Динамическое программирование

Лекция 20

План лекции

- Понятие динамического программирования
- Примеры
  - Сумма геометрической прогрессии
  - Суммирование набора
  - Задача о рюкзаке
  - Произведение матриц
  - Алгоритм Флойда-Уоршалла

Понятие динамического программирования

- Ричард Беллман ~1940
  - Какой алгоритм назван в честь этого ученого?

- Описание процесса решения задачи, при котором решение одной задачу вычисляется на основании решения "предшествующих" задач

- Один из методов численного решения задач оптимизации

- Первоначально часть системного анализа

Понятие динамического программирования

- Термин «динамическое программирование» происходит от термина «математическое программирование», который является синонимом слова «оптимизация»
- Слово «программирование» в словосочетании «динамическое программирование» к традиционному программированию почти никакого отношения не имеет
- Слово «программа» в Д.П. означает оптимальную последовательность действий для получения решения задачи
  - Расписание событий на выставке называют программой и понимают как допустимую последовательность событий

Понятие динамического программирования

- Последовательность действий в Д.П. имеет вид
  - Разбиение задачи на подзадачи меньшего размера
  - Нахождение оптимального решения подзадач рекурсивно
  - Вычисление оптимального решения исходной задачи на основании оптимальных решений подзадач

- Деление на подзадачи происходит до тех пор, пока не получатся тривиальные задачи, решаемые за константное время

- В общем случае требование оптимальности может отсутствовать (Д.П. с постоянной целевой функцией)
  - Например, вычисление $n!$ можно рассматривать как задачу Д. П. с постоянной целевой функцией и тривиальными задачами $1! = 1$ и $0! = 1$

Понятие динамического программирования

- Простая рекурсивная реализация будет расходовать время на вычисление решений для задач, которые уже один раз решались
- Чтобы избежать такого хода событий будем сохранять в таблице решение каждой решенной подзадачи
- Когда нам снова потребуется решение подзадачи, мы вместо того, чтобы вычислять его заново, просто возьмем его из таблицы
  - Этот прием называется *кэширование*

Понятие динамического программирования

- Динамическим программированием называется способ программирования, при котором задача разбивается на подзадачи, вычисление идет от малых подзадач к большим, решения подзадач запоминаются в таблице и используются при решении больших задач
- Заполнение таблицы в Д.П. называется прямым ходом
- Исходные данные подзадач называются параметрами
- Задачу можно рассматривать как функцию, аргументами которой являются параметры задачи
- Например, при нахождении суммы набора чисел параметрами задачи будут количество чисел и их значения

Понятие динамического программирования

- Под подзадачей понимается та же постановка задачи, но с меньшим числом параметров или с тем же числом параметров, но при этом хотя бы один из параметров имеет меньшее значение
- Преимущество Д.П. состоит в том, что каждую подзадачу мы решаем ровно один раз
- Сведение решения задачи к решению подзадач записывается в виде соотношений, которые выражают значение целевой функции для исходной задачи через значения целевой функции для подзадач
- Значения аргументов у любой из функций в правой части соотношения меньше значения аргументов функции в левой части соотношения
  - Если аргументов несколько, то достаточно уменьшения одного из них

Понятие динамического программирования

- Динамическое программирование может быть применено к задачам оптимизации, в которых решением является последовательность шагов, приводящая к достижению минимума или максимума целевой функции

- Процедура восстановления оптимального решения называется обратным ходом
- Соотношения между значением целевой функции для Оптимальное решение более сложной Чтобы выписать рекуррентные соотношения, связывающие оптимальные значения параметра для подзадач, двигаясь снизу вверх, вычислить оптимальные решения для подзадач и используя их построить оптимальное решение для поставленной задачи

- Для применения Д.П. бывает удобно решать не заданную задачу, а более общую задачу, и рассматривать исходную задачу как частный случай этой более общей задачи

Пример -- геометрическая прогрессия

Рассмотрим пример. Требуется вычислить сумму s следующего ряда при $x \neq 0$: $s=1 +1/x+ 1/x^2+ 1/x^3+\ldots+ 1/x^n$.

