	1 comp (2<=7 → swap)	1
2	[2, 7, **5**, 6, 3, 1, 4]	2 comps (5<=7 → swap 5>=2 → no swap)	1
3	[2, 5, 7, **6**, 3, 1, 4]	2 comps (6<=7 → swap 6>=5 → no swap)	1
4	[2, 5, 6, 7, **3**, 1, 4]	4 comps (3<=7 → swap 3<=6 → swap 3<=5 → swap 3>=2 → no swap)	3
5	[2, 3, 5, 6, 7, **1**, 4]	5 comps (1<=7 → swap 1<=6 → swap 1<=5 → swap 1<=3 → swap 1<=2 → swap)	5
6	[1, 2, 3, 5, 6, 7, **4**]	4 comps (4<=7 → swap 4<=6 → swap 4<=5 → swap 4>=3 → no swap)	3
-	**[1, 2, 3, 4, 5, 6, 7]**	**Total: 1+2+2+4+5+4 = 18**	**Total: 1+1+1+3+5+3 = 14**

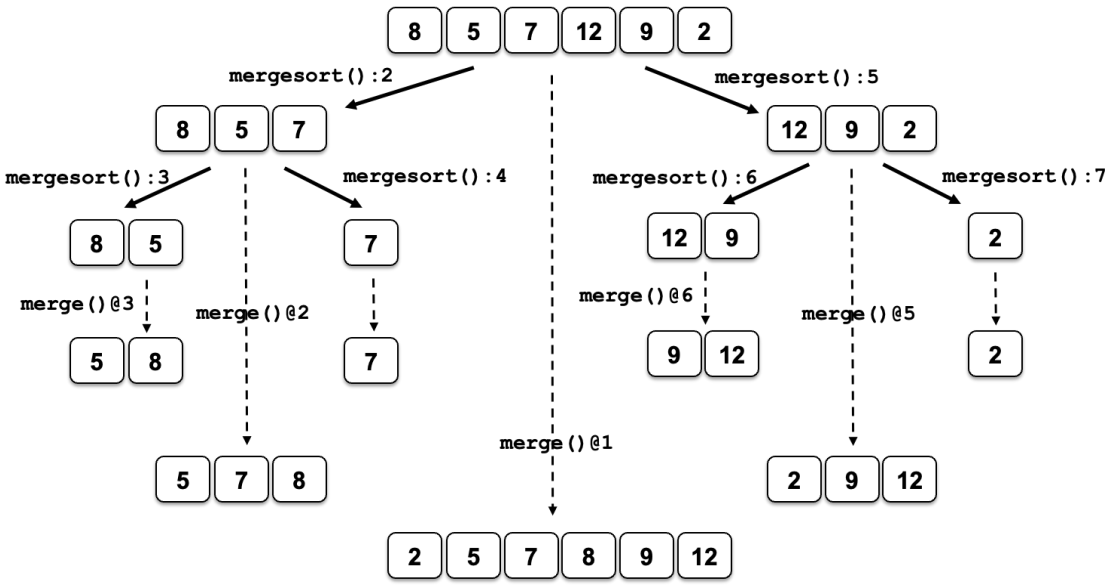
(iii) 6 key comparisons

Editor's note:
The question is asking for the **Best Case** arrangement which is a sorted array. The array is [1, 2, 3, 4, 5, 6, 7] and the number of key comparisons is n-1 where n is the number of elements in the array. Hence 7-1 = 6.

(iv) 21 key comparisons

Editor's note:
The question is asking for the **Worst Case** arrangement which is a reverse sorted array. The array is [7, 6, 5, 4, 3, 2, 1] and the number of key comparisons is $\frac{n(n-1)}{2}$ where n is the number of elements in the array. Hence 42 / 2 = 21.

