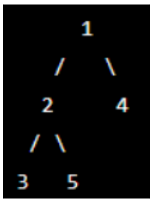
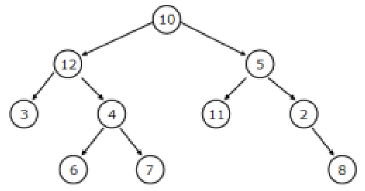
Min-Heap Insertion Example:

[3, 2, 4, 1, 5]



O(nlog(n))



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 then the right child

SiftDown/SiftUp:
O(log(n))

Maximal Bipartite:
O(V E)

BFS/DFS/Topological Sorting:
O(|E| + |V|)

Prim/Kruskal:
O(E log V)

Ford-Fulkerson:
O(|E|*f)

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

Edmonds-Karp:
O(VE^2)

Merge-Sort:

Time Complexity: O(n log n)

MergeSort(1,n) = MergeSort(1,n/2)+ MergeSort(n/2+1,n) + Merge(1,n)

```
MergeSort(A, p, r)
if p < r then
    q = ⌊(p+r) / 2⌋          splitting criteria
    MergeSort(A, p, q)    }  forward divide
    MergeSort(A, q+1, r)  }
    Merge(A, p, q, r)        backward merge
```

Knapsack:

Fractional Knapsack: O(n log(n))

- Algorithm:

```
function fillUpKnapsackGreedy(v, w, W)
  weight = 0;
  while weight < W do
    select element i with maximal v_i/w_i
    if (w_i + weight ≤ W) then
      x_i = 1; weight += w_i
    else
      x_i = (W-weight)/ w_i; weight = W
```

Execution Time:
O(n), or O(n log n)

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 restante.

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;
```

Dijkstra:

```
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
```

Prim:

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

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.

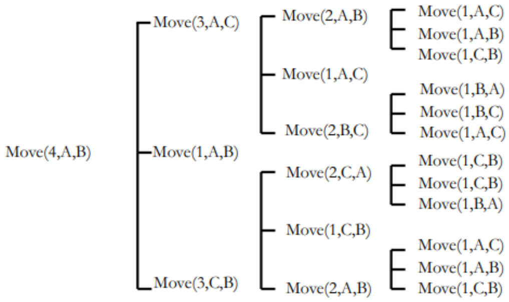
Hanoi Towers:

Time Complexity: O(2^n)

Given the recursive definition:

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

Move operations results in the sequence below:



Master Theorem:

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

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

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

$T(n) \in \Theta(n)$

- Property 1 - The Greedy-choice Property
  - Generally casted as a globally optimal solution that can be arrived at by making a locally optimal (greedy) choice.
- Property 2 – The Optimal substructure Property
  - Generally casted as an optimal solution to the problem contains within it optimal solutions to subproblems.

O is an **upper bound**

- Code takes at most this long to run

Ω is a lower bound

- Code takes at least this long to run

Θ is "equal to"

- Code takes exactly* this long to run
- *Except for constant factors and lower order terms
- Only exists when *O* = Ω!

To compare two functions and determine which one will dominate for larger and small instance sizes, we can look at their growth rates. If one function has a higher growth rate than the other, it will dominate for larger input sizes. For smaller input sizes, the dominating function may depend on the specific values of the input. We can also use the limit definition of asymptotic notation to compare two functions.

For example, consider the following two functions:

f(n) = 2n^2 + 5n + 1; g(n) = n^3 + 10n.

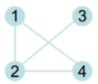
As n approaches infinity, the dominant term in f(n) is 2n^2, while the dominant term in g(n) is n^3. Therefore, g(n) will dominate f(n) for large input sizes. For small input sizes, however, f(n) may dominate g(n) depending on the specific values of n.

DFS/BFS using Adjacency Matrix: O(V^2) 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.

DFS/BFS using Adjacency List: O(V + E)

- Adjacency List vs. Adjacency Matrix

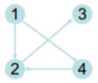
undirected:



```
1 |2  4  /
2 |1  3  4  /
3 |2  /
4 |1  2  /
```

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array}$$

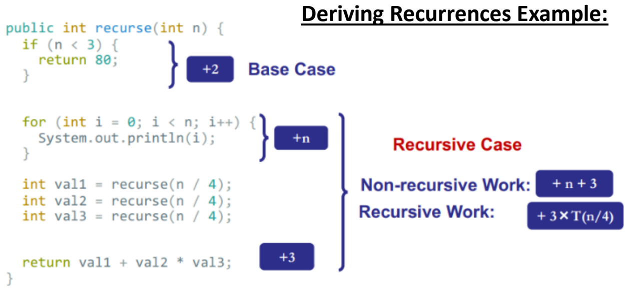
directed:



