Minimum jumps with fee

We'll cover the following

Problem Statement

Given a staircase with 'n' steps and an array of 'n' numbers representing the fee that you have to pay if you take the step. Implement a method to calculate the minimum fee required to reach the top of the staircase (beyond the top-most step). At every step, you have an option to take either 1 step, 2 steps, or 3 steps. You should assume that you are standing at the first step.

Example 1:

```
Number of stairs (n) : 6
Fee: {1,2,5,2,1,2}
Output: 3
Explanation: Starting from index '0', we can reach the top through: 0->3->top
The total fee we have to pay will be (1+2).
```

Example 2:

```
Number of stairs (n): 4
Fee: {2,3,4,5}
Output: 5
Explanation: Starting from index '0', we can reach the top through: 0->1->top
The total fee we have to pay will be (2+3).
```

Let's first start with a recursive brute-force solution.

Basic Solution

At every step, we have three option: either jump 1 step, 2 steps, or 3 steps. So our algorithm will look like:

Java | JS | Python3 | C++

```
def find_min_fee(fee):
```

```python
  return find_min_fee_recursive(fee, 0)


def find_min_fee_recursive(fee, currentIndex):
  n = len(fee)
  if currentIndex > n - 1:
    return 0

  # if we take 1 step, we are left with 'n-1' steps;
  take1Step = find_min_fee_recursive(fee, currentIndex + 1)
  # similarly, if we took 2 steps, we are left with 'n-2' steps;
  take2Step = find_min_fee_recursive(fee, currentIndex + 2)
  # if we took 3 steps, we are left with 'n-3' steps;
  take3Step = find_min_fee_recursive(fee, currentIndex + 3)

  _min = min(take1Step, take2Step, take3Step)

  return _min + fee[currentIndex]


def main():

  print(find_min_fee([1, 2, 5, 2, 1, 2]))
  print(find_min_fee([2, 3, 4, 5]))


main()
```

The time complexity of the above algorithm is exponential $O(3^n)$. The space complexity is $O(n)$ which is used to store the recursion stack.

Top-down Dynamic Programming with Memoization

To resolve overlapping subproblems, we can use an array to store the already solved subproblems. Here is the code:

Java | JS | Python3 | C++

```python
def find_min_fee(fee):
  dp = [0 for x in range(len(fee))]
  return find_min_fee_recursive(dp, fee, 0)


def find_min_fee_recursive(dp, fee, currentIndex):
  n = len(fee)
  if currentIndex > n-1:
    return 0

  if dp[currentIndex] == 0:
    # if we take 1 step, we are left with 'n-1' steps
    take1Step = find_min_fee_recursive(dp, fee, currentIndex + 1)
    # similarly, if we took 2 steps, we are left with 'n-2' steps
    take2Step = find_min_fee_recursive(dp, fee, currentIndex + 2)
    # if we took 3 steps, we are left with 'n-3' steps
    take3Step = find_min_fee_recursive(dp, fee, currentIndex + 3)
```

```python

    dp[currentIndex] = fee[currentIndex] + \
                       min(take1Step, take2Step, take3Step)

  return dp[currentIndex]


def main():

  print(find_min_fee([1, 2, 5, 2, 1, 2]))
  print(find_min_fee([2, 3, 4, 5]))


main()
```

Bottom-up Dynamic Programming

Let’s try to populate our `dp[]` array from the above solution, working in a bottom-up fashion. As we saw in the above code, every `findMinFeeRecursive(n)` is the minimum of the three recursive calls; we can use this fact to populate our array.

Code

Here is the code for our bottom-up dynamic programming approach:

Java | JS | Python3 | C++

```python
def find_min_fee(fee):
  n = len(fee)
  dp = [0 for x in range(n+1)]  # +1 to handle the 0th step
  dp[0] = 0  # if there are no steps, we don't have to pay any fee
  dp[1] = fee[0]  # only one step, so we have to pay its fee
  # for 2 steps, since we start from the first step, so we have to pay its fee
  # and from the first step we can reach the top by taking two steps, so
  # we don't have to pay any other fee.
  dp[2] = fee[0]

  # please note that dp[] has one extra element to handle the 0th step
  for i in range(2, n):
    dp[i + 1] = min(fee[i] + dp[i],
                    fee[i - 1] + dp[i - 1],
                    fee[i - 2] + dp[i - 2])

  return dp[n]


def main():

  print(find_min_fee([1, 2, 5, 2, 1, 2]))
  print(find_min_fee([2, 3, 4, 5]))


main()
```

The above solution has time and space complexity of $O(n)$.

Fibonacci number pattern

We can clearly see that this problem follows the Fibonacci number pattern. The only difference is that every Fibonacci number is a sum of the two preceding numbers, whereas in this problem every number (total fee) is the minimum of previous three numbers.