Параметры подзадач:

k <= n, x != 0

Подзадачи для k , x:

Вычисление сумм $S(k) = 1 +1/x+ 1/x^2+ 1/x^3+\ldots+ 1/x^k$

Вычисление степеней $P(k, x) = 1/x^k$

Соотношения:

S(0) = 1, P(0) = 1

S(k+1) = S(k)+P(k)/x, P(k+1) = P(k)/x

Пример – суммирование набора

- Имеется n неделимых предметов, вес i-го предмета есть w_i
- Найти список предметов, суммарный вес которых равен W кг. (если это возможно)

- Обозначим T(n, W) = 1, если искомый набор имеется
  0, если искомого набора нет

- Подзазача – вычисление T(i, j), где i -- макс. номер предмета, j – требуемый суммарный вес и $0 \le i \le n$, $1 \le j \le W$

- Параметры -- количество предметов и требуемый суммарный вес

Пример – суммирование набора

- Начальные значения функции T
  - T(0, j) = 0 при j ≥ 1
    - нельзя без предметов набрать массу j > 0
  - T(i, 0) = 1 при i ≥ 1
    - всегда можно набрать вес, равный 0

Пример – суммирование набора

- Для оптимального решения из двух возможных вариантов нужно выбрать наилучший
  - i-ый предмет в набор не берется
    - $T(i, j) = T(i - 1, j)$
    - Решение задачи с i предметами сводится к решению задачи с i – предметом
  - i-ый предмет в набор берется
    - $T(i, j) = T(i -1, j - w_i)$
    - Масса оставшихся предметов уменьшается на величину w_i
    - Эта ситуация возможна, если масса i-го предмета не больше значения j
- Соотношения
  $T(i, j)= T(i -1, j)$ при $j < w_i$
  $T(i, j)= \max (T(i -1, j), T(i -1, j - w_i))$ при $j \geq w_i$.

Пример – суммирование набора

- W = 16
- w1 = 4, w2 = 5, w3 = 3, w4 = 7, w5 = 6
- Результат прямого хода

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Пример – суммирование набора

- Обратный ход
  - Решение нашего примера определяется T[5, 16] = 1
  - T[5, 16] = T[4, 16] -- 5-ый предмет в набор не включаем
  - T[4, 16] ≠ T[3, 16] – 4-ый предмет включаем, оставшийся вес $16\text{-}w_4$ = 16-7 = 9
  - T[3, 9] =T[2, 9] – 3-ый предмет в набор не включаем
  - T[2, 9] ≠ T[1, 9]] – 2-ой предмет включаем, оставшийся вес равен $9\text{-}w_2$ = 9 - 5 = 4
  - T[1, 4] ≠ T[0, 4] – 1-ой предмет включаем, оставшийся вес равен 0

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0
3	1	0	0	1	1	1	0	1	1	1	0	0	1	0	0	0	0
4	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	1
5	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Задача о рюкзаке

- Определить наиболее ценную выборку из n предметов, подлежащих упаковке в рюкзак, вмещающий W килограммов
- Предмет i стоит c_i и весит w_i
- Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W, а суммарная стоимость была максимальна

Пример -- задача о рюкзаке

- Если перебирать все возможные подмножества данного набора из n предметов, то получится решение сложности не менее чем $O(2^n)$

- В настоящее время неизвестен алгоритм решения этой задачи, сложность которого является полиномом от n

- Построим с помощью Д.П. алгоритм со временем работы $O(nW)$ для решения данной задачи, когда все входные данные – целые числа

Пример -- задача о рюкзаке

Обозначим через T(n, W) функцию, значение которой соответствует решению нашей задачи. Аргументами функции является количество предметов n, по которому можно определить стоимость и массу каждого предмета, и ограничение по весу W.

Подзадачи – вычисление значений функции T(i, j) = max стоимость предметов, которые можно уложить в рюкзак с ограничением по весу j килограмм, если можно использовать только первые i предметов из заданных, где $0 \le i \le n$, $0 \le j \le n$.

Что является параметрами подзадачи?

Начальные значения функции T :

T(0, 0) = 0,

T(0, j) = 0 при $j \ge 1$ (нет предметов, максимальная стоимость равна 0),

T(i, 0) = 0 при $i \ge 1$ (можно брать любые из первых i предметов, но ограничение по весу равно 0).

Пример -- задача о рюкзаке

Для решения подзадачи, соответствующей функции $T(i, j)$, рассмотрим два случая.

1. i-ый предмет не упаковывается в рюкзак
   Решение задачи с i предметами сводится к решению задачи с i – 1 предметом:
   $T(i, j) = T(i - 1, j)$.
2. i-ый предмет упаковывается в рюкзак
   Масса оставшихся предметов уменьшается на величину w_i, а при добавлении i-го предмета стоимость выборки увеличивается на c_i:
   $T(i, j) = T(i - 1, j - w_i) + c_i$.
   При этом нужно учитывать, что эта ситуация возможна только тогда, когда масса i-го предмета не больше значения j.

Пример -- задача о рюкзаке

Для оптимального решения из двух возможных вариантов упаковки рюкзака нужно выбрать наилучший

Соотношение при $i \geq 1$ и $j \geq 1$:

$T(i, j) = T(i-1, j)$ при $j < w_i$

$T(i, j) = \max(T(i-1, j), T(i-1, j-w_i) + c_i)$ при $j \geq w_i$.

