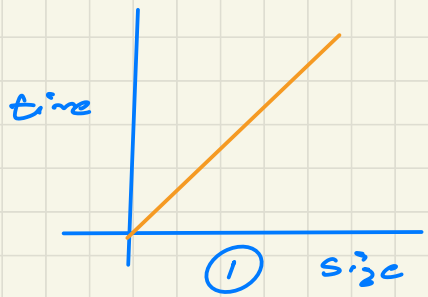
Time & Space Complexity

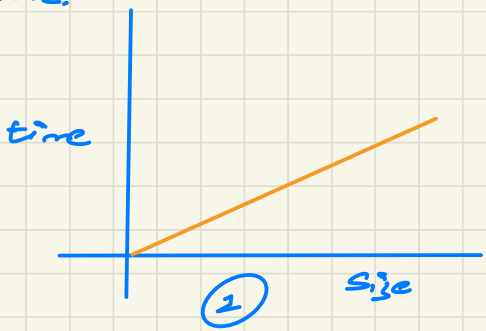
by Hridyesh

Time Complexity

1. Time complexity ≠ time taken by a program to complete.



①



②

Slope of ① > Slope of ② { both are linear functions }

2. Time complexity is the mathematical function in the graph.

function that gives us the relationship about how the time will grow as the input increases.

3. we always look at complexity for large/ ∞ data.

Eg. $O(n^3 + \log n)$ Let $n = 10^c$ ($\log c$)

$= O((10^c)^3 + \log 10^c)$

$= O(10^{18} + 6) \approx O(10^{18}) = O(n^3)$

for any $y = x^n + x^{n-1} + \dots$, we ignore the less dominant terms & constants

e.g. $O(3n^3 + 4n^2 + 5n + 6) \longrightarrow O(n^3)$

Notations

1. Big-O notation → describes the upper bound of the function of time complexity

e.g. if $f(n) = O(g(n))$

then $\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$ } finite value

2. Big-Omega notation → describes the lower bound of the function of time complexity

$\Omega(n^3)$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$

3. Theta notations → $\Theta(n^2)$ → $0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$

4. Little-O notation → also gives the upper bound but not strict to upper bound

Big-O

$f = O(g)$

$f \leq g$

(can't succeed)

more strong statement

← Little-O

$f = o(g)$

$f < g$

(strictly lower)

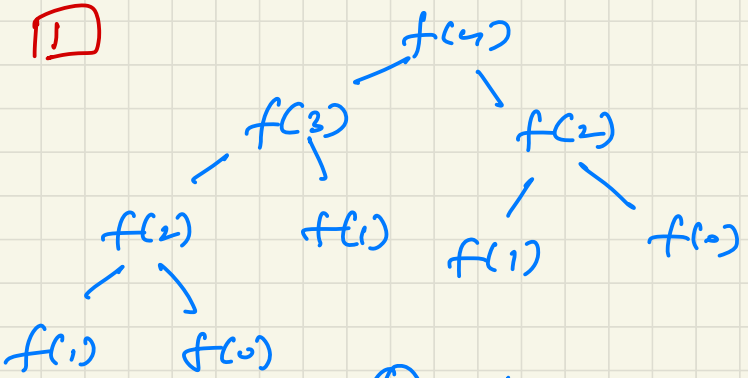
5. Little Omega → Strictly greater than the function.

i.e. $f = \omega(g) \rightarrow f > g$

Algorithms

1. Recursive Algorithm

1



at any particular time, no two function calls at the same will be in the stack at the same time.

(1) only calls that are interlinked with each other will be in the stack at the same time

(2) Space complexity is height of the tree.

