Лекция DP [9]

В этой лекции мы продолжим говорить про динамическое программирование, рассмотрим три классические задачи. Одна из них часто встречается, другая просто классическая задача, которую полезно уметь решать, а третья даёт общую идею решения некоторого класса задач.

Задача о рюкзаке 0-1

Формулировка. (Журавлёв)

Залезли вы в магазин электроники ночью, и там стоят всякие телевизеры, системные блоки, мониторы, мышки и так далее. У каждого объекта есть какой-то вес w_i, есть его ценность c_i. Вы можете унести за раз объекты с суммарным весом не более W. Ваша задача — набрать объектов, вес которых суммарно не превышают W на максимальную суммарную стоимость. При этом объект вы распилить не можете, потому что половина телевизера показывать ничего не будет. Собственно, такая задача.

Наивное решение.

Мы перебираем все подмножества объектов, смотрим, можем ли мы это подмножество взять, и если мы его можем взять, то смотрим, наиболее ли это дорогое подмножество, или нет. Если это наиболее дорогое подмножество, то это будет нашим новым ответом.

Какова сложность такого наивного решения? Вам нужно перебрать все подмножества данного множества. Всех подмножеств 2^n (1, если берём объект, 0, если нет), следовательно и сложность $O(2^n)$. С такой сложностью мы сможем решить задачу только для n порядка 20. Неэффективно.

```
1 void brute_force(int i, int sum, int weight) {
2     if (i == n || weight > W) {
3         return; // граничные условия
4     }
```

```
    // не берём предмет
    brute_force(i + 1, sum, weight);
    // берём предмет
    brute_force(i + 1, sum + c[i], weight + w[i]);

    ans = max(ans, sum);
}
```

Решение через динамику.

Заведём двумерный массив для динамического программирования $dp[i][s]$, где i — это индекс рассматриваемого объекта, s — вместимость рюкзака. А само значение, которое лежит в $dp[i][s]$ хранится максимальная суммарная стоимость предметов, набранных в рюкзак вместимости s из первых i объектов.

Допустим, мы рассматриваем объект с номером i. Всего есть два варианта — **брать** его (тогда стоимость рюкзака увеличится, а вместимость уменьшится), либо **не брать** в рюкзак (тогда стоимость останется такой же, какой была для набранных уже предметов). Будем выбирать максимальный по стоимости исход.

Пусть веса предметов хранятся в векторе w, а цены в веторе c.

Формула пересчёта динамики:

$$dp[i][s] = \max(dp[i-1][s-w[i]] + c[i],\ dp[i-1][s])$$

Проходимся по всем строкам, начиная с первой, вектора векторов в dp, и применяем формулу пересчёта.

Сделаем так, что индексация элементов в векторах w и c будет начинаться с 1. Тогда ответ будет лежать в $dp[n][W]$ — когда мы попробовали взять все n предметов в рюкзак вместимости W.

Нужно не забыть обработать момент, чтобы индексы нигде не оказались отрицательными.

Начальные условия нашей динамики — будем считать, что в первой строке матрицы dp будут стоять все нули. Потому что предмета с индексом 0 у нас нет, следовательно и стоимость рюкзака без предметов нулевая.

Сложность. Так как мы проходимся по всей матрице dp, временная сложность будет $O(nW)$. Также, если мы будем хранить всю матрицу из $n \times W$ элементов, памяти потребуется также nW. Однако заметим, что для пересчёта текущей строки, нам нужна только предыдущая строка, а так как в одной строке W элементов, памяти потребуется $2W$.

Замечание. Чтобы после того, как мы рассчитаем текущую строку, быстро присвоить её значение предыдущей, можно воспользоваться функцией $swap(prev, cur)$. Она поменяет вектора местами за $O(1)$.

```
...
vector<vector<int>> dp(n + 1, vector<int>(W + 1));
for (int i = 1; i <= n; i++) {
    for (int s = w[i - 1]; s <= W; s++) {
        dp[i][s] = max(dp[i - 1][s - w[i]] + c[i], dp[i -
        ↪  1][s]);
    }
    for (int s = 0; s < w[i - 1]; s++) {
        dp[i][s] = dp[i - 1][s];
    }
}
cout << dp[n][W];
```

Если хотите лучше понять, можете почитать на итмошной вики: https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_рюкзаке. Там же есть пример построенной таблицы.

