Оглавление

1 Динамическое программирование **2**
- 1.1 Постановка задачи 1 . . . 2
- 1.2 Решение: ДП . . . 2
- 1.3 Постановка задачи 2 . . . 3
- 1.4 Решение: ДП . . . 3
- 1.5 Постановка задачи 3 . . . 4
- 1.6 Решение: ДП по профилю . . . 4
- 1.7 Постановка задачи 4 . . . 6
- 1.8 Решение: ДП . . . 6
- 1.9 Задачи . . . 7

2 Задача о рюкзаке **9**
- 2.1 Постановка и решение задачи . . . 9

3 Наибольшая общая подпоследовательность (НОП) **10**
- 3.1 Постановка и решение задачи . . . 10

4 Наибольшая возрастающая подпоследовательность (НВП) **11**
- 4.1 Постановка и решение задачи . . . 11

5 Наибольшая общая возрастающая подпоследовательность (НОВП) **13**
- 5.1 Постановка задачи . . . 13
- 5.2 Решение за $O(n \cdot m^2)$ . . . 13
- 5.3 Решение за $O(n \cdot m)$ . . . 14
- 5.4 Оптимизация памяти . . . 15

6 Динамическое программирование по подотрезкам **16**
- 6.1 Постановка задачи . . . 16
- 6.2 Решение: ДП по подотрезкам . . . 16
- 6.3 Задачи . . . 17

7 Динамическое программирование по маскам **18**
- 7.1 Постановка задачи . . . 18
- 7.2 Решение: ДП по маскам . . . 18
- 7.3 Задачи . . . 20

Глава 1

Динамическое программирование

1.1 Постановка задачи 1

Дано целое неотрицательное число n. Необходимо найти n-ое число Фибоначчи, где $F(0) = 0$, $F(1) = 1$, а каждое последующее число равно сумме двух предыдущих.

1.2 Решение: ДП

Если $n = 0$ или $n = 1$, выведем ответ сразу. В противном случае создадим массив длины $n + 1$, где в каждой i-ой ячейке будем хранить i-ое число Фибоначчи. Для определенности назовем этот массив dp. Очевидно, что, исходя из условия: $dp[0] = 0$ и $dp[1] = 1$. Теперь, когда базовые случаи $dp[0] = 0$ и $dp[1] = 1$ определены, мы можем последовательно вычислить все остальные числа Фибоначчи до n-го включительно: на каждом шаге будем использовать уже вычисленные значения, чтобы получать новые.

Для каждого i от 2 до n выполним:

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

Очевидно, что итоговый ответ лежит в $dp[n]$. Метод динамического программирования позволяет избежать избыточных вычислений, сохраняя уже найденные значения в массиве dp. Каждое следующее число вычисляется за константное время $O(1)$ как сумма двух предыдущих, что дает общую сложность алгоритма $O(n)$ по времени и $O(n)$ по памяти.

i	$dp[i]$	Пояснение
0	0	Базовый случай $dp[0] = 0$
1	1	Базовый случай $dp[1] = 1$
2	1	$dp[2] = dp[1] + dp[0] = 1 + 0 = 1$
3	2	$dp[3] = dp[2] + dp[1] = 1 + 1 = 2$
4	3	$dp[4] = dp[3] + dp[2] = 2 + 1 = 3$
5	5	$dp[5] = dp[4] + dp[3] = 3 + 2 = 5$
6	8	$dp[6] = dp[5] + dp[4] = 5 + 3 = 8$
7	13	$dp[7] = dp[6] + dp[5] = 8 + 5 = 13$

1.3 Постановка задачи 2

Кузнечик находится в точке с координатой 0 на числовой прямой. Он может прыгать вправо на 1, 2 или 3 шага. Требуется найти количество различных способов, которыми кузнечик может достичь точки с координатой n (n - целое неотрицательное число).

1.4 Решение: ДП

Если $n = 0$ существует ровно один способ попасть в точку 0: ничего не делать. Если $n = 1$ существует ровно один способ попасть в точку 1: прыгнуть на 1 шаг. Если $n = 2$ существует ровно два способа попасть в точку 2: прыгнуть два раза на 1 шаг / прыгнуть один раз на 2 шага.

Итак, если $n = 0$ или $n = 1$ или $n = 2$, выведем ответ сразу. В противном случае создадим массив длины $n + 1$, где в каждой i-ой ячейке будем хранить количество способов достичь точки с координатой i. Для определенности назовем этот массив dp. Базовые случаи:

- $dp[0] = 1$
- $dp[1] = 1$
- $dp[2] = 2$

Теперь, когда базовые случаи определены, мы можем последовательно вычислить все остальные состояния dp до n-го включительно: на каждом шаге будем использовать уже вычисленные значения, чтобы получать новые. Для точки с координатой i ($i \geq 3$) кузнечик мог попасть в неё одним из следующих способов:

- прыжком на 1 шаг из точки $i - 1$,
- прыжком на 2 шага из точки $i - 2$,
- прыжком на 3 шага из точки $i - 3$.

