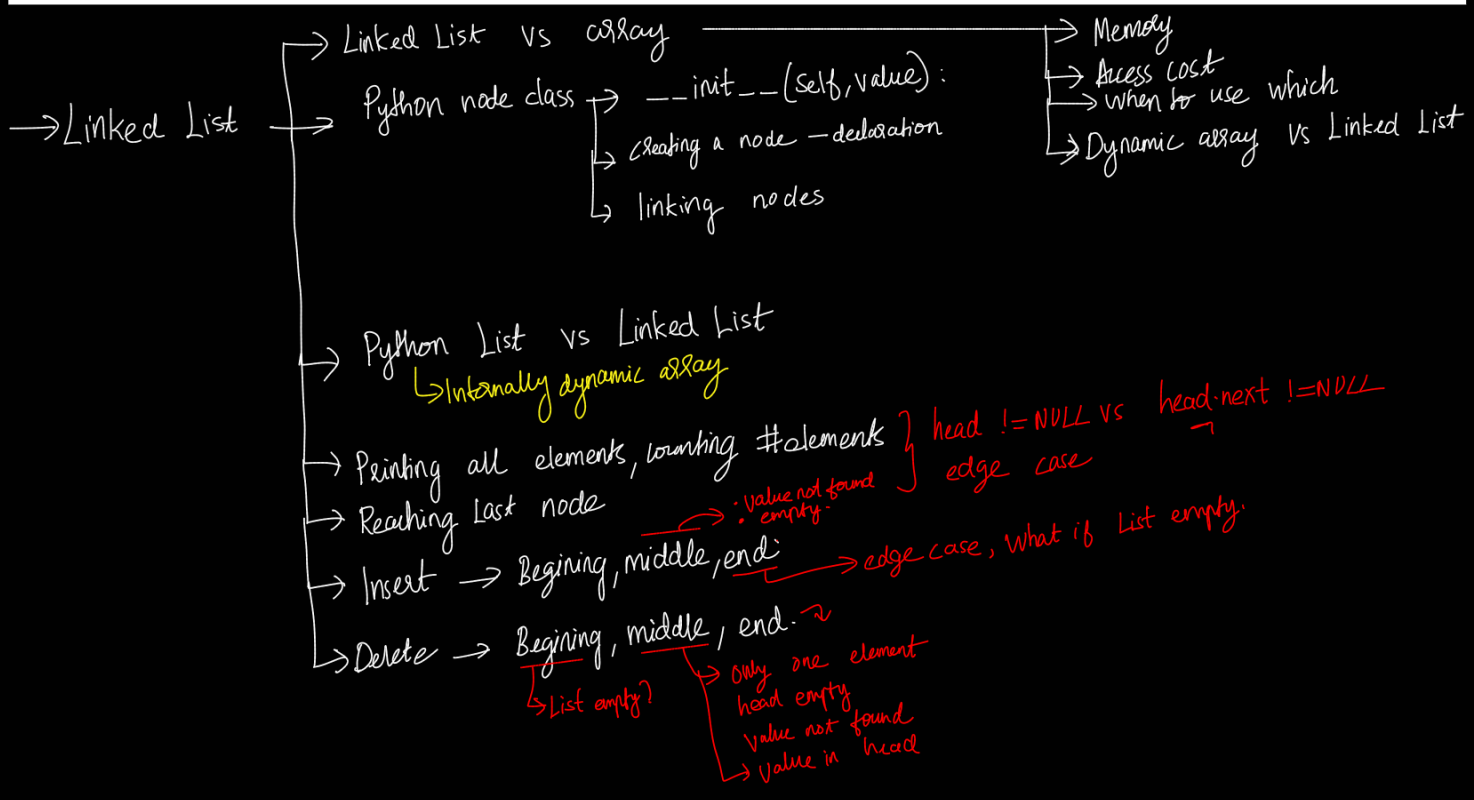
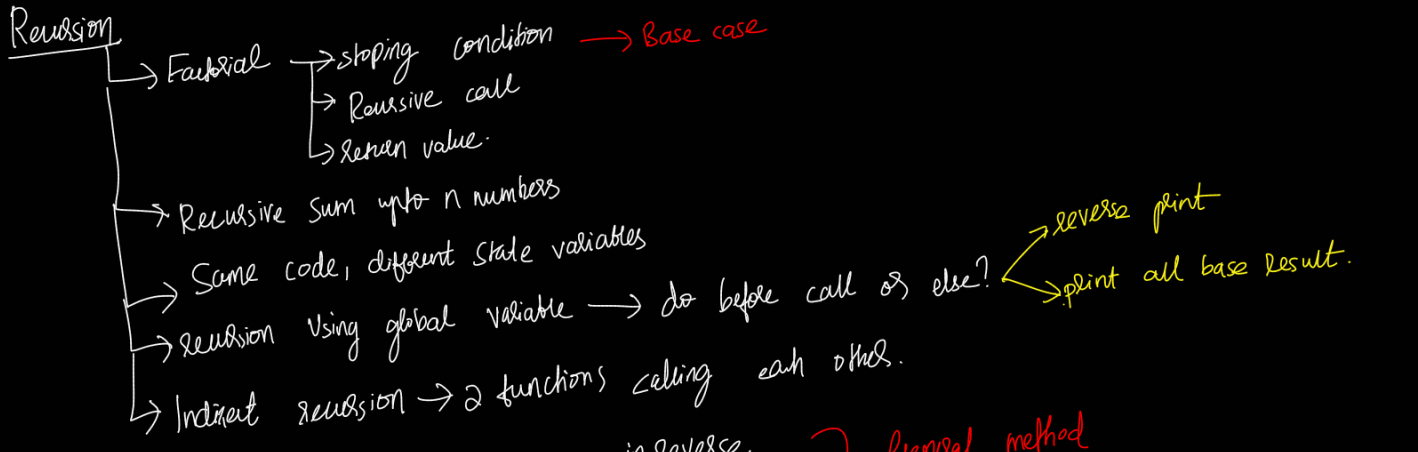
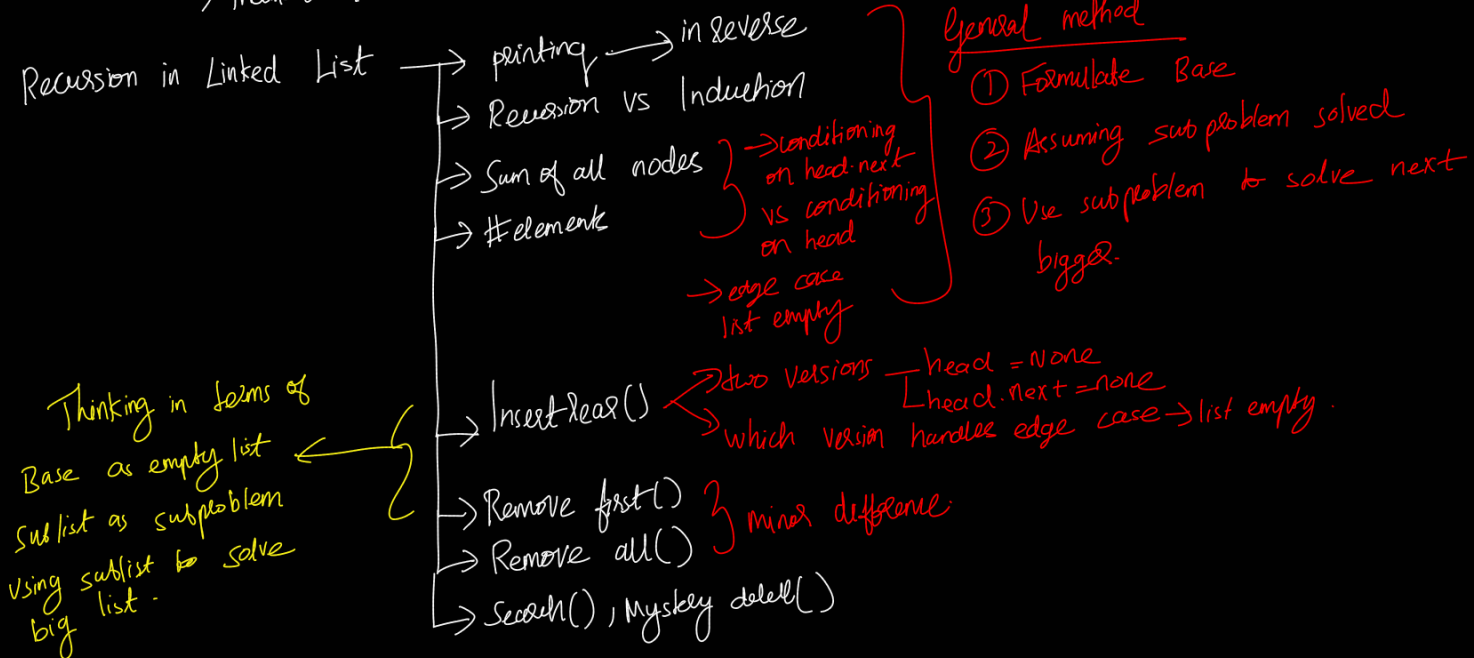
DSA pending
- Stack & Que — Practice set
- Stack & Que — Gate PYQ
- // — Python PYQ
- Binary Tree
  - Practice set
  - python Questions PYQ
  - SI
  - SA

Programming, Data Structures and Algorithms: Programming in Python, basic data structures: stacks, queues, linked lists, trees, hash tables; Search algorithms: linear search and binary search, basic sorting algorithms: selection sort, bubble sort and insertion sort; divide and conquer: mergesort, quicksort; introduction to graph theory; basic graph algorithms: traversals and shortest path.







Circular Linked List → Stopping criteria for traversal → current.next == head.
- → Printing, counting. → edge case → empty list, printing last node
- → Print last node
- → Insert at end, insert at start → why both are almost same
  - = insert at last + make last new node as head

Doubly Linked List → Insert at beginning/end

Question 22

```
def fun(head, pos):
    if pos <= 0 or head is None:
        return head

    if pos == 1:
        temp = head.next
        head = None
        return temp

    head.next = fun(head.next, pos - 1)
    return head
```

What output can be expected from the function call `fun(head,3)` called on the provided linked list?

4 → 3 → 7 → 9 → 2 → NULL

A. 4 → 7 → 9 → 2 → NULL
B. 4→ 3 → 9 → 2 → NULL
C. 4→ 3 → 7 → 2 → NULL
D. 4→ 3 → 7 → 2 → NULL

Answer: B

```
def removeNodes(head):

    if head is None or head.next is None:
        return head
    n = removeNodes(head.next)
    if n.data > head.data:
        return n
    head.next = n
    return head
```

Answer: C

What output can be expected from the function call `removeNodes(head)` called on the provided linked list?

5 → 2 → 13 → 3 → 8 → NULL

A. 5 → 2 → 13 → 8 → NULL
B. 13 → 3 → 8 → NULL
C. 13 → 8 → NULL
D. 13 → 3 → NULL

Asymptotic Analysis

- → A posteriori analysis → platform dep.
- → A priori analysis
- → Asymptotic analysis — Big-oh O; Big-Omega Ω; Θ Theta Notation | Like-oh o; Little-omega ω
- → Time vs space complexity
- → Complexity as function of input size n → T(n)

✅ 1. Growth Rate Hierarchy (from slowest to fastest)

This is the *most important cheat sheet* for asymptotics.

Slowest → Fastest

1. **Constant:** $1,\ 5,\ 100$
2. **Logarithmic:** $\log n,\ \log^2 n$
3. **Poly-logarithmic:** $(\log n)^k$
4. **Linear:** n
5. **Linearithmic:** $n \log n$
6. **Polynomial:** $n^2,\ n^3,\ n^k$
7. **Quasi-polynomial:** $n^{\log n}$
8. **Exponential:** $2^n,\ 3^n,\ a^n$
9. **Super-exponential:** $n!$
10. **Hyper-exponential / Tetration:** 2^{2^n}

$a = b^{\log_b a}$

$a^{\log_b c} = c^{\log_b a}$

$\log_c(ab) = \log_c(a) + \log_c(b)$

$\log_a b \cdot \log_b c = \log_a c$

$\log_b c = \frac{\log_a c}{\log_a b}$

$\log_a\left(\frac{1}{b}\right) = -\log_a b$

$\log_b a = \frac{1}{\log_a b}$

✅ 2. Common Algorithms and Their Time Complexities

Constant Time — O(1)
- Accessing array element
- Checking if stack is empty
- Hash table insert/search/delete (amortized)

Logarithmic — O(log n)
- Binary search
- Heap insert/delete
- Balanced BST operations (AVL, Red-Black)

Linear — O(n)
- Scanning array
- Finding max in list
- BFS/DFS on tree (no cycles)

Linearithmic — O(n log n)
- Merge sort
- Heap sort
- Quicksort (best/avg)
- Building a heap

Quadratic — O(n²)
- Selection sort
- Bubble sort
- Insertion sort (worst & avg)
- All-pairs comparisons
- Naive matrix multiplication

Cubic and Higher Polynomial — O(n³), O(n^k)
- Floyd–Warshall (all-pairs shortest paths)
- DP for edit distance / LCS: O(nm)

Exponential — O(2^n)
- Subset generation
- Traveling salesman brute force
- Exponential DP with bitmasks (often $2^n \cdot \text{poly}(n)$)

Factorial — O(n!)
- Permutation generation
- Brute forcing Hamiltonian cycles

Rules

→ When base is a constant, we can take log to compare
→ When base is not constant, we cannot take log to compare
→ we can use logarithmic identities to make base a constant
→ We can take log of T(n) & f(n) to compare & conclude only if they are not equal, if they are equal ⇒ no comment — eg n^3, n^2
→ Cancel out factors before comparing if any.

eg compare a^n & n^n

eg compare $n^a \log n$ & $n(\log n)^{10}$

$n < a^n$	$n < a^n$
$a^{\log_a n} < a^n$	$\log_a(n) < n \cdot \log_a 2$
$\log_a n < n$	$\log_a n < n$

→ log as orders preserving

→ Ignore constants only when ∈ f(n) form otherwise do not ignore.

→ Just like taking log preserves orders, exponentiating also preserves order
→ Assume some form of n → a^k a^{k^2}, a^{2^k}, a^{100k}, a^{200k^2} → eg $(\log n)^{\log\log(n)}$ vs $(\log\log n)^{\log n}$ put $n = a^{2^k}$

$(\log n)^k < n^\epsilon$ for any $\epsilon > 0$
asym

eg:- arrange

$\log n$, $(\log n)^{10}$ $\log(\log n)$ $(\log(\log n))^{10}$

$n = a^{2^k}$	2^k, $(2^k)^{10}$, k, k^{10}
$\log n = a^k$	A B C D
$\log(\log n) = k$	B > A > D > C

$n = a^k$	k, k^{10}, $\log k$, $(\log k)^{10}$
$\log n = k$	A B C D
	B > A > D > C

eg: a^{a^n}, $n!$, 4^n, a^n

$a^{a^n} > n! > 4^n > a^n$

→ Remember standard results
→ Not all are solvable using logs.

$a^{a^n} > n!$ $\quad 4^n > a^n$

$a^n > \log n!$ $\quad \left(\frac{4}{a}\right)^n > 1$

$a^n > n \log n$

Remember $\log n! = O(n \log n) = \log n^n$
but $n! = O(n^n)$
$n! <_{asym} n^n$

$n!$ = stirling's approx $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

but also

$\sqrt{\pi n}\left(\frac{n}{e}\right)^n$ vs n^n
$\sqrt{\pi n}\left(\frac{1}{e}\right)^n$ vs 1
$\sqrt{\pi n}$ vs e^n
$\sqrt{\pi n} < e^n$
$\Rightarrow n! < n^n$

$\log n! = \log(2\pi n)^{1/2} + \log\left(\frac{n}{e}\right)^n$

$\log n! = \frac{1}{2}\log(2\pi n) + n[\log n - \log e]$

$\log n! = n \log n - n \log e + \frac{1}{2}\log(2\pi n)$

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

$a^{\log n}$, $(\log n)^a$, $\sqrt{\log n}$, $\log(\log n)$

$n = a^{a^k}$
$\log n = a^k$
$\log \log(n) = k$

a^{a^k}, $(a^k)^a$, $(a^k)^{1/a}$, k
a^{a^k}, a^{ak}, $a^{k/a}$, k
$>$ $>$ $>$

→ Incomparability → $n^{1+\sin n}$ vs n → Graphical intuition

n^a vs $\begin{cases} n, & \text{odd} \\ n^3, & \text{even} \end{cases}$

→ For incomparable function we cannot say $f(n) = O[g(n)]$ or $g(n) = O[f(n)]$

→ Properties of Asymptotic Notations

① Transitivity — All 6 notations
② Reflexivity — O, Ω, Θ
③ Symmetry — Θ
④ Transpose Symmetry — $O \Leftrightarrow \Omega$, $o \Leftrightarrow \omega$

eg: $n^{\sqrt{n}}$, a^n, $n^{\log n}$, $n!$

taking log → $\sqrt{n}\log n$, $n \log a$, $\log n \cdot \log n$, $n \log n$

$n! > a^n > n^{\sqrt{n}} > n^{\log n}$

Notational convenience when we write

1. What does it mean when we say $f(n) = O(n)$?

It means:

There exists a constant $c > 0$ and n_0 such that

$$|f(n)| \le cn \quad \text{for all } n \ge n_0.$$

So:

- $O(n)$ is **a set of functions**.
- Saying $f(n) = O(n)$ means $f(n)$ **belongs to that set**.
- It is NOT an equality.
  It means **f grows at most linearly.**

So you should really read it as:

$$f(n) \in O(n).$$

In general $O(g(n)) = \{ f(n) \mid \exists_{c>0, n_0>0} [\forall_{n>n_0} f(n) \le c\, g(n)] \}$?

