Disjoint Sets DS – Disjoint set

Ref: https://cp-algorithms.com/data_structures/disjoint_set_union.htm

Ref: https://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/16/Small16.pdf

Q1 graph data structure(nodes/ edges) and its application to represent RW examples

Q2 Explain the array implementation and LL implementation (either in programming language or algorithm)

Q3 Visualization of array implementation of union-find(Ex: union of element X with element Y etc.)

Q4 What is path compression and union by rank? What is disadvantage of array implementation of Union Find.

Data structure Disjoint Set Union or DSU. Often it is also called Union Find because of its two main operations.

This data structure provides the following capabilities. We are given several elements, each of which is a separate set.

A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is.

The classical version also introduces a third operation, it can create a set from a new element.

Thus the basic interface of this data structure consists of only three operations:

- make_set(v) - creates a new set consisting of the new element v
- union_sets(a, b) - merges the two specified sets (the set in which the element a is located, and the set in which the element b is located)
- find_set(v) - returns the representative (also called leader) of the set that contains the element v.

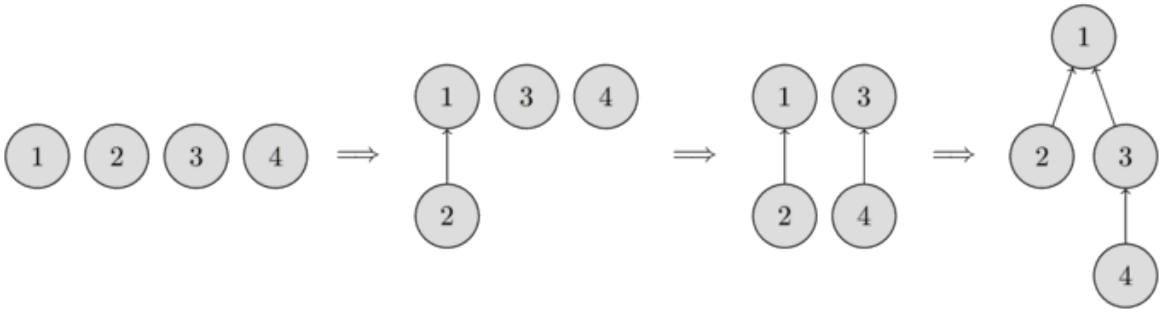
This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after union_sets calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set, if find_set(a) == find_set(b). Otherwise they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost O(1)

Build an efficient data structure

We will store the sets in the form of trees: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the following image you can see the representation of such trees.



In the beginning, every element starts as a single set, therefore each vertex is its own tree.

Then we combine the set containing the element 1 and the set containing the element 2

Then we **combine the set containing the element 3 and the set containing the element 4**. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

For the implementation this means that we will have to maintain an array parent that stores a reference to its immediate ancestor in the tree.

Code: Disjoint Set with Union and Find

```
#include <iostream>
#include <vector>
using namespace std;

// Define the parent array globally
vector<int> parent;

// Function to initialize a set for each element
void make_set(int v) {
    parent[v] = v;  // Each element is its own parent
initially
}



// Function to find the representative (root) of a set (with
path compression)
//
int find_set(int v) {
    if (v == parent[v])  // If `v` is the root of its set
        return v;
    return parent[v] = find_set(parent[v]);  // Path
compression
}
```

```
// Function to union two sets
void union_sets(int a, int b) {
    a = find_set(a);  // Find the root of set containing `a`
    b = find_set(b);  // Find the root of set containing `b`
    if (a != b)       // If `a` and `b` are not already in
the same set
        parent[b] = a;  // Merge the sets by attaching root
of `b` to root of `a`
}
```

```
int main() {
    // Number of elements
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    // Resize the parent vector to hold `n` elements
    parent.resize(n + 1);

    // Initialize each element as its own set
    for (int i = 1; i <= n; i++) {
        make_set(i);
    }

    // Example usage
    int num_operations;
    cout << "Enter the number of operations (union/find): ";
    cin >> num_operations;
```

```
    for (int i = 0; i < num_operations; i++) {
        string op;
        cout << "Enter operation (union/find): ";
        cin >> op;

        if (op == "union") {
            int u, v;
            cout << "Enter two elements to union: ";
            cin >> u >> v;
            union_sets(u, v);
            cout << "Union of " << u << " and " << v << "
completed.\n";
        } else if (op == "find") {
            int u;
            cout << "Enter the element to find the set: ";
            cin >> u;
            cout << "The representative of " << u << " is: "
<< find_set(u) << endl;
        } else {
            cout << "Invalid operation. Use 'union' or
'find'.\n";
        }
    }

    // Display final parent array
    cout << "Final parent array: ";
    for (int i = 1; i <= n; i++) {
        cout << parent[i] << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

However this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call `find_set(v)` can take

O(n)

This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work.

Explanation

1. Initialization (`make_set`):
    - Each element starts as its own set, i.e., `parent[v] = v`.
2. Find Operation (`find_set`):
    - This operation locates the root (representative) of the set containing the element `v`.
    - Path compression is used to flatten the structure, making future queries faster.
3. Union Operation (`union_sets`):
    - This operation merges the sets containing elements `a` and `b` by connecting their roots.
    - The root of one set is made the parent of the other.
4. Input Handling:
    - User specifies the number of elements (`n`) and operations (`union` or `find`).
    - Union and find operations are executed based on user input.
5. Output:

- After all operations, the final parent array is displayed, showing the representative of each element.

Sample Input and Output

Input:

mathematica

CopyEdit

```
Enter the number of elements: 5
Enter the number of operations (union/find): 4
Enter operation (union/find): union
Enter two elements to union: 1 2
Enter operation (union/find): union
Enter two elements to union: 3 4
Enter operation (union/find): union
Enter two elements to union: 2 3
Enter operation (union/find): find
Enter the element to find the set: 4
```

Output:

sql

CopyEdit

```
Union of 1 and 2 completed.
Union of 3 and 4 completed.
Union of 2 and 3 completed.
The representative of 4 is: 1
Final parent array: 1 1 1 1 5
```

Features

- Path compression ensures efficient `find` operations.
- `union` combines sets efficiently without redundant work.

- Time complexity for both `union` and `find` is nearly O(1) due to optimizations (amortized).

This implementation is compact and can be extended further for advanced applications like Kruskal's algorithm or connected component detection.

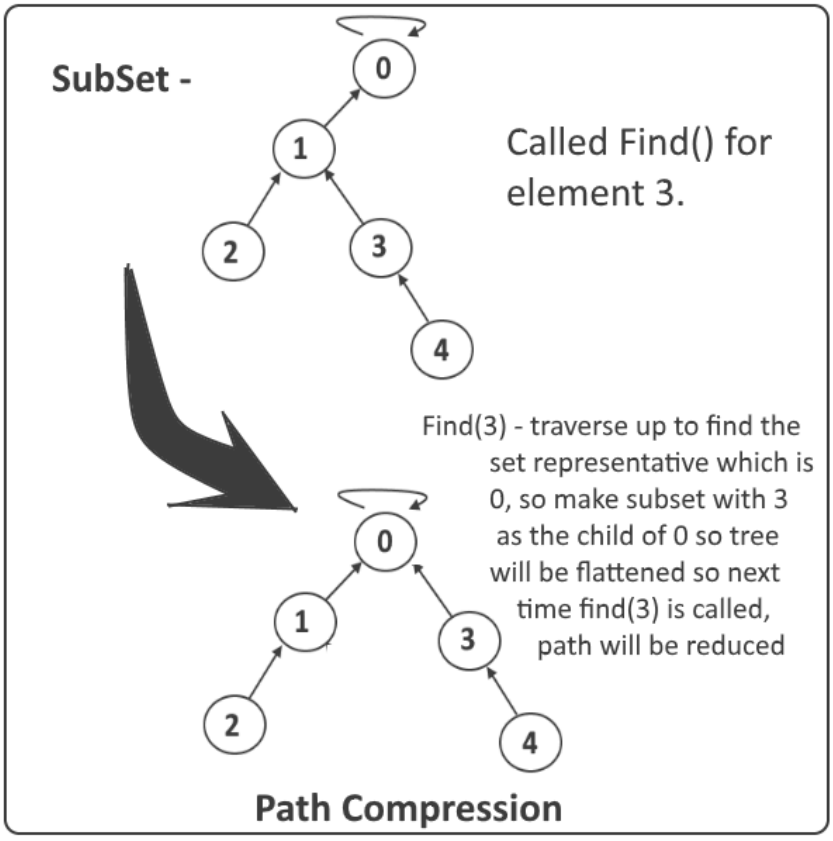
Visualization

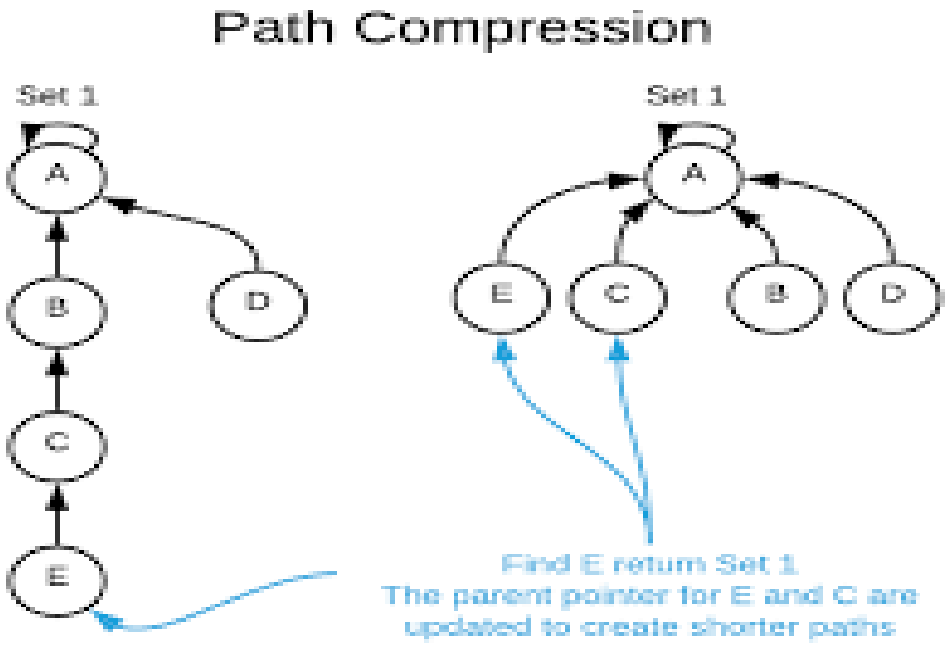
https://www.cs.usfca.edu/~galles/visualization/DisjointSets.html

Path compression optimization

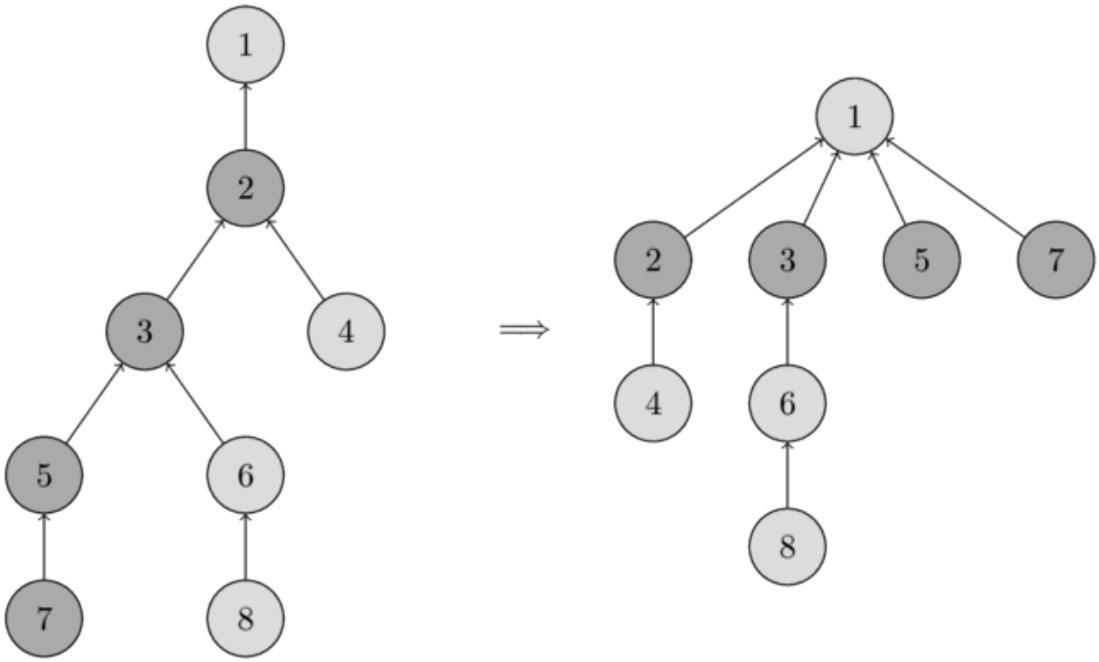
This optimization is designed for speeding up `find_set`.

If we call `find_set(v)` for some vertex `v`, we actually find the representative `p` for all vertices that we visit on the path between `v` and the actual representative `p`. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to `p`.


SubSet -
0
1
2
3
4
Called Find() for element 3.
Find(3) - traverse up to find the set representative which is 0, so make subset with 3 as the child of 0 so tree will be flattened so next time find(3) is called, path will be reduced
0
1
3
2
4
Path Compression


Path Compression
Set 1
A
B
D
C
E
Set 1
A
E
C
B
D
Find E return Set 1
The parent pointer for E and C are updated to create shorter paths

You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling `find_set(7)`, which shortens the paths for the visited nodes 7, 5, 3 and 2.



The new implementation of `find_set` is as follows:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

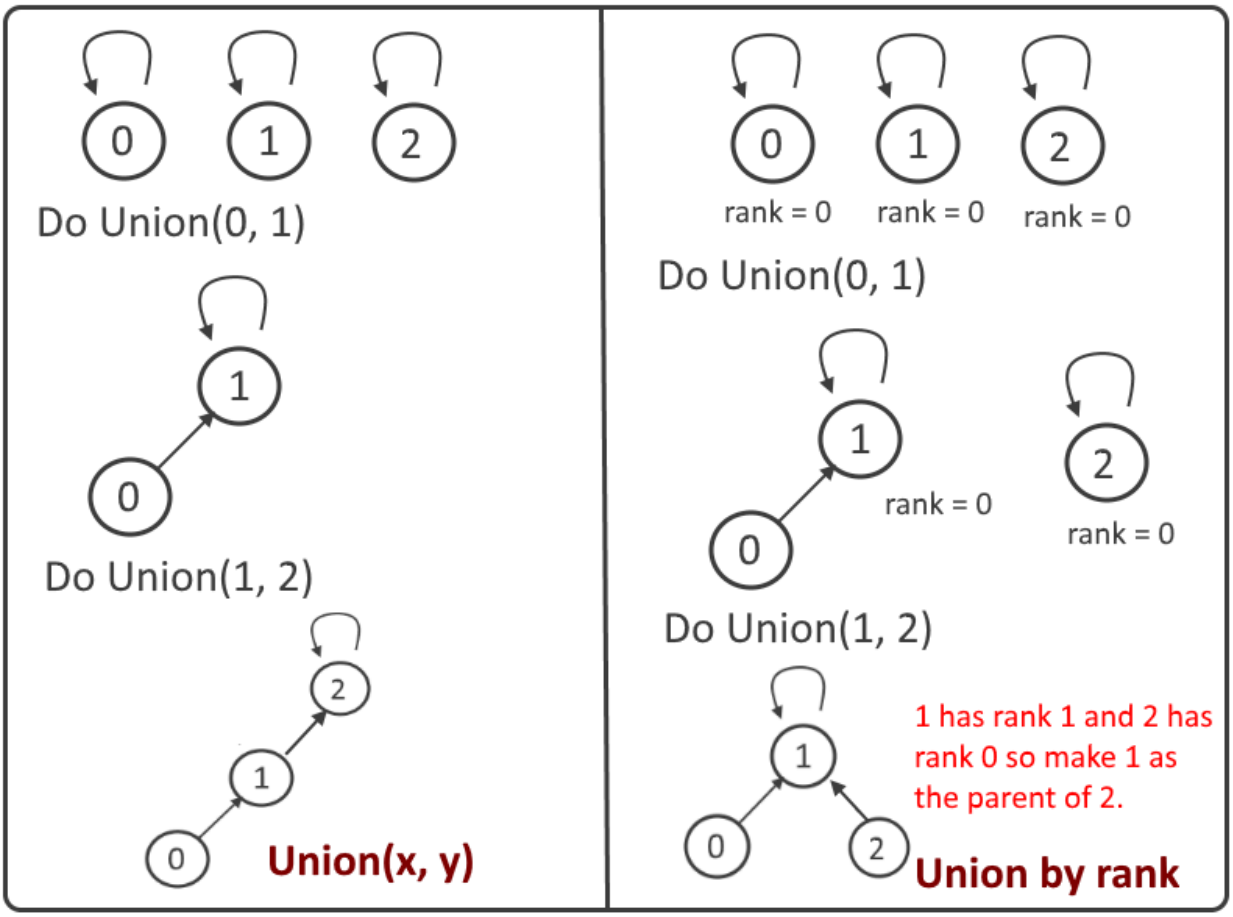
The simple implementation does what was intended: **first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.**

This simple modification of the operation already achieves the time complexity

O(logn) per call on average (here without proof). There is a second modification, that will make it even faster.

Union by size / rank

In this optimization we will change the `union_set` operation. To be precise, we will change **which tree gets attached to the other one**. In the naive implementation the second tree always got attached to the first one. In practice that can lead to trees containing chains of length



O(n). With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches:

In the first approach we use the size of the trees as rank,

and in the second one we use the depth of the tree (more precisely, the upper bound on the tree depth

, because the depth will get smaller when applying path compression).

In both approaches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank.

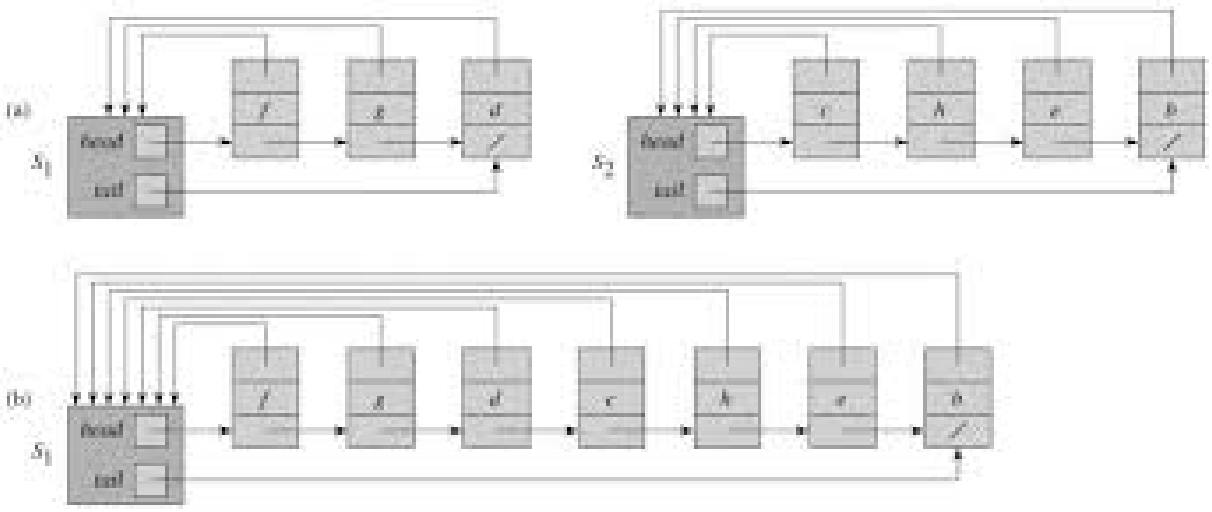
Here is the implementation of union by size:

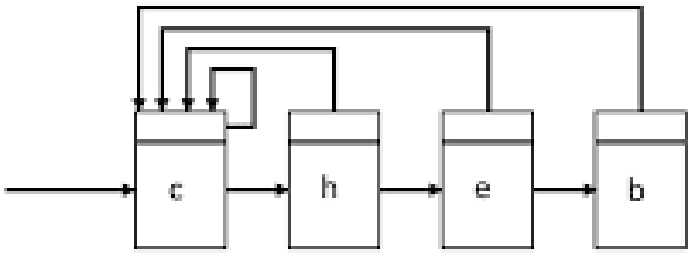
```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```

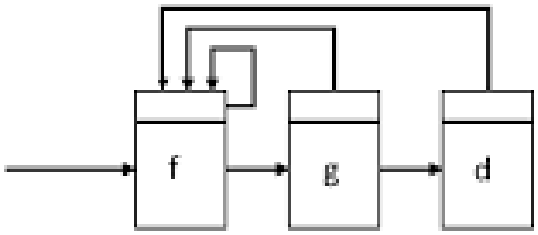
And here is the implementation of union by rank based on the depth of the trees:

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}


void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

Both optimizations are equivalent in terms of time and space complexity. So in practice you can use any of them.







Time complexity

As mentioned before, if we combine both optimizations - path compression with union by size / rank - we will reach nearly constant time queries. It turns out, that the final amortized time complexity is

$O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly. In fact it grows so slowly, that it doesn't exceed Amortized complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower then the amortized time. E.g. in our case a single call might take

$O(\log n)$ in the worst case, but if we do m such calls back to back we will end up with an average time of $O(\alpha(n))$

Also, it's worth mentioning that DSU with union by size / rank, but without path compression works in

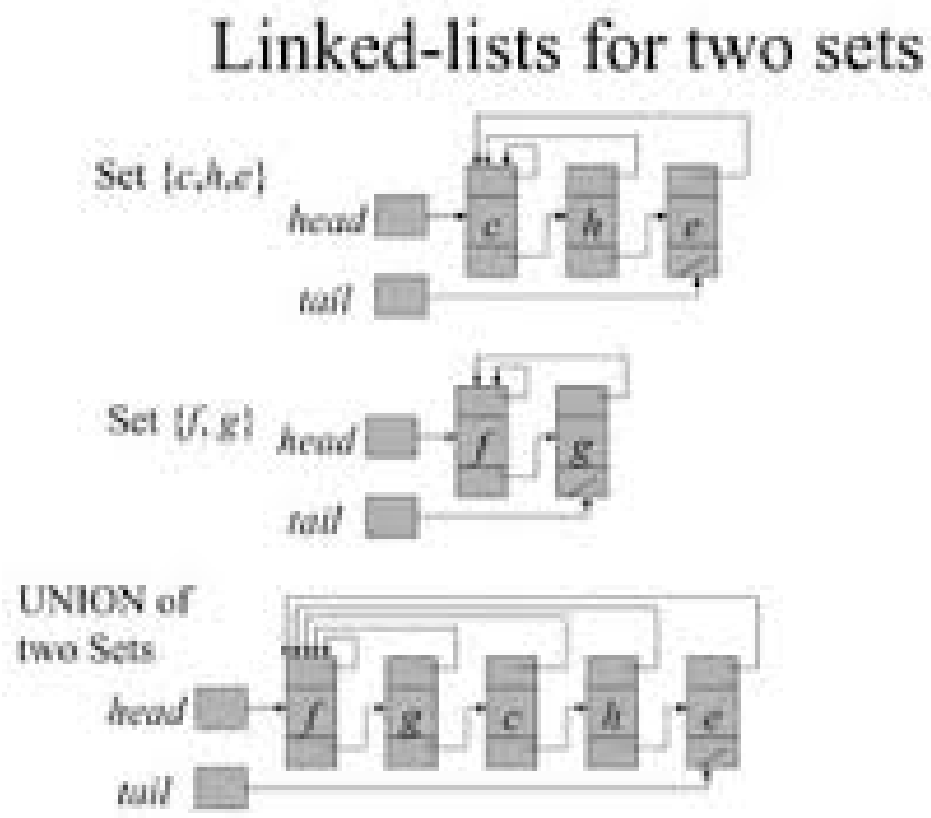
$O(\log n)$ time per query.

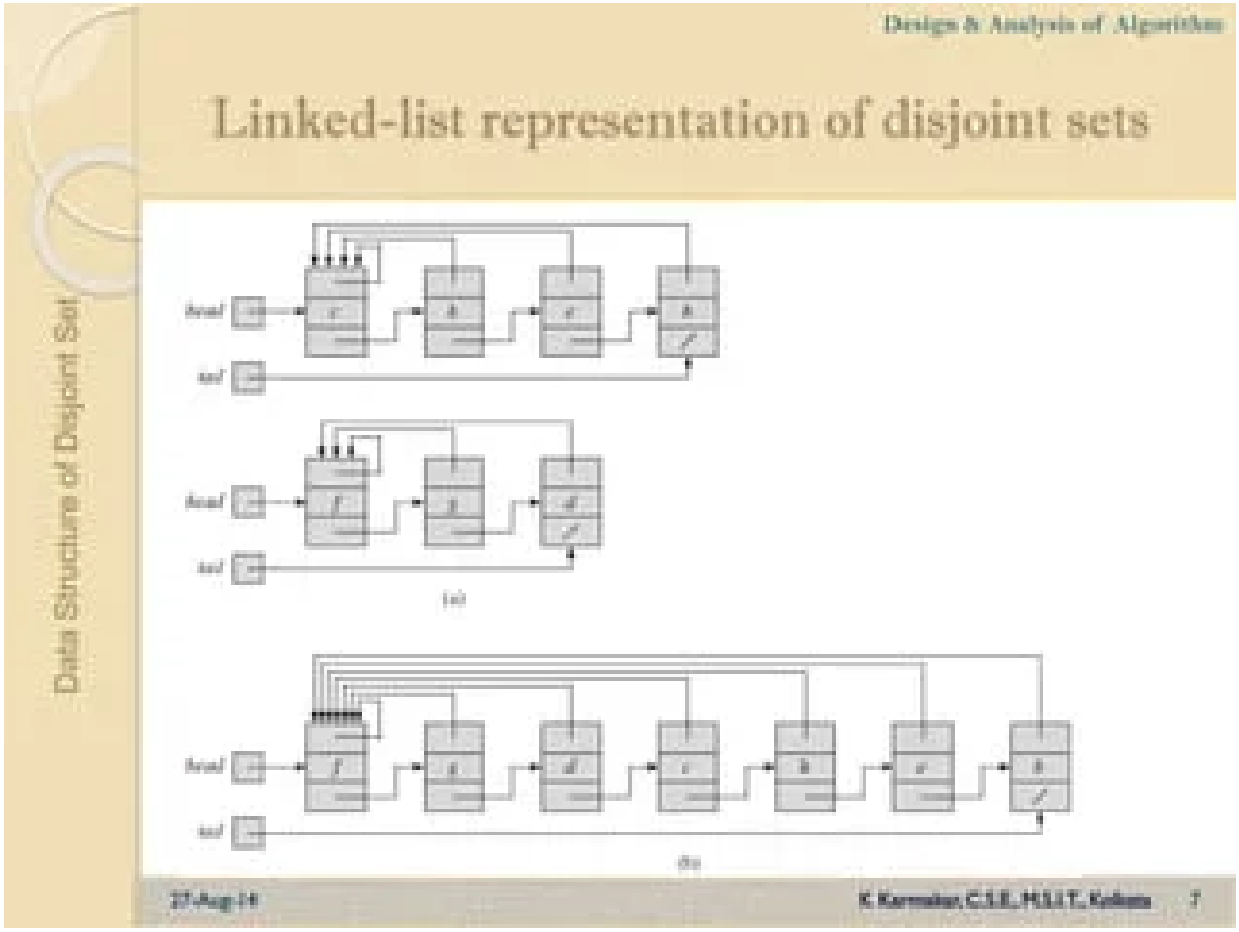
The linked list representation of disjoint sets is one way to implement the disjoint-set data structure. In this representation, each set is maintained as a linked list, and every element in the set points to its parent or the head of the list (which represents the set leader). Below is a detailed explanation of this approach:

Key Components

1. Linked List Representation:
    - Each set is represented as a linked list.
    - Each node contains:
        - The element (value).
        - A pointer to the next element in the set.
        - A pointer to the leader (representative of the set).