Следовательно, количество способов достичь точки i равно сумме способов достичь трёх предыдущих точек. Для каждого i от 3 до n выполним:

$$dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]$$

Очевидно, что итоговый ответ лежит в $dp[n]$. Метод динамического программирования позволяет избежать избыточных вычислений, сохраняя уже найденные значения в массиве dp. Каждое следующее число вычисляется за константное время $O(1)$ как сумма трех предыдущих, что дает общую сложность алгоритма $O(n)$ по времени и $O(n)$ по памяти.

i	$dp[i]$	Пояснение
0	1	Базовый случай $dp[0] = 1$
1	1	Базовый случай $dp[1] = 1$
2	2	Базовый случай $dp[2] = 2$
3	4	$dp[3] = dp[2] + dp[1] + dp[0] = 2 + 1 + 1 = 4$
4	7	$dp[4] = dp[3] + dp[2] + dp[1] = 4 + 2 + 1 = 7$
5	13	$dp[5] = dp[4] + dp[3] + dp[2] = 7 + 4 + 2 = 13$
6	24	$dp[6] = dp[5] + dp[4] + dp[3] = 13 + 7 + 4 = 24$
7	44	$dp[7] = dp[6] + dp[5] + dp[4] = 24 + 13 + 7 = 44$

1.5 Постановка задачи 3

Имеется последовательность из n позиций. В каждую позицию можно поставить шар одного из трёх цветов: белый (Б), серый (С) или чёрный (Ч), но два чёрных шара не могут стоять рядом. Найти количество различных последовательностей шаров длины n, удовлетворяющих данному ограничению.

1.6 Решение: ДП по профилю

Задача относится к классу задач на динамическое программирование по профилю, где "профилем"называется информация о последнем элементе последовательности, которая влияет на возможные переходы. Введем двумерный массив dp размера $(n+1)$ x 3, где:

- $dp[i][0]$ - количество последовательностей длины i, оканчивающихся на белый шар
- $dp[i][1]$ - количество последовательностей длины i, оканчивающихся на серый шар
- $dp[i][2]$ - количество последовательностей длины i, оканчивающихся на чёрный шар

База ДП

Для последовательности длины 1 (начальные условия):

- $dp[1][0] = 1$ - последовательность ["Б"]
- $dp[1][1] = 1$ - последовательность ["С"]
- $dp[1][2] = 1$ - последовательность ["Ч"]

Всего для $n = 1$: $1 + 1 + 1 = 3$ последовательности.

Переход

Для каждого $i \geq 2$ рассмотрим, какие последовательности длины i могли получиться из последовательностей длины $i - 1$:

- Белый шар можно добавить после любого цвета: $dp[i][0] = dp[i-1][0] + dp[i-1][1] + dp[i-1][2]$
- Серый шар можно добавить после любого цвета: $dp[i][1] = dp[i-1][0] + dp[i-1][1] + dp[i-1][2]$
- Чёрный шар можно добавить после белого или серого: $dp[i][2] = dp[i-1][0] + dp[i-1][1]$

Пример вычислений для $n = 3$

Покажем пошаговое заполнение таблицы:

i	$dp[i][0]$	$dp[i][1]$	$dp[i][2]$	Пояснение
1	1	1	1	Базовые случаи
2	3	3	2	$dp[2][0] = 1+1+1 = 3$, $dp[2][1] = 3$, $dp[2][2] = 1+1 = 2$
3	8	8	6	$dp[3][0] = 3+3+2 = 8$, $dp[3][1] = 8$, $dp[3][2] = 3+3 = 6$

Проверим для $n = 2$ вручную:

- Оканчивающиеся на белый: ББ, СБ, ЧБ (3 последовательности)
- Оканчивающиеся на серый: БС, СС, ЧС (3 последовательности)
- Оканчивающиеся на чёрный: БЧ, СЧ (2 последовательности)

Ответ

Общее количество последовательностей длины n: $dp[n][0] + dp[n][1] + dp[n][2]$

Реализация на C++

```cpp
int Solve(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 3;
    }

    vector<vector<int>> dp(n + 1, vector<int>(3, 0));

    dp[1][0] = 1;
    dp[1][1] = 1;
    dp[1][2] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i][0] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2];
        dp[i][1] = dp[i - 1][0] + dp[i - 1][1] + dp[i - 1][2];
        dp[i][2] = dp[i - 1][0] + dp[i - 1][1];
    }

    return dp[n][0] + dp[n][1] + dp[n][2];
}
```

- **Временная сложность**: $O(n)$
- **Пространственная сложность**: $O(n)$

Оптимизированная версия

Можно обойтись тремя переменными вместо массива, так как для вычисления $dp[i]$ нужны только значения $dp[i-1]$:

```cpp
int Solve(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 3;
    }

    int white = 1, gray = 1, black = 1;

    for (int i = 2; i <= n; i++) {
        int new_white = white + gray + black;
        int new_gray = white + gray + black;
        int new_black = white + gray;

        white = new_white;
        gray = new_gray;
        black = new_black;
    }

    return white + gray + black;
}
```

- **Временная сложность**: $O(n)$
- **Пространственная сложность**: $O(1)$

1.7 Постановка задачи 4

Дан целочисленный массив $a_1, a_2, \ldots, a_n$ и массив стоимостей изменений $c_1, c_2, \ldots, c_n$. Можно изменить любое подмножество элементов массива, где изменение элемента a_i стоит c_i и позволяет заменить его на любое целое число. Позиции, которые не изменяются, сохраняют исходные значения.

