# RC4

# Graph

$G = (V, E)$

nodes / vertices : $V = \{v_1, v_2, \ldots v_n\}$

edges / arcs : $E = \{e_1, e_2, \ldots e_m\}$

**neighbor/adjacent, simple graph, complete graph (m=$\frac{n(n-1)}{2}$)**

**Directed/Undirected graph, path, simple paths**

**Connected, strongly connected, weakly connected**

## Node degree

Undirected : $\sum degree(x) = 2|E|$

Directed : $\sum in\text{-}degree(x) = \sum out\text{-}degree(x) = |E|$

**source/sink**

## cycle

path starting and finishing at the same node

**simple cycle, acyclic graph, directed acyclic graph (DAG)**

**Sparse graph** : $|E| << |V|^2, |E| \approx \Theta(|V|)$

**Dense graph** : $|E| \approx \Theta(|V|^2)$

# Graph Representation

## Adjacency Matrix

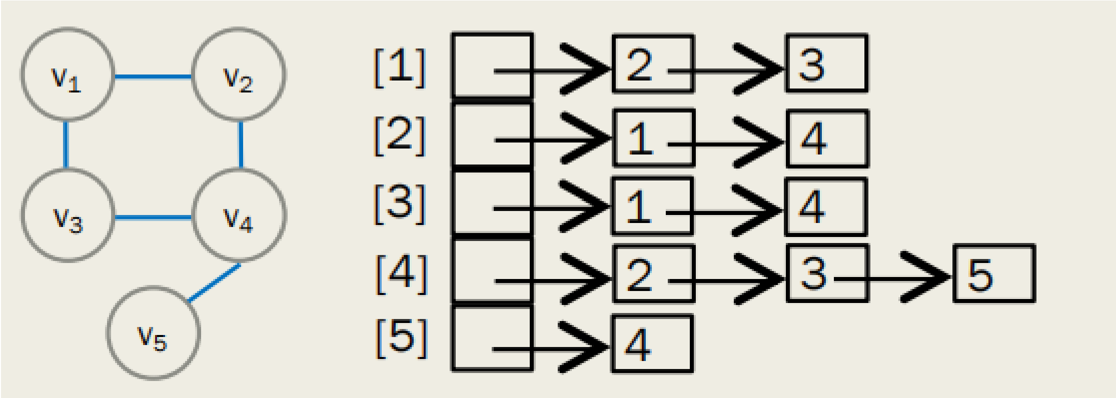
$|V| \times |V|$ matrix representing a graph

Unweighted graph : $A_{ij} = 1$ if there is an edge between $v_i$ and $v_j$ , $0$ if there is no edge.

Weighted graph : $A_{ij}$ is the weight of edge between $v_i$ and $v_j$ , $\infty$ if there is no edge.

## Adjacency List

Use a link list for each node to store all nodes adjacent to this node:




Space complexity $\mathcal{O}(|E|+|V|)$

# Graph Search, Topological Sorting

**Def:** visit every nodes exactly once.

Two common methods : BFS/DFS

## Depth-First Search (DFS)

```
void dfs(int u) {
    visited[u] = true;
    for(auto v:E[u]) if(!visited[v])
        dfs(v);
}
```

## Breadth-First Search (BFS)

```
queue<int>q;
void bfs(int S) {
    q.push(S); inqueue[S] = true;
    while(!q.empty()) {
        int u = q.front();
        for(auto v:E[u]) if(!inqueue[v])
            q.push(v) , inqueue[v] = true;
    }
}
```

