Зміст

Задача Kvadro

Інтерпертуємо умову задачі математично. Маємо формулу: $A + B \cdot N \leq C$. Звідси, очевидно $B \cdot N \leq C - A$, а оскільки B додатнє, маємо $N \leq \frac{C-A}{B}$.

Слід зазначити, що за $C \leq A$ розв'язок, очевидно, 0 - адже у нас не вистачить грошей на жодну лопасть. Інакше максимальне значення N можемо отримати як $\lfloor \frac{C-A}{B} \rfloor$. Зручніше за все використати цілочисленне ділення.

Розв'язок на C++:

```cpp
#include<iostream>
#include<algorithm>

using namespace std;

int main(){
    int a, b, c;
    cin >> a >> b >> c;
    cout << max((c - a)/b, 0);
    return 0;
}
```

Задача Diagonals

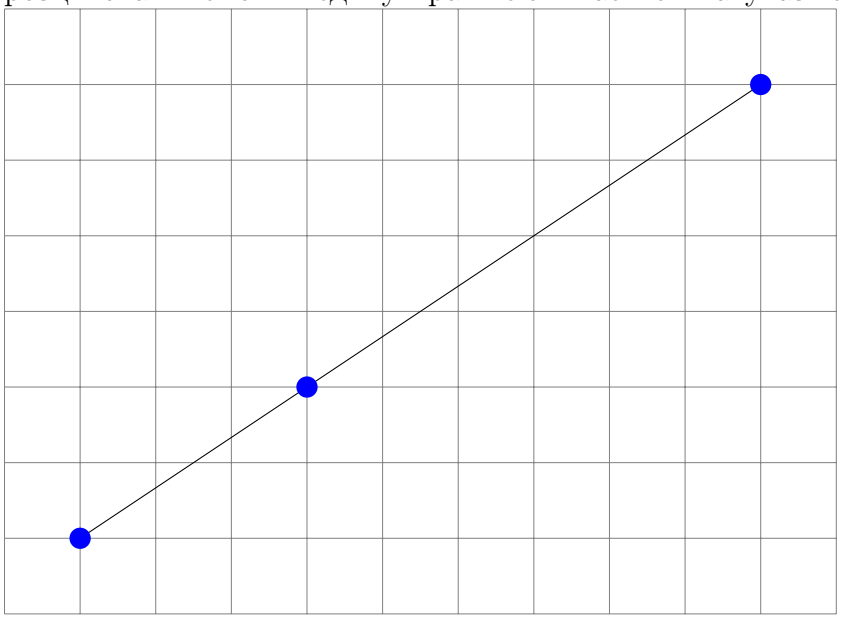
Зрозуміло, що у випадку трикутника у нас просто не існує діагоналей, а отже і шукати перетин ні в чого. Розглянемо N-кутник ($4 \leq N$). Виділивши 4 різні вершини цього N-кутника, ми отримаємо чотирикутник, дві діагоналі якого перетинаються. Крім того, до будь-яких двох діагоналей N-кутника, що перетинаються, ми можемо поставити у відповідність такий чотирикутник. Тож відповіддю до задачі буде C_N^4, тобто кількість способів вибрати 4 вершини з по-між N, якщо порядок вибору не має значення. $C_N^K = \frac{N!}{K!(N-K)!}$. Утім, якщо «в лоб» рахувати факторіали, вже на $N = 21$ ми зустрінемося з проблемами переповення навіть за умови використання 64-бітного типу данних. Тож необхідно зовсім трохи спростити наш вираз: $\frac{N!}{K!(N-K)!} = \frac{1 \cdot 2 \cdot \ldots \cdot N}{K! \cdot 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (N-K)} = \frac{(N-K+1) \cdot \ldots \cdot N}{K!}$, що для випадку $K = 4$ перетворюється у $\frac{N \cdot (N-1) \cdot (N-2) \cdot (N-3)}{1 \cdot 2 \cdot 3 \cdot 4} = \frac{N \cdot (N-1) \cdot (N-2) \cdot (N-3)}{24}$. Очевидно, що за $N \leq 100$ чисельник дробу не перевищить 10^8, що цілком вміщається в 32-бітний тип данних.

