<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford–Fulkerson Method (Maximum Flow)</td>
<td>$O(E</td>
</tr>
<tr>
<td>Edmonds–Karp algorithm (Implementation of FFM)</td>
<td>$O(VE^3)$</td>
</tr>
<tr>
<td>Ford–Fulkerson Method (Maximum Bipartite Matching)</td>
<td>$O(E</td>
</tr>
<tr>
<td>Approximate Minimum Vertex Cover</td>
<td>$O(V+E)$</td>
</tr>
<tr>
<td>Exact Subset-Sum</td>
<td>$O(\exp)$</td>
</tr>
<tr>
<td>Approx. Subset-Sum</td>
<td>$O(\text{poly})$</td>
</tr>
<tr>
<td><strong>[GraphTrav]</strong> BFS: Breadth-First Search</td>
<td>$O(V+E)$</td>
</tr>
<tr>
<td><strong>[GraphTrav]</strong> DFS: Depth-First Search</td>
<td>$O(V+E)$</td>
</tr>
<tr>
<td><strong>[ShortestPath]</strong> Topological Sort (DAG Only; Allows w&lt;0; Single-Source)</td>
<td>$O(V+E)$</td>
</tr>
<tr>
<td><strong>[ShortestPath]</strong> Dijkstra (Allows Cycles; No weight&lt;0; Single-Source)</td>
<td>$O(V^2)$ -&gt; $O(V \cdot \text{log} V)$</td>
</tr>
<tr>
<td><strong>[ShortestPath]</strong> Bellman-Ford (Allows Cycles; Allows weight&lt;0; Single-Source)</td>
<td>$O(V \cdot E)$</td>
</tr>
<tr>
<td>Bellman-Ford (Negative Cycle Detection)</td>
<td>$O(V \cdot E)$</td>
</tr>
<tr>
<td><strong>[ShortestPath]</strong> Matrix Multiplication (All-Pair)</td>
<td>$Θ(n^3 \cdot \text{log} n)$</td>
</tr>
<tr>
<td><strong>[ShortestPath]</strong> Floyd-Warshall (All-Pair)</td>
<td>$Θ(n^3)$</td>
</tr>
<tr>
<td><strong>[MST]</strong> Kruskal’s Algorithm (take shortest; for undirected)</td>
<td>$O(E \cdot \text{log} V)$</td>
</tr>
<tr>
<td><strong>[MST]</strong> Prim’s (take nearest; for undirected &amp; connected)</td>
<td>$O(E \cdot \text{log} V)$ -&gt; $O(E+V \cdot \text{log} V)$</td>
</tr>
<tr>
<td>Recursive Activity Selection</td>
<td></td>
</tr>
<tr>
<td>Iterative Activity Selection</td>
<td></td>
</tr>
<tr>
<td><strong>0-1 Knapsack Problem</strong></td>
<td></td>
</tr>
<tr>
<td>Fractional Knapsack Problem</td>
<td></td>
</tr>
<tr>
<td>Huffman (Optimal Prefix Coding)</td>
<td>$O(n \cdot \text{log} n)$ -&gt; $O(n \text{log} \text{log} n)$</td>
</tr>
<tr>
<td><strong>Maximum-Weight Indep. Subset of A Matroid</strong></td>
<td></td>
</tr>
<tr>
<td>Linear Select (Select the k-th-big item with linear time even in worst case)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quick Select ($T(n) = T(n/2) + n$)</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Quick Sort ($T(n) = 2T(n/2) + n$)</td>
<td>$O(n \cdot \text{log} n)$</td>
</tr>
<tr>
<td>Interleaves Two Halves of An Array ($T(n) = 2T(n/2) + n/4$)</td>
<td>$Θ(n \cdot \text{log} n)$</td>
</tr>
</tbody>
</table>

- 🧐: Greedy algorithm.
- ☢️: Dynamic Programming.
- ✫: Divide-and-Conquer.