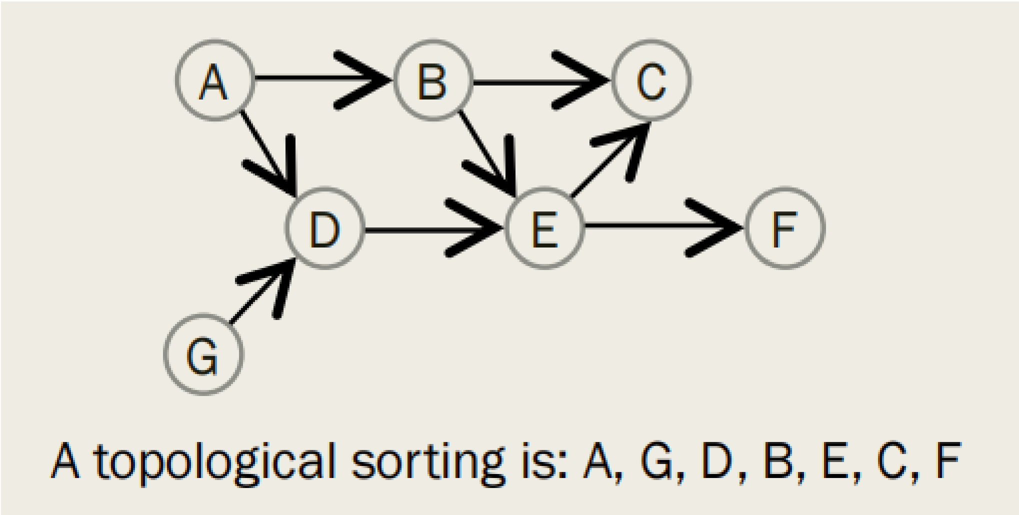
Time complexity : $\mathcal{O}(|V|^2)$ for adjacency matrix, $\mathcal{O}(|V|+|E|)$

# Topological Sorting

Sorting the nodes **(of a directed graph)** in a sequence such that for each directed edge $(v_i, v_j)$

Notice that the topological order is **not unique** for most random **DAG** ( a graph with cycle doesn't have a possible topological order )

example :




another possible order: $G, A, B, D, E, C, F$

**Code :**

```
queue<int>q;
vector<int>order;
vector<int> TopologicalSort() {
    for(int x = 1; x <= n; ++x) if(!in_degree[x])
        q.push(x); // Enqueue all in-degree 0 nodoes
    while(!q.empty()) {
        int u = q.front(); q.pop();
        order.push_back(u);
        for(auto v:E[u]) {
            in_degree[v]--;
            if( in_degree[v] == 0 )
                q.push(v); // neighbor's in_degree becomes 0
        }
    }
    return order;
}
```

**Time complexity :** $\mathcal{O}(|V| + |E|)$

# Minimum Spanning Tree

**Tree : acyclic, connected undirected** graph. $|E| = |V| - 1$ , any connected graph with $N$ nodes and $N - 1$ edges is a tree

**Spanning Tree :** Subgraph of $G$ that have all nodes of $G$ and is a tree.

**Minimum Spanning Tree :** The spanning tree with minimum sum of all edge weights

## Prim's Algorithm

Basic idea : keep adding nodes to the tree greedily until $T$ contains all $N$ nodes.

**Procedure :**

- Arbitrarily pick one nodes $s$ , $T = \{s\}, T' = V - \{s\}$.
- While $T' \neq \emptyset$, set the edge $e = (a, b, w)$ with **smallest weight** connecting nodes between $T$ and $T'$. That is, $a \in T, b \in T'$.
- To get the smallest edge dynamically, just keep track of $D(v)$ for each $v \in T'$ that $D(v)$ means the smallest edge from $T$ that connecting $v'$.
- Whenever adding a node $a$ into $T$, eunumurate all adjacent nodes $b$ and update $D(b)$.

**Code :**

```
int prim() {
    int ans = 0;
    for(int i = 0; i <= n; ++i) dis[i] = INF;
    added[1] = true;
    for(auto [v,w]:E[1]) dis[v] = w;
    for(int i = 1; i <= n-1; ++i) { // adding n-1 edges
        int u = 0;
        for(int j = 1; j <= n; ++j)
            if(!added[j] && dis[j] < dis[u])
                u = j;
        // find the smallest edge
        ans += dis[u];
        added[u] = true;
        for(auto [v,w]:E[u])
            dis[v] = min( dis[v] , w );
        // update the D(v)
    }
    return ans;
}
```

## Kruskal's Algorithm

# Shortest Path Problem

**Def :** Shortest path between the given nodes.

For unweighted graphs ( or say all weight is 1 ) , we can directly use $BFS$.

