Big-O-Notation

$f(n) = O(g(n))$ $\exists c>0, n_0 \in \mathbb{Z}^+$ s.t. $\forall n \geq n_0$, $0 \leq f(n) \leq c \cdot g(n)$

$f(n) = o(g(n))$ $\forall c>0, \exists n_0 \in \mathbb{Z}^+$ s.t. $n \geq n_0$, $0 \leq f(n) < c \cdot g(n)$

$f(n) = \Omega(g(n))$ $\exists c>0, n_0 \in \mathbb{Z}^+$ s.t. $\forall n \geq n_0$, $0 \leq c \cdot g(n) \leq f(n)$

$f(n) = \omega(g(n))$ $\forall c>0, \exists n_0 \in \mathbb{Z}^+$ s.t. $n \geq n_0$, $f(n) > c \cdot g(n)$

$f(n) = \Theta(g(n))$ $\exists c_1, c_2 > 0, n_0 \in \mathbb{Z}^+$, $\forall n \geq n_0$, $c_1 \cdot g(n) \leq f(n) \leq c_2 g(n)$

Asymptotic Limit Rules

$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$ $\quad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some $c>0 \Rightarrow f(n) = \Theta(g(n))$

Recursive Fibonacci

$T[n] = T[n-1] + T[n-2] + \left(\begin{array}{c}\text{Addition}\\ \& \\ \text{Returning}\end{array}\right)$

Claim1: $T[n] \geq \Omega\left(\left(\frac{3}{2}\right)^n\right)$

Inefficient: calculate the same quantity.

By induction we show $T[n] > \left(\frac{1}{4}\right)\left(\frac{3}{2}\right)^n$ for $n=1,2$. $T[1], T[2]$ are ≥ 1

$\therefore T[n] \geq T[n-1] + T[n-2]$

$\geq \frac{1}{4}\left(\frac{3}{2}\right)^{n-1} + \frac{1}{4}\left(\frac{3}{2}\right)^{n-2}$

$\geq \frac{1}{4}\left(\frac{3}{2}\right)^{n-2}\left(\frac{3}{2}+1\right) > \frac{1}{4}\cdot\left(\frac{3}{2}\right)^n$

Iterative Fibonacci: for: $f[i] = f[i-1] + f[i-2]$

$T[n] = O\left(n \times \left(\begin{array}{c}\text{time for}\\ 1 \text{ addition}\end{array}\right)\right)$

How many bits in n^{th} Fibonacci? Fact: $2^n > F_n > \left(\frac{3}{2}\right)^n$

Integer Multiplication

Grade School Algorithm

Input: $a[1\cdots n]$ $b[1\cdots n]$ n digit numbers

Output: $C[1\cdots 2n] = a \times b$ $2n$ digit numbers

Runtime: Adding n-digit numbers n times $\Rightarrow \Theta(n^2)$

Algorithm 2:

$a = (10)^{\frac{n}{2}} a_L + a_R$

$b = (10)^{\frac{n}{2}} b_L + b_R$

a_L	a_R	b_L	b_R

$a \times b = (10^n) a_L b_L + 10^{\frac{n}{2}} (a_L b_R + a_R b_L) + a_R b_R$ products of $\frac{n}{2}$ digits

```
MULT(a[1···n] b[1···n])
  IF n<2 RETURN a[1]*b[1]
  SPLIT a → a_L, a_R   b → b_L, b_R
  P1 ← MULT(a_L, b_L)   P2 ← (a_L, b_R)
  P3 ← MULT(a_R, b_R)   P4 ← (a_R, b_L)
  RETURN 10^n P1 + 10^(n/2) (P2+P3) + P4
```

$T[n] = 4T[\frac{n}{2}] + cn$

$= \Theta(n^2)$

KARATSUBA's Algorithm

$a \times b = 10^n a_L b_L + 10^{\frac{n}{2}} \underbrace{(a_L + a_R)(b_L + b_R)}_{P_3} - \underbrace{a_L b_L}_{P_1} - \underbrace{a_R b_R}_{P_2} + \underbrace{a_R b_R}_{P_2}$

```
MULT(a[1···n] b[1···n])
  IF n<2 RETURN a[1]*b[1]
  SPLIT: a → a_L, a_R   b → b_L, b_R
  P1 ← MULT(a_L, b_L)   P2 ← MULT(a_R, b_R)
  P3 ← MULT(a_L + a_R, b_L + b_R)
  RETURN 10^n P1 + 10^(n/2) [P3 - P1 - P2] + P2
```

