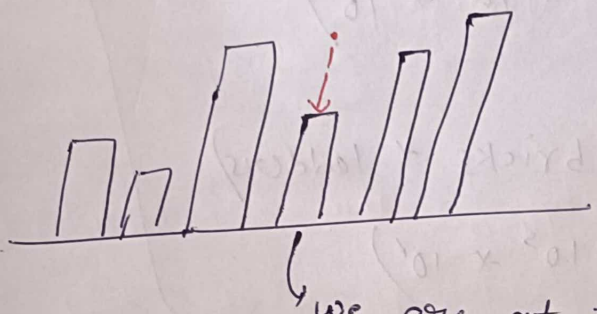
1642. Furthest Building You Can Reach



we are at the i^{th} building & we have 2 options

- ladder
- bricks

seems it like a DP ?

DP

```
int op1, op2, op3
if (h[i] >= h[i+1])
    op1 = 1 + solve(i+1, bricks, ladders);
else{
    int heightDiff = h[i+1] - h[i]
    if (bricks - heightDiff >=0)
        op2 = 1 + solve(i+1, bricks-heightDiff,
                        ladders)
    if(ladders > 0)
        op2 = 1+solve(i+1, bricks, ladders-1)
```

return d[i][bricks][ladders] = max(op1, op2, op3);

Time $\leq O(n * bricks * ladders)$

$\leq O(10^5 * 10^9 * 10^8)$

S.C. $\leq O(n * bricks * ladders)$

$O(10^5 * 10^5 * 10^1)$

How to remove MLE:- To optimize MLE we can may be use string & string concatination with unordered map like

string → "i # bricks # ladders" ← key

but it can also give us MLE

NOTE: We can apply DP but it will give us TLE or MLE

→ This is the similar problem to this ~~DP~~ problem. like this

871. Minimum Number of Refueling Stops

↳ DP greedy → Greed (Min Heap / Max Heap)

Approch :- 2

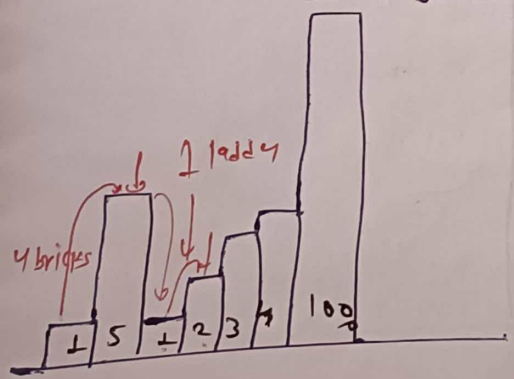
Our approch is like that first we will use the entire bricks & then we go for ladders.

: bcz from bricks we can go up to a limitied height but from ladder we can go at any height.

Note:- This will not work for any case

heights → [1, 5, 1, 2, 3, 4, 1000]

