```
InsertionSort(A)
1.  for j = 2 to length[A]
2.     key = A[j]
3.     i = j - 1
4.     while i > 0 and A[i] > key do
5.        A[i+1] = A[i]
6.         i = i - 1
7.     A[i+1] = key
```
Best case: n Worst: n^2

```
MergeSort(A, p, r)
1.  if p < r then
2.     q = ⌊(p+r) / 2⌋          splitting criteria
3.     MergeSort(A, p, q)     } forward divide
4.     MergeSort(A, q+1, r)   }
5.     Merge(A, p, q, r)        backward merge
```
worst **n log n**

Ω is a **lower bound**
- Code takes <u>at least</u> this long to run

O is an **upper bound**
- Code takes <u>at most</u> this long to run

Θ is "equal to"
- Code takes <u>exactly</u>* this long to run
- *Except for constant factors and lower order terms
- Only exists when $O = \Omega$!

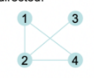
For an **acyclic** graph G:
- If G is connected, it is called a *tree*;
- Otherwise, it is called a *forest*.

For an **undirected** graph G:
- If every pair of vertices is connected by at least one path, then graph G is *connected*.

The adjacency matrix is a way of representing a graph as a matrix where the rows and columns correspond to the vertices, and the entries correspond to the edges. Specifically, if the graph has n vertices, then the adjacency matrix is an n-by-n matrix, where the entry in row i and column j is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.
To analyze the time complexity of a DFS traversal using an adjacency matrix, we need to count the number of basic operations performed by the algorithm as a function of the size of the instance. The basic operations in a DFS traversal are visiting a vertex, marking it as visited, and recursively visiting its unvisited neighbors.
If we use a standard depth-first search algorithm, the time complexity of the algorithm can be expressed in terms of the number of vertices and edges in the graph. In particular, the time complexity of DFS on a graph represented by an adjacency matrix is O(V^2), where V is the number of vertices in the graph. This is because each vertex needs to be visited at most once, and for each vertex, we need to examine all V adjacent vertices in the matrix.
It's worth noting that if the graph is sparse, i.e., it has relatively few edges compared to the number of vertices, then it may be more efficient to use an adjacency list representation instead of an adjacency matrix. In that case, the time complexity of DFS would be O (V + E), where E is the number of edges in the graph.

- Adjacency List vs. Adjacency Matrix

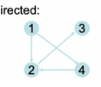
undirected:



1	2	4	/	
2	1	3	4	/
3	2	/		
4	1	2	/	

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

directed:



1	2	4	/
2	3	/	
3	/		
4	2		

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	0	0

BFS Strategy:
- Explores nodes expanded the frontier between visited and unvisted nod uniformly
- Nodes at distance k are visited before any node at distance k+1

In DFS, we start at a node, visit one of its neighbors, then visit one of that neighbor's neighbors, and so on. We keep track of the nodes we have visited so that we don't visit them again. We use a stack data structure to keep track of the order in which we visit nodes. The order in which we visit nodes using DFS is such that we go as deep as possible before backtracking. O(V + E)

Implementation:
- Uses a Worklist Algorithmic Strategy with a Queue Data Structures
- Examines each node and place its successors in a queue to be examined later
- Halts when all nodes have been visited.

```
function DFS(graph G)
  for each node u ∈ V[G] do
     color[u] = white;
     pred[u] = NIL;
     visit[u] = 0;
  time = 1;
  for each node u ∈ V[G] do
    if color[u] = white then
       DFS-Visit(u);
```

```
function DFS-Visit(node u)
  color[u] = gray;
  dist[u] = time;
  time = time + 1;
  for each v ∈ Adj[u] do
     if color[v] = white then
        pred[v] = u;
        DFS-Visit(v);
  color[u] = black;
  visit[u] = time;
  time = time + 1;
```

```
function BFS(Graph G, node s)
  for each node u ∈ V[G] - { s } do
    color[u] = white; dist[u] = ∞; pred[u] = NIL;
  color[s] = gray; dist[s] = 0; pred[s] = NIL;
  Q = { s };
  while (Q ≠ ∅) do
    u = Dequeue (Q);
    for each v ∈ Adj[u] do
      if color[v] = white then
        color[v] = gray;
        d[v] = d[u] + 1;
        pred[v] = u;
        Enqueue (Q, v);
    color[u] = black;
```