$T[n] = 3T[\frac{n}{2}] + cn$

Solving Recurrence

$T[n] = T[n-1] + \sqrt{n}$:

$T[n] = T[1] + \sqrt{2} + \sqrt{3} + \cdots + \sqrt{n}$

Each term $< \sqrt{n} \Rightarrow T[n] < n\sqrt{n} = n^{\frac{3}{2}}$

Look at last $\frac{n}{2}$ terms

$T[n] \geq \sqrt{\frac{n}{2}} + \sqrt{\frac{n}{2}+1} + \cdots + \sqrt{n}$

$\geq \frac{n}{2} \cdot \sqrt{\frac{n}{2}} = \frac{n^{\frac{3}{2}}}{2\sqrt{2}}$

$\therefore n^{\frac{3}{2}} \geq T[n] \geq n^{\frac{3}{2}}/2\sqrt{2}$

$T[n] = 3T[n^{\frac{1}{3}}] + O(\log n)$

let $x = \log n$

$\therefore n^{\frac{1}{3}} = e^{\frac{1}{3}x}$

$T(e^x) = 3T(e^{\frac{1}{3}x}) + O(x)$

let $S(x) = 3T(e^x) = T(n)$

$\therefore S(x) = 3S(\frac{x}{3}) + O(x)$

$\therefore T(n) = S(x) = O(x \log x)$

$= O(\log n \log\log n)$

$T(n) = T(n-1) + T(n-2)$

$\because T(n) \geq 2T(n-2)$ $\therefore T(n) = \Omega(2^{\frac{n}{2}})$

$T(n) \leq 2T(n-1)$ $\therefore T(n) = O(2^n)$

Guess $T(n) = 2^{\Theta(n)}$ more detailedly $T(n) = \Theta(\alpha^n)$

$\therefore \alpha^n = \alpha^{n-1} + \alpha^{n-2} \Rightarrow \alpha^2 = \alpha + 1 \Rightarrow \alpha = \frac{1+\sqrt{5}}{2}$

$\therefore T(n) = \Theta((\frac{1+\sqrt{5}}{2})^n)$

Master Theorem

$T[n] = aT(\frac{n}{b}) + O(n^c)$

Case 1: $c < \log_b a$: $T[n] = O(n^{\log_b a})$

Case 2: $c = \log_b a$: $T[n] = O(n^c \log n)$

Case 3: $c > \log_b a$: $T[n] = O(n^c)$

Matrix Multiplication

Naive Algo: compute z_{ij}s using $\Theta(n)$ times seperately

Runtime: $n^2 \cdot \Theta(n) = \Theta(n^3)$

Divide and Conquer:

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

$T[n] = 8T[\frac{n}{2}] + \Theta(n^2) \Rightarrow T[n] = \Theta(n^3)$

Strassen's Algo:

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

$P_1 = A(F-H)$ $P_2 = (A+B)H$ $P_3 = (C+D)E$

$P_4 = D(G-E)$ $P_5 = (A+D)(E+H)$

$P_6 = (B-D)(G+H)$ $P_7 = (A-C)(E+F)$

$XY = \begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_4-P_3-P_7 \end{bmatrix}$ $T[n] = 7T[\frac{n}{2}] + \Theta(n^2)$

Median

Naive: Sort the list and output $\Theta(n \log n)$

Randomised Algo run in $O(n)$ time Median$(a_1 \cdots a_n)$

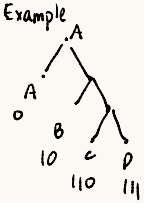
```
SELECT([a_1 ··· a_n], k)                    = SELECT([a_1 ··· a_n], ⌈n/2⌉)
PIVOT v ← random element from {a_i}         RUNTIME see 4th column.
Split S_< ← {a_i | a_i < v}   S_= ← {a_i | a_i = v}   S_> ← {a_i | a_i > v}
Cases: k ≤ |S_<| : return SELECT(S_<, k)                    O(n^2) worst
       |S_<| < k ≤ |S_<| + |S_=| : return v                 O(n) best
       |S_<| + |S_=| < k  return SELECT(S_>, k - |S_<| - |S_=|)
```

Huffman Codes

Prefix-freeness: No code can be prefix of another.

```
Huffman(f)
  H = priority queue
  For i=1···n : Insert(i, f[i])
  For k = n+1 ··· 2n-1
    i = Delmin(H)   j = Delmin(H)
    create node k with children i, j
    f[k] = f[i] + f[j]
    Insert(k, f[k]).
```

Example



Running Time $O(n \log n)$

Claim: $\exists$ an optimal tree T where f_i, f_j (the lowest two frequencies) are the deepest siblings.