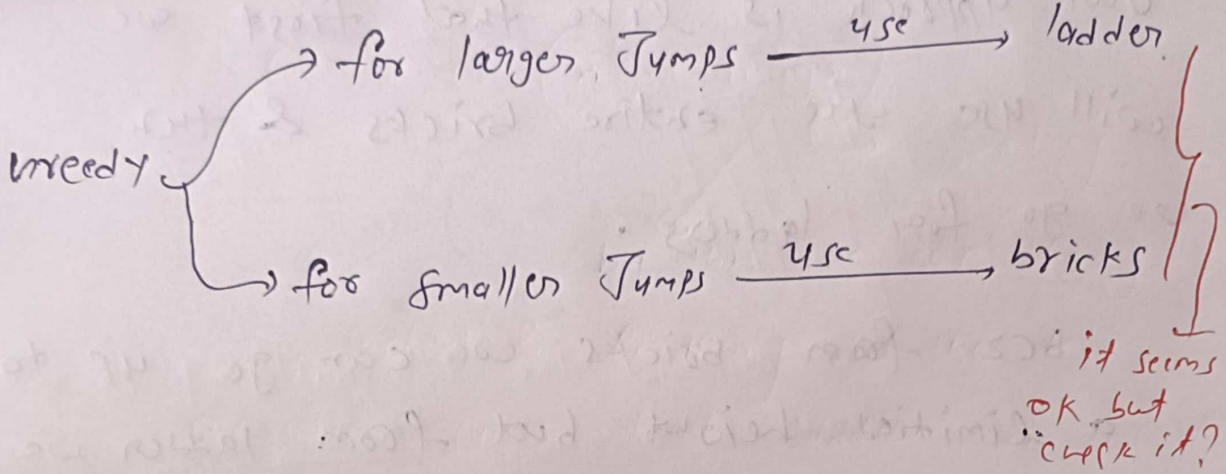
bricks = 4, ladder = 1



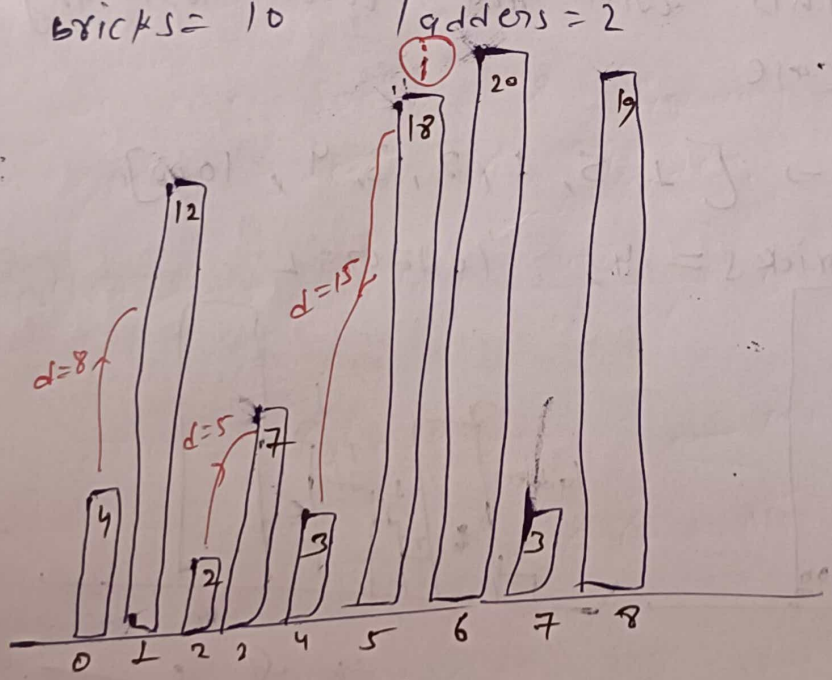
ans = 3

ans = 4

Correct approach of approach 2 :-



heights :- [4, 12, 2, 7, 3, 18, 20, 3, 19]



let's say we are at i=5 index & now we have to figure out that can we reach out that index or not.

How to figure out:-

This will tells us that to reach up till ith index what Jumps we have required.

~~Jumps~~

Jumps Vector = {8, 5, 15}

↳ but we use greedy approch i.e.

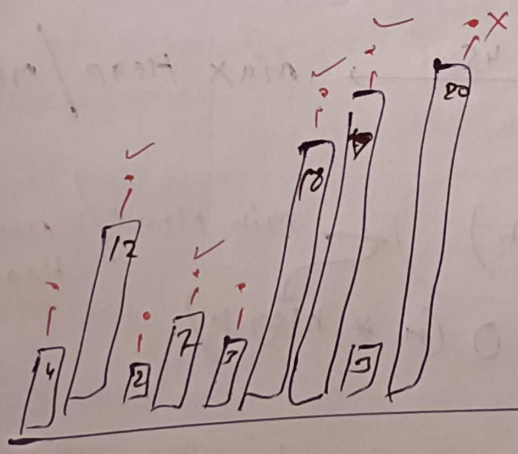
for lagre height —use→ ladder

small " —use→ bricks

so sort the Jump vector = {5, 8, 15}

Jump vector = {5, 8, 15} ⟶ max 2 Jumps are performed by ladder bcz we have 2 ladders

remaing Jums = {5} & we have brick = 10 i.e. we can perform this with bricks



i.e. we are check this (figuring out condi^{n}) for every i & some point this will ~~for~~ work

i.e. $(i-1)^{th}$ index we can reach

~~T.C. = O(n * nlogn)~~ → no for all index

T.C. = O (n * nlogn)

nlogn → sorting time

by this we are doing sorting & figuring out the 'k' maximum jumps that ladder can perform and the remaing jumps we sum them up & ask for bricks that it can do them.

i.e. we are sorting & figuring out the 'k' maximum jumps for ladder.

[Rather than sorting → use → max Heap / min Heap]

T.C. = ~~O(n * logn)~~ → O(n * nlogk) ← min Heap / max Heap

Note :- we can also use binary search.

T.C. = O(logn * nlogk)

bcz rather than going linearly we go for binary search if that index satisfie

the condition of figuring out then we go for right building else we go for left buildings.