→ set of functions $f(n)$ that satisfy $f(n) \le c\, g(n)$ for some $c>0$ & all $n>n_0$.

2. What happens when I replace $f(n)$ with $O(n)$ in an expression?

This is where people often get confused.

Important rule:

Big-O notation is *not* a number, it is a **class of functions.**
So replacing a function by a big-O term means you are replacing it with a **bound**, not a value.

Loop Complexity

① $1+2+3+4 \cdots n = \frac{n(n+1)}{2}$

② $1^2+2^2+3^2+4^2 \cdots n^2 = \frac{n(n+1)(2n+1)}{6}$

③ $\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{n} \approx \log_e n = \ln n$

④ $\sum_{k=1}^{n} \log k \approx n \log n$ → $\log(n!) \approx n \log n$

$\sum_{i=1}^{\infty} \left(\frac{1}{i^2}\right)$ = constant

1) `for(i=1; i<N; i+=a)` → $O(N/a)$
2) `for(i=1; i<N; i*=a)` → $O(\log_a N)$
3) `for(i=1; i^2<N; i+=1)` → $O(\sqrt{N})$
4) `for(i=1; i<a^N; i*=b)` → $O(\log_b a^N) = O(N \log_b a)$ → $1, b, b^2, b^3 \cdots b^k = a^N$, $k = N\log_b(a)$
   if $a=b$ → $O(N)$
5) `for(i=N; i>=1; i=i/a)` → $O(\log_a N)$
6) `for(i=a; i<=N; i=i^b)` → $O(\log_b \log_a N)$
   6.1) $a=b=a$ → $O(\log_a \log_a N)$
   6.1) $a=b=k$ → $O(\log_k \log_k N)$
7) `for(i=n; i>=a; i=i^{1/b})` → $O(\log_b \log_a N)$ → $n, n^{1/b}, n^{1/b^2}, n^{1/b^3}, n^{1/b^t} = a$, $n = a^{b^t}$, $\log_a(n) = b^t$, $\log_b \log_a(n) = t$
7.1) `for(i=n; i>=a; i=sqrt(i))` → $a=a$, $b=a$ → $O(\log_a \log_a N)$

$a, a^b, a^{b^2}, a^{b^3}, \cdots$ $a^{b^t} = N$, $b^t = \log_a N$, $t = \log_b \log_a N$

8) `for(i=n^a; i>=b; i=i^{1/c})` → $\log_c \log_b(n^a)$ → $n^a, n^{a/c}, n^{a/c^2}, n^{a/c^3}, \cdots$ $n^{a/c^t} = b$, $n = b^{c^t/a}$, $\log_b n = \frac{c^t}{a}$, $t = \log_c \log_b(n^a)$

8.1) $a=2; b=2; c=2$
`for(i=n^a; i>=a; i=sqrt(i))` → $\log_2 \cdot \log_2(n^a)$

- `for(i=1; i<=N; i*=2)` / `  for(j=1; j<=i; j++)` $\longrightarrow O(N)$
- `for(i=1; i<=N; i*=2)` / `  for(j=1; j<=i^2; j++)` $\longrightarrow O(N^2)$
- `for(i=1; i<=N; i*=2)` / `  for(j=1; j<=N; j++)` $\longrightarrow O(N \log_2 N)$
- `for(i=2; i<=n; i=i^2)` — $\log_2\log_2(n)$
  `  for(j=1; j<=n; j++)` — n
  `    for(k=1; k<=n; k=k+j)` — $\frac{n}{j}$
  $\Big\} \rightarrow O(n \ln n \cdot \log(\log_2 n))$

- `for(i=1; i<=N; i++)` / `  for(j=1; j<=N; j*=2)` $\longrightarrow O(N \log_2 N)$
- `for(i=1; i<=N; i=i+2)` / `  for(j=1; j<=i; j=j*2)` $\longrightarrow O(\log_2 n!) = O(n \log_2 n)$
- `for(i=1; i<=N; i++)` / `  for(j=1; j<=N; j=j+i)` $\longrightarrow N\left(\frac{1}{1}+\frac{1}{2}+\cdots\frac{1}{N}\right) = O(N \ln N)$

Practice Q —

```
for(int i=1; i<=n; i++)
  for(int j=i; j<=n; j+=i*2)
```

$\rightarrow$ $j = i,\ i+2i,\ i+4i,\ i+6i \ldots i+ak_i = n$

$i(1+ak) = n$

$k = \left(\frac{n}{i}-1\right)\frac{1}{2}$

$k = t-1$

$t = \frac{n}{2i}+\frac{1}{2}$

i	k
1	$\frac{n}{2}+\frac{1}{2}$
2	$\frac{n}{2\times2}+\frac{1}{2}$
3	$\frac{n}{3\times2}+\frac{1}{2}$
n	$\frac{n}{n\times2}+\frac{1}{2}$

$\text{Sum} = \frac{n}{2}+\frac{n}{2}\left(1+\frac{1}{2}+\frac{1}{3}\cdots\frac{1}{n}\right)$

$\text{Sum} = \frac{n}{2}+\frac{n}{2}\ln_e n$

$= O(n \ln n)$

Q) `for(i=1; i<=n; i*=2)` / `  for(j=1; j<=i^2; j*=2)` $\longrightarrow$

$i = 1, 2, 4, 8, 16, 32, \ldots\ldots n$

$\log_2(1)+\log_2(2^2)+\log_2(4^2)+\log_2(8^2)+\ldots\log_2(n^2)$

$= \log_2(2^0)+\log_2(2^2)+\log_2(2^4)+\log_2(2^6)+\ldots\log_2 2^k$

$= 2+4+6+\ldots k$

$= 2(1+2+3+\ldots k/2) = O\left(\frac{k}{2}+1\right)^2 = O[\log^2(n)]$

$2^k = n^2$

$k = \log_2(n^2)$

$\frac{k}{2} = \log(n)$

`for(i=1; i<n; i*=2)` $\rightarrow \log_2(n)$

`  for(j=n; j>0; j/=2)`

`    for(k=j; k<n; k+=2)` $\longrightarrow$

$j = n,\ n/2,\ n/4,\ n/8,\ \ldots\ldots n/n$

$0+n/4+\frac{n-n/4}{2}+\frac{n-\frac{n}{8}}{2}+\cdots\frac{n-1}{2}$

$= \frac{n}{2\times2}+\frac{3n}{4\times2}+\frac{7n}{8\times2}+\frac{15n}{16\times2}+\cdots\frac{n(n-1)}{n\times2}$

$= \frac{n}{2}\left[\frac{1}{2}+\frac{3}{4}+\frac{7}{8}+\frac{15}{16}+\cdots\frac{n-1}{n}\right] \approx \frac{n}{2}\log_2(n)$

$\Rightarrow \text{Total} = O(n \log_2^2(n))$

Notes from practices

① → When complexity of binomial coefficients are asked apply stirling iteratively

eg. $\binom{n}{n/2} = \frac{\sqrt{2\pi n}\,(n/e)^n}{\pi n \cdot (\frac{n}{2e})^{n/2} \cdot (\frac{n}{2e})^{n/2}} = \sqrt{\frac{2}{\pi n}} \cdot 2^n = O(2^n/\sqrt{n})$

$\Rightarrow \binom{n}{n/2} = O(\frac{2^n}{\sqrt{n}}) \Rightarrow \binom{n}{n/2} \neq \Theta(2^n)$

Para ②) $n,\ n^2+16^{\lg n},\ 2^n+\sqrt{4^n},\ 3^n,\ n^3 \cdot \lg n$

$16^{\lg n}$

$(2^4)^{\lg n}$

$(2^{\lg n})^4$

n^4

$n,\ n^2+n^4,\ 2^n \cdot 2,\ 3^n,\ n^3 \lg n$

$5,\ ③,\ ②,\ ①,\ ④$

STACK

- ADT → defines operation not implementation
- Teminology → top, push, pop, isfull, isempty, LIFO / FILO
- Why stack? → Reverse, activation record, expression eval, undo Mechanism
- Stack permutation → Catalan number = $\frac{1}{n+1}\binom{2n}{n}$

Array implementation
- top points → (when top = -1)
- push operation → a[++top] = value; } checking conditions of empty & full stack } Complexity O(1)
- pop operation → return a[top--];
- is empty → return (top == -1)
- is full → return (top == N-1)
- Limitation → array size initialised

Linked List Implementation → using head as top points — Why ?
- push → P = newnode(value); P->next = head; head = P
- pop → temp = head->value; head = head->next; return temp

} O(1)

QUEUE

- Why — events — OS — etc
- Teminology — Front, Rear, head, tail — Analogy → FIFO, LILO — Enqueue, Dequeue
- Reversing elements of Queue using stack
- Array implementation → Circular array based
  - elements always between front & rear
  - No need to decrement front or rear
  - Front or rear never cross each other
  - rear = (rear+1) % N — enqueue
  - front = (front+1) % N — dequeue
  - empty ⇒ front = rear = -1

→ getsize()

f<r : r-f+1
f>r : N-f+r+1
f=r≠-1 : 1
f=r=-1 : 0

→ f==-1 : 0
else : (N-f+r+1)%N

- check before enque → (r+1)%N == f ⇒ should not enque ⇒ array full
- check before deque → f == r ⇒ only one element ⇒ after deque set f=r=-1

Special treatment for first insert & last pop

- check before enque → r == f == -1 ? ⇒ empty que → set f=r=0
- check before deque → r == f == -1 ? ⇒ empty ⇒ return

→ Alternate implementation where rear points to first empty slot after last element
- f==r ⇒ empty
- f==(r+1)%N ⇒ full

→ rear always empty → even for full Queue
⇒ One place wasted ☆
⇒ What happens if we don't maintain this ⇒ We can't determine if array is full or empty.

- Enque → if ((r+1)%N == f) return 0; //full
  array[rear] = data;
  rear = (rear+1)%N;
- Deque → if (f == r) return 0; //empty
  temp = array[front];
  front = (front+1)%N
  return temp;

#element
- f==r : 0
- r>f : r-f
- f>r : r+N-f

⇒ unconditionally size = (N-f+r)%N

→ LinkedList Implementation → Why head as front is best option. Why not head as rear?
↳ Empty : f=r=NULL

• Enque
```
if(rear==NULL) front=rear=newNode(data)
else {
rear->next = NewNode(data)
rear = rear->next }
```

• Dequee
```
if (front ==NULL) return; //empty
data = front->data;
front = front->next;
if (front = NULL) rear=NULL
```

→ When we deque last element front becomes null but rear has to be forced
→ When we insert first element, front has to be forced.

→ Queue using 2 stacks S1, S2
Enque → push (S1)
Deque → if S2 empty → transfer all S1→S2
else pop S2.
↳ Complexity O(m+n)

→ Stack using 2 Queues
— 2 types of implementation < costly push / costly pop

Costly pop
→ Maintain one Q empty
→ other Q original order
→ When pop transfer n-1
→ pop out nth.

Costly push
→ Maintains one Q empty
→ Maintain other Q reversed
→ New element received in empty Q
→ Transfer from reversed Q to other Q

→ Balancing symbol : (1) Opening symbol → push
(2) closing symbol → pop & check whether corresponding
→ if not corresponding ⇒ error
(3) At end of string → stack not empty ⇒ error.

→ 2 Stack using 1 array → 2 implementation → which is efficient

→ Expressions — Infix, postfix, prefix

precedence:

↑	right to left
*, /	left to right
+, −	left to right

eg.
a+b↑c*d/e−f↑g*h/i↑j

bc↑	fg↑ ij↑
bc↑d*	fg↑h*
bc↑d*e/	fg↑h*ij↑/

abc↑d*e/+fg↑h*ij↑/−

↳ Infix: Max(4, Min(10,2))
10 2,
10 2, Min
postfix: 4 10 2, Min, Max

→ Infix to postfix using stack → if (operand) → pass to o/p
if (operator) → More priority than the one in stack → push
→ Less priority than one in stack → pop stack & pass to o/p until a lesser priority operator is available in stack
→ Equal priority → if not ↑ → pop stack & pass to o/p until a lesser priority operator is available in stack
if ↑ → push

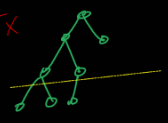
if (open '(') → push
if (close ')') → pop & pass to o/p until open '(' poped
↳ do not pass '(' & ')' to o/p.

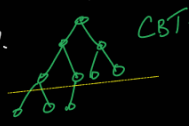
→ Postfix expression evaluation → if (operand) → push to stack
if (operator #) → pop 2 elements Y, X from stack
→ perform Z = X # Y
→ push result to stack

Binary Tree

- Full binary tree — #children = {0, 2} → All internal node = degree 2
- Complete Binary tree
  - All levels except last level is full
  - last level is left filled
  - Leaf node can only appear in last & 2nd last level
  - All nodes except the last level have exactly 2 children

eg. CBT ✗





CBT ✓

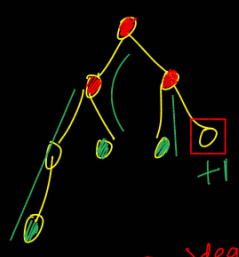
- Perfect binary tree
  - Complete Binary tree where last level is also full.
  - All nodes have exactly 2 children
  - All leafs in last level

Terminology

- Depth of node — #edges from root to node / depth(root) = 0
- Height of Node — #edges from node to leaf in the longest path / height(leaf) = 0
- Height of Tree = Height of root. = Depth of deepest leaf node

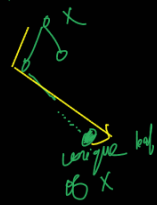
Binary Tree ⇒ #Leaf Nodes = 1 + #Nodes with degree = 2

#L = 1 + #M



Intuition

Every degree 2 node has a unique leaf node which is the rightmost leaf in the left subtree



- → degree 2 nodes
- → leaf nodes corresponding to the rightmost node in the left subtree of a unique degree 2 node

In full binary tree
all internal nodes = degree 2
⇒ FBT #L = 1 + #I
Total Nodes = 2#I + 1

Full Binary Tree height (h)

- # Internal nodes = $2^h - 1$
- # leaf nodes = $2^h = L$
- # Total nodes (n) = $2^{h+1} - 1$
- # Nodes in level i = 2^i

Full binary tree (#nodes n)

- # Leaf nodes = $\frac{n+1}{2}$ (L)
- height = $\log_2\left(\frac{n+1}{2}\right)$
- height = $\log_2(n+1) - 1$

Full binary tree (#leaf = L)

- #nodes = 2L − 1 (n)
- # Internal nodes = L − 1
- height = $\log_2 L = \log_2(2L) - 1$

$n = 2L - 1$
$2^h = L$
$L = \frac{n+1}{2}$

Complete Binary tree (height h)

- Min #Nodes = 2^h
- Max #Nodes = $2^{h+1} - 1$

Binary tree (height h)

- Min #Nodes = h + 1
- Max #Nodes = $2^{h+1} - 1$
- Min # leaf = 1
- Max # leaf = 2^h

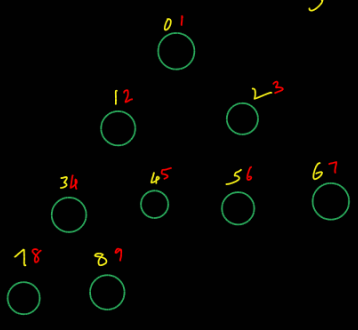
Binary tree (n nodes)

- Minimum height = $h = \left\lfloor \log_2\left(\frac{n+1}{2}\right) \right\rfloor$
- Max height = n − 1

Array Representation of Binary tree → only for CBT suitable

When index starts at 1:

- Index of child = { 2 × Index of parent (Left child), 2 × Index of parent + 1 (Right child) }
- Index of parent = $\left\lfloor \frac{\text{child Index}}{2} \right\rfloor$



When index starts at 0:

- Index of child = $(2 \times I_p)^{LC} + 1$, $(2 \times I_p)^{RC} + 2$
- Index of parent = $\left\lfloor \frac{I_c - 1}{2} \right\rfloor = \left\lceil \frac{I}{2} \right\rceil - 1$

Linked List representation

→ Calculating height using recursion → fun(t)

```
if (!t) return 0
if (t->left == NULL && t->right == NULL) return 0;
if (t->left)  a = fun(t->left)
if (t->right) b = fun(t->right)
return 1 + Max(a,b)
```