## Dijkstra's Algorithm

For more general situation, for weighted graph **with non-negative edge**.

Basic idea : each time, we choose the closest node to the start node, to update other's distance, obviously, this node's distance won't be updated again.

**Procedure** :

- Initialization : let $D(s) = 0$ and $D(v) = \infty$ for other nodes. $T = \{s\}, T' = V - \{s\}$
- While $T'$ is not empty, choose $u \in T'$ such that $D(u)$ is the smallest.
- Update other adjacent node's distance like $D(v) = \min(D(v), D(u) + w(u, v))$.

```
int Dijkstra(int S,int T) {
    for(int i = 0; i <= n; ++i) dis[i] = INF;
    dis[S] = 0; added[S] = true;
    for(auto[v,w]:E[S]) dis[v] = w;
    for(int i = 1; i <= n-1; ++i) {
        int u = 0;
        for(int j = 1; j <= n; ++j)
            if(!added[j] && dis[j] < dis[u])
                u = j;
        // find the closest node
        added[u] = true;
        for(auto [v,w]:E[u])
            dis[v] = min(dis[v] , dis[u] + w);
        // update D(v)
    }
    return dis[T];
}
```

**Time complexity :** $\mathcal{O}(|V|^2)$

How to optimize ?

Heap !

**Binary heap :** $\mathcal{O}(|V| \log |V| + |E| \log |V|)$

**Fibonacci heap :** $\mathcal{O}(|V|\log|V|+|E|)$

# Dynamic Programming

Optimization problem

characteristics of dynamic programming problems :

- Solving problem can be divided into solving subproblems
- This problem's answer can be deduced / calculated by subproblems' answer

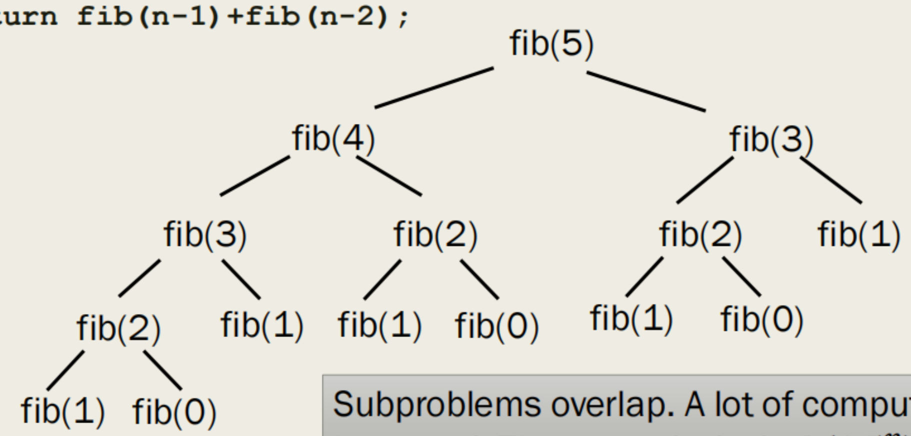
Then different from divide and conquer, we use array (usually) or other structures to store answers and don't recalculate or resolve a same subproblem.

Save both memory and time

Some progressive examples :

① Fibonacci Sequence :

```
int fib(int n) {
  if(n <= 1) return n;
  return fib(n-1)+fib(n-2);
}
```




Subproblems overlap. A lot of computation is wasted. Time complexity is $\Omega(1.5^n)$.

- We can also compute the Fibonacci sequence in iterative way:

```
int fib(int n) {
  f[0] = 0; f[1] = 1;
  for(i = 2 to n)
    f[i] = f[i-1]+f[i-2];
  return f[n];
}
```

- Time complexity is $\Theta(n)$.

② Unique Paths

## 62. Unique Paths

Medium 15.9K 420

Companies

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in time.

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

③ Matrix-Chain Multiplication