2. Head Node:
    - Each set has a "head node" or "leader," which acts as the representative of the set.
    - All elements in the linked list point to this head node.
3. Auxiliary Information:
    - To optimize operations, the head node can store the size of the set or a rank to help in merging sets efficiently.





Operations

1. `make_set(x)`:
    - Creates a new set with a single element `x`.
    - `x` becomes the head of the list and points to itself.
2. `find_set(x)`:
    - Returns the representative (leader) of the set containing `x`.
    - Traverse the linked list to find the head node.
3. `union_sets(a, b)`:
    - Merges the sets containing `a` and `b`.
    - Append the smaller linked list to the larger one (union by size).
    - Update all the elements in the smaller list to point to the leader of the larger list.

Code Implementation

Here's a simple C++ implementation of disjoint sets using a linked list:

cpp
CopyEdit

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

// Node structure for each element in the set
struct Node {
    int value;       // The element value
    Node* next;      // Pointer to the next node in the
linked list
    Node* leader;    // Pointer to the head of the linked
list (leader)
};

// Maps each element to its corresponding node
unordered_map<int, Node*> node_map;

// Create a new set with a single element
void make_set(int x) {
    Node* node = new Node();
    node->value = x;
    node->next = nullptr;
    node->leader = node; // Points to itself as the leader
    node_map[x] = node;  // Store in the global map
}

// Find the leader (head) of the set containing `x`
```

```
Node* find_set(int x) {
    return node_map[x]->leader;
}

// Union two sets containing `a` and `b`
void union_sets(int a, int b) {
    Node* leaderA = find_set(a);
    Node* leaderB = find_set(b);

    if (leaderA != leaderB) { // Only merge if they are in
different sets
        // Append the smaller list to the larger list
        Node* temp = leaderA;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = leaderB;

        // Update the leader of all nodes in the merged list
        Node* current = leaderB;
        while (current != nullptr) {
            current->leader = leaderA;
            current = current->next;
        }
    }
}

// Display the set containing `x`
void display_set(int x) {
    Node* leader = find_set(x);
    Node* current = leader;
```

```
    cout << "{ ";
    while (current != nullptr) {
        cout << current->value << " ";
        current = current->next;
    }
    cout << "}" << endl;
}

int main() {
    // Example usage
    make_set(1);
    make_set(2);
    make_set(3);
    make_set(4);
    make_set(5);

    union_sets(1, 2);
    union_sets(3, 4);
    union_sets(2, 3);

    cout << "Set containing 1: ";
    display_set(1);

    cout << "Set containing 5: ";
    display_set(5);

    return 0;
}
```

Explanation of the Code

1. `make_set(x)`:

    - Creates a new node for `x` and initializes its `leader` to itself.
2. `find_set(x)`:
    - Accesses the leader node through the `leader` pointer.
3. `union_sets(a, b)`:
    - Finds the leaders of the two sets.
    - Appends the second list to the first and updates the `leader` pointers of all elements in the second list.
4. `display_set(x)`:
    - Traverses the linked list starting from the leader and prints all elements in the set.

Advantages and Disadvantages

Advantages:

- Simple to implement.
- Intuitive representation of sets.

Disadvantages:

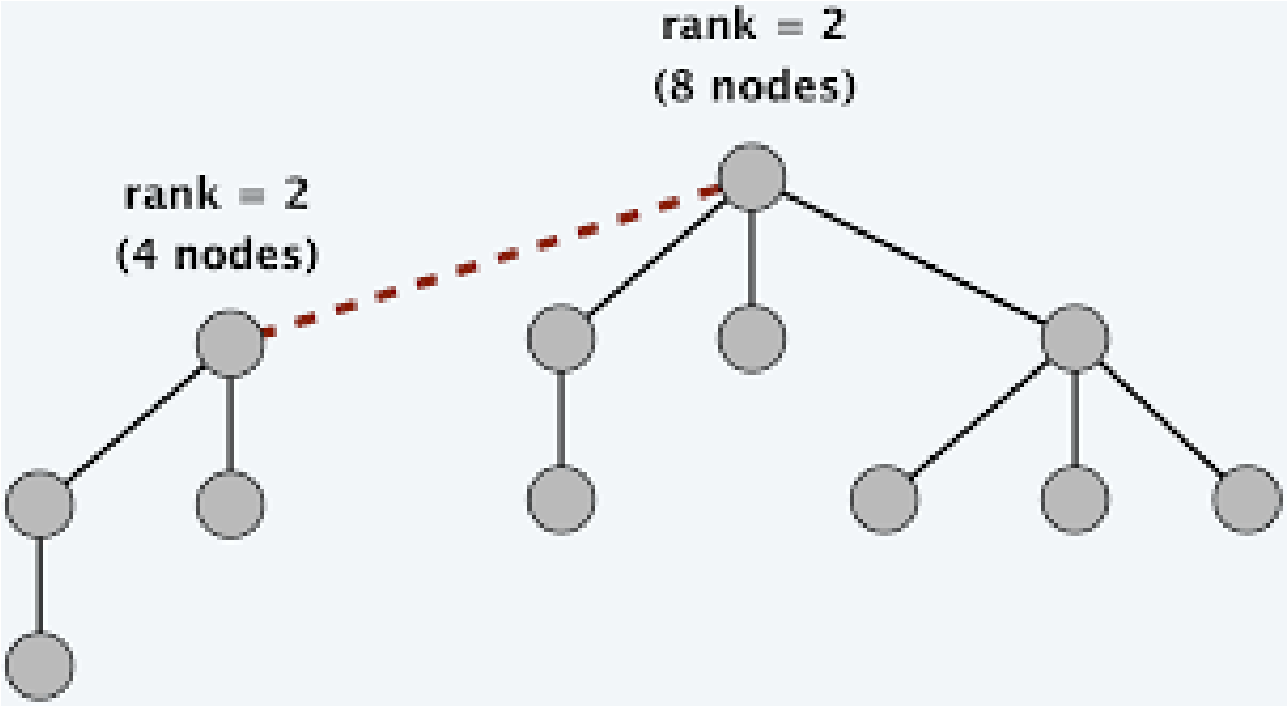
- The `find_set` operation is slow (`O(n)`) due to the need to traverse the entire list.
- Updating all elements in one set during `union_sets` can be inefficient.

Conclusion

The linked list representation of disjoint sets is useful for understanding the basic concept, but it is inefficient compared to more advanced implementations like the forest representation with union by rank and path compression, which achieve nearly constant time for both `union` and `find`.

Forest representation

The forest representation is a highly efficient way to implement disjoint sets. In this approach, each set is represented as a tree, where every node points to its parent. The root of the tree serves as the representative of the set. Two critical optimizations, union by rank and path compression, ensure that operations are nearly constant time.



Key Concepts

1. Tree Structure:
    - Each set is represented as a tree.
    - Each element has a parent pointer.
    - The root node of the tree is the leader/representative of the set.
    - A single array (`parent`) stores the parent of each element.
2. Optimizations:
    - Path Compression: During the `find_set` operation, the tree is flattened by directly linking every node to the root. This makes future lookups faster.

- Union by Rank: When two sets are merged, the smaller tree is attached to the root of the larger tree (based on rank or height). This keeps the tree shallow.

Operations

1. `make_set(x)`:
   - Creates a new set with a single element `x`.
   - The parent of `x` is set to itself, and its rank is initialized to 0.
2. `find_set(x)`:
   - Returns the representative of the set containing `x`.
   - Uses path compression to make future queries faster by linking all nodes on the path directly to the root.
3. `union_sets(a, b)`:
   - Merges the sets containing `a` and `b`.
   - Uses union by rank to attach the smaller tree to the root of the larger tree.

Code Implementation

Below is the implementation in C++:

```
#include <iostream>
#include <vector>
using namespace std;

const int MAX_SIZE = 1000; // Adjust based on the number of
elements
int parent[MAX_SIZE];      // Stores the parent of each
element
int rank[MAX_SIZE];        // Stores the rank (or height) of
each tree
```

```
// Create a new set with a single element
void make_set(int x) {
    parent[x] = x; // Parent of x is itself
    rank[x] = 0;   // Rank (height) is initially 0
}

// Find the representative of the set containing x
int find_set(int x) {
    if (x != parent[x]) {
        parent[x] = find_set(parent[x]); // Path compression
    }
    return parent[x];
}

// Union of two sets containing a and b
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);

    if (a != b) {
        // Union by rank
        if (rank[a] < rank[b]) {
            parent[a] = b; // Attach smaller tree to larger
tree
        } else if (rank[a] > rank[b]) {
            parent[b] = a;
        } else {
            parent[b] = a;
            rank[a]++; // Increase rank when ranks are equal
        }
    }
```

```
}

// Example usage
int main() {
    int n = 10; // Number of elements
    for (int i = 1; i <= n; i++) {
        make_set(i);
    }

    // Perform some union operations
    union_sets(1, 2);
    union_sets(2, 3);
    union_sets(4, 5);
    union_sets(6, 7);
    union_sets(5, 6);

    // Find representatives of sets
    cout << "Representative of 3: " << find_set(3) << endl;
    cout << "Representative of 7: " << find_set(7) << endl;

    // Check if two elements are in the same set
    if (find_set(3) == find_set(7)) {
        cout << "3 and 7 are in the same set." << endl;
    } else {
        cout << "3 and 7 are in different sets." << endl;
    }

    return 0;
}
```

Explanation of the Code

1. Initialization:
    - `make_set(x)` initializes each element as its own parent, and the rank is set to 0.
2. Path Compression in `find_set`:
    - While traversing up the tree to find the root, the `find_set` operation directly connects each node on the path to the root. This reduces the height of the tree drastically.
3. Union by Rank:
    - In `union_sets(a, b)`, the smaller tree (based on rank) is attached to the root of the larger tree. If both trees have the same rank, one root becomes the parent, and its rank is incremented.

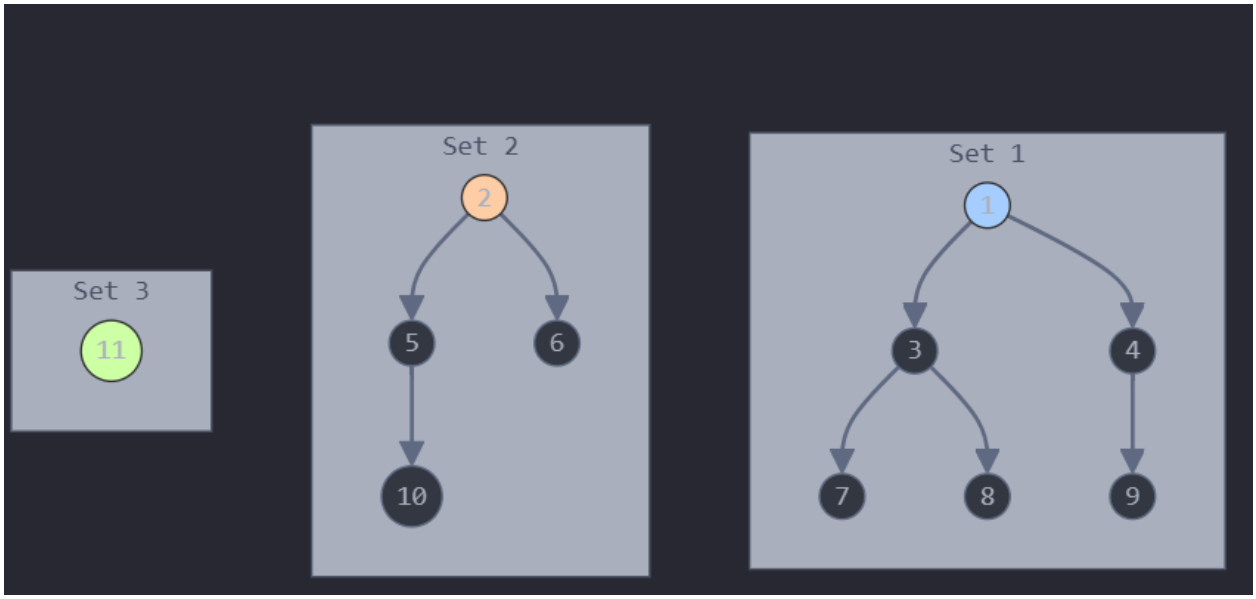
Advantages

1. Efficiency:
    - Both `find_set` and `union_sets` operations run in nearly constant time, specifically O(α(n))O(\alpha(n))O(α(n)), where α(n)\alpha(n)α(n) is the inverse Ackermann function (extremely small for all practical inputs).
2. Scalability:
    - Works well for large datasets and dynamic connectivity problems.

Example Execution

- Initial State:
    - `make_set(1)`, `make_set(2)`, ..., `make_set(7)` create individual sets:
        - `parent: [1, 2, 3, 4, 5, 6, 7]`
        - `rank: [0, 0, 0, 0, 0, 0, 0]`
- Union Operations:
    - `union_sets(1, 2)`: Attach 2 to 1, increase rank of 1:
        - `parent: [1, 1, 3, 4, 5, 6, 7]`
        - `rank: [1, 0, 0, 0, 0, 0, 0]`

    - `union_sets(2, 3)`: Attach 3 to 1:
        - `parent: [1, 1, 1, 4, 5, 6, 7]`
    - `union_sets(5, 6)` and `union_sets(4, 5)`: Merge 4, 5, 6 into one set:
        - `parent: [1, 1, 1, 4, 4, 4, 7]`
- Path Compression:
    - After `find_set(3)`, the parent of 3 points directly to 1.

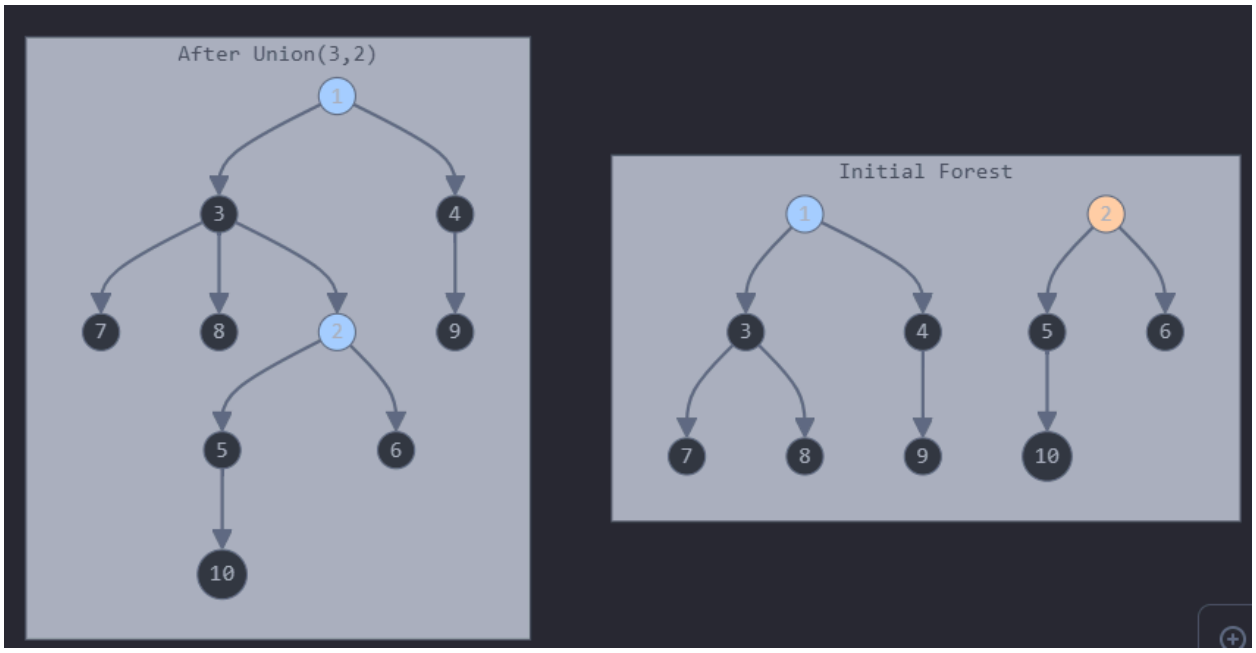


This diagram shows three disjoint sets represented as trees in a forest structure:

1. Set 1: A tree rooted at node 1, containing elements {1,3,4,7,8,9}
2. Set 2: A tree rooted at node 2, containing elements {2,5,6,10}
3. Set 3: A single-node tree with element {11}

Key characteristics shown in this representation:

- Each tree represents a separate set
- The root node serves as the representative element for the set
- Elements in the same set are connected in a tree structure
- Each element points to its parent (pointing upward in the tree)
- Different colors are used to distinguish different sets



1. FIND Operation:

- Purpose: Determines which set an element belongs to by finding the root of its tree
- Process:
    1. Start at the element
    2. Follow parent pointers until reaching the root (a node that points to itself)
    3. The root is the representative element of the set
- Example: FIND(8) would follow 8 → 3 → 1, returning 1 as the representative

2. UNION Operation:

- Purpose: Merges two sets into one
- Process:
    1. Find the roots of both sets using FIND
    2. Make one root point to the other (usually the smaller tree points to the larger one)
- Example in diagram: UNION(3,2)
    1. Find root of 3 (which is 1)
    2. Find root of 2 (which is 2)
    3. Make 2's tree point to node 3, combining the sets

Optimizations:

1. Path Compression (for FIND):
    - After finding the root, make all nodes in the path point directly to it
    - Makes future operations faster
2. Union by Rank/Size:

    - Attach the smaller tree to the root of the larger tree
    - Helps keep the tree balanced and operations efficient

Applications

1. Dynamic connectivity problems:
    - Networking (e.g., determining connected components in a graph).
2. Kruskal's Algorithm:
    - Finding a Minimum Spanning Tree (MST).
3. Image processing:
    - Grouping pixels into connected regions.
4. Social network analysis:
    - Detecting communities or friend groups.

This representation is widely used in scenarios where quick union and find operations are critical.

Minimum Spanning Tree - Prims and Kruskals

It is called a Minimum Spanning Tree, because it is a connected, acyclic, undirected graph, which is the definition of a tree data structure.

MST Thought Experiment

Let's imagine that the circles in the animation above are villages that are without electrical power, and you want to connect them to the electrical grid. After one village is given electrical power, the electrical cables must be spread out from that village to the others. The villages can be connected in a lot of different ways, each route having a different cost.

The electrical cables are expensive, and digging ditches for the cables, or stretching the cables in the air is expensive as well. The terrain can

certainly be a challenge, and then there is perhaps a future cost for maintenance that is different depending on where the cables end up.

All these route costs can be factored in as edge weights in a graph. Every vertex represents a village, and every edge represents a possible route for the electrical cable between two villages.

After such a graph is created, the Minimum Spanning Tree (MST) can be found, and that will be the most effective way to connect these villages to the electrical grid.

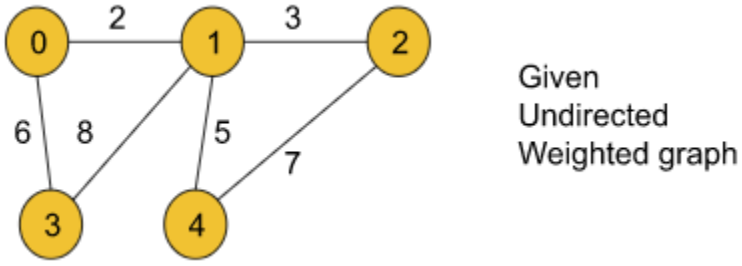
And this is actually what the first MST algorithm (Borůvka's algorithm) was made for in 1926: To find the best way to connect the historical region of Moravia, in the Check Republic, to the electrical grid.

Ref:
https://www.w3schools.com/dsa/dsa_theory_mst_minspantree.php

Minimum Spanning Tree - Theory: G-44

We will be discussing the minimum spanning tree. So, to understand the minimum spanning tree, we first need to discuss what a spanning tree is.

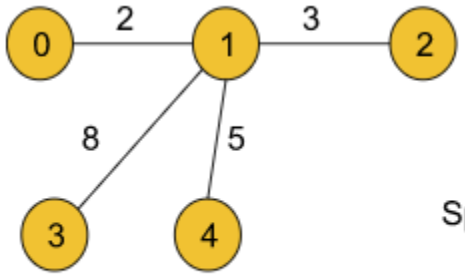
Let's further discuss this below:



Spanning Tree:

A spanning tree is a tree in which we have N nodes(i.e. All the nodes present in the original graph) and N-1 edges and all nodes are reachable from each other.

Let’s understand this using an example. Assume we are given an undirected weighted graph with N nodes and M edges. Here in this example, we have taken N as 5 and M as 6.



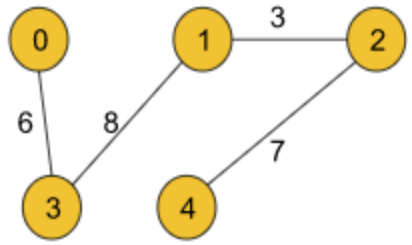
There are 5 nodes and 4 edges, and all nodes are reachable from each other. So, this is definitely a spanning tree.

Spanning Tree I

Sum of edge weights = 18

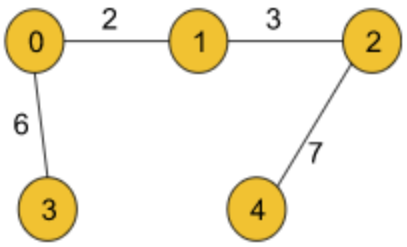
For the above graph, if we try to draw a spanning tree, the following illustration will be one:

We can draw more spanning trees for the given graph. Two of them are like the following:



Spanning Tree II

Sum of edge weights = 24



Spanning Tree III

Sum of edge weights = 18

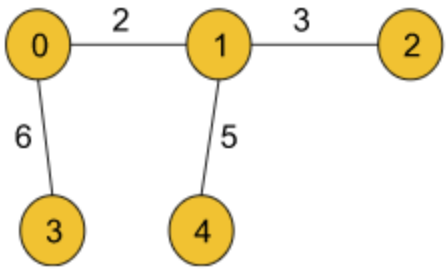
Note: Point to remember is that a graph may have more than one spanning trees.

All the above spanning trees contain some edge weights. For each of them, if we add the edge weights we can get the sum for that particular tree. Now, let's try to figure out the minimum spanning tree:

Minimum Spanning Tree:

Among all possible spanning trees of a graph, the minimum spanning tree is the one for which the sum of all the edge weights is the minimum.

Let's understand the definition using the given graph drawn above. Until now, for the given graph we have drawn three spanning trees with the sum of edge weights 18, 24, and 18. If we can draw all possible spanning trees, we will find that the following spanning tree with the minimum sum of edge weights 16 is the minimum spanning tree for the given graph:



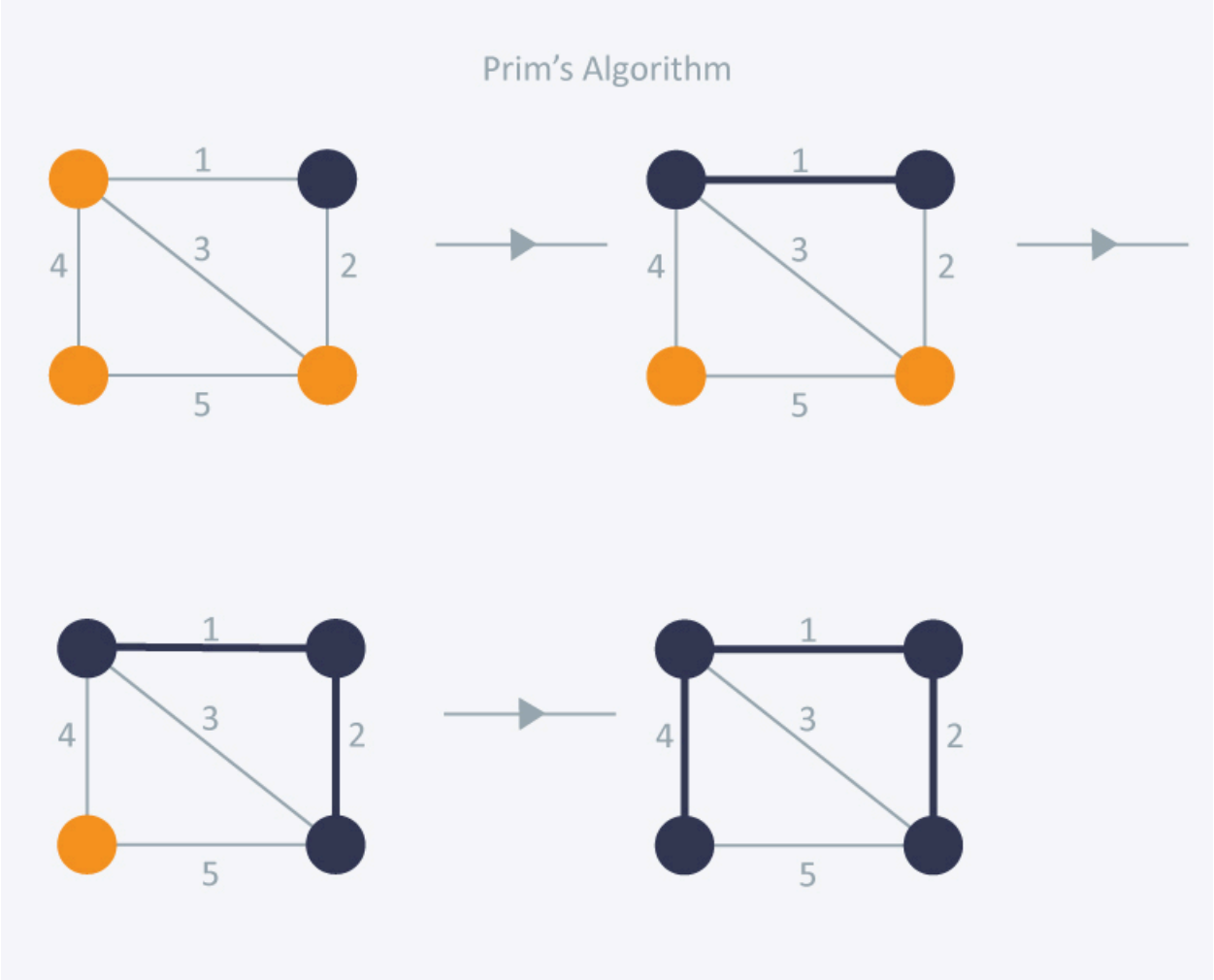
Minimum spanning tree
Sum of edge weights = 16

Note: There may exist multiple minimum spanning trees for a graph like a graph may have multiple spanning trees.

Prims MST

How it works:

1. Choose a random vertex as the starting point, and include it as the first vertex in the MST.
2. Compare the edges going out from the MST. Choose the edge with the lowest weight that connects a vertex among the MST vertices to a vertex outside the MST.
3. Add that edge and vertex to the MST.
4. Keep doing step 2 and 3 until all vertices belong to the MST.



In Prim's Algorithm, **we will start with an arbitrary node** and mark it.

In each iteration we will mark a new vertex that is adjacent to the one that we have already marked.

As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1.

In the next iteration we have three options, edges with weight 2, 3 and 4.

So, we will select the edge with weight 2 and mark the vertex

. Now again we have three options, edges with weight 3, 4 and 5.

But we can't choose edge with weight 3 as it is creating a cycle.

So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 (= 1 + 2 +4).

Unset

