CS 170 Final Note Sheet

Formal Limit Definition of $O, \Theta,$ and Ω

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}\begin{cases}\geq 0(\infty) & f(n)\in\Omega(g(n))\\ <\infty(0) & f(n)\in O(g(n))\\ =c, 0<c<\infty & f(n)\in\Theta(g(n))\end{cases}$$

Euclid's GCD: $O(n^3)$

```
def gcd(a,b):
    if b==0:
        return a
    return gcd(b, a mod b)
```

Extended GCD: $O(n^3)$

```
def extended-gcd(a,b):
    if b==0:
        return (1, 0, a)
    (x', y', d) = extended-gcd(b, a mod b)
    return (y', x' - floor(a/b)*y', d)
```

if d divides a and b and $d = ax + by$ for some integers s and y, then d = gcd(a, b)

Multiplicative Inverse

inverse of a,

$$ax \equiv 1(\texttt{mod}\ N)$$

for any $a(\texttt{mod}\ N)$, a has a multiplicative inverse if and only if they are relatively prime, gcd(a,N) = 1

Fermat's Little Theorem

given a prime (or carmichael) p,

$$a^{p-1} \equiv 1(\texttt{mod}\ p)$$

Proof:
Given a set of numbers, $S = \{1, 2, \ldots, p-1\}$ modulo p, there is a 1 to 1 mapping of numbers in S to thier inverses also in S
Then we can also say,

$$S = \{a * 1(\texttt{mod}\ p), a * 2(\texttt{mod}\ p), \ldots, a * (p-1)(\texttt{mod}\ p)\}$$

Multiplying all of these numbers together gives us,

$$(p-1)! \equiv a^{p-1}\cdot(p-1)!(\texttt{mod}\ p)$$

Here we can divide by $(p-1)!$ (which we can do because it is relatively prime to p)

$$a^{p-1} \equiv 1(\texttt{mod}\ p)$$

RSA Euler's Theorem

$$m^{(p-1)(q-1)} = 1(\texttt{mod}\ p)$$

Master's Theorem

If

$$T(n) = aT(\lceil n/b\rceil) + O(n^d)\ \texttt{for}\ a>0, b>1,\ \texttt{and}\ d\geq 0,$$

then,

$$T(n) = \begin{cases} O(n^d) & ifd > log_b a\\ O(n^d logn) & ifd = log_b a\\ O(n^{log_b a}) & ifd < lob_b a\end{cases}$$

Volker Strassen

Divide and conquer matrix multiplication.
Matrix multiplication can be broken into subproblems, because it can be performed blockwise. Ex, we can carve X into for $n/2$ x $n/2$ blocks

$$X = \begin{bmatrix}A & B\\ C & D\end{bmatrix}, Y = \begin{bmatrix}E & F\\ G & H\end{bmatrix}$$

Then the product can be expressed in terms of those blocks as if the blocks were a single element,

$$XY = \begin{bmatrix}A & B\\ C & D\end{bmatrix} * \begin{bmatrix}E & F\\ G & H\end{bmatrix} = \begin{bmatrix}AE+BG & AF+BH\\ CE+DG & CF+DH\end{bmatrix}$$

The runtime of this recurrence is,

$$T(n) = 8T(n/2) + O(n^2) \to O(n^3)$$

Using clever algebra, this can be improved with integer multiplication, XY can be computed from seven $n/2$ x $n/2$ subproblems via decomposition.

$$XY = \begin{bmatrix}P_5 + P_4 - P_2 + P_6 & P_1 + P_2\\ P_3 + P_4 & P_1 + P_5 - P_3 + P_7\end{bmatrix}$$

where,

$$\begin{array}{ll} P_1 = A(F-H) & P_2 = (A+B)H\\ P_3 = (C+D)E & P_4 = D(G-E)\\ P_5 = (A+D)(E+H) & P_6(B-D)(G+H)\\ P_7 = (A-C)(E+F) & \end{array}$$

and has a runtime of

$$T(n) = 7T(n/2) + O(n^2) \to O(n^{log_2 7}) \approx O(n^2)$$

Fast Fourier Transform

complex n^{th} roots of unity are given by $e^{\frac{2\pi i}{n}}$ The Vandermonde Matrix,

$$M_n(\omega) = \begin{bmatrix}1 & 1 & 1 & \cdots & 1\\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1}\\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)}\end{bmatrix}$$

Search Algorithms

Depth First Search

```
def explore(G,v): #Where G = (V,E) of a Graph
    visited(v) = true
    previsit(v)
    for each edge(v,u) in E:
        if not visited(u):
            explore(u)
    postvisit(v)

def dfs(G):
    for all v in V:
        if not visited(v):
            explore(v)
```

