1 Prologue

For any algorithm:

1. Is it correct?
2. How much time does it take, as function of n? Big-O Notation
3. Can we do better?

2 Divide-and-Conquer Algorithms

2.1

Divide-and-Conquer Method:

1. Breaking into subproblems that are smaller instances of same type of problem
2. Recursively solve subproblems
3. Appropriately combining solutions

Note: We want to reduce # of subproblems to make efficient

Ex) Can split a digit into

$x = \boxed{x_L} \boxed{x_R} = 2^{n/2} x_L + x_R$ (Karatsuba's Algorithm)

and take advantage of properties in a divide and conquer algorithm for mult

2.2

Master Theorem: problem size n, solve a subproblems of size $\frac{n}{b}$ and combine takes $O(n^d)$

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ constants $a > 0,\ b > 1,\ d \geq 0$

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

2.3

Ex) Mergesort – $O(n \log n)$ sort

- Split list into two halves, sort the half, merge the two sorted sublists

```
func merge(x[1...k], y[1...l]):
    if k=0  return y[1...l]
    if l=0  return x[1...k]
    if x[1] ≤ y[1]
        return x[1] + merge(x[2...k], y[1...l])
    else
        return y[1] + merge(x[1...k], y[2...l])
```

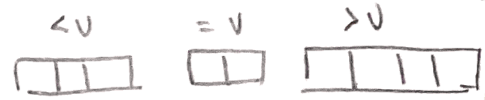
Iterative

```
Q = []
for i=1 to n:
    inject(Q, [a_i])
while |Q| > 1:
    inject(Q, merge(eject(Q), eject(Q)))
return eject(Q)
```

2.4

Find median by divide-and-conquer

1. Select a number v from list S randomly
2. Split list into $<v$, $=v$, $>v$
3. Search can be narrowed down to one of the lists by kth element



find kth element in

randomizing v $\quad T(n) \le T\left(\frac{3n}{4}\right) + O(n) \Rightarrow O(n)$

2.5

Can break down matrix into blocks to simplify matrix multiplication, then use optimization from Volker Strassen

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

2.6 Fast Fourier Transform

Multiplying two polynomials quicker in $O(n\log n)$ time

Degree d polynomial can be determined by $d+1$ distinct points

$$\boxed{\begin{matrix}\text{Coefficient Representation} \\ a_0, a_1, \ldots, a_d\end{matrix}} \underset{\text{Interpolation}}{\overset{\text{Evaluation}}{\rightleftarrows}} \boxed{\begin{matrix}\text{Value Representation} \\ A(x_0), A(x_1) \ldots A(x_{n-1})\end{matrix}}$$

Polynomial Multiplication

1) Selection : Pick points $x_0, \ldots x_{n-1}$ $\quad n \ge 2d+1$
2) Evaluation : Compute $A(x_0) \ldots A(x_{n-1})$ $\quad B(x_0) \ldots B(x_{n-1})$
3) Multiplication : $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$
4) Recover : Recover $C(x) = C_0 + C_1 x + \ldots + C_{2d} x^{2d}$

* FFT converts coefficient rep polynomial to value representation

* FFT^{-1} interpolates equation from value representation to coefficient rep

Fast Fourier Transform

1) Given $A(x)$, split into even and odd terms $A_e(x^2)$, $A_o(x^2)$

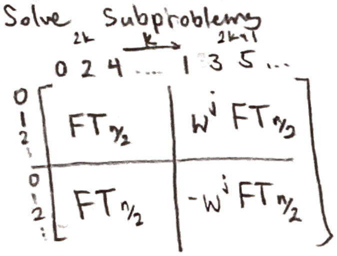
Then $\quad A(x) = A_e(x^2) + x_i A_o(x_i^2)$

$\quad\quad A(-x) = A_e(x^2) - x_i A_o(x^2)$

2) Choose complex nth roots of unity

$w^n = 1$ $\quad n=2: -1, 1$ $\quad n=4: 1, i, -1, -i$ $\quad n=2: \begin{bmatrix} 1 & 1 \\ 1 & w \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix}$

