Министерство образования и науки Украины

Зимняя школа по программированию

Харьков, ХНУРЭ
2010

Оглавление

День первый (19.02.2010г). Контест Антона Лунёва и Виталия Неспирного

Об авторах...

Лунёв Антон Андреевич, аспирант Донецкого национального университета, специальность 01.01.01 "Математический анализ".

Основные научные интересы:

- полнота и базисность систем корневых векторов некоторых классов обыкновенных дифференциальных операторов;
- точные константы в неравенствах для промежуточных производных;
- корневые решения обобщенного уравнения Маркова;
- игра Грюнди и ее обобщения. Увлекается программированием.

Основные достижения:

- трижды (2005, 2006, 2007) был награжден первым призом на международной математической олимпиаде для студентов IMC;
- в 2006, 2007 годах награжден серебряной медалью, а в 2008 — золотой, на международной студенческой математической олимпиаде в Иране;
- в составе команды DonNU United Донецкого национального университета занял 7-е место в 2008 году и 4-е место в 2009 году на региональной олимпиаде ACM по программированию (Southeastern European Regional Contest);
- в 2009 году занял в составе команды Донецкого национального университета 4-е место на Всеукраинской студенческой олимпиаде по программированию (г. Винница);

- занял 9-е место в зимней серии индивидуальных соревнований по программированию SnarkNews Winter Series 2009 и 3-е место в летней серии индивидуальных соревнований по программированию SnarkNews Summer Series 2009, по их итогам награжден дипломами третьей и первой степени соответственно.

Неспирный Виталий Николаевич, кандидат физико-математических наук (специальность 01.02.01 “Теоретическая механика”), младший научный сотрудник отдела технической механики Института прикладной математики и механики НАН Украины, ассистент кафедры теории упругости и вычислительной математики Донецкого национального университета, заместитель председателя жюри II-III этапов Всеукраинской олимпиады школьников по информатике в Донецкой области, координатор Донецкого сектора Открытого кубка по программированию, тренер команд математического факультета Донецкого национального университета.

Основные достижения:

- занял 1-е место на Всеукраинской студенческой олимпиаде по информатике (Николаев, 2000);
- в составе команды математического факультета ДонНУ занял 1-е место на Всеукраинской ACM-олимпиаде по программированию и 8-е в личном первенстве (Винница, 2001);
- подготовил призеров международных школьных олимпиад – Петр Луференко (2002), Роман Ризванов (2006), Вадим Янушкевич (2009);
- совместно с А.И.Парамоновым подготовил команду DonNU United, занявшую 7-е (2008) и 4-е место (2009) в ACM SEERC.

Теоретический материал. Лекция по теории чисел

Основные обозначения

$\mathbb{N}$ – множество натуральных чисел $\{1, 2, 3, \ldots\}$.

$\mathbb{Z}$ – множество целых чисел $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

$\mathbb{Z}_+$ – множество целых неотрицательных чисел $\{0, 1, 2, 3, \ldots\}$.

$a \bmod b$ – остаток числа a при делении на b.

(a, b) – наибольший общий делитель двух целых чисел a и b.

$(a_1, a_2, \ldots, a_n)$ – наибольший общий делитель n целых чисел $a_1, a_2, \ldots, a_n$.

$[a, b]$ – наименьшее общее кратное двух целых чисел a и b.

$[a_1, a_2, \ldots, a_n]$ – наименьшее общее кратное n целых чисел $a_1, a_2, \ldots, a_n$.

$O(f(n))$ – некоторая величина, зависящая от n, которая не превосходит константы, умноженной на $f(n)$, начиная с некоторого n.

$\tau(n)$ – количество натуральных делителей числа n.

$\sigma(n)$ – сумма натуральных делителей числа n.

$\varphi(n)$ – количество натуральных чисел, не превосходящих n и взаимно простых с n (так называемая функция Эйлера).

$n!$ – факториал n, равен $1 \cdot 2 \cdot \ldots \cdot n$.

$\deg_p(n)$ – наибольшее число k, для которого n делится на p^k.

$[x]$ – целая часть числа x, то есть наибольшее целое число, не превосходящее x.

$(\overline{a_{n-1} \ldots a_1 a_0})_p$ – p-ичная запись числа. Здесь $a_0, a_1, \ldots, a_{n-1} \in \{0, 1, \ldots, p-1\}$. Значение таким образом записанного числа вычисляется по формуле $a_{n-1}p^{n-1} + \ldots + a_1 p + a_0$.

Сравнения

Определение 1. *Два целых числа a и b называются сравнимыми по модулю натурального числа m (или равноостаточными при делении на m), если при делении на m они дают одинаковые остатки.*

Эквивалентные формулировки: a и b сравнимы по модулю m, если

- их разность $a - b$ делится на m,
- a может быть представлено в виде $a = b + km$, где k – некоторое целое число.

Например: 32 и -10 сравнимы по модулю 7, так как $32 = 7 \cdot 4 + 4$ и $-10 = 7 \cdot (-2) + 4$.

Утверждение "a и b сравнимы по модулю m" записывается в виде: $a \equiv b \pmod{m}$.

Свойства сравнений по модулю

1. Отношение сравнения по модулю обладает свойствами
   - рефлексивности: для любого целого a справедливо $a \equiv a \pmod{m}$.

- симметричности: если $a \equiv b \pmod m$, то $b \equiv a \pmod m$.
- транзитивности: если $a \equiv b \pmod m$ и $b \equiv c \pmod m$, то $a \equiv c \pmod m$.

2. Сравнения можно складывать почленно: если $a_1 \equiv b_1 \pmod m$ и $a_2 \equiv b_2 \pmod m$, то $a_1 + a_2 \equiv b_1 + b_2 \pmod m$.
3. Сравнения можно умножать почленно: если $a_1 \equiv b_1 \pmod m$ и $a_2 \equiv b_2 \pmod m$, то $a_1 a_2 \equiv b_1 b_2 \pmod m$.
4. Если $a \equiv b \pmod m$, то $a^k \equiv b^k \pmod m$ при любом натуральном k.
5. Если $a \equiv b \pmod m$ и m делится на d, то $a \equiv b \pmod d$.
6. Если $a \equiv b \pmod{m_1}$ и $a \equiv b \pmod{m_2}$, то $a \equiv b \pmod m$, где $m = [m_1, m_2]$.
7. Для того, чтобы числа a и b были сравнимы по модулю m, представленному в виде его канонического разложения на простые сомножители p_i: $m = \prod_{i=1}^{n} p_i^{\alpha_i}$, необходимо и достаточно, чтобы выполнялось условие:
$$a \equiv b \pmod{p_i^{\alpha_i}}, \quad i = 1, 2, \ldots, n.$$
8. Сравнения, нельзя, вообще говоря, делить на другие числа. Пример: $14 \equiv 20 \pmod 6$. Однако, сократив на 2, мы получаем ошибочное сравнение: $7 \equiv 10 \pmod 6$. Правила сокращения для сравнений следующие.
   - Можно делить обе части сравнения на число, взаимно простое с модулем: если $ac \equiv bc \pmod m$ и $(c, m) = 1$, то $a \equiv b \pmod m$.
   - Можно одновременно разделить обе части сравнения и модуль на их общий делитель: если $ac \equiv bc \pmod{mc}$, то $a \equiv b \pmod m$.
9. Нельзя также выполнять операции со сравнениями, если их модули не совпадают.

Бинарный алгоритм возведения в степень

Пусть нам требуется быстро вычислить $a^n \bmod m$. Тогда можно поступить следующим образом. Разложим n в двоичную запись:

$$n = 2^{r_k} + \ldots + 2^{r_2} + 2^{r_1}, \quad r_k > \ldots > r_2 > r_1 \geq 0.$$

Пусть $a_j = a^{2^j} \bmod m$. Тогда:

$$a^n \equiv a^{2^{r_k}+\ldots+2^{r_2}+2^{r_1}} \equiv a_{r_k} \cdot \ldots \cdot a_{r_2} \cdot a_{r_1} \pmod{m}. \quad (1)$$

Ясно, что $a_0 = a \bmod m$ и $a_{j+1} = a_j^2 \bmod m$. Эти формулы вместе со сравнением (1) дают алгоритм вычисления $a^n \bmod m$ со сложностью $O(\log n)$.

Его можно описать также следующим образом. Сначала присваиваем переменной p значение $1 \bmod m$ (взятие по модулю здесь имеет смысл, когда $m = 1$, но можно, конечно, рассмотреть этот случай отдельно), а затем в цикле выполняем следующие действия:

1. Если $n \bmod 2 = 1$, то умножить p на a по модулю m.
2. Поделить n на 2, округлив результат вниз, то есть присвоить n число $[n/2]$.
3. Если $n = 0$, то выйти из цикла.
4. Возвести a в квадрат по модулю m.

Тогда ответ будет содержаться в переменной p.
На C++ этот алгоритм можно записать совсем коротко:

```
p=1%m;
for(;n;)
{
    if(n%2) p=p*a%m;
    if(n/=2) a=a*a%m;
}
```

Аналогичный алгоритм может быть применен не только для возведения чисел в степень по модулю. Но и для вычисления степеней произвольных объектов, на которых введена бинарная ассоциативная мер, нахождения степени матрицы или перестановки. Эта идея развивается в следующем пункте.

Вычисление элементов возвратных последовательностей

Возвратной называется последовательность $a_0, a_1, a_2, \ldots$, удовлетворяющая соотношению вида:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k}.$$

Для того, чтобы возвратная последовательность была полностью определена, достаточно задать ее первые k значений: a_0, a_1, ..., a_{k-1}. Тогда все члены последовательности до n-ого могут быть последовательно вычислены по формуле за $O(kn)$ операций.

Если требуется вычислить лишь a_n, то это может быть выполнено за $O(k^3 \log n)$. При условии, что n – большое значение, а k невелико (в большинстве случае это имеет место), такой алгоритм будет предпочтительнее.

Обозначим через A_i вектор-столбец $(a_i, a_{i+1}, \ldots, a_{i+k-1})^T$. Рассмотрим преобразование перехода от этого вектора к A_{i+1}. Оно будет описываться матрицей размера $k \times k$:

$$C = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 \\ c_k & c_{k-1} & c_{k-2} & \ldots & c_1 \end{pmatrix}.$$

Тогда $A_n = CA_{n-1} = C^2A_{n-2} = \ldots = C^nA_0$. Таким образом, для получения a_n, как первого элемента A_n, необходимо возвести матрицу C в степень n, и домножить ее первую строку на вектор начальных значений A_0. Самое трудоемкое здесь – это возведение в степень, что может быть выполнено с помощью выше описанного бинарного алгоритма за $O(\log n)$ операций умножения. Однако поскольку умножение выполняется над матрицами $k \times k$, каждое из них потребует $O(k^3)$ обычных операций сложения и умножения.

В качестве примера можно рассмотреть последовательность Фибоначчи:

$$a_n = a_{n-1} + a_{n-2},\ a_0 = 0,\ a_1 = 1.$$

Здесь матрица C имеет вид $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, а $A_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Таким образом, n-ое число Фибоначчи будет вторым элементом первой строки матрицы C^n.

Алгоритм Евклида

Пусть a и b целые числа, не равные одновременно нулю, и последовательность чисел $r_{-1} = a$, $r_0 = b$, $r_1 > r_2 > r_3 > \ldots > r_n > 0$ определена тем, что r_k — это остаток от деления r_{k-2} на r_{k-1} при $k = 1, 2, \ldots, n$, а r_{n-1} делится на r_n нацело, то есть:

$$\begin{aligned}
a &= bq_0 + r_1, \quad 0 < r_1 < b, \\
b &= r_1 q_1 + r_2, \quad 0 < r_2 < r_1, \\
r_1 &= r_2 q_2 + r_3, \quad 0 < r_3 < r_2, \\
&\dots \\
r_{k-2} &= r_{k-1} q_{k-1} + r_k, \quad 0 < r_k < r_{k-1}, \\
&\dots \\
r_{n-1} &= r_n q_n.
\end{aligned} \tag{2}$$

Тогда $(a, b) = r_n$, то есть наибольший общий делитель (НОД) чисел a и b равен последнему ненулевому члену этой последовательности. Существование таких r_1,r_2,... доказывается индукцией по b.

Корректность этого алгоритма вытекает из того, что $(bq + r, b) = (b, r)$ и $(0, r) = r$ для любых $b, q, r \in \mathbb{Z}$ $(r \neq 0)$.

Реализовать этот алгоритм для неотрицательных чисел [1] можно следующим образом:

```
r1=a; r2=b;
while (r2)
{
  q = r1 / r2;
  r1 -= q * r2;
  swap(r1,r2);
}
```

По окончании алгоритма r_1 станет равным значению (a, b), а r_2 – нулю. Оценить сложность алгоритма поможет следующая теорема.

Теорема. 1. *Если $a > b > 0$ и числа $r_1, r_2, \dots, r_n$ удовлетворяют равенствам* (2)*, то $F_{n+2} \leq b$, где F_k – k-тое число Фибоначчи.*

Доказательство. По индукции можно доказать, что $b \geq F_{k-1} r_k + F_k r_{k-1}$ при $2 \leq k \leq n$, исходя из равенств (2) и того очевидного факта, что $q_k \geq 1$. Из равенства $r_{n-1} = q_n r_n$ и неравенства $0 < r_n < r_{n-1}$ следует, что $q_n \geq 2$. Поэтому по доказанной оценке для b имеем:

$$b \geq F_{n-1} r_n + F_n r_{n-1} = (F_{n-1} + F_n q_n) r_n \geq F_{n-1} + 2F_n = F_{n+2}.$$

□

Так как $F_n = \frac{\varphi^n + \varphi^{-n}}{2\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$ [2], то из теоремы 1 следует, что $n \leq \log_\varphi(2\sqrt{5}b) - 2$.

[1] отрицательное число всегда можно заменить на его модуль, НОД от этого не изменится

[2] это так называемая формула Бине, она легко доказывается по индукции

Откуда следует, что сложность алгоритма Евклида, примененного к числам a и b ($a > b$) равна $O(\log b)$.

Обобщенный алгоритм Евклида и уравнения вида $ax + by = c$

Рассмотрим теперь применение алгоритма Евклида к решению неопределенных линейных диофантовых уравнений вида $ax + by = c$.

Сначала покажем как решить уравнение $ax + by = (a, b)$. Для этого обобщим обычный алгоритм Евклида. Добавим переменные x_1, y_1 и x_2, y_2 и обеспечим сохранение равенств $a \cdot x_1 + b \cdot y_1 = r_1$ и $a \cdot x_2 + b \cdot y_2 = r_2$ после каждой итерации цикла:

```
x1=1; y1=0; r1=a;
x2=0; y2=1; r2=b;
while(r2)
{
    q = r1 / r2;
    r1 -= q * r2; swap(r1,r2);
    x1 -= q * x2; swap(x1,x2);
    y1 -= q * y2; swap(y1,y2);
}
```

Как уже отмечалось, по окончании алгоритма r_1 станет равным значению (a, b), а, значит, (x_1, y_1) будет решением уравнения $ax + by = (a, b)$, причем одно из этих чисел будет положительным, а другое неположительным. Важно также отметить, что все время в процессе вычислений выполнены неравенства $|x_1|, |x_2| \le b/(a, b)$ и $|y_1|, |y_2| \le a/(a, b)$. Из этого следует, что если переменные a и b были какого-то знакового целочисленного типа, то все вычисления можно смело проводить в нем, не боясь переполнений.

Перейдем теперь к описанию всех решений уравнения $ax + by = c$. Если c не делится на $d = (a, b)$, то такое уравнение не имеет целочисленных решений (действительно при любых целых x и y левая часть делится на (a, b)). Если же c делится на d, то необходимо найти хотя бы одно (частное) решение (допустим (x^*, y^*)). Тогда общее решение может быть записано в виде:

$$x = x^* + t \cdot (b/d),$$
$$y = y^* - t \cdot (a/d),$$

где t – произвольное целое число.

Найти же частное решение можно следующим образом: с помощью выше описанного обобщенного алгоритма Евклида находится решение (x_0, y_0) уравнения $ax + by = (a, b)$. Тогда, очевидно, $x^* = x_0 \cdot (c/d)$, $y^* = y_0 \cdot (c/d)$.

Обратный остаток по модулю

Пусть числа $a \in \mathbb{Z}$ и $m \in \mathbb{N}$ взаимно просты. Тогда по предыдущему пункту существуют такие $x, y \in \mathbb{Z}$, что $ax + my = 1$. Откуда $ax \equiv 1 \pmod{m}$. Из формулы для общего решения уравнения $ax + my = 1$ следует, что по модулю m такой x единственный. В связи с этим можно ввести следующее определение.

Определение 2. *Пусть числа $a \in \mathbb{Z}$ и $m \in \mathbb{N}$ взаимно просты. Число $x \in \{0, 1, \ldots, m-1\}$ называется обратным остатком к a по модулю m и обозначается* $\mathrm{inv}_m(a)$*, если выполнено сравнение $ax \equiv 1 \pmod{m}$.*

С обратным остатком в сравнениях можно работать также как с дробью $\frac{1}{a}$ в равенствах. А именно, выполняются следующие свойства:

1. $c \cdot \mathrm{inv}_m(a) + d \cdot \mathrm{inv}_m(b) \equiv (cb + da) \cdot \mathrm{inv}_m(a) \cdot \mathrm{inv}_m(b) \pmod{m}$.
2. $\mathrm{inv}_m(a) \cdot \mathrm{inv}_m(b) \equiv \mathrm{inv}_m(ab) \pmod{m}$.
3. Если n делится на a и $(a, m) = 1$, то $\frac{n}{a} \equiv n \cdot \mathrm{inv}_m(a) \pmod{m}$.
4. Если m делится на d, то $\mathrm{inv}_m(a) \equiv \mathrm{inv}_d(a) \pmod{d}$.

Первые два свойства проверяются домножением на ab, которое в данном случае является, равносильным переходом, так как $(a, m) = (b, m) = 1$. Последние два свойства проверяются домножением на a.

Обратный остаток может быть найден с помощью расширенного алгоритма Евклида. При этом, так как нам надо только x, то переменные y_1 и y_2 вообще можно не заводить.

Мультипликативные функции

Определение 3. *Функция f из $\mathbb{N}$ в $\mathbb{R}$ называется мультипликативной, если $f(1) = 1$ и $f(m)f(n) = f(mn)$ для любых взаимно простых натуральных чисел n и m.*

Функции $\tau(n)$, $\sigma(n)$ и $\varphi(n)$ являются примерами мультипликативных функций, что следует из следующих для них формул.

Теорема. 2. *Пусть $n = p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}$ – каноническое разложение числа n на простые множители. Тогда:*

$$\tau(n) = (a_1 + 1) \cdot \ldots \cdot (a_k + 1),$$

$$\sigma(n) = \frac{p_1^{a_1+1} - 1}{p_1 - 1} \cdot \ldots \cdot \frac{p_k^{a_k+1} - 1}{p_k - 1},$$

$$\varphi(n) = p_1^{a_1-1}(p_1 - 1) \cdot \ldots \cdot p_k^{a_k-1}(p_k - 1) = n \cdot \frac{p_1 - 1}{p_1} \cdot \ldots \cdot \frac{p_k - 1}{p_k}.$$

Малая теорема Ферма и теорема Эйлера

Теорема. 3 (Эйлер)**.** *Пусть* $m > 1$, $(a, m) = 1$, $\varphi(m)$ — *функция Эйлера от числа* m. *Тогда:*

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

Доказательство. Пусть $r_1, r_2, \ldots, r_c$ – все различные натуральные числа меньшие m и взаимно простые с m, где $c = \varphi(m)$ – их число. Пусть

$$\begin{aligned} a \cdot r_1 &\equiv \rho_1 \pmod{m}, \\ a \cdot r_2 &\equiv \rho_2 \pmod{m}, \\ &\ldots \\ a \cdot r_c &\equiv \rho_c \pmod{m}, \end{aligned}$$

где $\rho_k = a \cdot r_k \bmod m$ – остаток при делении числа $a \cdot r_k$ на m.

Так как a взаимно просто с m, то $\{r_1, r_2, \ldots, r_c\} = \{\rho_1, \rho_2, \ldots, \rho_c\}$.

Перемножим эти c сравнений. Получится:

$$a^c \cdot r_1 r_2 \ldots r_c \equiv \rho_1 \rho_2 \ldots \rho_c \pmod{m}$$

Так как $r_1 r_2 \ldots r_c = \rho_1 \rho_2 \ldots \rho_c$ и взаимно просто с модулем m, то, поделив последнее сравнение на $r_1 r_2 \ldots r_c$, получим $a^{\varphi(m)} \equiv 1 \pmod{m}$. □

Теорема. 4 (Малая теорема Ферма)**.** *Пусть* p – *простое число,* p *не делит* a. *Тогда* $a^{p-1} \equiv 1 \pmod{p}$.

Доказательство. Эта теорема является частным случаем теоремы Эйлера, поскольку при простом m функция Эйлера $\varphi(m) = m - 1$. □

Следствие 1. *Для произвольного целого* a *и простого* p *выполняется* $a^p \equiv a \pmod{p}$.

Доказательство. Если a не делится на p, то это следует из малой теоремы Ферма, а при a кратном p сравнение выполняется тривиально. □

С помощью теоремы Эйлера можно получить другой алгоритм нахождения обратного остатка. Пусть $(a, m) = 1$. Тогда по теореме Эйлера $a^{\varphi(m)} \equiv 1 \pmod{m}$. Или $a \cdot a^{\varphi(m)-1} \equiv 1 \pmod{m}$. Откуда следует, что $\mathrm{inv}_m(a) = a^{\varphi(m)-1} \bmod m$. В частности, если p — простое число, то $\mathrm{inv}_p(a) = a^{p-2} \bmod p$. Таким образом, если известна функция Эйлера от m, то обратный остаток к a может быть найден за $O(\log m)$ с помощью бинарного алгоритма возведения числа по модулю.

Степенные сравнения и теорема Вильсона

Рассмотрим сравнения вида $f(x) \equiv 0 \pmod p$, где p – простое число, а $f(x) = a_n x^n + \ldots + a_1 x + a_0$ – многочлен с целыми коэффициентами.

Лемма. 1. *Произвольное сравнение $f(x) \equiv 0 \pmod p$, где p – простое число, равносильно некоторому сравнению степени не выше $p-1$.*

Доказательство. Разделим $f(x)$ на многочлен $x^p - x$ с остатком: $f(x) = (x^p - x) \cdot Q(x) + R(x)$, где, как известно, степень остатка $R(x)$ не превосходит $p-1$. Но, по малой теореме Ферма, $x^p - x \equiv 0 \pmod p$. Это означает, что $f(x) \equiv R(x) \pmod p$, а исходное сравнение равносильно сравнению $R(x) \equiv 0 \pmod p$. □

Теорема. 5 (Лагранжа)**.** *Если сравнение $a_n x^n + \ldots + a_1 x + a_0 \equiv 0 \pmod p$ степени n по простому модулю p имеет более n различных решений, то все коэффициенты a_0, a_1, …, a_n кратны p.*

Доказательство. Пусть сравнение $a_n x^n + \ldots + a_1 x + a_0 \equiv 0 \pmod p$ имеет $n+1$ различных по модулю p решений x_1, x_2, …, x_n, x_{n+1}. Многочлен $f(x)$ можно представить в виде:

$$\begin{aligned} f(x) = & b_n(x-x_1)(x-x_2)\ldots(x-x_{n-1})(x-x_n) + \\ & + b_{n-1}(x-x_1)(x-x_2)\ldots(x-x_{n-1}) + \\ & \ldots \\ & + b_2(x-x_1)(x-x_2) + b_1(x-x_1) + b_0. \end{aligned}$$

Коэффициенты b_i являются при этом линейными комбинациями чисел a_i с целыми коэффициентами. Более того, b_i равен сумме a_i и линейной комбинации чисел $a_0, a_1, \ldots, a_{i-1}$. Откуда следует, что и числа a_i являются линейными комбинациями чисел b_i с целыми коэффициентами.

Положим теперь последовательно $x = x_1, x_2, \ldots, x_n, x_{n+1}$.

$f(x_1) = b_0 \equiv 0 \pmod p$, следовательно b_0 делится на p.

$f(x_2) = b_0 + b_1(x_2 - x_1) \equiv b_1(x_2 - x_1) \pmod p$. Поскольку x_1 и x_2 – различные решения, то $x_2 \not\equiv x_1 \pmod p$ и, значит, b_1 должно делиться на p.

Продолжая таким же образом дальше, получим, что все b_i кратны p, а значит и a_i, зависящие линейно от b_i, кратны p. □

Теорема. 6 (Вильсон)**.** *Пусть p – простое число. Тогда выполнено сравнение:*

$$(p-1)! \equiv -1 \pmod p$$

Доказательство. Пусть p – простое число. Если $p = 2$, то, очевидно, $1! + 1 \equiv 0 \pmod 2$. Если $p > 2$, то рассмотрим сравнение

$$[(x+1)(x+2)\ldots(x+(p-1))] - (x^{p-1} - 1) \equiv 0 \pmod p.$$

Ясно, что это сравнение степени не выше $p-2$, но оно имеет $p-1$ решение: 1, 2, 3, ..., $p-1$, т.к. при подстановке любого из этих чисел, слагаемое в квадратных скобках обращается в ноль, а $x^{p-1}-1$ сравнимо с нулем по малой теореме Ферма (x и p взаимно просты, т.к. $x < p$). Это означает, по теореме Лагранжа, что все коэффициенты выписанного сравнения кратны p, в частности, на p делится его свободный член, равный $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (p-1) + 1$. □

Китайская теорема об остатках

Теорема. 7. *Пусть $m_1, m_2, \ldots, m_n$ – попарно взаимно простые натуральные числа и $r_1, r_2, \ldots, r_n \in \mathbb{Z}$. Тогда существует единственное число a по модулю $M = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ такое, что выполняется следующая система сравнений:*

$$\begin{array}{rl} a & \equiv r_1 \pmod{m_1} \\ a & \equiv r_2 \pmod{m_2} \\ & \ldots \\ a & \equiv r_n \pmod{m_n} \end{array} \tag{3}$$

Доказательство. Сначала докажем единственность решения по модулю M. Пусть существует два числа a и b, удовлетворяющих системе (3), тогда их разность делится на каждое из m_i, а значит, и на M, поэтому по модулю M числа a и b совпадают.

Теперь докажем существование. Покажем, что число $a = \sum_{i=1}^{n} r_i \left(\frac{M}{m_i}\right)^{\varphi(m_i)}$ подходит. Так как $\frac{M}{m_i}$ кратно m_j, при $j \neq i$, и взаимно просто с m_i, то по тереме Эйлера:

$$a \equiv r_1 \cdot 0 + \ldots + r_{i-1} \cdot 0 + r_i \cdot 1 + r_{i+1} \cdot 0 + \ldots + r_n \cdot 0 \equiv r_i \pmod{m_i}.$$

□

О факториале

Пусть $n \in \mathbb{N}$ и p – простое число. Найдем формулу для $\deg_p(n!)$. Поделим n на p с остатком: $n = pq + r$, $0 \leq r < p$. Ясно, что на $\deg_p(n!)$ влияют только кратные p числа в произведении $1 \cdot 2 \cdot \ldots \cdot n$. Поэтому имеем:

$$\deg_p(n!) = \deg_p(p \cdot 2p \cdot \ldots qp) = \deg_p(p^q q!) = q + \deg_p(q!). \tag{4}$$

Из этой формулы получается очень простой и эффективный алгоритм нахождения $\deg_p(n!)$:

```
s=0;
for(;n;n/=p) s+=n/p;
```

По окончанию алгоритма s станет равным значению $\deg_p(n!)$.

Из формулы (4) также следует важная теоретически формула Лежандра:

$$\deg_p(n!) = \sum_{j=1}^{[\log_p n]} \left[\frac{n}{p^j}\right]. \tag{5}$$

Получим еще одну формулу для $\deg_p(n!)$ исходя из (5). Пусть $n = (\overline{a_{k-1}\ldots a_1 a_0})_p$ — p-ичная запись числа n. Тогда:

$$\left[\frac{n}{p^j}\right] = a_{k-1}p^{k-1-j} + \ldots + a_{j+1}p + a_j.$$

Используя это равенство после некоторых выкладок придем к формуле:

$$\deg_p(n!) = \frac{n - S_p(n)}{p-1}, \tag{6}$$

где $S_p(n) = a_{k-1} + \ldots + a_1 + a_0$ – сумма цифр числа n в p-ичной записи.

Рассмотрим теперь следующую задачу. Найти остаток числа $n!/p^{\deg_p(n!)}$ при делении на p. Другими словами $n!$ делят нацело на p, пока это возможно, и полученное число берут по модулю p. Самый простой подход состоит в том, чтобы последовательно рассматривать числа от 1 до n, делить каждое из них нацело на p, пока это возможно и домножать на полученный результат текущий ответ. Получается алгоритм со сложностью $O(n)$. Но часто бывает, что требуется вычислить эту величину для больших n и сравнительно небольших p. В таком случае этот наивный алгоритм неприемлем и надо поступать по другому. Поделим n на p с остатком: $n = pq + r$, $0 \le r < p$. Тогда имеем:

$$n! = \prod_{i=1}^{n} i = \left(\prod_{j=0}^{q-1}\prod_{k=1}^{p-1}(pj+k)\right)\left(\prod_{k=1}^{r}(pq+k)\right)\left(\prod_{j=1}^{q} pj\right) =$$

$$= \left(\prod_{j=0}^{q-1}\prod_{k=1}^{p-1}(pj+k)\right)\left(\prod_{k=1}^{r}(pq+k)\right) q!p^q.$$

Пусть $F(n) = n!/p^{\deg_p(n!)} \bmod p$ — величина, которую требуется найти. Тогда из полученного равенства следует, что:

$$F(n) \equiv F(q) \left(\prod_{j=0}^{q-1} \prod_{k=1}^{p-1} (pj + k) \right) \left(\prod_{k=1}^{r} (pq + k) \right) \equiv$$

$$\equiv F(q) \left(\prod_{j=0}^{q-1} \prod_{k=1}^{p-1} k \right) \left(\prod_{k=1}^{r} k \right) \equiv F(q)((p-1)!)^q r! \pmod{p}.$$

Откуда, по теореме Вильсона, получаем окончательный ответ:

$$F(n) \equiv F(q)(-1)^q r! \pmod{p}. \tag{7}$$

Если теперь перед вычислением $F(n)$ запомнить в массиве значения $r! \bmod p$ при $0 \le r < p$, что можно легко сделать в цикле за $O(p)$, то формула (7) дает нам простой алгоритм для вычисления $F(n)$ со сложностью $O(\log n)$.

Квадратичные сравнения

Квадратичные вычеты

Определение 4. *Целое число a называется квадратичным вычетом по модулю p, если $a \equiv b^2 \pmod{p}$ для некоторого целого b. В противном случае, число a называется квадратичным невычетом.*

Лемма. 2. *Пусть p – простое нечетное число. Тогда, среди чисел $1, 2, \ldots, p-1$ ровно половина квадратичных вычетов и ровно половина квадратичных невычетов по модулю p.*

Доказательство. Рассмотрим числа $j^2 \bmod p$ при $j = 1, 2, \ldots, \frac{p-1}{2}$. Все эти числа являются квадратичными вычетами по определению. Ясно, что все они различны, так как при $1 \le i < j \le \frac{p-1}{2}$ разность $i^2 - j^2 = (i-j)(i+j)$ не кратна p. С другой стороны, так как $j^2 \equiv (p-j)^2 \pmod{p}$, то все квадратичные вычеты находятся среди рассматриваемых чисел. Значит, их ровно $\frac{p-1}{2}$. □

Лемма. 3 (критерий Эйлера)**.** *Пусть p – простое нечетное число и a не кратно p. Число a является квадратичным вычетом по модулю p тогда и только тогда, когда:*

$$a^{\frac{p-1}{2}} \equiv 1 \pmod{p}.$$

Доказательство. Если $a \equiv x^2 \pmod{p}$, то по малой теореме Ферма $a^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \pmod{p}$. То есть все $\frac{p-1}{2}$ квадратичных вычетов являются решениями сравнения $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$. Но по теореме Лагранжа это сравнение имеет не более $\frac{p-1}{2}$ решений. Поэтому все его решения являются квадратичными вычетами. □

Определение 5. Для простого p символ Лежандра $\left(\frac{a}{p}\right)$ определяется так:

$$\left(\frac{a}{p}\right) = \begin{cases} 0, \text{если } a \text{ делится на } p, \\ 1, \text{если } a - \text{квадратичный вычет по модулю } p, \\ -1, \text{если } a - \text{квадратичный невычет по модулю } p. \end{cases}$$

Свойства:

1. $a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \pmod{p}$, при $p > 2$.
2. Если $a \equiv b \pmod{p}$, то $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.
3. $\left(\frac{a_1 a_2 \ldots a_n}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)\ldots\left(\frac{a_n}{p}\right)$.

Лемма. 4 (Гаусса)**.** *Пусть p – нечетное простое число, $q \in \mathbb{N}$ и q не кратно p. Для каждого натурального числа l от 1 до $\frac{p-1}{2}$ запишем $lq \equiv \pm r_l \pmod{p}$, где $1 \le r_l \le \frac{p-1}{2}$. Пусть μ – количество всех встречающихся здесь минусов, тогда:*

$$\left(\frac{q}{p}\right) = (-1)^{\mu} \quad \text{и} \quad \mu \equiv \sum_{l=1}^{\frac{p-1}{2}} \left[\frac{2lq}{p}\right] \pmod{2}.$$

Теорема. 8 (Закон взаимности Гаусса)**.** *Пусть p, q – нечетные простые числа. Тогда:*

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\cdot\frac{q-1}{2}} \quad \text{и} \quad \left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}.$$

Символ Якоби

Определение 6. *Пусть P – произвольное нечётное число, большее единицы, и $P = p_1 p_2 \ldots p_k$ – разложение его на простые множители (не обязательно различные). Тогда символ Якоби $\left(\frac{a}{P}\right)$ определяется равенством:*

$$\left(\frac{a}{P}\right) = \left(\frac{a}{p_1}\right)\left(\frac{a}{p_2}\right)\ldots\left(\frac{a}{p_k}\right),$$

где $\left(\frac{a}{p_i}\right)$ *– обычные символы Лежандра.*

Свойства:

1. Если $a \equiv b \pmod{p}$, то $\left(\frac{a}{P}\right) = \left(\frac{b}{P}\right)$.
2. $\left(\frac{1}{P}\right) = 1$.
3. $\left(\frac{-1}{P}\right) = (-1)^{\frac{P-1}{2}}$.
4. $\left(\frac{2}{P}\right) = (-1)^{\frac{P^2-1}{8}}$.
5. $\left(\frac{a_1 a_2 \ldots a_n}{P}\right) = \left(\frac{a_1}{P}\right)\left(\frac{a_2}{P}\right)\ldots\left(\frac{a_n}{P}\right)$. Следствие. $\left(\frac{ab^2}{P}\right) = \left(\frac{a}{P}\right)$.
6. *Закон взаимности символа Якоби*: если P и Q - нечётные взаимно простые числа, большие единицы, то:

$$\left(\frac{P}{Q}\right)\left(\frac{Q}{P}\right) = (-1)^{\frac{P-1}{2}\cdot\frac{Q-1}{2}}.$$

Квадратичные сравнения по составному модулю

Пусть $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ – многочлен с целыми коэффициентами. Следующая лемма является простым следствием китайской теоремы об остатках, и позволяет сводить степенное сравнение по составному модулю к сравнению по модулю степени простого числа.

Лемма. 5. *Если* $m_1, m_2, \ldots, m_k$ *– попарно взаимно простые числа, то сравнение:*

$$f(x) \equiv 0 \pmod{m_1 m_2 \ldots m_k} \tag{8}$$

равносильно системе сравнений:

$$\begin{aligned} f(x) &\equiv 0 \pmod{m_1}, \\ f(x) &\equiv 0 \pmod{m_2}, \\ &\ldots \\ f(x) &\equiv 0 \pmod{m_k}. \end{aligned} \tag{9}$$

И количество решений сравнения (8) *равно произведению количеств решений отдельных сравнений в* (9).

Теорема. 9. *Пусть* p *– нечетное простое число. Если* p *делит* $f(x)$*, но не делит* $f'(x)$*, то для любого* $\alpha \in \mathbb{N}$ *существует единственное* y *по модулю* p^α *такое, что* $x \equiv y \pmod{p}$ *и* $f(y) \equiv 0 \pmod{p^\alpha}$.

Доказательство этой теоремы довольно сложное и проводится индукцией по α, причем по ходу доказательства используются формула Тейлора для многочленов:

$$f(x+t) = f(x) + f'(x)t + \frac{f''(x)}{2!}t^2 + \ldots + \frac{f^{(n)}(x)}{n!}t^n$$

и тот факт, что многочлен $f(x)$ с целыми коэффициентами $f^{(k)}(x)$ делится на $k!$ при любом целом x.

Теорема. 10. *Пусть p – нечетное простое число, $a \in \mathbb{Z}$, $(a,p) = 1$ и $\alpha \in \mathbb{N}$. Тогда сравнение $x^2 \equiv a \pmod{p^\alpha}$ имеет ровно два решения при $\left(\frac{a}{p}\right) = 1$ и не имеет решений при $\left(\frac{a}{p}\right) = -1$.*

Эта теорема непосредственно следует из определения символа Лежандра и теоремы 9, так как для $f(x) = x^2 - a$ производная $f'(x) = 2x$ не кратна p при x не кратном p.

Теорема. 11. *Пусть a – нечетное число и $\alpha \in \mathbb{N}$. Тогда сравнение $x^2 \equiv a \pmod{2^\alpha}$*

1) *при $\alpha = 1$ имеет ровно одно решение,*
2) *при $\alpha = 2$ имеет ровно 2 решения при $a \equiv 1 \pmod 4$ и ни одного решения при $a \equiv 3 \pmod 4$,*
3) *при $\alpha \geq 3$ имеет ровно 4 решения при $a \equiv 1 \pmod 8$ и ни одного решения в противном случае.*

Основную трудность в этой теореме составляет случай $\alpha \geq 3$. Схема доказательства в этом случае такова: сначала доказывают, что если сравнение $x^2 \equiv a \pmod{2^\alpha}$ разрешимо, то оно имеет ровно 4 решения, причем, если b одно из них, то остальные три, это $-b$, $b + 2^{\alpha-1}$ и $-b + 2^{\alpha-1}$. А затем, используя этот факт, доказывают индукцией по α разрешимость сравнения.

Следующая теорема является простым следствием теорем 10, 11 и леммы 5.

Теорема. 12. *Сравнение $x^2 \equiv a \pmod m$, где $m = 2^\alpha p_1^{\alpha_1} \ldots p_k^{\alpha_k}$ и $(a,m) = 1$, разрешимо тогда и только тогда, когда $\left(\frac{a}{p_1}\right) = 1, \ldots, \left(\frac{a}{p_k}\right) = 1$; $a \equiv 1 \pmod 4$ при $\alpha = 2$, $a \equiv 1 \pmod 8$ при $\alpha \geq 3$. Если ни одно из этих условий не нарушено, то число решений будет 2^k при $\alpha = 0, 1$; 2^{k+1} при $\alpha = 2$; 2^{k+2} при $\alpha \geq 3$.*

Это главный результат настоящего пункта.

Тест Миллера-Рабина

Тест Миллера-Рабина – вероятностный полиномиальный тест простоты. Тест Миллера-Рабина позволяет эффективно определять, является ли данное число составным. Однако, с его помощью нельзя строго доказать простоту числа, поскольку при неудачном выборе параметра a алгоритм может признать простым составное число.

Определение 7. *Пусть m – нечетное число большее 1. Число $m-1$ однозначно представляется в виде $m-1=2^s t$, где t нечетно. Целое число a ($1<a<m$) называется свидетелем простоты числа m, если выполняются два условия:*

1. *m не делится на a*
2. *$a^t \equiv 1 \pmod m$ или существует целое k ($0 \le k < s$) такое, что $a^{2^k t} \equiv -1 \pmod m$.*

Теорема. 13 (Рабин)**.** *Составное нечетное число m имеет не более $\varphi(m)/4$ различных свидетелей простоты.*

Сам алгоритм проверки может быть записан на C++ в следующем виде:

```
for (t = m - 1, s = 0; !(t & 1); s++)
  t>>=2;

a = rand() % (m-2) + 1;
x = a в степени t по модулю m;
if (x == 1 && x == m-1)
  return возможно простое;
for (k=1; k<s; k++)
{
  x = x*x % m;
  if (x==1)
    return составное;
  if (x==m-1)
    return возможно простое;
}
return составное;
```

Как следует из теоремы Рабина, вероятность ошибки этого алгоритма (признания составного числа простым) составляет $\dfrac{\varphi(m)}{4m} < \dfrac{1}{4}$. Для уменьшения этой вероятности представленная проверка повторяется r раз (r ре-

комендуется брать величиной порядка $\log_2 m$). В этом случае вероятность ошибки алгоритма не превзойдет величины 4^{-r}.

Изначальный алгоритм, предложенный Миллером, был детерминированным и состоял в проверке всех a от 2 до $70 \ln^2 m$. Алгоритм Миллера гарантированно распознает простые и составные числа при условии выполнения обобщенной гипотезы Римана. Алгоритм Миллера-Рабина не зависит от справедливости обобщённой гипотезы Римана, но является вероятностным.

Задачи и разборы

Задача A. Две улитки

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Из верхнего левого угла прямоугольного поля, состоящего из M строк и N столбцов, улитка обошла по спирали по часовой стрелке. При этом все клетки она пронумеровала числами 1, 2, 3,... последовательно в порядке обхода.

Теперь на этом поле появляется другая улитка, которой необходимо попасть из клетки (i_1, j_1) в клетку (i_2, j_2). Каждую секунду она может перемещаться в соседнюю по горизонтали или вертикали клетку, но лишь при условии, что номер этой клетки отличается от номера предыдущей не более, чем на k.

Ваша задача – узнать сколько времени потребуется второй улитке, чтобы добраться до нужной клетки.

Формат входного файла

Во входном файле записаны целые числа M, N, k, i_1, j_1, i_2, j_2 ($1 \le M, N \le 10^{18}$, $1 \le k \le 2 \cdot 10^{18}$, $1 \le i_1, i_2 \le M$, $1 \le j_1, j_2 \le N$).

Формат выходного файла

В выходной файл выведите одно целое число – минимальное время, за которое вторая улитка сможет добраться до нужной клетки.

Примеры

a.in	a.out
3 5 1 1 2 3 4	6
5 5 7 3 1 3 3	4

Разбор задачи A. Две улитки

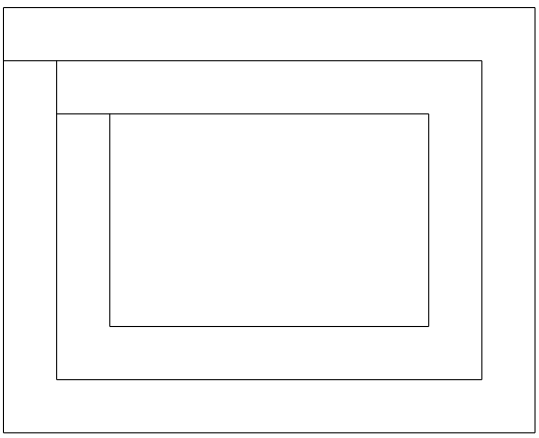
Вариант этой задачи для первой лиги (с размерами матрицы, не превышающими 1000) допускает непосредственное нахождение кратчайшего пути. Для этого достаточно заполнить матрицу числами от 1 до MN по спирали (как указано в условии) и воспользоваться классическим алгоритмом поиска в ширину, для определения длины кратчайшего пути, с учетом возможности перехода (см. например Т.Кормен и др. "Алгоритмы: построение и анализ", раздел 23.2). Сложность алгоритма – $O(MN)$.

В варианте для высшей лиги (когда размеры матрицы могут иметь порядок 10^{18}) это решение недопустимо уже хотя бы потому, что невозможно разместить в памяти матрицу таких размеров. Поэтому необходимо исследовать некоторые дополнительные свойства кратчайших расстояний.

Прежде всего отметим, что без ограничения общности можно считать, что $n_1 \leq n_2$, где n_1 и n_2 – номера (которые были даны первой улиткой) клеток (i_1, j_1) и (i_2, j_2) соответственно. Если это не так, то можно всегда поменять исходную клетку и клетку назначения местами.

Если $k = 1$, то вторая улитка имеет возможность перемещаться лишь вдоль пути первой улитки (то есть по спирали), и поэтому кратчайший (и единственный простой) путь будет иметь длину $n_2 - n_1$. В случае достаточно большого значения k (превышающего все разности между номерами соседних клеток) первая улитка может свободно перемещаться по матрице, и тогда кратчайшее расстояние между двумя заданными клетками будет очевидно равно манхэттеновскому расстоянию между ними – $|i_2 - i_1| + |j_2 - j_1|$.

В общем случае мы имеем некоторую фигуру P_k внутри матрицы, в которой разность номеров соседних клеток не превышает k. Попытаемся найти ее вид. Для удобства представим всю матрицу в виде слоев, состоящих из клеток, находящихся на определенном расстоянии от границы. Это расстояние будем называть уровнем слоя.



На следующей таблице представлена нумерация первых двух слоев.

1	2	3	$\ldots\rightarrow$	$N-2$	$N-1$	N
$2(M+N)-4$	$2(M+N)-3$	$2(M+N)-2$	$\ldots\rightarrow$	$2M+3N-5$	$2M+3N-6$	$N+1$
$\uparrow$ $\vdots$	$\uparrow$				$\vdots$ $\downarrow$	$\vdots$ $\downarrow$
$2M+N-1$	$4M+3N-10$	$4N+3M-9$	$\leftarrow\ldots$	$3(M+N)-8$	$3(M+N)-9$	$N+M-2$
$2N+M-2$	$2N+M-1$	$2N+M-1$	$\leftarrow\ldots$	$N+M+1$	$N+M$	$N+M-1$

Заметим, что разность между номерами клеток верхней полосы слоя первого уровня и соответствующих клеток верхней полосы второго слоя составляет $2M+2N-5$, между правыми полосами этих слоев – $2M+2N-7$, между нижними полосами – $2M+2N-9$, между левыми – $2M+2N-11$. Между верхними полосами второго и третьего слоев разность будет составлять $2M+2N-13$ и т.д. Каждый раз при переходе к следующей полосе разность уменьшается на 2. Так будет продолжаться до тех пор, пока для очередного слоя будет существовать следующий, и в нем будет соответствующая полоса. Если же это не так, то в качестве возможного перехода рассматривается противоположная полоса того же либо следующего слоя.

$MN-2N+M+1$	$MN-2N+M+2$	$\ldots\rightarrow$	$MN-N+M-1$	$MN-N+M$
MN	$MN-1$	$\leftarrow\ldots$	$MN-N+M+2$	$MN-N+M+1$

В такой полосе разность между номерами уменьшается на 2 при переходе к следующей клетке, пока не достигнет значения 1.

Таким образом, видим, что при увеличении номера клетки разность не увеличивается. Поэтому фигура P_k обладает тем свойством, что если клетка с некоторым номером принадлежит фигуре P_k, то ей принадлежат также и все клетки с большими номерами. Аналогично, если некоторая клетка не принадлежит фигуре P_k, то ей не принадлежат и все клетки с меньшими номерами. Это означает, что фигура P_k состоит из всех клеток с номерами не меньшими некоторого значения n_k. Этот номер (и координаты соответствующей клетки) без особого труда можно вычислить за $O(1)$.

Благодаря этому факту, фигура P_k будет выпуклой (в том смысле, что любые две клетки этой фигуры можно соединить путем, состоящим из клеток этой же фигуры, длины равной манхэттеновскому расстоянию между ними).

Теперь нетрудно найти решение задачи (напомним, что мы считаем $(n_1 \leq n_2)$. Если $n_1 \leq n_2 \leq n_k$, то улитка не имеет возможность перемещаться между слоями, и ей придется пройти расстояние $n_2 - n_1$. Если $n_k \leq n_1 \leq n_2$, то улитка находится внутри фигуры P_k и достаточно будет выполнить $|i_2 - i_1| + |j_2 - j_1|$ шагов. Наконец, в случае $n_1 \leq n_k \leq n_2$ улитка должна будет перемещаться по спирали от клетки (i_1, j_1) до клетки (i_k, j_k), на что потребуется $n_k - n_1$ шагов, после чего останется проделать

путь от (i_k, j_k) до (i_2, j_2), который внутри фигуры P_k будет иметь длину $|i_2 - i_k| + |j_2 - j_k|$.

Так как M и N могут достигать значений 10^{18}, то результат может иметь порядок 10^{36}, поэтому нужна длинная арифметика или арифметика двойной точности. В последнем случае значения представляются в виде $x = x_1 \cdot 10^{18} + x_2$, где x_1 и x_2 – числа, не превосходящие 10^{18}, для которых достаточно использовать 64-битный целочисленный тип.

Кроме того, нет смысла сразу вычислять номера клеток. Более простым будет преобразование $(i, j) \leftrightarrow (k, l, p)$, где k – номер слоя, l – номер полосы в слое, p – номер клетки в полосе. В этом варианте потребуется меньше операций над числами двойной точности

Чтобы сохранить сложность $O(1)$ при вычислении разности номеров n_1 и n_2, достаточно научиться быстро вычислять общее количество в слоях на уровнях с k_1 включительно по k_2, не включая последний. Поскольку на уровне k расположено $2M + 2N - 4k$ клеток, то соответствующая сумма легко вычисляется как сумма арифметической прогрессии и равна $2(M + N - k_1 - k_2 + 1)(k_2 - k_1)$.

Задача B. Треугольная комната

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Во многих книгах по занимательной математике приводится такая задача. *Расставить по периметру треугольной комнаты 3 стула так, чтобы у каждой стены стояло по 2. Ее решение — поставить по стулу в каждый из углов комнаты.*

Рассмотрим более общую задачу. Пусть комната представляет собой треугольник ABC. Дано общее количество стульев n, количество стульев n_{AB}, которое должно стоять у стены AB, количество стульев n_{BC}, которое должно стоять у стены BC, количество стульев n_{AC}, которое должно стоять у стены AC. Необходимо найти общее количество расстановок стульев, удовлетворяющих условию. Стулья можно ставить только в углы комнаты и вдоль стен, в центр комнаты стулья ставить нельзя. В любой из углов можно поставить произвольное количество стульев.

Формат входного файла

Входной файл содержит целые числа n, n_{AB}, n_{BC}, n_{AC} ($0 \le n, n_{AB}, n_{BC}, n_{AC} \le 10^{18}$).

Формат выходного файла

В первой строке выходного файла выведите количество различных вариантов расстановки стульев. В случае, когда есть хотя бы один вариант, выведите во второй строке 6 целых неотрицательных чисел: k_A, k_{AB}, k_B, k_{BC}, k_C, k_{AC} – соответственно количество стульев, которые необходимо поставить в угол A, вдоль стены AB, в угол B, вдоль стены BC, в угол C и вдоль стены AC.

Примеры

b.in	b.out
3 2 2 2	1 1 0 1 0 1 0
3 3 2 2	0

Разбор задачи B. Треугольная комната

Задача является некоторым обобщением задачи базового уровня 10-ой олимпиады цикла интернет-олимпиад для школьников 2005-2006 гг. (http://neerc.ifmo.ru/school/io). Легенда полностью сохранена.

Суть задачи заключается в нахождении количества различных целочисленных неотрицательных решений системы уравнений:

$$\begin{cases} k_A + k_B + k_C + k_{AB} + k_{BC} + k_{AC} = n, \\ k_A + k_B + k_{AB} = n_{AB}, \\ k_B + k_C + k_{BC} = n_{BC}, \\ k_A + k_C + k_{AC} = n_{AC}. \end{cases} \qquad (*)$$

Эта система состоит из 4 уравнений и содержит 6 неизвестных. Выбрав какие-либо 2 переменные (к примеру k_A и k_{AB}) и задав им некоторые значения, у нас останутся уже только 4 неизвестные, которые из системы (*) уже определяются однозначно:

$$\begin{cases} k_B = n_{AB} - (k_A + k_{AB}), \\ k_C = n_{BC} + n_{AC} - n + k_{AB}, \\ k_{BC} = n_{BC} - (k_B + k_C), \\ k_{AC} = n_{AC} - (k_A + k_C). \end{cases} \qquad (**)$$

Таким образом, достаточно перебрать все такие значения k_A и k_{AB}, что их сумма не превышает $\min(n_{AB}, n)$. Для каждого такого выбора вычислить по формуле (**) остальные величины, и если все они неотрицательны, то учесть этот вариант, как возможный вариант расстановки. Сложность

такого алгоритма – $O(n^2)$, что вполне приемлемо для варианта первой лиги, но не для высшей.

Чтобы уменьшить сложность, можно рассмотреть на плоскости (k_A, k_{AB}) множество, определяемое неравенствами $k_B(k_A, k_{AB}) \geq 0$, где $k_B(k_A, k_{AB})$ – выражение величины k_B через k_A и k_{AB}, и др. Поскольку все выражения являются линейными функциями, то каждое неравенство определяет полуплоскость, а их пересечение – некоторый многоугольник. Остается лишь вычислить количество целых точек, которые лежат внутри него или на границе.

Однако можно получить более очевидную структуру решений задачи. Преобразуем систему (*) следующим образом. Сложим второе, третье и четвертое уравнение и вычтем из него первое. Это будет первым уравнением новой системы. А затем из полученного уравнения вычтем второе уравнение (*). Результат будет вторым уравнением новой системы. Третье и четвертое уравнение получим также вычитанием из первого уравнения новой системы соответственно третьего и четвертого уравнений системы (*).

$$\begin{cases} k_A + k_B + k_C &= n_{AB} + n_{BC} + n_{AC} - n, \\ k_C - k_{AB} &= n_{BC} + n_{AC} - n, \\ k_A - k_{BC} &= n_{AB} + n_{AC} - n, \\ k_B - k_{AC} &= n_{AB} + n_{BC} - n. \end{cases} \qquad (***)$$

Легко видеть, что ранг системы (***), как и исходной (*), равен 4, следовательно они эквивалентны.

Второе, третье и четвертое уравнения говорят о том, что разность между количеством стульев в вершине и на противолежащей ей стороне является величиной постоянной. Расставим минимально возможное количество стульев, так чтобы удовлетворить только этим соотношениям (если в правой части стоит положительное число, то ставим стулья в вершине, а если отрицательное, то на стороне). Если мы использовали больше стульев, чем нам было дано, то расстановка невозможна. Если же остались стулья (а их будет точно четное количество), то половину из них можно произвольно расставить в вершинах, тогда оставшаяся половина однозначно расставится вдоль противолежащих сторон.

Таким образом, нам необходимо найти количество целых неотрицательных решений уравнения:

$$k_A^+ + k_B^+ + k_C^+ = P,$$

где $P = n_{AB} + n_{BC} + n_{AC} - n - (k_A^0 + k_B^0 + k_C^0)$, k_A^0, k_B^0 и k_C^0 – количество уже расставленных стульев по углам, а k_A^+, k_B^+ и k_C^+ – количество стульев, которые мы добавляем в соответствующие вершины.

Значение k_A^+ мы можем выбрать произвольно от 0 до P. Тогда после выбора k_A^+ для k_B^+ мы имеем $P - k_A^+ + 1$ вариантов (от 0 до $P - k_A^+$), в результате выбора которого k_C^+ получается однозначно. Таким образом, общее количество решений задачи равно:

$$(P+1) + P + (P-1) + \ldots + 1 = \frac{(P+1)(P+2)}{2}.$$

Задача C. Сказки Шахразады

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

...И Шахразаду застигло утро, и она прекратила дозволенные речи...

"Тысяча и одна ночь"

Когда царь Шахрияр узнал об измене своей жены, то вошел он во дворец и отрубил голову своей жене, и рабам, и невольницам. И с тех пор стал он каждый день брать невинную девушку в жены, а потом на утро убивал ее, и так продолжалось в течение трех лет.

И возопили жители Багдада, и бежали со своими дочерьми, и в городе не осталось ни одной девушки.

И вот потом царь приказал своему визирю привести ему, по обычаю, девушку, и визирь вышел и стал искать, но не нашел девушки и отправился в свое жилище, угнетенный и подавленный, боясь для себя зла от царя. А у царского визиря было две дочери: старшая – по имени Шахразада, и младшая – по имени Дуньязада.

Узнав у отца-визиря от начала до конца про все, что у него случилось с царем, Шахразада тот час стала упрашивать отца, чтобы выдал ее за Шахрияра. И не смог визирь отговорить свою дочь и тогда снарядил ее и отвел к царю Шахрияру. А Шахразада подучила сестру свою, чтобы, когда они будут прощаться, та попросила рассказать какую-либо историю.

И так все случилось, и царь, мучившийся бессонницей, обрадовался, и позволил дочери везиря начать свой рассказ. Но когда настало утро, Шахразада прекратила дозволенные речи на самом интересном месте, и Шахрияру ничего не оставалось, кроме как оставить ее в живых, чтобы услышать окончание истории.

Так продолжалось много ночей: Шахразада продолжала недорассказанную сказку прошлой ночи с того места, где остановилась на рассвете, и как только среди ночи заканчивалась одна сказка, Шахразада тут же начинала другую, увлекая царя новым сюжетом. Единственное за чем ей нужно было следить – чтобы сказки не повторялись и (быть может, кроме последней) не заканчивались под утро. Тогда уже либо царь убьет Шахразаду, либо к тому времени настолько полюбит ее, что рука не поднимется. Естественно, чем больше она историй расскажет, тем больше будет любовь Шахрияра, и тем больше шансов остаться в живых.

Формат входного файла

В первой строке задается целое число N ($1 \leq N \leq 10000$) – количество сказок, которые знает Шахразада, а в следующей строке – N натуральных чисел, не превышающих 100 – продолжительности по времени сказок в часах. Считается, что каждая ночь длится 8 часов.

Формат выходного файла

В первую строку выведите наибольшее количество сказок, которое сможет рассказать Шахразада, а во второй – длительности этих сказок, в том порядке, в котором она должна их рассказывать.

Примеры

c.in	c.out
6 1 2 3 4 5 6	6 1 2 3 4 5 6
2 8 9	2 9 8
4 4 4 8 12	3 4 8 12

Разбор задачи C. Сказки Шахразады

В задаче нужно переставить элементы массива $A[1..N]$ таким образом, чтобы среди в последовательности частичных сумм $S[1..N]$, где $S[i] = \sum_{j=1}^{i} a[j]$, не было элементов кратных некоторому числу M или, если это сделать невозможно, то добиться того, чтобы первое кратное M имело максимально возможный номер (по условию $M = 8$, но это не имеет значения для алгоритма).

Прежде всего заметим, что задача существенно не изменится, если все элементы A и число M разделить на одно и то же число d (достаточно будет лишь при выводе снова умножить на d). В качестве такого числа d можно взять НОД$(M, A[1], \ldots, A[N])$. Также не играют роли точные значения $A[i]$, а лишь остатки от деления их на M.

Отметим, что если M после деления на d стало равным 1, это означает, что все элементы A были кратны d и поэтому мы можем использовать лишь один из них. Поэтому в дальнейшем мы рассматриваем случай $M > 1$, причем существует по крайней мере один элемент в A взаимно простой с M.

Мы можем свести задачу к следующей. Есть $r[j]$ элементов со значением j, необходимо выстроить их в такой последовательности, чтобы все частичные суммы этой последовательности по модулю M были отличны от нуля. При этом мы можем расставить сначала все элементы, отличные от 0, а затем в любое место в середине добавить элементы, которые дают остаток 0. Поэтому достаточно рассмотреть как получить расстановку ненулевых остатков.

Рассмотрим сначала случай, когда:

$$2 \cdot \max_{0<j<M} r[j] \leq 1 + \sum_{0<j<M} r[j], \qquad (*)$$

т.е. максимальное из значений $r[i]$ превышает сумму остальных не более, чем на 1. Тогда можно выстроить последовательность элементов A таким образом, что не будет двух одинаковых остатков подряд. Это легко доказать по индукции (каждый раз мы выбираем и ставим элементы, остатки которых соответствуют двум максимальным значениям в массиве r, или, что то же самое, выбирать каждый раз элемент, соответствующий максимальному значению в r отличному от предыдущего).

Полученная, таким образом, последовательность не обязательно будет обладать тем свойством, что частичные суммы будут отличны от 0. Однако ее можно будет легко преобразовать к такому виду. Если некоторая сумма $S[i]$ равна нулю, то поменяв местами i-ый и $(i+1)$-ый элементы последовательности, оставим неизменными все суммы, кроме $S[i]$. Поскольку мы расставляли элементы таким образом, чтобы рядом не было одинаковых остатков, то $S[i]$ обязательно изменится и станет отличным от нуля. При этом может возникнуть ситуация, что на позициях $i+1$ и $i+2$ окажутся одинаковые остатки. Однако в силу того, что мы расставляли лишь ненулевые остатки $S[i+1]$ будет заведомо отлично от нуля. Выполнив такое преобразование во всех позициях, где частичные суммы были равны нулю, мы получим требуемую последовательность.

Теперь перейдем к случаю, когда неравенство (*) не выполняется. Пусть максимальный элемент в r соответствует остатку k. Если НОД$(M, k) \neq 1$, то на первое место в последовательности можно поставить какое либо значение взаимно простое с M (как оговаривалось выше, по крайней мере одно такое значение существует). После этого мы сможем поставить сколько угодно значений с остатком k. Достаточно поставить столько элементов, чтобы после удаления их из последовательности неравенство (*) стало верным, после чего воспользоваться схемой из предыдущего абзаца.

Если же НОД$(M, k) = 1$, то мы можем сначала поставить $M - 1$ элементов с остатком k. Далее если мы ставим число с остатком $i \cdot k \bmod M$, где i – любое значение от 2 до $M - 1$, то после него можно будет записать $M - i$ значений с остатком k. Очевидно, что нет никакого смысла ставить подряд два значения отличных от k до тех пор, пока не выполнится неравенство (*). В противном случае мы сможем избавиться от меньшего количества остатков k. Таким образом, если:

$$r[k] > (M - 1) + \sum_{i=2}^{M-1} (M - i) r[ik] + 1,$$

то мы не сможем выстроить все элементы (последняя единица учитывает возможность поставить значение с остатком k, когда все остальные элементы уже закончатся).

Существует и другой, более простой алгоритм решения задачи. Каждый раз необходимо выбирать остаток, соответствующий максимальному значению в r, не обращающий в 0 текущую частичную сумму. Такой выбор гарантирует максимальное избавление от остатков k, которых слишком много, и обеспечивает сохранение свойства (*), если оно было выполнено раньше (во всяком случае за 2 хода). Сложность этого алгоритма – $O(NM)$, но может быть улучшена до $O(M + N \log M)$, если для хранения значений $r[j]$ использовать кучу.

Задача D. Получить одинаковые

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

На столе лежат карточки, на каждой из которых написано некоторое число. Кроме того, имеется довольно большая стопка чистых карточек. За один ход разрешается убрать со стола две карточки с одинаковыми

числами, затем взять из стопки две чистые, написать на каждой из них по одному числу и положить на стол.

Требуется узнать как за минимальное количество ходов сделать так, чтобы на столе были карточки, на которых написаны одинаковые числа.

Формат входного файла

В первой строке задается количество лежащих на карточек N ($1 \leq N \leq 30000$), а во второй – N чисел, написанных на них. Все числа натуральные и не превышают 10^9.

Формат выходного файла

В первой строке выведите минимальное количество операций L, необходимых для того, чтобы получить на столе все одинаковые карточки. В последующих L строках выведите по четыре числа, определяющие, что нужно делать в соответствующий ход: первое и второе – числа, записанные на убираемых карточках, третье и четвертое – числа, которые нужно написать на новых карточках. Если невозможно сделать так, чтобы на столе оказались все одинаковые карточки, выведите одно число -1.

Примеры

d.in	d.out
3 1 2 2	1 2 2 1 1
6 3 3 3 4 4 4	2 3 3 4 3 3 3 4 4

Разбор задачи D. Получить одинаковые

Очевидно, что получить одинаковые карточки невозможно, если все исходные карточки были различными, поскольку мы не можем выполнить ни одного хода. Если же есть хотя бы одна пара одинаковых карточек, то всегда существует последовательность ходов, при помощи которой можно получить требуемый результат. Построим эту последовательность.

Итак, у нас есть две одинаковые карточки – пусть это будут первая и вторая карточки. Уберем их со стола и положим вместо первой карточку на которой будет, на которой будет написано число 0, а вместо второй – карточку, на которых напишем такое же число, что на третьей карточке. Теперь убираем вторую и третью, на новой второй пишем 0, а на третьей то же число, что на четвертой. На каждом шаге у нас количество карточек

с числом 0 увеличивается, и при этом обеспечивается существование еще одной пары одинаковых карточек. На последнем шаге убираем две последние карточки и кладем вместо них две карточки со значением 0. Всего мы выполним $N - 1$ ходов.

Очевидно этот алгоритм не дает оптимальное количество ходов, но для варианта задачи в первой лиге этого и не требуется. В оценке вычислительной сложности алгоритма преобладает время, связанное с поиском одинаковых элементов, что нельзя сделать за время меньшее $O(N \log N)$.

Перейдем теперь к построению оптимального алгоритма. Заметим, что нет смысла получать на карточках числа, отличные от тех, которые уже были записаны хотя бы на одной из них. Действительно пусть существует такая последовательность ходов, в которой используется некоторое значение x. Тогда, заменив в этой последовательности значения x на к примеру значение $A[1]$, мы не ухудшим последовательность. Более того, мы ее можем улучшить. Последним ходом в этой последовательности будет замена двух карточек с некоторым числом y на две карточки x. Заменив во всех операциях x на y, мы сохраним корректную последовательность и сможем не выполнять последний ход.