### Ford–Fulkerson Method (Maximum Flow) $O(E |f^*|)$

**Inputs:** Given a Network $G=(V,E)$ with flow capacity $c$, a source node $s$, and a sink node $t$.  
**Output:** maximum flow $f$ from $s$ to $t$. 

---

The Ford–Fulkerson Method (Maximum Flow) is an algorithm for finding the maximum flow in a flow network. It is based on the Edmonds–Karp algorithm, which is an implementation of the Ford–Fulkerson Method. The algorithm works by repeatedly finding augmenting paths in the residual graph and increasing the flow along these paths until no more augmenting paths can be found. The time complexity of the Ford–Fulkerson Method is $O(E |f^*|)$, where $E$ is the number of edges in the network and $|f^*|$ is the maximum flow. This complexity is due to the fact that the algorithm may take a large number of iterations to find the maximum flow, each iteration requiring a search for an augmenting path.

- **Inputs:** Given a Network $G=(V,E)$ with flow capacity $c$, a source node $s$, and a sink node $t$.  
- **Output:** maximum flow $f$ from $s$ to $t$. 

---

The Edmonds–Karp algorithm is a specific implementation of the Ford–Fulkerson Method, which uses the shortest augmenting path algorithm to find an augmenting path in the residual graph. The time complexity of the Edmonds–Karp algorithm is $O(V \cdot E^2)$, which is due to the use of Dijkstra's algorithm to find the shortest augmenting path at each iteration. Despite this additional computational overhead, the Edmonds–Karp algorithm is often used because it guarantees to find the maximum flow in a network, whereas the Ford–Fulkerson Method may not always find the maximum flow.

- **Inputs:** Given a Network $G=(V,E)$ with flow capacity $c$, a source node $s$, and a sink node $t$.  
- **Output:** maximum flow $f$ from $s$ to $t$. 

---

The Ford–Fulkerson Method (Maximum Bipartite Matching) is an algorithm for finding a maximum matching in a bipartite graph. It is based on the Ford–Fulkerson Method and works by repeatedly finding augmenting paths in the residual graph and increasing the matching along these paths until no more augmenting paths can be found. The time complexity of the Ford–Fulkerson Method (Maximum Bipartite Matching) is $O(E |f^*|)$, where $E$ is the number of edges in the graph and $|f^*|$ is the maximum matching.

**Inputs:** Given a Bipartite Graph $G=(U \cup V, E)$ with matching capacity $c$, a matching $M$, and a matching $M'$.  
**Output:** maximum matching $M'$ from $u$ to $v$. 

---

The Approximate Minimum Vertex Cover is an approximation algorithm for finding a minimum vertex cover in an undirected graph. It is based on the greedy algorithm and works by repeatedly selecting the vertex with the largest degree and removing all of its neighbors from the graph until no more vertices can be selected. The time complexity of the Approximate Minimum Vertex Cover is $O(V+E)$, where $V$ is the number of vertices in the graph and $E$ is the number of edges.

**Inputs:** Given an Undirected Graph $G=(V,E)$ with vertex set $V$ and edge set $E$, and a matching $M$.  
**Output:** minimum vertex cover $M'$ from $u$ to $v$. 

---

The Exact Subset-Sum algorithm is a brute-force algorithm for finding a subset of a given set of integers that sums to a target value. It works by generating all possible subsets of the given set and checking if any of these subsets sum to the target value. The time complexity of the Exact Subset-Sum is $O(\exp)$, which is due to the fact that the algorithm may need to generate and check all possible subsets of the given set.

**Inputs:** Given a Set of Integers $S$ and a Target Value $T$.  
**Output:** subset $S'$ from $u$ to $v$. 

---

The Approximate Subset-Sum algorithm is an approximation algorithm for finding an approximate subset of a given set of integers that sums to a target value. It is based on the greedy algorithm and works by repeatedly selecting the integer with the largest remainder and subtracting it from the target value until the remainder is less than the largest remaining integer or the remainder is negative. The time complexity of the Approximate Subset-Sum is $O(\text{poly})$, which is due to the fact that the algorithm may need to select and subtract multiple integers from the target value.