Пример -- задача о рюкзаке

W = 16,
$c1$ = 5,
$w1$ = 4;
$c2$ = 7,
$w2$ = 5;
c3 = 4,
$w3$ = 3;
$c4$ = 9,
$w4$ = 7;
$c5$ = 8,
$w5$ = 6.

i\j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5
2	0	0	0	0	5	7	7	7	7	12	12	12	12	12	12	12	12
3	0	0	0	4	5	7	7	9	11	12	12	12	16	16	16	16	16
4	0	0	0	4	5	7	7	9	11	12	13	14	16	16	18	20	21
5	0	0	0	4	5	7	8	9	11	12	13	15	16	17	19	20	21

- Решение примера определяется $T[5, 16] = 21$
- В примере суммарная масса предметов, подлежащих упаковке в рюкзак, совпадает с *W*, в общем-же случае она не должна превосходить величину *W*

Пример -- задача о рюкзаке

- Алгоритм обратного хода
- Требуется определить набор предметов, которые подлежат упаковке в рюкзак
- Сравним значение T[n, W] со значением T[n-1, W]
  - Если T[n, W] ≠ T[n-1, W], то предмет с номером n обязательно упаковывается в рюкзак, после чего переходим к сравнению элементов $T[n-1, W - w_n]$ и $T[n-2, W- w_n]$ и т.д.
  - Если T[n-1, W] = T[n-1, W], то n-ый предмет можно не упаковывать в рюкзак. В этом случае следует перейти к рассмотрению элементов T[n-1, W] и T[n-2, W] .

Пример -- задача о рюкзаке

- T[5, 16] = T[4, 16] -- 5-й предмет не кладем в рюкзак
- T[4, 16] != T[3, 16] – **4-й предмет кладем** в рюкзак, свободный вес равен 16 – w_4= 16 – 7 = 9
- T[3, 9] = T[2, 9] – 3-й предмет не кладем в рюкзак
- T[2, 9] != T[1, 9] -- **2-й предмет кладем** в рюкзак, свободный вес равен 9 - w_2 = 9 – 5 = 4
- T[1, 4] != T[0, 4] – **1-й предмет кладем** в рюкзак
- Итак, для нашего примера в рюкзак упакуются предметы с номерами 1, 2, 4

Пример -- задача о рюкзаке

```
void print_item(int i, int j)
{
  if (T[i][j]==0) return;              // набор предметов построен
  if (T[i-1][j] == T[i][j])
        Print_item (i-1,j);            //i-й предмет не берем
  else {
        print_item(i-1,j-w[i]);              // i-й предмет берем
        printf("%d ", i);              // печать i-го предмета
   }
}
```

Как обойтись без рекурсии?

Пример -- умножение матриц

Рассмотрим вычисление произведения *n* матриц

$$M = M_1 \times M_2 \times \ldots \times M_n. \qquad (1)$$

Порядок, в котором матрицы перемножаются, может существенно сказаться на общем числе операций, требуемых для вычисления матрицы *M,* независимо от алгоритма, применяемого для умножения матриц.

Умножение матрицы размера $[p \times q]$ на матрицу размера $[q \times r]$ требует pqr операций.

Пример – умножение матриц

Рассмотрим произведение матриц:

$$M = M_1 \times M_2 \times M_3 \times M_4$$

[10×20] [20×50] [50×1] [1× 100]

Если вычислять матрицу M в порядке: $M1 \times (M2 \times (M3 \times M4)),$ то потребуется 125 000 операций.

(50*1*100) → [50 ×100], 5000; (20*50*100) → [20 ×100], 100000;
(10*20*100) → [10 ×100], 20000.

Вычисление же в порядке: $(M1 \times (M2 \times M3)) \times M4$ требует лишь 2 200 операций.

(20*50*1) → [20 ×1], 1000; (10*20*1) → [10 ×1], 200;
(10*1*100) → [10 ×100], 1000.

Пример -- умножение матриц

Перебор с целью минимизировать число операций имеет экспоненциальную сложность.

На первом этапе определим за какое минимальное количество операций можно получить матрицу M из равенства (1).

Будем считать подзадачами вычисление минимального количества операций при перемножении меньшего, чем n, количества матриц.

В качестве параметров рассматриваемой задачи возьмем индексы i и j ($1 \le i \le j \le n$), обозначающие номера первой и последней матриц в цепочке $M_i \times M_{i+1} \times \ldots \times M_j$.

Сначала решим поздачи, когда j=i+1, т.е. когда перемножаются две рядом стоящие матрицы. Решения – количество затраченных операций, запишем в ячейке таблицы T с номерами (i. j).