→ return deepest node → deep(t)

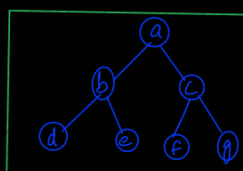
```
while (Q not empty(Q)) {
  temp = deque(Q)
  if (temp->left) enque(left)
  if (temp->right) enque(right)
} return temp;
```

→ return level level(t) {

```
level = 0
if (!t) return 0;
Enque(t);
Enque(NULL);
while (!Q empty()) {
  t = deque();
  if (t == NULL) {
    if (!Qempty) enque(NULL)
    level++;
  }
  else {
    if (t->left) enque(t->left)
    if (t->right enque(t->right)
  }
} return level;
}
```

→ Compare two tree using recursion

```
Is same (t1, t2) {
  if (t1 == NULL && t2 == NULL) return 1;
  else if (t1 == NULL || t2 == NULL) return 0;
  return (t1.data == t2.data &&)*
    Is same (t1->left, t2->left) &&
    Is same (t1->right, t2->right);
} //Comment out * to just compare structure & not data
```



post → left right parent d e b f g c a — Traversal kick 3rd time
in → left parent right d b e a f c g — 2nd time → By projection & squishing
pre → parent left right a b d e c f g — 1st time

Tree construction given element in

a) Inorder
b) post order
c) preorder

given any one of these → tree not unique — #possible tree = catalan(n) n → #nodes

given any two of these → tree unique

- Case 1: Inorder + Xorder
  - → Use X order to calculate the root
  - → Understand left & right subs from Inorder
  - → Iterate going Xorder from L to R or R to L
- Case 2: Post order + Preorder
  - → Not unique → Find root & child using intersection

General BT + Inorder + Preorder
General BT + Inorder + Postorder
Full BT + Post Order + Preorder
Complete + Any one
} Unique solution

→ divide evenly & left bias

☆ BST + Pre / Post order ⇒ Unique Tree

Binary Search Tree — left biased

- → Inorder traversal → sorted order
- → Insert, Search — O(h)
- → Max & Min in BST → Always go one direction until impossible → return node
- → k^{th} Max / Min → Inorder traversal — O(n)
- → Inorder / preorder / postorder — 3n visits per node ⇒ O(3n) = O(n)
- → Range Search
  - ☆ → if value in range → visit both
  - → if value < range → visit right child only
  - → if value > range → visit left child only

Given preorder or postorder the BST is uniquely determined if distinct keys
↓
⇒ Inorder can be derived.

```
RangePrint (node *R, a, b) {
  if (!R) return 0;
  if (R->data > a)  RangePrint(R->left, a, b);
  if (R->data < b)  RangePrint(R->right, a, b);
  if (a <= R->data <= b)  print(R->data)
}
```

→ Complexity ≡ #Nodes in Range $[a,b] \rightarrow m$
+ #Nodes visited but not printed

$\equiv O(m+2h)$ $\quad h$ = Height of tree

$\equiv O(\log_2 n + m)$ $\quad h = O(\log_2 n)$

BST deletion

Inorder predecessor → Max Value in left subtree (go right until no longer possible) → if there is no left subtree → immediate ancestor for which our node is in the right subtree

Inorder successor → Min Value in right subtree (go left until no longer possible) → if no right subtree → immediate ancestor for which our node is in the left subtree.

↳ either a leaf or has only one child

3 Cases of deletion → ① → leaf Node → Just delete
② → One child → Delete node, give child to grandparent ⑨
③ → Two children → Replace with Inorder predecessor or Inorder successor

} The predecessor/successor (inorder) will always be a leaf or a parent with degree 1.

⇒ Case ③ results in case ② or ①

→ Identifying possible & valid search traversals
↳ if you go left (right) from a node X, then all following nodes should be $\leq X$ ($\geq X$)

→ Identifying possible & valid preorder, inorder, postorder traversals of Binary Search tree.
↳ Tree will have smaller values to left child & larger values to right child.

→ Search traversal in BST → Given set S of traversed nodes while searching for a given value X

possible traversal ordering = $\dfrac{|S|!}{|S_{<x}|!\ |S_{>x}|!}$

$|S_{<x}|$ → #elements $<x$

$|S_{>x}|$ = #element $>x$

$|S|$ = #elements

→ Logic in a legal sequence the $S_{<x}$ has to appear in increasing order & $S_{>x}$ has to appear in decreasing order

→ For every correct legal sequence, there is $|S_{<x}|!$ ways & $|S_{>x}|!$ ways to mess up the given sequence so that only the original is legal.

→ It is not possible to get a legal sequence from another legal sequence by interchanging individually only among $S_{<x}$ set & $S_{>x}$ set.

→ All such permutations are unique ⟹ Total legal = $\dfrac{|S|!}{|S_{<x}|!\ |S_{>x}|!}$

→ the number of ways in which I can insert given X numbers to get max height BST = 2^{X-1} → Since at each point, I can only insert, either the largest or the smallest value among the remaining numbers, otherwise, some node will have degree 2 & max height not possible.

So since @ each point except the last node I have 2 choices = 2^{X-1} possibilities.

ways to insert numbers ≡ # skew trees that have no 2 degree nodes given X elements.

Hashing

→ We prefer, when there is no natural ordering btw data

→ Python dictionary is hash table

Randomised Quick sort → Random pivot variables

	Sorted array	Balanced BST	Hash table
Insertion	O(n)	O(logn)	O(1) Avg
Deletion	O(n)	O(logn)	O(1) Avg
Search	O(logn)	O(logn)	O(1) Avg

Direct Addressing Table

└ fancy name for array

Limitation

→ Range should be small, compact to avoid mem waste

→ Keys should be non negative unique integers without letters

→ Key must dense, compact

key = Bus #
unique Bus #

eg IPv4 address → 2^{32} very large huge wastage

Example: Bus Services

```
// a[] is an array (the table)

insert(key)
  a[key] = key data

delete(key)
  a[key] = NULL

find(key)
  return a[key]
```

O(1)

0	
1	
2	data_2
⋮	⋮
998	data_998

Hashing Ideas

① Map non integers key to integers → Prehashing

② Map large numbers to smaller integers → Hashing

③ Modulo operator Hash(K) = K%M m → #slots

a) if $M = 2^n$ ⇒ Hash(K) = K%M = K%2^n → last n bits of key, n LSB of key

↳ Hash not depending on all bits of K is generally not a good idea

b) if $M = 10^n$ ⇒ Hash(K) = K%10^n → last n digits of key (Base 10 representation)

c) Pick m = prime numbers close to 2^n → good final step to get keys in the range after a more general hash function, which properly depends on all bits of key

Desirable property

① h(k) – depends on every bit of K
– minute changes in K create completely different keys.

② h(K) – All buckets equally likely
– Data evenly distributed
– Least collisions for any given size of hash table

③ h(k) – Computationaly inexpensive

Simple Uniform Hashing

Simple uniform hashing: is when *any given element is equally likely to hash into any of the m slots*, independently of where any other element has hashed to

$$\textbf{Uniform:} \quad \Pr_{h \in \mathcal{H}}[h(x) = i] = \frac{1}{m} \quad \text{for all } x \text{ and all } i$$

input key, slot #, #slots

Collision Resolution

① Separate chaining → closed addressing ≡ open hashing

② Linear probing
③ Quadratic probing
④ Double Hashing
→ open addressing ≡ closed hashing

n ≤ m
→ no chain
→ Find another slot in hash table in collision

The use of "closed" vs. "open" reflects whether or not we are locked in to using a certain position or data structure (this is an extremely vague description, but hopefully the rest helps).

For instance, the "open" in "open addressing" tells us the index (aka. address) at which an object will be stored in the hash table is not completely determined by its hash code. Instead, the index may vary depending on what's already in the hash table.

The "closed" in "closed hashing" refers to the fact that we never leave the hash table; every object is stored directly at an index in the hash table's internal array. Note that this is only possible by using some sort of open addressing strategy. This explains why "closed hashing" and "open addressing" are synonyms.

Contrast this with open hashing - in this strategy, none of the objects are actually stored in the hash table's array; instead once an object is hashed, it is stored in a list which is separate from the hash table's internal array. "open" refers to the freedom we get by leaving the hash table, and using a separate list. By the way, "separate list" hints at why open hashing is also known as "separate chaining".

In short, "closed" always refers to some sort of strict guarantee, like when we guarantee that objects are always stored directly within the hash table (closed hashing). Then, the opposite of "closed" is "open", so if you don't have such guarantees, the strategy is considered "open".

Separate chaining

- Search(k) → Worst case = length of chain = $O(n)$
  - → Worst case similar to array, linked list
  - → But why hash table then?
  - → Worst case → very rare → $O(n)$
  - Avg case → most common → less expensive
  - $O(1)$
- Insert(k) → $O(1)$
- Delete(k) → $O(n)$
- Average case assuming simple uniform hashing
  - Unsuccessful search: $\Theta(1+\alpha)$ → for traversing chain; for calculating hash and table position
    - if we treat α a constant ⇒ m increases with n → design choice
    - then Unsuccessful search is $\Theta(\text{constant})$
  - Successful search: $\Theta(1+\alpha/2) = \Theta(1+\alpha)$

Load factor (α)

$\alpha = n/m$

n = #Elements (in table)

m = # slots = Hash table size

Intuitively α = #Elements per chain

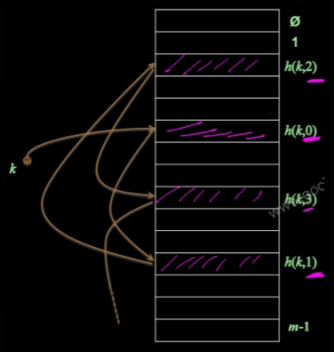
Probing for Next slot

We want to design a function h, with the property that for all $k \in \mathcal{U}$:

$$h : \mathcal{U} \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

universe of keys | trial count | slot in table

$(h(k,0), h(k,1), \ldots, h(k,m-1))$ is a permutation of $0, 1, \ldots, m-1$.



if collision → hash again by increasing trial count → repeat until free slot.

General case for hashing with probe

$$\text{hash}(\text{key}, i=\text{probe}) = [H(\text{key}) + f(i)] \% m$$

$i = 0, 1, 2 \ldots m-1$

① Linear probing $f(i) = i$ → Try next slot if collision until free slot

② Quadratic probing $f(i) = c_1 i + c_2 i^2$

```
while (A[i] != empty) {
    if (A[i] == key) break;
    i++;
}
```

Search → ① Linear probing → unsuccessful search → continuous looking in next slot until empty location
→ successful search → continuous looking in next slot until key found

Deletion → ① Linear probing → When we delete, we need to put a marker

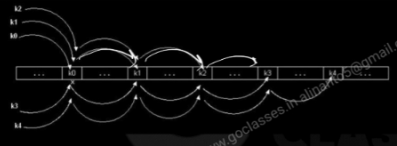
Marker → Indicates search algo to treat it as a filled spot
→ Indicates insert algo to treat it as empty slot

↑ edgecase → if no marker placed by delete then search may become unsuccessful even if key is in table

```
delete(key, H) {
    if(i index = search(key, H))
        print("not found");
    else H[index] == MARKER;
}
```

Disadvantage → Primary clusters

Problem with Quadratic Hashing → Secondary clusters



Secondary cluster

if two keys have the same initial probe position, then their probe sequences are the same,

since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.

Double hashing

Use two hash functions:

- h1 computes the hash code
- h2 computes the increment for probing

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

$h_1(k) = h_1(l)$

$\not\Rightarrow h_2(k) = h_2(l)$

⇒ even if h_1 hash collides h_2 hash will be different, & will result in different probing seq.

Distinct probing sequences possible

① Linear probing — m
② Quadratic probing — m
} Starting value decides entire sequence of probe

③ Double hashing — m^2 → $\frac{h_1(k)}{m} + i \times \frac{h_2(k)}{m} = m^2$

④ Simple uniform hashing with closed hashing $= m!$ — sequence non deterministic given starting value

sequence $m \cdot (m-1) \cdot (m-2) \cdots \cdots 1 = m!$

⑤ Hypothetical $h_1(k) + i h_2(k) + i^2 h_3(k) = m^3$

Analysis of open addressing w.r.t Simple uniform hashing

→ any of the $m!$ permutation of probing sequence is equally likely

- Load factor (α) in open addressing ≤ 1
- Since $n \leq m$

As in our analysis of chaining, we express our analysis of open addressing in terms of the load factor $\alpha = n/m$ of the hash table. Of course, with open addressing, at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$.

We assume that we are using uniform hashing. In this idealized scheme, the probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0, 1, \ldots, m-1\rangle$. Of course, a given key has a unique fixed probe sequence associated with it; what we mean here is that, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

We now analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Unsuccessful search #probes $\Theta\left(\frac{1}{1-\alpha}\right)$

Intuition

$P[\text{next probe getting empty space} / \text{n data, m slots}] = \frac{m-n}{m} = \frac{\#\text{empty slots}}{\#\text{slots}}$

$X = (\#\text{probes required} / \text{unsuccessful})$

$f_X(k=k) = \left(\frac{m-n}{m}\right)\left(\frac{n}{m}\right)^{k-1}$ $\quad 1 \leq X \leq \infty \rightarrow$ geometric distribution

- P(first $k-1$ probes should collide)
- P(Last probe hitting zero)

$X \sim$ geometric $\left(p = \frac{m-n}{m}, q = \frac{n}{m}\right)$

$E(X) = \frac{1}{p} = \frac{m}{m-n} = \frac{1}{1-n/m} = \frac{1}{1-\alpha}$

$\Rightarrow \boxed{X \sim \Theta\left(\frac{1}{1-\alpha}\right)}$

Geometric random (side Quest)

$X \sim$ Geometric(p, q) — each trial = Bernoulli(p)

X = #trails required to get a success (including success trial)

$E(X) = \frac{1}{p}$

$Var(X) = \frac{q}{p^2}$

Let $Y = X-1$

#Failures required to get a first success.

$E(Y) = E(X) - 1 = \frac{q}{p}$

$Var(Y) = Var(X-1) = Var(X) = \frac{q}{p^2}$

Cost of Successful search

→ Cost of successful search for i^{th} item with n items in the table — $1 \leq i \leq n$

≡

→ Cost of inserting i^{th} item with $i-1$ items in the table

≡

→ Cost of unsuccessful search for with $i-1$ items in the table

$\equiv \frac{1}{1-\frac{i-1}{m}}$

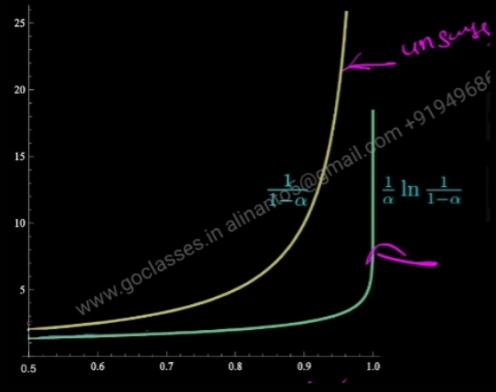
$\therefore$ Since it is equally likely that in successfull search, the item we are searching for can be in any of the i^{th} position $1 \leq i \leq n$

$\Rightarrow$ Cost of Successfull search

$= \frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i-1}{m}} = \frac{m}{n}\sum_{i=1}^{n}\frac{1}{m-(i-1)}$

$= \frac{m}{n}\sum_{k=0}^{n-1}\frac{1}{m-k} = \frac{1}{\alpha}\sum_{k=m-n+1}^{k=m}\frac{1}{k} \simeq \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx = \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

	unsuccessful	successful
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Open Addressing	$O\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$



3 expectation questions from Hashing

① Expected items per slot. ⇒ Separate chaining

X: No. of items per slot

$E[X_i] = ?$

Each item is equally likely to hash into any slot → Uniform hashing.

$\Rightarrow X = I_1 + I_2 + I_3 + \dots + I_n$

$I_i \rightarrow$ Indicator for i^{th} data point hashing into the slot under consideration

$n \rightarrow$ Total # datapoints.

$X \sim \text{Binomial}(n, \frac{1}{m})$

$I_i \sim \text{Bernoulli}(\frac{1}{m})$ → Since Uniform hashing

$E(X) = \frac{n}{m} = \alpha$ [load factor]

$m \rightarrow$ #slots.

---

② Expected #empty slots. → Assuming → a) Simple uniform hashing b) Separate chaining

X: # of empty slots

n = # data points

m = #slots

$X = I_1 + I_2 + I_3 + \dots + I_m$

where I_i is the indicator for i^{th} slot being empty

$P(I_i = 1) = \left(\frac{m-1}{m}\right)^n$

$E(X) = m E[I_i] = m\left(\frac{m-1}{m}\right)^n$

I_i identical but not independent ⇒ $X \not\sim$ Binomial

---

③ Expected #collisions Assuming → a) Simple uniform hashing b) Closed addressing.

X: # collisions

$n \rightarrow$ # datapoints

$m \rightarrow$ # slots.

