(A) Dynamic Programming

(1) DP - Prateek videos Lecture Notes (coding Ninjas)

- Recursion (Heart of Dynamic Programming)
- Fibonacci series (Recursion & DP way)
- Min^m steps to 1 (DP → (Bottom up (Tabulation) & Top Down (Memoization)))
- Min^m Coin to Given Sum (DP)

These are all 3 one dimensional DP questions only.

only 3 questions & Top Down approach is always readable to get substructure or get one from it's subproblem.

(2) Some Basic DP Intros & difference b/w DP and Greedy

General Knowledge

#1 Before going to Dynamic Programming Problems you should must be aware and master your concept in recursion.

#1 Introduction to Dynamic Programming

*This is video lecture seen from 'competitive programming courses' coding blocks' which I recorded & saved in my friend's Linux, harddrive, & uploaded on my gmail.

§ Dynamic Programming :-

→ "Those who can't remember their past are condemned to repeat it."

§ what?, where?, Examples?

⇒ e.g. $1+1+1+1+1 = 5$ (record it)

and again $\underbrace{1+1+1+1+1}_{\text{subproblem}} \underbrace{+1}_{O(1)} = 6$ → $O(n)$

so we took subproblem upto first 5 count and next count we just add to it.

what?

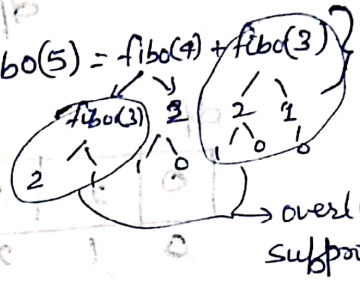
⇒ so DP is all about remembering the problems that you've already solved.

⇒ so it's just learning from past.

⑤ Where?

→ ① Optimal substructure ⟹ sum(6) = sum(5) + 1 (sum(6): Bigger problem; sum(5): subproblem) } This tech is called optimal substructu (other e.g. Merge sor Divide & Con

→ ② Overlapping subproblems ⟹ e.g. { Fibo(5) = fibo(4) + fibo(3) }

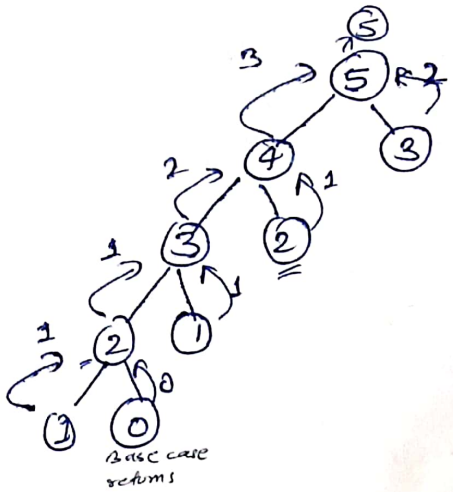


⟹ So DP will be helpful to memorize the subproblem which you've already solved.

⟹ so do not compute the same subproblems again and again.

---

⑤ e.g. fibonacci tree:



dp

0	1	1	2	3	5
0	1	2	3	4	5 →

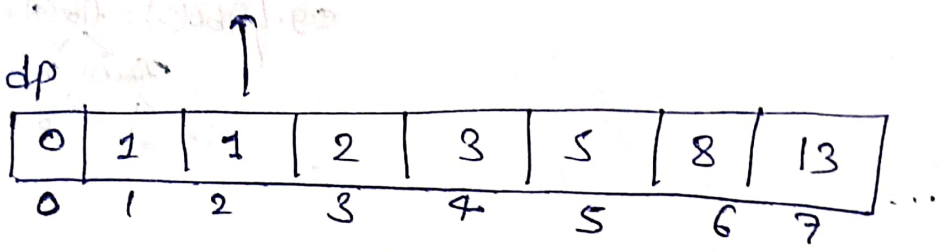
Top down approach

(Recursion + Memo(z)ation)

---

(8) Fibonacci (Button-up approach) (smaller to bigger)

$$dp[i] = dp[i-1] + dp[i-2]$$



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

⇒ So this is what top-down dp and button-up dp is.

⇒ We will see more in next lecture.

#2 Fibonacci Recursion & Call Stack

Same course continu

(8)

→ So as in last lecture video we concluded that we can use DP (where?) in problems where we've
→ overlapping subproblems and
→ optimal substructure (combine subproblems to get bigger problems)

→ we also talkeel about fibonacci series where when we want to get the n^{th} term, then we came out with ~~the~~ the very simple recurrence that n^{th} term is sum of previous two terms.

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

0 1 1 2 3 5 8

$f(n-2) + f(n-1)$ ← n^{th} term $f(n)$

(9) Recursion way of compiling fibonacci :-

→ Complexity $O(2^n)$ = exponential, which is not a good complexity.

→ so we can use DP to optimize the complexity. so while using dp. for this fibonacci the complexity will be $O(n)$. we will discuss both top-down and bottom-up approach using DP

⑤ Let's get into the coding and start implementing the code:-

⇒ so the code will be as :- (Recursive way)

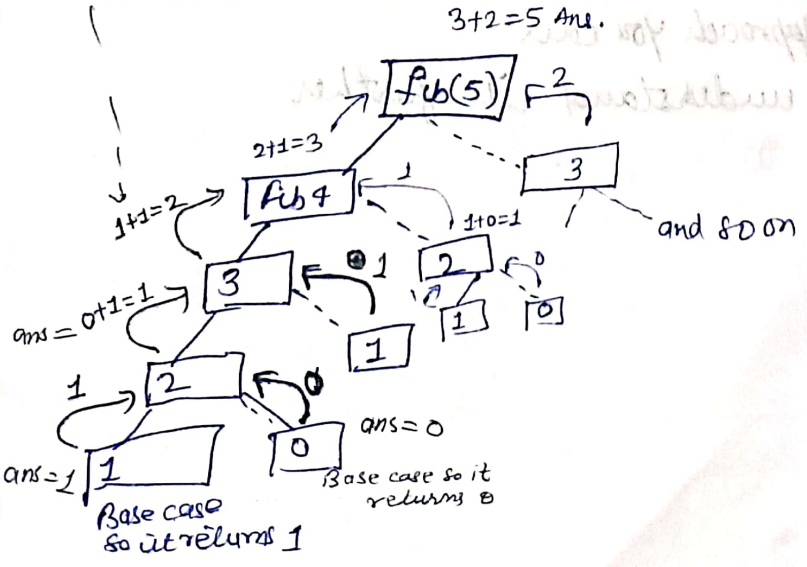
```
int fib(int n) {
    //Base case (every recursive program will have base case for termination.
    if (n == 0 or n == 1) {
        return n;
    }
    int ans;
    ans = fib(n-1) + fib(n-2);
    return ans;
}
```

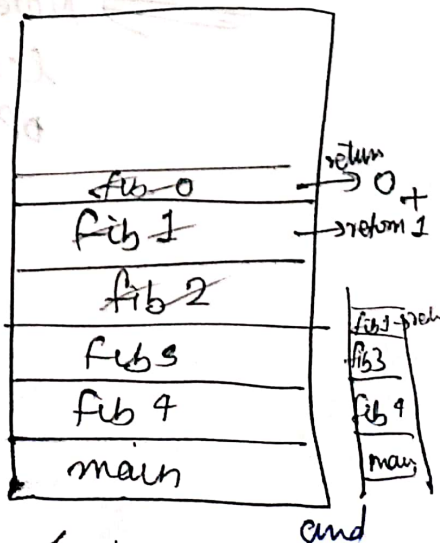
So :- fib(6) = 8 Ans

0 1 1 2 3 5 8..

base cases

⑤ More Depth of How Recursion works in Fibonacci :-

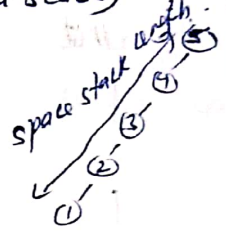




(8) So this is how the recursion tree or stack will work. But this is gonna take time.

Time Complexity → $O(2^n)$ (No. of nodes generating in the tree)

Space Complexity → Extra stack space → $O(n)$
(Max. depth of call stack)



⇒ But this is bad time complexity.
In Recursion,

⇒ So we need DP to bring down this time complexity.

⇒ Next we will see Top Down DP.

5 ☆☆☆☆

⇒ More Recursive Approach you can learn and understand in further DP questions

#3 Top Down Fibonacci DP & Space Optimization

same source

§

→ Now let's see how we can make that previous recursive solution into a dp solution.

