Heap

1) A complete tree is represented as an array.
2) Insert/delete is log(n):
    a) insert-> insert at the end and swap with a parent until heap property is satisfied
    b) delete->remove root, replace with the last element, and swap with children until heap property is satisfied.
    c) The last element is always touched because it ensures a complete tree
3) Heapify is done in place, there are two ways, siftUp, and siftDown.
    a) siftUp is like inserting one element at a time at the last position and takes **nlog(n)**
    b) siftDown takes less time **O(n)** as you skip n/2 elements at the bottom
    c) https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity
4) Push and pop can be done together in log(n) time instead of 2*log(n).
    a) If pop first then push, remove the [0] element, and put the new element in 0, and call siftDown.
    b) If push first then pop, simply compare [0] element with the new element, if the first element is still smallest then pop and put the new one at 0 and call siftDown. If the new element is smaller than 0, then simply return the new element.
5) Heapsort, heapify first then build by deletion. The deleted element goes in the last element's spot. So heapsort can be done entirely in-place in **nlog(n)**

Disjoint Set Union (DFU)

1) Useful to find disjoint sets given a list of edges, or useful for finding cycles (if both nodes of an edge are already in a given set).
2) Algo is makeset, findset, and union.
3) Makeset is associating a node with its parent in a tree-like fashion, the find method finds all related nodes. A set() may feel natural but is overkill as in union, you are always unioning a disjoint set so you can just use a linked list and remove the common element.
4) There are a couple of ways to implement find/union.
    a) Use a lookup table for find, during union update look up table of one of the sets. If the larger set is moved to the smaller set then the operation costs **O(n)** as in the worst case you will move all elements (n-1). The preferred way is to move the smaller set, the amortized cost of which is **log(n)**. Therefore overall cost is **nlog(n)**
    b) The other method is to use a tree based lookup where the root is the set-id. In this case the find becomes o(h) in the worst case o(n) and union is o(1). It is similar to the previous method except the cost is moved from union to find. And similarly if the root of the smaller set is added as a child of larger root during the union, then find is **log(n)** see c. for analysis. **The size of the set (called rank) is measured in height of the tree and not the number of elements**. Also the rank only increases when the rank of both sets are the same, otherwise the rank

does not change (use a simple example to see why). With path compression (update the parent to ultimate parent in find), the time complexity is **almost linear** through some fancy mathematics. It is linear as amortized cost of find is o(1).

c) With union-by-rank, a node of rank k has a minimum 2^k node (why induction). Therefore max rank is n ~= 2^k or k = log(n).
d) Vazirani page 139.
e) In the algo, figure out again when the rank should increase.. Not very intuitive

Minimum Spanning Trees

1) Kruskal's algorithm is similar to DFU above. Choose a smallest cost edge that does not form a cycle, until all nodes are selected.Takes **v log(e).** Min heap of edges, and you have to select |V| -1 edges. To take the smallest cost edge, the edges need to sorted first which make the minimum cost nlogn where n is |E|. Therefore path compression in DfU will not help reduce time complexity as the complexity is dominated by the sorting but if the edges are already sorted then it does.
2) Number of needed edges = number of vertices - 1 |E| = |V| - 1
3) Prim's algorithm, select smallest edge and then select smallest edge that is connected to the already selected vertices. Implementation is like dijekstra, start with a vertex then do dijekstra bfs using PQ, until all edges have been travelled or all nodes discovered.
4) Kruskal will find MST on disconnected forest, the mst will be for each connected graph. Prim's is for connected graphs only.
5) Prim's algorithm is very similar to Kruskal's: whereas Kruskal's "grows" a forest of trees, Prim's algorithm grows a single tree until it becomes the minimum spanning tree.

Tree and Graph

1) A connected acyclic undirected graph has |V| - 1 edges. If a undirected or directed connected graph has |V| - 1 edges then it is a tree and acyclic. However a directed graph with more |V| - 1 edges can be acyclic. Therefore to prove that a undirected graph is acyclic, just that they are connected and has |V| - 1 edges.
2) A tree has only one path between any two nodes.
3) On a graph, a cycle can be found by running DFS and keeping track of visited nodes. If a node has been visited, and if we are attempting to visit again then there is a cycle. In an undirected graph, you need to check that the visit is not due to the parent edge.
4) DFU above can also be used to find cycles.
5) DFS like DFU can be used to find disconnected components. All nodes that can be found in DFS search are connected, each restart of DFS starts in a different disconnected graph. On a directed graph, you may have to do multiple DFS until the root is used as a starting point.
6) On a DAG, there are 2 options
    a) Run DFS, multiple DFS may need to be run until the root is found.
    b) Use the topological sort and ensure that nodes can be sorted.