To analyze the time complexity of a DFS traversal using an adjacency matrix, we need to count the number of basic operations performed by the algorithm as a function of the size of the instance. The basic operations in a DFS traversal are visiting a vertex, marking it as visited, and recursively visiting its unvisited neighbors.
If we use a standard depth-first search algorithm, the time complexity of the algorithm can be expressed in terms of the number of vertices and edges in the graph. In particular, the time complexity of DFS on a graph represented by an adjacency matrix is O(V^2), where V is the number of vertices in the graph. This is because each vertex needs to be visited at most once, and for each vertex, we need to examine all V adjacent vertices in the matrix.
Topological - The algorithm works by first initializing the indegree of all vertices to 0 and then calculating the indegree of each vertex by iterating through the edges in the graph. The algorithm then creates a queue to store vertices with indegree 0, i.e., vertices that have no incoming edges.
The algorithm then performs a BFS-like traversal of the graph by repeatedly dequeuing a vertex u with indegree 0, adding it to the sorted list, and then reducing the indegree of its neighbors by 1. If any of the neighbors of u now have indegree 0, they are added to the queue.
The algorithm continues until all vertices have been added to the sorted list or until the queue is empty. I a topological sorting is found, the function returns the sorted list. If the sorted list has fewer than n vertices, then the graph contains at least one cycle, and the function returns an empty vector to indicate failure. The asymptotic complexity of the algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the graph, as the algorithm needs to visit each vertex and each edge exactly once
The Strongly-Connected components algorithm is used to identify the strongly-connected components of a directed graph. A strongly-connected component is a subgraph where every vertex is reachable from every other vertex in the subgraph. The algorithm works by performing a depth-first search (DFS) on the graph, and then performing a second DFS on the reverse of the graph, which is the same graph with all edges reversed. The vertices are then sorted in order of their finishing times from the second DFS, and this order is used to group the vertices into strongly-connected components.

The asymptotic complexity of both algorithms depends on the size of the graph and the way it is represented. For example, if the graph is represented using an adjacency list, BFS and DFS both have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. If the graph is represented using an adjacency matrix, BFS and DFS both have a time complexity of O(V^2).
In terms of the order in which nodes are visited and edges are explored, BFS visits nodes in order of their distance from the starting node, while DFS explores the edges of the graph in a depth-first manner, meaning it visits all the edges of one branch before moving on to another branch.

```
function Topological-Sort (Graph G)
  L = ∅ ;                  // List of nodes
  Q = ∅;                   // Queue of nodes
  for each v ∈ G do
    if v has no incoming edges (w,v) then
      Enqueue(Q,v);
  while Q ≠ ∅ do
    u = Head(Q);
    Eliminate edges (u,v);
    if v has no incoming edges (w,v) then
      Enqueue(Q,v);
    Dequeue(Q);
    Append u to end of list L;
```
O(V+E)

```
function Topological-Sort (Graph G)
  Execute DFS(G) and compute visit[v] for each node v;
  During DFS-visit of v when complete visit for v
        prepend node v to list L;
  return list L;
```

Ir tirando um a um e ir adicionando à quele os que ficam sem setas a apontar para eles e formar uma lista a partir daí

```
siftDown(A, i)
  left = Left(i)
  right = Right(i)
  if (left ≤ heapSize and A[left] > A[i]) then
    largest = left
  else
    largest = i
  if (right ≤ heapSize and A[right] > A[largest]) then
    largest = right
  if (largest ≠ left) then
    swap(A[i], A[largest])
    siftDown(A, largest)
```

Min-Heap Operations

A min-heap is a binary tree in which the parent node is always less than or equal to its children. It is a key data structure used for sorting and priority queue implementations. The siftUp and siftDown operations are used to maintain the heap property after a node is inserted or deleted.

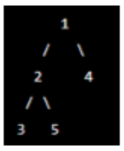
The siftUp operation is used after an element is inserted into the heap. It compares the element with its parent and swaps them if the parent is larger. This operation is repeated until the parent is smaller or until the root is reached. The complexity of the siftUp operation is O(log n), where n is the number of nodes in the heap.

The siftDown operation is used after an element is deleted from the heap. It swaps the deleted node with its smallest child until the heap property is restored. This operation is repeated until the smallest child is greater than the node or until the leaf is reached. The complexity of the siftDown operation is also O(log n).