```
#include <iostream>

#include <vector>

#include <queue>

#include <functional>

#include <utility>


using namespace std;

const int MAX = 1e4 + 5;

typedef pair<long long, int> PII;

bool marked[MAX];

vector <PII> adj[MAX];


long long prim(int x)

{

    priority_queue<PII, vector<PII>, greater<PII> > Q;

    int y;

    long long minimumCost = 0;

    PII p;

    Q.push(make_pair(0, x));

    while(!Q.empty())

    {
```

```
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
```

```
    long long weight, minimumCost;

    cin >> nodes >> edges;

    for(int i = 0;i < edges;++i)

    {

        cin >> x >> y >> weight;

        adj[x].push_back(make_pair(weight, y));

        adj[y].push_back(make_pair(weight, x));

    }

    // Selecting 1 as the starting node

    minimumCost = prim(1);

    cout << minimumCost << endl;

    return 0;

}
```

Introduction to Kruskal's Algorithm:

Ref: https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Illustration:
https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Kruskal's algorithm[1] finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. It is a greedy algorithm that in each step adds to the forest the lowest-weight edge that will not form a cycle.[2] The key steps of the algorithm are sorting and the use of a disjoint-set data structure to detect cycles. Its running time is dominated by the time to sort all of the graph edges by their weight.

A minimum spanning tree of a connected weighted graph is a connected subgraph, without cycles, for which the sum of the weights of all the edges in the subgraph is minimal. For a disconnected graph, **a minimum spanning forest is composed of a minimum spanning tree for each connected component.**

Kruskal's Algorithm Time Complexity and Efficiency

Kruskal's algorithm is a well-known algorithm for finding the minimum spanning tree of a graph. It is a greedy algorithm that makes use of the fact that the edges of a minimum spanning tree must form a subset of the edges of any other spanning tree.

The time complexity of Kruskal's Algorithm is O(ElogE), where E is the number of edges in the graph. This complexity is because the algorithm uses a priority queue with a time complexity of O(logE). However, the space complexity of the algorithm is O(E), which is relatively high.

Kruskal's Algorithm Applications

Kruskal's algorithm is popular in computer science for finding the minimum spanning tree in a graph. A greedy algorithm selects the cheapest edge that does not form a cycle in the graph. The following are some of the applications of Kruskal's algorithm:

- Network Design: Kruskal's algorithm can be used to design networks with the least cost. It can be used to find the least expensive network connections that can connect all the nodes in the network.

- Approximation Algorithms: Kruskal's algorithm can find approximate solutions to several complex optimization problems. It can also solve the traveling salesman problem, the knapsack problem, and other NP-hard optimization problems.
- Image Segmentation: Image segmentation is partitioning an image into multiple segments. Kruskal's algorithm can be used to break down an image into its constituent parts in an efficient manner.
- Clustering: Clustering is grouping data points based on their similarity.

Conclusion

Mastering Kruskal's Algorithm opens doors to understanding fundamental concepts in graph theory and tackling real-world problems efficiently. From network design to clustering in machine learning, this algorithm's applications are vast and impactful. By learning Kruskal's Algorithm from scratch, you are laying a strong foundation for more advanced topics in computer science.

How Does Kruskal's Algorithm Work?

Kruskal's Algorithm is a greedy algorithm for finding the Minimum Spanning Tree (MST) of a connected, weighted graph. It works by selecting the edges with the smallest weights and adding them to the spanning tree, provided they do not form a cycle.

Steps of Kruskal's Algorithm

1. Sort the Edges by Weight

- List all the edges in the graph and sort them in ascending order of their weights.
- This ensures that the edges with the smallest weights are considered first.

2. Initialize the Spanning Tree

- Start with an empty graph containing no edges.
- Ensure that each vertex is treated as an independent subset (using a Union-Find data structure).

3. Iterate Through the Edges

- For each edge (starting from the smallest weight):
    - Check if adding the edge to the spanning tree creates a cycle using the Union-Find technique.
    - If it doesn’t form a cycle, add the edge to the MST.
    - If it forms a cycle, skip the edge.

4. Repeat Until MST is Formed

- Continue adding edges until the spanning tree has exactly (V - 1) edges, where V is the number of vertices.

Example Walkthrough

Graph:

Vertices: {A, B, C, D}

Edges (with weights):

- A–B (1), B–C (4), A–C (3), C–D (2)

Step-by-Step Execution:

1. Sort edges by weight:

- A–B (1), C–D (2), A–C (3), B–C (4)

2. Initialize MST:

MST starts empty, and each vertex is its own subset.

3. Add edges to MST:

- Add A–B (1): No cycle is formed. Include it in MST.
- Add C–D (2): No cycle is formed. Include it in MST.
- Add A–C (3): No cycle is formed. Include it in MST.
- Skip B–C (4): Adding it would form a cycle.

4. Final MST:

- Edges in MST: A–B, C–D, A–C
- Total weight: 1 + 2 + 3 = 6

Key Concepts in Kruskal's Algorithm

- Greedy Strategy: Always select the edge with the smallest weight first.
- Union-Find Data Structure: Used to detect cycles efficiently:
    - Find: Determines the root of a vertex's subset.
    - Union: Merges two subsets when an edge connects them.

Complexity Analysis

1. Sorting Edges:

O(ElogE), where E is the number of edges

2. Union-Find Operations:

O(ElogV), where V is the number of vertices

3. Overall Time Complexity:

O(ElogE+ElogV)≈O(ElogV) for sparse graphs.

Ref: https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm

Algorithm[edit]

The algorithm performs the following steps:

- Create a forest (a set of trees) initially consisting of a separate single-vertex tree for each vertex in the input graph.
- Sort the graph edges by weight.

- Loop through the edges of the graph, in ascending sorted order by their weight. For each edge:
    - Test whether adding the edge to the current forest would create a cycle.
    - If not, add the edge to the forest, combining two trees into a single tree.

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Pseudocode[edit]

The following code is implemented with a disjoint-set data structure. It represents the forest *F* as a set of undirected edges, and uses the disjoint-set data structure to efficiently determine whether two vertices are part of the same tree.

```
algorithm Kruskal(G) is
    F:= ∅
    for each v in G.V do
        MAKE-SET(v)
    for each {u, v} in G.E ordered by weight({u, v}), increasing
do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F := F ∪ { {u, v} }
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Complexity[edit]

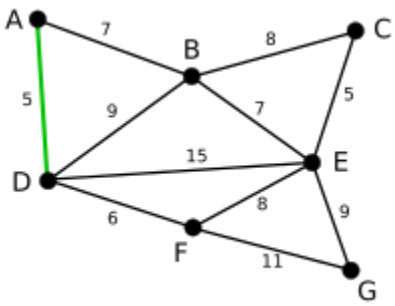
For a graph with E edges and V vertices, Kruskal's algorithm can be shown to run in time $O(E \log E)$ time, with simple data structures. This time bound is often written

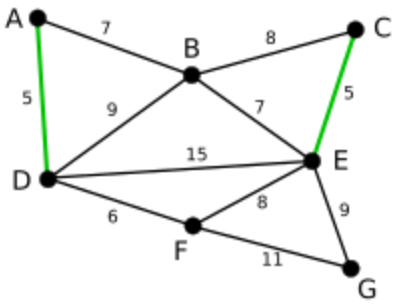
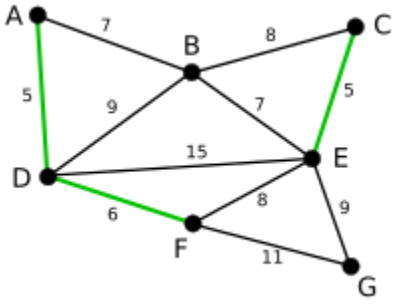
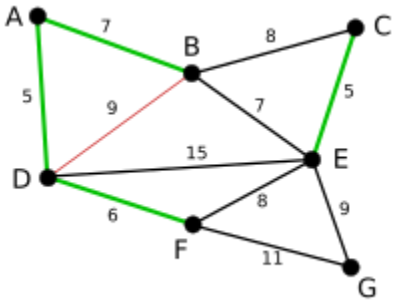
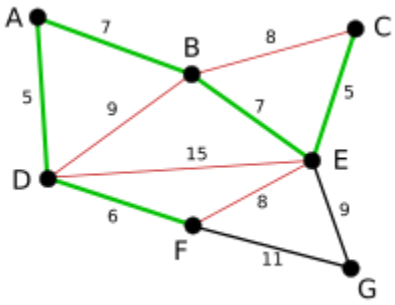
instead as $O(E \log V)$, which is equivalent for graphs with no isolated vertices, because for these graphs $V/2 \leq E < V^2$ and the logarithms of V and E are again within a constant factor of each other.

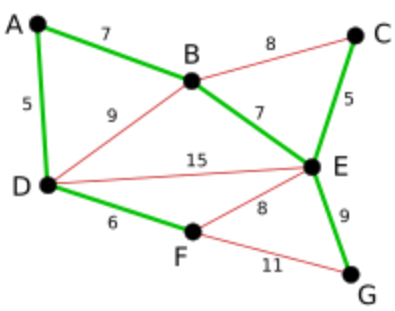
To achieve this bound, first sort the edges by weight using a comparison sort in $O(E \log E)$ time. Once sorted, it is possible to loop through the edges in sorted order in constant time per edge. Next, use a disjoint-set data structure, with a set of vertices for each component, to keep track of which vertices are in which components. Creating this structure, with a separate set for each vertex, takes V operations and $O(V)$ time. The final iteration through all edges performs two find operations and possibly one union operation per edge. These operations take amortized time $O(\alpha(V))$ time per operation, giving worst-case total time $O(E\ \alpha(V))$ for this loop, where α is the extremely slowly growing inverse Ackermann function. This part of the time bound is much smaller than the time for the sorting step, so the total time for the algorithm can be simplified to the time for the sorting step.

In cases where the edges are already sorted, or where they have small enough integer weight to allow integer sorting algorithms such as counting sort or radix sort to sort them in linear time, the disjoint set operations are the slowest remaining part of the algorithm and the total time is $O(E\ \alpha(V))$.

Example[edit]

Image	Description
	**AD** and **CE** are the shortest edges, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.

	**CE** is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.
	The next edge, **DF** with length 6, is highlighted using much the same method.
	The next-shortest edges are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The edge **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.
	The process continues to highlight the next-smallest edge, **BE** with length 7. Many more edges are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.

	Finally, the process finishes with the edge **EG** of length 9, and the minimum spanning tree is found.

Proof of correctness[edit]

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed spanning tree is of minimal weight.

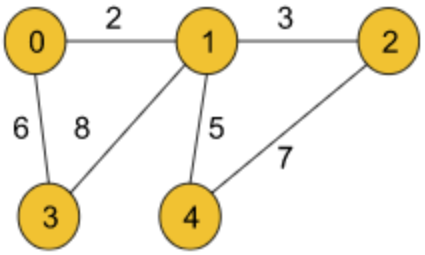
Kruskal's Algorithm - Minimum Spanning Tree : G-47

811

Problem Statement: Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree.

Example 1:

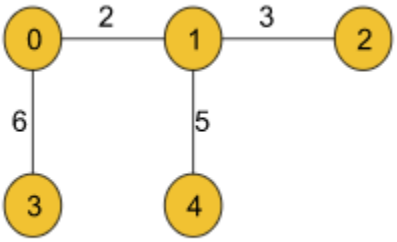
```
Input Format:

V = 5, edges = { {0, 1, 2}, {0, 3, 6}, {1, 2, 3}, {1, 3, 8}, {1, 4, 5}, {4, 2, 7}}
```



Result: 16

Explanation: The minimum spanning tree for the given graph is drawn below:

MST = {(0, 1), (0, 3), (1, 2), (1, 4)}



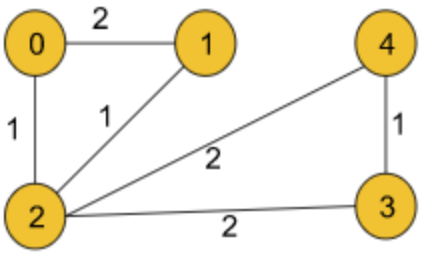
Sum of edge weights = 16

Example 2:

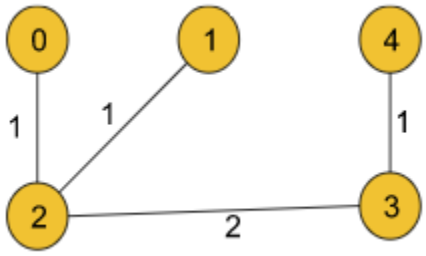
Input Format:

V = 5,

edges = { {0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}}



```
Result: 5

Explanation: The minimum spanning tree is drawn below:

MST = {(0, 2), (1, 2), (2, 3), (3, 4)}
```



Sum of edge weights = 5

Disclaimer: *Don't jump directly to the solution, try it out yourself first.*

Problem link.

Solution:

In the previous article on the minimum spanning tree, we had already discussed that there are two ways to find the minimum spanning tree for a given weighted and undirected graph. Among those two algorithms, we have already discussed Prim's algorithm.

In this article, we will be discussing another algorithm, named ***Kruskal's algorithm***, that is also useful in finding the minimum spanning tree.

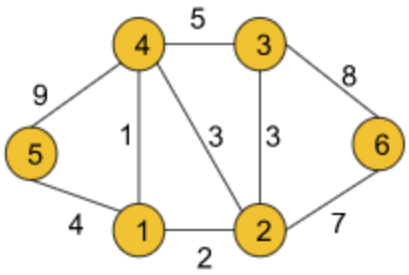
Approach:

We will be implementing Kruskal's algorithm using the ***Disjoint Set data structure*** that we have previously learned.

Now, we know Disjoint Set provides two methods named **findUPar()**(*This function helps to find the ultimate parent of a particular node*) and **Union**(*This basically helps to add the edges between two nodes*). To know more about these functionalities, do refer to the article on Disjoint Set.

The algorithm steps are as follows:

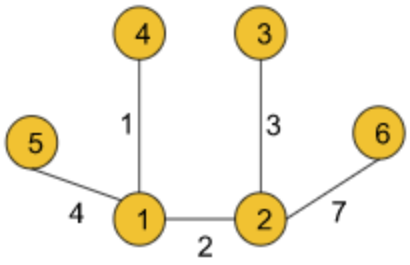
- First, we need to extract the edge information(*if not given already*) from the given adjacency list in the format of (wt, u, v) where u is the current node, v is the adjacent node and wt is the weight of the edge between node u and v and we will store the tuples in an array.
- Then the array must be sorted in the ascending order of the weights so that while iterating we can get the edges with the minimum weights first.
- After that, we will iterate over the edge information, and for each tuple, we will apply the following operation:
    - First, we will take the two nodes u and v from the tuple and check if the ultimate parents of both nodes are the same or not using the **findUPar()** function provided by the Disjoint Set data structure.
    - ***If the ultimate parents are the same***, we need not do anything to that edge as there already exists a path between the nodes and we will continue to the next tuple.
    - If the ultimate parents are different, we will add the weight of the edge to our final answer**(i.e. mstWt variable used in the following code)** and apply the **union operation(i.e. either *unionBySize(u, v)* or *unionByRank(u, v)*)** with the nodes u and v. The union operation is also provided by the Disjoint Set.
- Finally, we will get our answer (in the mstWt variable as used in the following code) successfully.



Sorted edges according to weights:

(wt	u	v)
1	1	4
2	1	2
3	2	3
3	2	4
4	1	5
5	3	4
7	2	6
8	3	6
9	4	5

The minimum spanning tree with sum of edge weights = 17



Note: *Points to remember if the graph is given as an adjacency list we must extract the edge information first. As the graph contains bidirectional edges we can get a single edge twice in our array (For example, (wt, u, v) and (wt, v, u), (5, 1, 2) and (5, 2, 1)). But we should not worry about that as the Disjoint Set data structure will automatically discard the duplicate one.*

Note: *This algorithm mainly contains the Disjoint Set data structure used to find the minimum spanning tree of a given graph. So, we just need to know the data structure.*

Ref:
https://takeuforward.org/data-structure/kruskals-algorithm-minimum-spanning-tree-g-47/

Unlike Prim's algorithm, Kruskal's algorithm can be used for such graphs that are not connected, which means that it can find more than one MST, and that is what we call a Minimum Spanning Forest.

Ref: https://www.w3schools.com/dsa/dsa_algo_mst_kruskal.php

Time Complexity for Kruskal's Algorithm

For a general explanation of what time complexity is, visit this page.

With E

as the number of edges in our graph, the time complexity for Kruskal's algorithm is

$$O(E \cdot logE)$$

We get this time complexity because the edges must be sorted before Kruskal's can start adding edges to the MST. Using a fast algorithm like Quick Sort or Merge Sort gives us a time complexity of

$O(E \cdot logE)$

for this sorting alone.

After the edges are sorted, they are all checked one by one, to see if they will create a cycle, and if not, they are added to the MST.

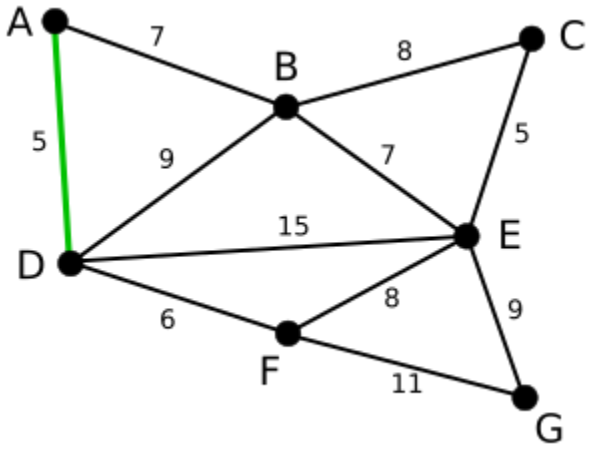
Although it looks like a lot of work to check if a cycle will be created using the `find` method, and then to include an edge to the MST using the `union` method, this can still be viewed as one operation. The reason we can see this as just one operation is that it takes approximately constant time. That means that the time this operation takes grows very little as the graph grows, and so it does actually not contribute to the overall time complexity.

Since the time complexity for Kruskal's algorithm only varies with the number of edges

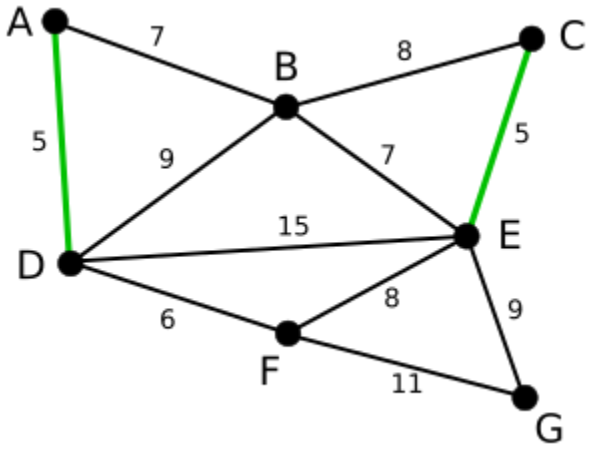
, it is especially fast for sparse graphs where the ratio between the number of edges and the number of vertices is relatively low.

Spanning tree[edit]

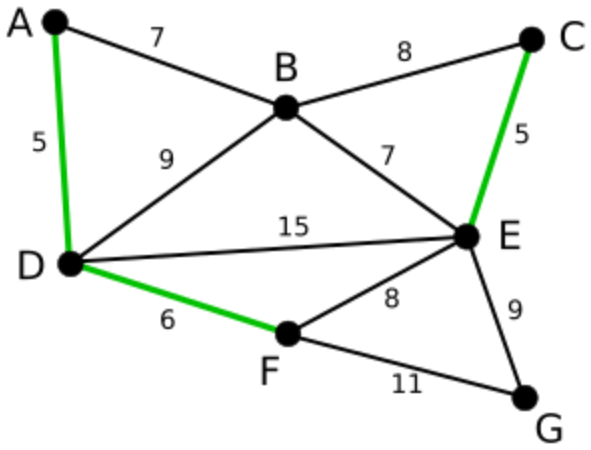
AD and **CE** are the shortest edges, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.



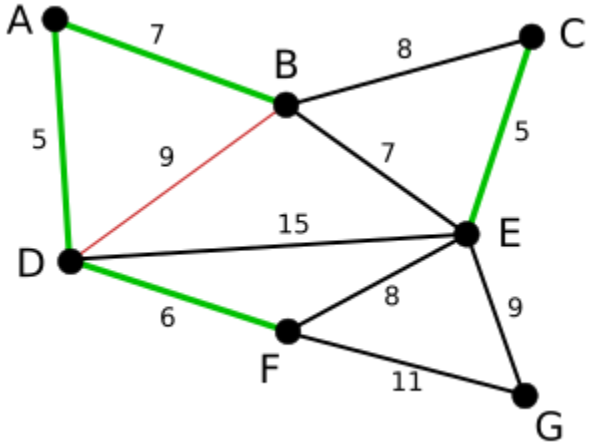
CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.



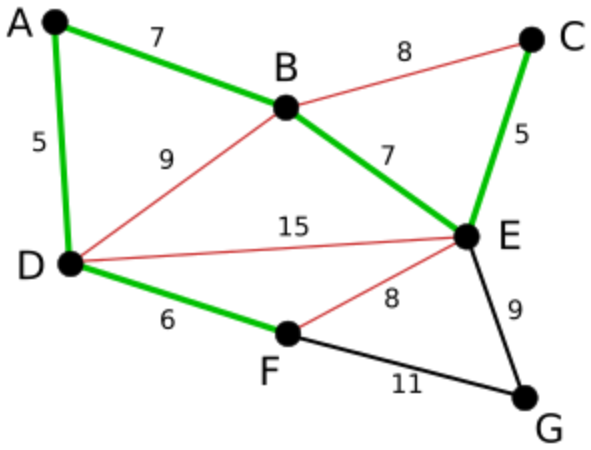
The next edge, **DF** with length 6, is highlighted using much the same method.



The next-shortest edges are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The edge **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.



The process continues to highlight the next-smallest edge, **BE** with length 7. Many more edges are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.



Finally, the process finishes with the edge **EG** of length 9, and the minimum spanning tree is found.

Minimality[edit]

We show that the following proposition ***P*** is true by induction: If *F* is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains *F* and none of the edges rejected by the algorithm.

- Clearly ***P*** is true at the beginning, when *F* is empty: any minimum spanning tree will do, and there exists one because a weighted connected graph always has a minimum spanning tree.
- Now assume ***P*** is true for some non-final edge set *F* and let *T* be a minimum spanning tree that contains *F*.
    - If the next chosen edge *e* is also in *T*, then ***P*** is true for *F* + *e*.
    - Otherwise, if *e* is not in *T* then *T* + *e* has a cycle *C*. The cycle *C* contains edges which do not belong to *F* + *e*, since *e* does not form a cycle when added to *F* but does in *T*. Let *f* be an edge which is in *C* but not in *F* + *e*. Note that *f* also belongs to *T*, since *f* belongs to *T* + *e* but not *F* + *e*. By ***P***, *f* has not been considered by the algorithm. *f* must therefore have a weight at least as large as *e*. Then *T* − *f* + *e* is a tree, and it has the same or less weight as *T*. However since *T* is a minimum spanning tree then *T* − *f* + *e* has the same weight as *T*, otherwise we get a contradiction and *T* would not be a minimum spanning tree. So *T* − *f* + *e* is a minimum spanning tree containing *F* + *e* and again ***P*** holds.

- Therefore, by the principle of induction, ***P*** holds when *F* has become a spanning tree, which is only possible if *F* is a minimum spanning tree itself.

ref: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

Here we will discuss Kruskal's algorithm to find the MST of a given weighted graph.

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

How to find MST using Kruskal's algorithm?

