Разбор задач отборочного тура Олимпиады по программированию «ИнфоТех Квест»

Автор разборов Плющенко Даниил Владимирович, преподаватель дополнительного образования, предмет Олимпиадное программирование.

Задача A. Необычная последовательность

Будем перебирать натуральные числа в порядке возрастания. Для каждого числа узнаем количество его делителей, прибавим это число к общей сумме. Как только сумма стала не меньше, чем k, выводим текущее число в качестве ответа. Чтобы узнать количество делителей, переберём возможные делители. Такое решение набирает 50 баллов, перебор делителей работает за O(n).

```
int divisors(int n) {
    int result = 0;
    for (int d = 1; i <= n; i++) {
        if (n % d == 0) {
            result++;
        }
    }
    return result;
}
```

Заметим, что если число a является делителем числа n, то число $b = n/a$ тоже является делителем, при этом одно из них не превосходит $\sqrt{n}$ иначе, если оба делителя больше $\sqrt{n}$, то $n = a \cdot b > \sqrt{a} \cdot \sqrt{b} > n$, то есть получаем противоречие. Таким образом, можно перебирать меньший из делителей до $\sqrt{n}$ и сразу получать парный ему делитель. Такое решение набирает 100 баллов, перебор делителей работает за O($\sqrt{n}$) .

```
int divisors(int n) {
    int result = 0;
    for (int d = 1; d * d <= n; i++) {
        if (n % d == 0) {
            result += 2;
        }
        if (d * d == n) {
            result--;
        }
    }
    return result;
}
```

Задача B. Хорошая строка

Переберём позиции символов, которые надо поменять местами, затем проверим, что строка является палиндромом. Такое решение работает за $O(n^3)$ и набирает 40 баллов.

```
bool is_palindrome(string s) {
    string t = s;
    reverse(t.begin(), t.end());
    return s == t;
}

for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        string t = s;
        swap(t[i], t[j]);
        if (is_palindrome(t)) {
            cout << "Yes";
            return 0;
        }
    }
}
cout << "No";
```

Можно ускорить это решение, если проверять, является ли строка палиндромом, используя хеши или использовать некоторые отсечения, чтобы получить 60 баллов.

Заметим, что если количество несовпадающих символов на симметричных позициях k равно 0, то ответ положительный. Если k равно 4, то для положительного ответа нужно, чтобы несовпадающие символы образовали две пары равных. Если k равно 2, то для положительного ответа нужно, чтобы строка была нечётной длины и один из несовпадающих символов был бы равен центральному. Такое решение работает за $O(n)$ и набирает 100 баллов.

```
s = input()
bad = []
for i in range(len(s) //2):
        if s[i] != s[len(s) - i - 1]:
                bad.append(s[i])
                bad.append(s[len(s) - i - 1])
bad.sort()
if len(bad) == 0:
        print("Yes")
elif len(bad) == 4 and bad[0] == bad[1] and bad[2] == bad[3]:
        print("Yes")
elif len(bad) == 2 and len(s) % 2 == 1 \
        and (s[len(s) // 2] == bad[0] or s[len(s) // 2] == bad[1]):
        print("Yes")
else:
        print("No")
```

Задача C. Binarycore

В этой задаче достаточно сделать то, что написано в условии. Будем поддерживать позицию элемента, с которого начинается подотрезок из одинаковых элементов, если текущий элемент не равен ему, то обновляем позицию, иначе увеличиваем длину текущего подотрезка.

```
def get_ans(a):
    last = a[0]
    length =1
    ans = 0
    for i in range(1, len(a)):
        if a[i] ==last:
            length += 1
        else:
            ans = max(ans, length)
            length = 1
            last = a[i]
    ans = max(ans, length)
    return ans

if__name__== "_main_":
    n =int(input())
    s = input()
    a = []
    last = s[0]
    length =1
    for i in range(1, n):
        if s[i] == last:
            length += 1
        else:
            a.append(length)
            length = 1
            last = s[i]
    a.append(length)
    print(get_ans(a))
```

Задача D. Квадратное фризби

Пусть имеющиеся точки считаны в массив *p[]*.

Переберём четырьмя вложенными циклами номера точек из массива p[], пусть эти номера *i, j, k, l*, при этом все они различны. Проверим, образуют ли эти четыре точки квадрат с противоположными сторонами *p[i] p[j]* и *p[k] p[l]*.