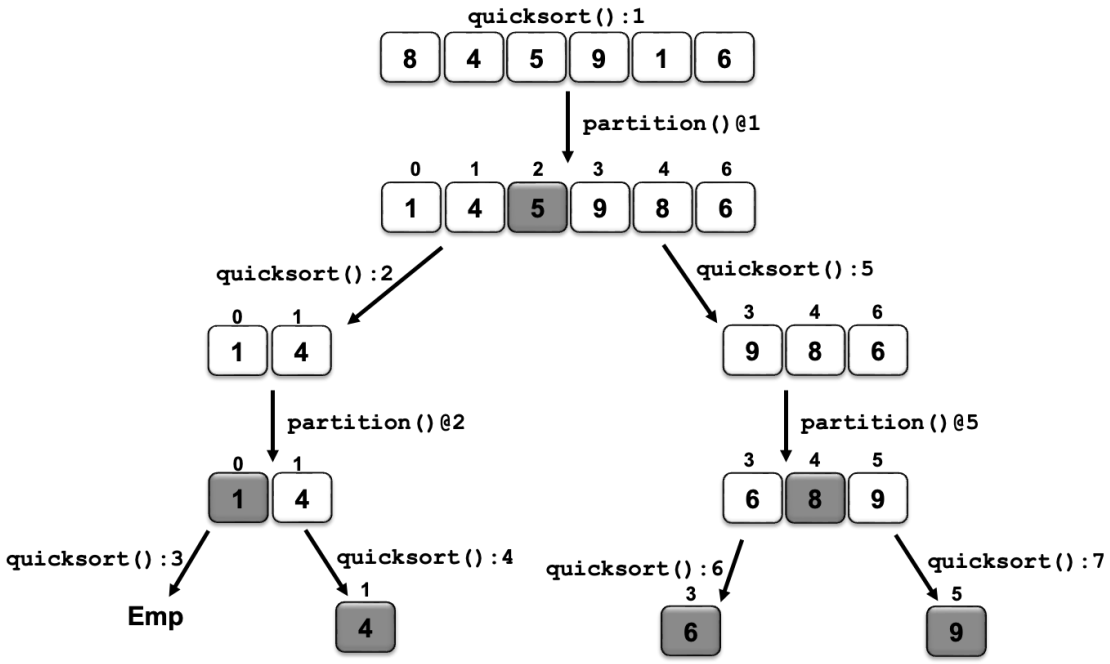
(b) **(i)**



(ii) n - 1 key comparisons

Editor's note:
Worst case for merge function.

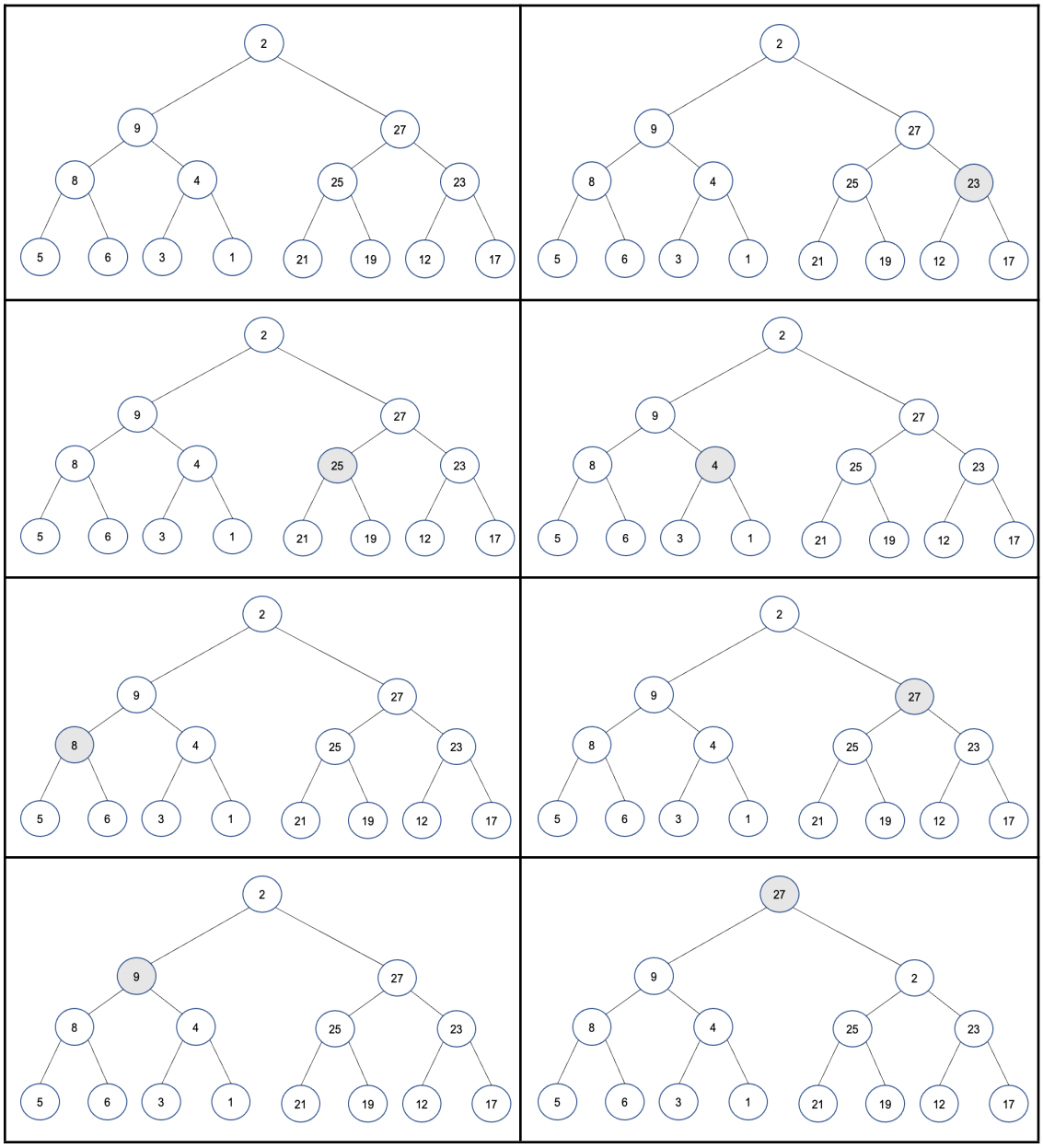
(c)