Strongly Connected Components

1) To be strong connected u is connected to v and vice versa
2) Within a graph, there may be multiple strongly-connected groups. The form a DAG. Otherwise they would be part of the same group.
3) Tarjan's algorithm and Kosaraju's algo to find all strongly connected components
4) Articulation points, find nodes instead of edges that separate the graphs

Graph Distances

1) BFS and DFS efficiency is O(V+E).. Though E = kV + c and therefore effectively O(V+E) is the same as O(E). However in some graphs, there can be no edges, and yet all vertices need to be represented so O(V+E) is preferred. In a connected graph algo is O(E).
2) Dijkstra's algorithm
    a) Start with a node and add it to min PQ with 0 distance
    b) Pop from PQ and add its neighbors to the PQ. The pop represents the shortest distance for the popped node.
    c) Add the neighbors to the PQ. Generally you would want to update the distance of nodes that are already in the PQ, but most PQ implementations (heaps) do not allow for update as it is the same as delete and add which is log(n). So keep track of popped nodes and add nodes to the PQ. When popping check against already popped nodes and ignore if already popped.
    d) IMP: The order of popped nodes does not indicate the path to a node. You need to explicitly track the prev node, that is, when adding to the PQ also add from which node it was added and when the node is popped you know the previous.
    e) IMP: The popped nodes are not necessarily in the eventual shortest path. They are just part of explored paths.
    f) IMP: In unweighted graph Dijkstra is same as BFS but efficiency is here as instead of PQ a simple Q is used.
    g) Efficiency is O(|V| + |E|)*log(|V|). Standard BFS cost linear, but worst case popping from PQ adds log(|V|) multiplier. Though in PQ implementation, it would be log(|E|) as we add the same node multiple times.
    h) Does not work with negative weights.
    i) Works on cyclic graphs.

3) Bellman-Ford - works with negative weights, not negative cycles
    a) relax V-1 times to get to the shortest path, that is for each edge (u,v), update distance of v.
    b) V-1 because that is the max edges between 2 vertices
    c) With each relax you increase the depth of the shortest distance. 1st relax you know the shortest distance to immediate neighbor, next you know the shortest distance to 2nd degree neighbor. Therefore V-1 relax you know V-1 depth

neighbor which is the max number of edges between any 2 vertices. This is like a growing frontier with each cycle.
   d) Detects -ve weight cycle (sum of all weights in a cycle is negative), if after |V|-1 updates the distances keeps changing.
   e) worst case o(v*e) or o(n**3) in a full graph
4) There is no shortest path solution for negative weight cycles, the problem is ill-defined.

Topological Sort - Khan's Algorithm

1) Calculate in-degree of all nodes, adjacency list representation works well.
2) Put all nodes with 0 in-degree into a Q
3) Pop Q and append to the sorted list, assume the node is removed from the original graph and update in-degree of remaining nodes. If in-degree is 0, add to Q.
4) If Q is empty and all nodes are not visited then there is a cycle.

Reservoir Sampling

1) Assume a reservoir of size k, that needs to be filled from a stream of items
2) From the stream fill the reservoir first, then for each item at position > k, check whether it needs to be selected or not and if selected with which item in the reservoir it should be replaced.
3) Consider element *i,* choose and replace it with a element in a reservoir , if a random number between 1 and *i* is less than k.
4) Probability to select *i*th item, *k/i*, probability that *i+1* item will not replace *i*th item 1-1/(*i*+1). Therefore probability that none of i+1...n will replace i. (1-1/i+1)*(1-1/i+2)*....(1-1/n) = i/n. Therefore probability of selecting ith item k/i*i/n=k/n.

Median of an array or kth largest

1. Choose a random value V and split the the array in place SL, SV, SR where SL contains all elements less than V and SV contains V and SR contains elements larger than V.
2. Depending on k, the kth largest is either in SL, SV or SR, so recurse as (SL, K) or V or (SR, (k - |SL| - |SV|))
3. Efficiency is O(n) in average case, o(n^2) in worst case. The same principle is used in quick sort. In the average case, each iteration will work on n/2.. So n + n/2 + n/4… = 2n.
4. Quick-select (practise partition)The partitioning of the array can be done in place. Revise as it is not straightforward. The usual is a 2-way partition instead of a 3 way as the algo suggests. TODO: Lomuto vs Hoare partitioning scheme. Lomuto is simpler, just start looking at each item and swap with the partition-index. Here are two algos:
   a. while(left < right):
      if item[left] > pivot:
         leftItem, rightItem = rightItem, leftItem #swap
         right -= 1 #after swap the right item is definitely greater than pivot, so move
      else:
   left += 1 #

Now remember to check the left pointer because it hasn't been in the while loop

b) Lomuto's algo starts with both pointer at 0. One pointer represents, the iterator over the list, the second pointer is the storage index. Any number less than pivot is moved to the storage which indicates the start of the larger set.

```
StorageIndex = 0
For i, num  in enumerate(nums):
    If num < pivot:
      Nums[storageIndex], nums[i] = nums[i], nums[storageIndex]
      storageIndex += 1i
```

5. This can also be used to find top-k elements, as when you find the kth element, everything on the right side is less than k.
   https://leetcode.com/problems/top-k-frequent-elements

Euler's Circuit or Path

1. Given an undirected or directed graph, the Eulerian path traverses each edge exactly once. If the start and end are the same then it is a circle. Only works if you know all nodes are connected, does not work on a disconnected graph. The following table helps to determine whether an eulerian path or circuit exists in a directed and undirected graph.

	Eulerian Circuit	**Eulerian Path**
Undirected graph	Every node has an even degree	OR exactly 2 nodes have odd degrees
Directed graph	Every node has the same in-degree and outdegree	OR utmost 1 node has +ve outdegree and utmost 1 node -ve outdegree

2. You must find the start node and then travel edges until a node has no more unvisited edge left. That will be the last node in the path, backtrack and continue.

Python sort

1. Python sort is a stable sort, if the item is a tuple, it will use all the elements to sort with the first element being the first sort order, followed by the second
2. You can also write your own comparator function and pass it to key.

```
d
[].sort(key=compare) sorted(myList, key=compare, reverse=False)
```