T_{ij} будет содержать число, равное количеству операций при умножении цепочки матриц $M_i \times M_{i+1}$, где $1 \le i \le 3$.

Пример -- умножение матриц

Для примера из четырех матриц в таблице будут определены следующие элементы T: $t_{1,2}$, $t_{2,3}$ и $t_{3,4}$

0	10000		
	0	1000	
		0	5000
			0

$M_1 \times M_2 \times M_3 \times M_4$
[10×20] [20×50] [50×1] [1× 100]

Далее перейдем к решению подзадач с параметрами j=i+2. Рассмотрим, например, цепочку матриц $M_1 \times M_2 \times M_3$. Решением этой подзадачи будет минимальное количество операций, выбранное из двух возможных порядков перемножения матриц: $(M_1 \times M_2) \times M_3$ и $M_1 \times (M_2 \times M_3)$
При этом для выражений в скобках ответы уже записаны в таблице T Результат запишем в ячейку $T_{1,3}$
Затем перейдем к решению подзадач с параметрами j=i+3

Пример -- умножение матриц

Обозначим через t_{ij} минимальную сложность умножения цепочки матриц $M_i \times M_{i+1} \times \ldots \times M_j$, где $1 \le i \le j \le n$. Ее можно получить следующим образом:

$$t_{ij} = \begin{cases} 0, \quad если\ i = j \\ \min\limits_{i \le k < j} (t_{ik} + t_{k+1.j} + r_{ikj}), если\ j > i \end{cases}$$

Здесь t_{ik} — минимальная сложность вычисления цепочки $M' = M_i \times M_{i+1} \times \ldots \times M_k$, а $t_{k+1,j}$ — минимальная сложность умножения цепочки $M'' = M_{k+1} \times M_{k+2} \times \ldots \times M_j$.
Третье слагаемое r_{ikj} равно сложности умножения M' на M″. Утверждается, что t_{ij} ($j > i$) — наименьшая из сумм этих трех членов по всем возможным значениям k, лежащим между i и j - 1.

Упражнение

- Задана строка, состоящая из вещественных чисел, разделенных арифметическими операциями

- Требуется расставить в строке скобки таким образом, чтобы значение полученного выражения было максимальнвым

-

Кратчайшие пути между всеми парами вершин

Строим матрицу стоимостей:

$$M[i, j] = \begin{cases} w(i, j), & \text{если ребро } (i, j) \in E \\ +\infty, & \text{если ребро } (i, j) \notin E \\ 0, & \text{если } i = j \end{cases}$$

Обозначим через d [i, j] матрицу кратчайших

путей между всеми вершинами.

Вершины занумеруем числами от 1 до n.

Алгоритм Флойда-Уоршолла

Обозначим через $d_{ij}^{(k)}$ стоимость кратчайшего пути из вершины с номером i в вершину с номером j с промежуточными вершинами из множества {1, 2, …, k}.

$$d_{ij}^{(k)} = \begin{cases} M[i, j], & \text{если } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & \text{если } k \geq 1 \end{cases}$$

$D^{(n)}$ содержит искомое решение

Алгоритм Флойда-Уоршолла

```
Floyd-Warshall(M, n) {
   D^(0) ← M;
   for (k = 0; k<n; k++)
    for (i = 0; i<n; i++)
        for j ← 1 to n do
            d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1));
   return D^(n);
}
```

Заключение

- Понятие динамического программирования
- Примеры
  - Сумма геометрической прогрессии
  - Суммирование набора
  - Задача о рюкзаке
  - Произведение матриц
  - Алгоритм Флойда-Уоршалла

На какое минимальное количество квадратов можно разложить число n?

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
dp==[0,1,2,3,1,2,3,4,2,1, 2, 3, 3, 4, 3, 4, 1, 2, 2 …]
```