```
1 |2  4  /
2 |3  /
3 |/
4 |2
```

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

Deriving Recurrences Example:



```
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 / 4);
  int val2 = recurse(n / 4);
  int val3 = recurse(n / 4);

  return val1 + val2 * val3;
}
```

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 3T\left(\frac{n}{4}\right) + n + 3 & \text{otherwise} \end{cases}$$

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

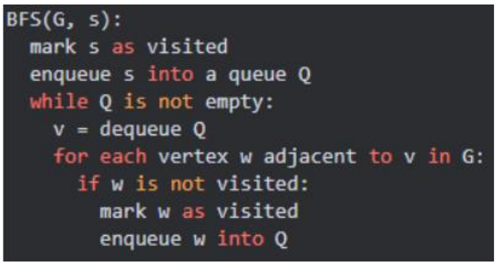
```
Partition(A, p, r)
   x = A[r]
   i = p - 1
   for j = p to r-1 do
     if (A[j] ≤ x) then
        i = i + 1
        swap(A[i], A[j])
   swap(A[i+1], A[r])
   return i+1
```

Activity Selection Example: Prove the two properties of optimal greedy:

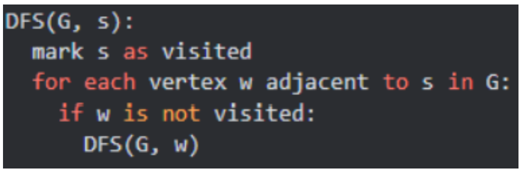
To prove the two properties of optimal greedy choice and optimal substructure for this algorithm, we can reason as follows:

1. 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.

2. 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.



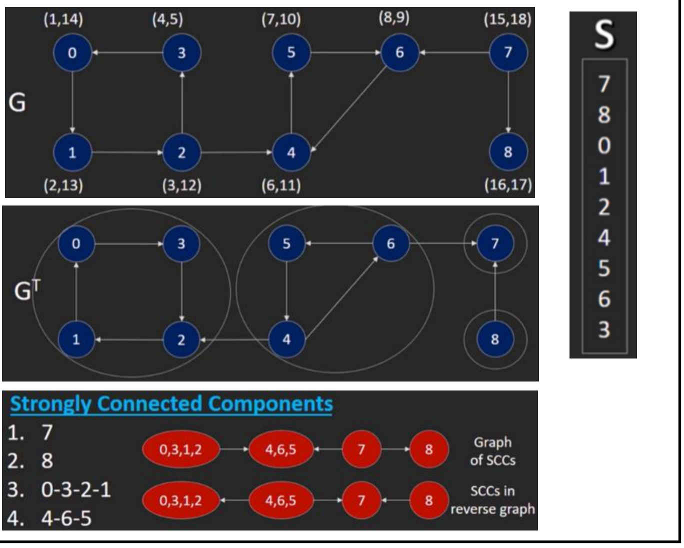
```
BFS(G, s):
  mark s as visited
  enqueue s into a queue Q
  while Q is not empty:
    v = dequeue Q
    for each vertex w adjacent to v in G:
      if w is not visited:
        mark w as visited
        enqueue w into Q
```



```
DFS(G, s):
  mark s as visited
  for each vertex w adjacent to s in G:
    if w is not visited:
      DFS(G, w)
```

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

Strongly Connected Components (Kosaraju's Algorithm Example):

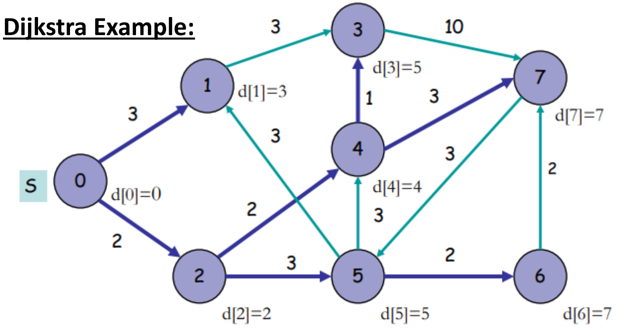


1. 7
2. 8
3. 0-3-2-1
4. 4-6-5

```
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;
```

```
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]
```

Dijkstra Example:



S
0
d[0]=0
1
d[1]=3
2
d[2]=2
3
d[3]=5
4
d[4]=4
5
d[5]=5
6
d[6]=7
7
d[7]=7
3
3
2
3
10
1
3
3
2
3
3
3
2
2