Top Most Important Programming Patterns for DSA Interviews

By

Prashant Kumar

https://www.linkedin.com/in/prashant-kumar-76b786168/

https://github.com/prashantt17

@prashantkumar

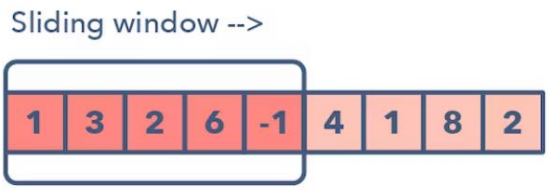
1. Sliding Window

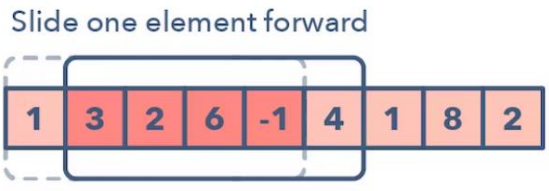
The Sliding window is a problem-solving technique of data structure and algorithm for problems that apply arrays or lists. These problems are painless to solve using a brute force approach in **$O(n^2)$** or **$O(n^3)$**. However, the **Sliding window** technique can reduce the time complexity to **O(n)**.

Sliding Window is used when we need to nested loops into single loop.

Below are some fundamental hints to identify this type of problem:

- The problem will be based on an array, list or string type of data structure.
- It will ask to find subrange in that array or string will have to give longest, shortest, or target values.
- Its concept is mainly based on ideas like the longest sequence or shortest sequence of something that satisfies a given condition perfectly.





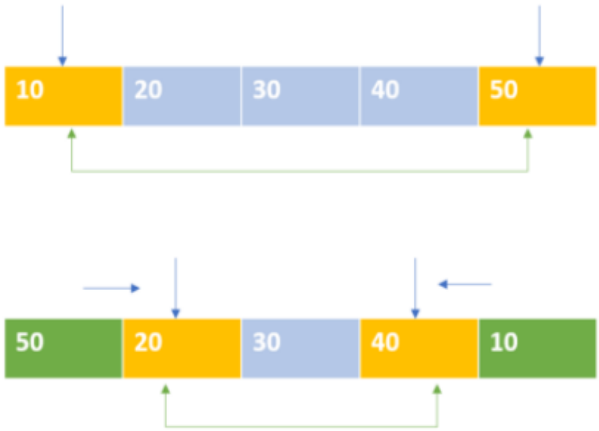
```
public int slidingWindowForMaxSum(int arr[], int n, int k)
    {
        int max_sum = Integer.MIN_VALUE;

        for (int i = 0; i < n - k + 1; i++) {
            int current_sum = 0;
            for (int j = 0; j < k; j++)
                current_sum = current_sum + arr[i + j];

            max_sum = Math.max(current_sum, max_sum);
        }
          return max_sum;
    }
```

2. The Two Pointer Technique

Two pointer algorithm is one of the most commonly asked questions in any programming interview. This approach optimizes the runtime by utilizing some order (not necessarily sorting) of the data. It is generally applied on lists (arrays) and linked lists. This is generally used to search pairs in a sorted array. This approach works in constant space.

