Март 12, 2022

Задача A. Скучная лекция
Задача B. Подсчет треугольников
Задача C. Последовательности
Задача D. Разрезание торта
Задача E. Анти палиндромные строки
Задача F. Рекурсивная последовательность
Задача G. a^b^c
Задача H. Элегантно переставленная сумма
Задача I. Задача сапожника
Задача J. Велосипед

4859. Скучная лекция

Леша сидел на лекции. Ему было невероятно скучно. Голос лектора казался таким далеким и незаметным...

Чтобы окончательно не уснуть, он взял листок и написал на нем свое любимое слово. Чуть ниже он повторил свое любимое слово, без первой буквы. Еще ниже он снова написал свое любимое слово, но в этот раз без двух первых и последней буквы.

Тут ему пришла в голову мысль – времени до конца лекции все равно еще очень много, почему бы не продолжить выписывать всеми возможными способами это слово без какой-то части с начала и какой-то части с конца?

После лекции Леша рассказал Максу, как замечательно он скоротал время. Максу стало интересно посчитать, сколько букв каждого вида встречается у Леши в листочке. Но к сожалению, сам листочек куда-то запропастился.

Макс хорошо знает любимое слово Леши, а еще у него не так много свободного времени, как у его друга, так что помогите ему быстро восстановить, сколько раз Леше пришлось выписать каждую букву.

Вход. Одна строка, состоящая из строчных латинских букв – любимое слово Леши. Длина строки лежит в пределах от 5 до 10^5 символов.

Выход. Для каждой буквы на листочке Леши, выведите ее, а затем через двоеточие и пробел сколько раз она встретилась в выписанных Лешей словах (как показано в примерах). Буквы должны следовать в алфавитном порядке. Буквы, не встречающиеся на листочке, выводить не нужно.

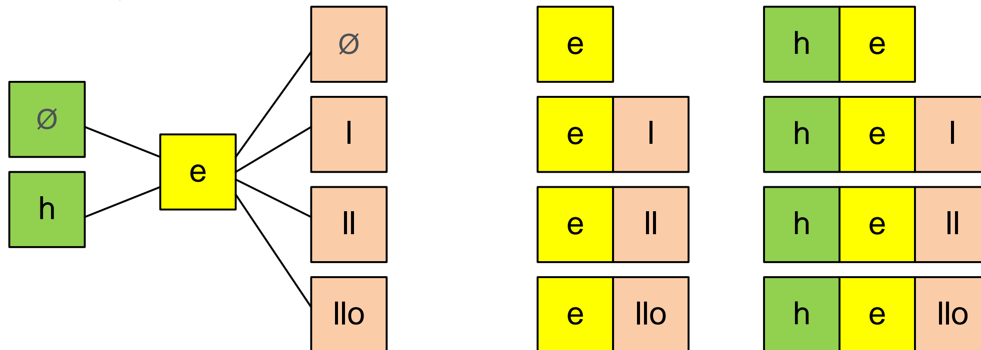
Пример входа 1	**Пример выхода 1**

`hello`	`e: 8`
	`h: 5`
	`l: 17`
	`o: 5`

Пример входа 2	**Пример выхода 2**

`abacaba`	`a: 44`
	`b: 24`
	`c: 16`

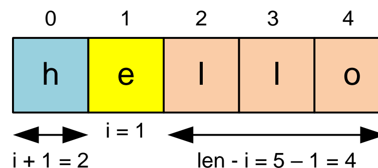
Возьмем слово ***hello*** и подсчитаем, сколько раз в выписанных словах встречается например буква ***e***. После буквы ***e*** могут находиться следующие подстроки (суффиксы): ∅ (пустая подстрока), *l*, *ll*, *llo* (4 подстроки). Перед буквой ***e*** могут находиться подстроки (префиксы) ∅, *h* (2 подстроки). Соединив любой префикс с любым суффиксом (и использовав один раз букву *e*), мы получим все возможные слова, выписанные Лешей.



Буква ***e*** будет выписана 2 * 4 = 8 раз.

Для каждой буквы следует подсчитать, сколько подстрок (включая пустую) может находиться до нее (пусть будет *a* подстрок), и сколько подстрок (включая пустую) может находиться после нее (пусть будет *b* подстрок). Тогда эта буква во всех выписанных словах будет встречаться $a * b$ раз.

Пусть *s* – входная строка, нумерация позиций начинается с нуля. Пусть *len* – длина строки. Тогда перед буквой s[*i*] может находиться $(i + 1)$ подстрока. После буквы s[*i*] может находиться $(len - i)$ подстрок. Всего имеется $(i + 1) * (len - i)$ слов, содержащих букву s[*i*].



Реализация алгоритма

Входную строку храним в массиве s. Количество каждой буквы в выписанных словах подсчитываем в массиве m.

```
#define MAX 100010
char s[MAX];
```

```
long long m[27];
```

Читаем входную строку s.

```
gets(s);
memset(m, 0, sizeof(m));
len = strlen(s);
```

Для каждой буквы s[i] подсчитываем сколько раз она встречается в выписанных Лешей словах.

```
for (i = 0; i < len; i++)
  m[s[i] - 'a'] += 1LL * (i + 1) * (len - i);
```

Выводим ответ: для каждой буквы печатаем количество раз, которое оно встречается в словах.

```
for (i = 0; i < 26; i++)
  if (m[i] > 0) printf("%c: %lld\n", i + 'a', m[i]);
```

8616. Подсчет треугольников

Имеется n стержней с длинами 1, 2, …, n. Вы можете выбрать любые три из них и построить треугольник. Сколько различных треугольников можно построить? Два треугольника считаются различными, если у них есть как минимум одна пара сторон с разными длинами.

Вход. Каждая строка содержит значение n ($3 \le n \le 10^6$). Последняя строка содержит $n < 3$ и не обрабатывается.

Выход. Для каждого теста выведите в отдельной строке количество различных треугольников, которое можно построить.

