B-Tree

2-3 Tree:

- All operation have TC of $O(logn)$
- 1 <= # of keys in node <= 2
- 2 <= # of children of a node <= 3

MFS (disjoint sets)

Find(S, value): *Find the root, with path compression*

- Time = 1: $O(n)$ n=# of element in the set
- Time > 1: $O(1)$ due to path compression*

- Keep an array of looped through elements
- Traverse until meet the root element. Return the root element
- Direct the parent of those elements to be the root (set representative) element

```
idx = S.values.index(value)
p = []
while ( s.parents[idx] != -1 ):
    p.append(idx)
    idx = s.parents[idx]

for i in p:
    s.parents[i] = idx

return s.values[idx]
```

Merge(S, value1, value2):

- Get the root of the 2 values

- Set root of 1 value to point at the root of the other value

```
    set1 = self.Find(s, value1)
    set2 = self.Find(s, value2)

    idx1 = s.values.index(set1)
    idx2 = s.values.index(set2)

    # Check if loop -> exit
    if ( set1 != set2 ):
        s.parents[idx1] = idx2
```

Graph

Discovery

DFS(G, v): (general preorder traversal) - $O(e)$

- Have a visited array
- Visit v, mark as visited. Then visit each adjacent elements of v recursively
- If dead end, choose at random an element that's not visited and do the same procedure until all elements are visited

```
def dfs_helper(tree, is_visited, i):
    is_leaf = True
    is_print = False
    for j in range(0, len(tree.parents)):
        if is_visited[j]:
            continue
        if tree.parents[j] == i:
            is_leaf = False
            is_visited[i] = 1
            if not is_print:
                print(tree.values[i], end=" ")
                is_print=True
            dfs_helper(tree, is_visited, j)
    if is_leaf:
        print(tree.values[i], end=" ")
```

Check for cycles

- Have a visited array. Init to all false
- Have a recursion stack array. Init to all false
- Loop through all nodes in graph. For each node, run check_if_cycles(node, visited, recstack)
- On each iteration: mark el as visited in both array. Loop through each of el's neighbor. If neighbor is

not visited, check_if_cycle(neighbor, visited, recstack). If neighbor is visited, means that there is a loop, return true. If loop finishes, mark recstack[node] false and return false.

```
def isCyclicUtil(self, v, visited, recStack):
    visited[v] = True
    recStack[v] = True
    for neighbour in self.graph[v]:
        if visited[neighbour] == False:
            if self.isCyclicUtil(neighbour, visited, recStack) == True:
                return True
        elif recStack[neighbour] == True:
            return True

    # The node needs to be poped from
    # recursion stack before function ends
    recStack[v] = False
    return False

def isCyclic(self):
    visited = [False] * self.V
    recStack = [False] * self.V
    for node in range(self.V):
        if visited[node] == False:
            if self.isCyclicUtil(node,visited,recStack) == True:
                return True
    return False
```

Strongly connected component

- Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0
- Reverse directions of all arcs to obtain the transpose graph
- One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS on v. The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack)

BFS(G, v):

- Have a visited array
- Have a stack. Add v to stack
- Pop from stack, mark el as visited. Add el's adjancents to the stack if el is not visited. And repeat until stack is empty

```
    is_visited = [0 for i in range(0, len(tree.parents)) ]
    queue = []
    root_idx = tree.parents.index(-1)
```

```
    queue.append(tree.values[root_idx])
    is_visited[root_idx] = 1
    while sum(is_visited) < len(is_visited):
        if queue:
            i = queue.pop(0)
            print(i, end=" ")

        for j in range(0, len(tree.parents)):
            if is_visited[j]:
                continue
            if tree.parents[j] == root_idx:
                queue.append(tree.values[j])
                is_visited[j] = 1
        root_idx = tree.values.index(i)
    while queue:
        print(queue.pop(0), end=" ")
    print()
```

Single-Source Shortest Path (SSSP)

Find the shortest path between v and all elements in graph

Unweighted graph: Use BFS

Weighted graph: Use Dijkstra

Dijkstra(G, v):

- Have a distance array. Initialize all distances to infinity
- Have a predecessor array
- If distance from v to a1 < dist[a1] => dist[a1] = dist_v_a1. Do this for all v's adjacents
- Pick the shortest distance from v to any of its adjacency. Update the predecessor array. Do the step above with the picked node until all nodes are reached
- *Dijkstra works because adding up all local minimum will result in a global minimum. Hence, the algorithm only works with non-negative edge weights*
- *With adjacency matrix -> loop is $O(n)$, executed n-1 times -> $O(n^2)$*
- *When $e << n^2$, it's better to use adjacency list (using a priority queue), which results in e updates, each costs $O(logn)$ -> $O(elogn)$*

Bellman-Ford:

Bellman-Ford is another example of a single-source shortest-path algorithm, like *Dijkstra*. Bellman-Ford and Floyd-Warshall are similar—for example, they're both dynamic programming algorithms—but Floyd-Warshall is not the same algorithm as "for each node v, run Bellman-Ford with v as the source node". In particular, Floyd-Warshall runs in $O(v^3)$ time, while repeated-Bellman-Ford runs in $O(v^2e*)time*(*O(*ve)$ time for each source vertex).