→ This appreoach will be valid for all problems where you are able to use recursion.
So if you've overlapping subpreoblems, then you can convevet it into dp solution.

→ So the idea is we want to stovee the overlapping subproblems.

→ So we will maintain an array (dp array) to store the overlapping subpreoblem. So by this recursion we will know what states are alveady computed.
so this is "Recursion+Memoization".

Let's suppose we've an dp[] array all initialized to zero and we pass it in our function. Code as below :-

```
int main(){
    int n;
    cin>>n;
    int dp[100]={0};
    cout<<fib(n,dp)<<endl;
}
```

Our function code now this time will store the value also for calculated fib. This is Top Down Approach of dp

```
int fib(int n, int dp[]){
    //Base Case
    if(n==0 or n==1){
        return n;
    }

    //Recursive Look up
    if(dp[n]!=0){
        retuen dp[n];
    }
    int ans;
    ans = fib(n-1,dp) + fib(n-2,dp);
    retuen dp[n]=ans;

}
```

① This says that if that specific value of n!=0 means it has been computed it's value then we can get that subproblem rather than going calculating again.

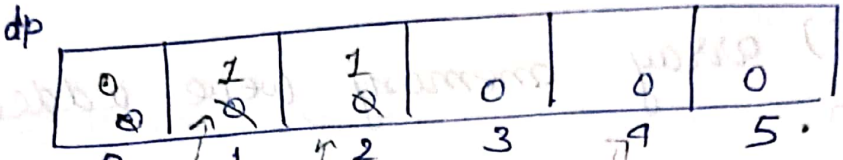
② whenever you are going to compute some value then store it in dp[n]

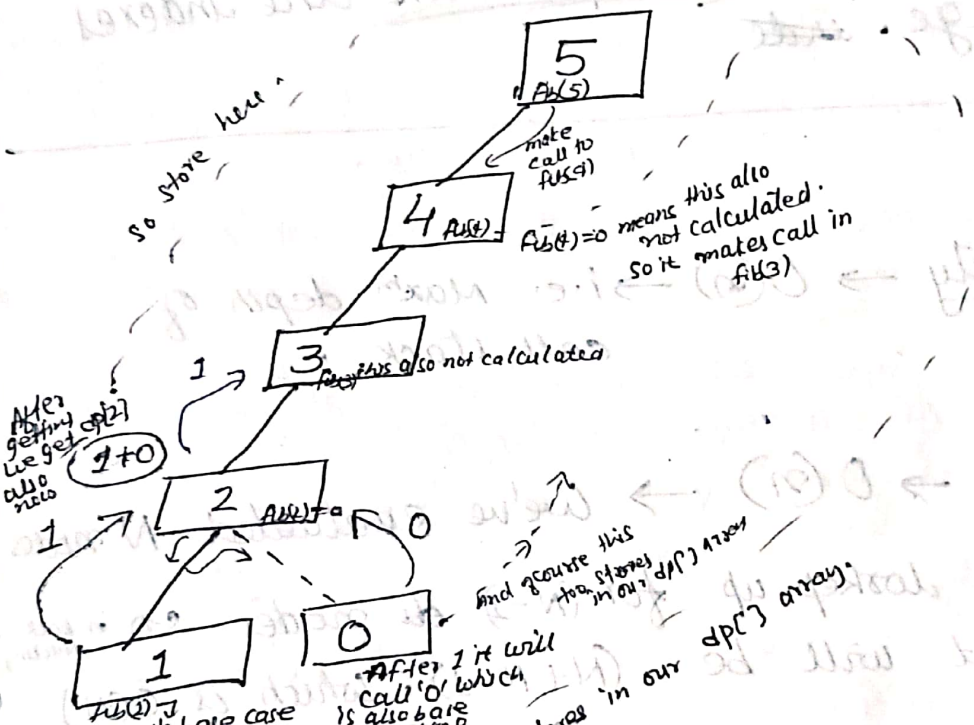
For the bottom-up approach we do

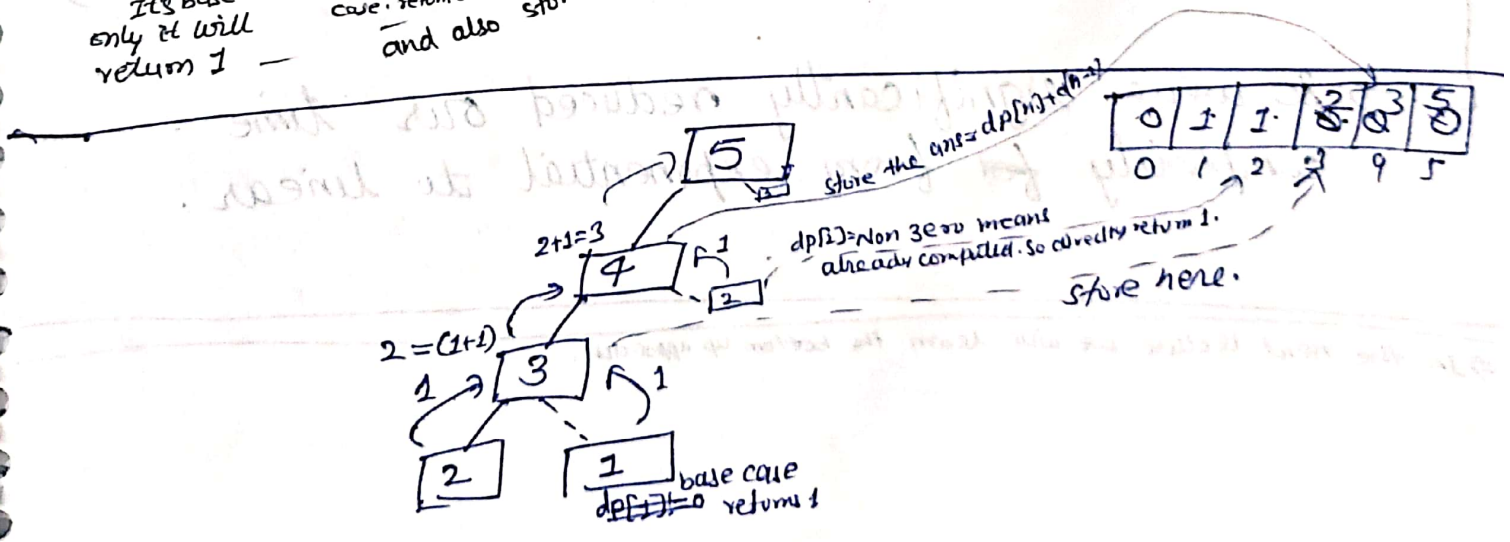
→ So this above ① & ② are 2 steps which you've do add to convert recursive solution in top down dp solution.

(5) Working of the function explained there:-

So let's have fib(5, dp) → dry run :-







⇒ So this way our code will run.

⇒ This approach is called top-down DP.

⇒ Because we are going from top i.e. 5 to bottom 1 or 0 and building up our solution.

(5) This Top-down DP can be also called as Recursion + Memoization.

⇒ So the dp[] array memory we've added to the function.

⇒ We could also use HashMap when the indexes are very large.

---

(5) Complexity:-

→ Space complexity → O(n) → i.e. Max^m depth of call stack.

→ Time Complexity → O(n) → We've evaluated N nodes and lookup up for (N-1) no nodes so in back tracking. it will be (N+N-1). which is O(N)

→ So we've significantly reduced our time complexity from exponential to linear.

⇒ In the next lecture we will learn the bottom up approach.

#4 Bottom Up Fibonacci DP & Space Optimizn

Same source.

Bottom up dp solution :-

→ The idea here is we won't be using recussion, instead we will be forming up a solution from the very basic.

In our code :-

→ Here also we will create dp[] array.

→ And instead of going from top to down, we will be going from bottom to up.

→ So after making dp[] array, we will initialize the base cases in this array. i.e. dp[0] = 0 & dp[1] = 1, and for all indexes starting from 2 to n, if we want to calculated some i^{th} index. then we can clearly say, it is sum of previous two values. i.e.

$$dp[i] = dp[i-1] + dp[i-2].$$

And we can repeat this for all states.

→ So we can implement this recurrence using loop. and time complexity will be O(n). and since we are using extra array it will take space of O(n).