Пример входа	**Пример выхода**
5	3
8	22
0	

Обозначим через T(n) количество различных треугольников, которое можно построить из n стержней. Вычислим некоторые значения функции:

- T(3) = 0, из стержней с длинами 1, 2 и 3 составить треугольник нельзя;
- T(4) = 1, треугольник (2, 3, 4);
- T(5) = 3, треугольники (2, 3, 4), (2, 4, 5), (3, 4, 5);
- T(6) = 7, треугольники (2, 3, 4), (2, 4, 5), (3, 4, 5), (4, 5, 6), (3, 5, 6), (2, 5, 6), (3, 4, 6);

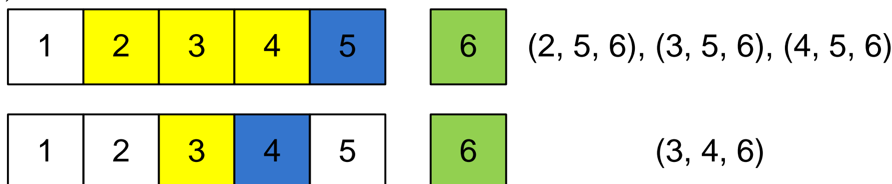
Очевидно, что T(n) включает в себя все тройки T(n – 1). Будем рассматривать множество T(n) как множество T(n – 1) плюс все тройки, которые содержат стержень длины n (назовем это множество S(n)). Например

$$T(6) = T(5) \cup \{ (4, 5, 6), (3, 5, 6), (2, 5, 6), (3, 4, 6) \}$$

Рассмотрим, какие числа кроме n содержит множество S(n). Например, если тройка из S(n) содержит n – 1, то вместе с n – 1 она может содержать числа n – 2, n – 3, …, 2 (сумма n – 1 и любого из этих чисел больше n, в результате чего тройка может образовывать треугольник). Если тройка из S(n) содержит n – 2, то вместе с n – 2 она может содержать числа n – 3, n – 4, …, 3. И так далее.

Рассмотрим множество S(6):

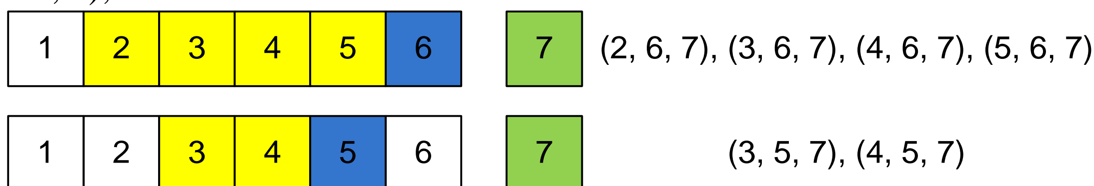
- Вместе с числом 5 могут быть числа 2, 3, 4 – образуя тройки (2, 5, 6), (3, 5, 6), (4, 5, 6);
- Вместе с числом 4 может быть только число 3 – образуется тройка (3, 4, 6);



Поскольку S(6) содержит 1 + 3 = 4 элемента, то T(6) = T(5) + |S(6)| = 3 + 4 = 7.

Рассмотрим множество S(7):

- Вместе с числом 6 могут быть числа 2, 3, 4, 5 – образуя тройки (2, 6, 7), (3, 6, 7), (4, 6, 7), (5, 6, 7);
- Вместе с числом 5 могут быть числа 3, 4 – образуя тройки (3, 5, 7), (4, 5, 7);



Поскольку S(7) содержит 2 + 4 = 6 элементов, то T(7) = T(6) + |S(7)| = 7 + 6 = 13.

Проводя подобные исследования, можно заметить что

- при четном n значение S(n) содержит 1 + 3 + 5 + … + (n – 3) = $\frac{n-2}{2}\left(\frac{n}{2}-1\right)$ элементов;
- при нечетном n значение S(n) содержит 2 + 4 + 6 + … + (n – 3) = $\frac{n-1}{2}\cdot\frac{n-3}{2}$ элементов;

Таким образом получим рекуррентность:

- при четном n имеем: $T(n) = T(n-1) + \frac{n-2}{2}\left(\frac{n}{2}-1\right)$;
- при нечетном n имеем: $T(n) = T(n-1) + \frac{n-1}{2} \cdot \frac{n-3}{2}$;

Значение $T(n)$ можно также выразить в явном виде:

$$T(n+1) = \begin{cases} T(n) + (k-1)k, n = 2k \\ T(n) + k \cdot k, n = 2k+1 \end{cases} \text{ или } T(n) = \begin{cases} \frac{1}{24} n(n-2)(2n-5), n \text{ четно} \\ \frac{1}{24}(n-1)(n-3)(2n-1), n \text{ нечетно} \end{cases}$$

Последовательность $T(n)$ имеет производящую функцию

$$f(x) = \frac{x^4}{(1-x)^3(1-x^2)} = x^4 + 3x^5 + 7x^6 + 13x^7 + \ldots$$

Пример

n	1	2	3	4	5	6	7	8	9	10
T(n)	0	0	0	1	3	7	13	22	34	50

Реализация алгоритма

Объявим массив для запоминания.

```
#define MAX 1000001
long long T[MAX];
```

Основная часть программы. Заполним массив T.

```
memset(T,0,sizeof(T));
for(i = 4; i < MAX; i++)
  if (i % 2 == 0) T[i] = T[i-1] + (i - 2) / 2 * (i / 2 - 1);
  else T[i] = T[i-1] + (i - 1) * (i - 3) / 4;
```

Обрабатываем последовательно запросы.

```
while(scanf("%lld",&n), n >= 3)
  printf("%lld\n",T[n]);
```

Реализация алгоритма – формула

Реализация функции $T(n)$.

```
long long T(long long n)
{
  if (n % 2) return (n - 1) * (n - 3) * (2 * n - 1) / 24;
  return n * (n - 2) * (2 * n - 5) / 24;
}
```

Основной цикл программы.

```
while(scanf("%lld",&n), n >= 3)
  printf("%lld\n",T(n));
```

8551. Последовательности

Вычислите количество неубывающих последовательностей длины n, содержащих целые числа от 1 до m, где каждый элемент встречается не более k раз.

Вход. Три целых числа n, m и k ($0 < n, m, k < 31$).

Выход. Выведите количество описанных последовательностей.

Пример входа	**Пример выхода**
`3 4 2`	`16`

Последовательностями будут: (1,1,2), (1,1,3), (1,1,4), (1,2,2), (1,2,3), (1,2,4), (1,3,3), (1,3,4), (1,4,4), (2,2,3), (2,2,4), (2,3,3), (2,3,4), (2,4,4), (3,3,4), (3,4,4).

Пусть f(n, m) равно количеству искомых последовательностей длины n, содержащие числа от 1 до m, где каждый элемент встречается не более k раз.

$$f(n, m) = f(n, m-1) + f(n-1, m-1) + \ldots + f(n-k, m-1)$$

Число m в последовательности может:

- отсутствовать, тогда следует строить f(n, m – 1) последовательностей;
- стоять в последней позиции. На n – 1 первых позициях строим f(n – 1, m – 1) последовательностей;
- стоять в двух последних позициях. На n – 2 первых позициях строим f(n – 2, m – 1) последовательностей;
- и так далее
- стоять в k последних позициях. На n – k первых позициях строим f(n – k, m – 1) последовательностей;

Базовые случаи:

f(0, m) = 1, в этом случае числа на всех n позициях уже фиксированы.