Наибольшая общая подпоследовательность

Задача о наибольшей общей подпоследовательности (LCS — Longest Common Subsequence) обычно формулируется для строк, но может быть применена для любых двух последовательностей элементов.

Подпоследовательность последовательности – это последовательность *subseq*, полученная из последовательности *seq*, удалением каких-то из элементов *seq*.

Формулировка.

Вам даны две строки s длины n и t длины m, ваша задача — найти их наибольшую общую подпоследовательность.

Наивное решение.

Переберём все возможные подпоследовательности строки s, проверим, каким из них являются подпоследовательностями t, запомним самую длинную из них. Сложность такого решения будет равна $O(2^n \cdot m)$. Не годится.

Решение через динамику.

Будем считать динамику $lcs(s_n, t_n)$, которая по смыслу значит: наибольшая общая подпоследовательность для строки s длиной n и строки t длиной m. Она будет естественно равна максимуму из возможных вариантов пересчёта.

1. Предположим, последние два символа совпали. Тогда возможно они принадлежат наибольшей общей подпоследовательности двух строк. $lcs(s_n, t_n) = lcs(s_{n-1}, t_{n-1}) + 1$. То есть, мы говорим, что совпадающий символ включаем в наибольшую подпоследовательность, и посчитай нам теперь её для строк на один символ короче.

2. Если последние два символа не совпали, нам может быть выгодно пропустить букву в одной из строк.

2.1. $lcs(s_n, t_n) = lcs(s_{n-1}, t_n)$ — пропускаем букву в строке s.
2.2. $lcs(s_n, t_n) = lcs(s_n, t_{n-1})$ — пропускаем букву в строке t.

Взятие максимума из этих трёх возможностей даст нам формулу пересчёта.

$$dp[i][j] = \max(dp[i-1][j-1] + (s[i] == t[i]),\ dp[i-1][j],\ dp[i][j-1])$$

Начальные условия — для всех i, j в $dp[i][0]$ и $dp[0][j]$ будут лежать нули, так как наибольшая общая подпоследовательность пустой строки и чего-либо ещё, очевидно равна нулю.

Тогда ответ будет лежать в $dp[n][m]$.

Также мы сможем запомнинать в какой-нибудь массив path[n][m] пару индексов (i, j) элемента, в который нужно перейти из текущего для получения следующего элемента наибольшей общей подпоследовательности. Таким образом, мы сможем вывести саму искомую подпоследовательность.

Сложность. Так как динамика проходится по всем $i \in [0, n)$ и для каждого i по всем $j \in [0, m)$, сложность будет равна $O(nm)$. Памяти требуется тоже nm, но если нам не потребуется восстановить саму подпоследовательность, можно хранить только две последние строки в динамике.

Ссылка на итмошную вики про LCS:
https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_наибольшей_общей_подпоследовательно

Динамическое программирование по подотрезкам

Динамическое программирование по подотрезкам — общая идея для некоторых задач, которую нужно знать. Давайте рассмотрим её на примере классической задачи об **оптимальном перемножении матриц**.

Формулировка.

Вам дано несколько матриц, нужно определить количество действий, которые потребуется совершить, для того чтобы перемножить их все.

Ассоциативность произведения матриц.

В курсе линейного анализа вам рассказывали, что для легального произведения матриц, при любой расстановке скобок в нём, ответ будет одинаковым.

$$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$$

Сложность умножения.

Из линейного анализа также известно, что чтобы умножить матрицу $A_{n \times k}$ на матрицу B, первая размерность матрицы B должна совпадать с последней размерностью матрицы A, то есть $B = B_{k \times m}$.

