COS 485 Crib Notes

Midterm 2

I. FORMULA YOU SHOULD KNOW ALREADY

$$\sum_{x=0}^{n} x = 1 + 2 + ... + n - 1 + n = (n+1)\left(\frac{n}{2}\right)$$

$$\sum_{x=0}^{\infty} \left(\frac{1}{2}\right)^x = \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8}.... = 2$$

$$\sum_{i=0}^{\lceil \lg n \rceil} \frac{n}{2^i} = \frac{n}{1} + \frac{n}{2} + \frac{n}{4} + \frac{n}{8}.... = 2n - 1$$

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

$$\frac{n(n+1)(2n+1)}{6} = 1^2 + 2^2 + 3^3 + ... + (n-1)^2 + n^2$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\lg x = k, \text{ where } 2^k = x$$

$$\lg(a^b + = b \lg a$$

$$a^{\lg b} = b^{\lg a}$$

$$\lg(n!) = n \lg(n)$$

II. TIME COMPLEXITY

Just a quick refresher. Featured below is an analysis for Selection Sort.

Inner loop iterations:			Total
0 1 2 3 4 5	...	n-1	n-1
1 2 3 4 5	...	n-1	n-2
2 3 4 5	...	n-1	n-3
3 4 5	...	n-1	n-4
4 5	...	n-1	n-5
5	...	n-1	n-6
	...	n-1	.
	...	n-1	.
	...	n-1	.
	...	n-1	3
	...	n-1	2
	...	n-1	1

This is is $T(n) = \frac{n^2}{2} - \frac{n}{2}$, or $O(n^2)$.

III. ORDER NOTATION

Definition III.1. For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some real positive constant c and some non-negative integer N s.t. $\forall n \geq N$, $g(n) \leq c \times f(n)$
This means, for example, $\{n^2, 5n + 2, n \lg n...\} \in O(n^2)$

Example III.1. Show that $n \in O(n^2)$.
By definition, $n \leq c \cdot n^2$. This is true starting at $c = 1$, $N = 1$: $1 \leq 1 \cdot n$.

Definition III.2. $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exists some real positive constant c and some non-negative integer N s.t. $\forall n \geq N$, $g(n) \geq c \times f(n)$
This means, for example, $\{n^2, n^3, 2^n...\} \in \Omega(n^2)$

Definition III.3. $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$, meaning it is the set of complexity functions $g(n)$ where there exists some real positive constants c, d and some non-negative integer N s.t. $\forall n \geq N$, $c \times f(n) \leq g(n) \leq d \times f(n)$
This means, for example, $\{n^2, 4n^2, n^2 + n \lg n...\} \in \Theta(n^2)$

Definition III.4. $o(f(n))$ is the set of complexity functions $g(n)$ satisfying the following: $\forall c \in \mathbb{R}^+, \exists N \in Z^+$ s.t. $\forall n \geq N$, $g(n) \leq c \times f(n)$.
This means, for example, $\{n, 5n + 2, n \lg n...\} \in O(n^2)$.

Example III.2. Show that $n \in o(n^2)$.
By definition, $n \leq c \cdot n^2$. To find a working expression that conforms to our definition, we note $\frac{1}{c} \leq n$, meaning this holds for any $N \geq \frac{1}{c}$.

Example III.3. Show that $n \notin o(5n)$.
Use a proof by contradiction. Let $c = \frac{1}{6}$. If $n \in o(5n)$, there must exist some N such that, for $n \geq N$, $n \leq \frac{1}{6}5n$. This reduces to $1 \leq \frac{5}{6}$, which cannot be true.

IV. DIVIDE AND CONQUER

Binary Search Recurrence equation: We halve our problem every step, and make one comparison on each level. This is $W(n) = W\left(\frac{n}{2}\right) + 1$ for $n > 1$, n a power of 2. $W(1) = 1$.
The solution is $\lg n + 1$ if n is a power of 2, or $\lfloor \lg n \rfloor + 1$ otherwise.

