Greedy Algorithms

Kruskal's algorithm: RT: O(nlogn), Finds the minimum spanning tree in an undirected edge-weighted graph
Basic algo: sort all edges in increasing order by weight, pick smallest edge and check if it forms a cycle with spanning tree formed so far. If no cycle, include it, otherwise discard, repeat until (V-1) edges in spanning tree
Starts with each vertex in its own component and repeatedly merges two components into one by choosing a light edge that connects them, scans the set of edges in monotonically increasing order by weight, uses a disjoint-set data structure to determine whether an edge connects vertices in different components

Kosaraju's Algo: RT: O(n), finds strongly connected components in graph
Check every node in the graph, if one of the nodes allows you to visit every other node in the graph, then it is a strongly connected component.
1. Call DFS(graph) to compute the finish time for each vertex, 2. Compute G(t), 3. Call DFS(Gt) on vertices in decreasing order of their finish times, 4. Output vertices as separate SCC's to do this. SSC's must have a path that is **cyclical**
1. Create empty stack 'S', 2. Do DFS traversal of a graph, while after calling recursive DFS for adjacent vertices of a vertex, push the vertex to the stack. 3. Reverse directions of all arcs to obtain the traverse graph. 4. One by one pop a vertex from S while it;s not empty. Let the popped vertex be 'v' Take v as source and do DFS call on V. The DFS staring from v prints strongly connected components of v

Dijkstra algorithm: RT: assume O(n), finds shortest path between nodes
Optimal substructure simply that subpath of any shortest path is itself a shortest path, and the shortest path length between some u and v is less than or equal to the shortest path from u to x to v. (triangle inequality)
See other side for algo
Dijkstra's is a greedy algorithm

BFS, RT: O(V+E), Shortest Path (Unweighted), Shortest Cycle (Unweighted, Directed), Can find Connected Components (A set of vertices in a graph that are linked to
The proof that vertices are in this order by breadth first search goes by induction on the level number. By the induction hypothesis, BFS lists all vertices at level k-1 before those at level k. Therefore it will place into L all vertices at level k before all those of level k+1, and therefore so list those of level k before those of level k+1.

Prim's algo: RT O(nlogn),Finds the minimum spanning tree in an undirected edge-weighted graph
1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph, initialize all key values as ∞, Assign key value as 0 for first vertex so it's picked first.
3. While mstSet doesn't include all vertices, either A: pick a vertex u which is not in mstSet and has minimum key value, B: Include u to mstSet, C: 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 a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. Builds on tree so A is always a tree, starts from an arbitrary "root" r, and at each step adds a light edge crossing cut (Va, V - Va) to A where Va = vertices that A is incident on. Using different data structures for representing and linearly searching array of weights to find the minimum weight edge: adj. matrix O(|V |^2), bin heap and adj lst O((|V | + |E|)log|V |) = O(|E|log|V |), Fibonacci heap and adj list O(|E| + |V |log|V |).
In the method that uses binary heaps, we can observe that the traversal is executed O(V+E) times (similar to BFS). Each traversal has operation which takes O(LogV) time. So overall time complexity is O(E+V)*O(LogV) which is O((E+V)*LogV) = O(E*LogV) (For a connected graph, V = O(E))

15 Common Recurrence Relations

$$T(n) = 2T(n/2) + n \quad (10)$$
$$T(n) = 4[2T(n/8) + n/4] + 2n \quad (11)$$
$$T(n) = 2^k T(n/(2^k)) + kn \quad (12)$$
$$T(n) = n + nlog_2 n \quad (13)$$
$$T(n) = 2log_2 nT(1) + (log_2 n)nT(n) = O(nlogn) \quad (14)$$

Recurrence	Algorithm	Big O
$T(n/2)+\Theta(1)$	Binary Search	O(log(n))
$T(n-1)+\Theta(1)$	Sequential Search	O(n)
$2T(n/2)+\Theta(1)$	tree traversal	O(n)
$T(n-1)+\Theta(n)$	Selection Sort (n2 sorts)	$O(n^2)$
$2T(n/2)+\Theta(n)$	Mergesort	O(n log n)
$T(n-1)+T(0)+\Theta(n)$	Quicksort	$O(n^2)$

Merge Sort (Divide and conquer), RT: O(nlogn), Split array into smaller arrays of size 2, sort and combine till back to original size

```
def mergesort(arr): if len(arr) == 1: return arr
        m = len(arr) / 2, l = mergesort(arr[:m]), r = mergesort(arr[m:])
        if not len(l) or not len(r): return l or r
        result = [],i = j = 0,
        while (len(result) < len(r)+len(l)):
                if l[i] < r[j]: result.append(l[i]) i += 1
                else: result.append(r[j]) j += 1
        if i == len(l) or j == len(r):
                result.extend(l[i:] or r[j:]), break
return result
```