Суть умножения матриц — каждую строку $A_{n \times k}$ мы умножаем на каждый столбец $B_{k \times m}$, а каждое такое действие $(\sum_{t=1}^{k} a_{i,t} \cdot b_{t,j})$ занимает у нас k операций. Всего действий $n \cdot m$. Значит, количество действий для перемножения двух матриц равно $n \cdot m \cdot k$. Притом в результате получится матрица размерности $n \times m$.

Решение.

Убедитесь сначала, что при различной расстановке скобок в последовательности перемножаемых матриц будут получаться разные ответы. Например, на матрицах $A_{10 \times 1000} \cdot B_{1000 \times 10} \cdot C_{10 \times 1000}$.

Пусть нам дано n матриц с количествами строк и рядов: вектора $rows$, $cols$.

Давайте посмотрим, в чём заключается идея динамики по подотрезкам. Рассмотрим последний шаг перемножения цепочки матриц.

$$(A_1 \cdot A_2 \cdot \ldots \cdot A_i) \cdot (A_{i+1} \cdot \ldots \cdot A_n)$$

Так как это последний шаг, произведения в скобках, предполагается, уже посчитаны, осталось только перемножить их между собой.

Притом i может быть произвольным. Поэтому переберём все возможные варианты i: от 1 до n. Из всех возможных вариантов выберем тот, при котором количество операций будет минимальным.

Заметим, что если у нас ещё не посчитано количество операций для какого-то из отрезков, мы можем перейти на его уровень и проделать с ним то же самое.

Будем считать значения в $dp[l][r]$, смысл которой: минимальное количество операций, которые нужно совершить для перемножения матриц с l по r.

То есть формула пересчёта динамики будет следующей (l и r — границы отрезка, который мы сейчас рассматриваем, k — позиция разбиения цепочки матриц на две части):

$$dp[l][r] = \min_{k \in [l,r)} (dp[l][k] + dp[k+1][r] + rows[l] \cdot cols[r] \cdot cols[k])$$

Начальные условия — $dp[i][i] = 0$, чтобы получить произведения из одной матрицы, ничего делать не нужно.

Обход динамики. Если раньше было понятно, как это делать, мы просто шли по строкам, то здесь порядок обхода не так понятен. Поэтому давайте воспользуемся техникой, которая называется мемоизация.

С помощью мемоизации мы просто перенесём формулу в код, не задумываясь, в каком порядке нужно считать нашу динамику.

Давайте заведём функцию dp, которая будет выводить ответ для отрезка $[l, r]$. Передадим в неё также матрицу $memo$ — вектор векторов, в котором будут храниться ответы для отрезков, которые мы уже посчитали, чтобы не пересчитывать их заново, а просто брать из массива.

Чтобы определять, лежит ли уже в массиве $memo$ ответ для данного отрезка, заполним сначала его элементами, которые точно не могут быть ответами для нашей динамики. Например, -1.

```
int dp(int l, int r, vector<vector<int>>& memo) {
    if (memo[l][r] != -1)  // значение уже посчитано
        return memo[l][r];
    if (l == r)  // одна матрица
        memo[l][l] = 0;

    int res = dp(l, l, memo) + dp(l + 1, r, memo) +
        rows[l] * cols[r] * cols[l];
    for (int k = l + 1; k < r; k++) {
        res = min(res, dp(l, k, memo) + dp(k + 1, r, memo) +
                  rows[l] * cols[r] * cols[k]);
    }
    memo[l][r] = res; // добавляем ответ в массив
    return res;
}
```

Сложность. Так как размерность массива $memo$ равна $n \times n$, и перебор происходит по всем отрезкам $[l, r], l \leq r$, то есть по верхнему треугольнику матрицы, (отрезок — координата (l, r) элемента массива), и для каждого отрезка в общем случае есть цикл по n элементам, то сложность алгоритма равна $O(n^3)$.

Итог

Мы рассмотрели три классические задачи динамического программирования. Ваша задача — понять их и научиться реализовывать. Пишите код сами, и тогда дальше будет становиться проще. Конечно же, дорешивайте, иначе не получите никакого прогресса.