После изменений спадом называется индекс i ($1 \leq i < n$), для которого итоговое значение на позиции i строго больше итогового значения на позиции $i+1$. Требуется найти минимальную стоимость изменений, необходимую для того, чтобы в массиве не было спадов.

1.8 Решение: ДП

Заметим, что условие отсутствия спадов эквивалентно тому, что итоговый массив должен быть строго возрастающим. Некоторые элементы должны остаться неизменными, и эти элементы должны образовывать возрастающую подпоследовательность исходного массива. Мы хотим максимизировать сумму стоимостей c_i тех элементов, которые остаются неизменными, поскольку это минимизирует общую стоимость изменений - мы платим только за те элементы, которые не входят в эту подпоследовательность.

Будем решать задачу следующим образом: сначала вычислим общую стоимость T изменения всех элементов массива. Затем найдем наибольшую неубывающую подпоследовательность с максимальным весом (суммой c_i). Для этого в заведем массив dp, где dp_i представляет максимальный вес неубывающей подпоследовательности, оканчивающейся на элементе a_i. Базовый случай - подпоследовательность из одного элемента имеет вес c_i. Для каждого последующего элемента a_i мы проверяем все предыдущие элементы a_j, и если $a_j \leq a_i$, то можем расширить подпоследовательность, оканчивающуюся на a_j, добавив элемент a_i, при этом вес увеличивается на c_i. Ответом будет разность между общей стоимостью и максимальным найденным весом неубывающей подпоследовательности, что и дает минимальную стоимость изменений, необходимых для устранения спадов в массиве.

1.8.1 Пример вычислений

Рассмотрим пример: $n = 9$, $a = [5, 3, 8, 2, 6, 4, 7, 1, 9]$, $c = [2, 4, 3, 1, 5, 2, 3, 4, 2]$

$a[i]$	$c[i]$	$dp[i]$	Пояснение вычислений
5	2	2	Базовый случай: подпоследовательность из одного элемента
3	4	4	$\max(4)$ - элемент 3 меньше предыдущего 5, поэтому только базовый случай
8	3	7	$\max(3, 2+3=5, 4+3=7)$ - можно добавить к элементам 5 и 3
2	1	1	$\max(1)$ - элемент 2 меньше всех предыдущих
6	5	9	$\max(5, 2+5=7, 4+5=9, 1+5=6)$ - можно добавить к элементам 5, 3, 2
4	2	6	$\max(2, 4+2=6, 1+2=3)$ - можно добавить к элементам 3 и 2
7	3	12	$\max(3, 2+3=5, 4+3=7, 1+3=4, 9+3=12, 6+3=9)$ - можно добавить ко всем предыдущим, кроме 8
1	4	4	$\max(4)$ - элемент 1 меньше всех предыдущих
9	2	14	$\max(2, 2+2=4, 4+2=6, 7+2=9, 1+2=3, 9+2=11, 6+2=8, 12+2=14, 4+2=6)$ - можно добавить ко всем

$T = 2+4+3+1+5+2+3+4+2 = 26$, $\max dp[i] = 11$, ответ: $26 - 14 = 12$

Примечание: Основные понятия

- Динамическое программирование — это способ решения сложных задач путём разбиения их на более простые подзадачи. ДП подходит для задач, где локально оптимальные выборы не ведут к глобальному оптимуму.

  Чтобы решить задачу методом динамического программирования нужно:

  – Определить, где хранить ДП и что находится в каждом его состоянии.
  – Определить шаг ДП (переход от одного состояния к другому / пересчет текущего состояния через предыдущие).
  – Определить, где будет находиться ответ в ДП.

1.9 Задачи

1.9.1 Задача 1: Черепашка

Условие: В прямоугольной таблице A размером $n \times m$ найти путь из левого верхнего в правый нижний угол (движение только вправо или вниз строго на 1 единицу) с максимальной суммой чисел в клетках.

Решение: Используем двумерный массив dp размером $n \times m$ - изначально заполненный нулями. $dp[i][j]$ хранит максимальную сумму для достижения клетки (i, j)

В первой строке и первом столбце существует только один возможный путь (либо только вправо, либо только вверх), поэтому сумма однозначно определяется последовательным сложением. База ДП:

- $dp[0][0] = A[0][0]$
- $dp[0][j] = dp[0][j-1] + A[0][j]$ —— $(j \in [1, m])$
- $dp[i][0] = dp[i-1][0] + A[i][0]$ —— $(i \in [1, n])$

Заполнение таблицы ДП происходит по строкам сверху вниз, а по столбцам слева направо. Таким образом пересчет: $dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) + A[i][j]$ будет корректным, так как в обеих прошлых ячейках значение ДП было уже верно посчитано и меняться не будет. Итоговый результат лежит в $dp[n-1][m-1]$.

