Лекция 5. Введение в динамическое программирование: одномерная и двумерная динамика.

Рекуррентные соотношения. Динамическое программирование: основные определения. Одномерная динамика: подсчет количества вариантов решения, поиск оптимального решения. Восстановление ответа. Двумерная динамика. Задача о рюкзаке

Общее представление

Задачи, в которых при наличии нескольких возможных вариантов решения, предполагается выбор в качестве ответа одного наилучшего (оптимального) решения, теоретически можно решать перебором всевозможных вариантов и выбором среди них наилучшего. При больших размерностях входных данных перебор всех вариантов (так называемый метод «грубой силы») не будет эффективен по времени. В этом случае можно попробовать решить задачу большей размерности при помощи решения подзадач меньшей размерности. Такой метод решения *называется динамическим программированием*.

Для того, чтобы задача могла быть решена методом динамического программирования должны быть выполнены следующие условия: *в задаче можно выделить подзадачи аналогичной структуры меньшего размера*; среди выделенных подзадач *есть тривиальные*, то есть имеющие «малый размер» и очевидное решение; *оптимальное решение* подзадачи большего размера может быть построено *из оптимальных решений подзадач*; *решения подзадач запоминаются в таблицы*, имеющие разумные размеры.

Задачи динамического программирования делятся на два типа. *Подсчет количества вариантов решения*. В этом случае находят суммарное количество решений подзадач. *Поиск оптимального решения (оптимизация целевой функции)*. В этом случае выбирают лучшее среди всех решений подзадач.

Одномерная динамика. Решение задач

Рассмотрим примеры двух указанных выше типов задач на примере задач, решаемых с помощью одномерной динамики, то есть динамики, в которой используется один параметр.

Подсчет количества вариантов решения. Последовательность из 0 и 1.

Требуется подсчитать количество последовательностей длины N , состоящих из 0 и 1, в которых никакие две единицы не стоят рядом.

Если воспользоваться полным перебором, то, например, для N=64 потребуется сгенерировать 2^{64} последовательностей из нулей и единиц и проверить каждую из них на соответствие условию задачи, что неэффективно по времени. В общем случае получаем сложность такого ния - $O(N2^N)$. Рассмотрим решение задачи с помощью динамического программирования.

Приведем примеры последовательностей, для некоторых небольших чисел N.

N=1	N=2	N=3	N=4
0	00 01	000, 001 010	0000, 0001, 0010 0100, 0101
1	10	100, 101	1000, 1001, 1010
2варианта	3 варианта	5 вариантов	8 вариантов

Из таблицы видим, что к каждой последовательности длиной ***(N-1)*** можно приписать ***0*** в любом случае и ***1*** только в том случае, если она заканчивается на ***0*** (по условию задачи). То есть,

каждая последовательность длины ***(N-1)*** может быть продолжена либо одним, либо двумя способами.

Обозначим за ***i*** длину последовательности. *F(i)* – количество последовательностей длины ***i***, удовлетворяющих условию задачи.

Присвоим начальные значения динамики: F[1]=2; F[2]=3. Это будет база динамики.

Правило перехода динамики – F(i+1)=F(i)+F(i-1) позволяет получить ответ для большей подзадачи на основе ответов к предыдущим подзадачам. Заметим, что мы получили правило динамики такое же, как и для вычисления чисел Фибоначчи.

Ответом к задаче будет являться число F(N).

Сложность такого решения O(N).

Подсчет количества вариантов решения. Оптимизация линейной динамики

Задача вычисления чисел Фибоначчи и ей подобные решаются при помощи линейной динамики. Если параметр N является слишком большим (например, порядка 10^{18}), то даже линейная динамика не будет давать оптимальное по времени решение. Последовательность чисел Фибоначчи является возрастающей. Приведем оценку верхней границы N-го числа Фибоначчи.

Оценка скорости роста чисел Фибоначчи: $F_N \geq 2^{\frac{N}{2}}$, для $N \geq 6$.

Оценку можно получить, применив метод математической индукции.

База индукции: $F_6 = 8 = 2^{\frac{6}{2}}$, $F_7 = 13 > 2^{\frac{7}{2}} = 8\sqrt{2}$.