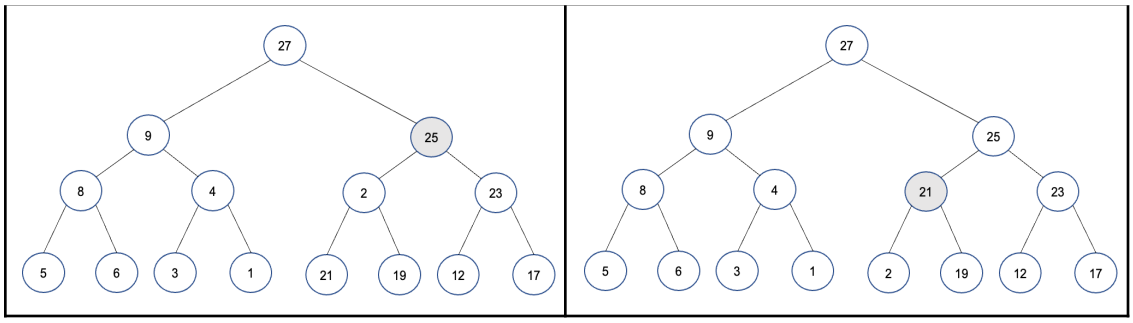


Editor's note:
Some students get confused with **partition()@1**. Here are the details to it:
[8, 4, **5**, 9, 1, 6]
Pivot is 5 at index 2 and will swap with 8 since it will be placed at index 0 of the array.
[5, 4, 8, 9, 1, 6]

We iterate through the elements to the right of the pivot and compare them with the pivot value. Eventually we derive this result:
[5, 4, 1, 9, 8, 6]
Now the pivot element has to return back to its original index position in the array. It swaps with 1 which is at index 2.
[1, 4, 5, 9, 8, 6]

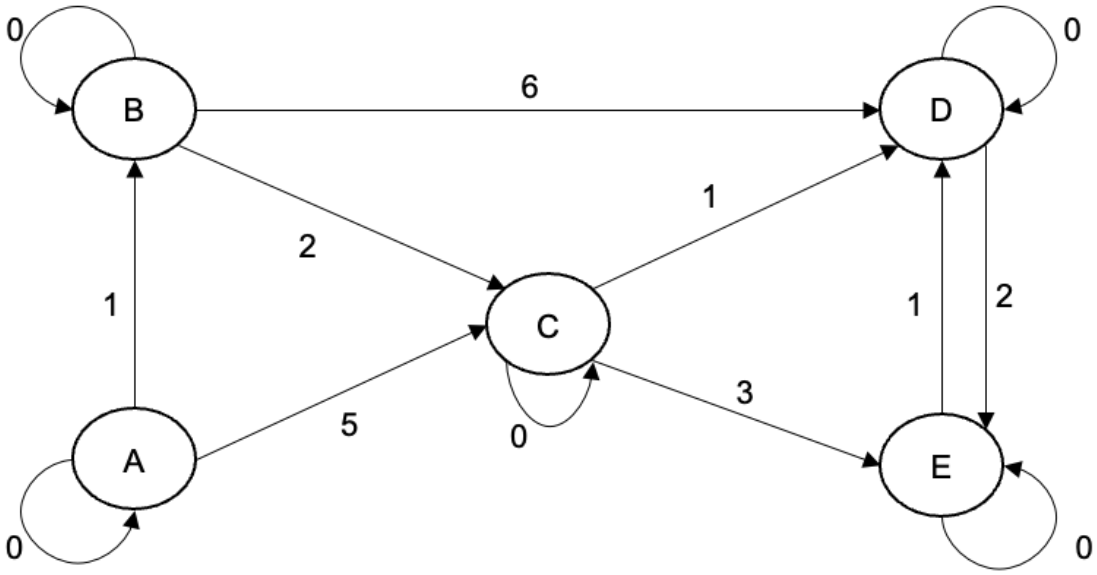
(d)





Editor's note:
Typically, fixHeap starts at the last non-leaf node and we work backwards till we reach the root.

2 (a) (i)



Editor's note:
Self-loops are necessary as there is no mention of a simple graph.