(2) Type of recursion → Linear : $f(n) = f(n-1) + f(n-2)$

→ Divide & Conquer : $f(N) = f(\frac{N}{2}) + O(1)$

(1) Divide & conquer recurrence

form : $T(n) = a_1 T(b_1 n + E_1(n)) + a_2 T(b_2 n + E_2(n)) + \dots\dots a_k T(b_k n + E_k(n)) + g(n)$

for all $n \geq n_0$ → constant

eg. $T(n) = T\left(\frac{n}{2}\right) + c$

$a_1 = 1 \quad \therefore E_1(n) = 0$

$b_1 = \frac{1}{2} \quad ; \quad g(n) = c$

(2) **Akra - Bazzi theorem**

$$T(n) = \Theta\left(n^p + n \int_1^n \frac{g(u)}{u^{p+1}} \, du\right)$$

$a_1 b_1^p + a_2 b_2^p + \cdots = 1 \quad \therefore \left\{ \sum_{i=1}^{k} a_i b_i^p = 1 \right\}$

Q $T(n) = 2T\left(\frac{n}{2}\right) + n - 1$ } $T(n)$ of merge sort

Sol^n

$a = 2 \quad , \quad g(n) = n - 1$

$b = \frac{1}{2}$

$\Rightarrow a_1 b_1^p = 1$

$2 \cdot \left(\frac{1}{2}\right)^p = 1 \quad \therefore p = 1$

$$T(n) = O\left(n + n\int_1^n \frac{u-1}{u^2}\,du\right)$$

$$= O\left(n + n\left(\ln u - \frac{1}{u}\right)_1^n\right)$$

$$= O\left(n + n\left(\ln n - \frac{1}{n}\right)\right)$$

$$\therefore T(n) = O(n \ln n) \quad \text{or} \quad \{T(N) = O(N \log N)\}$$

Q. $T(n) = 3T\left(\frac{n}{3}\right) + 4T\left(\frac{n}{4}\right) + n^2$

Sol:

$$3\cdot\left(\frac{1}{3}\right)^p + 4\cdot\left(\frac{1}{4}\right)^p = 1 \Rightarrow p > 1$$

for $p = 2$, $\frac{7}{12} < 1$ $\therefore 1 < p < 2$

Note: when $p <$ power of $g(n)$, then $g(n)$ is the answer

$$T(n) = O\left(n^p + n\int_1^n \frac{u^2}{u^{p+1}}\,du\right)$$

$$= O\left(n^p + n\int_1^n u^{1-p}\,du\right)$$

$$= O(n^p + n^2) \qquad p < 2$$

$$\therefore T(n) = O(n^2) \qquad \therefore n^p < n^2$$

(3) Solving linear recurrences

e.g. $f(n) = f(n-1) + f(n-2)$ {fibonacci no.}

form : $f(n) = a_1 f(n-1) + a_2 f(n-2) + a_3 f(n-3)$

$+ - - -$

$\therefore f(n) = \sum_{i=1}^{\infty} a_i f(n-i)$

Sol^n : put $f(n) = \alpha^n$ $\forall \alpha =$ some constant

$\Rightarrow \alpha^n = \alpha^{n-1} + \alpha^{n-2}$

$\alpha^2 = \alpha + 1 \Rightarrow \alpha^2 - \alpha - 1 = 0$

$\therefore \alpha = \frac{1 \pm \sqrt{5}}{2}$

$\Rightarrow f(n) = f(n-1) + f(n-2) = C_1 \alpha_1^n + C_2 \alpha_2^n$

$= C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$ - (2)

fact : no of roots = no of answers we've already got.

in case of fibonacci, we know two answers

$f(0) = 0$
$f(1) = 1$

$\Rightarrow c_1 + c_2 = 0$

$c_1\left(1 + \frac{\sqrt{5}}{2}\right) + c_2\left(1 - \frac{\sqrt{5}}{2}\right) = 1$

$c_1 = \frac{1}{\sqrt{5}}$ & $c_2 = -\frac{1}{\sqrt{5}}$

$$\Rightarrow \left\{ f(n) = \frac{1}{\sqrt{5}}\left(1 + \frac{\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(1 + \frac{\sqrt{5}}{2}\right)^n \right\}$$ → for nth fibonacci term

$$= \frac{1}{\sqrt{5}}\left\{\left(1 + \frac{\sqrt{5}}{2}\right)^n - \left(1 - \frac{\sqrt{5}}{2}\right)^n\right\}$$

as $n \to \infty$, $\left(1 - \frac{\sqrt{5}}{2}\right)^n \to 0$

$$\Rightarrow f(n) = \frac{1}{\sqrt{5}}\left[\left(1 + \frac{\sqrt{5}}{2}\right)^n\right]$$