1.9.2 Задача 2: Расстояние Левенштейна

Условие: Даны две строки a и b. Найти минимальное количество операций вставки, удаления и замены символа, необходимых для превращения одной строки в другую.

Решение: Используем двумерный массив dp размером $(n+1) \times (m+1)$ - изначально заполненный нулями. $dp[i][j]$ хранит минимальное число операций для решения задачи на префиксах: $a[1:i]$ и $b[1:j]$

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & \text{если } a[i] = b[j] \\ 1 + \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) & \text{иначе} \end{cases}$$

Заполнение таблицы ДП происходит по строкам сверху вниз, а по столбцам слева направо. Таким образом пересчет будет корректным, так как в обеих прошлых ячейках значение ДП было уже верно посчитано и меняться не будет. Итоговый результат лежит в $dp[n][m]$.

1.9.3 Задача 3: Набрать сумму

Условие: Имеется неограниченное количество монет трёх номиналов: 1, 3 и 5 рублей. Требуется набрать сумму в S рублей, используя минимально возможное количество монет. Одна монета каждого номинала может быть использована неограниченное количество раз.

Решение:

- $dp[i]$ - минимальное количество монет для суммы i
- База: $dp[0] = 0$
- Рекуррентное соотношение: $dp[i] = \min(dp[i-1], dp[i-3], dp[i-5]) + 1$
- Ответ: $dp[S]$

1.9.4 Задача 4: Последовательности без двух единиц подряд

Условие: Требуется найти количество всех возможных бинарных последовательностей (состоящих из нулей и единиц) длины n, в которых нет двух соседних единиц.

Решение:

- $dp[i][j]$ - количество последовательностей длины i, оканчивающихся на j (0 или 1)
- База: $dp[1][0] = 1$, $dp[1][1] = 1$
- Рекуррентное соотношение: $dp[i][0] = dp[i-1][0] + dp[i-1][1]$, $dp[i][1] = dp[i-1][0]$
- Ответ: $dp[n][0] + dp[n][1]$

1.9.5 Задача 5: Ограбление домов

Условие: Вдоль улицы выстроены в ряд n домов. В i-м доме хранится сумма денег a_i ($a_i >= 0$). Грабитель хочет ограбить эти дома, но сигнализация сработает, если он ограбит два соседних дома подряд. Требуется определить максимальную сумму денег, которую грабитель может украсть, не вызывая сигнализацию.

Решение:

- $dp[i]$ - максимальная сумма для первых i домов
- База: $dp[0] = 0$, $dp[1] = a[1]$
- Рекуррентное соотношение: $dp[i] = \max(dp[i-1], dp[i-2] + a[i])$
- Ответ: $dp[n]$

1.9.6 Задача 6: Разрезание стержня

Условие: Дан стальной стержень длиной n единиц. Известна таблица цен $price[0...n-1]$, где $price_i$ — это цена за стержень длиной i единиц. Разрешается разрезать стержень на куски целой длины (или не разрезать вовсе) с целью максимизировать общую выручку от продажи всех полученных кусков. Найти наибольшую выручку.

Решение:

- $dp[i]$ - максимальная прибыль для стержня длины i
- База: $dp[0] = 0$
- Рекуррентное соотношение: $dp[i] = \max\limits_{1 \le j \le i}(price[j] + dp[i-j])$
- Ответ: $dp[n]$

Глава 2

Задача о рюкзаке

2.1 Постановка и решение задачи

Дано n предметов, каждый с весом w_i и стоимостью $cost_i$. Максимальная вместимость рюкзака - W. Необходимо выбрать подмножество предметов с максимальной суммарной стоимостью, не превышающей W; каждый предмет можно использовать не более одного раза.

Идея решения

Используем двумерный массив dp (изначально заполненный нулями), где $dp[i][j]$ = максимальная стоимость для первых j предметов и рюкзака вместимости i. На каждом шаге предмет можно либо:

- не взять и пересчитаться из соседней левой ячейке, т.е. просто посмотреть ответ на префиксе длиной на 1 меньше, не изменяя при этом вместимость рюкзака
- взять и пересчитаться из ячейки, которая находится на 1 левее и на w_j выше - ведь на префиксе длины $j-1$ вместимость рюкзака должна была быть хотя бы $i - w_j$

$$dp[i][j] = \begin{cases} dp[i][j-1], & \text{если } w_i > j \\ \max(dp[i][j-1], cost_j + dp[i-w_j][j-1]), & \text{иначе} \end{cases}$$

Заполнение таблицы ДП происходит по строкам сверху вниз, а по столбцам слева направо. Таким образом пересчет будет корректным, так как в обеих прошлых ячейках значение ДП было уже верно посчитано и меняться не будет. Ответ, очевидно, будет находиться в ячейке $dp[W][n]$ (рассматриваем максимальную вместимость рюкзака и набор из всех предметов)