3) Solve Subproblems



$\Rightarrow$

$$\begin{bmatrix} \boxed{W_{n/2}} \begin{bmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{bmatrix} + w^j \boxed{W_{n/2}} \begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} \\ \boxed{W_{n/2}} \begin{bmatrix} a_0 \\ a_2 \\ \vdots \end{bmatrix} - w^j \boxed{W_{n/2}} \begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x + w^j y \\ x_1 + w^j y_1 \\ \vdots \\ x + w^j y \end{bmatrix} \begin{matrix} w_1 = 1 \\ w_2 = i \\ \vdots \\ w = -i \end{matrix}$$

$\Leftarrow$ fill in in order of w

4) Do multiplication in value representation

5) Interpolate back using $M_n(w)^{-1} = \frac{1}{n} M_n(w^{-1})$

3) Decompositions of graphs

3.1

Graphs used for variety of problems, can be represented w/ adjacency matrix or adjacency list

3.2

Depth First Search is linear time algorithm that finds parts of graph that are reachable from vertex

procedure explore(G, v):
Input: $G = (V, E)$ graph, $v \in V$
Output: visited(u) is true for nodes reachable
    visited(v) = true
    previsit(v)
    for each edge $(v, u) \in E$:
        if not visited(u): explore(G, u)
    postvisit(v)

procedure dfs(G)
    Set all $v \in V$ visited(v) = false
    for all $v \in V$:
        if not visited(v): explore(G, v)

Run time: $O(|V| + |E|)$

We can set pre/post numbers in graph by

procedure previsit(v)
    pre[v] = clock
    clock += 1

procedure postvisit(v)
    post[v] = clock
    clock += 1

3.3

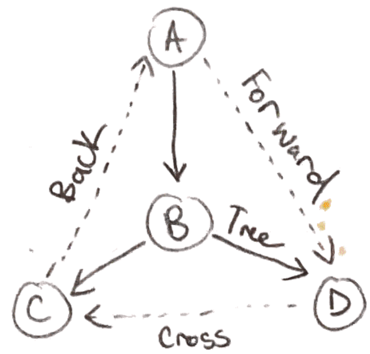
Edge Types:

Tree edge: part of DFS tree

Forward edge: from node to non child descendant in tree

Back edge: lead to an ancestor in tree

Cross edges: neither descendant or ancestor



pre/post ordering (u,v)

$[_u\ [_v\ ]_v\ ]_u$	Tree / forward
$[_v\ [_u\ ]_u\ ]_v$	Back
$[\ ]\ [\ ]$	Cross

Properties

- Directed graph has cycle iff DFS reveals a backedge
- DAG can be linearized by performing in decreasing order of post numbers
- DAG has at least one source and one sink

highest post number is source, lowest is sink

(3.4)

In directed graphs we have strongly connected components where it is only connected if there is a path from $u \to v$ & $v \to u$

- We can turn strongly connected components into a meta node and change any graph into a DAG

Properties

- If explore started at node v, terminates well all nodes reachable from v have been visited
- Node with highest post number must be in strongly connected component
- If C and C' are SCCs and edge from C to C', highest post in C > highest post in C'

To find SCC:

1. Run DFS on reversed G^R to find sink component
2. Run undirected connected component algo (in previsit set a cc[v] = count) on G and in DFS process nodes in decreasing order of post numbers in step 1

Runtime: $O(|V|+|E|)$

4 Paths in Graphs

(4.1)
Difference between two nodes is the length of shortest path between them

(4.2)
Breadth First Search by using queue instead of a stack

Search by level away

procedure BFS (G, s):

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

Runtime: $O(|V| + |E|)$

4.4

Dijkstra's Algorithm works on graphs with non-negative weighted edges by using priority queue and exploring by next shortest path

---

procedure dijkstra(G, d, s):

Input: Graph $G = (V, E)$ directed or undirected, positive edge lengths $\{l_e : e \in E\}$, $s \in V$

Output: For all vertices u reachable from s, dist(u) set to distance s to u

```
for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0
H = makeQueue(V)                    dist value 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 = u
            decreasekey(H, v)
```