```
dp[0] = 0;
for (int i = 1; i <= n; i++) {
   dp[i] = INT_MAX;
   for (int j = 1; j * j <= i; j++) {
    dp[i] = min(dp[i],dp[i-j*j] + 1);
   }
}
```

(j – размер квадрата)

Алгоритм Ахо (редакционное расстояние)

Пусть даны две строки S_1 и S_2. Необходимо за минимальное число *допустимых* операций преобразовать строку S_1 в строку S_2. Допустимой операцией являются следующие операции удаления символа из строки и вставки символа в строку:

DEL(S, i) – удалить i-ый элемент строки S;

INS(S, i, c) – вставить символ c после i-го элемента строки S.

Обозначим через S [i..j] – подстроку от i-го символа до j-го.

Пусть $M(i,j)$ – минимальное количество операций, которые требуются, чтобы преобразовать начальные i символов строки S_1 в начальные j символов строки S_2: $S_1[0..i]$ –> $S_2[0..j]$.

Считаем, что $S_1[0..0]$ и $S_2[0..0]$ – пустые строки.

Заметим, что для преобразования пустой строки в строку длины j требуется j операций вставки, т.е. $M(0, j) = j$.

Аналогично для преобразования строки длины i в пустую строку требуется i операций удаления, т.е. $M(i, 0) = i$.

Пусть мы решили подзадачу с параметрами $i-1$ и $j-1$.

Это означает, что из строки $S_1[0..i-1]$ построена строка $S_2[0..j-1]$ за минимальное число допустимых операций $M(i-1, j-1)$.

I) Пусть $S_1[i] = S_2[j]$. Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ не требуется никаких дополнительных операций. Следовательно, $M(i, j) = M(i-1, j-1)$.

II) Пусть $S_1[i] \neq S_2[j]$. Возможны два способа получения строки $S_2[0..j]$.

1. Пусть из строки $S_1[0..i–1]$ построена строка $S_2[0..j]$ за минимальное количество операций $M(i–1, j)$. Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ требуется удалить i-ый символ из строки S_1.
2. Пусть из строки $S_1[0..i]$ построена строка $S_2[0..j–1]$ за минимальное количество операций $M(i, j–1)$. Тогда для получения строки $S_2[0..j]$ из строки $S_1[0..i]$ потребуется одна операция вставки i-го символа строки S_1 после символа $S_2[j–1]$.

Из 2-х возможностей выбраем лучшую и получаем следующие рекуррентные соотношения:

$M(0, j) = j; \qquad M(i, 0) = i;$

$M(i, j) = \min(M(i-1, j-1), M(i-1, j) + 1, M(i, j-1) + 1),$

если $S_1[i] = S_2[j]$;

$M(i, j) = \min(M(i-1, j), M(i, j-1)) + 1,$

если $S_1[i] \neq S_2[j]$;

Решением задачи будет значение $M(m,n)$,
где m — длина строки S_1, а n — длина строки S_2.

Пример

$S_1 = ”abc”$, $S_2 = ”aabddc”$

Построим таблицу *M*, нумерация строк и столбцов которой начинается с нуля

и элементами которой будут числа, равные значениям функции, описанной

выше.

-1	0	1	2	3
c	6	5	4	**3**
d	5	4	3	4
d	4	3	2	3
b	3	**2**	1	2
a	2	1	2	3
a	1	0	1	2
	0	1	2	3
		a	*b*	*c*

Обратный ход

M[1,3] = 2, означает, что из строки "*a*" можно получить строку "*aab*", используя две допустимых операции. В примере за три допустимых операции можно преобразовать строку S_1 в S_2. Для определения операций нужно встать на последний символ строки S_1 и начать движение по таблице от правого верхнего угла. В примере движение начнется с ячейки *M*[3,6].

Находясь в ячейке *M*[*i*, *j*], будем рассматривать два случая.

1) Если *M*[-1, *i*] = *M*[*j*, -1], то будем сдвигаться по диагонали влево-вниз, попадая в ячейку *M*[*i*-1, *j*-1]. При этом будем перемещаться по строке S_1 на один символ влево, т.е. сделаем текущим в строке символ, находящийся в *i*-1 позиции.
2) Если *M*[-1, *i*] ≠ *M*[*j*, -1], то будем сдвигаться по таблице на одну позицию либо влево, попадая в ячейку *M*[*i*, *j*-1], либо вниз в ячейку *M*[*i*-1, *j*]. Этот выбор будет зависеть от того, какой из элементов, находящихся в этих ячейках, меньше. При движении влево будем удалять *i*-ый символ в строке S_1, перемещась на один символ влево. При движении вниз будем вставлять после *i*-го символа в строке S_1 символ $S_2[j]$.

Последовательность действий для примера

Изначально текущим в строке S_1 является последний символ –символ *c*.

Так как *M*[-1, 3] = *M*[6, -1], то осуществляем переход в ячейку *M* [5, 2] и текущим в S_1 становится предпослений символ – *b*.

Далее, так как *M*[-1, 2] ≠ *M*[5, -1], передвигаемся в ячейку *M*[4, 2]. При этом вставим после текущего символа *b* символ S_2 [5] = *d* (*j*=5).

Продолжая этот процесс вставим символ S_2 [4] = *d*, затем в строке S_1 сделаем текущим сивол *a*, вставим в строку S_1 символ *a*.

Процесс продолжается до тех пор, пока не достигнем ячейки *M*[0,0]. Для нашего примера последовательность операций будет следующая:

INS(S_1, 2, '*d*'), INS(S_1, 2, '*d*'), INS(S_1, 1, '*a*').

abc –> *abc* –> *abdc* –> *abddc* –> *abddc* –> *aabddc*

Отметим, что решений в данной задаче может быть несколько.

Движение по таблице представлено ниже.

↓	вниз по i-му столбцу из j-ой строки в j–1-ю	INS(S_1, i, $S_2[j]$)	вставка после i-й позиции в S_1 символа $S_2[j]$
←	движение влево по j-й строке из i-го столбца в i–1-й	DEL(S_1, i)	удаление i-го символа в S_1 и передвижение на i–1-ю позицию
↙	движение по диагонали влево вниз		перемещение текущей позиции в S_1 на один символ влево

Итак, t_{ij} вычисляются в порядке возрастания разностей нижних

индексов. Процесс начинается с вычисления t_{ii} для всех *i*, затем

$t_{i,i+1}$ для всех *i*, потом $t_{i,i+2}$ и т. д. При этом t_{ik} и $t_{k+1,j}$ будут уже вычислены, когда мы приступим к вычислению t_{ij}.

Оценка сложности данного алгоритма есть $O(n^3)$.

В результате работы алгоритма для примера из четырех матриц

будет построена следующая таблица *T*:

0	10000	1200	2200
	0	1000	3000
		0	5000
			0

Порядок, в котором можно произвести эти умножения, легко определить,

приписав каждой клетке то значение k, на котором достигается минимум.

Алгоритм