Реализация на C++

```
int Solve(int W, int n, vector<int>& weights, vector<int>& costs) {
    vector<vector<int>> DP(W + 1, vector<int>(n + 1));
    for (int i = 1; i <= W; i++) {
        for (int j = 1; j <= n; j++) {
            DP[i][j] = DP[i][j - 1];
            if (i - weights[j] >= 0) {
                DP[i][j] = max(DP[i][j], DP[i - weights[j]][j - 1] + costs[j]);
            }
        }
    }
    return DP[W][n];
}
```

- **Временная сложность**: $O(n \cdot W)$
- **Пространственная сложность**: $O(n \cdot W)$

Глава 3

Наибольшая общая подпоследовательность (НОП)

3.1 Постановка и решение задачи

Даны две последовательности $A[1..n]$ и $B[1..m]$. Необходимо найти их наибольшую общую подпоследовательность (подпоследовательность - это последовательность, которую можно получить из исходной удалением некоторых элементов без изменения порядка оставшихся элементов).

Идея решения

Используем двумерный массив dp (изначально заполненный нулями), где: $dp[i][j]$ = длина НОП для префиксов $A[1..i]$ и $B[1..j]$. На каждом шаге последние два элемента

либо совпали (и тогда выгодно посмотреть на ответ в ячейке dp[i - 1][j - 1] - фактически отбросив эти два равных числа, предварительно взяв их в ответ).

либо не совпали (и тогда выгодно отбросить последнее число в одном из двух массивов, то есть пересчитаться из ячеек $dp[i-1][j]$ и $dp[i][j-1]$, ведь во всех остальных НОП гарантированно $\leq max$(dp[i - 1][j], dp[i][j - 1])).

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & \text{если } A[i] = B[j] \\ \max(dp[i-1][j], dp[i][j-1]), & \text{иначе} \end{cases}$$

Заполнение таблицы ДП происходит по строкам сверху вниз, а по столбцам слева направо. Таким образом пересчет будет корректным, так как в обеих прошлых ячейках значение ДП было уже верно посчитано и меняться не будет. Ответ, очевидно, будет находиться в ячейке $dp[n][m]$ (рассматриваем ответ для полных массивов A и B)

Реализация на C++

```
int Solve(vector<int>& a, int n, vector<int>& b, int m) {
    vector<vector<int>> dp(n + 1, vector<int>(m + 1));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i-1] == b[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

- **Временная сложность**: $O(n \cdot m)$
- **Пространственная сложность**: $O(n \cdot m)$

Глава 4

Наибольшая возрастающая подпоследовательность (НВП)

4.1 Постановка и решение задачи

Дана последовательность $A[1..n]$. Необходимо найти её наибольшую возрастающую подпоследовательность (подпоследовательность, элементы которой строго возрастают).

Идея решения

Используем динамическое программирование с бинарным поиском:

- Поддерживаем массив d (изначально пустой), где $d[i]$ — минимальный возможный последний элемент возрастающей подпоследовательности длины $i+1$.
- Заметим два важных свойства этой динамики:
  - $d[i-1] < d[i]$ для всех $i = 1, \ldots, n$ (массив d отсортирован по возрастанию). Это следует из того, что для подпоследовательности длины $i+1$ последний элемент должен быть строго больше, чем последний элемент подпоследовательности длины i.
  - Каждый элемент $a[i]$ обновляет максимум один элемент $d[j]$. А именно, он может заменить первый элемент $d[j]$, который больше $a[i]$, тем самым улучшив $d[j]$ (сделав его меньше).
- Это означает, что при обработке очередного $a[i]$, мы можем за $O(\log n)$ с помощью двоичного поиска в массиве d найти первое число, которое больше текущего $a[i]$, и обновить его. Ответом будет получившийся массив d.

Пример работы алгоритма НВП

Рассмотрим последовательность $a = [3, 1, 4, 1, 5, 9, 2, 6]$. Алгоритм поддерживает массив d, где $d[i]$ — минимальный возможный последний элемент возрастающей подпоследовательности длины $i+1$.

Шаг	$a[i]$	d до обработки	Действие	d после обработки
0	3	[]	Добавить 3	[3]
1	1	[3]	Заменить $d[0] = 1$	[1]
2	4	[1]	Добавить 4	[1, 4]
3	1	[1, 4]	Заменить $d[0] = 1$	[1, 4]
4	5	[1, 4]	Добавить 5	[1, 4, 5]
5	9	[1, 4, 5]	Добавить 9	[1, 4, 5, 9]
6	2	[1, 4, 5, 9]	Заменить $d[1] = 2$	[1, 2, 5, 9]
7	6	[1, 2, 5, 9]	Заменить $d[3] = 6$	[1, 2, 5, 6]

Таблица 4.1: Пошаговая работа алгоритма НВП

Реализация на C++