Previsit = count till node added to the queue
Postvisit = count till you leave the given node
A directed Graph has a cycle if it has a back edge found during DFS

Breadth First Search

```
def bfs(G,s):
    for all u in V:
        dist(u) = infinity
    dist(s) = 0
    Q = [s] (Queue containing just s)
    while Q is not empty:
        u = eject(u)
        for all edges (u,v) in E:
            if dist(v) = infinity:
                inject(Q,v)
                dist(v) = dist(u) + 1
```

Dijkstra's Algorithm

```
def dijkstra(G,l,s):
    for all u in V:
        dist(u) = infinity
        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) in E:
            if dist(v) > dist(u)+l(u,v)
                dist(v) = dist(u)+l(u,v)
                prev(v) = u
                decreasekey(H,v)
```

Directed Acyclic Graphs

- Every DAG has a source and sink
- 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 directed graph is a DAG of its strongly connected components.
- If the explore subroutine is started at 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
- I C and C' are strongly connected components, and there is an edge from a node in C to a node in C', the the highest post number in C is bigger than the highest post number in C'.

Greedy Algorithms

Kruskal's MST Algorithm $O(E \log v)$

Repeatedly add the next lightest edge that doesn't produce a cycle.

```
Input: A connected undirected graph G = (V,E) with edge
        weights w
Output: A minimum spanning tree defined by the edges X

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

The above algorithm utilizes disjoint sets to determine whether adding a given edge creates a cycle. Basically by checking whether or not both sets have the same root ancestor.

Disjoint Sets Data Structure

Contains a function, "find" that returns the root a given set.

Properties of Trees (undirected acyclic graphs)

- A tree with n nodes has n-1 edges
- Any connected undirected graph G(V,E), with $|E| = |V| - 1$ is a tree
- An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

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.

Prim's Algorithm

(an alternative to Kruskal's Algorithm and similar to Dijkstras) On each iteration, the subtreedefined by x grows by one edge, the lightest between a vertex in S and a vertex outside S.

Huffman Encoding

A means to encode data using the optimal number of bits for each character given a distribution.

```
Huffman(f):
Input: An array f{1...n] of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i=1 to n: insert(H,i)
    i=deletemin(H), j=deletemin(H)
    create a node numbered k with children i,j
    f[k] = f[i]+f[j]
    insert(H,k)
```

Horn Formulas

Horn Formulas are a framework expressing logical facts and deriving conclusions. A Horn Clause is a possible solution to the Formulas. Variables are represented by two kinds of clauses:

1. Implications, whose left-hand side is an AND of any numbers of positive literals and whose right-hand side is a signle positive literal. ("If the conditions on the left hold, then the one on the right mush also be true.")

$$(z \wedge w) \Rightarrow u$$

2. Pure negative clauses, consisting of an OR of any number of negative literals.

$$(\bar{u} \vee \bar{v} \vee \bar{y})$$

The a greedy algorithm to solve a Horn Formula:

```
Input: a Horn formula
Output: a satisfying assignment, if one exists

set all variables to false
while there is an implication that is not satisfied:
    set the right-hand variable of the implication to true
if all pure negative clases are satisfied:
    return the assignment
return 'The formula is not satisfiable.'
```

Set Cover Algorithm

(example. This is the Schools distributed across towns problem.)

```
Input: A set of elements B; sets S1,...,Sm
Output: A selection of the Si whose union is B.

Repeat until all elements of B are covered:
    Pick the set Si with the largest number of
            uncovered elements.
```

Disjoint Sets Data Structure

```
def makeset(x):
    pi(x) = x
    rank(x) = 0

def find(x):
    while x != pi(x):
        x=pi(x)
    return x

def union(x,y):
    rx = find(x)
    ry = find(y)
    if rx == ry:
        return
    elif rank(rx) > rank(ry):
        pi(ry) = rx
    else:
        pi(rx)=ry
        if rank(rx) == rank(ry):
            rank(ry) = rank(ry) + 1
```

Dynamic Programming

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

The following algorithm starts at one side of the list and finds the max length of sequences terminating at that given node, recursively following backlinks. Then given all the lengths of paths terminating at that given node choose the max length. Without memoization, this solution would be exponential time.