$\therefore$ T.C $\longrightarrow O(1.618^n)$

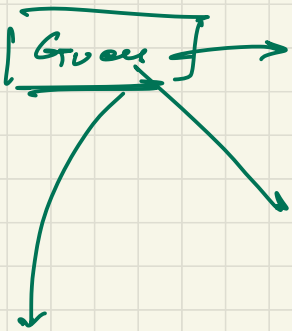
(4) Solving non-homogeneous linear eq^n / recurrences

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \ldots + a_d f(n-d) + g(n)$$

after adding some function, $f(n)$ is non homogeneous now

Step-1 : replace $g(n)$ with 0 & solve usually

Step-2 : Guess something that is similar to $g(n)$



if $g(n)$ is exponential, guess of same type : if $g(n) = 2^n + 3^n$ then $f(n) = a2^n + b3^n$

if $g(n)$ is polynomial, guess of same degree $g(n) = n^2 - 1$.

if $g(n)$ is a mixture of exponentials & polynomial,

i.e. $g(n) = 2^n + n$ $\therefore f(n) = a2^n + (bn + c)$

for exp for poly

* if $f(n) = a2^n$ fails, try for $f(n) = (an+b)2^n$, if fails too, increase the degree

Q $f(n) = 2f(n-1) + 2^n$ $\therefore f(0) = 1$ {Given}

Sol^n put $2^n = 0 \Rightarrow f(n) = 2f(n-1)$ [Let $f(n) = \alpha^n$]

$\Rightarrow \alpha^n = 2\alpha^{n-1} \Rightarrow \alpha = 2$

$g(n) = 2^n \therefore f(n) = a2^n$

$\therefore f(n) = (an+b)2^n$

$\Rightarrow a2^n = 2a2^{n-1} + 2^n$

$\Rightarrow a = a + 1$

(absurd)

$\Rightarrow (an+b)2^n = 2(a(n-1)+b)2^{n-1} + 2^n \Rightarrow a = 1$

(discarding b) $\therefore f(n) = 2^n + n2^n$ $\therefore f(n) = n2^n$

T.C $\longrightarrow O(n2^n)$

Analysis

1. Nested loops : **T.C → O(n·m)**

{ outer loop n times & inner loop m times }

2. Insertion sort : → sorting where we believe that first element is sorted

= K + 2K + ... + (n−1)K

K: checking the complete array

∴ **T.C → O(n²)**

3. Selection sort → which finds the min element in the array

: checking kn times to find the smallest element in the array.

= kn + k(n−1) + k(n−2) + ...

= k { n + n + n + ... − (1 + 2 + ...) }

= k { n(n−1)/2 + n² }

∴ **T.C → O(n²)**

4. For recursive algo : **T.C → O(n)**

→ $T_L = k + T(n-1)$

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

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

$T(n-2) = K + T(n-3)$

$\vdots \qquad \vdots \qquad \vdots$

$T(1) = K + T(0)$

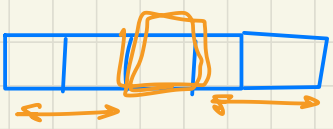
5. Binary search : $T(n) = K + T(n/2)$

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

$T(n/4) = K + T(n/8)$

$\vdots \qquad \vdots \qquad \vdots$

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

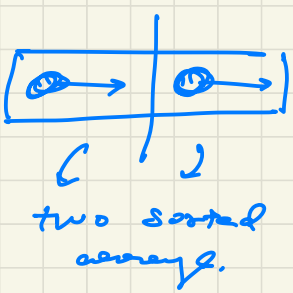


$n, \frac{n}{2}, \frac{n}{4}, \ldots 1$

$\therefore$ T.C $\longrightarrow$ $O(\log_2 N)$

6. Merge Sort Analysis $\longrightarrow$ Call on first half : $T(n/2)$

Call on second half : $T(n/2)$



two sorted arrays.