$X = I_1 + I_2 + I_3 + \dots + I_n$

$I_i \rightarrow$ #collisions in i^{th} insert

$E(X) = E(I_1) + \dots E(I_n)$

$E(X) = \frac{0}{m} + \frac{1}{m} + \frac{2}{m} + \frac{n-1}{m}$

$E(X) = \frac{1}{m} \frac{n(n-1)}{2}$

$I_1 = 0$

$I_2 = 1/m$

$I_3 = 2/m$ → since closed hashing I_i is independent of each other

$I_i = \frac{i-1}{m}$

$I_n = \left(\frac{n-1}{m}\right)$

Algorithms

Revisit time complexity

```
for(i=1; i<=N; i=i*2)
    for(j=2; j<=N; j=j*2)
```

k	1	2	3	4	k
i	1	2	4	8	$2^{k-1} = N$

$k-1 = \ln_2 N$

$k = \ln_2 N + 1$

$k = O(\ln_2 N)$

	1	2	3	4	k
j	2	4	16	16^2	$2^{2^k} = N$
	2	2^2	2^4	2^8	

$2^k = \ln_2 N$

$k = \ln_2 \ln_2 N$

Recursive equation method

①
```
fun(n) {
    if(n<=1) return 1;
    else return fun(n/2);
}
```
$T(n) = T(n/2) + 1$

$T(1) = 1$

②
```
A(n) {
    if(n<=1) return 1;
    else return A(n-1)+n;
}
```
$T(n) = T(n-1) + 1$

③
```
fun(n) {
    2fun(n/2)
}
```
→ $T(n) = 2T(n/2) + 1$

→ $T(n) = T(n/2) + 1$

④
```
fun(n) {
    fun(n/2) + fun(n/2) + n
}
```
→ $T(n) = 2T(n/2) + 1$

→ This is a difference equation

- → discrete Maths → exact solution
- → we only need approximation
  - ① Iterative Method / Repeated Substitution
  - ② Recursive Tree Method
  - ③ Master Theorem

$T(n) = T(n-1) + 1 \longrightarrow T(n) = O(n)$

$T(n) = T(n-1) + n \longrightarrow T(n) = O(n^2)$

$T(n) = T(n/2) + n \longrightarrow T(n) = O(n)$

$T(n) = 2T(n/2) + n \longrightarrow T(n) = O(n \lg n + n) = O(n \lg n)$

① Iterative sub Method

$T(n) = 2T(n/2) + n\log n \longrightarrow T(n/2) = 2T(n/4) + \frac{n}{2}\log\left(\frac{n}{2}\right)$

$T(n) = 2x2x2x\ldots \text{kms } T(1) + n\left[\log n + \log\frac{n}{2} + \ldots \log 1\right] \quad T(n/4) = 2T(n/8) + \frac{n}{4}\log\left(\frac{n}{4}\right)$

$T(n) = 2^k + n\left[\log(n) + \log\left(\frac{n}{2}\right) + \log\left(\frac{n}{4}\right) + \cdots \log\left(\frac{n}{2^k}\right)\right]$ $\quad 2^k = n,\ k = \ln_2(n)$

$T(n) = 2^k + n\left[\log(n) + \log(n) + \log\left(\frac{1}{2}\right) + \log(n) + \log\left(\frac{1}{4}\right) \cdots \log(n) + \log\left(\frac{1}{2^k}\right)\right]$

$= 2^k + n\left[k\log(n) - (1+2+3+4+\cdots k)\right] = 2^k + n\left[\log(n) - \frac{k^2}{2}\right]$

$= O(2^k + n\cdot\log^2(n)) = O(2^k)$

$T(n) = n^{1/2} T(n^{1/2}) + n$

$T(n) = n^{1/2}\cdot n^{1/4}\cdot n^{1/8}\cdot n^{1/k} T(2) + n + n^{1/2}\cdot n^{1/2} + n^{1/2}\cdot n^{1/4}\cdot n^{1/4} + n^{1/2}\cdot n^{1/4}\cdot n^{1/8}\cdot n^{1/8}$

$\Rightarrow T(n) = n\cdot k + n^{1/2 + 1/4 + 1/8 + \cdots 1/2^m}$

$T(n) = n\ln_2(n) + n\ln_2(n)$

$T(n) = O(n\ln_2(n))$

$T(n^{1/2}) = n^{1/4}\cdot T(n^{1/4}) + n^{1/2}$

$T(n^{1/4}) = n^{1/8}\cdot T(n^{1/8}) + n^{1/4}$

let $n^{1/k} = 2$

$\Rightarrow 2^k = n$

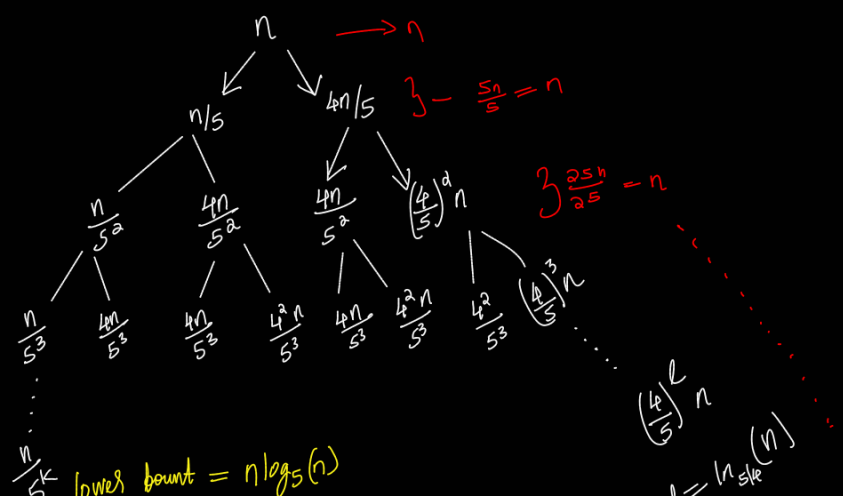
$k = \ln_2(n)$

$2^m = k$

$m = \ln k = \ln\ln(n)$

Tree Method

eg:- $T(n) = T(n/5) + T\left(\frac{4n}{5}\right) + n \quad \because T(1) = 1$



$\frac{5n}{5} = n$

$\frac{25n}{25} = n$

lower bound $= n \log_5(n)$

$\frac{n}{5^k} = 1$

$k = \ln_5(n)$

$T(n) = \Omega[n \log_5(n)]$

$= \Omega\left[n \frac{\log_2(n)}{\log_2(5)}\right]$

$l = \ln_{5/4}(n)$

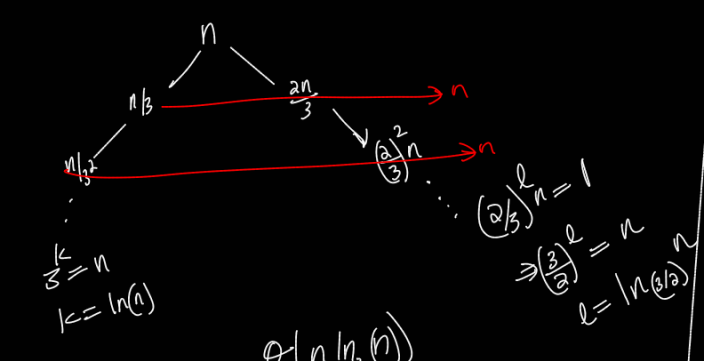
upper bound $= n \log_{5/4}(n)$

$T(n) = O[n \log_{5/4}(n)]$

$= O\left[n \frac{\log_2(n)}{\log_2(5/4)}\right]$

$\Rightarrow T(n) = \Theta[n \log_2(n)]$

$T(n) = T(n/3) + \left(\frac{2n}{3}\right) + n$



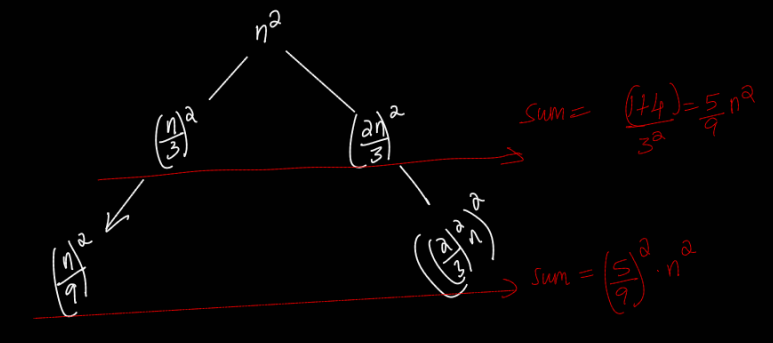
$\left(\frac{2}{3}\right)^l n = 1$

$\Rightarrow \left(\frac{3}{2}\right)^l = n$

$l = \ln_{(3/2)}(n)$

$\Theta(n \ln_2(n))$

$T(n) = T(n/3) + T\left(\frac{2n}{3}\right) + n^2$



$T(n) = n^2 + \frac{5}{9}n^2 + \left(\frac{5}{9}\right)^2 n^2 + \cdots \left(\frac{5}{9}\right)^k n^2$

$T(n) = n^2\left[1 + \frac{5}{9} + \left(\frac{5}{9}\right)^2 + \cdots \left(\frac{5}{9}\right)^k\right]$

$T(n) = n^2 \frac{1 - (5/9)^k}{4/9} = n^2\left(\frac{1 - \left(\frac{9}{5}\right)^{-\ln_2 n}}{4/9}\right) = n^2(1 - 1/n)$

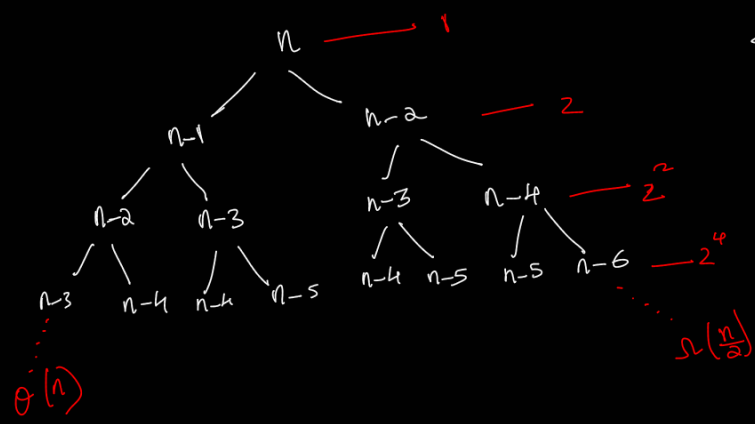
$= O(n^2)$

$k = \Omega(\log_3(n))$

$k = O(\log_{3/2}(n))$

decreasing powers
no need to solve
always → can take as some constant

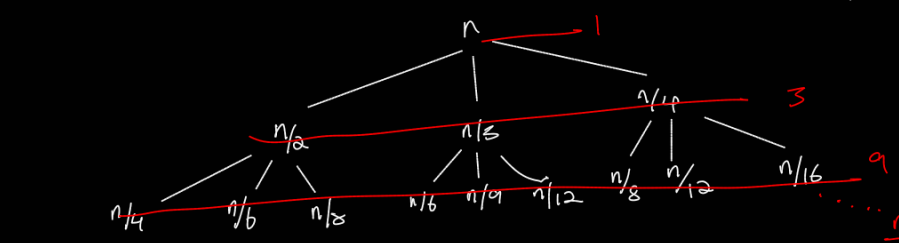
$T(n) = T(n-1) + T(n-2) + 1$



$T(n) = O(1+2+2^2+\cdots 2^n) = \frac{2^{n+1}-1}{2-1} = 2^{n+1}$

$T(n) = O(2^n)$

$T(n) = T(n/2) + T(n/2) + T(n/4) + 1$



$\frac{n}{2^k} = 1$

$2^k = \ln_2(n)$

$T(n) = \Omega(1+3+3^2+3^3+\cdots 3^k)$

$= \frac{3^{k+1}-1}{2} = O(3^{\ln_2 n})$

$= O(n^{\ln_2(3)})$

$\frac{n}{4^k} = 1$

$k = \ln_4 n$

$= \frac{\ln_2 n}{\ln_2 4} = \frac{1}{2}\ln_2 n$

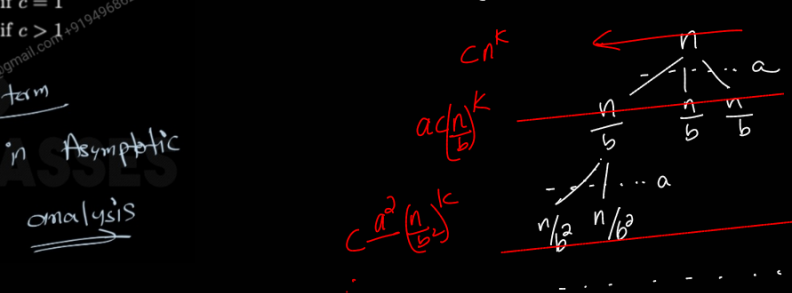
$1 + c + c^2 + \ldots + c^n = \sum_{i=1}^{n} c^i = \begin{cases} \Theta(1), & \text{if } c < 1 \\ \Theta(n), & \text{if } c = 1 \\ \Theta(c^n), & \text{if } c > 1 \end{cases}$

first term

last term

Short cuts for GTP in Asymptotic analysis

$T(n) = aT\left(\frac{n}{b}\right) + cn^k$



$m = \ln_b(n)$

$T(n) = cn^k + ca\left(\frac{n}{b}\right)^k + ca^2\left(\frac{n}{b^2}\right)^k + ca^3\left(\frac{n}{b^3}\right)^k + \ldots$

$ca^m\left(\frac{n}{b^m}\right)^k \qquad m = \ln_b m$

$T(n) = \frac{cn^k\left(\left(\frac{a}{b^m}\right)^{m+1} - 1\right)}{\left(\frac{a}{b^m} - 1\right)}$

$b^m = b^{\ln_b n}$

$b^m = n$

$\frac{a}{b^k} = 8$

$T(n) = cn^k\left[\frac{a\,a^{\ln_b n} - 1}{n\,n^{\ln_b n}}\right]$ — not solvable conventionally

$\frac{a}{n} - 1$

apply shortcut

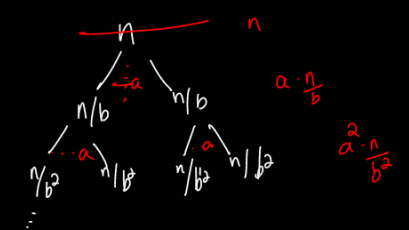
$T(n) = c n^k \sum_{i=1}^{\ln_b n} \left(\frac{a}{b^k}\right)^i$

$= \begin{cases} \Theta(n^k), & a < b^k \\ \Theta(n^k \cdot \ln_b n), & a = b^k \\ \Theta\left(n^k \cdot \left(\frac{a}{b^k}\right)^{\ln_b n}\right), & a > b^k \end{cases}$

Master Theorem

eg:- $T(n) = aT(n/b) + n$

$$T(n) = \begin{cases} \Theta(n) & \text{if } a<b \\ \Theta(n \ln_b n) & \text{if } a=b \\ \Theta(n^{\ln_b a}) & \text{if } a>b \end{cases}$$



$$T(n) = n + \left(\frac{a}{b}\right)n + \left(\frac{a}{b}\right)^2 n + \left(\frac{a}{b}\right)^3 n + \cdots \left(\frac{a}{b}\right)^k n$$

$$T(n) = n \sum_{i=0}^{k} \left(\frac{a}{b}\right)$$

$$T(n) = \begin{cases} \Theta(n) & \text{if } a<b \\ \Theta(n \ln_b n) & \text{if } a=b \\ \Theta(n (a/b)^{\ln_b n}) & \text{if } a>b \end{cases}$$

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

$$n\left(\frac{a}{b}\right)^{\ln_b n} = n\, n^{\ln_b(a/b)} = n^{1 + \ln_b(a) - \ln_b b} = n^{\ln_b(a)}$$

$2^n = \Omega(n^{1+\epsilon})$

$2^n = (n^{\log_n 2})^n = n^{n \log_n 2} = n^{n \cdot \frac{\log_2 2}{\log_2 n}} = n^{\frac{n}{\log_2 n}} = \Omega(n^{1+\epsilon})$

$T(n) = a\,T(n/b) + f(n)$

① if $f(n) = O(n^{\log_b a - \epsilon})$ | $\epsilon > 0$ $\Rightarrow$ $T(n) = \Theta(n^{\log_b a})$

② if $f(n) = \Theta(n^{\log_b a})$ $\Rightarrow$ $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

③ if $f(n) = \Omega(n^{\log_b a + \epsilon})$ | $\epsilon > 0$ $\Rightarrow$ $T(n) = \Theta(f(n))$

---

eg:- $T(n) = aT(\frac{n}{b}) + f(n)$ $\quad f(n) = n^{\log_b a}$ $\Rightarrow$ $T(n) = n^{\log_b a} \cdot \log(n)$

eg: $T(n) = 9T(\frac{n}{3}) + n$ $\quad n^{\log_3 9} = n^2$ $\quad f(n) = \Theta(n^{\log_b a})$ $\Rightarrow$

$T(n) = \Theta(n^2)$

eg: $T(n) = 3T(\frac{n}{4}) + \underbrace{n \log(n)}_{f(n)}$ $\quad n^{\log_b a} = n^{\log_4 3} = \Theta(n) = \Theta(n \log n)$

$\Rightarrow T(n) = \Theta(n \log(n))$ $\quad n \log n = \Omega(n^{\log_a 3})$

$T(n) = 2T(\frac{n}{2}) + \underbrace{n \lg(n)}_{f(n)}$ $\quad n^{\log_b a} = n$

$f(n) \neq \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ } ☆ Cannot apply Master's theorem.