Mergesort Recurrence equation: We split our array in half, mergesort both halves, and then merge the results together. $W(n) = 1 + 2W\left(\frac{n}{2}\right) + n - 1$. $W(1) = 1$
This is $(\lceil \lg n \rceil + 1)n$, or $\Theta(n \lg n)$.

Quicksort: *(Best-case)*
Recurrence equation: We partition our array on a pivot, quicksort both partitions, and we're done. In the best case, our partition is separates the array such that it is divided in half. $B(n) = n + 2B\left(\frac{n}{2}\right) + 0.\ B(1) = 1$

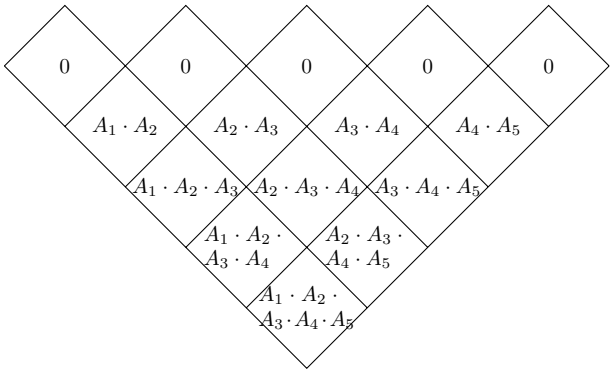
(Worst-case)
Array is sorted in nondecreasing order. We sort the left subarray, then the right, then partition with a cost of: $W(n) = W(0) + W(n-1) + n - 1 = W(n-1) + n - 1.$ $W(1) = 1$
As we keep reducing our problem size by one from $n-1$ to $n-2$, $n-3$..., this is $\frac{n(n-1)}{2}$.

V. DYNAMIC PROGRAMMING

Floyd's Algorithm: I know this one by heart, (un)fortunately. This uses the matrices $D^(0)...D^(n)$ to provide n^2 answers in n^3 time.

Matrix Chain Multiplication: Base form: $A_1[d_0 \times d_1] \cdot A_2[d_1 \times d_2] => R[d_0 \times d_2]$.
Not all orders of multiplication are equivalent. Consider the following:
$A_1[50 \times 10] \cdot A_2[10 \times 30] \cdot A_2[30 \times 20]$
$(A_1 \cdot A_2)A_3 = 50{\times}10{\times}30{+}50{\times}30{\times}20 = 15000 + 30000$
$A_1(A_2 \cdot A_3) = 10{\times}30{\times}20{+}50{\times}10{\times}20 = 6000 + 10000$

We can use dynamic programming in an n^3 algorithm to find the minimum number of multiplications for any given number of matrices. Shown is the setup for matrices $A_1...A_5$.



Sequence Alignment: You know how this works. Pull from the bottom diagonal. Assume a penalty of 1 for a mismatch and a penalty of 2 for a gap, use the dynamic programming algorithm to find an optimal alignment of the following sequences.

As an example from the last homework:

C C G G G T T A C C A
G G A G T T C A

Show both the table and the optimal alignment.

	j	0	1	2	3	4	5	6	7	8	9	10	11
i		C	C	G	G	G	T	T	A	C	C	A	_
0	G	**8**	**6**	5	4	6	7	8	9	11	12	14	16
1	G	9	7	**5**	5	4	6	6	7	9	10	12	14
2	A	11	9	**7**	5	5	4	5	5	7	8	10	12
3	G	12	10	8	**6**	**4**	4	3	4	5	6	8	10
4	T	14	12	10	8	6	**4**	3	2	3	4	6	8
5	T	16	14	12	10	8	6	**4**	3	1	2	4	6
6	C	18	16	14	12	10	8	6	**4**	**2**	**0**	2	4
7	A	20	18	16	14	12	10	8	6	4	2	**0**	2
8	_	22	20	18	16	14	12	10	8	6	4	2	**0**

VI. MINIMUM SPANNING TREES AND GREEDY ALGORITHMS

Spanning Tree For a graph G, it's spanning tree is a connected subgraph that contains all of G's verticies and is a tree.

