

Всероссийская олимпиада школьников по информатике
Муниципальный этап 9-11 класс

Задача 1. Справедливый дележ**100 баллов****Решение**

Нам нужно найти количество натуральных делителей числа N , включая 1 и само число N . Почему?

Если мы раздаём N пряников K людям поровну, то каждый получит N/K пряников. Чтобы это было целое число, K должно быть делителем N . $K \geq 1$, поэтому учитываем все натуральные делители.

Простой случай ($N \leq 10^6$)

Для небольших N можно использовать прямой перебор:

- Перебираем все числа от 1 до N
- Проверяем, делится ли N на это число
- Считаем количество делителей

Сложность: $O(N)$ — приемлемо для $N \leq 10^6$

Программа, набирающая 50 баллов, может иметь подобный вид на ЯП Python:

```
n = int(input()) # считывание N
count = 0 # счётчик ответа
for i in range(1, n + 1): # перебираем от значения 1 до n
    if n % i == 0: # и если нашли делитель
        count += 1 # увеличиваем ответ
print(count) # в конце выводим
```

Сложный случай ($N \leq 10^{12}$)

Для больших N прямой перебор до N невозможен (10^{12} операций — это слишком много).

Эффективный алгоритм:

- **Перебираем числа только до \sqrt{N}**
- Если i делит N , то мы находим сразу два делителя: i и N/i
- Особый случай: когда N — полный квадрат ($i = N/i$)

Сложность: $O(\sqrt{N})$ — приемлемо даже для $N = 10^{12}$ (всего 10^6 операций)

Из гипотетических трудностей - на больших значениях корень может вычисляться не очень-то и точно. Впрочем, ничего не мешает оставаться в целых числах.

Программа решения может принимать, например, следующий вид на ЯП Python:

```
# Чтение исходных данных
n = int(input())
# Инициализация переменных
count = 0
```

```

i = 1
# Проход до корня из n с подсчетом делителей
while i * i <= n:
    if n % i == 0:
        count += 1 if i * i == n else 2
    i += 1
# Вывод результата
print(count)

```

Программа решения может принимать, например, следующий вид на ЯП C++:

```

#include <iostream>
using namespace std;

int main() {
// Чтение исходных данных
long long n;
cin >> n;
// Инициализация переменных
long long count = 0;
// Проход до корня из n с подсчетом делителей
for (long long i = 1; i * i <= n; i++) {
    if (n % i == 0) {
        if (i * i == n) { // Если n - полный квадрат
            count += 1;
        } else {
            count += 2;
        }
    }
}
// Вывод результата
cout << count << endl;
return 0;
}

```

Задача 2. Треугольник из палочек

100 баллов

Решение

Для того чтобы три палочки могли образовать невырожденный треугольник, должно выполняться неравенство треугольника: сумма любых двух сторон должна быть строго больше третьей стороны.

Для сторон a, b, c (где $a \leq b \leq c$) это означает: $a + b > c$, $a + c > b$ (автоматически выполняется при $a \leq b \leq c$) и $b + c > a$ (автоматически выполняется при $a \leq b \leq c$)

Таким образом, основное условие: $a + b > c$

Простой случай ($n \leq 600$)

Для небольших n можно использовать полный перебор всех возможных троек:

- Перебираем все тройки (i, j, k) такие что $i < j < k$
- Проверяем условие треугольника для отсортированной тройки
- Считаем количество подходящих троек

Сложность: $O(n^3)$ — приемлемо для $n \leq 600$ Приведём решение по правилу треугольников.

Программа, набирающая 50 баллов, может иметь подобный вид на ЯП Python:

```
n = int(input())
a = list(map(int, input().split()))
count = 0
# Перебираем все возможные тройки
for i in range(n):
    for j in range(i + 1, n):
        for k in range(j + 1, n):
            # Сортируем очередную тройку так, как нам удобно
            x, y, z = sorted([a[i], a[j], a[k]])
            # Проверяем соблюдение неравенства треугольника
            if x + y > z:
                count += 1
print(count)
```

Сложный случай ($n \leq 10^4$)

Используем метод двух указателей:

- Сортируем массив палочек по возрастанию
- Фиксируем первую сторону i
- Используем два указателя j и k :
- j идет от $i+1$ до $n-1$
- k идет от $j+1$, но не сбрасывается для каждого j