```cpp
int bs(vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] < target) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return left;
}

vector<int> Solve(vector<int>& a, int n) {
    vector<int> d;
    for (int i = 1; i <= n; ++i) {
        int p = bs(d, a[i]);
        if (p == l.size()) {
            d.push_back(a[i]);
        }
        else {
            d[p] = a[i];
        }
    }
    return d;
}
```

- **Временная сложность**: $O(n \log n)$
- **Пространственная сложность**: $O(n)$

Глава 5

Наибольшая общая возрастающая подпоследовательность (НОВП)

5.1 Постановка задачи

Даны две последовательности $A[1..n]$ и $B[1..m]$. Необходимо найти их наибольшую общую возрастающую подпоследовательность - то есть последовательность C, которая:

- Является подпоследовательностью A (существует строго возрастающая последовательность индексов $i_1 < i_2 < \cdots < i_k$ такая, что $C[p] = A[i_p]$)
- Является подпоследовательностью B (существует строго возрастающая последовательность индексов $j_1 < j_2 < \cdots < j_k$ такая, что $C[p] = B[j_p]$)
- Строго возрастает ($C[1] < C[2] < \cdots < C[k]$)

5.2 Решение за $O(n \cdot m^2)$

Идея решения

Используем динамическое программирование, аналогичное задаче о наибольшей общей подпоследовательности (НОП), но с дополнительным условием возрастания.

Пусть $dp[i][j]$ - длина наибольшей общей возрастающей подпоследовательности для префиксов $A[1..i]$ и $B[1..j]$.

Рекуррентные соотношения

$$dp[i][j] = \begin{cases} \max\limits_{\substack{1 \le k < i \\ 1 \le l < j \\ A[k]=B[l]<A[i]}} (dp[k][l] + 1), & \text{если } A[i] = B[j] \\ \max(dp[i-1][j], dp[i][j-1]), & \text{иначе} \end{cases}$$

- Если $A[i] = B[j]$, то мы можем добавить этот элемент к НОВП, но только если предыдущий элемент был **меньше** $A[i]$ и присутствовал в обеих последовательностях ранее. Мы перебираем все возможные предыдущие пары (k, l) с $A[k] = B[l] < A[i]$
- Если $A[i] \neq B[j]$, то наследуем максимальное значение из предыдущих префиксов

Реализация на C++

```cpp
int SolveSlow(vector<int>& A, int n, vector<int>& B, int m) {
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (A[i-1] == B[j-1]) {
                int best = 0;
                for (int k = 1; k < i; k++) {
                    for (int l = 1; l < j; l++) {
                        if (A[k-1] == B[l-1] && A[k-1] < A[i-1]) {
                            best = max(best, dp[k][l]);
                        }
                    }
                }
                dp[i][j] = best + 1;
            }
            else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[n][m];
}
```

5.3 Решение за $O(n \cdot m)$

Идея решения

Заметим, что внутренний двойной цикл можно убрать. Для каждого фиксированного i мы можем поддерживать массив $best[x]$ - максимальная длина НОВП, заканчивающаяся на элемент со значением x.

Но поскольку нам нужны только элементы **меньше** текущего $A[i]$, мы можем поддерживать одну переменную $best$, которая хранит максимальную длину для всех элементов $B[j]$, которые **меньше** текущего $A[i]$.

Реализация на C++

```cpp
int Solve(vector<int>& A, int n, vector<int>& B, int m) {
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++) {
        int best = 0;
        for (int j = 1; j <= m; j++) {
            dp[i][j] = dp[i-1][j];
            if (A[i-1] == B[j-1]) dp[i][j] = max(dp[i][j], best + 1);
            if (B[j-1] < A[i-1]) best = max(best, dp[i-1][j]);
        }
    }
    int ans = 0;
    for (int j = 1; j <= m; j++) {
        ans = max(ans, dp[n][j]);
    }
    return ans;
}
```

- **Временная сложность**: $O(n \cdot m)$
- **Пространственная сложность**: $O(n \cdot m)$

- Для каждого i (элемент из A) мы проходим по всем j (элементы из B). Переменная *best* хранит максимальную длину НОВП, которая может быть продолжена текущим элементом $A[i]$
- Когда встречаем $B[j] < A[i]$, обновляем *best* значением $dp[i-1][j]$
- Когда встречаем $A[i] = B[j]$, используем $best + 1$ как возможное новое значение

Пример работы алгоритма НОВП

Рассмотрим последовательности:

- $A = [1, 3, 2, 4]$
- $B = [3, 1, 2, 4]$

Наибольшая общая возрастающая подпоследовательность: $[1, 2, 4]$ (длина 3)

$dp[i][j]$	$\emptyset$	3	1	2	4
$\emptyset$	0	0	0	0	0
1	0	0	**1**	1	1
3	0	**1**	1	1	1
2	0	1	1	**2**	2
4	0	1	1	2	**3**

Таблица 5.1: Таблица DP для НОВП

5.4 Оптимизация памяти

