Задача A. Великое Лайнландское переселение

Данная задача была рассмотрена на лекции. Ниже приведен пример решения задачи на языке C++:

```
#include <bits/stdc++.h>

using namespace std;

const int N = 100100;

int a[N], ans[N];

int main(void) {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    stack<int> st;
    for (int i = n - 1; i >= 0; --i) {
        while (!st.empty() && a[st.top()] >= a[i]) {
            st.pop();
        }
        if (st.empty()) {
            ans[i] = -1;
        } else {
            ans[i] = st.top();
        }
        st.push(i);
    }

    for (int i = 0; i < n; ++i) {
        cout << ans[i] << ' ';
    }
    cout << '\n';

    return 0;
}
```

Задача B. Минимум на отрезке

Данная задача была рассмотрена на лекции. Ниже приведен пример решения на языке C++. В данном решении стек с поддержкой минимума, а также очередь с поддержкой минимума реализованы в виде структур для удобства использования. У обеих структур есть функции `push(x)` (добавить элемент), `pop()` (удалить элемент), а также `get_min()` вернуть минимум среди всех хранящихся элементов.

```
#include <bits/stdc++.h>

using namespace std;

struct min_stack {
```

```
    stack<pair<int, int>> st;

    void push(int x) {
        if (st.empty()) {
            st.push({x, x});
        } else {
            st.push({x, min(x, st.top().second)});
        }
    }

    void pop() {
        st.pop();
    }

    int top() {
        return st.top().first;
    }

    int get_min() {
        return st.top().second;
    }

    bool empty() {
        return st.empty();
    }
};

struct min_queue {
    min_stack head, tail;

    void push(int x) {
        tail.push(x);
    }

    void pop() {
        if (head.empty()) {
            while (!tail.empty()) {
                head.push(tail.top());
                tail.pop();
            }
        }
        head.pop();
    }

    int get_min() {
        if (head.empty()) {
            return tail.get_min();
        }
        if (tail.empty()) {
            return head.get_min();
        }
        return min(head.get_min(), tail.get_min());
    }
```

```
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, k;
    cin >> n >> k;

    vector<int> a(n);
    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    min_queue q;
    for (int i = 0; i < k - 1; ++i) {
        q.push(a[i]);
    }
    for (int i = k - 1; i < n; ++i) {
        q.push(a[i]);
        cout << q.get_min() << '\n';
        q.pop();
    }

    return 0;
}
```

Задача C. ORные подотрезки

Для начала заметим, что $a \mid b \geqslant a$ и $a \mid b \geqslant b$ для любых целых неотрицательных чисел a и b. Это верно, так как если некоторый бит был равен единице, он останется таковым после выполнения операции. А если некоторый бит был равен нулю, он либо останется таковым, либо заменится на единицу после выполнения операции.

Также заметим, что можно реализовать очередь с поддержкой побитового «ИЛИ» всех хранящихся в ней элементов. Это делается аналогично очереди с поддержкой минимума. Достаточно лишь заменить операцию взятия минимума на операцию вычисления побитового «ИЛИ».

Теперь задача решается следующим образом. Зафиксируем правую границу отрезка r. Будем увеличивать левую границу до тех пор, пока значение побитового «ИЛИ» на отрезке не меньше, чем k. Для вычисления данного значения будем при расширении отрезка добавлять очередное число в очередь, а при сужении отрезка — удалять число из очереди. Как только значение побитового «ИЛИ» стало меньше, чем k, остановим процесс сужения отрезка и прибавим к ответу $l+1$, так как для любой левой границы, меньшей l, значение побитового «ИЛИ» будет не меньше, чем k. После этого увеличим правую границу на 1, добавив очередное число в очередь, и продолжим сужать отрезок, увеличивая левую границу.

Основная часть решения задачи, реализованная на языке C++, выглядит следующим образом:

```
int l = 0;
long long ans = 0;
for (int r = 0; r < n; ++r) {
    q.push(a[r]);
    if (q.get_or() >= k) {
        while (q.get_or() >= k && l <= r) {
            ++l;
            q.pop();
        }
```

```
        --l;
        q.push(a[l]);

        ans += l + 1;
    }
}
```

Задача D. Колонизаторы - 2

Данная задача решается «в лоб» при помощи алгоритма из первой задачи.