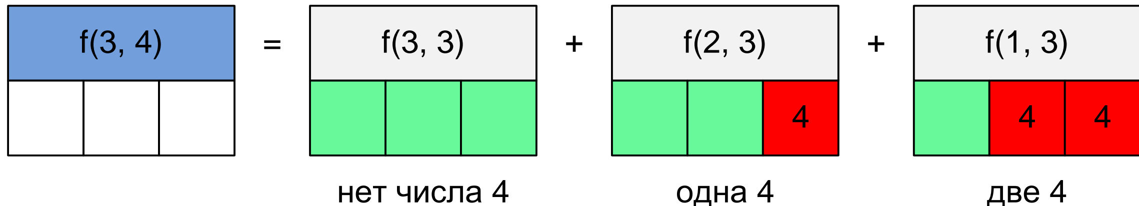
f(n, 0) = 0, нет возможности построить ни одной последовательности.

f(n, m) = 0 при $n < 0$.

f(n, m) = 0 при $m * k < n$. Максимум последовательность может содержать k единиц, k двоек, k троек, …, k m-ок. Если длина последовательности n больше $m * k$, то такой последовательности не существует.

Пример

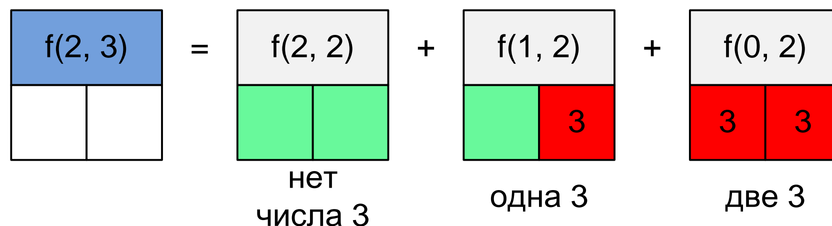
Пусть $n = 3$, $m = 4$, $k = 2$. Тогда



f(3, 3): генерируем последовательности длины 3 из чисел 1, 2, 3.

f(2, 3): генерируем последовательности длины 2 из чисел 1, 2, 3. На третьей позиции стоит 4.

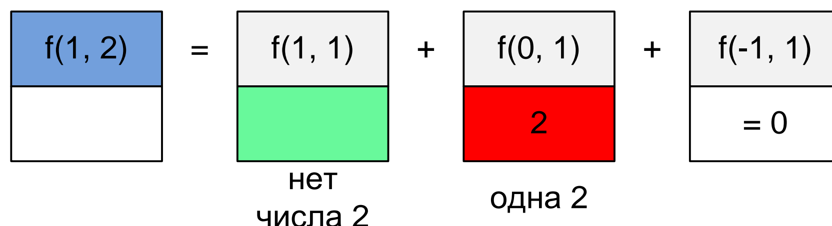
f(1, 3): генерируем последовательности длины 1 из чисел 1, 2, 3. На второй и третьей позиции стоит 4.



f(2, 3): в двух позициях могут стоять числа 1, 2, 3.

- f(2, 2): в двух позициях могут стоять числа 1, 2. Это последовательности (1, 1), (1, 2), (2, 2). Следовательно f(2, 2) = 3.
- f(1, 2): в одной позиции могут стоять числа 1, 2. Во второй позиции стоит 3. Это последовательности (1, 3), (2, 3). Следовательно f(1, 2) = 2.
- f(0, 2): все позиции заняты. Это последовательность (3, 3). Следовательно f(0, 2) = 1.

f(2, 3) = f(2, 2) + f(1, 2) + f(0, 2) = 3 + 2 + 1 = 6.



f(1, 2): в одной позиции могут стоять числа 1, 2.

Это последовательности (1) и (2). Следовательно f(1, 2) = 2.

Реализация алгоритма

Объявим массив для запоминания динамики: mas[n][m] = f(n, m).

```
long long mas[32][32];
```

Рекурсивная функция вычисления f(*n*, *m*).

```
long long f(int n, int m)
{
  if (n == 0) return 1;
  if (n < 0 || m == 0) return 0;
  if (m * k < n) return 0;

  if (mas[n][m] != -1) return mas[n][m];

  long long res = 0;
  for(int i = 0; i <= k; i++)
    res = res + f(n-i,m-1);
  return mas[n][m] = res;
}
```

Основная часть программы. Читаем входные данные и выводим ответ.

```
scanf("%d %d %d",&n,&m,&k);
memset(mas,-1,sizeof(mas));
printf("%lld\n",f(n,m));
```

1511. Разрезание торта

Имеется прямоугольный торт длины *length* и ширины *width*. Мы хотим разрезать его на *pieces* прямоугольных кусков равной площади. Каждый разрез должен совершаться параллельно сторонам торта, и должен полностью разрезать один из имеющихся кусков на две части. (Для разрезания торта на *n* кусков необходимо совершить *n* – 1 разрез)

Квадратные куски Вы предпочитаете тем, которые имеют большее отношение сторон. Под “отношением сторон” будем понимать отношение длины большей стороны куска к меньшей. Вам следует разрезать торт таким образом, чтобы минимизировать максимальное значение отношения сторон полученных кусков.

Например, если мы хотим разрезать торт 2x3 на шесть кусков, то это можно сделать, разрезав его на шесть кусков размера 1x1. Отношение сторон каждого куска равно 1.0, что является наименьшим возможным. Поэтому решение оптимально.

Один из возможных вариантов разрезать торт 5x5 на 5 кусков состоит в следующем: сначала разрезаем торт на две части размерами 2x5 и 3x5. Меньшую часть делим пополам (получаем две части размером 2 x 5/2), а большую часть делим на три части (каждая имеет размер 3 x 5/3). Большее отношение сторон достигается на куске 3 x 5/3 и равно 3/(5/3) = 1.8. Разделить торт на 5 частей равной площади с меньшим отношением сторон, нежели 1.8 невозможно.