```
for (i=0; i<n; i++)    m_{i,i} = 0;
for (l=1; l<n; l++)
  for (i=0; i<n; i++) {
   j = i + l;
   for (k = 0; k < j; k++)
      m_{ij} = min(m_{i,k}+ m_{k+1,j} + r_{i-1}*r_k* r_j)
  }
```

r_{i-1} – количество строк в M'

r_k – количество столбцов в M'

r_j – количество столбцов в M''

Задача "Divisibility“ 1999-2000 ACM NEERC (подключена в системе тестирования NSUTS в школьных тренировках)

Consider an arbitrary sequence of integers. One can place **+** or **–** operators between integers in the sequence, thus deriving different arithmetical expressions that evaluate to different values.
Let us, for example, take the sequence: 17, 5, –21, 15.
There are eight possible expressions:

17+5+ – 21+15=16	17+5+–21–15=–14
17+5– –21+15=58	17+5– –21–15=28
17–5 + –21+15=6	17–5+–21–15=–24
17–5– –21+15=48	17–5– –21–15=18

We call the sequence of integers **divisible** by *K* if + or – operators can be placed between integers in the sequence in such way that resulting value is divisible by *K*. In the above example, the sequence is divisible by 7 (17+5+–21–15=–14) but is not divisible by 5. You are to write a program that will determine divisibility of sequence of integers.
The first line of the input file contains two integers, *N* and *K* ($1 \le N \le 10000$, $2 \le K \le 100$) separated by a space.
The second line contains a sequence of *N* integers separated by spaces. Each integer is not greater than 10000 by its absolute value.

Задача "Gangsters" (подключена в системе тестирования NSUTS в школьных тренировках)

N gangsters are going to a restaurant. The i-th gangster comes at the time T_i and has the *prosperity* P_i. The door of the restaurant has K+1 *states of openness* expressed by the integers in the range [0, K]. The state of openness can change by one in one unit of time; i.e. it either opens by one, closes by one or remains the same. At the initial moment of time the door is closed (state 0). The i-th gangster enters the restaurant only if the door is opened specially for him, i.e. when the state of openness coincides with his *stoutness* S_i. If at the moment of time when the gangster comes to the restaurant the state of openness is not equal to his stoutness, then the gangster goes away and never returns.
The restaurant works in the interval of time [0, T].
The goal is to gather the gangsters with the maximal total prosperity in the restaurant by opening and closing the door appropriately.

The first line of the input file contains the values N, K, and T, separated by spaces. ($1 \le N, K \le 100$). he second line of the input file contains the moments of time when gangsters come to the restaurant T_1, T_2, ..., T_N, separated by spaces. The third line of the input file contains the values of the prosperity of gangsters P_1, P_2, ..., P_N, separated by spaces. The forth line of the input file contains the values of the stoutness of gangsters S_1, S_2, ..., S_N, separated by spaces. All values in the input file are integers.

Пример

t = 1 2 3 4 5 6

S = 1 2 3 4 5 1

P = 1 1 1 1 1 100

L

– состояние двери

4					**4**	**5**	**5**
3				**3**	**3**	**4**	**5**
2			**2**	**2**	**3**	**3**	**4**
1		**1**	**1**	**2**	**2**	**3**	**103**
0	**0**	**0**	**1**	**1**	**2**	**2**	**3**
0	**0**	**1**	**2**	**3**	**4**	**5**	**6**

← гангстеры

$$m_{i,j} = \max \{ [m_{i-1,j-1}, m_{i-1,j}, m_{i-1,j+1}] + f_i \}$$

где

