A Silent Voice → Netflix

Todays Content:

- → length of longest Increasing Subsequence
- → Colouring Houses
- → # Party
- → # Encoding

} # Homework

Q) Given N ar[], find length of longest Increasing Subsequence ⇒ LIS

Note: Subsequence, values should be strictly increasing

↳ Should be ordered based on index $a_1 < a_2 < a_3$:=

Ex1: ar[5] = { 9 2 4 3 10 } → fin len = 3

	0	1	2	3	4
	9	2	4	3	10

Constraints

$1 \le N \le 10^3$

ans1 = { 2 4 10 } → len = 3

ans2 = { 2 3 10 } → len = 3

Ex2: ar[6] = { 2 -1 6 3 7 9 } → fin len = 4

ans1 = { 2 6 9 } → len = 3

ans2 = { 2 6 7 9 } → len = 4

↳ { -1 3 7 9 } → len = 4

↳ { -1 6 7 9 } → len = 4

Idea: For every subsequence, check if its strictly increase or not & get max length of them.

→ generate all Subsequence

a) Bit Manipulation — TC: $2^N * N$

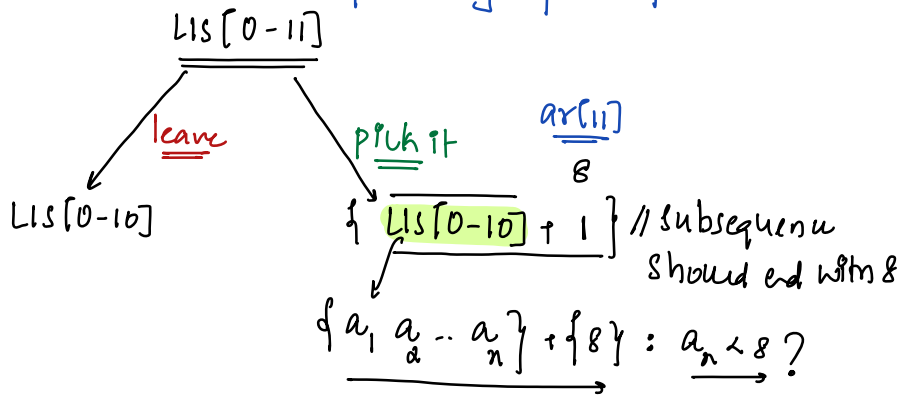
b) Back Tracking — TC: 2^N

If $N = 10^3 \rightarrow 2^{10^3} \rightarrow 2^{1000}$

Note: If as a brute force we only a Back-Tracking solution which is very huge for constraints, we can think interms of dynamic Programming

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] =	10	3	12	7	2	9	11	20	11	13	6	8

→ dp[i] = length of LIS from [0-i]



Issue: We don't know where subsequence ends

dp[i] = length of longest Increasing Subsequence from [0-i] ending at i^{th} index → { This should contain ar[i] }

	0	1	2	3	4	5	6	7	8	9	10	11
ar[12] =	10	3	12	7	2	9	11	20	11	13	6	8
dp[12] =	1	1	2	2	1	3	4	5	4	5	2	3

{3 12}, {3 7}, {2}, {3 7 9}, {3 7 9 11}, {3 7 9 11 20}, {3 7 9 11}, {3 7 9 11 13}, {3 6}, {2 6 8}

final ans = max of dp[]

dp[i] = { length of longest increasing subsequence ending at i }

$$dp[i] = \max_{\substack{j=0,\ \text{if}(ar[j] < ar[i]) \\ j=i-1}} \{ dp[j] \} + 1$$

↳ including i^{th} index element

Base: dp[0] = 1

dp table: → Items: dp[N+1]

↳ array: dp[N]

Code: ⟶ TC: # States * TC of each state

↳ O(N) * O(N)

```
int LIS(int ar[]) {
    int n = ar.length
    int dp[n]
    int dp[0] = 1
    i = 1; i < n; i++) {
        // dp[i] → length of longest increasing subsequence ending at i
        v = 0
        j = 0; j < i; j++) {
            if( ar[j] < ar[i]) {
                v = max(v, dp[j])
            }
        }
        dp[i] = v + 1
                ↳ because we are including i^th index element
    }
    return maxArr(dp)
            ↳ return max value of given arr[]
}
```

TC: O(N²) ⟶ O(N log N)

SC: O(N) optimize* ⟶ {dp + Binary Search}

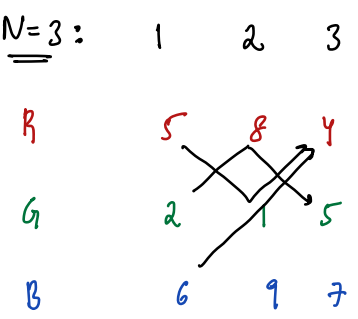
optimal:

208) N Houses :

Given N houses & Cost associated to Colour each house in R/G/B find min cost to paint all houses

Note: No 2 adjacent houses should have same colour

N=3 :



paints : G R G → 15

B G R → 11

R G R → 10

G G R → Invalid ? 2 adjacent cannot have same colour

min Cost = 10

Idea1 : Try out all Combinations

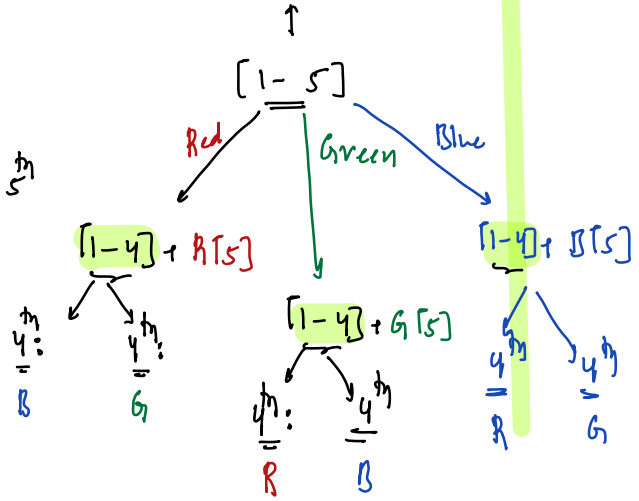
↳ # 3^N combinations

By Neglecting few combinations & making sure that, no 2 adjacent are same = $3 * 2^{N-1}$

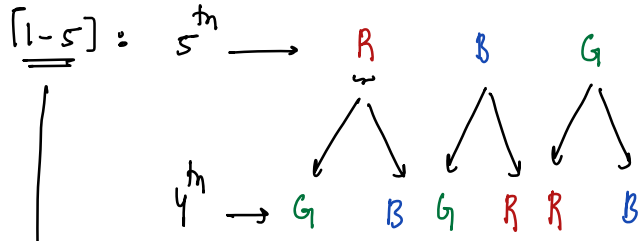
Idea2:

N=5	1	2	3	4	5
R	5	8	4	2	1
G	2	1	5	6	7
B	6	5	7	4	5

Min cost to fill all hous from [1-5] such that no 2 adjacent same colour



[1-4] : min cost to paint all hous from 1-4 such that no 2 adjacent have same colour



Note: We should also know colour we are ending with

R → 0, B → 1, G → 2 — i^{th} House

$dp[i][0]$ = {min cost to paint $[1-i]$, such i^{th} house ends Red}

$dp[i][1]$ = {min cost to paint $[1-i]$, such i^{th} house ends Blue}

$dp[i][2]$ = {min cost to paint $[1-i]$, such i^{th} house ends Green}

Dp Expression:

$dp[i][0] = R[i] + \min(dp[i-1][1], dp[i-1][2])$ — $R[i]$: cost to pain i^{th} House with Red

$dp[i][1] = B[i] + \min(dp[i-1][0], dp[i-1][2])$

$dp[i][2] = G[i] + \min(dp[i-1][1], dp[i-1][0])$

Base Condition: {i == 0}

$dp[0][0] = dp[0][1] = dp[0][2] = 0$

↳ {when i==0, there are no house}

Dp table: $dp[N+1][3]$;

Pseudocode :

// Given N houses, R[], G[], B[]

R[i] = cost to paint i^{th} house in Red

G[i] = cost to paint i^{th} house in Green

B[i] = cost to paint i^{th} house in Blue

```
int dp[N+1][3]                                  TC: O(3N + 1)  SC: O(N)
dp[0][0] = dp[0][1] = dp[0][2] = 0
i = 1 ; i <= n ; i++){
    dp[i][0] = R[i] + min(dp[i-1][1], dp[i-1][2])
    dp[i][1] = B[i] + min(dp[i-1][0], dp[i-1][2])
    dp[i][2] = G[i] + min(dp[i-1][1], dp[i-1][0])
}
return min( dp[N][2], dp[N][0], dp[N][1])
```

// Space Optimization : At any given point, we only need 6 dp States, hence we can optimize this to → O(1) ⇒ TODO

Trace:

N=5	0	1	2	3	4	5
R	0	5	2+8=10	4+6=10	12+2=14	1+14=15
G	0	2	5+1=6	5+7=12	10+6=16	7+14=21
B	0	6	5+2=7	7+6=13	10+4=14	5+14=19