```cpp
int SolveOptimized(vector<int>& A, int n, vector<int>& B, int m) {
    vector<int> dp_prev(m + 1, 0);
    vector<int> dp_curr(m + 1, 0);
    for (int i = 1; i <= n; i++) {
        int best = 0;
        for (int j = 1; j <= m; j++) {
            dp_curr[j] = dp_prev[j];
            if (A[i-1] == B[j-1]) {
                dp_curr[j] = max(dp_curr[j], best + 1);
            }
            if (B[j-1] < A[i-1]) {
                best = max(best, dp_prev[j]);
            }
        }
        swap(dp_prev, dp_curr);
    }
    int ans = 0;
    for (int j = 1; j <= m; j++) {
        ans = max(ans, dp_prev[j]);
    }
    return ans;
}
```

- **Временная сложность**: $O(n \cdot m)$
- **Пространственная сложность**: $O(m)$

Глава 6

Динамическое программирование по подотрезкам

6.1 Постановка задачи

Вам дается n шариков с индексом от 0 до $n-1$. На каждом шарике нанесен номер, представленный массивом a. Вам предлагается лопнуть все шарики. Если вы лопнете i-ый шарик, то получите монеты $a[i-1] \cdot a[i] \cdot a[i+1]$. Если $i-1$ или $i+1$ выходит за пределы массива, то рассматривайте это так, как если бы на нем был нарисован воздушный шар с цифрой 1. Нужно получить как можно больше монет.

6.2 Решение: ДП по подотрезкам

Вместо того чтобы думать о том, какой шарик лопнуть первым, мы будем думать о том, какой шарик лопнуть последним на каждом подотрезке. Это позволяет использовать динамическое программирование.

Пусть $dp[i][j]$ - максимальное количество монет, которое можно получить на подотрезке $[i : j]$. Для подотрезков длины 1 (отдельные шарики) значение равно произведению значения шарика на его соседей (с учетом границ массива).

Далее будем последовательно рассматривать подотрезки длины от 2 до n. Для каждого подотрезка $[i, j]$ перебираем все возможные варианты последнего лопнутого шарика k от i до j и обновляем ответ для текущего подотрезка. Вычисляем монеты для k как сумму: монет от левой части dp[i][k-1] (если $k > i$), монет от правой части dp[k+1][j] (если $k < j$), монет за лопание шарика k (с учетом, что на подотрезке $[i : j]$ шариков не осталось). Тогда ответ хранится в $dp[0][n-1]$ - ячейке, отвечающей за весь массив.

Реализация на C++ (массив расширен с помощью двух единиц по краям)

```
int Solve(vector<int>& a) {
    vector<int> b(a.size() + 2, 1);
    for (int i = 0; i < a.size(); i++) b[i + 1] = a[i];
    vector<vector<int>> dp(a.size() + 2, vector<int>(a.size() + 2, 0));
    for (int len = 1; len <= a.size(); len++) {
        for (int i = 1; i <= a.size() - len + 1; i++) {
            int j = i + len - 1;
            for (int k = i; k <= j; k++) {
dp[i][j] = max(dp[i][j], b[i - 1] * b[k] * b[j + 1] + dp[i][k - 1] + dp[k + 1][j]);
            }
        }
    } return dp[1][a.size()];
}
```

- **Временная сложность**: $O(n^3)$
- **Пространственная сложность**: $O(n^2)$

Примечание: Основные понятия

Динамическое программирование по подотрезкам применяется для задач, где нужно найти оптимальное решение для непрерывного участка (подотрезка) последовательности. Характерные черты:

- Подзадачи определяются для всех подотрезков исходной последовательности.
- Переходы между состояниями обычно осуществляются через:
  - Укорочение подотрезка с одного или обоих концов.
  - Разбиение подотрезка на две части.

Чтобы решить задачу методом динамического программирования по подотрезкам нужно:

1. Определить состояние DP: обычно dp[i][j] - решение для подотрезка от i до j, а затем инициализировать базовые случаи (подотрезки длины 1 или 2).
2. Заполнить таблицу **диагонально**, перебирая длины подотрезков от меньших к большим, определяя правило перехода.

6.3 Задачи

6.3.1 Задача 1: Наибольший подотрезок-палиндром нечетной длины

Условие: Дана строка s длины n. Найти длину наибольшего подотрезка, который является палиндромом нечетной длины.

Решение: Используем динамическое программирование:

- Состояние: $dp[i][j]$ - является ли подстрока $s[i..j]$ палиндромом.
- База: $dp[i][i] = \text{true}$ (подстрока длины 1).
- Переход:

$$dp[i][j] = \begin{cases} \text{true}, & \text{если } s[i] = s[j] \text{ и } dp[i+1][j-1] = true \text{ и (j-i+1) mod 2} = 1 \\ \text{false}, & \text{иначе} \end{cases}$$

- Ответ: максимальное $(j - i + 1)$, где $dp[i][j] = \text{true}$.

6.3.2 Задача 2: Максимальная правильная скобочная подпоследовательность

Условие: Дана строка s, состоящая из символов '(', ')', '[', ']', '{', '}'. Найти длину максимальной правильной скобочной подпоследовательности.

Решение: Используем динамическое программирование:

- Состояние: $dp[i][j]$ - длина наибольшей правильной подпоследовательности на подстроке $s[i..j]$.
- База: $dp[i][j] = 0$ для всех $i > j$.
- Переходы:
  1. Если скобки $s[i]$ и $s[j]$ образуют пару:
  $$dp[i][j] = dp[i+1][j-1] + 2$$
  2. Для всех k от i до $j - 1$:
  $$dp[i][j] = \max(dp[i][j], dp[i][k] + dp[k+1][j])$$
- Ответ: $dp[0][n-1]$.

Глава 7

Динамическое программирование по маскам

7.1 Постановка задачи

Дан граф из n городов с матрицей расстояний d_{ij}. Необходимо найти гамильтонов путь минимальной длины, проходящий через каждый город ровно один раз.

7.2 Решение: ДП по маскам

Любое подмножество множества из n элементов можно представить n-битным числом (битовой маской):

$$\text{mask} = \sum_{i=0}^{n-1} b_i \cdot 2^i, \quad \text{где } b_i = \begin{cases} 1, & \text{если } i\text{-й элемент принадлежит подмножеству,} \\ 0, & \text{иначе.} \end{cases}$$

Основные операции с битовыми масками

Операция	C++	Python
Добавить элемент	mask |= (1 « i)	mask | (1 « i)
Удалить элемент	mask &= ~(1 « i)	mask & ~(1 « i)
Проверить элемент	mask & (1 « i)	mask & (1 « i)
Переключить элемент	mask ^= (1 « i)	mask ^(1 « i)

Итак, пусть: $dp[mask][v]$ - минимальная длина пути, проходящего через все города в маске $mask$ и заканчивающегося в городе v. Тогда база ДП: $dp[1 << i][i] = 0$ для всех i - так как если маска содержит только один город i, минимальное расстояние для посещения такого набора городов равно 0. Далее для каждой маски и города v перебираем не посещенный город u:

$$dp[mask|(1 << u)][u] = \min(dp[mask|(1 << u)][u], dp[mask][v] + d_{vu})$$

.

Ответ: $\min\limits_{v}(dp[2^n - 1][v])$. $2^n - 1$ - маска, в которой все n городов посещены (все биты равны 1).

Реализация на C++