Proof:

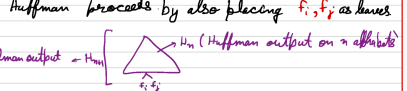
- Let f_u be the deepest node in an optimal tree O
- Since the tree is a full tree f_u has a sibling f_v
- The tree has O has f_i and f_j somewhere
- Swap f_i with f_u & f_j with f_v to get a new tree M
- Otherwise that $cost(M) \leq cost(O)$ as $f_i \leq f_u$ & $f_j \leq f_v$
- However, O is optimal & $cost(O)$ is the smallest
- Thus, $cost(M) = cost(O)$ □

Lemma: Huffman gives the optimal tree

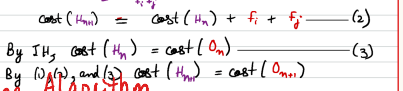
$n=2$:

$n \to n+1$:

- By claim on last slide, $\exists$ an optimal solution O_{n+1} with f_i, f_j as the deepest leaves.
- $cost(O_{n+1}) = cost(O_n) + f_i + f_j$ ——— (1)



- Huffman proceeds by also placing f_i, f_j as leaves



- $cost(H_{n+1}) = cost(H_n) + f_i + f_j$ ——— (2)
- By IH, $cost(H_n) = cost(O_n)$ ——— (3)
- By (1), (2), and (3) $cost(H_{n+1}) = cost(O_{n+1})$

Task Scheduling Algorithm

Input: n jobs with start time and end time $[s_i, t_i]$

Claim: first time finish is optimal

Proof: Base Case: $i=1$. obviously true $H_0 \equiv$ optimal

Inductive Step: Assume for $n=k$. $S_i = S_i'$, $t_i = t_i'$ for $i \leq k$

for $n=k+1$, $t_k \leq S_{k+1} < t_{k+1} \leq t_{k+1}' < S_{k+2}'$

$\therefore [S_{k+1}', t_{k+1}']$ can be replaced by $[S_{k+1}, t_{k+1}]$ without loss of optimality. proved.

```
Algo   Sort jobs by finish time                    → O(n log n)
       A = ϕ   t^A = -∞
       for j = 1 to n
           if t^A < s_j
               A = A ∪ {[s_j, t_j]}                → O(n)
               t^A = t_j
       Return A
```

Dijkstra's Algorithm

```
dist[u] ← ∞  ∀u ∈ V   dist[s] ← 0
Q ← makequeue(V, dist)
while Q is not empty
    u ← Q.deletemin( )
    for all edge u → v
        if dist[v] > dist[u] + W_uv
            dist[v] = dist[u] + W_uv
            Q.decreasekey(v, dist[v])
```

|V| deletemin
|E| decreasekey
$= \Theta((|V|+|E|)\log V)$
using binary heap

Bellman-Ford Algorithm

Repeatedly update all edges in some order until there are no structured bonds

Define: $D[\text{vertex } v, \text{hop } h]$ = length of shortest path that uses at most h hops (edge on the path)

```
Bellman-Ford Algo(G, W_e, s)
  D[v,0] ← ∞  ∀v ∈ V   D[s,0] ← 0
  for h=1 to |V|-1
    D[v,h] ← D[v,h-1]  ∀ v ∈ V
    for each edge u → v ∈ E
        D[v,h] = min(D[v,h], D[u,h-1] + W_u→v)
  return D[v, |V|-1]  ∀ v
```

2-d array → 1-d array.

```
Bellman-Ford Algo(G, W_e, s)
  dist[v] ← ∞  ∀v ∈ V   dist[s] ← 0
  for h=1 to |V|-1
    for each edge u → v ∈ E
        dist[v] = min(dist[v], dist[u] + W_u→v)
  return dist[:]
```

Runtime: $O(|V| \cdot |E|)$ Space: $O(V)$

RUNTIME For SELECT

$T[n]$ = expected runtime of Select$[a_i]$

Good pivot	
$\frac{n}{4}$	$\frac{3n}{4}$

$= ($expected runtime before good pivot$) + ($expected runtime after good pivot$)$

$\leq ($expected # of pivots before 1st good pivot$) \cdot n + ($expected runtime after good pivot$)$

$=$ (expected # of tosses before HEADS) $= 2$

$\leq 2n + T(\frac{3}{4}n)$ $\quad T(n) \leq T(\frac{3}{4}n) + O(n)$

$\Rightarrow T(n) = \Theta(n)$

Graphs Representation

	Adjacency Matrix	Adjacency Lists
size	$\Theta(n^2)$ space	$\Theta(n+m)$ space
$(u,v)\in E$?	$\Theta(1)$ time	$\Theta(\deg(u))$ time
Enumerate all neighbor of u	$\Theta(n)$ time	$\Theta(\deg(u))$ time