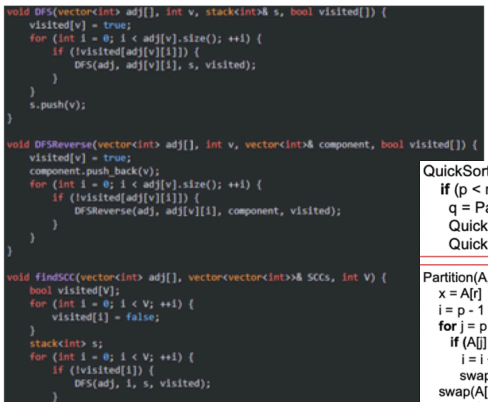
To determine the configuration of a min-heap or max-heap after a sequence of value insertions, we start with an empty heap and add each element one at a time using the siftUp operation. The final configuration of the heap depends on the order in which the elements were inserted.

For example, if we insert the elements [3, 2, 4, 1, 5] into an empty min-heap, the resulting heap would be:

```
siftUp(A, i)
  j = Parent(i)
  if (j > 0 and A[j] < A[i]) then
    swap(A[i], A[j])
    siftUp(A, j)
```



The complexity of this sequence of insertions is O (n log n), where n is the number of elements inserted, since each insertion requires a siftUp operation with a complexity of O (log n).



```
void DFS(vector<int> adj[], int v, stack<int>& s, bool visited[]) {
    visited[v] = true;
    for (int i = 0; i < adj[v].size(); ++i) {
        if (!visited[adj[v][i]]) {
            DFS(adj, adj[v][i], s, visited);
        }
    }
    s.push(v);
}

void DFSReverse(vector<int> adj[], int v, vector<int>& component, bool visited[]) {
    visited[v] = true;
    component.push_back(v);
    for (int i = 0; i < adj[v].size(); ++i) {
        if (!visited[adj[v][i]]) {
            DFSReverse(adj, adj[v][i], component, visited);
        }
    }
}

void findSCC(vector<int> adj[], vector<vector<int>>& SCCs, int V) {
    bool visited[V];
    for (int i = 0; i < V; ++i) {
        visited[i] = false;
    }
    stack<int> s;
    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
            DFS(adj, i, s, visited);
        }
    }
    for (int i = 0; i < V; ++i) {
        visited[i] = false;
    }
    while (!s.empty()) {
        int v = s.top();
        s.pop();
        if (!visited[v]) {
            vector<int> component;
            DFSReverse(adj, v, component, visited);
            SCCs.push_back(component);
        }
    }
}
```

```
QuickSort(A, p, r)
  if (p < r) then
    q = Partition(A, p, r)
    QuickSort(A, p, q-1)
    QuickSort(A, q+1, r)
```

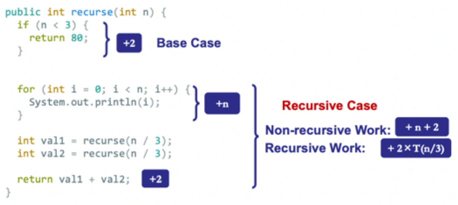
```
Partition(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r-1 do
    if (A[j] ≤ x) then
      i = i + 1
      swap(A[i], A[j])
  swap(A[i+1], A[r])
  return i+1
```

lere is a brief explanation of what each function does:

- DFS: performs a DFS on the graph to find the vertices in the order of their finishing times.
- DFSReverse: performs a DFS on the reverse of the graph to find the strongly-connected component.
- findSCC: performs the entire algorithm by first performing a DFS on the graph to find the finishing times, and then performing a DFS on the reverse of the graph to find the strongly-connected components.

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

<u>Master Theorem:</u>



```
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }
  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }
  int val1 = recurse(n / 3);
  int val2 = recurse(n / 3);

  return val1 + val2;
}
```

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

where $f(n)$ is $\Theta(n^c)$

If	$\log_b a < c$ then	$T(n) \in \Theta(n^c)$
If	$\log_b a = c$ then	$T(n) \in \Theta(n^c \log n)$
If	$\log_b a > c$ then	$T(n) \in \Theta(n^{\log_b a})$

a=2 b=3 and c=1
$y = \log_b x$ is equal to $b^y = x$
$\log_3 2 \cong 0.63$
$\log_3 2 < 1$
We're in case 1
$T(n) \in \Theta(n)$

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n + 2 & \text{otherwise} \end{cases}$$

1. Greedy choice property: A globally optimal solution can be obtained by making a locally optimal (greedy) choice.

2. Optimal substructure property: A problem can be solved optimally by breaking it down into subproblems and solving each subproblem optimally.

