1. A MinHeap is a complete binary tree where the minimum-valued element is stored at the root node and every node is less than or equal to both of its child nodes.

Note: Java code is provided for MaxHeap algorithm in attached file.

1. compile and test MaxHeap code.

Ans: Compiled and tested.

b) modify MaxHeap code to MinHeap, and then compile and test the code.

Ans: Modified, compiled and tested.

2. Consider the following Text and Pattern

Text: ABCADCBABABCDABCDABDE

Pattern: BCD

a) Apply Brute-Force substring search algorithm to scan Pattern in Text string. Show step-by-step of the algorithm. Write Java code for the algorithm for input data. What is time complexity?

Ans:

**Algorithm:**

do

if (text letter == pattern letter)

compare next letter of pattern to next

letter of text

else

move pattern down text by one letter while (entire pattern found or end of text)

**Step by step:**

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

ABCADCBABABCDABCDABDE

BCD

We found the string and pattern matched at index 10 and 14.

**Time Complexity:** O(MN) where M is length of characters in a pattern and N is length of characters in the text.

b) Apply Rabin-Karp substring search algorithm to scan pattern in the

text string. Show step-by-step of algorithm. Write Java code

for the algorithm for input data. What is time complexity?

Ans:

**Algorithm:**

hash\_p=hash value of pattern hash\_t=hash value of first M letters in

body of text

do

if (hash\_p == hash\_t)

brute force comparison of pattern

and selected section of text

hash\_t = hash value of next section of

text, one character over while (end of text or

brute force comparison == true)

**Step by step:**

Size of Text(M) = 21

Size of Pattern(N) = 3

Creating own hash code for the characters:

A - 1

B - 2

C - 3

D - 4

E - 5

H(pattern) = B + C + D = 9

H(text) = A + B + C = 6

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (6 - 1) + 1 = 6

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (6 - 2) + 4 = 8

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (8 - 3) + 3 = 8

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (8 - 1) + 2 = 9

H(text) == H(pattern)

The pattern isn’t matching upon comparison.

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (9 - 4) + 1 = 6

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (6 - 3) + 2 = 5

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (5 - 2) + 1 = 4

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (4 - 1) + 2 = 5

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (5 - 2) + 3 = 6

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (6 - 1) + 4 = 9

H(text) == H(pattern)

Comparing all characters, it matches. And since the string isn’t reached the end so, moving.

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (9 - 2) + 1 = 8

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (8 - 3) + 2 = 7

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (7 - 4) + 3 = 6

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (6 - 1) + 4 = 9

H(text) == H(pattern)

Comparing all characters, it matches. And since the string isn’t reached the end so, moving.

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (9 - 2) + 1 = 8

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (8 - 3) + 2 = 7

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (7 - 4) + 4 = 7

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Sliding it to the next,

H(Text) = (7 - 1) + 5 = 11

H(text) != H(pattern)

ABCADCBABABCDABCDABDE

BCD

Pattern found at index 10 and 14.

**Time Complexity:** O(NM) where M is length of characters in a pattern and N is length of characters in the text.

Worst case occurs when all characters of pattern and text are same as hash values of all the substrings of text match with hash value of the pattern.

c) What is the difference between the two time complexity?

Ans: In Rabin Karp, the time complexity is much **faster** as compared to Brut force. In only one case of Rabin Karp, i.e. when hash(pattern) == hash(text) then, the worst case is O(MN) else its O(M +N) however in Brute Force, if there is a mismatch, we jump back to the next character in the text. So there are lots of comparison of characters.