Сложность: $O(n^2)$ — оптимально для $n \leq 10^4$

Программа решения может принимать, например, следующий вид на ЯП Python:

```
n = int(input())
a = list(map(int, input().split()))
a.sort() # Сортируем массив
count = 0
for i in range(n):
    k = i + 2 # k всегда на шаг впереди j
    for j in range(i + 1, n):
        # Двигаем k пока выполняется a[i] + a[j] > a[k]
        while k < n and a[i] + a[j] > a[k]:
            k += 1
        # Все тройки (i, j, m) где m от j+1 до k-1 - валидны
        count += k - j - 1
print(count)
```

Программа решения может принимать, например, следующий вид на ЯП C++:

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    sort(a.begin(), a.end()); // Сортируем

    long long count = 0;
    for (int i = 0; i < n; i++) {
        int k = i + 2; // k всегда на 1 шаг впереди j
        for (int j = i + 1; j < n; j++) {
            // Двигаем k пока выполняется условие треугольника
            while (k < n && a[i] + a[j] > a[k]) {
                k++;
            }
            // Все тройки (i,j,m), где m от j+1 до k-1 - валидны
            count += k - j - 1;
        }
    }

    cout << count << endl;
    return 0;
}

```

Задача 3. Буквояд

100 баллов

Решение

Нам нужно расположить буквы в таком порядке, чтобы сумма абсолютных разностей между соседними буквами была максимальной.

Ключевая идея: Для максимизации суммы разностей нужно чередовать самые большие и самые маленькие значения. Оптимальная стратегия — расположить буквы в порядке "зигзага": минимальная, максимальная, следующая минимальная, следующая максимальная и т.д.

Простой случай ($|s| < 11$)

Для небольших строк можно использовать полный перебор всех перестановок:

- Генерируем все перестановки букв
- Для каждой перестановки вычисляем сумму разностей
- Находим максимальную сумму

Сложность: $O(n! \times n)$ — приемлемо для $n \leq 11$

Программа, набирающая 50 баллов, может иметь подобный вид на ЯП Python:

```
from itertools import permutations

alphabet = input().strip()
s = input().strip()

# Создаем словарь для быстрого получения позиции буквы
pos = {char: idx for idx, char in enumerate(alphabet)}

# Преобразуем строку в список позиций
positions = [pos[char] for char in s]

max_sum = 0
# Перебираем все перестановки
for perm in permutations(positions):
    current_sum = 0
    for i in range(len(perm) - 1):
        current_sum += abs(perm[i] - perm[i + 1])
    max_sum = max(max_sum, current_sum)

print(max_sum)
```

Сложный случай ($|s| \leq 300\,000$)

Для больших n полный перебор невозможен. Используем жадный алгоритм:

- Преобразуем буквы в их позиции в алфавите
- Сортируем позиции
- Чередуем минимальные и максимальные значения: берем самый маленький, потом самый большой, потом следующий маленький, потом следующий большой и т.д.
- Вычисляем сумму разностей для такой последовательности
- Берём лучший случай (начать с максимума или начать с минимума)

Сложность: $O(n \log n)$ — оптимально для $n \leq 300\,000$

Программа решения может принимать, например, следующий вид на ЯП Python:

```
alphabet = input().strip()
s = input().strip()

pos = {char: idx for idx, char in enumerate(alphabet)}
positions = sorted([pos[char] for char in s])
n = len(positions)

# Есть два основных варианта чередования:
# 1. мин-макс-мин-макс...
# 2. макс-мин-макс-мин...

# Вариант 1: начинаем с минимума
seq1 = []
l, r = 0, n - 1
while l <= r:
    seq1.append(positions[l])
```

```

    l += 1
    if l <= r:
        seq1.append(positions[r])
        r -= 1

# Вариант 2: начинаем с максимума
seq2 = []
l, r = 0, n - 1
while l <= r:
    seq2.append(positions[r])
    r -= 1
    if l <= r:
        seq2.append(positions[l])
        l += 1

# Вычисляем суммы для обоих вариантов
def calc_sum(seq):
    total = 0
    for i in range(len(seq) - 1):
        total += abs(seq[i] - seq[i + 1])
    return total

result = max(calc_sum(seq1), calc_sum(seq2))
print(result)

```

Программа решения может принимать, например, следующий вид на ЯП C++:

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <map>
#include <cmath>
using namespace std;

string alphabet, s;
vector<int> seq1, seq2;
map<char, int> pos; // Создаем словарь для позиций букв
vector<int> positions;

int main() {

    cin >> alphabet >> s;

    for (int i = 0; i < 26; i++) {
        pos[alphabet[i]] = i;
    }

    // Преобразуем строку в позиции и сортируем

    for (char c : s) {
        positions.push_back(pos[c]);
    }

