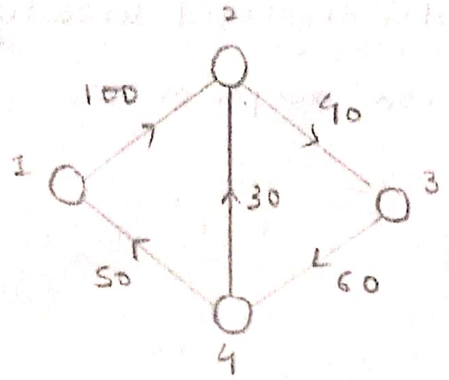
Day 31 DP I

1 Dimensional DP.

Q. Minimize the cash flow among a set of friends who have borrowed money from each other



(i) calculate net dues for each person

(ii) segregate into +ve & -ve trays

(iii) Take two values, one each from each tray, and settle the smaller abs value & the remaining value needs to be put back in the apt. tray.

1	2	3	4
-100	+100	+40	+60
+50	-40	-60	-30
	+30		-50
-50	+90	-20	-20

loop over all transactions and accordingly maintain an array to keep track of dues.

1, 3, 4 are owned more, whereas 2 owes money to others.

-ve	+ve
~~-50~~	~~+90~~
~~-20~~	~~+70~~
~~-20~~	~~+50~~
	0

minimum cash flow is the sum of abs. value of one tray for the entire system.

* each person just cares about the fact that their dues must be cleared. How that is done is of no care.

↓

Hence, finding the net dues, and then settling them minimises the transactional cashflow

minimizing the number of transactions → Strongly connected components

* optimising building of heap to O(n)

↓

There are two important operations involved with a heap : i) sift up

ii) sift down (using the sift down approach, it can be O(n))

i) sift up, is when the child is compared with its parent and then the order is checked.

ii) sift down is when the parent is compared to its children & replaced by the one more lesser. Generally used in deletion operation of heap.

↓

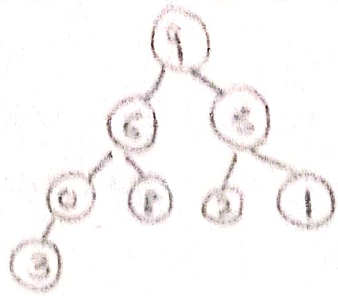
So here the premise is rather than ~~using~~ using sift up for each leaf, which would involve logn moves in worst case.

↓

sift down is employed for all non-~~leaf~~ leaf nodes, since half the nodes are not leaf. and the movements for other half would be of the order O(n) in worst case.

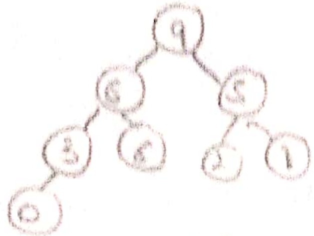
Scanned by CamScanner

for example 9 6 5 0 8 2 1 3

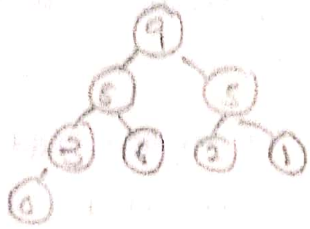


leaf nodes are ignored since they are proper heaps already.

↓ Starting at 0 for max heap, Compared with child, smaller, interchange



↓ Now 5 is correctly positioned, then 6 less than 8, so replaced



→ Now 9 (root) is already in correct position, so no movement.

↓

Now in the above process at worst case level 1 from bottom will require 1 level movement only. ones on level 2, 2, and so on.

↓

∴ Total moves required in worst case

Taking sum to ∞ for proper upperbound

$$\left\{ \frac{N}{4} \times 1 + \frac{N}{8} \times 2 + \frac{N}{16} + 3 + \dots \right.$$

↓

which totals to upper bound $O(n)$.

$$S = \frac{N}{4} * 1 + \frac{N}{8} * 2 + \frac{N}{16} * 3 + \dots$$

$$\frac{S}{2} = \frac{N}{8} + \frac{2N}{16} + \frac{3N}{32} + \dots$$

$$S - \frac{S}{2} = \frac{N}{4} + \frac{N}{8} + \frac{N}{16} + \dots$$

$$= N\left(\frac{1}{4} + \frac{1}{8} + \dots\right)$$

$$= \frac{N}{4}\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right)$$

$$= \frac{N}{4}(2) = \frac{N}{2}$$

(establishing an upper bound by taking an ∞ sum)

Since $S - \frac{S}{2}$ is $O(\frac{N}{\cancel{2}})$ $\therefore$ time complexity of building a heap can be proved to be worst case $O(n)$.

