**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**Отчёт по лабораторной работе № 6**

**«Работа с сортировками»**

**Выполнил работу**

**Кулинич Павел Васильевич**

**Академическая группа №3111**

**Принято**

**Ментор, Владислав Вершинин**

**Санкт-Петербург**

**2024**

**Оценка по памяти**

**Первый код:**

1. Переменные и параметры:
   * int n1 и int n2: Каждая из этих переменных занимает 4 байта (в зависимости от платформы, но обычно это так для int).
   * int argc: 4 байта.
   * char \*argv[]: Указатель на массив строк, который также занимает 4 байта (указатель).
2. Класс Solution:
   * Класс сам по себе не занимает значительного объема памяти, так как в нем нет данных, только метод.
3. Вектор dp:
   * Вектор dp создается с размером n + 1, где n — это количество ступенек. Каждый элемент вектора — это int, который занимает 4 байта.
   * Таким образом, память, занимаемая вектором dp, составляет 4 \* (n + 1) байт.

Общая оценка памяти

Теперь давайте подытожим:

* Память для переменных:
  + n1: 4 байта
  + n2: 4 байта
  + argc: 4 байта
  + argv: 4 байта (указатель)

**Второй код**

* **string s:**
  + **Хранит указатель на массив символов и дополнительную информацию о размере.**
  + **В зависимости от реализации std::string, размер может варьироваться, но обычно это около 24-32 байт для хранения указателя, размера и емкости (в зависимости от платформы и компилятора).**
  + **Для анализа будем считать, что строка s занимает O(m) байт, где m — длина строки s.**
* **string t:**
  + **Аналогично строке s, строка t занимает O(n) байт, где n — длина строки t.**
* **int m:**
  + **Переменная, хранящая длину строки s. Занимает 4 байта.**
* **int n:**
  + **Переменная, хранящая длину строки t. Занимает 4 байта.**

**2. Матрица dp**

* **vector<vector<bool>> dp:**
  + **Это двумерный вектор, который имеет размер (m + 1) x (n + 1).**
  + **Каждый элемент вектора bool обычно занимает 1 байт.**
  + **Таким образом, память, занимаемая матрицей dp, составляет (m + 1) \* (n + 1) байт.**

**3. Общая оценка памяти**

**Теперь подытожим использование памяти:**

* **Память для переменных:**
  + **m: 4 байта**
  + **n: 4 байта**
* **Память для строк:**
  + **s: O(m) байт**
  + **t: O(n) байт**
* **Память для матрицы dp:**
  + **dp: (m + 1) \* (n + 1) байт**

**Подсчет асимптотики**

1. Временная сложность алгоритма определяется циклом, который заполняет массив dp:

for (int i = 3; i <= n; ++i) {

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

}

Этот цикл выполняется n - 2 раз (от 3 до n), что дает временную сложность ( O(n) ). Таким образом, временная сложность алгоритма составляет:

[ O(n) ]

Пространственная сложность алгоритма определяется использованием массива dp, который хранит количество способов добраться до каждой ступеньки:

vector<int> dp(n + 1, 0);

Массив dp имеет размер n + 1, что требует ( O(n) ) памяти. Таким образом, пространственная сложность алгоритма составляет:

[ O(n) ]

Временная сложность: ( O(n) )

Пространственная сложность: ( O(n) ) (изначально) и ( O(1) ) (после оптимизации)

Таким образом, алгоритм эффективно решает задачу с линейной временной сложностью и оптимизированным использованием памяти.

1. Временная сложность алгоритма определяется двумя вложенными циклами, которые заполняют матрицу dp:

for (int i = 1; i <= m; ++i) {

for (int j = 1; j <= n; ++j) {

// Логика заполнения dp[i][j]

}

}

Внешний цикл выполняется m раз (где m — длина строки s).

Внутренний цикл выполняется n раз (где n — длина строки t).

Пространственная сложность алгоритма определяется использованием не

vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

Эта матрица требует ( O(m \times n) ) памяти.