```

```

sort(positions.begin(), positions.end());

int n = positions.size();

// Строим две возможные оптимальные последовательности

int l = 0, r = n - 1;

// Вариант 1: начинаем с минимума
while (l <= r) {
    seq1.push_back(positions[l++]);
    if (l <= r) seq1.push_back(positions[r--]);
}

// Вариант 2: начинаем с максимума
l = 0; r = n - 1;
while (l <= r) {
    seq2.push_back(positions[r--]);
    if (l <= r)
        seq2.push_back(positions[l++]);
}

// Вычисляем суммы для обоих вариантов
auto calc_sum = [](const vector<int>& seq) {
    long long sum = 0;
    for (int i = 0; i < seq.size() - 1; i++) {
        sum += abs(seq[i] - seq[i + 1]);
    }
    return sum;
};

long long result = max(calc_sum(seq1), calc_sum(seq2));
cout << result << endl;

return 0;
}

```

Задача 4. Кто не билечен?

100 баллов

Решение

У нас есть три типа "пакетов" поездок:

- Одиночная поездка: стоимость A за 1 поездку
 - Пара поездок: стоимость B за 2 поездки ($B \leq 2A$)
 - Тройка поездок: стоимость C за 3 поездки ($C \leq 3A$ и $C \leq A + B$)
- Нужно найти минимальную стоимость для N поездок.

Простой случай ($N \leq 10^7$)

Для небольших N можно (ради эксперимента и понимания) использовать динамическое программирование:

$dp[i]$ - минимальная стоимость i поездок

Переходы: из состояния i можно добавить 1, 2 или 3 поездки

Сложность: $O(N)$ — приемлемо для $N \leq 10^7$

Программа, набирающая 50 баллов, может иметь подобный вид на ЯП Python

```
A = int(input())
B = int(input())
C = int(input())
N = int(input())

# Инициализация ДП
dp = [float('inf')] * (N + 1)
dp[0] = 0

for i in range(N + 1):
    if i + 1 <= N:
        dp[i + 1] = min(dp[i + 1], dp[i] + A)
    if i + 2 <= N:
        dp[i + 2] = min(dp[i + 2], dp[i] + B)
    if i + 3 <= N:
        dp[i + 3] = min(dp[i + 3], dp[i] + C)

print(dp[N])
```

Сложный случай ($N \leq 10^9$)

У нас есть три варианта покупки поездок. Так как N может быть до 10^9 , нужно решение за $O(1)$.

Ключевая идея: Из условий следует, что самые выгодные предложения идут по возрастанию: тройка выгоднее пары, пара выгоднее одиночной. И есть лишь один маленький, но вносящий сумятицу и неопределённость случай когда: $2 * B > A + C$. Однако он влияет только на последнюю тройку купленных билетов

Эффективный алгоритм:

- Покупаем максимум троек (самый выгодный пакет)
- Для оставшихся 0, 1 или 2 поездки берем оптимальную комбинацию

Сложность: $O(1)$

Программа решения может принимать, например, следующий вид на ЯП Python:

```
A = int(input())
B = int(input())
C = int(input())
N = int(input())

# Считаем максимальное количество троек и остаток
triples = N // 3
remainder = N % 3

# Для имеющегося остатка выбираем лучший вариант
if remainder == 0:
    print(triples * C)
elif remainder == 1:
    print(min(triples * C + A, (triples - 1) * C + 2 * B))
```



```
else: # remainder == 2
    print(triples * C + B)
```

Программа решения может принимать, например, следующий вид на ЯП C++:

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    long long A, B, C, N, result;
    cin >> A >> B >> C >> N;

    long long triples = N / 3;
    long long remainder = N % 3;

    if (remainder == 0) {
        result = triples * C;
    } else if (remainder == 1) {
        result = min(triples * C + A, (triples - 1) * C + 2
* B);
    } else { // remainder == 2
        result = triples * C + B;
    }

    cout << result << endl;
    return 0;
}
```

Задача 5. Загадочное житие кузнечика

100 баллов

Решение

Кузнечик начинает в точке 0 и хочет достичь точки N. Из каждой точки i доступны два пути: телепортация в a_i за 0 секунд, прыжок в b_i за 1 секунду

Нужно найти минимальное время достижения точки N.

Простой случай ($N \leq 1000$)

Для небольших N можно использовать алгоритм Дейкстры или BFS. Строим граф, где вершины - точки от 0 до N. Из каждой вершины i есть ребра в a_i с весом 0 и в b_i с весом 1. Находим кратчайший путь из 0 в N

Сложность: $O(N \log N)$ с Дейкстрой или $O(N)$ с 0-1 BFS

Если телепортация заблокирована, то из каждой точки i можно прыгнуть только в b_i за 1 секунду. Это становится задачей поиска кратчайшего пути в ориентированном графе.

Программа, набирающая 50 баллов, может иметь подобный вид на ЯП Python:

```
from collections import deque

N = int(input())
```

```

points = []
for _ in range(N):
    a, b = map(int, input().split())
    points.append((a, b))

# BFS для поиска кратчайшего пути
dist = [-1] * (N + 1)
dist[0] = 0
queue = deque([0])

while queue:
    current = queue.popleft()

    if current == N:
        break

    if current < N:
        a, b = points[current]

        # Прыжок (единственный доступный вариант)
        if b <= N and dist[b] == -1:
            dist[b] = dist[current] + 1
            queue.append(b)

print(dist[N])

```

Когда телепортация возможна только вперед, можно использовать динамическое программирование с обходом вперед.

Программа, набирающая 75 баллов, может иметь подобный вид на ЯП Python:

```

N = int(input())
points = []
for _ in range(N):
    a, b = map(int, input().split())
    points.append((a, b))

dist = [float('inf')] * (N + 1)
dist[0] = 0

for i in range(N):
    if dist[i] == float('inf'):
        continue

    a, b = points[i]

    # Телепортация (только вперед)
    if a <= N:
        dist[a] = min(dist[a], dist[i])

    # Прыжок
    if b <= N:
        dist[b] = min(dist[b], dist[i] + 1)

```

```
print(dist[N])
```

Для полного решения нужно учесть, что телепортация может быть в любую точку (вперед или назад). Используем BFS с 0-1 весами:

Сложность: $O(N)$. Программа решения может принимать, например, следующий вид на ЯП Python:

```
import heapq

N = int(input())
points = []
for _ in range(N):
    a, b = map(int, input().split())
    points.append((a, b))

# Инициализация расстояний
dist = [float('inf')] * (N + 1)
dist[0] = 0

# Куча для Дейкстры
heap = [(0, 0)] # (время, точка)

while heap:
    time, current = heapq.heappop(heap)

    if time > dist[current]:
        continue

    if current == N:
        break

    if current < N: # Имеет смысл прыгать только из точек < N
        a, b = points[current]

        # Телепортация
        if a <= N and time < dist[a]:
            dist[a] = time
            heapq.heappush(heap, (time, a))

        # Прыжок
        if b <= N and time + 1 < dist[b]:
            dist[b] = time + 1
            heapq.heappush(heap, (time + 1, b))

print(dist[N])
```

Программа решения может принимать, например, следующий вид на ЯП C++:

```
#include <iostream>
#include <vector>
#include <deque>
using namespace std;
```

```

int main() {
    int N;
    cin >> N;

    vector<pair<int, int>> points(N);
    for (int i = 0; i < N; i++) {
        cin >> points[i].first >> points[i].second;
    }

    vector<int> dist(N + 1, -1);
    dist[0] = 0;
    deque<int> deq;
    deq.push_back(0);

    while (!deq.empty()) {
        int current = deq.front();
        deq.pop_front();

        if (current == N) break;
        if (current >= N) continue;

        int a = points[current].first;
        int b = points[current].second;

        // Телепортация (вес 0)
        if (a <= N && (dist[a] == -1 || dist[a] > dist[current]))
        {
            dist[a] = dist[current];
            deq.push_front(a);
        }

        // Прыжок (вес 1)
        if (b <= N && (dist[b] == -1)) {
            dist[b] = dist[current] + 1;
            deq.push_back(b);
        }
    }

    cout << dist[N] << endl;
    return 0;
}

```

Надеемся, многогранность этой задачи не станет её минусом.