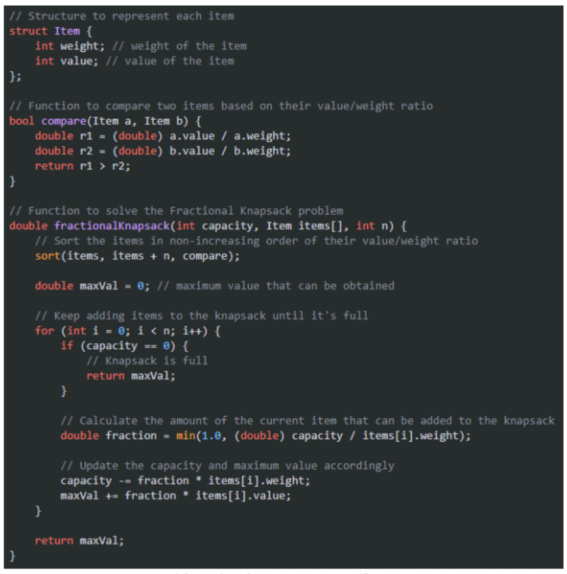
Activity Selection

Greedy choice property: At each step, the algorithm selects the activity that has the earliest end time among the remaining activities. This is a locally optimal choice, as it ensures that we can select the maximum number of non-overlapping activities starting from the current activity. By selecting the activity with the earliest end time, we leave room for selecting other activities that start later. We can show that this locally optimal choice leads to a globally optimal solution by contradiction. Suppose there exists a globally optimal solution that does not include the activity with the earliest end time. We can always replace this activity with the one with the earliest end time and obtain a new solution that is at least as good as the previous one.

Optimal substructure property: We can break the problem of selecting the maximum number of non-overlapping activities into subproblems by considering the activities starting from the second one and selecting the maximum number of non-overlapping activities from this subset. This is a subproblem of the original problem, and we can solve it optimally using the same algorithm recursively. The optimal solution to the original problem can then be obtained by adding the first activity to the solution of the subproblem.

Fractional Knapsack

O algoritmo greedy encontra a solução ótima para o fractional Knapsack problem porque ordena os objetos com base na relação valor/peso e seleciona primeiro os objetos com maior ratio. Contudo, a mesma abordagem não encontra a solução ótima para o integer Knapsack problem, onde os objetos não podem ser divididos em partes fracionárias. Um objeto com um ratio elevado pode ter um peso superior à capacidade restante da mochila, impedindo a adição de objetos com uma relação valor/peso menor, mas com um peso que caiba na capacidade.



```
// Structure to represent each item
struct Item {
    int weight; // weight of the item
    int value; // value of the item
};

// Function to compare two items based on their value/weight ratio
bool compare(Item a, Item b) {
    double r1 = (double) a.value / a.weight;
    double r2 = (double) b.value / b.weight;
    return r1 > r2;
}

// Function to solve the Fractional Knapsack problem
double fractionalKnapsack(int capacity, Item items[], int n) {
    // Sort the items in non-increasing order of their value/weight ratio
    sort(items, items + n, compare);

    double maxVal = 0; // maximum value that can be obtained

    // Keep adding items to the knapsack until it's full
    for (int i = 0; i < n; i++) {
        if (capacity == 0) {
            // Knapsack is full
            return maxVal;
        }

        // Calculate the amount of the current item that can be added to the knapsack
        double fraction = min(1.0, (double) capacity / items[i].weight);

        // Update the capacity and maximum value accordingly
        capacity -= fraction * items[i].weight;
        maxVal += fraction * items[i].value;
    }

    return maxVal;
}
```

Kruskal:

```
function MST-Kruskal(G,w)
  A = {};
  foreach v ∈ V[G] do
    MakeSet(v); // creates a cluster for v
  Sort edges ∈ E by non-decreasing order of weight;
  foreach (u,v) ∈ E[G] in sorted order do
    if FindSet(u) ≠ FindSet(v) then
      // (u,v) is the lightest and safe edge for A
      A = A U {(u,v)};
      Union(u,v); // merge clusters for u and v
return A;
```

Possible to define O(E log E)

Given that $E < V^2$, we get O(E log V)

Implementation:

- Algorithm maintains a forest of trees or subset $A \subset E$
- Uses a disjoint-sets data structure for representing and merging clusters
- Each set or cluster represents a sub-tree of the final MST