Теперь пусть у нас есть s_i карточек со значением i и мы будем собирать карточки x. Очевидно, что невозможно добиться того, чтобы записать на всех карточках значения x, меньше чем за $(N - s_x)/2$ ходов. При этом если все значения s_i четные, то этой величины можно добиться, заменяя карточками x пары других одинаковых карточек. Если же есть некоторое количество (обозначим его l) нечетных значений s_i, каждое из них (на каком-либо этапе) нужно будет преобразовать в четное. Для преобразования пары таких значений s_i и s_j необходимо будет взять любую имеющуюся в наличии пару одинаковых карточек и положить вместо них карточки i и j. Поэтому необходимо будет еще по крайней мере $l/2$ ходов. Если значение l было нечетным, то для преобразования последнего нечетного числа s_i в четное потребуется еще одно действие (любую пару одинаковых карточек заменяем на i и x). Наконец, может потребоваться еще одно дополнительное действие, если нет ни одной пары одинаковых карточек, кроме x. В этом случае придется для выполнения необходимой операции снять две карточки с числом x, что уменьшит значение s_x на 1 или 2. Таким образом, для каждого числа x мы можем точно определить количество шагов. Это количество зависит от значений s_x и l, при этом, чем больше s_x, тем меньше количество шагов, а l равно количеству нечетных s_i (при четном s_x) или на 1 меньше (при нечетном s_x). Поэтому в качестве x достаточно рассмотреть значение соответствующее максимальному четному и нечетному s_i.

Сложность алгоритма определяется сложностью получения значений s_i,

что является, по существу, сортировкой исходного массива A. Поэтому она имеет оценку $O(N \log N)$.

Задача E. Прыжки на полосе

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Горизонтальная полоса $1 \times N$ разбита на клетки размера 1×1. В клетке с номером s стоит шашка. В первый ход она должна переместиться на одну клетку (влево или вправо). Второй ход должен быть сделан уже на 2 клетки и т.д. Каждый следующий ход выполняется с шагом, на единицу больше, чем предыдущий. Так продолжается до тех пор, пока у шашки не останется ходов, не выводящих ее за пределы полосы. Определите какое максимальное и минимальное число ходов может сделать шашка.

Формат входного файла

В первой строке входного файла задано количество тестов T ($1 \le T \le 20$). В каждой из последующих T строк заданы по два целых числа N и s ($1 \le N \le 10^{18}$, $1 \le s \le N$), определяющих данные для соответствующего теста.

Формат выходного файла

Выходной файл должен состоять из T строк, в каждой из которых содержится по два целых числа – максимальное и минимальное количество ходов шашки для соответствующего теста.

Пример

e.in	e.out
2 3 2 6 2	2 2 5 3

Разбор задачи E. Прыжки на полосе

В рамках ограничений первой лиги можно воспользоваться динамическим программированием. Для этого определим функцию $F(i, k)$ – величина последнего прыжка в максимальной последовательности, если шашка находится в клетке с номером i и текущий прыжок должен быть выполнен

на k клеток. Эта функция может быть вычислена с помощью рекуррентной формулы $F(i,k) = \max(k-1, F(i-k,k+1), F(i+k,k+1))$. При выходе первого аргумента за границы поля считаем функцию F равной 0. Вычисляя значения этой функции от больших значений к меньшим, в конечном итоге найдем величину $F(s,1)$, что является ответом на поставленный в задаче вопрос. Аналогичную формулу можно составить и для минимальной последовательности. Сложность алгоритма – $O(N^2)$.

В варианте задачи для высшей лиги нужно определять требуемые значения за $O(1)$. Рассмотрим максимальное количество. Очевидно, что мы не можем добиться количества прыжков больше $N-1$. Для того, чтобы обеспечить такое количество прыжков, необходимо и достаточно, чтобы существовало такое число $k \le N/2$, чтобы к моменту, когда выполняется прыжок на k клеток, шашка находилась в клетке $\lceil (N+1-k)/2 \rceil$ или $\lfloor (N+1+k)/2 \rfloor$. Тогда, осуществляя в дальнейшем колебания относительно центра полосы, мы приведем шашку в одну из крайних клеток, выполнив в общей сложности $N-1$ прыжков.

Таким образом, достаточно научиться быстро проверять можно ли попасть из начальной вершины в заданную за $k-1$ прыжков. Перенумеруем клетки таким образом, чтобы начальная клетка получила номер 0, а клетка, в которую необходимо попасть – положительный номер d (в случае необходимости можно нумеровать клетки в обратном направлении). Если существует клетка с номером -1, то в клетку d можно попасть тогда и только тогда, когда d не превосходит $k(k-1)/2$ и имеет с ним одинаковую четность. Этот факт может быть доказан по индукции. Для $k=1$ он проверяется тривиальным образом. При переходе к значению $k+1$ достаточно убедиться в том, что в каждую клетку x от -1 до $(k+1)k/2$ соответствующей четности можно попасть из одной из клеток от -1 до $k(k-1)/2$, т.е. выполняется по крайней мере одно из неравенств $x-k \ge -1$ или $x+k \le k(k-1)/2$. Это возможно лишь тогда, когда $-1+2k \le k(k-1)/2$. После преобразований получим неравенство $k^2-3k+2 \ge 0$, которое верно для всех целых k.

Если N – четное, то клетки $\lceil (N+1-k)/2 \rceil$ и $\lfloor (N+1+k)/2 \rfloor$ имеют разную четность, и поэтому в одну из них обязательно будет возможность добраться. Если же N нечетное, то указанные клетки будут иметь одну четность, и в них мы сможем попасть только из клеток определенной четности. Можно показать, что из другой четности мы не может попадать в требуемые клетки. Зато можно рассмотреть всю полосу без одной клетки — на ней по доказанному выше мы можем попасть на клетки, из которых можно начать колебания относительно центра. Таким образом, ответом всегда является число $N-1$ либо $N-2$.

Мы рассмотрели лишь случай, когда исходная клетка не является край-

ней. В случае же когда нет запасной клетки -1, при $k \geq 10$ мы имеем преодолеть расстояние x в пределах от 0 до $k(k-1)-6$ и ровно на $k(k-1)/2$. Значения $k \geq 10$ рассматриваются лишь при $N \geq 20$, при этом диапазона $k(k-1)/2-6$ вполне достаточно, чтобы покрыть не только половину, но и всю полосу целиком. Поэтому при $N \geq 20$ для крайних клеток мы находимся в рамках общего случая, а при $N < 20$ этот вариант можно рассмотреть отдельно – с помощью динамического программирования. Как оказывается, при некоторых значениях N из крайних клеток максимальное число прыжков оказывается на 1 меньше стандартного значения.

Теперь рассмотрим минимальное количество. Ясно, что последний ход не может быть меньше чем $N/2$. Поскольку при меньших значениях обязательно есть из любой клетки, по крайней мере, есть один ход. При четных N нам необходимо попасть за $N/2$ ходов на одну из клеток $N/2$ или $N/2+1$, что в силу разной их четности всегда можно сделать. Для нечетного N прыжок $(N-1)/2$ будет последним, если будет возможность закончить этот ход на центральной клетке. Если это невозможно, то можно попытаться добраться за $(N+1)/2$ прыжков до клеток соседних с центральной (до самой центральной добраться за столько прыжков невозможно, поскольку клетки на этом расстоянии находятся за пределами полосы). Наконец, если и это не возможно сделать, то попытаться добраться за $(N+3)/2$ прыжков до одной из следующих двух клеток. Оказывается, что это уже всегда будет возможно, поскольку эти две клетки имеют ту же четность что и центральная, а значения $k(k-1)/2$ для k, отличающихся на 2 – разную. Исключение могут составить, опять-же, крайние клетки-полосы при $N < 20$, для которых минимальное количество может увеличиться на 1.

Доводя все вычисления до конца, можно убедиться в том, что какой из случаев имеет место зависит от четности s и остатка от деления N на 8 (для максимального количества прыжков достаточно рассмотреть лишь остаток от деления N на 4).

Задача F. Произведение взаимно простых

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Вам дано натуральное число m. Требуется найти произведение всех натуральных чисел не превосходящих m и взаимно простых с m, и выдать его остаток при делении на m.

Формат входного файла

В единственной строке входного файла задано натуральное число $m \leq 10^{18}$.

Формат выходного файла

В единственную строку выходного файла выведите ответ на задачу.

Примеры

f.in	f.out
1	0
2	1
3	2
4	3
5	4
6	5
7	6

Разбор задачи F. Произведение взаимно простых

При $m \leq 2$ ответ $m-1$, что проверяется непосредственным вычислением. Пусть $m > 2$. Разобьём все числа a от 1 до m, взаимно простые с m, на два класса: для которых $a = \mathrm{inv}_m(a)$ и для которых $a \neq \mathrm{inv}_m(a)$. Здесь $\mathrm{inv}_m(a)$ – обратный остаток числа a по модулю m. Это определение вводилось в лекции. Числа второго класса разбиваются на пары (a, b), где $b = \mathrm{inv}_m(a)$. Поэтому произведение чисел второго класса дает остаток 1 при делении на m. А значит, ответ на задачу – это остаток при делении на m произведения чисел первого класса. Числа этого класса — это все решения сравнения $a^2 \equiv 1 \pmod{m}$. Они тоже разбиваются на пары $(a; m-a)$, причем $a \neq m-a$ для любой такой пары, так как $m > 2$. Так как для чисел второго класса $a(m-a) \equiv -a^2 \equiv -1 \pmod{m}$, то ответ на задачу это $(-1)^{N/2} \bmod m$, где N – количество решений сравнения $a^2 \equiv 1 \pmod{m}$. По теореме из лекции о количестве решений квадратичного сравнения по составному модулю получаем, что при $m = 4, p^n, 2p^n$, где p – нечетное простое число, а $n \in \mathbb{N}$, искомое количество решений равно 2, а в остальных случаях это степень двойки, большая двух, и, значит, кратная 4. Поэтому, при $m = 4, p^n, 2p^n$ ответ равен $(-1)^{2/2} \bmod m = m-1$, а при остальных $m > 2$ ответ 1.

Итак, разобрав отдельно случай $m = 4$, и поделив, если нужно, m на 2, задача сводится к выяснению того, является ли m степенью нечетного

простого числа. Сначала проверяем является ли m простым. В варианте для первой лиги это можно делать методом последовательных проб со сложностью работы $O(\sqrt{m})$. Для высшей лиги этот алгоритм неприемлем и нужно использовать тест Миллера-Рабина, описанный в лекции. Далее нужно последовательно перебирать n, начиная с 2, извлекать из m корень n-ой степени, и если он целый, то проверять его на простоту. Здесь уже в обоих лигах можно использовать метод последовательных проб так как извлеченный корень не превосходит 10^9. Выход из цикла по n можно делать, если, например, извлеченный корень стал меньше 3. Таким образом, сложность алгоритма для первой лиги – $O(\sqrt{m})$, а для высшей – $O(\sqrt[4]{m})$, либо $O(r \log^2 m)$ если использовать тест Миллера-Рабина для всех проверок на простоту (здесь r – количество проб в тесте Миллера-Рабина).

Задача G. Делимость биномиальных коэффициентов

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Обозначим $C_n^i = \dfrac{n!}{i!(n-i)!}$, где $0 \le i \le n$ и n, i – целые числа. Вам даны натуральное число n и простое число p. Обозначим через k наибольшее целое неотрицательное число, для которого $p^k \le n$. Далее обозначим для $j \ge 0$ через a_j количество чисел $i \in \{0, 1, \ldots, n\}$, для которых C_n^i делится на p^j, но не делится на p^{j+1}. Легко проверить, что $a_j = 0$ при $j > k$. Поэтому, от вас требуется найти числа $a_0, a_1, \ldots, a_k$.

Формат входного файла

В единственной строке входного файла заданы натуральное число $n \le 10^{18}$ и простое число $p < 10^{18}$.

Формат выходного файла

В единственную строку выходного файла выведите через пробел числа $a_0, a_1, \ldots, a_k$.

Примеры

g.in	g.out
4 2	2 1 2
8 3	9 0
4 5	5

Разбор задачи G. Делимость биномиальных коэффициентов

Согласно формуле из лекции:

$$\deg_p(n!) = \frac{n - S_p(n)}{p-1},$$

где $\deg_p(m)$ – показатель вхождения числа p в каноническое разложение числа m на простые множители, а $S_p(n)$ – сумма цифр числа n в p-ичной записи.

Поэтому:

$$\deg_p(C_n^i) = \deg_p(n!) - \deg_p(i!) - \deg_p((n-i)!) = \frac{S_p(i) + S_p(n-i) - S_p(n)}{p-1}.$$

Из этой формулы следует, что $\deg_p(C_n^i)$ равно количеству переносов в следующий разряд при сложении чисел i и $n-i$ в столбик в p-ичной записи.

В варианте для первой лиги из этого наблюдения следует, что надо найти количество тех $i \in \{0, 1, \ldots, n\}$, для которых при сложении чисел i и $n-i$ в столбик в p-ичной записи не происходит переносов в следующий разряд. Разложим числа n, i и $n-i$ в p-ичную запись: $n = (\overline{a_k \ldots a_1 a_0})_p$, $i = (\overline{b_k \ldots b_1 b_0})_p$, $n-i = (\overline{c_k \ldots c_1 c_0})_p$. Тогда должны выполняться равенства $a_j = b_j + c_j$ $(0 \le j \le k)$. При фиксированном $j \in \{0, 1, \ldots, k\}$ количество способов выбрать $b_j, c_j \in \{0, 1, \ldots, p-1\}$, для которых $b_j + c_j = a_j$, равно $a_j + 1$. А так как для каждого конкретного j выбор не зависит от других j, то общее число способов равно $(a_0 + 1)(a_1 + 1) \ldots (a_k + 1)$. Это и есть ответ на задачу, так как p-ичная запись задает число однозначно.

Рассмотрим теперь вариант для высшей лиги. Требуется для каждого $h \in \{0, 1, \ldots, k\}$ найти количество тех $i \in \{0, 1, \ldots, n\}$, для которых количество переносов в следующий разряд при сложении чисел i и $n-i$ в столбик в p-ичной записи равно h. Введем в рассмотрение функции $f(j, h)$ и $g(j, h)$.

$f(j, h)$ – это количество пар наборов $(b_k, \ldots, b_{j+1}, b_j)$ и $(c_k, \ldots, c_{j+1}, c_j)$, для которых:

- $b_k, \ldots, b_{j+1}, b_j, c_k, \ldots, c_{j+1}, c_j \in \{0, 1, \ldots, p-1\}$,
- $(\overline{b_k \ldots b_{j+1} b_j})_p + (\overline{c_k \ldots c_{j+1} c_j})_p = (\overline{a_k \ldots a_{j+1} a_j})_p$,
- в этом сложении происходит ровно h переносов в следующий разряд.

Аналогичный смысл имеет функция $g(j, h)$, но в ней сумма равна $(\overline{a_k \ldots a_{j+1} a_j})_p - 1$.

Эти функции определены при $0 \le j \le k$ и всех h, но удобно считать, что при остальных значениях j они равны нулю при всех h, кроме значения $f(k+1,0)$, которое равно 1 (это можно объяснить так: два пустых набора в сумме дают пустой набор и не происходит переносов).

Легко видеть, что ответом на задачу будут значения $f(0,h), 0 \le h \le k$ (как и в варианте для первой лиги это следует из того, что $p-$ичная запись задает число однозначно).

Выразим $f(j,h)$ через предыдущие значения функций f и g.

Рассмотрим $j-$тый разряд. Если из него не происходит переноса, то должны выполняться равенства $b_j + c_j = a_j$ и $(\overline{b_k \dots b_{j+1}})_p + (\overline{c_k \dots c_{j+1}})_p = (\overline{a_k \dots a_{j+1}})_p$. Причем, в последнем сложении должно произойти ровно h переносов. Легко видеть, что всего таких пар наборов будет $(a_j + 1)f(j+1,h)$. Если же из $j-$того разряда происходит перенос, то должны выполняться равенства $b_j + c_j = p + a_j$ и $(\overline{b_k \dots b_{j+1}})_p + (\overline{c_k \dots c_{j+1}})_p = (\overline{a_k \dots a_{j+1}})_p - 1$. Причем, в последнем сложении должно произойти ровно $h-1$ переносов. Таких наборов будет $(p-1-a_j)g(j+1,h-1)$.

В итоге получаем, что:

$$f(j,h) = (a_j + 1)f(j+1,h) + (p-1-a_j)g(j+1,h-1).$$

Аналогично:

$$g(j,h) = a_j f(j+1,h) + (p-a_j)g(j+1,h-1).$$

Теперь в цикле по j от k до 1 с шагом -1, исходя из известных значений $f(k+1,h)$ и $g(k+1,h)$, мы найдем ответ на задачу. Сложность алгоритма $O(k^2) = O(\log^2 n)$.

Задача H. Попарно различные расстояния

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Для данного натурального N требуется построить на плоскости множество из N точек с целыми координатами, все попарные расстояния между которыми попарно различны.

Формат входного файла

В единственной строке входного файла задано натуральное число $N \le 200$ – количество точек в множестве.

Формат выходного файла

В выходной файл выведите координаты точек построенного множества (каждую точку в отдельной строке, координаты через пробел). Координаты точек не должны превышать по модулю 800. Гарантируется, что такое множество существует. Если таких множеств несколько, можно выдать любое.

Примеры

h.in	h.out
1	800 -800
2	0 0 0 1
4	0 0 0 1 0 3 0 7

Разбор задачи H. Попарно различные расстояния

Ясно, что задачу достаточно решить для $N = 200$, а для других N просто брать первые N точек найденного множества из 200 точек. Рассмотрим следующий жадный алгоритм построения требуемого множества. Пронумеруем целые точки плоскости, начиная с $(0, 0)$, по спирали, как показано на рисунке:

$\cdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$
$\cdots$	25	10	11	12	13	$\cdots$
$\cdots$	24	9	2	3	14	$\cdots$
$\cdots$	23	8	1	4	15	$\cdots$
$\cdots$	22	7	6	5	16	$\cdots$
$\cdots$	21	20	19	18	17	$\cdots$
$\cdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$

Будем просматривать точки в порядке возрастания номеров и поддерживать текущее множество точек с попарно различными попарными расстояниями между ними. Очередную точку будем добавлять в множество, если ее добавление не нарушает этого свойства. Это легко можно проверить поддерживая вместе с множеством массив пометок попарных расстояний между точками текущего множества. Но следует учитывать, что расстояния от очередной точки до всех предыдущих могут отличатся от попарных расстояний между предыдущими точками, но сама точка при этом может

быть равноудалена от каких-то двух предыдущих точек и поэтому ее добавлять нельзя. Если добавить точку не удается, то в дальнейшем мы к ней не возвращаемся (в этом и заключается жадность). Непосредственная реализация этого алгоритма показывает, что координаты первых 200 точек, добавленных в множество, не превосходят по абсолютной величине 800. Поэтому построенное таким способом множество действительно удовлетворяет всем условиям задачи. При этом программа укладывается в две секунды. Поэтому нет необходимости создавать константный массив с построенным множеством. Но и такое решение тоже вполне допустимо для этой задачи.

Задача I. Сумма двух квадратов

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Как известно, любое простое число p вида $4k + 1$ представимо в виде суммы двух квадратов натуральных чисел, причем единственным способом. В данной задаче вам предлагается найти такое представление. Чтобы облегчить задачу, будут рассматриваться только простые числа вида $8k+5$.

Формат входного файла

В первой строке входного файла задано натуральное число $T \leq 1000$, количество простых чисел вида $8k + 5$, которые вам надо представить в виде суммы двух квадратов натуральных чисел. В последующих T строках заданы сами эти числа. Гарантируется, что каждое из них является простым числом, дает остаток 5 при делении на 8 и не превосходит 10^{18}.

Формат выходного файла

Для каждого простого числа p из входного файла выведите в отдельной строке через пробел пару натуральных чисел x и y такую, что $x < y$ и $x^2 + y^2 = p$.

Примеры

i.in	i.out
4 5 13 29 9999999999999989	1 2 2 3 2 5 260483990 965478167

Разбор задачи I. Сумма двух квадратов

Пусть $x = 2^{2k+1} \pmod p$. Тогда по формуле Эйлера для квадратичных вычетов и закону взаимности Гаусса, которые приведены в лекции, имеем:

$$x^2 \equiv 2^{4k+2} \equiv 2^{\frac{p-1}{2}} \equiv \left(\frac{2}{p}\right) \equiv (-1)^{\frac{p^2-1}{8}} \equiv (-1)^{8k^2+10k+3} \equiv -1 \pmod p.$$

Полагая $y = 1$, получим что $x^2 + y^2 = mp$, где $m \in \mathbb{N}$ и $m < p$.

Теперь покажем, что по паре натуральных чисел $(x; y)$, для которой $x^2 + y^2 = mp$, где $1 < m < p$, можно построить пару натуральных чисел $(X; Y)$, для которой $X^2 + Y^2 = Mp$, где $1 \le M \le m/2$.

Если x и y оба делятся на m, то из равенства $x^2 + y^2 = mp$ следует, что p делится на m, что невозможно, так как $1 < m < p$.

Пусть $c, d \in \mathbb{Z}$ таковы, что $x = cm + x_1$, $y = dm + y_1$ и $|x_1|, |y_1| \le m/2$. Например, можно взять $c = [x/m + 1/2]$ и $d = [y/m + 1/2]$.

Так как хотя бы одно из чисел x и y не делится на m, то $0 < x_1^2 + y_1^2 \le 2(m/2)^2 = m/2 \cdot m$. А так как $x_1^2 + y_1^2 \equiv x^2 + y^2 \equiv 0 (\mathrm{mod}\, m)$, то $x_1^2 + y_1^2 = Mm$, где $1 \le M \le m/2$.

Далее имеем:

$$m^2 Mp = mp \cdot Mm = \left(x^2 + y^2\right)\left(x_1^2 + y_1^2\right) = (xx_1 + yy_1)^2 + (xy_1 - yx_1)^2.$$

Но:

$$xx_1 + yy_1 = x(x - cm) + y(y - dm) = x^2 + y^2 - m(cx + dy) = m(p - cx - dy) = mX$$

и

$$xy_1 - yx_1 = x(y - dm) - y(x - cm) = xy - yx - mdx + mcy = m(cy - dx) = mY.$$

Откуда $X^2 + Y^2 = Mp$, где $1 \le M \le m/2$. Если $X = 0$, то $Y^2 = Mp$, откуда Y делится на p, откуда M делится на p, что невозможно. Аналогично $Y \ne 0$. Поэтому, заменяя X на $|X|$, а Y на $|Y|$, получим требуемую пару натуральных чисел.

Таким образом, за один шаг мы уменьшаем m не менее, чем в два раза. Поэтому за $O(\log p)$ шагов мы получим пару $(x; y)$, для которой $x^2 + y^2 = p$.

Рассмотрим нюансы реализации.

Вначале нужно вычислить x по формуле $2^{2k+1} \bmod p$. Это можно сделать бинарным алгоритмом возведения в степень за $O(\log p)$ операций, который описан в лекции. При этом так как p может достигать порядка 10^{18}, то необходимо использовать длинную арифметику.

Далее идет цикл, одна итерация которого состоит в следующем:

1. По известным x и y вычислить m по формуле $m = (x^2 + y^2)/p$. Чтобы это сделать необходимо опять использовать длинную арифметику.
2. Если $m = 1$, то выходим из цикла — требуемая пара $(x; y)$ найдена.
3. Вычислить c и d по формулам $c = [x/m + 1/2]$, $d = [y/m + 1/2]$.
4. Наконец, заменить x и y на $|p - cx - dy|$ и $|cy - dx|$.

На 3-м и 4-м шаге все вычисления можно проводить в стандартном знаковом 64-битном целочисленном типе.

Задача J. Игра S-Грюнди

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Пусть S – некоторое множество целых неотрицательных чисел. Построим по нему последовательность $G_S(n)$ ($n > 0$) по следующей формуле:

$$G_S(n) = \operatorname{mex}\{G_S(x) \oplus G_S(y) : x, y > 0,\ x + y = n,\ |x - y| \notin S\},$$

где $\operatorname{mex} A$ – наименьшее целое неотрицательное число, не принадлежащее множеству A, а $a \oplus b$ – результат побитового сложения чисел a и b по модулю 2.

Будем называть множество S хорошим, если $G_S(n + 4) = G_S(n)$ при $n > 4$, а значения $G_S(n)$ при $n = 1, 2, 3, 4, 5, 6, 7, 8$ равны $0, 0, 0, 1, 0, 2, 1, 3$ соответственно.

Дано $n \geq 0$. Требуется найти количество хороших множеств, элементы которых не превосходят n, по модулю $10^9 + 7$.

Формат входного файла

В единственной строке входного файла задано целое неотрицательное число $n \leq 10^{18}$.

Формат выходного файла

В единственную строку выходного файла выведите ответ на задачу.

Примеры

j.in	j.out
0	0
1	1
2	1
3	1
4	1

Разбор задачи J. Игра S-Грюнди

Выполняется следующий критерий того, что множество S является хорошим.

Теорема. 14. *Множество S является хорошим тогда и только тогда, когда выполнены следующие условия:*

1) $0, 1 \in S$,

2) $2, 3, 4, 6, 8 \notin S$,

3) *для любого натурального n либо $4n - 1 \notin S$, либо $4n + 3 \notin S$,*

4) *для любого натурального n либо $4n - 4 \notin S$, либо $4n \notin S$.*

Введем последовательность $a(n)$ ($n \in \mathbb{N}$) следующим образом: значения $a(n)$ при $n = 1, 2, 3, 4, 5, 6, 7, 8$ равны $0, 0, 0, 1, 0, 2, 1, 3$ соответственно и $a(n + 4) = a(n)$ при $n > 4$. Тогда множество S будет хорошим тогда и только тогда, когда $G_S(n) = a(n)$ при всех $n \in \mathbb{N}$.

Теперь докажем теорему 14.

Докажем сначала, что если S хорошее множество, то оно удовлетворяет условиям (1), (2), (3), (4).

Если $0 \notin S$, то $G_S(2) = \operatorname{mex}\{G_S(1) \oplus G_S(1)\} = \operatorname{mex}\{0\} = 1 \neq a(2) = 0$. Значит, $0 \in S$. Проводя аналогичное рассуждение для $G_S(3)$, получим, что $1 \in S$. Значит, S удовлетворяет условию (1).

Так как $G_S(1) \oplus G_S(7) = 1$, $G_S(2) \oplus G_S(6) = 2$, $G_S(3) \oplus G_S(5) = 0$, то $G_S(8)$ будет равно 3, только в случае, если $2, 4, 6 \notin S$. Проводя аналогичное рассуждение для $G_S(3)$, получим, что $8 \notin S$. Похожий анализ значения $G_S(7)$ показывает, что $3 \notin S$. Значит, S удовлетворяет условию (2).

Проверим теперь выполнимость условия (3). При $n = 1$ оно выполняется в силу того, что $3 \notin S$. Пусть $n \geq 2$. Рассмотрим значение $G_S(4n + 7)$. Так как $G_S(4n + 7) = a(4n + 7) = 1$, то должно существовать k, для которого $1 \leq k \leq 4n + 7 - k$, $4n + 7 - 2k \notin S$ и $a(k) \oplus a(4n + 7 - k) = 0$. При $k \geq 5$

равенство $a(k) \oplus a(4n+7-k) = 0$ невозможно (если $a(k) \oplus a(4n+7-k) = 0$, то $a(k) = a(4n+7-k)$, откуда $4n+7-2k \vdots 4$, что невозможно). При $1 \le k \le 4$ это равенство выполняется лишь при $k = 2, 4$. Поэтому либо $4n+7-2 \cdot 2 = 4n+3$, либо $4n+7-2 \cdot 4 = 4n-1$ принадлежит множеству S. Тем самым свойство (3) доказано.

Похожим образом проверяется условие (4). При $n = 1, 2, 3$ оно выполняется, так как $4, 8 \notin S$. Пусть $n \ge 4$. Рассмотрим значение $G_S(4n+4)$. Так как $G_S(4n+4) = a(4n+4) = 3$, то должно существовать k, для которого $1 \le k \le 4n+4-k$, $4n+4-2k \notin S$ и $a(k) \oplus a(4n+4-k) = 2$. При $k \ge 5$ число $a(k) \oplus a(4n+4-k)$ равно либо 0, либо 1, поэтому такие k не подходят. При $1 \le k \le 4$ равенство $a(k) \oplus a(4n+4-k) = 2$ выполняется лишь при $k = 2, 4$. Поэтому либо $4n+4-2 \cdot 2 = 4n$, либо $4n+4-2 \cdot 4 = 4n-4$ принадлежит множеству S. Тем самым свойство (4) доказано.

Таким образом, в одну сторону теорема доказана.

Докажем теперь, что если множество S удовлетворяет условиям (1), (2), (3), (4), то оно является хорошим, то есть, что $G_S(n) = a(n)$ при $n \ge 1$.

При $1 \le n \le 12$ равенство $G_S(n) = a(n)$ легко проверяется из определения последовательности $G_S(n)$ и свойств (1) и (2) (при вычислении значений $G_S(7)$, $G_S(9)$ и $G_S(2)$ оказывается неважным принадлежат или нет множеству S числа 5, 7, 9).

Чтобы доказать равенство $G_S(n) = a(n)$ при $n \ge 13$, нам понадобится следующая:

Лемма. 6. *Для всех $n \ge 9$ выполнены соотношения:*

$$a(k) \oplus a(n-k) \ne a(n), \quad 1 \le k \le n-k. \tag{10}$$

Доказательство. Ясно, что $a(n-1), a(n-2), a(n-3) \ne a(n)$ и $a(n-4) = a(n)$ при $n \ge 9$. Так как $a(1) = a(2) = a(3) = 0$ и $a(4) = 1$, то из этого следует соотношение (10) при $1 \le k \le 4$.

Так как $a(n)$ имеет период 4 при $n \ge 5$, то при $5 \le k \le n-k$:

$$a(k) \oplus a(n-k) = a(l) \oplus a(n-l),$$

где $5 \le l \le 8$ и $k - l \vdots 4$. Поэтому достаточно проверить (10) лишь при $5 \le k \le 8$. Это легко сделать, используя явный вид последовательности $a(n)$, в зависимости от остатка числа n при делении на 4. □

Будем доказывать равенство $G_S(n) = a(n)$ индукцией по n. При $1 \le n \le 12$ оно уже доказано. Пусть $n \ge 13$ и для всех меньших n оно выполняется (предположение индукции). Так как $n \ge 13$, то по лемме 6:

$$G_S(k) \oplus G_S(n-k) = a(k) \oplus a(n-k) \ne a(n), \quad 1 \le k \le n-k.$$

Поэтому $a(n) \notin A_n = \{G_S(x) \oplus G_S(y) : x, y > 0,\ x + y = n,\ |x - y| \notin S\}$.
Докажем, что:

$$0, 1, \ldots, a(n) - 1 \in A_n. \tag{11}$$

Тогда из этого будет следовать, что $G_S(n) = \operatorname{mex} A_n = a(n)$ по определению операции mex. Разберем 4 случая:

1) $n = 4m + 1$. Здесь $a(n) = 0$ и условие (11) выполнено автоматически.

2) $n = 4m + 2$. В этом случае $a(n) = 2$ и надо проверить, что $0, 1 \in A_n$.

Так как $(2m + 3) + (2m - 1) = n$ и $(2m + 3) - (2m - 1) = 4 \notin S$ по условию (2), то $G_S(2m-1) \oplus G_S(2m+3) = a(2m-1) \oplus a(2m+3) = 0 \in A_n$ ($m \geq 3$ при $n \geq 13$, поэтому $a(2m - 1) = a(2m + 3)$).

Так как $(2m + 2) + 2m = n$ и $(2m + 2) - 2m = 2 \notin S$ по условию (2), то $G_S(2m) \oplus G_S(2m + 2) = a(2m) \oplus a(2m + 2) = 2 \oplus 3 = 1 \in A_n$ (так как $m \geq 3$, то $a(2m)$ и $a(2m + 2)$ равны 2 и 3 в каком-то порядке).

Итак, условие (11) в данном случае выполнено.

3) $n = 4m + 3$. Здесь $a(n) = 1$ и надо проверить, что $0 \in A_n$. Имеем:

$$G_S(2) \oplus G_S(4m + 1) = a(2) \oplus a(4m + 1) = 0 \oplus 0 = 0,$$
$$G_S(4) \oplus G_S(4m - 1) = a(4) \oplus a(4m - 1) = 1 \oplus 1 = 0.$$

По условию (3) либо $(4m + 1) - 2 = 4m - 1$, либо $(4m - 1) - 4 = 4m - 5$ не принадлежит множеству S. Поэтому в любом случае $0 \in A_n$.

4) $n = 4m+4$. В этом случае $a(n) = 3$ и надо проверить, что $0, 1, 2 \in A_n$. Соотношение $0, 1 \in A_n$, проверяется аналогично случаю 2), а соотношение $2 \in A_n$ – аналогично случаю 3) с использованием условия (4).

Таким образом, соотношение (11) проверено во всех случаях, поэтому $G_S(n) = a(n)$. Теорема доказана.

Теперь перейдем непосредственно к задаче. Пусть $f(n)$ – искомое количество хороших множеств, элементы которых не превосходят n. Введем также дополнительные функции $f_r(n)$. Это количество хороших множеств, элементы которых не превосходят n и дают остаток r при делении на 4. Тогда:

$$f(n) = f_0(n) \cdot f_1(n) \cdot f_2(n) \cdot f_3(n). \tag{12}$$

Действительно, из теоремы 14 следует, что поведение элементов множества, дающих фиксированный остаток при делении на 4, никак не зависит от элементов, дающих другие остатки.

Найдем отдельно значение $f_r(n)$ для каждого $r \in \{0, 1, 2, 3\}$.

При $r = 1$ единственное ограничение на S это $1 \in S$. Поэтому $f_1(n) = 2^{[(n-1)/4]_+}$, где $x_+ = \max\{0, x\}$.

При $r = 2$ единственное ограничение на S это $2, 6 \notin S$. Поэтому $f_2(n) = 2^{[(n-6)/4]_+}$.

При $r = 0$ ограничения на S сложнее, так как кроме свойств $0 \in S$, $4, 8 \notin S$ появляется еще свойство (4). Достаточно найти выражение для $g(n) = f_0(4n)$. Ясно, что $g(0) = g(1) = g(2) = 1$, а из свойства (4) можно легко получить рекуррентную формулу $g(n+1) = g(n)+g(n-1)$ при $n \geq 2$. Отсюда следует, что $g(n) = F_n$ при $n \geq 1$, где F_n – $n-$тый член последовательности Фибоначчи. Поэтому $f_0(n) = F_{[n/4]}$ при $n \geq 4$ и $f_0(n) = 1$ при $0 \leq n < 4$.

Наконец, при $r = 3$ должны выполняться условия $3 \notin S$ и (3). Откуда аналогично случаю $r = 0$ получаем, что $f_3(n) = F_{[(n+5)/4]}$.

Значения $f_1(n), f_2(n)$ могут быть вычислены с помощью бинарного алгоритма возведения в степень числа 2 по модулю за $O(\log n)$, а $f_0(n), f_3(n)$ – с помощью бинарного алгоритма возведения в степень соответствующей матрицы 2×2 по модулю также за $O(\log n)$. Оба алгоритма подробно рассмотрены в лекции. Таким образом, получаем решение задачи за $O(\log n)$.

Задача K. Две улитки

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Из верхнего левого угла прямоугольного поля, состоящего из M строк и N столбцов, улитка обошла по спирали по часовой стрелке. При этом все клетки она пронумеровала числами 1, 2, 3,... последовательно в порядке обхода.

Теперь на этом поле появляется другая улитка, которой необходимо попасть из клетки (i_1, j_1) в клетку (i_2, j_2). Каждую секунду она может перемещаться в соседнюю по горизонтали или вертикали клетку, но лишь при условии, что номер этой клетки отличается от номера предыдущей не более, чем на k.

Ваша задача – узнать сколько времени потребуется второй улитке, чтобы добраться до нужной клетки.

Формат входного файла

Во входном файле записаны целые числа M, N, k, i_1, j_1, i_2, j_2 ($1 \leq M, N \leq 1000$, $1 \leq k \leq 20000$, $1 \leq i_1, i_2 \leq M$, $1 \leq j_1, j_2 \leq N$).

Формат выходного файла

В выходной файл выведите одно целое число – минимальное время, за которое вторая улитка сможет добраться до нужной клетки.

Примеры

k.in	k.out
3 5 1 1 2 3 4	6
5 5 7 3 1 3 3	4

Разбор задачи K. Две улитки

См. разбор задачи A.

Задача L. Треугольная комната

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Во многих книгах по занимательной математике приводится такая задача. *Расставить по периметру треугольной комнаты 3 стула так, чтобы у каждой стены стояло по 2. Ее решение — поставить по стулу в каждый из углов комнаты.*

Рассмотрим более общую задачу. Пусть комната представляет собой треугольник ABC. Дано общее количество стульев n, количество стульев n_{AB}, которое должно стоять у стены AB, количество стульев n_{BC}, которое должно стоять у стены BC, количество стульев n_{AC}, которое должно стоять у стены AC. Необходимо найти общее количество расстановок стульев, удовлетворяющих условию. Стулья можно ставить только в углы комнаты и вдоль стен, в центр комнаты стулья ставить нельзя. В любой из углов можно поставить произвольное количество стульев.

Формат входного файла

Входной файл содержит целые числа n, n_{AB}, n_{BC}, n_{AC} ($0 \le n, n_{AB}, n_{BC}, n_{AC} \le 1000$).

Формат выходного файла

В первой строке выходного файла выведите количество различных вариантов расстановки стульев. В случае, когда есть хотя бы один вариант, выведите во второй строке 6 целых неотрицательных чисел: k_A, k_{AB}, k_B, k_{BC}, k_C, k_{AC} – соответственно количество стульев, которые необходимо поставить в угол A, вдоль стены AB, в угол B, вдоль стены BC, в угол C и вдоль стены AC.

Примеры

l.in	l.out
3 2 2 2	1 1 0 1 0 1 0
3 3 2 2	0

Разбор задачи L. Треугольная комната

См. разбор задачи B.

Задача M. Получить одинаковые

Имя входного файла:	m.in
Имя выходного файла:	m.out
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

На столе лежат карточки, на каждой из которых написано некоторое число. Кроме того, имеется довольно большая стопка чистых карточек. За один ход разрешается убрать со стола две карточки с одинаковыми числами, затем взять из стопки две чистые, написать на каждой из них по одному числу и положить на стол.

Требуется узнать как сделать так, чтобы на столе были карточки, на которых написаны одинаковые числа.

Формат входного файла

В первой строке задается количество лежащих на карточек N ($1 \leq N \leq 30000$), а во второй – N чисел, написанных на них. Все числа натуральные и не превышают 10^9.

Формат выходного файла

В первой строке выведите количество операций L, необходимых для того, чтобы получить на столе все одинаковые карточки. Значение L необязательно должно быть минимально возможным, но не должно превосходить N. В последующих L строках выведите по четыре числа, определяющие, что нужно делать в соответствующий ход: первое и второе – числа, записанные на убираемых карточках, третье и четвертое – числа, которые нужно написать на новых карточках. Если невозможно сделать так, чтобы на столе оказались все одинаковые карточки, выведите одно число -1.

Примеры

m.in	m.out
3 1 2 2	1 2 2 1 1
6 3 3 3 4 4 4	2 3 3 4 3 3 3 4 4

Разбор задачи M. Получить одинаковые

См. разбор задачи D.

Задача N. Прыжки на полосе

Имя входного файла:	n.in
Имя выходного файла:	n.out
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Горизонтальная полоса $1 \times N$ разбита на клетки размера 1×1. В клетке с номером s стоит шашка. В первый ход она должна переместиться на одну клетку (влево или вправо). Второй ход должен быть сделан уже на 2 клетки и т.д. Каждый следующий ход выполняется с шагом на единицу больше, чем предыдущий. Так продолжается до тех пор, пока у шашки не останется ходов, не выводящих ее за пределы полосы. Определите какое максимальное и минимальное число ходов может сделать шашка.

Формат входного файла

В первой строке входного файла задано количество тестов T ($1 \le T \le 20$). В каждой из последующих T строк заданы по два целых числа N и s ($1 \le N \le 1000$, $1 \le s \le N$), определяющих данные для соответствующего теста.

Формат выходного файла

Выходной файл должен состоять из T строк, в каждой из которых содержится по два целых числа – максимальное и минимальное количество ходов шашки для соответствующего теста.

Пример

n.in	n.out
2 3 2 6 2	2 2 5 3

Разбор задачи N. Прыжки на полосе

См. разбор задачи E.

Задача O. Произведение взаимно простых – 2

Имя входного файла:	o.in
Имя выходного файла:	o.out
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Вам дано натуральное число m. Требуется найти произведение всех натуральных чисел не превосходящих m и взаимно простых с m, и выдать его остаток при делении на m.

Формат входного файла

В единственной строке входного файла задано натуральное число $m \leq 10^{12}$.

Формат выходного файла

В единственную строку выходного файла выведите ответ на задачу.

Примеры

o.in	o.out
1	0
2	1
3	2
4	3
5	4
6	5
7	6

Разбор задачи O. Произведение взаимно простых – 2

См. разбор задачи F.

Задача P. Неделимость биномиальных коэффициентов

Имя входного файла:	`p.in`
Имя выходного файла:	`p.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Обозначим $C_n^i = \dfrac{n!}{i!(n-i)!}$, где $0 \le i \le n$ и n, i – целые числа. Вам даны натуральное число n и простое число p. Требуется найти количество чисел $i \in \{0, 1, \ldots, n\}$, для которых C_n^i не делится на p.

Формат входного файла

В единственной строке входного файла заданы натуральное число $n \le 10^{18}$ и простое число $p < 10^{18}$.

Формат выходного файла

В единственную строку выходного файла выведите ответ на задачу.

Примеры

p.in	p.out
4 2	2
8 3	9
4 5	5

Разбор задачи P. Неделимость биноминальных коэффициентов

См. разбор задачи G.

Задача Q. Иррациональные попарные расстояния

Имя входного файла:	`q.in`
Имя выходного файла:	`q.out`
Ограничение по времени:	0.5 с
Ограничение по памяти:	256 Мб

Для данного натурального N требуется построить на плоскости множество из N точек, удовлетворяющее следующим условиям:

- координаты точек являются целыми числами;
- расстояние между любыми двумя точками множества является иррациональным числом;
- никакие три точки множества не лежат на одной прямой.

Формат входного файла

В единственной строке входного файла задано натуральное число $N \leq 1000$ – количество точек в множестве.

Формат выходного файла

В выходной файл выведите координаты точек построенного множества (каждую точку в отдельной строке, координаты через пробел). Координаты точек не должны превышать по модулю 1000000. Гарантируется, что такое множество существует. Если таких множеств несколько, можно выдать любое.

Примеры

q.in	q.out
1	1000000 -1000000
2	0 0 1 1
3	0 0 1 1 -1 2

Разбор задачи Q. Иррациональные попарные расстояния

Рассмотрим точки $\left(i, \frac{i(i+1)}{2}\right)$, где $0 \leq i < n$. Докажем, что это множество точек подходит. Во-первых, ясно, что их координаты целые. Далее,

так как $n \le 1000$, то их координаты неотрицательные и меньше 500000, то есть удовлетворяют условию задачи. Так как эти точки лежат на параболе $f(x) = \dfrac{x^2 + x}{2}$, график которой строго выпуклый вверх при $x \ge 0$, то никакие три из них не лежат на одной прямой. Наконец, расстояние между $i-$той и $j-$той точками равно:

$$d_{i,j} = \sqrt{(i-j)^2 + \left(\frac{i^2+i}{2} - \frac{j^2+j}{2}\right)^2} = \frac{|i-j|}{2}\sqrt{4 + (i+j+1)^2}.$$

При $i \ne j$ выполнено неравенство $i + j + 1 \ge 2$, поэтому число $4 + (i + j + 1)^2$ не может быть квадратом целого числа, а значит, $d_{i,j}$ является иррациональным числом.

Задача R. Минимаксное паросочетание

Имя входного файла:	`r.in`
Имя выходного файла:	`r.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Задан полный двудольный граф, каждая доля которого состоит из N вершин. Каждой из вершин приписан вес – целое число. Вес ребра определяется как произведение весов вершин, которые оно соединяет.

Как известно, паросочетанием в графе называется множество ребер, попарно не имеющих общих вершин. Паросочетание называют совершенным, если оно покрывает все вершины графа, то есть каждая вершина графа инцидентна какому-либо ребру паросочетания.

Ваша задача – найти совершенное минимаксное паросочетание, то есть такое, что максимальный из весов ребер паросочетания будет минимально возможным.

Формат входного файла

В первой строке входного файла задано целое число N ($1 \le N \le 10^5$). Во второй строке – N целых чисел, не превосходящих по абсолютной величине 10^9. i-ое число в строке обозначает вес i-ой вершины первой доли графа. В третьей строке аналогичным образом представлены веса вершин второй доли. Вершины каждой доли имеют номера от 1 до N.

Формат выходного файла

В первой строке выходного файла выведите одно целое число – вес совершенного минимаксного паросочетания, т.е. максимальный из весов

его ребер. А во второй строке описание этого паросочетания – N целых чисел, i-ое из которых обозначает номер вершины во второй доле, с которой соединена ребром паросочетания i-ая вершина первой доли.

Пример

r.in	r.out
3 1 2 3 9 10 11	27 2 3 1

Разбор задачи R. Минимаксное паросочетание

За весьма громоздкой формулировкой скрывается довольно простая задача: задано два массива $A[1..N]$ и $B[1..N]$. Необходимо найти такую перестановку P чисел от 1 до N, что величина $\max\limits_{i=\overline{1,N}} A[i]\cdot B[P_i]$ была минимально возможной.

Предположим сначала, что все элементы в массивах – положительны. Тогда необходимо поставить в соответствие меньшему значению A большее значение B. Если это не так, и существует пара таких значений i и j, что $A[i] < A[j]$ и $A[P_i] < A[P_j]$, то максимальное из произведений в этой паре $(A[j]\cdot A[P_j])$ будет больше любого из произведений $A[j]\cdot A[P_i]$ и $A[i]\cdot A[P_j]$. Поэтому обменяв значения P_i и P_j в перестановке мы точно не ухудшим паросочетание.

Если все элементы отрицательны, то можно изменить знак всех элементов на противоположный и свести задачу к уже решенной. Предположим теперь, что количество отрицательных элементов в A равно p, а в B – q. Если $p + q < N$, то мы сможем p наибольших положительных элементов из B сочетать с отрицательными значениями A (причем произвольным образом), q наибольших положительных элементов A – с отрицательными элементами B. Для оставшихся $N - p - q$ положительных элементов A и B сочетаем согласно указанному выше правилу. Аналогично рассматривается случай $p + q > N$ – здесь придется сочетать малые по модулю отрицательные числа между собой.

Наконец остается самый опасный случай – $p + q = N$. Здесь снова p положительных элементов из B сочетаются с отрицательными элементами A, а q положительных элементов A – с отрицательными элементами B. В этом случае все произведения (а значит и наибольшее из них) будут отрицательными. Если хотя бы один элемент сочетался бы с элементом своего знака, то результат их умножения будет положительным, и значит такое сочетание заведомо не будет оптимальным. Однако в этом случае

мы уже не можем произвольно сочетать элементы каждой группы – нам необходимо обеспечить минимальность максимального произведения, или максимальность минимального из абсолютных значений произведения. Для этого меньшему по модулю значению A должно соответствовать большее по модулю значение B.

Если некоторые элементы равны нулю, то их можно отнести либо к положительным, либо к отрицательным. Они корректно укладываются в рассмотренную нами схему.

Таким образом, необходимо отсортировать массивы A и B вместе с номерами в противоположных порядках. В случае, если $p+q=N$, нужно перевернуть отрицательные фрагменты в каждом массиве в обратном порядке. Таким образом, мы получим соответствие, гарантирующее минимальность максимального произведения. Сложность алгоритма – $O(N \log N)$.

Задача S. Количество квадратичных вычетов

Имя входного файла:	`s.in`
Имя выходного файла:	`s.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Пусть m некоторое натуральное число. Число $a \in \{0, 1, \ldots, m-1\}$ называется квадратичным вычетом по модулю m, если существует такое целое число x, что x^2-a делится на m. Вам дано m и требуется найти количество квадратичных вычетов по модулю m.

Формат входного файла

В первой строке входного файла задано натуральное число $T \leq 100$, количество натуральных чисел m в файле. В последующих T строках заданы сами эти числа. Гарантируется, что каждое из них не превосходит 10^{12}.

Формат выходного файла

Для каждого натурального числа m из входного файла выведите в отдельной строке количество квадратичных вычетов по модулю m.

Примеры

s.in	s.out
5 1 2 3 4 12	1 2 2 2 4

Разбор задачи S. Количество квадратичных вычетов

Пусть $f(m)$ – искомое количество квадратичных вычетов по модулю m. Разложим m на простые множители: $m = p_1^{a_1} \cdot \ldots \cdot p_s^{a_s}$. Тогда из китайской теоремы об остатках следует, что:

$$f(m) = f(p_1^{a_1}) \cdot \ldots \cdot f(p_s^{a_s}). \tag{13}$$

Найдем $f(p^n)$ для простого p и натурального n. Пусть a – ненулевой квадратичный вычет по модулю p^n и $a = p^k b$, где $k \geq 0$ и b не кратно p. Так как $0 < a < p^n$, то $k < n$. Если k нечетно, то из сравнения $x^2 \equiv a \pmod{p^n}$ следует, что x^2 делится на p^k. Откуда x делится на $p^{(k+1)/2}$, а значит x^2 делится на p^{k+1}. Поэтому так как $k < n$, то и a делится на p^{k+1}. Откуда b делится на p. Противоречие. Значит k четно. Пусть $k = 2j, j \geq 0$. Тогда как и ранее заключаем, что x делится на p^j, то есть $x = p^j y$. И тогда сравнение $x^2 \equiv a \pmod{m}$ равносильно сравнению $y^2 \equiv b \pmod{p^{n-2j}}$.

Из этого рассуждения следует, что:

$$f(p^n) = 1 + \sum_{j=0}^{\left[\frac{n-1}{2}\right]} g(p^{n-2j}), \tag{14}$$

где $g(m)$ – количество квадратичных вычетов по модулю m, взаимно простых с m, а первая единица отвечает за нулевой квадратичный вычет.

Найдем теперь $g(p^n)$.

Пусть вначале $p = 2$. Тогда ясно, что $g(2) = g(4) = 1$, а при $n \geq 3$ по теореме из лекции нечетное число a является квадратичным вычетом по модулю 2^n тогда и только тогда, когда $a \equiv 1 \pmod{8}$. Поэтому $g(2^n) = 2^{n-3}$, при $n \geq 3$. Объединяя ответы при разных n в один, получим, что:

$$g(2^n) = 2^{\max\{0, n-3\}}, \ n \geq 1. \tag{15}$$

Если p – нечетное простое число, то по теореме из лекции число a, не кратное p, будет квадратичным вычетом по модулю p^n тогда и только

тогда, когда остаток a при делении на p является квадратичным вычетом по модулю p. Так как по модулю p ровно $(p-1)/2$ ненулевых квадратичных вычетов (см. лекцию), то:

$$g(p^n) = \frac{p-1}{2} p^{n-1},\ n \geq 1. \tag{16}$$

Теперь используя формулы (13), (14), (15) и (16), мы можем легко найти $f(m)$ по разложению m на простые множители.

Так как во входном файле может быть порядка 100 чисел m, то стандартный алгоритм разложения на простые множители за $O(\sqrt{m})$, вообще говоря, может не пройти по времени. Поэтому необходимо вначале до считывания входных данных найти с помощью решета Эратосфена все простые числа до $\sqrt{10^{12}} = 10^6$ и сохранить их в массиве. Всего их будет около 80000. А потом считывать m и раскладывать его на множители, деля последовательно на все простые числа до $\sqrt{m}$. В итоге сложность алгоритма будет равна $O(T\pi(\sqrt{m}))$, где $\pi(n)$ – количество простых чисел не больших n. По замечанию выше $T\pi(\sqrt{m}) < 100 \cdot 80000 = 8 \cdot 10^6$. То есть алгоритм с такой сложностью вполне уложится по времени.

Задача T. Факториал и 4-я степень

Имя входного файла:	`t.in`
Имя выходного файла:	`t.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Для натурального m и простого p обозначим через $\deg_p(m)$ показатель вхождения числа p в каноническое разложение числа m на простые множители. Вам даны натуральное число n и простое число p. Требуется вычислить остаток числа $n!/p^{\deg_p(n!)}$ при делении на p^4. Другими словами $n!$ делят нацело на p, пока это возможно, и полученное число берут по модулю p^4. Число n будет задано в p-ичной записи, то есть $n = d_{L-1}\ p^{L-1} + d_{L-2}\ p^{L-2} + \ldots + d_1 p + d_0$, где $d_{L-1}, d_{L-2}, \ldots, d_1, d_0$ – некоторые целые неотрицательные числа, меньшие p (цифры числа n в p-ичной записи).

Формат входного файла

В первой строке входного файла задано простое число p ($3 < p < 55000$) и натуральное число $L \leq 500\,000$ – длина p-ичной записи числа n. Во второй строке записаны через пробел числа $d_{L-1}, d_{L-2}, \ldots, d_1, d_0$, при этом $d_{L-1} > 0$.

Формат выходного файла

В единственную строку выходного файла выведите ответ на задачу.

Примеры

t.in	t.out
5 5 1 2 1 3 4	607
11 10 2 0 3 6 9 0 6 2 6 10	8891

Разбор задачи T. Факториал и 4-я степень

Пусть $F(n) = n!/p^{\deg_p(n!)} \bmod p^4$ – величина, которую требуется найти. Поделим n на p с остатком: $n = pq + r$, где $0 \le r < p$ и $q \ge 0$. Тогда:

$$n! = \prod_{i=1}^{n} i = \left(\prod_{j=0}^{q-1}\prod_{k=1}^{p-1}(pj+k)\right)\left(\prod_{k=1}^{r}(pq+k)\right)\left(\prod_{j=1}^{q} pj\right) = \left(\prod_{j=0}^{q-1}\prod_{k=1}^{p-1}(pj+k)\right)\left(\prod_{k=1}^{r}(pq+k)\right) q!p^q.$$

Откуда:

$$F(n) \equiv F(q)\left(\prod_{j=0}^{q-1} f(j)\right) f_r(q) \pmod{p^4}, \tag{17}$$

где $f_r(j) = (\prod_{k=1}^{r}(pj+k)) \bmod p^4$, а $f(j) = f_{p-1}(j)$.

Ясно, что:

$$f_r(j) \equiv r! + pj \sum_{1\le x\le r} \frac{r!}{x} + (pj)^2 \sum_{1\le x<y\le r} \frac{r!}{x\cdot y} + (pj)^3 \sum_{1\le x<y<z\le r} \frac{r!}{x\cdot y\cdot z} \pmod{p^4}$$

или:

$$f_r(j) = a_0(r)\left(1 + pj\cdot a_1(r) + (pj)^2\cdot a_2(r) + (pj)^3\cdot a_3(r)\right) \bmod p^4, \tag{18}$$

где:

$$a_0(r) = r! \bmod p^4,$$
$$a_1(r) = \left(\sum_{1 \le x \le r} \mathrm{inv}_{p^4}(x) \right) \bmod p^4,$$
$$a_2(r) = \left(\sum_{1 \le x < y \le r} \mathrm{inv}_{p^4}(x)\mathrm{inv}_{p^4}(y) \right) \bmod p^4,$$
$$a_3(r) = \left(\sum_{1 \le x < y < z \le r} \mathrm{inv}_{p^4}(x)\mathrm{inv}_{p^4}(y)\mathrm{inv}_{p^4}(z) \right) \bmod p^4,$$

а $\mathrm{inv}_m(x)$ – обратный остаток к x по модулю m (см. лекцию).

Из определения функций $a_0(r)$, $a_1(r)$, $a_2(r)$, $a_3(r)$ легко получить эффективные формулы для их вычисления:

$$a_0(0) = 1, a_1(0) = a_2(0) = a_3(0) = 0, \tag{19}$$

а при $0 < r < p$:

$$a_0(r) = r \cdot a_0(r-1) \bmod p^4, \tag{20}$$
$$a_1(r) = (a_1(r-1) + \mathrm{inv}_{p^4}(r)) \bmod p^4, \tag{21}$$
$$a_2(r) = (a_2(r-1) + \mathrm{inv}_{p^4}(r)a_1(r-1)) \bmod p^4, \tag{22}$$
$$a_3(r) = (a_3(r-1) + \mathrm{inv}_{p^4}(r)a_2(r-1)) \bmod p^4. \tag{23}$$

Число $\mathrm{inv}_{p^4}(r)$ может быть найдено либо с помощью расширенного алгоритма Евклида, либо с помощью теоремы Эйлера и бинарного алгоритма возведения в степень. Оба способа подробно описаны в лекции и имеют сложность $O(\log p^4) = O(\log p)$.

Обозначим $A_0 = a_0(p-1)$, $A_1 = A_0 \cdot a_1(p-1) \bmod p^4$, $A_2 = A_0 \cdot a_2(p-1) \bmod p^4$, $A_3 = A_0 \cdot a_3(p-1) \bmod p^4$. Тогда $f(j) = \left(A_0 + pjA_1 + (pj)^2A_2 + (pj)^3A_3\right) \bmod p^4$.

Следующая лемма является ключевой для быстрого вычисления произведения $\left(\prod_{j=0}^{q-1} f(j) \right) \bmod p^4$.

Лемма. 7.

В предыдущих обозначениях выполнены следующие соотношения:

1) A_1 *делится на* p^2.

2) A_2 *делится на* p.

3) A_3 *делится на* p.

Доказательство. Рассмотрим многочлен $P(x) = (x+1)(x+2)\dots(x+p-1) - (x^{p-1}-1) = (p-1)! + 1 + \sum_{i=1}^{p-2} B_i x^i$. Он был введен при доказательстве теоремы Вильсона в лекции, и там было доказано, что все его коэффициенты кратны p. Значит, числа B_1, B_2 и B_3 кратны p. С другой стороны, ясно, что, так как $p > 3$, то $B_j \equiv A_j \pmod p$ при $j = 1, 2, 3$. Отсюда следуют пункты **(2)** и **(3)** леммы. Докажем пункт **(1)**. Имеем:

$$A_1 \equiv \sum_{x=1}^{p-1} \frac{(p-1)!}{x} \equiv \sum_{x=1}^{\frac{p-1}{2}} \frac{p!}{x(p-x)} \equiv -p! \sum_{x=1}^{\frac{p-1}{2}} (\mathrm{inv}_p(x))^2 \equiv -\frac{p!}{2} \sum_{x=1}^{p-1} (\mathrm{inv}_p(x))^2 \equiv$$

$$\equiv -\frac{p!}{2} \sum_{x=1}^{p-1} x^2 \equiv -\frac{p!}{2} \cdot \frac{p(p-1)(2p-1)}{6} \equiv 0 \pmod{p^2}$$

так как $p > 3$. □

Из этой леммы следует, что коэффициенты pA_1 и p^2A_2 при j и j^2 в выражении для $f(j)$ делятся на p^3, а последнее слагаемое $(pj)^3 A_3$ вообще делится на p^4, поэтому его можно опустить при вычислении. То есть:

$$f(j) = \left(A_0 + pjA_1 + (pj)^2 A_2\right) \pmod{p^4}. \tag{24}$$

Таким образом, $f(j)$ зависит лишь от остатка числа j при делении на p. Поделим q на p с остатком: $q = pq_1 + r_1$, где $0 \le r_1 < p$ и $q_1 \ge 0$. Учитывая это, имеем:

$$\prod_{j=0}^{q-1} f(j) \equiv \left(\prod_{j_1=0}^{q_1-1} \prod_{k=0}^{p-1} f(pj_1 + k)\right) \left(\prod_{k=0}^{r_1-1} f(pq_1 + k)\right) \equiv$$

$$\equiv \left(\prod_{k=0}^{p-1} f(k)\right)^{q_1} \left(\prod_{k=0}^{r_1-1} f(k)\right) \equiv g(p-1)^{q_1} g(r_1 - 1) \pmod{p^4}$$

или:

$$\prod_{j=0}^{q-1} f(j) \equiv g(p-1)^{q_1} g(r_1 - 1) \pmod{p^4}, \tag{25}$$

где:

$$g(j) = \left(\prod_{k=0}^{j} f(k)\right) \bmod p^4. \tag{26}$$

Обозначим $G = g(p-1)$. Объединяя формулы (17), (25) получим:

$$F(n) \equiv F(q)G^{q_1}g(r_1-1)f_r(q) \pmod{p^4}. \tag{27}$$

По условию $n = \sum\limits_{i=0}^{L-1} d_i p^i$. Обозначим $n_k = \sum\limits_{i=k}^{L-1} d_i p^{i-k}$. Кроме того, будем считать, что $d_i = n_i = 0$ при всех $i \geq L$. Ясно, что $n = n_0$, $q = n_1$, $r = d_0$, $q_1 = n_2$ и $r_1 = d_1$. Поэтому формула (27) примет вид:

$$F(n_0) \equiv F(n_1)G^{n_2}g(d_1-1)f_{d_0}(n_1) \pmod{p}^4.$$

Применяя все те же рассуждения к n_k вместо $n = n_0$ получим, что:

$$F(n_k) \equiv F(n_{k+1})G^{n_{k+2}}g(d_{k+1}-1)f_{d_k}(n_{k+1}) \pmod{p^4}. \tag{28}$$

Итерируя формулу (28) получим, что:

$$F(n) \equiv G^{\sum\limits_{k=0}^{L-1} n_{k+2}} \left(\prod_{k=0}^{L-1} g(d_{k+1}-1)f_{d_k}(n_{k+1}) \right) \pmod{p^4}. \tag{29}$$

Теперь опишем более подробно алгоритм вычисления $F(n)$ по формуле (29).

Сначала необходимо посчитать значения функций $a_0(r)$, $a_1(r)$, $a_2(r)$, $a_3(r)$ при $0 \leq r < p$ по формулам (19), (20), (21), (22), (23). Затем вычислить числа A_0, A_1, A_2, чтобы потом посчитать по ним значения функций $f(j)$ и $g(j)$ при $0 \leq j < p$ по формулам (24), (26). Весь этот препроцессинг происходит за $O(p \log p)$ операций ($\log p$ возникает из-за нахождения $\mathrm{inv}_{p^4}(x)$).

Теперь произведение $\left(\prod\limits_{k=0}^{L-1} g(d_{k+1}-1)f_{d_k}(n_{k+1}) \right) \bmod p^4$ может быть вычислено за $O(L)$, так как значения $g(j)$ сохранены в массиве для всех $j \in \{0, 1, \ldots, p-1\}$, и, значит, вычисление $g(d_{k+1}-1)$ происходит за $O(1)$, а значение $f_{d_k}(n_{k+1}) = f_{d_k}(d_{k+1} + d_{k+2}p + d_{k+3}p^2)$ может быть вычислено за $O(1)$ по формуле (18) по известным величинам $a_0(r)$, $a_1(r)$, $a_2(r)$, $a_3(r)$.

Чтобы вычислить $G^{\sum\limits_{k=0}^{L-1} n_{k+2}} \bmod p^4$, нам понадобится следующая лемма.

Лемма. 8. *В предыдущих обозначениях выполнено соотношение* $G \equiv A_0^p \pmod{p^4}$.

Доказательство. По лемме 7 имеем

$$G \equiv \prod_{j=0}^{p-1} f(j) \equiv \prod_{j=0}^{p-1} (A_0 + (pA_1)j + (p^2A_2)j^2) \equiv A_0^p + p^3A_0^{p-1}\left(\frac{A_1}{p^2}\sum_{j=0}^{p-1} j + \frac{A_2}{p}\sum_{j=0}^{p-1} j^2 \right) \equiv$$

$$\equiv A_0^p + p^3A_0^{p-1}\left(\frac{A_1}{p^2} \cdot \frac{p(p-1)}{2} + \frac{A_2}{p} \cdot \frac{p(p-1)(2p-1)}{6} \right) \equiv A_0^p \pmod{p^4}$$

□

так как $p > 3$.

По лемме 8 и теореме Эйлера (см. лекцию) имеем для любого $s \geq 0$

$$G^s \equiv A_0^{p\,s} \equiv A_0^{(ps) \bmod p^3(p-1)} \equiv A_0^{p\cdot(s \bmod p^2(p-1))} \equiv G^{s \bmod p^2(p-1)} \pmod{p^4}. \qquad (30)$$

Замечание. 1. *Непосредственно по теореме Эйлера можно заменить* s *на* $s \bmod p^3(p-1)$*, что тоже приемлемо, но лемма 8 позволяет заменить* s *на более меньшее число.*

Таким образом, нужно вычислить $s = \left(\sum_{k=0}^{L-1} n_{k+2}\right) \bmod p^2(p-1)$, а потом с помощью бинарного алгоритма возведения в степень за $O(\log s) = O(\log p)$ вычислить $G^s \bmod p^4$. Так как $n_k = pn_{k+1} + d_k$, то мы легко можем найти s за $O(L)$.

Все вычисления можно проводить в беззнаковом 64-битном целочисленном типе, поскольку ограничение на p таково, что $p^4 < 2^{63}$. Чтобы не было переполнения при вычислении произведения двух чисел по модулю p^4 надо работать с ними в p^2-ичной системе счисления. То есть, если нам надо вычислить $(a \cdot b) \bmod p^4$, то представим a и b в виде $a = a_0 + a_1 \cdot p^2$ и $b = b_0 + b_1 \cdot p^2$, где $0 \leq a_0, a_1, b_0, b_1 < p^2$. Тогда:

$$(a \cdot b) \bmod p^4 = (a_0 \cdot b_0 + ((a_0 \cdot b_1 + a_1 \cdot b_0) \bmod p^2) \cdot p^2) \bmod p^4.$$

Здесь уже все промежуточные вычисления не дают переполнения типа.

В итоге, получаем алгоритм вычисления $F(n)$ со сложностью $O(p \log p + L)$ и памятью $O(p + L)$.

День второй (20.02.2010г). Контест Виталия Гольдштейна

Об авторе...