Minimum Spanning Tree For a graph G, it is a spanning tree of minimum weight.

Prim's Algorithm Builds a minimum spanning tree vertex by vertex using an arbitrary starting vertex. Each step it will add the cheapest possible connection from the current vertex to the next. The algorithm is $\Theta(n^2)$.

Kruskal's Algorithm Builds a minimum spanning tree edge by edge, taking the cheapest edge available unless it causes a cycle. The steps to implement are as follows:

1. Sort the edges in time $\Theta(E \lg(E))$.
2. Loop through the list of edges for a candidate edge. In worst case, it is $\Theta(E)$.

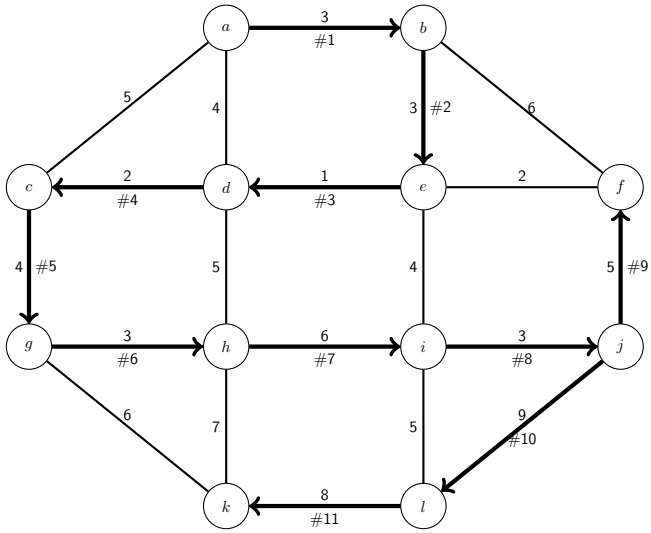
In the worst case, though, every vertex is connected to every other vertex. This means $E \in \Theta(V^2)$. Thus, the worst case complexity is $\Theta((V^2 \lg(E))$.

From the above descriptions, we can quickly conclude that Kruskal's Algorithm is better for sparse graphs, and Prim's Algorithm is better for dense graphs.

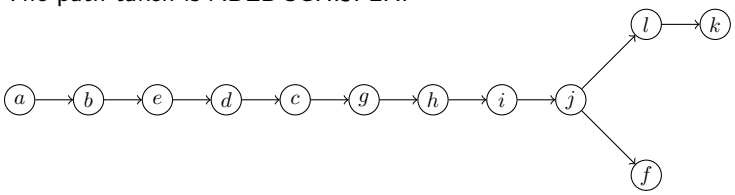
Dijkstra's Algorithm Builds a shortest path from a vertex v_0 to a destination v_n vertex by vertex using an arbitrary starting vertex. Each step it will add the cheapest possible connection from the tree to another vertex, *using the available connections already made in order to reduce possible costs*. The algorithm is $\Theta(n^2)$.

VII. BACKTRACKING

Note that the book explicitly states that backtracking is a modification of DFS. We just DFS a lot.
You remember DFS. Recall the following? *Demonstrate the use of depth first search on this graph starting at vertex A. Draw the search tree and number the vertices in the order they are first encountered.*



The path taken is ABEDCGHIJFLK.



DFS DFS is a $\Theta(V + E)$ traversal of any graph. Pseudocode for the recursive form is as follows.

```
mark the curr vertex, u, visited
enter u in the discovery order set

foreach vertex, v, adjacent
to the curr vertex, u:
  if v !visited:
    Set parent of v to u
    DFS starting at v
mark u finished
enter u in the 'finished' set
```

Backtracking Create a state space tree and traverse it. After determining that a node can only lead to death and destruction, go back to the parent and try again. The only way we can determine this is either by running out of options to come up with a solution, or using an upper bound to state a solution will get no worse. These algorithms are pretty terrible; the $n - queens$ problem uses a $\Theta(n^n)$ solution. The sum-of-subsets problem uses an upper bound and is a $T_n \approx 2^{n+1} - 1$ algorithm

VIII. BRANCH-AND-BOUND

Note that the book explicitly states that backtracking is a modification of *breadth*-first search. We just BFS a lot.