↓

heapsort is still $O(n\log n)$

Dynamic programming :

$1+1+1+1+1+1+1+1+1+1 = 10$

$+1 = 10+1$

↓

calculate the given sum? → added one to it

↓

Now, what's the sum?

↓

option 1 : calculate the whole sum again

$O(n^2)$

option 2 : use the pre computed sum and add 1 to that to get the new sum.

$O(n)$

without repeated computation

↑

in real world, we operate with memory, & we can store already computed small problems, so that if req those values can be used.

more efficient ←

essence of dp

prefix sum : example of dp . as

prefix_sum[i] = prefix_sum[i-1] + A[i];

↓

remember things that may need to be reused

↓

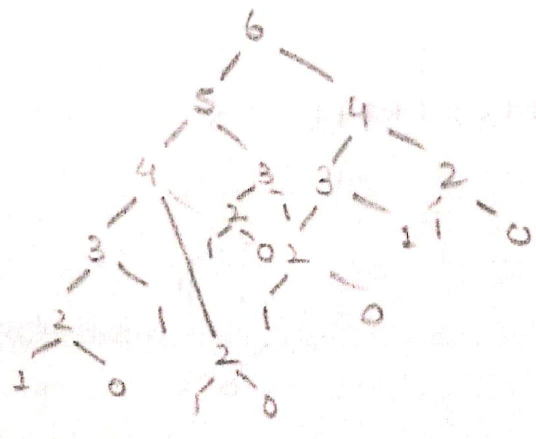
Always think of the next step . when writing the brute force soln, can remembering some computations help ?

↓

Recursively .

fibo[i] = fibo[i-2] + fibo[i-1]

```
int fibo( i ){
    #Base
    return fibo(i-1) + fibo (i-2);
}
```



repeated work since comp. of 4, 3, 2, is done so many times

↓

Store the value when they are called first, and use it next time

↓

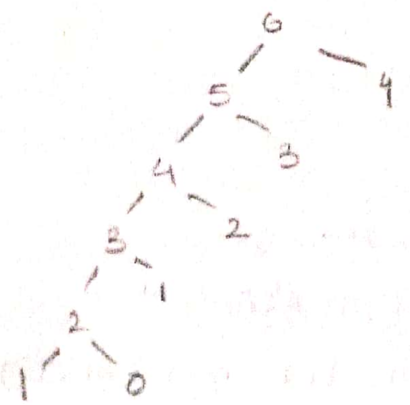
making it O(1) operation

→ huge optimization → $O(2^n) \to O(n)$

memoization ← (store the value of the fn with the parameters, and at each fn. call check if comp. for those parameters is already done)

Using array or hashmap

* 1 D DP
* 2D DP
* DP on trees
* Knapsack DP
* DP for optimizing NP hard



(the computations not being repeated).

* memoization based solns are acceptable in interviews
* online judges might not accept recursive ones, and iterative solns, would be req.

Q. Stairs.

At any step, you can either climb 1 stair, or you can climb 2 stairs in 1 go. Given n stairs, how many ways can it be climbed.

for k steps (we can iterate from 1 to k) → and memorize the computations

Base cases: 0 → 1, 1 → 1

func(n) → func(n-1) + func(n-2),
↓
and apply memoization

* store the base cases first and then build on them till the current or required parameters.

* direction of calculation changes. In iterative, stack memory is saved.

Q. Given an array, find the longest increasing subsequence (non-contiguous)

1	2	5	3	4	10	8	9

* O(nlogn) approach as well.

$O(n^3) \to O(n^2)$

```
int lis(index){
    //base conds.
    int len = 1;
    for(i=0; i<index; i++)
    {
        if(A[i] < A[index])
        { len = max(len,
                    lis(i)+1); }
    }
    return max. lis[i] = max;
```

Li. subsequence that ends at index i
↓
lis for numbers less than A[i], and then A[i] can be inserted after it.

```
ans[0]=1
for (int i=1 ; i<N; i++) {
        ans[i]=1;
        for (int j =0; j<i ; j++) {
                if ( A[j]<A[i] ){
                        ans[i] = max (ans[i], ans[j]+1);
                }
        }
}
```

query for max element in dp array with own main array value less than A[i].