Prim's O(E log V)

```
function MST-Prim(G,w,r)                                   root
  Q = V[G];                // Priority queue Q
  foreach u ∈ Q do         // Initialization              weights
      key[u] = ∞;
  key[r] = 0;
  pred[r] = NIL;           // Keep track of tree A
  while Q ≠ ∅ do
      u = ExtractMin(Q); // Pick closest unprocessed node      O(E)
      // ∃ (u,v) safe and light edge, for tree A
O(V)  foreach v ∈ Adj[u] do
        if (v ∈ Q and w(u,v) < key[v]) then  // Check if node is not in MST
          pred[v] = u;
O(log V)  key[v] = w(u,v);  // Min Heap Q is updated!

      O(log V)
```

Builds MST from a Root node

- Algorithm Starts with the Root node
- Expands Tree one Edge at a time
- **At each step the Algorithm choose the Lightest Safe Edge**

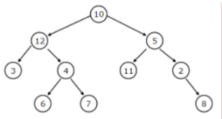
Using Priority Queue Q

key[v]:

- lowest edge weight connecting v to a node in the Tree

pred[v]:

- predecessor of v in the Tree



Levelorder tree traversal
10, 12, 5, 3, 4, 11, 2, 6, 7, 8
Inorder tree traversal
3, 12, 6, 4, 7, 10, 11, 5, 2, 8
Preorder tree traversal
10, 12, 3, 4, 6, 7, 5, 11, 2, 8
Postorder tree traversal
3, 6, 7, 4, 12, 11, 8, 2, 5, 10

- **Preorder:** visit the root; then visit preorder the children left to right
- **Postorder:** visit postorder the children left to right, then the root
- **Inorder (binary trees):** visit inorder the left child, then the root and the

Dijkstra's Algorithm is a greedy algorithm that solves the shortest path problem for a weighted directed graph with non- negative edge weights, producing a shortest path tree. It works by maintaining a set of vertices whose shortest distance from the source node is known, and a set of vertices whose shortest distance is unknown. At each step, it selects the vertex with the smallest known distance from the source node and updates the distances of its adjacent vertices if a shorter path is found.

Dijkstra: O((V+E) log V)

```
Dijkstra(Graph G, Function w, Node s)
  InitializeSingleSource(G,s);
  S = ∅;
  Q = V[G];                    // Priority queue Q
  while Q ≠ ∅ do
      u = ExtractMin(Q);                              O(log V)
      S = S ∪ {u};
      foreach v ∈ Adj[u] do
        Relax(u,v,w);          // update Q

O(V)        O(E)        O(log V)
```

```
Relax(u, v, w)
 if d[v] > d[u] + w(u, v) then
    d[v] := d[u] + w(u, v)
    parent[v] := u
```

Negative link weight: The Bellman-Ford algorithm works; Dijkstra's algorithm doesn't.

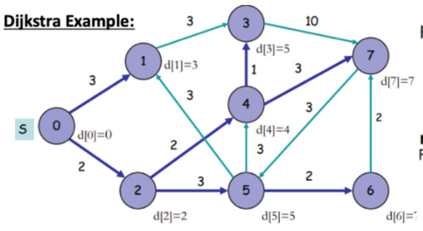
- Also based on Edge-Relaxation
- Maintains distance associated with each node

Dijkstra vs Prim:

1. **Dijkstra's algorithm finds the shortest path, but Prim's algorithm finds the MST**
2. Dijkstra's algorithm can work on both directed and undirected graphs, but Prim's algorithm only works on undirected graphs
3. Prim's algorithm can handle negative edge weights, but Dijkstra's algorithm may fail to accurately compute distances if at least one negative edge weight exists

In practice, Dijkstra's algorithm is used when we want to save time and fuel traveling from one point to another. Prim's algorithm, on the other hand, is used when we want to minimize material costs in constructing roads that connect multiple points to each other.