Гольдштейн Виталий Борисович, ассистент и аспирант Московского Физико-Технического Института. Член научного комитета всероссийской олимпиады школьников по информатике и сборов по подготовке к международной олимпиаде школьников. Стажер компании Google (зима-весна 2008). Завуч Летней Компьютерной Школы (Август, 2009) и Летней Компьютерной Школы (Зима, 2009-2010). Разработчик компании Яндекс.

Основные достижения:

- чемпион России на олимпиаде по информатике среди школьников 2004;
- золотая медаль на международной олимпиаде школьников по информатике 2004 (IOI Athens 2004);
- 17-е место чемпионата мира по программированию среди студентов (ACM ICPC Shanghai 2005);
- серебряная медаль чемпионата мира по программированию среди студентов (ACM ICPC Tokyo 2007);
- участник финалов TopCoder Open, TopCoder Collegiate Open, Global Google Code Jam, Google Code Jam Europe;
- тренер призеров (4-е место, золотая медаль) чемпионата мира по спортивному программированию ACM-ICPC World Finals 2009, Швеция, Стокгольм.

Теоретический материал. Структуры данных

Дерево поиска

Дерево поиска — двоичное дерево, в каждой вершине которого хранится ключ. Для каждой вершины дерева с ключом x (корня поддерева) выполнено, что ключи потомков левого поддерева не превосходят x, а ключи правого поддерева не меньше x.

Высота дерева — длина самого длинного пути от корня до некоторого листа. Аналогично можно говорить о высоте поддерева.

Сбалансированное дерево

Дерево называется сбалансированным по количеству вершин, если для каждой вершины выполнено, что количество вершин в левом поддереве отличается от количества вершин в правом поддереве не более, чем на единицу.

Дерево называется сбалансированным по высоте, если для каждой вершины выполнено, что высота в левом поддереве отличается от высоты в правом поддереве не более, чем на единицу.

Высота сбалансированных деревьев (по-любому из двух определений) равна $O(\log n)$, где n — количество элементов в дереве.

Поиск элемента

Для поиска элемента x в двоичном дереве необходимо сравнить его со значением в корне. Если x меньше значения в корне, то элемент нужно искать в левом поддереве, если x больше значения в корне, то нужно искать в правом поддереве. Если наблюдается равенство, то элемент найден. На каждом шаге алгоритма, мы опускаемся на шаг вниз, а следовательно высота поддерева уменьшается как минимум на единицу. Следовательно, время работы поиска есть $O(h)$.

Куча

Куча — двоичное дерево, в каждой вершине которого хранится ключ. Для каждой вершины дерева с ключом x (корня поддерева) выполнено, что у всех непосредственных детей ключ не больше x.

Операция подъема

При увеличении ключа в некотором элементе кучи, основное свойство кучи может быть нарушено. А именно ключ потомка может стать меньше, чем у измененной вершины. Чтобы исправить это используется операция подъема. Подъем происходит следующим образом: пока ключ измененного элемента меньше, меняем его местами с его потомком. Операция подъема работает за высоту кучи.

Операция спуска

При уменьшении ключа в некотором элементе кучи, основное свойство кучи может быть нарушено. А именно ключ одного (или обоих) из де-

тей может стать больше, чем у измененной вершины. Чтобы исправить это используется операция спуска. Спуск происходит следующим образом: пока ключ измененного элемента больше одного из своих детей, меняем его местами с наибольшим из детей. Если поменять не с максимальным ребенком, то свойство кучи может остаться нарушенным.

Операция удаления

Для удаления элемента, поменяем его с произвольным листом кучи (возможно с ним самим), а сам лист удалим. После этого нужно восстановить, возможно, испорченное свойство кучи. Ошибка может возникнуть только в элементе, ключ которого изменился. При этом ключ мог как увеличится, так и уменьшится. Поэтому, следует применить и подъем, и спуск. Одна из этих операций заведомо остановится на первом же шаге.

Чтобы сохранить сбалансированность, имеет смысл отрезать от наиболее глубокого листа.

Операция добавление

Добавим вершину к произвольному листу, после чего проведем операцию подъема. Чтобы сохранить сбалансированность имеет смысл привешивать к наименее глубокой вершине.

Декартово дерево

Декартово дерево — двоичное дерево, в каждой вершине которого хранится два ключа x и y, для которого выполнены два условия:

- если рассматривать только ключи x, то декартово дерево представляет собой дерево поиска;
- если рассматривать только ключи y, то декартово дерево представляет собой кучу.

Дерево отрезков

Дерево отрезков строится над некоторым массивом и позволяет искать min(max) или сумму, а также совершать некоторую групповую операцию на отрезке. *Дерево отрезков* — двоичное дерево, в листах которого хранятся элементы заданного массива. То есть листья отвечают за операцию на отрезке состоящем из одного элемента. В промежуточных вершинах хранится минимум (максимум или сумма) на объединение отрезков соответствующих ее потомкам. Данное дерево позволяет за время $O(h)$ находить минимум на отрезке.

Построение декартова дерева с использованием дерева отрезков

Пусть нам заданы пары чисел (x_i, y_i), необходимо построить декартово дерево. Какой из элементов будет корнем декартова дерева? Корень декартова дерева имеет максимальный y (по условию кучи). Пусть (x_i, y_i) — пара с максимальным y. Тогда все пары с меньшими x будут в левом поддереве, а остальные в правом поддереве. Каждое из поддеревьев строится

аналогично. Теперь основная задача — находить максимум для ключей y, соответствующих поддереву. Каждое поддерево представляет собой отрезок по ключу x.

Отсортируем все пары чисел по ключу x. Найдем максимальный y_m. Тогда все элементы с индексами $1, \ldots, m-1$ будут в левом поддереве, а $m+1, \ldots, n$ будут в правом поддереве. При дальнейшем построении будет необходимо находить максимальный элемент на некоторых отрезкам. Так, например, на втором шаге необходимо найти максимум на отрезках $[1, m-1]$ и $[m+1, n]$. Если все y различны, то максимальный элемент на каждом шаге будет выбираться однозначно. Поэтому все построение будет выполнено однозначно, а следовательно существует единственное декартово дерево.

Построение декартова дерева за линейное время

Пусть нам задана последовательность пар (x_i, y_i), отсортированных по x. Будем добавлять пары чисел в дерево по очереди. Вновь добавленный элемент каждый раз будет оказываться самым правым элементом в дереве. После добавления каждого элемента будем сохранять ссылку на вновь добавленный элемент. Процесс добавления будет выглядеть следующим образом:

- если y самого правого элемента не меньше, чем у добавляемого, то просто добавим новый элемент справа;
- в противном случае найдем элемент на самом правом пути, который имеет y больше добавляемого. Вместо правого поддерева этого элемента поместим вновь добавленный. А все правое поддерево подвесим слева;
- если же добавляемый элемент имеет максимальный y, тогда все дерево подвешивается к новой вершине слева и новая вершина становится корнем.

Операция merge

Пусть у нас есть два декартова дерева l и r, такие что ключи x дерева l не больше ключей дерева r. Операция merge объединяет два этих дерева. При этом дерево строится из “кусков” деревьев l и r, то есть деревья l и r перестают существовать. Это позволяет проводить объединение за время меньшее, чем линейное.

Объединенное дерево будет состоять из тех же элементов, что и l с r. Попробуем определить корень объединенного дерева. Очевидно, что это элемент с максимальным y. Максимальный y находится в одном из корней двух деревьев.

Пусть максимум находится в левом поддереве (а значит корень l будет корнем объединения), тогда необходимо объединить правого сына l со всем деревом r. Полученное объединение нужно будет прикрепить справа к l.

Если максимум в правом поддереве, то симметрично необходимо объединить все дерево l с левым сыном r.

На каждом шаге сумма высот деревьев уменьшается на единицу, поэтому объединение работает за $O(h_l + h_r)$

Операция split

Операция split — это операция, обратная операции merge. На вход операции подается одно декартово дерево и ключ x, а результатом являются два декартовых дерева l и r, ключи которых не больше и больше x соответственно.

Пусть x_r в корне $\leq x$, тогда все левое поддерево вместе с корнем должно находится в l, а правое поддерево необходимо разделить на l' и r'. l' нужно будет присоединить справа к первоначальному дереву. Тогда первоначальное дерево будет результатом l, а $r = r'$. Обратный случай обрабатывается симметрично. На каждом шаге высота разделяемого дерева уменьшается на единицу, поэтому split работает за $O(h)$.

Стоит заметить, что в операции merge используется только ключ y, а в операции split используется только ключ x.

Добавление

Для добавления пары (x_i, y_i) необходимо разделить дерево по ключу x_i на l и r, после чего создать дерево m из одной вершины. Затем объединение $merge(merge(l, m), r)$ будет результатом добавления.

Все операции проводятся за $O(h)$, а, значит, и все добавление работает за $O(h)$.

Удаление

Для удаления необходимо найти удаляемый элемент, после чего заменить удаленную вершину на объединение двух ее потомков.

Декартово дерево со случайным y

Декартово дерево при некоторых парах (x_i, y_i) может иметь высоту порядка $O(n)$. Мы уже говорили, что существует единственное декартово дерево с различными y. В дальнейшем мы будем использовать декартово дерево со случайным y. Фактически, y будет использоваться только для поддержания структуры дерева. Единственным, имеющим смысл для нас будет ключ x. Можно доказать, что количество декартовых деревьев с высотой не больше $4 \log n$ близко к 100 процентам. Из этого следует, что с огромной вероятностью декартово дерево со случайными ключами y имеет небольшую высоту. Это позволяет нам не задумываться о сбалансированности дерева, а только лишь поддерживать свойства декартова дерева.

Поддержка дополнительных параметров

Для некоторых дополнительных операций требуется хранение дополнительных параметров, таких как количество элементов в поддереве или минимум некоторых дополнительных информационных значений. Для этого необходимо поддерживать инвариант, чтобы все функции, изменяющие дерево, возвращали деревья с корректно подсчитанными значениями. Так, например, в операции объединения достаточно вызвать функцию пересчета дополнительных параметров в конце перед возвратом значения.

Функция пересчета подразумевает, что для потомков детей все параметры подсчитаны верно. Из этих соображений вычисляется значение в корне. Особенно важным для нас будет умение поддерживать количество элементов в поддереве. Это позволит нам искать k-ый по порядку ключ.

Множественные операции

Множественные операции, аналогично деревьям отрезков, реализуются запоминанием и последующим выполнением операций. Если нам необходимо, например, прибавить ко всему дереву некоторое число a, то запишем в корень информацию о том, что нужно прибавить ко всему дереву. При последующих операциях перед обработкой вершины необходимо будет выполнять проталкивание информации. Проталкивание информации подразумевает, что нужно прибавить число к корню и перенести информацию для детей. То есть, для детей мы запишем информацию, которая будет обработана в будущем.

Для того, чтобы применить множественную операцию на отрезке, необходимо вырезать необходимый отрезок операциями split, применить операцию к корню вырезанного отрезка, затем с помощью merge вернуть дерево обратно.

Декартово дерево по неявному ключу

Представление массива в виде дерева

Декартово дерево по неявному ключу — вообще говоря заменяет собой обычный массив, который поддерживает некоторое количество дополнительных операций. Такое дерево не имеет как такового ключа x, он здесь неявный. Обход дерева, фактически, показывает порядок элементов в массиве.

Интерпретация и модернизация операции split

Операция split фактически означает разрез массива на две части. Так, split по неявному ключу x есть разделение на два массива, первый из которых состоит из первых x элементов, а другой – из всех остальных.

Для реализации такой операции необходимо немного модифицировать алгоритм. А именно: если в левом поддереве количество элементов $k_l \geq x$, то необходимо разделять левое поддерево по ключу x, а в противном случае необходимо разделять правое поддерево по ключу $x - k_l - 1$.

Добавление, удаление

Здесь добавление и удаление происходит фактически с массивом.

Добавление — это добавление элемента в середину массива.

Удаление — так же удаление некоторого элемента из середины массива.

Эти операции, аналогично обычному дереву, делаются через модернизированные операции split и merge.

Циклический сдвиг

Циклический сдвиг — есть разрез массива на две части и обмен их местами. То есть, циклический сдвиг есть ни что иное как применение одной операции split и merge.

Динамическое дерево отрезков

Говоря про обычное декартово дерево, мы упоминали как поддерживать минимум в поддереве по некоторой дополнительной информации. Так и в дереве по неявному ключу мы можем поддержать эту дополнительную информацию. Благодаря этому декартово дерево может заменить нам дерево отрезков. Для нахождения минимума на отрезке, достаточно вырезать этот отрезок двумя операциями split. Посмотреть на информацию в корне, после чего двумя операциями merge вернуть дерево в исходное состояние. Аналогично обычному декартову дереву, в дереве по неявному ключу можно поддержать множественные операции на отрезке.

Разворот дерева

Одной из разновидностей множественной операции, которая имеет смысл только в дереве по неявному ключу, является разворот дерева. Если представлять себе дерево, эмулирующее массив, то, фактически, это разворот массива в обратном порядке. Аналогично тому, как мы поступали с прибавлением на отрезке, мы только пометим в корне дерева, что его необходимо развернуть. После этого, перед каждым обращением к вершине дерева будем проверять, не нужно ли перевернуть это дерево. Переворот эквивалентен обмену левого и правого дерева плюс перевороту каждого из деревьев. То есть операция разворот так же может быть отложенной и передаваться потомкам.

Примером операции, которая не может быть отложена, служит операции обмена элементов с четными номерами и элементов с нечетными номерами.

Задачи и разборы

Задача A. Сумма на отрезке

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	3 с
Ограничение по памяти:	6 Мб

Задан набор чисел $a_1 \dots a_n$. Для заданных запросов l и r найдите $s_{l,\,r} = \sum_{j\,=\,l}^{r} a_j$.

Формат входного файла

В первой строке записано количество чисел — n ($1 \le n \le 10^6$). Во второй строке записаны числа a_i ($1 \le a_i \le 1000$), разделенные пробелом. На третьей строке записано число m ($1 \le m \le 10^6$) — количество запросов. Далее на отдельных строках записаны сами запросы l_i и r_i ($1 \le l_i \le r_i \le n$).

Формат выходного файла

Выведите на отдельных строках m чисел s_{l_i,r_i}.

Примеры

a.in	a.out
5 1 2 3 4 5 5 1 5 2 3 3 4 2 5 1 4	15 5 7 14 10

Пояснение

В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи A. Сумма на отрезке

Для решения этой задачи необходимо посчитать массив частичных сумм $s_i = \sum_{j=1}^{i} a_j$ по рекуррентной формуле $s_0 = 0$, $s_i = s_{i-1} + a_i$, для $i > 0$. После этого ответ на запрос вычисляется как $s_{r_i} - s_{l_i-1}$.

Задача B. Частичная сумма матрицы

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	3 с
Ограничение по памяти:	64 Мб

Задана матрица чисел $a_{i,j}$, где $1 \le i \le n, 1 \le j \le m$. Для всех i, j найдите $s_{i,j} = \sum_{k=1,t=1}^{k \le i, t \le j} a_{k,t}$.

Формат входного файла

В первой строке записаны размеры матрицы — n, m ($1 \le n, m \le 1000$). В следующих n строках записано по m чисел $a_{i,j}$ ($1 \le a_{i,j} \le 1000$), разделеннные пробелом.

Формат выходного файла

Выведите n строк по m чисел $s_{i,j}$.

Примеры

b.in	b.out
3 5 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4	1 3 6 10 15 6 12 18 24 30 8 17 24 35 45

Пояснение

В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи B. Частичная сумма матрицы

Для получения s необходимо вычислять ее по возрастанию i и j. Будем считать, что $s_{0,j} = s_{i,0} = 0$, тогда верна рекуррентная формула $s_{i,j} = s_{i-1,j} + s_{i,j-1} - s_{i-1,j-1} + a_{i,j}$. Эта задача является подсказкой к решению следующей задачи.

Задача C. Сумма на матрице

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	4 с
Ограничение по памяти:	64 Мб

Задана матрица чисел $a_{i,j}$, где $1 \le i \le n, 1 \le j \le m$. Для заданных lx, ly, rx, ry найдите $s_{lx,ly,rx,ry} = \sum_{k=lx,t=ly}^{k\le rx,t\le ry} a_{k,t}$.

Формат входного файла

В первой строке записаны размеры матрицы — n, m ($1 \le n, m \le 1000$). В следующих n строках записано по m чисел $a_{i,j}$ ($1 \le a_{i,j} \le 1000$), разделеннные пробелом. В $n+2$ строке записано число q ($1 \le q \le 10^6$) — количество запросов. В следующих q строках описаны запросы lx_i, ly_i, rx_i, ry_i ($1 \le lx_i \le rx_i \le n, 1 \le ly_i \le ry_i \le m$).

Формат выходного файла

Выведите q чисел в отдельных строках — ответы на запросы.

Примеры

c.in	c.out
3 5 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4 5 1 1 3 3 2 2 3 4 2 3 3 5 3 2 3 4 1 4 2 5	24 18 16 9 12

Пояснение

В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи C. Сумма на матрице

Для решения задачи предварительно найдем $s_{i,j} = \sum_{k=1,t=1}^{k \le i, t \le j} a_{k,t}$ как описано в разборе предыдущей задачи. После чего, из аналогичных соображений, применяя формулу включения-исключения, получаем, что $s_{lx,\ ly,\ rx,\ ry} = s_{rx,ry} - s_{lx-1,ry} - s_{rx,\ ly\ -\ 1} + s_{lx\ -\ 1,\ ly\ -\ 1}$.

Задача D. Сумма на параллелепипеде

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	4 с
Ограничение по памяти:	64 Мб

Задана трехмерная таблица чисел $a_{i,j,t}$, где $1 \le i \le n, 1 \le j \le m, 1 \le t \le k$. Для заданных $lx,\ ly,\ lz,\ rx,\ ry,\ rz$ найдите $s_{lx,ly,lz,rx,ry,rz} = \sum_{i=lx,j=ly,t=lz}^{i \le rx, j \le ry, t \le rz} a_{i,j,t}$.

Формат входного файла

В первой строке записаны размеры таблицы — n, m, k ($1 \le n, m, k \le 100$). Далее записано n блоков по m строк, в каждой из которых записано по k чисел $a_{i,j,t}$ ($1 \le a_{i,j,t} \le 1000$). Блоки разделены пустой строкой. В очередной строке записано число q ($1 \le q \le 10^6$) — количество запросов. В следующих q строках описаны запросы lx_i, ly_i, lz_i, rx_i, ry_i, rz_i ($1 \le lx_i \le rx_i \le n, 1 \le ly_i \le ry_i \le m, 1 \le lz_i \le rz_i \le k$).

Формат выходного файла

Выведите q чисел в отдельных строках — ответы на запросы.

Примеры

d.in	d.out
2 3 5 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4 5 1 1 1 1 2 2 1 1 1 2 2 2 1 2 3 2 3 4 1 3 4 2 3 5 1 2 4 2 2 5	12 24 22 18 6

Пояснение

В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи D. Сумма на параллелепипеде

Данная задача является обобщением предыдущей задачи. Для ее решения вычислим сумму на параллелепипеде $s_{i,j,k}$ с левым верхним углом в точке $(1,1,1)$, аналогично задаче "Частичная сумма матрицы". Единственное отличие лишь в размерности. Количество слагаемых будет уже не 4, а 8. Формально можно записать следующим образом:

$$s_{i,j,k} = a_{i,j,k} + \sum_{x,y,z \in \{0,1\}}^{x+y+z \neq 0} (-1)^{x+y+z+1} \cdot s_{i-x,j-y,k-z}$$

.

Тогда вычисление ответа выглядит как:

$$s_{li,lj,lk,ri,rj,rk} = \sum_{x \in \{li-1,ri\}, y \in \{lj-1,rj\}, z \in \{lk-1,rk\}} (-1)^{p(x,y,z)} \cdot s_{x,y,z} \ ,$$

где p — количество среди аргументов, равных меньшему из двух возможных значений (например, $li - 1$).

Задача E. Следующий

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Реализуйте структуру данных, которая поддерживает множество S целых чисел, с которым разрешается производить следующие операции:

- $add(i)$ — добавить в множество S число i (если оно там уже есть, то множество не меняется);
- $next(i)$ — вывести минимальный элемент множества, не меньший i. Если искомый элемент в структуре отсутствует, необходимо вывести -1.

Формат входного файла

Исходное множество S пусто. Первая строка входного файла содержит n — количество операций ($1 \le n \le 300\,000$). Следующие n строк содержат операции. Каждая операция имеет вид либо "+ i", либо "? i". Операция "? i" задает запрос $next(i)$.

Если операция "+ i" идет во входном файле вначале или после другой операции "+", то она задает операцию $add(i)$. Если же она идет после запроса "?", и результат этого запроса был y, то выполняется операция $add((i + y) \bmod 10^9)$.

Во всех запросах и операциях добавления параметры лежат в интервале от 0 до 10^9.

Формат выходного файла

Для каждого запроса выведите одно число — ответ на запрос.

Пример

e.in	e.out
6 + 1 + 3 + 3 ? 2 + 1 ? 4	3 4

Разбор задачи E. Следующий

Данная задача является упражнением на стандартные структуры данных. Не стоит забывать, что вы не первые люди во вселенной, которым потребовались структуры данных. Многие структуры данных уже реализованы и ими необходимо уметь пользоваться. В данном случае в C++ можно воспользоваться классом std::set и методом lower_bound, а в Java интерфейсом SortedSet и методом tailSet. В случае самостоятельной реализации можно использовать любое дерево поиска.

Задача F. K-ый максимум

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также находить k-й максимум.

Формат входного файла

Первая строка входного файла содержит натуральное число n — количество команд ($n \leq 100\,000$). Последующие n строк содержат по одной команде каждая. Команда записывается в виде двух чисел c_i и k_i — тип и аргумент команды соответственно ($|k_i| \leq 10^9$). Поддерживаемые команды:

- $+1$: Добавить элемент с ключом k_i;
- 0: Найти и вывести k_i-й максимум;
- -1: Удалить элемент с ключом k_i.

Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе k_i-го максимума, он существует.

Формат выходного файла

Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число — k_i-й максимум.

Пример

f.in	f.out
8 +1 5 +1 3 +1 7 0 2 -1 3 0 1 1 10 0 3	5 7 5

Разбор задачи F. K-ый максимум

Задача звучит похоже на предыдущую, однако стандартные структуры данных не поддерживают таких операций. Это связано с тем, что это редко возникающая задача на практике. Для решения задачи необходимо воспользоваться любым деревом поиска с поддержкой количества элементов в поддереве. Например, можно использовать разобранное на лекции декартово дерево.

Задача G. В начало строя!

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Капрал Студип любит командовать своим отрядом. Его любимый приказ “в начало строя”. Он выстраивает свой отряд в шеренгу и оглашает последовательность приказов. Каждый приказ имеет вид “Солдаты с l_i по r_i — в начало строя!”.

Пронумеруем солдат в начальном положении с 1 до n, слева направо. Приказ “Солдаты с l_i по r_i — в начало строя!” означает, что солдаты, стоящие с l_i по r_i включительно перемещаются в начало строя, сохраняя относительный порядок.

Например, если в некоторый момент солдаты стоят в порядке $2, 3, 6, 1, 5, 4$, после приказа: “Солдаты с 2 по 4 — в начало строя!” порядок будет $3, 6, 1, 2, 5, 4$.

По данной последовательности приказов найти конечный порядок солдат в строю.

Формат входного файла

В первой строке два целых числа n and m ($2 \leq n \leq 100\,000$, $1 \leq m \leq 100\,000$) — количество солдат и количество приказов. Следующие m строк содержат по два целых числа l_i и r_i ($1 \leq l_i \leq r_i \leq n$).

Формат выходного файла

Выведите n целых чисел — порядок солдат в конечном положении после выполнения всех приказов.

Пример

g.in	g.out
6 3 2 4 3 5 2 2	1 4 5 2 3 6

Разбор задачи G. В начало строя!

В этой задаче необходимо реализовать декартово дерево по неявному ключу. Фактически, здесь необходимо вырезать из массива отрезок и переместить его в начало. Это можно сделать двумя операциями split и двумя операциями merge декартового дерева.

Задача H. Своппер

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Перед возвращением в штаб-квартиру, корпорации Ааазу и Скиву пришлось заполнить на местной таможне декларацию о доходах за время визита. Получилась довольно внушительная последовательность чисел. Обработка этой последовательности заняла весьма долгое время.

— Своппер кривой, — со знанием дела сказал таможенник.

— А что такое своппер? — спросил любопытный Скив.

Ааз объяснил, что своппер — это структура данных, которая умеет делать следующее:

- взять отрезок чётной длины от x до y и поменять местами число x с $x+1$,$x+2$ с $x+3$, и т.д;

- посчитать сумму чисел на произвольном отрезке от a до b.

Учитывая, что обсчёт может затянуться надолго, корпорация "МИФ" попросила Вас решить проблему со своппером и промоделировать ЭТО эффективно.

Формат входного файла

Во входном файле заданы один или несколько тестов. В первой строке каждого теста записаны число N — длина последовательности и число M — число операций ($1 \le N, M \le 100\,000$). Во второй строке теста содержится N целых чисел, не превосходящих 10^6 по модулю — сама последовательность. Далее следуют M строк — запросы в формате 1 x_i y_i — запрос первого типа, и 2 a_i b_i — запрос второго типа. Сумма всех N и M по всему файлу не превосходит $200\,000$. Файл завершается строкой из двух нулей. Гарантируется, что $x_i < y_i$, а $a_i \le b_i$.

Формат выходного файла

Для каждого теста выведите ответы на запросы второго типа, как показано в примере. Разделяйте ответы на тесты пустой строкой.

Пример

h.in	h.out
5 5 1 2 3 4 5 1 2 5 2 2 4 1 1 4 2 1 3 2 4 4 0 0	Swapper 1: 10 9 2

Разбор задачи H. Своппер

Не нужно придумывать, что бы такого хитрого хранить в вершине декартового дерева, чтобы поддержать все операции. Будем хранить массив в виде двух отдельных деревьев: нечетные и четные позиции. Тогда объединение и разделение массивов будет требовать дополнительных проверок. Например, при объединении массива с нечетным количеством элементов, элементы второго массива изменят свою четность. Аккуратно разобрав случаи, все операции можно свести к обычному декартову дереву по неявному ключу.

Задача I. Эх, дороги...

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	2 с
Ограничение по памяти:	64 Мб

В многострадальном Тридесятом государстве опять готовится дорожная реформа. Впрочем, надо признать, дороги в этом государстве находятся в довольно плачевном состоянии. Так что реформа не повредит. Одна проблема "— дорожникам не развернуться, поскольку в стране действует жесткий закон" — из каждого города должно вести не более двух дорог. Все дороги в государстве двусторонние, то есть по ним разрешено движение в обоих направлениях (разумеется, разметка отсутствует). В результате реформы некоторые дороги будут строиться, а некоторые другие закрываться на бессрочный ремонт.

Петя работает диспетчером в службе грузоперевозок на дальние расстояния. В связи с предстоящими реформами, ему необходимо оперативно определять оптимальные маршруты между городами в условиях постоянно меняющейся дорожной ситуации. В силу большого количества пробок и сотрудников дорожной полиции в городах, критерием оптимальности маршрута считается количество промежуточных городов, которые необходимо проехать.

Помогите Пете по заданной последовательности сообщений об изменении структуры дорог и запросам об оптимальном способе проезда из одного города в другой, оперативно отвечать на запросы.

Формат входного файла

В первой строке входного файла заданы числа n — количество городов, m — количество дорог в начале реформы и q — количество сообщений об изменении дорожной структуры и запросов ($1 \le n, m \le 100\,000$, $q \le 200\,000$). Следующие m строк содержат по два целых числа каждая — пары городов, соединенных дорогами перед реформой. Следующие q строк содержат по три элемента, разделенных пробелами. "+ i j" означает строительство дороги от города i до города j, "- i j" означает закрытие дороги от города i до города j, "? i j" означает запрос об оптимальном пути между городами i и j.

Гарантируется, что в начале и после каждого изменения никакие два города не соединены более чем одной дорогой, и из каждого города выходит не более двух дорог. Никакой город не соединяется дорогой сам с собой.

Формат выходного файла

На каждый запрос вида "? i j" выведите одно число — минимальное количество промежуточных городов на маршруте из города i в город j. Если проехать из i в j невозможно, выведите -1.

Примеры

i.in	i.out
5 4 6 1 2 2 3 1 3 4 5 ? 1 2 ? 1 5 - 2 3 ? 2 3 + 2 4 ? 1 5	0 -1 1 2

Разбор задачи I. Эх, дороги...

Описанный в задаче граф представляет собой совокупность компонент в виде циклов и прямых линий. Будем хранить каждую компоненту в виде декартового дерева и отдельной пометки, является ли данная компонента прямой или циклом. При добавлении ребра может произойти одно из двух: объединятся две компоненты или линия замкнется в цикл. В первом случае одну из компонент может потребоваться развернуть, что можно сделать с помощью декартового дерева. При удалении ребра произойдет так же одно из двух: либо цикл разомкнется, либо компонента развалится на две. В первом случае необходимо сделать циклический сдвиг, во втором разрезать массив на два. При запросе расстояния необходимо проверить лежат ли вершины в одном дереве (в противном случае вывести -1). Если выполнено, то найти индексы в массиве каждого из элементов. Ответом будет модуль разности найденных индексов.

Задача J. Разреженные таблицы

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	4 с
Ограничение по памяти:	256 Мб

Дан массив из n чисел. Требуется написать программу, которая будет отвечать на запросы следующего вида: найти минимум на отрезке между u и v включительно.

Формат входного файла

В первой строке входного файла даны три натуральных числа n, m ($1 \le n \le 10^5, m \le 10^7$) и a_1 ($0 \le a_1 < 16714589$) — количество элементов в массиве, количество запросов и первый элемент массива соответственно. Вторая строка содержит два натуральных числа u_1 и v_1 ($1 \le u_1, v_1 \le n$) — первый запрос.

Элементы $a_2, a_3, \ldots, a_n$ задаются следующей формулой:

$a_{i+1} = (23 \cdot a_i + 21563) \bmod 16714589$.

Например, при $n = 10$, $a_1 = 12345$ получается следующий массив:

a = (12345, 305498, 7048017, 11694653, 1565158, 2591019, 9471233, 570265, 13137658, 1325095).

Запросы генерируются следующим образом:

$$u_{i+1} = ((17 \cdot u_i + 751 + ans_i + 2i) \bmod n) + 1,$$
$$v_{i+1} = ((13 \cdot v_i + 593 + ans_i + 5i) \bmod n) + 1,$$

где ans_i — ответ на запрос номер i.

Формат выходного файла

В выходной файл выведите u_m, v_m и ans_m (последний запрос и ответ на него).

Пример

j.in	j.out
10 8 12345 3 9	5 3 1565158

Разбор задачи J. Разреженные таблицы

В задаче необходимо реализовать разреженную таблицу. Суть этой таблицы заключается в следующем: предподсчитаем минимум на всех

отрезках длины равной какой-нибудь степени двойки. Таких отрезков $O(N \log N)$. После чего любой отрезок $[l, r]$ можно накрыть двумя перекрывающимися отрезками $[l, l + 2^k - 1]$ и $[r - 2^k + 1, r]$, где $2^k \le r - l + 1$ и максимально. Минимум из значений на этих отрезках и будет ответом. Нахождение требуемого k должно работать за $O(1)$ и быть предподсчитано для всех длин отрезков. Для проведения предподсчета воспользуемся рекурентной функцией $s[l][0] = a[l]$, $s[l][i] = s[l][i-1] + s[l + 2^{i-1}][i-1]$, где $s[l][i]$ — сумма на отрезке $[l, l + 2^i - 1]$.

Задача K. K-мерная частичная сумма

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	10 с
Ограничение по памяти:	64 Мб

Задана k-мерная таблица чисел $a_{i_1,i_2,\ldots,i_k}$, где $1 \le i_j \le n_j$ для j от 1 до k. Для заданных $l_1, \ldots, l_k, r_1, \ldots r_k$ найдите:

$$s_{l_1,\ldots,l_k,r_1,\ldots r_k} = \sum_{i_j=l_j}^{i_j \le r_j} a_{i_1,\ldots,i_k}.$$

Формат входного файла

В первой строке записано число k ($1 \le k \le 6$).

Во второй строке записаны размеры таблицы — n_j ($1 \le \prod n_j \le 10^6$). Далее записано $\prod_{j=1}^{k-1} n_j$ строк по n_k чисел, непревосходящих 1000, описывающих таблицу.

В очередной строке записано число q ($1 \le q \le 10^6$) — количество запросов. В следующих q строках описаны запросы:

$l_1, \ldots, l_k, r_1, \ldots r_k$ ($1 \le l_j \le r_j \le n_j$).

Формат выходного файла

Выведите q чисел в отдельных строках — ответы на запросы.

Примеры

k.in	k.out
3 2 3 5 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4 1 2 3 4 5 5 4 3 2 1 2 3 1 5 4 5 1 1 1 1 2 2 1 1 1 2 2 2 1 2 3 2 3 4 1 3 4 2 3 5 1 2 4 2 2 5	12 24 22 18 6

Пояснение

В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи K. K-мерная частичная сумма

Задача является обобщением предыдущей и требует лишь аккуратной реализации. Для реализации рекомендуется хранить данные в одномерном массиве, чтобы не зависеть от заданной размерности. Фактически, нужно уложить K-мерный массив в одномерный, а при индексации вычислять, где именно соответствующая ячейка находится в одномерном массиве.

Задача L. Минимум в стеке

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	3 с
Ограничение по памяти:	64 Мб

На вход вашей программе подается набор операций со стеком. Каждая операция — это добавить или удалить элемент из стека. После выполнения каждой операции вычислите наименьшее из всех чисел. Сложите

все полученные числа и получите ответ. Если после некоторой операции стек оказался пуст, то ничего не прибавляйте к ответу. Если выполнить удаление невозможно так как стек пуст, то не выполняйте его.

Формат входного файла

В этой задаче входные данные будут генерироваться прямо в вашей программе. На вход программе будут поданы параметры, чтобы вы смогли сгенерировать входную последовательность.

Первое число во входном файле n ($1 \le n \le 10^6$) — количество операций проводимых со стеком. Затем идут четыре неотрицательных целых числа a, b, c, x_0 не превосходящие 10000.

Для получения входных данных сгенерируем последовательность x.

Первое число в генерируемой последовательности — x_1. Первое как и каждое очередное число вычисляется из предыдущего: $x_i = (a \cdot x_{i-1}^2 + b \cdot x_{i-1} + c) \ / \ 100 \mod 10^6$.

'/' — это деление нацело, а 'mod' — это остаток от деления.

Если $x_i \bmod 5 < 2$, то необходимо удалить число из стека, в противном случае — нужно добавить в стек число x_i.

Формат выходного файла

Выведите единственное число — искомую сумму.

Примеры

l.in	l.out
2 0 0 1 81	0
3 1 1 1 13	0

Пояснение

Используйте 64-битный тип для генерации и вычисления ответа. В этой задаче очень большие входные данные, поэтому не используйте медленные способы чтения данных. В C++ не стоит использовать ifstream и ofstream (используйте fscanf и fprintf), а в Java не стоит использовать класс Scanner (используйте InputStreamReader).

Разбор задачи L. Минимум в стеке

Для решения этой задачи необходимо реализовать стек, каждый элемент которого дополнительно будет хранить в себе минимум от начала стека до его позиции. Тогда при удалении ничего дополнительного делать не придется, а при добавлении необходимо вычислить требуемый минимум.

Это легко сделать, выбрав минимум из значения, записанного в вершине стека и вновь добавленного значения.

Задача M. Минимум в очереди

Имя входного файла:	`m.in`
Имя выходного файла:	`m.out`
Ограничение по времени:	0,5 с
Ограничение по памяти:	64 Мб

На вход вашей программе подается набор операций с очередью. Каждая операция — это добавить или удалить элемент из очереди. После выполнения каждой операции вычислите наименьшее из всех чисел. Сложите все полученные числа и получите ответ. Если после некоторой операции очередь оказалась пуста, то ничего не прибавляйте к ответу. Если выполнить удаление невозможно так как очередь пуста, то не выполняйте его.

Формат входного файла

В этой задаче входные данные будут генерироваться прямо в вашей программе. На вход программе будут поданы параметры, чтобы вы смогли сгенерировать входную последовательность. Первое число во входном файле n ($1 \le n \le 10^6$) — количество операций проводимых с очередью. Затем идут четыре неотрицательных целых числа a, b, c, x_0 не превосходящие 10000.

Для получения входных данных сгенерируем последовательность x.

Первое число в генерируемой последовательности — x_1. Первое, как и каждое очередное число, вычисляется из предыдущего: $x_i = (a \cdot x_{i-1}^2 + b \cdot x_{i-1} + c)/100 \bmod 10^6)$.

'/' — это деление на цело, а 'mod' — это остаток от деления.

Если $x_i \bmod 5 < 2$, то необходимо удалить число из очереди, в противном случае нужно добавить в очередь число x_i.

Формат выходного файла

Выведите единственное число — искомую сумму.

Примеры

m.in	m.out
2 0 0 1 81	0
3 1 1 1 13	0

Пояснение

Используйте 64-битный тип для генерации и вычисления ответа.

Разбор задачи M. Минимум в очереди

Хранить дополнительные значения в очереди так, чтобы посчитать необходимый минимум невозможно. Самый очевидный выход из ситуации — это хранить параллельно очередь и кучу, синхронизируя их содержимое. Однако ограничения в задаче подобраны таким образом, чтобы требовалось решение за $O(N)$. Для достижения цели промоделируем очередь двумя стеками следующим образом: первый стек будет входящим, а второй — выходящим. Если нам требуется добавить элемент, то мы просто добавляем его во входящий стек, который, по сути, имитирует хвост очереди. Если требуется удалить элемент, то удаляем его из выходящего стека, который имитирует голову очереди. Если стек пуст, то перекладываем все элементы из "хвоста" в голову. При перекладывании "хвост" "переворачивается" и как раз на вершине оказывается первый пришедший элемент из "хвоста". Каждый элемент за свою жизнь добавляется и удаляется из двух стеков, поэтому суммарное количество действий — линейно.

Задача N. Range Variation Query

Имя входного файла:	`n.in`
Имя выходного файла:	`n.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

В начальный момент времени последовательность a_n задана следующей формулой: $a_n = n^2 \ mod \ 12345 + n^3 \ mod \ 23456$

Требуется много раз отвечать на запросы следующего вида:

- найти разность между максимальным и минимальным значением среди элементов $a_i, a_{i+1}, ..., a_j$;
- присвоить элементу a_i значение j.

Формат входного файла

Первая строка входного файла содержит натуральное число k — количество запросов ($k \leq 100\,000$). Следующие k строк содержат запросы, по одному на строке. Запрос номер i описывается двумя целыми числами x_i, y_i.

Если $x_i > 0$, то требуется найти разность между максимальным и минимальным значением среди элементов $a_{x_i}...a_{y_i}$. При этом $1 \leq x_i \leq y_i \leq 100\,000$.

Если $x_i < 0$, то требуется присвоить элементу a_{-x_i} значение y_i. При этом $-100\,000 \le x_i \le -1$ и $|y_i| \le 100\,000$.

Формат выходного файла

Для каждого запроса первого типа в выходной файл требуется вывести одну строку, содержащую разность между максимальным и минимальным значением на соответствующем отрезке.

Пример

n.in	n.out
7 1 3 2 4 -2 -100 1 5 8 9 -3 -101 2 3	34 68 250 234 1

Разбор задачи N. Range Variation Query

Данная задача является упражнением на дерево отрезков. В данном случае необходимо дерево отрезков, поддерживающее одновременно минимум и максимум.

День третий (21.02.2010г). Контест Михаила Левина

Об авторе...

Левин Михаил Владимирович родился в Ростове-на-Дону, там закончил английскую гимназию №36, музыкальную школу, занимался разными боевыми искусствами. С 6 класса участвовал в олимпиадах по математике, был кандидатом в сборную на международную олимпиаду школьников. Закончил мех-мат МГУ с красным дипломом, учится в аспирантуре кафедры алгебры. Чемпион России и NEERC ACM 2006 года, бронзовая медаль ACM ICPC World Finals 2007, серебряная медаль ACM ICPC World Finals 2008. 23 место на Global Google Code Jam 2006. Проработал 2.5 года в московском офисе Google инженером, с августа 2009 работает руководителем группы разработки в Яндексе, второй год ведет семинары по курсу “Алгоритмы и структуры данных поиска” в Школе Анализа Данных Яндекса, руководит стажировками ШАД.

Теоретический материал. Сканирующая прямая

Задача: на плоскости дано n отрезков. Есть ли среди них два пересекающихся? Есть тривиальное решение за $O(n^2)$. Мы улучшим его до $O(n \log n)$ с помощью метода сканирующей прямой. Изначально применим следующие ограничения: во-первых, пусть у нас нет вертикальных отрезков. Во-вторых, пусть никакие три отрезка не пересекаются в одной точке. Далее будет видно, что эти ограничения можно снять. Будем двигать вертикальную сканирующую прямую слева направо из минус бесконечности в плюс бесконечность. У нас иногда будут происходить “события”. Событий бывает два вида:

1. Начался отрезок — встретился левый конец какого-то отрезка.
2. Закончился отрезок — встретился правый конец какого-то отрезка.

Т.к. вертикальных отрезков нет, то левый и правый концы однозначно определены у всех отрезков.

Кроме того, в каждый момент времени мы будем поддерживать множество всех отрезков, которые сейчас пересекают сканирующую прямую. Это множество полностью упорядочено снизу вверх по ординате точки пересечения отрезка с прямой. Т.к. вертикальных отрезков нет, то каждый отрезок пересекает прямую ровно в одной точке.

При этом легко видеть, что при движении прямой слева направо порядок всех отрезков не меняется до тех пор, пока какие-то два из них не пересекутся и прямая не пройдет через эту точку пересечения. Этот факт мы и будем использовать в решении задачи.

Итак, процедура определения того, есть ли пересекающиеся отрезки, и нахождения какой-то пары пересекающихся в случае, когда она есть.

Изначально множество отрезков S, пересекающих прямую l, пусто. Все события упорядочим по возрастанию абсциссы, а при равенстве абсцисс сначала обрабатываем левые концы отрезков, потом правые. Внутри упорядочение снизу вверх по ординате.

Когда происходит событие начало отрезка s, нам нужно добавить отрезок в множество S, затем найти там его соседей сверху и снизу a и b и попробовать пересечь s с ними (если какого-то из соседей нет, то его не рассматриваем). Если с кем-то из них s пересекается, то мы нашли точку пересечения и пару отрезков и выходим.

Когда происходит событие конец отрезка s, нам нужно найти его соседей сверху и снизу a и b в текущем множестве (если какого-то из соседей нет, не рассматриваем его). Затем попробовать пересечь a с b, a с s, b с s. Если какая-то пара пересекается, то мы нашли точку пересечения и пару отрезков и выходим. Если не пересекаются, то удаляем s из множества.

Очевидно, что если никакие два отрезка не пересекаются, то и наш алгоритм не получит пересечения. Осталось доказать, что если есть пара пересекающихся отрезков, то алгоритм найдет пересечение. Пусть p — самая левая из всех точек пересечения отрезков. Утверждается, что, во-первых, алгоритм остановится прежде чем сканирующая прямая пройдет за точку p, а во-вторых, что алгоритм остановится и найдет при этом какую-то точку пересечения. В частности это означает, что порядок сегментов в множестве S ни разу не поменяется за время выполнения алгоритма.

Действительно, пусть в точке p пересекаются два отрезка AB и CD (мы предположили, что в каждой точке пересекаются не более чем два отрезка), A и C — их левые концы. Пусть CD был добавлен в множество S позже (т.е. в частности абсцисса точки C не меньше абсциссы точки A). Если в этот момент соседом CD был AB, то алгоритм нашел пересечение, и все в порядке. Если же нет, то это означает, что между CD и AB есть какой-то отрезок. Точнее, если обозначить за M точку пересечения вертикальной прямой, проходящей через точку C, с отрезком AB, то есть

какой-то отрезок, пересекающий треугольник CMp. Рассмотрим все такие отрезки. Ни один из них не может пересекать ни сторону Cp, ни сторону Mp треугольника CMp, т.к. тогда точка пересечения будет строго левее p, либо совпадать с p, но этого также не может быть, т.к. в точке p уже и так пересекаются отрезки AB и CD. Значит, ни один из этих отрезков не пересекает ломаную CpM. А это означает, что если мы возьмем самый правый из всех концов этих отрезков x, то x левее p. Рассмотрим момент, когда происходит событие x. В этот момент последний отрезок, разделяющий AB и CD в множестве S, удаляется из множества S, и мы проверяем AB и CD на пересечение, находим его, и т.д.

Что касается вертикальных отрезков, то с ними можно проводить те же самые рассуждения, если левым концом вертикального отрезка считать его нижний конец, правым — верхний, а точкой пересечения со сканирующей прямой — нижний конец. Упорядочение отрезков остается корректным, за исключением случая, когда какой-то отрезок пересекает вертикальный отрезок, но это пересечение будет отловлено еще до добавления вертикального отрезка в множество S.

Насчет многих отрезков, пересекающихся в одной точке - рассуждение выше легко исправляется, если вместо пары отрезков AB и CD рассмотреть сразу множество всех отрезков $A_1B_1, A_2B_2, \ldots, A_kB_k$, пересекающихся в точке p. Пусть $A_1, A_2, \ldots, A_k$ к тому же упорядочены по углу относительно p. Ни один из остальных отрезков не может пересекать ни один из отрезков $A_1p, A_2p, \ldots, A_kp$. Поэтому либо в момент добавления одного из A_iB_i мы найдем его пересечение с соседом A_jB_j, либо в момент удаления самого правого конца всех отрезков, пересекающихся с треугольником A_1pA_k, какие-то два из наших A_iB_i станут соседями, и мы найдем их пересечение. За счет разрешения вертикальных отрезков один из A_iB_i может быть вертикальным, но это ничего не меняет. А вертикальные отрезки левее p не играют роли.

Таким образом, алгоритм работает правильно. Чтобы оценить его сложность, необходимо сначала определиться со структурой данных для множества S. Конечно же мы захотим воспользоваться для него сбалансированным бинарным деревом поиска, а попросту говоря `set`ом. Свойство алгоритма о неизменности порядка сегментов в S при перемещении сканирующей прямой в ходе алгоритма гарантирует, что дерево всегда будет оставаться корректным. А функция сравнения при этом может даже формально зависеть от времени, реально вычисляя ординаты точек пересечения отрезков с *текущей* сканирующей прямой и сравнивая их. Можно и по-другому: про непересекающиеся отрезки можно определить, какой из них выше, используя только векторные произведения (и таким образом не выходя за пределы целых чисел), а факт пересечения отрезков можно

также определить не выходя за пределы целых чисел и с помощью векторного произведения. В случае сравнения двух пересекающихся отрезков в процессе вставки в дерево можно сразу же завершать работу алгоритма и выводить пересечение.

Поворачивающаяся сканирующая прямая

Многие задачи решаются не с помощью обычной сканирующей прямой, а с помощью поворачивающейся вокруг одной из точек заданного множества. В качестве центральной точки по очереди выступают все точки множества. Приведем пример такой задачи:

На плоскости дано n точек. Найти три из них, образующие треугольник наибольшей площади.

Решение.

Предположим, что мы зафиксировали две из трех вершин треугольника A и B. Тогда какая третья вершина даст наибольшую площадь при таких ограничениях? Та, что наиболее удалена от прямой AB. В нашем множестве есть наиболее удаленная среди точек слева от AB точка C_1 и наиболее удаленная среди точек справа от AB точка C_2. Треугольники ABC_1 и ABC_2 являются кандидатами на треугольник наибольшей площади со стороной AB. Ясно, что точки C_1 и C_2 являются вершинами выпуклой оболочки множества всех наших точек. При этом прямые, параллельные AB, проходящие через эти точки, называются опорными прямыми нашего множества, т.к. они содержат точку из множества, при этом все точки множества лежат с одной стороны от каждой из этих прямых, т.е. множество точек "опирается" на эти прямые. Для каждого направления v существует две опорные прямые множества, параллельные направлению v. А если поворачивать v по часовой стрелке, то опорные прямые будут поворачиваться в таком же направлении вокруг выпуклой оболочки, иногда совпадая с ее сторонами, иногда проходя через вершину. Это соображение дает нам идею быстрого решения (за $O(n^2 \log n)$) исходной задачи.

Давайте по очереди каждую точку зафиксируем в качестве точки A, а затем все остальные точки посортируем по углу вокруг вершины A и будем выбирать последовательно каждую из них в качестве точки B. Для самой первой точки B явным образом найдем две самые удаленные от AB точки C_1 и C_2 и опорные прямые, проходящие через них. Вычислим площади треугольников ABC_1 и ABC_2 и сравним с текущим ответом. Далее, переходя от точки B к следующей точке B', мы всегда поворачиваем вектор AB в одном и том же направлении. При этом опорные прямые, соответствующие направлению AB, тоже поворачиваются в том же направлении. Чтобы их найти, нужно знать выпуклую оболочку множества всех точек

и “бегать” опорными прямыми вокруг этой выпуклой оболочки до тех пор, пока не повернемся до нужного угла. Выпуклую оболочку множества точек мы предрассчитаем за $O(n \log n)$, а бегать будем просто двумя номерами вершин, через которые проходят текущие опорные прямые. За все время обхода точек B мы пройдем ровно один круг, соответственно номера вершин пройдут один полный круг, что произойдет за линейное время. В каждый момент времени мы знаем вершину A, текущую вершину B и опорные прямые, параллельные AB, а также вершины выпуклой оболочки, через которые они проходят — текущие C_1 и C_2. Имея эту информацию, мы опять считаем площади двух соответствующих треугольников и сравниваем с текущим ответом.

Выпуклую оболочку находим за $O(n \log n)$, затем для каждой из n точек выполняем сортировку за $O(n \log n)$, нахождение изначальных опорных прямых за $O(n)$, обход всех точек по порядку за $O(n)$. Итого получаем сложность $O(n^2 \log n)$. Затраты памяти $O(n)$.

Минимальное остовное дерево

Некоторые геометрические задачи тесно связаны с минимальным остовным деревом.

Например такая задача:

Задача про секретных агентов. Несколько секретных агентов ловят преступника и патрулируют местность, в которой он предположительно находится. Чтобы общаться между собой, они используют рации, передающие сигнал на ограниченное расстояние. Все рации одинаковые, и у них есть максимальное расстояние, на которое можно передать сигнал — R. Агенты весьма ленивы, поэтому они стоят на своих точках и не двигаются.

Как только один из агентов заметит преступника, он должен тут же передать эту новость вместе с координатами преступника всем остальным агентам. Для этого он может передать новость некоторым из тех агентов, до которых добивает его рация, те — расскажут своим соседям и т.д., но ни один агент при этом не должен сойти с места. Необходимо выяснить, какое минимальное значение R устраивает агентов, если задано их изначальное положение.

Решение.

У этой задачи есть несколько решений. Самое простое решение разумной сложности работает за $O(n^2 \log MAX)$, где MAX — максимально возможный модуль координаты агента. Здесь работает идея бинарного поиска по ответу. Понятно, что если R — ответ задачи, то при всех $R' < R$ граф $G_{R'}$, состоящий из вершин — агентов и ребер — отрезков между

ними длины не более R' — несвязен, а для любого $R' >= R$ $G_{R'}$ — связен. Соответственно, можно запустить бинарный поиск по всем возможным значениям R.

В качестве множества значений можно выбрать все вещественные числа от 0 до $2MAX$. Но в этом варианте придется во-первых вести бинарный поиск только до тех пор, пока не будет достигнута определенная точность (т.к. бинарный поиск по действительным числам сам по себе никогда не заканчивается), а во-вторых в программе придется производить вычисления в вещественных числах, что существенно дольше, чем если производить все вычисления в целых числах.

Как же нам не выйти за пределы целых чисел? Понятно, что в конце концов нам придется вычислить R, и это значение может и не быть целым. Однако можно как можно дольше отдалять этот момент: ведь квадраты расстояний — всегда целые числа. Можно, значит, делать бинарный поиск по множеству всех целых чисел в пределах от 0 до $2(2MAX)^2$. А еще ясно, что ответ R равен корню из длины отрезка от какого-то агента i до агента j. Поэтому можно вообще вести поиск по множеству квадратов всех длин отрезков между агентами. Их $O(n^2)$ – можно посортировать за $O(n^2 logn)$, и потом провести бинарный поиск с $\log(n)$ итерациями. На каждой итерации нужно проверить полный граф на связность (какие-то ребра на самом деле отсутствуют в графе в силу ограничения, но мы заранее не знаем, какие, поэтому как и в случае хранения графа с помощью матрицы смежности сложность поиска в ширину или глубину будет равна $O(n^2)$, а не $O(m+n)$). Итоговая сложность — $O(n^2 \log(n))$. Если бы мы делали бинарный поиск по всем целым числам от 0 до $2(2MAX)^2$, то получилось бы $O(n^2 \log(MAX)$. Здесь отличие небольшое.

В этом решении существенное ускорение (хоть и в константу раз) дает предрасчет всех попарных квадратов расстояний между агентами. Это происходит из-за того, что нам очень много раз нужно рассчитывать эти длины и сравнивать их с текущим ограничением. Хранение всех попарных расстояний требует $O(n^2)$ памяти. Поэтому если использовать эту оптимизацию, то затраты памяти будут $O(n^2)$. Заметим, что если мы пользуемся алгоритмом за $O(n^2 \log n)$ с сортировкой ребер, то нам в любом случае понадобится столько памяти, чтобы их отсортировать. Если же мы пользуемся алгоритмом за $O(n^2 \log(MAX)$ и не предрассчитываем длины ребер, то можно обойтись $O(n)$ памяти.

Существует решение быстрее, ненамного сложнее по теории и даже проще в реализации.

Речь пойдет о минимальном остовном дереве графа и алгоритме Прима [1].

Минимальное остовное дерево определяется как поддерево графа G,

содержащее все его вершины, и при этом сумма весов его ребер — минимально возможная. Нам в задаче про агентов нужно найти такой связный подграф графа G, что он содержит все вершины G, и при этом максимальное ребро подграфа — минимально возможное (если за G взять полный граф с вершинами — агентами, ребрами — отрезками между ними на плоскости, при этом длина ребра равна длине соответствующего отрезка). Дерево, обладающее таким свойством, называется узким остовным деревом. Изначально по определению это не одно и то же.

Однако несложно доказать, что любое минимальное остовное дерево является также и узким остовным деревом: действительно, рассмотрим минимальное остовное дерево T. Пусть его максимальное по длине ребро e не является минимально возможным. Удалим на время это ребро из дерева. Тогда оно распадется на две компоненты связности. Рассмотрим теперь какое-нибудь узкое остовное дерево. В нем эти два множества вершин каким-то образом связаны, а значит есть ребро узкого дерева e', соединяющее вершину из одного из этих множеств с вершиной из другого. При этом это ребро строго короче ребра e, т.к. является ребром узкого остовного дерева, а ребро e длиннее всех ребер узкого остовного дерева. Теперь возьмем и добавим в $T-e$ ребро e'. Две компоненты связности соединятся обратно, поэтому получившийся граф тоже будет остовным деревом. При этом сумма весов его ребер строго меньше, чем сумма весов ребер T — противоречие с тем, что T является минимальным остовным деревом.

Таким образом, чтобы решить задачу, нам достаточно построить минимальное остовное дерево полного графа G с вершинами — агентами и ребрами — отрезками между ними. Это можно сделать с помощью алгоритма Прима за $O(n^2)$. При этом мы сразу же будем знать длину максимального ребра, и она будет являться ответом к задаче. Затраты памяти $O(n)$.

Есть решение еще намного быстрее, однако оно использует алгоритм, сильно превосходящий по своей сложности все предыдущие, и изучать его мы здесь не будем. Однако решение на основе этого алгоритма (т.е. если считать, что мы знаем его и умеем его реализовывать), уже не настолько сложное, и его можно понять. Имеется в виду алгоритм построения диаграммы Вороного для n точек на плоскости за время $O(n \log n)$ [2, 3]. Чтобы понять, как нам в задаче помогут диаграммы Вороного, — нужно на время вернуться к узким остовным деревьям. Рассмотрим произвольное ребро минимального остовного дерева на наших n точках. Представим, что граф у нас действительно нарисован на плоскости, и проведены ровно те отрезки между точками, которые соответствуют ребрам дерева. Рассмотрим точку O — середину нашего ребра. Утверждается, что строго внутри круга с центром в точке O не лежит ни одна точка нашего множества. Действительно, пусть концы ребра — точки A и B, а точка C лежит внутри

круга. Тогда т.к. AB — диаметр, то $AC < AB$ и $BC < AB$. Если удалить ребро AB из дерева, то оно распадется ровно на две компоненты связности, и точка C будет лежать в одной из них. Предположим, не ограничивая общности, что точка C будет лежать в той же компоненте, что и точка A. Тогда добавим в наш граф из двух компонент связности ребро BC. Граф станет связным, значит станет остовным деревом. При этом сумма весов его ребер — строго меньше, чем сумма весов ребер нашего изначально дерева, т.к. $BC < AB$, мы удалили AB и добавили BC вместо него. Получаем противоречие с минимальностью исходного остовного дерева.

Значит, доказано следующее утверждение: для любого ребра любого минимального остовного дерева T графа на наших n точках середина отрезка, соответствующего ребру, ближе к концам этого отрезка, чем к любой другой из наших n точек (это лишь переформулировка утверждения про окружность с центром в точке O: расстояние от точки O до точек A и B равно $\frac{AB}{2}$, а расстояние до любой другой точки — не меньше $\frac{AB}{2}$). Это в свою очередь означает, что точка O лежит на ребре диаграммы Вороного, соответствующем паре точек (A, B), и не только лежит, но и является точкой пересечения этого ребра диаграммы Вороного с отрезком AB.

Что это нам дает? Это дает нам следующий алгоритм: построим диаграмму Вороного, и при построении для каждого ребра запомним, какой паре точек это ребро соответствует (т.е. для какой пары точек это ребро представляет собой геометрическое место точек, равноудаленных от этой пары точек и при этом удаленных от них менее, чем от любой другой из n точек). Затем построим такой граф на наших n точках: пройдем по всем ребрам диаграммы Вороного и для каждого из них добавим в граф ребро между парой точек, соответствующей этому ребру. Вследствие вышесказанного, этот граф будет содержать любое минимальное остовное дерево полного графа на наших n точках. С другой стороны, в диаграмме Вороного лишь линейное по n количество ребер, поэтому и в нашем графе будет $O(n)$ ребер. Тогда с помощью алгоритма Крускала можно построить в нем минимальное остовное дерево за $O(n \log n)$ [4]. Максимальное ребро этого дерева и даст нам ответ к задаче. Суммарная сложность — $O(n \log n)$. Затраты памяти $O(n)$.

Задачи и разборы

Задача A. Наибольший круг

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Ура! Вам, наконец, удалось купить клочок земли почти в 400х километрах от Москвы. Только ... только это было так дорого, что Вы не могли позволить себе построить там дом. Таким образом, Вы решили построить бассейн. Он должен иметь форму круга и быть как можно больше внутри вашего участка. Тщательно измерив границы вашего участка, теперь Вы знаете, что это выпуклый N-угольник.

Каков максимально возможный радиус круглого бассейна в нем?

Формат входного файла

В первой строке входного файла содержится целое число N, $3 \le N \le 10\,000$. Следующие N строк содержат по два целых числа каждая, x_i и y_i, не превышающих 10^7 по абсолютной величине — координаты вершин участка (выпуклый многоугольник) против часовой стрелки. Никакие три вершины не лежат на одной прямой.

Формат выходного файла

Выведите искомый радиус. Ваши решения будут приниматься, если он округляется не более, чем на 10^{-3}.

Пример

a.in	a.out
4 0 0 1 0 1 1 0 1	0.5

Разбор задачи A. Наибольший круг

Пусть $a_i x + b_i y + c_i$ — уравнение прямой, на которой лежит i-я сторона многоугольника, относительно которой весь многоугольник лежит слева. Пусть при этом уравнение нормализовано, т.е. $a_i^2 + b_i^2 = 1$. Тогда известно,

что ориентированное расстояние от точки (x_0, y_0) до этой прямой равно как раз $a_i x_0 + b_i y_0 + c_i$.

Нам нужно найти наибольший круг, помещающийся в многоугольник, или, другими словами, найти внутри многоугольника точку, наиболее удаленную от его ближайшей стороны. Таким образом, наша задача сводится к задаче нахождения максимума функции

$f(x0, y0) = \min_i a_i x_0 + b_i y_0 + c_i$.

Функция f, как минимум из n выпуклых (линейных) функций, является выпуклой. Поэтому ее максимум возможно искать с помощью тернарного поиска [6]. Более того, при фиксированном x_0 функция f все еще остается выпуклой, поэтому даже при фиксированном x_0 ее максимум можно искать тернарным поиском. Таким образом, всю задачу можно решить с помощью двух вложенных тернарных поисков, первый по x, второй — по y. Внутри необходимо вычислить значения n линейных функций — уравнений сторон, которые необходимо предрассчитать. Сложность такого решения $O(n \log^2(\frac{1}{precision}))$.

Т.к. ограничения довольно большие, а тернарных поиска два и они вложенные, здесь имеет смысл (и приходится) применять распространенную оптимизацию тернарного поиска, ускоряющую его примерно в два раза. А именно, делить отрезок поиска на каждой итерации надо не на 3 равные части, как обычно, а делить нужно обеими точками исходный отрезок в соотношении золотого сечения. Это дает возможность при переходе к следующему, уменьшенному отрезку не вычислять еще раз значение самой функции в одной из новых точек деления, т.к. оно к этому моменту уже вычислено: вторая точка деления исходного отрезка становится первой точкой деления уменьшенного, если мы идем направо, и наоборот, первая точка деления исходного отрезка становится второй точкой деления уменьшенного отрезка. Этот метод имеет смысл рассматривать всегда, когда вычисление самой функции f, у которой мы ищем максимум, трудоемко (в нашем случае, $O(n)$ — существенное время), т.к. за счет этого ускорения мы добиваемся того, что вызывать подсчет f нужно только один раз на итерацию, а не два раза, при этом число итераций растет, но эта константа очень мала.

Задача B. Лужи

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Посчитайте общую площадь, покрытую водой n круго-подобных луж.

Формат входного файла

В первой строке входного файла содержится количество тестов K ($1 \leq K \leq 10$).

Каждый тест начинается с строки, содержащей целое число $1 \leq n \leq 100$. n следующих строк содержат координаты центра и радиус каждой лужи в формате $x_i\ y_i\ r_i$. Гарантируется, что все числа являются целыми числами. Они не превышают 1000 по абсолютному значению. Радиус всегда положителен.

Формат выходного файла

Выведите для каждого теста одно реальное число с восьмью цифрами после запятой — общая площадь, покрытая водой.

Пример

b.in	b.out
1 3 0 0 3 1 0 1 6 0 2	40.84070450

Разбор задачи B. Лужи

Задача решается с помощью метода сканирующей прямой. Найдем все “самые левые”, “самые правые” точки окружностей, а также все точки пересечения пар окружностей — особые точки.

Будем двигать вертикальную прямую слева направо из минус бесконечности в плюс бесконечность. По пути у нас будут происходить события — когда абсцисса прямой совпадет с одной из абсцисс особых точек. Между последовательными событиями находится полоса плоскости. В пересечении с этой полосой каждый круг представляет собой криволинейную трапецию

или пустое множество. Причем известно, что границы этих трапеций пересекаются лишь в вершинах. Давайте найдем площадь объединения всех этих трапеций для каждой конечной полосы, а потом просуммируем — и получим ответ к задаче.

В каждой полосе можно все стороны криволинейных трапеций отсортировать по возрастанию ординаты левого конца. При этом правые концы будут отсортированы в том же порядке, т.к. стороны трапеций не пересекаются нигде, кроме вершин. Далее можно применить стандартный алгоритм нахождения длины объединения n отрезков на прямой: идем слева направо и считаем вложенность. Когда находим полную отдельную связную компоненту, добавляем ее длину к ответу. В данном случае нужно будет добавлять не длину отрезка, а площадь криволинейной трапеции, образованной двумя дугами окружности. Для того, чтобы посчитать эту площадь, достаточно посчитать площадь обычной трапеции по формуле, а затем добавить к ней с правильными знаками площади двух круговых сегментов по краям криволинейной трапеции. Это также легко сделать по формуле, зная радиусы соответствующих окружностей.

Задача C. Не курить!

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Вася — хороший парень. Но у него есть плохая привычка — он курит. Всё то время, сколько Петя дружит с Васей, он пытается отучить его от этого. Но ему это так и не удалось, потому что Вася не хочет бросать курить.

Недавно Петя придумал способ, как отучить своего друга от курения. Вася — неряха, поэтому его сигареты не лежат в пачке, а разбросаны по огромному столу. Петя хочет брать несколько сигарет в день незаметно для Васи. Вася не заметит пропажи сигарет, если в день будет пропадать не более одной сигареты. Кроме того, Петя должен брать только ту сигарету, которая пересекается с какой-нибудь другой сигаретой на столе. Помогите Пете узнать, сможет ли он начать реализацию своего плана.

Формат входного файла

Сигарета представляется как отрезок прямой. В первой строке входного файла записано число N ($1 \leq N \leq 125\ \ 000$) — количество сигарет на Васином столе. Следующие N строк содержат описания сигарет: $(i+1)$-я

строка содержит координаты концов i-й сигареты — целые числа x_1, y_1, x_2, y_2 ($-10\,000 \le x_1, y_1, x_2, y_2 \le 10\,000$).

Формат выходного файла

В первой строке выходного файла выведите слово “YES”, если Пете удастся начать реализацию своего плана. Вторая строка должна содержать числа i и j: i — номер сигареты, которую должен взять Петя, j — номер сигареты, с которой она пересекается.

Если Петя не сможет взять ни одной сигареты, выведите в единственной строке выходного файла “NO”.

Пример

c.in	c.out
2 0 0 2 2 0 2 2 0	YES 1 2
2 0 0 0 5 5 0 5 10	NO

Разбор задачи C. Не курить!

См. лекцию.