T (n) = aT (n/b) + f (n) where a ≥ 1 and b > 1 are constants and f (n) is asymptotically positive. Case 1: if f (n) = ($n^{\log_b a-\varepsilon}$) for Some ε then T (n) = Θ($n^{\log_b a}$)
Case 2: if f (n) = Θ($n^{\log_b a}\log^k n$) with k ≥ 1 then T (n) = Θ($n^{\log_b a}\log^{k+1} n$)
Case 3: if f (n) = Ω($n^{\log_b a+\varepsilon}$) with ε > 0 and f (n) satisfies the regularity condition, then T (n) = Θ(f (n)).
Regularity condition is *af*(n/b) ≤ *cf*(n) for some c < 1 and sufficiently large n. Case on right hand side for examples

$T(n) = 3T(n/2) + n^2 = \Theta(n^2)$ 3 (15)
$T(n) = 4T(n/2) + n^2 = \Theta(n^2 log(n))$ 2 (16)
$T(n) = T(n/2) + 2^n = \Theta(2^n)$ 3 (17)
$T(n) = 2^n T(n/2) + n^n = ...$ a not constant (18)
$T(n) = 16T(n/4) + n = \Theta(n^2)$ 1 (19)
$T(n) = 2T(n/2) + nlog(n) = \Theta(nlog^2 n)$ 2 (20)
$T(n) = 2T(n/2) + n/log(n)$ non poly (21)
$T(n) = 2T(n/4) + n^{0.51} = \Theta(n^{0.51})$ 3 (22)
$T(n) = .5T(n/2) + 1/n$ a < 1 (23)
$T(n) = 16T(n/4) + n! = O(n!)$ 3 (24)
$T(n) = \sqrt{2}T(n/2) + log(n) = \Theta(\sqrt{n})$ 1 (25)
$T(n) = 3T(n/3) + \sqrt{n} = \Theta(n)$ 1 (26)
$T(n) = 4T(n/2) + cn = \Theta(nlog(n))$ 3 (27)
$T(n) = T(n/2) + n(2 - cos(n))$ regularity violate (28)

- Example:
  - The algorithm solves the problem by dividing them into four subproblems of size n/3 using $4\log_4(n)$ time, solving them recursively, and then combining the solutions in log_4(n) time.
  - a = 4, b = 3, d < (1 + e) since nlogn is faster than n but slower than $n^{(1+e)}$, Thus, T(n) = O($n^{\log_3(4)}$)

Format: $T(n) = a\,T(n/b) + \Theta(n^k (\log n)^i)$.

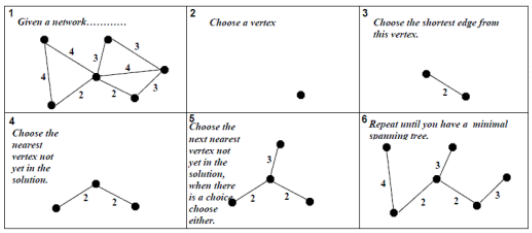
Quick Sort, RT: O(nlogn) Worst Case: O(n^2), Choose pivot and sort around the pivot This is a divide and conquer algorithm, first divide, conquer, then combine

```
def quickSort(alist):
        quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
        if first<last: splitpoint = partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1),quickSortHelper(alist,splitpoint+1,last

def partition(alist,first,last):
        pivotvalue = alist[first],leftmark = first+1, rightmark = last, done = False
        while not done:
                while leftmark <= rightmark and \ alist[leftmark] <= pivotvalue:
                        leftmark = leftmark + 1
                while alist[rightmark] >= pivotvalue and \ rightmark >= leftmark:
                        rightmark = rightmark -1
                if rightmark < leftmark: done = True
                Else: temp = alist[leftmark], alist[leftmark] = alist[rightmark]
                        alist[rightmark] = temp
        temp = alist[first], alist[first] = alist[rightmark] ,alist[rightmark] = temp
        return rightmark
```

Prim's Algorithm



Random Lingo
● NP-Complete problems to reduce from:

Independent Set: A set of k vertices that don't have any edges connecting them, This algorithm finds maximum vertices you can select that aren't connected to each other. You can also make a yes/no algorithm by asking "are there more than n vertices that are independent".
Vertex Cover: A set of vertices that cover all the edges.This algorithm finds the minimum vertices that cover all the edges. You can make this a yes/no algorithm by asking "are there fewer than n vertices that are independent". Vertex Cover and Independent Set are closely related. (With v being the number of vertices) If you have an n size independent set, you'd have a v-n size vertex cover. And it'd just be all other vertices. And if you have an n size vertex cover, you'd have a v-n size independent set.
Clique : Literally just "is there a complete subgraph of size k?" Set of vertices all connected
Rudrata Path (directed or undirected): A path that visits every vertex once
Rudrata Cycle (directed or undirected):A cycle that visits every vertex once, Find median in linear time
K-selection: Find x-th element in linear time

Topological Sort :RT: O(n), Organizes a DAG with all edges pointing from left to right
1. Call DFS to compute finishing times for all vertices, as each vergtex finishes, insert it onto the front of a linked list, and return the linked list of vertices, this is Θ(V+E).
Proof: show that if (u, v) ∈ E then f [v] < f [u]: when we explore (u,v) what are the colors of u and v, if us is grey then v cannot be grey because then v would be ancestor of u therefore (u, v) is a black edge, which is a contradiction of DAG, if v is white than becomes descendant ofu and parenthesis theorem work

PRIMS algo

```
def prim(G,start):
        pq = PriorityQueue()
        for v in G: v.setDistance(sys.maxsize), v.setPred(None) #loop
        start.setDistance(0) ,pq.buildHeap([(v.getDistance(),v) for v in G])
        while not pq.isEmpty():
        currentVert = pq.delMin()
                for nextVert in currentVert.getConnections():
                        newCost = currentVert.getWeight(nextVert) \
                        + currentVert.getDistance()
                        if v in pq and newCost<nextVert.getDistance():
                                nextVert.setPred(currentVert)
                                nextVert.setDistance(newCost)
                                pq.decreaseKey(nextVert,newCost)
```

Dijsktras algo

```
def minDistance(self, dist, sptSet):
    min = sys.maxint
    for u in range(self.V):
        if dist[u] < min and sptSet[u] == False:
            min = dist[u]
            min_index = u
    return min_index
def dijkstra(self, src):
    dist = [sys.maxint] * self.V , dist[src] = 0, sptSet = [False] * self.V
    for cout in range(self.V):
        x = self.minDistance(dist, sptSet), sptSet[x] = True
        for y in range(self.V):
            if self.graph[x][y] > 0 and sptSet[y] == False and \
            dist[y] > dist[x] + self.graph[x][y]:
                dist[y] = dist[x] + self.graph[x][y]
    Return dist
```

Kruskal algo

```
  def KruskalMST(self):
      result = []  # This will store the resultant MST
      i = 0,      e = 0
      self.graph = sorted(self.graph, key=lambda item: item[2]) #sort edges in non dec
      parent = [],      rank = []
      for node in range(self.V): parent.append(node),          rank.append(0)
      while e < self.V - 1:
          u, v, w = self.graph[i], i = i + 1 ,x = self.find(parent, u), y = self.find(parent, v)
          if x != y: e = e + 1, result.append([u, v, w]) , self.union(parent, rank, x, y)
      minimumCost = 0
      for u, v, weight in result:
          minimumCost += weight
       Return minimumCost
```

```
def bfs(graph, start):
        visited, queue = set(), [start]
        while queue:
                vertex = queue.pop(0)
                        if vertex not in visited:
                        visited.add(vertex)
                        queue.extend(graph[vertex] - visited)
        return visited
```

```
def dfs(graph, start):
        visited, stack = set(), [start]
        while stack:
                vertex = stack.pop()
                if vertex not in visited:
                        visited.add(vertex)
                        stack.extend(graph[vertex] - visited)
        return visited
```

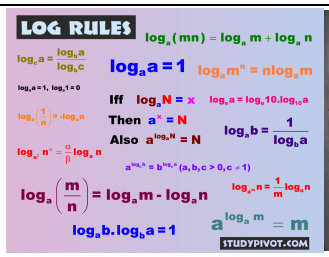
Greedy algorithms- algorithms that make the optimal choice at each step as it tries to find the overall optimal way to solve the problem. This can backfire when the solution is hidden behind a less optimal path

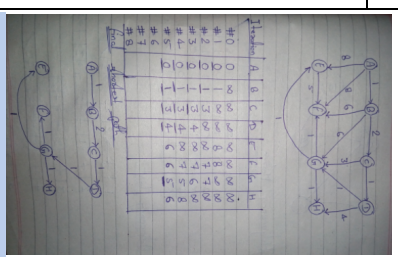
If both of the properties below are true, a greedy algorithm can be used to solve the problem.
Greedy choice property: A global (overall) optimal solution can be reached by choosing the optimal choice at each step.
Optimal substructure: A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.
This can be made by identifying an optimal substructure or subproblem

