Разбор некоторых задач контеста «Задачи-2» из темы «3. Динамическое программирование»

Егор Подлесов

02.02.2022

Задача A. Минимум предметов

Как набрать вес в точности M, используя как можно меньше предметов из множества $\{m_1, \ldots, m_N\}$?

► Решать будем при помощи двумерной динамики.
Пусть $dp[i][j]$ равно минимальному количеству предметов, которое нужно взять из множества $\{m_1, \ldots, m_i\}$, чтобы собрать вес j. Если вес j собрать нельзя, то $dp[i][j] = \text{inf}$, где inf – достаточно большое число, например, $\text{inf} = 10^9$.
Переход будет выглядеть следующим образом:
$dp[i][j] = \min\{dp[i-1][j], dp[i-1][j-m_i]\}$, если конечно $j \geq m_i$, иначе у нас нет выбора, и мы берём значение $dp[i-1][j]$.
Таким образом осталось определиться с граничными значениями. Предлагается сделать следующее: так как чтобы собрать вес равный 0 из любого подмножества предметов нам потребуется 0 предметов, то
$\forall i \in [1, N] \Rightarrow dp[i][0] = 0$, а также $dp[1][m_1] = 1$, все остальные элементы двумерного массива dp сделаем равными inf. ◄
Временная сложность решения: $O(NM)$

Задача B. Гирьки: кучки одного размера

Разделите набор $\{m_1, \ldots, m_N\}$ на две кучки равной массы, содержащие равное число гирек.

►
Сперва нужна парочка костылей, а именно: сразу отметаем варианты когда общая сумма или общее количество на 2 не делится. Если всё же обе величины чётные, то сразу делаем так $m = m/2$, ведь насинтересует лишь возможность формирования из гирек в сумме веса $m/2$ и более того лишь из $n/2$ гирек. Данное решение подразумевает динамику по трём аргументам.
Пусть $able[i][k][j]$ равно **true**, если используя k гирек из множества $\{m_1, \ldots, m_i\}$, можно собрать вес j и **false** в противном случае.

Переход:

```
able[i][k][j] = able[i-1][k][j];
if (k > 0 && j >= w[i]) able[i][k][j] |= able[i-1][k-1][j-w[i]];
```

где $w[i]$ означает вес i-й гирьки.

Теперь к граничным значениям. Тут всё просто, всё, что нужно, это установить $able[0][0][0] =$ **true**, вполне логично, что из 0 гирек можно собрать вес 0, притом, как ни странно, этого достаточно, все остальные значения массива *able* устанавливаем в **false**.

Где хранится ответ и как восстанавливать путь?

Ответ хранится в $able[n][n/2][m]$, при условии, что мы поделили m на 2. Восстанавливаем его мы следующим образом:

```
vector<int> b, c;
int cnt = n/2;
for (int i = n; i >= 1; --i) {
    if (cnt > 0 && m >= w[i] && able[i-1][cnt-1][m-w[i]]) {
        b.PB(i);
        --cnt;
        m -= w[i];
    } else c.PB(i);
}
```

Где b и c контейнеры для первой второй кучки ответа. ◀

Временная сложность решения: $O(N^2M)$

Задача J. Жадный калькулятор.

Задано алгебраическое выражение, составленное из неотрицательных вещественных чисел и знаков операций +, - и *. Требуется так расставить в этом выражении скобки, чтобы его значение стало максимально возможным.

▶

Для начала научимся считывать алгебраическое выражение записывая числа в массив **nums** и операции в массив **operations**.

Будем рассматривать наше выражение (строку) s поэлементно. Строка k будет постепенно набирать в себя цифры крайнего числа, и как только мы встречаем символ $+$ $*$ - становится понятно что цифры этого числа кончились, и, преобразуя строку k, мы добавляем ее новым элементом массива.

```
1   getline(cin, s);
2   string k = "";
3   for (int i=0; i < s.size(); ++i) {
4       if (s[i] == '+' || s[i] == '*' || s[i] == '-') {
5           if (k != "") {
6               nums.push_back(atof(k.c_str()));
7               k = "";
8           }
9
10          k = "";
11          operations.push_back(s[i]);
12      }
13      else if (s[i] == ' ') {
14          if (k != "") {
15              nums.push_back(atof(k.c_str()));
16              k = "";
17          }
18      }
19      else {
20          k += s[i];
21      }
22  }
23  if (k != "")  nums.push_back(atof(k.c_str()));
24  if(nums.size() == 1) {
25      cout << nums[0] << endl << nums[0];
26      return 0;
27  }
```

Строки с 24 по 27 проверяют случай при котором выражение состоит из 1 числа и сразу выводят ответ.

