Dynamic Programming

K^th Min Element: → (n – k + 1) ➔ k = 3, ans = 4

[5, 2, 1, 7, 4, 9]

Create a heap using the first 3 digits.

If you are finding the kth min element, which is equal to n - k + 1 largest element,

Kth min element = (n – k + 1) largest element.

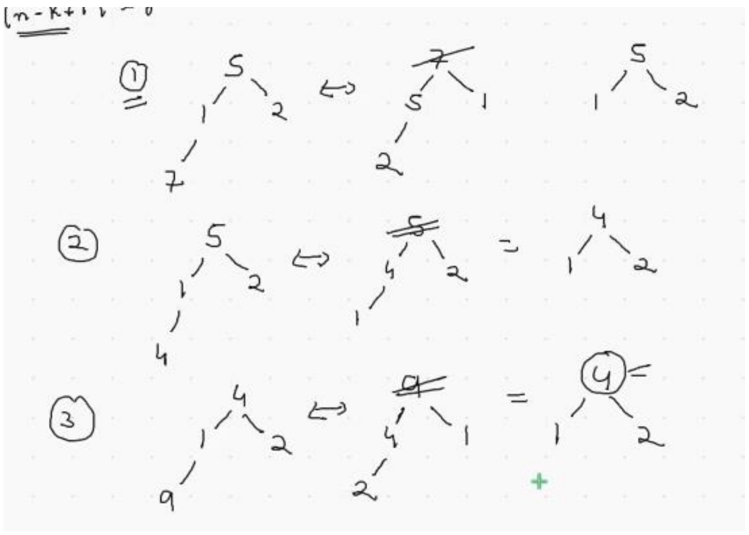
We have to find the 4th largest element. The simple way is creating a min heap and we pop from min heap k time elements. Since k times we are pop, klogn. The space complexity is ***O(n)***. How to optimize it?

We can create a max heap of size k, [5, 1, 2]. We understand to find n – k + 1 elements, so we have to pop n – k + 1 times. So the (n -k + 1)th element will be my minimum element.

So, first step we will add 7 to the min heap. This heap will be like [5, 1, 2, 7]. So the max heap would be [7, 5, 1, 2]. We will pop 7 and our heap would be again [5, 1, 2].

In the second step, we will add 4 to the heap. This heap will be like [5, 1, 2, 4]. So the max heap would be [5, 4, 2, 1]. We will pop 5 and our heap would be again [4, 1, 2]

If we add 9, heap would be [4, 1, 2, 9]. This heap would be like [9, 4, 1, 2]. So the largest element 9 would be popped. The element would be [4, 1, 2]



So, the time complexity of this problem would be ***O ((n – k + 1)logk)***

Introduction:

What is Dynamic Programming?

Dynamic programming is a problem-solving technique for resolving complex problems by recursively breaking them up into sub-problems, which are then each solved individually. Dynamic programming optimizes recursive programming and saves us the time of re-computing inputs later.

This differs from the Divide and Conquer technique in that sub-problems in dynamic programming solutions are overlapping, so some of the same identical steps needed to solve one sub-problem are also needed for other sub-problems.

This leads us to the main advantage of dynamic programming. Instead of recomputing these shared steps, dynamic programming allows us to simply store the results of each step the first time and reuse it each subsequent time.

NOTE: Dynamic programming is a special case of the larger category of recursive programming, however not all recursive cases can use dynamic programming

Why is Dynamic Programming efficient? Recursion vs. DP

If the two are so closely entwined, why is dynamic programming favoured whenever possible? This is because brute force recursive programs often repeat work when faced with overlapping steps, spending unneeded time and resources in the process.

Dynamic programming solves this issue by ensuring each identical step is only completed once, storing that step's results in a collector such as a hash table or an array to call whenever it's needed again. In doing so, dynamic programming allows for less repeated work and therefore better runtime efficiency.

NOTE: Dynamic programming essentially trades space efficiency for time efficiency as solution storage requires space not used in brute force recursive solutions.

It will be based on recursion. DP is an optimization on recursion.

If you can't forget the past, you are condemned to repeat it.