(ii) **Array S:** the set of vertices whose shortest paths from the source node have already been determined.
Array d: an array of size |V| that stores the estimated lengths of shortest paths from the source node to all vertices
Array pi: an array of size |V| to store the predecessors for each vertex

Initialisation

Array S:

A	B	C	D	E
0	0	0	0	0

Array d:

A	B	C	D	E
0	∞	∞	∞	∞

Array pi:

A	B	C	D	E
null	null	null	null	null

After 1st Iteration

Array S:

A	B	C	D	E
1	0	0	0	0

Array d:

A	B	C	D	E
0	1	5	∞	∞

Array pi:

A	B	C	D	E
null	A	A	null	null

After 2nd Iteration

Array S:

A	B	C	D	E
1	1	0	0	0

Array d:

A	B	C	D	E
0	1	3	6	∞

Array pi:

A	B	C	D	E
null	A	B	B	null

After 3rd Iteration

Array S:

A	B	C	D	E
1	1	1	0	0

Array d:

A	B	C	D	E
0	1	3	4	6

Array pi:

A	B	C	D	E
null	A	B	C	C

After 4th Iteration

Array S:

A	B	C	D	E
1	1	1	1	0

Array d:

A	B	C	D	E
0	1	3	4	6

Array pi:

A	B	C	D	E
null	A	B	C	C

Editor's note:
The shortest path A->B->C->D->E has a cost of 6 which is equal to the cost of path A->B->C->E which was derived first. As such the predecessor of E remains as C.

After 5th Iteration

Array S:

A	B	C	D	E
1	1	1	1	1

Array d:

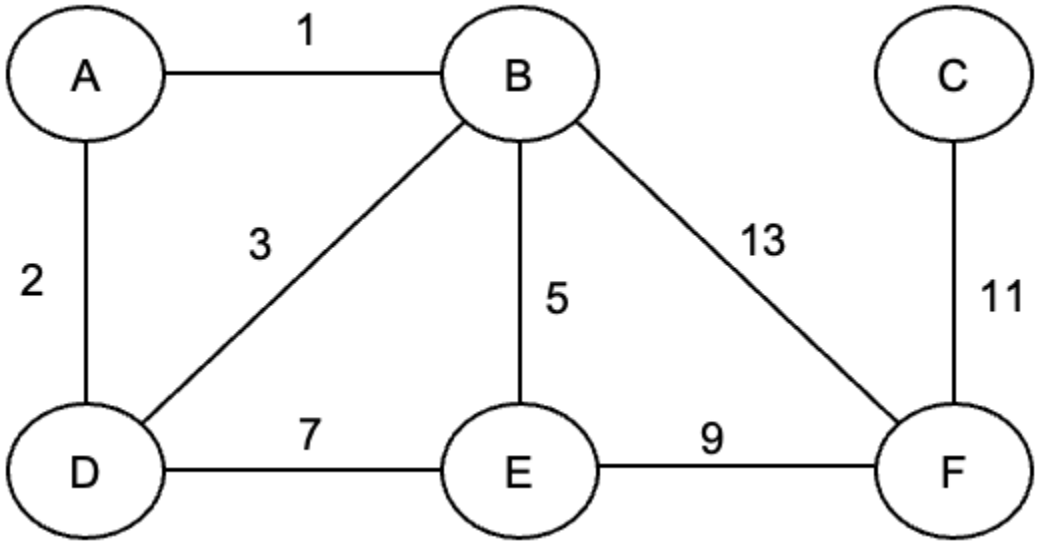
A	B	C	D	E
0	1	3	4	6

Array pi:

