Dijkstra's Algorithm: O((m+n)log(n)) with queues.
(Priority Queue steps in yellow)

Dijkstra(Graph G, vertex S)
  For every vertex, set dist(V) = infinity and prev(V) = nil.
  dist s = 0 (no distance to starting node)
  make a new empty priority queue Q with costs as keys Insert S with cost 0
  For every other vertex, insert Q with cost infinity)
  Until all vertices are visited: (for i = 1 to i = V)
    set vertex u = to the front of the queue, remove front of queue.
    for all edges (u, v)
      if dist(v) > dist(u) + length(u,v)
        dist(v) = dist(u) +length(u,v);
        prev(v) = u;
        decreasekey(H, v);

- **Summary:** Initializes the distance to each node as infinity, then compares distance from start vertex to each adjacent vertex to infinity. Goes to least-cost vertex and repeats process, comparing each vertex's cost from the new vertex 'u' to its previous cost. This continues until all vertices have been visited.
- Used to find shortest path to all vertices .
- Dijkstra's algorithm will fail if negative edges exist unless those edges are solely coming out of the source node.

Bellman-Ford Algorithm: Shortest path w/ negative weights. O(V*E)

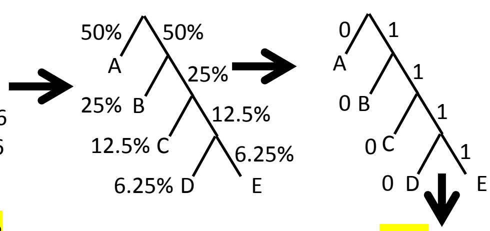
Bellman_Algo(Graph G, length l, vertex S)
  For every vertex, set dist(V) = infinity and prev(V) = nil.
  dist s = 0 (no distance to starting node)
  Repeat the following v-1 times (for i = 1 to i = V)
    for each edge (u, v)
      if dist(v) > dist(u) + cost(u,v)
        dist(v) = dist(u) + cost(u,v);
        prev(v) = u;

- **Summary:** The difference between this and Dijkstra's Algorithm is that Bellman-Ford merely iterates v-1 times, not v times.
- Works for negative
  - Negative cycles will be detected if it's updated past v-1 times.

Huffman Encoding Example:

Convert the following alphabet into Huffman encoding:

A = 1/2
B = 1/4
C = 1/8
D = 1/16
E = 1/16



A = 50%
B = 25%
C = 12.5%
D = 6.25%
E = 6.25%

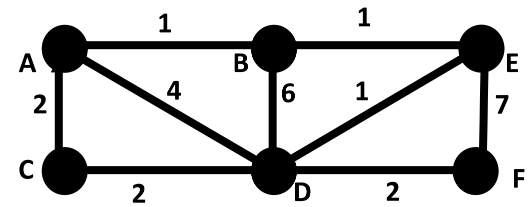
A = 0
B = 10
C = 110
D = 1110
E = 1111

Now, for a file of 10000 chars (assuming all chars are one of these five), the length of the file in bits will be computed as follows:

Number of As = 50% of 10000 = 5000	Bitlength A = 1 bit * 5000 chars = 5000
Number of Bs = 25% of 10000 = 2500	Bitlength B = 2 bits * 2500 chars = 5000
Number of Cs = 12.5% of 10000 = 1250	Bitlength C = 3 bits * 1250 chars = 3750
Number of Ds = 6.25% of 10000 = 625	Bitlength D = 4 bits * 625 chars = 2500
Number of Es = 6.25% of 10000 = 625	Bitlength E = 4 bits * 625 chars = 2500

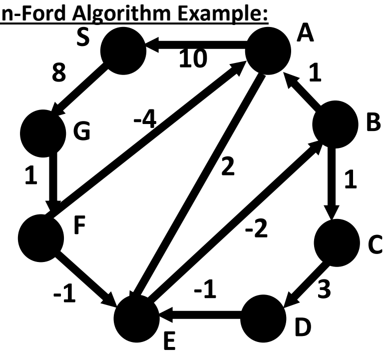
Total Bitlength = Bitlength A + Bitlength B + Bitlength C + Bitlength D + Bitlength E
Total Bitlength = 5000 + 5000 + 3750 + 2500 + 2500
Total Bitlength = 18750

Dijkstra's Algorithm Example:



Node	Iteration					
	0	1	2	3	4	5
A	**0**	0	0	0	0	0
B	∞	**1**	1	1	1	1
C	∞	2	**2**	2	2	2
D	∞	4	4	4	**3**	3
E	∞	∞	2	**2**	2	2
F	∞	∞	∞	∞	9	**5**

Bellman-Ford Algorithm Example:



Node	Iteration						
	0	1	2	3	4	5	7
S	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5
B	∞	∞	∞	10	6	5	5
C	∞	∞	∞	∞	11	7	6
D	∞	∞	∞	∞	∞	14	10
E	∞	∞	12	8	7	7	7
F	∞	∞	9	9	9	9	9
G	∞	8	8	8	8	8	8

Huffman Encoding:
Input array of integers

Huffman (Array of frequencies a[1… n])
  (make a new empty priority queue, Q and insert all frequencies
  for 1 over the number of frequencies to 1 less than twice the number of frequencies **(for (i=1 to n)) . Order queue by frequency.**
    insert (i) into the queue.
  for (k = n+1 to k = 2n-1)
    i = deletemin(Q), j = deletemin(Q)
    create node numbered k with children I, j;
    f[k] = f[i] + f[j] (This creates a node equal to the sum of its' children's frequencies)
    insert (Q, k);

- **Summary:** Creates a binary tree to encode characters of various frequencies into binary form. Can then be used to determine the length of a file via characters with various frequencies.