Теперь перейдем к разбору алгоритма нахождения наибольшего значения выражения содержащего N элементов. Пусть $Max(i, j)$ обозначает максимально возможное после расстановки скобок значение этого куска, а $Min(i, j)$ - минимально возможное. Эти числа слежует вычислять парами в порядке увеличения длин кусков $j - i + 1$. Ясно, что для всех i от 1 до N значения $Max(i, i)$ и $Min(i, i)$ совпадают и равны i-му члену выражения. При $i < j$ число $Max(i, j)$ вычисляем следующим образом. Перебираем все значения k, такие что $i \leq k \leq j$, и каждый раз предполагаем, что при вычислении значения рассматриваемого куска самой последней выполняется операция, записанная между числами с номерами k и $k+1$. Если это, к примеру операция вычитания, то чтобы максимизировать значение части выражения от i-го числа до j-го, мы должны взять максимальное значение куска от i-го числа до k-го (которое уже вычислено) и вычесть из него минимальное значение куска от $k+1$-го числа до j-го (которое также уже вычислено). Из значений, полученных при анализе различных k, выбираем максимальное.

```
    for (int i = 0; i < nums.size(); i++) {
        dp[i][i] = nums[i];
        mn[i][i] = nums[i];
    }

    for (int l = 1; l < nums.size(); l++) {
        for (int st = 0; st < nums.size() - l; st++) {
            for (int j = st; j < st + l; j++) {
                if (operations[j] == '+') {
                    mn[st][st + l] = min(mn[st][st + l], mn[st][j] + mn[j + 1][st + l]);
                    dp[st][st + l] = max(dp[st][st + l], dp[st][j] + dp[j + 1][st + l]);
                }
                if (operations[j] == '-') {
                    mn[st][st + l] = min(mn[st][st + l], mn[st][j] - dp[j + 1][st + l]);

                    dp[st][st + l] = max(dp[st][st + l], dp[st][j] - mn[j + 1][st + l]);
                }
                if (operations[j] == '*') {
                    mn[st][st + l] = min(mn[st][st + l], dp[st][j] * dp[j + 1][st + l]);
                    mn[st][st + l] = min(mn[st][st + l], mn[st][j] * mn[j + 1][st + l]);
                    mn[st][st + l] = min(mn[st][st + l], dp[st][j] * mn[j + 1][st + l]);
                    mn[st][st + l] = min(mn[st][st + l], mn[st][j] * dp[j + 1][st + l]);

                    dp[st][st + l] = max(dp[st][st + l], dp[st][j] * dp[j + 1][st + l]);
                    dp[st][st + l] = max(dp[st][st + l], mn[st][j] * mn[j + 1][st + l]);
                    dp[st][st + l] = max(dp[st][st + l], mn[st][j] * dp[j + 1][st + l]);
                    dp[st][st + l] = max(dp[st][st + l], dp[st][j] * mn[j + 1][st + l]);
                }
            }
        }
    }
    cout << dp[0][nums.size() - 1] << endl;
```

$dp[i][j]$ обозначает максимально возможное после расстановки скобок значение куска от i-го числа до j-го, а $mn[i][j]$ минимально возможное после расстановки скобок значение куска от i-го числа до j-го. Так как значения $dp[i][j]$ и $mn[i][j]$ следует вычислять парами в порядке увеличения кусков, будем считать их циклом для l от 1 до количества чисел в выражении, где l - количество элементов рассматриваемого куска. Внутри этого цикла нужно написать второй для st от 0 до количества чисел в выражении минус l, где st первый элемент куска, а $st + l$ тогда будет последним. Так мы вычислим значения всех $dp[st][st + l]$ и $mn[st][st + l]$, где $0 \leq st \leq st + l \leq N$, N - количество чисел в выражении.
Исходя из алгоритма написанного выше, мы понимаем что теперь чтобы посчитать $dp[st][st+l]$ и $mn[st][st+l]$ нам нужно перебрать все $operations[j]$ такие, что $st \leq j < st+l$. $operations[j]$ является операцией между $nums[j]$ и $nums[j + 1]$. Для этого и существует третий цикл for.
Рассмотрим третий цикл подробнее.
Если операция $operations[j]$ это $+$, то $mn[st][st + l]$ будет минимумом среди всех $mn[st][j] + mn[j + 1][st + l]$, потому что минимальное значение суммы достигается при наименьших слагаемых. Аналогично $dp[st][st + l]$ максимум среди всех $dp[st][j] + mn[j + 1][st + l]/$
Если операция $operations[j]$ это $-$, то $mn[st][st+l]$ будет минимумом среди всех $mn[st][j] - dp[j+1][st+l]$, так как минимальное значение разности достигается при наименьшем уменьшаемым и наибольшем вычитаемым. Аналогично $dp[st][st + l]$ максимум среди всех $dp[st][j] - mn[j + 1][st + l]$.
В случае, когда $operations[j]$ это $*$ есть небольшие отличия. Значения $dp[st][j]$, $mn[st][j]$, $dp[j + 1][st + l]$ и $mn[j + 1][st + l]$ могут быть как положительными, так и отрицательными, поэтому значение $dp[st][st + l]$ и $mn[st][st + l]$ лучше брать как максимум и минимум соответственно среди всех возможных комбинаций умножения $mn[st][j]$ или $dp[st][j]$ на $mn[j+1][st+l]$ или на $dp[j+1][st+l]$, что можно видеть в коде со строки 67 по 75.
Таким образом в $dp[0][nums.size() - 1]$ будет храниться максимально возможное значение выражения.