$= K_1 n + K_2 n$ (for one half)

$= K_3 n$

Expression : $T(n) = K + T(n/2) + (K_1 n + K_2 n)$

- K : base case or one operation
- $T(n/2)$: merging two half arrays
- $K_1 n + K_2 n$: while making a third array.

$$2[T(n/2) = K + 2T(n/4) + K_2 n k]$$

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

Space Complexity

1. Space complexity of an algorithm represents the amount of extra space needed by its life cycle.

$$S(P) = A + S_P(I)$$

- A: fixed part
  - fixed part is a space required to store certain data & variables, that are not dependent on size of variable.
- $S_P(I)$: variable part.
  - variable part is a space that is dependent on size of problem. e.g. recursive stack, dynamic memory allocation.

2. Quick sort space complexibility analysis

(1) Best case scenario

↓

occurs when the partitions are as evenly balanced as possible.

(2) Worst case scenario

↓

happens when we encounter the most unbalanced partitions possible

3. (1) Space complexity is calculated on the basis of space used in the recursion stack.

(2) Average case space used → $O(\log n)$

(3) Worst case space used → $O(n)$

College Notes

Red & Black tree → self balancing BST

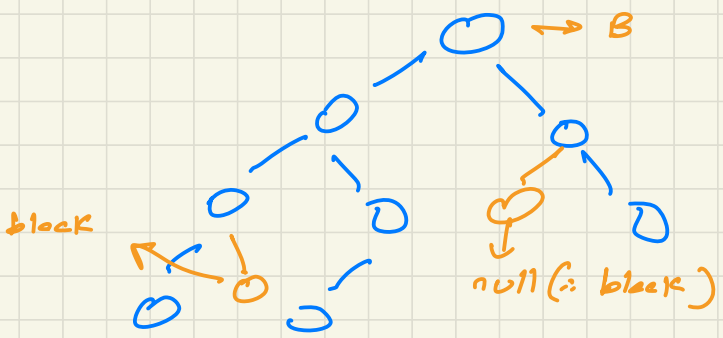
↳ includes colour instead of number balancing

Rules: (1) Root: always be black {root node height = 1}

(2) Children of red node will always be black

(two consecutive red nodes aren't allowed)

(3) Black height must be same for all the paths from root to leaf.

