Data Structures

Graph

Description: has V verticies (nodes) and E edges

path	set of edges between two verticies
connected graph	all pairs of nodes have a path between them
directed graph	an edge from node u to node v does not imply an edge from v to u
directed acyclic graph (DAG)	directed graph with no cycles
cycle	path from any node to itself
tree	connected graph with no cycles

Heap (minimum heap)

Description: a binary tree structure. value of each node is larger than the parent's value

To build a heap, use these functions

$heap\_up(i)$	$O(\log n)$	swaps node with parent repeatedly until greater than parent
$heap\_down(i)$	$O(\log n)$	swaps node with child with min value until greater than both children
$make\_heap()$	$O(n)$	from i from $\frac{n}{2}$ to 1, call $heap\_up(i)$

Once a heap is built, you can use these functions

$find\_min()$	$O(1)$	returns the value at the root
$insert(v)$	$O(\log n)$	inserts value after last leaf. calls $up\_heap()$ on that node
$delete(i)$	$O(\log n)$	replace value with last leaf value. decrease heap size. call $heap\_down(i)$
$reduce\_key(i)$	$O(\log n)$	reduce value. call $heap\_up(i)$

To do **heap sort**, first call $make\_heap()$ then $find\_min()$ and $delete(1)$ (delete the minimum from heap) until there are no nodes left in the heap

Queue

Description: the first element inserted will be the first to come out. Use a queue to do breadth first search: $insert()$ neighbors to eventually visit and $remove()$ from the queue to process nodes. Stop when queue is empty

$insert()$	inserted node is added to the *end of the line*
$remove()$	remove node from the *front of the line*

Stack

Description: the first element inserted will be the last to be taken out. Use a stack to do depth search: *push*() neighbors to eventually visit and *pop*() from the stack to process nodes. Stop when stack is empty. Depth first search can also be done with recursion.

push()	inserted node is added to the *top of the stack*
pop()	remove node from the *top of the stack*

Union-Find

Description: Used to group nodes in a graph (for Kruskal's minimum spanning tree algorithm). Contains three data structures. The *items* data structure can be implemented as a tree or an array of lists.

sizes	array. index is the set number and value is size of set (ex. Set 2 has 5 nodes)
sets array	index is the node and value is which set it belongs to (ex. vertex 7 belongs to set 2)
items	**tree** implementation: distinct trees for each set. root of each tree defines the set number
	array implementation: array of lists. distinct lists for each set. array index is set number

To manage a **tree** Union-Find data structure, use these functions

$make\_union\_find()$	$O(n)$	create *sizes*, *sets array*, and *items* data structures
$find(i)$	$O(\log n)$	start at node i. go up tree until root is reached. return root index
$union(x, y)$	$O(1)$	x and y are sets. point root of smaller set at root of larger set

To manage a **array/list** Union-Find data structure, use these functions

$make\_union\_find()$	$O(n)$	same as tree implementation
$find(i)$	$O(1)$	start at node i. go up tree until root is reached. return root index
$union(x, y)$	$O(n)$	x and y are sets. point head of list to second list. point tail of second list to head of first list

The runtime for $union(x, y)$ using array implementation looks long, but calling $union()$ repeatedly until there is only one set is actually bounded by $O(n \cdot \log n)$

Algorithms

Primm's Minimum Spanning Tree | $O(|E| \cdot \log |V|)$

Use: Creating a minimum spanning tree (MST)

Method: Start with any node. Continuously add frontier edges with minimum weight.

Correctness: Use cut property of a minimum spanning tree. One subgraph is growing tree. The other contains frontier nodes. Lowest cost edge crossing cut is added to minimum spanning tree.

```
for v in V do           # initialize distance from each vertex to the growing tree
  dist_to_T(v) = inf
end

u = s                   # some start node s
while u not nil
  dist_to_T[u] = -inf   # u is now in the tree
  for v neighbors of u
    if dist(u, v) < dist_to_T[v] # if we found a shorter distance
      dist_to_T[v] = dist(u, v)  #   update the distance
      parent[v] = u              #   and the parent node
    end
  end
  u = closest_vertex_to_tree(dist_to_T) # the next u is the closest to the tree
end
```

Kruskal's Minimum Spanning Tree | $O(|E| \cdot \log |V|)$

Use: Creating a minimum spanning tree (MST)

Method: Add edges from minimum to max weight. Do not add edges that create cycles. Uses $Union - Find$ data structure to grow small minimum spanning trees into one minimum spanning tree.

Correctness: Adds edges with weights in increasing order so always using shortest edges. Never adds cycles since algorithm will never union the same growing tree with itself

```
sort all (u, v) edges   # runs in O(n * log n) time

UF.make_union_find()    # runs in O(n) time

for each edge (u, v) in sorted order

  u's_set = UF.find(u)  # find which set u belongs to
  v's_set = UF.find(v)  #            set v belongs to

  if u's_set not v's_set
    UF.union(u's_set, v's_set)  # union sets if not in same set
  end
end
```

To **cluster** nodes, run Kruskal's algorithm and stop after adding the k^{th} edge to get $k-1$ clusters.

Depth First Search | $O(|E| + |V|)$

Use: Traversing a graph so neighbors of neighbors are visited before other neighbors of the starting node.