(5) Code :- Let's assume max n = 100 as constraint taken for large arrays.

```
int fib(int n) {
    int dp[100] = {0}; // Initializing all values to 0
    dp[1] = 1;         // Base case & 0 & 1. (0) is 0 dp[1] = 1 so
    for (int i=2; i<=n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

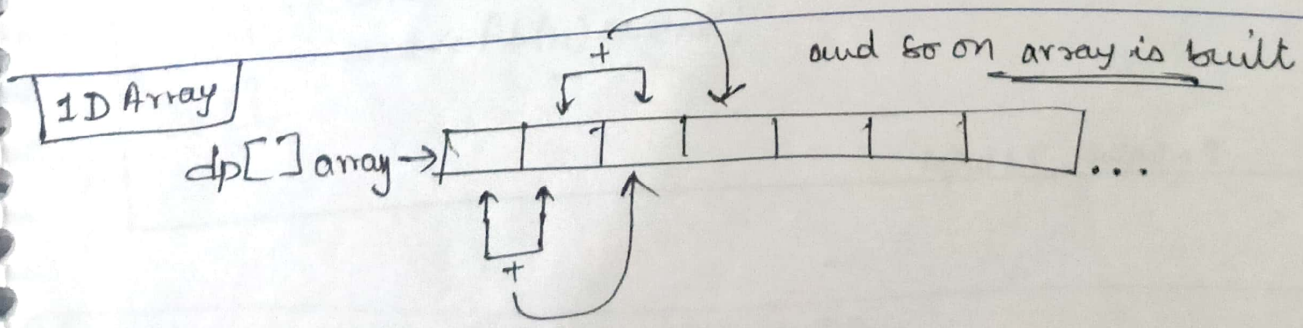
We just did was find the subproblem or substructure i.e

$dp[n] = dp[n-1] + dp[n-2]$

with initial base case & termination case and memoizing the old value.

```
int main() {
    int n;
    cin >> n;
    cout << fib(n) << endl;
}
```

input ⇒ 5 ⇒ output ⇒ 8.



Actual Code starts from here.

(5) In this code we can fwether make space optimization in our approach.

⇒ clearly we can see that in order to calculate the current number, we just need to store the last two values.

⇒ e.g. if we want to calculate 5fib of 5, we just need ans of dp[n-1] & dp[n-2]. we dont need to store entire array.

⇒ so what we can do is, we can take two vaeiables let's say, 'a' & 'b' that stores the value of last two fibonacci, and we can eliminate that array.

⇒ so by doing this, our space complexity can reduce feom O(n) to O(1).

⇒ so let's write the code ⇒

---

(5) Code for space optimized ~~Top-down~~ Bottom-up dp fibonacci :-

```
int fib(int n) {
    if (n==0 or n==1) {
        return n;
    }
    int a=0; → // as first or (n-2) stores
    int b=1; → // as second or (n-1) stores
    int c; → // our ans.
    for(int i=2; i<=n; i++) {
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
```

After every calculation & dp[n]=c we will update previous two states i.e. dp[n-1]=b & dp[n-2]=a.

→ Any time we need just previous 2 states.

```
int main() {
    int n;
    cin>>n;
    cout<<fib(n)<<endl;
}
```

Input → 5 Output → 8

⇒ So this is all about space optimization.

⇒ There is also one another approach called Matrix exponentiation. We'll also reduces time complexity to O(logn) complexity. We will cover this later as extra topic. Advanced Topic

#5 Min Steps to One (DP Problem explained and solved)

same source.

Problem statement :-

→ In this problem you are given a number 'n', and you need to convert this number into 1.

$$n \longrightarrow 1.$$

→ Also you've been given cetain conditions that you can reduce

① $n \Rightarrow n/3$ iff $n\%3 = 0$

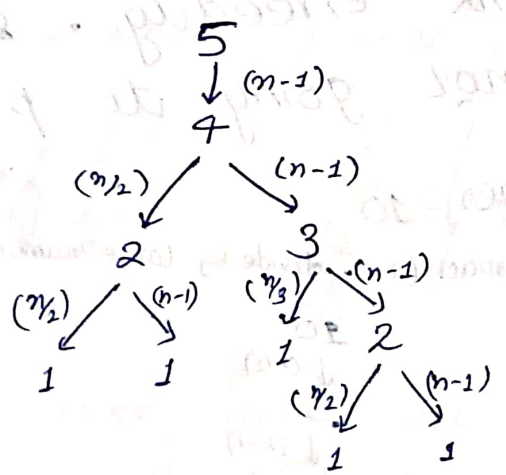
or

② $n \Rightarrow n/2$ iff $n\%2 = 0$

or

③ $n \Rightarrow n-1$

→ e.g. we've given number "5" and to reduce this to 1 we've following steps :-

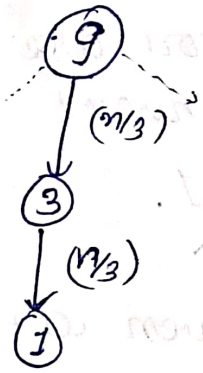


→ So we need to find what is minimum ~~no.~~ number of steps we need to make a number to 1.

(9) continue...

→ e.g. 2:- Let's take a number '9'.

We will do as following steps :-



So here we only require two steps.

⇒ But do you think that, taking a number to be divided by larger number to reach a smaller subproblem is optimal. (this kind of approach that)

⇒ so you can think of some testcases and find out the answers.

⇒ But this approach is not optimal.

⇒ So if we think 'Greedily'. So Greedy approach is not going to pass.

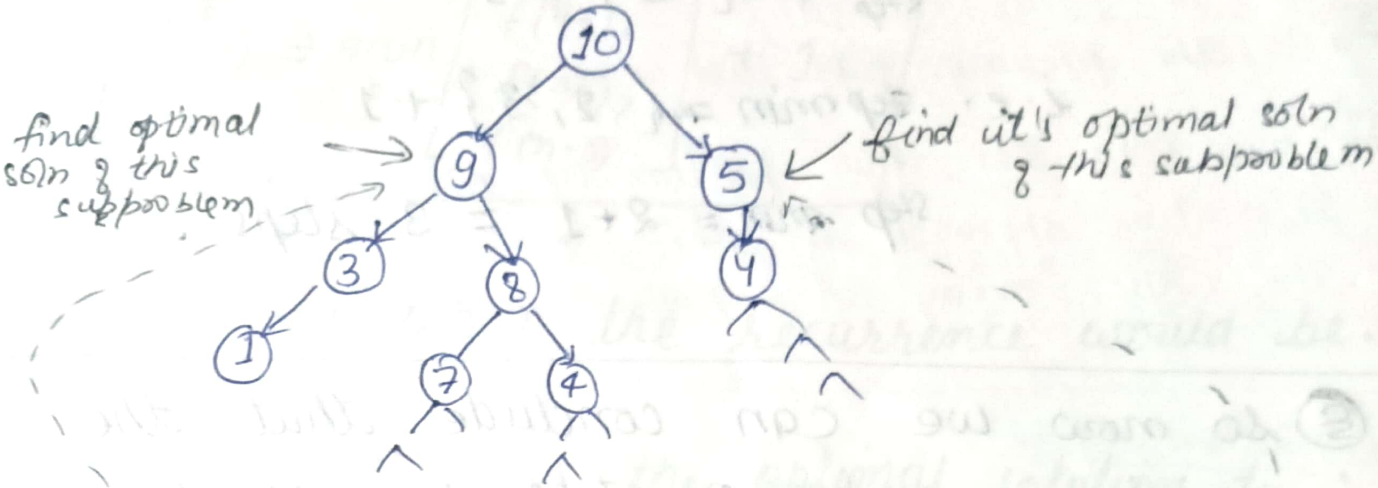
⇒ e.g. given number = 10

Applying Greedy Approach (i.e. divide by large number to reduce it to 1 if possible)

10
↓ (n/2)
5
↓ (n-1)
4
↓ (n/2)
2
↓ (n/2)
1

So Greedy approach says (we require) min 4 steps.

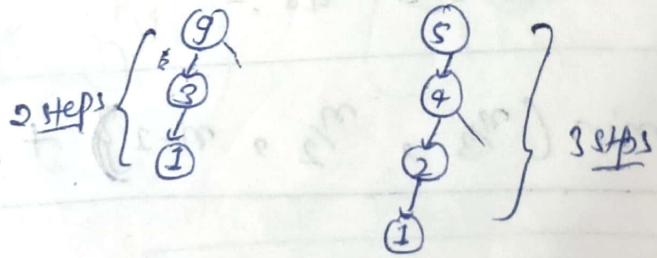
⇒ But in the ideal case :- Trying all possibilities



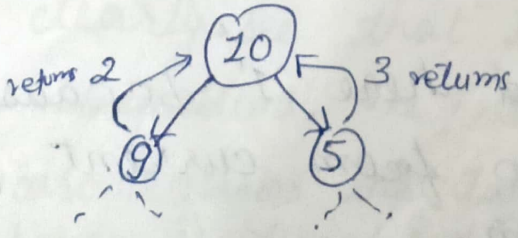
⇒ So in this approach we need to find the shortest path.

⇒ But here we only need to find out is, what is optimal answer for (9) subproblem and for (5) subproblem

⇒ For (9) we already know that we need 2 steps we computed previous page see.
and for (5) we know optimal answer will be 3 steps.



⇒ So our tree now will be as



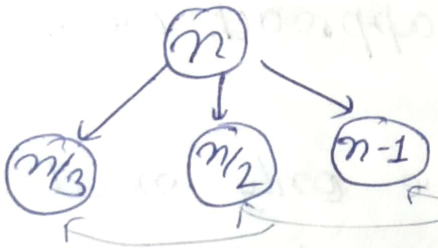
⇒ so now in order to convert 10 into 1 we need $\min_{step} = \min\{9, 5\} + 1$

i.e. step $\min = \min\{2, 3\} + 1$

step $\min = 2 + 1 = 3$ steps.

---

(5) so now we can conclude that the minimum number of steps require to go from 1 subproblem to another like below



So we need to compute answer for these subproblems. let's say $x = n/2$ $y = n/2$ & $z = n-1$.

⇒ so let $f(n) \rightarrow$ min^m no. of steps to make a number 'n' to 1 can be written as :-

$$f(n) = \min(n/3, n/2, n-1) + 1$$

$$f(n) = \min(x, y, z) + 1$$

⇒ we've added here '1' because we need '1' step to go from current problem to its subproblem.

⑤

⇒ so this eqn. further can be written as :-

$$f(n) = \min \begin{bmatrix} f(n/3), \\ f(n/2), \\ f(n-1) \end{bmatrix} + 1$$

⇒ so this is what the recurrence would be. ~~like~~.

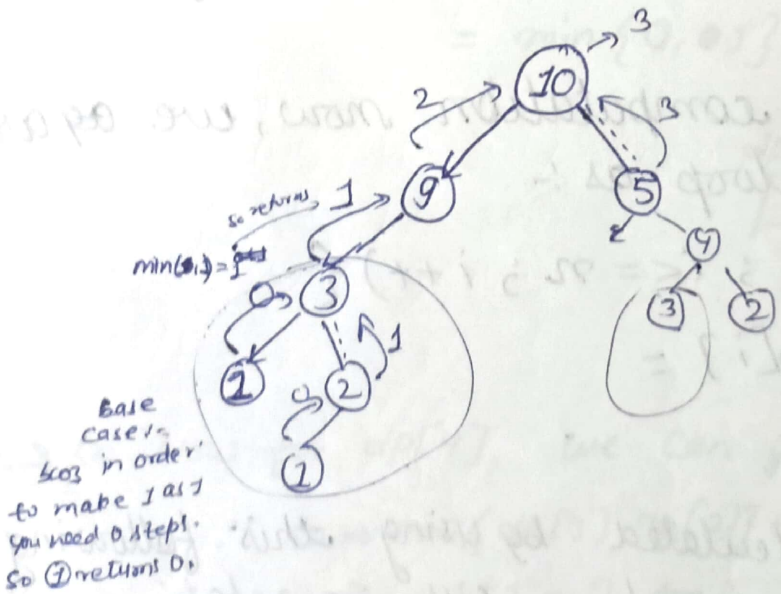
⇒ And also this is the optimal solution to solve the problem.

↳ so don't pick which is going to reduce your problem immediately,

↳ but pick the one which is globally optimally. which gives the best solution.

---

⑤ so let's dry run this problem :-



⇒ Here we can clearly see that there are many overlapping subproblems. like in ③ & ④ above etc.

⇒ And And it also satisfies the two property that we talked about :- ① optimal substructure &

② Overlapping

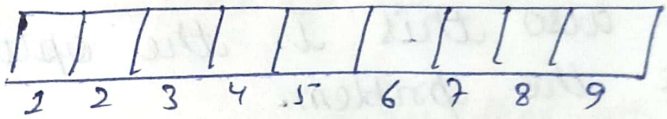
⑤ So if we've this two properties then we can solve this problem using DP.

So the previous approach was Top-down approach. (Recursive + Memoisation)

Now let's see for bottom-up approach. (It's just an iterative approach in which we just need to compute the dp[] array in a bottom-up manner)

---

⑤ Bottom-up :-



→ In this firstly we need to know is what is dp[] of 1st index for best possible case.

→ we can clearly say that dp[1] = 0, because in order to convert 1 to 1 we only need zero steps.

→ so for further computation now, we are going to use a loop as :-