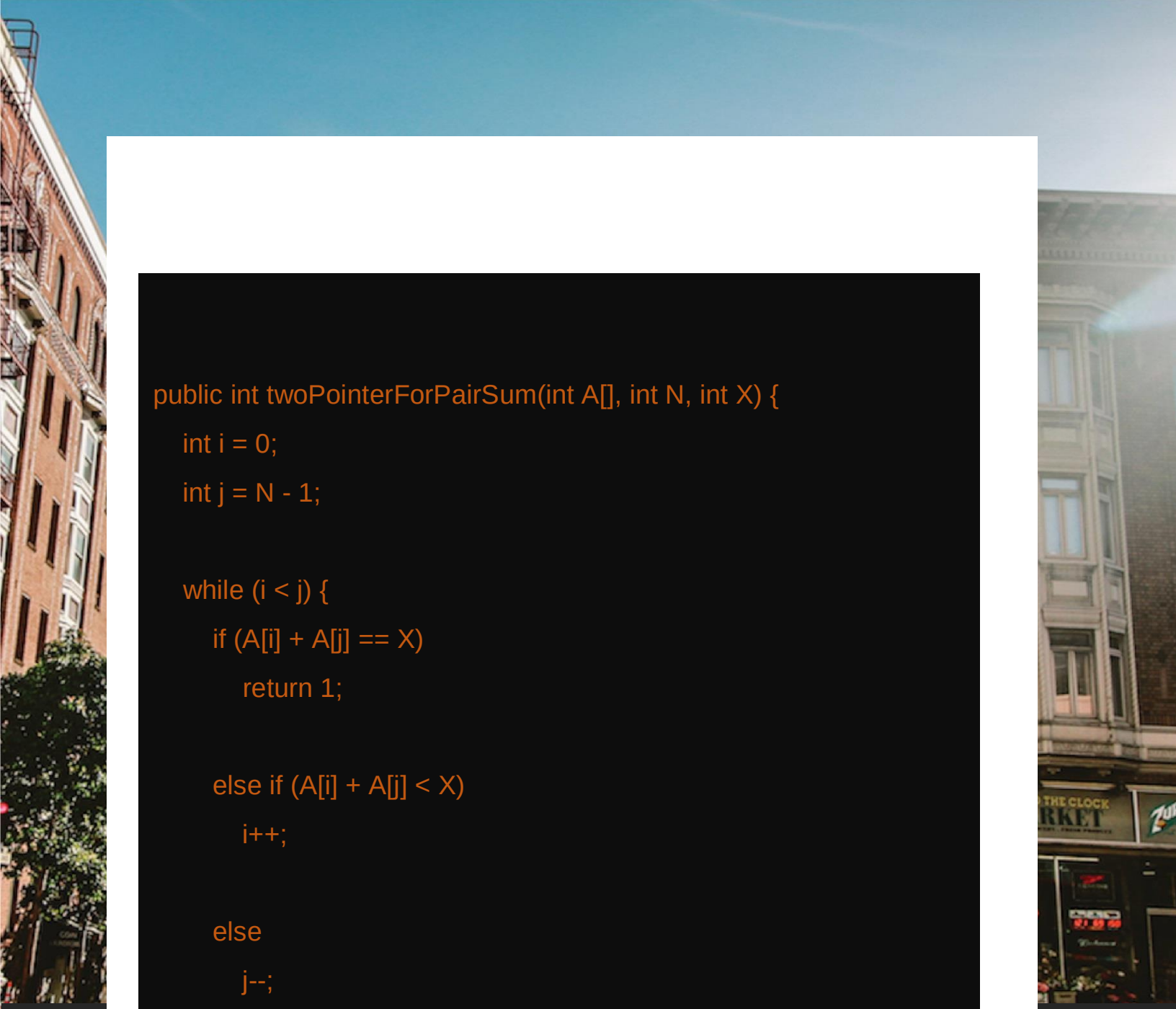
Common patterns in the two-pointer approach entail:

1. Two pointers, each starting from the beginning and the end until they both meet.
2. One pointer moving at a slow pace, while the other pointer moves at twice the speed.



Here are some problems that feature the Two Pointer pattern:

1. Squaring a sorted array (easy)
2. Triplets that sum to zero (medium)
3. Comparing strings that contain backspaces (medium)

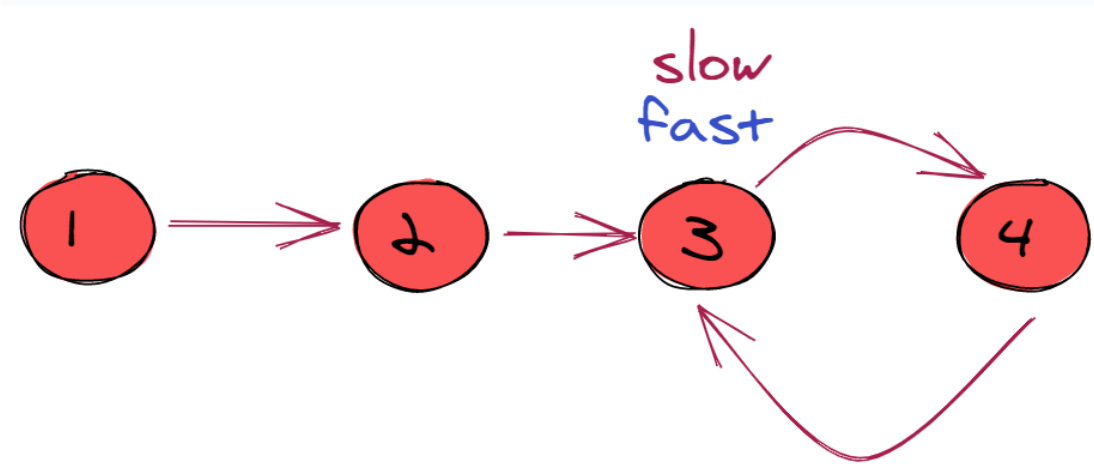


```
public int twoPointerForPairSum(int A[], int N, int X) {
    int i = 0;
    int j = N - 1;

    while (i < j) {
        if (A[i] + A[j] == X)
            return 1;

        else if (A[i] + A[j] < X)
            i++;

        else
            j--;
    }
    return 0;
}
```