Задача D. Длина объединения

Имя входного файла:	d.in
Имя выходного файла:	d.out
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Рассмотрим множество отрезков на прямой с целыми концами. Изначально множество пустое, в него могут добавляться отрезки, из него могут удаляться отрезки. После каждой операции вставки или удаления отрезка необходимо вывести общую длину объединения всех отрезков, лежащих на данный момент в множестве.

Формат входного файла

В первой строке входа записано целое число $1 \le n \le 100\,000$ — общее количество проделанных операций. Далее идут n строк, каждая из них устроена следующим образом. Первый символ — “+” , если это операция

вставки отрезка и "-", если это операция удаления отрезка. Далее в строке записаны через пробел два целых числа — левый и правый концы отрезка. Координаты концов по модулю не превосходят 1 000 000 000. Гарантируется, что удаляться будут только отрезки, которые перед этим были добавлены в множество. Одинаковые отрезки можно добавлять в множество. Каждый из них считается отдельным отрезком.

Формат выходного файла

В выход необходимо вывести ровно n чисел, по одному в строке — общую длину объединения всех отрезков в множестве после каждой из n операций добавления/удаления.

Пример

d.in	d.out
4 + 1 3 + 2 6 + 5 7 - 2 6	2 5 6 4

Разбор задачи D. Длина объединения

Вначале считаем все отрезки из входного файла и отсортируем все координаты концов всех отрезков. Получим набор точек (заданных координатой по оси Ox) $a_0, a_1, \ldots, a_{n-1}$, где n — общее количество различных точек среди концов отрезков. Построим дерево отрезков, листьями которого будут все отрезки вида $a_i a_{i+1}$. Вершины дерева на уровень выше будут соответствовать отрезкам $a_{2i} a_{2i+2}$ и т.д. Каждая вершина дерева отрезков соответствует по определению какому-то "каноническому" отрезку $a_i a_j$. Любой отрезок с концами в данных точках естественным образом распадается на $O(\log(n))$ канонических отрезков, причем это разбиение: это известно из лекций.

В каждой вершине дерева отрезков будем хранить несколько величин. Во-первых, будем хранить длину `real_length` реального отрезка $a_i a_j$ на прямой, соответствующего этой вершине. Во-вторых, будем хранить количество `covering_segments_count` отрезков в множестве, содержащих данный канонический отрезок в своем разложении. В-третьих, будем хранить общую длину `covered_length` подмножества данного канонического отрезка, покрытую отрезками из текущего множества. Иными словами, длину объединения всех отрезков, пересеченного с данным каноническим отрезком.

Как поддерживать значения этих трех величин при добавлении и удалении отрезков из множества?

Величина `real_length` не меняется при этих операциях. Мы подсчитываем ее одновременно с инициализацией дерева отрезков: длина отрезков вида $a_i a_{i+1}$ считается тривиально как $a[i+1] - a[i]$, а длина канонического отрезка `s.real_length` считается как `left(s).real_length + right(s).real_length`, где `left(s)` — левый сын `s` в дереве, а `right(s)` — правый сын.

Величина `covering_segments_count` увеличивается на один, если при обработке добавления отрезка в множества мы рассматриваем текущий канонический отрезок (в этом и только в этом случае он в ходит в разложение отрезка на канонические отрезки). Аналогично, она уменьшается на единицу, если текущий канонический отрезок обрабатывается при удалении отрезка из множества.

Наконец, величина `covered_length` равна `real_length` в случае если текущее значение `covering_segments_count` положительно. Если же оно равно нулю, то только в этом случае нужно проводить реальные вычисления и записать `s.covered_length = left(s).covered_length + right(s).covered_length`

Сложность этого решения зависит от способа реализации. Можно реализовать так: при каждом добавлении и удалении отрезка сначала пересчитаем в соответствующих канонических отрезках значение `covering_segments_count`. Всего их $O(\log(n))$, и сложность этой операции тоже получается $O(\log(n))$. После чего нужно пересчитать значение `covered_length` во всех отрезках, в которых оно могло измениться. А именно, пусть мы пересчитали значение `covering_segments_count` в канонических отрезках $s_1, s_2, \ldots, s_k$. Тогда во всех вершинах на пути от каждого s_i к корню дерева отрезков могло измениться `covered_length`: в них могло быть `covering_segments_count = 0`, а значение `covered_length` могло измениться в их детях, а значит и в них. Если проапдейтить значения по порядку сначала во всех вершинах на пути от s_1 до корня, потом во всех вершинах от s_2 до корня и т.д., то т.к. $k = O(\log(n))$, и на пути от каждого s_i до корня $O(\log(n))$ вершин, то общая сложность каждой операции получается $O(\log^2(n))$. В этом случае общая сложность решения $O(n \log^2(n))$.

Однако можно поступить умнее. А именно, в стандартную рекурсивную реализацию дерева отрезков включить подсчет величины `covered_length` (величина `covering_segments_count` и так представляет собой стандартную величину для дерева отрезков, и ее подсчет делается так же, как в задаче RSQ). Как же это сделать? Пусть мы нахо-

димся в каком-то каноническом отрезке. Если он полностью содержится в нашем текущем рассматриваемом отрезке (добавляемом или удаляемом из множества), то дальнейшая рекурсия не нужна, мы пересчитаем наши величины только в текущем каноническом отрезке и выше, а в его поддерево не пойдем. Величина `covering_segments_count` пересчитывается очевидным образом. После этого, если `covering_segments_count = 0`, то

```
covered_length = left(s).covered_length + right(s).covered_length,
```

иначе `covered_length = real_length`.

Таким образом, мы рассмотрим, как всегда, лишь логарифмическое число канонических отрезков, и в каждом из них проведем $O(1)$ действий, поэтому сложность добавления и удаления отрезка $O(\log(n))$, а общая сложность решения — $O(n \log(n))$.

Затраты памяти в обоих случаях — $O(n)$ (в дереве отрезков $O(n)$ канонических отрезков).

Задача E. Внешние прямоугольники

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

На плоскости нарисовано n прямоугольников с вершинами в точках с целочисленными координатами, со сторонами, параллельными осям координат. Границы любых двух прямоугольников не имеют общих точек. То есть другими словами любые два прямоугольника либо расположены отдельно друг от друга, либо один из них — строго внутри другого. В такой ситуации, некоторые из прямоугольников — “внешние”, т.е. такие что ни один из них не лежит внутри никакого другого прямоугольника, а остальные прямоугольники — “внутренние”. Необходимо посчитать количество “внешних” прямоугольников.

Формат входного файла

В первой строке входа задано число n ($1 \le n \le 10^5$). В каждой из последующих n строк — по четыре целых числа $x_1,\ y_1,\ x_2,\ y_2$ ($-10^9 \le x_1,\ y_1,\ x_2,\ y_2 \le 10^9$), задающих координаты двух противоположных вершин соответствующего прямоугольника.

Формат выходного файла

В выход выведите одно число — количество “внешних” прямоугольни-

ков.

Пример

e.in	e.out
3 -3 -3 3 3 -2 2 2 -2 -1 -1 1 1	1
4 0 0 3 3 1 1 2 2 100 100 101 101 200 200 201 201	3

Разбор задачи E. Внешние прямоугольники

Будем двигать горизонтальную прямую снизу вверх, начиная из какого-то положения под всеми прямоугольниками и заканчивая положением над всеми прямоугольниками. У нас по пути будут происходить события. Событие “добавился прямоугольник”, когда прямая первый раз пересекла прямоугольник и событие “удалился прямоугольник”, когда прямая пересекла его последний раз и вышла за пределы прямоугольника. Эти события можно отсортировать по времени (если считать, что прямая двигается с постоянной скоростью; это то же самое, что отсортировать горизонтальные стороны прямоугольников по ординате). Если же какие-то две стороны различных прямоугольников находятся на одной высоте, то отсортируем их, например, по абсциссе самой левой точки стороны.

Заметим, что если какой-то “внутренний прямоугольник” A лежит в каком-то “внешнем” прямоугольнике B, то B добавится раньше, чем A, удалится позже, чем A, и при этом еще горизонтальная сторона A полностью помещается между концами горизонтальной стороны B.

Будем хранить множество отрезков, которые на данный момент лежат на прямой — все отрезки пересечения прямой с “внешними” прямоугольниками, с которыми она пересекается на данный момент. Это множество может изменяться только в момент, когда добавляется или удаляется прямоугольник. Будем хранить это множество в виде сбалансированного бинарного дерева, например красно-черного дерева. Отрезки в дереве отсортируем по левому концу. Неважно, на самом деле, по какому концу, т.к. отрезки не могут пересекаться: “внешние” прямоугольники не имеют общих точек.

Рассмотрим реализацию обоих событий:

Если добавляется прямоугольник A, то найдем, в какое место дерева его можно добавить (его горизонтальную сторону a). Найдем его потенциального левого соседа в дереве s — наибольший отрезок, левый конец которого не правее левого конца a. Проверим, не содержит ли отрезок s отрезок a целиком. Если содержит, то это означает, что соответствующий "внешний" прямоугольник S содержит прямоугольник A, а значит A не является "внешним", и его не нужно добавлять в дерево. В противном случае добавляем a в дерево.

Если удаляется прямоугольник A, то это — "внешний прямоугольник", в этот момент нам надо его посчитать: добавить единицу к ответу. Затем удалить его из дерева.

Операции с деревом производятся за $O(\log(n))$, где n — общее количество отрезков, поэтому общая сложность решения $O(n\log(n))$. Затраты памяти $O(n)$.

Задача F. Четырехугольники

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Вам дано N точек общего положения на плоскости, то есть, нет двух точек, которые совпадают, и нет трех таких точек, которые лежат на одной прямой. Узнайте, сколько существует различных выпуклых четырехугольников с вершинами в этих точках.

Формат входного файла

Одно целое число N ($4 \leqslant N \leqslant 1\,500$) в первой строке входного файла. Следующие N строк содержат два целых числа X_i and Y_i каждая — координаты точек. Все координаты не превышают 10^8 по абсолютному значению.

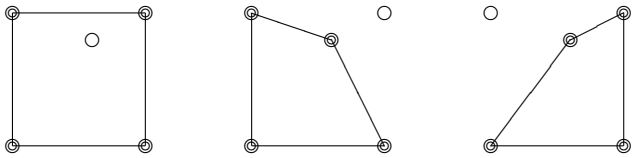
Формат выходного файла

Выведите одно целое число — количество различных выпуклых четырехугольников с вершинами в данных точках.

Пример

f.in	f.out
4 -1 -1 -1 1 1 1 1 -1	1
5 0 0 5 0 3 4 0 5 5 5	3

Последний пример проиллюстрирован ниже.



Разбор задачи F. Четырехугольники

Задача решается с помощью метода поворачивающейся сканирующей прямой.

Количество выпуклых четырехугольников совпадает с количеством четверок точек, таких что ни одна из них не лежит в треугольнике, образованном другими тремя. Посчитаем количество остальных четверок точек и вычтем ответ из C_n^4. Т.е. нам необходимо посчитать количество четверок точек таких, что одна из точек лежит в треугольнике, образованном остальными точками. Для этого посчитаем для каждой точки количество треугольников, в которых она лежит, и сложим.

Зафиксируем точку O, для которой будем считать количество треугольников из остальных точек, в которых она лежит. Это число в свою очередь можно выразить как общее количество треугольников из остальных точек (C_{n-1}^3) минус количество треугольников из остальных точек, в которых точка O не лежит. Отсортируем все точки по углу относительно O. Рассмотрим произвольный треугольник, в котором O не лежит. В нем есть ровно одна вершина (обозначим ее A) такая что остальные две вершины

(обозначим их B и C) лежат слева от прямой OA. Иначе O лежало бы внутри ABC. Таким образом, для того чтобы посчитать количество треугольников, в которых O не лежит, надо перебрать все точки в качестве A, и для каждой такой посчитать количество пар точек (B, C), лежащих слева от прямой OA. Такой подсчет можно сделать после сортировки точек по углу относительно O за линейное время: для самой первой точки A явным образом находим последнюю в отсортированном порядке точку, еще лежащую слева от OA. Все предыдущие тоже лежат слева от OA — знаем их количество k для OA, тогда количество искомых пар равно C_k^2. Переходя к следующей по порядку точке A', мы сдвигаем указатель на последнюю точку, лежащую слева от OA' только вперед, и так будет каждый раз при переходе к следующей по порядку точке. Поэтому у нас будет два указателя, один из которых пройдет ровно один круг, а другой никогда его не обгонит — значит они оба пройдут в сумме линейное количество шагов. В процессе этого для каждой точки A мы посчитаем количество пар (B, C) слева от прямой OA, просуммируем эти числа и таким образом получим количество треугольников ABC, таких что O не лежит внутри ABC. Чтобы получить из этого количество треугольников, содержащих O, надо вычесть это число из C_{n-1}^3.

Задача G. Радиопередатчики

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	4 с
Ограничение по памяти:	256 Мб

Известно, что если рядом расположены два передатчика, вещающие на одной частоте, то качество приема резко ухудшается. Вам даны местоположения N передатчиков. Известно, что каждый из них может использовать одну из двух доступных частот. Мощности всех передатчиков равны между собой и равны W. В данном случае под мощностью понимается радиус действия уверенного приема вокруг передатчика. Какую наибольшую мощность W могут иметь эти передатчики, чтобы не существовало области положительной площади уверенного приема двух передатчиков, вещающих на одинаковой частоте? Выбор частоты вещания каждого передатчика остается за вами. Профессор Кнутмен утверждает, что существует решение этой задачи за время, пропорциональное квадрату числа N, и это решение достаточно эффективно для полного решения этой задачи.

Формат входного файла

В первой строке входного файла содержится целое число N ($3 \leq N \leq 7500$) — количество передатчиков. Далее в N строках записаны координаты каждого передатчика. Все координаты — целые числа, не превосходящие 10000 по абсолютной величине. Все передатчики имеют различные координаты.

Формат выходного файла

В первую строку выведите максимальную мощность передатчиков W. Мощность выводите не менее чем с шестью знаками после запятой. Во вторую строку выведите N чисел — номера частот передатчиков. Число 1 в i-ой позиции обозначает, что i-ый передатчик должен вещать на первой частоте. Число 2 обозначает, что на второй. Если решений несколько, выведите любое.

Пример

g.in	g.out
4 0 0 0 1 1 0 1 1	0.7071067811865475 2 1 2 2 1

Разбор задачи G. Радиопередатчики

В этой задаче требовалось решение за $O(n^2)$. Оно получается с помощью построения минимального остовного дерева за $O(n^2)$. Построим какое-нибудь MST. Раскрасим все его вершины в два цвета. Рассмотрим все пары вершин одного цвета. Для любой такой пары вершин, если добавить соответствующее ребро e в дерево, в графе появится цикл. Новое ребро обязано быть максимальным ребром в этом цикле, т.к. у нас минимальное остовное дерево. Кроме того, этот цикл обязательно нечетной длины. А это значит, что при любом способе выбора частот в каждой вершины в этом цикле есть две соседние вершины с одной и той же частотой. И расстояние между этими двумя вершинами уж никак не больше длины ребра e. А это означает, что мощность передатчиков не может быть более $\frac{length(e)}{2}$. Это верно для любой пары вершин одного цвета и соответствующего ребра e. Давайте рассмотрим число m — минимум по всем парам вершин величины $\frac{length(u,v)}{2}$. С одной стороны, ответ задачи, как мы уже

выяснили, не больше m. С другой стороны, если сделать все передатчики ровно мощности m, а частоты передатчиков выбрать в соответствии с изначальной раскраской нашего минимального остовного дерева, то мы получим корректное распределение частот. Таким образом, ответ к задаче равен m. Сколько нужно времени для вычисления m? Сначала $O(n^2)$ на построение минимального остовного дерева, затем $O(n)$ на раскраску дерева в два цвета, а потом еще $O(n^2)$ на перебор всех пар вершин одинакового цвета. Таким образом, общая сложность решения $O(n^2)$, необходимые затраты памяти $O(n)$.

Задача H. Автомобильная стоянка

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Имеется циклическая автомобильная стоянка, то есть такая, в которой все парковочные места расположены по кругу. Всего есть n парковочных мест, и они пронумерованы от 1 до n. На стоянку приезжают и уезжают автомобили. Каждый автомобиль заранее целенаправленно подъезжает к определенному парковочному месту — месту, выделенному для хозяина автомобиля. Однако частенько случается, что кто-то уже занял это место, тогда автомобиль начинает ехать по часовой стрелке (числа расположены в порядке $1, 2, 3, \ldots, n, 1, 2, 3, \ldots$ по часовой стрелке) до тех пор, пока не найдет первое свободное место и не встанет на него. Если же мест нет, то автомобиль уезжает, так и не попав на стоянку.

Формат входного файла

В первой строке входа записано два целых числа $1 \le n, m \le 100\ 000$, где n — количество мест на стоянке, а m — количество событий, произошедших за день. Событие — это либо приезд, либо отъезд автомобиля. В следующих m строках — по событию в строке. Прибытие автомобиля записывается как "+ i" (между знаком "+" и числом есть пробел), где i — номер места, к которому изначально направляется автомобиль. Отъезд записывается как "- i" (между знаком "-" и числом есть пробел), где i — номер места на стоянке.

Формат выходного файла

Для каждого приезда автомобиля выведите в выход единственную строку, в которой записан номер места, на которое он встал, либо -1, если

места не нашлось. Для каждого отъезда автомобиля выведите 0, если на данном месте действительно был автомобиль, и -2, если автомобиля на месте i не было.

Пример

h.in	h.out
4 8 + 2 + 2 + 2 + 2 + 2 + 2 - 2 - 2	2 3 4 1 -1 -1 0 -2

Разбор задачи H. Автомобильная стоянка

Первый способ.
В этой задаче по сути нужно реализовать три операции:

- найти первое по часовой стрелке свободное место от данной стоянки;
- проверить, свободно ли данное место;
- сделать свободное место занятым или наоборот.

Можно просто реализовать нашу циклическую стоянку как булевский массив, в котором значение `true` соответствует свободному месту и `false` занятому. Тогда вторая и третья операция тривиально реализуются за $O(1)$. Однако первая операция в этом случае работает за $O(n)$, т.к. нам нужно пройтись от текущего места по всему массиву циклически до первого свободного места, которого, в принципе, может и не быть, и тогда будет сделано n действий. Поэтому общая сложность решения в худшем случае составит $O(n^2)$, что нам не подходит при данных ограничениях ($n = 100000$).

Вместо первых двух операций реализуем одну: посчитать общее количество свободных мест от i-го места для стоянки до j-го включительно. На базе этой операции реализуются обе исходные операции. Вторая операция просто проверит количество свободных мест от i-го до i-го места и сравнит с единицей. Первая операция реализуется чуть сложнее. Пусть нужно найти первое свободное место в циклическом массиве, начиная с i-го места. Для начала проверим, есть ли свободное место от i-й до $n-1$-й

позиции. Если есть, то дальше будем искать уже только на этом отрезке. Если нет, то проверим, есть ли свободное место от 0-й до $i-1$-й позиции. Если там тоже нет свободных мест, значит свободных мест нет вообще. Если же есть, то будем искать только на этом отрезке. Итак, в общем случае мы будем искать первое свободное место либо на отрезке от 0 до $i-1$, либо на отрезке от i до $n-1$. Эти задачи аналогичны, так что будем решать более общую задачу: найти первое свободное место на отрезке от i до j включительно, если известно, что между двумя этими позициями точно есть свободное место. Как мы будем это делать? Запустим бинарный поиск. Посчитаем количество свободных мест на позициях от i-й до mid(i, j) включительно, где $mid(i,j)$ — середина отрезка от i-й до j-й позиции. Если есть хотя бы одно свободное место, то переходим от отрезка (i,j) к отрезку $(i, mid(i,j))$. Иначе свободное место обязательно есть от позиции $mid(i,j)+1$ до позиции j, и мы переходим от отрезка (i,j) к отрезку $(mid(i,j)+1, j)$. Таким образом, мы на каждом шаге будем уменьшать длину отрезка в два раза, и вторая операция реализуется за $O(\log(n))$ умножить на сложность операции "посчитать количество свободных мест на отрезке".

Теперь осталось понять, как нам побыстрее справиться с этой операцией вместе с третьей из изначально указанных. Проще всего здесь воспользоваться деревом Фенвика. Оно умеет за $O(\log(n))$ считать сумму чисел на отрезке от i до j, а также прибавлять к ячейке число. Соответственно, построим дерево Фенвика над нашим изначальным массивом из нулей и единиц: ноль — занятое место, единица — свободное. Тогда подсчет количества свободных мест на отрезке — это и есть сумма на отрезке. А занять свободное место или освободить место равносильно прибавлению единицы или соответственно минус единицы к значению в ячейке. Соответственно, первая из трех изначальных операций работает за $O(\log(n)) \cdot O(\log(n) = O(\log^2(n))$, а остальные две — за $O(\log(n))$. Итоговая сложность решения — $O(m \log^2(n))$, где m — количество запросов. Затраты памяти — $O(n)$, т.к. дереву Фенвика требуется только $O(n)$ памяти.

Второй способ.

Вместо подсчета сумм на отрезке с помощью дерева Фенвика можно воспользоваться деревом отрезков, правда с необычной функцией: наше дерево отрезков будет просто-напросто находить самую левую единицу на отрезке от i до j. При этом еще умеет проверять значение в точке и изменять его с нуля на единицу и наоборот. Это покрывает все три операции, перечисленные в начале первого решения. Проверка значения в точке и изменение значения в одной точке — стандартная функция для дерева отрез-

ков. А самая первая единица на отрезке — локальная функция, поэтому ее тоже можно считать с помощью дерева отрезков. Если на каком-то отрезке нет единиц, то в качестве позиции самой левой единицы будем записывать -1, иначе будем записывать индекс этой позиции. Для отрезков единичной длины посчитать ответ тривиально. А дальше для каждого канонического отрезка смотрим на его детей: если в левом сыне не -1, то переписываем значение из левого сына. Если в левом сыне -1, то переписываем значение из правого сына. При изменениях нуля на единицу или единицы на ноль идем снизу вверх и пересчитываем по этому алгоритму позицию самой левой единицы. Соответственно, все операции реализуются за $O(\log(n))$, и сложность всего решения $O(m\log(n))$, где m — количество запросов. Затраты памяти этого решения — $O(n)$, т.к. дереву отрезков требуется $O(n)$ канонических отрезков.

Задача I. Пересечение отрезков

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Два отрезка $[A,\ B]$ и $[C,\ D]$ на плоскости заданы координатами своих концов — точек $A,\ B,\ C,\ D$: $(X_a, Y_a),\ (X_b, Y_b),\ (X_c, Y_c),\ (X_d, Y_d)$.

Требуется найти пересечение этих отрезков и вывести:

- слово `Empty`, если эти отрезки не пересекаются;
- координаты точки пересечения, если пересечение состоит из единственной точки;
- координаты точек — начала и конца отрезка пересечения в лексикографическом порядке, если пересечение заданных отрезков — отрезок.

Формат входного файла

Четыре строки файла исходной информации содержат по два целых значения, по модулю не превосходящих 1000 — координаты концов точек $A,\ B,\ C,\ D$. Отрезки могут быть вырожденными.

Формат выходного файла

Числовые значения в ответе следует округлить до десяти знаков после десятичной точки.

Пример

i.in	i.out
0 0 9 9 9 5 0 5	5.0000000000 5.0000000000
0 0 9 9 15 15 7 7	7.0000000000 7.0000000000 9.0000000000 9.0000000000
0 0 9 9 10 10 10 10	Empty

Разбор задачи I. Пересечение отрезков

Чтобы определить, пересекаются ли прямые, содержащие отрезки AB и CD, нужно построить уравнения прямых $a_1x + b_1y + c_1 = 0$ и $a_2x + b_2y + c_2 = 0$ по двум точкам и посчитать определитель этой системы уравнений. Если определитель ненулевой, то прямые пересекаются, и по формулам Крамера можно найти их точку пересечения. Далее нужно реализовать функцию `between(a, x, b)`, определяющую, лежит ли точка x между точками a и b, и с помощью нее определить, лежит ли точка пересечения прямых на обоих отрезках. Функция `between` для точек пишется на основе функции `between` для вещественных чисел, с учетом погрешностей вычислений. При этом `between` должна возвращать нестрогое “между”.

Если прямые не пересекаются, то они либо параллельны, либо совпадают. Это можно определить, посчитав векторное произведение AB на AC. Если оно не равно нулю, то прямые параллельны, и общих точек нет. Иначе отрезки лежат на одной прямой. В этом случае можно, например, для каждого из 4 концов всех отрезков проверить, принадлежит ли он другому отрезку, создать множество всех концов отрезков, принадлежащих обоим отрезкам, затем взять в нем два крайних элемента и вывести как пересечение этих двух отрезков. Если множество пусто, то и пересечения нет.

В этой задаче при желании можно обойтись без вычислений в вещественных числах за исключением одного деления в случае, когда нужно вывести точку пересечения обычных пересекающихся отрезков. Сравни-

вать вещественные числа не обязательно. Очевидно, в случае параллельных и совпадающих прямых, все делается с помощью векторных произведений, поэтому и не нужно выходить за пределы целых чисел. Чтобы определить, пересекаются ли два отрезка с целыми концами, достаточно сначала определить, пересекаются ли их ограничивающие прямоугольники, а если да, то посчитать еще несколько векторных произведений, на основе которых вычисляется, лежат ли точки A и B по разные стороны от прямой CD или нет, и то же самое насчет точек C и D и прямой AB.

Но главное здесь просто аккуратно рассмотреть все случаи. На удивление эту простую задачу редко сдают с первой попытки даже опытные люди.

Задача J. Наибольший круг (Юниорская лига)

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Ура! Вам наконец, удалось купить клочок земли почти в 400х километрах от Москвы. Только ... только это было так дорого, что Вы не могли позволить себе построить там дом. Таким образом, Вы решили построить бассейн. Он должен иметь форму круга, и быть как можно больше внутри вашего участка. Тщательно измерив границы вашего участка, теперь Вы знаете, что это выпуклый N-угольник.

Каков максимально возможный радиус круглого бассейна в нем?

Формат входного файла

В первой строке входного файла содержится целое число N, $3 \le N \le 50$. Следующие N строк содержат по два целых числа каждая, x_i и y_i, не превышающей 10^7 по абсолютной величине — координаты вершин участка (выпуклый многоугольник) против часовой стрелки. Никакие три вершины не лежат на одной прямой.

Формат выходного файла

Выведите искомый радиус. Ваши решения будут приниматься, если он округляется не более, чем на 10^{-3}.

Пример

j.in	j.out
4 0 0 1 0 1 1 0 1	0.5

Разбор задачи J. Найбольший круг (Юниорская лига)

Перебираем все тройки сторон многоугольника. Какой-то тройки наш круг обязан касаться. Чтобы касался трех сторон, центр должен лежать на пересечении биссектрис углов между этими прямыми. Перебираем между двумя парами сторон 4 пары биссектрис, все попарно пересекаем и проверяем такой центр. Для фиксированного центра радиус – минимум из расстояний до вершин и до сторон многоугольника. Выбираем максимальный $O(n^4)$.

Задача K. Радиопередатчики 2 (Юниорская лига)

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Известно, что если рядом расположены два передатчика, вещающие на одной частоте, то качество приема резко ухудшается. Вам даны местоположения N передатчиков. Известно, что каждый из них может использовать одну из двух доступных частот. Мощности всех передатчиков равны между собой и равны W. В данном случае под мощностью понимается радиус действия уверенного приема вокруг передатчика. Какую наибольшую мощность W могут иметь эти передатчики, чтобы не существовало области положительной площади уверенного приема двух передатчиков, вещающих на одинаковой частоте? Выбор частоты вещания каждого передатчика остается за вами.

Формат входного файла

В первой строке входного файла содержится целое число N ($3 \le N \le 1000$) — количество передатчиков. Далее в N строках записаны координаты каждого передатчика. Все координаты — целые

числа, не превосходящие 10000 по абсолютной величине. Все передатчики имеют различные координаты.

Формат выходного файла

В первую строку выведите максимальную мощность передатчиков W. Мощность выводите не менее чем с шестью знаками после запятой. Во вторую строку выведите N чисел — номера частот передатчиков. Число 1 в i-ой позиции обозначает, что i-ый передатчик должен вещать на первой частоте. Число 2 обозначает, что на второй. Если решений несколько, выведите любое.

Пример

k.in	k.out
4 0 0 0 1 1 0 1 1	0.70710678118654752 1 2 2 1

Разбор задачи K. Радиопередатчики 2 (Юниорская лига)

Запоминаем в массив все квадраты попарных расстояний между вершинами. В двумерный массив для обращения и в еще один большой одномерный — для сортировки. Отсортировали одномерный, далее по нему бинарный поиск. В бинарном поиске квадрат стороны из большого одномерного массива является ограничением на квадрат длины ребра. Для соответствующего графа проверяем, можно ли его раскрасить в 2 цвета, с помощью поиска в глубину. Если можно, идем направо в бин. поиске, иначе идем налево. Возвращаем корень из найденного значения, деленный еще пополам $O(n^2 \log n)$.

День четвертый (22.02.2010г). Контест Эльдара Богданова и Андрея Луценко

Об авторах...

Эльдар Богданов родился в 1988 году в г.Тбилиси. Закончил грузинский научно-технический лицей, занимался программированием в учебном центре “Мзиури”. В 2009 году окончил Тбилисский Государственный Университет со степенью бакалавра компьютерных наук. Ныне студент первого курса магистратуры Грузинского Университета им. св. Андрея Первозванного.

Основные достижения:

- призер республиканской Олимпиады школьников по информатике 2005 года;
- третий призер Открытого чемпионата Москвы 2008 года, Кубка Векуа 2008 и Открытого чемпионата Украины 2007 и 2009 годов;
- второй призер Открытого чемпионата Южного Кавказа 2007 года, Олимпиады им. Лебедева и Глушкова 2008 года и Зимней Школы по программированию 2009 года;
- чемпион Грузии, Южного Кавказа и Украины 2008 года;
- полуфиналист Google Code Jam 2008 года;
- бронзовый призёр финала ACM ICPC 2009 года.

Андрей Луценко родился в 1988 году в г. Тбилиси. В 2009-ом году закончил Тбилисский Государственный Университет со степенью бакалавра компьютерных наук. Ныне студент первого курса магистратуры ТГУ. Основные профессиональные интересы: программирование для мобильных устройств, проектирование и разработка многопользовательских систем с использованием клиент-серверных СУБД. Обладатель дипломов личного зачета Чемпионата Южного Кавказа 2007, 2008 и 2009 годов, командного зачета Кубка Векуа 2007. Хобби: горный велосипедный спорт, футбол.

Теоретический материал. Комбинаторная теория игр. Теорема Шпрага-Гранди

Комбинаторная теория игр изучает так называемые *комбинаторные игры*. Обычно под этим термином подразумевают игры, удовлетворяющие следующим условиям:

- в игре участвуют два игрока;
- определено множество всех возможных позиций игры. Обычно также задается некоторая начальная позиция;
- правила игры указывают для обоих игроков и каждой позиции, в какие позиции можно из неё перейти в результате допустимого хода;
- игроки делают ходы по очереди;
- игра заканчивается, когда у игрока, которому предстоит ходить, нет допустимых ходов. Тогда один из игроков объявляется победителем. Правилами игры гарантируется, что такой момент наступит, как бы ни играли противники;
- оба игрока располагают полной информацией;
- в игре не присутствует элемента случайности.

Как видите, большинство из привычных нам игр не удовлетворяют всем условиям. В карточных играх и играх с использованием костей многое зависит от случая. “Морской бой” подразумевает, что игроки располагают не всей информацией, то есть в данном случае не знают расположения кораблей противника. В крестиках-ноликах игра может закончиться ничьей. С той же проблемой мы сталкиваемся в случае шахмат и шашек.

Что же тогда относится к комбинаторным играм? Рассмотрим несколько примеров.

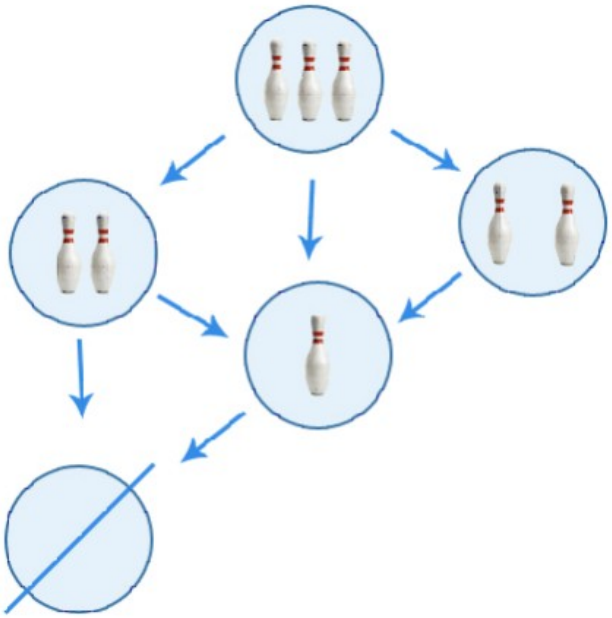
Ним. У двух игроков имеется несколько кучек с камнями. Они ходят по очереди. Ход заключается в том, чтобы взять ровно из одной кучки некоторое количество камней — не меньше одного и не больше количества камней в этой кучке. Игра заканчивается, когда камней ни в одной кучке больше не остается. Побеждает тот, кто последним взял камень.

Kayles. Есть несколько выстроенных в один ряд кегель. Два игрока по очереди делают ход, который заключается в том, чтобы сбить одну или две соседние кегли. Выигрывает сбивший последнюю в игре кеглю.

Dawson’s Kayles. Эта игра отличается от предыдущей тем, что за один ход сбивают ровно две кегли (то есть в некоторых случаях игра заканчивается даже при наличии кегель, если они все стоят в одиночку) и что последний, сделавший ход, проигрывает.

Hackenbush. В этой игре на листе бумаги нарисовано несколько кривых, некоторые из которых соединены концами друг с другом или с “землей” — обычно горизонтальной линией внизу листа. Два игрока по очереди стирают кривые с листа. Если в результате стирания какие-либо кривые полностью теряют связь с землей, они тоже стираются. В ранних версиях игры оба игрока могли стереть любые кривые. Таким образом, игра заканчивалась, когда от рисунка ничего не оставалось. Позднее, кривые сделали разноцветными. В Blue-Red Hackenbush кривые красного и синего цвета, и один из игроков может стирать только красные, а другой — только синие кривые. Игра заканчивается, если у игрока, которому предстоит сделать ход, больше нет кривых соответствующего цвета.

Каждую комбинаторную игру можно представить в виде ориентированного ациклического графа, в котором вершины соответствуют позициям в игре, а ребра — допустимым ходам. К примеру, граф для Kayles с тремя кеглями выглядит так (здесь учтены изоморфизмы некоторых позиций):



***P*- и *N*-позиции.**

Так как в рассматриваемых нами играх оба игрока обладают полной информацией, нет случайностей, и игра заканчивается победой одного из противников, позиции в игре можно разделить на две категории: *P*-позиции (сокращенно от слова *Previous* — предыдущий), в которых при оптимальной стратегии игру выигрывает сделавший предыдущий ход, и *N*-позиции (от слова *Next* — следующий), в которых победителем должен стать игрок, которому предстоит ходить. Возникает вопрос — как определить статус каждой позиции и упомянутую оптимальную стратегию, то есть какие ходы делать, чтобы гарантированно одержать победу?

Давайте проведем анализ для графа, составленного для 3 кегль в Kayles. Позиция, в которой кегель уже не осталось, очевидно, является *P*-позицией. Позиция с одной или двумя соседними кеглями является *N*-позицией, потому что из них обеих можно перейти в позицию без кегель и выиграть. Из позиции с двумя отдельно стоящими кеглями можно перейти только в позицию с одной кеглей, в которой выигрывает следующий игрок - следовательно, это *P*-позиция. И наконец, из позиции с тремя кеглями можно, сбив среднюю кеглю, перейти в позицию с двумя отдельными кеглями, где следующий игрок проигрывает — значит, это *N*-позиция. Таким образом, *P*- и *N*- позиции можно определить рекурсивно:

1. Любая конечная позиция (то есть позиция, из которой нет допустимых ходов) является *P*-позицией.

2. Неконечная позиция является P-позицией, если все позиции, в которые можно из неё перейти, являются N-позициями.

3. Неконечная позиция является N-позицией, если из неё можно перейти в хотя бы одну P-позицию.

Так можно определить P- и N-позиции для любой игры, в которой делающий последний ход побеждает. Если же делающий последний ход проигрывает, то вместо пункта 1 мы имеем “все конечные позиции являются N-позициями”.

Оптимальную стратегию на основе этой информации определить уже не составляет труда: если игрок находится в N-позиции, он должен сделать ход в любую P-позицию и в итоге гарантированно выиграет. Если же игрок находится в P-позиции, то ему всегда приходится перейти в N-позицию, таким образом при оптимальной игре противника он проиграет.

В играх с небольшим графом концепцией P- и N-позиций можно ограничиться, но во многих случаях этого бывает недостаточно. Например, в Kayles уже при 60 кеглях получается более миллиона неизоморфных позиций. Тем не менее, в большинстве этих позиций присутствуют независимые компоненты: если позиция состоит из нескольких рядов кегель, то ни один ход в дальнейшем не может повлиять одновременно на какие-либо два из этих рядов. За счет таких независимых компонент и возможно производить анализ намного более продуктивно, но для этого необходимо наложить на игру два условия.

1. **Равноправность.** Вспомним приведенные ранее примеры комбинаторных игр. У Red-Blue Hackenbush было от остальных игр существенное отличие: множества ходов, доступные игрокам в конкретной позиции, различались (так как один может стирать только синие, а второй — только красные кривые). Игры, в которых хотя бы для одной позиции игрокам дозволены разные ходы, называются партизанскими. Если для всех позиций возможности игроков идентичны, игра называется равноправной. В дальнейшем мы будем рассматривать только равноправные игры.

2. **Нормальная игра.** В игре Dawson’s Kayles, в отличии от остальных, победителем объявляется не сделавший последний ход, а его противник. Игры с таким условием называются мизерными. В противном случае игра называется нормальной. Анализ мизерных игр на порядок сложнее нормальных и не позволяет применять те же приемы, поэтому мы остановимся на нормальных играх.

Ним. Теорема Бутона.

Рассмотрим игру Ним повнимательнее. Как и для любой комбинаторной игры, для позиции в Ним можно определить, выигрышная она или проигрышная, анализируя весь граф игры. Но с увеличением размера каждой кучки граф разрастается полиномиально, а с увеличением количества кучек — вообще экспоненциально. Решение, не требующее подобных вычислений, было открыто Чарльзом Бутоном ещё в 1901 году.

Обозначим позицию в Ним с N кучками, в i-ой из которых x_i камней, через $(x_1,x_2,\ldots,x_N)$. Ним-суммой двух чисел A и B назовем число $A \oplus B$, где $\oplus$ — операция побитового *исключающего или*. Теорема Бутона в этих обозначениях выглядит так:

Позиция $(x_1, x_2, \ldots, x_N)$ в Ним является P-позицией тогда и только тогда, когда ним-сумма всех x_i равна нулю, т.е. $x_1 \oplus x_2 \oplus \ldots \oplus x_N = 0$.

Чтобы доказать эту теорему, покажем, что все три условия вышеуказанного определения P- и N- позиций выполняются:

1. *Любая конечная позиция является P-позицией. Единственная конечная позиция в Ним - когда все кучки пустые, соответственно* $x_1 \oplus x_2 \oplus \ldots \oplus x_N = 0 \oplus 0 \oplus \ldots \oplus 0 = 0$.

2. *Неконечная позиция является P-позицией, если все позиции, в которые можно из неё перейти, являются N-позициями.* Допустим, мы имеем позицию $(x_1, x_2, \ldots, x_N)$ и $x_1 \oplus x_2 \oplus \ldots \oplus x_N = 0$. Без потери общности допустим, что мы уменьшаем первую кучку до размера $x^* < x_1$. Тогда $x^* \oplus x_2 \oplus \ldots \oplus x_N \neq 0$, в противном случае мы имеем $x^* \oplus x_2 \oplus \ldots \oplus x_N = 0 = x_1 \oplus x_2 \oplus \ldots \oplus x_N \Leftrightarrow x^* = x_1$.

3. *Неконечная позиция является N-позицией, если из неё можно перейти в хотя бы одну P-позицию*. Покажем, что такой ход всегда существует. Найдем в числе $S = x_1 \oplus x_2 \oplus \ldots \oplus x_N$ самую левую позицию с единицей. Изменим любое из чисел x_i, у которого в этой позиции единица, таким образом, чтобы все позиции в новой ним-сумме содержали нули. Это всегда возможно, так как для этого в таком числе должны измениться только биты правее выбранного, а выбранный становится вместо единицы нулем - соответственно, число уменьшается.

Теперь мы умеем быстро определять, является данная позиция в Ним P- или N-позицией. Покажем, что к Ним можно свести любую равноправную нормальную игру. Для этого введем несколько понятий и сформулируем теорему Шпрага-Гранди.

Сумма игр.

Здесь и далее под термином “игра” будет иногда пониматься конкретная

позиция в соответствующей игре, а иногда — игра в общем, то есть и правила, её определяющие, и конкретная позиция. Надеюсь, в каждом случае будет понятно, что именно имеется в виду.

Допустим, у нас есть две игры G_1 и G_2. Суммой игр $G_1 + G_2$ назовем игру со следующими правилами. На каждом ходу, игрок выбирает одну из игр и делает там ход, оставляя вторую игру нетронутой. Игра продолжается до тех пор, пока тот, кому предстоит сделать ход, не окажется без допустимых возможностей в обеих играх. Формально, если $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ — соответствующие играм графы, то их сумма $G = (V, E)$ определяется как:

$V = V_1 \times V_2$,

$E = \{(\nu_1\nu_2, \omega_1\nu_2) | (\nu_1, \omega_1) \in E_1\} \cup \{(\nu_1\nu_2, \nu_1\omega_2) | (\nu_2, \omega_2) \in E_2\}$.

Очевидно, что Kayles в случае нескольких рядов кегель можно представить как сумму игр, в каждой из которых всего один непрерывный ряд. В Ним каждую кучку можно рассматривать как отдельную игру. Заметьте, что игры, входящие в сумму, могут быть и разными: можно иметь $G_1 = \{$*ряд Kayles с 7-ю кеглями*$\}, G_2 = \{$*кучка Nim с 5-ю камнями*$\}$.

Теорема Шпрага-Гранди.

Для начала, введем некоторые определения и обозначения.

Обозначим через *o(G)* для позиции *G*, является она *P*- или *N*-позицией.

Назовем две игры *G* и *H* эквивалентными, если они ведут себя одинаково в любой сумме игр, то есть для любой игры *X* выполняется *o(G+X)=o(H+X)*. Записывается это как *G=H*.

Покажем теперь, что для любой игры *G* и *P*-позиции *A* выполняется *A+G=G*. Для этого удостоверимся, что *o(G+H)=o(A+G+H)*. Если *o(G+H)=P*, то у второго игрока есть выигрышная стратегия и в игре *G+H*, и в игре *A*, соответственно, он может закончить последним в обеих из них. Если же *o(G+H)=N*, то первый игрок может сделать выигрышный ход в *G+H*, после чего второй игрок окажется в аналогичной рассмотренной до этого ситуации.

Также покажем, что *o(G+G)=P* для любой игры *G*. Здесь все просто: на любой ход в одной копии игры противник может ответить точно таким же ходом во второй, что в итоге приводит к его победе.

Из доказанных утверждений следует, что если *o(G+H)=P* для некоторых игр *G* и *H*, то *G = G+(G+H) = (G+G)+H = H*, то есть игры *G* и *H* эквивалентны.

Обозначим игру из одной кучки Ним размера *x* через $^*x$. Также обозначим через *mex(S)* наименьшее целое неотрицательное число, которое не входит во множество *S*. Тогда теорема Шпрага-Гранди выглядит так:

Если из позиции G можно с помощью допустимого хода перейти в позиции $^*a_1, ^*a_2, \ldots, ^*a_N$*, то* $G = {}^*m$*, где* $m = mex(\{a_1, a_2, \ldots, a_N\})$.

Как видите, теорема гласит, что любая игра эквивалентна игре с одной кучкой Ним некоторого размера. Для доказательства достаточно показать, что $G + {}^*m$ является *P*-позицией. Рассмотрим два случая:

1. Первый игрок делает ход в игре *G*. Тогда у нас остается игра ${}^*a + {}^*m$, где a принадлежит $\{a_1, a_2, \ldots, a_N\}$. В свою очередь, m этому множеству не принадлежит, значит $m \neq a$. Если $a > m$, второй игрок может перейти в ${}^*m + {}^*m$, а если $a<m$, он может перейти в ${}^*a + {}^*a$. В обоих случаях, мы имеем *P*-позицию.

2. Первый игрок делает ход в ${}^*m$. Тогда мы имеем $G + {}^*a$ при $a<m$. Так как m — наименьшее неотрицательное число, не входящее в $\{a_1, a_2, \ldots, a_N\}$, одно из a_i равняется a и второй игрок может перейти в ${}^*a + {}^*a$, которое является *P*-позицией.

Часто вводят понятие функции Гранди. Значение функции Гранди *F(G)* для данной позиции *G* равняется такому m, что $G = {}^*m$.

Итоги.

Таким образом, мы показали, что любая равноправная нормальная игра эквивалентна кучке Ним некоторого размера и научились находить этот размер. Ранее была рассмотрена оптимальная стратегия для игры Ним. Следовательно, вместе эти два метода представляют мощный инструмент для анализа комбинаторных игр.

Напоследок рассмотрим один пример, чтобы наглядно показать применение пройденного материала.

Допустим, два игрока играют в такую игру. У них есть кучка из *A* камней, ряд из *B* кегель (в дальнейшем в игре может возникнуть несколько кучек и рядов) и ещё некоторая равноправная игра, которой соответствует некоторый граф. Игроки делают ходы по очереди. Ход представляет собой ровно одно из перечисленных:

1. Выбрать какую-либо из кучек с камнями и разделить её на две непустые кучки неравного размера.

2. Выбрать какой-либо из рядов кегель и сбить в нём ровно две соседние кегли.

3. Сделать допустимый ход в игре, заданной графом.

Игра заканчивается, когда больше нельзя сделать ни одного хода, и сыгравший последним объявляется победителем.

Посмотрим, как определить для конкретной позиции, является она *P*-позицией или *N*-позицией. Для этого сначала посчитаем значения функции Гранди для каждой из игр в отдельности.

Для первой игры при A камнях функция Гранди будет иметь вид:

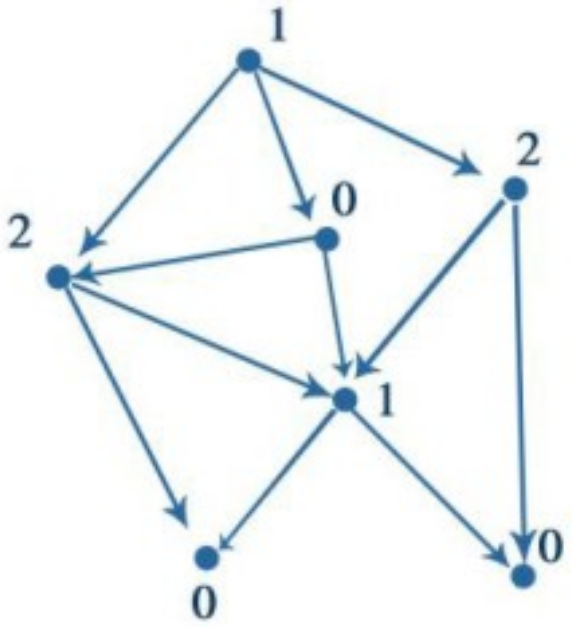
$$F(A) = \begin{cases} mex(F(1) \oplus F(A-1), F(2) \oplus F(A-2), \ldots, F(A/2-1) \oplus F(A/2+1)) \text{ для } A \text{ вида } 2k \\ mex(F(1) \oplus F(A-1), F(2) \oplus F(A-2), \ldots, F((A-1)/2) \oplus F((A+1)/2)) \text{ для } A \text{ вида } 2k+1. \end{cases}$$

Для второй игры при B кеглях функция Гранди будет иметь вид:

$$F(B) = \begin{cases} mex(F(B-2), F(1) \oplus F(B-3), \ldots, F((B-2)/2) \oplus F((B-2)/2)) \text{ для } B \text{ вида } 2k \\ mex(F(B-2), F(1) \oplus F(B-3), \ldots, F((B-3)/2) \oplus F((B-1)/2)) \text{ для } B \text{ вида } 2k+1. \end{cases}$$

Посчитать значения функции Гранди для обоих игр можно на основе данных выражений за время $O(A^2)$ и $O(B^2)$ соответственно. Что касается графа, значения для его вершин можно вычислить рекурсивно за время *O(M)*, где *M* - количество ребер в графе. Так как ребра графа соответствуют допустимым ходам, значение для каждой вершины будет наименьшим значением, которое не присутствует среди детей этой вершины.

Конкретно, если *A*=5, *B*=6 и граф *G* выглядит, как показано на рисунке ниже, мы получаем *F(A)*=2, *F(B)*=3 и *F(G)*=1. Так как $F(A) \oplus F(B) \oplus F(G) = 2 \oplus 3 \oplus 1 = 0$, это *P*-позиция.



Задачи и разборы

Задача A. Akhmed

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Ахмед – террорист-камикадзе, и завтра у него первое задание. Как известно, после выполнения задания на том свете Ахмеда будет ожидать некоторое количество девственниц. Их точное количество зависит от величины нанесенного им урона.

Ахмеду предстоит отправиться в зону с N объектами, о которых он располагает следующей информацией: i-ый из объектов имеет форму круга с радиусом R и центром в точке (X_i, Y_i), а за его уничтожение полагается W_i девственниц. В указанной зоне Ахмед может добраться до любой точки плоскости, после чего он должен взорвать бомбу. Все объекты, полностью попавшие в радиус её действия, будут уничтожены.

Ахмед желает, чтобы в раю его встретили по крайней мере K красавиц. Теперь его задачей является подобрать бомбу для выполнения своего задания. Контора лишними средствами не располагает, поэтому на складе ему выдадут бомбу с наименьшим радиусом действия, достаточным для исполнения его замысла. Найдите этот радиус или определите, что это невозможно.

Ограничения

$1 \le N \le 100$,
$1 \le K \le 111$,
$1 \le R \le 1\,000$,
$-1\,000 \le X_i, Y_i \le 1\,000$.

Формат входного файла

Первая строка входных данных содержит числа N, R и K. Далее следует N строк, каждая из которых содержит по три числа – X_i, Y_i и W_i.

Формат выходного файла

Выведите минимальный радиус действия бомбы, позволяющей Ахмеду воплотить в жизнь свои намерения. Если это невозможно, выведите -1. Ваш ответ не должен отличаться от судейского более чем на $1e-5$.

Пример

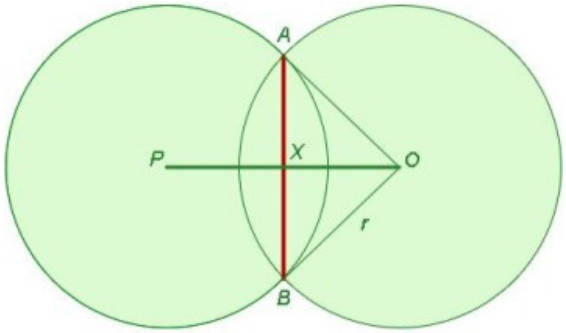
a.in	a.out
3 1 5 0 2 2 -2 0 2 0 -2 2	3.000000

Разбор задачи A. Akhmed

Для начала, избавимся от лишних подробностей и перефразируем задачу: дано N кругов равного радиуса различной ценности. Требуется определить минимальный радиус круга, которым можно полностью покрыть подмножество заданных кругов с суммарной ценностью не менее K.

Отдельно рассмотрим случаи, когда сумма ценностей всех объектов меньше K, соответственно ответ -1, или же существует объект с ценностью, больше или равной K, когда ответом является радиус объекта. Теперь перейдём к общему решению.

Первым делом заметим, что если некоторая точка X находится внутри круга с радиусом r, то концентрический круг радиуса $r + R$ будет содержать любую точку, отдаленную от X на расстояние не более R. Поэтому круги можно рассматривать как точки. Если при этом найти круг, содержащий множество точек с суммарной ценностью большей или равной K, то концентрический круг с радиусом, большим на R, будет одним из решений исходной задачи.



Один из способов решения поставленной задачи состоит в том, чтобы перебрать искомый радиус бинарным поиском. Это возможно, так как если существует круг радиуса $R1$, охватывающий необходимое множество точек, то, очевидно, существует круг с радиусом $R2$ большим $R1$ с таким же свойством. Отметим также, что если круг содержит некоторое множество точек, то его центр всегда можно сдвинуть так, чтобы он содержал ровно то же множество точек, но при этом проходил через хотя бы две из них. Используем этот факт для того, чтобы проверить, существует ли круг с заданным радиусом, охватывающий нужное нам множество точек: построим все возможные окружности, проходящие через каждую пару точек. Если искомый круг

существует, то его сдвигом мы обязательно получим одну из построенных фигур. Остаётся для каждого круга проверить, какие точки в него входят и посчитать их суммарную ценность.

Построить окружность заданного радиуса r, проходящую через данную пару точек A и B, можно следующим образом. Найдём расстояние d между A и B. Если $d > 2*r$, то окружность построить невозможно. Если $d = 2*r$, то мы получим одну окружность с центром в середине отрезка AB. В случае $d < 2 * r$, можно построить две окружности. Обозначим центр отрезка AB через X. Отрезок PO, соединяющий центры окружностей, должен быть перпендикулярен к AB и пересекать его в точке X. Следовательно, можно найти центр O одной из окружностей посредством параллельного переноса точки X по направлению вектора PO (который легко найти, как перпендикулярный вектору AB) на расстояние XO, которое из треугольника AXO равняется $\sqrt{AO^2 - AX^2} = \sqrt{r^2 - (\frac{d}{2})^2}$. Аналогично, центр P второй окружности можно найти с помощью параллельного переноса в обратном направлении на то же расстояние.

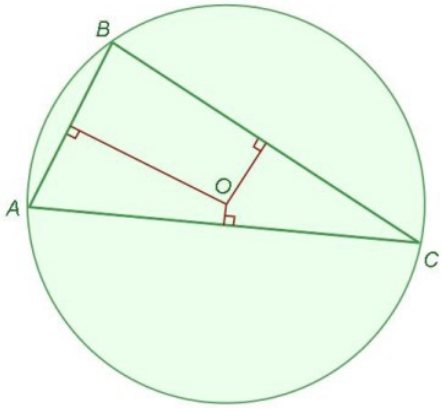
В целом сложность алгоритма составляет $O(\log(maxdist) * N^3)$, где $maxdist$ — максимальное расстояние между точками, $O(\log(maxdist))$ соответственно время, требуемое на перебор оптимального радиуса, $O(N^2)$ — количество окружностей, проходящих через все пары заданных точек, а последнее $O(N)$ — сложность проверки принадлежности точек к построенной окружности.

Есть также другой поход к этой задаче, основывающийся на наблюдении, что оптимальный круг обязательно удовлетворяет одному из следующих условий:

1. Круг проходит через две, диаметрально расположенные точки из заданного множества.

2. Круг проходит через какие-либо три точки из заданного множества.

Таким образом, нам требуется перебрать все пары точек и строить на них окружности как на концах диаметра, и перебрать все тройки точек и строить окружности, через них проходящие.



В первом случае центром окружности будет центр отрезка, соединяющего соответствующую пару точек, а её радиусом — половина расстояния между этими точками.

Рассмотрим второй случай. Назовём выбранные три точки A, B и C. Окружность, проходящая через них — это окружность, описан-

ная вокруг треугольника ABC. Как известно, центром такой окружности является точка пересечения серединных перпендикуляров. Радиусом же будет расстояние от центра до любой из точек A, B, C.

В сумме количество кругов $O(N^3)$, а после построения каждого требуется проверить каждую точку на принадлежность этому кругу, что даёт итоговую сложность $O(N^4)$.

Задача B. Shift

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Заданы N строк. Вашей задачей является стереть первые K_1 символов в первой строке, первые K_2 символов во второй и так далее, то есть первые K_i символов в каждой i-ой строке таким образом, чтобы полученные в результате строки имели как можно более длинный общий префикс. Если есть несколько возможных последовательностей K_i, найдите лексикографически минимальную.

Ограничения

$1 \le N \le 10$.

Суммарная длина всех строк не превосходит 100 000. Строки содержат только заглавные латинские буквы. У заданных строк будет хотя бы один общий символ.

Формат входного файла

Первая строка входного файла содержит количество строк N. Далее следует N заданных строк.

Формат выходного файла

В первой строке выведите максимальную длину общего префикса строк, получающихся из заданных посредством стирания их начальных символов. Далее выведите числа K_1, K_2, ..., K_N, каждое на отдельной строке:

Примеры

b.in	b.out
3 ABCACCBCACCBC CCCBCCBABACCBC BCCBCBCAACCB	4 3 9 8

Пояснение

Стертые символы выделены наклонным, а общий префикс – жирным шрифтом.

*ABC***ACCB***C*ACCBC
*CCCBCCBAB***ACCB***C*
*BCCBCBCA***ACCB**

Разбор задачи B. Shift

Очевидно, что под "общим префиксом после стираний" скрывается просто-напросто общая подстрока данных строк. Найдём её максимальную возможную длину. Заметим, что для любого фиксированного L, если строки имеют общую подстроку длины L, то у них найдутся и общие подстроки длины 1, 2, ... , $L-1$. Следовательно, если мы сможем достаточно быстро отвечать на запросы вида "имеют ли данные строки общую подстроку длины L?", максимальное удовлетворяющее этому условию L можно перебрать бинарным поиском.

Обрабатывать запросы для фиксированного L можно следующим образом. Для каждой строки, получим хеш-значения всех её подстрок длины L. Возьмём хеш-функцию вида:

$$H(S) = (A^{L-1} * S[0] + A^{L-2} * S[1] + \ldots + A^0 * S[L-1]) mod M,$$

где S — некая строка длины L, индексированная, начиная с 0, а A и M — натуральные числа. В таком случае хеш-значение для каждой следующей подстроки в данной строке можно вычислять за время $O(1)$ по формуле:

$$H(NEXT) = ((H(CUR) - (A^{L-1} * CUR[0]) mod M) * A + NEXT[L-1]) mod M,$$

где CUR — подстрока, для которой хеш уже посчитан, а $NEXT$ — подстрока той же длины, начинающаяся в строке на одну позицию правее, чем CUR.

Даже при оптимальном выборе параметров A и M могут происходить коллизии, то есть значения хеш-функции могут совпасть для разных строк. Значительно уменьшить вероятность этого явления можно, вычисляя для каждой подстроки значения двух разных хеш-функций.

Получив для каждой строки множество значений хеш-функции, нам остаётся проверить, сколько элементов содержит пересечение U этих множеств. Если оно непустое, у данных строк есть хотя бы одна общая подстрока соответствующей длины. Подобную проверку можно совершить с помощью сортировки или балансированных деревьев за время $O(LEN * \log(LEN))$, где LEN — суммарная длина всех строк.

Итак, мы можем найти максимальную длину $Lmax$ общей подстроки за время $O(\log(\frac{LEN}{N}) * LEN * \log(LEN))$, где первый множитель соответствует времени, требуемом для бинарного поиска. Столько операций потребуется в худшем случае, когда все строки равной длины $\frac{LEN}{N}$.

Найти лексикографически первый список позиций, с которых начинается общая подстрока длины $Lmax$, можно с помощью того самого пересечения множеств. Нам требуется найти в первой строке минимальную позицию, с которой начинается подстрока длины $Lmax$ с каким-либо хеш-значением из U. После этого в каждой из оставшихся строк надо найти минимальную позицию, с которой начинается подстрока длины $Lmax$ с тем же хешем. Всё это можно сделать за время $O(LEN * \log(\frac{LEN}{N}))$, если хранить U в виде упорядоченного массива или балансированного дерева: длина первой строки не превосходит LEN, а количество элементов в U не может быть больше длины самой короткой строки, которая в худшем случае равна $\frac{LEN}{N}$.

Итоговая сложность алгоритма составляет:

$$O(\log \tfrac{LEN}{N}) \ * \ LEN \ * \ \log(LEN)).$$

Задача C. Multi

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Разложением натурального числа N на множители называется его представление в виде произведения одного или более целых чисел, каждое из которых больше единицы. Два разложения считаются одинаковыми, если одно получается из другого перестановкой множителей. Для данного N, посчитайте количество его различных разложений на множители.

Ограничения

$2 \leq N \leq 10^{11}$.

Формат входного файла

Входной файл содержит число N.

Формат выходного файла

Выведите количество различных разложений N на множители.

Примеры

c.in	c.out
12	4

Пояснение

$12 = 12 = 2 \cdot 6 = 3 \cdot 4 = 2 \cdot 2 \cdot 3$.

Разбор задачи C. Multi

Забудем на первых порах о величине числа N и попытаемся считать ответ задачи с помощью динамического программирования. Обозначим количество разложений числа i на множители, наибольший среди которых равен j, через $WAYS(i, j)$. Если допустить $WAYS(1, 1) = 1$, будет выполняться следующее рекуррентное соотношение:

$$WAYS(i,j) = \begin{cases} WAYS(\frac{i}{j},1) + WAYS(\frac{i}{j},2) + \ldots + WAYS(\frac{i}{j},j), \text{если } j \text{ делит } i \\ 0, \text{если } j \text{ не делит } i. \end{cases}$$

Если нам удастся посчитать $WAYS(N, K)$ для всех $K = 1 \ldots N$, то их сумма и будет ответом.

Заметим, что у N при данных ограничениях в худшем случае будет чуть более 4000 делителей. Обозначим количество делителей через D. Очевидно, что нам интересны лишь те значения $WAYS(i, j)$, где i и j оба делят N. Соответственно, количество интересующих нас значений понизилось до $O(D^2)$. Но для вычисления каждого значения всё ещё требуется $O(D)$ операций. Чтобы избавиться от этого лишнего $O(D)$, немного изменим определение $WAYS(i, j)$ - пусть это будет количество разложений i на множители, наибольший из которых не превосходит j. Тогда вышеуказанное соотношение предстанет для интересующих нас значений i и j в следующем виде:

$$WAYS(i,j) = \begin{cases} WAYS(i, last\_j) + WAYS(\frac{i}{j},j), \text{если } j \text{ делит } i \\ WAYS(i, last\_j), \text{если } j \text{ не делит } i. \end{cases}$$

Таким образом, мы получили способ считать значения $WAYS(i, j)$ за время $O(1)$, а всего таких значений $O(D^2)$. Тем не менее, так как эти значения непоследовательны, для обращения к ним тяжело обойтись без сбалансированных деревьев или бинарного поиска, что повысит сложность этого фрагмента алгоритма минимум в $O(\log(D))$ раз.

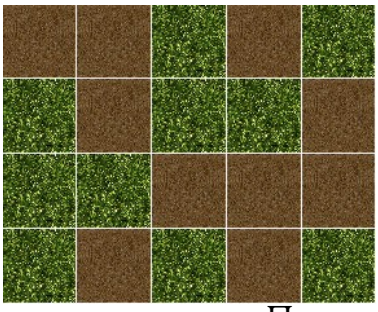
Избежать этого можно следующим образом. Последовательно пронумеруем все делители N в порядке возрастания. Пусть $NUM(i)$ будет i-ый

по возрастанию делитель. $WAYS(i,j)$ теперь будет соответствовать количеству разложений числа $NUM(i)$ с наибольшим множителем, не превосходящим $NUM(j)$. При вычислении $WAYS(i,j)$ для фиксированного i в порядке возрастания j неясно, какой индекс будет у числа $\frac{NUM(i)}{NUM(j)}$, но зато известно, что он будет обязательно меньше, чем индекс любого числа $\frac{NUM(i)}{NUM(k)}$ при $k < j$. Соответственно, запоминая индекс найденного числа, на следующей итерации j нам нужно проверять NUM-ы, начиная с этого индекса, что в сумме для каждого i даст $O(D)$.

Итоговая сложность алгоритма будет $O(\sqrt{N}+D^2)$, где $O(\sqrt{N})$ — время, нужное на поиск делителей N.

Задача D. Pasture

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб



У фермера Гиоргия есть пастбище прямоугольной формы. Параллельно границам пастбища проведены линии, делящие его на $N \times M$ равных участков. Каждый из участков или заполнен травой, или является пустырем (Гиоргий применяет к разным участкам разную стратегию ухода за почвой). Также у фермера Гиоргия есть K коров. Несмотря на то, что он любит их всех одинаково, они слишком ревнуют друг к другу и не могут пастись вместе. Поэтому Гиоргий решил выделить каждой корове какую-то часть пастбища и огородить её. Делить пастбище он хочет согласно следующим условиям:

1) Каждой из коров должен достаться хотя бы один участок.

2) Для каждой коровы, все доставшиеся ей участки должны быть достижимы друг из друга напрямую или посредством других участков этой коровы. Пара участков напрямую достижима друг из друга, если они имеют общую сторону.

3) Каждый участок пастбища должен достаться ровно одной корове.

4) Разница между максимальным количеством заполненных травой участков, доставшихся какой-либо корове, и минимальным должна быть минимальной возможной среди всех удовлетворяющих первым трем условиям делений пастбища.

Найдите деление пастбища, удовлетворяющее всем условиям.

Ограничения

$2 \leq N,\ M \leq 50$,
$2 \leq K \leq MIN(10, N \cdot M)$.

Формат входного файла

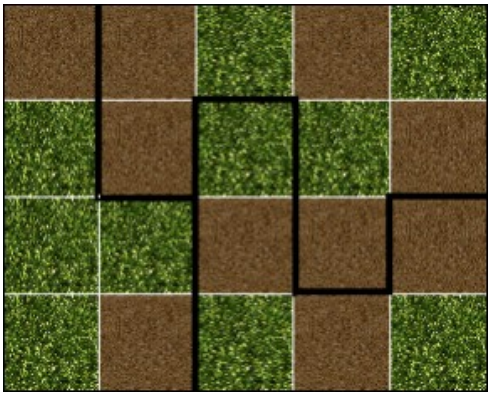
Первая строка входных данных содержит числа N, M и K. Далее следует N строк, каждая из которых содержит M символов – описание пастбища. Участки-пустыри помечены символом '.', а участки с травой – символом 'X'.