Розв'язок на C++:

```cpp
#include<iostream>

using namespace std;

int main(){
    int n;
    cin >> n;
    if(n < 4){
        cout << 0;
    } else {
        cout << n * (n - 1) * (n - 2) * (n - 3) / 24;
    }
    return 0;
}
```

Задача Guardian

Перш за все, уточнимо один момент: дві колінеарні траншеї, що мають спільну точку, розцінюватимемо як єдину траншею. Мається на увазі ситуація, подібна до наступної:



У цьому разі дві траншеї ((1, 1), (4, 3)) та ((4, 3), (10, 7)) слід розглядати як єдину траншею ((1, 1), (10, 7)).

Умова задачі не передбачала наявність в траншей більш як однієї точки перетину, до чого можна було дійти логічними міркуваннями (або отримати відповідь від журі).

Тепер перейдемо безпосередньо до задачі. Відмітимо той факт, що троє вартових не можуть сидіти в одній траншеї - у такому випадку двоє крайніх не бачать одне одного через наявність між ними третього. Однак кожен з вартових повинен бачити крізь траншею двох інших, звідки логічно випливає, що кожен вартовий розміщений у точці перетину якихось траншей. Якщо перший вартовий розміщений на перетині траншей A та B, другий на перетині B та C, третій на перетині A і C, то умова задачі виконується. Єдиним винятком є випадок, коли три траншеї перетинаються в одній точці. У цьому випадку вартових поставити все-таки не вийде.

З цих міркувань випливає алгоритм розв'язку: спершу об'єднуємо колінеарні траншеї, що мають спільну точку в одну. Далі перебираємо трійки різних траншей. Якщо з цієї трійки кожні дві перетинаються усі вони не перетинаються в одній точці, то збільшуємо відповідь на одиницю. Однак на словах простий розв'язок потребує досить великого коду, адже підзадачі досить громіздкі. Далі розв'язок буде наведено частинами, кожен шматок коду буде коментуватися (тобто що та чи інша частина програми покликана робити).

Деякі деталі коду можуть здатися незнайомими читачеві, що не заглиблювався у деталі мови програмування C++. Наведена програма використовує деякі можливості об'єктно-орієнтованого програмування, крім того активно використовується передача аргументів у функції за послиланнями «&» та модифікатори **const**, що не дозволяють змінювати значення аргументів, переданих за посиланням. Крім того, використані також і можливості перевантаження операторів, що знову-таки може здатися незнайомим. Сподіваємося, що у такому разі білі плями будуть легко відновленні за допомогою пошуку в веб-джерелах і це не стане на заваді розумінню розв'язку, а лише розширить знання технік програмування.

Перш за все нам знадобиться структура для зручної роботи з точками. По суті своїй це буде пара цілих змінних. Додатково ми перевантажимо операто порівняння, а також оператор різниці. Останній буде повертати, по суті, вектор з двох координат, але зрозуміло, що вектор, як і точка, має дві координати (на площині), тож на практиці оператор різниці

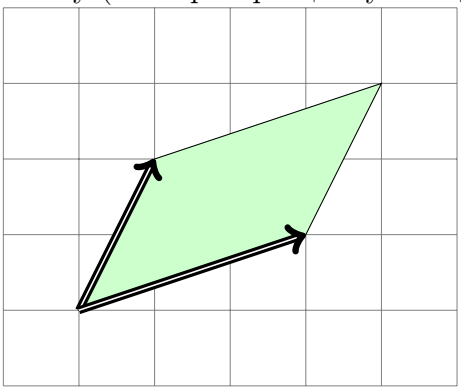
буде повертати точку (при чому ми все ще тримаємо в голові той факт, що по суті свої це все-таки вектор). Крім того, нам знадобиться функція, що буде обчислювати значення векторного добутку двох векторів, побудованих на 3 точках (точніше, довжину вектора, що є результатом векторного добутку). Знадобиться також цілочисленний $abs()$ (оскільки стандартна бібліотека C++ дає нам таку функцію лише для чисел з плаваючою комою), а також $sign(x)$. Функція $sign$ є еквівалентом математичної sgn повертає 0, якщо аргумент рівний 0, 1 у випадку якщо аргумент додатній та -1 у випадку, якщо аргумент від'ємний. Ну і завершує набір $vector\_mul\_sign$ з доволі красномовною назвою, шо буде повертати «знак» векторного добутку двох векторів. Необхідність цих функцій буде обгрунтована трохи пізніше.

```
1  struct point{
2      long long x, y;
3      point(long long x, long long y)
4      : x(x), y(y){}
5  };
6
7  bool operator == (const point & A, const point & B){
8      return (A.x == B.x) && (A.y == B.y);
9  }
10
11 point operator - (const point & A, const point & B){
12     return point(A.x - B.x, A.y - B.y);
13 }
14
15 long long vector_mul(const point & A, const point & B){
16     return A.x*B.y - A.y*B.x;
17 }
18
19 int abs(int a){
20     if(a < 0)
21         return -a;
22     return a;
23 }
24 int sign(long long a){
25     if(a < 0)
26         return -1;
27     if(a > 0)
28         return 1;
29     return 0;
30 }
31
32 int vector_mul_sign(const point & a, const point & c, const point & b){
33     return sign(vector_mul(a - c, b - c));
34 }
```