3. Fast and Slow Pointers

Fast and Slow pointers is an algorithm that works by setting two pointers that move at different speeds, one faster than the other. If the faster pointer “meets” the slower pointer at any point, a cycle is detected. I came across an interesting problem recently that can be solved using the Fast and Slow pointers technique.

- By moving at different speeds, the algorithm proves that the two pointers are going to meet eventually. The fast pointer should catch the slow pointer once both the pointers are in a cyclic loop.
- The slow and fast pointers algorithm (also known as Floyd's Cycle Detection algorithm or the Tortoise and Hare algorithm)
- Time Complexity: O(N) where N is the number of nodes in the Linked Lists.
- Space Complexity: O(1), algorithm runs in constant space.



```
public void FastandSlowPointerToDetectLoop() {
    Node slow_p = head, fast_p = head;
    int flag = 0;
    while (slow_p != null && fast_p != null &&
        fast_p.next != null) {
        slow_p = slow_p.next;
        fast_p = fast_p.next.next;
        if (slow_p == fast_p) {
            flag = 1;
            break;
        }
    }
    if (flag == 1)
        System.out.println("Loop found");
    else
        System.out.println("Loop not found");
}
```

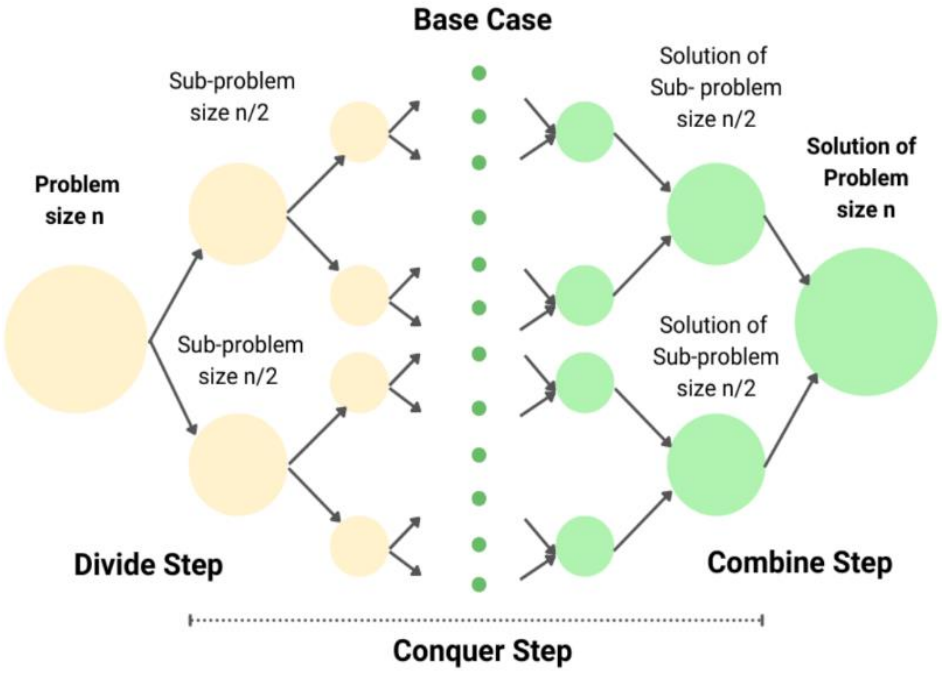
4. Divide and Conquer Approach

The definition of divide and conquer is to strategically break up entities into chunks. This algorithm is a strategy for solving a large problem. It has three steps:

1. Breaking up a large problem into smaller, solvable subproblems.
2. Solving, or ''conquering,'' each of the subproblems.
3. Combining the solutions to the subproblems to achieve the overall solution to the original problem.