DFS

```
visited[u]=FALSE ∀u∈V.
EXPLORE (vertex v)
  visited[v]=True
  for each edge(v,w) ∈ E
    if not visited[w]
      EXPLORE(w)
```

```
DFS(GRAPH G)
  visited[u]=FALSE ∀u∈V
  for each vertex v
    IF NOT visited[v]:
      EXPLORE(v).
```

	List	Matrix
Time complexity	$O(V+E)$	$O(V^2)$
Space complexity	$O(V+E)$	$O(V^2)$

Connected Components in An Graph

```
EXPLORE (vertex v)
  visited[v]=True
  ccnum[v]: count
  for each edge(v,w) ∈ E
    if not visited[w]
      EXPLORE(w)
```

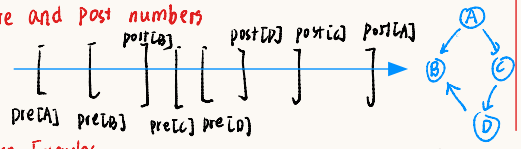
```
DFS(GRAPH G)
  visited[u]=FALSE ∀u∈V
  count=1
  int array ccnum[n]
  for each vertex v
    IF NOT visited[v]:
      EXPLORE(v).
      count=count+1
```

DFS in Directed Graph

```
EXPLORE (vertex v)
  visited[v]=True
  pre[v] = clock
  clock = clock+1
  for each edge(v,w) ∈ E
    if not visited[w]
      EXPLORE(w)
  post[v] = clock
  clock = clock+1
```

```
DFS(GRAPH G)
  visited[u]=FALSE ∀u∈V
  clock = 0
  int array pre[n] post[n]
  for each vertex v
    IF NOT visited[v]:
      EXPLORE(v).
```

Pre and post numbers



Horn Formulas

pure negative clause $(\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n)$

implication clause $(\neg x_1 \vee \neg x_2 \vee \cdots \vee x_n)$ $\quad (w \wedge \lambda) \Rightarrow x \Leftrightarrow (\neg w \vee \neg \lambda \vee x)$ $\quad x \Rightarrow y \Rightarrow (\neg x \vee y)$

$F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$? satisfy

Edge: $u \to v$

$[\,[\,]\,]$ (u, v, v, u) = Tree or Forward Edge

$[\,[\,]\,]$ (v, u, u, v) = Back Edge $\quad [\,]\,[\,]$ (v, v, u, u) = Cross Edge

$[\,]\,[\,]$ (u, u, v, v) , $[\,[\,]\,]$ (u, v, v, u) = Impossible

Observation: $post[u] < post[v]$ iff $u \to v$ is a back edge.

Topological Sort

Algorithm: ① Run DFS to compute Pre & Post value

② sort vertices in decreasing post values.

DAG has NO back edges. $\Rightarrow \forall u \to v$, $post[u] > post[v]$.

Connectivity

Strongly connected: $\exists$ a path $u \to v$ & $v \to u$

FACT: Every Directed Graph can be decomposed as a DAG of SCCs

Kosaraju's Algorithm

Input: Directed Graph $G(V,E)$

Output: Decompose G into DAG of SCCs

1) $G_R \leftarrow G$ with edges reversed.

2) $pre_R[v]$, $post_R[v]$ for all $v \in V \leftarrow DFS(G_R)$

3) DFS on G exploring in decreasing $post_R$ order.

```
a) visited[v]=FALSE ∀v∈V
b) count=0
   ccnum[1...n]: int array
c) for vertices v in
     decreasing post_R order:
     if NOT visited[v]
       explore[v]
       count = count+1
```

```
Explore[v]:
  visited[v]=True
  ccnum[v]=count
  for each edge v→w do
    if NOT visited[w]
      EXPLORE(w).
```

BFS

Input: G Output: $\forall v \in V$, $dist[v]$

```
dist[v] ← ∞  ∀v∈V
dist[s] = 0
Q ← queue with {s}
while Q NOT EMPTY
  v ← eject(Q)
  for all edges v→w
    if (dist[w]==∞)
      dist[w] = dist[v]+1
      Q.add(w)
```

Time complexity:
- List: $O(V+E)$
- Matrix: $O(V^2)$

Space Complexity:
- List: $O(V+E)$
- Matrix: $O(V^2)$

MST Properties

1: Removing a cycle edge in a connected graph does not disconnect it.