Переход индукции. При $n \geq 8, F_n = F_{n-1} + F_{n-2} \geq 2^{\frac{n-1}{2}} + 2^{\frac{n-2}{2}}$>=2· $2^{\frac{n-2}{2}}$=$2^{n/2}$.

Таким образом, $F_N \geq 2^{\frac{N}{2}}$

Рассмотрим на примере простейшей задачи вычисления чисел Фибоначчи *оптимизацию линейной динамики.* По условию задачи F[0]=0; F[1]=1. F(i+1)=F(i)+F(i-1). Для вычисления следующего элемента достаточно знать значение двух предыдущих элементов. Два предыдущих значения образуют вектор (F(i-1), F(i)). На следующем шаге мы получаем вектор (F(i), F(i+1)). Чтобы из первого вектора получить второй мы можем умножить его на матрицу $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Проверим корректность перехода. Действительно, $\big(\mathrm{F(i-1)}, \mathrm{F(i)}\big)\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (\mathrm{F(i)}, \mathrm{F(i+1)})$. Получаем:

$$\big(\mathrm{F(N-1)}, \mathrm{F(N)}\big) = \big(\mathrm{F(N-2)}, \mathrm{F(N-1)}\big)\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \ldots = \big(\mathrm{F(0)}, \mathrm{F(1)}\big)\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{N-1} = \big(\mathrm{F(0)}, \mathrm{F(1)}\big)A^{N-1}.$$

Возведение матрицы в степень, то есть, вычисление A^{N-1} вычисляется при помощи бинарного возведения в степень. Затем вектор $\big(\mathrm{F(0)}, \mathrm{F(1)}\big)$ просто умножается на полученную матрицу. Такое решение будет работать за время $O(\log(N)\, K^3)$, где N – показатель степени матрицы (номер числа Фибоначчи), K – размерность матрицы. В рассмотренной задаче случае K=2. Таким образом, получаем лучшую асимптотику вычисления N-го числа Фибоначчи. Заметим, что при таком подходе мы можем вычислить только N-е число, а не все числа.

Определение оптимального решения. Задача о кузнечике, который собирает монеты

Задачи, в которых требуется определить наилучшее решение среди возможных (производится поиск оптимального решения), также удобно решать при помощи динамического программирования. Проиллюстрируем применение метода динамического программирования для поиска оптимального решения на конкретной задаче.

Задача. Кузнечик собирает монеты

Кузнечик прыгает по столбикам, расположенным на одной линии на равных расстояниях друг от друга. Столбики имеют порядковые номера от 1 до N . Вначале Кузнечик сидит на столбике с номером 1. Он может прыгнуть вперед на расстояние от 1 до K столбиков, считая от текущего.

На каждом столбике Кузнечик может получить или потерять несколько золотых монет (для каждого столбика это число известно). Определите, как нужно прыгать Кузнечику, чтобы собрать наибольшее количество золотых монет. Учитывайте, что Кузнечик не может прыгать назад.

Входные данные

В первой строке вводятся два натуральных числа: N и K ($2 \le N$, $K \le 10000$), разделённые пробелом. Во второй строке записаны через пробел N - 2 целых числа – количество монет, которое Кузнечик получает на каждом столбике, от 2-го до N - 1 -го. Если это число отрицательное, Кузнечик теряет монеты. Гарантируется, что все числа по модулю не превосходят 10000.

Выходные данные

В первой строке программа должна вывести наибольшее количество монет, которое может собрать Кузнечик. Во второй строке выводится число прыжков Кузнечика, а в третьей строке – номера всех столбиков, на которых побывал Кузнечик (через пробел в порядке возрастания).

Если правильных ответов несколько, выведите любой из них.

Примеры

Входные данные	Выходные данные
5 3 2 -3 5	7 3 1 2 4 5

Для решения задачи будем использовать массив динамики d[], где d[i] – наибольшее количество золотых монет, которое может собрать кузнечик, находясь в данный момент на столбике *i*.

База динамики. Начальное заполнение массива динамики d[1]=0 – если кузнечик находится на первом столбике, то он собрал 0 монет.