**Inputs:** Given a Set of Integers $S$ and a Target Value $T$.  
**Output:** subset $S'$ from $u$ to $v$. 

---

**[^1]:** Greedy algorithm.

---

**[^2]:** Dynamic Programming.

---

**[^3]:** Divide-and-Conquer.
Edmonds–Karp algorithm
(Implementation of FFM) \[O(V \cdot E^2)\]

An implementation of the Ford–Fulkerson method.

```
for all edges (u,v):
    f[u, v] := 0
while there is a path p from s to t in Gf, such that cf(u,v) > 0 for all edges (u,v) in p:
    cf(p) := min([cf(u, v) for each edge (u, v) in p])
    for each edge (u,v) in p:
        f(u, v) += cf(p) # Send flow along the path
        f(v, u) -= cf(p) # The flow might be "returned" later
```

Ford–Fulkerson Method (Maximum Bipartite Matching) \[O(E \mid f^* \mid)\]

Input: a bipartite graph \( G = (V, E) \) with \( V = L \cup R \).

Output: Size of maximum matching.

1. Build the flow network:
   1. For every \((u,v) \in E\), assign capacity \( c(u, v) = 1 \).
   2. Add source node \( s \) and sink node \( t \).
   3. For every \( u \in L \), add edge \((s, u)\) with capacity \( c(s, u) = 1 \).
   4. For every \( v \in R \), add edge \((v, t)\) with capacity \( c(v, t) = 1 \).

2. Apply \( \text{Ford–Fulkerson} \). Return the output value.

```
Ford–Fulkerson (Approx. Minimum Bipartite Vertex Cover) \[O(E \mid f^* \mid)\]
```

Input: an undirected graph \( G = (V, E) \).
Output: 2-approximation to the minimum size of vertex cover in G.

Just use 😊 Ford–Fulkerson Method (Maximum Bipartite Matching) \( O(E |f*|) \).

This is because the Maximum Bipartite Matching is a 2-approximation to the Min. Bipartite Vertex Cover.

😊 Approximate Minimum Vertex Cover

\( O(V+E) \)

Input: an undirected graph \( G = (V, E) \).

Output: 2-approximation to the minimum size of vertex cover in G.

```python
C = []
E' = G.E
while E' is not []:
    Randomly select edge (u, v) from E'
    C.append((u, v))
    remove every edge connecting u or v in E'
return C
```

😊 Exact Subset-Sum

\( O(\exp) \)

```python
def exact_subset_sum (S, t):
    n = len(S)
    L[0] = {0}
    for i in range(n):
        L[i] = sorted( unique( L[i-1] + L[i-1] + {S[i]} ) )
        L[i] = filter(lambda x: x<=t, l[i])
    return max([sum(l) for l in L])
```

😊 Approx. Subset-Sum

\( O(\text{poly}) \)

```python
def approx_subset_sum (S, t, e):
    def trim(l, d):
        '''removes elements within `d` of its predecessor.'''
        m = len(l)
        l` = {l[0]}
        last = l[0]
        for i in range(2, m):
```
if l[i] > list*(1+d): # because l is sorted
    l`.append(l[i])
last = l[i]
return l` # a trimmed, sorted list

n = len(S)
L[0] = {0}
for i in range(n):
    L[i] = sorted( unique( L[i-1] + L[i-1] + {S[i]} ) )
    L[i] = trim(L[i], e/2/n)
    L[i] = filter(lambda x: x<t, l[i])
return max([sum(l) for l in L])

def BFS(G, s):
    # Mark all the vertices as not visited
    visited = [False]*(len(G.V))
    # Create a queue for BFS, enqueue s:
    queue = [s]
    # Mark the source node as visited:
    visited[s] = True
    while queue:
        # Dequeue a vertex from queue and print it
        s = queue.pop()
        print s,
        # Get all adjacent vertices of the dequeued
        # vertex s. If a adjacent has not been visited,
        # then mark it visited and enqueue it
        for i in G.neighbors[s] if not visited[i]:
            queue.append(i)

def DFSUtil(G,v,visited):
    '''A function used by DFS'''
    visited[v] = True # Mark the current node as visited
    print v, # print the current node
    # Recur for all the vertices adjacent to this vertex
Topological Sort (DAG Only; Allows $w<0$; Single-Source) \( O(V+E) \)