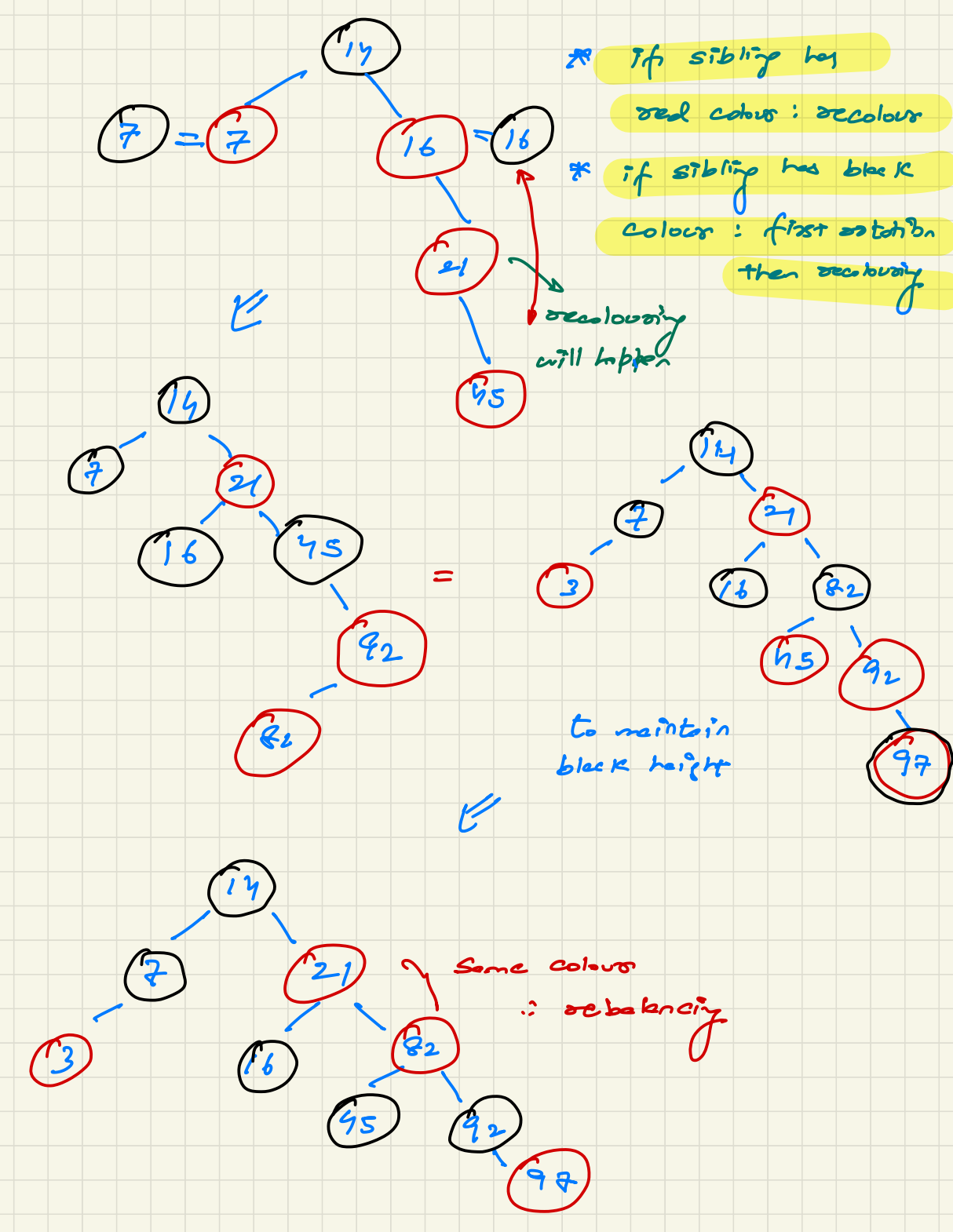


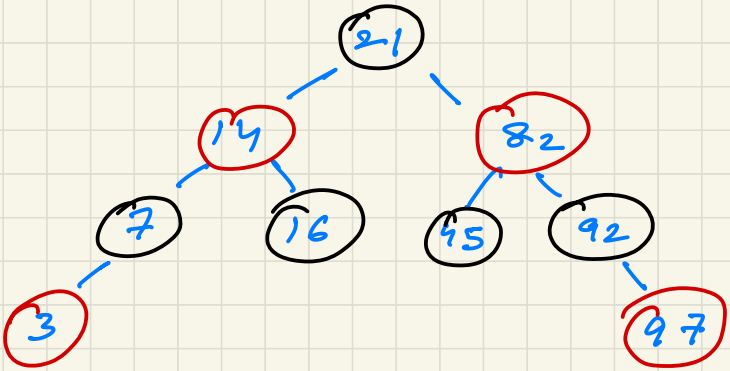
{Black depth should be same}

(4) All the null pointer nodes will be black

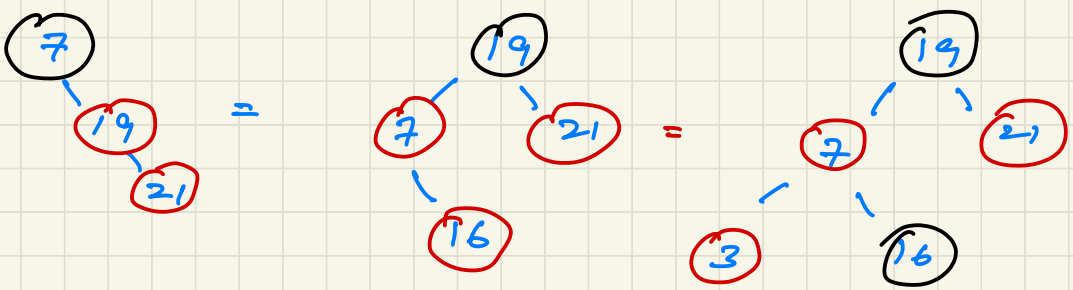
* Every new inserted node will be red

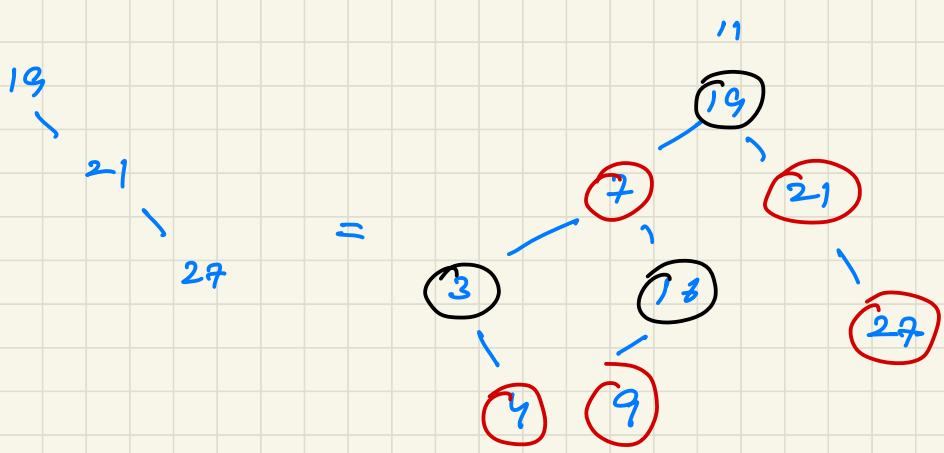
14, 7, 16, 21, 45, 42, 82, 3, 97


14
7
7
16
16
21
25
* If sibling has red colour : recolour
* if sibling has black colour : first rotation then recolouring
recolouring will happen
14
7
21
16
25
42
42
14
7
21
3
16
42
25
97
to maintain black height
14
7
21
3
16
42
25
97
Same colours
∴ rebalancing



(2) 7, 19, 21, 16, 3, 4, 9, 29, 42, 13





Question

1. Koi mil gya

Key : ① Sum of value of a number

② Smallest no. that has sum of digits equal to the no.

E.g. 15 = 6 + 9 ∴ Smallest no. = 69

Input = 15 ⟶ Output = 69

⇒ Two variables : one stated as -1 in the beginning

Recursion

1. $$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

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

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

$T(n) = T(n-3) + 3$

$T(n) = T(n-k) + k$

$n - k = 0$

$n = k$

$T(0) + n$

$= 1 + n$

