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 $d < log_b(a)$

Sorting Algorithm Runtimes

$HeapSort$ and $MergeSort \rightarrow \Theta(n \log n)$
$InsertionSort \rightarrow \Omega(n)$ to $O(n^2)$
$QuickSort \rightarrow \Omega(n \log n)$ to $O(n^2)$

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)
```

A forward edge goes to a node already in the explored set. Pre and post ordering and the edge types look like this:

pre/post ordering for (u, v)	Edge type
$[_u \; [_v \;]_v \;]_u$	Tree/forward
$[_v \; [_u \;]_u \;]_v$	Back
$[_v \;]_v \; [_u \;]_u$	Cross

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. Does not work with negative edge weights.

```
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. To find a negative cycle, run V times, if values are updated after the $V - 1$ update, there must be a negative cycle. $update()$ is harmless.

```
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 with positive or negative edge weights.

Kruskal: O(E log V)

Use the disjoint set trees to add edges in order of increasing edge weight that doesn'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((V + 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. Similar implementation as Dijkstra'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 O(n log n)

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: $cos(\frac{2\pi j}{n}) + i \cdot sin(\frac{2\pi j}{n})$

Dynamic Programming

Find a recursive relationship for sub problems and then figure out the order in which we need to solve them. Runtime can easily be deduced by the number of for loops we have to run. Following are some basic examples of dynamic programming:

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)
```

Longest Palindromic Subsequence: $O(n^2)$

For $0 \leq i, j \leq n$ define $f(i, j)$ = length of the longest palindromic subsequence of $x[i \ldots j]$.
Base case: when $i : j$ (empty string), $f(i, j) = 0$; when $i == j$ (single character), $f(i, j) = 1$
When $i < j$,

$$f(i, j) = max \begin{cases} f(i + 1, j) \\ f(i, j - 1) \\ 2 + f(i + 1, j - 1) \quad \text{if } x[i] = x[j] \end{cases}$$

Optimal Binary Search Tree with Probabilities:

Let $S(i, j)$ be the cost of the cheapest tree formed by words i to j, for $1 \leq i \leq j \leq n$. Also $S(i, j) = 0$ if $i > j$. Compute $S(i, j)$ by looking at all choices of which word to use as the root of the tree. If k is at the root, the cost of the left subtree is $S(i, k - 1)$ an the

right is $S(k+1, j)$. If we place word k at the root, the total cost of the tree will be:

$$\sum_{t=i}^{k} p_t + S(i, k-1) + S(k+1, j)$$

giving $S(i, j) = \min_{i \le k \le j}\{\text{the above cost}\}$

Linear Programming

Linear Programming is a way to state problems as systems of equations and then we can solve them in polynomial time as long as we don't need Integer Linear Programming, which is NP Complete. It can be solved with the simplex method which we don't need to know, can just treat it like a black box.
Remember: when writing out your solutions make sure you add the non-negative constraints.

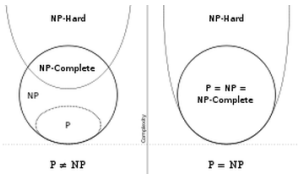
Max Flow

This is bounded by $O(C \cdot E)$ or $O(V \cdot E^2)$
Max-flow min-cut theorem: the size of the maximum flow in a network equals the capacity of the smallest (s,t) cut.
Maximum flow property: if all edge capacities are integers, then the optimal flow found by our algorithm is integral. We can see this directly from the algorithm, which in such cases would increment the flow by an integer amount on each iteration.
When drawing the residuals make sure you draw the flow you've already used backwards, so some paths with have an arrow going forward and an arrow going backward.

P/NP

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.

