DYNAMIC PROGRAMMING

Day 2

Find the time complexity of the below codes

1.

```
typedef long long ll;
ll f ( ll x)
{
if(x==0)
   return 2;
else
   return f(x-1) + f(x-1);
}
```

Time complexity : O(2^n)

Visualise at https://recursion.now.sh/

2.

```
typedef long long ll;
ll f ( ll x)
{
if(x==0)
   return 2;
else
   return 2 * f(x-1);
}
```

Time complexity : O(n)
Visualise at https://recursion.now.sh/

Finding n^{th} Fibonacci Number using 1-d DP [Iterative Method: using loop]

f(n) = f(n-1) + f(n-2); // in recursion
F[n] = F[n-1]+F[n-2]; // when using loop

```
vector<int> DP(n);
DP[0]=0;  // Use base conditions
first
DP[1]=1;
for(int i=2;i<=n;i++)
DP[i]=DP[i-1]+DP[i-2];
```

Using constant space

0,1,1,2,3,5….

```
int prev, prev_prev, curr;
prev=1;
prev_prev=0;
for(int i=2; i<=n; i++){
    curr = prev+prev_prev;
    prev_prev=prev;
    prev=curr;
}
cout<< "f(n) = "<<curr<<endl;
```

Space Complexity = O(1);

Time Complexity = O(n);

Here, we can find the n^{th} fibonacci number in constant space, because f(n) depends only the previous 2 values f(n-1) and f(n-2), which can be stored in 2 variables, instead of taking a large array.

Iterative DP

1. Speed
2. Easier complexity
3. Shorter code
4. Slightly Harder techniques
5. Known as Bottom-up DP

Recursive DP

1. Easier to apply
2. Fewer states
3. Order doesn't matter
4. Known as Top-down DP

Order doesn't matter in recursive DP

```
Void fun(int left, int right){}
Void fun(int right, int left){}
```

Bottom Up DP

In bottom up DP, we run base condition first and then using it, we solve the larger problems
Eg. Iterative fibonacci. See above

Top Down DP

We solve larger problem using smaller sub-problem, assuming we already have solution for the smaller sub-problem.
Eg. Recursive fibonacci with memoisation

```
const int MAX=100000;
dp[MAX]; // initialise all values
with -1

int f( int x)
{
 if(x==0)
```

```
    return 0;
if(x==1)
    return 1;
if( dp[x]!=-1 )
  {
     return dp[x];
  }
return dp[x] = f(x-1) + f(x-2);
}
```

Time Complexity : O(n)

// Call f(1000); This occurs in a top down way i.e. base condition is used at last

f(1000) = f(999) + f(998)

f(999) = f(998) + f(997)

f(998) = f(997) + f(996)

……

f(1)= 1

f(0) = 0

Primitive Calculator Problem :

You are given a primitive calculator that can perform the following three operations with the current number x:
I. multiply x by 2
II. multiply x by 3
III. add 1 to x.
Your goal is given a positive integer n, find the minimum number of operations needed to obtain the number n starting from the number 1. Also, output the sequence of numbers obtained during the operations.
Link to similar problem "K. Best Wishes" :
https://codeism.contest.codeforces.com/group/qXv2tukHZE/contest/299746/problem/K
eg.
A.
Input :
1

Output:
0
1

B.
Input :
5

Output:
3
1 3 4 5

```
int dp[MAX+1];
for(int i=0; i<=MAX; i++)
    dp[i]=-1;
// dp[x] = min. Operations to reach x
from 1
int f(x)
{
    if(x==1)
        return dp[x]=0;
    if(dp[x] != -1)  // memoisation
        return dp[x];
```

```
    int ans = INT_MAX;

     if(x%2==0)
        ans=min(ans, f(x/2)+1);
     if(x%3==0)
         ans=min(ans, f(x/3)+1);
     if(x-1>=1)
         ans=min(ans, f(x-1)+1);
    dp[x] = ans; // memoisation
  return dp[x];
}
```

Time Complexity : O(n)

If memoisation is removed, time Complexity : $O(3^n)$

```
// Now, code to calculate sequence using DP
// This is called "traceback" in DP
int n;
cin>>n;
cout<<f(n); // Min. operations for n
vector<int> seq; // to store sequence
```

```
int i=n;  // start from n
while(i>1)
{ // Perform a traceback
 seq.push_back(i);
 if( i -1 >=0 && dp[i] == dp[i-1] + 1)
        i=i-1;
 else if( i %2 ==0 && dp[i] == dp[i/2] + 1)
        i=i/2;
 else if( i %3 ==0 && dp[i] == dp[i/3] + 1)
        i=i/3;
}
seq.push_back(1);
 // Reverse the sequence
reverse(seq.begin(), seq.end());
```

Short Circuiting in C++

```
int c=2;
int x=3;
 if(x!=3 && c++)
```

```
   {

   }
cout<<c;
```

Output : 2

```
int c = 2;
int x = 3;

if (x == 3 && c++) {
}
cout << c;
```

Output : 3

Reason : In && operator, when left condition is false, compiler doesn’t need to check for the second condition.