Runtime: Binary Heap: $O((|V| + |E|) \log |V|)$

Bellman-Ford algorithm allows us to find shortest path on graph with negative weights by updating all edges $|V|-1$ times

procedure shortest-paths (G, l, s)
Input: graph G edge length $\{l_e : e \in E\}$ no negative cycles, vertex $s \in V$

for all $u \in V$
  $dist(u) = \infty$, $prev(u) = nil$
$dist(s) = 0$
repeat $|V|-1$ times
  for all $e \in E$:
    update (e)

procedure update $((u,v) \in E)$
  $dist(v) = \min\{dist(v), dist(u) + l(u,v)\}$

Runtime: $O(|V| + |E|)$

1.) Update gives correct distance if u is 2nd to last node in shortest path
2.) Never makes $dist(v)$ too small, safe

If there is a negative cycle in the graph, there is no shortest path
To check, perform one last check to see if anything changes in all edges

We can also find shortest paths in dags in linear time by doing a topological sort

procedure dag-shortest-paths (G, l, s)
Input: Dag $G = (V, E)$
  edge lengths $\{l_e : e \in E\}$; vertex $s \in V$
Output: $dist(u)$ is distance from s to u

for all $u \in V$:
  $dist(u) = \infty$
  $prev(u) = nil$
$dist(s) = 0$
Linearize G
for each $u \in V$, in linearized order
  for all edges $(u,v) \in E$:
    update (u,v)

5 Greedy Algorithms

Greedy Algorithms choose next step that offers most obvious and immediate benefit, leading to locally optimal choices.

- Works for problems where making locally optimal choice leads to global optimum

Minimum Spanning Tree (MST)

Def: tree w/ min total weight on graph
Input: undirected Graph $G = (V,E)$; edge weights w_e
Output: A tree $T = (V, E')$ with $E' \subseteq E$ minimizing weight$(T) = \sum_{e \in E'} w_e$

Cut Property: On MST X with subset nodes S, for $V-S$ any lightest edge between S and $V-S$, e, is part of some MST. "Always safe to add lightest edge across any cut"

Kruskal's Algorithm:

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

```
procedure Kruskal(G,w)
  Input: connected, undirected graph G=(V,E) edge weights We
  Output: A minimum spanning tree defined by edges X
  for all u ∈ V:
     make set(u)
  X = {}
  Sort 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)
```

Runtime: $O(|E| \log |V|)$

Prim's Algorithm:

Intermediate set forms subtree and grows by one each iteration

```
procedure prim(G, w):
    for all u ∈ V:
        cost(u) = ∞
        prev(u) = nil
    Pick any initial node u₀
    cost(u₀) = 0

    H = makequeue(V)
    while H is not empty:
        v = deletemin(H)
        for each {v, z} ∈ E
            if cost(z) > w(v, z):
                cost(z) = w(v, z)
                prev(z) = v
                decreasekey(H, z)
```

Runtime: $O(|E| \log |V|)$

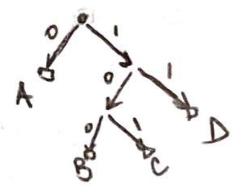
5.2

Huffman Encoding

Want to encode data efficiently

- Use variable length encoding, prefix free, binary tree to represent
- Symbols w/ smallest freq at bottom of tree,

letter	encoding
A	0
B	100
C	101
D	11




```
procedure huffman(f):
    Input: An array f[1,...n] of frequencies
    Output: An encoding tree w/ n leaves

    let H be a priority queue of integers, ordered by f
    for i=1 to n: insert(H, i)
    for k=n+1 to 2n-1
        i = deletemin(H), j = deletemin(H)
        create node numbered k w/ children i, j
        f[k] = f[i] + f[j]
        insert(H, k)
```

Runtime: $O(n \log n)$

Set Cover

Given set of points, want to find smallest num of subsets to cover all points

Set Cover Algorithm:

Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$

Output: A selection of the S_i whose union is B

Cost: number of sets picked

Repeat until all elements of B are covered

Pick the set S_i w/ largest number of uncovered elements

If optimal k sets, greedy algo returns at most $k \ln n$ sets

6) Dynamic Programming

Solve problem by finding subproblems, tackling them smallest first and using answers from smaller problems to solve larger ones

Shortest Paths in DAGs

Compute in single pass

initialize all dist(·) values to 0

$dist(s) = 0$

for each $v \in V \setminus \{s\}$, in linearized order:

$dist(v) = \min_{(u,v)\in E} \{dist(u) + l(u,v)\}$

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

find longest increasing subsequence in list

for $j = 1, 2, \ldots, n$:

$L(j) = 1 + \max\{L(i) : (i,j) \in E\}$

return $\max_j L(j)$

Min Edit Distance : $O(mn)$

Edit distance between two words, remove, add, change

for $i = 0, 1, 2, \ldots m$:

$E(i,0) = i$

for $j = 1, 2, \ldots n$:

$E(0,j) = j$

for $i = 1, 2, \ldots m$:

for $j = 1, 2, \ldots n$:

$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1) + diff(i,j)\}$

return $E(m,n)$

Runtime: $O(mn)$

Knapsack : $O(nW)$

Bag fits at most W weight each object weight $w_1, w_2 \ldots w_n$ dollar value $v_1, \ldots v_n$

① Unlimited quantity : subproblem by knapsack capacity w

$K(0) = 0$

for $w = 1$ to W:

$K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$

return $K(W)$

② No repetition : capacity w and keep track of items left

Initialize all $K(0,j) = 0$ and $K(w,0) = 0$

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 $K(W,n)$

All pairs shortest paths

Find shortest paths between all s,t

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

Start from one node expand set of intermediate nodes

for i=1 to n:
  for j=1 to n:
    dist(i,j,0) = ∞

for all (i,j) ∈ 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)}

Independent sets in trees : $O(|V|+|E|)$

Find independent set where no edges between nodes

I(u) = size of largest independ set from u

Either includes or doesn't

$$I(u) = \max\left\{1 + \sum_{\substack{\text{grandchildren} \\ w \text{ of } u}} I(w),\ \sum_{\substack{\text{children } + \\ w \text{ of } u}} I(w)\right\}$$

if independent set, can't include children.

7) Linear Programming and reductions

Optimization tasks where constraints and optimization criterion are linear functions

Optimum typically achieved at a vertex of feasible region, or infeasible (too tight constraints), unbounded (too loose)

- Can be solved with Simplex method to start at vertex repeatedly looking for better obj value

Primal

max $c_1x_1 + \dots + c_nx_n$

$a_{i1}x_1 + \dots + a_{in}x_n \le b_i$ $i \in I$

$a_{i1}x_1 + \dots + a_{in}x_n = b_i$ $i \in E$

$x_j \ge 0$ $j \in N$

max c^Tx
$Ax \le b$
$x \ge 0$

Dual

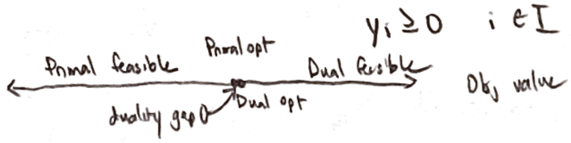
min $b_1y_1 + \dots + b_my_m$

$a_{1j}y_1 + \dots + a_{mj}y_m \ge c_j$ $j \in N$

$a_{1j}y_1 + \dots + a_{mj}y_m = c_j$ $j \notin N$

$y_i \ge 0$ $i \in I$

min y^Tb
$y^TA \ge c^T$
$y \ge 0$



Constraint Transformations:

① Changing Objective:
max c^Tx = min $-c^Tx$
min c^Tx = max $-c^Tx$

② Inequality to Equality
$ax \le b \to ax + s = b,\ s \ge 0$

③ Equality to Inequality
$ax = b \to ax \le b,\ ax \ge b$

④ Unrestricted variable
$x \in \mathbb{R} \to x = x_+ - x_-$
$x_+, x_- \ge 0$

Ex) Chocolates to max profit

Objective: max $x_1 + 6x_2$

Constraint: $x_1 \le 200$
$x_2 \le 300$
$x_1 + x_2 \le 400$
$x_1, x_2 \ge 0$

LP: use Simplex

$(x_1, x_2) = (100, 300)$

Dual:

Multiplier	Inequality
y_1	$x_1 \le 200$
y_2	$x_2 \le 300$
y_3	$x_1 + x_2 \le 400$

$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \le 200y_1 + 300y_2 + 400y_3$

min $200y_1 + 300y_2 + 400y_3$
$y_1 + y_3 \ge 1$
$y_2 + y_3 \ge 6$ $y_1, y_2, y_3 \ge 0$