BFS BFS is also a $\Theta(V + E)$ traversal of any graph, using a dequeue.

Branch-and-Bound Just like Backtracking, with the modification that we visit the children of the current node only if its bound (an estimate guessing what will follow the current node) indicates that that path will lead to a better solution than the current best solution. That's the only effective difference. This is a **best**-first search.

IX. MONTE CARLO

Determines the number of nodes on a search tree. Start at a random staring point and solve the problem as we normally would. Repeat until we have either exhausted our search space, or have found a solution.

·	·	Q_1	·	·	·	·	·	·
·	x_1	x_1	x_1	·	·	·	·	Q_2
x_1	Q_3	x_1	·	x_1	·	·	x_2	x_2
x_3	x_3	x_1	·	·	x_1	x_2	Q_4	x_2
Q_5	x_3	x_1	x_3	·	x_2	x_1	x_4	x_2
x_5	x_3	x_1	Q_6	x_2	x_4	·	x_1	x_2
x_5	x_3	x_1	x_2	x_4	x_3	Q_7	x_4	x_1
x_5	x_3	x_1	x_4	Q_8	x_6	x_3	x_4	x_2
x_5	x_2	x_1	x_6	x_5	x_8	x_6	x_3	x_2

Row #	Choices	Cost
1	9	9
2	6	$9 \cdot 6$
3	4	$9 \cdot 6 \cdot 4$
4	3	$9 \cdot 6 \cdot 4 \cdot 3$
5	2	$9 \cdot 6 \cdot 4 \cdot 3 \cdot 2$
6	2	$9 \cdot 6 \cdot 4 \cdot 3 \cdot 2 \cdot 2$
7	1	$9 \cdot 6 \cdot 4 \cdot 3 \cdot 2 \cdot 2 \cdot 1$
8	1	$9 \cdot 6 \cdot 4 \cdot 3 \cdot 2 \cdot 2 \cdot 1 \cdot 1$
9	0	0
Total		**9999**

X. COMPUTATIONAL COMPLEXITY

There are three categories of algorithms

P	Problems with known polynomial-time solutions
NP	Problems whose solutions can be verified in polynomial time (VP)
Intractable	Problems proven to be unsolvable in polynomial time. This can happen in two waays. Either we produce a non-polynomial amount of output or we can prove that the problem takes exponential time to solve.

In the set of problems designated **NP** there are four subcategories

NP-Complete	A problem B is called NP-complete if both of the following are true. 1. $B \in NP$ 2. For every other problem $A \in NP$, $A \propto B$.
NP-Easy	A problem A is called NP-easy if, for some problem $B \in NP$ $A \propto_T B$.
NP-Hard	A problem B is called NP-hard if, for some NP-complete problem A. $A \propto_T B$.
NP-Equivalent	A problem is called NP-equivalent if it is both NP-hard and NP-easy.

In the above we use the operations $\propto$ and $\propto_T$. They are defined below:

$\propto$	If there exists a polynomial-time transformation algorithm from decision problem A, to decision problem B, then problem A is polynomiaal-time many-one reducable to problem B. This is $A \propto B$.
$\propto_T$	If a problem A can be solved in polynomial-time with a hypothetical polynomial-time algorithm for problem B, then problem A is polynomial-time Turing reducable to problem B. This is $A \propto_T B$.

Examples of Problem Transforms:

$P_x \propto P_{\text{easy}}$	P_x is easy
$P_y \propto P_{\text{hard}}$	P_y is unknown
$P_{\text{hard}} \propto P_z$	P_z is hard
$P_{\text{easy}} \propto P_a$	P_a is unknown
$P_1 \propto P_2$ and $P_2 \propto P_1$	P_1 and P_2 are both easy or both hard.
$P_h, \cdots, P_k \propto P_\pi$	If P_π is hard, then the others are unknown. If P_π is easy, then they all are easy. If any of $P_h, \cdots, P_k$ is hard, then P_π is hard.