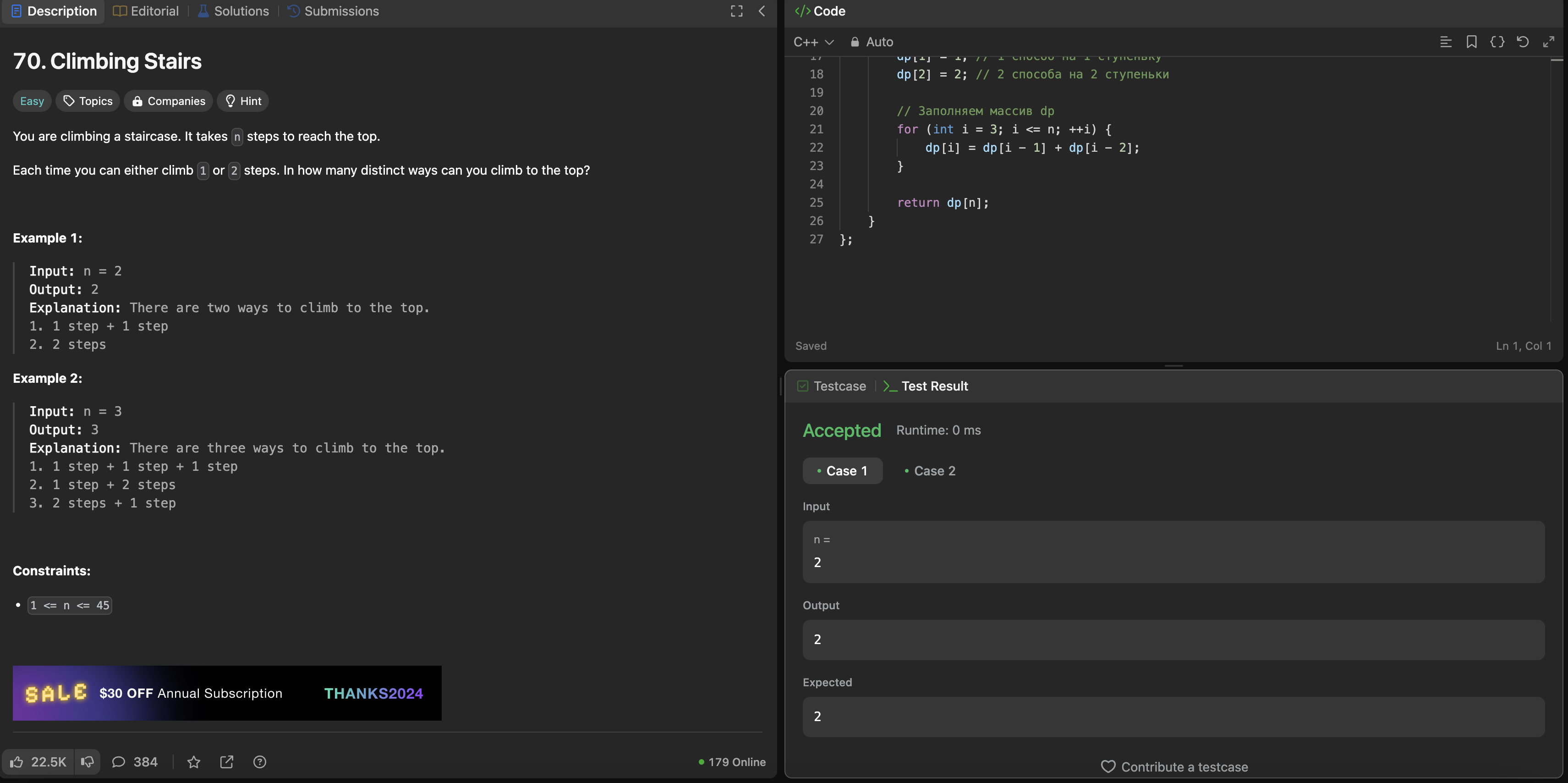
Таким образом, пространственная сложность алгоритма составляет:

[ O(m \times n) ] Временная сложность: ( O(m \times n) )

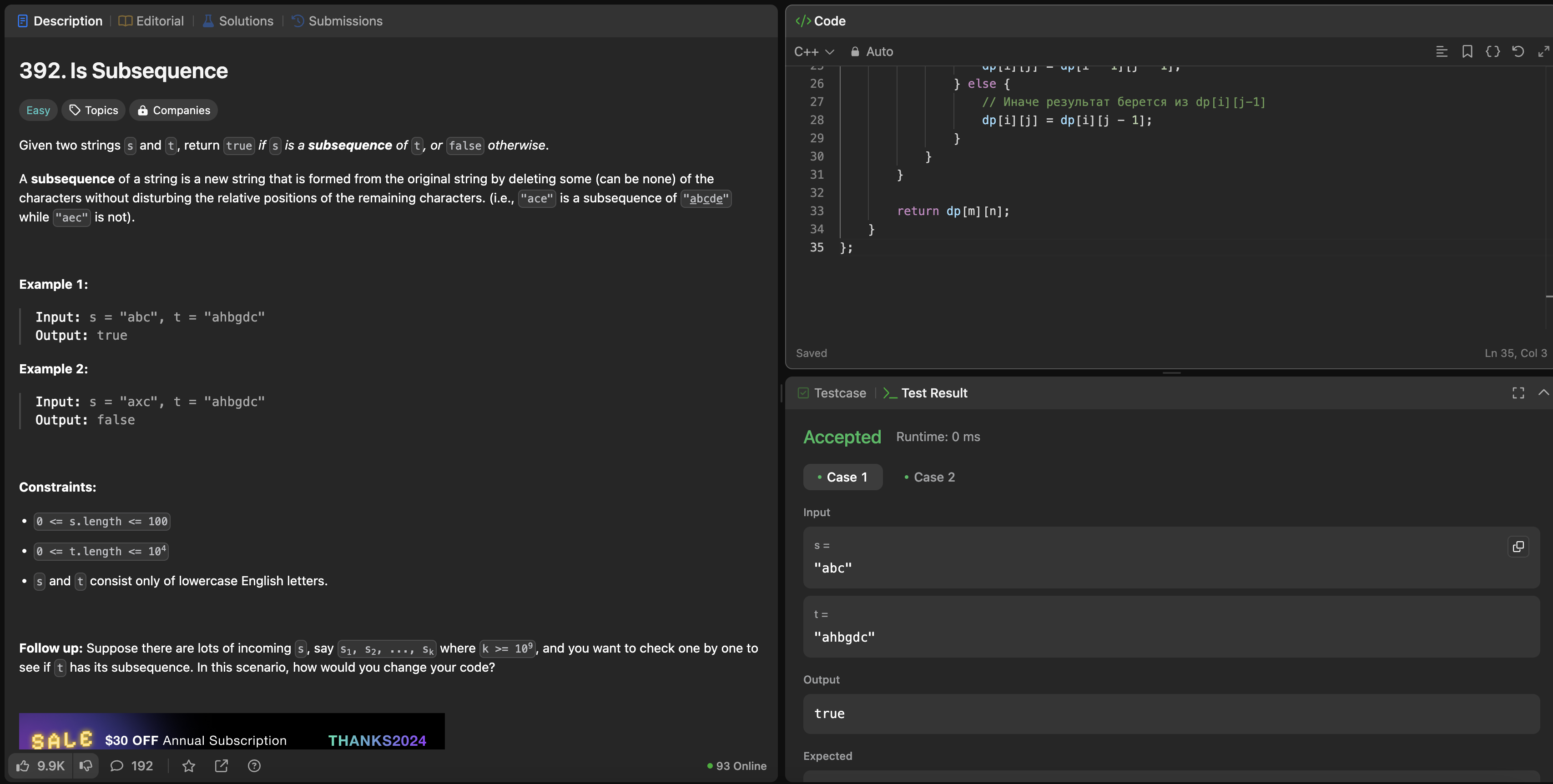
Пространственная сложность: ( O(m \times n) )