$T(n) = 8T(n/a) + n^2$ $\quad n^{\log_b a} = n^3$

$\exists \epsilon > 0 \mid n^2 = O(n^{3-\epsilon})$ } $\Rightarrow T(n) = \Theta(n^3)$

$T(n) = T\left(\frac{n}{2}\right) + \sqrt{n}$ $\quad n^{\log_b a} = n^0 =$

$n^{1/2} = \Omega(n^{0+\epsilon}) \Rightarrow T(n) = O(\sqrt{n})$

$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \log n$ $\quad n^{\log_b a} = n^2$

$n^3 \log(n) = \Omega(n^{2+\epsilon}) \Rightarrow T(n) = O(n^3 \log(n))$

$T(n) = 4T(n/2) + n^2 \log n$ $\quad n^{\log_b a} = n^2$ } we cannot apply master's theorem

---

$T(n) = 4T(n/2) + n^2$ $\quad n^{\log_b a} = n^2$

$f(n) = \Theta(n^2) \Rightarrow T(n) = n^2 \log n$

$T(n) = T\left(\frac{n}{2}\right) + 2^n$ $\quad n^{\log_2 1} =$

$2^n = \Omega(n^{0+\epsilon}) \Rightarrow T(n) = O(2^n)$

$T(n) = 2T(n/2) + n!$ $\quad n^{\log_2 2} = n^1$

$n! = \Omega(n^1) \Rightarrow T(n) = O(n!)$

$T(n) = 2T(n/3) + [\lg(n)]^2$ $\quad n^{\log_3 2}$

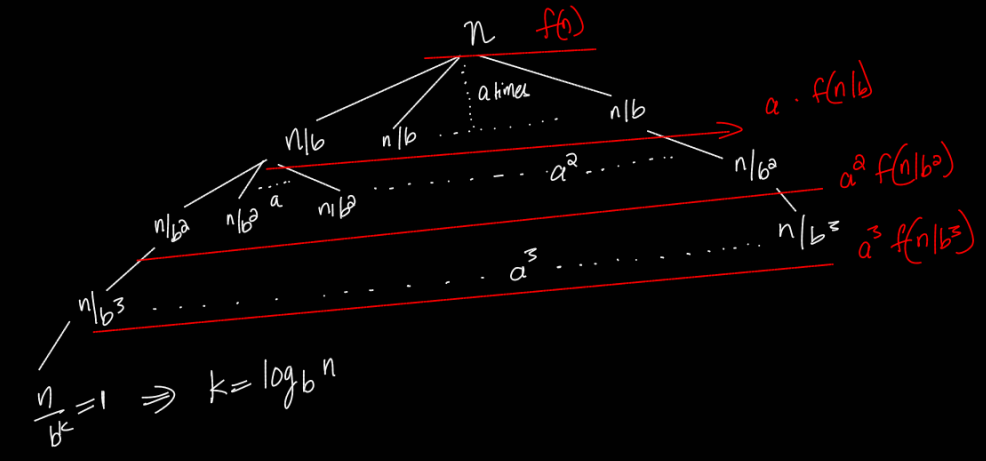
$\lg^2(n) = O(n^{\log_2 2^2 - \epsilon}) \Rightarrow T(n) = O(n^{\log_3 2})$

$T(n) = 4T(n/2) + \overbrace{\Theta(n \log n)}^{f(n)}$ $\quad n^{\log_b a} = n^2$

$f(n) = \Theta(n \log n) = O(n^{2-\epsilon}) \Rightarrow T(n) = O(n^2)$

Masters theorem derivation

$T(n) = aT(n/b) + f(n)$ $\quad a \geq 1$ $b > 1$ $f(n)$ asymptotically +ve



$\frac{n}{b^k} = 1 \Rightarrow k = \log_b n$

$T(n) = f(n) + a\, f(n/b) + a^2 f(n/b^2) + \cdots$

Last term $a^k \cdot f(n/b^k)$

$= a^{\log_b n} \cdot f(1)$

$= n^{\log_b a} \cdot f(1)$

$T(n) = O\left[\sum_{i=1}^{\log_b n} f\left(\frac{n}{b^i}\right) \cdot n^{\log_b a}\right]$

if $f(n) = \Omega(n^{\log_b a + \epsilon})$

$T(n) = O[f(n)]$

Master Theorem (Generalised)

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$. → Generalised version.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

$T(n) = 2T(n/2) + n\log n$ $\quad n^{\log_b a} = n$

$n\log n \neq O(n^{1+\epsilon})$ $\quad \epsilon > 0$

$\neq \Omega(n^{1-\epsilon})$

$n\log n = \Theta(n^1 \log^k n)$ $\quad k = 1$

$\Rightarrow T(n) = \Theta(n\log^2 n)$

$T(n) = 2T(n/2) + n\log^{10} n$ $\quad n^{\log_b a} = n$

$n\log^{10} n \neq O(n^{1+\epsilon})$

$n\log^{10} n \neq \Omega(n^{1-\epsilon})$

$n\log^{10} n = \Theta(n^1 \log^k n)$ $\quad k = 10$

$\Rightarrow T(n) = \Theta(n\log^{11} n)$

---

(3) $T(n) = 9T(n/3) + n^2\log^2 n \Rightarrow T(n) = \Theta(n^2 \log^3 n)$

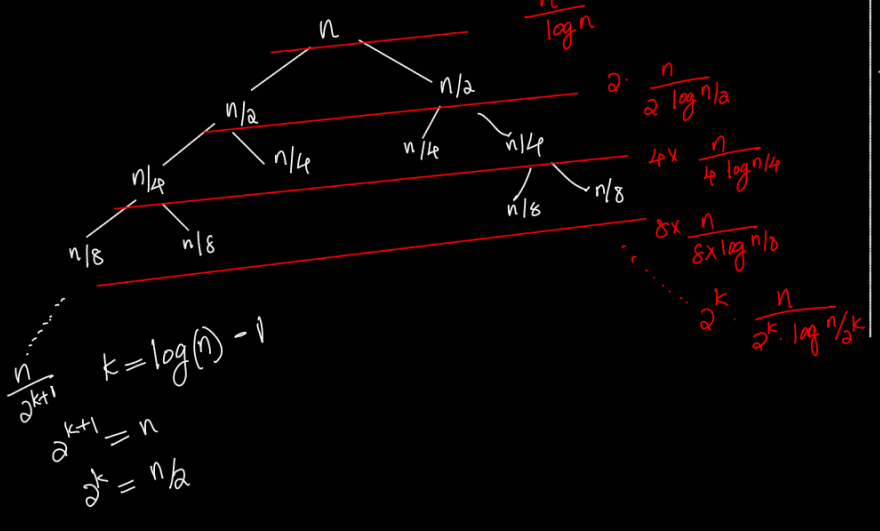
(4) $T(n) = 2T(n/2) + \frac{n}{\log n}$ $\quad n^{\log_b a} = n^1$

$\frac{n}{\log(n)} \neq O(n^{1-\epsilon})$

$\frac{n}{\log(n)} \neq \Omega(n^{1+\epsilon})$

$\frac{n}{\log(n)} = \Theta(n^1 \log^{-1}(n)) \Rightarrow k = -1$ → masks theorem not applicable when $k < 0$

☆☆ ☆☆ only method to solve



$$T(n) = n\left[\frac{1}{\log n} + \frac{1}{\log n/2} + \frac{1}{\log n/4} + \cdots \frac{1}{\log 2}\right]$$

$$T(n) = n \sum_{i=0}^{\ln(n)-1} \left[\frac{1}{\log n - i}\right]$$

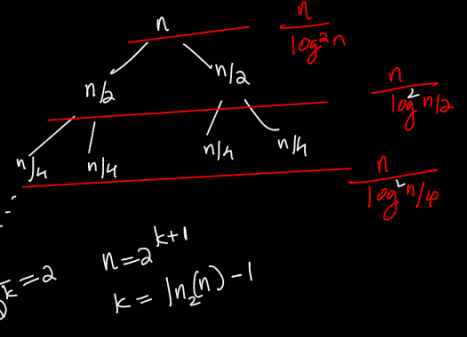
$$T(n) = n \cdot \sum_{i=1}^{\ln(n)} \left(\frac{1}{i}\right) = \Theta(n \lg\lg(n))$$

$$\sum_{i=1}^{n} \frac{1}{i} = \log n$$

$$\sum_{i=1}^{f(n)} \frac{1}{i} = \log f(n)$$

☆☆☆ Remember

Q) $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log^2 n}$



$\frac{n}{2^k} = 2 \quad n = 2^{k+1} \quad k = \log_2(n) - 1$

$$T(n) = n\left[\frac{1}{\log^2 n} + \frac{1}{\log^2\left(\frac{n}{2}\right)} + \frac{1}{\log^2\left(\frac{n}{4}\right)} + \cdots \frac{1}{\log^2 2}\right]$$

$$T(n) = n\left[\frac{1}{\log^2 n} + \frac{1}{(\log n - 1)^2} + \frac{1}{(\log n - 2)^2} + \cdots \frac{1}{(\log n - (k-1))^2} \frac{1}{(\log n - k)^2}\right]$$

$$T(n) = n\left[\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots \frac{1}{\log^2 n}\right]$$

$$T(n) = n \boxed{\sum_{i=1}^{\log n} \left(\frac{1}{i^2}\right)} = \Theta(n)$$

$= \text{Constant}$

The sum of squared inverses up to n, $S_n = \sum_{k=1}^{n} \frac{1}{k^2} = 1 + \frac{1}{4} + \frac{1}{9} + \ldots + \frac{1}{n^2}$, has an asymptotic notation of $\Theta(1)$.

☆☆

The series is convergent, meaning that as n approaches infinity, the sum approaches a finite constant value, which is a key aspect of its asymptotic behavior.

Explanation

- **Convergence:** The infinite sum of the series is a well-known result from the Basel Problem: $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. Basel problem
- **Asymptotic Behavior:** Since the sum approaches a constant ($\pi^2/6$) as n becomes very large, the growth rate of the partial sum S_n is constant relative to n.

$T(n) = 2T(n/2) + n\log^k(n) = \Theta(n \log^{k+1} n)$

$T(n) = aT(n/a) + n\log^k(n) = \Theta(n \log^{k+1} n)$

$T(n) = 2T(n/2) + \frac{n}{\log n} = O(n \log\log n)$

$T(n) = 2T(n/2) + \frac{n}{\log^2 n} = O(n)$

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are 3 cases:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$ with $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$.
   - 2.2 If $f(n) = \Theta\left(\frac{n^{\log_b a}}{\log n}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \lg\lg n\right)$.
   - 2.3 If $f(n) = \Theta\left(\frac{n^{\log_b a}}{(\log n)^p}\right)$ with $p \geq 2$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.

   extra
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ with $\epsilon > 0$, then $T(n) = \Theta(f(n))$.

Extended

$T(n) = 4T(n/2) + \frac{n^2}{\log(n)^2} \qquad n^{\log_b a} = n^2$

$\frac{n^2}{\log(n)^2} \neq O(n^{2-\epsilon})$

$\neq \Omega(n^{2+\epsilon})$

$= \Theta\left(\frac{n^2}{\log(n)^p}\right) \quad p \geq 2 \Rightarrow T(n) = \Theta(n^2)$

$T(n) = 9T(n/3) + n^2\log^2(n) \qquad n^{\log_b a} = n^2$

$n^2\log^2(n) = \Theta(n^2 \log^k n) \quad k = 2 \quad k \geq 0$

$\Rightarrow T(n) = \Theta(n^2 \log^3 n)$

$T(n) = 2T(n/2) + \frac{n}{\log^3 n} \Rightarrow T(n) = \Theta(n)$

$T(n) = 4T(n/2) + \frac{n}{\log^2(n)}$ $\quad \frac{n}{\log^2(n)} = \theta(n^{2-\epsilon}) \Rightarrow T(n) = \theta(n^2)$

$T(n) = T(n/2) + 1$ $\quad n^{\log_2 1} = n^0$

$1 = \theta(n^0) \Rightarrow T(n) = \log(n)$

Recurrences not Solvable using the Master Theorem

Example 1: $T(n) = \sqrt{n}\,T\left(\frac{n}{2}\right) + n$

$a = \sqrt{n}$ is not a constant

Example 2: $T(n) = 2T\left(\frac{n}{\log n}\right) + n^2$

$b = \log n$ is not a constant

Example 3: $T(n) = \frac{1}{2}T\left(\frac{n}{2}\right) + n^2$

$a = \frac{1}{2}$ is not ≥ 1

Example 4: $T(n) = 2T\left(\frac{4n}{3}\right) + n$

$b = \frac{3}{4}$ is not > 1.

Example 5: $T(n) = 3T\left(\frac{n}{2}\right) - n$

$f(n) = -n$ is not positive

Example 6: $T(n) = 2\,T\left(\frac{n}{2}\right) + n^2 \sin n$

violates regularity condition of case 3

Example 7: $T(n) = T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{4}\right) + n$

a and b are not fixed

$aT(\frac{n}{b})$

$a, b \rightarrow$ Constant > 1

$a \geq 1$, $b > 1$, $k \geq 0$, $p \geq a$

$aT(n/b) + f(n)$

2.1 $\rightarrow f(n) = \theta\left(n^{\log_b a} \cdot \log^k n\right) \Rightarrow T(n) = \theta\left(n^{\log_b a} \log^{k+1} n\right)$

2.2 $f(n) = \theta\left(\frac{n^{\log_b a}}{\log n}\right) \Rightarrow T(n) = \theta(n^{\log_b a} \log\log n)$

2.3 $f(n) = \theta\left(\frac{n^{\log_b a}}{\log^p n}\right) \Rightarrow T(n) = \theta(n^{\log_b a})$

Change of Variable

$T(n) = T(\sqrt{n}) + \log n \Rightarrow T(2^m) = T(2^{m/2}) + m$

assume $n = 2^m$

$m = \ln_2 n$

assume $T(2^m) = S(m)$ $\quad S(m) = S(m/2) + m$

Using MT

$m = \Omega(m^{0+\epsilon})$

$\Rightarrow S(m) = \theta(m)$

$T(n) = \theta(\log_2 n)$

if $T\left(\frac{2^m}{k}\right) = S\left[\log_2\left(\frac{2^m}{k}\right)\right] = S(m - \log_2 k)$

---

eg. $T(n) = T(\sqrt{n}) + \sqrt{n} \rightarrow T(2^m) = T(2^{m/2}) + 2^{m/2}$

$2^m = n$

$m = \log_2 n$

$S(m) = S(m/2) + 2^{m/2}$

$m^{\log_2 1} = m^0$

$2^{m/2} = \Omega(m^{0+\epsilon})$

$\Rightarrow S(m) = \theta[2^{m/2}]$

$T(n) = \theta[\sqrt{n}]$

---

eg. $T(n) = 2T(\sqrt{n}) + \log n \Rightarrow T(2^m) = 2T(2^{m/2}) + m$

$2^m = n$

$m = \log_2 n$

$S(m) = 2S(m/2) + m$

$m^{\log_2 2} = m^1$

$m = \theta(m^1) \Rightarrow S(m) = \theta(m^1 \log m)$

$T(n) = \theta(\log_2 n \cdot \log\log n)$

$T(n) = T(\sqrt{n}) + \log\log n \longrightarrow T(2^m) = T(2^{m/2}) + \log m$

$S(m) = S(m/2) + \log m$

$n = 2^m$

$m = \log_2 n$

$m^{\log_2 1} = m^0$

$\log m \neq O(m^{0-\epsilon}) \mid \epsilon > 0$

$\log m \neq \Omega(m^{0+\epsilon})$

$\log m = \Theta(m^0 \cdot \log^k m)$, $k = 1$

$S(m) = \Theta(m^0 \cdot \log^2 m)$

$T(n) = \Theta(1 \cdot \log^2 \log n)$

$T(n) = 3T(\sqrt{n}) + 1 \qquad T(2^m) = 3T(2^{m/2}) + 1$

$S(m) = 3T(m/2) + 1$

$2^m = n$

$m = \log_2 n$

$1 = O(m^{\log_2 3 - \epsilon})$

$\Rightarrow S(m) = \Theta(m^{\log_2 3})$

$T(n) = \Theta((\log_2 n)^{\log_2 3})$

$T(n) = T(\sqrt{n}) + n^2 \qquad T(2^m) = T(2^{m/2}) + 2^{2m}$

$S(m) = S(m/2) + 4^m$

$2^m = n$

$m = \log_2 n$

$m^{\log_2 1} = m^0$

$4^m = \Omega(m^{0+\epsilon})$

$\Rightarrow S(m) = \Theta(4^m)$

$S(m) = \Theta(n^2)$

$T(n) = 2T(\sqrt{n}) + \log^3 n \qquad T(2^m) = 2T(2^{m/2}) + m^3$

$S(m) = 2S(m/2) + m^3$

$2^m = n$

$m = \log_2 n$

$m^3 = \Omega(m^{1+\epsilon}) \Rightarrow S(m) = \Theta(m^3)$

$T(n) = \Theta(\log_2^3 n)$

$T(n) = 4T(\sqrt{n}) + \log^2 n \qquad S(m) = 4S(m/2) + m^2$

$m^2 = \Theta(m^{\log_2 4}) \Rightarrow S(m) = \Theta(m^2 \cdot \ln m)$

$T(n) = \Theta[\ln^2(n) \cdot \ln\ln(n)]$

$T(n) = 12T(n^{1/3}) + \log^2 n \qquad S(m) = 12S(m/3) + m^2$

$m^2 = \Theta(m^{4-\epsilon}) \Rightarrow S(m) = \Theta(m^4)$

$T(n) = \Theta(\ln^4 n)$

GATE CSE 2006

$S(m) = 2S(m/2) + 1 \Rightarrow S(m) = \Theta(m)$

$1 = O(m^{1-\epsilon}) \qquad T(n) = \Theta(\ln n)$

Consider the following recurrence:

$T(n) = 2T(\sqrt{n}) + 1, T(1) = 1$

Which one of the following is true?