```
for (i=2 ; i<=n ; i++) {
    dp[i] =
}
```

→ dp[i] can be calculated by using this following or can be expressed as following relation

$$f(n) = \min[f(n/3), f(n/2), f(n-1)] + 1$$

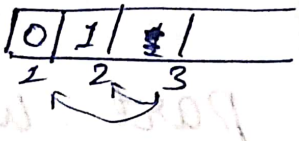
→ So what we are going do do is for dp[2], by what states can we reach '1'. So its either by dividing by 2 or subtracting 1 from.

⇒ so you will reach to $f(\frac{n}{2})$ or $f(n-1)$ so

min^n of $\{dp[1], dp[1]\} = 0$ and add 1 to it.

Therefore our final answer will be 1.

so $dp[2] = 1$.



→ for $dp[3]$, for 3 we can end up at 2 by subtract 1.

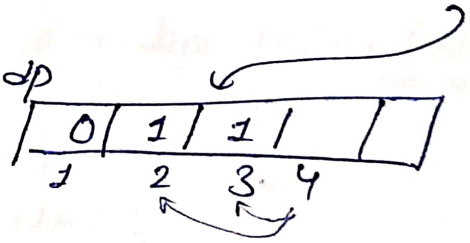
or can end up at 1 by dividing it by 3.

so these are the two states that we can reach, so min of 0 & 1 is going to be 0 and if we add 1 it will become 1.

i.e. $dp[3] = \min\{dp[\frac{3}{3}], dp[3-1]\} + 1$

$= \min\{dp[1], dp[2]\} + 1$

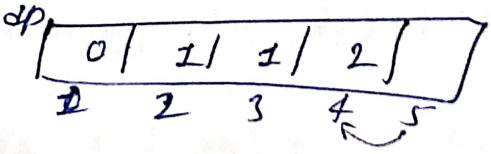
$= \min\{0, 1\} + 1 = 0 + 1 = 1$



→ so Now for $dp[4]$, we can go either to 3 $(4-1)$ or 2 $(4/2)$

$\therefore$ so $\Rightarrow \min\{dp[3], dp[2]\} + 1$

$\Rightarrow 1 + 1 = 2$ Ans.



→ for $dp[5]$, here we can end up in only one state: i.e $5-1=4$

$\therefore$ so $\Rightarrow \min\{dp[4]\} + 1 \Rightarrow 2+1 \Rightarrow 3$

and so on this way it continues.

(5) So this way you compute upto n.

⇒ So the overall complexity will be O(n).
For both Top-Down and Bottom-up DP.

⇒ In the next part we will see the code implementation.

#6 Minimum Steps to One - Top Down DP

Same source.

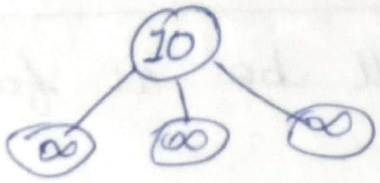
So the code will be as follows:-

```
int main()
  dp[100]={0}
ans = minSteps(n, dp);
```

```
int minSteps(int n, int dp[]) {
    //Base case
    if(n==1) {
        return 0;
    }

    //Recursive case
    //Lookup if n is already computed
    if(dp[n] !=0){
        return dp[n];
    }