$= O(n)$

2. Draw the recurrence tree

```
T(n) ⟶ void fun(int n){
         (n>0)
       for(i=0 ; i<n ; i++){
         cout << "i";
       fun(n-1);
```

Time complexity

1. Sorting Algorithm

Algorithm	Time complexity		Space complexity
	Worst case	Best Case	
Selection Sort	$\Omega(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$\Omega(n)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(1)$
Heap Sort	$\Omega(n \log n)$	$O(n \log n)$	$O(1)$
Quick Sort	$\Omega(n \log n)$	$O(n^2)$	$O(n)$
Merge Sort	$\Omega(n \log n)$	$O(n \log n)$	$O(n)$
Bucket Sort	$\Omega(n+K)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nK)$	$O(nK)$	$O(n+K)$
Count Sort	$\Omega(n+K)$	$O(n+K)$	$O(K)$

2. Graph Algorithms → All algos have $O(N+E)$ as T.C

Algorithm	Time complexity		Space complexity
	Worst case	Best Case	
BFS	$O(V+E)$	$O(V+E)$	$O(V)$
DFS	$O(V+E)$	$O(V+E)$	$O(V)$
Cycle det. (DAG + BFS)	$O(N+E)$	$O(N+E)$	$O(N+E) + O(N) + O(N)$

3. Different Algorithms

Algorithm	Time complexity	Space complexity
Dijkstra's Algorithm	$O(n \log n)$	$O(n) + O(n)$
Disjoint Set	$O(4)$	$O(n)$
Kruskal's Algorithm	$O(n \log n)$	$O(n)$