P/NP Basics

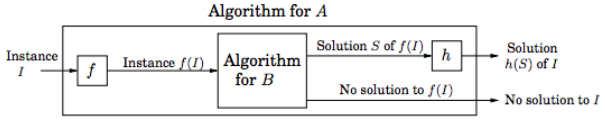


Here are some examples of **P** and **NP** algorithms:

NP	P
3+ SAT	2 SAT
TSP	MST
Longest Path	Shortest Path
3D Matching	Bipartide matching
Knapsack	Unary Knapsack
Independent Set	Independent Set on Trees
Integer LP	Linear Programming
Rudrata Path	Euler Path
Balanced Cut	Minimum cut

Reductions

A search problem is NP-complete if all other search problems reduce to it.
Make sure that the conversions between the two algorithms are of polynomial size and time, otherwise the reduction does not work.
To reduce X to Y means to find a solution for X using Y.



Dealing with NP

There are different ways to deal with NP, one is to do intelligent but exhaustive searching. This basically involves stopping looking at a branch as soon as we know that what we've selected won't work. This is basically just pruning on a tree.
Approximations
These would be like the example where we use the MST tree to find an approximation which is at most 2 times worse than the optimal solution to the TSP problem.
There are certain classes of approximations which are as follows:

- Those for which, like the TSP, no finite approximation ratio is possible.
- Those for which an approximation ratio is possible, but there are limits to how small this can be. VERTEX COVER, k-CLUSTER, and the TSP with triangle inequality belong here
- Approximability has no limits, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist. KNAPSACK resides here.
- Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about log n. SET COVER is an example

Local Search Heuristics
These can often be useful in practice but provide no guarantees for how well or how fast they will approximate the actual value.
We then deal with local optimum by using **randomization and restarts**. This means that we will effectively run the search multiple times with different random starting points. This gives us a better probability of finding the correct (or better) local optima.
Simulated annealing is another technique where we sometimes follow the optimal path, and with some random probability we actually go backwards in our search.

```
let s be any starting solution
repeat
   randomly choose a solution s' in the neighborhood of s
   if Δ = cost(s') − cost(s) is negative:
      replace s by s'
   else:
      replace s by s' with probability e^{−Δ/T}.
```

Generally we will choose our T to start large and then decrease as we go on so that with time we jump down the hills less frequently but early on we are likely to crawl out of local maximum.

Machine Learning

Classification Let $\mathcal{Y}$ denote a set of classes. For a boolean classification, such as spam filters, $\mathcal{Y} = \{0, 1\}$. Let the observation $x \in \mathbb{R}^d$ denote a d-dimensional vector of features.
K-Nearest Neighbors: $\Theta(n(d + \log k))$
Let $x_i \dots x_{i+k}$ be the k nearest observations to x from the trained set. Classify x with the class with the plurality of votes from the k feature vectors.
Note: for efficiency, normalize feature vectors to have the same range of values before applying k-NN. Algorithm performs worse if the dimension d is too large.
Decision Tree Inference (NP Hard) A *DecisionTree* is a binary tree that represents a function $\mathbb{R}^2 \to \mathcal{Y}$ where each leaf is labeled with a decision of a classification $y \in \mathcal{Y}$. If there is an observation $x \in \mathbb{R}^2$ then we start at the root of the tree and go to the left or right child depending on the threshold value t. Repeat until a leaf is reached. Inference on decision trees is a greedy approach where test points in S are split into S_L and S_R by estimating the relation of test data into similar camps (left subtrees and right subtrees). Recurse the subtrees until
1) all remaining subtrees have the same classification
2) after a certain node depth then choose most-common class or
3) after class entropy falls beneath a threshold
Random Forests A *RandomForest* is a collection of decision trees. The goal is to reduce the error rate by classifying an observation x through many decision trees and selecting the most common class as the result. Assumes each decision tree is different. Can create random trees through either
1) Bagging (used more): train each tree on a random subset of the training set S. If S contains n test observations, then sample with replacement n times to create our random subset $S^* \in S$. The expected size of S^* is about 63% the size of S. Use the Decision Tree Inference algorithm to construct a tree from S^*.
2) Feature Sampling: using a subset of d features when constructing the tree. Reduces the likely hood that each tree has the same prominent feature at its root.