    //compute if dp[n] is not known (for first time it's being computed then)
    int op1, op2, op3;
    op1 = op2 = op3 = INT_MAX;

    if(n%3==0) { //we compute option 1 only when it's divisible by 3
        op1 = minSteps(n/3, dp)+1; //and then store it's value from previous dp[n/3]+1
    }
    if(n%2==0) { //we compute option 2 only when its divisible by 2
        op2 = minSteps(n/2, dp)+1; //and then store it's value from previous dp[n/2]+1
    }
    op3 = minSteps(n-1, dp)+1; //this option 3 is always applicable
    int ans = min(min(op1, op2), op3); //final ans will be min^m of above 3 options.
    return dp[n] = ans;
}
```

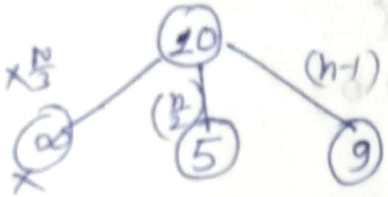
(8) Now let's see dry run of this code :-

e.g. N = 10

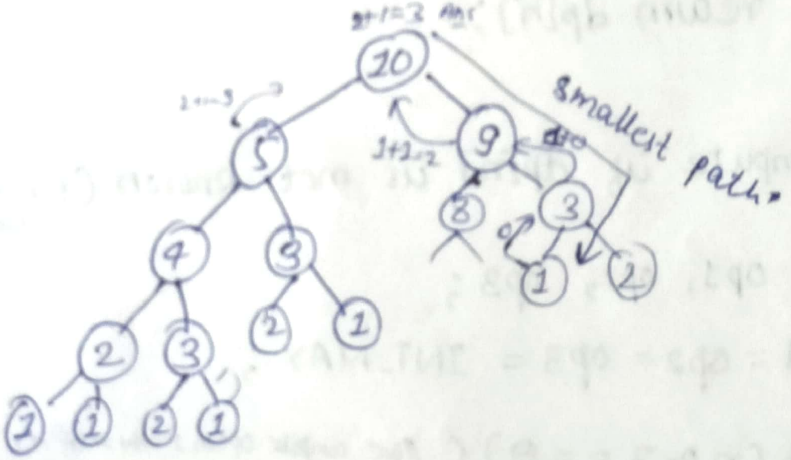


← op1 = op2 = op3 = INT_MAX.

→ But N=10 is divisible by 2 :- but Not by 3 so



→ Now '5' will make call on 4 and then 4 on 2 & 3 as follows and so on if possible operations are possible this.



→ So we've mentioned above the smallest path. So final answer is going to be 3.

→ And if some states are already computed then you won't be computing that again. So we use memoization thing.

→ Our Complexity will be as O(n).

#7 Minimum steps to one – Bottom-up DP

same source.

(5) Codes as will be :-