Крім того, нам знадобиться структура, що представлятиме траншею. У ній ми будемо зберігати координати початку і кінця траншеї. Тут же оголосимо глобальний масив траншей розміру **MAXN** (20) і розмір масиву n, що буде використовуватися (буде зчитаний у тілі головної функції програми і змінюватись у ході розв'язку). Крім того, для зручності були перевантажені оператори для роботи з потоками вводу і виводу (cin та cout, зокрема).

```
struct trench{
    point a, b;

    trench(point a, point b)
    : a(a), b(b){}

    trench(long long x1 = 0, long long y1 = 0, long long x2 = 0, long long
        y2 = 0)
    : a(x1, y1), b(x2, y2){}
};

trench tr[MAXN];
int n;


istream & operator >> (istream & in, trench & to_read){
    in >> to_read.a.x >> to_read.a.y >> to_read.b.x >> to_read.b.y;
    return in;
}

ostream & operator << (ostream & out, trench & to_read){
    out << to_read.a.x << ' ' << to_read.a.y << ' ' << to_read.b.x << ' ' <<
        to_read.b.y;
    return out;
}
```

Далі нам знадобиться функція, що буде «намагатися» злити траншеї (у випадку якщо вони колінеарні та мають спільну точку) у одну довгу. Тут нам знадобиться раніше оголошена функція, що повертає знак векторного добутку. Справа в тім, що числове значення векторного добутку дорівнює площі паралелограма, побудованого на векторах як на сторонах (правда, ще воно може бути додатнє чи від'ємне в залежності від взаємного розташування векторів). Тобто якщо у нас є два вектори (1, 2) та (3, 1), маємо наступну картинку (вектори при цьому зміщені у спільну точку початку):



Замальована область - побудований паралелограм. Зрозуміло, що в колінеарних векторів площа паралелограма, побудованого на цих векторах буде нульовою. Саме цей факт ми і використовуємо у функції злиття двох відрізків.

Інша функція буде перевіряти, чи мають траншеї спільну точку перетину. Це класична задача, вирішена в нашому випадку за допомогою векторних добутків. Детально можна прочитати в книжці «Алгоритми. Побудова і аналіз.» Т.Кормена, Ч.Лейзерсона, Р.Рівеста, К.Штайна у розділі обчислювальної геометрії [1] або ж на сайті `http://e-maxx.ru/algo/segments_intersection_checking`.

[1] Екземпляр цієї книжки можна знайти у сімнадцятому кабінеті ФМГ №17

```
bool try_to_unite_trenches(int i, int j){
    if(tr[i].a == tr[j].a && vector_mul_sign(tr[i].b, tr[i].a, tr[j].b) ==
       0){
        tr[i] = trench(tr[i].b, tr[j].b);
        tr[j] = tr[n - 1];
        --n;
        return 1;
    }
    if(tr[i].a == tr[j].b && vector_mul_sign(tr[i].b, tr[i].a, tr[j].a) ==
       0){
        tr[i] = trench(tr[i].b, tr[j].a);
        tr[j] = tr[n - 1];
        --n;
        return 1;
    }
    if(tr[i].b == tr[j].a && vector_mul_sign(tr[i].a, tr[i].b, tr[j].b) ==
       0){
        tr[i] = trench(tr[i].a, tr[j].b);
        tr[j] = tr[n - 1];
        --n;
        return 1;
    }
    if(tr[i].b == tr[j].b && vector_mul_sign(tr[i].a, tr[i].b, tr[j].a) ==
       0){
        tr[i] = trench(tr[i].a, tr[j].a);
        tr[j] = tr[n - 1];
        --n;
        return 1;
    }

    return 0;
}

bool intersected(const trench & L, const trench & K){
    int signs[4] = {vector_mul_sign(L.a, L.b, K.a),
                    vector_mul_sign(L.a, L.b, K.b),
                    vector_mul_sign(K.a, K.b, L.a),
                    vector_mul_sign(K.a, K.b, L.b)};
    int res_1 = abs(signs[0] - signs[1]),
        res_2 = abs(signs[2] - signs[3]);
    if(res_1 && res_2){
        return 1;
    }

    return 0;
}
```

