CS 170 Cheat Sheet

Big O notation

$f, g \in \mathbb{N}$, $f = O(g)$ means that f grows no faster than g if $\exists c > 0$ s.t. $F(n) \leq cg(n)$
$f = \Theta(g)$ means $g = O(f)$
$f = \Theta(g)$ IFF $f = O(g)$ & $g = \Theta(g)$

Master Theorem

Given: $T(n) = a \times T(\frac{n}{b}) + O(n^d)$

a) $O(n^d)$ if $d > log_b(a)$

b) $O(n^d log(n))$ if $d = log_b(a)$

c) $O(n^{log_b(a)})$ if $d < log_b(a)$

Graph Algorithms

DFS: O(V + E)

Guaranteed to visit every node reachable by v before returning from v. Can create topological sort of DAG.

```
procedure explore(G,v)
Input:    G = (V,E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes

visited(v) = true
previsit(v)
for each edge (v,u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

BFS: O(V + E)

Used to find shortest path through an unweighted graph.

```
for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s]  (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u,v) ∈ E:
      if dist(v) = ∞:
         inject(Q,v)
         dist(v) = dist(u) + 1
```

Dijkstras: O((V + E) log V)

Like BFS but with priority queue, used to find shortest path between two nodes on a weighted graph.

```
for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0

H = makequeue(V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u,v) ∈ E:
      if dist(v) > dist(u) + l(u,v):
         dist(v) = dist(u) + l(u,v)
         prev(v) = u
         decreasekey(H,v)
```

Bellman Ford: O((V E))

Find shortest paths with negative edges as long as there are no negative cycles. Runs $V - 1$ updates on all E edges.

```
procedure update((u,v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u,v)}

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

Floyd-Warshall: $O((V^3)$

Find the shortest path between all pairs of vertexes in a graph.

Kruskal: O((E log(V))

Use the disjoint set trees to add edges in ascending order that don't complete a cycle. Used to find MST.

```
for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u,v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u,v} to X
      union(u,v)
```

Prim's: O((E log(V))

On each iteration, the subtree defined by X grows by one edge, namely, the lightest edge between a vertex in S and a vertex outside S

```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
   pick a set S ⊂ V for which X has no edges between S and V − S
   let e ∈ E be the minimum-weight edge between S and V − S
   X = X ∪ {e}
```

Cut Property: Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across the partition. Then $X \cup e$ is part of some Minimum Spanning Tree.

Huffman Encoding

Make a tree of decisions of whether to pick a 0 or a 1, make all the leaves values. Then we can follow the tree down to decode a Huffman encoding of values.

FFT

It is a black box which represents 2 polynomials as a list of points and then multiplies them together to create a new polynomial. Takes **O(N log N)** time. Uses roots of unity to determine where to multiply two polynomials together.
N^{th} **Roots of Unity** can be found by: $cos(\frac{2\pi j}{n}) + i \cdot sin(\frac{2\pi j}{n})$

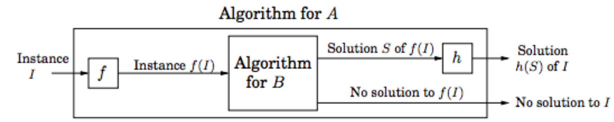
P/NP Basics

P: search problem that can be solved in polynomial time.
NP: search problem that can be checked to be correct in polynomial time.
NP complete: search problem that is at least as hard as every other NP complete problem. Basically it reduces to circuit sat, could possibly be solved in polynomial time but we doubt it.
NP hard: there exists NP complete Y such that Y is reducible to X but can't go the other way around. Not actually in NP.

Reductions

A search problem is NP-complete if all other search problems reduce to it.

To reduce X to Y means to find a solution for X using Y.



Dynamic Programming

These are normally straight forward, mainly we just need to find a recursive relationship for sub problems and then figure out the order in which we need to solve them. Normally runtime can easily be deduced by the number of for loops we have to run to. Following are some basic examples of dynamic programming in case we see something like these on the final.

Edit Distance: $O(n^2)$

```
for i = 0,1,2,...,m:
   E(i,0) = i
for j = 1,2,...,n:
   E(0,j) = j
for i = 1,2,...,m:
   for j = 1,2,...,n:
      E(i,j) = min{E(i−1,j)+1, E(i,j−1)+1, E(i−1,j−1)+diff(i,j)}
return E(m,n)
```

Knapsack with repetition: $O(nW)$

K(w) is the maximum value we can achieve with weight w.

```
K(0) = 0
for w = 1 to W:
   K(w) = max{K(w − w_i) + v_i : w_i ≤ w}
return K(W)
```

Knapsack without repetition: $O(nW)$

$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, ..., j$.

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
   for w = 1 to W:
      if w_j > w:   K(w,j) = K(w,j−1)
      else:   K(w,j) = max{K(w,j−1), K(w−w_j,j−1) + v_j}
return K(W,n)
```