**Тесты**

1)



2)



**Использование ДП**

1. **1)** Оптимизация рекурсивного подхода:
   * Если бы мы использовали простой рекурсивный подход для решения этой задачи, мы бы столкнулись с экспоненциальной временной сложностью ( O(2^n) ). Это происходит из-за того, что для каждой ступеньки мы можем сделать два выбора (подняться на 1 или 2 ступеньки), что приводит к множеству повторяющихся вычислений.
   * Динамическое программирование позволяет избежать этих повторений, сохраняя результаты уже вычисленных подзадач.
2. Хранение промежуточных результатов:
   * В данном случае мы используем массив dp, где dp[i] хранит количество способов добраться до i-й ступеньки. Это позволяет нам использовать уже известные значения для вычисления новых, что значительно ускоряет процесс.
   * Например, чтобы вычислить количество способов добраться до i-й ступеньки, мы можем просто сложить количество способов добраться до (i-1)-й и (i-2)-й ступенек: dp[i] = dp[i - 1] + dp[i - 2].
3. Линейная временная сложность:
   * Используя динамическое программирование, мы снижаем временную сложность до ( O(n) ), что делает алгоритм эффективным даже для больших значений n.
4. Простота реализации:
   * Динамическое программирование позволяет легко реализовать решение, разбивая задачу на более простые подзадачи и комбинируя их результаты.

Итог

Использование динамического программирования в данной задаче позволяет эффективно решать проблему с линейной временной сложностью и минимизировать количество вычислений за счет хранения промежуточных результатов. Это делает алгоритм не только более быстрым, но и более понятным и удобным для реализации.

2. 1) Оптимизация вычислений:

* + Задача проверки подпоследовательности может быть решена с помощью рекурсивного подхода, но это приведет к множеству повторяющихся вычислений. Например, при рекурсивном подходе для каждой пары символов в строках s и t мы могли бы повторно вычислять, является ли оставшаяся часть s подпоследовательностью оставшейся части t.
  + Динамическое программирование позволяет сохранить результаты промежуточных вычислений, что значительно ускоряет процесс.

1. Хранение промежуточных результатов:
   * В вашем коде используется двумерный массив dp, где dp[i][j] указывает, является ли строка s[0..i-1] подпоследовательностью строки t[0..j-1]. Это позволяет избежать повторных вычислений, так как мы можем использовать уже известные значения для вычисления новых.
   * Например, если текущие символы совпадают (s[i-1] == t[j-1]), мы можем использовать значение dp[i-1][j-1], что позволяет нам избежать повторного анализа тех же символов.
2. Структурированное решение:
   * Динамическое программирование позволяет разбить задачу на более простые подзадачи. В данном случае, мы можем рассматривать каждую подстроку s и t и постепенно строить решение, основываясь на меньших подзадачах.
   * Это делает алгоритм более понятным и структурированным.
3. Линейная временная сложность:
   * Временная сложность данного алгоритма составляет ( O(m \times n) ), где ( m ) — длина строки s, а ( n ) — длина строки t. Это приемлемая сложность для большинства практических случаев, особенно по сравнению с экспоненциальной сложностью рекурсивного подхода.
4. Пространственная сложность:
   * Хотя в данном случае используется двумерный массив, который требует ( O(m \times n) ) памяти, можно оптимизировать решение, используя только одномерный массив, что снизит пространственную сложность до ( O(n) ).

Итог

Использование динамического программирования в данной задаче позволяет эффективно решать проблему проверки подпоследовательности с линейной временной сложностью и минимизировать количество вычислений за счет хранения промежуточных результатов. Это делает алгоритм не только более быстрым, но и более понятным и удобным для реализации.