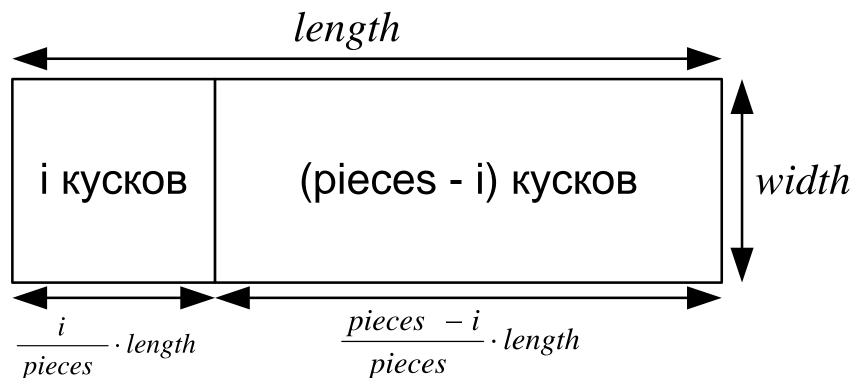
Вход. Состоит из нескольких тестов, каждый из которых задается в одной строке и содержит три целых числа: длину *length* и ширину *width* торта, а также

количество прямоугольных кусков *pieces*, на которое следует разрезать торт. Известно, что $1 \le length, width \le 1000$, $1 \le pieces \le 10$.

Выход. Следует разрезать торт так, чтобы минимизировать максимальное значение отношения сторон полученных кусков. Для каждого теста вывести в отдельной строке полученное отношение сторон с 4 десятичными цифрами. Помните, что все полученные куски должны иметь одинаковую площадь!

Пример входа	**Пример выхода**
`2 3 6`	`1.0000`
`5 5 5`	`1.8000`
`950 430 9`	`1.2573`

Благодаря верхнему ограничению на число кусков *pieces*, задача может быть просто решена полным перебором разрезаний торта. Если кусок размером *length* на *width* следует разбить на *pieces* кусков, то после совершения первого разреза (который по условию задачи можно совершить либо по горизонтали, либо по вертикали) площади двух полученных кусков должны быть пропорциональны числам 1 и *pieces* – 1, или 2 и *pieces* – 2 и так далее. То есть после первого разреза должны получаться два куска размером *i* * *length* / *pieces* на *width* и ((*pieces* – *i*) * *length* / *pieces* на *width*, или *length* на *i* * *width* / *pieces* и *length* на (*pieces* – *i*) * *width* / *pieces*, где $1 \le i < pieces$. При этом дальше первый кусок следует делить на *i* кусков, а второй на *pieces* – *i* кусков.



Совершаем разрез исходного куска на две части и далее рекурсивно запускаем разрезание каждой из полученных частей. Ищем такое разрезание, при котором максимум отношения кусков является наименьшим.

Реализация алгоритма

Функция cut возвращает наименьшее возможное максимальное значение отношения сторон полученных *pieces* кусков в результате разрезания прямоугольника длины *length* и ширины *width*.

```
double cut(double length, double width, int pieces)
{
  double temp, res = 1e100;
  int i;
```

Если *pieces* равно 1, то возвращаем отношение сторон текущего куска.

```
  if (pieces == 1) return max(length,width) / min(length,width);
```

Совершаем вертикальный разрез торта на две части, каждая из которых дальше будет делиться на *i* и *pieces* – *i* кусков.

```
  for(i = 1; i < pieces; i++)
  {
    temp = max(cut(i * length / pieces,width,i),
               cut((pieces - i) * length / pieces,width,pieces - i));
    if (temp < res) res = temp;
  }
```

Совершаем горизонтальный разрез торта на две части, каждая из которых дальше будет делиться на *i* и *pieces* – *i* кусков.

```
  for(i = 1; i < pieces; i++)
  {
    temp = max(cut(length,i * width / pieces,i),
               cut(length,(pieces - i) * width / pieces,pieces - i));
    if (temp < res) res = temp;
  }
  return res;
}
```

Основной цикл программы.

```
while(scanf("%d %d %d",&length, &width, &pieces) == 3)
{
  double res = cut(length,width,pieces);
  printf("%.4lf\n",res);
}
```

Временная оценка работы алгоритма

Пусть f(*pieces*) – функция, которая возвращает количество вызовов функции cut в зависимости от значения *pieces*.

Очевидно, что f(1) = 1, так как в этом случае сразу после вызова функции выйдем по команде return.

При *pieces* = 2 первый вызов функции cut будет со значением *pieces* = 2. Далее кусок будем стараться разделить пополам вертикальным разрезом, в результате чего дважды будет вызвана f(1). Потом попробуем совершить горизонтальный разрез, снова будет дважды вызвана f(1). Итого получим f(2) = 5 вызовов функции cut.

Пусть *pieces* = 3. Первый вертикальный разрез разобьет кусок на две части, первый из которых далее надо будет делить на 1 часть, а второй на 2 части. Второй вертикальный разрез также разобьет кусок на две части, первый из которых далее надо будет делить на 2 части, а второй на 1 часть. Аналогичное количество вызовов функции следует произвести при горизонтальных разрезах. Итого

$$f(3) = 1 + 2 * ((f(1) + f(2)) + (f(2) + f(1))) = 1 + 2 * 2 * (1 + 5) = 25$$

В общем случае $f(n) = 1 + 2\sum_{i=1}^{n-1}\left(f(i) + f(n-i)\right) = 1 + 4\sum_{i=1}^{n-1} f(i)$.

Например:

$$f(4) = 1 + 4\sum_{i=1}^{3} f(i) = 1 + 4 * (1 + 5 + 25) = 125,$$

$$f(5) = 1 + 4\sum_{i=1}^{4} f(i) = 1 + 4 * (1 + 5 + 25 + 125) = 625$$

Докажем методом математической индукции, что $f(n) = 5^{n-1}$.

База индукции. $f(1) = 5^0 = 1$.

Шаг индукции. $f(n) = 1 + 4\sum_{i=1}^{n-1} f(i) = 1 + 4\,(5^0 + 5^1 + \ldots + 5^{n-2}) = 1 + 4\frac{1-5^{n-1}}{1-5} = 5^{n-1}$.

Временная оценка работы программы составляет $O(5^{pieces})$.

9616. Анти палиндромные строки

Даны два числа n и m. Посчитайте количество строк длины n, символы которых принадлежат алфавиту размера m, которые не содержат в качестве подстроки палиндромов длины больше единицы.

Вход. В первой строке записано целое число t – количество тестов. Каждый тест представляет собой строку, в которой записано два целых числа n и m ($1 \le n, m \le 10^9$).

Выход. Для каждого теста выведите требуемое количество по модулю $10^9 + 7$.