$$f_i = \begin{cases} p_i, & \text{если } L = s_i \\ 0 & \end{cases}$$

КОНЕЦ ЛЕКЦИИ

Разбиение чисел

Разбиением называется представление натурального числа в виде суммы натуральных слагаемых, а сами слагаемые — *частями разбиения*. Порядок слагаемых не играет роли. Будем записывать разбиения, перечисляя их части через запятую в невозрастающем порядке. Например, разбиение 4=2+1+1 записывается как (2, 1, 1).

Пусть $p(n)$ обозначает количество всех разбиений натурального числа n. Например, $p(5) = 7$, $p(100) = 190\ 569\ 292$.

$p(100)$ было известно ещё в XIX веке.

Задача вычисления $p(n)$ имеет почтенный возраст. Впервые она была сформулирована Лейбницем в 1654 году, а в 1740 — предложена немецким математиком Филиппом Ноде Леонарду Эйлеру.

Занимаясь разбиениями, Эйлер открыл целый ряд их свойств, среди которых главное место занимала знаменитая «пентагональная теорема». С исследований Эйлера начинается история теории разбиений, в развитии которой принимали участие крупнейшие математики последующих поколений.

Исследования Эйлера

Изучение функции $p(n)$ Эйлер начинает с рассмотрения бесконечного произведения

$(1 + x + x^2 + ...)(1 + x^2 + x^4 + ...) ... (1 + x^k + x^{2k} + ...) ...$

Каждый член произведения получается в результате умножения мономов, взятых по одному из каждой скобки. Если в первой скобке взять x^{m1}, во второй — x^{2m2} и т.д., то их произведение будет равно $x^{m1+2m2+3m3+...}$. Значит, после раскрытия скобок получится сумма сумма мономов вида $x^{m1+2m2+3m3+...}$

Сколько раз в этой сумме встретится x^n? Столько, сколькими способами можно представить n как сумму $m1 + 2m2 + 3m3 + ...$ Каждому такому представлению отвечает разбиение числа n на $m1$ единиц, $m2$ двоек и т.д. Так получаются все разбиения, так как каждое из них, конечно, состоит из нескольких единиц, нескольких двоек и т.д. Поэтому коэффициент при x^n равен числу разбиений $p(n)$.

$d(n) = l(n)$ (теорема Эйлера)

Обозначим через $d(n)$ количество разбиений числа n на различные слагаемые, а через $l(n)$ — на нечётные. Например, среди выписанных выше разбиений числа 5 различные части имеют (5), (4, 1) и (3, 2), а нечётные — (5), (3, 1, 1) и (1, 1, 1, 1, 1). Значит, $d(5) = l(5) = 3$.

Тогда:

$d(0) + d(1)\,x + d(2)\,x^2 + d(3)\,x^3 + \ldots = (1 + x)(1 + x^2)(1 + x^3) \ldots,$

$l(0) + l(1)\,x + l(2)\,x^2 + l(3)\,x^3 + \ldots = 1\,(1 - x)(1 - x^3)(1 - x^5) \ldots$

.

Изучая $p(n)$, Эйлер сосредоточил внимание на произведении $(1–x)(1–x^2)(1–x^3)$... Раскрывая в нём скобки, Эйлер получил удивительный результат:

$(1 - x)(1 - x^2)(1 - x^3) \ldots = 1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + x^{22} + x^{26} - x^{35} - x^{40} + \ldots$

Показатели в правой части — *пятиугольные числа*, т.е. числа вида $(3q^2 \pm q)/2$, а знаки при соответствующих мономах равны $(–1)^q$.

Исходя из этого наблюдения, Эйлер предположил, что должна быть верна следующая теорема

Пентагональная теорема:

$$\prod_{k=1}^{\infty}\left(1-x^{k}\right)=\sum_{q=-\infty}^{\infty}(-1)^{q}x^{(3q^{2}+q)/2}$$

Используя ее:

$(p(0) + p(1)x + p(2)x^2 + \ldots)(1 - x - x^2 + x^5 + x^7 - x^{12} - x^{15} + \ldots) = 1.$

формула Эйлера, позволяющую последовательно находить числа $p(n)$:

$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + \ldots$

$+ (-1)^{q+1}(p(n-(3q^2 - q)/2) + p(n-(3q^2 + q)/2))$

Решение (динамика)

1. 1 1 1 1 //исходный массив
2. 1 1 2
3. 1 3
4. 2 2
5. 4