2: A tree with n node has $n-1$ edges

3: A connected graph on n nodes & $n-1$ edges is a tree.

MST Greedy Approach

Add the least weight edge does not introducing a cycle; and iterate

Main Theorem: (i) let $X \subseteq E$ be part of T

(ii) $S \subseteq V$ be a set s.t. no edges in X from S to $V-S$

(iii) let $e \in E$ be the lightest edge from S to $V-S$

Case I: $e \in T$. done

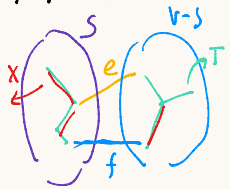
Case II: $e \notin T$

$T' = T + e - f \quad w_f \geq w_e$

(i) By 1. T' is connected

(ii) By 3. T' is a tree

$cost(T') = cost(T) + w_e - w_f \leq cost(T) \Rightarrow X \cup e \subseteq T'$ (a MST)



Kruskal's Algorithm

① Sort Edges in order of increasing weights

② Add if doesn't introduce a cycle and skip otherwise

```
For all u∈V, makeset(u)
X = ϕ
Sort edges in E by w
∀(u,v)∈E in that order
  if find(u) ≠ find(v)
    X = X ∪ {(u,v)}
    union(u,v)
return X
```

$|E|\log|V|$ sorting

$|V|\times$ makeset $\to O(1)$

$2|E|\times$ Find $\to O(\log|V|)$

$|V|-1\times$ unions $\to O(\log|V|)$

$O((|V|+|E|)\log|V|)$

Union Find Data Structure

```
makeset(x)
  set π(x)=x
  rank(x)=0
```

```
find(x)
  if x≠π(x): π(x)=find(π(x))
  else: return x
```

original one

$\pi(x)$: parent of x $\quad$ rank(x): height of tree under x

```
Union(x,y)                    union by rank
  r_x = find(x)  r_y = find(y)
  if rank(r_x) ≤ rank(r_y):
    π(r_x) = r_y
    if rank(r_x)=rank(r_y): rank(r_y)=rank(r_y)+1
  else π(r_y) = r_x
```

```
find(x):
  if x≠π(x): π(x)=find(π(x))
  return π(x)
```

Path Compression

Runtime: makeset → $O(1)$ find → O(rank of root) union → O(rank of the roots)

Claim: Any root node of rank k has at least 2^k node in its tree.

If there are n elements overall, there can be at most $\frac{n}{2^k}$ nodes of rank k.

Prim's Algorithm

Description: $X=\{\}$, repeat until $|X|=|V|-1$:

(i) repeat a set $S \subset V$ for which X has no edges between S and $V-S$

(ii) let $e \in E$ be the minimum weight edge between S and $V-S$

(iii) $X = X \cup \{e\}$

```
Prim(G, w)
  for all u∈V: cost(u)=∞, prev(u)=nil
  cost(u_0)=0
  H = makequeue(V) // priority-queue
  while H is not empty:                  O(n) inserts & deletes
    v = deletemin(H)                     O(m) decrease keys
    for each (v,z)∈E:                    O(n+m log n)
      if cost(z) > w(v,z):
        cost(z) = w(v,z)
        prev(z) = v
        decreasekey(H,z)
```

Closest Pair

Naive Algorithm: Try every pair $\Theta(n^2)$

Divide and Conquer:

CLOSEST($\{p_i=(x_i,y_i) \mid i=1\cdots n\}$)

- SORT $\{P_1 \ldots P_n\}$ in increasing x
- $(P_L, P_L') \leftarrow$ CLOSEST($\{P_1 \cdots P_{\frac{n}{2}}\}$)
- $(P_R, P_R') \leftarrow$ CLOSEST($\{P_{\frac{n}{2}+1} \cdots P_n\}$)

  let d_L = distance(P_L, P_L') d_R = distance(P_R, P_R')

  $d_{min} = \min(d_L, d_R)$
- $A_L \leftarrow S_L$ restricted to strip $[x_{\frac{n}{2}}-d, x_{\frac{n}{2}}+d]$

  $A_R \leftarrow S_R$ restricted to strip $[x_{\frac{n}{2}}-d, x_{\frac{n}{2}}+d]$
- Sort A_L, A_R in increasing y
- For each $p=(x,y)$ in A_L

  $R \leftarrow \{q=(x',y') \text{ in } A_R \text{ with } y-d \leq y' \leq y+d\}$

  compute all distances between P and $q \in R$ and update minimum.