**Minimum jumps with fee**

**WE'LL COVER THE FOLLOWING**

* + - [Problem Statement](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#problem-statement)
    - [Basic Solution](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#basic-solution)
    - [Top-down Dynamic Programming with Memoization](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#top-down-dynamic-programming-with-memoization)
    - [Bottom-up Dynamic Programming](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#bottom-up-dynamic-programming)
      * [Code](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#code)
    - [Fibonacci number pattern](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#fibonacci-number-pattern)

**Problem Statement**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

class Staircase {

  public int findMinFee(int[] fee) {

    return findMinFeeRecursive(fee, 0);

  }

  private int findMinFeeRecursive(int[] fee, int currentIndex) {

    if( currentIndex > fee.length - 1)

      return 0;

    // if we take 1 step, we are left with 'n-1' steps;

    int take1Step = findMinFeeRecursive(fee, currentIndex + 1);

    // similarly, if we took 2 steps, we are left with 'n-2' steps;

    int take2Step = findMinFeeRecursive(fee, currentIndex + 2);

    // if we took 3 steps, we are left with 'n-3' steps;

    int take3Step = findMinFeeRecursive(fee, currentIndex + 3);

    int min = Math.min(Math.min(take1Step, take2Step), take3Step);

    return min + fee[currentIndex];

  }

  public static void main(String[] args) {

    Staircase sc = new Staircase();

    int[] fee = {1,2,5,2,1,2};

    System.out.println(sc.findMinFee(fee));

    fee = new int[]{2,3,4,5};

    System.out.println(sc.findMinFee(fee));

  }

}





RUN

SAVERESET

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

**Top-down Dynamic Programming with Memoization**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#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](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

  public int findMinFee(int[] fee) {

    int dp[] = new int[fee.length];

    return findMinFeeRecursive(dp, fee, 0);

  }

  private int findMinFeeRecursive(int[] dp, int[] fee, int currentIndex) {

    if( currentIndex > fee.length - 1)

      return 0;

    if(dp[currentIndex] == 0) {

      // if we take 1 step, we are left with 'n-1' steps;

      int take1Step = findMinFeeRecursive(dp, fee, currentIndex + 1);

      // similarly, if we took 2 steps, we are left with 'n-2' steps;

      int take2Step = findMinFeeRecursive(dp, fee, currentIndex + 2);

      // if we took 3 steps, we are left with 'n-3' steps;

      int take3Step = findMinFeeRecursive(dp, fee, currentIndex + 3);

      dp[currentIndex] = fee[currentIndex] + Math.min(Math.min(take1Step, take2Step), take3Step);

    }

    return dp[currentIndex];

  }

  public static void main(String[] args) {

    Staircase sc = new Staircase();

    int[] fee = {1,2,5,2,1,2};

    System.out.println(sc.findMinFee(fee));

    fee = new int[]{2,3,4,5};

    System.out.println(sc.findMinFee(fee));

  }

}





RUN

SAVERESET

**Bottom-up Dynamic Programming**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#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**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#code)

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

* [Java](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [JS](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [Python3](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)
* [C++](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

class Staircase {

  public int findMinFee(int[] fee) {

    int dp[] = new int[fee.length + 1]; // +1 to handle the 0th step

    dp[0] = 0; // if there are no steps, we dont have to pay any fee

    dp[1] = fee[0]; // only one step, so we have to pay its fee

    // for 2 or 3 steps staircase, 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 or three steps, so we don't

    // have to pay any other fee.

    dp[2] = dp[3] = fee[0];

    for (int i = 3; i < fee.length; i++)

      dp[i + 1  ] = Math.min(fee[i] + dp[i], Math.min(fee[i - 1] + dp[i - 1], fee[i - 2] + dp[i - 2]));

    return dp[fee.length];

  }

  public static void main(String[] args) {

    Staircase sc = new Staircase();

    int[] fee = { 1, 2, 5, 2, 1, 2 };

    System.out.println(sc.findMinFee(fee));

    fee = new int[] { 2, 3, 4, 5 };

    System.out.println(sc.findMinFee(fee));

  }

}





RUN

SAVE \*RESET

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

**Fibonacci number pattern**[#](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nDNy6JDP1G#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.

**MARK AS COMPLETED**

[**←    Back**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/7nAKN0Qz67r)

Minimum jumps to reach the end

[**Next    →**](https://www.educative.io/courses/grokking-dynamic-programming-patterns-for-coding-interviews/m2EOxJ0Nkp3)

House thief

Stuck? Get help on

[**DISCUSS**](https://discuss.educative.io/c/grokking-dynamic-programming-patterns-for-coding-interviews-design-gurus/pattern-4-fibonacci-numbers-minimum-jumps-with-fee)

Send feedback

7 Recommendations