Правило перехода динамики d[i]=d[num_max]+a[i], где num_max – номер столбика с максимальным количеством собранных кузнечиком монет, который предшествовал данному. То есть, кузнечик, находясь на столбике с номером *i*, может собрать такое максимальное количество монет, которое равно сумме предыдущего максимального количества монет и количеству монет на текущем столбике.

Ответом к задаче будет являться число d[n], где n – номер последнего столбика.

```
     d[1]=0; a[1]=0; a[n]=0;
     for (int i=2;i<=n;++i){
          int num_max = i - 1;
//поиск предыдущего столбика с максимальным количеством монет
          for(int j=max(1,i-k);j<=i-1;++j)
               if (d[j]>d[num_max]) {
                    num_max=j;
               }
          d[i]=d[num_max]+a[i]; //Текущее максимальное значение
          p[i] = num_max;
     }
     printf("%d\n", d[n]);
```

Дополнительно в программе используется массив p[], где p[i] – номер столбика с максимальным количеством монет, предшествующего i-му столбику. Этот массив будет использоваться для восстановления ответа – перечисления всех номером столбиков, на которых побывал кузнечик.

Восстановление ответа

В задаче про кузнечик и многих других нужно не только дать ответ в виде количества способов решения или наилучшего решения, но и указать всевозможные способы или способ достижения этого наилучшего решения. Например, в задаче про кузнечика требовалось указать номера столбиков, по которым перемещался кузнечик. В программе был использован массив предков p[], который для каждого i столбика хранил значение p[i] – номер того столбика, с которого кузнечик прыгнул на i столбик. Теперь мы просто можем переходить к предыдущему столбику с конца решения вот таким образом - i=p[i]. Это позволит восстановить ответ – указать цепочку столбиков, на которых был кузнечик. Получаемые на каждом шаге цикла номера столбиков будем записывать в массив ans. Поскольку заполнение массива происходит, начиная с последнего столбика и, заканчивая первым, то вывод вектора ans нужно осуществлять в обратном порядке. Приведем фрагмент программы.

```
int i=n;i
vector<int> ans;
ans.push_back(i);
do   {
     i=p[i];
     ans.push_back(i);
}
while (i>1);
printf("%d\n",ans.size()-1);
for(auto i=ans.rbegin();i!=ans.rend();++i)
     printf("%d ",*i);
```

Для теста, указанного в задаче проиллюстрируем заполнение массива динамики d и массива предков p. Входные данные:5 – количество столбиков, 3 – количество прыжков, которые может делать кузнечик (от 1 до 3), 2 -3 5 – количество монет, которые лежат на столбиках.

i - номер столбика	**1**	**2**	**3**	**4**	**5**
a[i] - количество монет	0	2	-3	5	0
d[i]	0	2	-1	7	7
p[i]	-1	1	2	2	4

Из заполненной таблицы следует, что d[5]=7 – максимальное количество собранных монет. Восстановление ответа (столбиков, на которых был кузнечик) проходит следующим образом: 5 столбик – последний, на котором он был; его предком является столбик 4; в свою очередь его предком является столбик 2, а его предком, столбик 1. То есть получается последовательность элементов вектора ans: 5, 4, 2, 1. Выведем ее в обратном порядке и получим: 1, 2, 4, 5.

Задача о наибольшей возрастающей подпоследовательности

Рассмотрим классическую задачу динамического программирования о наибольшей возрастающей подпоследовательности (НВП). В ней требуется найти оптимальное решение и восстановить ответ.

Дана последовательность, требуется найти длину её наибольшей возрастающей подпоследовательности. Подпоследовательностью последовательности называется некоторый набор её элементов, не обязательно стоящих подряд.

Входные данные

В первой строке входных данных задано число N - длина последовательности ($1 \le N \le 1000$). Во второй строке задается сама последовательность (разделитель - пробел). Элементы последовательности - целые числа, не превосходящие 10000 по модулю.

Выходные данные

Требуется вывести длину наибольшей строго возрастающей подпоследовательности и саму наибольшую возрастающую подпоследовательность.

Разбор решения задачи. Определим для каждого i – номера элемента в последовательности ту наибольшую возрастающую подпоследовательность, которая построена из элементов промежутка $[a_0,..,a_i]$ и заканчивается элементом a_i. В массиве d[] будем в качестве d[i] хранить количество элементов в соответствующей НВП, заканчивающейся элементом a[i].