Dynamic Programming:
Splitting a problem into smaller, typically overlapping/dependent sub-problems and tackling one by one, smallest first.

1. Find efficient recursion for problem.
2. Eliminate Recursion and find iterative algorithm to compute problem.
3. Estimate number of subproblems, evaluate running time.
4. Optimize.

- **Sequence:** ordered list a1, a2, … an. Length is number of elements in list.
  - Sequence is increasing if 1 < i1 < … < ik < n.
    - (Decreasing is opposite)
    - (Non-Increasing means each value is less than/equal to previous value. Non-decreasing means each is greater than/equal to prev)
  - A subsequence is a sequence enclosed by another sequence.

Example:

- **Sequence:** 6, 3, 5, 2, 7, 8, 1
- **Increasing Subsequences**: 6, 7, 8 and 3, 5, 7, 8 and 2, 7
- **Longest Increasing Subsequence:** 3, 5, 7, 8

Longest Subsequence Algorithm:
Find longest subsequence: Inputs vector of integers.

```
Longest_ sub(A)
    Create another vector, B, of equal length to A.  Set all indices of B to 1.
    for each index in B,
        for (int I = 0; I < j; i++) {
            #for each index I, while I is less than j, update according to
            #desired type of substring (if increasing, check if A[i] > A[j], etc)
            #example for nondecreasing
            if (A[i] < A[j])
                if (B[j] < B[i] + 1)
                    B[j] = B[i] + 1;

    Sort B, return B;
```

Knapsack Problem:
Given a weight and a value for each item, and a maximum weight for entire knapsack, find the maximum value for that weight.

```
Knapsack( list of items with their weight (w) and value (v) , maximum weight X)
    v(0) = 0;
    for (w=1 to w) {
        for (i = 1 to n) {
            if (weight of item I <= w) {
              max value equals greater of the  previous max and V( w-i(w)) +
i(v) )
```

Coin Changing Problem:
To separate given value into coins, use a greedy algorithm, changing largest coins possible at each stage. Input = value to be changed. Output = number of each coin.

```
Coin_Change(int value)
initialize number of quarters, dimes, nickles, pennies, all at 0;
while the value is greater than or equal to 25
    increment the number of quarters (as you're adding one quarter to
change)
    value = value – 25
while the value is greater than or equal to 10
    increment the number of dimes (as you're adding one dime to change)
    value = value – 10
while the value is greater than or equal to 5
    increment the number of nickels (as you're adding one nickel to change)
    value = value – 5;
while the value is greater than or equal to 1
    increment the number of pennies (as you're adding one penny to change)
```

Linear Programming:

Process of representing a problem through linear functions.

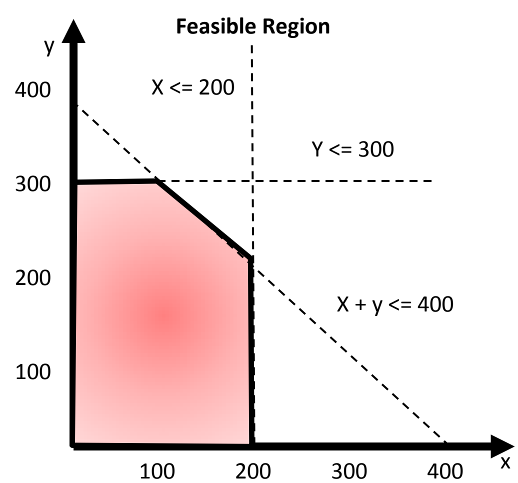
1. Define objective function: This is usually an algebraic formula.
2. Define constraints, as determined by problem.
3. Graph the program's 'feasibility region', the area in which all variables satisfy their constraints.
   a. If this area doesn't exist, program is *infeasible* (I.E. with x<= 1, x > 3)
   b. If constraints are so loose, max value is infinite, region is *unbounded*.
4. The optimal feasible point within this region is usually the max value of the objective function within the region.

Simplex Algorithm for finding optimal feasible point: O(mn)

```
Let v = any vertex in feasible region
while there's a nearby vertex v' with better objective value
    v = v'
```

EXAMPLE: If a workshop creates x boxes of product daily, selling at $1 each, and y boxes of another product daily, selling at $6 each, how much should it make to maximize profit? Daily demand for x <= 200, and for y <= 300. Manufacturing limits only allow production of 400 total as well. What are optimal production levels?

Objective function; max (x + 6y)
Constraints x <= 200
y <= 300
x + y <= 400
x >0 and y > 0.