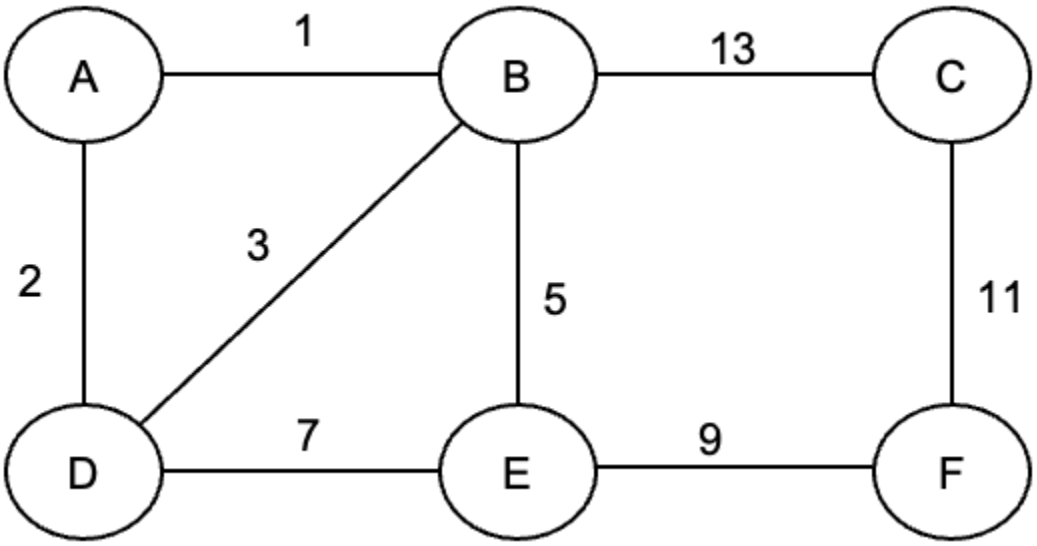
A	B	C	D	E
null	A	B	C	C

(b) Let T be a spanning tree of G, where G is a connected, weighted graph. Suppose that for every edge that is present in the graph G but not in the spanning tree T, if that edge is being added to T, it forms a cycle such that the edge that was added is the maximum-weight edge in that cycle (i.e to say that the edge that was added has the largest weight value than all other edges in that cycle)

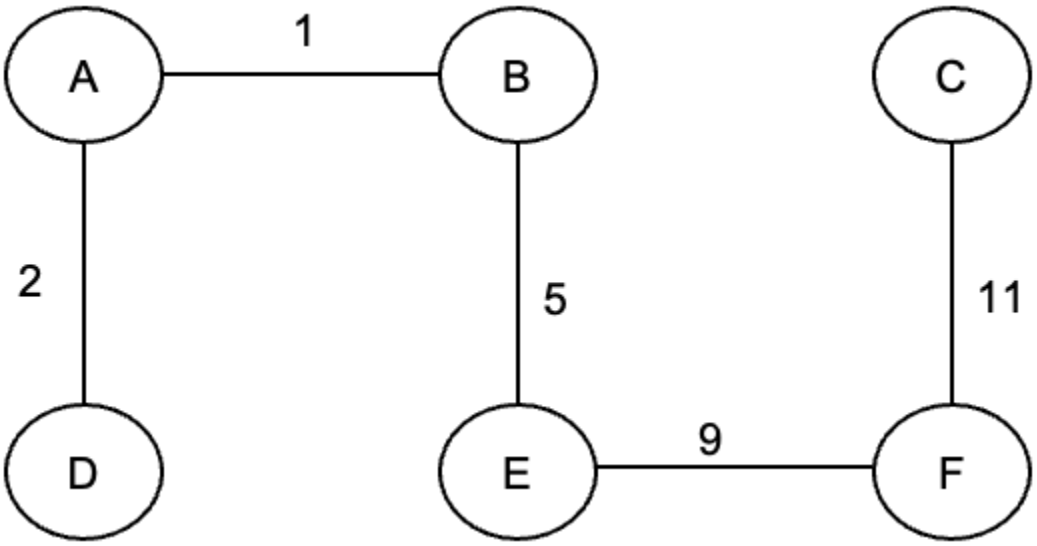
(c) **Graph G:**



OR

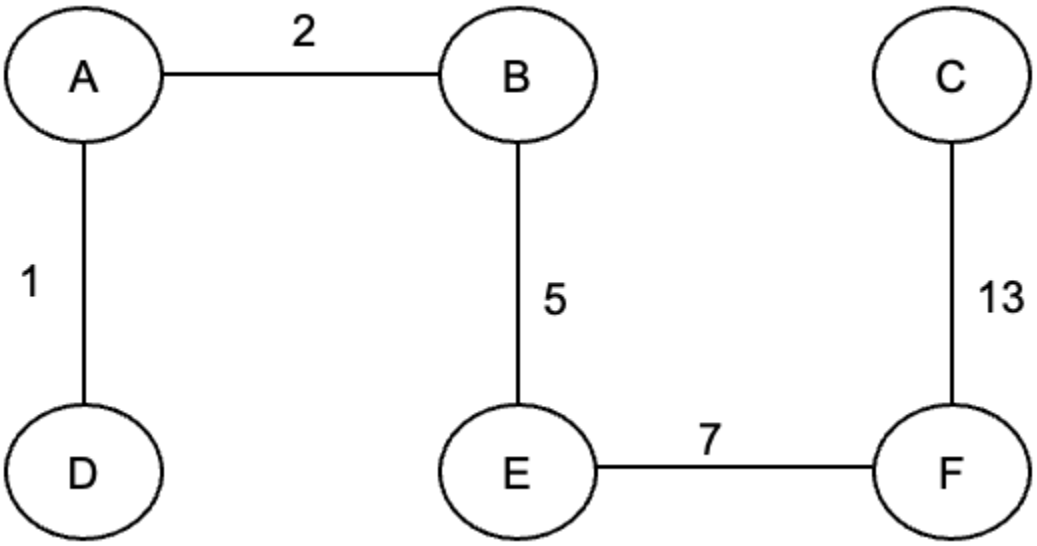


Minimal Spanning Tree T (for both answers above):