Пример входа	**Пример выхода**
2	1920
5 6	1620
6 5	

Если строка не содержат в себе подстроку – палиндром длины 2 или длины 3, то она и не содержат в себе подстроку – палиндром длины больше единицы.

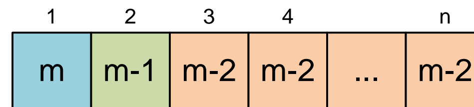
Если $n = 1$, то строка длины 1 может содержать любую из m букв. Искомое число строк равно m.

Если $m = 1$ и $n > 1$, то ответ 0, так как имеется единственная строка *aa..aa* и она содержит палиндром *aa*.

Если $n = 2$, то на первой позиции строки может находиться любая из m букв, а на второй позиции – любая из $m – 1$ букв (буквы в первой и второй позиции не

должны совпадать, образуя палиндром). Количество искомых строк равно (m * (m – 1)) mod $10^9 + 7$.

Пусть длина входной строки больше 2. На первой позиции может находиться любая из m букв, а на второй позиции – любая из m – 1 букв. Третья позиция должна отличаться от второй (вторая и третья буква не должны образовывать палиндром). Третья позиция должна отличаться от первой (первые три буквы не должны образовывать палиндром). Значит на третьей позиции может находиться любая из m – 2 букв.



Следуя этой логике, заключаем что буква на i-ой позиции должна отличаться от букв на позициях (i – 1) и (i – 2). Количество искомых строк равно (m * (m – 1) * $(m – 2)^{n-2}$) mod $10^9 + 7$.

Реализация алгоритма

Объявим модуль, по которому будут проходить вычисления.

```
#define MOD 1000000007
```

Функция ***powmod*** вычисляет значение x^n mod m.

```
long long powmod(long long x, long long n, long long m)
{
  if (n == 0) return 1;
  if (n % 2 == 0) return powmod((x * x) % m, n / 2, m);
  return (x * powmod(x, n - 1, m)) % m;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d", &tests);
while (tests--)
{
  scanf("%lld %lld", &n, &m);
```

Вычисляем ответ в зависимости от значений n и m.

```
  if (n == 1) res = m; else
  if (m == 1) res = 0; else // n > 1, m = 1:  aa....a
  if (n == 2) res = (m * (m - 1)) % MOD; else
  res = ((m * (m - 1)) % MOD * powmod(m - 2, n - 2, MOD)) % MOD;
```

Выводим ответ.

```
  printf("%lld\n", res);
}
```

4075. Рекурсивная последовательность

Последовательность a_i целых чисел задана следующим образом:

$a_i = b_i$ (для $i \le k$),

$a_i = c_1 a_{i-1} + c_2 a_{i-2} + ... + c_k a_{i-k}$ (для $i > k$),

где b_j и c_j ($1 \le j \le k$) – заданные целые числа. Для заданного n следует вычислить a_n и вывести его по модулю 10^9.

Вход. Первая строка содержит количество тестов t. Каждый тест состоит из четырех строк:

k – количество элементов (c) и (b) ($1 \le k \le 10$);

$b_1, ..., b_k$ ($0 \le b_j \le 10^9$) – k целых чисел, разделенных пробелом;

$c_1, ..., c_k$ ($0 \le c_j \le 10^9$) – k целых чисел, разделенных пробелом;

n – натуральное число ($1 \le n \le 10^9$).

Выход. В точности t строк, каждая из которых содержит значение a_n по модулю 10^9 для каждого теста.

Пример входа	**Пример выхода**
3	8
3	714
5 8 2	257599514
32 54 6	
2	
3	
1 2 3	
4 5 6	
6	
3	
24 354 6	
56 57 465	
98765432	

При $n \ge k$ воспользуемся соотношением:

$$\begin{pmatrix} a_n \\ a_{n-1} \\ ... \\ a_{n-k+1} \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & ... & c_{k-1} & c_k \\ 1 & 0 & ... & 0 & 0 \\ 0 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & 0 \end{pmatrix}^{n-k} \begin{pmatrix} b_k \\ b_{k-1} \\ ... \\ b_1 \end{pmatrix}$$

Возведение матрицы в степень реализуем за время $O(\log_2(n - k))$.

Если $n < k$, то ответ выведем из соответствующей ячейки массива b.

Реализация алгоритма

Объявим модуль MOD, по которому будут производиться все вычисления. Переопределим типы vi (вектор) и vvi (матрица).

```
#define MOD 1000000000
typedef vector<long long> vi;
typedef vector<vector<long long> > vvi;
```

Умножение матрицы a на матрицу b. Все вычисления производятся по модулю *mod*.

```
vvi mult(vvi a, vvi b, int mod)
{
  int i, j, k, s, n = a.size();
  vvi c(n, vi(n));
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
    {
      for(s = k = 0; k < n; k++) s = (s + a[i][k] * b[k][j]) % mod;
      c[i][j] = s;
    }
  return c;
}
```

Умножение матрицы a на вектор b.

```
vi mult(vvi a, vi b, int mod)
{
  int i, j, k, s, n = a.size();
  vi c(n, 0);
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      c[i] = (c[i] + a[i][j] * b[j]) % mod;
  return c;
}
```

Возведение матрицы a в степень *k*.

```
vvi pow(vvi a, int k, long long mod)
{
  int i, n = a.size();
  vvi res(n, vi(n, 0));
  for(i = 0; i < n; i++) res[i][i] = 1;
  while (k > 0)
  {
    if (k % 2) res = mult(res, a, mod);
    a = mult(a, a, mod);
    k /= 2;
  }
  return res;
}
```

Основная часть программы. Для каждого теста читаем входные данные.

```
scanf("%d",&tests);
while(tests--)
{
  scanf("%d",&k);
  b.assign(k,0);
```

Изначально положим $b = (b_k, b_{k\text{-}1}, \ldots, b_2, b_1)$.

```
  for(i = k - 1; i >= 0; i--) scanf("%lld",&b[i]);

  m.assign(k, vector<long long>(k,0));
```

Значения c_i считываем прямо в первую строку матрицы m. Заполняем остальные строки матрицы m.

```
  for(i = 0; i < k; i++) scanf("%lld",&m[0][i]);
  for(i = 1; i < k; i++) m[i][i-1] = 1;
```

Если $n \geq k$, то производим вычисления по указанной в анализе задачи формуле.

```
  scanf("%d",&n);
  if (n >= k)
  {
    m = pow(m, n - k, MOD);
    b = mult(m, b, MOD);
    printf("%lld\n",b[0]);
  }
```

Если $n < k$, то ответ берем из соответствующей ячейки массива b.

```
  else
    printf("%lld\n",b[k-n]);
}
```

9627. a^b^c

Вычислите значение

$$a^{b^c} mod(10^9 + 7)$$

Вход. Первая строка содержит количество тестов n. Каждая из следующих n строк содержит три числа a, b, c (a, b, $c \leq 10^9$).