Нам також знадобиться функція, що повертатиме точку перетину двох відрізків. Координати точки A перетину відрізків $((x_1, y_1), (x_2, y_2))$ та $((x_3, y_3), (x_4, y_4))$ можна знайти як визначники матриць:

$$A_x = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}, A_y = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}$$

Якщо розкрити визначники, отримаємо наступні формули:

$$A_x = \frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)},$$

$$A_y = \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

Чудово, однак якщо ділити напряму, можлива втрата точності, що уноможливить перевірку рівності точок. Помітимо, що знаменник в обох дробів рівні, тож можемо повернути його у якості другої частини пари, не виконуючи ділення в явному вигляді (першою частиною пари буде точка, однак (уявно) домножена на власний той самий знаменник).

Ще одна функція, що нам знадобиться - функція перевірки трьох траншей на виконання умови «попарно перетинаються у трьох різних точках». Тут все легко - ми перевіряємо кожну пару траншей на перетин. У випадку, якщо вони всі перетинаються, ми шукаємо, до прикладу, точки перетину траншей A і B та B і C. Тепер залишається всього лише переконатися в тому, що це дві різних точки (не забуваючи про те, що у нас окремо чисельники і знаменники).

```
1  pair<point, long long> point_of_intersection(const trench & L, const trench
       & K){
2      return make_pair(point((L.a.x*L.b.y - L.a.y*L.b.x)*(K.a.x - K.b.x) - (L.
           a.x - L.b.x)*(K.a.x*K.b.y - K.a.y*K.b.x),
3                              (L.a.x*L.b.y - L.a.y*L.b.x)*(K.a.y - K.b.y) - (L.
                                  a.y - L.b.y)*(K.a.x*K.b.y - K.a.y*K.b.x)),
4                              (L.a.x - L.b.x)*(K.a.y - K.b.y) - (L.a.y - L.b.y)
                                  *(K.a.x - K.b.x));
5  }
6
7  bool intersected_in_different_points(const trench & A, const trench & B,
       const trench & C){
8      if(intersected(A, B) && intersected(B, C) && intersected(A, C)){
9          pair<point, long long> p1 = point_of_intersection(A, B), p2 =
               point_of_intersection(B, C);
10         if(p1.first.x*p2.second == p2.first.x*p1.second &&
11            p1.first.y*p2.second == p2.first.y*p1.second){
12             return 0;
13         }
14         return 1;
15     } else {
16         return 0;
17     }
18 }
```

Ну і останній штрих - головна функція. Спершу ми зчитуємо вхідні данні, потім об'єднуємо траншеї, які можемо об'єднати, після чого перебираємо трійки траншей і перевіряємо їх на виконання умови. В кінці виводимо результат.

```
int main(){
    cin >> n;
    for(int i = 0; i < n; i++){
        cin >> tr[i];
    }
    {
        bool flag = 1;
        while(flag){
            flag = 0;
            for(int i = 0; i < n; i++)
                for(int j = i + 1; j < n; j++){
                    flag |= try_to_unite_trenches(i, j);
                }
        }
    }
    int answer = 0;
    for(int i = 0; i < n; i++)
        for(int j = i + 1; j < n; j++)
            for(int k = j + 1; k < n; k++){
                answer += intersected_in_different_points(tr[i], tr[j], tr[k
                    ]);
            }
    cout << answer;

    return 0;
}
```