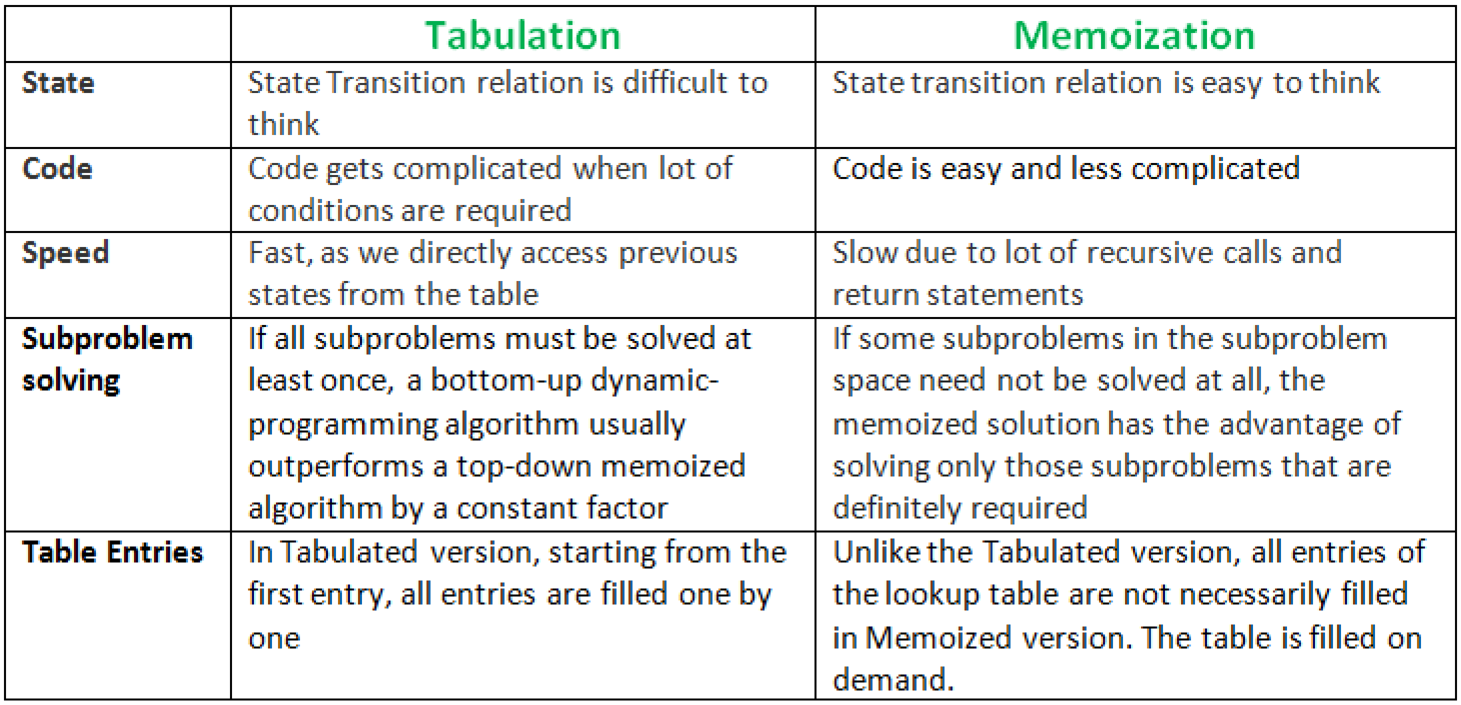
Dynamic Programming is based on this phrase. If you don't store the answer or a value, then you would keep repeating the same process again every time you need that value.

Tabulation vs Memoization:

There are following two different ways to store the values so that the values of a sub-problem can be reused. Here, will discuss two patterns of solving DP problem:

1. **Tabulation:** Bottom Up
2. **Memoization:** Top Down

Both the above versions say the same thing, just the difference lies in the way of conveying the message and that’s exactly what Bottom-Up and Top-Down DP do. Version 1 can be related to as Bottom-Up DP and Version-2 can be related as Top Down Dp.



	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

If you can’t forget the past you care condemn to repeat it

The two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

- Optimal Substructure:
    - If somebody asks you count, max, min we need to think if we can apply for dynamic programming.
- Overlapping Subproblem
    - You are again and again doing the same computation, then dynamic programming can be applied.

Let’s suppose we have x variable, and we did some computations. Again in future, if we are doing the computation again, if we have stored the computations previously done then we can use those computations again.

1. Whenever we see dynamic programming problems, we would be actually make recurrence.

2. We will be storing the repeated recurrence.

First, we will be creating recurrence and then we will be storing the values in array or dictionaries.

a) These are hacks, first we create recurrence and then will store the values

b) A function f(x), the arguments which are changing arguments in a recurrence are called state. In this recurrence the x is changing, which is called **state of Dynamic Programming**. Any argument in recurrence which will change is called “state”. If we have one argument that change, we are called **1D DP.** If there are two arguments which are changing, we called it 2D DP.

Q) Fibonacci Series: Given a number ‘n’, find the nth Fibonacci Number.

[0, 1, 1, 2, 3, 5, 8]. Find the 4th Fibonacci Number.