Below are the steps for finding MST using Kruskal's algorithm:

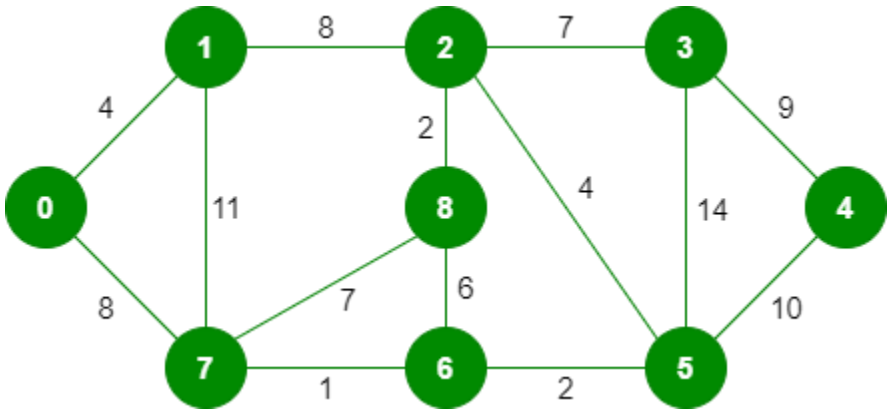
1. **Sort all the edges in non-decreasing order of their weight.**
2. **Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.**
3. **Repeat step#2 until there are (V-1) edges in the spanning tree.**

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example:

Illustration:

Below is the illustration of the above approach:

Input Graph:



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.
After sorting:

Weight	*Source*	*Destination*
1	*7*	*6*
2	*8*	*2*
2	*6*	*5*

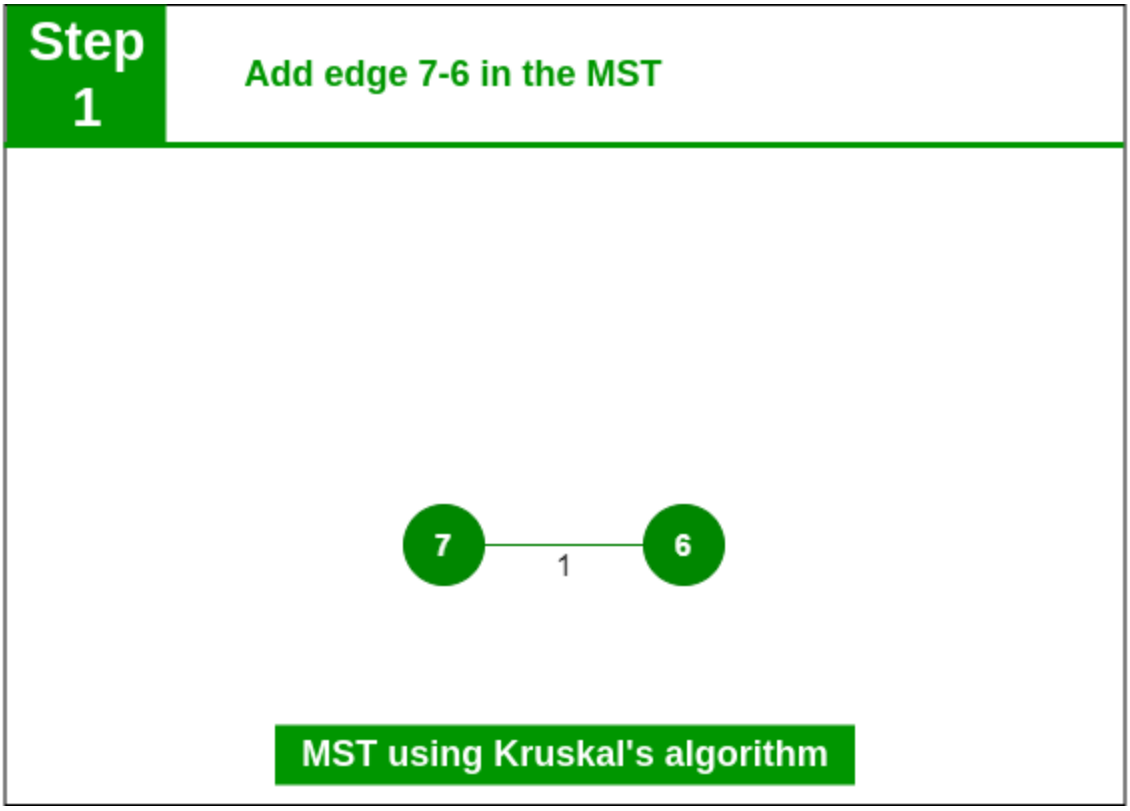
4	*0*	*1*
4	*2*	*5*
6	*8*	*6*
7	*2*	*3*
7	*7*	*8*

8	*0*	*7*
8	*1*	*2*
9	*3*	*4*
10	*5*	*4*
11	*1*	*7*

14	*3*	*5*

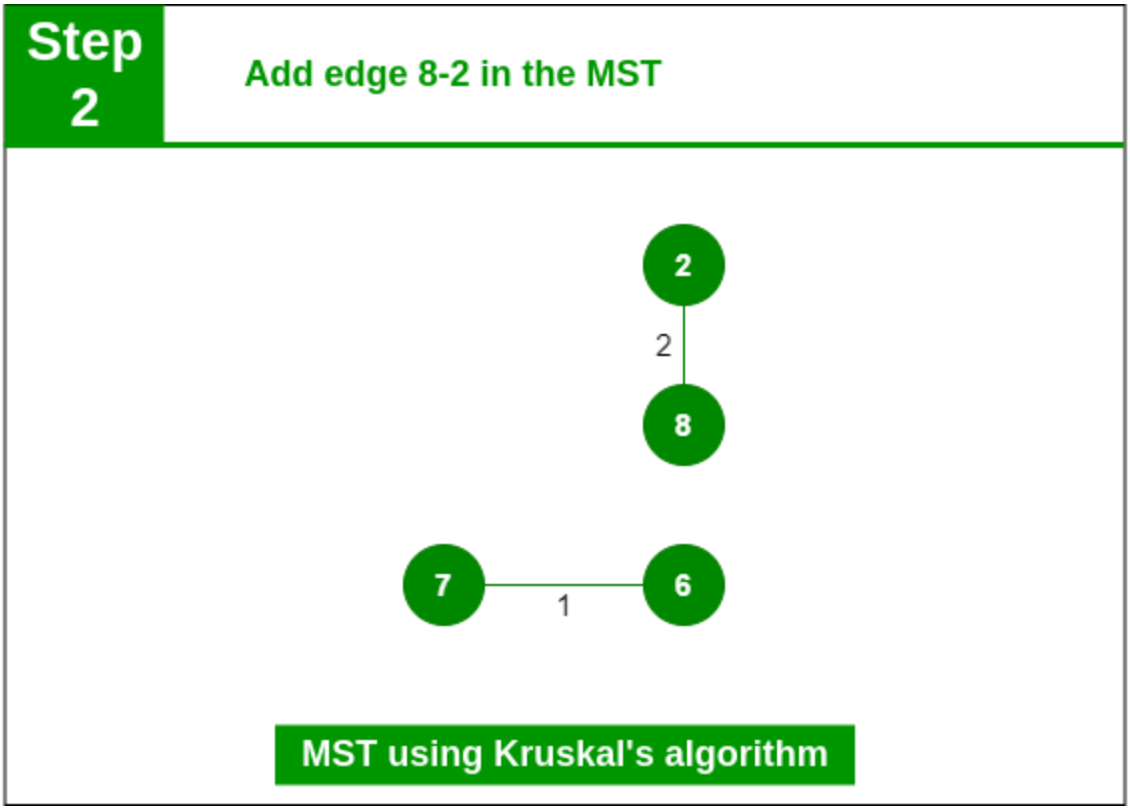
Now pick all edges one by one from the sorted list of edges

Step 1: Pick edge 7-6. No cycle is formed, include it.



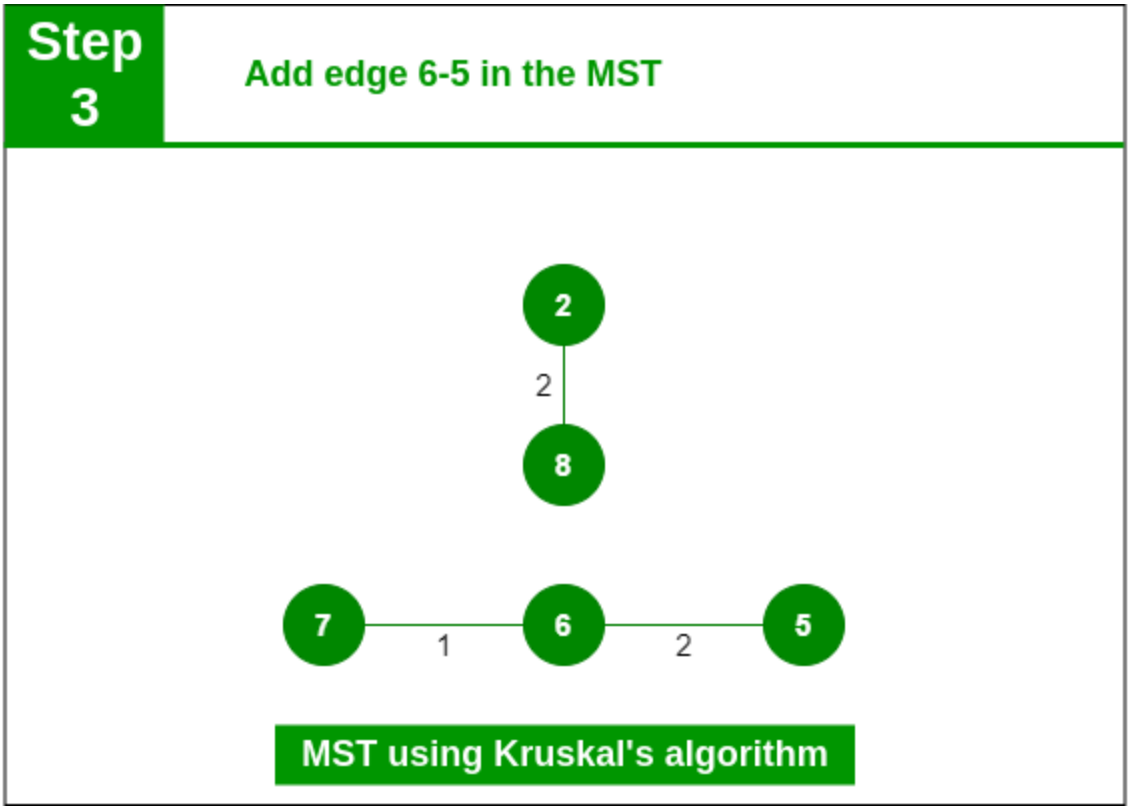
Add edge 7-6 in the MST

Step 2: Pick edge 8-2. No cycle is formed, include it.



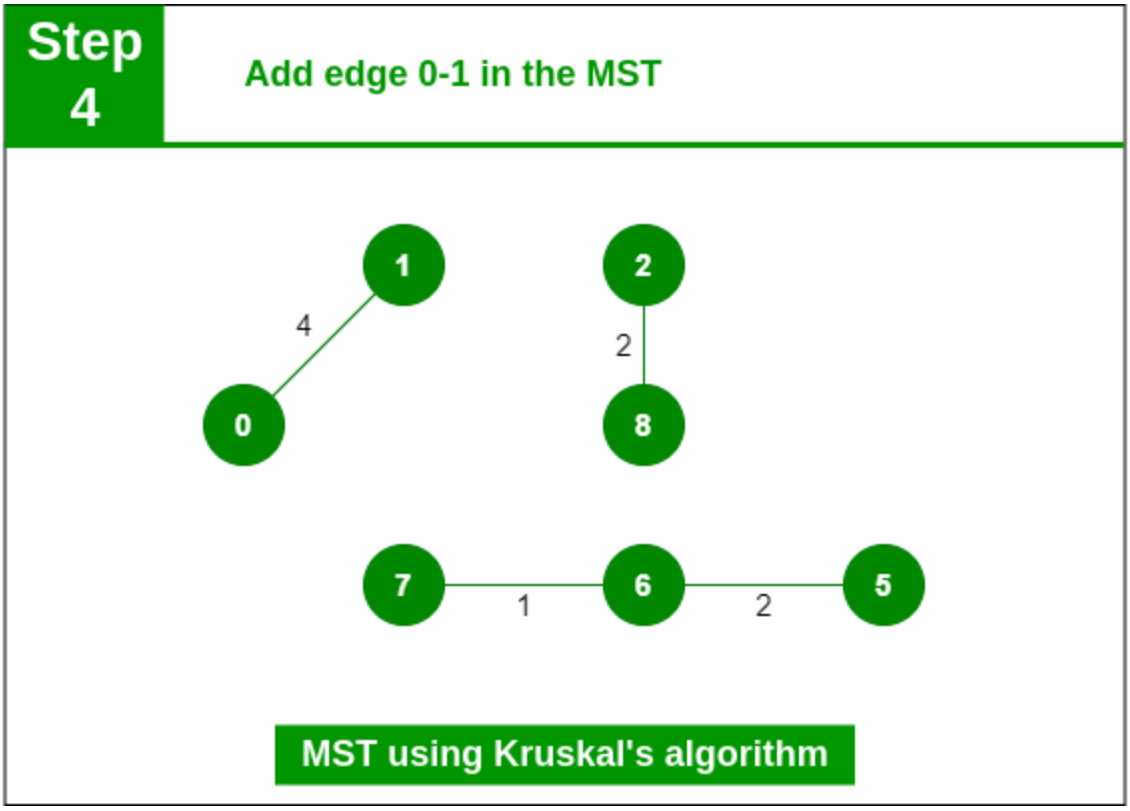
Add edge 8-2 in the MST

Step 3: Pick edge 6-5. No cycle is formed, include it.



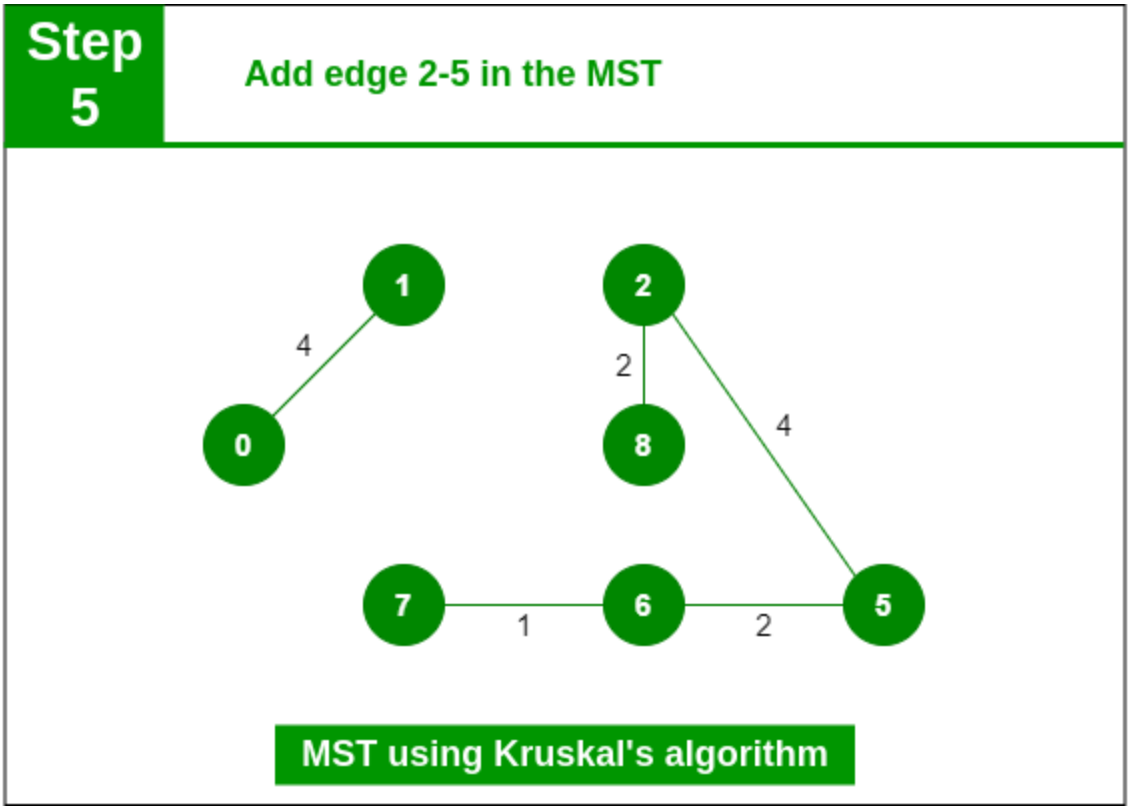
Add edge 6-5 in the MST

Step 4: Pick edge 0-1. No cycle is formed, include it.



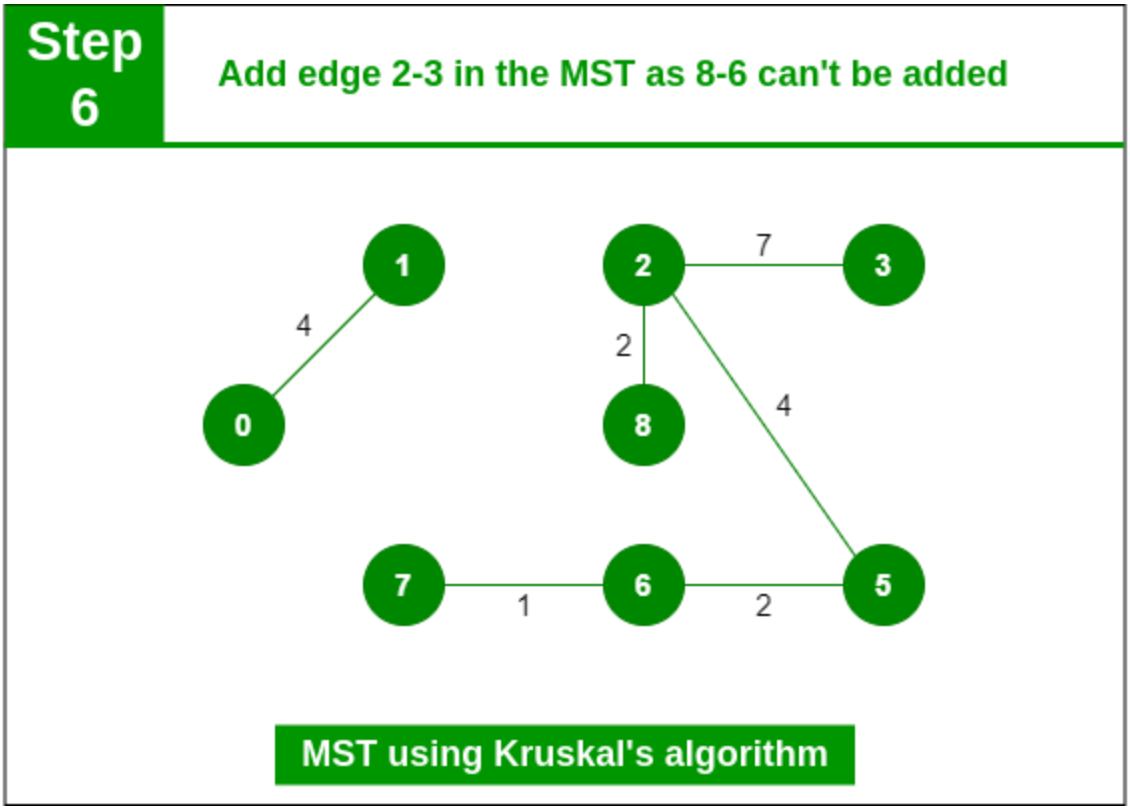
Add edge 0-1 in the MST

Step 5: Pick edge 2-5. No cycle is formed, include it.



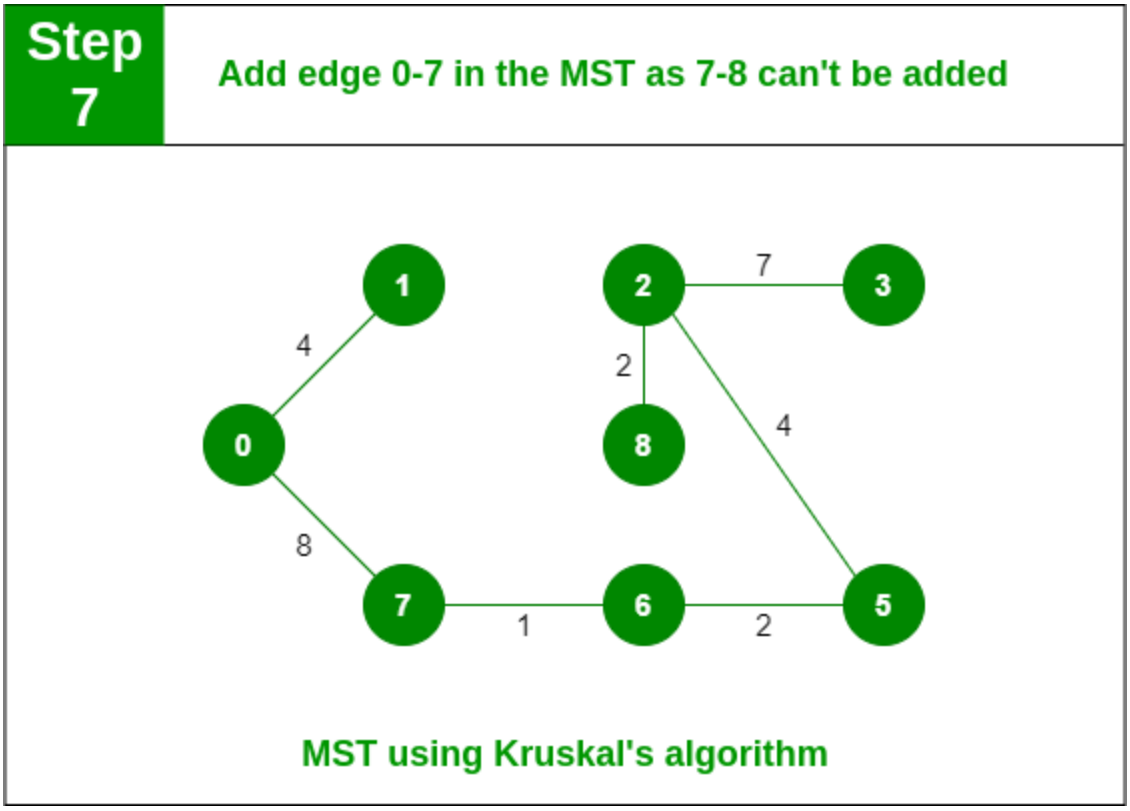
Add edge 2-5 in the MST

Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.



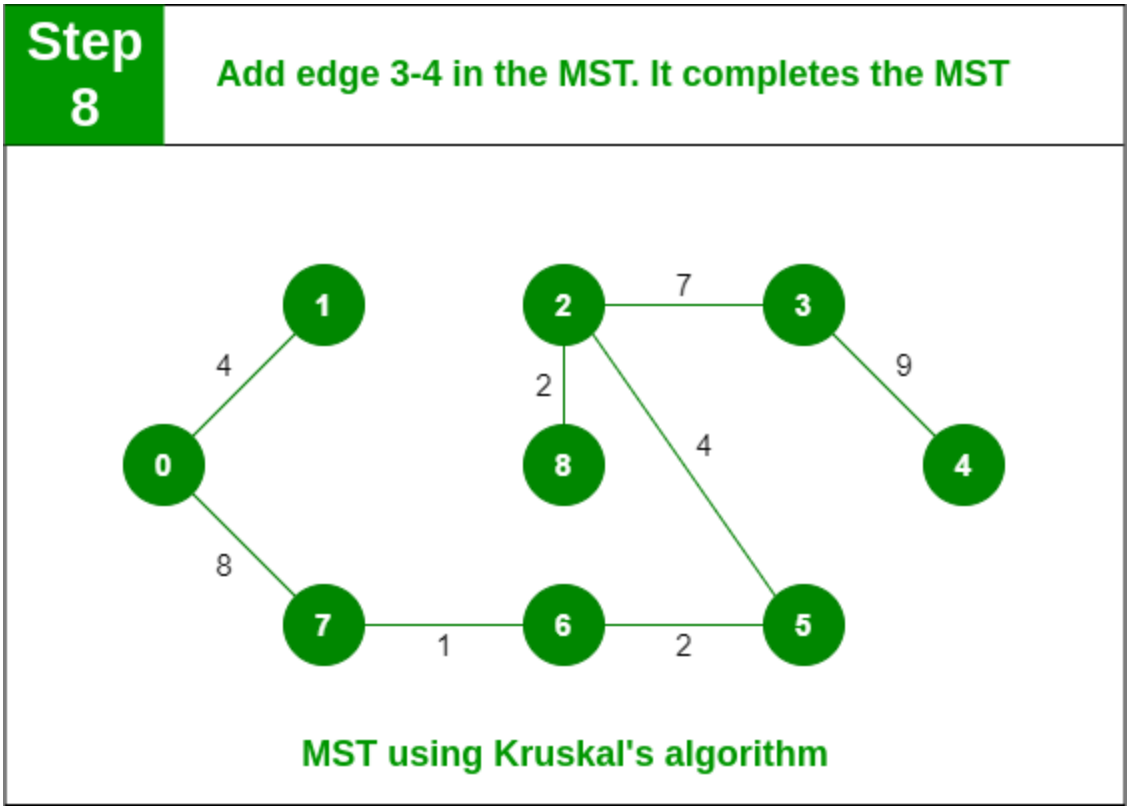
Add edge 2-3 in the MST

Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.



Add edge 0-7 in MST

Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.