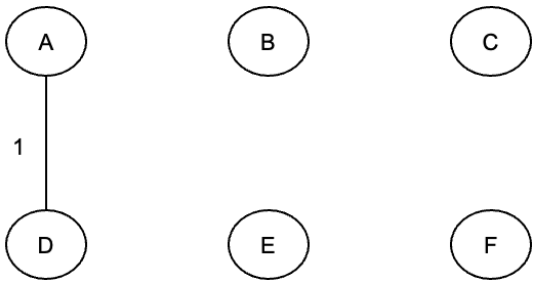
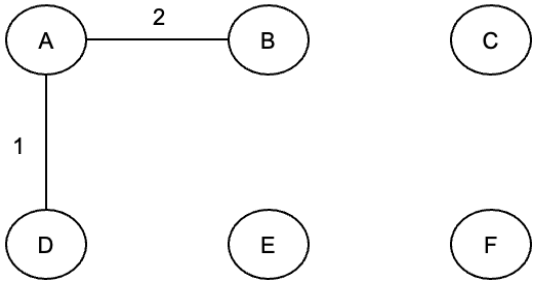


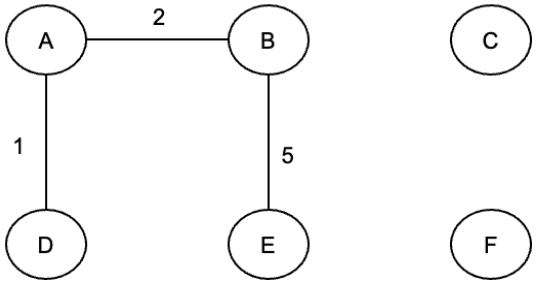
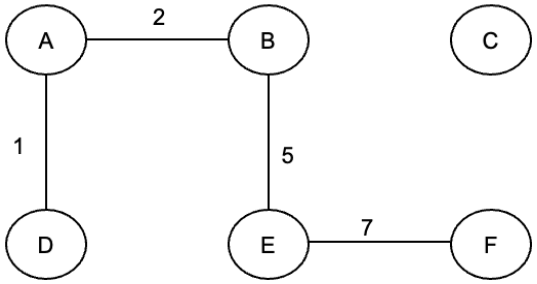
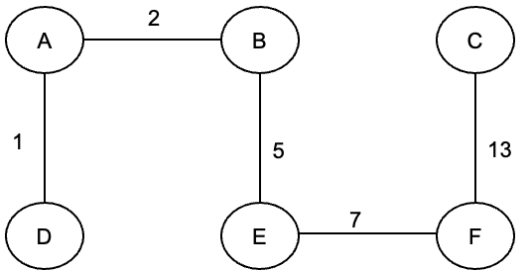
Editor's note:
The approach to such questions is to have the MST property in mind. This along with the given constraints will aid immensely in deriving the answer.

(d)



Editor's note:
Kruskal's algorithm is essentially as such: from the set of all edges of the graph, pick the minimum-weight edge. If adding that edge to the MST does not form a cycle, then add that edge to the MST and repeat with the remaining edges.

E = {~~1~~, 2, 3, 5, 7, 8, 9, 13}	
E = {~~1~~, ~~2~~, 3, 5, 7, 8, 9, 13}	
E = {~~1~~, ~~2~~, ~~3~~, 5, 7, 8, 9, 13}	**Adding edge of weight 3 forms a cycle**

E = {~~1~~, ~~2~~, ~~3~~, ~~5~~, 7, 8, 9, 13}	
E = {~~1~~, ~~2~~, ~~3~~, ~~5~~, ~~7~~, 8, 9, 13}	
E = {~~1~~, ~~2~~, ~~3~~, ~~5~~, ~~7~~, ~~8~~, 9, 13}	**Adding edge of weight 8 forms a cycle**
E = {~~1~~, ~~2~~, ~~3~~, ~~5~~, ~~7~~, ~~8~~, ~~9~~, 13}	**Adding edge of weight 9 forms a cycle**
E = {~~1~~, ~~2~~, ~~3~~, ~~5~~, ~~7~~, ~~8~~, ~~9~~, ~~13~~}	

3 (a) (i) Typically, its rather time consuming to keep checking for the tightest bound when using the substitution method.

Thus, an approach would be to solve it first using other methods (if possible) before proving.

We can attempt to solve the recurrence equation using characteristic equation with some manipulation. Observe that the given equation is **NOT** a homogenous equation.