All-Pairs Shortest Path (APSP)

Find the shortest path between v and w in graph G

Use Dijkstra:

- Can run Dijkstra on all vertices
- Cost = \$O(nelogn)$foradjlistand$O(n^3)\$ for adj matrix

Use Floyd-Warshall:

- Use n x n distance matrix. Initialize to all inf. All diagonals are 0's
- Make n iterations over dist matrix. After k_{th} iteration, dist[i, j] will have for its value the smallest length of any path from vertex i to vertex j that does not pass through a vertex numbered higher than k. In the k_{th} iteration, $A_k[i,j] = min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$

```
    init_dist_mat()
    init_pred_mat()

    for k in range (0, len(vtx_array)):
        for i in range (0, len(vtx_array)):
            for j in range (0, len(vtx_array)):
                if distance_matrix[i][j] > distance_matrix[i][k] +
distance_matrix[k][j]:
                    distance_matrix[i][j] = distance_matrix[i][k] +
distance_matrix[k][j]
                    predecessor_matrix[i][j] = k
```

Minimum Spanning Tree

Find the sub tree (all vertices are connected) such that it's weight is minimum

Prim:

- Create a set mstSet that keeps track of vertices already included in MST
- Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first
- While mstSet doesn’t include all vertices
    - Pick a vertex u which is not there in mstSet and has minimum key value
    - Include u to mstSet
    - Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

Kruskal:

- Sort all the edges in non-decreasing order of their weight
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it
- Repeat the step above until there are (V-1) edges in the spanning tree

Dependency Graph

Directed Acyclic Graph (DAG):

- Directed graph with no cycles

Topological Sort

- Topological sort is a process of assigning a linear ordering to the vertices of a DAG so that if there is an arc from vertex i to vertex j, then i appears before j in the linear ordering.
- Can be achieved with a print statement after DFS
- This technique works because there are no back arcs in a dag. Consider what happens when depth-first search leaves a vertex x for the last time. The only arcs emanating from v are tree, forward, and cross arcs. But all these arcs are directed towards vertices that have already been completely visited by the search and therefore precede x in the order being constructed.

SCC

- Technically explained above:
    - Get Topological Sort of Graph (push on a stack)
    - Get Transpose of Graph
    - Run DFS on the top-sort stack with the tranposed graph

Critical Path

- The Longest Path through all components (in a DAG). Reverse of SSSP.

Bipartite Matching

- A graph whose vertices can be divided into two disjoint groups with each edge having one end in each group is called bipartite.
- The matching problem can be formulated in general terms as follows:
    - Given a graph G=(V, E), a subset of the edges in E with no two edges incident upon the same vertex in V is called a matching.
    - The task of selecting a maximum subset of such edges is called the maximal matching problem. A complete matching is a matching in which every vertex is an endpoint of some edge in the matching.

Augmenting paths

- Start with M = Ø.
- Find an augmenting path P relative to M and replace M by M ⊕ P.

- Repeat step (2) until no further augmenting paths exist, at which point M is a maximal matching.

Recurrence Relations

Master Method

> $$T(n) = aT(n/b) + f(n)$$

3 Cases

1. The running time is dominated by the cost at the leaves:

> If $f(n) = O(n^{log_b(a)-\epsilon})$, then $T(n) = \theta(n^{log_b(a)})$ for an $\epsilon > 0$

2. The running time is evenly distributed through out the tree:

> If $f(n) = \theta(n^{log_b(a)})$, then $T(n) = \theta(n^{log_b(a)} * log(n))$

3. The running time is dominated by the cost at the root:

> If $f(n) = \Omega(n^{log_b(a)+\epsilon})$, then $T(n) = \theta(f(n))$ for an $\epsilon > 0$

Procedures

- Extract $a, b, f(n)$
- Calculate $n^{log_b(a)}$. Compare this to $f(n)$ asymtotically
- Select one of the 3 cases and get the answer