```cpp
int Solve(const vector<vector<int>>& d) {
    int n = d.size();
    vector<vector<int>> dp(1 << n, vector<int>(n, 1e9));

    for (int i = 0; i < n; i++) {
        dp[1 << i][i] = 0;
    }

    for (int mask = 0; mask < (1 << n); mask++) {
        for (int v = 0; v < n; v++) {
            if (!(mask & (1 << v))) {
                continue;
            }
            for (int u = 0; u < n; u++) {
                if (mask & (1 << u))) {
                    continue;
                }
                int new_mask = mask | (1 << u);
                dp[new_mask][u] = min(dp[new_mask][u], dp[mask][v] + d[v][u]);
            }
        }
    }

    int final_mask = (1 << n) - 1;

    int res = 1e9;
    for (int v = 0; v < n; v++) {
        res = min(res, dp[final_mask][v]);
    }

    return res;
}
```

- **Временная сложность**: $O(2^n \cdot n^2)$ (перебор всех масок и пар городов)
- **Пространственная сложность**: $O(2^n \cdot n)$ (хранение таблицы DP)

Примечание: Основные понятия

Динамическое программирование по маскам - это метод решения задач, где состояние описывается битовой маской. Фактически одно из состояний ДП хранит информацию целиком об одном из его измерений. Особенно эффективен для задач на подмножествах.

Перебирайте маски в порядке возрастания количества установленных битов:

```cpp
for (int k = 0; k <= n; ++k) {
    for (int mask = 0; mask < (1 << n); mask++) {
        ...
    }
}
```

Полезные трюки

- Быстрое вычисление количества установленных битов:

```
int cnt = __builtin_popcount(mask);
```

- Нахождение первой установленной позиции:

```
int first = __builtin_ctz(mask);
```

- Итерация по всем подмаскам:

```
for (int submask = mask; submask; submask = (submask-1) & mask) {
    ...
}
```

7.3 Задачи

7.3.1 Задача 1: Покрытие множества

Условие: Дано множество S и набор его подмножеств $A_1, ..., A_m$. Найти минимальное число подмножеств, покрывающих S.

Решение: Используем динамическое программирование по маскам:

- Состояние: $dp[mask]$ - минимальное число подмножеств для покрытия $mask$.
- База: $dp[0] = 0$.
- Переход: Переход: Для каждого подмножества A_i и маски mask:

$$dp[mask|A_i] = \min(dp[mask|A_i], dp[mask] + 1)$$

- Ответ: $dp[2^n - 1]$.

7.3.2 Задача 2: Раскраска графа в минимальное число цветов

Условие: Дан неориентированный граф из n вершин. Найти минимальное число цветов, необходимых для раскраски графа так, чтобы любые две смежные вершины имели разные цвета.

Решение: Используем динамическое программирование по маскам:

- **Состояние**: $dp[mask]$ - минимальное число цветов, необходимых для раскраски подмножества вершин $mask$.
- **База**: $dp[0] = 0$.
- **Переход**:
  1. Найдем все независимые множества в графе (подмножества вершин, где никакие две вершины не соединены ребром)
  2. Для каждой маски $mask$ перебираем все независимые подмножества $S \subseteq mask$:

$$dp[mask] = \min_{S \subseteq mask,\ S \text{ - независимое}} (dp[mask \setminus S] + 1)$$

- **Ответ**: $dp[2^n - 1]$.