$W(n) = 2W(n-1) + 3 \ldots (eqn\ 1)$
$W(n-1) = 2W(n-2) + 3 \ldots (eqn\ 2)$
$W(n) - W(n-1) = 2W(n-1) - 2W(n-2) + 3 - 3$
$W(n) = 3W(n-1) - 2W(n-2)$

After some manipulation we have obtained a linear and homogenous equation. We can now proceed to solve it using a characteristic equation.

$Characteristic\ eqn$:
$x^2 - 3x + 2 = 0$
$(x - 2)(x - 1) = 0$

The characteristic roots are $x = 2$ and $x = 1$, each of multiplicity 1.
Therefore, the Recurrence equation is:
$W(n) = a(2)^n + b(1)^n\ where\ a\ and\ b\ are\ constants$
$W(n) = a(2)^n + b\ (simplified)$

Note that we have 2 unknowns to solve and 2 equations. However, the question only gives us 1 initial condition.

$W(n) = 2W(n - 1) + 3$ (Given equation)
$W(1) = 1$ … $(eqn\ 1)$
$W(2) = 2W(1) + 3$ … $(eqn\ 2)$

Sub eqn 1 into eqn 2:

$W(2) = 2 + 3 = 5$

With 2 initial conditions we can now solve for the unknown constants:

$W(1) = 1 = a(2)^1 + b$ … $(eqn\ 1)$
$W(2) = 5 = a(2)^2 + b$ … $(eqn\ 2)$
$2a + b = 1$ … $(eqn\ 1)$
$4a + b = 5$ … $(eqn\ 2)$

eqn 2 – eqn 1

$2a = 4$
$a = 2$

Sub a = 2 into eqn 2

$b = 5 - 8 = -3$

Therefore,

$W(n) = 2^{n+1} - 3$

Now we can make a pretty accurate guess that: $W(n) = O(2^n)$

Subsituition method

Guess that $W(n) = O(2^n)$

Proof: We will prove that $W(n) \leq 2 * (2^n) - 3$

(1) Base Case: $W(1) = 1 \leq 2 * (2) - 3 = 1$

(2) Inductive step: assume that $W(k) \leq 2 * 2^k - 3$, prove that $W(k+1) \leq 2 * 2^{k+1} - 3$.

$W(k+1) = 2W(k) + 3$ (From given equation)

$\leq 2(2 * (2^k) - 3) + 3$ (Based on our assumption)

$\leq 2 * 2^{k+1} - 6 + 3$

$\leq 2 * 2^{k+1} - 3$ (Proven)

Thus $W(n) = O(2 * (2^n) - 3) = O(2^n)$

(ii) $W(n) = W(\frac{n}{3}) + lgn$

$a = 1, b = 3$ by comparing with the Master Method formula $W(n) = aW(\frac{n}{b}) + f(n)$

$f(n) = lgn$

$n^{log_b a} = n^{log_3 1} = n^0 = 1$

$f(n) = \theta(lgn) = \theta(1 * lgn) = \theta(n^{log_b a} lg^1 n)$ where $k = 1 \geq 0$

By using the third formula of the Master Method,

$W(n) = \theta(n^{log_b a} lg^{k+1} n) = \theta(lg^2 n)$

(b)

charJump array

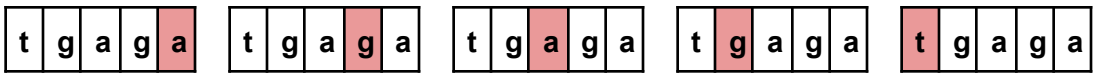
charJump['a'] = 0

charJump['g'] = 1

charJump['t'] = 4

matchJump array

matchJump[i] = mJ[i]



match: 0	match: 1	match: 2	match: 3	match: 4
slide: 1	slide: 5	slide: 2	slide: 5	slide: 5
mJ[4]: 1	mJ[3]: 6	mJ[2]: 4	mJ[1]: 8	mJ[0]: 9

Pattern	t	g	a	g	a
matchJump	9	8	4	6	1

(c) **(i)** The Rabin-Karp string matching algorithm with hashing works as such, every window in the text string will be compared with the pattern.

In this case, the binary value in each window will be converted into a decimal value and subsequently hashed with the value 11. This is also done for the pattern string.

Both the values obtained from the text window and the pattern will be compared. Character comparisons are **ONLY** done when the hashed values are equal.

It is done to verify that both the text window and the pattern match.

The binary pattern is "00101" which is "5" in decimal. Hashing this value with 11 gives us 5 ($5 \ mod \ 11 = 5$).