**Topological Sort: Example**

```
for i in G.neighbors[v]:
    if visited[i] == False:
        G.DFSUtil(i, visited)

def DFS(G,v):
    '''The function to do DFS traversal. It uses recursive DFSUtil()'''
    visited = [False]*(len(G.V)) # Mark all the vertices as unvisited
    for i in range(V):
        if visited[i] == False:
            G.DFSUtil(v,visited) # Call the recursive helper function to print
    DFS traversal
```

Original graph | DFS forest

1. Run DFS(G), computing finish time for each vertex;
2. As each vertex is finished, insert it onto the front of a list;

**Final order:** \( \langle b, f, g, a, c, d, e, h \rangle \).
3. Output the list.

def topologicalSortUtil(G, v, visited, stack):
    '''A recursive function used by topologicalSort'''
    visited[v] = True  # Mark the current node as visited.
    # Recur for all the vertices adjacent to this vertex
    for i in G.neighbors[v]:
        if not visited[i]:
            G.topologicalSortUtil(i, visited, stack)
    stack.insert(0,v)  # Push current vertex to stack which stores result

def topologicalSort(G):
    '''The function to do Topological Sort.
    It uses recursive topologicalSortUtil()'''
    visited = [False]*G.V  # Mark all the vertices as not visited
    stack = []  # Call the recursive helper function to store Topological
    # Sort starting from all vertices one by one
    for i in range(G.V):
        if not visited[i]:
            G.topologicalSortUtil(i, visited, stack)
    print stack  # Print contents of stack

[ShortestPath] Dijkstra (Allows Cycles; No weight<0; Single-Source)
O(V^2)\rightarrow O(V \cdot \log V)

def initialize_single_source(graph, source):
    for each vertex v in graph:
        v.d = \infty
        v.\pi = None
    s.d = 0

def relax(u, v, weight_of_edge_uv):
    if v.d > u.d + weight_of_edge_uv:
        v.d = u.d + weight_of_edge_uv
        v.\pi = u

def extract_min(set_of_vertices):
    a = vertex in set_of_vertices whose distance d is min
    set_of_vertices.pop(a)
    return a
def dijkstra(G, w, s):
    initialize_single_source(G, s)
    S = []
    Q = G.Vertices
    while Q is not empty:
        u = extract_min(Q)
        S.append(u)
        for each vertex v in G.adj[u]:
            relax(u, v, w[u, v])

procedure BellmanFord(list vertices, list edges, vertex source)
    // 该实现读入边和节点的列表，并向两个数组（distance和predecessor）中写入最短路径信息

    // 步骤1：初始化图
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := infinity
        predecessor[v] := null

    // 步骤2：重复对每一条边进行松弛操作
    for i from 1 to size(vertices)-1: // repeat n-1 times -- iteration ID not important:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]: // if taking this edge yields shorter dist.:
                distance[v] := distance[u] + w // relax dist. to v via this route:
                predecessor[v] := u // record the current best solution.

    // 步骤3：检查负权环
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            raise "图包含了负权环"
Bellman-Ford (Negative Cycle Detection) \( O(V \cdot E) \)

1. Color every node white.
2. For each node \( u \) (in an arbitrary order),
   1. set \( v := u \);
   2. while \( v \) is white and has a predecessor,
      1. recolor \( v \) gray;
      2. set \( v := \text{predecessor}[v] \).
3. If \( v \) is gray, we found a cycle:
   loop through again to read it off.
Else, none of the gray nodes are involved in a cycle;
loop through again to recolor them black.