Выход. Для каждого теста выведите в отдельной строке a^b^c (mod $10^9 + 7$).

Пример входа	**Пример выхода**
3	2187
3 7 1	50625
15 2 2	763327764
3 4 5	

По малой теореме Ферма $a^{p-1} = 1 \pmod p$, где p простое. Число $p = 10^9 + 7$ простое. Отсюда например следует что $a^{(p-1) * l} = 1 \pmod p$ для любого l.

Для вычисления выражения a^b^c сначала следует найти $k = b$^c, после чего вычислить a^k. Однако число b^c большое, представим его в виде b^$c = (p - 1) * l + s$ для некоторых l и $s < p - 1$. Тогда

$$a\text{^}(b\text{^}c) \bmod p = a^{(p-1) * l + s} \bmod p = (a^{(p-1) * l} * a^s) \bmod p = a^s \bmod p$$

Очевидно что $s = b$^$c \bmod (p - 1)$. Следовательно

$$a\text{^}(b\text{^}c) \bmod p = a\text{^}(b\text{^}c \bmod (p - 1)) \bmod p$$

Пример. Вычислим выражение 3^2^3 mod 7. Модуль 7 специально выбран простым. Значение выражения равно

$$3\text{^}(2\text{^}3) \bmod 7 = 3^8 \bmod 7 = 6561 \bmod 7 = (937 * 7 + 2) \bmod 7 = 2$$

Из теоремы Ферма следует что $3^6 \bmod 7 = 1$. Следовательно для любого натурального k

$$(3^6 \bmod 7)^k = 3^{6k} \bmod 7 = 1$$

Поскольку $2\text{^}3 = 2^3 = 8$, то $3^8 \bmod 7 = 3^{6 * 1 + 2} \bmod 7 = 3^2 \bmod 7 = 9 \bmod 7 = 2$

Исходное выражение также можно вычислить как

$$3\text{^}(2\text{^}3) \bmod 7 = 3^8 \bmod 7 = 3^{8 \bmod 6} \bmod 7 = 3^2 \bmod 7 = 9 \bmod 7 = 2$$

Реализация алгоритма

Функция ***powmod*** вычисляет значение $x^n \bmod m$.

```
long long powmod(long long x, long long n, long long m)
{
 if (n == 0) return 1;
 if (n % 2 == 0) return powmod((x * x) % m, n / 2, m);
 return (x * powmod(x, n - 1, m)) % m;
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d", &n);
while (n--)
{
  scanf("%lld %lld %lld", &a, &b, &c);
```

Вычисляем $res = b^c \bmod (p - 1)$.

```
  res = powmod(b, c, 1000000006LL);
```

Вычисляем $res = a^{res} \bmod p$.

```
  res = powmod(a, res, 1000000007LL);
```

Выводим ответ.

```
  printf("%lld\n", res);
}
```

1593. Элегантно переставленная сумма

Задана последовательность из n целых чисел $\{a_1, a_2, \ldots, a_n\}$. Необходимо найти такую ее перестановку, для которой сумма модулей разниц всех соседних элементов максимальна. Эту наибольшую сумму будем называть элегантной.

Рассмотрим, например, последовательность {4, 2, 1, 5}. Искомой является перестановка {2, 5, 1, 4}, а ее элегантная сумма равна |2 – 5| + |5 – 1| + |1 – 4| = 3 + 4 + 3 = 10. Для всех других 24 перестановок значение элегантной суммы не больше 10.

Вход. Первая строка содержит количество тестов t ($t < 100$). Каждая следующая строка является отдельным тестом. Каждая входная строка начинается числом n ($1 < n < 51$), за которым следует последовательность из n неотрицательных чисел. Каждое число в последовательности не более 1000.

Выход. Для каждого теста вывести его номер и значение элегантной суммы.

Пример входа

```
3
4 4 2 1 5
4 1 1 1 1
2 10 1
```

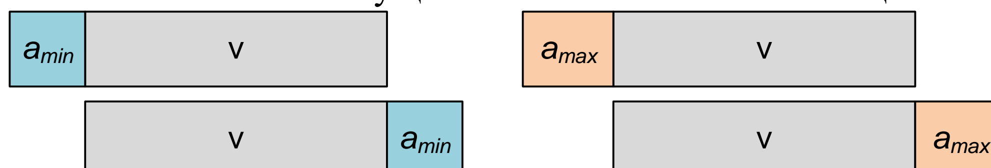
Пример выхода

```
Case 1: 10
Case 2: 0
Case 3: 9
```

Отсортируем числа входной последовательности a. Создадим новый массив v, в котором будем строить искомую перестановку. Изначально занесем в него минимальный и максимальный элементы последовательности a (и соответственно эти элементы удалим из a). Элегантную сумму будем подсчитывать в переменной s. Сначала положим $s = | v[0] – v[1] |$.

Пока массив a не пустой, жадным методом делаем наилучший выбор среди следующих четырех возможностей:

1. Наименьший элемент текущего массива a ставим в начало массива v.
2. Наименьший элемент текущего массива a ставим в конец массива v.
3. Наибольший элемент текущего массива a ставим в начало массива v.
4. Наибольший элемент текущего массива a ставим в конец массива v.



Для каждого случая пересчитываем новое значение s. Совершаем тот выбор, для которого новое значение s будет наибольшим. Для каждого теста в качестве ответа выводим значение s.

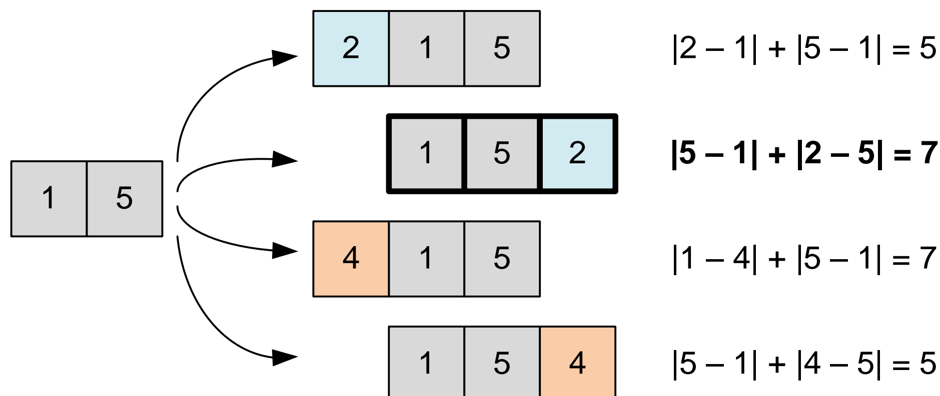
Поскольку массивы a и v обновляются динамически, в качестве контейнеров будем использовать двусторонние очереди.