База динамики. d[0]=1. Длина НВП для одного первого символа равна 1. В качестве НВП в данном случае выступает сам элемент.

Правило перехода динамики. Предположим, что каждое d[i] – уже вычисленное для каждого элемента i оптимальное значение длины НВП, состоящей из подходящих элементов последовательности $[a_0,..,a_i]$ и заканчивающейся элементом a[i]. Чтобы определить $d[i+1]$ надо посмотреть все предыдущие элементы $d[0..i]$, для которых выполняется a[i]<a[i+1], выбрать наибольший такой элемент d[k] и прибавить к нему 1. То есть, найти в предыдущей последовательности наибольшую длину НВП, заканчивающуюся элементом, меньшим, чем текущий элемент a[i] последовательности. $d[i+1] = max_{0 \le k \le i}(d[k]|a[k]<a[i])+1$.

Ответом к задаче будет наибольшее число массива d[].

Проиллюстрируем решение примером. Дана последовательность a[]={1 4 2 5 6 3 7}. Рассмотрим изменение массива d[].

i	0	1	2	3	4	5	6
a[i] Массив	1	4	2	5	6	3	7
d[i] Динамика	1	2	2	3	4	3	5
НВП. Пример конкретной НВП	1	1, 4	1, 2	1, 2, 5	1, 2, 5, 6	1, 2, 3	1, 2, 5, 6, 7

Ответом для задачи будет длина НВП, равная 5. Это максимальное значение среди элементов массива d[]. Для восстановления ответа, начиная с $max_{i=0..n}d[i]$ (ответа к задаче), будем двигаться к началу массива динамики. Каждый раз, осуществляя поиск элемента массива динамики, на единицу меньшего, чем текущий элемент, перемещаемся к началу массива d[] до тех пор, пока не дойдем до какого-либо d[i]=1. Второй способ восстановления ответа заключается в том, чтобы хранить в массиве last[i] значение индекса предпоследнего элемента в НВП, заканчивающейся i-м элементом. Вначале значение элементов массива last[] заполняются значениями (– 1) – барьерами, указывающими предварительно на то, в НВП нет элементов, предшествующих данному элементу. В приведенном примере last={0,2,3,4}. По такому вспомогательному массиву легко восстанавливается ответ: a[0]=1, a[2]=2, a[3]=5, a[4]=6, a[6]=7.

Фрагмент кода программы, соответствующий поиску оптимального решения (поиску длины НВП) приведен ниже.

```
d[0]=1; last[0]=-1; //База динамики
for (int i=1;i<n;++i){ // Правило перехода динамики
     max=0; last[i]=-1;
     for (int k=i-1; k>=0; --k){
          if (d[k]>max&&a[k]<a[i]) {
               max=d[k];
               last[i]=k;  //Запоминаем номер предпоследнего
          }                //элемента НВП
     }
          d[i]=max+1;
}
```

Для вывода ответа – длины НВП, необходимо вывести максимальное значение динамики.

```
max=d[1];
for (int i=0;i<n;++i)
     if (d[i]>max) {
               max=d[i]; l=i; //Индекс элемента,
     }                        //на который заканчивается НВП
```

Восстановление ответа при помощи массива last[]. В ответе ans[] помещаются элементы в обратном порядке.

```
ans.push_back(a[l]); //добавили последний элемент НВП
while(last[l]!=-1){ //Пока не дошли до барьерного элемента
   ans.push_back(a[last[l]]);//добавляем в ans предыд.эл.НВП
     l=last[l];              //переход на индекс пред.эл.НВП
}
for (auto i=ans.rbegin();i!=ans.rend(); ++i)
     cout<<*i<<" ";  //Вывод ответа в обратном порядке
```

Сложность решения $O(N^2)$.

Двумерная динамика. Задача о рюкзаке

В одномерной динамике для правил перехода динамики используется одномерный массив динамики. Двумерная динамика использует два параметра динамики и соответственно двумерные массивы динамики. Рассмотрим классическую задачу о рюкзаке, на примере которой проиллюстрируем двумерное динамическое программирование.