A. $T(n) = \Theta(\log\log n)$
B. $T(n) = \Theta(\log n)$ ✓
C. $T(n) = \Theta(\sqrt{n})$
D. $T(n) = \Theta(n)$

(c) $T(n) = 161^2 \cdot T(\sqrt[161]{n}) + 161 \cdot (\log n)^2$.

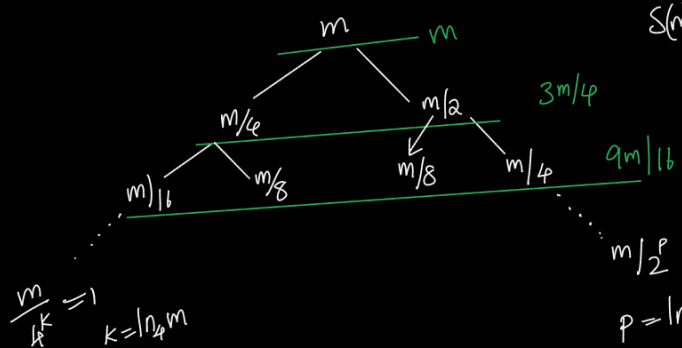
$2^m = n$
$m = \log_2 n$

$S(m) = 161^2 \, S\left(\frac{m}{161}\right) + 161 \cdot m^2$

$161 m^2 = \Theta(m^2) \Rightarrow S(m) = \Theta(m^2 \cdot \ln m)$
$= \Theta(\ln^2 n \cdot \ln \ln n)$

(f) $T(n) = T(\sqrt[4]{n}) + T(\sqrt{n}) + \log n$.

$S(m) = S(m/4) + S(m/2) + m$



$S(m) = m + \frac{3}{4}m + \left(\frac{3}{4}\right)^2 m + \left(\frac{3}{4}\right)^3 m + \cdots \left(\frac{3}{4}\right)^p m$

$S(m) = m \cdot \left[\left(\frac{3}{4}\right)^0 + \left(\frac{3}{4}\right)^1 + \left(\frac{3}{4}\right)^2 + \cdots \left(\frac{3}{4}\right)^p\right]$

GP $R < 1$ do not solve it is constant

$S(m) = \Theta(m)$
$T(m) = \Theta(\ln n)$

$\frac{m}{4^k} = 1 \quad k = \log_4 m$

$m/2^p \quad p = \log_2 m$

(d) $T(n) = T\left(\frac{n}{161}\right)^{161} \cdot n$ ★

Take log

$\log T(n) = 161 \log T\left(\frac{n}{161}\right) + \log n$

$G(n) = 161 \, G\left(\frac{n}{161}\right) + \log n$

$\log n = \Theta(n^{1-\varepsilon})$

$\Rightarrow G(n) = \Theta(n)$
$T(n) = \Theta(2^n)$

$\log T(n) = G(n)$
$T(n) = e^{G(n)}$

(e) $T(n) = \sqrt{n}\, T(\sqrt{n}) + 100n$

divide by n

$2^m = n$
$m = \ln n$ ★

$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 100$

$S(m) = S(m/2) + 100$

$S(m) = \Theta(\log_2 m)$
$\frac{T(n)}{n} = \Theta(\ln \cdot \ln n)$
$T(n) = \Theta(n \cdot \ln \ln n)$

(f) $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

$\frac{T(n)}{n} = \frac{T\sqrt{n}}{\sqrt{n}} + 1$

$S(m) = S(m/2) + 1$

$S(m) = \Theta(\log m)$
$\frac{T(n)}{n} = \Theta(\log \log n)$
$T(n) = \Theta(n \log \log n)$

GATE IT 2008 | Question: 44

When $n = 2^{2k}$ for some $k \geqslant 0$, the recurrence relation $T(n) = \sqrt{(2)}T(n/2) + \sqrt{n}, T(1) = 1$ evaluates to :

A. $\sqrt{(n)}(\log n + 1)$
B. $\sqrt{(n)} \log n$
C. $\sqrt{(n)} \log \sqrt{(n)}$
D. $n \log \sqrt{n}$

$T(n) = \sqrt{2}\, T(n/2) + \sqrt{n} \qquad T(1) = 1$

$n^{\log_2 \sqrt{2}} = n^{1/2} = \Theta(\sqrt{n})$

$\Rightarrow T(n) = \Theta(\sqrt{n} \cdot \ln n)$

Divide & Conquer algorithms

→ Bubble sort → Idea → Time complexity
→ Stable, In place
→ Optimisation using flag → complexity change? → early termination
  - Best case → O(n)
  - Worst case no change O(n²)

→ Merge sort (Recursive merge)
→ Idea → Recursive definition
→ Complexity → #levels → log(n) → cost of each level — O(n)
→ Partition not binary → k partitions with n/k elements
→ Merge algorithm → 5 ways to merge given k partitions of n/k
1) • Concatenate & Sort — n log n
2) • Bottom-up — n log k
3) • Successive merge — nk
4) • Heap Merge — n log k
5) •

→ Bottom up merge (iterative merge)
→ Sorting Bitonic array using merge sort in O(n)
Find t reverse merge

→ Selection sort → Complexity — O(n²)
→ Stable, In place
No → why.

→ Insertion sort → Idea
→ Complexity — O(n²) Best case O(n)
→ Optimisation for array — search — O(log n) insert — O(1) } per step
→ Optimisation for linked list — search — O(n) insert — O(1)
→ Connection with inversion — O(n+I) I = #inversions

→ Inversion → definition
→ Trick using diagram → largest possible given n size array = $\frac{n(n-1)}{2} = {}^nC_2 = O(n^2)$
→ Inversion & Insertion sort → Counting inversion in bounding inversion in bitonic array.
→ Counting inversion — algorithm → Brute force
→ Using merge sort
→ # Inversions = # Inversions + # elements in left array whenever element in right array is added to merge buffer.

→ Swapping X.....Y & change in inversion count
if x<y — max increase = 2x difference in index − 1
if x>y — max decrease = 〃

Given inversion (i,j) — make at least j−i inversions

Min Heap → Definition → Complete Binary tree
→ All children smaller than parent — Minheap
→ Inplace array implementation always — i — parent
2i+1 — left child
2i+2 — right child
child i → parent ⌊(i−1)/2⌋

1. Find max / Find min → O(1)
2. Insert 1 element → O(log n) — n → size of heap → Insert at end of heap array → Sift up.
3. make heap → O(n) — n → size of input array
   ① Assume input array as complete binary tree
   ② Start with last non leaf node. This is at ⌊n/2⌋ − 1
   ③ Use heapify from $\frac{n}{2}-1$ to 0 in the input array.
4. Remove 1 element — O(log n)
   ↓ exchange root with last element of array — call heapify on root.

Sift down
O(log n) ← heapify (assumes, subtrees satisfy heap property, only current root node violates)
n → size of the current subtree that heapify is called on
→ ① Find largest among root, left child, right child
② if root is not largest, swap with largest child
③ Call heapify for affected child — subtree

Quick sort

```
QS(a, left, right) {
    if (left = right) return;
    # swap(A[left], A[random(left, right)])
    pivot = partition(a, left, right)
    QS(a, left, pivot-1)
    QS(a, pivot+1, right)
}
```

Randomized quick sort — No change for partition function

```
partition(a, left, right)        {
    pivot = left; left++
    while (left <= right) {
        while a[left] <= a[pivot]   && left <= right   left++
        while a[right] > a[pivot]   && right >= left   right--
        swap(a[left], a[right])   left++; right--;
    }
    swap a[right], a[pivot]
    return right
}
```

Worst case pivot ends up in one of the edges

→ $T(n) = T(n-1) + n$ | Which input produces worst case

$T(n) = O(n^2)$

→ Best case $T(n) = 2T(n/2) + n$

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

Unstable Inplace

partition — in general need not choose the first element as pivot always.

$P[\text{The smallest partition has size} \geq \alpha n]_{\text{for random pivot}} = 1 - 2\alpha \quad 0 < \alpha < 1$

→ Sorted array

all elements out of n equally likely.

$\Rightarrow$ $n - 2\alpha n$ favorable choices

$\Rightarrow$ $P(\text{Smallest(partition size)} > \alpha n) = 1 - 2\alpha$

→ Average case complexity of Quick sort

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left(\frac{2}{j-i+1}\right) = O(n \log n)$$

$i < j$

→ Select k^{th} smallest element using partition algorithm. → $O(n^2)$ — worst case ($T(n) = T(n-1) + n$); $O(n)$ — best case ($T(n) = T(n/2) + n$) — that's why used

↳ Naive method → sort $O(n \log n)$ → Find k^{th} smallest

→ Binary search — algo — recursive

↳ Complexity — Best case, worst case, Avg case

Heap Sort → Make heap in $O(n)$
→ Remove Max/Min $O(\log n)$ → $T(n) = \log n + \log(n-1) + \log n-2 \cdots \log 2$
$= \log(n!) = O(n \log n)$
→ Stable → No
→ Inplace → Yes

Sorting algorithm	Best Case	Average Case	Worst Case	Stable	Inplace
Bubble sort	$O(n)$*	$O(n^2)$	$O(n^2)$	✓	✓
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	X	✓
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	✓	✓
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	X	✓
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	X
Quick sort	$O(n \log n)$	$O(n \log n)$ ☆	$O(n^2)$ #	X	✓

* → With optimisation flag.
# → Pivots are consistently going to the sides $T(n) = T(n-1) + n$
☆ → Pivots are on average splitting the data $T(n) = 2T(n/2) + n$

Linear Time Sorting

Counting — Assumption — Integers values $\leq k$ → k bucket idea
Complexity — $O(n+k)$
Stable — ✓
Inplace — X

Radix — Integers $\leq M^t$ → Counting sort on LSB first, then progressively until MSB.
→ Since count sort is stable, at any stage t, if the t^{th} digit is same, then arrangement will be in the order of $t-1^{th}$ stage.

Complexity → t stages ($t_{last} = \log_{base}(M^t)$ ⇒ last stage = $t \log_{base}(M)$
→ Each stage counting sort $= O(B+n) = O(n)$ | B = #buckets, B = sorting base
→ Radix sort ($\leq B^t$) $= O((B+n) t \log_B M)$

$= O(n t \log_B M)$

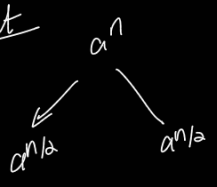
$= O(n t \log_B n)$ if $M = n$

$= O(nt)$ linear if $B = n$

Closest Pairs

- Brute force method → Find distance between all possible pairs — nC_2 pairs $= O(n^2)$
  - $O(n^2)$
  - Keep a note of the minima when finding — $O(1)$
- 1-d case → $O(n \log n)$
  - sort — $O(n \log n)$
  - Find distance between only $n-1$ adjacent pairs — $O(n)$

Exponent



$T(n) = T(n/2) + O(1)$

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

```
exponent(a, n):
    if(n==0):
        return 1
    if(n==1):
        return a
    X = exponent(a, n/2)
    if even(n):
        return X*X
    else:
        return a*X*X
```

Matrix multiplication $A_{n \times n} B_{n \times n} = C_{n \times n}$

- Normal way $= O(n^3)$ # multiplication
  (n multiplications & n-1 addition for finding 1 value of result.)
- Divide & Conquer

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix}$$

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

$T(n) = O(n^3)$

Current theoretical best is $O(n^{2.371339})$ year 2024 by Coppersmith

---

Graph Algorithms

→ Directed vs Undirected graph

- Directed
  - Connected graph
  - Weakly connected
  - Complete
- Undirected
  - Connected graph — path from every u to V
  - Completely connected — all possible edges u, V

V = # nodes
E = # edges

→ Nodes, Vertices → How to represent & store a graph

1. Adjacency Matrix (A) = V×V → a_{ij} = weight of edge from i to j
   - A — symmetric for undirected graph
2. Incidence Matrix (I) = V×E
3. Adjacency list

① I_{ij} = weight if edge i enters node j

② I_{ij} = −weight if edge i leaves node j.

③ Sum of any column = 0

$L = I I^T \Rightarrow k_{ij} = \sum_{a=1}^{E} I_{ia} I_{ja}$ → dot product of i^{th} & j^{th} nodes

(V×V)

only for unweighted graph ⇒ A(0,1)

$k_{ij} = 0$ → nodes have no common edge

$= $ Degree of node, $i = j$ & weights = 1

$= -\text{weight}^2$ edge exist btw node i & j

$= -1$ if weights = 1

⇒ II^T → Laplacian Matrix

$A = D - L$ } for simple graph

$L = D - A$ }

Adjacency list

→ Total memory required for linked list = $2E$ → undirected
E → directed

→ Total memory = $O(V+E)$

	Adjacency List	Adjacency Matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test $u \to v \in E$	$O(\text{degree}(u))$	$O(1)$
List V neighbours	$\Theta(\text{degree}(V))$	$O(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge $u \to v$	$O(1)$	$O(1)$
Delete edge $u \to v$	$O(\deg(u)+\deg(V))$	$O(1)$

→ In most applications of graph, generating neighbours is the main theme

Preferred Method

Dense graph → $|E| \approx |V|^2$

$|E| \leq |V|\text{-}1C_2$

$|E| \leq \frac{|V|(|V|-1)}{2}$

upper bound on #edges in a undirected simple graph.

Sparse graph $|E| \approx |V|$

Search → Why search → Most problems related to graph is reducible to search

DFS → start/discovery time $d(v)$ → when the node is expanded & children generated
→ Finish time $f(v)$ → When the last node in search subtree of v is popped from stack

DFS forest

properties ① if $f(v) = d(v)+1$ ⇒ no children ⇒ v is a leaf in search tree
⇒ v has no neighbours in graph.

② if $d(u) \in [d(v), f(v)]$ ⇒ u is in V subtree
⇒ $f(u)$ also in $[d(v), f(v)]$

③ if $d(u) < d(v)$ → u discovered before V
① $f(u) > f(v)$ → V is in U subtree
② $f(u) < d(v)$ → U & V are in separate search subtrees when considering either of them as subtree root
③ $d(v) < f(u) < f(v)$ — not possible

(f) Given an n-vertex DAG with a single source, what is the maximum number of trees in the DFS search forest among any ordering of the vertices (in DFS)?

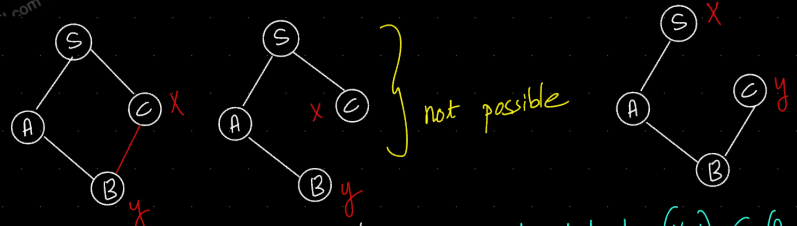
Solution: n. If the vertices are arranged in reverse topological order each call of explore from depth-first-search only explores one vertex.

(e) If a root r in a DFS tree of undirected graph G has more than one child, then $G - r$ (the graph resulting from removing r and any incident edges from G) has more than one connected component.

Solution: True. This is from the homework. Basically, there is no path from the descendants of the first child to any subsequent child that does not use the root r, as explore would have explored that path.

8. Let T be a depth -first search tree of a undirected graph. Let (x, y) be an edge of G that is not an edge of T, then one of x or y is an ancestor of the other. True / False

Solution. True .