Пример. Рассмотрим работу алгоритма на первом тесте. Отсортируем массив:

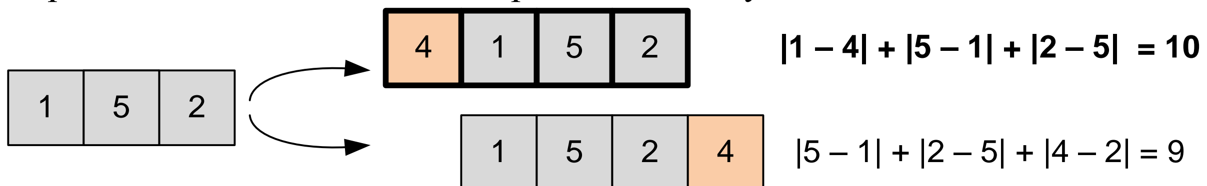
$$a = \{1, 2, 4, 5\}$$

Шаг 1. Занесем в массив v наименьший и наибольший элементы: $v = \{1, 5\}$. Удалим их из a, после чего $a = \{2, 4\}$.

Наименьший и наибольший элементы массива a приписываем справа и слева от массива v. Наибольшее значение суммы достигается, например, на массиве $\{1, 5, 2\}$.



Шаг 2. $v = \{1, 5, 2\}$, $a = \{4\}$. В массиве a остался один элемент. Приписываем его справа и слева от массива v. Пересчитываем суммы.



Искомая сумма равна 10, достигается она например на перестановке

$$\{4, 1, 5, 2\}$$

Реализация алгоритма

Объявим рабочие очереди.

```
deque<int> a, v;
```

Читаем количество тестов *tests*.

```
scanf("%d", &tests);
for (i = 1; i <= tests; i++)
{
```

Начинаем обработку следующего теста. Чистим содержимое массивов.

```
  scanf("%d", &n);
  a.clear(); v.clear();
```

Читаем входной массив a.

```
for (j = 0; j < n; j++)
{
  scanf("%d", &val);
  a.push_back(val);
}
```

Сортируем входной массив.

```
sort(a.begin(), a.end());
```

Заносим минимальный и максимальный элементы последовательности a в массив v.

```
v.push_back(a.back());
v.push_front(a.front());
```

Положим изначально $s = | v[1] – v[0] |$.

```
s = abs(v.back() - v.front());
```

Удаляем эти два элемента из массива a.

```
a.pop_back();
a.pop_front();
```

Пока массив a не пустой, рассматриваем 4 случая и делаем среди них оптимальный выбор жадным методом.

```
while (!a.empty())
{
```

Объявим целочисленный массив mx из четырех элементов.

```
  mx[0] = abs(v.front() - a.front());
  mx[1] = abs(v.back() - a.front());
  mx[2] = abs(v.front() - a.back());
  mx[3] = abs(v.back() - a.back());
  rmax = *max_element(mx, mx + 4);

  if (rmax == mx[0])
  {
    v.push_front(a.front());
    a.pop_front();
  } else
  if (rmax == mx[1])
  {
    v.push_back(a.front());
    a.pop_front();
  } else
  if (rmax == mx[2])
  {
    v.push_front(a.back());
    a.pop_back();
```

```
    } else
    {
      v.push_back(a.back());
      a.pop_back();
    }
    s += rmax;
  }
```

Выводим ответ.

```
  printf("Case %d: %d\n", i, s);
}
```

1591. Задача сапожника

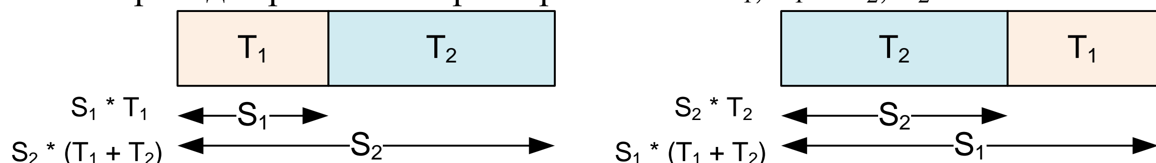
Сапожнику необходимо выполнить *n* работ. Каждый день сапожник может выполнять только одну работу. Для каждой *i* - ой работы известно время ее выполнения T_i и штраф S_i, который каждый день должен платить сапожник до тех пор, пока он не отдаст *i* - ую работу заказчику. Найти последовательность выполнения работ, при которой сумма штрафа будет минимальной.

Вход. Состоит из нескольких тестов. Первая строка каждого теста содержит количество работ *n* ($1 \le n \le 1000$), за которой следуют *n* строк, содержащие характеристики работ T_i ($1 \le T_i \le 1000$) и S_i ($1 \le S_i \le 10000$).

Выход. Для каждого теста в отдельной строке вывести порядок работ, при котором сумма штрафа минимальна. При существовании нескольких оптимальных порядков работ выводить лексикографически наименьший.

Пример входа	**Пример выхода**
4 3 4 1 1000 2 2 5 5	2 1 3 4

Рассмотрим две работы с характеристиками T_1, S_1 и T_2, S_2.



Если сначала будет выполняться первая работа, а потом вторая, то сумма штрафа составит

$$S_1 * T_1 + S_2 * (T_1 + T_2)$$

Если за второй будет выполняться первая работа, то в качестве штрафа следует заплатить

$$S_2 * T_2 + S_1 * (T_1 + T_2)$$

Рассмотрим, при каком условии сумма штрафа при выполнении работ 1, 2 лучше, нежели при выполнении работ в порядке 2, 1:

$$S_1 * T_1 + S_2 * (T_1 + T_2) < S_2 * T_2 + S_1 * (T_1 + T_2)$$

Раскроем скобки и приведем подобные:

$$S_2 * T_1 < S_1 * T_2$$

Или то же самое, что

$$\frac{T_1}{S_1} < \frac{T_2}{S_2}.$$

Пусть теперь имеется n работ. Если существуют i - ая и j - ая работы, для которых $T_i / S_i > T_j / S_j$, то поменяв их местами в последовательности выполнения, мы уменьшим общую сумму штрафа. Таким образом, для минимизации штрафа следует отсортировать работы по неубыванию отношения времени их выполнения к сумме штрафа.