Furthermore, with the divide-and-conquer approach, computational complexity is estimated using mathematical equations known as recurrence relations. The solution to a divide-and-conquer recurrence, represented as f(n). This concept is illustrated with the following Master Theorem formula: f(n)=af(n/b)+g(n)

where f(n) represents the number of operations needed to solve a problem.

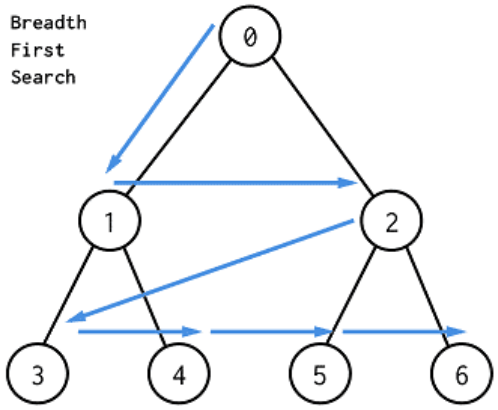


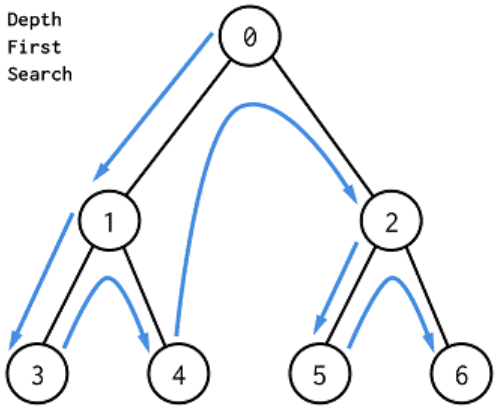
```
static int DACtoFindMaxElement(int a[], int index, int l) {
    int max;
    if (l - 1 == 0) {
        return a[index];
    }
    if (index >= l - 2) {
        if (a[index] > a[index + 1])
            return a[index];
        else
            return a[index + 1];
    }
    max = DACtoFindMaxElement(a, index + 1, l);

    if (a[index] > max)
        return a[index];
    else
        return max;
}
```

5. BFS and DFS Traversal

BFS is an algorithm that is used to graph data or searching tree or traversing structures. The algorithm efficiently visits and marks all the key nodes in a graph in an accurate breadthwise fashion. This algorithm selects a single node (initial or source point) in a graph and then visits all the nodes adjacent to the selected node. Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them.

DFS is an algorithm for finding or traversing graphs or trees in depth-ward direction. The execution of the algorithm begins at the root node and explores each branch before backtracking. It uses a stack data structure to remember, to get the subsequent vertex, and to start a search, whenever a dead-end appears in any iteration. The full form of DFS is Depth-first search.





➔ BFS

```
void BFSToFindCurrentLevel(Node root, int level) {
    if (root == null)
        return;
    if (level == 1)
        System.out.print(root.data + " ");
    else if (level > 1) {
        printCurrentLevel(root.left, level - 1);
        printCurrentLevel(root.right, level - 1);
    }
}
```

➔ **DFS**

```
void DFStoFindInorder(Node node) {
    if (node == null)
        return;

    /* first recur on left child */
    printInorder(node.left);

    /* then print the data of node */
    System.out.print(node.key + " ");

    /* now recur on right child */
    printInorder(node.right);
}
```

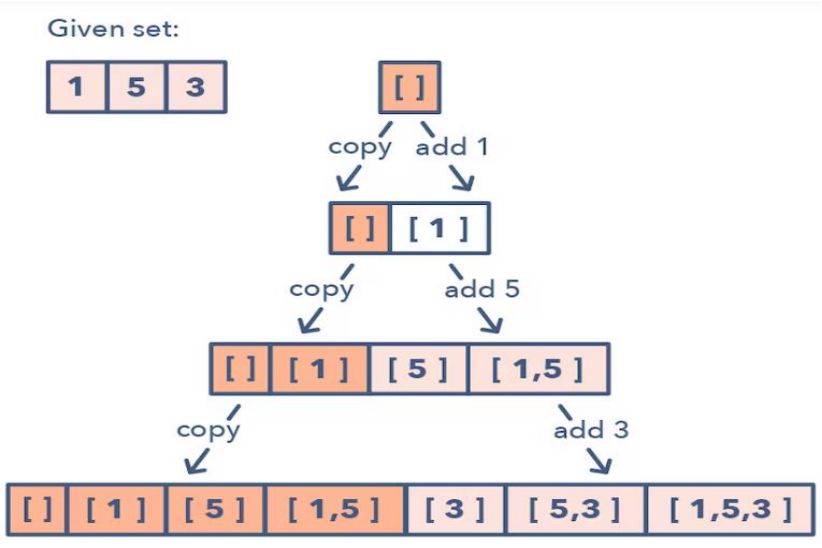
6. Subsets

A huge number of coding interview problems involve dealing with Permutations and Combinations of a given set of elements. The pattern Subsets describes an efficient Breadth First Search (BFS) approach to handle all these problems.