Уся програма виглядатиме наступним чином:

```
#include<iostream>
#include<cmath>

using namespace std;

const int MAXN = 20;

struct point{
    long long x, y;
    point(long long x, long long y)
    : x(x), y(y){}
};

bool operator == (const point & A, const point & B){
    return (A.x == B.x) && (A.y == B.y);
}

point operator - (const point & A, const point & B){
    return point(A.x - B.x, A.y - B.y);
}

long long vector_mul(const point & A, const point & B){
    return A.x*B.y - A.y*B.x;
}

int abs(int a){
```

```
27      if(a < 0)
28          return -a;
29      return a;
30  }
31  int sign(long long a){
32      if(a < 0)
33          return -1;
34      if(a > 0)
35          return 1;
36      return 0;
37  }
38
39  int vector_mul_sign(const point & a, const point & c, const point & b){
40      return sign(vector_mul(a - c, b - c));
41  }
42
43  struct trench{
44      point a, b;
45
46      trench(point a, point b)
47      : a(a), b(b){}
48
49      trench(long long x1 = 0, long long y1 = 0, long long x2 = 0, long long
            y2 = 0)
50      : a(x1, y1), b(x2, y2){}
51  };
52
53  trench tr[MAXN];
54  int n;
55
56
57  istream & operator >> (istream & in, trench & to_read){
58      in >> to_read.a.x >> to_read.a.y >> to_read.b.x >> to_read.b.y;
59      return in;
60  }
61
62  ostream & operator << (ostream & out, trench & to_read){
63      out << to_read.a.x << ' ' << to_read.a.y << ' ' << to_read.b.x << ' ' <<
            to_read.b.y;
64      return out;
65  }
66
67  bool try_to_unite_trenches(int i, int j){
68      if(tr[i].a == tr[j].a && vector_mul_sign(tr[i].b, tr[i].a, tr[j].b) ==
            0){
69          tr[i] = trench(tr[i].b, tr[j].b);
70          tr[j] = tr[n - 1];
71          --n;
72          return 1;
73      }
74      if(tr[i].a == tr[j].b && vector_mul_sign(tr[i].b, tr[i].a, tr[j].a) ==
            0){
75          tr[i] = trench(tr[i].b, tr[j].a);
76          tr[j] = tr[n - 1];
77          --n;
78          return 1;
79      }
80      if(tr[i].b == tr[j].a && vector_mul_sign(tr[i].a, tr[i].b, tr[j].b) ==
            0){
81          tr[i] = trench(tr[i].a, tr[j].b);
```

```
82          tr[j] = tr[n - 1];
83          --n;
84          return 1;
85      }
86      if(tr[i].b == tr[j].b && vector_mul_sign(tr[i].a, tr[i].b, tr[j].a) ==
            0){
87          tr[i] = trench(tr[i].a, tr[j].a);
88          tr[j] = tr[n - 1];
89          --n;
90          return 1;
91      }
92
93      return 0;
94  }
95
96  bool intersected(const trench & L, const trench & K){
97      int signs[4] = {vector_mul_sign(L.a, L.b, K.a),
98                      vector_mul_sign(L.a, L.b, K.b),
99                      vector_mul_sign(K.a, K.b, L.a),
100                     vector_mul_sign(K.a, K.b, L.b)};
101     int res_1 = abs(signs[0] - signs[1]),
102         res_2 = abs(signs[2] - signs[3]);
103     if(res_1 && res_2){
104         return 1;
105     }
106
107     return 0;
108 }
109
110 pair<point, long long> point_of_intersection(const trench & L, const trench
        & K){
111     return make_pair(point((L.a.x*L.b.y - L.a.y*L.b.x)*(K.a.x - K.b.x) - (L.
            a.x - L.b.x)*(K.a.x*K.b.y - K.a.y*K.b.x),
112                            (L.a.x*L.b.y - L.a.y*L.b.x)*(K.a.y - K.b.y) - (L.
                               a.y - L.b.y)*(K.a.x*K.b.y - K.a.y*K.b.x)),
113                            (L.a.x - L.b.x)*(K.a.y - K.b.y) - (L.a.y - L.b.y)
                               *(K.a.x - K.b.x));
114 }
115
116 bool intersected_in_different_points(const trench & A, const trench & B,
        const trench & C){
117     if(intersected(A, B) && intersected(B, C) && intersected(A, C)){
118         pair<point, long long> p1 = point_of_intersection(A, B), p2 =
                point_of_intersection(B, C);
119         if(p1.first.x*p2.second == p2.first.x*p1.second &&
120            p1.first.y*p2.second == p2.first.y*p1.second){
121             return 0;
122         }
123         return 1;
124     } else {
125         return 0;
126     }
127 }
128
129
130 int main(){
131     cin >> n;
132     for(int i = 0; i < n; i++){
133         cin >> tr[i];
134     }
```

```
135     {
136         bool flag = 1;
137         while(flag){
138             flag = 0;
139             for(int i = 0; i < n; i++)
140                 for(int j = i + 1; j < n; j++){
141                     flag |= try_to_unite_trenches(i, j);
142                 }
143         }
144     }
145     int answer = 0;
146     for(int i = 0; i < n; i++)
147         for(int j = i + 1; j < n; j++)
148             for(int k = j + 1; k < n; k++){
149                 answer += intersected_in_different_points(tr[i], tr[j], tr[k
                        ]);
150             }
151     cout << answer;
152 
153     return 0;
154 }
```