3. Consider this undirected graph:

A close up of a clock

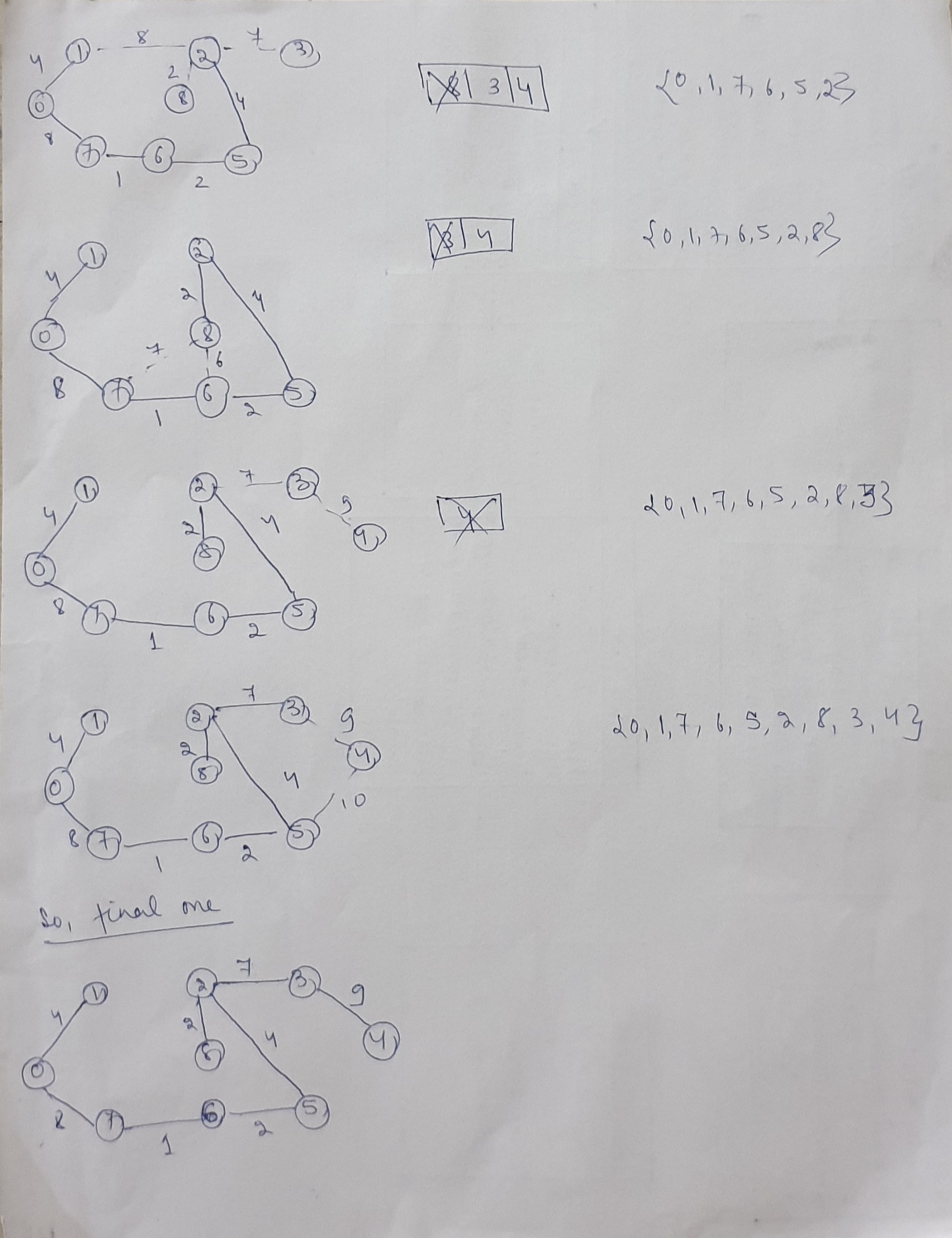