Similarly, In || operator, when left condition is true, compiler doesn’t need to check for the second condition.

Example of short-circuiting (more use in graph problems):

```
while(i>1)
{
 seq.push_back(i);
 if( i -1 >=0 && dp[i] == dp[i-1] + 1 &&
i=i-1)
{
}
 else if( i %2 ==0 && dp[i] == dp[i/2] + 1 &&
i=i/2)
  {
}
 else if( i %3 ==0 && dp[i] == dp[i/3] + 1 &&
i=i/3)
{
}

}
```

Passing vector to function

```
int fun(int x, vector<int> vec)
{
// O(log n)
}

int main()
{
fun(2,vec);
}
```

Note : Here, call-by-value is used to pass vector to fun()

So, time complexity : O(size of vector)

How to avoid ?

Use call by reference when passing vector

Reference is just alias (other name) of variable

```
  fun(int x, vector<int> & vec)
{
// O(log n)
}
```

Calling, code inside function, etc. would be same
Only use & sign when passing a vector
Now, Time complexity : O(log N)

Wine Selling Problem

Q) Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the i^{th} wine is pi. (prices of different wines can be different).

Because the wines get better every year, supposing today is the year 1, on year y the price of the i^{th} wine will be y*pi, i.e. y-times the value that current year.

You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).

You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?

Link : https://www.spoj.com/problems/TRT/

n wines..sell

yth year..
profit=y*p[i]

n=3-->no of bottles..
p[ ]=1 2 3
1*1=1 -->2 3
2*2=4+1=5 -->3
5+3*3==14

1 2 3
1*3==3 -->1 2
2*2+3==7 -->1
7+3*1==10

n=5
p[ ]=2 3 5 1 4-->2-->2
3 5 1 4-->3-->3*2=6
5 1 4-->4-->4*3=12
5 1-->1-->4*1=4
5-->5-->5*5=25

2 3 5 1 4-->2-->2*1=2
3 5 1 4-->4-->4*2=8
3 5 1-->1-->3*1=3
3 5-->3-->4*3=12
5-->5-->5*5=25

dp
recursive rel
no of dimensions..

city a-->city b

t -->avg speed?

total dist?
total time travelling?

yth year pass
some are left

years passed?

no of bottles left?
leftmost bottle(l)??
rightmost bottle(r)?

1 4 5
2nd year
l=2
r=3
2 bottles left
4 5
l,r

years passed?
no of bottles left?
initially no of bottles=n
p[]-->initial profit of each bottle..

l,r→(r-l+1)-->bottles left
current year-->n-(r-l)-->y
profit=cur_year*price

fun(l,r)-->return max profit left bottles..

fun(l,r)==max(fun(l+1,r)+p[l]*y,fun(l,r-1)+p[r]*y);-->recursive rel

int fun(int l,int r)-->return max of profit left

int dp[ ][ ]
fun(2,5)
fun(2,4)
fun(3,5)

fun(2,5)

bottles(from l to r)..

```
{
  y=n-(r-l);
  if(l==r)
   return (y*p[l]);
return max(fun(l+1,r)+p[l]*y,fun(l,r-1)+p[r]*y);
}
```

fun(2,5)

fun(2,4)

fun(2,5)
fun(3,5)

fun(2,5)

dp[l][r]

1 3→(2,3),(1,2)
(2,3)-->(2,2),(3,3)
(1,2)-->(1,1),(2,2)

```
for(int i=0;i<n;i++)
 for(int j=0;j<n;j++)
  dp[i][j]=-1;

bottles(l,r)
{
  y=n-(r-l);
```

```
  if(l==r)//base case
   return (y*p[l]);
if(dp[l][r]!=-1)//already calculated
  return dp[l][r];
return
dp[l][r]=max(fun(l+1,r)+p[l]*y,fun(l,
r-1)+p[r]*y);
}
```

l,r-->2d dp

states→