Method: Use a *stack* data structure. Start at a node. Add any unvisited neighbors to the stack. Process the current node. Remove from the stack and repeat the process.

```
mark each node u as not visited     # each node has not been visited yet

S.push(s)                           # stack only contains the starting node

while S not empty

  u = S.pop()                       # remove a node to process

  if not u.visited
    u.visited = true
    for each neighbor n of u        # add any not visited neighbors to the queue
      S.push(n)
    end
  end

  # process your node. if you do something to each node, do it here

end
```

Depth first search also has a **recursive** implementation.

```
function dfs(G, u):

  # pre-process here. if you need to do anything before you visit all child nodes, do it here

  for each neighbor n of u   # visit each neighbor of u
    if not n.visited
      n.visited = true       # perform depth first search on all unvisited neighbors of u
      dfs(G, n)
    end
  end

  # post-process here. if you need to do anything after you visit all child nodes, do it here

end
```

Breadth First Search | $O(|E| + |V|)$

Use: Traversing a graph so all neighbors of a node are visited before the neighbors of other visited nodes are visited

Method: Use a *queue* data structure. Start at a node. Add all neighbors of current node to the *queue*. Process the current node. Remove node from the *front* of the *queue* and repeat this process.

```
mark each node u as not seen          # each node has not been seen yet

s.seen = true
Q.insert(s)                           # queue only contains the starting node

while Q not empty

  u = Q.remove()                      # remove a node to process

  for each neighbor n of u
    if not n.seen                     # add any unseen neighbors to the queue
      n.seen = true
      Q.insert(n)
    end
  end

  # process your node. if you do something to each node, do it here

end
```

Visited nodes in breadth first search can be split into **layer**. The first layer (L_1) only has start node s. Layer L_2 has any neighbor of s. Layer L_i has nodes with edges to layer L_{i-1} but no edges to any layers before that.

Topological Sort | $O(|E| + |V|)$

Use: Only used on a directed acyclic graph (DAG). Sort nodes in order so each node will have no edge that points to a previously visited node.

Method: Find a node with no incoming edges. Mark this node as the i^{th} node to visit in topologically sorted order. Delete this node from the graph. Repeat this until no nodes are left in the graph.

```
for i to |V|
  u = node with no incoming edges
  topological_order(u) = i
  G.delete(u)
end
```

Even though nodes are removed from the graph, we can always do topological sort on a copy of the graph so we do not damage the original graph.

Dijkstra's Shortest Path | $O(|E| \cdot \log |V|)$

Use: Finding the shortest path from one node to all other nodes

Method: All edges must be positive. Start at a node s with distance to s as 0. Add all neighbors to a heap of unvisited nodes. The root of the Heap is the node with the smallest edge weight to any visited nodes. If neighbors are in the heap and we find a shorter path through the current node to that neighbor, change that neighbor's shortest path distance in the heap.

Correctness: The path through visited node u to "to be added" node v will be shorter than any other path that goes through unvisited nodes since we visit nodes in order of their edge weight to any previously visited node.

```
for u in V do         # initialize distance from each node to the start node
  dist_to_s(u) = inf
end

H = make_heap()       # add start node to heap
H.insert(s)

while H not empty

  u = H.remove(1)
  for each neighbor n of u

    dist_to_s_thru_u = dist_to_s(u) + dist(u, n)

    if dist_to_s_thru_u < dist_to_s(n)   # if the distance to s through u is smaller
      dist_to_s(n) = dist_to_s_thru_u    #   change n's distance to s

      if n not in H                      # insert n into heap of nodes to visit
        H.insert(n)                      # if already in heap, just reduce n's dist
      else
        H.reduce_key(n, dist_to_s_thru_u)
      end

      parent[n] = u                      # remember that the shortest path to s is thru u
    end
  end
end
```

A Star Shortest Path | runtime dependent on heuristic

Use: Finding the shortest path from one node to another. Uses heuristic to estimate distance from current node to destination.

Method: Same as Dijksta's except the keys for the heap of not visited nodes is the edge weight plus a heuristic distance. A good heuristic is any function that gives a distance that is less than or equal to the actual distance to the destination.

Bellman-Ford's Shortest Path | $O(|E| \cdot |V|)$

Use: Finding the shortest path from one node to all other nodes

Method: Edges can have any weight (can be negative). For each edge (u, v) in the graph, see if the distance from s to u can be updated by going through v. Do this each edge thing, $|V| - 1$ times to ensure on the i^{th} iteration, any node at most i edges away from s will have the shortest path.

```
for u in V do         # initialize distance from each node to the start node
  dist_to_s[u] = inf
end
```

```
for i from 1 to |V|-1                          # do the following |V|-1 times
  for each edge (u, v)                         # for every edge in the graph...
    if dist_to_s[v] + d(u, v) < dist_to_s[u]
      dist_to_s[u] = dist_to_s[v] + d(u, v)    # if we find a shorter path, update distance to v
      parent[u] = v
    end
  end
end

for each edge (u, v)
  if dist_to_s[v] + d(u, v) < dist_to_s[u]
    report there is a negative weight cycle     # check for negative weight cycles
  end
end
```