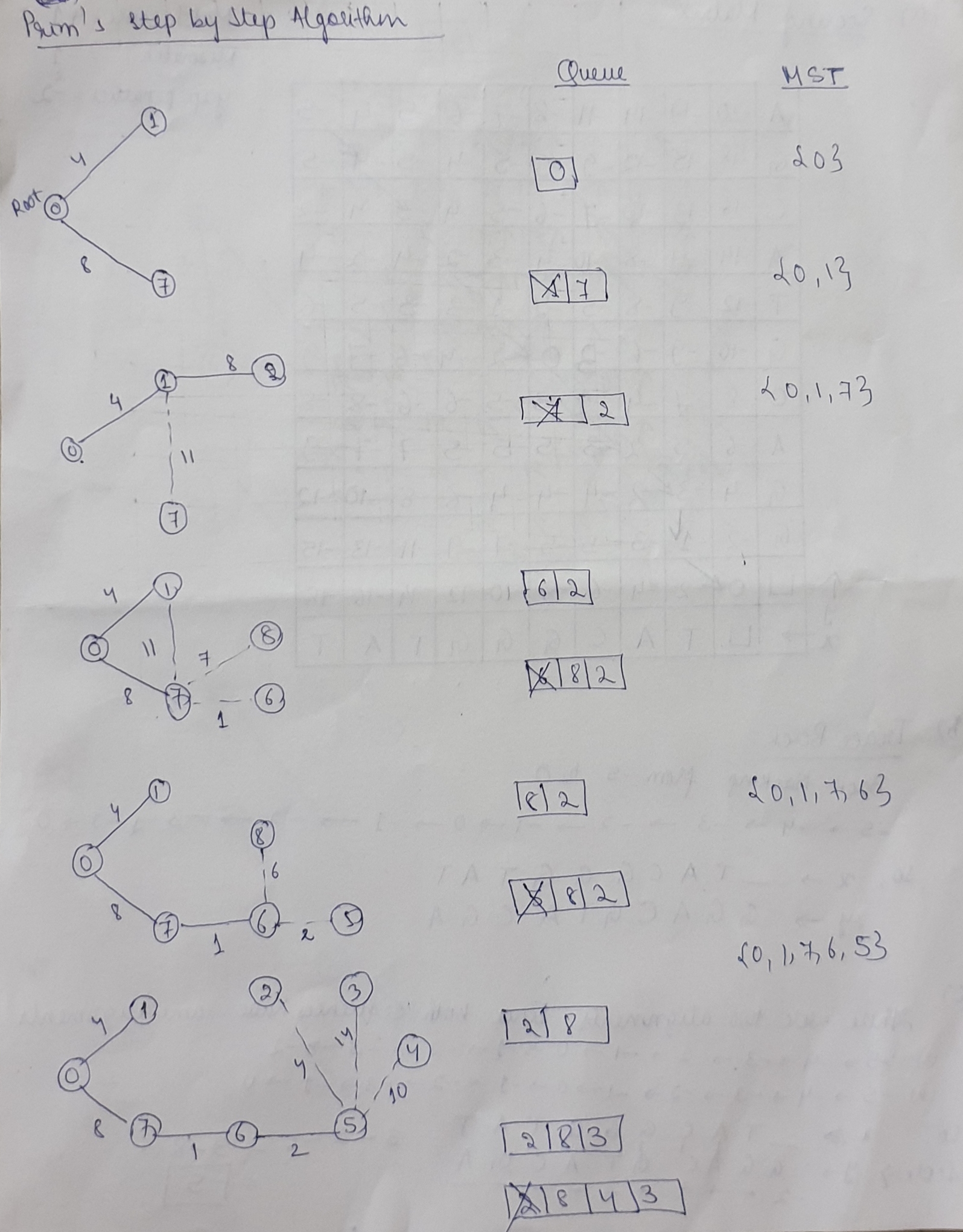
Description automatically generated