Add edge 3-4 in the MST

Note: Since the number of edges included in the MST equals to (V – 1), so the algorithm stops here

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- **form a tree that includes every vertex**

- **has the minimum sum of weights among all the trees that can be formed from the graph**

How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

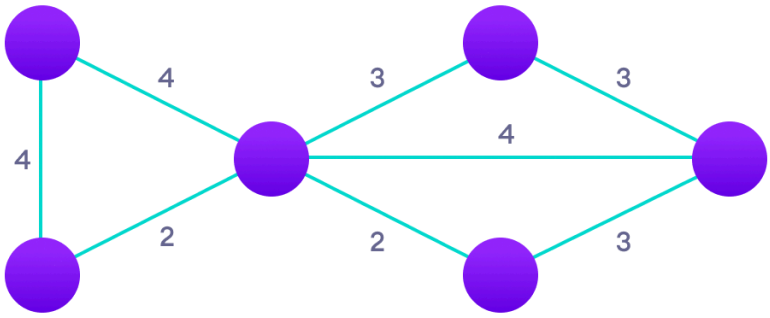
We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. **Sort all the edges from low weight to high**
2. **Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.**
3. **Keep adding edges until we reach all vertices.**

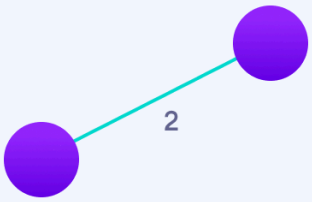
Example of Kruskal's algorithm

Ref: https://www.programiz.com/dsa/kruskal-algorithm



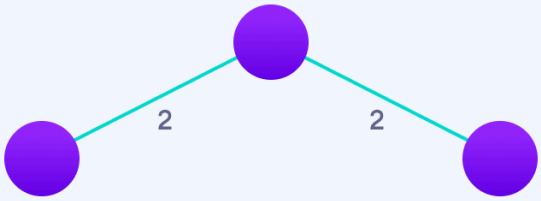
Step: 1

Start with a weighted graph



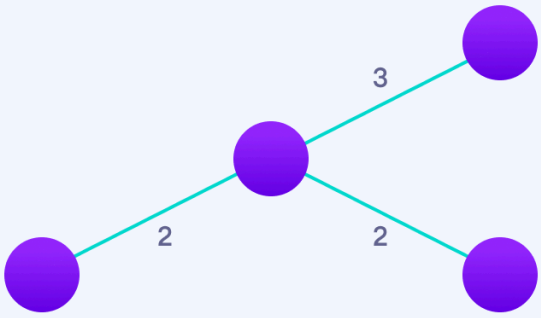
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



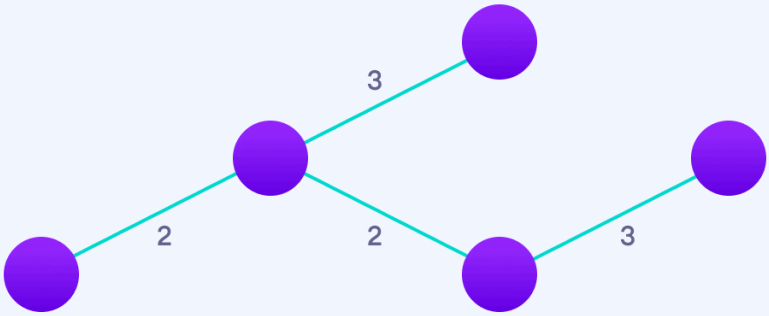
Step: 3

Choose the next shortest edge and add it



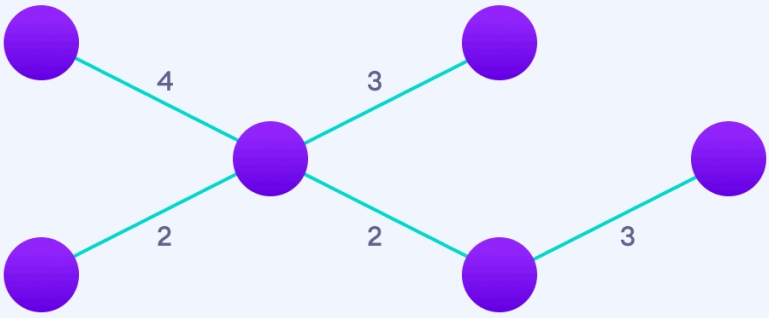
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called Union FInd. The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

```
Unset
KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V:
```

```
    MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order by
weight(u, v):

    if FIND-SET(u) ≠ FIND-SET(v):

    A = A ∪ {(u, v)}

    UNION(u, v)

return A
```

Kruskal's algorithm

This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

The simplest implementation

The following code directly implements the algorithm described above, and is having

O(MlogM+N2)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>M</mi><mo>+</mo><msup><mi>N</mi><mn>2</mn></msup><mo stretchy="false">)</mo></math>

$O(M \log M + N^2)$ **time complexity. Sorting edges requires**

O(MlogN)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>N</mi><mo stretchy="false">)</mo></math>

$O(M \log N)$ **(which is the same as**

O(MlogM)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>M</mi><mo stretchy="false">)</mo></math>

$O(M \log M)$**) operations. Information regarding the subtree to which a vertex belongs is maintained with the help of an array `tree_id[]` - for each vertex `v`, `tree_id[v]` stores the number of the tree , to which `v` belongs. For each edge, whether it belongs to the ends of different trees, can be determined in**

O(1)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mn>1</mn><mo stretchy="false">)</mo></math>

$O(1)$**. Finally, the union of the two trees is carried out in**

O(N)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>N</mi><mo stretchy="false">)</mo></math>

$O(N)$ **by a simple pass through** `tree_id[]` **array. Given that the total number of merge operations is**

N−1

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>N</mi><mo>−</mo><mn>1</mn></math>

$N-1$ **, we obtain the asymptotic behavior of**

O(MlogN+N2)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>N</mi><mo>+</mo><msup><mi>N</mi><mn>2</mn></msup><mo stretchy="false">)</mo></math>

$O(M \log N + N^2)$ ●

```
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
```

```
vector<int> tree_id(n);

vector<Edge> result;

for (int i = 0; i < n; i++)

   tree_id[i] = i;


sort(edges.begin(), edges.end());


for (Edge e : edges) {

   if (tree_id[e.u] != tree_id[e.v]) {

      cost += e.weight;

      result.push_back(e);


      int old_id = tree_id[e.u], new_id = tree_id[e.v];

      for (int i = 0; i < n; i++) {

         if (tree_id[i] == old_id)

            tree_id[i] = new_id;

      }

   }

}
```

Proof of correctness

Why does Kruskal's algorithm give us the correct result?

Ref: https://cp-algorithms.com/graph/mst_kruskal.html#practice-problems

If the original graph was connected, then also the resulting graph will be connected. Because otherwise there would be two components that could be connected with at least one edge. Though this is impossible, because Kruskal would have chosen one of these edges, since the ids of the components are different. Also the resulting graph doesn't contain any cycles, since we forbid this explicitly in the algorithm. Therefore the algorithm generates a spanning tree.

So why does this algorithm give us a minimum spanning tree?

We can show the proposal "if

F

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi></math>

$_{SFS}$ **is a set of edges chosen by the algorithm at any stage in the algorithm, then there exists a MST that contains all edges of**

F

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi></math>

$_{SFS}$**" using induction.**

The proposal is obviously true at the beginning, the empty set is a subset of any MST.

Now let's assume

F

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi></math>

$_{SFS}$ **is some edge set at any stage of the algorithm,**

T

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi></math>

T **is a MST containing**

F

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi></math>

F **and**

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e **is the new edge we want to add using Kruskal.**

If

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e **generates a cycle, then we don't add it, and so the proposal is still true after this step.**

In case that

T

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi></math>

T **already contains**

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e , the proposal is also true after this step.

In case

T

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi></math>

T doesn't contain the edge

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e , then

T+e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mo>+</mo><mi>e</mi></math>

$T+e$ will contain a cycle

C

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>C</mi></math>

C . This cycle will contain at least one edge

f

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi></math>

f, that is not in

F

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi></math>

F. The set of edges

T−f+e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mo>−</mo><mi>f</mi><mo>+</mo><mi>e</mi></math>

$T - f + e$ will also be a spanning tree. Notice that the weight of

f

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi></math>

f cannot be smaller than the weight of

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e, because otherwise Kruskal would have chosen

f

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi></math>

f earlier. It also cannot have a bigger weight, since that would make the total weight of

T−f+e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mo>−</mo><mi>f</mi><mo>+</mo><mi>e</mi></math>

$T-f+e$ **smaller than the total weight of**

T

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi></math>

T **, which is impossible since**

T

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi></math>

T **is already a MST. This means that the weight of**

e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math>

e **has to be the same as the weight of**

f

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>f</mi></math>

f **. Therefore**

T−f+e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>T</mi><mo>−</mo><mi>f</mi><mo>+</mo><mi>e</mi></math>

$T - f + e$ **is also a MST, and it contains all edges from**

F+e

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>F</mi><mo>+</mo><mi>e</mi></math>

$F + e$**. So also here the proposal is still fulfilled after the step.**

This proves the proposal. Which means that after iterating over all edges the resulting edge set will be connected, and will be contained in a MST, which means that it has to be a MST already.

Minimum spanning tree - Kruskal with Disjoint Set Union

For an explanation of the MST problem and the Kruskal algorithm, first see the main article on Kruskal's algorithm.

In this article we will consider the data structure "Disjoint Set Union" for implementing Kruskal's algorithm, which will allow the algorithm to achieve the time complexity of

O(MlogN)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>N</mi><mo stretchy="false">)</mo></math>

$O(M \log N)$ ●

Description

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the `make_set` function - it will take a total of

O(N)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>N</mi><mo stretchy="false">)</mo></math>

$O(N)$. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two `find_set` calls in

O(1)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mn>1</mn><mo stretchy="false">)</mo></math>

$O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU `union_sets` function will be called - also in

O(1)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mn>1</mn><mo stretchy="false">)</mo></math>

$O(1)$. So we get the total time complexity of

O(MlogN+N+M)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>N</mi><mo>+</mo><mi>N</mi><mo>+</mo><mi>M</mi><mo stretchy="false">)</mo></math>

$O(M \log N + N + M)$ =

O(MlogN)

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>O</mi><mo stretchy="false">(</mo><mi>M</mi><mi>log</mi><mo data-mjx-texclass="NONE">⁡</mo><mi>N</mi><mo stretchy="false">)</mo></math>

SOJM Vlog KiS ●

Implementation

Here is an implementation of Kruskal's algorithm with Union by Rank.

```
vector<int> parent, rank;


void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}


int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}


void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
```

```
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}

struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());
```

```
for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

Notice: since the MST will contain exactly

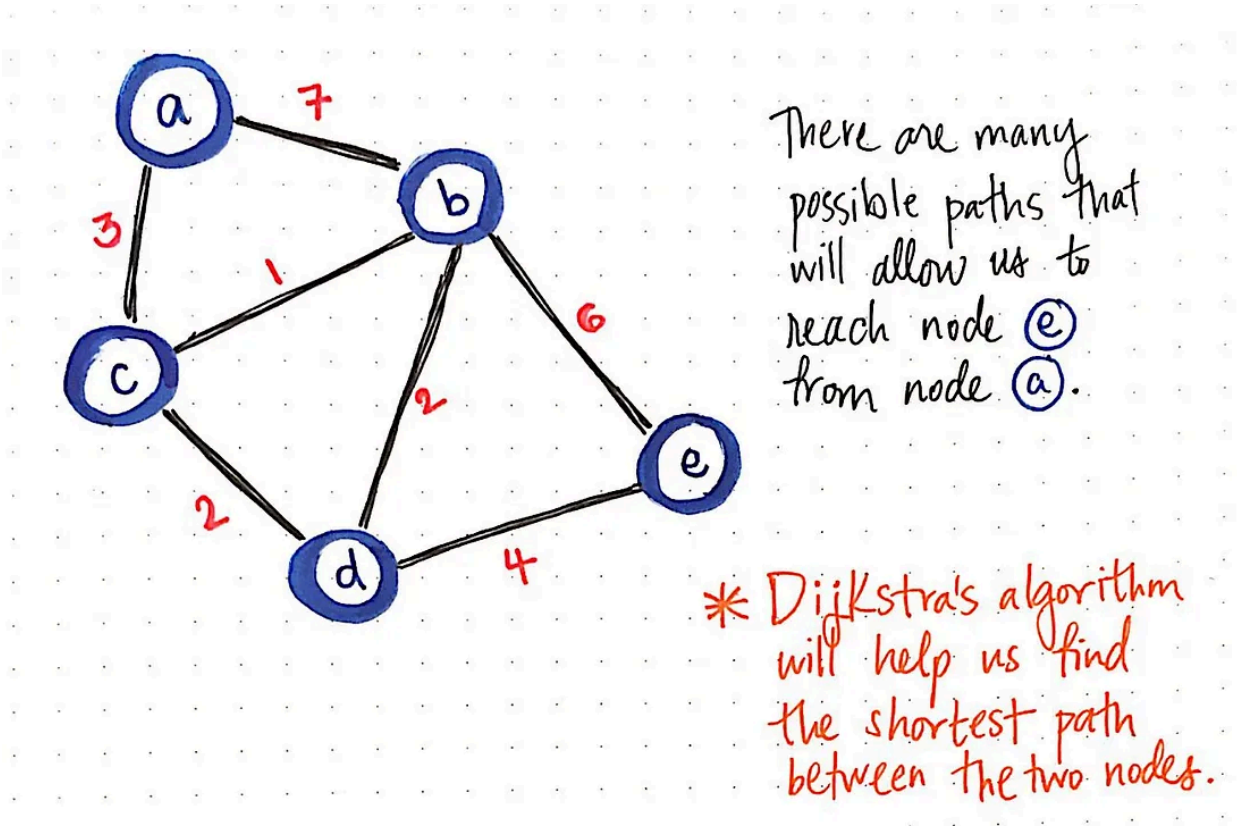
N−1

<math xmlns="http://www.w3.org/1998/Math/MathML"><mi>N</mi><mo>−</mo><mn>1</mn></math>

$N-1$ **edges, we can stop the for loop once we found that many.**

Dijkstra's algorithm

is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, a road network. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[4][5][6]



Dijkstra's algorithm finds the shortest path from a given source node to every other node.[7]:196–206 It can be used to find the shortest path to a specific destination node, by terminating the algorithm after determining the shortest path to the destination node. For example, if the nodes of the graph represent cities, and the costs of edges represent the average distances between pairs of cities connected by a direct road, then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A common application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in algorithms such as Johnson's algorithm.

The algorithm uses a min-priority queue data structure for selecting the shortest paths known so far. Before more advanced priority queue structures were discovered, Dijkstra's original algorithm ran in

$$\Theta(|V|2)$$

time, where

$$|V|$$

is the number of nodes.[8][9] Fredman & Tarjan 1984 proposed a Fibonacci heap priority queue to optimize the running time complexity to

$$\Theta(|E|+|V|\log|V|)$$

. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can be improved further. If preprocessing is allowed, algorithms such as contraction hierarchies can be up to seven orders of magnitude faster.

How it works:

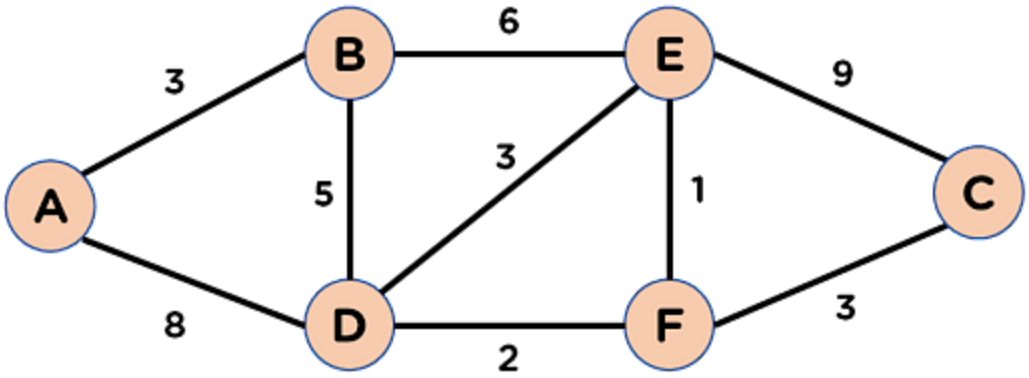
1. **Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.**
2. **Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.**
3. **For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.**
4. **We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.**
5. **Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.**
6. **In the end we are left with the shortest path from the source vertex to every other vertex in the graph.**

Applications of Dijkstra’s Algorithm

- Google maps uses Dijkstra algorithm to show shortest distance between source and destination.
- In computer networking , Dijkstra's algorithm forms the basis for various routing protocols, such as OSPF (Open Shortest Path First) and IS-IS (Intermediate System to Intermediate System).
- Transportation and traffic management systems use Dijkstra's algorithm to optimize traffic flow, minimize congestion, and plan the most efficient routes for vehicles.
- Airlines use Dijkstra's algorithm to plan flight paths that minimize fuel consumption, reduce travel time.
- Dijkstra's algorithm is applied in electronic design automation for routing connections on integrated circuits and very-large-scale integration (VLSI) chips.

Implementing Dijkstra's Algorithm

Let's apply Dijkstra's Algorithm for the graph given below, and find the shortest path from node A to node C:



Solution:

1. All the distances from node A to the rest of the nodes is ∞.

2. Calculating the distance between node A and the immediate nodes (node B & node D):

For node B,

Node A to Node B = 3

For node D,

Node A to Node D = 8

3. Choose the node with the shortest distance to be the current node from unvisited nodes, i.e., node B. Calculating the distance between node B and the immediate nodes:

For node E,

Node B to Node D = 3+5 = 8

For node E,

Node B to Node E = 3+6 = 9

4. Choose the node with the shortest distance to be the current node from unvisited nodes, i.e., node D. Calculating the distance between node D and the immediate nodes:

For node E,

Node D to Node E = 8+3 = 11 ([9 < 11] > TRUE: So, No Change)

For node F,

Node D to Node F = 8+2 = 10

5. Choose the node with the shortest distance to be the current node from unvisited nodes, i.e., node E. Calculating the distance between node E and the immediate nodes:

For node C,

Node E to Node C = 9+9 = 18

For node F,

Node E to Node F = 9+1 = 10

6. Choose the node with the shortest distance to be the current node from unvisited nodes, i.e., node F. Calculating the distance between node F and the immediate nodes:

For node C,

Node F to Node C = 10+3 = 13 ([18 < 13] FALSE: So, Change the previous value)

So, after performing all the steps, we have the shortest path from node A to node C, i.e., a value of 13 units.

Dijkstra's Algorithm

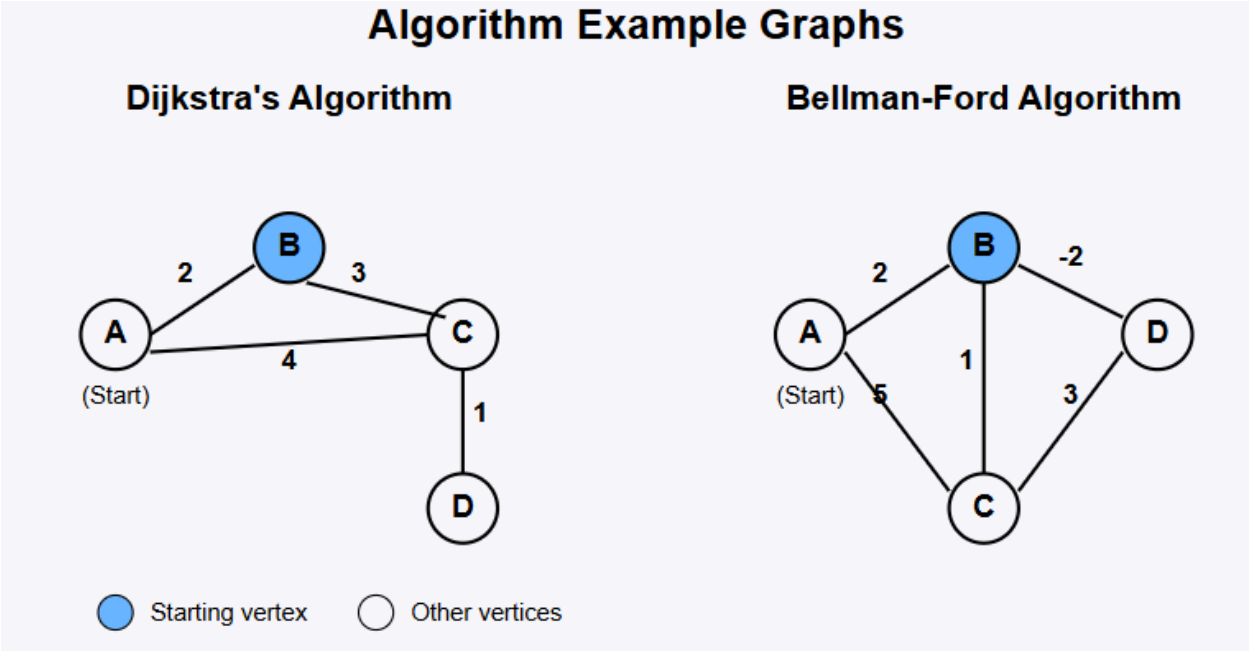
Dijkstra's algorithm is a method for finding the shortest path from a starting node to all other nodes in a weighted graph. Here's a simple explanation with an example:

How It Works

1. **Assign a tentative distance value to every node: set the starting node to 0 and all other nodes to infinity**
2. **Set the starting node as current and mark it as visited**
3. **For the current node, consider all unvisited neighbors and calculate their tentative distances**
4. **When done considering all neighbors, mark the current node as visited**
5. **Select the unvisited node with the smallest tentative distance as the new current node**
6. **Repeat steps 3-5 until all nodes are visited**

Example

Let's use this simple graph:



Where A is our starting node, and each edge has a weight (the number shown).

Step-by-Step Execution:

Initialize:

- **A: distance = 0 (starting node)**
- **B: distance = ∞**
- **C: distance = ∞**
- **D: distance = ∞**
- **Unvisited set = {A, B, C, D}**

Iteration 1:

- **Current node = A (distance = 0)**

- **Neighbors of A:**
    - **B: min(∞, 0+2) = 2**
    - **C: min(∞, 0+4) = 4**
- **Mark A as visited**
- **Unvisited set = {B, C, D}**
- **Distances: A=0, B=2, C=4, D=∞**

Iteration 2:

- **Current node = B (smallest unvisited distance = 2)**
- **Neighbors of B:**
    - **C: min(4, 2+3) = 4 (no change)**
- **Mark B as visited**
- **Unvisited set = {C, D}**
- **Distances: A=0, B=2, C=4, D=∞**

Iteration 3:

- **Current node = C (smallest unvisited distance = 4)**
- **Neighbors of C:**
    - **D: min(∞, 4+1) = 5**
- **Mark C as visited**
- **Unvisited set = {D}**
- **Distances: A=0, B=2, C=4, D=5**

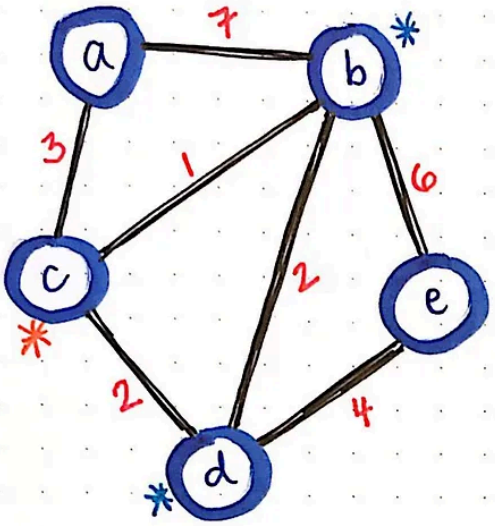
Iteration 4:

- **Current node = D (smallest unvisited distance = 5)**
- **No unvisited neighbors**
- **Mark D as visited**

- **Unvisited set = {}**
- **Distances: A=0, B=2, C=4, D=5**

Final Result:

- **Shortest path from A to B: 2**
- **Shortest path from A to C: 4**
- **Shortest path from A to D: 5**



VERTEX	SHORTEST DIST. FROM (a)	PREVIOUS VERTEX
a	0	
b	~~∞~~ 7	a
c	~~∞~~ 3	a
d	∞	
e	∞	

Visited = [a]

Unvisited = [b, (c), d, e]
↑ current vertex

* We'll next head over to vertex (c) — remember, we need to visit the node with the smallest-known cost. Since (c)'s cost is the smallest of our unvisited nodes, that's what we'll check next.

* Two out of three of (c)'s neighbors are unvisited, so we'll check them & their shortest paths, from the start vertex, via (c).

⟹
- distance to (b): 3+1 = 4
- distance to (d): 3+2 = 5

☆ Notice that the distance to (b) via node (c) is 4. This is shorter than the currently-known shortest distance in our table, which is 7.

→ We can update our shortest distance & previous vertex values for (b), since we found a better path:

b	~~∞~~ ~~7~~ 4	~~a~~ c

Ref:

https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e

Ref:

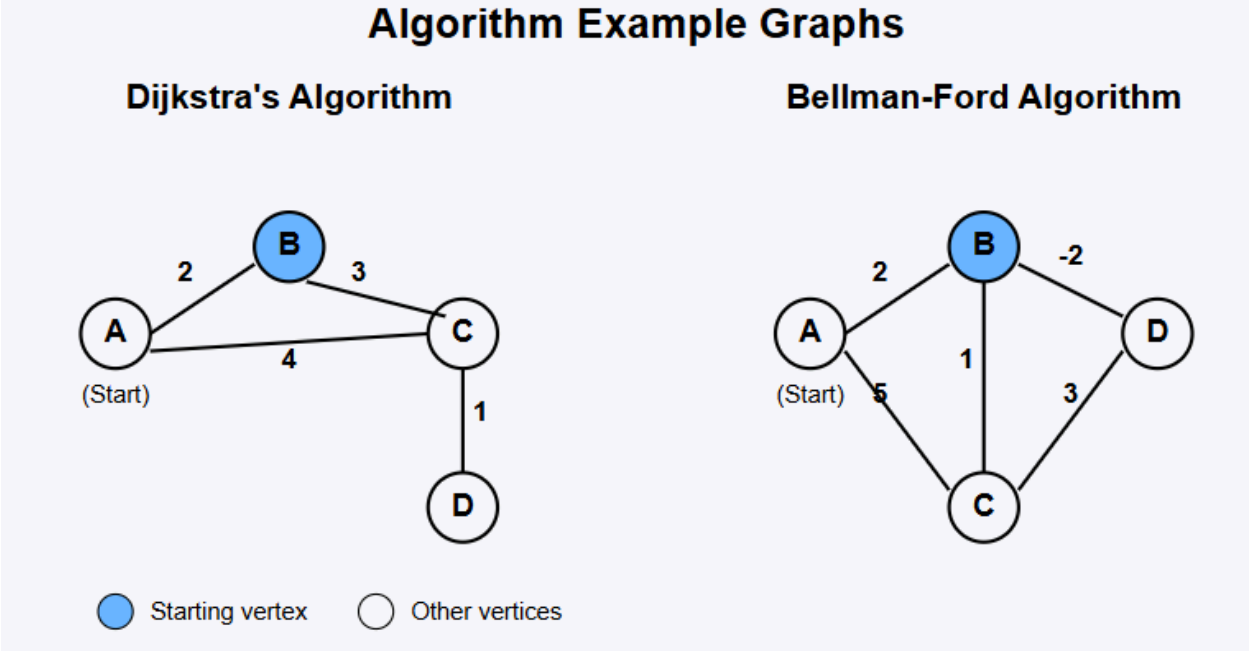
https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e

Simulator:
https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

DSA Bellman-Ford Algorithm

The two algorithms are compared which are Dijkstra and Bellman-Ford algorithms to conclude which of them is more efficient for finding the shortest path between two vertices. Our results show that the Dijkstra algorithm is much faster than the algorithm of the Bellman ford and commonly used in real-time applications.

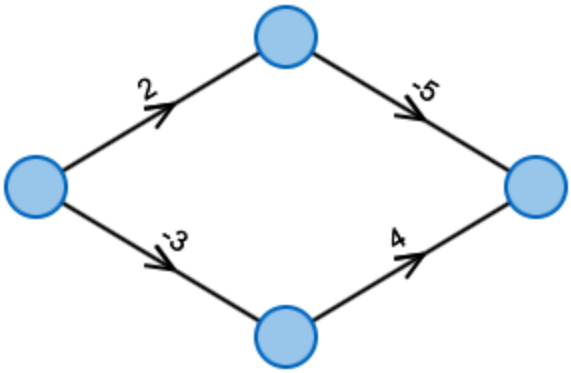
The key advantage of Bellman-Ford over Dijkstra's algorithm is its ability to handle negative weights and detect negative cycles, though it has a higher time complexity of O(|V|×|E|) compared to Dijkstra's O(|E| + |V|log|V|) with a priority queue implementation.



How It Works

1. Initialize distances from the source to all vertices as infinite and distance to the source itself as 0
2. Relax all edges |V| - 1 times, where |V| is the number of vertices
    - For each edge (u, v) with weight w, if distance[u] + w < distance[v], then update distance[v] = distance[u] + w
3. Check for negative weight cycles by trying to relax edges one more time
    - If any distance gets updated, then there is a negative weight cycle

Shortest Paths



In many applications one wants to obtain the shortest path from a to b. Depending on the context, the length of the path does not necessarily have to be the length in meter: One can as well look at the cost of a path – both if we have to pay for using it – or if we receive some.

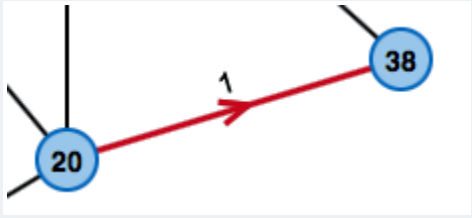
In general we speak of cost. Therefore one assigns cost to each part of the path – also called "edge".

Dijkstra's Algorithm computes shortest – or cheapest paths, if all cost are positive numbers. However, if one allows negative numbers, the algorithm will fail.

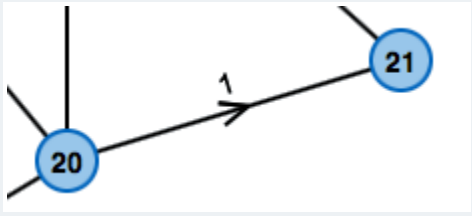
The Bellman-Ford Algorithm by contrast can also deal with negative cost.

These can for example occur when a taxi driver receives more money for a tour than he spends on fuel. If he does not transport somebody, his cost are positive.

Idea of the Algorithm



This edge is a short-cut:
We know that we have to pay 20 in order to go from the starting node to the left node. The path from the left to the right node has cost 1.
Therefore one can go from the starting node to the node on the right with a total cost of 21.



The Bellman-Ford Algorithm computes the **cost** of the cheapest paths from a starting node to all other nodes in the graph. Thus, he can also construct the paths afterwards.

The algorithm proceeds in an interactive manner, by beginning with a bad estimate of the cost and then improving it until the correct value is found.

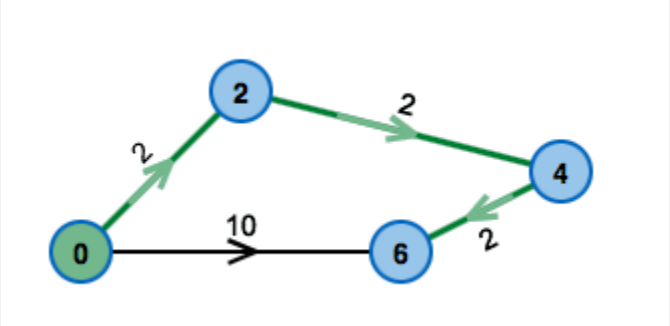
The first estimate is:

- **The starting node has cost 0**, as his distance to itself is obviously 0.
- **All other node have cost infinity**, which is the worst estimate possible.

Afterwards, the algorithm checks every edge for the following condition: **Are the cost of the source of the edge plus the cost for using the edge smaller than the cost of the edge's target?**

If this is the case, we have found a **short-cut**: It is more profitable to use the edge which was just checked, than using the path used so far. Therefore the cost of the edge's target get updated: They are set to the cost of the source plus the cost for using the edge (compare example on the right).

Looking at all edges of the graph and updating the cost of the nodes is called a **phase**. Unfortunately, it is not sufficient to look at all edges only once. After the first phase, the cost of all nodes for which the shortest path only uses one edge have been calculated correctly. After two phases all paths that use at most two edges have been computed correctly, and so on.



The green path from the starting node is the cheapest path. It uses 3 edges.

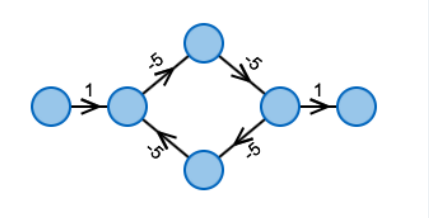
How many phases ware necessary? To answer this question, the observation that **a shortest path has to use less edges than there are nodes in the graph.** Thus, we need at most one phase less than the number of nodes in the graph. A shortest path that uses more edges than the number of nodes would visit some node twice and thus build a circle.

Construction of the shortest path

Each time when updating the cost of some node, the algorithm saves the edge that was used for the update as the **predecessor** of the node.

At the end of the algorithm, the shortest path to each node can be constructed by going backwards using the predecessor edges until the starting node is reached.

Circles with negative weight



A cheapest path had to use this circle infinitely often. The cost would be reduced in each iteration.

If the graph contains a circle with a negative sum of edge weights – a **Negative Circle**, the algorithm probably will not find a cheapest path.

As can be seen in the example on the right, paths in this case can be infinitely cheap – one keeps on going through the circle.

This problem occurs if the negative circle can be reached from the starting node. Luckily, the algorithm can detect whether a negative circle exists. This is checked in the last step of the algorithm.

A negative circle can be reached if and only if after iterating all phases, one can still find a short-cut. **Therefore, at the end the algorithm checks one more time for all edges** whether the cost of the source node plus the cost of the edge are less than the cost of the target node. If this is the case for an edge, the message "Negative Circle found" is returned.

One can even find the negative circle with the help of the predecessor edges: One just goes back until one traversed a circle (that had negative weight).