Dynamic Programming- solving a problem by recursively breaking it down into simpler subproblems as the optimal solution to the overall problem depends on the optimal solution to the individual subproblems. Each subproblem is done once and the solution is stored, this makes things less complex and is good for optimization.
This fails when there isn't an overlapping sub problem, or an optimal sub-structure, for example shortest path can be solved using DP but longest path can't solved, because longest path doesn't hold optimal sub-structure property.
Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems
Not going to give an instance of a problem and then slave to solve it (building a table, bottom up)

Memoization (Top Down): memoized program for a problem is like the recursive version with a small modification that looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom-up fashion and returns the last entry from the table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3), and so on. So literally, we are building the solutions of subproblems bottom-up.





Chapter 3 | Graphs

- Adjacency List: $O(|V| + |E|)$
- Adjacency Matrix: $O(|V|^2)$

```
function runner_dfs(G, v):
    for all v in V:
        visited(v) = false
    for all v in F:
        if not visited(v): dfs(v)

function dfs(v):
    visited(v) = true
    previsit(v) # used for labeling
    for each child:
        if not visited(child): dfs(child)
    postivisit(v) # used for labeling
```

Pre/Post Ordering Edge types

[pre(u) [pre(v) , post(v)] post(u)] – **Forward**

[pre(v) [pre(u) , post(u)] post(v)] – **Back**

[pre(v) , post(v)], [pre(u) , post(u)] – **Cross**

Graph Algorithms

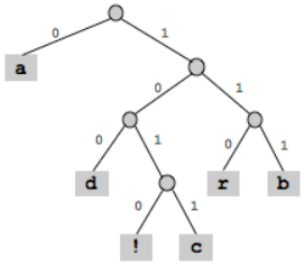
- Topological Sort - $O(|V| + |E|)$ - DFS with post ordering. Reverse post ordering to get a topological sort.
- Kosaraju's / SCC Algorithm - $O(|V| + |E|)$ - Run **DFS** search on G^R w/ post ordering. Run **BFS** on highest post order on G, everything it reaches is an SCC. Repeat on node w/ next highest post roder.
- Dijkstra's Algorithm - $O(|V|^2)$ - Minimum distance between a node and all other nodes reachable from it. Add all verticces to an array / heap with values of infinity. Start at a source node. For all the edges of a node, add the current distance (the value for the current vertex) to all the edge weights and update nodes within the heap to this new value. Pop from heap. Repeat this process.
- Bellman-Ford - $O(|V|^2)$ - Minimum distabce between a node and all other nodes reachable from it. Works on negative edge weights, which Dijkstra's does not.
- Kruskal's Algorithm - $O(E \log V)$ - Finds MST for an undirected, weighted graph - Sort (low $\rightarrow$ high) edges by weight. Add edges s.t. cycles are not formed.
- Prim's Algorithm - $O(E \log V)$ - Finds MST for an undirected, weighted graph - Breadth first search except have a heap sorted on edge weights. Pop from the top of heap. Add edge if node has not been visited.

Graphs Properties

- For any nodes u and v, the two intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or one is contained within the other.
- Reverse post ordering will give a linearized graph. Called a topological sort. $O(|V| + |E|)$
- A directed graph has a cycle if and only if its depth-first search reveals a back edge.
- In a DAG, every edge leads to a vertex with a lower **post** number.
- Every DAG has at least one source and at least one sink
- Every directed graph is a dag of its strongly connected components.
- If the **dfs** function is started at a node u, then it will terminate precisely when all nodes reachable from u have been visited.
- The node that receives the highest **post** number in a depth-first search must lie in a source strongly connected component
- If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest *post* number in C is bigger than the highest *post* number in C'.

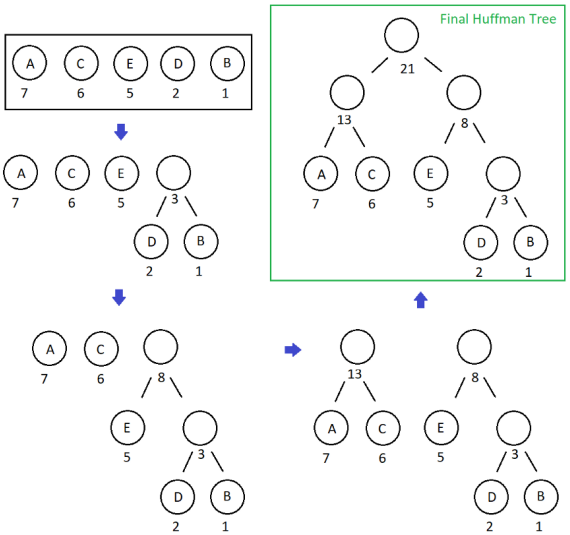
Huffman Encoding

A binary tree where more frequent characters have smaller encodings.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

1. Create key, value pairs for the characters and their frequencies. Sort low to high based on frequency.
2. From the collection, pick out the two nodes who has the smallest sum of frequency.
3. The root has a sum of the frequencies. Add to the graph.
4. Repeat this process until all probabilities are used.



