

CS 170 Final Note Sheet

Formal Limit Definition of O, Θ , and Ω

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

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 ,

$$ax \equiv 1 \pmod{N}$$

for any $a \pmod{N}$, 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 ,

$$a^{p-1} \equiv 1 \pmod{p}$$

Proof:

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

$$S = \{a * 1 \pmod{p}, a * 2 \pmod{p}, \dots, a * (p-1) \pmod{p}\}$$

Multiplying all of these numbers together gives us,

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

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

$$a^{p-1} \equiv 1 \pmod{p}$$

RSA Euler's Theorem

$$m^{(p-1)(q-1)} = 1 \pmod{pq}$$

Master's Theorem

If

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

then,

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

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 $n/2 \times n/2$ 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,

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

Using clever algebra, this can be improved with integer multiplication, XY can be computed from seven $n/2 \times n/2$ 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,

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

and has a runtime of

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

Fast Fourier Transform

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

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

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
- 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' .

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 coefficients of the objective function by -1.
- 2. To turn an inequality constraint like sum_{i=1}^n a_i x_i <= b into an equation, introduce a new variable S and use, sum_{i=1}^n a_i x_i + s > b, s >= 0 (S is known as a slack variable)
- 3. To change an inequality constraint into inequalities rewrite ax = b, as ax <= b and ax >= 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 x1 + 6x2
Inequality multiplier
x1 <= 200 y1
x2 <= 300 y2
x1 + x2 <= 400 y3
x1, x2 >= 0
(y1 + y2)x1 + (y2 + y3)x2 <= 200y1 + 300y2 + 400y3
resulting in,
min 200y1 + 300y2 + 400y3
y1 + y3 >= 1
y2 + y3 >= 6
y1, y2, y3 >= 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