Дано N предметов массой $m_1, \ldots, m_N$ и стоимостью $c_1, \ldots, c_N$ соответственно.

Ими наполняют рюкзак, который выдерживает вес не более M. Какую наибольшую стоимость могут иметь предметы в рюкзаке?

Входные данные

В первой строке вводится натуральное число N, не превышающее 100 и натуральное число M, не превышающее 10000.

Во второй строке вводятся N натуральных чисел m_i, не превышающих 100.

Во третьей строке вводятся N натуральных чисел c_i, не превышающих 100.

Выходные данные

Выведите одно целое число: наибольшую возможную стоимость рюкзака.

Примеры

Входные данные	Выходные данные
4 6 2 4 1 2 7 2 5 1	13
Комментарии. Дано 4 предмета, имеющих параметры (масса, стоимость): {(2,7), (4,2), (1,5), (2,1)}. Рюкзак выдерживает вес не более 6 кг. Чтобы набрать в рюкзак предметы наибольшей стоимости оптимальнее всего взять предметы {(2,7), (1,5), (2,1)}. В этом случае рюкзак будет иметь массу 5 кг и стоимость 13.	

Разбор решения задачи

Информацию о предметах будем хранить в векторе пар s(n), каждый элемент которого – пара (масса, стоимость): Двумерный массив динамики d состоит из элементов d[i][j], где d[i][j] - максимальное значение стоимости рюкзака в случае возможности использования первых i предметов, которыми наполняется рюкзак вместимостью j. 0<=i<=N, 0<=j<=m. Размер таблицы динамики *n x m.*

База динамики. Начальное заполнение массива d[i][j]=0. Первоначальные стоимости рюкзака равны нулю.

Правило перехода динамики. Каждый предмет либо будет помещен в рюкзак, либо не будет помещен в него. Тип данной задачи – рюкзак без повторений. В случае типа задачи – рюкзак с повторениями, каждый предмет может быть использован необходимое количество раз. Оптимальное значение d[i][j] выбирается как максимум из двух возможных случаев:

- предмет i не помещают в рюкзак. В этом случае стоимость рюкзака вместимостью j остается такой же, какой она была без данного предмета, то есть d[i-1][j]. Разумеется, предмет не помещается в рюкзак если масса предмета больше вместимости рюкзака.
- предмет i помещается в рюкзак. В этом случае стоимость рюкзака вычисляется как сумма стоимости рюкзака без данного предмета и стоимости этого предмета $d[i-1][j-m_i]+c_i$.

Таким образом, получаем правило перехода динамики. $d[i][j]=max\ \{d[i-1][j], d[i-1][j-mi]+ci\}$.
Ответом к задаче будет являться элемент d[n][m].

Приведем фрагмент программы, иллюстрирующий правило перехода динамики.

```
  for(int i = 1; i<=n; ++i){//перебор по предметам
    for(int j = 1; j<=m; ++j){//перебор по массе рюкзака
//если предмет имеет допустимую массу для помещения его в рюкзак
//и если предмет выгоднее поместить в рюкзак, то помещаем его
     if((j>=s[i].first)&&(d[i-1][j]<(d[i-1][j-
s[i].first]+s[i].second)))
                d[i][j]=d[i-1][j-s[i].first]+s[i].second;
            else
//если не выгодно добавлять предмет в рюкзак, то не добавляем его
                d[i][j] = d[i-1][j];
        }
    }
```

Сложность алгоритма $O(n \times m)$.

Рассмотрим иллюстрацию решения задачи о рюкзаке на примере.

W=10 – вместимость рюкзака. N=4 - количество предметов.

1 предмет. m1=6 c1=30

2 предмет. m2=3 c2=14

3 предмет. m3=4 c3=16

4 предмет. m4=2 c4=9

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	30	30	30	30	30
2	0	0	0	14	14	14	30	30	30	44	44
3	0	0	0	14	16	16	30	30	30	44	46
4	0	0	9	14	16	23	30	30	39	44	46

Максимальная стоимость рюкзака – 46.

В таблице цветом отображено восстановление ответа – выделены ячейки, которые представляют собой оптимальную промежуточную стоимость рюкзака d[i][j].