However, hashing 5 with 11 is not the only value that will also result in a hashed value of 5 ($5 \ mod \ 11 \equiv 16 \equiv 27 \ldots$)

Analysing all text windows till a pattern match is found:
00100000101 (Text string)
00100 $\rightarrow 4 \ mod \ 11 = 4$ (No character comparisons)
01000 $\rightarrow 8 \ mod \ 11 = 8$ (No character comparisons)
10000 $\rightarrow 16 \ mod \ 11 = 5$ (5 character comparisons)
00000 $\rightarrow 0 \ mod \ 11 = 0$ (No character comparisons)
00001 $\rightarrow 1 \ mod \ 11 = 1$ (No character comparisons)
00010 $\rightarrow 2 \ mod \ 11 = 2$ (No character comparisons)
00101 $\rightarrow 5 \ mod \ 11 = 5$ (5 character comparisons)

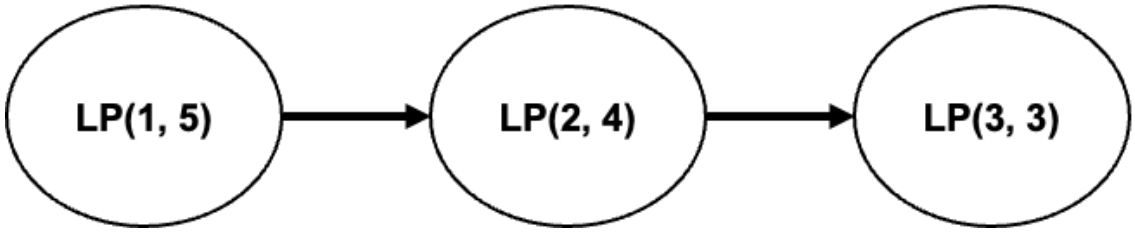
Total character comparisons: 5 + 5 = 10

(ii) m: pattern length
n: text length

The worst case occurs when spurious hits occur for all the text windows. A spurious hit refers to the scenario where the hash values are the same but a pattern match is not found. When hash values are the same, a total of m character comparisons will occur. When a pattern match is found, the algorithm is completed, else the next text window will be analysed. The maximum number of possible text windows is n - m + 1. In the case of the straightforward solution, as there is no hashing involved, m character comparisons are performed for each text window. Hence, in the worst case, the number of character comparisons is m(n - m +1) for both approaches.

The advantage to the Rabin Karp algorithm is that only one comparison per text window / sub-sequence is required. Character comparisons are only required when hash values match.

4 (a) (i)



(ii) String X = "ACCACA"

	index	0	1	2	3	4	5	6
index			**A**	**C**	**C**	**A**	**C**	**A**
0								
1	**A**		**TRUE**	**FALSE**	**FALSE**	**TRUE**	**FALSE**	**FALSE**
2	**C**			**TRUE**	**TRUE**	**FALSE**	**FALSE**	**FALSE**
3	**C**				**TRUE**	**FALSE**	**TRUE**	**FALSE**
4	**A**					**TRUE**	**FALSE**	**TRUE**
5	**C**						**TRUE**	**FALSE**
6	**A**							**TRUE**

Editor's note:
The question states to fill in the table entries $p[i][j]$ for all $j \geq i$. As such, we do not fill in the entries where $j < i$. Also the indexes start from 1.

(iii) The table p allows us to see which substrings in the given string X is palindromic. The logic behind the algorithm is that given a string ABBA, YABBAY will be a palindrome if firstly the string ABBA is a palindrome and that the added characters at both ends, indicated by Y, are equal. This idea applies to the condition where $p[i][j] = (x_i == x_j \; and \; p[i+1][j-1] = true) \; for \; j > i+1$.

The base cases are taken into account. If a string has an odd number of characters, then the base case $p[i][i] = true \; for \; i = 1..n$ applies, where a single character is considered as palindromic. Conversely, if a string has an even number of characters, then the base case $p[i][i+1] = true \; if \; x_i == x_{i+1}, \; false \; otherwise$ applies, where a string of length 2 is a palindrome only if both characters are equal.

```
def LP(m,n):
    # Initialise the n x n table
    p = [[False for x in range(n+1)] for y in range(n+1)]

    for i in range(1, n+1):
        for j in range(i, n+1):
```

```
if (i == j): p[i][j] = True
elif (j == i+1 and X[i] == X[j]): p[i][j] = True
elif (X[i] == X[j] and p[i+1][j-1] == True):
     p[i][j] = True
```

Time Complexity: $O(n^2)$ as two nested traversals are required.

Editor's note:
The algorithm LP(m,n) only computes the table p as stated in the question.

(b) A **NP (Non-deterministic Polynomially bounded)** problem is a decision problem whereby there exists a polynomially bounded non-deterministic algorithm. i.e it is a problem that can be verified by a polynomial time algorithm.

A **NP-complete** problem D is one that is in NP and every problem Q in NP is reducible to D in polynomial time.

The graph coloring problem is a NP problem. We can see that it is a decision problem and can be verified in polynomial time by iterating through each edge {u, v} in the graph and verifying that the color c(u) != c(v).

Solver: Jabez Ng Yong Xin (JABE001@e.ntu.edu.sg)