We can see recursion call

To make a recursion of this function, we can write,

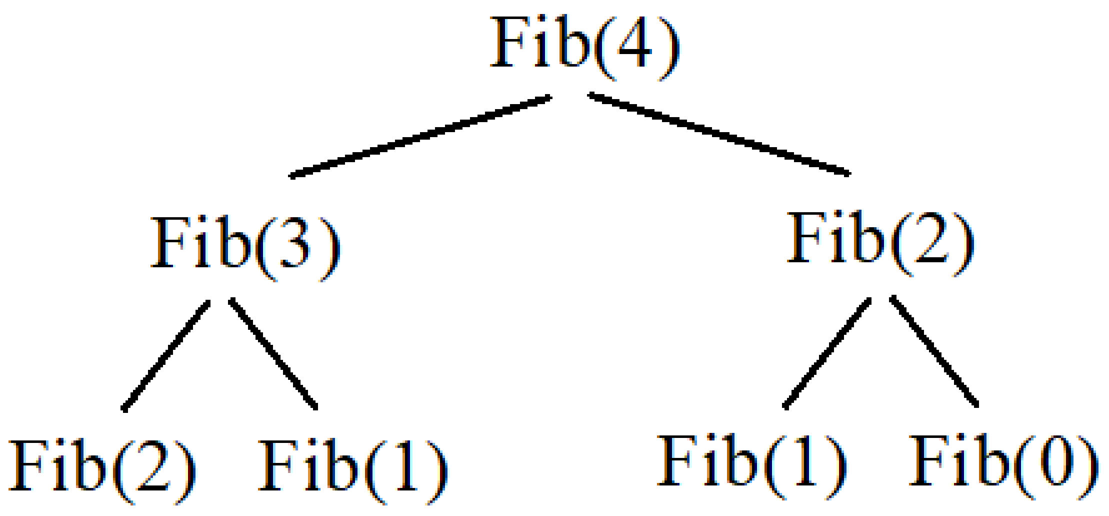
```python
def Fib(num):
    if num == 0:
        return 0
    if num == 1:
        return 1

    return Fib(num - 1) + Fib(num - 2)
```

If we call Fib(4), it will give Fib(3) + Fib(2). Now,

Fib(3) = Fib(2) + Fib(1) and further more

Fib(2) = Fib(1) + Fib(0)



We are calling Fib(3) & Fib(2), which is recursion. So, it means we can use dynamic programming in this problem.

The first way to optimize is "**memorization**" and the second way is called "**Tabular DP**".

Since only number is changing, it is 1d. We will only create a 1d array. For a 2d dp, we will create a 2d array. In this question we would be creating a one-dimension array.

If we create a DP array and assign everything to [-1] * 10005 (a very big value).

Now, in this array, if 3rd index value is -1 then we have not yet got the answer of ***fib(3)***. If 5th index is -1, then we have not yet got any answer of ***fib(5)***

Note: *-1 means, we have not got the answer yet or we have not saved the answer yet.*

We will store our answers in the dp array. If at a particular index, the value is not -1, lets say it is 5, it means that we already have the answer for Fib(3) and we don't have to do recursion for that value and we can simply return the value.

Remember -1 means we have not saved the answer or not saved the history yet.

CODE:

```
def Fib(no):
    if no == 0:
        return 0

    if no == 1:
        return 1
    return Fib(no -1) + Fib(no - 2)

if __name__ == '__main__':
    print(Fib(50))
```

The time complexity of this code is 2^n which is very huge. So the code is running so many times.

If we pass a dp value,

```
# Memoization

def Fib(no, dp):
    if no == 0:
        return 0

    if no == 1:
        return 1

    if dp[no] is not None:
        return dp[no]

    ans = Fib(no - 1, dp) + Fib(no - 2, dp)
    dp[no] = ans
    return ans

if __name__ == "__main__":
    dp = [None] * 10005
    print(fib(35, dp))
```

in this problem DP is 1D and every element in DP is none.

[None, None, None, None]

Fib(1) will return 1. So in DP when Fib(1) return 1, then store the answer as 1.

[None, 1, None, None]

In Fib(2), we will call Fib(1) and Fib(0).

[None, 1, 2, None]

In Fib(3) we will call Fib(2) and F(1).

[None, 1, 2, 3]

For Fib(4), we will call Fib(3) = 3 and Fib(2) = 2. At Fib(2), you will check,

`dp[2] is not None`

hence, Fib(4) = 3 + 2 = 5

We are not doing computation again. Time Complexity would be O(n).

Tabulation:

In **Memorization**, we were breaking bigger problem into smaller problems; also called as top down dp, but in **Tabulation** is called bottom up dp. We will build our problem from bottom to up.

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

In this we will take all the answers to zero.

Now dp[2] = dp[1] + dp[0]

dp[3] = dp[2] + dp[1]

dp[4] = dp[3] + dp[2]

dp[5] = dp[4] + dp[3]

dp[6] = dp[5] + dp[4]

dp[7] = dp[6] + dp[5]

dp[8] = dp[7] + dp[6

this approach looks easy for this, but for bigger problems this will be tough. Memoization is very handy, when compared to Tabulation

```python
# Tabulation
def FibBottomUp(n):
    dp = [0] * [n-1]
    dp[0] = 0
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```