```
int minSteps(int n) {
    int dp[100] = {0};
    dp[1] = 0; //Base case (Initial case since Min^m steps to '1'
    for(int i=2; i<=n; i++){
        int op1, op2, op3;
        op1 = op2 = op3 = INT_MAX;
        if(n%3==0){
            op1 = dp[i/3];
        }
        if(n%2==0){
            op2 = dp[i/2];
        }
        op3 = dp[i-1];
        dp[i] = min(min(op1,op2),op3)+1;
    }
    return dp[n];
}
```

Here also we can have created full dp[100000] like in very last page you see for understanding to get for each 'n'

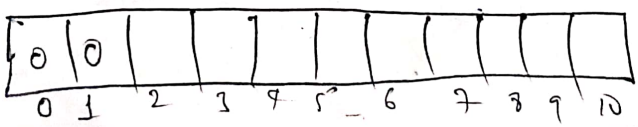
100000 for full dp

So to reach 2 from 2 requires 0 step

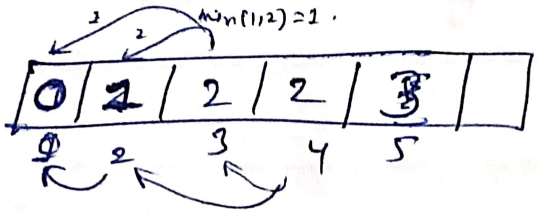
// i is the current n to be computed.

rest just do +1 for each upper to lower subproblem. or say for each step count.

understood in one go these days



⑤ Dry run of this code :-



i=2

for dp[2] ⇒ dp[2-1] only so ⇒ dp[2] = 1 + 1 ⇒ 2 Ans.

---

i=3 for dp[3] ⇒ min{dp[3/2] or dp[3-1]} + 1

⇒ min(1, 2) + 1 = 1+1 = 2 Ans

---

i=4

for dp[4] ⇒ min{ dp[4/2], dp[4-1] } + 1

⇒ min{ dp[2], dp[3] } + 1

⇒ min(2, 2) + 1 = 2 + 1 = 3 Ans

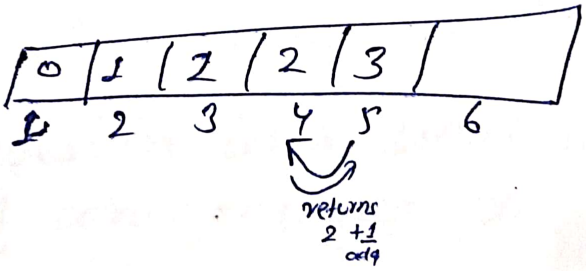
---

i=5

for dp[5] ⇒ min{ dp[5-1] } + 1

⇒ min(dp[4]) + 1

⇒ 2 + 1 ⇒ 3



and so on it goes.

---

⑤ so time complexity ⇒ O(n)
space complexity ⇒ O(n) (array of dp space).

#8 Minimum Coin Change :-

same source.

Problem statement :-

→ You have coins of certain denominations.. Let's say :- Coins = [1, 2, 5, 10, 20, 50, 100, 200, 500, 2000]

infinite coins of same type.

→ Now you need to make the change for certain sum rupees. Let's say I want a change for N = 137 rupees.

→ So we need to make N = 137 rupees change by using minimum number of coins from the available denomination.

→ For each type of coins/currencies you've infinite coins of same type. Which means you can use the same coins multiple times.

→ So our Question is :- what is the minimum number of coins required to make n = 137 rs.

↓

So we will do by asking min coin requirement value possible to give your answers. Thats how substructure or sub-problem is found.

(5) Approach :-

→ The first approach that might come to your mind again would be a greedy approach.

→ So in Greedy approach we will try and pick larger possible currency to get the required sum in few coins only.

coins = [1, 2, 5, 10, 20, 50, 100, 200, 500, 2000]

So for n = 137

let's first pick the largest coins less than 137.

so we pick 100 Rs and now we need

137 - 100 = 37 Rs.

In order to get 37 we pick next largest currency i.e. 20 :

so 37 - 20 = 17 Rs.

So since this coins[] array is a sorted array so you can use binary search to find that largest coin less than required

so now we are left with 17 Rs.

To get 17 we pick 10 ⇒ 17 - 10 = 7 Rs.

For 7 we pick 5 ⇒ 7 - 5 = 2 Rs.

For 2 Rs we pick 2 ⇒ 2 - 2 = 0.

⇒ So we've picked [100, 20, 10, 5, 2]. So min^m number of coins = 5.

(5) continue... :-

→ But now we need to discuss whether this Greedy approach is optimal or not.

→ So you may say that this is the right appreoach, but actually it is not.
However, this type of approach is true for Indian currency.
But you've to make testcases that it works for all other currency also.

→ So let's check one counter case in which this Greedy approach will not give optimal solution.

→ So let's we go to another country where coins denomination care as follows:-

coins = [1, 7, 10].
and we want sum N=15.

so by Greedy approach ⇒
15 - 10 = 5
5 - 1 = 4
4 - 1 = 3
3 - 1 = 2
2 - 1 = 1
1 - 1 = 0

so we got to need ⇒ [5, 1, 1, 1, 1, 1] → 6 coins which is not correct answer.

→ So the best answer for above test case would be [7, 7, 1] → is es only using 3 coins.
So clearly Greedy approach in this case is not applicable

(5) So other solution approach that we will be using here is dynamic programming approach where we will see all poesible options.

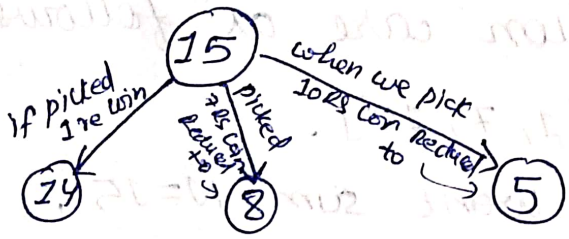
⇒ So DP is just optimisation of Recursive formulation, so recursive formulation will allow us to consider all possible cases.

---

(5) DP approach :-

→ So taking that same previous test case :- i.e. coins = [1, 7, 10] and N = 15.

so let's start with 15 rs. i.e. 'N=15' now on what other case can we land up.



→ So these (14), (8) and (5) are our subproblems. And if we know solution to these subproblems, we can definitely get the solution for (15).

→ So let's say to make sum of
(14) → we need x no. of coins.
(8) → we need y no. of coins
(5) → we need z no. of coins.

(5) continue.... :-

→ So our answer will be as follows

$$ans = \min(x, y, z) + 1$$

or

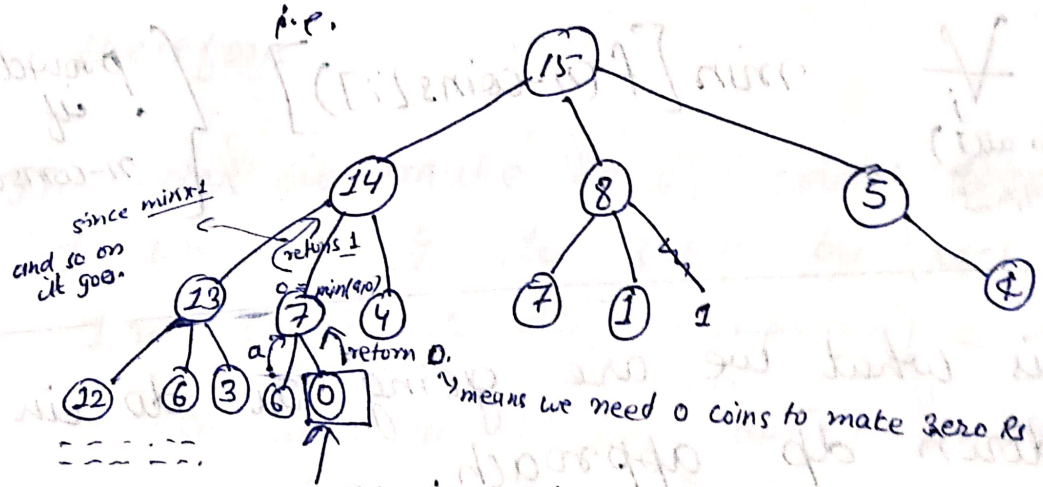
$$ans = \min(n - 0^{th}coin,\ n - 1^{st}coin,\ n - 2^{nd}coin, \ldots\ n - (last\ coin)) + 1$$

$$ans = \min(n - coins[i]) + 1 \quad \text{for all } i = 0 \text{ to } n$$

→ So it's similar to "minimum steps to 1 problem" that we discussed in last lecture videos.

→ Also since this approach follows optimal substructure, so you can breakdown every problem in a same manner.

i.e.



"0" means we need 0 coins to make zero Rs sum.

So this zero becomes our base case.

→ So this way we compute and return in Top-Down approach.

→ So here we are trying all possible coins.

(§) Therefore we finally concluded :-

$$f(n) = \min\left[f(n - coins[i]) \right] \quad ; \forall \text{ (for all } i\text{)}$$

provided if $n - coins[i] \geq 0$.

so you don't pick a coin whose denomination is greater than the value of n.

e.g. to get sum = 7 rs we cannot pick 10 or ~~Rs~~ 20 Rs coin.

(§) Our DP formula :-

$$f(n) = \underset{\substack{i \\ \text{(for all } i\text{)} \\ coins = [i^{th}\ coin]}}{\forall} \min\left[f(n - coins[i]) \right] \left\{ \begin{array}{l} \text{provided} \\ \text{if} \\ n - coins[i] \geq 0 \end{array} \right\}$$

⇒ So this what we are going to do in a top down dp approach.

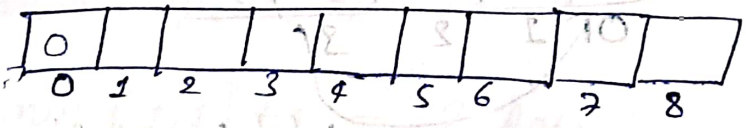
(5) In case of bottom-up DP we will doing as follows

→ Let's say coins = [1, 3, 5] and N = 8 Rs.

→ So there are multiple ways to make 8 Rs. But shortest we can see will be 3 & 5 rs coin.

→ So let's say we've a dp[] array. and since this '8' i.e. 'N' is described by one variable, so this would be 1D array.

→ so dp[] =



→ So in order to make a change for '0', i.e. N=0, dp[0] we need 0 coins.

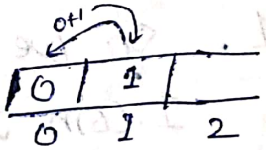
therefore

→ In order to make 1 Re coin:- Then from 1 we can go to zeroo by picking a 1 re coin. ∴ dp[1 - 1rupee coin] = dp[0] = 0

So this would be dp[1] = dp min (dp[0]) + 1

= 0 + 1 = 1 Ans.

So we need 1 coin to make sum = 1.

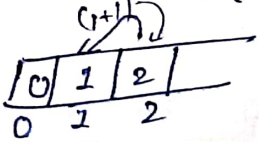


→ Now for making 2 Rs coins:- so we can pick 1 re coin and end up at dp[1].

i.e. dp[2] = min (dp[2 - 1re coin]) + 1

= min (dp[1]) + 1 = min (1) + 1 = 2

So we need two coins to make sum



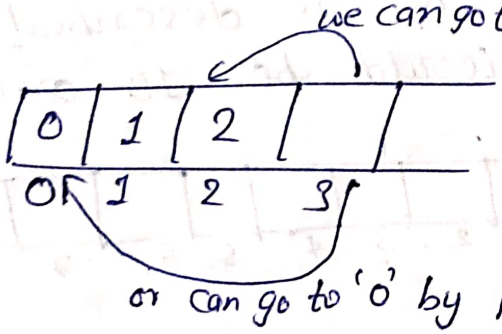
and so on it goes.

(8) continue...:-

→ for N = 3 :- coins = [1, 3, 5]

we can either pick one rupee coin
or pick three rupees coin.
But can't pick five rupees coin because
it does not satisfy condition → (n - coins[i] ≥ 0).

so from 3 :-



So let's check which of these two choices which are optimal.

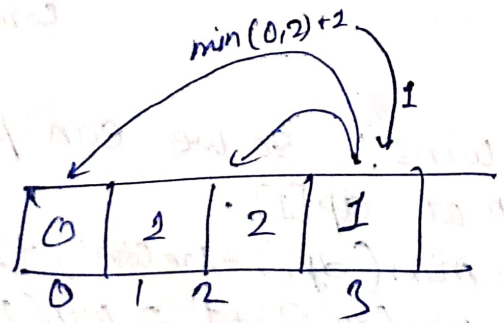
$$\therefore dp[3] = \min(dp[3-1\text{re coin}], dp[3-3\text{rs coin}]) + 1$$

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

$$= \min(2, 0) + 1$$

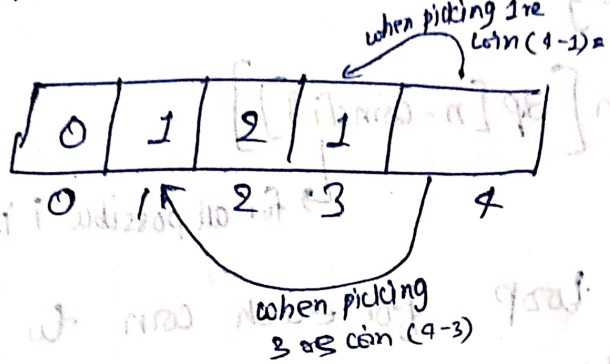
$$= 0 + 1 = 1.$$

So we require min^m 1 coin to make sum of Rs 3 from given coin set.
And also this is optimal answere.



⑤ .continue.. :-

→ for N = 4 :- coins = [1, 3, 5]

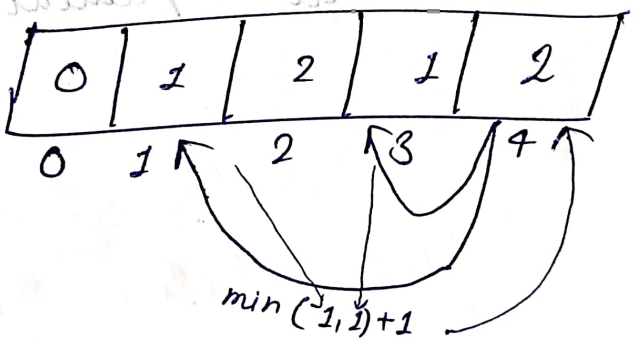


so :- $dp[4] = \min(dp[4-1\text{re coin}], dp[4-3\text{re coin}]) + 1$

$= \min(dp[3], dp[1]) + 1$

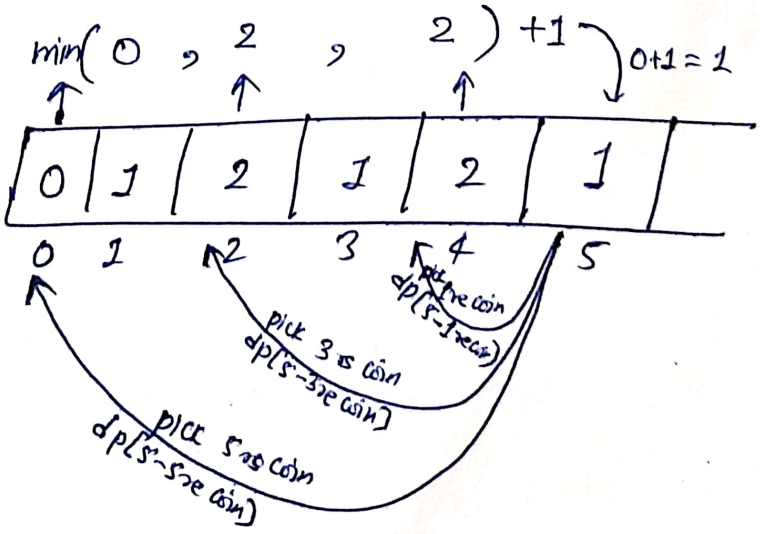
$= \min(1,1) + 1 = 2$ Ans.

so we require 2 coins to make sum = 4 Rs.



⑤

→ For N = 5 :-



and so on it goes.

(5) So this is how bottom-up dp works.

⇒ About time complexity :-

$$dp[n] = \min\left[dp[n - coins[i]]\right]$$

↳ for all possible i in coin set.

So we make loop for each coin to get upto N sums.

So let's say we have 'T' type of coin.

then the time complexity = $O(TN)$

T = Coins.length

N = Required sum we need to make.

⇒ So now in next lecture we will implement these codes.

#9 Minimum Coin Change – Top Down DP

Same source

Approach :-

→ As we see in last lecture the formula we depict is :-

$$dp[n] = \min_{i}\left(\forall \; dp[N - coins[i]]\right) + 1$$

Top Down DP Code :-