This type of problems can be solved by two different approaches

1. **Backtracking Approach -** It's a recursive approach where we backtrack each solution after appending the subset to the resultset.
    - ➔ Time Complexity: **O(2^n)**
    - ➔ Space Complexity: **O(n)** for extra array subset.

2. **BitMasking Approach -** The binary representation of a number in range 0 to 2^n is used as a mask where the index of set bit represents the array index to be included in the subset.
    - ➔ Time Complexity: **O(n*2^n)**
    - ➔ Space Complexity: **O(n)**



➔ BACKTRACKING

```
public void
findSubsets(List < List < Integer >> subset, ArrayList < Integer >
nums, ArrayList < Integer > output, int index) {
    // Base Condition
    if (index == nums.size()) {
        subset.add(output);
        return;
    }

    // Not Including Value which is at Index
    findSubsets(subset, nums, new ArrayList < > (output), index + 1);

    // Including Value which is at Index
    output.add(nums.get(index));
    findSubsets(subset, nums, new ArrayList < > (output), index + 1);
}
```

```
static void BitMaskingToFindsubSet(int arr[], int N) {
    List < String > list = new ArrayList < > ();
    int size = (int) Math.pow((double) 2, (double) N);
    for (int i = 0; i < size; i++) {
        String s = "";
        for (int j = 0; j < N; j++) {
            if ((i & (1 << j)) > 0)
                s += arr[j] + " ";
        }
        if (!(list.contains(s))) {
            list.add(s);
        }
    }
    for (int ii = 0; ii < list.size(); ii++) {
        String s = list.get(ii);
        String str[] = s.split(" ");
        System.out.print("{ ");
        for (int jj = 0; jj < str.length; jj++) {
            if (ii == 0)
                System.out.print(str[jj])
            else
                System.out.print(Integer.parseInt(str[jj]) + " ");
        }
        System.out.print(" }\n");
    }
}
```

7. Greedy Algorithm

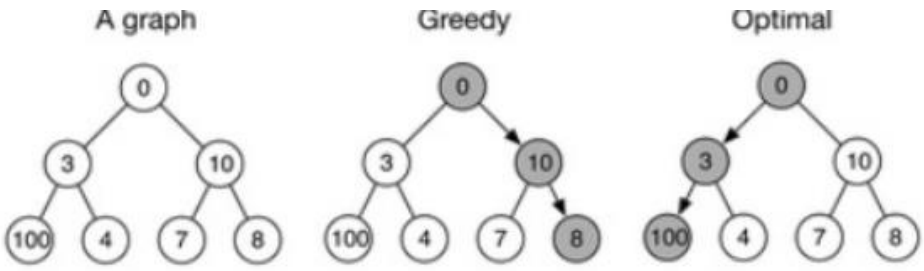
A **greedy algorithm** is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

To solve a problem based on the greedy approach, there are two stages

1. Scanning the list of items
2. Optimization

wo conditions define the greedy paradigm.

- Each stepwise solution must structure a problem towards its best-accepted solution.
- It is sufficient if the structuring of the problem can halt in a finite number of greedy steps.



A greedy algorithm fails to maximise the sum of nodes along a path from the top to the bottom because it lacks the foresight to choose suboptimal solutions in the current iteration that will allow for better solutions later

```
void primMST(int graph[][]) {
    int parent[] = new int[V];
    int key[] = new int[V];
    Boolean mstSet[] = new Boolean[V];
    for (int i = 0; i < V; i++) {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }
    key[0] = 0; // Make key 0 so that this vertex is
    parent[0] = -1; // First node is always root of MST
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
            if (graph[u][v] != 0 && mstSet[v] == false &&
                graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
    }
    printMST(parent, graph);
}
```

https://www.linkedin.com/in/prashant-kumar-76b786168/

https://github.com/prashantt17

@prashantkumar