Формат выходного файла

Выведите N строк, по M цифр на каждой. j-ая цифра в i-ой строке обозначает номер коровы (начиная с нуля), которой достанется участок (i, j).

Примеры

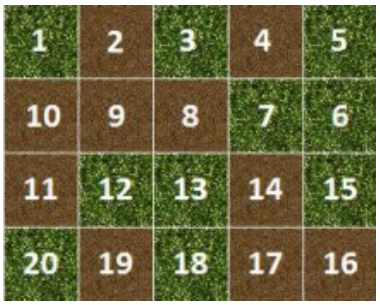
d.in	d.out
4 5 3 ..X.X X.XX. XX... X.X.X	02222 02122 00121 00111



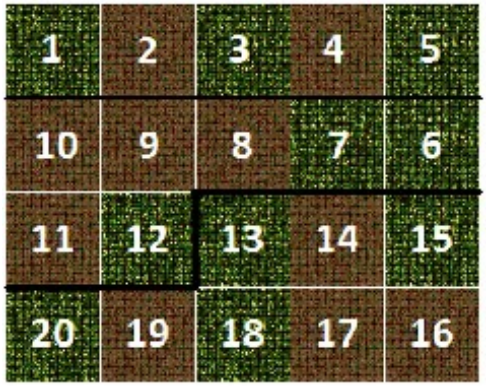
Разбор задачи D. Pasture

Определим, сколько участков с травой достанется каждой корове. Пусть всего плодородных участков P. Очевидно, что самый лучший вариант деления участков такой, чтобы $(P \mod K)$ коровам достался на один плодородный участок больше чем остальным, а именно $(\frac{P}{K} + 1)$ участок. Остальным $(K - P \mod K)$ коровам достанется $\frac{P}{K}$ участков. Покажем, что такое деление всегда достижимо с помощью следующего простого алгоритма.

Пройдем по полю в верхнем ряду слева направо и пронумеруем посещаемые участки, затем спустимся в следующий ряд и пройдем справа налево, снова спустимся, пойдем направо и так далее.



Теперь предположите, что все пронумерованные участки расположены в одну линию по возрастанию номера. Если поделить эту линию на K отрезков и отдать каждой корове по одному, то соответствующее деление на исходном поле будет удовлетворять первым трем требованиям условия. Осталось выполнить четвертое, для чего сгодится жадный алгоритм: правый конец первого отрезка двигать вправо до тех пор, пока он не будет содержать $\frac{P}{K}$ (или $(\frac{P}{K}+1)$) участков с травой. Следующий отрезок снова будем постепенно расширять вправо, пока не получим хотя бы $\frac{P}{K}$ плодотворных участков, и так далее. Здесь нужно быть осторожным со случаями, когда $\frac{P}{K}=0$.



Задача E. MST Game

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Графом называется упорядоченная пара $G := (V, E)$, где V - множество вершин, а E – множество пар вершин, называемых ребрами. Вершина j называется достижимой из вершины i, если существует ребро (i,j), или если из i достижима некоторая вершина k и существует ребро (k,j). Остовным деревом графа называется минимальное по включению множество ребёр, которые в совокупности обеспечивают, что любая пара вершин в графе достижима друг из друга. Над множеством E часто вводят функцию веса $W : E \to N$, то есть каждому ребру сопоставляют некоторое натуральное число, называемое его весом. Минимальным остовным деревом графа называют такое его остовное дерево, сумма весов ребер которого минимальна.

Андрей и Гиоргий только что выучили алгоритм Прима для построения минимального остовного дерева. Алгоритм этот очень простой. Сначала будущее дерево содержит одну произвольную вершину. Затем добавляется минимальное ребро, исходящее из этой вершины. Следующим добавляется ребро, один конец которого входит в дерево, а второй пока нет. Это шаг повторяется до тех пор, пока все вершины не войдут в дерево.

Поразмыслив, Гиоргий заметил, что если на каждом шаге брать ребро с максимальным весом вместо минимального, то в результате получится максимальное остовное дерево. Андрей же предложил сыграть в такую игру: каждый из них по очереди будет добавлять ребро к дереву, причем Андрей будет стремиться получить в результате дерево как можно меньшего веса, а Гиоргий, наоборот, как можно большего. Первой вершиной дерева пусть всегда будет вершина 1.

Для данного графа вычислите вес остовного дерева, которое получится при оптимальной игре обоих друзей.

Ограничения

Количество вершин в графе N будет между 2 и 20. Количество ребер M не менее $N - 1$ и не более $\frac{N \cdot (N-1)}{2}$. Граф связный и содержит не более одного ребра между каждой парой вершин. Веса ребер в диапазоне между 1 и 100. Вершины графа пронумерованы начиная с единицы.

Формат входного файла

Первая строка входного файла содержит два целых числа N и M и имя

начинающего игру – “ANDREW” или “GIORGI”. Далее следует M строк, описывающих ребра графа. Каждая содержит по три целых числа: номера вершин, которые соединяет это ребро, и его вес.

Формат выходного файла

Выведите единственное число – вес дерева, которое получится при оптимальной игре.

Примеры

e.in	e.out
4 5 ANDREW 1 2 6 2 3 4 2 4 8 4 3 2 1 4 11	19

Разбор задачи E. MST Game

На первый взгляд, может показаться, что Андрей всегда должен брать из всех доступных ребра наименьшего, а Гиоргий - наибольшего веса. На деле обоим иногда бывает выгоднее взять какое-то другое ребро, чтобы не дать противнику добраться до лакомых кусочков.

Допустим, что игра в процессе и остовное дерево уже содержит некоторое множество вершин S. Оптимальной стратегией для Андрея является выбор такого ребра (u, v) веса w, где u принадлежит S, что (w + [стоимость достроения остова от дерева $S \cup \{v\}$ при ходе Гиоргия]) минимально. Гиоргий, соответственно, пытается на каждом ходу максимизировать аналогичную величину. Поэтому, решение сводится к такой рекурсивной функции:

optimalCost (S, $move$) : integer;
если S содержит все N вершин, $optimalCost := 0$;
  для каждого ребра (u, v) веса w, где u принадлежит S, а v нет:
    если ходит Андрей,
      то $optimalCost := MIN(optimalCost, w + optimalCost(S \cup \{v\}, GIORGI)$;
    если ходит Гиоргий,
      то $optimalCost := MAX(optimalCost, w + optimalCost(S \cup \{v\}, ANDREW)$.

Заметим, что различных деревьев, содержащих вершину 1, в худшем случае 2^{N-1}, но функция будет обращаться к одному и тому же множеству S много раз. Поэтому результаты вычислений надо запоминать.

Таким образом, у нас получится алгоритм сложности $O(2^{N-1} * N^2)$. Эту оценку можно понизить до $O(2^{N-1} * N)$ за счёт следующего наблюдения. На самом деле, нас интересуют не все ребра, инцидентные S, а лишь ребра минимальной и максимальной стоимости, соединяющие S с каждой из оставшихся вершин. Допустим, мы располагаем этой информацией для некоторого множества S. Когда к S добавляется какая-либо вершина v, нужно рассмотреть соседние с ней ребра и в случае надобности обновить информацию для множества $S \cup \{v\}$. И это обновление, и выбор вершины для присоединения к дереву таким образом требует $O(N)$ операций.

Задача F. Dictionary

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Андрей и Гиоргий решили написать систему подсказок к коду, которая бы сильно упростила труд (ну или спровоцировала рецидив лени) программистов со всего мира.

Часть их грандиозного замысла уже выполнена: они написали программу, которая на основе словаря имен функций и переменных выдает для заданной строки все имена, содержащие эту строку как префикс.

Решив потестить свое творение и в то же время расслабиться, они придумали такую игру: первый игрок набирает на клавиатуре какую-либо букву, на которую начинается одно из слов словаря, затем второй набирает следующую букву из тех, которые ему предлагает система, следующую набирает снова первый игрок из оставшихся подсказок и так далее. Побеждает набравший такую букву, после которой система не предложит ни одного слова.

Вам дан словарь. Предположите невероятное – что система не содержит багов с самого начала, а также, что оба друга играют оптимально. Определите, кто выиграет игру, и если это первый игрок, выведите также лексикографически первую букву с которой ему можно для этого начать.

Ограничения

Суммарная длина всех слов в словаре не превышает 100 000. Слова содержат только заглавные латинские буквы.

Формат входного файла

Первая строка содержит целое число N. Следующие N строк содержат

слова, каждое на отдельной строке.

Формат выходного файла

Если выигрывает игрок, который ходит первым, выведите через пробел слово “First” и лексикографически первую букву, которая оставляет за ним победную инициативу. В противном случае выведите “Second”.

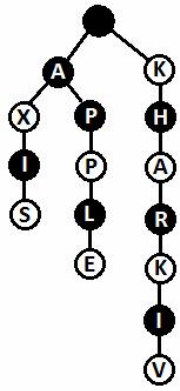
Примеры

f.in	f.out
4 A APPLE KHARKIV AXIS	First K

Пояснение

Если первый игрок напишет букву ‘A’, второй может ответить буквой ‘X’, что в конечном счёте приведёт к концу игры на слове “AXIS”. Начиная же с ‘K’, первый игрок в итоге победит (в данном случае все оставшиеся ходы будут форсированными).

Разбор задачи F. Dictionary



Рассмотрим дерево всех допустимых развитий игры с корнем в вершине, соответствующей пустой строке. Такое дерево известно как $trie$-дерево. Для решения нашей задачи дадим вершинам дополнительную нагрузку: каждая из них будет или N-вершиной, что означает, что игрок, которому предстоит написать следующую букву выигрывает при оптимальной игре, или P-вершиной, что означает, что он проигрывает. Очевидно, что все листья дерева являются проигрышными позициями, так как если игра дошла до листа, слово уже написано и более длинного слова, содержащего данное как префикс, не существует. Статус всех остальных вершин определяется следующим образом: если из вершины X достижима какая-либо P-вершина, то X является N-вершиной, так как из неё можно перевести игру в проигрышную для следующего игрока позицию. Если же все вершины, достигаемые из X, являются N-вершинами, то это, очевидно, P-вершина. Определить статус всех вершин можно с помощью одного поиска в глубину, запущенного из корня. Время работы поиска в глубину

будет $O(S)$, где S — количество вершин в построенном дереве и не будет превосходить суммарную длину L слов в словаре более, чем на 1 (за счёт корневой вершины). Построить такое дерево можно за $O(L)$, следовательно итоговая оценка сложности алгоритма — $O(L)$.

Задача G. Biotronic

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Наверное, многие из вас играли в Biotronic на Facebook-е. Игровое поле состоит из $N \times M$ разноцветных блоков. За один ход вы можете поменять местами два блока, имеющих общую сторону. Если в результате хода на поле появятся горизонтальные или вертикальные полосы из трех или более блоков одного цвета, то все блоки, принадлежащие какой-либо полосе, разом уничтожаются и вместо них в соответствующих клетках образуются пустыри. Если над пустырем находится закрашенный блок, то он падает на место этого пустыря. Когда все эти падения завершатся, на поле снова могут оказаться полосы из расположенных в ряд трех или более блоков, которые, в свою очередь, исчезнут, а на их место упадут закрашенные блоки сверху, если таковые имеются. Этот процесс будет продолжаться до тех пор, пока на поле будут оставаться непрерывные закрашенные полосы длиной хотя бы 3.

Поле игры задается как матрица размером $N \times M$, содержащая символы из множества “R”, “B”, “G”, “W”, “V”, “Y”, “O”. Символ “R” соответствует блоку красного цвета (red), “B” – голубого (blue), “G” – зелёного (green), “W” – белого (white), “V” – фиолетового (violet), “Y” – желтого (yellow), а “O” – оранжевого (orange).

Ваша задача — найти такой ход, в результате которого на поле в результате процесса уничтожений останется минимальное количество блоков.

Ограничения

$3 \leq N, M \leq 20$.

Каждый символ в матрице будет из множества “R”, “B”, “G”, “W”, “V”, “Y”, “O”. Изначально на поле нет трех или более блоков одного цвета, расположенных в ряд.

Формат входного файла

Первая строка файла содержит числа N и M. Далее следует N строк,

по M символов в каждой. Строки подаются от самой верхней к самой нижней.

Формат выходного файла

Выведите единственное число — минимальное возможное количество оставшихся блоков.

Примеры

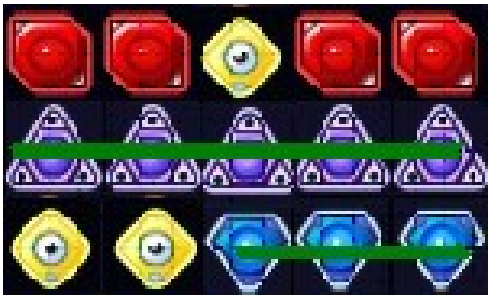
g.in	g.out
3 5 RRYRR VVBVV YYVBB	4

Пояснение

Изначально поле игры выглядит так:

Оптимальным ходом является поменять местами третие блоки во второй и третьей строках.

В результате этого хода во второй строке образуется полоса из пяти фиолетовых блоков, а в третьей – из трех синих. Обе полосы уничтожаются.



После того, как на место исчезнувших блоков упадут блоки сверху, мы получим следующую ситуацию, где полоса из трёх желтых блоков внизу, очевидно, собирается исчезнуть:

В итоге у нас остаётся четыре красных блока, но так как они разделены пустырем, ничего больше не происходит и процесс заканчивается.

Разбор задачи G. Biotronic

Уже ограничения в задаче настоятельно предлагают симуляцию. Для всех пар соседних клеток, поменяем их местами и произведем итерационную симуляцию процесса. На каждой итерации найдём все блоки, принадлежащие хотя бы одной полосе длиной 3. После идентификации всех таких блоков, сотрём их и спустим на их место закрашенные блоки сверху, если таковые имеются. Потом повторим то же самое заново. Когда на какой-либо итерации ни одного блока не сотрется, посчитаем количество оставшихся закрашенных блоков и, если оно меньше текущего минимума, обновим его.

Задача H. Yet Another Roads Problem

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Гиоргий бросил программирование и решил заняться инженерией. Ему сразу подвернулся чрезвычайно выгодный проект: построение системы дорог в X-ляндии.

В X-ляндии есть N городов. Задачей проекта является соединить некоторые пары городов дорогами таким образом, чтобы из любого города можно было попасть в любой другой напрямую или проездом через другие города. Существующие на данный момент дороги находятся в настолько плохом состоянии, что их в расчёт уже не берут.

Между некоторыми парами городов можно построить дорогу, но стоимости дорог значительно отличаются из-за расстояний, особенностей рельефа и других препятствий. Так как бюджет у проекта фиксированный, Гиоргий хочет потратить на постройку как можно меньше средств, чтобы остальное положить в карман.

От постройки части дорог Гиоргий может даже выиграть. Леса между городами заселены разбойниками, которых устраивает существование некоторых дорог, потому что потенциальных жертв станет больше. Таким образом, они готовы заплатить Гиоргию за каждую из выгодных им дорог. Более того, последний настолько хорошо торгуется, что если разбойники готовы заплатить ему за какую-либо дорогу, то они обязательно заплатят больше, чем ему придется потратить на постройку этой дороги.

Вам дана матрица размером $N \times N$, элемент (i,j) которой равен 0, если построить дорогу между городами i и j невозможно, некоторому положительному числу X – стоимости постройки этой дороги, если разбойники не собираются вознаграждать Гиоргия за её проведение, или некоторому отрицательному числу Y – разности стоимости постройки этой дороги и суммы вознаграждения за данное действие, если оно последует. Посчитайте, сколько есть различных способов построить сеть дорог в X-ляндии таким образом, чтобы выгода Гиоргия была максимальной. Так как это число может быть очень большим, выведите его остаток при делении на 1 000 000 007.

Ограничения

$5 \leq N \leq 100$.

Элементы заданной матрицы будут целыми числами в диапазоне [-100, 100]. Матрица будет симметричной, а все диагональные элементы будут равны 0.

Формат входного файла

Первая строка входного файла содержит число N. Каждая из следующих N строк содержит по N чисел – элементы матрицы.

Формат выходного файла

Выведите остаток при делении количества оптимальных для Гиоргия вариантов постройки дорог на 1 000 000 007.

Примеры

h.in	h.out
5 0 2 0 4 0 2 0 −3 2 −2 0 −3 0 2 −1 4 2 2 0 0 0 −2 −1 0 0	2

Пояснение

Очевидно, что построить дороги между парами городов 2 и 3, 2 и 5, и 3 и 5 выгодно для Гиоргия при любом раскладе. Чтобы минимизировать свои затраты, он должен также провести дорогу между городами 1 и 2. А соединить город 4 с остальной сетью можно или проведя дорогу (4,2), или дорогу (4,3). Таким образом, есть 2 оптимальных плана постройки.

Разбор задачи H. Yet Another Roads Problem

Очевидно, что Гиоргия устраивает построить все дороги за которые ему заплатят разбойники, поэтому первым делом добавим в граф все ребра с отрицательным весом. Если граф становится после этого связным, больше строить дорог не нужно и ответ задачи 1. В противном случае, рассмотрим новый граф, каждая вершина которого соответствует связной компоненте в предыдущем графе. Теперь перед нами стоит задача о нахождении количества остовных деревьев минимального веса.

При построении минимального остовного дерева нам всегда нужно брать максимальное количество ребер минимального веса, которые не создают циклов. Воспользуемся этим наблюдением: посчитаем, сколькими различными способами можно добавить в граф максимальное количество ребер минимального веса так, чтобы не возник цикл. Для этого сначала рассмотрим граф, в который все эти ребра добавлены. В общем случае, в результате в графе появится некоторое число L компонент связности. Независимо от множества ребер минимального веса, которые мы включим в остовное дерево, каждая компонента всегда будет содержать одни и те же вершины. Но построить эту компоненту иногда можно несколькими способами, то есть у неё может быть несколько остовных деревьев. Подсчитать их количество можно, применив матричную теорему Киргхофа к компоненте как к отдельному графу. Когда мы найдём количества остовных деревьев $X_1, \ldots, X_L$, будущий ответ надо будет умножить на их произведение. Теперь, если L=1, процесс заканчивается и мы имеем ответ. В противном

случае, снова рассмотрим новый граф, каждая вершина которого соответствует связной компоненте в предыдущем графе и повторим описанный приём уже для ребер следующего наименьшего веса. В конечном счёте, если у первоначального графа есть хотя бы одно остовное дерево, то на каком-либо шаге количество компонент связности станет равно 1 и процесс закончится.

Оценим сложность этого алгоритма. Веса ребер в диапазоне $[-100, 0) \cup (0, 100]$, и со всеми отрицательными мы справляемся одной проверкой. Для каждого положительного веса нам приходится рассмотреть все ребра этого веса, а далее для полученных компонент найти количество остовных деревьев. Самый дорогой шаг здесь именно последний, поэтому остановимся на нём. Теорема Киргхофа требует нахождения дополнительного минора для какого-либо элемента в матрице порядка количества вершин в графе. Это можно сделать с помощью метода исключения Гаусса за время $O(K^3)$, где K обозначает количество вершин. Более того, так как каждый раз, когда мы рассматриваем компоненту размером X, количество вершин в будущем графе уменьшается на $X - 1$, сложность всего алгоритма, вне зависимости от ограничения на вес ребер, будет $O(N^3)$.

Задача I. Soccer

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Футбольный матч – это поединок между двумя командами. За время матча, каждая из них забивает некоторое неотрицательное количество голов. Если обе команды забили равное количество голов, объявляется ничья, в противном случае команда, забившая большее количество голов, становится победителем.

Самый престижный турнир на клубном уровне в футболе – Лига Чемпионов. В первом раунде соревнования, команды разделены на группы, по 4 команды в каждой. Каждой из команд приходится провести по два матча с каждой другой командой в её группе. Если матч заканчивается победой одной из команд, эта команда получает 3 очка, а в случае ничьей обе команды награждаются 1-им очком.

Вам предоставлена статистическая информация о нескольких группах, а конкретно количество очков и количество забитых и пропущенных голов для каждой из команд. Для каждой группы вам предстоит определить,

могла ли такая ситуация возникнуть после завершения всех матчей в этой группе.

Ограничения

Количество групп будет между 1 и 10. Количество очков для каждой команды будет между 0 и 18. Количество забитых голов для каждой команды будет между 0 и 20. Количество пропущенных голов для каждой команды будет между 0 и 20.

Формат входного файла

Первая строка входных данных содержит число G – количество групп. Далее следует $4 \cdot G$ строк, по 3 числа в каждой. Первые четыре строки описывают результаты команд первой группы, следующие четыре – второй, и так далее. В каждой тройке чисел первым будет следовать количество забитых соответствующей командой голов, далее – количество пропущенных, а затем количество набранных командой очков.

Формат выходного файла

Для каждой группы входного файла, выведите одну строку с надписью “YES”, если информация об этой группе не противоречива, и “NO” – в противном случае.

Примеры

i.in	i.out
3 10 6 9 4 7 8 5 6 6 5 5 8 0 0 6 0 0 6 0 0 6 0 0 6 10 5 13 10 5 12 5 10 2 5 10 8	YES YES NO

Пояснение

Один из вариантов результатов матчей, приводящих к ситуации в первой группе, выглядит так:

Team1 – Team2 5:1
Team1 – Team3 2:2
Team1 – Team4 3:2
Team2 – Team1 0:0
Team2 – Team3 0:0
Team2 – Team4 0:1
Team3 – Team1 1:0
Team3 – Team2 1:2
Team3 – Team4 0:1
Team4 – Team1 0:0
Team4 – Team2 0:1
Team4 – Team3 1:1

Ситуация во второй группе возникает, если все матчи между командами закончились вничью.

Ситуация в третьей группе недостижима ни при каком комплексе результатов матчей.

Разбор задачи I. Soccer

Очевидно, что перебирать множества всех возможных результатов при данных ограничениях не представляется возможным. Тем не менее, в результате нехитрых наблюдений можно построить схему переборов, гарантированно быстро обрабатывающих каждую из групп.

Для каждой упорядоченной пары команд i и j, обозначим количество голов, забитое командой i в сумме двух матчей, через A_{ij}. Заметьте, что пара (A_{ij}, A_{ji}) хоть и не всегда отвечает однозначно на вопрос, какие результаты были зафиксированы в матчах между командами i и j, но некоторую информацию всё же содержит. Например, если (A_{ij}, A_{ji}) = (0,0), т.е. в сумме двух матчей обе команды остались на нуле, то оба матча однозначно закончились ничьей 0:0. А если (A_{ij}, A_{ji}) = (0,5), то у нас есть два варианта: или команда j выиграла оба матча, или закончила один вничью и выиграла в другом. Так как $0 \leq A_{ij} \leq 20$, всего различных таких пар мы можем иметь 21*21 штук. Тем не менее, их можно поделить на классы, каждый из которых будет иметь своё множество возможных результатов. Рассмотрим все такие классы (для простоты будем называть $A = A_{ij}$, $B = A_{ji}$):

1. A=0, B=0. Очевидно, что единственная пара результатов в соответствующей паре матчей — две ничьи.

2. A=1, B=0. Первая команда выиграла один из матчей, второй завершился вничью.

3. A=0, B=1. Вторая команда выиграла один из матчей, второй завершился вничью.
4. A>=2, B=0. Здесь мы имеем две возможности: или первая команда победила оба раза, или была одна ничья.
5. A=0, B>=2. Это аналогичный случай для второй команды.
6. $A = B$, A>0. Здесь также имеется две возможности: или оба матча закончились ничьей, или один матч выиграла первая команда, а второй - вторая (очевидно, с таким же счетом, хотя для решения это значения не имеет).
7. $A = B$-1, A>0. Две возможности: или победа и поражение первой команды, или ничья и поражение.
8. $A = B$+1, B>0. Аналогично для второй команды.
9. $A >= B$+2, B>0. Первая команда точно выиграла один матч, а второй мог закончиться с любым из трех результатов.
10. $B >= A$+2, A>0. Соответственно, вторая команда выиграла один матч, а второй закончился с любым из трех результатов.

Всего у нас шесть пар матчей. Голы в каждом из них будут соответствовать одному из перечисленных десяти классов. Попытаемся определить, какие шестерки $[c_1, c_2, c_3, c_4, c_5, c_6]$, где c_i — класс, в который попал результат i-ой пары команд, можно получить, чтобы при этом каждая команда набрала заданное для неё количество очков. Для этого сначала переберем все возможные результаты (то есть ничья или победа одной из команд) всех 12-ти матчей за время 3^{12} и выделим те, которые приводят к заданному распределению очков. В худшем случае таких множеств результатов будет около 800 (для распределения очков 8,8,8,8). Для каждого такого множества определим все шестерки классов, которые могли привести к этому множеству. Для этого попытаемся сопоставить каждому результату в каждой паре матчей между двумя командами каждый класс, которому он удовлетворяет. Сложность этого перебора в худшем случае 5^6, так как результат победа, поражение удовлетворяет 5-ти классам и всего есть 6 результатов. Таким образом, количество операций во всём этом фрагменте не превышает $(3^{12} + 800 * 5^6)$, что вполне разумно.

В результате предыдущего шага мы уже знаем, какие из шестерок классов удовлетворяют данному в группе распределению очков. Теперь переберем все возможные значения всех A_{ij} и определим, соответствует ли хотя

бы одному “хорошая” шестерка классов. Для A_{ij} у нас есть следующие восемь уравнений:

$A_{12} + A_{13} + A_{14} = GF_1$,
$A_{21} + A_{23} + A_{24} = GF_2$,
$A_{31} + A_{32} + A_{34} = GF_3$,
$A_{41} + A_{42} + A_{43} = GF_4$,
$A_{21} + A_{31} + A_{41} = GA_1$,
$A_{12} + A_{32} + A_{42} = GA_2$,
$A_{13} + A_{23} + A_{43} = GA_3$,
$A_{14} + A_{24} + A_{34} = GA_4$.

Если мы переберём какие-либо пять значений A_{ij}, все остальные зафиксируются — то есть нам нужно перебрать в худшем случае 21^5 вариантов. А имея все эти числа, мы можем определить соответствующий результату каждой пары команд класс и проверить, устраивает ли нас эта шестёрка.

Задача J. Rims

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Давайте рассмотрим вариацию небезызвестной игры “Rims”. Играют в нее вдвоем. В начале игры на плоскости имеется некоторое количество пронумерованных точек и, возможно, несколько непересекающихся петель (замкнутых кривых без самопересечений). Ход заключается в том, чтобы нарисовать некоторую петлю, которая будет проходить через не менее 1 и не более K данных точек и не будет касаться ни одной из имеющихся петель. Игроки делают ходы по очереди. Победителем является тот, кто сделал последний ход.

Вам задано начальное положение в игре в виде N различных точек и M петель, которые для простоты являются просто окружностями (заметьте, что в течение игры можно рисовать петли любой формы).

Очевидно, что на плоскости можно нарисовать бесконечное количество петель. Тем не менее, только их конечное число принципиально отличается друг от друга в этой игре. Будем называть два хода различными, если соответствующие им петли пересекают разные множества точек, или же внутри петель остаются разные множества точек.

Вашей задачей является посчитать, сколько есть у начинающего игру различных вариантов первого хода, которые обеспечивают ему победу в

игре при оптимальной стратегии. Так как таких ходов может быть очень много, выведите остаток при делении этого числа на 1 000 000 007.

Ограничения

$1 \leq N \leq 5000$,
$1 \leq K \leq 5000$,
$0 \leq M \leq 1000$.

Координаты всех точек будут в диапазоне $[-10^6, 10^6]$. Координаты центров всех окружностей, соответствующих начальным петлям, будут в диапазоне $[-10^6, 10^6]$. Радиусы окружностей, соответствующих начальным петлям, будут между 1 и 10^6. Заданные петли не будут пересекаться.

Формат входного файла

Первая строка входных данных содержит числа K, N и M. Далее следует N строк, i-ая из которых содержит пару чисел – координаты i-ой точки. Далее следует ещё M строк, по три числа на каждой - координаты и радиус окружности, соответствующей одной из петель.

Формат выходного файла

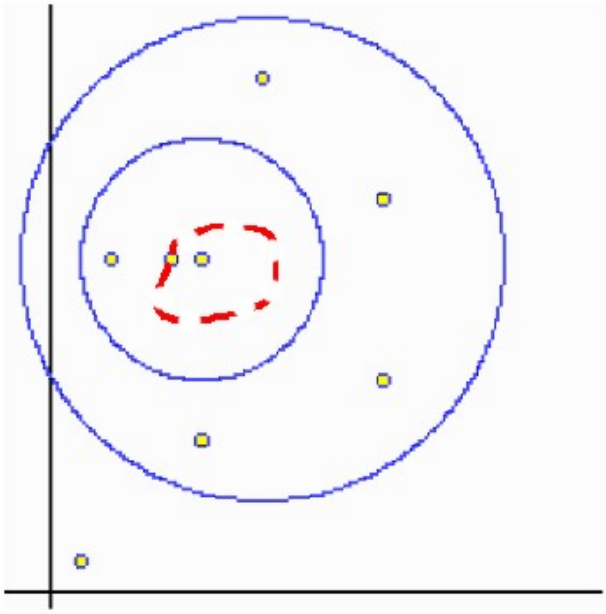
Выведите остаток от количества начальных выигрышных ходов первого игрока при делении на 1 000 000 007.

Примеры

j.in	j.out
2 8 2 5 5 1 1 2 11 5 11 3 11 11 7 11 13 7 17 7 11 8 5 11 4	18

Пояснение

Один из победных ходов выделен пунктиром.



Разбор задачи J. Rims

Нетрудно понять, что за геометрической постановкой задачи скрывается следующая игра. У пары игроков имеется несколько кучек с камнями, и они по очереди делают ходы. Ход заключается в том, чтобы взять из какой-либо одной кучки не более K камней, а оставшиеся разделить на две, возможно пустые, кучки. Если назвать размеры этих двух кучек A и B, то визуально этот ход соответствует проведению некоторой петли, внутри которой содержится A камней, а снаружи - B.

Получить из заданного геометрического представления размеры кучек можно, определив для каждой точки заданную окружность минимального радиуса, которая содержит эту точку. Этот шаг требует $O(N*M)$ времени.

Допустим, в начале игры мы имеем кучки размерами $N_1, \ldots, N_L$ и мы знаем значение функции Гранди F для каждого аргумента вплоть до максимального среди всех N_i. Обозначим $X = F(N_1) \oplus F(N_2) \oplus \ldots \oplus F(N_L)$. Теперь для каждой кучки попробуем сделать в ней все допустимые ходы. Нас интересуют те из них, которые переводят игру в проигрышную позицию, то есть если мы делаем в кучке N_i ход, делящий её на кучки размерами A и B, должно выполняться условие $X \oplus N_i \oplus A \oplus B = 0$. Каждому такому ходу в оригинальной задаче может соответствовать множество ходов, так как в видоизмененной игре ходы отличаются только размерами кучек, а в оригинальной - множествами точек. Конкретно, если мы забрали из кучки Q $(Q = N_i - (A + B))$ камней, то в первоначальной задаче мы могли провести петлю через $C(N_i, Q)$ различных множеств из Q точек.

В свою очередь, оставить ровно A из $(A+B)$ точек внутри петли можно $C(A+B,A)$ способами. Соответственно, такому ходу в новой задаче соответствует $C(N_i,Q)*C(A+B,A)$ ходов в старой.

Для вычисления биномиальных коэффициентов нам потребуется $O(N^2)$ времени, а проверить все возможные ходы из всех кучек можно за суммарное время $O(N*K)$, так как сумма размеров всех кучек равна N. Осталось научиться считать значения функции Гранди.

Напрямую подсчитывать значения F, пытаясь взять из кучки каждого размера по $1,2,\ldots,K$ камней и потом деля оставшееся на две кучки – слишком дорого, так как требует $O(N^2*K)$ времени. Чтобы увидеть более быстрый способ, давайте для начала представим, что $K \geq N$, то есть из кучки всегда можно взять любое количество камней. В таком случае, для любого $k>0$, все взятия $(i-k)$ камней из кучки размером $i \geq k$ равносильны друг другу, так как приводят к одинаковой ситуации: мы должны разделить оставшиеся k камней на две кучки. Поэтому можно попросту запоминать, какие значения функции Гранди были ранее достигнуты, а для каждого i рассматривать только взятие 1 камня из кучки размером i. В случае $K<N$, мы уже не можем этого сделать, потому что для кучек размером больше $(i+K)$ значения, достигнутые из кучек i и меньше, уже неактуальны. Но нам ничего не мешает запомнить для каждого возможного значения, от кучки какого размера можно было его достигнуть, и соответственно рассматривать только достаточно "новые" значения. Таким образом, вычислять значения функции Гранди также можно за $O(N^2)$ и суммарная оценка алгоритма получается $O(N*(N+M+K))$.

Задача K. Division

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

В X-ляндии есть N городов. Некоторые пары городов соединены дорогами. Дорог всего $N-1$ штук и проведены они таким образом, чтобы из каждого города можно было достигнуть любой другой город напрямую или посредством других городов.

Обитающие в лесах разбойники решили захватить власть в стране. Но для начала они хотят завладеть одним городом. Естественно, что между теми городами, путь между которыми пролегал через захваченный город, сообщение пропадёт. Таким образом, страна разделится на несколько частей, по несколько городов в каждой.

Цель разбойников – захватить такой город, который разобъёт X-ляндию на максимальное количество частей. Если таких городов несколько, они хотят выбрать тот, который минимизирует количество городов в самой большой части. Если всё ещё остаётся несколько возможностей, они хотят минимизировать количество городов во второй по размеру части, и так далее. Найдите оптимальный план захвата.

Ограничения

$2 \le N \le 100\,000$.

Формат входного файла

Первая строка входного файла содержит количество городов N. Далее следует $(N-1)$ строк, каждая из которых содержит по два числа – номера городов, которые соединяет соответствующая дорога.

Формат выходного файла

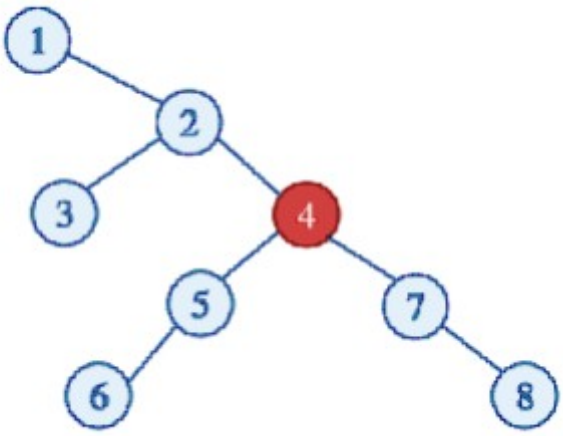
В первой строке выведите максимальное возможное количество частей K. Далее выведите K строк, по одному числу на каждой – размеры частей, упорядоченные от большего к меньшему.

Примеры

k.in	k.out
8 1 2 2 3 2 4 5 4 7 8 7 4 6 5	3 3 2 2

Пояснение

Захватив город №4, разбойники разобьют страну на 3 части: {1,2,3}, {5,6}, {7,8}. Также 3 части получается, если захватить город №2, но тогда одна из частей будет содержать 5 городов.



Разбор задачи K. Division

Переводя условие задачи на язык теории графов, мы получаем следующее: для данного дерева, найти вершину с максимальной степенью, для которой упорядоченный в порядке убывания список размеров компонентов, получающийся при её удалении из дерева, лексикографически минимален.

Для решения этой задачи воспользуемся видоизмененным поиском в глубину. Пускай вершина информирует своего непосредственного родителя о размере поддерева, корнем которого она является. Тогда для каждой вершины, обойдя всех её детей, мы можем составить список размеров компонентов, на которые она делит данный граф. Этими числами будут полученные от детей размеры их поддеревьев, плюс одно число - количество вершин, не вошедших в поддерево данной вершины, а соответственно оставшихся выше неё в нашем дереве. Это количество равно $N - k$, где k—размер поддерева с корнем в данной вершине. Единственный случай, когда это число равно 0 и его учитывать не нужно – для корневой вершины поиска в глубину. Получив список размеров компонентов, отсортируем его и обновим текущий ответ, если данная вершина его улучшает.

На первый взгляд, может показаться, что сложность этого алгоритма слишком велика: для каждой вершины, мы получаем соответствующий ей список компонентов (которых в худшем случае $N - 1$) и сортируем его. Тем не менее, так как каждый компонент в списке соответствует одному ребру, в сумме их получится $2 * (N - 1)$, а сложность алгоритма таким образом выражается как $O(N * \log N)$.

Задача L. Prime Distance

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Давайте определим расстояние от целого $N > 1$ до простого числа как функцию:

$$F(N) = \begin{cases} 0, \text{если } N \text{ – простое число,} \\ F(\text{сумма всех различных простых делителей } N) + 1, \text{если } N \text{ – составное.} \end{cases}$$

Найдите $F(N)$ для данного N.

Ограничения

$2 \le N \le 1\,000\,000\,000$.

Формат входного файла

Единственная строка входного файла содержит целое число N.

Формат выходного файла

Выведите число $F(N)$.

Примеры

l.in	l.out
28	2
100	1

Пояснение

Число $28 = 2 \cdot 2 \cdot 7$, следовательно оно составное и $F(28) = F(2+7)+1$. $2 + 7 = 9 = 3 \cdot 3$, следовательно оно также составное и $F(9) = F(3) + 1$. 3-е простое число, соответственно $F(3) = 0$, $F(9) = 1$, $F(28) = 2$.

$100 = 2 \cdot 2 \cdot 5 \cdot 5 => F(100) = F(2 + 5) + 1 = F(7) + 1 = 1$.

Разбор задачи L. Prime Distance

Начнем с того, что научимся находить простые делители данного N за время $O(\sqrt{N})$. Для этого пройдём по числам от 2 до $\sqrt{N}$ включительно и проверим для каждого, делит ли оно N. Если делит, то сохраним этот делитель и поделим N на него столько раз, сколько получится. Это деление

обеспечивает, что каждый найденный делитель будет простым. Если после обхода всех указанных чисел мы не нашли делителей, то N — простое число. В противном случае, N, очевидно, составное. Если после всех делений оно всё ещё больше 1, то оно равно последнему простому делителю и его тоже нужно учесть.

Решать задачу можно просто симуляцией процесса нахождения значений F, как описано в примерах. Чтобы понять, почему это работает достаточно быстро, рассмотрим поведение функции суммы различных простых делителей. В худшем случае, их сумма равна $(\frac{N}{2}+2)$, если N равно произведению двух и некоторого простого числа. Следовательно, для больших N можно сказать, что число уменьшается в два раза. Поэтому можно грубо оценить максимальное значение F как $\log N$. Верхняя оценка сложности всего алгоритма получается $O(\log N * \sqrt{N})$, что вполне достаточно при данных ограничениях.

Задача M. Octal Game

Имя входного файла:	`m.in`
Имя выходного файла:	`m.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Восьмеричные (octal) игры представляют собой обширный класс равноправных игр. Для них выработан специальный восьмеричный код, позволяющий компактно представить правила игры. В общем случае, в восьмеричную игру играют двое, делая ходы по очереди. У них имеется некоторое количество камней, сложенных в кучки. Ход представляет собой проведение одной из следующих трех операций, над какой-либо одной кучкой:

1. Забрать все камни из кучки, опустошая её.
2. Забрать часть камней из кучки, соответственно уменьшая её размер.
3. Забрать часть камней из кучки, а оставшиеся разделить на две непустые кучки.

Все остальные кучки остаются без изменений в течение хода. Игрок, который делает последний ход, побеждает. Восьмеричный код представляет собой восьмеричную дробь "$0.d_1d_2d_3d_4\ldots$", где цифра di обозначает, какие из трех вышеописанных операций игрок может выполнить над кучкой, забирая из неё i камней. Конкретно, di представляет собой сумму A, B и C, вычисляемых следующим образом:

- A равно 1, если можно выполнить операцию (1), то есть взять все камни из кучки величины i, и 0 в противном случае;
- B равно 2, если можно выполнить операцию (2), то есть уменьшить размер кучки из i+x камней до x, и 0 в противном случае;
- C равно 4, если можно выполнить операцию (3), то есть взять из кучки i камней, а оставшиеся камни разделить на две непустые кучки, и 0 в противном случае.

Вам даётся восьмеричный код, описывающий некую игру. Код может быть периодическим (см. пример №2). Далее даётся несколько ситуаций (количество кучек в игре и размер каждой из них), для каждой из которых вы должны определить, кто выигрывает при оптимальной игре с соответствующим восьмеричным кодом – начинающий игру или его противник.

Ограничения

Длина восьмеричного кода не будет превосходить 500. Количество ситуаций во входном файле не будет превышать 100. Количество кучек в каждой ситуации не будет превышать 100. Размер каждой кучки во входном файле будет между 1 и 500.

Формат входного файла

Первая строка входного файла содержит восьмеричный код игры. Непериодический код будет задан в виде “0.abcd...”, а периодический – “0.abcd...(efg...)”. Вторая строка содержит число T - количество ситуаций, которые вам предстоит обработать. Каждая из следующих T строк описывает одну ситуацию. Первое число в каждой строке соответствует количеству кучек, а далее следуют размеры кучек.

Формат выходного файла

Для каждой ситуации, выведите одну строку с надписью “First”, если при оптимальной игре побеждает начинающий игру, и “Second” в противном случае.

Примеры

m.in	m.out
0.77 3 3 5 10 8 3 7 6 4 6 1 2 3 4 5 6	First Second First

Value	Grundy Number
0	0
1	1
2	2
3	3
4	1
5	4
6	3
7	2
8	1
9	4
10	2

Пояснение

Код 0.77 означает, что на каждом ходе игрок может взять из кучки ровно 1 или 2 камня без других ограничений. Первые значения функции Гранди для этой игры выглядят так:

Примеры

m.in	m.out
0.03(12) 3 1 4 3 9 3 5 3 8 2 5	Second First Second

Пояснение

Раскрывая период, мы получаем следующий восьмеричный код: 0.03121212... Соответственно, брать ровно 1 камень из кучек запрещено; взяв 2 камня, нельзя разбивать оставшуюся кучку на две части, взять 3, 5, 7, ... камней можно только из кучки, содержащей ровно столько камней (то есть забрать все); а взять 4, 6, 8, ... камней можно только если в кучке ещё остаются камни, причём разбивать оставшиеся на две части не позволено.

Разбор задачи M. Octal Game

Первым шагом в решении этой задачи является подсчёт значений функции Гранди. Так как во всех предлагающихся ситуациях размеры кучек не

превосходят 500, значения функции можно считать вплоть до этого числа за кубическое время: для каждого i, попытаться взять из кучки величиной i все камни, если это позволено данным восьмеричным кодом, попытаться взять $j < i$ камней, и попытаться взять $j < i - 1$ камней, а оставшиеся камни разделить на две непустые кучки - если эти операции дозволены кодом для соответствующего j. Минимальное значение Гранди, которое не было достигнуто с помощью дозволенных операций для данного i, будет значением функции для i.

Просчитав функцию Гранди, остаётся ввести ситуации и для каждой из них определить, проигрышная она или выигрышная. Напомню, что ситуация проигрышная тогда и только тогда, когда выполняется $F(N_1) \oplus F(N_2) \oplus \ldots \oplus F(N_L) = 0$, где $N_1, \ldots, N_L$ - размеры кучек.

День пятый (23.02.2010г). Контест Ильи Порублёва и команды ЧНУ

Об авторе...

Порублёв Илья Николаевич

- выпускник кафедры ТК факультета кибернетики Киевского университета;
- старший преподаватель Черкасского университета;
- автор некоторых задач и член жюри школьных черкасских областных и городских олимпиад, олимпиады NetOI, Всеукраинских олимпиад по информатике 2003–2005 гг.;
- один из авторов книги “Алгоритмы и программы. Решение олимпиадных задач”.

Команда Черкасского университета:

- студент 3-го курса, призер Всеукраинской олимпиады по информатике 2006г. и NetOI-2006 Черненко Роман Вячеславович;
- студент 1-го курса, призер XV комплексной олимпиады “Турнир чемпионов”, неоднократный призер Всеукраинских олимпиад по физике, призер Всеукраинской олимпиады по информатике 2008г. Радченко Игорь Владимирович;
- студент 4-го курса Регент Константин Валериевич.

При разработке заданий дня использованы некоторые идеи, предложенные Полищуком Евгением Юрьевичем, учеником 11-го класса Черкасского физико-математического лицея, обладателем первых дипломов Всеукраинских олимпиад по информатике 2008 и 2009 гг., призёром комплексной олимпиады “Турнир чемпионов” в номинациях “математика”, “информатика” и “комплекс”.

Теоретический материал. Динамическое программирование и другие способы решения оптимизационных задач

Динамическое программирование (ДП) — общий метод решения широкого класса задач, главным образом оптимизационных. ДП задаёт общее направление алгоритма, но не подробности.

Рассмотрим очень простой пример — задачу «Игра» (олимпиада 12.11.2005 на `neerc.ifmo.ru/school/io`, текст сокращён).

Пример:

Герой прыгает по платформам, висящим в воздухе. Он должен перебраться от одного края экрана до другого. При прыжке с платформы на соседнюю, у героя уходит $|y_2 - y_1|$ энергии, где y_1 и y_2 — высóты этих платформ. Суперприём позволяет перескочить через платформу, но на это затрачивается $3 \cdot |y_3 - y_1|$ энергии.

Известны высóты платформ в порядке слева направо. Найдите минимальное количество энергии, достаточное, чтобы добраться с 1-й платформы до n-й (последней).

Для решения с помощью ДП введём серию подзадач «Сколько энергии $E(i)$ необходимо, чтобы добраться с 1-й платформы до i-й?» ($1 \leqslant i \leqslant n$). Искомую энергию для каждой следующей платформы можно получать, опираясь на известные энергии для предыдущих:

$$E(i) = \underbrace{\min}_{\text{лучший из 2-х способов}}\{\overbrace{E(i-1)}^{\text{достичь } (i-1)\text{-й платформы}} + \overbrace{|y_i - y_{i-1}|}^{\text{и перепрыгнуть с неё на } i\text{-ю}}, \quad \underbrace{E(i-2)}_{\text{достичь } (i-2)\text{-й платформы}} + \underbrace{3 \cdot |y_i - y_{i-2}|}_{\text{и перепрыгнуть с неё на } i\text{-ю}}\}, \qquad (1)$$

Тривиальные подзадачи имеют решения $E(1) = 0$ (двигаться не надо) и $E(2) = |y_2 - y_1|$ (возможен только прыжок с 1-й).

Когда нужно найти не только минимальное суммарное количество энергии, а и оптимальный маршрут (последовательность платформ), используют *обратный ход*. Исходя из $E(n)$, $E(n-1)$ и $E(n-2)$, можно определить, с какой именно платформы (последней или предпоследней) произошёл последний прыжок оптимального маршрута. Потом аналогично узнаём, откуда произошёл предпоследний прыжок, и так до самого начала.

Чтобы упростить обратный ход, можно при решении нетривиальных подзадач запоминать не только оптимальный ответ, а и выбор, при котором он был достигнут. Иногда, чтобы не “разворачивать” результат обратного хода, реализуют само ДП в обратном направлении (тривиальные подзадачи — как наиболее экономно допрыгать до конца с n-й и $(n-1)$-й платформ, целевая задача — как допрыгать до конца с 1-й платформы).

Для анализа применимости и целесообразности динамического программирования следует проверять такие условия:

1. В задаче можно выделить *однотипные подзадачи* разных размеров.
2. Среди выделенных подзадач есть *тривиальные*, имеющие малый размер и очевидное решение.
3. Оптимальное решение подзадачи большего размера можно построить *из оптимальных решений* меньших подзадач (часто говорят “у оптимально решенной задачи все подзадачи решены оптимально”).
4. Одни и те же нетривиальные меньшие подзадачи используются при решении различных бóльших подзадач (*подзадачи перекрываются*).
5. Количество различных подзадач разумно, и для запоминания результатов нужно *не слишком много памяти*.

Изначальная задача не обязана быть одной из подзадач серии — достаточно, чтобы её можно было решить, опираясь на решения одной или нескольких подзадач. Для некоторых задач можно выделить сразу несколько разных серий, и нужно грамотно выбрать правильный и, по возможности, эффективный способ.

Проверка п. 3 — главный этап в анализе примени́мости ДП к задаче, и зачастую требует особого внимания. Даже в задаче “Игра” аргументация правильности уравнения (1) вообще-то требует, кроме приведённых пояснений его частей, ещё трёх фактов. Будет ли герой прыгать с некоторой платформы дальше или там и закончит маршрут не влияет на то, какой маршрут, достигающий этой платформы, требует больше энергии, а какой меньше. Когда герой находится на некоторой промежуточной платформе, варианты его очередного прыжка не зависят от того, как он попал на эту платформу. Ещё один важный фактор(о котором специально не было упомянуто на лекции перед туром, дабы сохранить интригу задач I и K) — герою гарантированно невыгодно прыгать обратно. Повторно посещать одну и ту же платформу невыгодно, т. к. фрагмент пути между посещениями (требующий неотрицательных затрат энергии) всегда можно “вырезать”. Единственный (при прыжках длиной не более 2) фрагмент пути, включающий возвраты, но не включающий повторов — $(i)\to(i{+}2)\to(i{+}1)\to(i{+}3)$, но он также невыгоден, т. к. с учётом общеизвестных неравенств $|x|\geqslant 0$, $|a-c|\leqslant|a-b|+|b-c|$ и равенства $|a-b|=|b-a|$ получаем, что

$$\begin{gathered}\underbrace{3}_{=2+1}\cdot|y_{i-2}-y_i|+|y_{i-1}-y_{i-2}|+3\cdot|y_{i-3}-y_{i-1}|=\\ =2\cdot\underbrace{|y_{i-2}-y_i|}_{\geqslant 0}+\underbrace{|y_i-y_{i-2}|+|y_{i-2}-y_{i-1}|}_{\geqslant|y_i-y_{i-1}|}+3\cdot|y_{i-3}-y_{i-1}|\geqslant\\ \geqslant|y_i-y_{i-1}|+3\cdot|y_{i-3}-y_{i-1}|,\end{gathered}$$

т. е. вариант $(i) \to (i{+}1) \to (i{+}3)$ будет гарантированно не хуже. И решения, полученные с помощью (1), правильны *благодаря* тому, что задача имеет все описанные свойства.

Если подзадачи не перекрываются, бессмысленно запоминать промежуточные результаты, и следует либо считать рекурсивно, либо искать алгоритм, разбивающий на совершенно другие подзадачи или не разбивающий вообще. В любом случае это не ДП.

Аналогично с п. 5: если подзадач настолько много, что на запоминание их результатов не хватает памяти — значит, ДП с такой серией подзадач не даст эффективного алгоритма.

В "Игре" min выбирался из 2-х вариантов. Обычно их больше.

> В абзаце есть блоки разной высоты (например, обычные слова и математические системы). Абзац длинный, и его нужно разбить на стро́ки. Высота строки́ определяется по наивысшему из блоков в ней. Высота абзаца равна сумме высот строк. Длина строки́ равна суммарной ширине включённых в неё блоков (пробелы не учитываем). Разбивать блок для переноса со строки́ на строку нельзя. Изменять порядок следования блоков нельзя. Нужно найти такое разбиение абзаца на стро́ки, чтобы высота абзаца была минимальной. Максимальная допустимая длина строк TW и ширина и высота каждого блока $(w(i), h(i))$ задаются во входных данных.

Задача разбиения на стро́ки на практике обычно решается жадно ("размещать в строку всё, что помещается, а остальное переносить дальше"). Это просто и быстро, но не всегда даёт наилучший результат — и в данной задаче, и в некоторых других её вариантах.[3]

Поставим серию подзадач "Как оптимально разбить на стро́ки часть абзаца с 1-го по i-й блок?", обозначив оптимальную высоту $T(i)$.

Тривиальными будут несколько первых подзадач: если суммарная ширина блоков с 1-го по i-й позволяет разместить их в одну строку, то нужно так и сделать. Ведь если все блоки помещаются в одну строку, то высота абзаца равна высоте наивысшего блока. При любом ином разбиении на стро́ки этот блок задаст высоту одной из строк, и к ней надо будет прибавить высо́ты остальных строк. Стало быть,

$$\text{при } \sum_{k=1}^{i} w_k \leqslant TW, \qquad T(i) = \max_{1 \leqslant k \leqslant i} h_k. \tag{2}$$

[3] TEX, с помощью которого свёрстан этот сборник, решает её как раз через ДП, благодаря чему редко встречаются стро́ки с очень растянутыми пробелами

При размещении i блоков в более чем одной строке, некий k-й блок оказывается последним в предпоследней строке; очевидно, должны выполняться условия $k < i$ и $w_{k+1} + \ldots + w_{i-1} + w_i \leqslant TW$ (блоки с $k{+}1$-го по i-й помещаются в одну строку). Высота само́й последней строки́ равна $\max\limits_{k<p\leqslant i} h_p$, независимо от способа разбиения на стро́ки предыдущих блоков. Значит, чем лучше решена k-подзадача, тем лучше будет и решение i-подзадачи. Т. е., принцип оптимальности выполняется и уравнение ДП имеет вид:

$$T_i = \min_k \left(T_k + \max_{k<p\leqslant i} h_p \right), \text{ при условиях } k < i \text{ и } \sum_{p=k+1}^{i} w_p \leqslant TW. \qquad (3)$$

Как видим, минимум берётся не из нескольких (заранее известного количества) величин, а переменная k перебирает много значений, и из всех возможных сумм $T_k + \max\limits_{k<p\leqslant i} h_p$ выбирается минимальная.

Математическая запись (2) и (3) некрасива, но реализация через `while` проста. Скажем, для (2) она выглядит примерно так:

```
max_h:=0;
i=1; sum_w:=w[1];
while (i<=N)and(sum_w<=TW) do begin
  if h[i]>max_h then
    max_h:=h[i];
  T[i]:=max_h;
  i:=i+1;
  sum_w:=sum_w+w[i];
end;
```

“Лобовая” реализация (2), использующая цикл по `k`, возможна, но не нужна́. Для реализации (3) нужен цикл `for` для перебора `i`, внутри него аналогичный цикл `while`, перебирающий `k` начиная с `i-1` в сторону уменьшения, но никакого цикла по `p`.

В рассмотренных задачах ДП было однопараметрическим, но это далеко не всегда так. В задаче о разбиении абзаца можно было сделать ДП от двух параметров (количество расставленных блоков и количество задействованных строк), но там это было невыгодно (менее эффективно и при этом сложнее для понимания). Однако есть очень много задач, к которым ДП может быть применено только при условии многопараметричности.

Дано арифметическое выражение, состоящее из одноцифровых (от 0 до 9) чисел, между которыми поставлены знаки операций “`+`”, “`-`” и “`*`”.

Нужно вставить в заданное выражение круглые скобки так, чтобы получить правильное арифметическое выражение, имеющее наибольшее возможное значение.

Занумеруем чи́сла, начиная с 0, знаки операций — с 1, так что i-я операция будет между $(i{-}1)$-м и i-м числами: $d_0 \operatorname{op}_1 d_1 \operatorname{op}_2 \ldots \operatorname{op}_n d_n$.

Чтобы решить целевую задачу, найдём: оптимальное решение для расстановки скобок $d_0 \operatorname{op}_1 (d_1 \operatorname{op}_2 d_2 \operatorname{op}_3 \ldots \operatorname{op}_n d_n)$, т. е. последней выполняется op_1, а в каком порядке выполняются операции со 2-й по n-ю, надо ещё решить (за счёт чего и выбрать оптимум); оптимальное решение для расстановки $(d_0 \operatorname{op}_1 d_1) \operatorname{op}_2 (d_2 \operatorname{op}_3 \ldots \operatorname{op}_n d_n)$, т. е. последней выполняется op_2; и т. д., до $(d_0 \operatorname{op}_1 d_1 \ldots \operatorname{op}_{n-1} d_{n-1}) \operatorname{op}_n d_n$. Из всех этих n вариантов выберем лучший.

Каждое из подвыражений в скобках либо уже не нуждается в дальнейшей расстановке скобок (например, внутри "$d_0 \operatorname{op}_1 d_1$"), либо может быть решено путём такого же разбиения на подзадачи. Т. е., разные однотипные подзадачи отличаются друг от друга началом и концом, и речь разумно вести об (i, j)-подзадачах. Для всего выражения в целом $i = 0$, $j = N$.

Сказано "оптимальный" и "лучший", а не "максимальный", потому что знание только максимальных решений меньших подзадач не всегда даёт возможность получить максимум бóльшей подзадачи. Для сложения и впрямь достаточно максимизировать каждое из слагаемых, но для вычитания это не так: уменьшаемое должно быть как можно бóльшим, а вычитаемое — наоборот, как можно меньшим.

Вспомним, что изначальная задача не обязана быть одной из задач серии — достаточно, чтоб её можно было решить, опираясь на решения одной или нескольких подзадач. Это приводит к идее поставить серию подзадач как "Найти максимальное возможное $EMax(i, j)$ и минимальное возможное $EMin(i, j)$ значения при всех расстановках скобок внутри $d_i \operatorname{op}_{i+1} \ldots \operatorname{op}_j d_j$". Оптимумы для сложения и вычитания имеют вид:

$$\begin{aligned}
\max\{T(i..k{-}1) + T(k..j)\} &= EMax(i, k{-}1) + EMax(k, j),\\
\min\{T(i..k{-}1) + T(k..j)\} &= EMin(i, k{-}1) + EMin(k, j),\\
\max\{T(i..k{-}1) - T(k..j)\} &= EMax(i, k{-}1) - EMin(k, j),\\
\min\{T(i..k{-}1) - T(k..j)\} &= EMin(i, k{-}1) - EMax(k, j),
\end{aligned} \tag{4}$$

где $T(i..j)$ — подвыражение $d_i \operatorname{op}_{i+1} \ldots \operatorname{op}_j d_j$, в котором ещё не выбрана расстановка скобок, и opt (min или max) берётся по всем расстановкам.

Для умножения сложнее, т. к. из-за вычитаний значения подвыражений могут быть отрицательными. Но, оказывается, и минимум, и максимум $T(i..k{-}1) \times T(k..j)$ содержатся среди четырёх произведений $EMax(i, k{-}1) \times EMax(k, j)$, $EMax(i, k{-}1) \times EMin(k, j)$, $EMin(i, k{-}1) \times EMax(k, j)$ и $EMin(i, k{-}1) \times EMin(k, j)$.

Докажем для максимума (доказательство для минимума аналогично). Рассмотрим произвольную пару $\langle e(i..k{-}1), e(k..j)\rangle$ (где $e(i'..j')$ — значение $T(i'..j')$ при какой-то расстановке скобок). Может быть или $e(k..j) \geqslant 0$, или $e(k..j) < 0$. Если $e(k..j) \geqslant 0$, то

$$\underbrace{(EMax(i, k{-}1) - e(i..k{-}1))}_{\geqslant 0} \cdot \underbrace{e(k..j)}_{\geqslant 0} \geqslant 0,$$

т. е. $EMax(i, k{-}1) \cdot e(k..j) \geqslant e(i..k{-}1) \cdot e(k..j)$. Если $e(k..j) < 0$, то

$$\underbrace{(EMin(i, k{-}1) - e(i..k{-}1))}_{\leqslant 0} \cdot \underbrace{e(k..j)}_{< 0} \geqslant 0,$$

т. е. $EMin(i, k{-}1) \cdot e(k..j) \geqslant e(i..k{-}1) \cdot e(k..j)$.

В обоих случаях удаётся построить произведение, не меньшее $e(i..k{-}1) \times e(k..j)$, такое, что его первым множителем становится или $EMax(i, k{-}1)$, или $EMin(i, k{-}1)$. Аналогично, дальнейший переход к произведению, где вторым множителем является (в зависимости от знака первого множителя) $EMax(k, j)$ или $EMin(k, j)$, также не уменьшит значения произведения.

Доказанный факт необходим для обоснования алгоритма решения данной задачи, но доказательство важно ещё и как пример подхода “Докажем, что никакой способ не может быть лучше сразу всех перечисленных, и это даст нам право не перебирать другие способы”. Эта идея применяется очень широко (не только в ДП).

Окончательно, уравнение ДП для данной задачи имеет вид:

$$\begin{aligned} EMax(i, j) &= \max_{k\,:\,i<k\leqslant j} \{\max\{T(i..k-1)\,\mathrm{op}_k\,T(k..j)\}\}, \\ EMin(i, j) &= \min_{k\,:\,i<k\leqslant j} \{\min\{T(i..k-1)\,\mathrm{op}_k\,T(k..j)\}\}, \end{aligned} \tag{5}$$

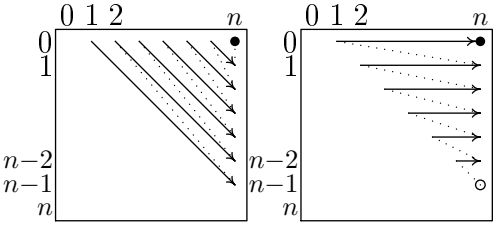
где выражения $\mathrm{opt}\{T_{i..k-1}\,\mathrm{op}_k\,T_{k..j}\}$ следует разворачивать или по формуле (4), или согласно рассуждениям следующего после неё абзаца, в зависимости от знака операции op_k (“+”, “−” или “*”).

А что будет, если добавить в задачу действие деления? Могут появиться недопустимые (из-за деления на 0) расстановки скобок... Но это мелочь. Кардинальное изменение задачи кроется в другом.

Рассмотрим $\max\{T(i..k{-}1) \,/\, T(k..j)\}$. Попытаемся, аналогично произведению, построить набор выражений, среди которых гарантированно будет максимум частного. Среди них несомненно должен быть случай, когда делимое — как можно бóльшее неотрицательное, а делитель — по возможности меньший строго положительный.

Но решительно непонятно, *как* искать такую оценку для делителя. Попытка дополнительно расширить серию подзадач, введя кроме $EMax(i,j)$ и $EMin(i,j)$ также минимальное положительное $EMinPos(i,j)$ и максимальное отрицательное $EMaxNeg(i,j)$, не приводит к успеху. Пусть в (i,j)-подзадаче операция op_k, $i<k\leqslant j$, является вычитанием. Чтобы построить $EMinPos(i,j)$ и $EMaxNeg(i,j)$, нужно рассмотреть в т. ч. и ситуацию, когда скобки расставлены $(T(i..k{-}1))-(T(k..j))$. Разность будет ближайшей к нулю, но не нулем, когда уменьшаемое и вычитаемое как можно ближе друг к другу, но не равны между собой. Для такого критерия *нельзя* провести оптимизации $(i,k{-}1)$-подзадачи и (k,j)-подзадачи *независимо* друг от друга. Значит — нарушается принцип оптимальности динамического программирования, т. е. эту задачу невозможно решить с помощью модификаций вышеизложенного алгоритма ДП.

При реализации вышеизложенного алгоритма (для “+”, “−” и “*”) может возникнуть вопрос: в каком порядке решать подзадачи, чтобы избежать ситуаций, когда для решения бóльшей подзадачи нужно решение меньшей, а эта меньшая ещё не решена. Представим графически, как ответы подзадач размещаются в двумерном массиве. Тривиальные подзадачи займут главную диагональ[4]. Подзадача, охватывающая весь диапазон (в данном случае $(0,n)$), из которой и будет получен ответ изначальной задачи, окажется “угловой” правой-верхней.



Классический порядок — по линиям, параллельным главной диагонали (см. левый рис.). Сначала решают все $(i,i{+}1)$-подзадачи (длины 2), потом все $(i,i{+}2)$ (длины 3), и т. д. Некоторые алгоритмы (для некоторых других задач) требуют именно такого порядка.[5] Но когда это свойство не требуется, может оказаться удобнее порядок, изображённый на правом рис. (стрóки снизу вверх, внутри строк слева направо) или симметричный ему (столбики слева направо, внутри столбиков снизу вверх).

Для многих алгоритмов ДП возможно завести глобальную таблицу для хранения результатов и реализовать рекурсивную функцию, которая при вызове будет сначала смотреть по этой таблице, решали ли эту подза-

[4]возможно, ещё соседнюю с ней линию элементов вида $(i,i{+}1)$; в данной задаче спорный вопрос, удобнее ли считать $(i,i{+}1)$-подзадачи тривиальными или решать по общей формуле (5), выполняя в цикле перебора k лишь одну итерацию

[5]например, в задаче Wiercenia (Drilling) конкурса Potyczki Algorytmiczne 2009 (см. `www.konkurs.adb.pl`) для очевидной реализации со сложностью $\theta(n^3)$ порядок решения подзадач безразличен, но если их решать в порядке увеличения размеров, можно уменьшить время работы до $O(n^2\log n)$ или $\theta(n^2)$

дачу раньше, и если решали, то брать готовый ответ, а если не решали — действовать как обычная рекурсия , после чего запоминать результат в той же таблице. Такой подход называется *рекурсия с запоминаниями* (англ. *memorized recursion*).

Рекурсия с запоминаниями обычно работает медленнее, чем аналогичный алгоритм, заполняющий таблицу итерационно. Но эта разница обычно невелика́, а иногда рекурсия с запоминаниями быстрее. Например, если особенности алгоритма позволяют, проанализировав результаты части рекурсивных вызовов, увидеть, что остальные вызовы уже́ не нужны́, и таким образом решать не все подзадачи.

Говоря о рекурсии с запоминаниями, следует упомянуть вариант её реализации, пока что почти не рассмотренный в официальной печатной литературе, но широко используемый на соревнованиях.

На рис. изображён пример начальной позиции пасьянса “Fourteen”. За один ход можно снять любые две верхние карты, сумма которых равна 14 (туз (A) считается равным 1, валет (J) — 11, дама (Q) — 12, король (K) — 13). Из изображённой позиции возможны три варианта хода: можно снять либо пару (2♠, Q♠), либо (2♡, Q♠), либо (3♡, J♣). Снятые карты в дальнейшей игре не участвуют. Других способов перемещать карты нету. Цель пасьянса — снять все карты.