```
def bellman_ford(graph, source):
    # Step 1: Initialize distances
    distance = {node: float('infinity') for node in graph}
    distance[source] = 0

    # Step 2: Relax edges |V| - 1 times
    for _ in range(len(graph) - 1):
        for u in graph:
            for v, weight in graph[u].items():
                if distance[u] + weight < distance[v]:
                    distance[v] = distance[u] + weight

    # Step 3: Check for negative weight cycles
    for u in graph:
        for v, weight in graph[u].items():
            if distance[u] + weight < distance[v]:
                print("Graph contains negative weight cycle")
                return None

    return distance
```

```
Unset
Input: Weighted, undirected graph G=(V,E) with weight
function l.
Output: A list {d(v[j]) : j = 1,..,n} containing the
distances dist(v[1],v[j]) = d(v[j]),
         if there are no negative circles reachable from
v[1].
         The message "Negative Circle" is shown, if a
negative circle can be reached from v[1].
```

```
Unset

BEGIN
   d(v[1]) ← 0
   FOR j = 2,..,n DO
      d(v[j]) ← ∞
   FOR i = 1,..,(|V|-1) DO
      FOR ALL (u,v) in E DO
         d(v) ← min(d(v), d(u) + l(u,v))
   FOR ALL (u,v) in E DO
      IF d(v) > d(u) + l(u,v) DO
         Message: "Negative Circle"
END
```

Running time of the Bellman-Ford Algorithm

We assume that the algorithm is run on a graph with n nodes and m edges.

At the beginning, the value ∞ is assigned to each node. We need n steps for that.

Then we do the $n-1$ phases of the algorithm – one phase less than the number of nodes. In each phase, all edges of the graph are checked, and the distance value of the target node may be changed. We can interpret this check and assignment of a new value as one step and therefore have m steps in each phase. In total all phases together require $m \cdot (n-1)$ steps.

Afterwards, the algorithm checks whether there is a negative circle, for which he looks at each edge once. Altogether he needs m steps for the check.

The total running time of the algorithm is of the magnitude $m \cdot n$, as the n steps at the beginning and the m steps at the end can be neglected compared to the $m \cdot (n-1)$ steps for the phases.

Simulator
https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

Activity Selection Problem

Ref: https://www.studytonight.com/data-structures/activity-selection-problem

The Activity Selection Problem is an optimization problem which deals with the selection of non-conflicting activities that needs to be executed by a single person or machine in a given time frame.

Each activity is marked by a start and finish time. Greedy technique is used for finding the solution since this is an optimization problem.

What is Activity Selection Problem?

Let's consider that you have n activities with their start and finish times, the objective is to find solution set having maximum number of non-conflicting activities that can be executed in a single time frame, assuming that only one person or machine is available for execution.

Some points to note here:

- It might not be possible to complete all the activities, since their timings can collapse.
- **Two activities, say i and j, are said to be non-conflicting if `si >= fj` or `sj >= fi` where `si` and `sj` denote the starting time of activities i and j respectively, and `fi` and `fj` refer to the finishing time of the activities i and j respectively.**
- Greedy approach can be used to find the solution since we want to maximize the count of activities that can be executed. This approach will greedily choose an activity with earliest finish time at every step, thus yielding an optimal solution.

Input Data for the Algorithm:

- `act[]` array containing all the activities.
- `s[]` array containing the starting time of all the activities.
- `f[]` array containing the finishing time of all the activities.

Ouput Data from the Algorithm:

- `sol[]` array refering to the solution set containing the maximum number of non-conflicting activities.

Steps for Activity Selection Problem

Following are the steps we will be following to solve the activity selection problem,

Step 1: Sort the given activities in ascending order according to their finishing time.

Step 2: Select the first activity from sorted array `act[]` and add it to `sol[]` array.

Step 3: Repeat steps 4 and 5 for the remaining activities in `act[]`.

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of previously selected activity, then add it to the `sol[]` array.

Step 5: Select the next activity in `act[]` array.

Step 6: Print the `sol[]` array.

Activity Selection Problem Example

Let's try to trace the steps of above algorithm using an example:

In the table below, we have 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities:

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2

3	4	a3
0	6	a4
5	7	a5
8	9	a6

A possible solution would be:

Step 1: Sort the given activities in ascending order according to their finishing time.

The table after we have sorted it:

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3

0	6	a4
5	7	a5
5	9	a1
8	9	a6

Step 2: Select the first activity from sorted array `act[]` and add it to the `sol[]` array, thus sol = {a2}.

Step 3: Repeat the steps 4 and 5 for the remaining activities in `act[]`.

Step 4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to `sol[]`.

Step 5: Select the next activity in `act[]`

For the data given in the above table,

1. Select activity a3. Since the start time of a3 is greater than the finish time of a2 (i.e. `s(a3) > f(a2)`), we add a3 to the solution set. Thus sol = {a2, a3}.
2. Select a4. Since `s(a4) < f(a3)`, it is not added to the solution set.

3. Select a5. Since `s(a5) > f(a3)`, a5 gets added to solution set. Thus sol = {a2, a3, a5}
4. Select a1. Since `s(a1) < f(a5)`, a1 is not added to the solution set.
5. Select a6. a6 is added to the solution set since `s(a6) > f(a5)`. Thus sol = {a2, a3, a5, a6}.

Step 6: At last, print the array `sol[]`

Hence, the execution schedule of maximum number of non-conflicting activities will be:

```
(1,2)
(3,4)
(5,7)
```

```
(8,9)
```

Time Complexity Analysis

Following are the scenarios for computing the time complexity of Activity Selection Algorithm:

- Case 1: When a given set of activities are already sorted according to their finishing time, then there is no sorting mechanism involved, in such a case the complexity of the algorithm will be `O(n)`
- Case 2: When a given set of activities is unsorted, then we will have to use the `sort()` method defined in bits/stdc++ header file for sorting the activities list. The time complexity of this method will be `O(nlogn)`, which also defines complexity of the algorithm.

Real-life Applications of Activity Selection Problem

Following are some of the real-life applications of this problem:

- Scheduling multiple competing events in a room, such that each event has its own start and end time.
- Scheduling manufacturing of multiple products on the same machine, such that each product has its own production timelines.
- Activity Selection is one of the most well-known generic problems used in Operations Research for dealing with real-life business problems.

Approach 1: Greedy algorithm

Ref: https://www.scaler.in/activity-selection-problem/

- **Intuition:**
  This method uses the `greedy approach` and as the name suggests greedy means that at every step we have to make a choice that looks best at the moment and can provide us with an optimal solution for that problem. Since we have to maximize the number of performed activities, so we will be choosing the activity that will finish first as that will leave us with the maximum time to process the remaining activities. This greedy intuition enables us to make choices and provide us with an optimal solution and also helps us to get started with the solution. Therefore, our first task is to sort the activities based on their finish time. *If we try to solve this problem by sorting the activities based on start time then there might be a case where the activity with the least start time takes maximum duration to complete thus preventing us from maximizing the number of activities.*
- **Algorithm:**
  1. Sort all activities based on their finish time.
  2. Choosing the first activity from the sorted list.
  3. Select the next activity from the sorted list only if its `start` time is greater than or equal to the `finish` time of the previously selected activity.
  4. Repeat `Step 3` for all the remaining activities in the sorted list.
- **Implementation**

Python Program to solve Activity Selection Problem using Greedy Algorithm:

```
Unset
```

```
   def activitySelection(arr, n):

        # Sorting activities by finish time
        arr.sort(key = lambda x : x[1])

        # First activity will bealways selected
        i = 0
        print(arr[i], end=' ')

        for j in range(1, n):
        # If the current activity has start time >= finish time
of previously selected
        # activity, then select it
            if arr[j][0] >= arr[i][1]:
                print(arr[j], end=' ')
                i = j

    arr = [[3, 4], [2, 5], [1, 3], [5, 9], [0, 7], [11, 12],
[8, 10]]
    n = len(arr)

    print("Selected activities are :", end=' ')
    # Calling the selection function
    activitySelection(arr, n)
```

Output

```
Unset
```

```
   Selected activities are : [1, 3] [3, 4] [5, 9] [11, 12]
```

- **Complexity Analysis:**

Complexity Analysis:

Unset

```
Case 1:

When the provided list of activities is already sorted by
finish time, then no need for sorting it again. Therefore,
Time Complexity in such a case will be `O(n)`.

Case 1:

When the provided list of activities is not sorted, then we
will have to either write a `sort()` function from scratch or
we can use the in-built Standard Template Library function.
Therefore, Time Complexity, in this case, will be `O(nlogn)`.

Space Complexity:`O(1)`, Since no Auxillary space is required.
```

Algorithm for brute force approach

1. Insert all the activities in a min priority queue of pairs. We will make pairs like this {finish_time, start_time} for an activity and then insert it in the priority queue.
2. Take the top element of the priority queue and pop it. This is our first activity which is completed.
3. Store the end time of this activity in a variable end and start popping elements of the priority queue one by one.
4. Now, if the activity on the top of the priority queue has a start time greater than the end time of the previously selected activity, then cout this activity and update the end variable with the value of the end time of the currently selected activity.
5. Repeat this till the priority queue is not empty.

Algorithm for optimal approach

1. Sort the activities in increasing order of their finish time.
2. Select the activity with the least finish time and execute it.
3. Now, we will select the next activity only when the start time of the current activity would be greater than the previously selected activity's finish time.
4. After selecting an activity, we will update the finish time by the current activity's finish time.
5. Repeat this till the array is completely traversed.

Conclusion

- At every step in the Greedy Approach we have to make a choice that looks best at the moment and can provide us with an optimal solution for that problem.
- Since we had to maximize the number of performed activities, we chose the activity that will finish first as that will leave us with the maximum time to process the remaining activities.
- Time and Space Complexity for the Greedy approach when the given set of activities are not sorted is `O(nlogn)` and `O(1)` respectively.

Why will the greedy algorithm work for this problem?

REf: https://www.naukri.com/code360/library/activity-selection-problem-and-greedy-algorithm

A greedy algorithm works for the activity selection problem because of the following properties of the problem:

1. The problem has the 'greedy-choice property', which means that the locally optimal choice (the activity with the earliest finish time) leads to a globally optimal solution.

2. The problem has the 'optimal substructure' property, which means that an optimal solution to the problem can be constructed efficiently from optimal solutions to subproblems.

The greedy algorithm makes the locally optimal choice at each step by selecting the activity with the earliest finish time. This choice ensures that there is no overlap between the selected activities and that the maximum number of activities can be completed. Since the problem has the 'optimal substructure' property, a globally optimal solution can be constructed efficiently by making locally optimal choices at each step.

Introduction to Fractional Knapsack Problem

Given two arrays named **value** and **weight** of size

n

each, where **value[i]** represents the value and **weight[i]** represents the weight associated with the

ith

item. Also, we have been provided a knapsack with a maximum capacity of `W`.

The task is to pick some items (possibly all of them) such that the sum of values of all the items is maximized and the sum of weights of all the items is at most `W`.

As the name `Fractional` suggests, here we are allowed to take a fraction of an item. For example, if the weight of an

ith

item is

x

units, then we can have the following choices :

1. Do not take any fraction of the
2. ith
3. item.
4. Take the item as a whole.
5. Take any integer weight of the
6. ith
7. item
8. i.e.
9. For an item of weight
10. 20
11. Kg. we can either take
12. 1
13. Kg,
14. 2
15. Kg, …, or
16. 19
17. Kg but we are not allowed to take
18. 1.2
19. Kg or
20. 5.7
21. Kg.

Note that if we are taking a fraction of an item, we get the value in the same fraction.

Example

Input :

```
Unset

weight[] = {10, 30, 20, 50}
value[] = {40, 30, 80, 70}
W = 60
```

Output :

```
Unset

162
```

Explanation :
Let’s say we take the

1st

item as a whole, so we obtain a value of`40`and the capacity of the knapsack is now reduced to`50`. Then we pick the

3rd

item as a whole, after which we obtain a value of`120`and the capacity of the knapsack is now reduced to`30`.

Now we take `30` Kg. of the

4th

item, which will add

7050×30=42

to our value due to which we obtained a value of `162` and we are left with no space in the knapsack hence we can conclude the result to be `162`.

It can be proved that there is no other choice of picking item(s) such that the value of the chosen item(s) exceeds `162`.

Techniques to Solve the Fractional Knapsack Problem

Brute Force Approach

Since for an item of weight

x

, we have

x+1

choices

i.e.

taking `0` Kg. of the item, taking `1` Kg. of the item, taking `2` Kg. of the item, …., taking

x

Kg. of the item. Therefore, with the brute force approach, we can check the value of all the possibilities from the set of items provided.

As discussed above, for the i^{th} item we have $weight_i + 1$ choices, and hence the total number of possibilities is KaTeX parse error: $ within math mode. Therefore, the time complexity of the brute force approach is $O(\prod_{i=0}^{n-1}(\text{weight}_i + 1))$

Example :

Let's say we have the following items :

Item no.	Weight	Value
1	1	3

2	1	6
3	1	5

And a knapsack with a capacity of 3.

So, as per the brute force algorithm, we can have the following choices :
In the below-given matrix, the value in the

$cell_{i,j}$

denotes the amount of weight of

j^{th}

item we are taking in the

i^{th}

choice.

Item1	**Item2**	**Item3**	**Total Value**
0	0	0	0
0	0	1	5
0	1	0	6
0	1	1	11

1	0	0	3
1	0	1	8
1	1	0	9
1	1	1	**14**

The highest possible value we can obtain is **14**

i.e.

by picking the `1` Kg. of all the items.

Greedy Approach

We have seen that the brute force approach works every time but on a reasonable large input, it will take a very huge amount of time to calculate the answer.

So, we can opt for a greedy algorithm. We can have several potentially correct strategies, some of the obvious ones are :

1. Picking the items with the largest values first.
2. Picking the items with the lowest weights first.
3. Picking the items based on some sort of ratio among their values and weights.

Check the below illustration for a better understanding:

Consider the example: ***arr[] = {{100, 20}, {60, 10}, {120, 30}}, W = 50****.*

Sorting: *Initially sort the array based on the profit/weight ratio. The sorted array will be* ***{{60, 10}, {100, 20}, {120, 30}}****.*

Iteration:

- *For* ***i = 0****, weight = 10 which is less than W. So add this element in the knapsack.* ***profit = 60*** *and remaining* ***W = 50 – 10 = 40****.*
- *For* ***i = 1****, weight = 20 which is less than W. So add this element too.* ***profit = 60 + 100 = 160*** *and remaining* ***W = 40 – 20 = 20****.*
- *For* ***i = 2****, weight = 30 is greater than W. So add 20/30 fraction =* ***2/3*** *fraction of the element. Therefore* ***profit*** *= 2/3 * 120 + 160 = 80 + 160 =* ***240*** *and remaining* ***W*** *becomes* ***0****.*

*So the final profit becomes **240** for **W = 50**.*

Follow the given steps to solve the problem using the above approach:

- Calculate the ratio (**profit/weight**) for each item.
- Sort all the items in decreasing order of the ratio.
- Initialize **res = 0**, curr_cap = given_cap.
- Do the following for every item **i** in the sorted order:
    - If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
    - Else add the current item as much as we can and break out of the loop.
- Return **res**.

Pseudocode

```
Unset
```

```
// Function to find the maximum

// value that can be obtained.

function getMax (weight, value,  capacity):



    // Finding the size of the array.

    n = weight.length



    // Sorting the weight and value

    // array  in the decreasing order

    // of the value per unit weight ratio.

    Sort(weight, value, value[i]/weight[i])



    // Initializing answer with 0.

    ans = 0
```

```
// Iterating over the arrays

For i = 0 To i = n - 1:

    wt = weight[i]

    val = value[i]



    valuePerUnitWeight = value[i]/weight[i]



    // If we can take the whole item.