Source: algorithms - Finding the path of a negative weight cycle using Bellman-Ford - Computer Science Stack Exchange

Shortest Path Matrix Multiplication (All-Pair) \( \Theta(n^3 \lg n) \)

```python
def extend_shortest_paths(L, W):
    n = L.rows
    M = []
    for i in range(n):
        M.append([])
        for j in range(n):
            M[i].append(\infty)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                M[i][j] = \min(M[i][j], M[i][k] + W[k][j])
                # If by taking route k i can reach j faster, then take this path.
        return M

def faster_all_pairs_shortest_paths(W):
    n = W.rows  # get size of square matrix W
    L = \{1: W\}
    m = 1
    while m < n-1:
        L[2*m] = extend_shortest_paths(L[m], L[m])
```

[ShortestPath] **Floyd-Warshall (All-Pair)** \(\Theta(n^3)\)

let dist be a \(|V| \times |V|\) array of minimum distances initialized to \(\infty\)
for each vertex \(v\):
  dist\([v][v] \leftarrow 0\)
for each edge \((u,v)\):
  dist\([u][v] \leftarrow w(u,v)\) // the weight of the edge \((u,v)\)
for \(k\) from 1 to \(|V|\):
  for \(i\) from 1 to \(|V|\):
    for \(j\) from 1 to \(|V|\):
      if dist\([i][j] > dist[i][k] + dist[k][j]\) :
        dist\([i][j] \leftarrow dist[i][k] + dist[k][j]\)

[MST] **Kruskal's Algorithm (take shortest; for undirected)** \(O(E \cdot \log V)\)

**Kruskal's Algorithm**

1. Given a network...........
2. Choose the shortest edge (if there is more than one, choose any of the shortest)........
3. Choose the next shortest edge and add it........
4. Choose the next shortest edge which wouldn’t create a cycle and add it.
5. Choose the next shortest edge which wouldn’t create a cycle and add it.
6. Repeat until you have a minimal spanning tree.
A = {}
for v in G.V:
    v = set(v)
for (u, v) in G.E increasingly ordered by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v): # if adding this edge won't incur cycles:
        A.append( (u, v) )
        UNION(u, v)
return A

[MST] Prim's (take nearest; for undirected & connected)
O(E \cdot \log V) \rightarrow O(E + V \cdot \log V)

Prim’s Algorithm

1. Given a network............
2. Choose a vertex
3. Choose the shortest edge from this vertex.
4. Choose the nearest vertex not yet in the solution.
5. Choose the next nearest vertex not yet in the solution, when there is a choice choose either.
6. Repeat until you have a minimal spanning tree.

T = {}
U = { random.choice(V) }
while U ≠ V: # Before U includes all vertices in G, repeat:
    Find the "light edge" (u, v) s.t. u ∈ U and v ∈ V - U # Find the nearest vertex to (and thus not yet in) U:
    T.append( (u, v) )
    U.append( v )

Recursive Activity Selection
s = { array of starting times }  
f = { array of finishing times }  # we assume that activities are ordered by monotonically increasing finish time  
n = number of activities  
def recursively_select_activity(k):
    m = k+1  # Start search from the next planned activity.
    while m<n and s[m]<f[k]:  # As long as m is not the last planned activity and that m wants to start before k ends:
        m += 1  # Go on searching.
    if m<n:  # if finally found such one:
        return {a_m} cup recursively_select_activity(m)
    else:  # if not:
        return {}
recursively_select_activity(0)

Iterative Activity Selection

# Input:  
s = { array of starting times }  
f = { array of finishing times }  # we assume that activities are ordered by monotonically increasing finish time  
n = number of activities  
# Init:  
A = {a_1}  
k = 1  
# Main loop:  
for m = 2 to n:  
    if s[m]>=f[k]:  
        A.append(a_m)  
        k = m  
return A