Решим эту задачу алгоритмически, обобщив цель: будем максимизировать количество снятых карт (если снять все возможно, то все и будут сняты). Карты перемещаются только из стопок в отбой, поэтому, зная начальную позицию, любую позицию той же игры можно задать последовательностью из 12 чисел — количеств карт в каждой стопке. Это даёт возможность поставить серию подзадач “Какое максимальное количество карт K можно переместить в отбой, начиная с позиции $(i_1, i_2, \ldots, i_{12})$?”, где $i_1, i_2, \ldots, i_{12}$ — количества карт в стопках, и записать очень простое уравнение ДП:

$$(i_1, i_2, \ldots, i_{12}) = 2 + \max\{K(\mathrm{next}(i_1, i_2, \ldots, i_{12}))\}, \qquad (6)$$

где $\mathrm{next}(i_1, i_2, \ldots, i_{12})$ — множество всех позиций, куда можно попасть из (i_1, i_2, …, i_{12}) за один ход, и max берётся по позициям этого множества. Если множество пустое (т. к. $i_1 = i_2 = \ldots = i_{12} = 0$ или т. к. ни одна пара верхних карт не даёт сумму 14), имеем тривиальную подзадачу с ответом 0. Кстати, после достижения $i_1 = i_2 = \ldots = i_{12} = 0$ сто́ит прекращать какие бы то ни было рекурсивные вызовы, т. к. ещё лучшего решения не будет.

Нетрудно убедиться, что подзадачи тут перекрываются. Значит, надо хранить ответы решённых подзадач. Причём, значение надо получать по параметру подзадачи вида (i_1, i_2, …, i_{12}).

Работать с 12-мерным массивом можно, но неудобно (а обобщить этот способ, чтобы количество стопок задавалось во входных данных, почти нереально). Можно кодировать каждую последовательность одним числом, например по формуле $5 \cdot (5 \cdot (5 \cdot (5 \cdot (5 \cdot (5 \cdot (5 \cdot (6 \cdot (6 \cdot (6 \cdot (6 \cdot i_1 + i_2) + i_3) + i_4) + i_5) + i_6) + i_7) + i_8) + i_9) + i_{10}) + i_{11}) + i_{12}$, и работать с *одно*мерным массивом (того же размера, что и 12-мерный). Но массив всё же великоват ($6^4 \cdot 5^8 = 506\,250\,000$ элементов). Можно даже подумать, будто формула (6) неприменима из-за слишком большого количества подзадач.

Но большинство последовательностей (i_1, i_2, …, i_{12}) задают невозможные позиции — например, не может оставаться одна десятка при трёх четвёрках. Если рассматривать и хранить только возможные позиции (куда действительно можно дойти из начальной), они “с запасом” помещаются в памяти современного компьютера. Но при этом *теряется прямой доступ*, т. е. по набору (i_1, i_2, …, i_{12}) уже нельзя быстро обратиться непосредственно к элементу, хранящему значение $K(i_1, i_2, \ldots, i_{12})$, а надо *искать* элемент с ключом (i_1, i_2, …, i_{12}). Подробное рассмотрение способов такого поиска требует отдельной лекции, но один из способов кратко упомянем.

В современных версиях C++, подключив библиотеку `map` и написав `map<T1,T2> mmm`, получим так называемый “ассоциативный массив» `mmm` с “индексами” (точнее говоря, *ключами*) типа `T1` и значениями типа `T2`. Доступ к значению по ключу записывается аналогично обычным массивам — например, после объявления `map<string, double> nums;` может следовать `nums["one"] = 1.0;`. Проверку, содержит ли экземпляр `map`-а некоторый ключ, следует делать методом `find` — например, `if(nums.find(s) != nums.end()) /*действия если ключ s нашли*/ else /*действия если не нашли*/`.

“Внутри” `map` обычно устроен как красно-чёрное дерево (разновидность сбалансированных бинарных деревьев поиска). Следовательно, ключи и соответствующие им значения хранятся в узлах дерева, а поиск требует сравнений ключей вдоль пути от корня дерева до элемента с искомым ключом.

Поэтому для хранения ответов подзадач задачи о пасьянсе “Fourteen” лучше всё же кодировать каждую исследуемую позицию $(i_1, i_2, \ldots, i_{12})$ одним числом и использовать `map<int, int>` (или, если код может не помещаться в `int`, то `map<long long, int>`), но не `map<vector<int>, int>`, где `vector<int>` хранит саму последовательность $(i_1, i_2, \ldots, i_{12})$.

В языке Java аналогичная структура называется `TreeMap`. В стандартных поставках Паскаля (по крайней мере, распространённых) никакого аналога нет.

Наконец, здесь не утверждается, что изложенные `map`-ы — наилучший способ. Возможно, хорошая реализация, например, хеш-таблиц (hash tables) окажется в среднем более эффективной.

> Пусть даны N целых неотрицательных чисел a_1, a_2, …, a_N, сумма которых равна S (гарантированно $1 < N < S \leqslant 50\,000$). Нужно выбрать некоторые из них, так, чтобы сумма выбранных была не больше $S/2$ и при этом как можно ближе к $S/2$.

Широко известно, что эту задачу можно решить с помощью ДП, поставив серию подзадач “Какую наибольшую сумму $M(i, w)$, не превышающую w, можно сформировать, выбрав некоторые из чисел от 1-го по i-е?”. Для $i = 1$, при $w < a_1$ все ответы $M(1, w) = 0$, а при $w \geqslant a_1$ все ответы $M(1, w) = a_1$. Для дальнейших i:

$$M(i, w) = \begin{cases} M(i-1, w), & w < a_i; \\ \max\{M(i-1, w), M(i-1, w-a_i) + a_i\}, & w \geqslant a_i. \end{cases} \qquad (7)$$

Вариант $M(i-1, w)$ предусматривает не брать текущее число a_i, а поступить так же, как поступали для того же w при предыдущем i. Вариант $M(i-1, w-a_i) + a_i$ предусматривает, что надо взять текущее число a_i, а остаток суммы $w - a_i$ набрать наилучшим возможным образом, использовав некоторые из предыдущих чисел.

Поскольку надо выбрать сумму, как можно более близкую снизу к $S/2$, можно ограничиться диапазоном $0 \leqslant w \leqslant S \operatorname{div} 2$. Стро́ки же надо рассмотреть во всём диапазоне $1 \leqslant i \leqslant N$. Таким образом, и временна́я, и пространственная сложность алгоритма — $\theta(S{\cdot}N)$.

Этот алгоритм является *псевдополиномиальным*, т. к. оценка $S{\cdot}N$, казалось бы, полиномиальна, но зависит от S — *значения* суммы, а не *размера* входных данных в разумном смысле.

Псевдополиномиальные алгоритмы обычно невозможно обобщить, если задача изменится так, что значения вместо целых станут числами с плавающей точкой.[6] Псевдополиномиальные алгоритмы хороши там, где ничего

[6] В некоторых частных случаях можно попытаться использовать идею задачи Q

лучшего наука всё равно не знает — например, по причине NP-полноты задачи (именно такова ситуация с данной задачей.) Но там, где возможно, обычно предпочтительны "по-настоящему полиномиальные" алгоритмы.

Независимо от названий, $\theta(S{\cdot}N)$ памяти при таких ограничениях никуда не годится. Поэтому рассмотрим модификацию алгоритма. Отсортируем числа и сгруппируем одинаковые значения. Отныне будем считать, что есть m_1 чисел со значением b_1, m_2 чисел со значением b_2, ..., m_K чисел со значением b_K, причём $b_1 < b_2 < \ldots < b_K$, все m_i натуральные (целые строго положительные).

Переформулируем серию подзадач как "Какую наибольшую сумму $T(i,w)$, не превышающую w, можно сформировать, выбрав некоторые из чисел видов от 1-го по i-й?" и переделаем уравнение ДП (7) с учётом того, что теперь чисел вида b_i может быть до m_i штук:

$$M(i,w) = \max_{j\,:\,0\leqslant j\leqslant \min(m_i,\, w\,\mathrm{div}\,b_i)} \bigl(M(i-1, w-j\cdot b_i) + j\cdot b_i\bigr). \qquad (8)$$

с тривиальными подзадачами $T(1,w) = b_1 \min\{m_1,\ w\,\mathrm{div}\,b_1\}$. Т. е., начинаем рассматривать не два варианта (брать ли текущее число), а все варианты, сколько штук можно взять чисел текущего вида.

Неочевидно, что такое изменение алгоритма приведёт к *заметной* экономии памяти... И всё же исследуем взаимосвязь между K и S. Поскольку все b_i целые неотрицательные и $b_1 < b_2 < \ldots < b_K$, гарантированно выполняются $b_1 \geqslant 0$, $b_2 \geqslant 1$, ..., $b_K \geqslant K{-}1$. Следовательно, $\sum_{j=1}^{K} b_j \geqslant \frac{K(K-1)}{2}$. Поскольку все m_j натуральные, $\sum_{j=1}^{K} b_j \leqslant \sum_{j=1}^{K} m_j b_j$, что очевидно равно S. Из рассмотренных соотношений получаем неравенство $\frac{K(K-1)}{2} \leqslant S$, откуда следует $K \leqslant 2\sqrt{S+1/4} + \frac{1}{2} < 2\sqrt{S} + 1$. Следовательно, количество строк таблицы ДП теперь не превысит $\min\bigl\{N,\ \bigl\lceil 2\sqrt{S}{+}1 \bigr\rceil\bigr\}$, что не хуже, чем для предыдущего алгоритма, а в случае большого N — гораздо лучше.

Теперь оценим время работы. Ширина таблицы по-прежнему $S/2$. Для каждой клеточки запускается цикл, выполняемый $1 + \min\{m_i,\ w\,\mathrm{div}\,b_i\}$ раз. Заменив на не меньшее $m_i{+}1$, получим: время построения всех оценок i-ой строки не больше $(S/2) \times (m_i{+}1)$. Время заполнения 1-й строки $S/2$ также не больше $(S/2) \times (m_1{+}1)$. Сложив оценки по всем строкам и вынеся $S/2$, получим: суммарное время заполнения таблицы не больше $(S/2) \times \sum_{j=1}^{K} (m_j{+}1)$, что с учётом $\sum_{j=1}^{K} m_j = N$ равно $(S/2) \times (N{+}K)$. Поскольку $K \leqslant N$, окончательно время выполнения (8) может быть оценено как $O(S{\cdot}N)$, что не хуже, чем для (7), требующего больше памяти.

Задачи и разборы

Задача A. Платформы

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

В старых играх можно столкнуться с такой ситуацией. Герой прыгает по платформам, висящим в воздухе. Он должен перебраться от одного края экрана до другого. При прыжке с платформы на соседнюю, у героя уходит $|y_2 - y_1|$ энергии, где y_1 и y_2 — высóты, на которых расположены эти платформы. Кроме того, есть суперприём, позволяющий перескочить через платформу, но на это затрачивается $3 \cdot |y_3 - y_1|$ энергии.

Известны высóты платформ в порядке от левого края до правого. Найдите минимальное количество энергии, достаточное, чтобы добраться с 1-й платформы до n-й (последней) и список (последовательность) платформ, по которым нужно пройти.

Формат входного файла

Первая строка содержит количество платформ N ($2 \leqslant N \leqslant 100\,000$), вторая — N целых чисел, значения которых не превышают по модулю 4000 — высóты платформ.

Формат выходного файла

В первой строке выведите минимальное количество энергии. Во второй — количество платформ, по которым нужно пройти, а в третьей выведите список этих платформ.

Пример

a.in	a.out
4 1 2 3 30	29 4 1 2 3 4

Разбор задачи A. Платформы

Задача полностью разобрана на лекции.

Задача B. Покупка билетов

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

За билетами на премьеру нового мюзикла выстроилась очередь из N человек, каждый из которых хочет купить 1 билет. На всю очередь работала только одна касса, поэтому продажа билетов шла очень медленно, приводя "постояльцев" очереди в отчаяние. Самые сообразительные быстро заметили, что, как правило, несколько билетов в одни руки кассир продаёт быстрее, чем когда эти же билеты продаются по одному. Поэтому они предложили нескольким подряд стоящим людям отдавать деньги первому из них, чтобы он купил билеты на всех.

Однако для борьбы со спекулянтами кассир продавала не более 3-х билетов в одни руки, поэтому договориться таким образом между собой могли лишь 2 или 3 подряд стоящих человека.

Известно, что на продажу i-му человеку из очереди одного билета кассир тратит A_i секунд, на продажу двух билетов — B_i секунд, трёх билетов — C_i секунд. Напишите программу, которая подсчитает минимальное время, за которое можно обслужить всех покупателей.

Обратите внимание, что билеты на группу объединившихся людей всегда покупает первый из них. Также, никто в целях ускорения не покупает лишних билетов (то есть билетов, которые никому не нужны).

Формат входного файла

Во входном файле записано сначала число N — количество покупателей в очереди ($1 \leqslant N \leqslant 5000$). Далее идет N троек натуральных чисел A_i, B_i, C_i. Каждое из этих чисел не превышает 3600. Люди в очереди нумеруются начиная от кассы.

Формат выходного файла

В выходной файл выведите одно число — минимальное время в секундах, за которое можно обслужить всех покупателей.

Пример

b.in	b.out
5 5 10 15 2 10 15 5 5 5 20 20 1 20 1 1	12
2 3 4 5 1 1 1	4

Разбор задачи B. Покупка билетов

Весьма популярная задача, используется на многих российских сайтах; играла на Московской городской олимпиаде 08.02.2004, автор — В. А. Матюхин.

Тоже очень простое ДП. Подзадачи можно поставить, например, как "За какое минимальное время $T(i)$ может быть обслужена часть очереди от начала по i-го покупателя включительно?". Изначальная задача оказывается N-подзадачей, уравнение ДП имеет вид:

$$T(i) = \min(T(i-1) + A(i), T(i-2) + B(i-1), T(i-3) + C(i-2)). \qquad (9)$$

1-я сумма задаёт вариант, когда последний i-й покупатель ни с кем не объединялся, 2-я — когда его покупку сделал впереди стоящий, 3-я — когда $(i-2)$-й сделал покупку за себя, $(i-1)$-го и i-го.

Несомненно, тривиальной подзадачей будет $T(1) = A(1)$ (очередь из одного только 1-го человека, он покупает один билет). Но кроме того надо добавить к тривиальным подзадачам, например, $T(0) = 0$ и $T(2) = \min(A(1) + A(2), B(1))$. При $i=2$ уже есть выбор (объединяются ли 1-й и 2-й покупатели), но его ещё нельзя делать по формуле (9). При $i=3$, благодаря $T(0) = 0$, применима общая формула.

Задача C. Easy MaxSum

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Есть прямоугольная таблица размером M строк на N столбиков. В каж-

дой клетке записано целое число. По ней нужно пройти сверху вниз, начиная из любой клетки верхней строки, дальше каждый раз переходя в одну из "нижних соседних" клеток (иными словами, из клетки с номером (i, j) можно перейти или на $(i+1, j-1)$, или на $(i+1, j)$, или на $(i+1, j+1)$; в случае $j = N$ возможны только 1-й и 2-й из трёх описанных вариантов, в случае $j = 1$ — только 2-й и 3-й) и закончить маршрут в какой-нибудь клетке нижней строки.

Напишите программу, которая будет находить максимально возможную сумму значений пройденных клеток среди всех допустимых путей и количество путей, на которых эта сумма достигается. Гарантированно, что при проверке будут использованы только такие входные данные, для которых количество путей не превышает 10^9.

Формат входного файла

В первой строке записаны M и N — количество строчек и количество столбиков ($2 \leqslant M \leqslant 200$, $2 \leqslant N \leqslant 40$, $M \geqslant N$), дальше в каждой из следующих M строк записано ровно по N разделённых пробелами целых чисел (каждое не превышает по модулю 10^6) — значения клеток таблицы.

Формат выходного файла

Вывести два числа, разделенные пробелом — максимальную сумму и количество путей.

Пример

c.in	c.out
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	42 1

Разбор задачи C. Easy MaxSum

Сначала рассмотрим задачу поиска просто максимальной суммы (без количества путей). Серию подзадач поставим как "Какую максимальную сумму $e(i, j)$ можно насобирать, дойдя из 1-й строки до j-й клеточки i-й строки?". Номер клеточки в 1-й строке (откуда начинать путь) *не* задаётся как параметр. Целевая задача не принадлежит серии, но, зная решение (M, j)-подзадач при всех j, остаётся лишь найти максимум в одномерном массиве.

Решениями тривиальных $(1, j)$-подзадач при всех j будут просто $e(1, j) = a(1, j)$ (для каждой клеточки 1-й строки есть ровно один допустимый путь — состоящий только из этой клеточки). Решения нетривиальных задач будут строиться по уравнению ДП:

$$e(i, j) = \max\{e(i-1, j-1),\ e(i-1, j),\ e(i-1, j+1)\} + a(i, j) \qquad (10)$$

(с поправкой, что при $j = 1$ пропускается первый из вариантов, а при $j = N$ — последний). Для доказательства преобразуем формулу к виду $e(i, j) = \max\{e(i-1, j-1) + a(i, j),\ e(i-1, j) + a(i, j),\ e(i-1, j+1) + a(i, j)\}$. Первый из вариантов даёт максимально возможную сумму среди всех путей, приходящих в (i, j) через левого верхнего соседа, второй — среди путей через верхнего соседа, и третий — среди путей через правого верхнего. Максимум из них и будет оптимальным решением для клеточки (i, j).

Теперь перейдём к поиску количества $q(i, j)$ путей с максимальной суммой, приводящих в клеточку (i, j). Для любой клеточки 1-й строки $q(1, j) = 1$ — путь только один, он и есть максимальный. Разные пути с одинаковой максимальной суммой возникают, когда два или три варианта, из которых выбирается $\max$ в (10), имеют одинаковые максимальные значения. В таком случае надо сложить количества путей с максимальной суммой всех максимальных направлений.

```
max:=e[i-1][j];
if (j>1)and(e[i-1][j-1]>max) max:=e[i-1][j-1];
if (j<N)and(e[i-1][j+1]>max) max:=e[i-1][j+1];
e[i][j]:=max+a[i][j];
q[i][j]:=0;
if(j>1)and(e[i-1][j-1]=max)then q[i][j]+=q[i-1][j-1];
if        (e[i-1][j]=max)  then q[i][j]+=q[i-1][j];
if(j<N)and(e[i-1][j+1]=max)then q[i][j]+=q[i-1][j+1];
```

(где `a+=b` означает `a:=a+b`).

Задача D. Старые песни о главном – 3

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Во время трансляции концерта “Старые песни о главном – 3” предприниматель К. решил сделать бизнес на производстве кассет. Он имеет M

кассет с длительностью звучания D каждая и хочет записать на них максимальное число песен. Эти песни (их общее количество N) передаются в порядке 1, 2, ..., N и имеют заранее известные длительности звучания $L(1)$, $L(2)$, ..., $L(N)$. Предприниматель может выполнять одно из следующих действий:

- записать очередную песню на кассету (если она туда помещается) или пропустить её;
- если песня на кассету не помещается, то можно пропустить песню или начать её записывать на новую кассету. При этом старая кассета откладывается и туда уже ничего не может быть записано.

Определить максимальное количество песен, которые предприниматель может записать на кассеты.

Формат входного файла

Первая строка содержит число M ($\leqslant 100$).
Вторая строка — D ($\leqslant 300$).
Третья строка — N ($\leqslant 300$).
Четвёртая строка — $L(1)$ $L(2)$ ... $L(N)$. Все $L(i) \leqslant D$.
Все входные параметры — натуральные числа.

Формат выходного файла

Необходимо вывести единственное число — искомое количество песен.

Пример

d.in	d.out
2 10 4 6 6 6 4	3

Разбор задачи D. Старые песни о главном – 3

Задача взята из "Динамического программирования" В. М. Котова; по-видимому, сам Котов примерно такое псевдополиномиальное решение и подразумевал. Но в старшей лиге оно не годится — там необходим "по-настоящему" полиномиальный алгоритм.

Поставим серию подзадач "Какое максимальное количество песен $T(i, j, k)$ возможно записать, выбрав некоторые из песен от 1-й по j-ю

включительно, ранее финализировав i штук кассет и использовав k единиц времени на текущей $(i{+}1)$-й?".

В пределах кассеты $T(i,j,k)$ можно получать как:

$$T(i,j,k) = \max\{T(i,j{-}1,k),\ 1 + T(i,j{-}1,k{-}L[j]),\ T(i,j,k{-}1)\}. \quad (11)$$

Смысл 1-го варианта — не писать текущую песню, 2-го — писать, 3-го — занять единицу времени тишиной. Для каждого варианта надо проверять, применим ли он: для 1-го — $j{>}0$, для 2-го — $(j{>}0)$ and $(k{\geq}L[j])$, для 3-го — $k > 0$. А при переходе на новую кассету:

$$T(i,j,0) = T(i{-}1,j,D). \quad (12)$$

Если бы в (11) не рассматривали вариант "писать тишину", в массиве ответов могли бы появиться "дырки", где хранится непонятно что, а вместо (12) нужно было бы $T(i,j,0){=}\max_{0\leq k\leq D} T(i{-}1,j,k)$.

Итак, порядок действий таков: начинаем с тривиальной подзадачи $T(0,0,0){=}0$; находим $T(0,j,k)$ для всех j, k; переходим ко 2-й кассете по $T(1,j,0){=}T(0,j,D)$ для всех j; находим $T(1,j,k)$ для всех j, k; и т. д. Окончательный ответ равен $T(M{-}1,N,D)$.

Задача E. Разбиение на две группы

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

С полярниками произошла необычная для исследователей история — о них вспомнили. Плохо, что вспомнили не для того, чтобы пополнить запасы продовольствия и топлива. Вспомнили о них лишь потому, что недалеко решили создать новую станцию, а опытных полярников найти сложно. Вот и решили разбить всех полярников этой станции на две части и одну из них направить на новую станцию. Но возникла проблема: некоторые полярники настолько сблизились друг с другом, что отказываются работать на разных станциях. Все полярники на станции разбились на группы, которые будут работать только вместе или не будут работать вообще. Помогите разделить группы полярников на две максимально одинаковые части (т. е. разность количеств полярников в частях по модулю должна быть минимальна), не разрывая группы. Обе части должны быть не пустыми.

Формат входного файла

Первая строка содержит число N — количество групп людей. Вторая строка содержит N натуральных (целых строго положительных) чисел —

количество людей в каждой группе. $2 \leqslant N \leqslant 30000$, суммарное количество полярников во всех N группах не превышает 50000.

Формат выходного файла

В первой строке необходимо вывести размеры каждой из двух частей.

Во второй и третьей строке необходимо вывести список номеров групп, которые входят в первую и вторую части соответственно (группы нумеруются в порядке перечисления во входных данных, начиная с единицы). Если существует несколько правильных ответов — выводите любой из них.

Пример

e.in	e.out
5 3 2 5 1 2	6 7 1 2 4 3 5

Разбор задачи E. Разбиение на две группы

При внимательном анализе становится ясно, что следует реализовать описанный на лекции псевдополиномиальный алгоритм (включая обратный ход).

Задача F. Абзац

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

В абзаце есть блоки очень разной высоты (например, обычные слова и математические системы). Абзац длинный, поэтому его нужно разбить на стро́ки. Высота строки́ определяется по наивысшему из блоков в ней. Высота абзаца равна сумме высот всех строк. Длина каждой строки определяется как суммарная ширина блоков, включённых в эту строку (учитывать пробелы не нужно). Возможность разбиения блока для переноса со строки́ на строку не рассматривается. Изменять порядок следования блоков нельзя. Нужно найти такое разбиение абзаца на строки, чтобы высота абзаца была минимальной. Ширина и высота каждого блока $(w(i), h(i))$ и максимально допустимая длина строк TW задаются во входных данных.

Формат входного файла

В первой строке записано два числа — TW (максимально допустимая длина строки) и N (количество блоков в абзаце), где $5 \leqslant N \leqslant 5000$. В следующих N строках — по два числа (ширина и высота блока).
Все размеры — натуральные числа не больше 10^6. Гарантировано, что для всех блоков $w(i) \leqslant TW$.

Формат выходного файла

В первой строке выведите минимальную высоту абзаца. Во второй — количество строк M, на которые нужно разбить абзац, а в следующих M строках выведите количество блоков в соответствующих строках абзаца.

Пример

f.in	f.out
7 6 3 1 2 1 3 3 1 1 3 3 3 1	5 3 2 3 1

Разбор задачи F. Абзац

Задача полностью разобрана на лекции.

Задача G. Пасьянс

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

“N-T пасьянс” — карточная игра для одного игрока. В игре используется $4N$ $(3 \leqslant N \leqslant 15)$ карт, причем каждой карте соответствует уникальная пара её значения (целое число в диапазоне $1..N$) и масти (♠, ♣, ♡ или ♢). В начальном положении все карты разложены в T $(4 \leqslant T \leqslant 12)$ стопок; при этом каждая из первых $(4N)\%T$ стопок содержит по $(4N/T+1)$ карт, остальные — по $4N/T$ карт (здесь “/” и “%” — целочисленное деление и остаток от деления соответственно). Если сумма значений верхних карт

двух стопок равна $N+1$, то эти две карты можно переместить в отбой (независимо от их мастей). Это единственный способ перемещать карты.

Напишите программу, которая будет определять, какое максимальное количество карт можно переместить в отбой.

Формат входного файла

Первая строка содержит два целых числа N и T, далее идут T строк с описаниями карт соответствующей стопки. Каждая карта описывается её значением (целое число) и мастью (символ с ASCII-кодом 03(♡), 04(♢), 05(♠), или 06(♣)) без пробела между ними. Описания разных карт одной стопки разделены ровно одним пробелом, направление описания слева направо соответствует порядку карт снизу вверх.

Формат выходного файла

Ваша программа должна вывести единственное целое число — максимально возможное количество карт, которые можно переместить в отбой.

Пример

g.in	g.out
3 5 2♠ 2♣ 2♡ 2♢ 3♢ 1♡ 3♣ 1♠ 1♣ 3♡ 1♢ 3♠	10

Для заданной начальной позиции, игрок может переместить в отбой все карты, кроме пары нижних двоек в 1-й стопке.

Разбор задачи G. Пасьянс

Задача полностью разобрана на лекции. Может показаться, будто наличие (на самом деле ненужных) символов мастей сильно усложняет ввод. Но на самом деле большинство стандартных способов ввода даже не заметит этих символов. . .

Задача H. Normal MaxSum

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Авторы дня думают, что вам приходилось решать такую задачу: “Есть прямоугольная таблица размером M строк на N столбиков. В каждой клетке записано целое число. По ней нужно пройти сверху вниз, начиная из любой клетки верхней строки, дальше каждый раз переходя в одну из “нижних соседних” клеток (иными словами, из клетки с номером (i, j) можно перейти или на $(i+1, j-1)$, или на $(i+1, j)$, или на $(i+1, j+1)$; в случае $j = N$ возможны только 1-й и 2-й из трёх описанных вариантов, в случае $j = 1$ — только 2-й и 3-й) и закончить маршрут в какой-нибудь клетке нижней строки. . . ”

Пусть добавлено дополнительное ограничение: допускаются только пути, которые проходят (хотя бы по одному разу) через все столбики.

Напишите программу, которая будет находить максимально возможную сумму значений пройденных клеток среди всех допустимых путей.

Формат входного файла

В первой строке входа записаны M и N — количество строк и количество столбиков ($2 \leqslant M \leqslant 1024$, $2 \leqslant N \leqslant 768$, $M \geqslant N$), дальше в каждой из следующих M строк записано ровно по N разделённых пробелами целых чисел, не превышающих по модулю 10^6 — значения клеток таблицы.

Формат выходного файла

Выход должен содержать единственное число — максимальную сумму.

Пример

h.in	h.out
4 3 1 15 2 9 7 5 9 2 4 6 9 -1	28

Разбор задачи H. Normal MaxSum

Алгоритм решения базовой задачи (без ограничения насчёт всех столбиков) описан в разборе задачи C. Данная (модифицированная) задача игра-

ла на 4-ом туре NetOI–2008, но тогдашнее авторское решение не проходит (по времени) часть теперешних тестов.

Разумеется, для решения модифицированной задачи надо расширить серию подзадач базовой задачи. И, оказывается, её можно расширить, увеличив общее количество подзадач и время работы алгоритма всего в 4 раза: вместо серии (i,j)-подзадач (где i и j — номера строки и столбика), поставить серию (i,j,wl,wr)-подзадач, где смысл целочисленных i и j тот же, а смысл *булевых* wl и wr — "был ли хотя бы раз посещён крайний (левый/правый) столбик?".

Многие такие подзадачи невозможны (скажем, $(1,j,\texttt{false},wr)$ при любых j и wr, или $(i,N\operatorname{div}2,\texttt{true},\texttt{true})$ при $i<N\operatorname{div}2$). Чтобы хранить какие-то ответы для этих подзадач, можно задать ответы невозможных тривиальных подзадач равными "машинной $(-\infty)$". Но, чтобы и избежать переполнений, и гарантировать, что $(-\infty)$ плюс любое осмысленное значение меньше любого отрицательного осмысленного значения, приходится работать в типе `_int64`. (Разумеется, можно и не пользоваться "машинной $(-\infty)$", а хранить или определять невозможность подзадачи как-то иначе.)

Само уравнение ДП для $i>1$ имеет вид:

$$\begin{aligned}
&\text{для } 2\le j\le N{-}1,\quad e(i,j,wl,wr)=\max\big(e(i{-}1,j{-}1,wl,wr),\\
&\qquad e(i{-}1,j,wl,wr),e(i{-}1,j{+}1,wl,wr)\big)+a(i,j);\\
&e(i,1,\texttt{false},wr)=-\infty;\\
&e(i,1,\texttt{true},wr)=\max\big(e(i{-}1,1,\texttt{true},wr),\\
&\qquad e(i{-}1,2,\texttt{true},wr),e(i{-}1,2,\texttt{false},wr)\big)+a(i,j);\\
&e(i,N,wl,\texttt{false})=-\infty;\\
&e(i,N,wl,\texttt{true})=\max\big(e(i{-}1,N,wl,\texttt{true}),\\
&\qquad e(i{-}1,N{-}1,wl,\texttt{true}),e(i{-}1,N{-}1,wl,\texttt{false})\big)+a(i,j).
\end{aligned}\tag{13}$$

Задача I. Платформы – 3

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

В старых играх можно столкнуться с такой ситуацией. Герой прыгает по платформам, висящим в воздухе. Он должен перебраться от одного края экрана до другого. При прыжке с платформы на соседнюю, у героя уходит $|y_2-y_1|^2$ энергии, где y_1 и y_2 — высóты, на которых расположены

эти платформы. Кроме того, есть суперприём, позволяющий перескочить через платформу, но на это затрачивается $3 \cdot |y_3 - y_1|^2$ энергии.

Известны высóты платформ в порядке от левого края до правого. Найдите минимальное количество энергии, достаточное, чтобы добраться с 1-й платформы до n-й (последней).

Формат входного файла

Первая строка содержит количество платформ N ($2 \leqslant N \leqslant 100\,000$), вторая — N целых чисел, значения которых не превышают по модулю 4000 — высóты платформ.

Формат выходного файла

Выведите единственное целое число — искомую величину энергии.

Пример

i.in	i.out
4 1 2 3 30	731

Разбор задачи I. Платформы - 3

На первый взгляд, задача элементарна — берём уравнение ДП с лекции, заменяем модули разностей высот на квадраты этих же разностей... Но это не проходит тесты (с вердиктом Wrong Answer)! Перечитываем условие, исправляем выход за пределы 32-битного `int`-а... Всё равно Wrong Answer!

В условии *не запрещено прыгать в обратную сторону*. Просто при функциях затрат энергии из предыдущей версии задачи прыгать назад всё равно оказывалось невыгодно, а при функциях этой задачи — в зависимости от входных данных. Например, при y_1=10, y_2=32, y_3=18, y_4=40 минимум суммарных затрат энергии (580) достигается при порядке $1 \to 3 \to 2 \to 4$.

Наивная попытка изменить уравнение ДП на

$$E(i) = \min \left\{ \begin{array}{ll} E(i-1) + (y_i - y_{i-1})^2, & E(i-2) + 3(y_i - y_{i-2})^2, \\ E(i+1) + (y_i - y_{i+1})^2, & E(i+2) + 3(y_i - y_{i+2})^2 \end{array} \right\}$$

не приводит к немедленному успеху. Это математически правильно, но его непросто алгоритмизировать из-за циклических зависимостей (например, $E(2)$ зависит от $E(3)$, а $E(3)$ — от $E(2)$).

При желании тут всё же можно применить ДП. Поскольку затраты энергии на любой прыжок неотрицательны, бессмысленно рассматривать маршруты, содержащие циклы (повторные посещения той же платформы) — если эти циклы "вырезать", расход энергии не увеличится. Поскольку максимальная длина прыжка равна 2, последовательность $(+2)$, (-1), $(+2)$ — *единственная*, использующая перемещения назад, но не создающая циклов. Поэтому можно считать, будто двигаться разрешено только вперёд, причём есть три вида прыжков: 1) прыжок с $(i-1)$ на i, требующий $(y_i - y_{i-1})^2$ энергии; 2) прыжок с $(i-2)$ на i, требующий $3 \cdot (y_i - y_{i-2})^2$ энергии; 3) прыжок с $(i-3)$ на i, требующий $3 \cdot (y_{i-1} - y_{i-3})^2 + (y_{i-2} - y_{i-1})^2 + 3 \cdot (y_i - y_{i-2})^2$ энергии.

Ещё один способ решения — рассмотреть платформы как вершины графа, возможные прыжки — как рёбра, и найти в этом взвешенном неориентированном графе путь минимальной длины (например, алгоритмом Дейкстры, причём надо использовать реализацию со сложностью $O(M \log N)$, а не $O(N^2)$).

Задача J. Сомневающееся начальство

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

На координатной плоскости заданы N вершин, и надо многократно находить кратчайший среди путей, проходящих через эти вершины и возвращающихся в начальную. (Длина пути равна сумме длин составляющих его отрезков, длину одного отрезка можно посчитать по формуле $\sqrt{\Delta x^2 + \Delta y^2}$.) Зачем один и тот же путь находить многократно? А он не один и тот же — начальство никак не определится, какие из этих N вершин в самом деле нужны, а какие нет. И потому приходится для одного и того же набора вершин обрабатывать разные запросы, отличающиеся набором нужных вершин. В каком порядке проходить нужные вершины и какую из них выбрать в качестве начальной (она же конечная), выбирает не начальство, а тот, кто ищет минимальный путь.

Формат входного файла

В первой строке задано количество вершин N ($4 \leqslant N \leqslant 17$). В каждой из последующих N строк заданы по два целых числа, не превышающих по модулю 10^6 — x - и y - координаты очередной вершины. В следующей строке задано число Q — количество запросов от начальства (оно

не ограничено явно, но гарантировано, что все запросы разные). Каждая из следующих Q строк содержит запрос, в формате: сначала количество K ($3 \leqslant K \leqslant N$) вершин, через которые надо пройти, потом номера этих вершин (каждый номер в пределах от 1 до N, все вершины в одном запросе разные).

Формат выходного файла

Программа должна вывести Q строк, в каждой из которых — единственное действительное число, равное ответу на соответствующий запрос. Ответы будут засчитываться, если абсолютная погрешность не превысит 10^{-2}.

Пример

j.in	j.out
4 0 0 10 0 0 10 10 10 2 4 1 2 3 4 3 1 2 4	40 34.142136

Разбор задачи J. Сомневающееся начальство

Вообще-то, и задача (коммивояжёра), и способ применить к ней ДП давным-давно известны — более того, это одно из исторически первых применений ДП к дискретным задачам (как ни странно, изначально ДП разрабатывалось для непрерывных задач). Решение задачи коммивояжёра с помощью ДП не популярно, т. к. требует экспоненциальных затрат и времени, и памяти. В данной задаче, надеемся, удалось добиться, чтобы тесты проходили только реализации ДП, но только при ма́лом N и за счёт не очень естественного требования, что задачу надо решать многократно для различных подмножеств одного и того же множества вершин.

Серия подзадач имеет вид “Какова минимальная длина пути $L(s, f, X)$, начинающегося в вершине s, заканчивающегося в вершине f, и проходящего (неважно, в каком порядке) через множество вершин X?” (Естественно, подзадачи имеют смысл только при $(s{\in}X){\wedge}(f{\in}X)$). Целевая задача не будет одной из подзадач, но может быть решена, например, как $\min_{2\leq k\leq n}\{L(1,k,A)+d(k,1)\}$, где A — множество всех чисел от 1 до n, $d(\cdot,\cdot)$ — расстояние (длина одного ребра). Тривиальными будут все подзадачи вида $(s, f, \{s, f\})$, т. е. в которых множество пройденных вершин

состоит только из старта и финиша. Уравнение ДП имеет вид:

$$L(s, f, X) = \min_{k \in X \setminus \{s,f\}} \big\{ L(s, k, X \setminus \{f\}) + d(k, f) \big\}. \tag{14}$$

Чтобы иметь доступ к решениям подзадач по множествам как по ключам, закодируем их. Сместим нумерацию так, чтобы она начиналась с 0, предста́вим это множество в виде битового вектора, и будем пользоваться этим битовым вектором как целым индексом массива. Например, множество $\{2, 4, 6\}$ сначала преобразуется в $\{1, 3, 5\}$, потом в $1 \cdot 2^1 + 1 \cdot 2^3 + 1 \cdot 2^5 = 2 + 8 + 32 = 42$, т. е. ответ для множества $\{2, 4, 6\}$ хранится в элементе с индексом 42.

Может показаться, будто при таком подходе на хранение всех ответов в типе `double` понадобится $8 \cdot 17 \cdot 17 \cdot 2^{17}$ байтов памяти, что превышает отведённые 256 Mb. Но, поскольку расстояния симметричны (неориентированны), можно всё время рассматривать только подзадачи при $s < f$, и организовать двумерный массив указателей, в котором только `L[s][f]` при `s < f` показывает на один из выделенных в свободной памяти массивов `double`-ов с диапазоном индексов от 0 до $2^N - 1$, а прочие указатели показывают “в никуда”. (Разумеется, при этом нельзя считать, будто все промежуточные вершины обязаны принадлежать диапазону от s до f).

Использовать здесь `map`-ы нецелесообразно: хотя в массивах и будут “дырки” (неиспользуемые элементы), при достаточно большом наборе запросов окажется, что подзадачи надо решить для *большинства* подмножеств, и то, что `map` хранит только нужные элементы, будет поглощено тем, что затраты памяти на один элемент в `map`-е гораздо больше, чем в массиве. Да и время доступа к элементу в `map`-е значительно больше, чем в массиве.

Если бы надо было не находить ответы на разные запросы, а просто решить задачу для полного множества, можно было бы не вводить два параметра s и f, а, например, заменить все s на 1. Но тогда было бы целесообразно применять вообще другие алгоритмы — тоже экспоненциальные по времени, но хотя бы не по памяти.

Задача K. Путь через горы

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Поверхность Земли в горной местности можно представить в виде ломаной линии. Вершины ломаной расположены в точках

(x_1, y_1), (x_2, y_2), ..., (x_N, y_N), при этом $x_i < x_{i+1}$. Обычный горный маг находится в точке (x_1, y_1) и очень хочет попасть в точку (x_N, y_N). При этом он может перемещаться только пешком. Он может ходить по поверхности Земли (т. е. вдоль ломаной). А может сотворить в воздухе мост и пройти по нему. Мост может соединять две вершины ломаной: мост не может начинаться и заканчиваться не в вершине ломаной, и мост не может проходить под землей (в т. ч. не может быть туннелем в горе), но мост может каким-то своим участком проходить по поверхности земли. Длина моста не может быть больше R. Суммарно маг может построить не более K мостов. После прохождения моста, он (мост) растворяется в воздухе. Какое наименьшее расстояние придётся пройти магу, чтобы оказаться в точке (x_N, y_N)?

Формат входного файла

Программа должна прочитать сначала натуральное число N ($2 \leqslant N \leqslant 200$); затем натуральное число K ($1 \leqslant K \leqslant 100$) — максимальное количество мостов; далее целое число R ($0 \leqslant R \leqslant 10000$) — максимальную возможную длину моста. Далее координаты (x_1, y_1), (x_2, y_2), ..., (x_N, y_N). Все координаты — целые числа, не превышающие по модулю 10000, для всех i от 1 до $N-1$ выполняется $x_i < x_{i+1}$.

Формат выходного файла

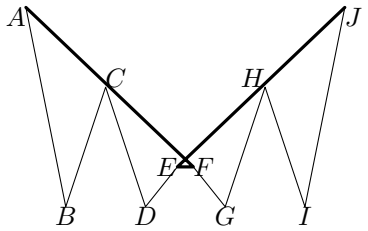
Программа должна вывести одно число — минимальную длину пути, которую придётся пройти магу (как по земле, так и по мостам). Ответ выведите с точностью 5 цифр после десятичной точки.

Пример

k.in	k.out
5 2 5 0 0 2 2 3 -1 4 1 5 0	6.47871

Разбор задачи K. Путь через горы

На первый взгляд, задача взята с Московской командной олимпиады 19.10.2008 (автор — О. Пакуляк). Но не все решения, проходящие тот набор тестов, пройдут наш набор.



Как и в задаче I, тут *можно* возвращаться. Например, пусть для изображённого (тонкой линией) рельефа можно строить не более двух мостов; длина моста меньше AJ, AH, CJ, но достаточна для AF и EJ; строить мост AE или FJ нельзя из-за вершин C и H соответственно. В такой ситуации кратчайшим может быть маршрут $A{\to}F{\to}E{\to}J$. Более того, возможны случаи, когда выгодно строить мост в обратном направлении.

Поставим серию подзадач "Пройдя какое минимальное расстояние $d(i,j)$, можно дойти от 1-й вершины ломаной до j-й, построив при этом ровно i мостов?". Ответ целевой задачи можно получить как $\min_{0\le i\le K}\{d(i,N)\}$. Тривиальными объявим все $d(0,j)$-подзадачи, их решения — длины путей до соответствующих вершин *вдоль ломаной*. Уравнение ДП для нетривиальных подзадач имеет вид:

$$d(i,j)=\min\left(\begin{array}{c} d(i,j{-}1)+l(j{-}1,j),\\ \min\limits_k\{d(i{-}1,k)+l(k,j)\},\\ d(i,j{+}1)+l(j{+}1,j)\end{array}\right), \qquad (15)$$

где $l(\cdot,\cdot)$ — длина одного перехода (из соседней вершины или по мосту). Верхняя строка задаёт вариант "прийти по земле слева", средняя — способы прийти по мосту (k перебирает вершины, из которых можно построить мост в j), нижняя — "прийти по земле справа".

Таблицу решений удобно заполнять сначала по всем j при $i{=}1$, потом по всем j при $i{=}2$, и т. д. — значит, $d(i{-}1,k)$ готово независимо от того, $j{>}k$ или $j{<}k$. С вариантами "прийти по земле" сложнее, т. к. на первый взгляд $d(i,j{+}1)$ и $d(i,j)$ циклически зависят друг от друга. Но, по причине положительности расстояний, прийти в j-ю вершину по земле справа может быть выгодно *только* если в некоторую $(j{+}c)$-ю вершину пришли по мосту, а дальше шли по земле в $(j{+}c{-}1)$-ю, …, $(j{+}1)$-ю, j-ю. Поэтому достаточно сначала, меняя j от 2 до N, посчитать оценки по таблице решений без учёта варианта "прийти по земле справа", потом для того же i дополнительно проверить только этот вариант встречным циклом по j от $N{-}1$ до 2.

Для эффективной реализации средней строки́ надо перебирать вершины, из которых можно построить мост (в прямой видимости и не дальше R). Запомним эти перечни в виде списков смежности взвешенного графа. При построении графа, внешний цикл перебирает вершины в переменной i, внутренний — в j, начиная с $i{+}1$. Если при нахождении ребра добавлять две его копии (из i в j и из j в i), рассматривать $j{<}i$ не надо. Чтобы быстро определять, видна ли вершина j из i или её закрывают предыдущие,

поддерживаем вектор направления на последнюю из видимых вершин и сравниваем (по формуле "ориентированной площади" $a_x b_y - a_y b_x$) каждый очередной вектор $\overrightarrow{v_i v_j}$ с этим направлением.

Как и в задаче I, возможно альтернативное решение с помощью алгоритма Дейкстры, но теперь граф должен быть "многослойный": вершинами орграфа надо объявить *пары* (вершина ломаной, количество использованных мостов), дугами — соседние вершины ломаной при одинаковом количестве мостов и сами мосты (там, где они геометрически допустимы, причём дуги направлены из i-го "слоя" графа в $(i+1)$-й).

Задача L. Квадратные и круглые

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Чую, брате, шо буде війна,
Холодна і мокра, довга і зла.
Квадратні на круглих будуть іти
За то шо ті рівні, як їх не крути...

гурт "Скрябін"

Вождь Квадратных, находясь в состоянии аффекта, выдал каждому из своих бойцов настолько большую дозу Озверина, что каждый из них сначала нападает, а потом уже разбирается, напал он на Квадратного или на Круглого. Круглые же никогда не умели воевать врукопашную, но они владеют некоторыми магическими приёмами.

Итак, отряд из S Квадратных напал на посёлок, где жили R Круглых. Вследствие наложения действия Озверина и магических приёмов Круглых, оказалось, что бой происходит по таким правилам. Ежеминутно случайно выбирается пара существ, и:

- если выбраны Круглый и Квадратный, то Квадратный убивает Круглого и воюет дальше;
- если выбраны два Круглых, они просят друг у друга прощения и воюют дальше;
- если выбраны два Квадратных, они убивают друг друга.

Пара выбирается случайно и равновероятно среди всех живых так, чтобы встретились два разных существа (возможно, одного вида). Например,

когда живы два Квадратных и один Круглый, то с одинаковыми вероятностями $1/3$ состоится или стычка 1-го Квадратного и Круглого, или 2-го Квадратного и Круглого, или 1-го и 2-го Квадратных. В первых двух случаях Круглый погибнет, а в третьем — погибнут Квадратные.

Напишите программу, которая найдет вероятность того, что при соблюдении данных правил боя хоть один Круглый выживет, а все Квадратные погибнут.

Формат входного файла

В единственной строке содержится два числа — количество Квадратных S ($1 \leqslant S \leqslant 2010$) и количество Круглых R ($1 \leqslant R \leqslant 2010$).

Формат выходного файла

Вывести одно число — искомую вероятность (с относительной погрешностью не более 10^{-9}).

Пример

l.in	l.out
2 20	0.80545497225
2 1	0.333333333333

Разбор задачи L. Квадратные и круглые

Квадратные могут гибнуть только парами, поэтому для любого нечётного S ответ равен в точности 0. Для решения же задачи при чётных S поставим серию подзадач “Какова вероятность $p(s,r)$, что все Квадратные погибнут, а хотя бы один Круглый выживет, если сначала было s Квадратных и r Круглых?”. Тривиальными подзадачами будут: $p(s,0){=}0$ для любого $s{\geq}0$; $p(0,r){=}1$ для любого $r{\geq}1$.

Когда живы s Квадратных и r Круглых, общее количество пар различных существ равно $C^2_{s+r}{=}\frac{(s+r)(s+r-1)}{2}$. Из этого количества, $s\cdot r$ приходится на пары “Круглый–Квадратный”, $C^2_s{=}\frac{s(s-1)}{2}$ — на пары “Квадратный–Квадратный”, $C^2_r{=}\frac{r(r-1)}{2}$ — на пары “Круглый–Круглый”.

Следовательно, по формуле полной вероятности,

$$p(s,r){=}\tfrac{2sr}{(s+r)(s+r-1)}p(s,r{-}1){+}\tfrac{s(s-1)}{(s+r)(s+r-1)}p(s{-}2,r){+}\tfrac{r(r-1)}{(s+r)(s+r-1)}p(s,r),$$

а это почти уравнение ДП.[7] Разумеется, его надо преобразовать, убрав зависимость $p(s,r)$ от себя же. Но это просто (домножаем всё равенство на

[7] есть разные мнения, относить ли такие алгоритмы к ДП: тут есть взаимосвязь однотипных одзадач разных размеров, но нет оптимизации

$(s+r)(s+r-1)$, переносим $r(r-1)p(s,r)$ в левую часть, делим, чтоб получить слева в точности $p(s,r)$:

$$p(s,r) = \frac{2sr \cdot p(s,r-1) + s(s-1) \cdot p(s-2,r)}{(s+r)(s+r-1) - r(r-1)} \tag{16}$$

Задача M. Старые песни о главном – 4

Имя входного файла:	`m.in`
Имя выходного файла:	`m.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Во время трансляции концерта “Старые песни о главном – 4” предприниматель К. решил сделать бизнес на производстве дисков. Он имеет M дисков с длительностью звучания D каждый и хочет записать на них максимальное число песен. Эти песни (их общее количество N) передаются в порядке 1, 2, ..., N и имеют заранее известные длительности звучания $L(1)$, $L(2)$, ..., $L(N)$. Предприниматель может выполнять одно из следующих действий:

- записать очередную песню на диск (если она туда помещается) или пропустить её;
- если песня на диск не помещается, то можно пропустить песню или начать её записывать на новый диск. При этом старый диск откладывается и туда уже ничего не может быть записано.

Определить максимальное количество песен, которые предприниматель может записать на диски.

Формат входного файла

Первая строка содержит число M ($\leqslant 200$).
Вторая строка — D ($\leqslant 10^9$).
Третья строка — N ($\leqslant 500$).
Четвёртая строка — $L(1)$ $L(2)$ ... $L(N)$. Все $L(i) \leqslant D$.
Все входные параметры — натуральные числа.

Формат выходного файла

Необходимо вывести единственное число — искомое количество песен.

Пример

m.in	m.out
2 10 4 6 6 6 4	3

Разбор задачи M. Старые песни о главном - 4

На первом этапе вычислим массив m, где $m[i][j]$ — максимальное количество песен из диапазона от i-й по j-ю ($i \le j$), которые можно записать на один диск. Заполним массив по строкам, начиная с последней и перебирая элементы внутри каждой строки́ слева направо (от $j{=}i$ до $j{=}N$). Заполняя каждую строку, будем хранить множество длин песен, входящих сейчас в максимальный набор, и сумму этого множества. В начале заполнения строки множество пустое. При заполнении j-го элемента строки есть 3 варианта:

1. Текущую j-ю песню можно добавить в набор (суммарная длина не превысит размер диска) — так и делаем.
2. Текущую j-ю песню нельзя добавить в набор, но в текущем наборе есть песня, которая длиннее текущей. Заменим ту песню текущей, что уменьшит общую длину текущего набора.
3. Текущую j-ю песню нельзя добавить в набор и длина текущей песни больше или равна максимальной длины песни в наборе. В таком случае эту песню нужно пропустить.

Хранить текущий набор песен имеет смысл в очереди с приоритетами, т. к. из него нужно многократно вынимать максимум.

Вычислив массив m, используем его для ДП (второго этапа). Серия подзадач ДП — “Какое максимальное количество песен $H(i,j)$ можно записать на j дисков, использовав некоторые из песен от 1-й по i-ю?”. Решения тривиальных подзадач — $H(i,1){=}m[1][i]$. Уравнение ДП для нетривиальных имеет вид

$$H(i,j) = \max\left(m[1][i], \max_{1\le k<i}\left\{H(k,j{-}1) + m[k{+}1][i]\right\}\right). \qquad (17)$$

Задача N. Конфликт

Имя входного файла:	`n.in`
Имя выходного файла:	`n.out`
Ограничение по времени:	1 с
Ограничение по памяти:	128 Мб

С полярниками произошла обычная для исследователей история — о них забыли. Хорошо, что запасов продовольствия и топлива хватало на долгие годы. Так и работали — не смотря ни на что, день за днём. Однако начал проявляться фактор психологической совместимости: люди, вынужденные долго общаться в узком кругу, со временем иногда начинают не переносить друг друга. Психологи считают, что такое отношение всегда взаимно и всегда проявляется между парами людей, независимо от присутствия или отсутствия других. Для проведения очередной серии исследований полярникам нужно разделиться на две группы, которые некоторое время будут работать отдельно одна от другой. Помогите распределить всех полярников на две группы так, чтобы в составе каждой группы не было потенциально конфликтных пар, причём размеры групп должны быть максимально одинаковыми (т. е. разность количеств полярников в группах по модулю должна быть минимальна), обе группы должны оказаться не пустыми.

Формат входного файла

В первой строке записано количество полярников N ($3 \leqslant N \leqslant 50000$), затем количество потенциально конфликтных пар M ($0 \leqslant M \leqslant 70000$), затем сами пары, в которых полярники заданы их номерами от 1 до N.

Формат выходного файла

Программа должна вывести в первой строке 0 (если распределить полярников нужным образом невозможно) либо 1 (если возможно). Если распределение возможно, то вторая и третья строки должны содержать разделенные пробелами списки полярников, относящихся к соответствующей группе. Нужно позаботиться о том, чтобы размеры групп были максимально одинаковыми (т. е. разность количеств полярников в группах по модулю должна быть минимальна), обе группы должны оказаться не пустыми. Если правильных ответов несколько, выводите любой из них.

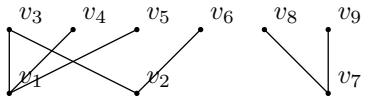
Пример

n.in	n.out
5 3 1 2 1 4 3 5	1 1 3 2 4 5
5 5 1 5 1 4 1 3 2 5 5 3	0

Разбор задачи N. Конфликт

Решение задачи состоит из двух этапов. На первом этапе существенно используется теория графов. Полярники — вершины, факты взаимной неприязни — рёбра (неориентированные). Запускаем поиск из какой-нибудь вершины. Что это будет за поиск (в ширину, в глубину или ещё какой) — неважно[8], главное — чтобы он отметил стартовую вершину, например, единичкой, а потом все вершины, соседние (соединённые ребром) с помеченными единичками, помечал двоечками, а соседние с помеченными двоечками — единичками. Если возникла хотя бы одна ситуация, что соседние вершины помечены одинаково, распределение по группам невозможно. Если не возникло — единички и двоечки задают распределение, кого в какую группу включить.

В вышеизложенном алгоритме нет и не может быть никаких оптимизаций. Поиск наилучшего приближения к равенству размеров групп происходит, когда вышеописанный граф состоит из нескольких (более чем одной) компонент связности, за счёт того, что внутри *некоторых* из компонент связности можно обменять местами доли.



Например, для графа, изображённого на рис., можно все нижние вершины (v_1, v_2, и v_7) назначить в одну долю, все верхние (v_3, v_4, v_5, v_6, v_8 и v_9) — в другую. А можно, "перевернув" правую компоненту связности, получить доли $\{v_1, v_2, v_8, v_9\}$ и $\{v_3, v_4, v_5, v_6, v_7\}$.

Отдельно взятая компонента связности, порождённая вершинами $\{v_1, v_2, v_3, v_4, v_5, v_6\}$, всегда заносит в одну из долей 2 вершины, в другую — 4 вершины. Иными словами, заносит в одну из долей на 2 вершины больше, чем в другую. В некотором смысле — прибавляет $4{-}2{=}2$ либо в одну долю, либо в другую.

Таким образом, второй этап решения данной задачи совпадает с задачей E, где роль размера отдельно взятой неделимой группы играет разность размеров долей отдельно взятой компоненты связности.

[8] рекурсивным поиском в глубину лучше всё же не пользоваться во избежание переполнения стека

Задача O. Русское лото

Имя входного файла:	`o.in`
Имя выходного файла:	`o.out`
Ограничение по времени:	2 с
Ограничение по памяти:	256 Мб

Для игры в “русское лото” используются карточки, удовлетворяющие условиям:

- карточка состоит из 3-х строк и N столбиков;

  K клеток в каждой строке заняты целыми числами (остальные свободны);

- в i-м столбце могут находиться только числа из диапазона $a_i \ldots b_i$;
- одно и то же число не может повторяться на одной и той же карточке.

Пример карточки:

8		25	30		57			81
3		26		43			79	85
	15	29	35		50	68		

Найти количество различных карточек.

Формат входного файла

В первой строке записано два числа — N и K ($3 \leqslant N \leqslant 9$, $1 < K < N$). В каждой из следующих N строк записано два целых неотрицательных числа a_i и b_i, не превышающие 1000. Гарантированно, что $a_i < b_i$, $b_i < a_{i+1}$ и $b_i - a_i < 20$.

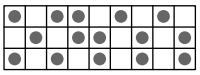
Формат выходного файла

Программа должна вывести одно число — искомое количество карточек.

Пример

o.in	o.out
9 5 1 9 10 19 20 29 30 39 40 49 50 59 60 69 70 79 80 90	817687046204481600000

Разбор задачи O. Русское лото



Временно представим, будто клеточки не содержат чи́сла, а просто заняты либо свободны (пример карточки см. справа). Тогда количество способов расставить метки (занять клеточки) внутри одной строки́ равно C_N^K, а по всем трём строкам $\left(C_N^K\right)^3$. При $N \leq 9$ это не очень много, и их можно перебрать.

Когда зафиксирован конкретный способ расстановки меток, легко сосчитать, сколько клеточек занято в 1-м, 2-м, ..., N-м столбиках — обозначим эти количества u_1, u_2, ..., u_N. Посчитаем количество возможных значений в каждом из столбиков: $d_i = b_i - a_i + 1$.

Тогда общее количество "настоящих" карточек с числами, соответствующих такой карточке с метками, равно $A(d_1, u_1) \cdot A(d_2, u_2) \cdot \ldots \cdot A(d_N, u_N)$, где $A(d, u)$, обычно обозначаемое A_d^u — количество размещений из d по u в смысле комбинаторики. Поскольку каждой карточке с числами соответствует только одна карточка с метками, сумма упомянутых произведений для всех возможных расстановок меток и будет ответом задачи.

Условие задачи допускает случай $b_i = a_i + 1$, т. е. $d_i = 2$. А среди всех возможных расстановок меток будут также и по три в одном столбике. Это можно пытаться как-то отбрасывать заранее. А можно просто полагать, что $A_2^3 = 0$.

Для указанных ограничений ответ не всегда будет помещаться в стандартные типы (в т. ч. `_int64`), поэтому нужна длинная арифметика. Точнее, инициализация длинного числа коротким, сложение длинных чисел и умножение длинного на короткое.

Наконец, как перебирать все возможные расстановки меток на карточке? Есть много способов, один из них — представить всё поле в

`vector`-е, содержащем три `vector`-а, каждый из которых содержит $N - K$ штук нулей и K штук единиц (изначально именно в таком порядке), и в трёх циклах (вложенных) применять к этим `vector`-ам функцию `next_permutation`.

Похоже, что рассмотренный тут способ практически во всех смыслах лучше рекурсивного, рассмотренного в книге Порублёв–Ставровский (даже после попыток грамотно применить к тому рекурсивному способу запоминания подзадач).

Задача P. Максимальное значение выражения

Имя входного файла:	`p.in`
Имя выходного файла:	`p.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Дано арифметическое выражение, состоящее из одноцифровых (от 0 до 9) чисел, между которыми поставлены знаки операций “`+`”, “`-`” и “`*`”. Нужно вставить в заданное выражение круглые скобки так, чтобы получить правильное арифметическое выражение, имеющее наибольшее возможное значение.

Формат входного файла

Вход содержит единственную строку — арифметическое выражение, состоящее из одноцифровых (от 0 до 9) чисел, между которыми поставлены знаки операций “`+`”, “`-`” и “`*`”. Длина строки не превышает 555 символов.

Формат выходного файла

Первая строка — максимальное значение выражения.

Вторая строка — способ расстановки круглых скобок, при котором достигается максимальное значение выражения.

Расстановка скобок должна удовлетворять таким условиям:

- отдельно взятое число нельзя брать в скобки;
- всё выражение в целом нельзя брать в скобки;
- любое другое подвыражение необходимо брать в скобки, даже если порядок вообще-то не важен (например, “`(2+2)+2`”) или определяется приоритетами (например, “`(2*2)-2`”).

Если существуют различные расстановки скобок, при которых достигается максимальное значение выражения, выводите любую одну из них.

Ответ будет засчитан, если относительные погрешности и числового ответа (1-й строки), и значения выражения из 2-й строки не превышают 10^{-9}.

Пример

p.in	p.out
1+2-3*4	0 ((1+2)-3)*4

Разбор задачи P. Максимальное значение выражения

Задача, в основном, разобрана на лекции. Главный неочевидный момент — для обратного хода удобно хранить и то k, при котором достигнут max или min, и, для произведений, какой из четырёх вариантов $EMax(i,k-1)\times EMax(k,j)$,$EMax(i,k-1)\times EMin(k,j)$, $EMin(i,k-1)\times EMax(k,j)$ или $EMin(i,k-1)\times EMin(k,j)$ был выбран. А раз это надо сделать для произведений, то ради единообразия можно сделать и для других действий.

Задача Q. Путь через горы 2

Имя входного файла:	q.in
Имя выходного файла:	q.out
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Поверхность Земли в горной местности можно представить в виде ломаной линии. Вершины ломаной расположены в точках (x_1, y_1), (x_2, y_2), ..., (x_N, y_N), при этом $x_i < x_{i+1}$. Обычный горный маг находится в точке $(x_1,\ y_1)$ и очень хочет попасть в точку (x_N, y_N). При этом он может перемещаться только пешком. Он может ходить по поверхности Земли (т. е. вдоль ломаной). А может сотворить в воздухе мост и пройти по нему. Мост может соединять две вершины ломаной: мост не может начинаться и заканчиваться не в вершине ломаной, и мост не может проходить под землей (в т. ч. не может быть туннелем в горе), но мост может каким-то своим участком проходить по поверхности земли. Длина моста не может быть больше R. Маг может построить мосты суммарной длиной не более K. После прохождения моста, он (мост) растворяется в воздухе. Какое наименьшее расстояние придётся пройти магу, чтобы оказаться в точке (x_N, y_N)?

Формат входного файла

Программа должна прочитать сначала натуральное число N ($2 \leqslant N \leqslant 64$); затем целое число R ($0 \leqslant R \leqslant 10000$) — максимальную возможную длину одного моста; далее натуральное число S ($R \leqslant S \leqslant 10000$) — максимальную суммарную длину мостов. Далее координаты (x_1, y_1), (x_2, y_2), ..., (x_N, y_N). Все координаты — целые числа, не превышающие по модулю 10000, для всех i от 1 до $N-1$ выполняется $x_i < x_{i+1}$.

Формат выходного файла

Программа должна вывести одно число — минимальную длину пути, которую придётся пройти магу (как по земле, так и по мостам). Ответ выведите с точностью 5 цифр после десятичной точки.

Пример

q.in	q.out
5 2 2 0 0 2 2 3 -1 4 1 5 0	9.64099

Разбор задачи Q. Путь через горы 2

Будем характеризовать рёбра и пути вместо длины (одного числа) парой чисел: общей длиной `totd` и магической длиной `magd`. При этом `totd` любого отрезка равна его геометрической длине; для прохождений по земле `magd` $= 0$, для мостов `magd` $=$ `totd`.

Оценкой расстояния до вершины (аналогом длины кратчайшего из известных путей к этой вершине) будем считать набор всех разных пар $\langle$`totd`, `magd`$\rangle$, минимальных[9] по отношению $A \preceq B \overset{\text{def}}{\Longleftrightarrow} (A.\texttt{totd} \leqslant B.\texttt{totd}) \wedge (A.\texttt{magd} \leqslant B.\texttt{magd})$. Иными словами, пара хранится $\Leftrightarrow$ её нельзя заменить на гарантированно лучшую пару (с не бóльшими и `totd`, и `magd`). Разумеется, пары, в которых `magd` превышает заданный лимит S, хранить бессмысленно (даже если они минимальны).

В *данной* версии задачи гарантированно невыгодно возвращаться, т. е. делать какие бы то ни было ходы справа налево (строя мосты или идя вдоль рельефа). Докажем это.

[9] элемент минимален, если нет ни одного элемента, меньшего, чем он; для нелинейных отношений порядка минимальными могут быть сразу несколько элементов

Предположим, будто существует минимальный маршрут, содержащий возвраты. Тогда в нём существует вершина B, в которую приходят слева и уходят влево же. Прийти вдоль рельефа слева и уйти вдоль рельефа налево бессмысленно (можно вырезать пройденное дважды подряд звено ломаной, так что такой маршрут не минимален). Следовательно, B — один из концов магического моста. Рассмотрим этот мост; причём, если таких мостов два (и приходят, и уходят по мостам), рассмотрим верхний из них. (Для любителей абсолютной строгости скажем, что если провести вертикальную прямую $x = B.x - \varepsilon$, то верхним будет тот из отрезков, чья точка пересечения с этой прямой имеет бóльшую y-координату.) Обозначим другой конец рассматриваемого моста как A (неважно, приходят ли в B слева по мосту AB или уходят из B налево по мосту BA).

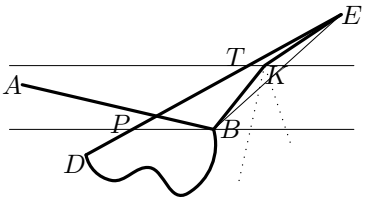
Обозначим вершину, в которую пошли после AB (из которой пришли перед BA) как C. Нетрудно убедиться, что C находится ниже отрезка AB и что $A.x < C.x < B.x$. Пока будут происходить дальнейшие переходы (по мостам или вдоль рельефа) *внутри* диапазона от A до B, оба эти соотношения будут сохраняться. Минимальный маршрут не может содержать повторов вершин (иначе фрагмент между посещениями одной и той же вершины можно было бы вырезать). Следовательно, должна быть некоторая вершина D, удовлетворяющая тем же двум условиям и являющаяся концом моста, выводящего за пределы диапазона от A до B.

Следовательно, из допущения о возможности минимальных маршрутов, содержащих возвраты, следует, что такой маршрут обязательно должен иметь один из двух видов, изображённых на рис.: имеются мосты AB и DE, и при этом либо $A.x < D.x < B.x < E.x$, либо $E.x < A.x < D.x < B.x$.