```
int main() {
    int N = 15 ; // change sum we want to make
    int coins[] = {1, 7, 10} ;
    int dp[100] = {0} ;
    int T = sizeof(coins) / sizeof(int) ;
    cout << minCoins ( N, coins, T, dp ) << endl ;
    return 0 ;
}
```

⑤ Min Coins :-

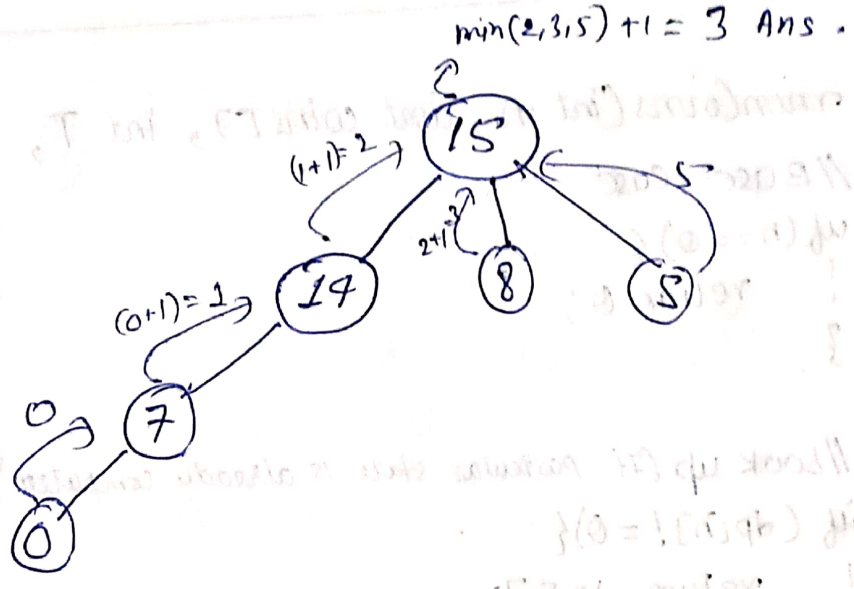
```
int minCoins(int n, int coins[], int T, int dp[]){
    //Base case
    if(n==0){
        return 0;
    }

    //Look up (If particular state is already computed)
    if(dp[n]!=0){
        return dp[n];
    }

    //Recurrence Case (In case answer is not known, means first time)
    int ans = INT_MAX;
    for(int i=0; i<T; i++){
        if(n-coins[i]>=0){ //means if you want sum(n)=5, then you can't pick coin with denomination greater than 5.
            int subprob = minCoins(n-coins[i], coins, T, dp);
            ans = min(ans, subproblem+1);
        }
    }
    dp[n] = ans;
    return dp[n];

}
```

(∞, 1, 7, 10, 2) 15
(14, 8, 5) from these.

Input: N=15, coins[] = [1, 7, 10]
Output = 3 (ie, 7, 7, 1)

⇒ So we've already seen how this works.

(5) N = 15



---

(5) So in the for loop we've just made the call of on all subproblems, and we are taking minm at every state.

⇒ And answers to the current problem will be answers to the subproblem + 1.

⇒ So this is Top-Down DP approach.

⇒ Next we will see for Bottom-up DP approach.

#10 Minimum Coin Change → Bottom-up DP

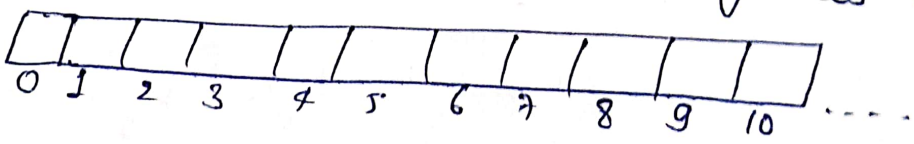
Some source

Bottom-up Approach :-

→ In Bottom-up approach we will use similar only as in Min steps to 1 in previous lecture.

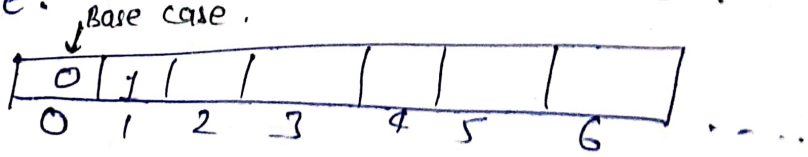
→ so here we will have to find the answer for every value of n.
and for every value we've multiple subproblems to which we can break down.

→ so we are going to form an array as below



and we are going to see what is the number of minimum coins that we are going to make denomination sum as N=1, or 2, or, 3 .... upto N and starting from N=0.

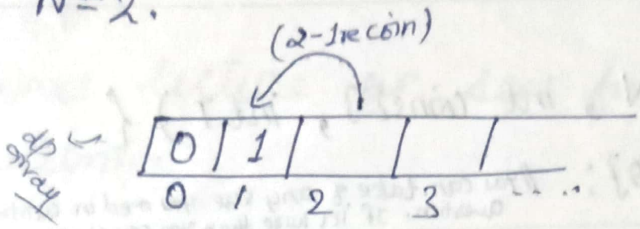
→ we also know that for N=0 the answer is going to be '0', so this would form our base case.



→ And for all other values, i.e. for N=1, we have coins[] = {1, 7, 15}, so we can pick '1' and not 7 & 15. So we just go to dp[1-1 coin] i.e. dp[0] = 0, so ans will whatever the minimum + 1. i.e. dp[1] = min(dp[0]) + 1 ⇒ dp[1] = 1.

⑤ continue ... :-

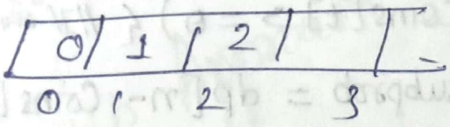
→ For N=2.



→ So we will iterate over all possible coins. where the values i.e. (n-coins[i]>=0) is possible. so in above case we choose 1re coin.

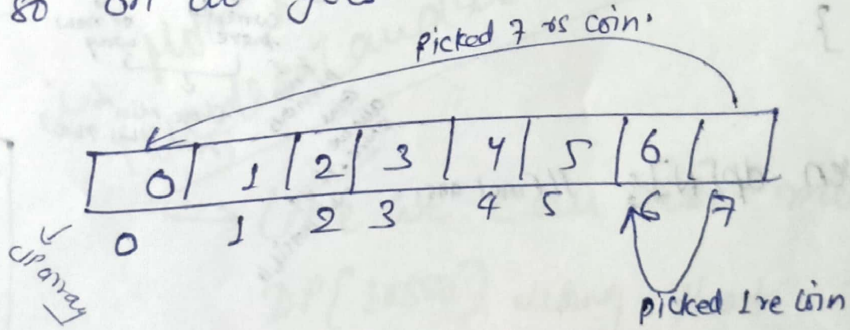
so dp[2] = min (dp[2-1re coin]) + 1

= min (dp[1]) + 1 ⇒ min(1) + 1 = 2 coins requ



→ And so on it goes.

→ So for N=7.



∴ dp[7] = min (dp[7-1re coin], dp[7-7re coin]) + 1

= min (dp[6], dp[0]) + 1

= min (6, 0) + 1

= 0 + 1 = 1 coins (ie. 7 rupees coin) only required to get N=7 as sum total

→ And so on it goes.