0-1 Knapsack Problem

def knapSack(W, wt, val, n):
    ...  
    # A Dynamic Programming based Python Program for 0-1 Knapsack problem  
    # Returns the maximum value that can be put in a knapsack of capacity W  
    W = total weight carry-able  
    wt = { array of items' weights }

val = { array of items' values }  
n = total number of items

K = {{ (n+1)-by-(W+1) matrix of 0 }}
# Build table K[][] in bottom up manner
for i in range(n+1):  # When taking the first i items:
    for w in range(W+1):  # When there is w capacity left:
        if i==0 or w==0:  # if it's "nothing" or that this slot is empty:
            K[i][w] = 0  # Max value we can get from this situation is 0.
        elif wt[i-1] <= w:  # else, if the remaining capacity can
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])  # set
            the value at this slot to be the max one of the two options: (1) add this
            item, shrinking the remaining capacity by its weight; (2) pass this item,
            leaving the remaining capacity unoccupied.
        else:  # there's no space to accomodate this item:
            K[i][w] = K[i-1][w]  # we can only pass this item.

return K[n][W]

**Fractional Knapsack Problem**

Sort list of items by value-to-weight ratio.
While knapsack is not full and list of items is not exhausted:
    A = first item in the list.
    Put as much A as possible into the knapsack.

**Huffman (Optimal Prefix Coding)**

O(n·1gn)→O(n1g1gn)
Maximum-Weight Indep. Subset of A Matroid

Given a matroid $M = \{S, I\}$ and its associated weight vector $w$.

A = []
Sort M.S by monotonically decreasing weight $w$.
for x in M.S:
    if A+{x} is still independent: # i.e. A+{x} is in M.I:
        A.append(x)
return A

Linear Select (Select the k-th-big item with linear time even in worst case) $O(n)$

def select(a, i):
    if len(a)<5: return sorted(a)[i]
    else:
        a_rect = reshape_to_5(a) # 5 items per group (row).
        m = [ median(row) for row in a_rect ]
        if len(m) % 2 == 0: # if even items
            median_to_get = (len(m)-1)/2
        else: # odd items:
            median_to_get = len(m)/2
        x = select(m, i = median_to_get) # use SELECT to find the median-of-mediants.
        # partition:
Quick Select (T(n) = T(n/2) + n) $O(n)$

```python
def select(a, k):
    n = len(a)
    if n==1: return a[0]
    pivot = random.choice(a)
    l = []
    e = []
    h = []
    for this in a:
        if this<pivot:  l.append(this)
        elif this>pivot: h.append(this)
        else: e.append(this)
    if len(l)+len(e)<=k:
        k -= len(l) + len(e)
        a = h # find in the higher group
    elif len(l)<=k:
        k -= len(l)
        a = e # find in the "equal" group
    else: # k<len(l)
        a = l # find in the lower group
    if len(h)==0 and len(l)==0:
        return pivot # A-hah! The pivot happens to be just the target value!
    else:
        return select(a, k)
```

```python
l = a[ np.where( a < x ) ] # lower half
h = a[ np.where( a > x ) ] # higher half
# locate desired value:
k=len(l)
if  i==k:
    return x
elif i<k:
    return select(l, i)
elif i>k:
    return select(h, i-k-1)
```

result = select(a,i)
assert result==sorted(a)[i]
Quick Sort \( T(n) = 2T(n/2) + n \) \( \Theta(n \log n) \)

```python
def sort(a):
    n = len(a)
    if n<=1: return a
    # else:
    pivot_id = np.random.choice(n)
    pivot = a[pivot_id]
    # construct a result array:
    l = []
    h = []
    for i in range(n):
        this = a[i]
        if i!=pivot_id: # Be aware that we use the ID to allow for non-pivot
            items with the same value as the pivot.
            if this<pivot:
                l.append(this)
            else:
                h.append(this)
    return sort(l)+[pivot]+sort(h)
```

Interleaves Two Halves of An Array
\( T(n) = 2T(n/2) + n/4 \) \( \Theta(n \log n) \)

```python
def interleave(start, end):
    n = (end-start)/2
    mid = n/2
    cycle = n-mid
    for i in range(start+mid,start+n):
        swap(a[i], a[i+cycle])
    if n > 2:
        interleave(start, start+n)
        interleave(start+n, start+2*n)
```