Перейдем к нахождению положения скобок.
Для начала создадим два массива **open_brackets** и **close_brackets**. $open\_brackets[i]$ будет содержать количество открывающихся скобок перед i-м числом выражения и $close\_brackets[i]$ количество открывающихся скобок после i-м числом выражения.
Будем использовать рекурсивную функцию
textbffind_brackets
для нахождения скобок. Она перебирает все опрерации $operations[i]$ которые находятся между элементами $nums[start]$ и $nums[finish]$, и как только находит такое i что, например, если $operations[i]$ это минус, то $dp[start][finish] = mn[start][i] - dp[i+1][finish]$
Как только такое i было найдено мы запускаем функцию от значений $start$ и i, а также от $i+1$ и $finish$. Третий аргумент $flag$ равен 1 если нам нужно вызвать функцию **find_brackets** для dp[i][j], то есть $find\_brackets(i, j, 1)$ если же нам нужно вызвать функцию для mn[i][j] $flag$ будет равен 0, то есть функцию нужно будет вызвать так $find\_brackets(i, j, 0)$.
Аргумент $flag$ нужен по той причине, что иногда нам нужно найти такое расположение скобок, чтобы кусок выражения между $start$ и $finish$ имел максимальную сумму, а иногда такое расположение, чтобы имел минимальную.
Так например продолжая пример выше, нами будут вызваны $find\_brackets(start, i, 0)$ и $find\_brackets(i+1, finish, 1)$

Вывод итогового выражения

Для каждого элемента i выражения мы сначала выводим все открывающиеся скобки, количество которых содержится в $open\_brackets[i]$, затем сам элемент, все закрывающиеся скобки, количество которых содержится в $close\_brackets[i]$ и потом операцию после этого элемента. ◀

Задача L. Оптимальная триангуляция - 1.

Триангуляцией N-угольника называется набор из N-3 непересекающихся (кроме как в вершинах многоугольника) диагоналей, разбивающих N-угольник на N-2 треугольника. Для заданного выпуклого N-угольника найдите триангуляцию, у которой сумма длин диагоналей, входящих в триангуляцию, минимальна.

►

Пронумеруем все вершины от 0 до n - 1. $dp[i][j]$ - минимальная сумма длин диагоналей по всем триангуляциям многоугольника, образованного вершинами $i, i+1, ..., j$. Тогда ответ на задачу это $dp[0][n-1]$. Как считать $dp[i][j]$?

Рассмотрим отрезок, соединяющий вершины i и j. Он должен входить в какой-то треугольник триангуляции. Переберем третью вершину $k(i < k < j)$ этого треугольника. Для данного k понятно, что ответ равен

$dp[i][j] = dp[i][k] + dp[k][j] + dist[i][k] + dist[k][j]$

где $dist[i][j]$ это длина хорды между двумя вершинами (после выбрасывания треугольника (i, j, k) мы получаем два многоугольника от i до k и от k до j). Если вершины имеют соседние номера, то хорды между ними нет, следовательно ее длина равна нулю.

При $j = i + 2$ получится не совсем то, так как прибавятся стороны. Поэтому для $i + 2 \geq j$ $dp[i][j] = 0$.

Для $j > i + 2$ получаем формулу

$dp[i][j] = min\{i < k < j : dp[i][k] + dp[k][j] + dist[i][k] + dist[k][j]\}$ ◄

Временная сложность решения: $O(N^3)$