Dijkstra Example:



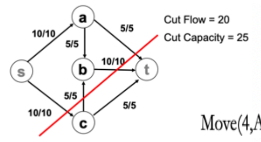
Number of flow increases is O(V E)

Execution Time is O(V E²)

- O(E) due to BFS and the increase of flow at each step

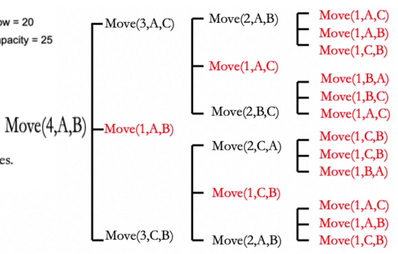
```
Ford-Fulkerson-Method(Graph G, node s, node t)
  initialize flow f to 0;
  while (exists an augmenting path P) do
      increase flow along P;
      update residual network;
return f;                                        O(E |f*|)
```

```
Ford-Fulkerson(Graph G, node s, node t)
  foreach (u,v) ∈ E[G] do
      f[u,v] = 0;
      f[v,u] = 0;
  while exists an augmenting path p in residual network G_f do
    compute c_f(p);
    foreach (u,v) ∈ p do
      f[u,v] = f[u,v] + c_f(p)  // Increase flow value
      f[v,u] = - f[u,v]
```



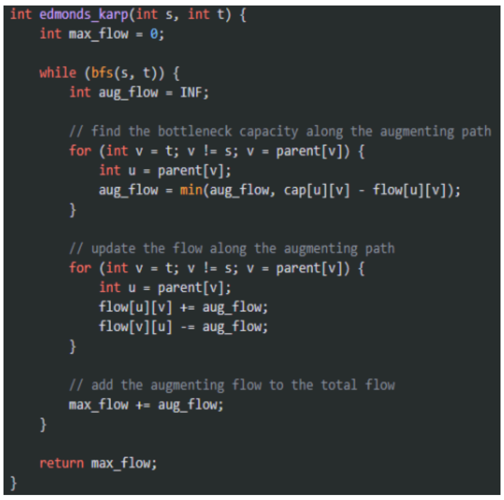
Cut includes only positive capacity edges.

liquid flow negative flows.



Hanoi(n,A,B,C) = Hanoi(n-1,A,C,B)+ Hanoi(1,A,B,C) + Hanoi(n-1,C,B,A)

The Edmonds-Karp algorithm can be used to solve many types of network flow problems, including the Maximal Bipartite Matching problem. In this problem, we are given a bipartite graph and we want to find a matching that covers as many vertices as possible. A matching is a set of edges such that no two edges share a common vertex.
To use the Edmonds-Karp algorithm for this problem, we first convert the bipartite graph into a flow network by adding a source node and a sink node and connecting the source node to one partition of the bipartite graph and the other partition to the sink node. We then set the capacity of each edge to 1, indicating that each vertex can be matched with at most one other vertex. The maximum flow in this network corresponds to the maximum matching in the bipartite graph.
The correctness of the Edmonds-Karp algorithm can be proved using the Max-Flow Min-Cut theorem. This theorem states that the maximum flow in a network is equal to the minimum cut, where a cut is a partition of the vertices into two disjoint sets such that the source is in one set and the sink is in the other set. The minimum cut corresponds to the minimum number of edges that need to be removed to disconnect the source from the sink.
In the Edmonds-Karp algorithm, we always choose the shortest augmenting path from the source to the sink, which corresponds to a cut in the residual graph. Therefore, the flow along this path is equal to the capacity of the minimum cut. By repeating this process, we eventually find the maximum flow, which is equal to the capacity of the minimum cut.
The time complexity of the Edmonds-Karp algorithm can be improved to O((VE)log(U)) using the Dinic's algorithm, where U is the maximum capacity of any edge in the network. However, this improvement is not necessary for most



```
int edmonds_karp(int s, int t) {
    int max_flow = 0;

    while (bfs(s, t)) {
        int aug_flow = INF;

        // find the bottleneck capacity along the augmenting path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            aug_flow = min(aug_flow, cap[u][v] - flow[u][v]);
        }

        // update the flow along the augmenting path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            flow[u][v] += aug_flow;
            flow[v][u] -= aug_flow;
        }

        // add the augmenting flow to the total flow
        max_flow += aug_flow;
    }

    return max_flow;
}
```

Divide and conquer algorithms are a class of algorithms that solve a problem by breaking it down into smaller subproblems, solving those subproblems independently, and then combining the solutions to the subproblems to obtain the solution to the original problem. The key idea is to divide the problem into smaller subproblems that are easier to solve, and then combine the solutions to these subproblems to obtain the solution to the original problem.

One of the most important aspects of divide and conquer algorithms is the formulation of a recurrence relation that describes the running time of the algorithm. The recurrence relation expresses the running time of the algorithm as a function of the size of the input.