In the above graph, the function optimizes with y = 300, x = 100, creating a maximum, optimum, gain of 6(300) + 100, which = $1900

Depth First Search: O(m + n)
Input: graph G, start vertex S; Creates tree T rooted at u;
(Directed in yellow)

DFS(graph G, vertex S)
    Mark all vertices as unvisited. Initialize tree T to be empty.
    Have time variable set to 0, this will be incremented with each visit.
    As long as one vertex is still unvisited
        DFS (S);
    Return T;

DFS(vertex U)
    Visit U;
    pre(u) = ++time;
    For each vertex V adjacent to U if edge FROM v TO u exists
        if V hasn't been visited;
            Add edge (u,v) to T;
            DFS(V);
    post(u) = ++time;

- **Summary:** The DFS creates an empty tree, here labeled T. Each vertex, starting with the start vertex S, is visited, and given a prenumber as it's visited. Adjacent vertices will be recursively visited, and then U is given a postnumber is it is visited for the final time.
- Good for exploring graph structure.
- If DFS reveals a back edge, G has a cycle and is not a DAG.
- If G is a DAG and post(v) > post(u), edge (u, v) is not in G.
- Highest post visit is always a source vertex, for DAG. If not a DAG, will be a source strongly connected component (perhaps in a cycle).
    - Same goes for lowest post-order and sinks.

DFS Search Result Properties:

- Vertex v is in T if and only if v is reachable from u;
- To do a topological sort, output nodes in decreasing post-visit order

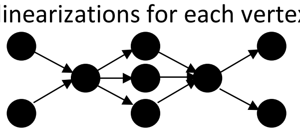
EDGE TYPES:

- Tree edge: Belong to T as determined by DFS;
- Forward edge: Non-tree edge (x, y): pre(x) < pre(y) < post(y) < post(x)
- Back edge: Non-tree edge (x,y): pre(y) < pre(x) < post(x) < post(y)
- Cross edge: Non-tree edge (x,y): [pre(x), post(x)] and [pre(y), post(y)] are disjoint.

Directed Acyclic Graph: Directed graph G has no cycle, so it's a DAG.
Source: Node with no incoming edge.
Sink: Node with no outgoing edge.
*Every DAG has at least one source and at least one sink.
*In a DAG, all edges lead to a vertex with lower post number.

Topological Sorting: Ordering of a DAG where all edges are from left to right; linearization of DAG.

- Each DAG will have n! linearizations for each vertex with n branches.
- For example:



Total linearizations = 2! * 1 * 3! * 1 * 2! = 24

Strongly Connected Components: If vertices U and V are strongly connected, there are paths both from U to V and from V to U. A strongly connected component of a graph is a subgraph consisting of only strongly connected vertices.

Breadth First Search: O(m + n)
Input: graph G, start vertex S. Creates tree T.
(cost added in yellow)

BFS(vertex S)
    Mark all vertices as unvisited (have a visited bool set to false for all vertices)
    For each vertex, set cost(v) = ∞
    Initialize search tree T to be empty;
    Mark S as visited, set cost(s) = 0;
    Create empty queue Q, add s to end of Q;
    While Q isn't empty;
        vertex u = whatever vertex was at the front of the queue;
        remove u from the front of the queue;
        for each vertex adjacent to u
            if that vertex isn't visited,
            add edge (u,v) to T;
                Mark v as visited and add v to the end of the queue;
                Set cost(v) = cost(u) + 1;

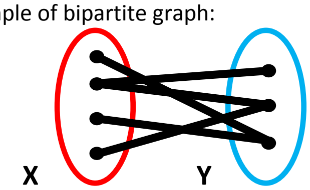
- Processes vertices in order of their shortest distance from S.
    - Will find the shortest path to each vertex; useful for minimum spanning tree.
- Good for exploring distances.

BFS Search Result Properties:

- Search tree contains exactly the set of vertices in the connected component of S;
    - If directed, contains set of vertices REACHABLE from S;
- If cost(u) < cost(v) then u is visited before v.
- For every vertex u,cost(u) is the length of the shortest path from s to u.
- For Undirected:
    - If u, v are in connected component of S, and there's an edge between them, then if that edge isn't in the search tree, the distance between u and v is between 0 and 1.
    - This isn't always true for directed graphs.
- Treats all edges as equal length.

Bipartite Graph: A graph that can be partitioned into two subgraphs X and Y such that all edges are between them.

- ALL trees are bipartite.
- Odd length cycles are not bipartite.
- Any subgraph of a bipartite graph is bipartite.
- Example of bipartite graph:



Spanning Tree/Spanning Graphs:

- A Spanning Tree is a tree that touches every vertex of a graph. A spanning graph follows the same concept but can have a cycle.
    - Minimum spanning tree is smallest edge connected to each vertex.

Kruskal's Algorithm: used to create minimum spanning tree.

1) All vertices exist in tree T, disconnected.
2) Sort list of edges by increasing weight
3) Add edges (u, v) starting with least weight until all V are connected. Don't add edges if 'v' is already in T.

- To find maximum spanning tree, negate all edge weights first, so that the minimum cost edge will have maximum cost, and vice versa. THEN do Kruskal's.