G – graph T – tree

x = ancestor of (y)

undirected edge $(x,y) \in G$ not in T because
→ either X is visited first in tree or Y is visited first in tree
→ X is visited & Y is not visited but still edge x,y is not used before finishing X

Finishing time

(c) In a DFS on a directed graph $G = (V, E)$, if u was explored before v, and there is a path from u to v, $post(v)$ is greater than $post(u)$. False

→ This means Y has to be visited using some other path before finishing X

(d) Suppose DFS is called on a DAG G, and the resulting DFS forest has a single tree. Then G has exactly one source vertex. True

(e) If a root r in a DFS tree of undirected graph G has more than one child, then $G - r$ (the graph resulting from removing r and any incident edges from G) has more than one connected component. True

(f) Let T be any tree which contains all the vertices of a connected undirected graph G. There is a way to break ties in DFS that will output T.

DAG $\Rightarrow \exists$ path $(u,v) \Rightarrow \nexists$ path (v,u)

47

Let G be an undirected graph. Consider a depth-first traversal of G, and let T be the resulting depth-first search tree. Let u be a vertex in G and let v be the first new (unvisited) vertex visited after visiting u in the traversal. Which of the following statement is always true?

A. $\{u, v\}$ must be an edge in G, and u is a descendant of v in T
B. $\{u, v\}$ must be an edge in G, and v is a descendant of u in T
C. If $\{u, v\}$ is not an edge in G then u is a leaf in T
D. If $\{u, v\}$ is not an edge in G then u and v must have the same parent in T

A depth-first search is performed on a directed acyclic graph. Let $d[u]$ denote the time at which vertex u is visited for the first time and $f[u]$ the time at which the DFS call to the vertex u terminates. Which of the following statements is always TRUE for all edges (u, v) in the graph ?

A. $d[u] < d[v]$
B. $d[u] < f[v]$
C. $f[u] < f[v]$
D. $f[u] > f[v]$

DFS Implementation → Recursion →

u, v are nodes

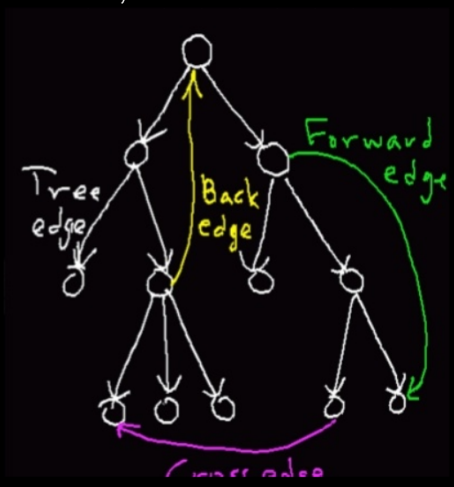
$T(u) = \sum_{v = u \text{ neighbour}} T(v) + O(\text{degree } v)$

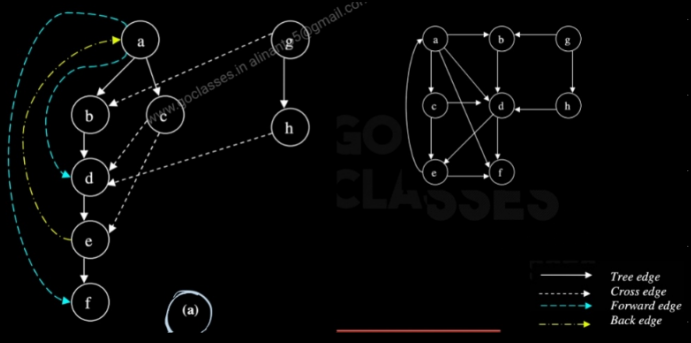
$T(u) = O(V+E)$ — using adjacency list

$T(u) = O(V^2)$ — using adjacency matrix

```
DFS(G, a, goal, visited):
  if a == goal:
    return [a]

  mark a as visited

  for each neighbor x of a in G:
    if x not in visited:
      path = DFS(G, x, goal, visited)
      if path ≠ None:
        return [a] + path

  return None
```

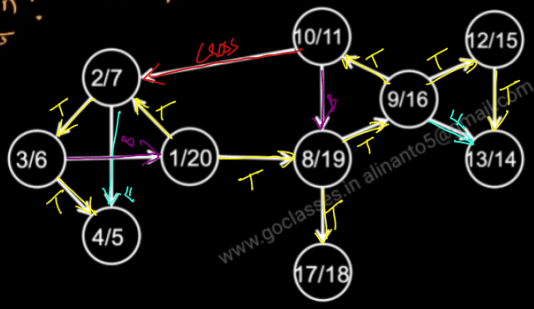
→ Parenthesis theorem — only 3 possible cases





Example : - Discovery time/finishing time numbers of DFS are given for directed graph. Classify edges

Given : Graph and $d[u]$ / $f[u]$ Ask : classify edges



Cross edge → $d(u) < f(u) < d(v) < f(v)$

$v \to u$ cross

$d.u < d.v < f.v < d.v$

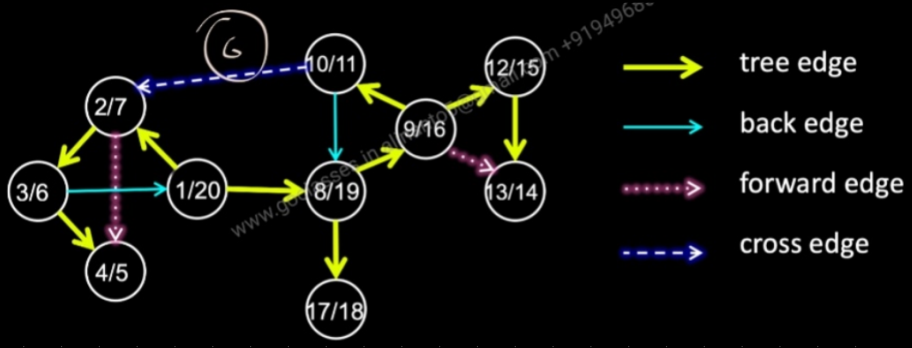
$u \to v \notin T \Rightarrow$ Forward

$d.u < d.v < f.v < d.v$

$v \to u \in T \Rightarrow$ back

- A ***forward edge*** is a non-tree edges (x, y) such that $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.
- A ***backward edge*** is a non-tree edge (x, y) such that $\text{pre}(y) < \text{pre}(x) < \text{post}(x) < \text{post}(y)$.
- A ***cross edge*** is a non-tree edges (x, y) such that that the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are disjoint.

Forward Edge & Cross Edge Not in undirected graph



3. Let G be undirected graph with n vertices and m edges.

   a. True or false: All its DFS forests (for traversals starting at different vertices) will have the same number of trees?

   b. True or false: All its DFS forests will have the same number of tree edges and the same number of back edges?

3. a. The number of DFS trees is equal to the number of connected components of the graph. Hence, it will be the same for all DFS traversals of the graph.

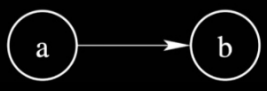
   b. For a connected (undirected) graph with $|V|$ vertices, the number of tree edges $|E^{(tree)}|$ in a DFS tree will be $|V|-1$ and, hence, the number of back edges $|E^{(back)}|$ will be the total number of edges minus the number of tree edges: $|E|-(|V|-1) = |E|-|V|+1$. Therefore, it will be independent from a particular DFS traversal of the same graph.

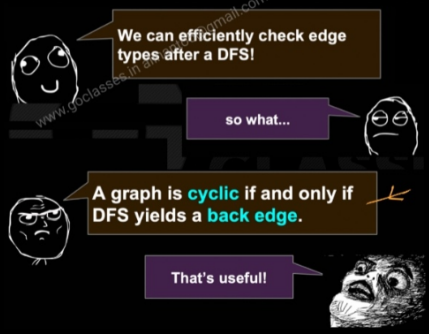
☆ back edge ⇒ atleast one cycle in graph both directed & undirected.

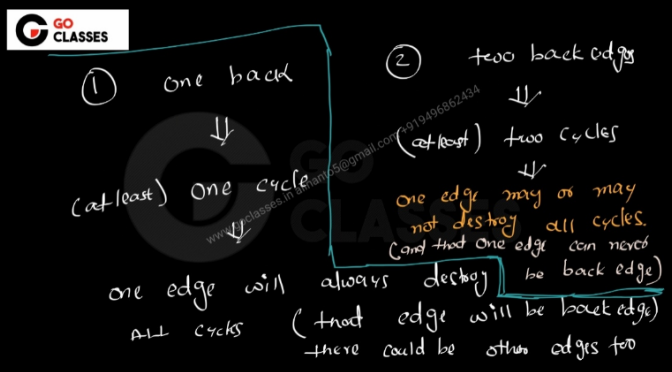
n back edge ⇒ Atleast n cycles in graph

T F A depth-first search of a directed graph always produces the same number of tree edges (i.e. independent of the order in which the vertices are provided and independent of the order of the adjacency lists).
Explain:

Solution: **False.** The DFS forest may contain different numbers of trees (and tree edges) depending on the starting vertex and upon the order in which vertices are searched.
Consider the example below. If the DFS starts at a, then it will visit b next, and (a, b) will become a tree edge. But if the DFS visits b first, then a and b become separate trees in the DFS forest, and (a, b) becomes a cross edge.



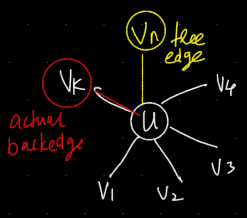




T F If a directed graph G is cyclic but can be made acyclic by removing one edge, then a depth-first search in G will encounter exactly one back edge.

Solution: False. You can have multiple back edges, yet it can be possible to remove one edge that destroys all cycles. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (b, a), (c, a)\})$, there are two cycles ($[a, b, a]$ and $[a, b, c, a]$) and a DFS from a in G returns two back edges ((b, a) and (c, a)), but a single removal of edge (a, b) can disrupt both cycles, making the resulting graph acyclic.

⟶ Cycle detection using dfs — in undirected graph




```
DFS(u)
visited[u] = true
for each v adjacent to u
    if not visited[v]:
        parent[v] = u
        DFS(v)
    else if parent[u] != v
        cycle = found;
```

Reaching an already visited node in DFS may be a due to going back a tree edge or a back edge. If it is a tree edge that we are going back up, then we would have marked current node as the child of the subject V.

Cycle detection using dfs → directed graph — DFS(u):

discovered but not finished[u] = true;
visited[u] = true
for each v adjacent to u:
  if not visited[v]:
    DFS[v]
  else # already visited → could be cross, back or forward edge
    if discovered but not finished[v]
      return cycle
discovered but not finished = false

cross edges u→v — v will already be discovered but finished
d.v < f.v < d.u < f.u

forward edge u→v — v will already be discovered but finished
d.u < d.v < f.v < f.u

back edge u→v — v will be discovered but not finished
d.v < d.u < f.u < f.v

DAG → Directed Acyclic graph

① ∃ path(u,v) ⇒ ∄ path(v,u) → no cyclics ⇒ No backedge

② No back edge (u,v) ⇒ any edge u→v in any case

u→v edge ⇒ f.u > f.v

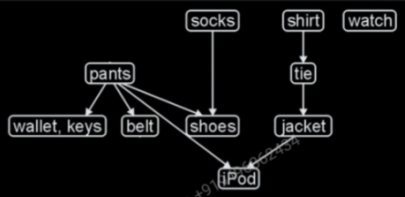
We can't generalise discover time

① d.u < d.v < f.v < f.u → Forward edge / Tree edge

× ② d.v < d.u < f.u < f.v → Backedge not possible

③ d.v < f.v < d.u < f.u → cross edge

A topological sort is a linear ordering of vertices in a **Directed Acyclic Graph (DAG)** where for every directed edge from vertex u to vertex v ($u \to v$), u always comes before v in the sequence, essentially mapping out dependencies like tasks that must be completed in order (e.g., prerequisites). It only works on DAGs because cycles make a valid linear order impossible, and multiple valid sorts can exist for the same graph.

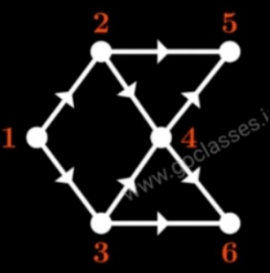


- One topological sort is:
  - pants, wallet, keys, belt, socks, shoes, shirt, tie, jacket, iPod, watch
- A more reasonable topological sort is:
  - socks, pants, shirt, belt, tie, jacket, wallet, keys, iPod, watch, shoes

GATE CSE 2007

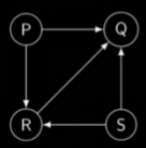
20

Consider the DAG with $V = \{1, 2, 3, 4, 5, 6\}$ shown below.



GATE CSE 2014

Consider the directed graph below given.



Which one of the following is **TRUE**?

A. The graph does not have any topological ordering.
B. Both PQRS and SRQP are topological orderings.
C. Both PSRQ and SPRQ are topological orderings. ✓
D. PSRQ is the only topological ordering.

Kahn's algo → Find topo sort without DFS.

How Kahn's Algorithm Works (Step-by-Step)

1. **Calculate In-Degrees:** Compute the in-degree for every vertex in the graph (number of incoming edges).
2. **Initialize Queue:** Add all vertices with an in-degree of 0 to a queue.
3. **Process Nodes:** While the queue is not empty:
   - Dequeue a vertex `u` and add it to the result list (topological order).
   - For each neighbor `v` of `u`:
     - Decrement the in-degree of `v`.
     - If `v`'s in-degree becomes 0, enqueue `v`.
4. **Check for Cycles:** If the count of vertices in the sorted list equals the total number of vertices, a valid topological sort is found; otherwise, the graph contains a cycle.

Which of the following is not a topological ordering?

A. 1 2 3 4 5 6
B. 1 3 2 4 5 6
C. 1 3 2 4 6 5
D. 3 2 4 1 6 5

Find topological sort orders using DFS →

1. Call DFS to compute finishing time f[v] for every vertex
2. As every vertex is finished insert it onto the front of a linked list
3. Return the list

Decreasing orders of finishing time is a valid topological orders

u→v ∈ DAG(G) ⇒ f.u > f.v

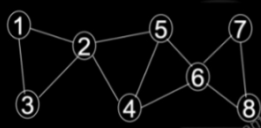
Cut Vertex or Articulation Point (Undirected)

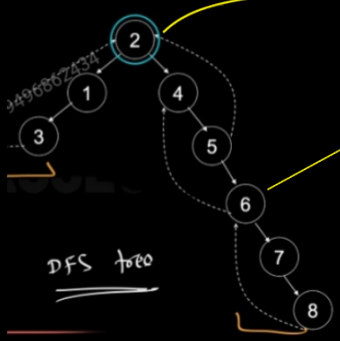
↳ A vertex whose removal disconnects G

Brute force method to check → for all vertex i in G
remove i and all edges associated with i.
DFS(i)
if 2 trees ⇒ disconnected
articulation.append[vertex i]

↳ $O(V(V+E)) = O(V^2+VE)$

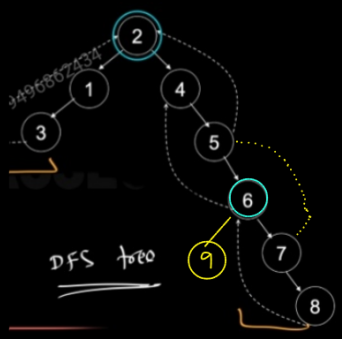
DFS approach → property → if in a undirected graph G
DFS(G) → Root has atleast 2 children
⇒ Even with all possible back edges, removing Root splits G.





Root has atleast 2 children ⟺ Root is articulation point
Root has only one child ⟺ Root is not articulation point

② A non Root vertex in DFS tree is an articulation point if atleast one it's subtree is not connected to any of it's ancestors through any back edge



③ A leaf node of a DFS tree can never be articulation point in a undirected graph.

Run DFS ⇒ make DFT ⇒ Pick leaf node say v.
Run DFS again but this time using v as root ⇒ How many children will be there in 2nd run of DFS on v as root.
⇒ Answer 1

True/False

false

u is an articulation point If and only If in DFS tree,
All of its descendent should connect to its ancestor's through u only.
In other words, there should not be any direct edge in graph from any of the descendent to any of the ancestor.

Correct statement: Atleast one of it's child subtrees should be connected to it's ancestors through u only.

There should not be any direct edge in graph from all of it's child subtrees to any of it's ancestors

If u is an articulation point in G such that x is an ancestor of u in T and y is a descendent of u in T, then all paths from x to y in G must pass through u. false

MSQ

13

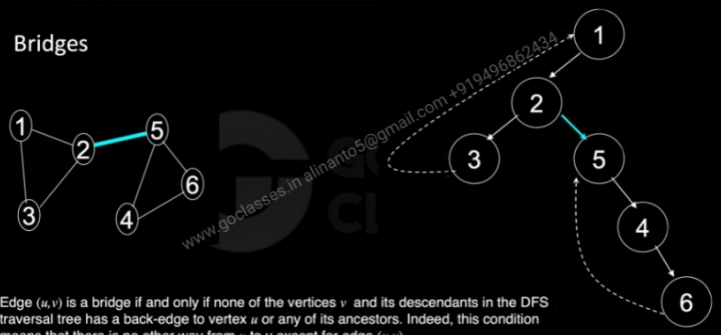
An *articulation point* in a connected graph is a vertex such that removing the vertex and its incident edges disconnects the graph into two or more connected components.