Dynamic Programming

Procedure for finding and writing a DP solution.

1. Subproblem: Define what a index of the array represents.
2. Recurrence: Define how to find a solution of a subproblem of your subproblems based on smaller subproblems.
3. Algorithm: **1.)** Define size and default values of your array. **2.)** Set base case values. **3.)** Apply recurrence ot the array. **4.)** Give the return value. May be a single index, or may involve iterating through the array.
4. Correctness: Doesn't and shouldnt be a formal proof. Give a description of the cases for the recurrence, and why they model the problem.
5. Running time: Doesn't need to be long or formal, give a brief description of why/what the running time is.

Knapsack Variations

– Total capacity `W`. Also w_i, v_i is weight and value for object i.
– Knapsack w/ Repetition: If $\exists$ a solution for $K(W)$ then removing item i gives the solution for $K(W - w_i)$. Thus:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

– Knapsack w/out Repetition: $K(w, j)$ is the maximum value achievable using a knapsack of capacity w and items $1, ..., j$.

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Linear Programming

A broad set of problems that take in constraints and optimization criterion as linear functions. This boils down to assigning real values to a set of variables such that they satisfy a set of linear equations/inequalities and maximize/minimize a linear objective function.

– **Duality**: Every linear maximization problem has a **dual** minimization porblem.
– **Duality Theorem**: If a linear program has a bounded optimum, then so does its

dual, and the two optimum values coincide.

– **Max-flow min-cut theorem**: size of maximum flow == capacity of smallest min-cut.
– **Bipartite matching**: Given a bipartite graph, find a set of edges st everything is paired exactly once. Add a source to one side and a sink to other and all edges have a weight of one. */exists* a perfect matching iff network flow == number of couples

Primal Linear Program:

$$\begin{aligned} \min / \max\{\alpha x_1 + \beta x_2 + \gamma x_3\} \\ k_{1,1}x_1 + k_{1,2}x_2 + k_{1,3}x_3 \leq a \\ k_{2,1}x_1 + k_{2,2}x_2 + k_{2,3}x_3 \leq b \\ k_{3,1}x_1 + k_{3,2}x_2 + k_{3,3}x_3 \leq c \\ x_1, x_2, x_3 \geq C \end{aligned}$$

The Dual:

$$\begin{aligned} \max / \min\{ay_1 + by_2 + cy_3\} \\ k_{1,1}y_1 + k_{2,1}y_2 + k_{3,1}y_3 \geq \alpha \\ k_{1,2}y_1 + k_{2,2}y_2 + k_{3,2}y_3 \geq \beta \\ k_{1,3}y_1 + k_{2,3}y_2 + k_{3,3}y_3 \geq \gamma \\ y_1, y_2, y_3 \geq C \end{aligned}$$

NP-Complete

– **NP** - all search problems, nondeterministic polynomial time
– **P** - can be solved in polynomial time
– **NP-Complete** - subset of problems in NP but not in P
– Changing a search problem into an optimization problem is quite easy as the two reduce into one another.
– **SAT** - Given a Boolean formula in conjunctive normal form (CNF), set each boolean variables to a True/False value such that all the clauses are satisfied. If each clause has at most three boolean values then it is **3SAT**.

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

– Euler / Rudrata - Can every vertex within a graph be reached without visiting a vertex twice?
– **Traveling Salesman** - Given a graph G and a maximum cost M, find a path through n vertices such that the total distance traveled is less than M.
– Longest Path - Finding the path with maximum length between two vertices.
– 3D matching - Given three disjoint sets and a set of triplets representing edges (a, b, c) where $a \in A$, $b \in B$, and $c \in C$. Find a subset of the triplets such that each point in $A \cup B \cup C$ appears exactly once in the triplets.
– Knapsack - Given a set of items with weights and values, find which items to fill a knapsack such that the knapsack does not exceed a certain weight.
– Independent Set - Given a graph G with a set of edges E, determine what is the largest number of vertices such that there does not exist an edge between any two vertices.
– **Integer Linear Programming** - Linear proramming except the answer must be integer values. This turns linear programming into a NP-complete problem.