Чтобы проверить, образуют ли четыре точки a, b, c, d квадрат с противоположными сторонами *ab* и *cd*, проверим, что длины отрезков *ab, ac, ad, bc, bd* и *cd* совпадают. Расстояние между двумя точками *a* и *b* вычисляется по формуле $\sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$.

Чтобы избежать погрешности при использовании вещественных чисел, можно сравнивать длины на равенство с точностью до некоторого Ɛ, например, Ɛ = 10^{-6}. Также можно сравнивать на равенство квадраты расстояний, а не сами расстояния. Поскольку координаты точек целочисленные, квадраты расстояний тоже будут целочисленными, это поможет избежать использования вещественных чисел.

Если данные четыре точки образуют квадрат, то проверим, правда ли, что текущий ответ меньше значения площади этого квадрата. Если это так, то обновим текущий ответ. В конце выведем текущий ответ.

Такое решение работает за O(n^4) и набирает 40 баллов.

```
struct point
    { int x;
    int y;

    bool operator <(const point& other) const {
        return x < other.x || x == other.x && y < other.y;
    }
};
long double get_dist(point &a, point& b)
    { long double dx = a.x - b.x;
    long double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
bool is_square(vector<point> &p) {
    long double a = get_dist(p[0],
    p[1]);    long double    b    =
    get_dist(p[1], p[2]); long double c
    = get_dist(p[2], p[3]); long double
    d = get_dist(p[3], p[0]); return
    abs(a - b) < 1e-6
        && abs(a - c) < 1e-6
        && abs(a - d) < 1e-6
        && abs(b - c) < 1e-6
        && abs(b - d) < 1e-6
        && abs(c - d) < 1e-
        6;
}
long double get_area(vector<point>& p) {
    return get_dist(p[0], p[1]) * get_dist(p[2], p[3]);
}
int main() {
```

```
        int n;
        cin >>
        n;
        vector<point> p(n);
        for (int i = 0; i < n; i++) {
               cin >> p[i].x >> p[i].y;
        }
        long double ans = 0;
        for (int i = 0; i < n; i++)
          for (int j = i + 1; j < n; j++)
            for (int k = j + 1; k < n;
            k++)
              for (int l = k + 1; l < n; l++) {
                vector<point> current = {p[i], p[j], p[k],
                p[l]}; sort(current.begin(), current.end());
                do {
                  if (is_square(current)) {
                    ans = max(ans,
                    get_area(current)); break;
                  }
                } while (next_permutation(current.begin(), current.end()));
              }
    cout << fixed << setprecision(8) <<
    ans; return 0;
}
```

Переберём двумя вложенными циклами номера точек из массива *p[]*, пусть эти номера *i≠ j*. Проверим, существует ли квадрат, в котором точки *p[i]* и p*[j]* образуют сторону, для этого должны существовать ещё две точки, которые будут противоположной стороной.

Один из способов это проверить – повернуть вектор $\overrightarrow{p[i]\,p[j]}$ на 90 градусов и отложить этот вектор как от точки *p[i],* так и от *p[j]*. Тогда концы отложенных векторов вместе с *p[i]* и *p[j]* будут образовывать квадрат.

Значит, надо проверить, что концы отложенных векторов присутствуют среди имеющихся точек, для этого сложим все точки в множество (хранить точки в хеш-таблице или двоичном дереве). Такое решение работает за $O(n^2)$ или $O(n^2 * \log n)$ в зависимости от используемой структуры данных и набирает 100 баллов.

Заметим, что решение за $O(n^3)$ или $O(n^3 * \log n)$, перебирающее три точки возможного квадрата, также получало 100 баллов.

```
struct point
    { long
    long x;
    long long
    y;
    bool operator <(const point& other) const {
        return x < other.x || x == other.x && y < other.y;
    }
    bool operator ==(const point& other) const
        { return x == other.x && y == other.y;
    }
};
```

```
set<point> finds;
int main() {
    int n;
    cin >>
    n;
    vector<point> p;
    for (int i = 0; i < n; ++i)
        { ll x, y;
        cin >> x >> y;
        p.push_back({x,
        y});
        finds.insert({x, y});
        }
    ll ans = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            point new_v = { p[i].x - p[j].x, p[i].y - p[j].y
            }; point rot_v = { -new_v.y, new_v.x };
            point a = { p[j].x + rot_v.x, p[j].y + rot_v.y };
            point b = { p[i].x + rot_v.x, p[i].y + rot_v.y
            }; long long dx = p[i].x - p[j].x;
            long long dy = p[i].y - p[j].y;
            if (finds.find(a) != finds.end() && finds.find(b) !=
                finds.end()) ans = max(ans, abs(dx * dx + dy * dy));
            a = { p[j].x - rot_v.x, p[j].y - rot_v.y };
            b = { p[i].x - rot_v.x, p[i].y - rot_v.y };
            if (finds.find(a) != finds.end() && finds.find(b) !=
                finds.end()) ans = max(ans, abs(dx * dx + dy * dy));
        }
    }
    cout <<
    ans;
    return 0;
}
```