Задача Carriage

Для легшого розуміння подальших формул залишимо тут картинку з умови задачі:

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36
53	54	51	52	49	50	47	48	45	46	43	44	41	42	39	40	37	38

Перш за все розділимо задачу на два пункти. Нам треба знайти повністю пусті купе (адже лише вони нас влаштовують), після чого знайти довжину найбільшої послідовності купе, що стоять поруч.

Пронумеруємо купе від 0 до 8. Нехай місця 1, 2, 3, 4, а також 53 та 54 належать 0-му купе; 5, 6, 7, 8, 51 та 52 належать 1-му купе і так далі. Неважко побачити, що формула визначення номеру купе за номером місця виглядатиме наступним чином:

$$stateroom(place) = \begin{cases} \lfloor \frac{place-1}{4} \rfloor & , place \leq 36 \\ 8 - \lfloor \frac{place-37}{2} \rfloor & , place > 36 \end{cases}$$

Тож заведемо табличку на 9 елементів, спершу заповнену нулями, кожна комірка якої буде показувати кількість вільних місць у відповідному купе. Читаючи вхідні данні (вільні місця), будемо знаходити за вищенаведеною формулою потрібне купе і збільшувати відповідну комірку на 1. Якщо після обробки вхідних даних значення якоїсь комірки буде рівне 6, це означатиме, що відповідне купе повністю вільне (адже має 6 вільних місць).

Перейдемо до другої частини алгоритму. Це типова підзадача - знайти найбільший за довжиною підвідрізок таблиці, де кожне значення рівне певному числу (у нашому випадку 6). Заведемо допоміжні змінні: max_seq_len, що відповідатиме за довжину найбільшого підвідрізка, що вже знайдена, а також cur_seq_len, що відповідатиме за максимальну довжину послідовності з шісток, що починається десь раніше поточної позиції, а закінчуєтья рівно на цій позиції. Обидві змінні з очевидних міркувань спершу встановлюємо нулями. Оброблятимемо купе починаючи з 0-го і до 8-го. У разі, якщо поточне купе вільне (значення відповідної комірки рівне 6) - збільшуємо cur_seq_len. Інакше послідовність з шісток «щойно закінчилась», тож у разі якщо довжина поточної послідовності більше за max_seq_len - змінюємо останне, але в будь-якому разі обнуляємо значення cur_seq_len. Спробу оновити результат слід зробити і у кінці, після обробки 8-го купе, адже у нас може бути послідовність з шісток, що закінчується на останньому елементі.

Розв'язок на C++ виглядатиме наступним чином:

```
1  #include<iostream>
2  #include<algorithm>
3
4  using namespace std;
5
6  int main(){
7       int staterooms[9] = {};
8
9       int n;
10      cin >> n;
11
12      for(int i = 0; i < n; i++){
13           int place;
14           cin >> place;
15           if(place <= 36){
16                staterooms[(place - 1)/4]++;
```

```
17            } else {
18                 staterooms[8 - (place - 37)/2]++;
19            }
20       }
21
22       int max_seq_len = 0, cur_seq_len = 0;
23       for(int i = 0; i < 9; i++){
24            if(staterooms[i] == 6){
25                 cur_seq_len++;
26            } else {
27                 max_seq_len = max(max_seq_len, cur_seq_len);
28                 cur_seq_len = 0;
29            }
30       }
31       max_seq_len = max(max_seq_len, cur_seq_len);
32
33       cout << max_seq_len;
34
35       return 0;
36  }
```