Tree Method

https://www.youtube.com/watch?v=sLNPd_nPGIc

Substitution method

https://www.youtube.com/watch?v=Ob8SM0fz6p0

Sort

Insertion Sort

Worst case $O(n^2)$ **Best case** $O(n)$

- Keep swaping until condition is met (e.g. A[i+1] > A[i])

```
i ← 1
while i < length(A)
    x ← A[i]
    j ← i - 1
```

```
        while j >= 0 and A[j] > x
            A[j+1] ← A[j]
            j ← j - 1
        end while
        A[j+1] ← x[4]
        i ← i + 1
    end while
```

Selection Sort (Bubble)

Worst case $O(n^2)$ **Best case** $O(n^2)$

- Find min of the unsorted set, then swap with the last element in sorted set

```
    for i = 0->n:
        min = i
        for j = i+1->n:
            if A[j] < min:
                min = j
        if i != j:
            A.swap(i, j)
```

Merge Sort

Worst case $O(nlogn)$ **Best case** $O(nlogn)$ **Memory** $O(n)$ **(need a temp array)**

- Keep dividing in halfs using the middle element until there is one element. Then call merge() on the halfs (this sort them)

```
def mergeSort(arr, l,  r):
    If r > l
        1. Find the middle point to divide the array into two halves:
                middle m = (l+r)/2
        2. Call mergeSort for first half:
                Call mergeSort(arr, l, m)
        3. Call mergeSort for second half:
                Call mergeSort(arr, m+1, r)
        4. Merge the two halves sorted in step 2 and 3:
                Call merge(arr, l, m, r)

def merge(arr, l, m, r):
    # Copy data to temp arrays L[] and R[]
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]
            i+=1
        else:
            arr[k] = R[j]
```

```
            j+=1
        k+=1

    # Checking if any element was left
    while i < len(L):
        arr[k] = L[i]
        i+=1
        k+=1

    while j < len(R):
        arr[k] = R[j]
        j+=1
        k+=1
```

Quick Sort

Worst case $O(n^2)$ **Best case** $O(nlogn)$

- Instead of dividing in halfs using the middle element, use the ***pivot***, which can be chosen wisely to improve the algorithm run time depending on the dataset.
- After choosing the pivot, the keys are sorted such as all the elements < pivot is on the left and all the elements > pivot are on the right
- Run quicksort() recursively on the left set and the right set until the set only have 1 element in it.

```
def quickSort(arr, low, high):
{
    if (low < high)
    {
        pi = partition(arr, low, high); // partition and return the position of
pivot

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}

def partition (arr, low, high):
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
```

```
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Heap Sort

Worst case $O(nlogn)$ Best case $O(n)$ or $O(nlogn)$ if all keys are distinct

- Heap sort is performed on a heap. A heap is a Complete Binary Tree.
- Build a max-heap from the input array -> get a binary tree where the root is the largest element and the left-most element is the smallest element.
- Hence, we swap the left-most and the root, push the root on to a stack and run heapify to get a max-heap again. Do this until the size of stack is same as arr.

```
# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        # Heapify the root.
        heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
```

```
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)
```

Radix Sort

*Let there be d digits in input integers. Radix Sort takes $O(d(n+b))=O((n+b) * log_b(k))$ time, where b is the base and k is the max possible value**

If we have $log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers

```
def countingSort(arr, exp1):
    n = len(arr)

    # The output array elements that will have sorted arr
    output = [0] * (n)

    # initialize count array as 0
    count = [0] * (10)

    # Store count of occurrences in count[]
    for i in range(0, n):
        index = (arr[i]/exp1)
        count[ (index)%10 ] += 1

    # Change count[i] so that count[i] now contains actual
    #  position of this digit in output array
    for i in range(1,10):
        count[i] += count[i-1]

    # Build the output array
    i = n-1
    while i>=0:
        index = (arr[i]/exp1)
        output[ count[ (index)%10 ] - 1] = arr[i]
        count[ (index)%10 ] -= 1
        i -= 1

    # Copying the output array to arr[],
    # so that arr now contains sorted numbers
    i = 0
    for i in range(0,len(arr)):
        arr[i] = output[i]

# Method to do Radix Sort
def radixSort(arr):
```

```
    # Find the maximum number to know number of digits
    max1 = max(arr)

    # Do counting sort for every digit. Note that instead
    # of passing digit number, exp is passed. exp is 10^i
    # where i is current digit number
    exp = 1
    while max1/exp > 0:
        countingSort(arr,exp)
        exp *= 10
```

Search

Linear Search

- It's just linear search 😃

Binary Search

- Cooler search. Only works on sorted sets. Run time = $O(logn)$