```
L = {}
for j=1,2,...,n:
    L[j] = 1+max{L[i]:(i,j) in E}
    # The (i,j) represents all the edges that go from
    # a node to j.
return max(L)
```

Edit Distance (Spelling Suggestions)

This algorithm works by basically choosing the min of the options for every given letter. (The 3 options being adding a gap inbetween letters of one of the strings or matching the two letters and moving on.)
ex) Snowy and Sunny have an edit distance of 3 with this configuration

```
S _ N O W Y
S U N N _ Y

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 $O(nW)$

Items have a weight and a value, you want to maximize the value within a given weight. (The amount you can carry in your knapsack)
With repetition:

$$K(\omega) = max_{\texttt{items}}\{K(\omega - \omega_{\texttt{item}}) + value\}$$

Without repetition:

$$K(\omega, j) = max_{\texttt{available items}}\{K(\omega - \omega_j, j-1) + V_j, k(\omega, j-1)\}$$

Chain Matrix Multiplication $O(n^3)$

$$C(i,j) = min\{C(i,k) + c(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

Floyd-Warshall Algorithm for Shortest Paths $O(|V|^3)$

```
for i=1 to n:
    for j=1 to n:
        dist(i,j,0) = infinity
        for all (i,j) in E:
            dist(i,j,0) = l(i,j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i,j,k) = min{dist(i,k,k-1)+
                        dist(k,j,k-1),dist(i,j,k-1)}
```

Traveling Salesman Problem $O(n^2 2^n)$

```
C({1},1)=0
for s = 2 to n:
    for all subsets S in {1,2,...,n} of size s and has 1:
        C(S,1) = infinity
        for all j in S,j != 1:
            C(S,j) = min{C(S-{j},i)+dij:i in S,i not in j}
return min over j, C({1,...,n},j)+dj1
```

Linear Programming

Properties of Linear Programs

1. To turn a maximization problem into a minimization (or vice versa) just multiply the coeficients of the objective function by -1.
2. To turn an inequality constraint like $\sum_{i=1}^{n} a_i x_i \leq b$ into an equation, introduce a new variable S and use, $\sum_{i=1}^{n} a_i x_i + s > b$, $s \geq 0$ (S is known as a slack variable)
3. To change an inequality constraint into inequalities rewrite $ax = b$,
   as $ax \leq b$ and $ax \geq b$
4. If a linear program has an unbounded value then its dual must be infeasible.

Solving Linear Programs with the Simplex method

typically polynomial time, but in worst case, exponential

```
let v be any vertex of the feasible region
while there is a neighbor v’ of v with a better value:
    set v = v’
return v
```

This is easily seen in a 2d or even sometimes a 3d graph of the constraints

Proving Optimality of a Linear Program Result, Duality

max $x_1 + 6x_2$

Inequality	multiplier
$x_1 \leq 200$	y_1
$x_2 \leq 300$	y_2
$x_1 + x_2 \leq 400$	y_3
$x_1, x_2 \geq 0$	

$(y_1 + y_2)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3$
resulting in,
min $200y_1 + 300y_2 + 400y_3$
$y_1 + y_3 \geq 1$
$y_2 + y_3 \geq 6$
$y_1, y_2, y_3 \geq 0$
Which both result in the same optimum (via simplex) thus proving optimality.

Zero Sum Games

Max Flow Algorithm

Start with zero flow.
Repeat:
Choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible.

Max Flow Min Cut Theorem

The size of the maximum flow in a network equals the capacity of the smallest (s,t)-cut, where and (s,t)-cut partitions the vertices into two disjoint groups L and R such that s (start) is in L and t (goal) is in R.

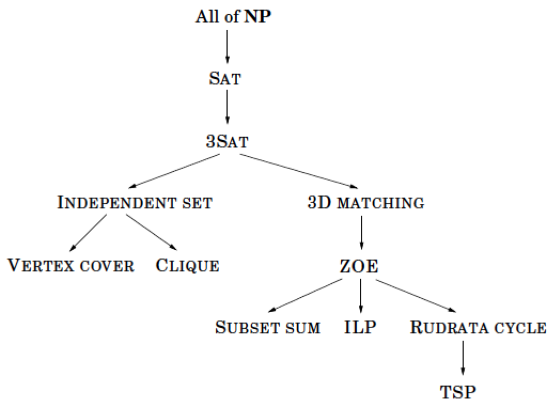
Bipartite Matching

example is given a graph with two sets, Girls and Boys where lines between the sets are who likes who. Find a graph where every Boy and Girl is matched up with someone they like. This problem reduces to a maximum-flow problem solvable by linear programming.

NP-Complete Problems

Hard problems(NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
Traveling Salesman Problem	Minimum Spanning Tree
Longest Path	Shortest Path
3D Matching	Bipartite Matching
Knapsack	Unary Knapsack
Independent Set	Independent Set on trees
Integer Linear Programming	Linear Programming
Rudrata Path	Euler Path
Balanced Cut	Minimum Cut

SAT = Search Algorithm Time


All of NP
SAT
3SAT
INDEPENDENT SET
3D MATCHING
VERTEX COVER
CLIQUE
ZOE
SUBSET SUM
ILP
RUDRATA CYCLE
TSP