1. What is Minimum Spanning Tree Prim’s method for this graph, Use required Data Structures, show step by step algorithm

Ans: Prim’s algorithm is a greedy algorithm for finding Minimum Spanning Tree of a graph.

**Algorithm:**

* Create an empty tree, M, which will act as the MST.
* Choose a random vertex v from the graph.
* Add the edges that are connected to v into some data structure E.
* Find the edge in E with minimum weight, and add edge to M. Now make v equal to the other vertex in the edge and repeat from Step 3.

**Step by Step:**

Total minimum key = **37**

b) What is the space and time complexity of this algorithm?

Ans: Time Complexity of the algorithm is O(V^2).

Space Complexity of the algorithm is O(V).

c) Write Java code, compile and test.

Ans: Compiled and tested.

4. Consider Knapsack problem in this article:

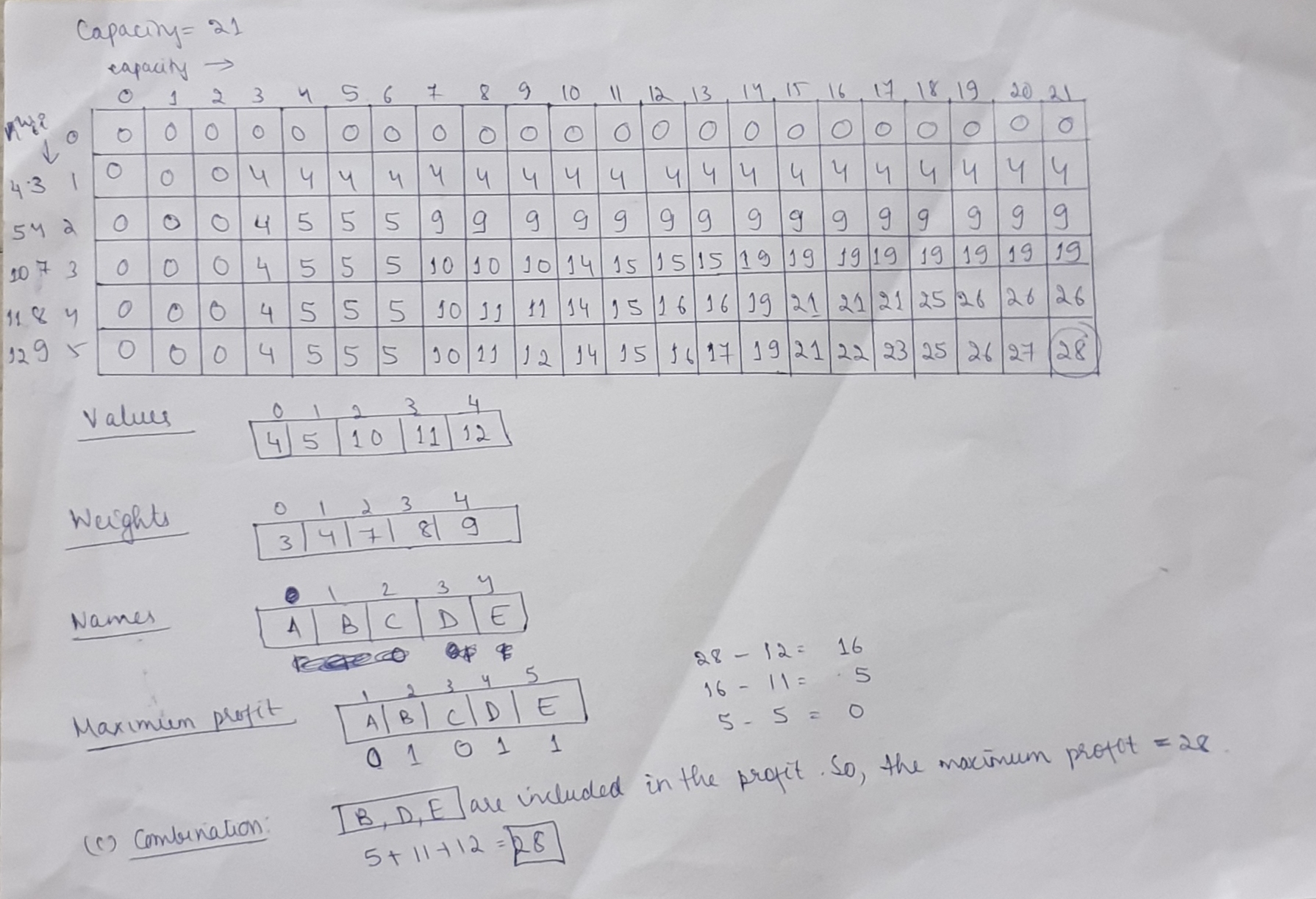
<https://www.radford.edu/~nokie/classes/360/dp-knapsack.html>

1. Find solutions for m=21

Ans: Maximum value = **28**

Attached image solving the Knapsack problem and getting the maximum value:

b) How does it use Dynamic Programming?

Ans: Dynamic programming requires an optimal substructure and overlapping sub-problems, both of which are present in the 0–1 knapsack problem. Dynamic programming also includes sequence of decisions for every object which Knapsack has. Also, Dynamic Programming says that it should look into all possible solutions and pick up the best one. Total possible solutions is 2^n. Then time complexity is 2^n. So, we’ll use indirectly to get solution in optimal way. While building the table, we are adding one object at once and ignoring the rest of the rows but while adding rest of the objects, we are considering the previous rows. So, that is how Knapsack is using Dynamic programming.