Часть между B и D нарисована волнисто — она может быть хоть одним отрезком ломаной, хоть последовательностью звеньев ломаной, хоть содержать как звенья ломаной, так и мосты. Эти подробности нас сейчас не интересуют. Не интересует даже, действительно ли весь путь от B до D идёт в одном и том же направлении или внутри него есть свои изменения направления.

Проанализируем эти случаи подробнее, начав с более правдоподобного $A.x < D.x < B.x < E.x$. Рассмотрим его подслучай $D.y \leqslant B.y$. Проведём через B горизонтальную прямую, обозначим точку её пересечения с DE

как P. Покажем, что всегда можно построить путь из B в E, лучший[10], чем рассматриваемый (через D).



Если бы всегда можно было построить мост BE, доказательство было бы тривиальным: для непосредственного перехода $B \to E$, и `magd` и `totd` равны длине моста $|BE|$, а для перехода через D и `magd` и `totd` *включают* в себя *как составляющую* часть PE моста DE. А $|PE| > |BE|$, т. к. $\angle PBE$ тупой.

Ломаная может содержать некую вершину K, такую, что построить мост DE можно, а BE — нет. Однако в *данной* задаче это можно учесть и "спасти" рассуждение предыдущего абзаца. Просто построим не мост BE, а два моста: BK и KE, и отдельно получим $|TE| > |KE|$, отдельно $|PT| > |BK|$ (где T — точка пересечения горизонтальной прямой, проведенной через K, с отрезком DE).

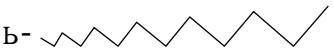
Если же "мешает" не одна вершина K, а много, надо построить несколько мостов вдоль верхней выпуклой оболочки всех этих "мешающих" вершин; доказательство аналогично.

Мы завершили доказательство, что возвращающийся обратно маршрут не минимален. Это было доказательство для одного из случаев, но для остальных случаев всё или аналогично, или проще. Если $A.x < D.x < B.x < E.x$, но $D.y > B.y$, то вместо того, чтоб заменять фрагмент пути $B \to D \to E$ на новые мосты, заменим фрагмент $A \to B \to D$. Если же $E.x < A.x < D.x < B.x$, то можно, например, проложить новые мосты вдоль выпуклой оболочки между A и E, заменив на них весь рассматриваемый фрагмент (то ли $A \to B \to D \to E$, то ли $E \to D \to B \to A$).

Отметим, что если концы маршрута могут не совпадать с концами ломаной, то и доказательство, и само доказываемое утверждение неверны. Например, при ограничении на длину моста 4, ломаной $P_1 = (0, 4)$, $P_2 = (1, -1000)$, $P_3 = (2, 0)$, $P_4 = (3, 3)$, старте P_1 и финише P_3, при некоторых ограничениях на S оптимальным окажется маршрут $P_1 \to P_4 \to P_3$.

Итак, в общих чертах алгоритм решения данной задачи может выглядеть так: пройти по всем вершинам ломаной от 2-й до последней, вычисляя для каждой вершины набор всех минимальных в вышеописанном смысле оценок. Когда этот набор будет найден для последней вершины — найти в этом наборе пару с максимальным не превышающим S значением `magd`. (Из определения конкретно данного отношения порядка и определения минимальности следует: эта пара будет иметь минимальную `totd` среди всех допустимых.)

[10] по всё тому же нелинейному отношению порядка, т. е. с меньшими и `totd`, и `magd` одновременно



Увы, это ещё не всё: хотя для подавляющего большинства входных данных описанный алгоритм работает достаточно быстро, на некоторых специально сконструированных рельефах (пример на рис. справа) количество различных минимальных путей к последней вершине может достигать почти $2^{N/2}$, что для $N \approx 60$ недопустимо. Но хранить несколько копий структур размера $N \cdot 2^{N/2}$ при $N \leqslant 32$ ещё вполне допустимо, так что ситуацию можно спасти с помощью такого приёма.

Организуем вычисления по принципу “для данной вершины учтём, каким образом её набор минимальных оценок влияет на наборы минимальных оценок последующих вершин (соседней и тех, куда можно провести мост)”. Например, при существовании мостов 1–3, 3–5, 1–5 и 1–7, сначала (при обработке 1-й вершины) будет учтено, каким образом мосты 1–3, 1–5 и 1–7 влияют на наборы минимальных оценок 3-й, 5-й и 7-й вершин, и только потом (при обработке 3-й вершины) будет учтено влияние моста 3–5 на набор 5-й вершины.

Проведём такие вычисления только для “левой половины” ($i \leqslant N/2$). По построению, при этом будут найдены полные наборы минимальных оценок для $i \leqslant N/2$, а для $i > N/2$ будут найдены минимальные среди тех путей, которые на последнем шаге приходят[11] из левой половины. И построим наборы минимальных “встречных” оценок, т. е. считая от финиша ($i{=}N$).

После этого можно для каждой вершины провести поиск такой пары прямой минимальной оценки и встречной минимальной оценки, для которой сумма `magd` не превышает лимит S, а сумма `totd` минимальна среди всех аналогичных сумм. Среди этих минимумов для вершин следует найти всеобщий минимум — он и будет окончательным ответом. (Перечень вершин, для которых следует провести слияние, можно сократить, взяв лишь среднюю, т. е. на границе левой и правой половин, а также те, для которых существует хотя бы один мост, другой конец которого находится в другой половине.)

Идеологически алгоритм описан, но как технически эффективно работать с такими наборами минимальных оценок? Один из способов — использовать `map<double, double>`, где 1-й элемент (ключ) — `magd`, 2-й (значение) — `totd`. АТД `map` автоматически упорядочивает пары по строгому возрастанию ключа (`magd`). Чтоб не работать с пустыми `map`-ами, изначально запишем для каждой вершины способ “дойти вдоль ломаной, не строя мостов”.

Пусть набор мин. оценок хранится в `map<double, double> E`,

[11] обычно по мосту, исключения только для крайней левой точки правой половины

оценка нового пути — в экземпляре `A` структуры с полями `totd` и `magd`. Тогда проверку, минимальна ли `A`, можно провести так: `it = E.upper_bound(A.magd+EPS)` найдёт итератор на первый элемент набора, чей ключ строго больше `A.magd`. Делаем `it--` и получаем итератор на ближайшую снизу к `A.magd` пару набора.[12] Если `it->second<=A.magd`, значит новая пара `A` не лучше уже включённой в набор, и `A` надо "забыть". Иначе, `A` минимальна, и надо проверить, не лучше ли она некоторых уже хранящихся в `map`-е пар (если лучше, те пары надо исключить). Последовательность минимальных пар, упорядоченная по возрастанию `magd`, одновременно упорядочена по убыванию `totd` (это не гарантируется `map`-ом, мы сами поддерживаем это свойство).

Благодаря этому, достаточно с помощью `if(it->first < A.magd-EPS) it++;` установить `it` на первую пару, чей ключ больше-равен `A.magd`, сделать копию итератора `it2=it;` и увеличивать `it2`, пока не дошли до конца `E` и пока `it2->second > A.totd+EPS`. Таким образом, мы узнаём границы удаляемого диапазона, после чего вызываем `erase`.

Задача R. Шахматы

Имя входного файла:	`r.in`
Имя выходного файла:	`r.out`
Ограничение по времени:	3 с
Ограничение по памяти:	256 Мб

Напомним некоторые существенные для данной задачи стандартные правила игры в шахматы. Играют два игрока, один играет белыми фигурами, другой чёрными. Игра происходит на доске 8×8, столбики обозначаются буквами от "a" до "h" слева направо, строки — цифрами от 1 до 8 снизу вверх. Каждая клетка доски или пустая, или содержит одну фигуру. Если фигура A (не пешка) может походить согласно правилам на клетку, занятую чужой фигурой B, то вследствие такого хода фигуру B бьют, т. е. снимают с доски. Поэтому обо всех клетках, куда некоторая фигура может походить, говорят, что они находятся "под боем" данной фигуры. Королю запрещено ходить в клетку, которая находится под боем любой чужой фигуры. Если один из игроков сделал такой ход, что король противника оказался под боем (это называют "шах"), противник обязан

[12] путь вдоль рельефа (`magd` $= 0$) уже включён в `E`, а при вызове `upper_bound` мы прибавили `EPS` (например, `1e-9`), поэтому `it` будет показывать куда-то дальше, чем `E.begin()`, и `it--` не выведет за пределы разумных значений

ответить таким ходом, чтобы его король уже не был под боем. Если такого хода не существует, то это называют “мат”.

Король может ходить на одну клетку в любом из 8-ми направлений (влево, вправо, вперед, назад, в любом направлении по любой диагонали). Ферзь может ходить в любом из 8-и направлений на любое количество клеток, но не пересекая клеток, занятых фигурами.

Пусть на шахматной доске находится три фигуры: белый король, белый ферзь и чёрный король. Сейчас ход белых. За какое минимальное количество ходов они гарантированно смогут поставить мат? Чёрные будут делать всё, допустимое правилами игры, чтобы избежать мата.

Формат входного файла

Программа должна прочитать число $TEST\_NUM$ — количество тестовых блоков, потом сами блоки. Каждый блок является отдельной строкой, в которой записаны три обозначения клеток, где находятся белый король, белый ферзь и чёрный король соответственно, разделенные пробелом (обозначение клетки состоит из записанных слитно обозначения столбца и номера строки).

Все заданные позиции гарантированно допустимы с точки зрения шахматных правил (например, чёрный король не под боем).

Формат выходного файла

Ваша программа должна вывести для каждого теста единственное число — минимальное количество ходов.

Пример

r.in	r.out
2 a3 b3 a1 a3 e3 b1	1 2

Разбор задачи R. Шахматы

Возможно, существуют (а может, и нет) более эффективные решения, опирающиеся на знания теории шахмат. Но мы рассмотрим решение, не требующее узкоспециализированных знаний.

Методы, обычно используемые в шахматных программах, и сложны́, и используют совместно с деревом ходов эвристические оценки позиций, так что нет уверенности, найдут ли они именно минимальное количество ходов. К задаче вряд ли примени́мо ДП.

Если считать параметром ДП позицию игры, возникнет ситуация, что из позиции А можно перейти за два полухода[13] в позицию Б, а из Б за два полухода — в А, и чтоб вычислить ответ для А, надо знать ответ для Б, а чтоб вычислить для Б, надо знать для А.

Если бороться с такими зацикливаниями, запоминая, какие позиции уже рассмотрены (например, аналогично поиску в глубину), оценка зависит не только от само́й позиции, но и от предыдущих — следовательно, позиция игры опять не может быть параметром ДП. Если же считать параметром ДП и позицию, и множество всех предыдущих позиций, то подзадач слишком много. Добавляет сложностей и то, что пат (король не под боем, ходов нет) — *не* проигрыш.

Так что задача особенна не столько сложностью (она умеренна), сколько провоцированием неправильных подходов к решению.

Закодируем позиции 19-битовыми числами: один бит — чей ход, трижды по шесть бит — координаты на доске каждой из трёх фигур.

Таких кодов, включая и коды невозможных позиций, 2^{19} (примерно полмиллиона), что относительно немного. Сначала рассортируем их по трём группам: маты (ход чёрных, король чёрных под боем, ходить некуда), невозможные (более одной фигуры в одной клеточке; короли на соседних клеточках; ход белых, а чёрный король под боем белого ферзя) и все остальные позиции.

Пройдя по перечню всех матов, найдём все *однополуходовки*, т. е. позиции, из которых за один полуход (белых) *можно* прийти в мат. Затем, зная о каждой позиции, является ли она однополуходовкой, пересмотрев все “остальные” позиции, выберем из них *двухполуходовки*, т. е. позиции, из которых *любой* допустимый правилами ход чёрных ведёт в однополуходовку. Таким образом, если позиция — двухполуходовка, то после двух полуходов (сначала чёрных, потом белых) чёрным поставят мат (при правильной игре белых, для любой допустимой игры чёрных).

Далее повторяем аналогичные действия с небольшим отличием. Пройдя по перечню всех 2-полуходовок, найдём все *3-полуходовки*, т. е. позиции, из которых за один полуход (белых) можно прийти в 2-полуходовку. При этом 1-полуходовки следует пропускать — если белые могут поставить мат за один полуход, незачем растягивать игру на три полухода. Пересмотрев все “остальные” позиции, выберем из них *4-полуходовки*, т. е. позиции, из которых *любой* допустимый правилами ход (чёрных) ведёт либо в 1-полуходовку, либо в 3-полуходовку, причём хотя бы один ход — в 3-полуходовку. Таким образом, если позиция — 4-полуходовка, то при правильной игре обеих сторон после четырёх полуходов (чёрных, белых,

[13]полуход — ход одной из сторон (белых или чёрных)

чёрных, белых) чёрным поставят мат. Если белые будут играть неправильно, мат может наступить позже или не наступить вообще; если чёрные — мат может наступить уже через два полухода.

Дальнейшие действия (зная $2k$-полуходовки, строим $(2k{+}1)$- и $(2k{+}2)$-полуходовки) полностью аналогичны. Процесс обрывается, когда соответствующих полуходовок нет (реально, 20-полуходовки ещё есть, а 21-полуходовок уже нет).

Чтобы быстро определять, является ли позиция 1-полуходовкой, 2-полуходовкой, ... будем поддерживать массив `short a[1<<19]` (т. е. с диапазоном индексов от 0 до $2^{19}{-}1$), значения которого будут как-то кодировать ситуации “невозможная”, “мат”, “1-полуходовка”, “2-полуходовка”, ..., “ещё не известно”.

Этот алгоритм выполняет довольно много действий; грубая верхняя оценка — $P{\cdot}D{\cdot}M$, где P — количество позиций ($\approx 2^{19}$), D — максимальное количество полуходов (20), M — среднее количество допустимых из позиции ходов (≈ 10). Но эти действия мало зависят от количества позиций, для которых надо решить задачу (задача решается для всех позиций, и остаётся лишь брать готовые ответы).

При неправильной игре, белые могут потерять ферзя (подставив его под бой чёрного короля в клеточке, не соседней со своим королём), после чего уже не смогут поставить мат. Это надо учесть при отборе позиций-матов и невозможных позиций.

День шестой (24.02.2010г). Контест Дмитрия Кордубана

Об авторе...

Кордубан Дмитрий Александрович, студент 6 курса специальности “системный анализ и управление” Национального технического университета Украины “КПИ”. С 2006 года заместитель главы жюри Киевской городской олимпиады школьников по информатике.

Научные интересы:

- распознавание образов;
- машинное обучение.

Основные достижения:

- участник финала ACM ICPC 2009 (34 место);
- победитель Всеукраинской студенческой олимпиады 2009.

Теоретический материал. Введение в суффиксные массивы

Мотивация

Основой всех строковых задач является задача поиска точного вхождения подстроки в строку. Возникающие при её решении объекты — префикс-функция, Z-функция (или основной препроцессинг согласно [5]) — хорошо описывают совпадения самой строки со всеми её суффиксами. Однако на практике часто возникают задачи, эффективное решение которых требует более глубокого анализа внутреннего строения строки. Приведем некоторые из них.

Задача 1. Для заданной строки $S = s_1 s_2 \dots s_n$ лексикографически отсортировать все ее циклические сдвиги $s_i \dots s_n s_1 \dots s_{i-1}$.

Задача 2. Для заданных строк $S = s_1 s_2 \dots s_n$ и $T = t_1 t_2 \dots t_m$ найти их наибольшую общую подстроку.

Задача 3. Задан текст $S = s_1 s_2 \dots s_n$. Необходимо предобработать его за разумное время так, чтобы затем эффективно искать вхождения произвольного образца $P = p_1 p_2 \dots p_m$ в S. В отличии от задачи словарного поиска, где заранее задано множество образцов P_i, здесь запросы на поиск поступают один за другим.

Задача 1 лежит в основе преобразования Берроуза-Уилера, используемого для сжатия данных. Её наивное решение с помощью общих методов сортировки сравнениями требует $\Omega(n^2 \log n)$ времени в худшем случае. Использование поразрядной сортировки позволяет уменьшить оценку до $\Omega(n^2))$, что по-прежнему может быть неприемлемо на практике.

Задача 2 является типичным представителем задач, возникающих в вычислительной биологии [5]. Ее наивное решение с помощью линейных алгоритмов поиска подстроки требует по крайней мере $\Omega(nm)$ времени.

Задача 3 может возникать при разработке базы данных, допускающей поиск точных вхождений очень длинных строк. Одним из примеров является база данных для геномных ДНК [5]. Очевидное решение с использованием алгоритмов поиска подстроки требует по крайней мере $\Omega(n + m)$ времени. Это слишком долго для типичных приложений, где размер текста может достигать гигабайт при сравнительно небольшой длине образцов.

Суффиксные массивы

Для любой строки $S = s_1 s_2 \dots s_n$ обозначим через S_i её суффикс $s_i s_{i+1} \dots s_n$. Очевидно, $S = S_1$.

Суффиксным массивом строки S называется последовательность индексов $a_1, a_2, \dots, a_n$ всех её суффиксов, упорядоченных лексикографически. Таким образом, S_{a_1} есть лексикографически наименьший суффикс, S_{a_2} — второй по величине, S_{a_n} — наибольший.

Отметим, что задача построения суффиксного массива и задача 1 практически эквивалентны. В самом деле, если мы построим суффиксный массив для строки SS и вычеркнем из него все суффиксы длиной не больше n, то оставшиеся суффиксы будут соответствовать решению задачи 1. Если же мы умеем быстро решать задачу 1, то можем дополнить строку S специальным символом \$, лексикографически меньшим всех остальных. Тогда упорядоченная последовательность циклических сдвигов строки $S\$$ практически совпадает с суффиксным массивом S.

Покажем, как эффективно решать задачу 1. Для удобства изложения “зациклим” все рассматриваемые строки, отождествив между собой индексы с одинаковыми остатками по модулю n. Иными словами, вместо

конечной строки S будем работать с бесконечной строкой вида $SSS\ldots$. Поскольку длина всех строк равна n, это не повлияет на их лексикографический порядок.

Определим на строках отношение h-порядка $\leq_h$:

$$S \leq_h T \iff s_1 s_2 \ldots s_h \leq t_1 t_2 \ldots t_h (\text{лексикографически})$$

Аналогично определим отношения $=_h, <_h$ и их дополнения.

Утверждение 1. $\leq_h$ — отношение порядка.

Утверждение 2. Если $h \geq n$, то h-порядок для циклических строк длины n совпадает с лексикографическим:

Утверждение 3.

$$S \leq_{2h} T \iff \begin{cases} S <_h T & \text{, либо} \\ (S =_h T) \wedge (S_h \leq_h T_h) \end{cases}$$

Последнее утверждение содержит ключевую идею нашего метода. Из него следует, что имея h-порядок на множестве циклических сдвигов S легко получить его $2h$-порядок. Мы можем вычислить отношение $\leq_{2h}$, сделав не более 2 обращений к отношению $\leq_h$, т.е. за время $O(1)$ при должной реализации. Это приводит к алгоритму для решения задачи 1:

1. Вычислим 1-порядок всех циклических сдвигов S с помощью непосредственной сортировки. Положим $h = 1$.
2. Пока $h < n$, будем выполнять следующее: преобразуем h-порядок в $2h$-порядок и удвоим h.

Временная сложность первого шага равна $O(n \log n)$ при использовании сортировки сравнениями или $O(n + |\Sigma|)$ для сортировки подсчетом. Количество итераций на втором шаге равно $\Theta(\log n)$. Корректность следует из утверждения 2.

Внутренний цикл на втором шаге можно реализовать по-разному, но один простой способ очевиден: использовать сортировку сравнениями за $O(n \log n)$. Отсюда немедленно получаем общую оценку времени работы $O(n(\log n)^2)$, что намного лучше $\Omega(n^2)$. При использовании библиотечной сортировки с пользовательским компаратором такая реализация оказывается достаточно компактной, а после применения серии эвристик — вполне практичной.

Более того, мы можем преобразовать h-порядок в $2h$-порядок для множества циклических сдвигов за $O(n)$ [11]. Для этого воспользуемся идеей цифровой сортировки. Пусть нам дан h-порядок в виде массива индексов.

Пусть также нам дано разбиение всех циклических сдвигов на классы эквивалентности по отношению $=_h$. Ясно, что после $2h$-сортировки поменяться местами могут только h-эквивалентные строки. Выделим память под новый массив индексов и запомним в нем позиции начала блоков, соответствующих различным классам эквивалентности. Это можно сделать с помощью массива указателей. Затем пройдемся по циклическим сдвигам в порядке h-неубывания. Каждый раз, когда мы встречаем сдвиг S_i, поместим сдвиг S_{i-h} на первое место в соответствующем блоке и сдвинем указатель на начало блока. Обоснование этого действия простое: среди всех сдвигов в классе h-эквивалентности $\{S_{i-h}\}$ наименьшим из еще не рассмотренных будет именно S_{i-h}, т.к. сдвиг S_i наименьший из еще не рассмотренных и справедливо утверждение 3.

Для использования этого метода нам необходимо поддерживать разбиение на классы h-эквивалентности на каждой итерации. Его легко вычислить на 1 шаге и затем поддерживать после каждого шага за $O(n)$, опираясь на аналог утверждения 3 для отношения эквивалентности. Таким образом, общее время работы алгоритма составит $O(n \log n)$.

Как было сказано ранее, оба алгоритма тривиально превращаются в алгоритмы построения суффиксного массива.

С практической точки зрения, реализация обоих методов требует примерно одинаковых усилий, хотя первый идейно проще. В результете экспериментов автором установлено, что метод со сложностью $O(n(\log n)^2)$ работает примерно в 2 раза медленнее метода со сложностью $O(n \log n)$ на строках длиной до $5 \cdot 10^5$ символов при использовании следующих двух эвристик:

1. На каждой итерации сортировать не все строки вместе, а каждый класс эквивалентности по отдельности. Это упрощает вид компаратора и ускоряет работу на большинстве тестов.

2. Прекращать работу, когда число классов эквивалентости достигло n. Эта эвристика использовалась и во втором методе.

Несмотря ни на что, строка вида $S = aa \ldots a$ по-прежнему остается худшим случаем для обоих методов. Поэтому выбор реализации на соревновании остается на совести участника.

Наибольшие общие префиксы

Перейдем теперь к решению задачи 2. Для этого кроме суффиксного массива нам понадобиться еще кое-что.

Массивом LCP строки S называется последовательность чисел $lcp_1, lcp_2, \ldots, lcp_{n-1}$ такая, что lcp_i равен длине наибольшего общего префикса суффиксов S_{a_i} и $S_{a_{i+1}}$. Здесь $a_1, a_2, \ldots, a_n$ — суффиксный массив строки S.

Оказывается, что массив LCP вместе с суффиксным массивом содержат всю важную информацию о внутренней структуре S, занимая всего лишь $O(n)$ памяти. Это иллюстрирует следующее основное

Утверждение 4. Для любых двух суффиксов S_{a_i} и S_{a_j} (без потери общности $i < j$) длина их наибольшего общего префикса равна:

$$LCP(S_{a_i}, S_{a_j}) = \min\{lcp_i, lcp_{i+1}, \ldots, lcp_{j-1}\}$$

Предположим, что мы умеем быстро вычислять массив LCP. Как использовать эту информацию для решения задачи 2? Найдем суффиксный массив и массив LCP для строки $C = S\$T$, где \$ означает специальный символ-разделитель, не встречающийся ни в одной из строк. Из-за этого все суффиксы C распадаются на 2 класса: суффиксы S и суффиксы T (не считая неинтересного нам суффикса $\$T$). Очевидно следующее

Утверждение 5. Если строка P является подстрокой S, то существуют такой суффикс S_i, что P является его префиксом.

Из этого следует следующее решение задачи 2: перебрать все пары суффиксов S_i и T_j и для каждой пары найти наибольший общий префикс, используя утверждение 4 и имеющиеся массивы для строки C. Наибольшее из найденных значений и будет ответом задачи. Однако, такой способ требует $\Omega(nm)$ вычислений минимума в массиве LCP, что не дает никакого преимущества по сравнению с наивным подходом.

Но можно заметить, что искомый максимум достигается на множестве *соседних в суффиксном массиве* C суффиксов S_i и T_j, т.е. среди пар суффиксов вида $C_{a_k}, C_{a_{k+1}}$, где $(a_k \leq n) \wedge (a_{k+1} > n+1)$ или наоборот $(a_k > n+1) \wedge (a_{k+1} \leq n)$. Это следует из того же утверждения 4. Просмотреть такое множество можно за один линейный проход по массиву LCP, так что задача сводится к его нахождению.

Покажем, как найти массив LCP S за $O(n)$, используя суффиксный массив S $a_1, a_2, \ldots, a_n$. Будем вычислять lcp_i не в порядке увеличения i, а в порядке увеличения номеров соответствующих суффиксов. Рассмотрим суффикс S_1. В общем случае он не наименьший, так что найдем его предшественника в суффиксном массиве S_y (т.е. $\exists\, i{:}a_i{=}y, a_{i+1}{=}1$). Найдем lcp_i – наибольший общий префикс S_1 и S_y непосредственным сравнением символов, начиная с первого. Затем перейдем к суффиксам S_2 и его предшественнику S_z ($\exists\, i' : a_{i'} = z, a_{i'+1} = 2$). Понятно, что если $lcp_i > 0$, то длина их наибольшего общего префикса равна по крайней мере $lcp_i - 1$. Поэтому найдем lcp'_i непосредственным сравнением начиная с символа $lcp_i + 1$, а не

со 2-го символа. Повторяя этот процесс, вычислим все lcp. Единственный особый случай — когда очередной суффикс оказывается наименьшим и не имеет предшественника. Тогда ничего не делаем, а на следующем шаге непосредственно сравниваем символы, начиная с первого.

Онлайновый поиск подстрок

Наконец мы знаем достаточно для того, чтобы эффективно решать задачу 3. Именно она послужила стимулом для разработки суффиксных массивов [11].

Если бы множество образцов было известно заранее, всё было бы несколько проще. Задача поиска вхождений образцов из заданного словаря решается алгоритмом Ахо-Корасик за оптимальное время $O(n+\sum m_i)$. Как правило, на соревнованиях все входные данные известны заранее и можно решать оффлайн-версию задачи. Требование онлайновости качественно ее усложняет.

Пусть, как всегда, нам уже дан суффиксный массив. Воспользуемся утверждением 5 и тем фактом, что суффиксы упорядочены по невозрастанию. Это позволяет применить двоичный поиск, чтобы найти суффикс $S_i =_m P$ или установить его отсутствие. Количество итераций будет равно $\Theta(\log n)$. Если сравнивать строки непосредственно, то в худшем случае это займет линейное от длины P время. Поэтому общая оценка равна $O(m \log n)$. Такой метод может быть как лучше, так и хуже наивного со временем работы $O(n + m)$ в зависимости от соотношения между n и m, но на практике $n \gg m$ и выигрыш заметен. Более того, часто средняя величина наибольшего общего префикса P и S_i невелика (например, в естественных тестах), что тоже ускоряет поиск.

Покажем, как реализовать поиск за $O(m + \log n)$. Рассмотрим подробнее каждую итерацию двоичного поиска. Пусть на какой-то итерации мы сузили диапазон поиска до промежутка от S_{a_L} до S_{a_R}, и хотим сравнить P с суффиксом S_{a_M} где $M = (L + R)/2$. Пусть также $l = LCP(S_{a_L}, P)$, $r = LCP(S_{a_R}, P)$. Пусть без потери общности $r \leq l$. Тогда возможны 3 случая в зависимости от значения $LCP(S_{a_L}, S_{a_M})$:

1. $LCP(S_{a_L}, S_{a_M}) > l$. Это значит, что $S_{a_M} =_{l+1} S_{a_L} \neq_{l+1} P$, а значит P может лежать только в правой части диапазона поиска. При этом величина l остается неизменной.

2. $LCP(S_{a_L}, S_{a_M}) = l$. В этом случае мы знаем, что первые l символов P и S_{a_M} совпадают. Поэтому достаточно провести сравнение символов $l+1, l+2$ и так далее, пока не будет найдено вхождение P в S_{a_M} или различие в символе $l + k$. В последнем случае мы сужаем диапазон

поиска до нужной части в зависимости от знака неравенства. Также мы узнаем новое значение l или r — оно равно $l + k - 1$.

3. $LCP(S_{a_L}, S_{a_M}) < l$. Поскольку $P =_l S_{a_L}$ и $P \neq_l S_{a_M}$, очевидно что P может лежать только в левой части диапазона поиска. При этом новое значение r равно $LCP(S_{a_L}, S_{a_M})$.

Проанализируем этот алгоритм. Начальные значения l и r можно найти за $O(m)$ непосредственно. В основном цикле будет выполнено не более m успешных сравнений символов P с символами S, поскольку мы никогда не сравниваем один символ успешно дважды. Кроме того, количество неуспешных сравнений не превосходит 1 на каждой итерации. Поэтому суммарное время работы равно $O(m + \log n)$ при условии, что мы находим значения $LCP(S_{a_i}, S_{a_j})$ за $O(1)$.

Осталось показать способ нахождения $LCP(S_{a_i}, S_{a_j})$ за $O(1)$. Согласно утверждению 4, эта задача сводится к известной задаче нахождения минимума на отрезке (Range Min Query) в массиве LCP. Покажем простой алгоритм ее решения со временем предобработки $O(n \log n)$.

Пусть задана последовательность $a_1, a_2, \ldots, a_n$. Вычислим значения:

$$b_i^k = \min_{i \le j < i+2^k} a_j$$

для всех $i, k : 1 \le i < i+2^k \le n$. Всего таких пар значений будет $O(n \log n)$, и мы можем вычислить их одно за другим в порядке увеличения k за время $O(n \log n)$ пользуясь тождеством:

$$b_i^{k+1} = \min\{\, b_i^k, b_{i+2^k}^k \,\}$$

Тогда любой запрос на минимум в a сводится к двум обращениям к таблице b_i^k:

$$\min_{l \le i < r} a_i = \min\{\, b_l^k, b_{r-2^k}^k \,\} \quad \text{при } k : 2^k \le r - l < 2^{k+1}$$

На самом деле, при решении задачи онлайнового поиска можно существенно сэкономить память, поскольку нас интересуют минимум не на всех отрезках, а только на отрезках вида $[l, m]$ или $[m, r]$ где l, r — пара границ в двоичном поиске. Всего таких пар будет $O(n)$, так что мы можем хранить информацию о минимумах в $O(n)$ памяти. Подробнее смотри основополагающую работу [11].

Заключение

Суффиксные массивы обеспечивают оптимальный или субоптимальный по времени и очень эффективный по памяти подход к решению многих строковых задач. Идейная простота и большая гибкость делают их отличным выбором в техническом арсенале участников соревнований по программированию. Другими методами аналогичной мощности являются суффиксные деревья [5, 10] и суффиксные автоматы.

Метод непосредственного построения суффиксного массива за линейное время описан в работе [12]. Большое количество приложений можно найти в работах [5, 10].

Задачи и разборы

Задача A. Сны Антона

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Антон — математик, которому часто снятся “профессиональные” сны. К сожалению, наутро он редко может точно вспомнить происходившее. Не так давно ему приснилась прекрасная перестановка $(a_1, a_2, ..., a_n)$ множества $\{1, ..., n\}$. Какая именно — неизвестно, но Антон четко запомнил все характерные участки ее графика.

Точнее, он запомнил *множество спуска* перестановки $S = \{i \mid a_i > a_{i+1}\}$, благо оно было не очень большим.

Помогите Антону найти количество перестановок, которые могли ему приснится, по модулю $1\,000\,003$.

Формат входного файла

Первая строка входного файла содержит два целых числа n и k ($1 \le n \le 10^6$, $0 \le k \le 200$, $k < n$). Следующая строка содержит k различных натуральных чисел s_i — все элементы множества S ($1 \le s_i < n$).

Формат выходного файла

Выведите единственное целое число — искомое количество перестановок по модулю $1\,000\,003$.

Пример

a.in	a.out
3 1 2	2
3 2 1 2	1

Разбор задачи A. Сны Антона

Первый полиномиальный алгоритм, который приходит в голову — вычислить величины c_{ik}, равные количеству перестановок длины i удовлетворяющих (частичным) ограничениям на спуски S и последний элемент которых равен k. c_{ik} можно выразить рекуррентно через значения c с меньшими индексами. С помощью динамического программирования можно вычислить все значения c_{nk} за время $O(n^2)$, что очевидно не укладывается в ограничения. И хотя есть способ ускорить этот метод до $O(n \log n)$, его реализация все равно не уложится в ограничения времени.

Подойдем к проблеме с другой стороны. Пусть искомая величина равна $f(S)$. Определим $g(S)$ как количество перестановок длины n, множество спуска которых содержится в S. Тогда согласно принципу включения-исключения:

$$f(S) = \sum_{T \subseteq S} (-1)^{|S \backslash T|} g(T) \tag{1}$$

Неформально это можно обосновать так. Из определения $g(S)$ справедлива формула:

$$g(S) = \sum_{T \subseteq S} f(T)$$

Будем считать $g(S)$ первым приближением для $f(S)$. Правда, $g(S)$ больше $f(S)$, хотя бы потому что в ней учтено количество перестановок с множеством спуска $T \subset S$, $|S \backslash T| = 1$. Поэтому отнимем от $g(S)$ величину:

$$\sum_{\substack{T \subseteq S \\ |S \backslash T| = 1}} g(T)$$

Полученное число будет более точным приближением, но меньше $f(S)$, поскольку мы лишний раз вычли количество перестановок с множеством спуска $T \subset S$, $|S \backslash T| = 2$. Чтобы получить лучшее приближение, снова

прибавляем:

$$\sum_{\substack{T \subseteq S \\ |S \backslash T| = 2}} g(T)$$

и так далее, пока, наконец, не получим точную формулу (1).

Что нам это дает? Оказывается, функция $g(S)$ легко вычисляется. Пусть $s_1 < s_2 < ... < s_k$ — элементы множества S, тогда:

$$g(S) = \frac{n!}{s_1!(s_2 - s_1)! \cdots (n - s_k)!} \tag{2}$$

Действительно, чтобы получить перестановку $(a_1, a_2, ..., a_n)$, чье множество спуска принадлежит S, сначала выберем последовательность $a_1 < a_2 < ... < a_{s_1}$ $\binom{n}{s_1}$ способами. Затем выберем последовательность $a_{s_1+1} < a_{s_1+2} < ... < a_{s_2}$ $\binom{n-s_1}{s_2-s_1}$ способами и так далее. В итоге получаем:

$$g(S) = \binom{n}{s_1}\binom{n - s_1}{s_2 - s_1} \cdots \binom{n - s_k}{n - s_k} = \frac{n!}{s_1!(s_2 - s_1)! \cdots (n - s_k)!}$$

На основании формул (1), (2) можно получить алгоритм со временем работы $O(2^k)$. На первых взгляд кажется, что экспоненциальный алгоритм хуже полиномиального, но это не совсем так.

Раскроем (1) до вида:

$$f(S) = n! \sum_{1 \le i_1 < i_2 < ... < i_j \le k} (-1)^{k-j} \frac{1}{s_{i_1}!(s_{i_2} - s_{i_1})! \cdots (n - s_{i_j})!} \tag{3}$$

Здесь суммирование идет по всем подмножествам индексов из S, записанным в порядке возрастания, а $n!$ вынесен за знак суммы. Рассмотрим матрицу:

$$\begin{pmatrix} \frac{1}{s_1!} & \frac{1}{s_2!} & \frac{1}{s_3!} & \cdots & \frac{1}{n!} \\ 1 & \frac{1}{(s_2-s_1)!} & \frac{1}{(s_3-s_1)!} & \cdots & \frac{1}{(n-s_1)!} \\ 0 & 1 & \frac{1}{(s_3-s_2)!} & \cdots & \frac{1}{(n-s_2)!} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & \frac{1}{(n-s_k)!} \end{pmatrix}$$

Тогда члены суммы из правой части (3) есть в точности ненулевые слагаемые в разложении определителя этой матрицы. Если положить для удобства $s_0 = 0$, $s_{k+1} = n$, $1/(s_i - s_j)! = 0$ при $s_i < s_j$, то можно записать

$$f(S) = n! \det[1/(s_{j+1} - s_i)!] \tag{4}$$

Число $p = 1\,000\,003$ простое, так что все элементы $\mathbb{Z}_p^*$ обратимы и мы можем непосредственно найти определитель матрицы в формуле (4) методом Гаусса. Для того чтобы время работы алгоритма составило $O(k^3)$, нужно предварительно вычислить все обратные и все факториалы в $\mathbb{Z}_p^*$, что можно сделать за время $O(p)$. Общая сложность алгоритма равна $O(k^3 + p)$. Заметим, что этот метод работает только для $n < p$.

Задача B. Буковель

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	2.5 с
Ограничение по памяти:	64 Мб

У владельцев одного горнолыжного курорта[14] проснулась совесть, и они наконец-то решили проложить несколько новых трасс. Для удобства пользования нижние станции новых подъемников (и, соответственно, спуски новых трасс) должны находится недалеко друг от друга. Кроме того, трассы не должны пересекаться.

Рельеф местности задан прямоугольной матрицей высот. Каждый элемент матрицы содержит высоту соответствующей квадратной *клетки* местности. Две клетки называются соседними, если они имеют общее ребро. *Трассой* называется непустая последовательность соседних клеток, высоты которых убывают. По определению *длина* трассы на единицу меньше количества клеток в ней.

Согласно требованиям, все трассы должны заканчиваться в одной и той же клетке. Кроме того, никакая другая клетка местности не должна одновременно принадлежать нескольким трассам.

Вычислите максимальную суммарную длину новых трасс, которые можно проложить согласно требованиям.

Формат входного файла

Первая строка входного файла содержит два натуральных числа n и m — размерность матрицы высот ($1 \le n, m \le 30$). Следующие n строк содержат по m целых чисел h_{ij} каждая — высоты соответствующих клеток местности ($0 \le h_{ij} \le 2061$).

Формат выходного файла

Выведите единственное целое число — максимальную суммарную длину трасс, которые можно проложить согласно требованиям.

[14] Условие задачи приснилось автору вместе с названием. Все совпадения случайны.

Пример

b.in	b.out
6 5 9 9 9 9 9 7 6 5 7 8 9 3 4 6 9 9 2 4 5 9 9 3 9 9 9 9 5 9 9 9	16
3 3 5 5 5 5 5 5 5 5 5	0

Разбор задачи B. Буковель

Требование "непересекаемости" трасс наводит на мысль, что задача связана с потоками в сетях. Дополнительные требование максимальности суммарной длины подсказывает нам, что задачу надо сводить к задачам о взвешенном потоке.

Для начала научимся решать задачу в том случае, когда расположение общей нижней точки трасс фиксировано. Обозначим множество клеток местности $\{v_i\}$. Пусть общая нижняя точка всех трасс находится в клетке v_s.

Построим взвешенную сеть с множеством вершин $V = \{v_i^{in}\} \cup \{v_i^{out}\} \cup \cup\{t\}$. Здесь каждой клетке местности v_i соответствуют 2 вершины v_i^{in} и v_i^{out}, плюс выделена вершина t. Все ребра в нашей сети будут иметь пропускную способность 1, но различную стоимость. Для начала, проведем ребра стоимостью -1 от каждой вершины v_i^{in} к соответствующей v_i^{out}. Для каждой пары соседних клеток $(u, v) : h_u < h_v$ проведем ребро от u^{out} к v^{in} стоимостью 0. Наконец, проведем ребра стоимостью 0 от всех v_i^{out} $(i \neq s)$ к t.

Легко убедиться, что каждой допустимой конфигурации трасс соответствует поток из вершины $s = v_s^{out}$ в t и наоборот, каждому потоку соответствует допустимая конфигурация трасс. При этом суммарная длина трассы равна стоимости потока, взятому со знаком минус. Нетрудно сообразить, что для максимизации суммарной длины величина потока должна быть максимальной, потому что любой дополняющий путь из s в t имеет отрицательную стоимость. Поэтому алгоритм нахождения максимального потока минимальной стоимости решает нашу частную задачу. Чтобы ре-

шить общую задачу, достаточно перебрать расположение нижней точки v_s. Общее время работы составит $O(nm \cdot T(n,m))$, где $T(n,m)$ — время поиска максимального потока минимальной стоимости в описанной сети.

Простой алгоритм нахождения максимального потока минимальной стоимости состоит в последовательном увеличении потока вдоль самого дешевого дополняющего $s-t$ пути в остаточной сети [7]. Поэтому мы можем последовательно применять какой-то из алгоритмов поиска кратчайшего $s-t$ пути в графе, допускающем ребра отрицательного веса, например, алгоритм Беллмана-Форда. Его сложность равна $O(|V||E|)$, что для нашей сети превращается в $O((nm)^2)$. Заметим также, что максимальный поток ограничен константой 4 — степенью вершины s, так что $T(n,m)$ тоже равно $O((nm)^2)$. Собрав все вместе, получим общее время работы $O((nm)^3)$. Такое решение задачи не пройдет по времени при указанных ограничениях на n и m.

Попробуем где-то сэкономить. Для этого нам понадобятся такие утверждения:

Утверждение 1. Рассмотрим взвешенный граф $G = (V, w)$ с функцией веса $w : V \times V \to \mathbb{R}$, выделенную вершину $s \in V$ и функцию $\varphi : V \to \mathbb{R}$. Определим на том же множестве вершин граф $G_\varphi = (V, w_\varphi)$ с функцией веса $w_\varphi(u,v) = w(u,v) + \varphi(u) - \varphi(v)\ \forall\ u,v \in V$. Тогда дерево кратчайших путей из s в G_φ будет также деревом кратчайших путей из s в G.

Утверждение 2. Если в предыдущем утверждении положить $\varphi(v)$ равной кратчайшему расстоянию от s до v в G, то все веса ребер в G_φ будут неотрицательны: $w_\varphi(u,v) \geq 0\ \forall\ u,v \in V$.

Утверждение 3. Пусть граф $G = (V, w)$ порожден некоторой взвешенной остаточной $s-t$ сетью N, а $\varphi(v)$ равно кратчайшему расстоянию от s до v в G. Обозначим $\tilde{N}$ остаточную сеть, полученную из N путем увеличения потока вдоль самого дешевого дополняющего $s-t$ пути. Ей соответствует взвешенный граф $\tilde{G} = (V, \tilde{w})$. Тогда по-прежнему справедливо $\tilde{w_\varphi}(u,v) \geq 0\ \forall\ u,v \in V$.

Вышесказанное приводит к следующему алгоритму нахождения максимального потока минимальной стоимости. Сначала вычислим все кратчайшие пути из s в графе G, определив таким образом $\varphi(v)$. Затем будем последовательно применять алгоритм Дейкстры для поиска всех кратчайших путей из s в графе G_φ, проталкивать поток вдоль кратчайшего пути и обновлять $\varphi(v)$.

В общем случае нахождение всех кратчайших путей в начальном графе G требует времени $O(|V||E|)$, что не дает выигрыша в нашем частном случае: $O(|V||E| + 4|E|\log|V|) = O(|V||E|) = O((nm)^2)$. Однако граф G ациклический и мы можем вычислить все кратчайшие пути из

s с помощью топологической сортировки за $O(E)$. Это приводит к оценке $O(|E| + 4|E| \log |V|) = O(nm \log nm)$ для $T(n, m)$ и общему времени работы $O((nm)^2 \log nm)$.

Задача C. Хитрый ним

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Двое членов жюри Зимней школы по программированию любят играть друг с другом в различные модификации игры ним. Однако, после лекции двухдневной давности, игра стала неинтересной — оба научились с легкостью определять выигрышность позиции и оптимальную стратегию. Поэтому для поддержания интереса, они решили сыграть в поддавки.

Версия нима, в которую они играли в последний раз, имела следующие правила. На столе лежат несколько кучек камней. За один ход разрешается либо взять произвольное число камней из любой кучки, либо разделить любую кучку на две непустых. Согласно обычным правилам игрок, который не может сделать ход, проигрывает. Но в поддавках проигрывает тот, после чьего хода останется пустой стол.

Формат входного файла

Первая строка входного файла содержит натуральное число n ($n \leq 10$) — количество кучек. Вторая строка содержит n натуральных чисел a_i — размеры кучек ($a_i \leq 50\,000$).

Формат выходного файла

Выведите слово “First” (без кавычек), если первый игрок может обеспечить себе победу в описанной игре в поддавки при оптимальной игре обоих. Иначе выведите “Second”.

Пример

c.in	c.out
1 1	Second
1 5	First
2 2 2	Second

Разбор задачи C. Хитрый ним

Несмотря на то, что игра в поддавки не раскладывается простым образом в сумму игр, ее анализ не намного сложнее игры по обычным правилам.

Сначала вычислим числа Шпрага-Гранди $\mathcal{G}$ для всех позиций игры по обычным правилам, состоящим из одной кучки.

Утверждение 6. Для всех $n \in \mathbb{N}$:

$$\mathcal{G}(\{n\}) = \begin{cases} n-1, & \text{если } n \pmod 4 \equiv 0 \\ n, & \text{если } n \pmod 4 \equiv 1 \text{ или } 2 \\ n+1, & \text{если } n \pmod 4 \equiv 2 \end{cases}$$

Доказательство индукцией по n. База $n = 1$ тривиальна. Пусть 6-е утверждение справедливо для всех $i < n$, проверим его для n. Во-первых, из такой кучки можно получить кучку любого меньшего размера удалением камней, т.е. позицию с числом Шпрага-Гранди $\mathcal{P}(\{i\})$ $\forall i < n$. Во-вторых, непосредственным перебором случаев можно убедиться, что разделение кучки на две не позволяет получить позицию с числом Шпрага-Гранди отличным от $\mathcal{P}(\{i\})$ $(\forall\, i < n)$, кроме случая $n = 4k+3$: тогда $\mathcal{P}(\{1, n-1\}) = \mathcal{P}(\{1\}) \oplus \mathcal{P}(\{n-1\}) = 1 \oplus (4k+2) = 4k+3 = n$. Таким образом, для $n = 4k+3$ $\mathcal{P}(\{n\}) = n+1$. В остальных случаях разделение кучки на две не дает ничего нового в терминах чисел $\mathcal{P}$. $\square$

Теперь нетрудно проанализировать игру в поддавки.

Утверждение 7. Позиция $P = \{a_1, a_2, ..., a_k\}$ проигрышная (для первого игрока) при игре в поддавки тогда и только тогда, когда

$$\mathcal{G}(a_1) \oplus \mathcal{G}(a_2) \oplus \cdots \oplus \mathcal{G}(a_k) = 0, \tag{5}$$

кроме случая, когда все a_i равны 1. Тогда позиция проигрышная, так как $\mathcal{G}(a_1) \oplus \mathcal{G}(a_2) \oplus \cdots \oplus \mathcal{G}(a_k) = 1$.

Доказательство. Обозначим описанное множество позиций буквой $\mathcal{P}$. Мы хотим показать, что все позиции из $\mathcal{P}$ проигрышные, а все из $\overline{\mathcal{P}}$ — выигрышные. Легко убедиться, что из позиций из $\mathcal{P}$ нет ходов в другие позиции из $\mathcal{P}$ (согласно определению чисел Шпрага-Гранди $\mathcal{G}$).

Покажем, что из любой позиции из $\overline{\mathcal{P}}$ существует ход в $\mathcal{P}$. Для этого применим стратегию, аналогичную стратегии для обычной игры: будем ходить таким образом, чтобы в новой позиции была справедлива формула (5). Но если в результате получилось множество кучек из 1 камня, то мы всегда можем скорректировать ход, чтобы выполнить вторую часть утверждения. Скажем, если выбранным ходом мы оставляем 1 камень в кучке,

то вместо этого заберем всю кучку; если мы забираем всю кучку, то вместо этого можем оставить в ней 1 камень; если мы делим кучку на 2 по 1 камню, то вместо этого мы можем оставить 1 камень. □

Задача D. Эпидемия

Имя входного файла:	`d.in`
Имя выходного файла:	`d.out`
Ограничение по времени:	2 с
Ограничение по памяти:	64 Мб

В результате последнего исследования микробиолог Вася обнаружил несколько новых штаммов патогенных бактерий. Оказалось, что разные бактерии конкурируют между собой в питательной среде, так что более сильные угнетают более слабых. Для удобства Вася пронумеровал различные штаммы натуральными числами так, что штамм с большим номером сильнее всех штаммов с меньшим номером.

Будем считать, что питательная среда двумерна и разбита квадратной сеткой на одинаковые клетки. Для простоты предположим, что время дискретно. Тогда динамика распространения бактерий определяется следующим правилом. Некоторая клетка в момент времени t будет заселена самым сильным из штаммов, которые находились в момент времени $t-1$ в одной из 5 клеток: текущей или такой, которая имеет с ней общее ребро (т.е. находится непосредственно сверху, снизу, справа или слева). Если же все эти 5 клеток не были заселенны в момент $t-1$, то текущая клетка останется незаселенной и в момент t.

Васе необходимо провести наблюдение динамики распространения бактерий. Для этого в момент времени 0 он одновременно заселил несколько различных штаммов в разных местах питательной среды. К сожалению, после этого в лаборатории отключили электричество и эксперимент не состоялся.

Помогите Васе сфабриковать результаты наблюдений.

Формат входного файла

Первая строка входного файла содержит три натуральных числа n, m и q — размеры питательной среды и количество наблюдений ($n, m \le 300$, $q \le 3 \cdot 10^5$).

Следующие n строк содержат по m целых чисел a_{ij} — содержимое соответствующих клеток в начальный момент времени ($0 \le a_{ij} \le 10^9$). Значение 0 означает незаселенную клетку, положительное число — номер штамма.

Следующие q строк содержат по три целых числа r_i, c_i, t_i каждая — координаты (строка, столбец) и время i-го наблюдения ($1 \le r_i \le n$, $1 \le c_i \le m$, $0 \le t_i \le 10^9$).

Формат выходного файла

Для каждого наблюдения, в порядке поступления, выведите в отдельной строке содержимое заданной клетки в заданный момент времени. Выведите 0, если клетка не заселенна, иначе — номер штамма.

Пример

d.in	d.out
6 5 6 0 0 0 0 0 0 1 0 0 0 0 0 0 3 0 0 0 0 0 0 0 2 0 0 0 0 0 0 2 0 1 1 0 5 2 0 4 4 1 5 4 2 1 1 4 1 1 5	0 2 3 3 1 3

Разбор задачи D. Эпидемия

По аналогии с эвклидовой плоскостью будем называть множество клеток $S_r(i,j) = \{(i',j') \in \mathbb{Z}^2 : |i-i'| + |j-j'| \le r\}$ кругом радиуса r с центром в (i,j).

Продлим питательную среду до бесконечной плоскости, считая пространство за границей незаселенным в начальный момент времени. Тогда справедливо следующее Утверждение 1. Содержимое клетки (i,j) в момент времени $t \ge 0$ равно:

$$a_{ij}^t = \max_{(k,m) \in S_t(i,j)} a_{km}^0 \stackrel{\text{def}}{=} M(i,j,t).$$

Таким образом, задача свелась к быстрому вычислению $M(i,j,t)$. Покажем, как выполнить предобработку за время $O(nm \log nm)$, чтобы потом отвечать на такие запросы за $O(1)$. Идея ничем не отличается от той, которая использовалась в структуре данных *RMQ* в лекции.

Действительно, если $2 \le 2^k \le t < 2^{k+1}$, то

$$\begin{aligned} S_t(i,j) =& S_{2^k}(i + 2^k - t, j) \cup S_{2^k}(i - 2^k + t, j) \cup \\ & S_{2^k}(i, j + 2^k - t) \cup S_{2^k}(i, j - 2^k + t) \end{aligned}$$

откуда немедленно следует:

$$\begin{aligned} M(i,j,t) = \max\{& M(i + 2^k - t, j, 2^k), M(i - 2^k + t, j, 2^k), \\ & M(i, j + 2^k - t, 2^k), M(i, j - 2^k + t, 2^k)\}. \end{aligned}$$

Поэтому достаточно вычислить значения $M(i,j,2^k)$ для всех интересных троек i,j,k. Всего таких троек будет $O(nm \log nm)$. Каждое значение вычисляется за $O(1)$, так что общее время предобработки равно $O(nm \log nm)$. Запросы с $t = 0$ и $t = 1$ обрабатываются отдельно, с $t \ge 2$ — по общей схеме за $O(1)$.

Задача E. Развлечение

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Один инструктор решил развлечь участников своей группы, мирно сидящих вечером у костра. Для этого он бросил в огонь пластиковую бутылку с остатками бензина.

Участники знают, что через некоторое время бутылка взорвется и вылетит из костра по прямой траектории в случайном направлении, пока не воткнется в кого-то или не улетит очень далеко. Каждого из них интересует один вопрос: с какой вероятностью бутылка попадет в него, если все останутся сидеть на своих местах?

Костер находится в центре координат $(0,0)$. Участники для простоты представлены кругами радиуса 1. Угол вылета бутылки распределен равномерно на $[0, 2\pi)$.

Формат входного файла

Первая строка входного файла содержит натуральное число n — количество участников ($n \le 5 \cdot 10^4$). Следующие n строк содержат пары целых

чисел x_i, y_i — координаты центра круга, представляющего i—го участника ($|x_i|, |y_i| \leq 10^4$). Никакие два круга не имеют общих внутренних точек. Никакой круг не содержит начало координат.

Формат выходного файла

Выведите n строк. В i—й строке выведите вероятность поражения бутылкой i—го участника с абсолютной или относительной погрешностью до 10^{-6}.

Пример

e.in	e.out
5 0 2 0 4 -1 -10 0 -15 -4 1	0.1666666666666667 0.0 0.03172551743055352 0.010618204803539803 0.07797913037736935

Разбор задачи E. Развлечение

Для начала решим задачу для единственного круга. Понятно, что вероятность попадания пропорциональна углу, под которым круг виден из начала координат.

Как вычислить этот угол? Заметим, что при повороте системы координат вокруг $(0, 0)$ угол остается постоянным. Повернем систему так, чтобы координаты центра круга P равнялись $(r, 0)$. Треугольник, образованный центром координат O, центром круга P и точкой касания K_1 будет прямоугольным. $|OP| = r, |PK_1| = 1$, откуда $\sin\theta = 1/r$. Угловой размер круга равен 2θ.

Что делать, когда кругов много и они могут заслонять друг друга? Используем прием, называемый заметанием. Представим себе бесконечный луч с началом в O, в начальный момент времени совпадающий с осью абсцисс и вращающийся с единичной угловой скоростью вокруг начала координат. Вычислим моменты времени, когда луч коснется какого-то из кругов, и моменты, когда он оторвется от какого-то из кругов. Ясно, что на каждом промежутке между двумя такими событиями луч пересекает постоянное множество кругов. Это можно использовать для решения задачи.

Вычислим моменты событий и отсортируем их по невозрастанию. В случае равенства углов — события отрыва расположим до событий касания. Может случится, что какой-то круг пересекает ось абсцисс — тогда его

можно условно разрезать на 2 кусочка, каждый из которых порождает 2 события.

Теперь пройдемся по событиям в порядке увеличения времени (или угла, что то же самое). Во время прохода будем хранить в приоритетной очереди множество кругов, которые пересекает в данный момент наш воображаемый луч. Эта структура данных должна хранить круги в порядке увеличения расстояния от них до начала координат. Тогда на каждом промежутке между событиями мы сможем узнать номер участника, подвергающегося риску быть пораженным бутылкой в соответствующем диапазоне углов вылета, сделав один запрос на минимальный элемент в структуре. Проэмулировав оборот луча на 2π, мы вычислим угловой размер каждого из кругов с учетом заслонений. Обработка самих событий тривиальна и состоит в добавлении или удалении кругов в приоритетную очередь.

Если реализовать приоритетную очередь на основе двоичной кучи, каждая из операций потребует $O(\log n)$ времени. Поскольку общее число событий линейно, а сортировку можно реализовать за $O(n \log n)$, общая сложность алгоритма тоже не превзойдет $O(n \log n)$.

Задача F. Феечка

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Одна феечка любила складывать числа. Поэтому, когда Петя написал на доске n натуральных чисел $a_1, a_2, ..., a_n$ по кругу и вышел за мелом, она заменила каждое число на его сумму с двумя соседними числами (все значения были взяты до момента изменений на доске). Петя вернулся и очень расстроился из-за того, что исходные числа утрачены. Помогите ему найти хотя бы одну из возможных исходных последовательностей (благо, феечка складывает без ошибок).

Формат входного файла

Первая строка входного файла содержит число n — количество чисел на доске ($3 \le n \le 10^5$). Следующие n строк содержат по одному натуральному числу b_i каждая ($b_i \le 10^9$) — это числа, оказавшиеся на доске после действий феечки. На месте числа a_1 оказалось b_1, a_2 было заменено на b_2 и так далее.

Формат выходного файла

Выведите n строк, содержащих исходную последовательность. В строке i выведите число a_i. Если возможных последовательностей несколько, выведите любую.

Пример

f.in	f.out
3 10 10 10	3 6 1
5 11 6 10 16 14	1 2 3 5 8

Разбор задачи F. Феечка

Задача взята из материалов хорватских школьных олимпиад по информатике [9].

Фраза в условии о том что “феечка складывает без ошибок” означает, что хотя бы одно решение существует.

Заметим, что

$$S = \sum_{i=1}^{n} a_i = \frac{1}{3} \sum_{i=1}^{n} b_i \tag{6}$$

Также заметим, что

$$b_{i+2} - b_{i+1} = a_{i+3} - a_i$$

так что

$$a_{i+3} = a_i + b_{i+2} - b_{i+1} \tag{7}$$

Здесь и далее мы отождествляем индексы 0 и n, 1 и $n+1$ и т.д. для удобства записи.

Рассмотрим случай, когда $n \pmod 3 \not\equiv 0$. Тогда решение находится однозначно. Положим $a'_1 = 0$ и вычислим остальные $a'_4, a'_7, \ldots, a'_{n-2}$ по формуле (7). Ясно, что равенство (6) не будет выполнятся для a'_i. Но из такой последовательности легко сделать допустимую, положив $a_1 = (S - \sum_{i=1}^{n} a'_i)/n$ и вычислив остальные a_i тем же способом.

Если же $n \pmod 3 \equiv 0$, то последовательность распадается на три подпоследовательности $a_1, a_4, \ldots, a_{n-2}$, $a_2, a_5, \ldots, a_{n-1}$ и $a_3, a_6, \ldots, a_n$. Положим $a'_1 = 0$, $a'_2 = 0$ и вычислим все a'_i с номерами, не кратными 3 по формуле (7). Глядя на них, мы можем найти минимальные значения a_1 и a_2, при которых все a_i с номерами, не кратными 3, будут натуральными (изменяя первый член подпоследовательности, мы фактически прибавляем константу ко всем членам). После чего осталось решить задачу для $a_3, a_6, \ldots, a_n$ — требуемую сумму этих элементов можно вычислить, отняв от S сумму элементов с номерами, не кратными 3.

Понятно, что если во втором случае существует хотя бы одно решение, то существует и решение с минимальными a_1 и a_2, которое и будет найдено таким алгоритмом. Временная сложность алгоритма равна $O(n)$.

Задача G. Джентельмен удачи

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	4 с
Ограничение по памяти:	64 Мб

Охотник за сокровищами древних царей пробрался в место захоронения упомянутых сокровищ. Оно представляет собой широкий водоем прямоугольной формы. Внутри водоема насыпаны небольшие островки, на которых и спрятаны бесценные золотые артефакты. В воде когда-то кишели стаи злобных крокодилов, но, в связи с глобальным похолоданием[15], они впали в спячку, а сам водоем даже покрылся корочкой льда. Этим обстоятельством и хочет воспользоваться расхититель гробниц и прочих достопримечательностей.

Последователь Индианы Джонса за годы скитаний развил способность переносить на себе груз любого веса. Опасность заключается лишь в том, что лед может проломиться под суммарным весом его тела и груза.

Формат входного файла

Первая строка входного файла содержит 3 натуральных числа n, m и w_0 — размеры водоема и масса тела охотника за сокровищами в килограммах ($n \le 300, m \le 300, w_0 \le 100$). Следующие n строк содержат описание водоема. Каждая строка содержит m натуральных чисел или больших латинских букв, разделенных пробелами. Натуральное число x в строке равно максимальной нагрузке в килограммах, которую выдерживает соответствующий участок льда ($x \le 1\,000$). Большая латинская буква означает,

[15] На дворе XXII век.

что в соответствующем участке водоема находится островок с золотым артефактом. Буква A означает артефакт массой 1 кг, B — 2 кг, ..., Z — 26 кг. Островок выдерживает любую массу. Количество островков не превышает 8.

Формат выходного файла

Выведите единственное целое число — максимальный суммарный вес артефактов, который может вынести охотник за сокровищами за одну ходку, не провалившись под лед. Искомый маршрут должен начинаться и заканчиваться на берегу. Маршрут может проходить по берегу несколько раз. Когда охотник находится на берегу, он может перейти в любой из участков на границе водоема. Находясь внутри водоема он может перейти в любой из 4 соседних по горизонтали или вертикали участков. Аналогично, он может выйти на берег из любого участка на границе водоема.

Охотник не может оставлять часть груза на берегу, островках или на льду — боится, что украдут. Так же он не может взять с собой часть артефакта.

Пример

g.in	g.out
4 4 80 100 100 100 100 100 J J J 100 100 100 100 100 100 100 100	30
4 4 100 100 100 100 100 100 J 100 J 100 100 100 100 100 J 100 100	20
3 7 80 100 100 100 100 100 100 100 100 G 100 G 100 H 100 100 100 100 100 100 100 100	15

Разбор задачи G. Джентельмен удачи

Несмотря на большие размеры водоёма, для нахождения решения задачи важен лишь факт наличия или отсутствия пути между двумя островками или между островком и берегом при определенной нагрузке.

Переберем все возможные значения нагрузки w от w_0 до $w_0 + 26k$ и для каждого из них построим граф G_w из $k + 1$ вершин, содержащий информацию о связности между названными ключевыми точками. Это можно сделать с помощью серии поисков в ширину или в глубину. После чего переберем $k!$ способов обхода островков и выберем тот, который приносит наибольшую прибыль. Нужно учитывать возможность завершения обхода после посещения части островков.

Сложность такого алгоритма формально равна $O(n \cdot m \cdot k + k!)$, где в константе скрыт максимальный вес одного артефакта. Но при заданных ограничениях решение проходит.

Задача H. Гусарская рулетка

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Двое гусар не поделили даму сердца. Поскольку дама — дискретный объект, они решили ее разыграть.

Игра состоит в следующем. Имеется n-зарядный револьвер, заряженный одним боевым патроном. Вероятность нахождения патрона распределена равномерно по всему барабану. Гусары ходят по очереди, начинает первый. На каждом шаге игрок выбирает, отправиться ли ему в кабак (тогда он получит полезность 0, а дама достанется другому) или приставить револьвер к виску и нажать на курок. Смерть означает полезность равную -1 (при этом дама, естественно, достается другому). Если гусар остается жив, то право и обязанность выбора кабак/стреляться переходит к другому. Барабан револьвера при этом не перекручивается, т.е. вероятность смерти увеличивается. Процесс продолжается до тех пор, пока кто-то не спасует или не умрет. Ценность[16] дамы для первого гусара равна a, для второго — b.

Гусары играют оптимально, т.е. каждым ходом максимизируют математическое ожидание полезности для себя.

[16] В данном контексте ценность – синоним полезности

Формат входного файла

Первая строка входного файла содержит три числа n, a и b — емкость барабана револьвера ($1 \le n \le 100$) и ценность дамы для первого и второго игроков ($0 < a, b \le 10$, числа вещественные). Гарантируется, что у обоих игроков оптимальная стратегия единственна.

Формат выходного файла

Напечатайте в первой строке два числа: вероятность того, что первый гусар умрет, и вероятность того, что дама достанется первому. Во второй строке напечатайте аналогичные вероятности для второго игрока. Абсолютная или относительная погрешность чисел в ответе не должна превышать 10^{-6}.

Пример

h.in	h.out
1 3.14 2.71	0.0 0.0 0.0 1.0
6 3.14 2.71	0.5000000000000002 0.5000000000000001 0.3333333333333335 0.5

Разбор задачи H. Гусарская рулетка

Ясно, что холостой выстрел из $n-$зарядного револьвера превращает его в виртуальный $(n-1)$-зарядный с точки зрения игроков.

Проанализируем игру с конца. Для всех i от 1 до n вычислим величины p_i^k и E_i^k — вероятность того, что даму получит k-й игрок и матожидание полезности k-го игрока при условии, что емкость револьвера равна i, а право стрельбы перешло к игроку с номером $x = 1 + (n - i) \pmod 2$.

Для удобства обозначим полезность дамы для первого игрока a^1, а для второго — a^2. Если в какой-то момент право стрельбы перешло к игроку x, то номер другого игрока будет равен $3 - x$.

Если $i = 1$, то стреляться самоубийственно. Поэтому или

$$p_1^1 = 1,\ E_1^1 = a^1,\ p_1^2 = 0,\ E_1^2 = 0,$$

или же

$$p_1^1 = 0,\ E_1^1 = 0,\ p_1^2 = 1,\ E_1^2 = a^2$$

в зависимости от четности n.

Иначе у активного игрока x есть два варианта поведения. Или пойти в кабак (тогда полезность равна 0), или стреляться. Во втором случае с вероятностью $1/i$ игрок погибает и с вероятностью $(1 - 1/i)$ передает ход, уменьшая эффективную емкость револьвера до $i - 1$. Матожидание полезности при этом равно $-1/i + (1 - 1/i)E^x_{i-1}$.