```
const nmax=120;
procedure Summ(N:integer);
var List : array [0..nmax] of byte;{вспомогательный массив для хранения значений
      слагаемых}
CountVariants : longint;{количество вариантов}
procedure Generate(k, Count, max:longint);
{номер элемента, количество,максимальное начение=числу}
begin    {Текущее разложение}
inc(CountVariants);  {первое разложение на единицы}
while (List[k] < max) and (k < (Count-1)) do
 {пока значение элемента меньше числа и его номер меньше количества элементо-1}
       begin   dec(Count); inc(List[k]);    {уменьшаем размер, переходим в следующий разряд
                                        влево, сумма не изменяется}
              Generate(k+1, Count, List[k]);   {генерируем следующее  разложение}
     end;
List[k] := 1;   {снова в правую крайнюю ячейку}
end;
begin    if (N < 1) or (N > nmax) then exit;
 FillChar(List, sizeOf(List), 1);   {заполняем массив единицами}
CountVariants := 0;
Generate(0, N, N);    {генерируем разбиения}
 WriteLn('Всего вариантов: ', CountVariants); end;
var N:integer;
 begin readln(N);   Summ(N);  end.
```

x(m) разбиений натурального числа m

Для решения исходной задачи перейдем к рассмотрению обобщенной задачи. Подсчитаем количество $P(m,n)$ разбиений натурального числа m со слагаемыми, не превосходящими n. Ясно, что x(m)=$P(m,m)$.

1) $P(m,1)=1$ – существует только одно разбиение m, в котором
слагаемые не превосходят единицы, а именно: $m=1+1+\ldots+1$.

2) $P(1,n)=1$ – число 1 имеет одно представление при любом n.

3) $P(m,n)=P(m,m)$ при $n>m$ - слагаемых, больших m, в разбиениях нет

4) $P(m,m)=P(m,m-1)+1$ - существует лишь одно разбиение со
слагаемым, равным m. Все иные разбиения имеют слагаемые, не
превосходящие m-1.

5) $P(m,n)=P(m,n-1)+P(m-n,n)$ ($n<m$). (см. следующий слайд)

$P(m,n) = \underline{P}(m,n\text{-}1) + P(m\text{-}n,n)$ $(n<m)$

Все разбиения *m* на сумму слагаемых, не превосходящих *n*, можно разбить на два непересекающихся класса:

- суммы, не содержащие *n* в качестве слагаемого,
- суммы, содержащие *n*.

Количество элементов первого класса равно $P(m,n\text{-}1)$.

Количество элементов второго класса:

без учета слагаемого *n* суммы элементов второго класса равны $m\text{-}n$. Значит, их общее количество равно $P(m\text{-}n,n)$ и, следовательно, общее количество элементов второго класса также равно этой величине.

$P(5,5) = P(5,4\ ) + P(1,5) = P(5,4) + 1;$

$P(5,4) = P(5,3) + P(1,4) = 5 + 1.$

Задача о телефонном номере (подключена в системе тестирования NSUTS в школьных тренировках)

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MN
7 PRS	8 TUV	9 WXY
	0 OQZ	

Если вы обратили внимание, то клавиатура многих телефонов выглядит как показано –> Использование изображенных на клавишах букв позволяет представить номер телефона в виде легко запоминающихся слов. Многие фирмы пользуются этим и стараются подобрать себе номер телефона так, чтобы он содержал как можно больше букв из имени фирмы.

Напишите программу, которая преобразует исходный цифровой номер телефона в соответствующую последовательность букв и цифр, содержащую как можно больше символов из названия фирмы. При этом буквы из названия фирмы должны быть указаны в полученном номере в той же последовательности, в которой они встречаются в названии фирмы. Например, если фирма называется *IBM*, а исходный номер телефона — **246**, то замена его на ***BIM*** не допустима, тогда как замена его на **2*IM*** или ***B*4*M*** является правильной.

S_1= “IBM”, S_2= “246”. При этом, если в S_1 встречаются буквы, которые соответствуют цифрам номера телефона в нужном порядке, то они останутся без изменения.

Формат входных данных:

Первая строка входного файла содержит название фирмы. Она состоит только из заглавных букв латинского алфавита, количество которых не превышает 80 символов. Вторая прока содержит номер телефона в виде последовательности цифр. Цифр в номере телефона также не более 80.

Формат выходных данных:

В единственной строке выходного файла должно содержаться число букв из измененного номера.

Пример файла входных данных:

IBM

246

Пример файла выходных данных:

2