Получим оптимальный набор вещей: в рюкзак помещают 1 и 3 предмет.

LR динамика

В случае обработки строк и поиска в таких задачах оптимального решения удобно использовать подход LR динамики (динамики по подстрокам). L – левая граница рассматриваемой подстроки, R – правая граница рассматриваемой подстроки. Решение подзадач представляет собой поиск оптимального решения для подстрок строки. Рассмотрим применение LR динамики на примере конкретной задачи.

Задача. Удаление скобок

Дана строка, составленная из круглых, квадратных и фигурных скобок. Определите, какое наименьшее количество символов необходимо удалить из этой строки, чтобы оставшиеся символы образовывали правильную скобочную последовательность.

Входные данные

Строка из круглых, квадратных и фигурных скобок. Длина строки не превосходит 100 символов.

Выходные данные

Выведите строку максимальной длины, являющуюся правильной скобочной последовательностью, которую можно получить из исходной строки удалением некоторых символов. Если возможных ответов несколько, выведите любой из них.

Примеры

Входные данные	Выходные данные
([)]	[]
{([(]{)})]	[({})]

Разбор решения задачи

Считываем скобочную последовательность в строку s. Двумерный массив динамики состоит из элементов dp[i][j] – наименьшего количества символов, которые надо удалить из подстроки s[i..j], левый символ которой имеет позицию *i*, правый символ – *j*. Размер таблицы $n \times n$, где n – длина строки *s*.

База индукции - dp[i][i]=1. Если строка состоит из одной скобки (левая граница подстроки совпадает с правой границей), то нужно удалить эту единственную скобку. Заметим, что в данной задаче начальное заполнение двумерной таблицы динамики проходит по диагонали. Значения dp[i][j] при i>j заполняются нулями (левая граница строки не может быть больше правой границы).

Правило перехода динамики. Рассмотрим сначала отдельно случай, когда в строке s[l..r] на позициях l и r стоят соответствующие друг другу по типу открывающаяся и закрывающаяся скобки. В этом случае количество удаляемых скобок будет равно этому количеству для подстроки s[l+1..r-1]. Например, для строки ({{}}) количество удаляемых скобок равно количеству удаляемых скобок в последовательности {{}} – 2 скобки. Исключение составляют ситуации совокупности изначально правильных последовательностей, например (){}(). Если переходить к подстроке s[l+1..r-1] =”){}(“, то получим ответ 2. В то же время, ответ к строке (){}() – 0 (ноль) удаляемых скобок. Поэтому, нужно учесть такого рода ситуации для строк s[l..r] с соответствующими друг другу по типу открывающейся и закрывающейся скобками, и выбрать наименьшее значение – или из таблицы динамики, или при переходе к подстроке s[l+1..r-1].

Рассмотрим общий случай. Чтобы вычислить d[l][r] – наименьшее количество скобок, которое нужно удалить из подстроки s[l..r] разобьем всевозможными способами строку s[l..r] на две подстроки s[l,k] и s[k+1,r], где k=l…r-1. Очевидно, что если подстроки s[l,k] и s[k+1,r] сделать правильными скобочными последовательностями, удалив лишние строки в них, то и строка s[l..r] станет правильной скобочной последовательностью. Поэтому остается найти минимальное значение суммарного количества удаляемых скобок для всевозможных разбиений строки на две подстроки

– $min_{l \le k \le r-1}(s[l,k] + s[k+1,r])$. Это значение и будет давать наименьшее количество удаляемых скобок в строке s[l..r].

Рассмотрим заполнение таблицы динамики для конкретного примера.

Входные данные. ({ {) } } – скобочная последовательность.						
L\R	0	1	2	3	4	5
0	1	2	3	2	3	2
1	0	1	2	3	2	1
2	0	0	1	2	1	2
3	0	0	0	1	2	3
4	0	0	0	0	1	2
5	0	0	0	0	0	1

Значение динамики	Значение динамики
Пример вычисления значения d[0][5]	Min({0,0}+{1,5},{0,1}+{2,5},{0,2}+{3,5},{0,3}+{4,5}, {0,4}+{5,5}) = 2