В случае равенства отношения ($T_i / S_i = T_j / S_j$), работы следует сортировать по возрастанию их номеров.

Пример. Отсортируем работы по неубыванию отношения времени их выполнения к сумме штрафа:

номер работы	2	1	3	4
T_i	1	3	2	5
S_i	1000	4	2	5

$$\frac{1}{1000} \le \frac{3}{4} \le \frac{2}{2} \le \frac{5}{5}$$

Получим соответственно оптимальный порядок выполнения работ, указанный в примере. Третья и четвертая работы имеют одинаковые отношения (2 / 2 = 5 / 5), поэтому располагаем их в порядке возрастания номеров работ.

Реализация алгоритма

Информацию о работах будем заносить в массив jobs, элементами которого являются вектора длины 3. После считывания данных jobs[i][0] содержит время выполнения i - ой работы T_i, jobs[i][1] содержит значение штрафа S_i, а jobs[i][2] содержит номер работы i.

```
vector<int> j(3,0);
vector<vector<int> > jobs;
```

Функция сортировки. Сравнение $\frac{a[0]}{b[0]} < \frac{a[1]}{b[1]}$ эквивалентно a[0] * b[1] < b[0] * a[1]. Если отношения a[0] / b[0] и a[1] / b[1] равны, то раньше должна следовать

работа с меньшим номером. Поэтому в этом случае следует сравнивать номера работ, которые содержатся в a[2] и b[2].

```
int lt(vector<int> a, vector<int> b)
{
  if (a[0] * b[1] == b[0] * a[1]) return a[2] < b[2];
  return a[0] * b[1] < b[0] * a[1];
}
```

Основная часть программы. Читаем входные данные. Заполняем массив jobs.

```
while(scanf("%d",&n) == 1)
{
  jobs.clear();
  for(i = 1; i <= n; i++)
  {
    scanf("%d %d",&j[0],&j[1]); j[2] = i;
    jobs.push_back(j);
  }
```

Сортируем работы согласно компаратору ***lt***.

```
  sort(jobs.begin(),jobs.end(),lt);
```

Выводим результат как требуется в условии задачи.

```
  for(i = 0; i < n; i++)
    printf("%d ",jobs[i][2]);
  printf("\n");
}
```

765. Велосипед

Велосипедист собирается проехать из пункта А в пункт В, расстояние между которыми составляет l м. У него есть велосипед, который может развивать скорость v м/с. Однако перед тем как выехать, велосипедист может выполнить некоторые модернизации своего велосипеда. Для каждой модернизации известно на сколько она увеличивает скорость велосипеда, а также время, за которое она может быть сделана. Можно выполнить несколько различных модернизаций, однако каждая модернизация может быть выполнена не более одного раза. Помогите велосипедисту добраться до пункта В как можно быстрее.

Вход. Сначала идут три целых числа: расстояние между пунктами l ($0 \le l \le 10^9$), исходная скорость велосипеда v ($1 \le v \le 10^6$) и количество различных модернизаций n ($0 \le n \le 100$). Далее идут n пар целых чисел, каждая из которых определяет соответствующую модернизацию: прирост скорости после модернизации v_i ($0 \le v_i \le 1000$) и время t_i ($0 \le t_i \le 1000$), затрачиваемое на эту модернизацию. Все величины заданы в системе СИ (метры и секунды).

Выход. Вывести минимальное время с шестью десятичными знаками, которое потребуется велосипедисту для того чтобы доехать из пункта A в пункта B с учетом времени на модернизации.

Пример входа 1	**Пример выхода 1**
`100 5 2 5 3 5 3`	`12.666667`

Пример входа 2	**Пример выхода 2**
`100 4 3 1 2 2 4 3 6`	`20.285714`

Пусть dp[i] хранит наименьшее время, через которое можно добиться прибавки скорости i. Изначально положим dp[i] = $+\infty$, dp[0] = 0.

Рассмотрим пару модернизации (v_i, t) означающую что за время t можно получить прирост по скорости v_i. Рассмотрим прирост скорости j ($j > 0$), который можно получить за время dp[j] Тогда используя модернизацию (v_i, t), можно получить прирост скорости $j + v_i$ за время dp[j] + t. И если это время окажется меньше dp[$j + v_i$], то значение dp[$j + v_i$] следует улучшить.

Далее ищем время, за которое можно добраться из пункта A в пункт B при условии что прирост скорости велосипедиста составит $i = 0, 1, 2, \ldots \sum_{i=1}^{n} v_i$. Например, если велосипедист будет ехать со скоростью $v + i$, то его время поездки составит $l / (v + i)$ + dp[i] секунд. Среди всех таких времен находим наименьшее.

Пример. Рассмотрим второй тест. Рассмотрим как изменяется состояние массива dp с обработкой очередной модернизации.

	0	1	2	3	4	5	6	
(v_i, t)	0	∞	∞	∞	∞	∞	∞	инициализация
(1, 2)	0	2	∞	∞	∞	∞	∞	
(2, 4)	0	2	4	6	∞	∞	∞	
(3, 6)	0	2	4	6	8	10	12	

Например dp[5] = 10 означает, что прирост скорости 5 можно набрать за 10 секунд. Для этого достаточно воспользоваться второй и третьей модернизациями.

Реализация алгоритма

Объявим рабочий массив dp.

```
#define MAX 100010
int dp[MAX];
```

Читаем входные данные. Инициализируем массив dp.

```
scanf("%d %d %d",&l,&v,&n);
memset(dp,0x3F,sizeof(dp)); dp[0] = mx = 0;
```

Совершим пересчет массива dp для всех модернизаций.

```
for(i = 0; i < n; i++)
{
  scanf("%d %d",&vi,&t);
  for(j = mx; j >= 0; j--)
    if (dp[j] + t < dp[j + vi]) dp[j + vi] = dp[j] + t;
  mx += vi;
}
```

Для каждого значения прироста скорости i вычислим время, необходимое для преодоления расстояния из пункта A в пункт B. Оно равно величине пути l, деленное на скорость велосипедиста $v + i$, плюс время модернизации dp[i]. Наименьшее из таких времен и будет искомым.

```
res = 1e10;
for(i = 0; i <= mx; i++)
{
  temp = 1.0 * l / (v + i) + dp[i];
  if (temp < res) res = temp;
}
```

Выводим ответ.

```
printf("%.6lf\n",res);
```