XI. NP-COMPLETE PROBLEMS

There are a famous set of problems which we have discussed in class for which Boothe is fairly sure there is no polynomial-time solution.

Bin Packing	Given a set of numbers, how do you pack them into the minimum number of bins with a specified capacity.
Subset Sum	Given a set of numbers, using only one bin, how can you pack the bin as close to capacity as possible without overflow
0-1 Knapsack	What is the maximum value you can fit in a pack with a specified weight-limit
Traveling Salesmen	Find the shortest path through a weighted graph while visiting all of the nodes.
Hamiltonian Circuit	Find a complete tour of a graph.
Graph Coloring	Color a graph such that no two adjacent nodes have the same color with the fewest possible colors.
Max Clique	What is the largest set of verticies all directly connected to one another.
Vertex Cover	Find the smallest set of verticies such that every edge in the graph is attached to a vertex in that set.
Satisfiability	Evaluate if a given statement of boolean logic can ever be true.
N-Queens	Find a placement of N queens on an $N \times N$ board so that no queen threatens another queen.

XII. PROBLEM SIZE EXPANSION

In order to solve this problem we must define what the Fibonacci sequence is:

> The Fibonacci sequence, which is discussed in Subsection 1.2.2 is defined as follows:
>
> $$t_n = t_{n-1} + t_{n-2}$$
> $$t_0 = 0$$
> $$t_1 = 1$$

This definition gives us the recurrence equation for calculating the nth term in the Fibonacci sequence, regardless of implementation.

$$t_n = \frac{\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n}{\sqrt{5}}$$

This is clearly not a polynomial. Also, this recurrence equation gives us a representation of the amount of output produced by this algorithm and not the time complexity.

Even though we subtract one exponential from another, as $n \to \infty$ the second term approaches zero as $\frac{1-\sqrt{5}}{2} < 1$. Thus we are left with a single exponential term. This means that the output produced is non-polynomial in n.

The size of the input to this problem is not n, but rather is 2^d if we assume a binary representation scheme for numbers. d is the number of binary digits, which is defined by the text as the input size. Solving $n = 2^d$ for d gives us $d = \lfloor \lg(n) \rfloor + 1$. Thus our equation for the amount of output produced to represent the nth term in the Fibonacci sequence is

$$t_n = \frac{\left[\frac{1+\sqrt{5}}{2}\right]^{2^d} - \left[\frac{1-\sqrt{5}}{2}\right]^{2^d}}{\sqrt{5}}$$

Thus the amount of output produced in representing the nth term in the Fibonacci sequence is non-polynomial with respect to d, where d is the input size. Thus this problem is inherently intractable by definition as it produces a non-polynomial amount of output.

XIII. TRANSFORMATION COMPLEXITY AUGMENTATION

We are given two problems A, B, such that A can be converted to B in $t_n \in \Theta(n^a) \in P$ where P is the set of all polynomials. Thus $A \propto B$. We also note that B is solvable in $t_n \in \Theta(n^b) \in P$. To bound the complexity of this solution, consider the following:

Proof. Suppose we have an instance of problem A that is of size n. Because at most there are $\Theta(n^a)$ steps in the transformation algorithm, and at worst the algorithm outputs a symbol at each step, the size of the instance of B produced by the transformation is at most a quantity s such that $s \in \Theta(n^a)$. When s is the input to the algorithm for B, there are at most $\Theta\left(s^b\right) = \Theta\left((n^a)^b\right) = \Theta\left(n^{ab}\right)$ steps. Therefore, the maximum number of steps required to transform the instance of problem A to an instance of problem B and then solve problem B to get the correct answer for problem A is at most

$$\Theta(n^a) + \Theta\left(n^{ab}\right)$$

which is a polynomial in n. □