    If (capacity >= wt):

        // Adding value of current item.

        ans = ans + val



        // Reducing capacity by wt.

        capacity = capacity - wt
```

```
        // Otherwise we can take a fraction.
        Else:
            // Adding the value of the part that we can take.
            ans = ans + valuePerUnitWeight * capacity

            // Now we are left with no space so
            // we will terminate the loop.
            capacity = 0
            break

    // Returning the answer
    return ans
end function
```

A thief goes to a house to steal n objects having price values but his knapsack can hold only a fixed amount of weight. He needs to select the items he steals very smartly so that he maximizes the amount of money he can make from them. The bonus is that the object can be broken into pieces. So lets see how can help the thief!

So now you see, the entire aim of this problem is to help the thief efficiently steal the items so they fit in his knapsack while giving him the maximum value. But this isn't the only application of the Knapsack problem, it is used in automated data mining any industries like transportation, logistics, etc.

In simple words, the fractional knapsack problem can be described as — Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Let us break down the problem statement:

1. We will have n objects and each object will have a value.
2. There will be a maximum capacity W of the knapsack.
3. We need to select the objects such that the total weight of the objects does not exceed the maximum value W.
4. While selecting the objects, we need to select them in such a way that the total value is maximized, that is, price is maximized.
5. The objects can be divided in fractions, hence the name is fractional knapsack. As the object is divided into fractions, the value of the object will be divided in the same fraction.

If we have n objects:

1. i is the object
2. x_i is the fraction of the particular object
3. w_i is the weight of the fraction selected
4. p_i is the price of the fraction selected

then:

- $\sum_{1 \le i \le n} \boldsymbol{p_i x_i}$ needs to be maximized
- $\sum_{1 \le i \le n} \boldsymbol{w_i x_i} \le \boldsymbol{W}$
- $\boldsymbol{0 \le x_i \le 1}$
- $\boldsymbol{1 \le i \le n}$

Formulation of the problem

A feasible solution will satisfy equations 2,3,4 but an optimal solution will satisfy equation 1.

Example

Problem Statement: We have 5 objects having weights {30,50,10,70,40} and the price of the respective weights as {150,100,90,140,120}. Given the maximum capacity of the knapsack

as 150, find the set of objects to be selected such that the price is maximized.

We can summarize the given problem statement in a tabular form:

Item Number	Weight	Price
1	30	150
2	50	100
3	10	90
4	70	140
5	40	120

Problem Statement in tabular form

The maximum price comes out to be 500. One combination to get that is when we take the whole items 3,1,5,2 and a 2/7th fraction of item 4.

We will see various approaches to solve this problem ahead.

Brute Force Approach

If we try the brute force approach then we will have various permutations and combinations to satisfy all the equations. Let us try one combination.

Let W=remaining weight= 150 and P=total price=0.

If we take item one first, W=150–30=120 and P=0+150=150.

Then we take item 2, W=120–50=70 and P=150+100=250.

Taking item 3 next, W=70–10=60 and P=250+90=340.

As the remaining weight is 60 and the weight of item 4 is 70, we take the fraction 6/7 for item 4. So W=60-((6/7)*70)=0 and P=340+((6/7)*140)=340+120=460.

Now the remaining weight has become 0 so we don't consider item 5. The total price comes out to be 460 which is less than the maximum price which can be achieved. Hence this is not an efficient solution.

Here we have just considered one combination, we can try numerous such combinations taking different fractions of different items till we reach the most optimal solution. So it is safe to say that the Brute Force approach is very tedious and inefficient.

Greedy By Price Approach

When the brute force approach doesn't work, the next thing that comes to our mind is the greedy by price method, i.e, we take the items having the maximum price first.

Item Number	Weight	Price	Ranking by Price
1	30	150	1
2	50	100	4
3	10	90	5
4	70	140	2
5	40	120	3

Items ranked by price

Item 1 has the maximum price so we consider that first. W becomes 120 and P becomes 150.

Next is item 4, W=120−70=50 and P=150+140=290.

Next item 5, W=50−40=10 and P=290+120=410.

Next we take 1/5 fraction of item 2 (as remaining weight is less than the item weight) and so W=10-((1/5)*50)=10−10=0 and P=410+((1/5)*100)=410+20=430.

Item	Weight	Price	Rank	W calculation	P calculation
1	30	150	1	150-30=120	0+150=150
4	70	140	2	120-70=50	150+140= 290
5	40	120	3	50-40=10	290+120=410
2	50	100	4	10-((1/5) *50) =0	410+((1/5)*100)=430
3	10	90	5	-	-

Greedy by price calculation

So the total max price is 430 which is still less than 500. Hence we can conclude that the greedy by price method gives us a feasible solution but not the most optimal solution.

Greedy by Weight

In the greedy by weight method, we rank the weights in increasing order of the weight. We first take the item having the smallest weight and then so on. This is so that we can fit the maximum number of items in the knapsack so the price can be maximized.

Item Number	Weight	Price	Ranking by Weight
1	30	150	2
2	50	100	4
3	10	90	1
4	70	140	5
5	40	120	3

Items ranked in order of weight

Let us do the calculations now :

Initial values: W=150 and P=0

Item 3: W=150−10=140 and P=0+90

Item 1: W=140−30=110 and P=90+150

Item 5: W=110−40=70 and P=240+120=360

Item 2: W=70−50=20 and P=360+100=460

Item 4: W=20-((2/7)*70)=0 and P=460+((2/7)*140)=500

Item	Weight	Price	Rank	W calculation	P calculation
3	10	90	1	150-10=140	0+90=90
1	30	150	2	140-30=110	90+150=240
5	40	120	3	110-40=70	240+120=360
2	50	100	4	70-50=20	360+100=460
4	70	140	5	20-((2/7)*70)=0	460+((2/7)*140)=500

Greedy by weight calculation

We have finally gotten the maximum amount possible price (500) in this case **but this approach does not *always* give the most optimal solution!**

We will look at another approach now which will always give the most optimal solution.

Greedy by Price Per Unit

We combine the greedy by price and weight method and try the **greedy by price per unit** approach. Price per unit is the ratio of price/weight for each item. The item having the highest value of the ratio is considered first and so on. The steps for implementing this are:

1. Calculate the P/W ratio for all items
2. Arrange the items in descending order according to the ratio
3. Set W=max weight and P=0
4. While (W!=0)

4.i. Take the item having the highest ratio value

4.ii. Consider the fraction of the selected item

4.iii. Substract the weight of the item from W

4.iv. Add the price of the item to P

Let us solve the above example using this algorithm.

Step 1: Calculate the P/W ratio for all items

Item Number	Weight	Price	P_i/W_i
1	30	150	$\frac{150}{30} = 5$
2	50	100	$\frac{100}{50} = 2$
3	10	90	$\frac{90}{10} = 9$
4	70	140	$\frac{140}{70} = 2$
5	40	120	$\frac{120}{40} = 3$

Calculating ratios

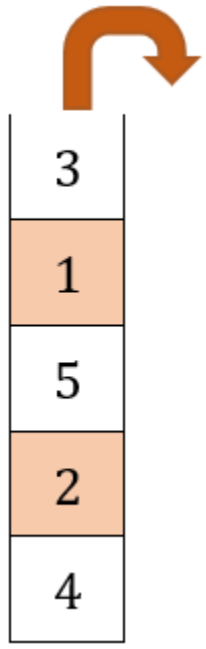
We calculate the ratio by dividing the price of every item by the weight of the respective item.

Step 2: Arrange the items in descending order according to the ratio

From the above table calculations in step 1, we now arrange in such a manner that the item having the largest ratio value is first and the one having the smallest ratio is last.

The ratios are in the order: 9>5>3>2=2

We arrange the items accordingly:



Order of the items to be considered

3>1>5>2>4

The item having a greater ratio value will be considered first and so on.

Note that here both the items 2 and 4 have the same ratio values. We can take any item first and it won't make any difference as the ratio P/W is same for both.

So this is the order in which we will select the item: 3 -> 1 -> 5 -> 2 -> 4.

Step 3: Set W=max weight and P=0

We set variable W=max weight = 150 (given) and price as P=0.

We also initialize the table for our calculation:

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W	P
3	10	90					
1	30	150					
5	40	120					
2	50	100					
4	70	140					

Table for calculation

The Xi column in the table is for the fraction of the weight that we will be selecting of the particular item. The WiXi column stores the product of the fraction selected and the weight, i.e. the total weight of the item to be considered and the PiXi does the same for price.

If Wi(weight of the item) is less than the total remaining weight (W), then we take Xi=1, that is we take the entire item.

Otherwise if the total remaining weight W is less than the item weight Wi we take the fraction of item. The fraction is calculated by: Xi=W/Wi.

W keeps a track of the remaining weight and P keeps a track of the total price.

For every step, W=W- (Wi*Xi) and P=P+(Pi*Xi)

Step 4 : Check if W is not equal to 0

Currently W=150 !=0 so we continue.

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W ($W-W_i$)	P ($P+P_i$)
3	10	90	1	10*1=10	90*1=90	140	90
1	30	150					
5	40	120					
2	50	100					
4	70	140					

For item 3

The highest ratio is of item 3, so we consider that first. Since Wi<W, we have Xi=1. So W=150–10=140 and P=0+90=90.

W is still not equal to 0 so we take the next item.

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W ($W-W_i$)	P ($P+P_i$)
3	10	90	1	10*1=10	90*1=90	140	90
1	30	150	1	30*1=30	150*1=150	110	240
5	40	120					
2	50	100					
4	70	140					

For item 1

The next item is item 1. Again Wi<W so Xi=1. We calculate the rest of the columns by the formulae given above.

W=110!=0, consider next item.

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W ($W-W_i$)	P ($P+P_i$)
3	10	90	1	10*1=10	90*1=90	140	90
1	30	150	1	30*1=30	150*1=150	110	240
5	40	120	1	40*1=40	120*1=120	70	360
2	50	100					
4	70	140					

For item 5

The next item is item 5. Again Wi<W so Xi=1. We calculate the rest of the columns by the formulae given above.

W=70!=0 so next item.

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W ($W-W_i$)	P ($P+P_i$)
3	10	90	1	10*1=10	90*1=90	140	90
1	30	150	1	30*1=30	150*1=150	110	240
5	40	120	1	40*1=40	120*1=120	70	360
2	50	100	1	70*1=70	100*1=100	20	460
4	70	140					

For item 2

The next item is item 2. Again Wi<W so Xi=1. We calculate the rest of the columns by the formulae given above.

W=20!=0, take next item.

Item	Weight	Price	X_i	W_iX_i	P_iX_i	W ($W-W_i$)	P ($P+P_i$)
3	10	90	1	10*1=10	90*1=90	140	90
1	30	150	1	30*1=30	150*1=150	110	240
5	40	120	1	40*1=40	120*1=120	70	360
2	50	100	1	70*1=70	100*1=100	20	460
4	70	140	2/7	$70*\frac{2}{7}=20$	$140*\frac{2}{7}=40$	0	500

Last item 4

The last item that we have is item 4. The weight of item 4 is 70 but the remaining weight is 20 so W<Wi. The value of W can never be less than 0 as W is the maximum weight the knapsack can hold so we consider the fraction of item 4 where Xi=Wi/W therefore, Xi=20/70=2/7.

As we are taking the fraction of item 4, the weight and price gets divided accordingly.

So the weight to be considered will be 70*(2/7)=20 and price will be 140*(2/7)=40.

Now W has become 0, so we terminate the algorithm over here.

The total price P comes out to be 500 which is the optimal solution.

Hence we can say that this approach works efficiently.

From the tabular calculation we can see that we have taken the whole items 1,2,3,5 and 2/7th item of 4. The answer comes out to be : {1,1,1,2/7,1}.

Overview

In the article we have seen 4 methods to solve the Fractional Knapsack Problem:

1. Brute force
2. Greedy by Price

3. Greedy by Weight
4. Greedy by Price per Unit

The first method gives many feasible solutions but numerous combinations need to be tried to get the most optimal one.

The second and third methods may not always give the most optimal solution.

The fourth method always gives the most optimal solution and is an easy approach towards finding the solution.

You can find the code to implement the Fractional Knapsack Problem using Greedy by price per unit method (same logic as above) on my GitHub- https://github.com/riya1620/FractionalKnapsack

Ref:

https://medium.com/@riya.tendulkar/how-to-solve-the-fractional-knapsack-problem-d2c11b56aa38

Ref: https://www.geeksforgeeks.org/fractional-knapsack-problem/

Job Sequencing Problem

Ref: https://www.geeksforgeeks.org/job-sequencing-problem/

Given three arrays **id[]**, **deadline[]**, **profit[]**, where each job **i** is associated with id[i], deadline[i], and profit[i]. Each job takes **1 unit** of time to complete, and only **one** job can be scheduled at a **time**. You will earn the **profit** associated with a job only if it is **completed** by its **deadline**. The **task** is to find the **maximum** profit that can be gained by completing the jobs and the **count** of **jobs** completed to earn the maximum profit.

Examples:

Input: *id[] = [1, 2, 3, 4]*
deadline[] = [4, 1, 1, 1]
profit[] = [20, 10, 40, 30]
Output: *2 60*
Explanation: *All jobs other than the first job have a deadline of 1, thus only one of these and the first job can be completed, with the total profit gain of 20 + 40 = 60.*

Input: *id[] = [1, 2, 3, 4, 5]*
deadline[] = [2, 1, 2, 1, 1]
profit[] = [100, 19, 27, 25, 15]
Output: *2 127*
Explanation: *The first and third job have a deadline of 2, thus both of them can be completed and other jobs have a deadline of 1, thus any one of them can be completed. Both the jobs with a deadline of 2 is having the maximum associated profit, so these two will be completed, with the total profit gain of 100 + 27 = 127.*

Job Sequencing Problem

Problem Statement: You are given a set of N jobs where each job comes with a deadline and profit. The profit can only be earned upon completing the job within its deadline. Find the number of jobs done and the maximum profit that can be obtained. Each job takes a single unit of time and only one job can be performed at a time.

Examples

Example 1:

Input: N = 4, Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

Output: 2 60

Explanation: The 3rd job with a deadline 1 is performed during the first unit of time .The 1st job is performed during the second unit of time as its deadline is 4.
Profit = 40 + 20 = 60

Example 2:

Input: N = 5, Jobs = {(1,2,100),(2,1,19),(3,2,27),(4,1,25),(5,1,15)}

Output: 2 127

Explanation: The first and third job both having a deadline 2 give the highest profit.
Profit = 100 + 27 = 127

Job Sequencing with Deadlines Algorithm

The simple and brute-force solution for this problem is to generate all the sequences of the given set of jobs and find the most optimal sequence that maximizes the profit.

Suppose, if there are n number of jobs, then to find all the sequences of jobs we need to calculate all the possible subsets and a total of

2^n

subsets will be created. Thus, the time complexity of this solution would be

$O(2^n)$

To optimize this algorithm, we can make use of a greedy approach that produces an optimal result, which works by selecting the best and the most profitable option available at the moment.

(Greedy Approach)

problem -

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Job ID	A	B	C	D	E
Deadline	2	1	2	1	3
profit	100	19	27	25	15

Sol:

Greedy Approach

(1) Sort all jobs in decreasing order of profit.

(2) Initialize the result sequence as first job in sorted jobs.

(3) Do following for remaining n-1 jobs
   a) If the current job can fit in the current

\Ref:
https://medium.com/@dillihangrae/job-sequence-with-deadlines-greedy-algorithm-18d3734d6d1c

result sequence without missing the deadline, add current job to the result, else ignore the current job.

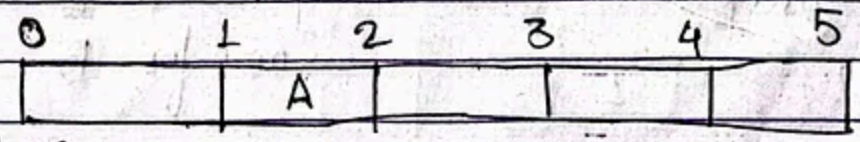
Greedy Approach:

1) Sort all jobs in decreasing order of profit.

Sorted -

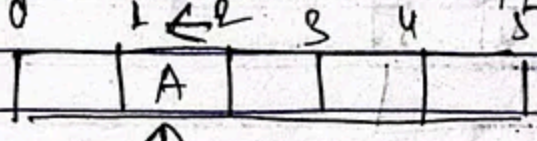
JobID	A	C	D	B	E	
Deadline	2	2	1	1	3	
profit	100	27	25	19	15	→
						→ decreasing

2) Schedule the Job from 0-1 second 1-2 sec.



Now

for Deadline 2 - 27 prft so for indexed at 2 there is A then look opposite it is occupied



occupied

0	1	2	3	4	5
C	A	E			

Now, we have 0-1 → C Job Sequences..
1-2 → A
2-3 → E

Time Complexity : $O(n^2)$
n = number of jobs

JOB ID	A	B	C	D	E
Profit	20	15	10	5	1
Death	2	2	1	3	3

(1) Step sort profit Decreasing — already sorted — n = 5

0	1	2	3	4	5	6
B	A	D				

B → A → D

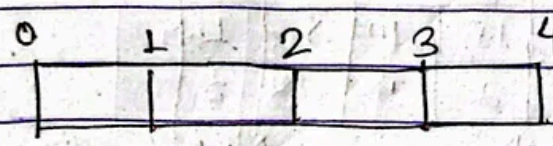
20 → 15 → 5
20 + 15 + 5 ⇒ 40

JOBID	A	B	C	D
Deadline	2	1	2	1
profit	100	10	15	07

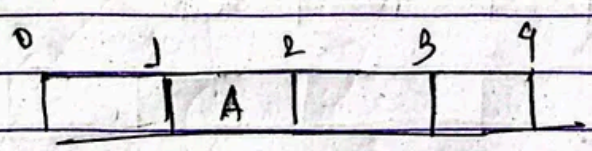
(1) Sorted jobsID in decreasing order of Pi

JOBID	A	D	C	B
Deadline	2	1	2	1
profit	100	27	15	10

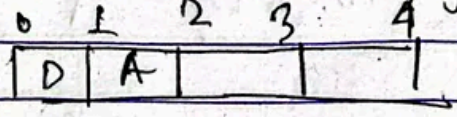
(2) Sequence initialize $n = 4$



(3) first is A with 2 deadline



then D with 1 - Empty slot found



Naive string matching algorithm

REf: https://www.geeksforgeeks.org/naive-algorithm-for-pattern-searching/

Given **text** string with length **n** and a **pattern** with length **m,** the task is to prints all occurrences of **pattern** in **text**.
Note: You may assume that n > m.

Examples:

Input: *text = "THIS IS A TEST TEXT", pattern = "TEST"*
Output: *Pattern found at index 10*

Input: *text = "AABAACAADAABAABA", pattern = "AABA"*
Output: *Pattern found at index 0, Pattern found at index 9, Pattern found at index 12*

```
def search_pattern(pattern, text):

    # Get the lengths of the pattern and the text

    m = len(pattern)

    n = len(text)



    # A loop to slide pattern over text one by one

    for i in range(n - m + 1):

        # For current index i, check for pattern match

        j = 0

        while j < m and text[i + j] == pattern[j]:

            j += 1



        # If the entire pattern matches the text starting at index i

        if j == m:

            print(f"Pattern found at index {i}")
```

Time Complexity: $O(N^2)$

Auxiliary Space: $O(1)$

Complexity Analysis of Naive algorithm for Pattern Searching:

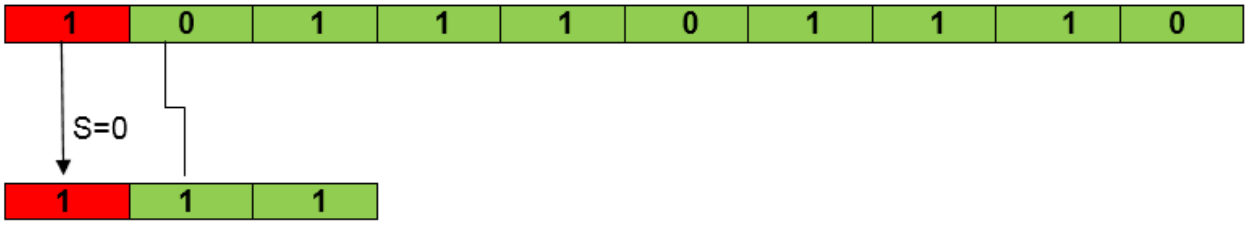
Best Case: O(n)

- *When the **pattern** is found at the very beginning of the **text** (or very early on).*
- *The algorithm will perform a constant number of comparisons, typically on the order of **O(n)** comparisons, where n is the length of the **pattern**.*

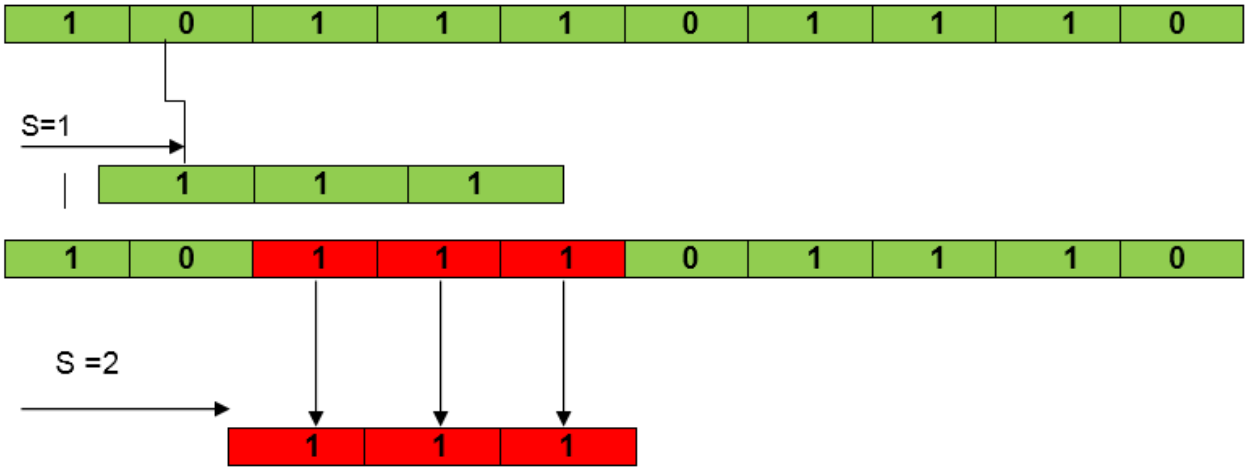
Worst Case: O(n^2)

- *When the **pattern** doesn't appear in the **text** at all or appears only at the very end.*
- *The algorithm will perform **O((n-m+1)*m)** comparisons, where **n** is the length of the **text** and **m** is the length of the **pattern**.*
- *In the worst case, for each position in the **text**, the algorithm may need to compare the entire **pattern** against the text.*

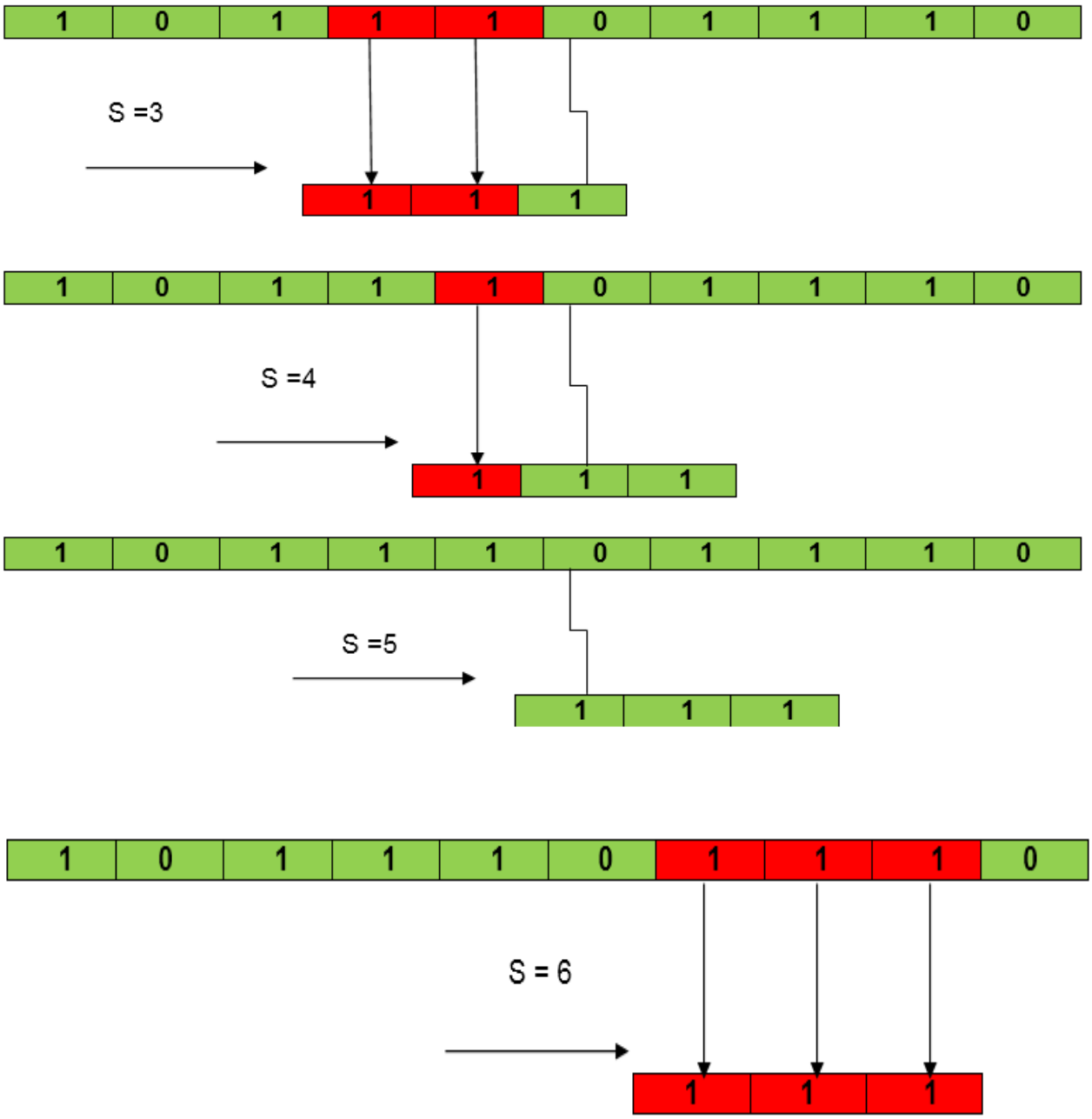
T = Text



P = Pattern



So, S=2 is a Valid Shift



Ref: https://www.tpointtech.com/daa-naive-string-matching-algorithm

Work Mechanism

Ref: https://www.naukri.com/code360/library/string-matching-naive

- ***Pattern and Text:*** *We have a pattern and textual content*
- ***Comparison:*** *Start from the start of the text and examine every character with the pattern*
- ***Match Check:*** *If the characters match, continue evaluating the next characters in each text and pattern*
- ***Mismatch:*** *If there may be a mismatch, one character in advance inside the text and start evaluating from the beginning of the pattern.*
- ***Repeat:*** *Keep repeating this system till you discover a match or attain the end of the text.*
- ***Result:*** *When a suit is observed, or you look at the textual content, you know whether the pattern exists in the text.*

Naive Pattern Searching Approach

Naive Pattern Searching Approach is one of the easy pattern-searching algorithms. All the characters of the pattern are matched to the characters of the corresponding/main string or text. In this approach, we will check all the possible placements of our input pattern with the input text.
Refer to the algorithm below for a better understanding.

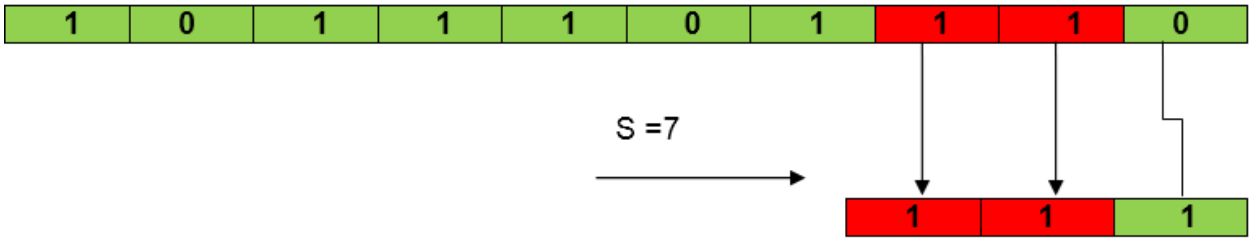
Algorithm

1. *There will be two loops: the outer loop will range from i=0 to i≤n-m, and the inner loop will range from j=0 to j<m, where 'm' is the length of the input pattern and n is the length of the text string.*
2. *Now, At the time of searching algorithm in a window, there will be two cases:*

- ***Case 1****: If a match is found, we will match the entire pattern with the current window of the text string. And if found the pattern string is found in the current window after traversing, print the result. Else traverse in the next window.*

- ***Case 2:*** *If a match is not found, we will break from the loop(using the 'break' keyword), and the j pointer of the inner loop will move one index more and start the search algorithm in the next window.*

So, S=6 is a Valid Shift



Rabin-Karp Algorithm for Pattern Searching

Ref: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

Given a text T**[0. . .n-1]** and a pattern P**[0. . .m-1]**, write a function search(char P[], char T[]) that prints all occurrences of P[] present in T[] using Rabin Karp algorithm. You may assume that **n > m**.

Examples:

> **Input:** T[] = “THIS IS A TEST TEXT”, P[] = “TEST”
> **Output:** Pattern found at index 10
>
> **Input:** T[] = “AABAACAADAABAABA”, P[] = “AABA”
> **Output:** Pattern found at index 0
> Pattern found at index 9
> Pattern found at index 12

Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the **hash value** of the **pattern** with the **hash value** of the current substring of **text**, and if the **hash values** match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.

- **Pattern** itself
- All the substrings of the **text** of length **m** which is the size of pattern.

Step-by-step approach:

- Initially calculate the hash value of the pattern.
- Start iterating from the starting of the string:
  - Calculate the hash value of the current substring having length **m**.
  - If the hash value of the current substring and the pattern are same check if the substring is same as the pattern.
  - If they are same, store the starting index as a valid answer. Otherwise, continue for the next substrings.
- Return the starting indices as the required answer.

How is Hash Value calculated in Rabin-Karp?

Hash value is used to efficiently check for potential matches between a **pattern** and substrings of a larger **text**. The hash value is calculated using a **rolling hash function**, which allows you to update the hash value for a new substring by efficiently removing the contribution of the old character and adding the contribution of the new character. This makes it possible to slide the pattern over the **text** and calculate the hash value for each substring without recalculating the entire hash from scratch.

Here's how the hash value is typically calculated in Rabin-Karp:

Step 1: Choose a suitable **base** and a **modulus**:

- Select a prime number '**p**' as the modulus. This choice helps avoid overflow issues and ensures a good distribution of hash values.
- Choose a base '**b**' (usually a prime number as well), which is often the size of the character set (e.g., 256 for ASCII characters).

Step 2: Initialize the hash value:

- Set an initial hash value '**hash**' to **0**.

Step 3: Calculate the initial hash value for the **pattern**:

- Iterate over each character in the **pattern** from **left** to **right**.
- For each character **'c'** at position **'i'**, calculate its contribution to the hash value as **'c * ($b^{pattern\_length - i - 1}$) % p'** and add it to '**hash**'.
- This gives you the hash value for the entire **pattern**.

Step 4: Slide the pattern over the **text**:

- Start by calculating the hash value for the first substring of the **text** that is the same length as the **pattern**.

Step 5: Update the hash value for each subsequent substring:

- To slide the **pattern** one position to the right, you remove the contribution of the leftmost character and add the contribution of the new character on the right.
- The formula for updating the hash value when moving from position **'i'** to **'i+1'** is:

$$hash = (hash - (text[i - pattern\_length] * (b^{pattern\_length - 1})) \% p) * b + text[i]$$

Step 6: Compare hash values:

- When the hash value of a substring in the **text** matches the hash value of the **pattern**, it's a **potential match**.
- If the hash values match, we should perform a character-by-character comparison to confirm the match, as hash collisions can occur.

The idea behind the string hashing is the following: we map each string into an integer and compare those instead of the strings. Doing this allows us to reduce the execution time of the string comparison to O(1).

1. Compute the hash of the string pattern.
2. Compute the substring hash of the string text starting from index 0 to *m*-1.
3. Compare the substring hash of text with the hash of pattern.
- If they are a match, then compare the individual characters to ensure the two strings are an exact match.
- If they are not a match, then slide the substring window by incrementing the index and repeat step 3 to compute the hash of the next *m* characters until all *n* characters have been traversed.

Hash Function

First of all, the algorithm is only as good as its hash function. If a hash function which results in many false positives is used, then character comparisons will be done far too often to deem this method any more performant than a naive approach.

Secondly, you might have noticed that a new hash is computed each time the substring window traverses through the text. This is highly inefficient as it results in the same performance (if not worse) as a naive approach.

Both these problems can be solved using polynomial hashing with additions and multiplications. Although this is not a Rabin-fingerprint, it works equally well.

> **Hashing**
>
> Ref: https://brilliant.org/wiki/rabin-karp-algorithm/
>
> Hashing is a way to associate values using a hash function to map an input to an output. The purpose of a hash is to take a large piece of data and be able to be represented by a smaller form. Hashing is incredibly useful and

> but does have some downfalls, namely, collisions. The pigeonhole principle tells us that we can't put xx balls into x−1x−1 buckets without having at least one bucket with two balls in it. With hashing, we can think of the inputs to a function as the balls and the buckets as the outputs of the function—some inputs must inevitably share an output value, and this is called a collision.

Polynomial Rolling Hash

Example:

Say you have a hash function that maps each letter in the alphabet to a unique prime number, [prime_a…prime_z][prime_a…primez_]. You are searching a text TT of length nn for a word WW of length mm. Let's say that the text is the following string "abcdefghijklmnopqrstuvwxyz" and the word is "fgh". How can we use a rolling hash to search for "fgh"?

At each step, we compare three letters of the text to the three letters of the word. At the first step,

we can multiply prime_a×prime_b×prime_c to get the hash of the string "abc".

Similarly, the hash value of "fgh" is prime_f×prime_g×prime_h. Compare the hash value of "abc" with "fgh" and see that the numbers do not match.

To move onto the next iteration, we need to calculate the hash value of "bcd". How can we do this? We could compute

prime_b×prime_c×prime_d,

but we would actually be repeating work we've already done!

We know

prime_a×prime_b×prime_c

and we know that

(prime_a×prime_b×prime_c)/(prime_a) = prime_b * prime_c

If we divide out the prime_a and multiply in the prime_d,we can get the hash value of "bcd". We compare this value with "fgh", and so on

We can compute the polynomial hash with multiplications and additions as shown below.

$$H = c_1 \times b^{m-1} + c_2 \times b^{m-2} + c_3 \times b^{m-3} ... + c_m \times b^0$$

c = characters in the string **m** = length of string **b** = constant

Example

For the sake of brevity, let's use integers directly instead of character conversions in this example.

Given the pattern '135' and a text '2135' with a base *b = 10.*

First we compute the hash of the pattern '135'.

$$H(135) = 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 = 135$$

Next, we compute the hash of the first *m = 3* characters of the text which is '213'.

$$H(213) = 2 \times 10^2 + 1 \times 10^1 + 3 \times 10^0 = 213$$

This is clearly not a match. So, let's slide the window by dropping the first character of the previous window and adding the next character to it. The window now represents '135'.

$$H(135) = 1 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 = 135$$

Now our hashes are a match and the algorithm essentially comes to an end.

Rolling Hash

Notice that we had to compute the entire hash of '213' and '135' after moving the sliding window. This is undesirable as we had to compute the hash of some integers we had already accounted for in the previous window.

The rolling hash function can effectively eliminate these additional computations by acknowledging that the hash of a new window skips the first character of a previous window and adds the computation of a new character.

In theory, we can get rid of the hash value of the skipped character, multiply the resulting value by the base (to restore the correct order of the exponents of the previous untouched characters), and finally add the value of the new character.

Therefore, we can compute the hash of the new window by using the equation shown below.

$$H = (H_p - C_p \times b^{m-1}) \times b + C_n$$

Hp = previous hash **Cp** = previous character **Cn =** new character **m** = window size **b** = constant

Using the previous example of moving from ‘213’ to ‘135’, we can plug in the values to get the new hash.

$$H = (213 - 200) \times 10 + 5 = 135$$

By using the rolling hash function, we can calculate the hash of each sliding window in linear time. This is the main component of the Rabin-Karp algorithm that provides its best case time complexity of ***O(n+m)***.

Modular Arithmetic

All math in the Rabin-Karp algorithm needs to be done in modulo ***Q*** to avoid manipulating large ***H*** values and integer overflows. This is done at the expense of increased hash collisions, also known as spurious hits.

> The value for Q would usually be a large prime number — as large as it can be without compromising arithmetic performance. The smaller the value of Q, the higher the chances of spurious hits.

$$H = (c_1 \times b^{m-1} + c_2 \times b^{m-2} ... + c_m \times b^0) \, mod \, Q$$

There is a potential problem with the above approach. To understand what that is, let’s have a look at a simple JavaScript code snippet of it.

Notice that there are two multiplications and an addition done inside the loop. Not only is that inefficient, but it also fails to prevent integer overflows as larger sums are calculated before the modulo operator is even used. We can overcome this problem by using Horner's method.

Horner's Method

Horner's method simplifies the process of evaluating a polynomial by dividing it into monomials (polynomials of the 1st degree).

$$c_0 + c_1 \times b^1 + c_2 \times b^2 + c_3 \times b^3 = c_0 + b(c_1 + b(c_2 + c_3 \times b))$$

Using this method, we can eliminate one multiplication from our previous implementation. This leaves us with only one multiplication and one addition at each step in the loop, which in turn allows us to prevent integer overflows.

Complexity

Given a word of length m and a text of length n, the best case time complexity is ***O(n + m)*** and space complexity is ***O(m)***. The worst case time complexity is ***O(nm)***. This can occur when an extremely poor performing hash function is chosen, but a good hash function, such as polynomial hashing, can fix this problem.

Finite Automata algorithm for Pattern Searching

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9

        Pattern found at index 12
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

The string-matching automaton is a very useful tool which is used in string matching algorithm.
String matching algorithms build a finite automaton scans the text string T for all occurrences of the pattern P.

FINITE AUTOMATA

- Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P.
- This approach examines each character of text exactly once to find the pattern. Thus it takes linear time for matching but preprocessing time may be large.
- It is defined by tuple M = {Q, Σ, q, F, d} Where Q = Set of States in finite automata

Σ=Sets of input symbols

Algorithm-