Игрок x выбирает вариант с большим матожиданием полезности (согласно условию, такой вариант единственный). Так что если

$$-\frac{1}{i} + (1 - \frac{1}{i})E^x_{i-1} < 0,$$

то

$$\begin{aligned} E^x_i &= 0 \\ p^x_i &= 0 \\ E^{3-x}_i &= a^{3-x} \\ p^{3-x}_i &= 1 \end{aligned}$$

Иначе

$$-\frac{1}{i} + (1 - \frac{1}{i})E^x_{i-1} > 0,$$

откуда следует

$$\begin{aligned} E^x_i &= \frac{-1}{i} + (1 - \frac{1}{i})E^x_{i-1} \\ p^x_i &= (1 - \frac{1}{i})p^x_{i-1} \\ E^{3-x}_i &= \frac{a^{3-x}}{i} + (1 - \frac{1}{i})E^{3-x}_{i-1} \\ p^{3-x}_i &= \frac{1}{i} + (1 - \frac{1}{i})p^{3-x}_{i-1} \end{aligned}$$

На основании этих рекуррентных соотношений легко построить алгоритм со временем работы $O(n)$, который вычисляет p^k_n и E^k_n.

Наконец заметим, что вероятность получить даму p^k_n, вероятность погибнуть d^k_n и матожидание полезности E^k_n связаны соотношением

$$E^k_n = -d^k_n + a^k p^k_n,$$

откуда легко выразить d^k_n.

Задача I. Интересные строки

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	2.5 с
Ограничение по памяти:	64 Мб

Назовем строку P $k-$*интересной* подстрокой строки S, если P входит в S по крайней мере в k позициях (вхождения могут перекрываться). Например, строка “aba” является 4-интересной подстрокой строки “abababababa”, потому что входит в неё начиная с позиций 1, 3, 5, 7; она также является 1, 2, 3 - интересной.

По данной строке S и числу k найдите максимальную возможную длину $k-$интересной подстроки S. Так, в предыдущем примере максимальной 3-интересной подстрокой будет “ababa”.

Формат входного файла

Первая строка входного файла содержит два натуральных числа n и k ($n \leq 10^5, k \leq n$). Вторая строка содержит S, состоящую из n символов — маленьких и больших латинских букв, цифр.

Формат выходного файла

Выведите единственное число — максимальную длину $k-$интересной подстроки S.

Пример

i.in	i.out
9 3 abababababa	5
11 2 abracadabra	4
11 5 abracadabra	1
11 6 abracadabra	0

Разбор задачи I. Интересные строки

Один из детерминированных способов решения этой задачи состоял в использовании суффиксного массива строки S. Нетрудно проверить следующее. Утверждение 1. Если строка S имеет $k-$интересную подстроку

длины m, то в последовательности *LCP* соседних по порядку суффиксов S найдется подотрезок длины $k-1$, все элементы которого больше или равны m.

Для реализации было достаточно найти суффиксный массив и массив *LCP* за время $O(n \log n)$, после чего рассмотреть все числа

$$a_i = \min\{\mathrm{lcp}_i, \mathrm{lcp}_{i+1}, \ldots, \mathrm{lcp}_{k-2}\}$$

и выбрать среди них наибольшее. Это достаточно сделать за время $O(n \log n)$, например использовав приоритетную очередь со сложностью операций $O(\log n)$. Альтернативный способ со временем работы $O(n)$ описан в [8]. Тогда общая сложность алгоритма тоже составит $O(n \log n)$.

Однако, эта задача имеет довольно простое вероятностное решение в духе алгоритма Рабина-Карпа. Заметим, что если некоторая подстрока $k-$интересная, то и все её подстроки меньшего размера тоже k-интересные. Поэтому можно делать двоичный поиск по ответу.

Пусть $m = (l+r)/2$ — догадка на очередном шаге двоичного поиска. Тогда нам надо уметь достаточно быстро отвечать на вопрос "существует ли хотя бы k различных подстрок S длины m?" Для этого можно вычислить хеш-функции вида $s_i r^{m-1} + s_{i+1} r^{m-2} + \cdots + s_{i+m-2} r + s_{i+m-1} \pmod p$ от всех подстрок S длины m за общее линейное время, а затем с помощью хеш-таблицы найти количество повторений каждого значения. Если вероятность коллизий маленькая, мы получим ответ с высокой вероятностью. Суммарное время одной итерации двоичного поиска равно $O(n)$, а сложность всего алгоритма будет равна $O(n \log n)$.

Задача J. Игра

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Двое играют в такую игру. Есть круглый стол радиуса R и бесконечный запас круглых монет радиуса r. Игроки по очереди выкладывают по одной монете на стол так, что она не выходит за границы стола и не пересекает ни одну из уже находящихся на столе монет. Проигрывает тот, кто не может сделать ход.

По заданным параметрам определите, кто победит при оптимальной игре: первый или второй?

Формат входного файла

Первая строка входного файла содержит два натуральных числа R и r ($R \leq 100, r \leq 100$) — радиусы стола и монет.

Формат выходного файла

Выведите слово “First” (без кавычек), если первый игрок может обеспечить себе победу. Иначе выведите слово “Second”.

Пример

j.in	j.out
6 5	First

Разбор задачи J. Игра

Суть этой шуточной задачи состоит в том, что у первого игрока почти всегда есть выигрышная стратегия. Она проста: первым ходом класть монету в центр доски, а затем ходить центрально-симметрично ходам противника. Такой ответ всегда возможен, а общее количество ходов конечно из-за ограниченности доски. Поэтому рано или поздно второй не сможет сделать ход.

Единственный случай, когда это не срабатывает — когда радиус монеты больше радиуса стола. Тогда первый игрок не может походить и сразу проигрывает.

Задача K. Анаграммы

Имя входного файла:	k.in
Имя выходного файла:	k.out
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Анаграммами называются пары слов, составленные из одних и тех же букв. Более формально, у анаграмм совпадают мультимножества букв. Например, “if” и “fi”, “orchestra” и “carthorse” — анаграммы, а “hello” и “world”, “if” и “iff” — нет. Любое слово является анаграммой самому себе.

Задан словарь (множество слов). Найдите размер максимального подмножества, все слова которого являются друг другу анаграммами.

Формат входного файла

Первая строка входного файла содержит натуральное число n — количество слов в словаре ($n \leq 10^4$). Следующие n строк содержат слова из

словаря, состоящие из маленьких латинских букв. Все слова различны, длина каждого не превосходит 100 символов.

Формат выходного файла

Выведите единственное число — искомый размер максимального подмножества анаграмм.

Пример

k.in	k.out
2 hello world	1
5 kolun uklon kloun kulak kulon	4

Разбор задачи K. Анаграммы

Одно из возможных решений было таким. Сначала отсортируем мультимножества букв каждого слова в алфавитном порядке. Тогда анаграммы превратятся в одинаковые строки, а не анаграммы — в разные. Затем нам надо подсчитать максимальное количество одинаковых строк среди полученных таким образом. Это можно сделать с помощью еще одной сортировки и одного прохода по упорядоченной последовательности. Общее время работы составляет $O(nl(\log n + \log l))$ при использовании асимптотически оптимальной сортировки (например слиянием, быстрой, сортировки кучей).

Задача L. Полиномы

Имя входного файла:	`l.in`
Имя выходного файла:	`l.out`
Ограничение по времени:	1 с
Ограничение по памяти:	64 Мб

Маленький Петя любит вычислять значения полиномов в уме. Как только он видит полином второго порядка $P(x) = ax^2 + bx + c$, он тут же вычисляет и записывает его значения в точках $P(1)$, $P(2)$, ..., $P(k)$, ... Этот

процесс мог бы продолжаться бесконечно, но к счастью Петя еще не умеет записывать произвольно большие числа. Максимальное число, которое он может записать, равно D.

Петя вычисляет значение полинома за 1 секунду, а записывает его моментально (если может). Сколько времени потратит Петя на один полином?

Формат входного файла

Первая строка входного файла содержит два натуральных числа n и D ($n \leq 5 \cdot 10^4, D \leq 10^{12}$). Следующие n строк содержат по три целых числа a_i, b_i, c_i — коэффициенты полинома $P_i(x)$ ($1 \leq a \leq 10, -10 \leq b, c \leq 10$).

Формат выходного файла

Для каждого из полиномов $P_i(x)$ выведите в отдельной строке время в секундах, которое потратит на него Петя.

Пример

l.in	l.out
3 10 1 1 1 1 -10 -10 10 10 10	3 12 1

Разбор задачи L. Полиномы

Суть этой задачи сводится к нахождению большего корня квадратного уравнения. При заданных ограничениях подойдет любой разумный метод — начиная от точного вычисления корней за $O(1)$ с последующим округлением и заканчивая двоичным поиском по ответу за время $O(\log D)$. Полный перебор за $\Omega(\sqrt{D})$ в худшем случае не пройдет по времени.

День седьмой (25.02.2010г). Контест Теодора Заркуа

Об авторе...

Теодор Ясонович Заркуа, родился в 1951 году в Тбилиси. Закончил мехмат Тбилисского ГУ. Служил офицером-двухгодичником на Дальнем Востоке. Преподает в ВУЗах с 1975 года. Имеет опыт преподавания информатики в средних школах. Тренирует команды программистов Тбилисского ГУ с 1999 года, а команды Грузинского университета им.Святого Андрея Первозванного Патриаршества Грузии с 2009 года (практически с момента открытия этого ВУЗа). Команды, руководимые Теодором Заркуа, систематически занимают высшие места в масштабах Грузии и Южного Кавказа, успешно участвуют в Открытом Кубке им. Панкратьева, неоднократно занимали призовые места на международных олимпиадах им. Векуа, Поттосина, Глушкова и Лебедева, становились призерами Открытых Всеукраинских олимпиад, Октрытых чемпионатов Москвы, а в 2008 году команда Тбилисского ГУ заняла первое место на Открытой Всеукраинской олимпиаде. В ноябре 2008 года команда ТбГУ в составе Николоз Джимшелеишвили, Эльдар Богданов, Георгий Леквеишвили, тренер Теодор Заркуа, впервые среди команд Южного Кавказа завоевала право участвовать в финале чемпионата Мира по программированию среди ВУЗов. В финале, который состоялся в апреле 2009 года в Стокгольме, эта команда заняла 11-е место и завоевала бронзовые медали.

Основные достижения:

- в 1980,1981 годах участвовал в работах по адаптации операционной системы ДИСПАК на многопроцессорный вычислительный комплекс “Эльбрус”, которые проводились в НИИ Точной Механики и Вычислительной Техники им. Лебедева АН СССР;
- несколько лет был руководителем департамента информационных тех-

нологий одного из крупнейших коммерческих банков Грузии — “Банка Грузии”;

- является Директором международной олимпиады по программированию на Кубок Векуа, Открытого чемпионата Южного Кавказа, а также полуфинала чемпионата Мира по программированию среди ВУЗов для Южнокавказского подрегиона;
- является координатором Открытого Кубка им. Панкратьева по Грузии;
- основатель и Президент Ассоциации Поддержки Программирования (Грузия);
- имеет 19 научно-педагогических публикаций, среди них один учебник по программированию;
- с 2009 года — профессор Грузинского университета им. Святого Андрея Первозванного Патриаршества Грузии.

Имеет среднее музыкальное образование, а также спортивные разряды по гандболу, футболу и шахматам. Женат, трое детей.

Теоретический материал. О позиционных системах счисления

Как известно, позиционные системы счисления отталкиваются от идеи определять величину числа посредством некоторых символов, называемых цифрами, которые составляют запись числа и вносят лепту в значение этого числа в соответствии не только со своим собственным значением, но и в зависимости от позиции, занимаемой этой цифрой, в записи числа.

На самом деле, любая система записи чисел полностью определяется правилом, применимым к любому числу и позволяющим увеличивать его на 1. Действительно, если мы можем получить число на 1 большее любого заданного, это значит, что мы можем получить и число, превосходящее заданное на 2, затем и на 3 и т.д. Т.е. мы можем сказать, что в этом случае полностью владеем операцией сложения. Но раз мы владеем операцией сложения, то мы владеем и операцией умножения — ведь умножение это суть кратное сложение. Кроме того, владея сложением, мы немедленно овладеваем и вычитанием, как операцией, обратной сложению. Для этого достаточно определить $x - y$, как число, которое при увеличении на y дает x:

$$(x - y) + y = x$$

Совершенно аналогично, владея умножением, мы моментально овладеваем возведением в целую степень (кратное умножение), а также делением (операция, обратная умножению)... Таким образом, благодаря умению определять число, непосредственно превосходящее заданное, мы сразу же покрываем весь ассортимент арифметических действий над целыми числами и встаем лицом к лицу с необходимостью расширить множество целых, так как обнаруживаем, что результат деления не всегда удается найти среди целых...

Рассмотрим более детально процесс получения последовательных целых чисел, начиная числом 0. Для простоты, будем рассматривать этот процесс в привычной для нас десятичной системе.

	9999	999
	8888	888
	7777	777
	...	...
	2222	222
	1111	111
...	0000	000

Будем считать, что в нашем распоряжении неограниченное количество вертикальных столбиков, составленных десятичными цифрами, как изображено выше. Будем также считать, что запись числа дает комбинация актуальных цифр столбиков, а актуальными будем считать цифры, расположенные в самом нижнем ряду. Так называемые ведущие нули, как правило, не записываются. Таким образом, стартовая позиция, изображенная выше дает число “0”.

Для того, чтобы получить последующее число, необходимо сделать ход вперед (вниз) на самом правом столбике. При этом все цифры, кроме актуальной, сдвигаются на 1 позицию вниз (т.е. приближаются к актуальной позиции), а цифра, бывшая актуальной до этого хода, оказывается на самом верху столбика (т.е. в позиции, наиболее удаленной от актуальной).

Все остальные столбики не меняют своих значений и в итоге получаем следующую картину:

	9999	990
	8888	889
	7777	778
	...	...
	2222	223
	1111	112
...	0000	001

Таким образом, получаем число “1”. Аналогично после следующего хода получаем “2”, затем “3” и т.д., пока в актуальную позицию не вернется снова “0”. В этот момент полностью повторится стартовая позиция, которая соответствовала числу “0” и наступает очередь делать 1 ход второму столбику. В результате получим такую картину:

	9999	909
	8888	898
	7777	787
	...	...
	2222	232
	1111	121
...	0000	010

Эта картина соответствует числу “10”. Так и должно быть. Действительно, после “9” идет “10”. Продолжая процесс, через 9 ходов получим:

	9999	908
	8888	897
	7777	786
	...	...
	2222	231
	1111	120
...	0000	019

И после следующего хода крайнего правого столбца вновь наступает необходимость сделать ход второму справа столбцу, после чего получаем:

	9999	919
	8888	808
	7777	797
	...	...
	2222	242
	1111	131
...	0000	020

Продолжая этот процесс, в какой-то момент получим:

	9999	988
	8888	877
	7777	766
	...	...
	2222	211
	1111	100
...	0000	099

Здесь следующий ход правого столбика вновь вызывает необходимость сделать ход следующему столбику, но на этот раз он, в свою очередь, делает необходимым сделать ход следующему столбику, и получаем следующую картину:

	9999	899
	8888	788
	7777	677
	...	...
	2222	322
	1111	211
...	0000	100

что соответствует числу “100”.

Из вышеизложенного совершенно очевидно, что, не имея ограничений относительно количества столбцов (которые по мере необходимости будут “пристраиваться” спереди), описанным выше способом можно получить запись любого неотрицательного целого числа. Если циклом ходов столбика будем называть последовательность ходов, начинающуюся ходом, когда нуль находится в актуальной позиции, и заканчивающуюся ходом, после которого нуль вновь возвращается в актуальную позицию (впервые после того, как покинул ее в начале цикла), то можно сказать, что каждый столбик, исключая самый правый, делает 1 ход каждый раз, как его **правый сосед завершает свой очередной цикл**. Соответственно, второй столбик делает ходы в 10 раз реже, чем первый. Но точно такое же соотношение имеет место, если сравнивать второй столбик с его левым соседом, а именно, третий столбик делает очередной ход только в те моменты, когда второй завершает очередной цикл. Поэтому, третий столбец оказывается в 10 раз “медлительнее”, чем второй... Совершенно очевидно, что точно такое же соотношение наблюдается при сравнении скорости выполнения ходов каждого столбика с предыдущим. А если сравнивать столбик с предыдущим

предыдущего (разумеется, если таковой существует), то, очевидно, окажется, что он медленнее его уже в 100 раз. И более обобщенно, столбик расположенный левее некоторого другого на n позиций будет более "тяжеловесным" по сравнению с ним в 10^n раз. Таким образом получаем, что появление 1 в актуальной позиции второго столбца означает то, что уже было сделано 10 ходов первым столбцом, т.е. было добавлено единиц 10 раз. А появление цифры 7 в актуальной позиции второго столбца означает то, что к этому моменту единиц было добавлено 70 раз. Учитывая все вышесказанное, мы можем получить привычную расшифровку десятичной записи любого числа. К примеру,

$$573509 = 9 * 1 + 0 * 10 + 5 * 10^2 + 3 * 10^3 + 7 * 10^4 + 5 * 10^5.$$

Как видим, все предельно просто — вес самой правой позиции (по сути, коэффициент, на который умножается самая правая цифра) всегда равен 1, а каждая следующая позиция в 10 раз тяжелее предыдущей (при движении справа налево). По большому счету здесь для нас суть важно, что:

1. каждый столбик составлен из 10 символов (цифр);
2. все цифры каждого отдельного столбика различны.

После всего вышесказанного не составит труда заметить, что процесс, аналогичный вышеописанному, вполне можно осуществить и в том случае, если наши столбики составлены из количества цифр, которое не равно 10. Пусть, к примеру, в нашем распоряжении столбики, составленные из 7 цифр. Тогда абсолютно аналогичный процесс зафиксирует то, что каждый последующий столбик в 7 раз "тяжелее" предыдущего. Поэтому расшифровка семеричной записи наугад взятого числа будет иметь такой вид:

$$(523224)_7 = 4 + 2 * 7 + 2 * 7^2 + 3 * 7^3 + 2 * 7^4 + 5 * 7^5 \qquad (1)$$

Если выполнить действия над правой частью, то получим десятичную запись этого же числа. Точно таким же способом можно расшифровать (перевести в десятичный вид) любое число, записанное с использованием столбиков, состоящих из 7 цифр. Если же для записи числа использовали столбики, состоящие из, скажем, 9 цифр, тогда при расшифровке записи (переводе в десятичный вид), веса позиций будут выражены степенями числа 9. И, вообще, если для записи числа использовались столбики из p цифр, то при переводе такой записи в десятичную систему, веса позиций окажутся равными степеням числа p, причем степень, в которую будет возводиться p, будет равна номеру позиции, отсчитанному справа налево, при условии, что отсчет начался с нуля. Наверняка, аудитории известно, что

p принято называть основанием позиционной системы счисления. Таким образом, можно говорить, что имеются позиционные системы счисления с разными основаниями. В частности, мы успели рассмотреть системы с основаниями 10, 7, 9. Когда система счисления имеет основание p ($p > 1$), то для записи чисел в этой системе используется ровно p символов (цифр), значения которых покрывают диапазон целых чисел от 0 до $p-1$. Обычно, в качестве цифр используются символы, применяемые для записи десятичных чисел, но когда их не достаточно, в качестве цифр могут быть привлечены символы какой-либо другой группы. В частности, для обозначения цифр распространенной в среде специалистов информационных технологий шестнадцатеричной системы, значения которых превосходят 9, принято использовать необходимое количество первых букв латинского алфавита. Выполним записанные в (1) действия. Получим $(523224)_7 = 89982$.

А теперь допустим, что нам было необходимо число 89982 перевести в семеричную форму записи. Как мы могли это сделать? Ответ содержится в правой части равенства (1). Перепишем ее, вынося за скобки 7 везде, где это возможно. Получим:

$$89982 = 4 + 7 * (2 + 7 * (2 + 7 * (3 + 7 * (2 + 7 * 5))))$$

Заметим, что данная запись является более экономичной с точки зрения организации вычислений, так как позволяет вычислять степени числа 7 не с самого начала, а с использованием уже вычисленной степени, показатель которой на 1 меньше по сравнению с вычисляемой. Именно на этой идее основана известная схема Горнера (*William George Horner*) вычисления многочленов.

Вернемся к составленному выражению. Легко замечаем, что если мы возьмем остаток от деления числа 89982 на 7, то получим крайнюю правую цифру искомой семеричной записи этого числа. Если затем точно так же поступить с результатом деления этого числа на 7, то получим уже соседнюю цифру искомой записи и так до тех пор, пока в результате очередного деления не получим число, меньшее чем 7. Выписывая в столбик справа — остатки, а слева — результаты деления, получим:

89982	4
12854	2
1836	2
262	3
37	2
5	5

Выписывая полученные справа цифры снизу вверх (вспомним, что мы получали цифры искомой записи справа налево) в итоге получаем искомую

семеричную форму $(523224)_7$. Однако зададимся вопросом — чем лучше десятичная форма записи по сравнению с другими формами записи, кроме того, что мы к ней уже привыкли. Если мы будем до конца объективны, то должны будем однозначно ответить — "ничем". Единственное преимущество десятичной системы — это наличие 10 пальцев на руках, позволяющее прибегать к их помощи при устном счете. Кстати, в названиях чисел в грузинском языке явно присутствуют следы двадцатеричной системы. По-грузински, 40 — это "ормоци", что означает дважды двадцать, 60 — это "самоци" (трижды двадцать), 70 — "самоцдаати", что означает трижды двадцать и десять (один — по-грузински "ерти", два — "ори", три — "сами", десять — "ати", двадцать — "оци", а "да" означает "и"). Учитывая южное месторасположение Грузии, логично предположить, что было время, когда предки современных грузин помогали себе при подсчете пальцами не только рук, но и ног.

Для того, чтобы окончательно утвердиться в отсутствии особых преимуществ десятичной системы, посмотрим, как можно было осуществить приведенные выше преобразования из десятичной в семеричную и наоборот, **пользуясь только семеричной системой**.

Для этого, вначале составим таблицу умножения для семеричной системы:

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	11	13	15
3	3	6	12	15	21	24
4	4	11	15	22	26	33
5	5	13	21	26	34	42
6	6	15	24	33	42	51

Теперь попытаемся расшифровать 89982, пользуясь вышеизложенным методом, но производя вычисления исключительно в семеричной системе.

Воспользуемся экономичной схемой и запишем выражение, которое следует вычислить в семеричной системе (для большей ясности воспользуемся квадратными скобками):

$$89982 = 2+10*(8+10*(9+10*(9+10*8))) = [2+13*(11+13*(12+13*(12+13*11)))]_7$$

Мы уже использовали, что $8 = (11)_7$, $9 = (12)_7$, $10 = (13)_7$.

В дальнейшем процессе вычислений будем пользоваться приведенной выше таблицей умножения, а также известными правилами вычисления суммы и произведения многозначных чисел. Например, выражение в самых внутренних скобках будет вычисляться следующим образом:

$$
\begin{array}{r}
13 \\
* \\
11 \\
\hline
13 \\
+ \\
13 \\
\hline
143 \\
+ \\
12 \\
\hline
155
\end{array}
$$

Как видим, здесь внешне пока нет никаких отличий от вычислений в десятичной системе. В результате на данный момент будем иметь:

$$[2+13*(11+13*(12+13*(12+13*11)))]_7 = [2+13*(11+13*(12+13*155))]_7$$

Вычисляем дальше:

$$
\begin{array}{r}
155 \\
* \\
13 \\
\hline
531 \\
+ \\
155 \\
\hline
2411 \\
+ \\
12 \\
\hline
2423
\end{array}
$$

Дальнейшие вычисления приведем без подробностей:

$$[2 + 13 * (11 + 13 * (12 + 13 * 155))]_7 = [2 + 13 * (11 + 13 * 2423)]_7 =$$

$$= [2 + 13 * (11 + 35132)]_7 = [2 + 13 * 35143]_7 = [2 + 523222]_7 = [523224]_7.$$

Возможно, кому-либо такой способ перевода из десятичной в семеричную покажется более удобным. . .

Перевод числа $(523224)_7$ в десятичную систему пользуясь семеричной системой будет выглядеть следующим образом:

$$
\begin{array}{r|l}
523224 & 2 \\
35143 & 11 \\
2423 & 12 \\
155 & 12 \\
11 & 11
\end{array}
$$

Теперь остается выписать полученные цифры искомого десятичного представления, беря их в последовательности снизу вверх.

Очевидно, для физической реализации наиболее удобной является двоичная система, в которой для изображения любого числа достаточно всего 2-х цифр.

Здесь же отмечаем фактическое отсутствие таблицы умножения для двоичной системы как таковой! Умножение на 0 всегда дает 0, а умножение на 1 дает значение другого сомножителя.

Заметим, что хорошее владение двоичной системой облегчает решение целого ряда задач.

Например, пусть биологи проводят серию экспериментов над n особями. Причем необходимо проверить гипотезу о том, что результат может сильно зависеть от набора особей, взятых для конкретного сеанса. Разумеется, тут важно суметь рассмотреть все подмножества рассматриваемого множества особей. Для этого проще всего, перенумеровав все объекты эксперимента числами, начиная с нуля, каждому из подмножеств поставить в соответствие двоичную комбинацию, в которой в разрядах, имеющих номера присутствующих в этом подмножестве особей, стоят единицы, а во всех прочих разрядах — нули. Таким образом, для рассмотрения всех возможных подмножеств достаточно просто перебрать все числа, начиная нулем и кончая числом 2^{n-1} и для каждого из них выбирать в множество те и только те особи, которые имеют номера, которым соответствуют единицы в двоичном представлении текущего числа.

Например, для трех особей (a,b,c) будем иметь следующее соответствие между числами (по сути, номерами подмножеств) и подмножествами (для удобства справа приписаны трехзначные двоичные представления этих чисел):

0	{}	000
1	{a}	001
2	{b}	010
3	{a,b}	011
4	{c}	100
5	{a,c}	101
6	{b,c}	110
7	{a,b,c}	111

Однако, в тех случаях, когда экспериментатору важно подметить какое изменение повлияло на результат, очевидно, приведенный вариант не подойдет. Действительно, в этом случае желательно, чтобы полученная последовательность была как можно более гладкой в том смысле, чтобы

каждый последующий член последовательности как можно меньше отличался от предыдущего, а у нас, к примеру, подмножества с номерами 1 и 2, а также 5 и 6 отличаются двумя составляющими, а подмножества с номерами 3 и 4, вообще всеми тремя составляющими.

Очевидно, нам бы подошел следующий вариант:

0	000	{}	000
1	001	{a}	001
2	010	{a,b}	011
3	011	{b}	010
4	100	{b,c}	110
5	101	{a,b,c}	111
6	110	{a,c}	101
7	111	{c}	100

Легко почувствовать, что уже для $n = 4$ “кустарными” способами такую последовательность составить не так уж легко. К счастью, существует остроумная формула, предложенная Фрэнком Греем (*Franc Grey*), которая носит его имя и решает проблему формирования подобной последовательности, позволяя прямо определить член искомой последовательности, зная его порядковый номер.

Если обозначить i-ый разряд двоичного представления порядкового номера кода Грея через N_i, а через G_i - i-ый разряд двоичного представления соответствующего кода Грея, то имеет место формула:

$$G_i = N_i \wedge N_{i+1}$$

Отметим, что разряды считаются занумерованными, как и раньше, справа налево (с нуля). Функция на языке C, реализующая получение кода Грея для заданного целого параметра, равного номеру этого кода, будет выглядеть так:

```
unsigned long long G(long long N)
{return N ^ N>>1; }
```

Вспомним, что сдвиговые операции имеют более высокий приоритет, чем разрядное сложение по модулю 2.

Бывает важно быстро определить мощность множества, представленного двоичной комбинацией. В этом случае можно использовать то обстоятельство, что для данного целого $N(N > 0)$ операция разрядного умножения на $N - 1$ дает двоичную комбинацию, отличается от исходной только

тем, что самый правый разряд, содержащий единицу, обнуляется. Таким образом, программный код, определяющий количество единиц в двоичном представлении числа без знака, заданного в качестве параметра, на языке C будет выглядеть так:

```
unsigned long long G(long long N)
{int R;
for(R=0; N; N &= N-1, R++);
return R; }
```

Замечательным образом используются особенности двоичного кода при создании целого ряда информационных структур и алгоритмов. В частности, стоит рассмотреть остроумную идею организации сумматора — информационной структуры, которая обеспечивает подсчет суммы членов отрезка последовательности (т.е. членов последовательности, имеющих номера из заданного диапазона). Основная идея заключается в том, чтобы вместо самих членов последовательности хранить суммы членов из диапазона, начинающегося членом, номер которого получится, если в двоичном представлении номера текущего члена обнулить самую правую единицу и потом добавить 1. В частности, все члены с нечетными номерами будут равны члену с таким же номером исходной последовательности. Например, если обозначить члены исходной последовательности через A_i, тогда первые несколько членов "рабочей" последовательности B будут получены так:

$B_1 = A_1$
$B_2 = A_1 + A_2$
$B_3 = A_3$
$B_4 = A_1 + A_2 + A_3 + A_4$
$B_5 = A_5$
$B_6 = A_5 + A_6$
$B_7 = A_7$
$B_8 = A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8$
...

Пусть $SUM(A, n, m)$ означает сумму всех членов последовательности A, имеющих индексы от m до n включительно. Тогда, $SUM(A, n, m)$, где $n < m$ можно будет получить следующим образом $SUM(A, 1, m) - SUM(A, 1, n - 1)$.

При этом процедура $SUM(A, 1, n)$ будет выполняться за логарифмическое, относительно n, время, так как может схематически выглядеть так. Вначале берем в сумму B_n, затем обнуляем самый правый разряд двоичного представления n и если результат не нуль, то берем B с индексом,

равным полученному числу и т.д., до тех пор, пока не получим нуль. Так как движение происходит в худшем случае продвижением по всем разрядам двоичного представления n, то вышеприведенная оценка оказывается обоснованной. Например, для вычисления суммы первых 7 элементов приходится суммировать B_7, B_6, B_4.

Естественно, весь этот сыр-бор может быть оправдан, если существует приемлемый способ модифицировать последовательность B, когда возникает необходимость изменить (для определенности сложением) A_i. Очевидно, в этом случае нужно изменить соответствующим способом все члены рабочей последовательности, значения которых получаются с участием Ai. Таких не больше, чем единиц в двоичном значении числа i, а пройтись по всем членам "рабочей" последовательности начиная с того, номер которого совпадает с i можно осуществляя операцию, обратную той, с помощью которой мы продвигались, когда получали начальные индексы суммируемых членов исходной последовательности, когда формировали последовательность B. Например, при изменении A_6, будет необходимо "затронуть" B_6, B_8 и т.д. до тех пор, пока не исчерпаются все члены "рабочей" последовательности (естественно, мы проходим только по "нужным" членам). Для двоичного представления индекса, для которого следует получить последующий, эта процедура выглядит следующим образом — взять самую крайнюю единицу и добавить число, двоичное представление которого содержит единственную единицу именно в этой позиции. На языке C функция, которая выдает номер следующего элемента "рабочей" последовательности, выглядит так:

```
unsigned long long NEXT(unsigned long long n)
{return (n<<1) - (n & (n-1));  }
```

Совершенно очевидно, что подобная структура эффективна не только для получения суммы элементов некоторого диапазона последовательности, но и для получения таких численных характеристик интервалов последовательности, которые однозначно реагируют на изменение каждого элемента исходной последовательности (таковыми являются, например, такие характеристики, как максимум и минимум).

Отметим, что двоичная система имеет один недостаток — эта система значительно более требовательна к количеству разрядов. Иными словами, проведение вышеописанного процесса получения последовательности целых чисел в двоичной системе чаще других приводит к необходимости увеличивать разрядность. Это обстоятельство привело к появлению спроса на системы с основаниями, равными целым степеням двойки. В частности, в качестве системы для представления информации, представленной в ком-

пьютере, часто используются либо шестнадцатеричная, либо восьмеричная системы.

Обе эти системы не так требовательны к разрядности представленных ими чисел, одновременно перевод из двоичной в эти системы и наоборот можно осуществлять максимально просто — посимвольно. При этом одной шестнадцатеричной цифре соответствует секция из 4-х двоичных цифр и наоборот ($16 = 2^4$), а одной восьмеричной цифре соответствует секция из 3-х двоичных цифр и наоборот ($8 = 2^3$).

Действительно, $(1011010110001)_2 = (13261)_8 = (16B1)_{16}$.

Легко убедиться, что подобная тесная связь существует между любыми двумя системами, если основание одной из них является целой степенью основания другой.

Теперь давайте ненадолго вернемся к “нашим” столбикам. Мы уже заметили, что изменение основания позиционной системы просто означает соответствующее изменение количества элементов наших столбиков. Но, в принципе, не составляет труда заметить, что суть процесса продвижения по подряд идущим целым числам не изменится, если столбики содержат разное количество элементов. Например, для системы, в которой количество элементов столбиков справа налево равно 2, 3, 2,. . . , первые несколько комбинаций (записей чисел) будут выглядеть так (для определенности, дадим по три разряда):

000	0
001	1
010	2
011	3
020	4
021	5
100	6
101	7
110	8
111	9
120	10

Для удобства, справа приписаны десятичные значения этих записей.

Как видим, главный принцип, заложенный в позиционные системы, в соответствии с которым каждая следующая позиция “тяжелее” предыдущей во столько раз, каково основание этой позиции, срабатывает и при разных основаниях. В частности, если мы имеем $(n + 1)$-значную запись, $(A_n \dots A_1 A_0)$, для которой основания позиций даются последовательностью

$B_0, B_1 \dots B_n$, соответствующая расшифровка будет иметь вид:

$$(A_n \dots A_1 A_0) = A_0 + B_0 * (A_1 + B_1 * (A_2 + \dots (A_{n-1} + B_{n-1} * A_n) \dots))$$

Для приведенного примера, в частности, $(110) = 0 + 2 * (1 + 3 * 1)$.

Такие системы принято называть системами с неоднородным основанием. Заметим, что на практике мы постоянно имеем дело с такими системами. В частности, время задается именно такой системой (секунды и минуты в шестидесятеричной, часы в двадцатеричной, дни, вообще, в системе с переменным основанием, значение которого зависит от месяца, а в некоторых случаях и от года, месяцы в двенадцатеричной системе. . .).

Регламент данной лекции не позволяет затронуть множество полезных и интересных вопросов, связанных с системами счисления. Будем считать, что это удовольствие у нас впереди.

Задачи и разборы

Задача A. АнтиГрей

Имя входного файла:	`a.in`
Имя выходного файла:	`a.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Как известно, английский математик Фрэнк Грей (Franc Grey) предложил остроумную формулу, которая позволяет по порядковому номеру определить соответствующий член последовательности, обладающей тем свойством, что двоичное представление каждого последующего члена отличается ровно в одном разряде от двоичного представления предыдущего. Если обозначить i-ый разряд двоичного представления порядкового номера кода Грея через N_i, а через G_i - i-ый разряд двоичного представления соответствующего кода Грея, то имеет место формула:

$$G_i = N_i \wedge N_{i+1}$$

Отметим, что разряды считаются занумерованными справа налево (с нуля). Функция на языке C, реализующая получение кода Грея для заданного целого параметра, равного номеру этого кода будет выглядеть так:

```
unsigned long long G(long long N)
{return N ^ N>>1;}
```

Последовательность (G_i) будем называть последовательностью Грея. Наша задача, по заданному коду Грея получить число, которому этот код соответствует (т.е. номер члена последовательности Грея, равного этому коду).

Формат входного файла

Файл состоит из единственной строки — кода Грея, заданного в шестнадцатеричном виде. Длина заданной строки не превосходит 200 000, в качестве цифр, значения которых превосходят 9, взяты первые английские буквы верхнего регистра.

Формат выходного файла

Выходной файл состоит из единственной строки - шестнадцатеричного значения числа, код Грея которого был задан во входном файле. У результата не должно быть ведущих нулей, в качестве цифр, значения которых превосходят 9, следует взять первые английские буквы верхнего регистра.

Примеры

a.in	a.out
3F	2A
5B9	6D1

Разбор задачи A. АнтиГрей

Так как исходное число задано в шестнадцатеричной системе, то за один проход поциферным переводом можно получить соответствующий двоичный код, преобразовать его, а затем уже посекторным переводом получить искомый результат в требуемом виде. Что касается самого преобразования, то вначале заметим, что если обозначить самый старший двоичный разряд искомого числа через N_n, а соответствующий разряд кода Грея G_n, то получим:

$$N_n = G_n$$

Дальше согласно формуле Грея будем иметь:

$$G_{n-1} = N_{n-1} \wedge N_n,$$

отсюда

$$N_{n-1} = G_{n-1} \wedge N_n,$$

Очевидно, пользуясь последней формулой, двоичный результат можно получить за один проход.

Задача B. Снова A + B

Имя входного файла:	`b.in`
Имя выходного файла:	`b.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Рассмотрим множество строк, составленных только из латинских букв малого регистра и десятичных цифр. Назовем две такие строки подобными, если:

1. Они имеют одинаковую длину;
2. Элементы с одинаковыми индексами у этих строк: либо оба — буквы, либо оба — цифры.

На множестве всевозможных строк, подобных данной, вводится операция сложения. Если упорядочить все строки, подобные данной в порядке обратном лексикографическому, тогда все эти строки могут быть снабжены своими порядковыми номерами, начиная с нуля. Пусть $N(A)$ — это порядковый номер строки A, а $N(B)$ — порядковый номер строки B, тогда суммой строк A и B будем считать строку, порядковый номер которой в вышеописанной последовательности будет равен $(N(A) + N(B)) \bmod M$, где M — общее количество строк, подобных строкам, участвующим в операции сложения.

Ограничения

Длина каждого слагаемого не меньше 1 и не превосходит 300 000. Тесты гарантируют, что слагаемые удовлетворяют вышеизложенному условию подобия.

Формат входного файла

Первая строка содержит A, а вторая строка содержит B.

Формат выходного файла

Выходной файл состоит из единственной строки — результата сложения заданных строк.

Примеры

b.in	b.out
z y	y
7b 1c	8e

Разбор задачи B. Снова A + B

Согласно условию, мы имеем дело с позиционной системой счисления с неоднородным основанием. Некоторые позиции имеют основание 26 (символьные позиции), а некоторые позиции имеют основание 10 (цифровые позиции). Причем, символы на столбиках расположены в убывающей последовательности. Т.е. в роли нуля в соответствующих позициях выступают "z" и "9". Таким образом, если символ в строке соответствует букве, тогда его порядковый номер в начальной позиции соответствующего столбика (снизу вверх начиная от нуля) равен $25 - (s - `a')$, а если это цифра, то его аналогичный номер равен $9 - (s - `0')$, где s — это код упомянутого символа. А то, что номер результирующей строки определяется путем взятия остатка от деления на общее количество таких строк, означает просто потерю данных выходящих за пределы длины слагаемых строк.

Задача C. Площадь

Имя входного файла:	`c.in`
Имя выходного файла:	`c.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Для заданной функции $F(x) = a_0 \times f_0(b_0 \times x) + \ldots + a_n \times f_n(b_n \times x)$, а также, для заданных A и B, определить ориентированную площадь, ограниченную графиками функций $y = F(x)$, $y = -C$, $x = A$, $x = B$. Где $0 \le n \le 25$, $-25 \le a_i, b_i \le 25$, $b_i \,!= 0$, $C = \text{abs}(a_0/b_0) + \ldots + \text{abs}(a_n/b_n)$, $-1000 \le A, B \le 1000$, а каждая из f_i либо sin, либо cos.

Формат входного файла

В начале файла идут значения чисел A и B, а затем строка, содержащая $F(x)$, причем эта строка пробелов не содержит, кроме того a_i и b_i могут быть пропущены (естественно, в этих случаях их значения должны быть приняты равными 1). Аналогично, $-\sin(-x)$ следует воспринимать как $-1*\sin(-1*x)$.

Формат выходного файла

Выходной файл состоит из единственной строки, в которой содержится искомая площадь с точностью до 6 дробных десятичных цифр.

Примеры

c.in	c.out
-15 27 5*sin(2*x)+6*cos(3*x)	191.900934
0 3.14 -sin(x)	1.140001

Разбор задачи C. Площадь

Следует упорядочить A и B по возрастанию и вычислить результат по формуле:

$$S = \int_A^B (F(x) + C)dx$$

Остается аккуратно получить необходимые значения чисел a_i и b_i, а затем правильно применить формулы получения первообразных для функций $\sin(x)$ и $\cos(x)$.

Задача D. Пустыня

Имя входного файла:	d.in
Имя выходного файла:	d.out
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Требуется определить минимально необходимый запас топлива в стартовой точке, для того, чтобы на вездеходе преодолеть трассу заданной длины, проходящую через пустыню при условии, что каждый раз можно заправляться только в бак, размер которого известен заранее, при условии, что расход топлива равен одному литру на километр, что можно брать с собой неограниченное количество пустых резервуаров, устанавливать их в любом месте и заливать в них горючее только из бака.

Формат входного файла

Во входном файле через пробел записаны два числа — длина трассы в километрах и размер бака вездехода в литрах. Данные в тестах гаранти-

руют, что отношение чисел, выражающих длину трассы и размер бака не превосходит 5.

Формат выходного файла

В выходном файле единственное число — ответ задачи с точностью до 6 знаков после десятичной точки.

Примеры

d.in	d.out
10 3	330.886044
499 123	57670.182247

Разбор задачи D. Пустыня

Известная задача с красивым и поучительным решением. Если размер бака (B) не меньше длины трассы (D), то ответ — длина трассы (D). Иначе зададимся вопросом — каково наибольшее расстояние до конца трассы, стартуя с которого при заполненном баке можно преодолеть трассу до конца. Очевидно, это расстояние равно размеру бака. Обозначим его x_1 ($x_1 = B$). После этого следует пытаться ответить на вопрос — а с какого наибольшего расстояния от точки, отстоящей от конца на расстояние x_1 можно стартовать, имея в точке старта два бака с горючим, чтобы прийти к ней с полностью опустошенным баком, обеспечить наличие там B литров горючего, заправиться в этой точке и пересечь пустыню до конца. Если обозначить это расстояние через x_2, то очевидно будем иметь: $2 \times B - 3 \times x_2 = B$. Здесь мы учли, что одной ходки для того, чтобы доставить B литров горючего в точку $D - x_1$ явно недостаточно, а каждый поход длиной x_2 расходует x_2 литров горючего (таких походов всего 3 — два туда и один обратно). В результате имеем $x_2 = B/3$. Теперь пытаемся определить максимальное от полученной точки расстояние до точки, стартуя с которой с запасом $3 \times B$ можем прибыть в точку $D - x_1 - x_2$ с пустым баком и обеспечить наличие в этой точке $2 \times B$ литров горючего. Очевидно, получим $x_3 = B/5$ (только двукратного выезда из точки $D - x_3$ для доставки в $D - x_1 - x_2$ $2 \times B$ литров горючего явно недостаточно). В какой-то момент очередная точка либо совпадет с точкой старта, либо окажется перед стартовой точкой. В первом случае ответ уже готов, а во втором случае придется решить уже тривиальную задачу — определить минимальное стартовое количество горючего, которое позволит оказаться в предыдущей точке с минимально необходимым для дальнейшего успешного продвижения количеством горючего. Ведь мы знаем расстояние от

старта до последней определенной нами точки, а также необходимое количество полных баков горючего для старта от этой точки.

Задача E. Больше всех

Имя входного файла:	`e.in`
Имя выходного файла:	`e.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Для заданной последовательности целых чисел и заданного основания системы счисления p $(0 \leq p \leq 16)$ определить цифру, которая присутствует в p-ичных значениях большего числа членов последовательности. Если претендентов несколько, то выдать наименьшее среди них.

Формат входного файла

В начале файла идет значение числа p, а затем подряд идут члены исследуемой последовательности. Все числа во входном файле задаются в десятичном виде.

Формат выходного файла

Выходной файл состоит из единственной строки - цифры, встретившейся в p-ичных значениях большего числа членов заданной последовательности. Результат выдается в терминах p-ичной системы. Причем, **при необходимости использовать латинские буквы, следует брать буквы верхнего регистра.**

Примеры

e.in	e.out
10 535500222 3777660099 55532222	3

Разбор задачи E. Больше всех

Важно организовать подсчет “очков”, полученных цифрами-претендентами таким образом, чтобы цифра, присутствующая в данном числе, получила от него только одно очко, а не присутствующая — 0. При просмотре чисел и добывании цифр нужно иметь в виду, что в реализациях, как правило, остатки от деления отрицательных чисел на

положительные получаются отрицательные результаты (вопреки математическому определению). Следует учитывать риск недодать очков нулю, если не обрабатывать отдельные числа циклом, предусматривающим выполнение тела цикла хотя бы один раз. Чисто технически: вводится массив для накопления очков, полученных цифрами, а затем просто надо будет определить индекс максимального значения этого массива.

Задача F. Сколько префиксных?

Имя входного файла:	`f.in`
Имя выходного файла:	`f.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Как известно, еще в 20-е годы прошлого столетия польский математик Ян Лукасевич (*Jan Lukasiewicz*) предложил бесскобочные формы записи алгебраических выражений, называемые в его честь польскими записями. Префиксная польская запись получается путем выставления знака операции перед соответствующими (соответствующим) операндами (операндом). Например, если имеем инфиксное выражение $(b-c/d)/(e*f-(g+h*k))$, то префиксной формой фрагмента "c/d" будет "$/cd$", префиксной формой фрагмента "$b-c/d$" будет "$-b/cd$". Префиксной формой фрагмента "$e*f$" будет "$*ef$", фрагмента "$h*k$" будет "$*hk$", а фрагмента "$g+h*k$" - "$+g*hk$". Тогда выражению "$e*f-(g+h*k)$" будет соответствовать префиксная запись "$-*ef+g*hk$", и рассматривая полученные префиксные записи как операнды заключительной операции — деления, окончательно получим: "$/-b/cd-*ef+g*hk$".

Перед нами стоит задача по заданному целому $N(1 <= N <= 50)$ определить число, равное общему количеству всевозможных префиксных выражений длины N, содержащему только двуместные операции '+' '-' '*' '/', а также необходимое количество неповторяющихся первых букв древнегрузинского, либо последних букв современного украинского алфавитов, либо неповторяющихся и тех и других, **взятое по модулю 1 000 009**, при условии, что всевозможные префиксные выражения, удовлетворяющие приведенным условиям, упорядочены в порядке получения больших значений, если в качестве операндов взято **необходимое количество подряд идущих цифр девятеричного представления числа Непера, начиная с 753-й цифры дробной части**. Для того, чтобы исключить неоднозначность толкования подчеркнем, что искомое число подсчитывается для **фиксированного набора необходимого количества неповторяющихся букв**.

Примечание

Будем считать доказанным тезис о пустоте множества общих букв современного украинского и древнегрузинского алфавитов на данный момент.

Формат входного файла

Файл содержит одну строку — число N.

Формат выходного файла

Файл содержит единственное число (разумеется целое :)) - искомый результат.

Примеры

f.in	f.out
1	1
5	192

Разбор задачи F. Сколько префиксных?

В первую очередь замечаем, что каждому префиксному выражению взаимнооднозначно соответствует постфиксное выражение. Замечаем также, что результат можно получить, не зная особенностей современного украинского и древнегрузинского алфавитов, также без знания разложения девятеричного значения числа Непера... Замечаем также, что для четных N — ответ 0. Для нечетных же N сразу обозначим $n = N/2$. Тогда легко заметить, что искомый ответ получается формулой $(A{*}B{*}C)\ mod\ 1\,000\,009$, где A — это количество всевозможных двоичных комбинаций длины $2 * n$, в которых по n единиц и нулей, и ни в какой момент количество нулей не превосходит количество единиц, что соответствует числу Каталана C_n. Поясняю — так как мы заменили поиск количества префиксных записей поиском количества постфиксных записей, то сразу замечаем, что первый элемент записи обязательно операнд, а дальше количество операций не должно стать равным количеству операндов, а в итоге операндов должно быть ровно на 1 больше, чем операций. Поэтому приведенным выше способом мы определяем количество возможных структур постфиксных записей длины N в разрезе взаиморасположения обезличенных операндов и операторов. Понятно, что операнды мы отождествили с единицами, а операторы с нулями.

B — это количество всевозможных перестановок из $n+1$ элементов (каждое из этих перестановок будет соответствовать определенному взаиморасположению фиксированного набора из $n+1$ неповторяющихся операндов — тут важно не забыть о первом операнде соответствующего постфиксного выражения). C — это количество расположений четырех операций (возможно повторяющихся) по n местам. Соответственно, $B = (n+1)!$, $C = 4^n$.

Задача G. Из префиксного в инфиксное

Имя входного файла:	`g.in`
Имя выходного файла:	`g.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Как известно, еще в 20-е годы прошлого столетия польский математик Ян Лукасевич (*Jan Lukasiewicz*) предложил бесскобочные формы записи алгебраических выражений, называемые в его честь польскими записями. Префиксная польская запись получается путем выставления знака операции перед соответствующими (соответствующим) операндами (операндом). Например, если имеем инфиксное выражение $(b-c/d)/(e*f-(g+h*k))$, то префиксной формой фрагмента "c/d" будет "$/cd$", префиксной формой фрагмента "$b-c/d$" будет "$-b/cd$". Префиксной формой фрагмента "$e*f$" будет "$*ef$", фрагмента "$h*k$" будет "$*hk$", а фрагмента "$g+h*k$" - "$+g*hk$". Тогда выражению "$e*f-(g+h*k)$" будет соответствовать префиксная запись "$-*ef+g*hk$", и рассматривая полученные префиксные записи как операнды заключительной операции — деления, окончательно получим: "$/-b/cd-*ef+g*hk$".

Перед нами стоит задача по заданному префиксному выражению получить соответствующее инфиксное, удовлетворяющее следующим условиям:

1. Все операнды исходного выражения участвуют в инфиксной форме, причем именно в той последовательности, в которой они шли в заданном выражении;

2. В результирующем выражении скобки используются только в необходимых случаях (т.е. когда без скобок смысл выражения другой).

Ограничения

В исходном выражении нет пробелов, в качестве операндов использованы латинские буквы малого регистра, в качестве операций только двуместные операции “+” “-” “*” “/” . Длина исходного выражения не превосходит 50. Тесты гарантируют, что в исходном выражении ошибок нет.

Формат входного файла

Файл содержит одну строку — исходное префиксное выражение.

Формат выходного файла

Файл должен содержать единственную строку — результат преобразования.

Примеры

g.in	g.out
-bc	b-c
/b/cd	b/(c/d)

Пояснение

В последнем примере, в принципе, возможен вариант вообще без скобок: $b/c * d$, но такой ответ противоречит условию (а именно, пункту 1).

Разбор задачи G. Из префиксного в инфиксное

В первую очередь следует вспомнить, что если отреверсировать префиксную запись, то получим запись, сопряженную к соответствующей постфиксной записи. Как известно, алгоритм восстановления исходной инфиксной записи из записи, сопряженной к постфиксной отличается от стандартного алгоритма восстановления инфиксной записи на основе постфиксной только тем, что при обработке операций, операнды считываются из стека в порядке, соответствующем их появлению в инфиксной записи. Я позволю себе привести этот алгоритм:

1. Взять пустой стек.
2. Взять очередной элемент исходного выражения. Если это невозможно (т.е. исходное выражение исчерпалось), то перейти к пункту 5.
3. Если очередной элемент является операндом, то поместить его в стек и перейти к пункту 2.

4. Т.к. очередной элемент является знаком операции, то считать из стека необходимое количество операндов (сколько необходимо для данной операции), сформировать элемент, соответствующий выполнению текущей операции над считанными операндами с учетом получения операндов из стека в порядке их появления в инфиксном выражении, записать этот элемент в стек и перейти к пункту 2.

5. Взять результат из стека, завершить выполнение алгоритма.

Осталось аккуратно действовать для удовлетворения требования о минимуме скобок. Для этого удобно рабочий стек сопроводить информацией об операции, являющейся самой последней (верхней) для соответствующего элемента стека, после чего необходимость обрамления элементов, считанных из стека круглыми скобками, можно определить в соответствии со следующей таблицей:

Обрабатываемая операция	Последняя операция в первом операнде					Последняя операция во втором операнде				
	#	+	-	*	/	#	+	-	*	/
+										
-							( )	( )		
*		( )	( )				( )	( )		
/		( )	( )				( )	( )	( )	( )

Знак “#” означает отсутствие какой-либо операции в соответствующем операнде. Смысл остальных элементов таблицы очевиден.

Задача H. Равносильность

Имя входного файла:	`h.in`
Имя выходного файла:	`h.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Школьник Бека обнаружил, что одно и то же соотношение можно записать по-разному. Например: “x>=25” можно записать и как “x-25>=0”, и как “25<=x” и даже как “$2 * x - 3 * y - 12 >= x - y * 3 + 13$”. Он стал проверять равносильность различных соотношений, приведенных в одном толстом задачнике по математике. Напомним, что два соотношения называются равносильными, если для любого комплекта значений переменных, использованных в них, либо оба эти соотношения верны, либо оба они не верны (т.е. если их т.н. истинностные значения равны всегда).

Задавшись целью проверить эквивалентность абсолютно всех соотношений из книги, Бека вскоре понял, что ему необходима программа, которая

поможет ускорить эту работу. Программировать он еще не умеет, поэтому просит о помощи Вас.

Нужно составить программу, которая по заданным двум соотношениям выдаст заключение об их равносильности на английском языке — либо YES, либо NO. Каждое из соотношений задано в виде строки. В качестве знака соотношения может быть один из следующих: “<” “<=” “>” “>=” “=” “<>”. В качестве операндов выражений, составляющих левую и правую части соотношения, могут быть либо двухсимвольные сочетания $X0$, $X1$,. . . , $X9$, обозначающие переменные, либо не более чем двузначные десятичные числа (возможно с ведущим нулем). В качестве знаков операций в выражениях могут быть использованы следующие: “-” “+” “*”.

Ограничения

Длина каждой строки, содержащей исследуемое соотношение, не меньше 3 и не больше 1000. В последовательности, составленной только из знаков операций отдельно для левой и отдельно для правой частей, знак умножения не может встречаться дважды подряд. Строки не содержат пробелов. В выражении переменная может умножаться только на число. Аналогично, число может умножаться только на переменную.

Тесты гарантируют, что соотношения заданы корректно. Т.е. у каждого соотношения есть ровно один знак сравнения, а также есть и правая, и левая части, заданные в соответствии с условием.

Формат входного файла

Входной файл содержит две строки, удовлетворяющие приведенным ограничениям. В каждой строке по одному соотношению, удовлетворяющему приведенным условиям.

Формат выходного файла

Выходной файл содержит единственную строку. В этой строке с первой позиции должен быть текст YES, если строки эквивалентны, а в противном случае должен быть (также с первой позиции) текст NO.

Примеры

h.in	h.out
2<-1 X0+32=0*X1-X0-25+X0*2	YES
X0-07*X3>=17+3*X4 X4+15+X0+X3*7<=2*X0-2*X4-3	NO

Разбор задачи H. Равносильность

Вначале замечаем, что правая и левая части рассматриваемых соотношений могут быть только линейными выражениями, содержащими не более чем 10 переменных. Учитывая еще и свободный член, каждую из частей соотношения можно взаимно однозначно отобразить на вектор, состоящий из 11 целых чисел. Очевидно, удобно выделить подстроку, содержащую выражение, составляющее одну из частей соотношения и иметь подпрограмму, которая переводит строку в соответствующий вектор. Дальше, вычитая почленно из одного вектора другой, получаем эквивалент заданного выражения с нулем в правой части. Добьёмся положительного знака у первого ненулевого элемента вектора (при необходимости изменим знак сравнения на противоположный). Получив подобное представление для обоих сравниваемых соотношений, останется рассмотреть 3 случая:

1. Оба соотношения не содержат переменных. В этом случае просто необходимо сравнить их единственные значения.
2. Хотя бы одно из соотношений содержит переменную, но знаки сравнения у них разные. В этом случае, ответ отрицательный.
3. Сравниваем почленно векторы представлений. Если встретился нуль, которому соответствует не нуль, либо соответствующие элементы имеют разные знаки, то ответ отрицательный. Осталось предусмотреть возможность разных векторов, соответствующие элементы которых пропорциональны.

Задача I. Троичная логика

Имя входного файла:	`i.in`
Имя выходного файла:	`i.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Говорят, что на заре зарождения программирования супруги программистов, не являющиеся программистами (представьте себе — такое тоже иногда имеет место), указывали своим супругам на оторванность от жизни строго двоичной логики. Представьте себе, что критике подвергался даже всем известный оператор ветвления в том или ином виде реализованный практически во всех программистских инструментальных средствах. Скажу по секрету, что известен проект расширенного оператора ветвления ЕСЛИ ТО ИНАЧЕ НО !? — с непременным использованием двойного символа “!?” в качестве его признака завершения. Говорят, что именно это

обстоятельство стимулировало развитие различных видов недвоичной логики, а также появление различных вариантов оператора выбора (переключения)... Пока еще нет официального статистического отчета о том, как повлияли сии новации на внутрисемейный климат семей, в которых только один из супругов является программистом, но вывод о чрезвычайной важности роли супругов программистов в процессе развития программирования можно сделать однозначно.

Пусть дано выражение, составленное из чисел 0,1,2, переменных, которые могут получить только одно из этих значений (0,1,2) а также операций сравнения, логического сложения, умножения, отрицания и "строгого" логического сложения (аналога исключающего или), определенных в соответствии со следующими таблицами:

x	y	$x < y$	$x = y$	$x <= y$	$!x$	$x\&y$	$x\|y$	$x \hat{} y$
0	0	0	2	2	2	0	0	0
0	1	2	0	2	2	0	1	1
0	2	2	0	2	2	0	2	2
1	0	0	0	0	1	0	1	1
1	1	0	2	2	1	1	1	2
1	2	2	0	2	1	1	2	0
2	0	0	0	0	0	0	2	2
2	1	0	0	0	0	1	2	0
2	2	0	2	2	0	2	2	1

Будем считать также определенными операции ">" ">=" "<>" в соответствии со следующими определениями. Для любых двух x,y, значения которых 0, 1 либо 2:

- x>y равносильно !(x<y) & !(x=y)
- x>=y равносильно (x>y) | (x=y)
- x <> y равносильно !(x=y)

Приоритеты операций приведены ниже в порядке убывания:

!

< > <= >=

= <>

&

^

|

В выражении могут быть использованы круглые скобки. Для заданного выражения и (при необходимости) значений входящих в него переменных, определить значение этого выражения.

Ограничения

Выражение не содержит пробелов и может содержать не более трех односимвольных переменных. В качестве переменных могут быть использованы только ‘x’, ‘y’, ‘z’. Длина каждого выражения не превосходит 100.

Формат входного файла

Первая строка содержит одно число, количество тестовых примеров, которое не превосходит 1 000 и не меньше 1. Дальше идет соответствующее количество тестовых примеров. Каждый тестовый пример начинается со строки, содержащей исходное выражение. За строкой, содержащей выражение, следует последовательность из не менее одной строки, каждая из которых содержит по 3 числа, записанные через пробел — соответственно, значения переменных x,y,z. Признаком конца этой последовательности строк будет строка с единственным числом, равным 3.

Формат выходного файла

Выходной файл содержит значения выражений, приведенных в отдельных строках в последовательности, соответствующей, приведенной во входном файле.

Примеры

i.in	i.out
2 2&1|1<=1 (2|1) 1 1 1 3 x&z|x<=z (x|1) 2 1 1 0 0 1 3	1 2 0

Разбор задачи I. Троичная логика

Структура входного файла подсказывает, что вначале необходимо определить постфиксное выражение, соответствующее вычисляемому выражению. При этом необходимо предусмотреть два вида параметров - константы и переменные. Константы будут представлены их значением, а переменные их номером. При вычислении переменная будет замещаться значением, соответствующим номеру. После этого останется применить

стандартный алгоритм вычисления к полученному постфиксному выражению необходимое количество раз. При внимательном рассмотрении замечаем следующие способы реализации операций, определенных в условии задачи:

$x\&y \equiv \min(x, y)$
$x|y \equiv \max(x, y)$
$x \hat{} y \equiv (x + y) \bmod 3$
$!x \equiv 2 - x$

Реализация операций сравнения очевидна.

Задача J. Контакты

Имя входного файла:	`j.in`
Имя выходного файла:	`j.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Два Мира вышли на контакт и обнаружили, что они формулируют свои мысли в оцифрованном виде вполне понятным друг для друга способом. Одно мешает полноценному общению — один из этих Миров пользуется позиционной системой с основанием p, а другой — системой с основанием q и эти основания различны. Наша задача — помочь общению.

Формат входного файла

Во входном файле задаются два числа p и q (в десятичном виде), а затем одна строка — оцифрованное сообщение (число) в системе с основанием p. Длина строки не более 1000 символов. $2 \leq p, q \leq 16, p <> q$. При необходимости, в качестве цифр, превосходящих 9 используются большие латинские буквы.

Формат выходного файла

В выходном файле строка — значение исходного числа в q-ичной системе без ведущих нулей. При этом, в качестве цифр, превосходящих 9 использовать необходимое количество больших латинских букв.

Примеры

j.in	j.out
7 10 523224	89982
10 7 89982	523224
16 12 AC	124

Разбор задачи J. Контакты

Следует аккуратно реализовать один из алгоритмов, приведенных на лекции.

Задача K. Мощность

Имя входного файла:	`k.in`
Имя выходного файла:	`k.out`
Ограничение по времени:	1 с
Ограничение по памяти:	256 Мб

Пусть задана последовательность байтов, каждый из которых определяет конкретное подмножество некоторой группы из 8 объектов, причем разные байты соответствуют разным непересекающимся группам объектов. Считается, что группы пронумерованы в определенном порядке — именно в том, в котором задается последовательность соответствующих байтов. Объекты внутри групп также перенумерованы начиная с нуля. Поэтому можно установить взаимно однозначное соответствие между объектами внутри группы и двоичными позициями соответствующего байта. Наличие объекта в подмножестве означает выставление единицы в соответствующей позиции соответствующего байта, а его отсутствие соответствует нулю в этой позиции.

Интерес представляют объединения подмножеств, соответствующих подряд расположенным группам.

Требуется определить ненулевую мощность, которая чаще всего встречается среди мощностей объединений подряд идущих подмножеств, обладающих свойством быть равными учетверенной длине соответствующего диапазона в байтах. Среди претендентов, имеющих равные показатели, выбрать самое большое число.

Вспомним, что мощностью конечного множества считается количество его элементов.

Формат входного файла

Во входном файле задается строка, представляющая собой исходную последовательность байтов. Одному байту соответствуют 2 подряд идущих символа, задающих двузначное шестнадцатеричное число — значение соответствующего байта. Строка имеет четную длину, не превосходящую 4 000.

Формат выходного файла

В выходном файле выводится единственное целое число — искомая мощность.

Примеры

k.in	k.out
457A	8

Разбор задачи K. Мощность

Сразу замечаем, что для нас интерес представляет последовательность мощностей подмножеств, соответствующих входной последовательности байтов. Разумеется, мощность подмножества, соответствующего некоторому байту равна количеству единиц в двоичном представлении этого байта. Соответственно, мощность подмножеств групп некоторого диапазона равна сумме единиц соответствующих байтов. Не следует поддаваться провокации подсчитать для каждого байта количество единиц. Нужно заранее подготовить массив из 16 элементов, где элемент с индексом i $(0 \leq i \leq 15)$ будет равен количеству единиц в двоичном представлении числа i. Тогда по ходу считывания заданной последовательности мощность подмножества, имеющего порядковый номер i будет равна $B[S[2*i]] + B[S[2*i+1]]$, где $S[i]$ — это числовое значение i-го символа входного файла, а $B[i]$ — это i-ый элемент вышеописанного массива.

Дальше не следует поддаваться провокации организовать структуру сумматора. Значительно проще и оправданнее просматривать диапазоны в такой последовательности, чтобы легче было для подсчета новой мощности максимально продуктивно использовать уже подсчитанные мощности. Здесь может быть несколько простых, но эффективных вариантов.

Список литературы

1. Алгоритм Прима, построение минимального остовного дерева взвешенного связного неориентированного графа [Электронный ресурс] / Wikimedia Foundation. - Режим доступа : www/ URL: http://ru.wikipedia.org/wiki/Алгоритм_Прима - 10.12.2009 г. - Загл. с экрана.

2. Диаграмма, мозаика Вороного, разбиение Дирихле [Электронный ресурс] / Wikimedia Foundation. - Режим доступа : www/ URL: http://ru.wikipedia.org/wiki/Диаграмма_Вороного - 19.11.2009 г. - Загл. с экрана.

3. Красиво и доступно оформленная презентация на английском, разъясняющая построение диаграммы [Электронный ресурс] / Algolist Study. - Режим доступа : www/ URL: http://algolist.manual.ru/maths/geom/voronoi/index.php - 29.08.2009 г. - Загл. с экрана.

4. Алгоритм Крускала, построения минимального остовного дерева взвешенного связного неориентированного графа [Электронный ресурс] / Wikimedia Foundation. - Режим доступа : www/ URL: http://ru.wikipedia.org/wiki/Алгоритм_Крускала - 19.06.2008 г. - Загл. с экрана.

5. Гасфилд Дэн. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология [Текст] / Дэн Гасфилд. - СПб.: Невский Диалект; БХВ-Петербург, 2003.

6. Тернарный поиск, поиск максимумов и минимумов функции [Электронный ресурс] / Wikimedia Foundation. - Режим доступа : www/ URL: http://en.wikipedia.org/wiki/Ternary_search - 12.06.2009 г. - Загл. с экрана.

7. Пападимитриу Х. Комбинаторная оптимизация. Алгоритмы и сложность [Текст] / Пападимитриу Х., Стайглиц К. - Москва: Наука, 1984.

8. Сборник Зимней школы 2008 [Текст] / Вечур О. В. - ХНУРЕ, 2008, день 6, задача F. Minima.

9. Хорватская школьная олимпиада по информатике [Электронный ресурс] / Croatia Znanost. - Режим доступа : www/ URL: http://hsin.hr/coci - 19.06.2009 г. - Загл. с экрана.

10. Maa Moritz. Suffix Trees and their Applications [Текст] / Moritz Maa. - Ferien-Akademie, 1999.

11. Manber Udi. Suffix arrays: A new method for on-line string searches [Текст] / Udi Manber, Gene Myers. - SIAM Journal on Computing, 1993.

12. Karkkainen Juha. Simple linear work suffix array construction [Текст] / Juha Karkkainen, Peter Sanders. - ACM, 2006.