Задача E. Перетягивание каната

Пусть c[i][j] == 1, если i и j знакомы между собой. Тогда тремя вложенными циклами переберём номера людей i, j и k, при этом все они различны, и проверим, что *c[i][j] = c[i][k] = c[j][k] = 1*. Если это правда, то прибавим единицу к счётчику.

Тогда ответ – это значение счётчика, разделенное на 6, потому что мы посчитали количество упорядоченных троек, а нас интересуют неупорядоченные, их в 3! раз меньше. Можно перебирать тройки так, чтобы i < j < k, тогда делить на 6 не придётся. Такое решение работает за $O(n^3)$ и набирает 40 баллов.

Тогда переберём все тройки, проверим, что в соотвествующих ячейках матрицы с стоят единицы.

```
int cnt = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    for (int k = 0; k < n; k++)
      if (c[i][j] && c[i][k] && c[j][k])
        cnt++;
cout << cnt / 6;
```

Переберём пару знакомых человек и третьего человека. Проверим, что третий человек знаком с обоими. Чтобы перебирать пары знакомых человек, сохраним все пары в списке, а чтобы проверять, подходит ли третий человек, будем поддерживать матрицу c[][]. Такое решение работает за O(nm) и набирает 100 баллов.

```
int edge = 0;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  a--;
  b--;
  if (c[a][b])
    continue;
  c[a][b] = c[b][a] = 1;
  from[edge] = a;
  to[edge] = b;
  edge++;
}

int cnt = 0;
for (int i = 0; i < edge; i++) {
  for (int j = 0; j < n; j++) {
    if (c[from[i][j] && c[to[i][j])) {
      cnt++;
    }
cout << cnt / 3;
```

Задача F. Стрельба по движущемуся противнику

В этой задаче достаточно было промоделировать процесс, для этого нужно понимать, в какие моменты времени враг будет проходить каждый из отрезков ломаной. Если суммарное время будет меньше T, то нужно пойти по следующему отрезку, иначе нужно понять, в какой точке отрезка остановится враг.

Будем поддерживать, сколько времени T осталось идти врагу. Изначально $T = t$. Идём по списку точек и вычисляем расстояние d между соседними точками, то есть длину звена ломаной. Также вычисляем, сколько времени c займёт переход от текущей точки до следующей, это значение равно d / v.

Если $T > c$, то обновим $T = T - c$ и будем моделировать ход врага дальше. Иначе посчитаем, где именно между двумя точками будет находиться враг.

$r = T / c$ – такую долю расстояния между текущей и следующей точкой пройдёт враг.

Тогда нужно от текущей точки отложить вектор, идущий в следующую точку, но его длину надо домножить на r. Именно в этой точке и остановится враг.

```
struct point {
    long double x, y;
};
long double get_dist(point& a, point& b) {
    long double dx = a.x - b.x;
    long double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
int main() {
    long double v;
    cin >> v;
    int n;
    cin >> n;
    vector<point> p(n);
    for (int i = 0; i < n; i++) {
        cin >> p[i].x >> p[i].y;
    }
    int t;
    cin >> t;
    long double time_left = t;
    for (int i = 1; i < n; i++) {
        long double dist = get_dist(p[i - 1], p[i]);
        long double ct = dist / v;
        if (time_left > ct) {
            time_left -= ct;
        } else {
            long double t = time_left / ct;
            point dp = { p[i].x - p[i - 1].x, p[i].y - p[i - 1].y };
            point ans = { p[i - 1].x + dp.x * t, p[i - 1].y + dp.y * t };
            cout << fixed << setprecision(3) << ans.x << " " << ans.y;
            return 0;
        }
    }
    cout << -1;
    return 0;
}
```