```
FINITE AUTOMATA (T, P)
State <- 0
for l <- 1 to n
State <- δ(State, ti)
If State == m then
Match Found
end
end
```

Algorithm Overview:

The Finite Automaton-based pattern searching algorithm utilizes the concept of a deterministic finite automaton (DFA) to efficiently search for patterns within text. It involves two main steps: constructing the Transition Function (TF) table and traversing the DFA over the text.

Algorithm

Step 1: Constructing the Transition Function (TF) Table:

- The TF table represents the transitions between states of the DFA based on the input characters.
- It is constructed by iterating through all possible states and characters, determining the next state based on the current state and the input character.
- The get Next State function calculates the next state based on the current state and input character.

Step 2: Traversing the DFA over the Text:

- Once the TF table is constructed, the DFA is traversed over the text

to search for occurrences of the pattern.

- Starting from the initial state (state 0), the DFA transitions to subsequent states based on the characters in the text.
- When the DFA reaches the final state (which represents the end of the pattern), a match is found, and the index of the occurrence is recorded.

Step 3:Reporting Results:

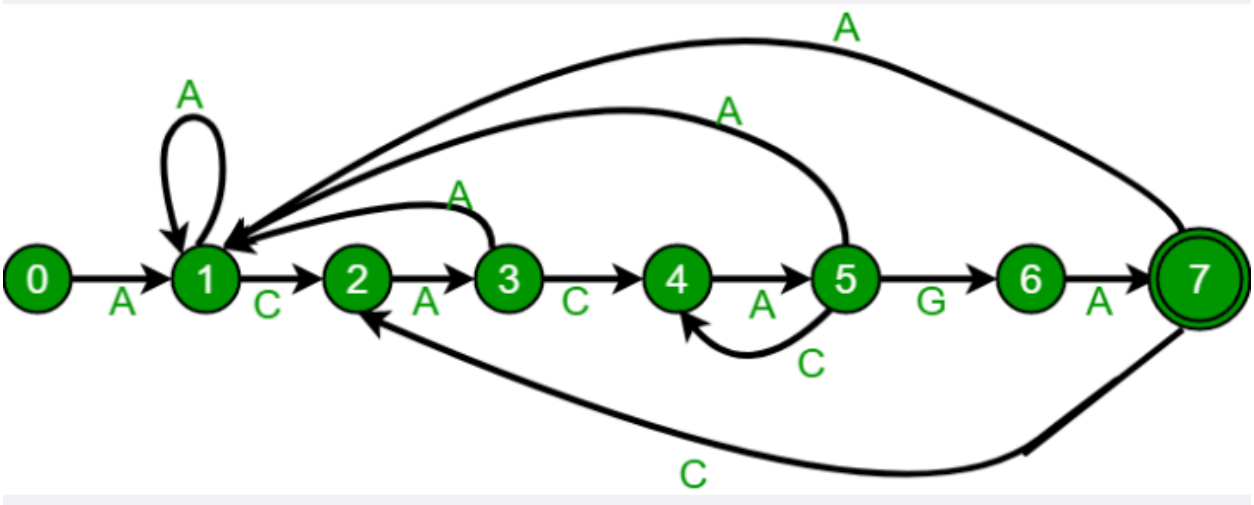
- If a match is found during the traversal, report the index of the occurrence.
- If no matches are found, report that the pattern was not found in the text.

Implementation:

These string matching automaton are very efficient because they examine each text character exactly once, taking constant time per text character. The matching time used is O(n) where n is the length of Text string.

<u>But the preprocessing time i.e. the time taken to build the finite automaton can be large if ? is large.</u>

Before we discuss Finite Automaton construction, let us take a look at the following Finite Automaton for pattern ACACAGA.



	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The above diagrams represent graphical and tabular representations of pattern ACACAGA.

Number of states in Finite Automaton will be **M+1** where M is length of the pattern. The main thing to construct Finite Automaton is to get the next state from the current state for every possible character.\

Knuth–Morris–Pratt algorithm (or KMP algorithm)

is a string-searching algorithm that searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

Matching Overview

txt = "AAAAABAAABA"

pat = "AAAA"

We compare first window of ***txt*** *with* ***pat***

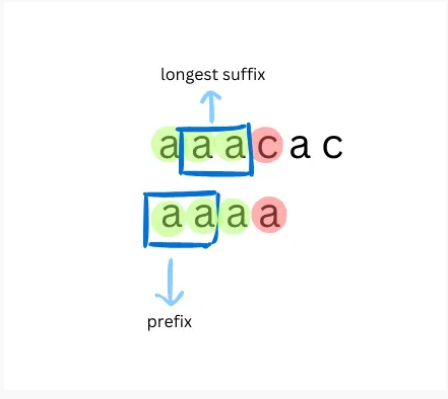
In KMP Algorithm,

- *We preprocess the pattern and build LPS array for it. The size of this array is same as pattern length.*
- *LPS is the **Longest Proper Prefix** which is also a **Suffix**. A proper prefix is a prefix that doesn't include **whole** string. For example, prefixes of "abc" are "", "a", "ab" and "abc" but proper prefixes are "", "a" and "ab" only. Suffixes of the string are "", "c", "bc", and "abc".*
- *Each value, **lps[i]** is the length of longest proper prefix of **pat[0..i]** which is also a suffix of **pat[0..i]**.*

Preprocessing Overview:

- *We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.*
- *For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].*

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].



REF: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

*txt = "**AAAA**ABAAABA"*

*pat = "**AAAA**" [Initial position]*

We find a match. This is same as Naive String Matching.

*In the next step, we compare next window of **txt** with **pat**.*

*txt = "**AAAAA**BAAABA"*

*pat = "**AAAA**" [Pattern shifted one position]*

This is where KMP does optimization over Naive. In this second window, we only compare fourth A of pattern

with fourth character of current window of text to decide whether current window matches or not. Since we know

first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing?

An important question arises from the above explanation, how to know how many characters to be skipped. To know this,

we pre-process pattern and prepare an integer array lps[] that tells us the count of characters to be skipped

Let me break down the key aspects of this KMP algorithm implementation with a solved example:

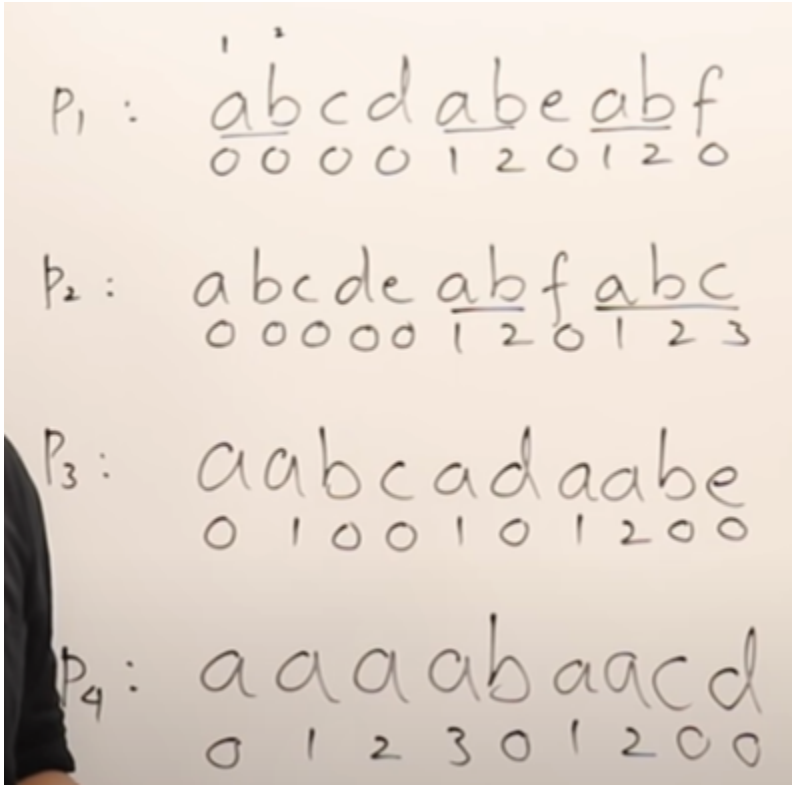
1. **LPS (Longest Proper Prefix Suffix) Computation**:
    - The `compute_lps()` function creates an array that helps in efficient pattern matching
    - For the pattern "ABABCABAB", the LPS array will be

    - `[0, 0, 1, 2, 0, 1, 2, 3, 4]`
    - This array helps in quickly skipping unnecessary comparisons during pattern searching
2. **Pattern Searching**:
    - The `kmp_search()` function uses the LPS array to efficiently search for the pattern
    - In our example, the pattern "ABABCABAB" is found at index 10 in the text "ABABDABACDABABCABAB"
3. **Time Complexity**:
    - O(m + n), where m is the length of the text and n is the length of the pattern
    - Much more efficient than naive string matching which is O(m * n)

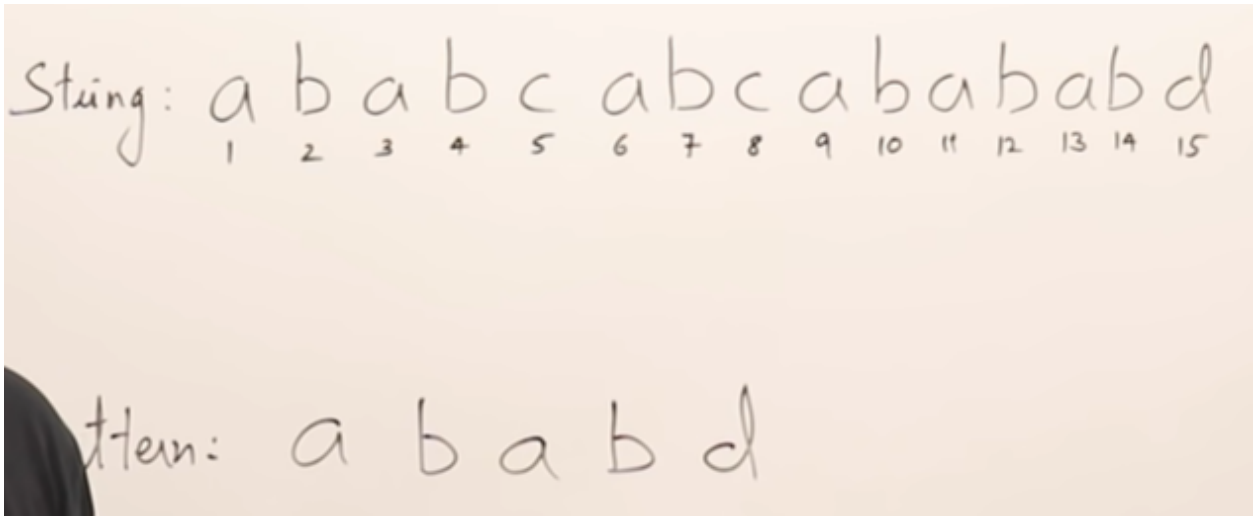
When you run this code, you'll see:

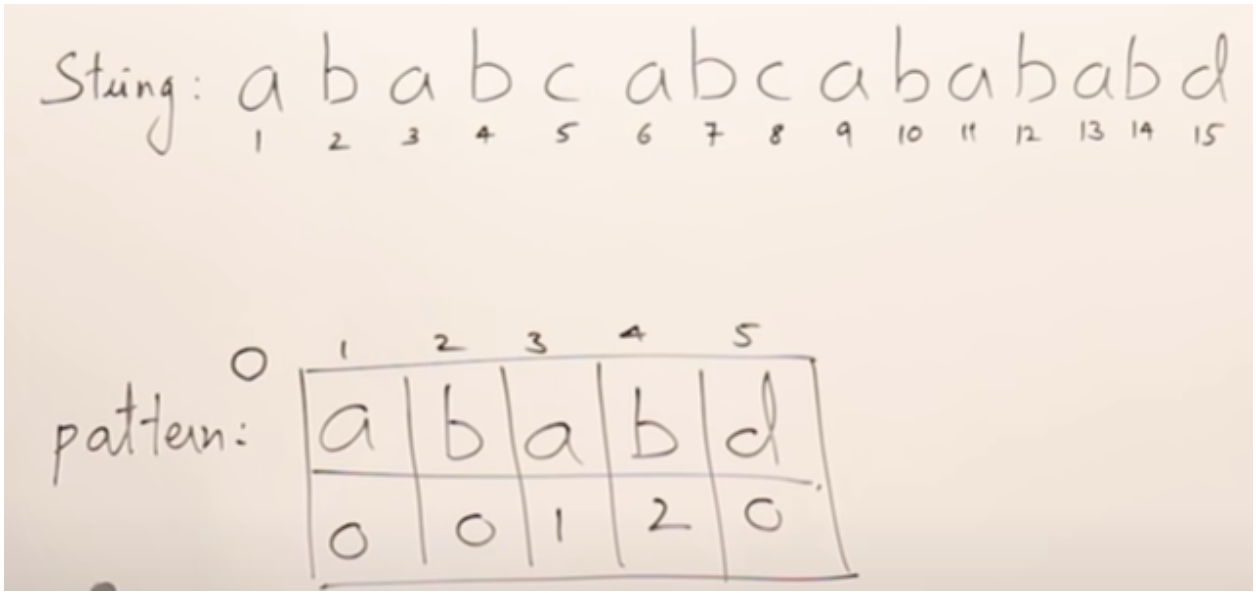
- The LPS array for the pattern
- The indices where the pattern is found in the text

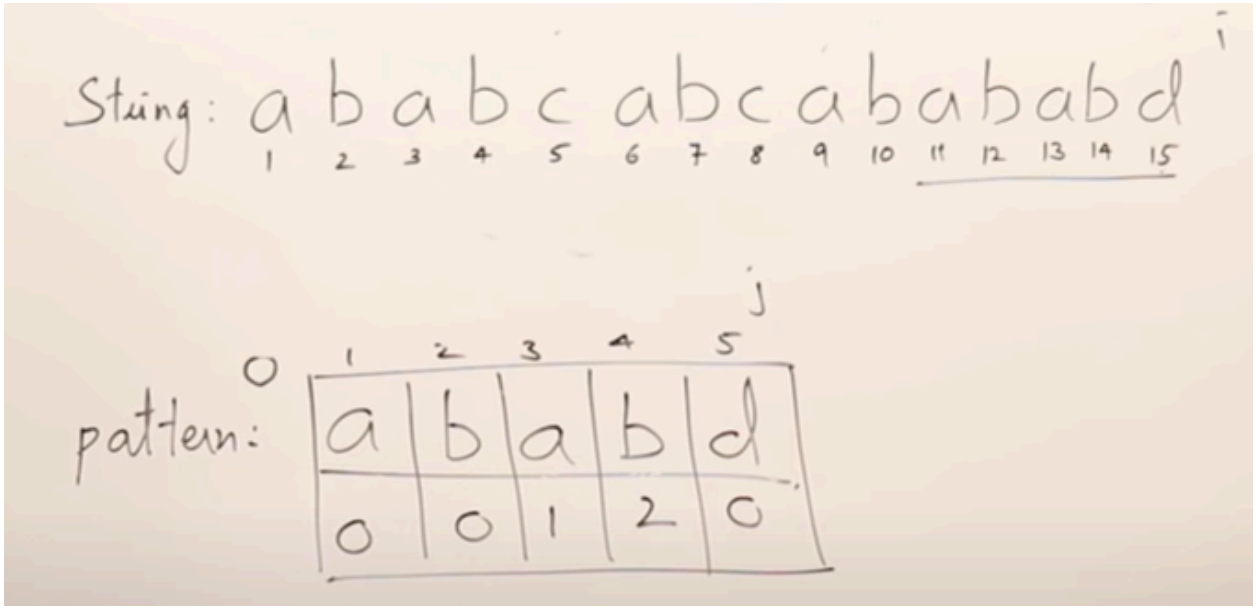
The key advantage of KMP is its ability to avoid backtracking in the text, making it much faster for large texts and patterns with repetitive subpatterns.



Note: the P3 should have Pi table as 0100101230


String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
ttern: a b a b d


String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0
pattern:
1 2 3 4 5
a b a b d
0 0 1 2 0



ref: https://www.youtube.com/watch?v=V5-7GzOfADQ

```
def compute_lps(pattern):
    """
    Compute the Longest Proper Prefix which is also Suffix (LPS) array
    This helps in efficient pattern matching
```

```
"""

lps = [0] * len(pattern)

length = 0  # Length of the previous longest prefix suffix

i = 1



while i < len(pattern):

    if pattern[i] == pattern[length]:

        # If characters match, increase length and assign to LPS

        length += 1

        lps[i] = length
```

```
        i += 1

    else:

        # If characters don't match

        if length != 0:

            # Go back to the previous longest prefix suffix

            length = lps[length - 1]

        else:

            # If length is 0, set LPS to 0 and move to next character

            lps[i] = 0

            i += 1
```

```
    return lps


def kmp_search(text, pattern):
    """
    KMP algorithm for pattern searching
    Returns all starting indices where pattern is found in text
    """
    # Compute the LPS array for the pattern
    lps = compute_lps(pattern)
```

```
# Indices to track positions in text and pattern

i = 0  # Index for text

j = 0  # Index for pattern



# Results to store all match starting positions

matches = []



while i < len(text):

    # If characters match, move both indices forward
```

```
if text[i] == pattern[j]:

    i += 1

    j += 1




# Pattern fully matched

if j == len(pattern):

    matches.append(i - j)

    # Move pattern index back using LPS

    j = lps[j - 1]
```

```
    # Mismatch after some matches

    elif i < len(text) and text[i] != pattern[j]:

        # If pattern index is not at start, use LPS

        if j != 0:

            j = lps[j - 1]

        else:

            # If pattern index is at start, move text index

            i += 1



return matches
```

Solved Example

text = "ABABDABACDABABCABAB"

pattern = "ABABCABAB"

Compute LPS array for the pattern

lps_array = compute_lps(pattern)

print("LPS Array:", lps_array)

Search for pattern in text

```
result = kmp_search(text, pattern)


print(f"Pattern '{pattern}' found at indices: {result}")
```