Приведем ниже фрагмент программы решения задачи. Заметим, что матрица динамики заполняется по столбцам, начиная с диагонали каждого столбца (в противном случае придется обращаться к невычисленным еще значениям). В программе используется вспомогательный массив, элементы которого ep[l][r] хранят индексы k, указывающие на оптимальное разбиение строк $s[l,k]$ на две подстроки $s[l,k]$ и $s[k+1,r]$.

```
int n = s.size();
     for (int r = 0; r < n; ++r)
     for (int l = r; l >= 0; --l){
          if (l == r)
               dp[l][r] = 1; // База динамики
          else{
               int best = 1000000; int mk = -1;
               if (s[l] == '(' && s[r] == ')' || s[l] == '[' &&
s[r] == ']' || s[l] == '{' && s[r] == '}')
               //Случай соответствующих скобок
               best = dp[l + 1][r - 1];
               //Общий случай правила перехода динамики
               for (int k = l; k < r; ++k)
               if (dp[l][k] + dp[k + 1][r] < best){
                    best = dp[l][k] + dp[k + 1][r];
                    mk = k; //поиск оптимального разбиения строки
               }
               dp[l][r] = best; ep[l][r] = mk;
          }
     }
```

Восстановление ответа для задачи про удаление скобок удобно реализовать при помощи рекурсивной процедуры rec(int l, int r). Выход из рекурсии происходит в том случае, если из скобочной последовательности нужно удалить все скобки. Если из последовательности s[l..r] ни одной скобки удалить не нужно, то это правильная скобочная последовательность – в этом случае печатается полностью подстрока s[l..r] и осуществляется выход из рекурсии.

```
void rec(int l, int r){
     if (dp[l][r] == r - l + 1)
          return;
```

```
    if (dp[l][r] == 0){
        cout << s.substr(l, r - l + 1);
        return;
    }
    if (ep[l][r] == -1) {//Если подстрока имеет в начале и конце
                  //соответствующего типа правильные скобки,
        cout << s[l]; //то печатаем левую скобку
        rec(l + 1, r - 1);//вызов рекурсию вложенной подстроки
        cout << s[r];// печатаем правую скобку
        return;
    }
    rec(l, ep[l][r]); //вызов рекурсии от левой подстроки
    rec(ep[l][r] + 1, r);//вызов рекурсии от правой подстроки
}
```

В основном тексте программы процедура rec() вызывается для всей строки.

```
    rec(0, n - 1); // Вызов рекурсии в основном тексте программы
```

По такому же принципу, как и рассмотренная задача, решается задача добавления наименьшего количества скобок в скобочную последовательность для того, чтобы она стала правильной.

Задания

1. Для задачи вычисления биномиальных коэффициентов C_n^m укажите базу динамики, правило перехода динамики и рассмотрите пример заполнения таблицы динамики для конкретного значения n и m.
2. Укажите способ восстановления ответа в задаче о рюкзаке.
3. Для задачи о рюкзаке рассмотрите пример заполнения таблицы динамики для тестового примера, приведенного в задаче лекции.
4. Укажите решение задачи о рюкзаке, если предметы можно помещать в рюкзак неограниченное количество раз.
5. Классическая задача на вычисления расстояния редактирования предполагает определение минимального количества вставок, удалений и замен символов, необходимое для преобразования строки s1 в строку s2 (так называемое расстояние Левенштейна). Предложите способ вычисления расстояния Левенштейна для двух строк.

Литература

Ворожцов А.В., Винокуров Н.А. Лекции «Алгоритмы: построение, анализ и реализация на языке программирования Си». – М.: Издательство МФТИ, 2007.

Зимняя школа по программированию. Харьков, ХНУРЭ, 2014.

Курс сайта www.stepic.org «Алгоритмы: теория и практика. Методы». Александр Куликов

Курс сайта www.stepic.org «Введение в теоретическую информатику». Александр Шень.

Курс «Олимпиадные задачи по программированию». М.С. Густокашин. Электронный ресурс. http://prog.mathbaby.ru/dokuwiki/lib/exe/fetch.php?media=2015-9m:9m-binsearch-lection.pdf

Порублев И.Н., Ставровский А.Б. Алгоритмы и программы. Решение олимпиадных задач. – М.: ООО «И.Д. Вильямс» , 2007.