```
int minCoins(int N, int coins[], int T) {
    int dp[100]={0}; //You can take 8 any size you need in constraint given in question. If it's huge than you can choose arraylist.

    //Iterate over all states of Coin Sum from 1 to N
    for(int n=1; n<=N; n++) {
        //Initialize the current ans as int_max
        dp[n] = INT_MAX; //∞ initially take min coins

        for(int t=0; t<T; t++) {
            if (n-coins[t] >= 0) {
                int subprob = dp[n- Coins[t]];
                dp[n] = min( dp[n], subprob+1);
            }
        }
    }
    return dp[N]; //Final ans.
}
```

dp[0]=0 always

Amount

Array for full DP preparation.

num of different of coins

//Negative amount. (5-Jars) → so don't take Jars coin to reach 3

//If any sum is unreachable then e.g. N=1 then ∞ and '1' coin not then never reach to 0

currently there

or new way

Chose Min^m from this two

From previous coins also we store. It is min possible.

e.g. Simple e.g.

min(∞, x, 3) → 1, 7, 10

N=15 then

t=0 min(∞, 14) +1

t=1 min(14, 8) +1

t=2 min(14, 10) +1

So final ans

```
int main() {
    int N=15;
    int coins = {1,7,10};
    int T = sizeof(coins)/sizeof(int);
    cout<< minCoins(N, coins, T)<<endl;
    return 0;
}
```

output = 3 (required)

(§) So this all about Bottom-up DP approach for minimum coin requirement.

⇒ In next lecture we see furthermore DP related questions.

(§) Time complexity = $O(TN)$

space complexity = $O(N)$ → Array & dp size.

Minimum Coins

bottom-up Approach

is more Readable and understandable for me.

↳ like we can also make DP[10000] using that T & N loop and then given the ans directly by DP[N]

↓

sum value required.

Overall Understanding :- (Minimum Coin Change)

→ We can understood by taking a whole dp[] array whose each index is the required sum we need to collect with possible minimum use of coin present. With each designated coin is present or can be used infinite or multiple times.

→ So we better update of 'dp[]' array from 0 to 10^5 as sum taken. So that for bigger sum we can take it back from previous sum present in dp[].

→ So it's just bottom-up approach. But in competitive programming taken constraints as :- ⇒ collecting subproblems in dp.

coins[] = {1, 7, 5} → Testcase
Sum = 15 (required sum)

Constraints
↳ $0 < Sum < 10^5$

Code in JAVA :-

```
int[] dp[100000] = new int[];

dp[0] = 0; // First updated case when sum=0 we need '0' coins only.

Arrays.fill(1,100000) = Integer.MAX_VALUE; // Rest of the dp for now fill with ∞.

//We will find minm coin required for each sum (dp[i]), else ∞ (i.e. Int.MAX_VALUE) means never found.

for(int sum=1; sum<=100000; sum++){
    dp[sum] = Integer.MAX_VALUE;
    for(int c=0; c<coins.length; c++){
        if(sum-coins[c]>=0){ //Coin taken should not exceed sum required
            int previousDpValueOrSubproblem = dp[sum-coins[c]];
            //This is previous update dp value for smaller sums,
            //which we also call as subproblems.
            //Finally ans for sum
            dp[sum] = Math.min(dp[sum], previousDpValueOrSubproblem+1)
        }
    }
}
```

In case there was other way to coin change but was not minm enough then. Use.

So there for compare to minm value.

+1 Below 1 step to move to that previous stored dp[] array valued sum subproblem.

From these 3 subproblems we would min possible

Sum = 15 Coins[] = {1,7,10}

So Ans

Syso(dp[Sum]) = {7,7,1} = Ans = 3

DSA

(First learning of DP)

Dynamic Programming

&

Greedy Algorithm

Related Questions and Approach

Pre-requisites

↳ You need to first understand how recursion works till deep end

Dynamic Programming :-

⇒ Dynamic Programming has similarities with backtracking and divide-conquer in many respects. Here is how we generally solve a problem using dynamic programming

1. Split the problem into overlapping sub-problems.
2. Solve each sub-problem recursively.
3. Combine the solutions to sub-problems into a solution for the given problems.
4. Don't compute the answer to the same problem more than once.

---

e.g. Normal (recursive) way of solving fibonacci :-

```
int fibo(int n){
    if (n<=2) return 1;
    else
        return fibo(n-2) + fibo(n-1);
}
```

The complexity of this program would be (exponential), $O(2^n)$

How let us solve fibonaci problem dynamically.

→ Lineare

```
int memory[500];
memset(memory, -1, 500);

int fibo(int n) {
    if(n<=2) return 1;
    if (memory[n] != -1) {
        return memory[n];
    }
    int s = fibo(n-1) + fibo(n-2);
    Memory[n] = s;
    return s;
}
```

Here we have n possible inputs to the function: 1, 2, ... n, . Each input will either :-

① be computed, and the result saved
② be returned from the memory.

Each input will be computed at most once
Time complexity is O(n*k), where k is the time complexity of computing an input if we assume that the recursive calls are returned directly from memory
O(1).

Consider a few examples of Dynamic Programming

(1) Find the ~~smallest~~ minm number of coins required to make a specific amount of change.

⇒ Look at all combinations of coins that add up to an amount, and count the fewest number.

(2) Find the most valve of item that can fit in your knapsack.

⇒ Find all combinations of items and determine the highest valve combination.

(3) Find the number of different paths to the top of a stair case.

⇒ Enumerate all the combinations of steps.

More about Dynamic Programming:-

⇒ Dynamic programming is mainly an optimization over plain recursion. (Wherever we see a recursion solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming). The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

```
int fib (int n){
    if(n<=1) return n;
    return fib(n-1)+fib(n-2);
}
```

→ Recursion : Exponential
$O(2^n)$

→ Here recursively again calling (Here repitions)

```
f[0] = 0;
f[1] = 1;
for (i=2; i<=n; i++) {
    f[i] = f[i-1]+f[i-2];
}
return f[n];
```

→ Dynamic Programming : Lineare.
$O(n)$

Here storing so no need to go recursively

(Here no repetions)

Difference between Greedy method and Dynamic Programming :-

⇒ Greedy method and DP both are used for solving optimization problems.
Both are different strategies but the purpose are same.

⇒ Optimization problems are those which either requires a minimum result or maximum result. The methods are different.

⇒ In Greedy method we try to follow predefined procedure to get optimal result.
The procedure is known to be optimal. we follow this procedure to get best results.

e.g. kruskals method for finding minimum cost spanning tree always select a minimum cost edge and thod gives us best result.

e.g. or Dijkstra's shortest path method always select a shortest path vertex and continue relaxing the vertices so you get a shortest path.

⇒ In Dynamic Programming, we will try to find out all possible solutions and then pick up the best solution (optimal solution)

This approach is different and little more time consuming than greedy method. of course

⇒ so here we go as :- for many problems there can be many solutions, so we first get all those solutions and then pick up the best one.

⇒ Mostly dp probums are solved by using recursive formulas. Though we not use recursion & programming but the formulas are recursive, they are mostly solved using iteration.

⇒ DP follows principal of optimality.
i.e. problem can be solved by taking sequence of decisions to get optimal solution.

⇒ In greedy one procedure but in DP we take decision after every step.

5