Approach:-

↳ $(n \log n)$

↳ maintaing the greedy behaviour but use that behaviour in emergency :)

bcz in Linear search we are checking greedy behaviour for 'n' places & for binary search we are checking greedy behaviour for 'log n' places.

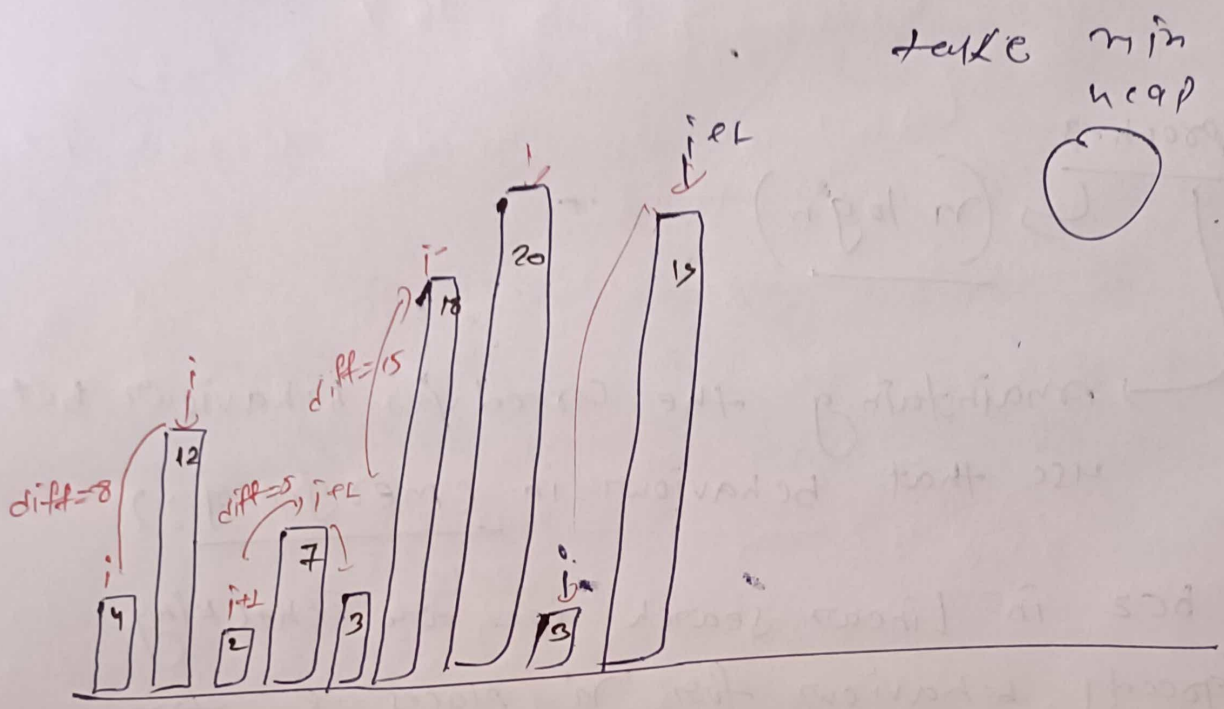
but here we only check greedy behaviour only in emergency.

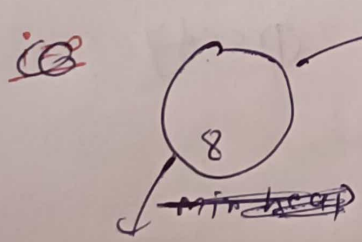
↳ emergency comes when you have Jumps more than ladders.

ex→ you have to do Jumps more than ladder like ladder = 2 but you have Jumps '3' i.e. this 3rd Jump should perform by bricks.

$h = [4, 12, 2, 7, 3, 18, 20, 3, 19]$

bricks = 10 ladder = 2





→ here we can use priority que or multiset } both have same insertion & deletion complexity

we use ds which will return minimum element.

(8, 5) → ladder size = 2. & min heap size = 2
i.e. we can do those jump by ladder.

(8, 15, 5) → size > ladder i.e. grab out the minimum Jumps & perform that Jump

By bricks.

brick -= minheap.pop();

bricks = 5

(8, 15, 2) min = 2 bricks = 5 - 2 = 3

(8, 15, 13) min = 8 brick = 3 - 8 = -5

brick < 0 → return -1

→ we can also we max heap, but we have to modific according that.

T.C. = O(n log k)

S.C. = O(k)

k is numbers of ladder

O(n³) (Brute force DP) → O(n² log n) (Brute force greedy) → O(n k log n) Better (greedy)

O(n log²n) (Better Binary search) → O(n log n) (optimal minheap)