$(y_1, y_2, y_3) = (0, 5, 1)$

$5x_2 \le (300)5$
$x_1 + x_2 \le 400$

$x_1 + 6x_2 \le 1900$

$(x_1, x_2) = (100, 300)$
$100 + 6(300) \le 1900$

Zero Sum Games

Can represent some situations with matrix games with payoff matrix
- row wants to maximize and col to minimize
- Each player can have a mixed strategy to decide each play with probability x_i
- Expected (avg) payoff is $\sum_{i,j} G_{ij} \cdot \text{Prob}[\text{row plays } i, \text{col plays } j]$

Column

row	r	p	s
r	0	-1	1
p	1	0	-1
s	-1	1	0

In cases, where opponent move is known, want to play defensively

For row:

(x_1, x_2) to max $\underbrace{\min\{3x_1 - 2x_2, -x_1 + x_2\}}_{\text{columns best response}}$

	m	t
e	3	-1
s	-2	1

$z = \min\{3x_1 - 2x_2, -x_1 + x_2\}$

max z
$z \le 3x_1 - 2x_2$
$z \le -x_1 + x_2$

max z

$$-3x_1 + 2x_2 + z \le 0$$
$$x_1 - x_2 + z \le 0$$
$$x_1 + x_2 = 1$$
$$x_1, x_2 \ge 0$$

Both have same optimum

For column. $\min \max\{3y_1 - y_2, -2y_1 + y_2\}$

min w

$$-3y_1 + y_2 + w \ge 0$$
$$2y_1 - y_2 + w \ge 0$$
$$y_1 + y_2 = 1$$
$$y_1, y_2 \ge 0$$

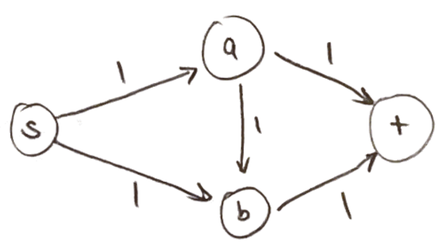
$$\max_x \min_y \sum_{i,j} G_{ij} x_i y_j = \min_y \max_x \sum_{i,j} G_{ij} x_i y_j$$

Max Flow

For graph, want to send as much flow through, each edge has capacity
- from source s to target t
- The shipping scheme is flow w/ f_e on every edge, must:

1. Doesn't violate edge capacities: $0 \le f_e \le c_e$ for all $e \in E$
2. All nodes u except s, t flow entering equals flow leaving

flow conserved $\sum_{(w,u) \in E} f_{wu} = \sum_{(u,z) \in E} f_{uz}$



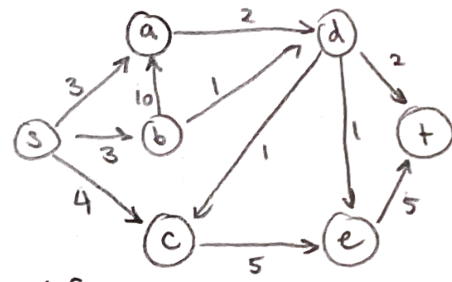
Size of flow = quantity leaving s $\sum_{(s,u) \in E} f_{s,u}$

Ford Fulkerson Algorithm

Start with zero flow
Find path from s to t and route max flow through
Update capacities by updating residual graph
Repeat until no more paths s to t

Residual Graph

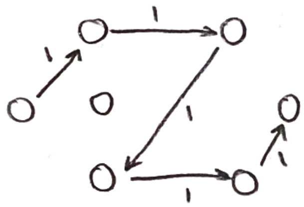
Graph



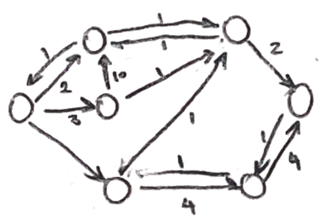
Two types of edges   residual capacity c^f.

$$c^f_{uv} = \begin{cases} c_{uv} - f_{uv} & \text{if } (u,v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (v,u) \in E \text{ and } f_{vu} > 0 \end{cases}$$

flow



residual



forward: capacity left

back: flow through edge

For any (s,t) cut on graph, $\text{size}(f) \leq \text{capacity}(L, R)$
capacity of cut

Max-flow min-cut theorem

Size of max flow in a network equals capacity of smallest (s,t) cut

Max flow creates residual graph, using graph, find min cut by finding vertices(L) reachable from s and V−L separation is the min cut in graph

Runtime: $O(|V| \cdot |E|^2)$