Let T be a DFS tree obtained by doing DFS in a connected undirected graph G.

Which of the following options is/are correct?

A. Root of T can never be an articulation point in G.
B. Root of T is an articulation point in G if and only if it has 2 or more children.
C. A leaf of T can be an articulation point in G.
D. If u is an articulation point in G such that x is an ancestor of u in T and y is a descendent of u in T, then all paths from x to y in G must pass through u.

Bridges



Edge (u,v) is a bridge if and only if none of the vertices v and its descendants in the DFS traversal tree has a back-edge to vertex u or any of its ancestors. Indeed, this condition means that there is no other way from u to v except for edge (u,v).

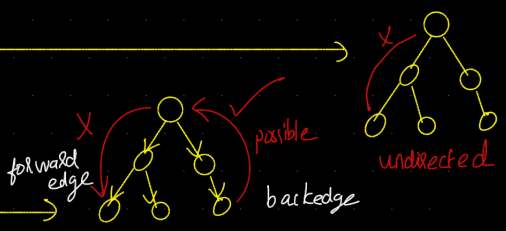
BFS → Complexity → Adjacency list → $O(V+E)$
→ Adjacency Matrix → $O(V^2)$

Frontier property → In a partial BFS Tree [partial → paused at some instant] the internal nodes cannot be in frontier → since they have been expanded already.

→ leafs in the last two levels only present in the frontier → leaf nodes of partial BFS tree which have height > 2 are not in frontier.

→ We cannot have graph edges between two different levels separated by another level — undirected

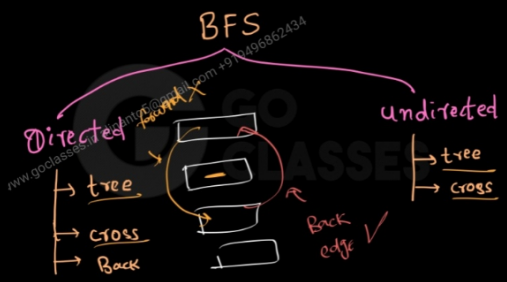
→ We cannot have graph edges from higher level to lower level, with atleast one level in between — directed



Cross edge
① within same level edges can be present in both directed & undirected.
② Across consequtive levels

BFS tree

When cross edges are present in BFS tree of undirected graph this can lead to cycles of
① odd length → Cross edge within same level
② Even length → Cross edge across consequtive levels



Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail.

Then,

True/False $d[v_i] \le d[v_{i+1}]$

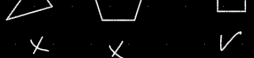
True/False $d[v_r] \le d[v_1] + 1$

$d[v]$ → distance from source

at any point the max difference in distance from source for any two points will be 1.

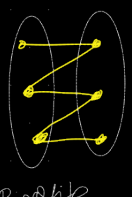
$$d[v_i] \le d[v_{i+k}] \le d[v_i] + 1$$

Bipartite graph ⟺ There is no odd length cycle.



→ Either there is no cross edge
or if cross edge exists, then it only exists across level & not within the same level

→ Actual definition
Bipartite means — graph can be divided into two sets of vertices V & V such that, there is no edge btw any two vertice in V & there is no edge btw any two vertices in V



Bipartite

☆ A tree graph is always bipartite

→ Given undirected graph → How to check using BFS → ① Do BFS tree
② No cross edge ⇒ Bipartite
③ No cross edge within same level ⇒ Bipartite

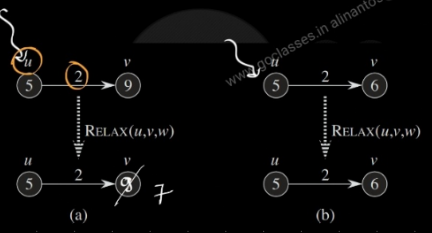
Greedy algorithm

Properties of Greedy Algorithms

- Optimal Substructure Property:-
  The optimal solution contains optimal solutions to subproblems
- Greedy Choice Property:-
  A global optimum can be arrived at by selecting a local optimum.

DP

Relaxing an edge (u,v)




```
RELAX(u,v,w):
  if (d[v] > d[u] +w)
     d[v] = d[u]+w
     parent[v] = u
```

DIJKSTRA CODE

```
DIJKSTRA(G,s):
  key[v] = ∞ for all v in V
  key[s] = 0
  S = ϕ
  initialise priority
  queue Q to all vertices

  while Q is not empty:
  u = EXTRACT-MIN(Q)
  S = S U {u}

  for each adjacent v of u
      RELAX(u,v,w)
```

0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞

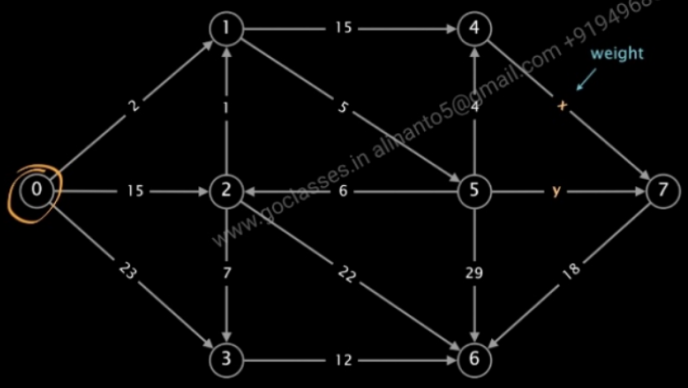
```
RELAX(u,v,w)
 if (d[v] > d[u] +w)
    d[v] = d[u]+w
    parent[v] = u
```

in loop
pick min
relax all outgoing edges

---

5. Shortest paths. (8 points)

Suppose that you are running Dijkstra's algorithm on the edge-weighted digraph below, starting from vertex 0.



	Cost	Parent
1	2	0
2	13	5
3	23	0
4	11	5
5	7	1
6	36	5
7	7+y 11+x	5 (7+y < 11+x) 4 (11+x < 7+y)

The table below gives the Priority Queue and Parent values immediately after vertex 4 has been deleted from the priority queue and relaxed.

v	Priority Queue	Parent
0	0.0	*null*
1	2.0	0 → 1
2	13.0	5 → 2
3	23.0	0 → 3
4	11.0	5 → 4
5	7.0	1 → 5
6	36.0	5 → 6
7	19.0	4 → 7

The table below gives the Priority Queue and Parent values immediately after vertex 4 has been deleted from the priority queue and relaxed.

Based on the state of priority queue (given in table), what is the condition that x and y must satisfy?

A. $x = 8.0$ and $y \geq 12.0$
B. $x > 8.0$ and $y = 11.0$
C. $x = 7.0$ and $y = 11.0$
D. $x > 8.0$ and $y = 12.0$

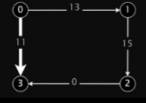
11+x < 7+y
11+x = 19 < 7+y
x = 8
y ≥ 12

Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.

Dijkstra selects vertex 3 immediately after 0. But shortest path from 0 to 3 is 0→1→2→3.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the shortest path from 0→1→2→3 to 0→3.

negative weights

reweighting

→ If dijistka's algorithm is applied for a graph with -ve edges, but no -ve cycle, then it may / maynot give the optimum path

→ Negative edge but cycle cost is non negative
⇒ Dijiska's might fail but the graph has a well definied minimum cost path.

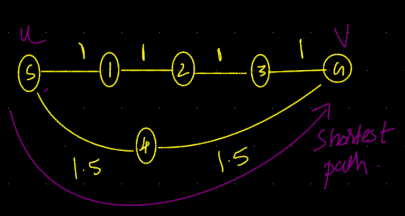
1. Consider the following strategy to solve the single source shortest path problem with positive integer edge weights from source s.

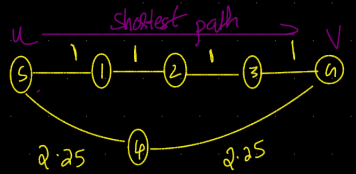
   - Replace each edge with weight w by w edges of weight 1 connected by new intermediate nodes.
   - Run BFS(s) on the modified graph to find the shortest path to each of the original vertices in the graph.

   Which of the following statements is correct?

   (a) This strategy will not solve the problem correctly.
   (b) This strategy will solve the problem correctly and is as efficient as Dijkstra's algorithm.
   (c) This strategy will solve the problem correctly but is not as efficient as Dijkstra's algorithm.
   (d) This strategy will only work if the graph is connected.

Let $G = (V, E)$ be an edge-weighted graph where all edges have distinct positive weights, and $u, v \in V$ be two vertices. Let P be a shortest path between u, v. Now suppose we replace the edge weight c_e by c_e^2 for each edge $e \in E$. Is it true that P will still be a shortest path between u and v in the new scenario? NO





GO CLASSES

Time complexity of Dijkstra =
(Adjacency list)
E.T(decrease key) + V.T(remove min)
extract min performed V times
relax performed E times

Time complexity of Dijkstra =
(Adjacency matrix)
E.T(decrease key) + V.T(remove min) + V^2
for loop cost
run V times
$= O(V^2)$

```
DIJKSTRA(G,s):
  key[v] = ∞ for all v in V
  key[s] = 0
  S = ϕ
  initialise priority
  queue Q to all vertices

  while Q is not empty:
    u = EXTRACT-MIN(Q)
    S = S U {u}

    for each adjacent v of u
        RELAX(u,v,w)
```

$$\sum_{u \in V}\left(\text{T(removeMin)} + \left(\sum_{v \in u.\text{neighbors}} \text{T(updateKey)}\right)\right)$$

$$\sum_{u \in V}\left(\text{T(removeMin)}\right) + \sum_{u \in V}\left(\sum_{v \in u.\text{neighbors}} \text{T(updateKey)}\right)$$

Adjacency List $V\,T(\text{removeMin}) + E\,T(\text{updateKey})$

Adjacency Matrix $V\,T(\text{removeMin}) + E\,T(\text{updateKey}) + V^2$

Graph Representation	Priority Queue Data Structure	Time Complexity Dijkstra
Adjacency List	Heap	$E \log V + V \log V = (E+V) \log V$
Adjacency Matrix	Heap	$(E+V) \log V + V^2$
Adjacency List	Unsorted Array	$E + V^2 = O(V^2)$
Adjacency Matrix	Unsorted Array	$E + V^2 = O(V^2)$
Adjacency List	Sorted Array	$EV + V$
Adjacency Matrix	Sorted Array	$EV + V + V^2$
Adjacency List	Fibonacci Heap	$E + V \log V$
Adjacency Matrix	Fibonacci Heap	$E + V \log V + V^2$

$E = O(V^2)$

$E \leq {}^{V}C_2$

$E \leq \frac{V(V-1)}{2}$

Dijiska's optimisation in DAG

Remember we can find topo order using DFS in $O(V+E)$

unsorted array

Algorithm Shortest path in DAG

1: Set $d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$.
2: topologically sort the vertices of G — $O(V+E)$
3: **for** each vertex u, taken in topologically sorted order **do** — V times
4: **for** every edge (u, v) **do** — $u \to v$ → $1 + \text{degree}(u_1) + 1 + \text{degree}(u_2) + \ldots = V+E$
5: Relax(u, v) — $O(1)$ → $\log(V)$ heap
6: **end for**
7: **end for**

if $(d[v] > d(u) + w)$
$\{ d[v] = d(u) + w$

but we don't need heap to extract min so array can be used in $O(1)$

Priority que implementations & complexity

	insert	deleteMin	decreaseKey
Un sorted linked list	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(1)$	Delete + insert(e) $O(n)$
Un sorted Array	$O(1)$	$O(n)$	$O(n)$
Sorted Array	$O(n)$	$O(1)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
heap ☆	$O(\log n)$	"	"

Used 99.9%

→ for all these data structure decrease equalent to delete key + insert key.

→ Bubble up the node
No need to insert & delete

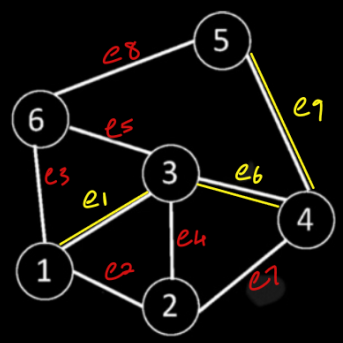
Bellman ford

GO CLASSES

Path - relaxation property

if we relax in the order of shortest path (along with intermixed other relaxation) then we will get shortest path cost.

Idea → ① If we relax the edges that appears in the shortest path in the correct order then the cost we find for goal is optimum even if we intermix any other edge relations.



→ Shortest path is $e_1 \to e_6 \to e_7$ edges
→ as long as we relax e_1 before e_6 before e_7 → in the order as they appear in the shortest path.
the other edge relations does not matter

$e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9$ ✓
$e_1 e_3 e_6 e_4 e_5 e_7 e_9 e_8$ ✓
$e_3 e_1 e_6 e_4 e_5 e_9 e_7 e_8$ ✓
$e_6 e_2 e_1 e_9 e_8$ ✗

} Valid ideas in Bellman ford

Idea ② How to ensure that shortest path edges are relaxed in correct order
→ Repeat V-1 times → guaranteed to relax in shortest path order for every path u→v.

$e_1 e_2 e_9 e_6 e_7 e_9$ → This is also valid → This is core idea behind bellman ford.

① Shortest path will visit atmost all V vertex
② shortest path will not revisit any vertex
③ shortest path will have atmost V-1 edges — $s_1\ s_2 \cdots s_{V-1}$
④ if I repeat V-1 times, each time making sure to relax all edges, then I am guaranteed to cover the shortest path order.

R_1	R_2	R_3	.....	R_{V-1}
Relaxed all edges in some order $e_1 e_2 e_3 .. s_1 ... e_{2n}$	Relaxed all edges in some order $e_1 e_2 e_3 .. s_2 ... e_{2n}$	Relaxed all edges in some order $e_1 e_2 e_3 .. s_3 ... e_{2n}$		Relaxed all edges in some order $e_1 e_2 e_3 .. s_{V-1} ... e_{2n}$

- Negative edges are allowed in bellman ford as long as there are no negative cost cycles
  ↳ still gives optimum solution
- Check for cycle → Do V-1 times, then perform 1 more time → if cost of any Vertex decreases ⇒ negative cycle

```
BELLMAN-FORD(G,s):
    d[v] = ∞ for all v in V
    d[s] = 0
for i = 1 to n-1
    for each edge (u,v) in E
        RELAX(u,v,w)
for each edge (u,v) in E
    if (d[v] > d[u] +w)
        return false
return true
```

(V-1)E

```
RELAX(u,v,w):
    if (d[v] > d[u] +w)
        d[v] = d[u]+w
        parent[v] = u
```

Relax O(1) since array instead of heap.
- Heap is not required since no extract-min

O(1)

E

check for negative cycle

Total Bellmanford	O(EV)
Total dijiskar with heap	O((E+V) log V)

→ Just after k iteration ($1 \le k \le V-1$) → we already have shortest path answers to all vertices reachable within k steps/edges

if you relax 5 times
⇓
you will get answer to any vertices where shortest path length is either 1, 2, 3, 4 or 5.

Summary of Bellman ford

① if works on -ve edges
② if runs V-1 times to calculate costs and then last one more time to check cycle — Detect negative cycles
③ time complexity O(VE)
④ if we run K times ⇒ we have shortest path to all vertices which are having atmost K edges

Bellman-ford : Early Termination

```
BELLMAN-FORD(G,s):
    d[v] = ∞ for all v in V
    d[s] = 0
for i = 1 to n-1
    relaxed = false
    for each edge (u,v) in E
        RELAX(u,v,w)
    if(relaxed == false) break
for each edge (u,v) in E
    if (d[v] > d[u] +w)
        return false
return true
```

```
RELAX(u,v,w):
    if (d[v] > d[u] +w)
        d[v] = d[u]+w
        relaxed = true
```

→ if no relaxation happens in a iteration ⇒ can stop early.

→ Best case 1 iteration ⇒ O(E)

3E ⇒ O(2E) in best case

Bellman-Ford can work with negative edges and can also work with cycles in the graph if the cycles have a net positive cost and can only fail or return unsuccessful if the graph is having a cycle with a net negative cost.

Your statement is **almost correct**, but it needs a bit of **precision**. Here's the accurate version with the right nuances:

What Bellman–Ford does handle

- ✅ **Negative edge weights** — yes.
- ✅ **Cycles in the graph** — yes, including:
  - positive-cost cycles
  - zero-cost cycles
  - even negative cycles **as long as they are not reachable from the source**

When Bellman–Ford fails

- ❌ Bellman–Ford **fails (or reports failure) only if there exists a negative-weight cycle that is reachable from the source**.
- In that case, shortest paths are **undefined**, because you can keep looping around the cycle to reduce the path cost indefinitely.