Для начала найдем для каждого элемента ближайший элемент справа, больший данного. После этого пройдем по всем элементам, вычтем единицу из текущего элемента и прибавим единицу к найденному ближайшему справа большему элементу.

После этого найдем для каждого элемента ближайший элемент слева, больший данного. Наконец, пройдем по всем элементам, вычтем единицу из текущего элемента и прибавим единицу к найденному ближайшему слева большему элементу.

Пример кода, находящего ближайший меньший элемент, можно увидеть в разборе первой задачи. Ближайший больший элемент ищется аналогично.

Задача E. Наибольший общий делитель

В данной задаче достаточно вспомнить идею очереди с поддержкой минимума и применить ее для вычисления НОД. Реализацию очереди с поддержкой минимума можно увидеть во второй задаче. Далее для решения данной задачи необходимо заменить вычисления минимума на вычисление НОД. Для этого в языках C++ и Python есть функция `gcd`.

Задача G. Гистограмма

Данная задача была рассмотрена на лекции. Для начала найдем для каждого элемента ближайшие элементы слева и справа, меньшие данного. Пусть индексы этих элементов равны `l[i]` и `r[i]` (для удобства скажем, что `l[i] == -1`, если все элементы слева больше, чем `h[i]`; аналогично скажем, что `r[i] == n`, если все элементы справа больше, чем `h[i]`).

Код для нахождения ближайшего меньшего элемента справа можно увидеть в разборе первой задачи. Далее необходимо перебрать текущий элемент и обновить ответ. Пример основной части решения на языке C++:

```
long long ans = 0;
for (int i = 0; i < n; ++i) {
    ans = max(ans, 1ll * h[i] * (r[i] - l[i] - 1));
}
```

Задача H. Очередь в магазине

Подзадача 1

Для получение 40 баллов по данной задаче необходимо было проэмулировать процесс, описанный в условии. Можно, например, поддерживать очередь в контейнере `deque`. Операции первого и третьего типов выполняются за $\mathcal{O}(1)$. Операцию второго типа можно выполнять явно за $\mathcal{O}(n)$, где n — количество элементов в деке. Таким образом, время работы составит $\mathcal{O}(n^2)$.

Подзадача 2

Рассмотрим решение, позволяющее набрать полный балл по данной задаче. Понятно, что для того, чтобы получить достаточно быстрое решение, необходимо научиться обрабатывать событие второго типа достаточно быстро. Заведем некоторую переменную d, изначально равную нулю, в которой мы будем накапливать некоторое значение агрессивности, которое нужно прибавить ко всем людям в очереди.

Теперь рассмотрим, как, имея посчитанное значение d, обработать каждое из трех возможных событий. Пусть произошло событие первого типа. Тогда в очередь нужно добавить число $a - d$, так как мы подразумеваем, что число d должно быть прибавлено ко всем элементам очереди. Если произошло второго типа, то нужно увеличить d на y, а первое число очереди увеличить на $x - y$. В случае, если произошло событие третьего типа, нужно извлечь первое число из очереди, и записать в ответ это число, увеличенное на d.

В таком случае все операции обрабатываются за $\mathcal{O}(1)$, а итоговое время работы равно $\mathcal{O}(n)$.

Пример решения задачи на языке C++:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;

    queue<ll> q;
    ll push = 0;
    while (n--) {
        int type;
        cin >> type;

        if (type == 1) {
            int a;
            cin >> a;
            q.push(a - push);
        } else if (type == 2) {
            int x, y;
            cin >> x >> y;

            push += y;
            q.front() += x - y;
        } else {
            cout << q.front() + push << '\n';
            q.pop();
        }
    }

    return 0;
}
```

Задача I. Ксюша и покемоны

Воспользуемся очередью с минимумом для того, чтобы найти минимум на каждом подотрезке массива длины k. Также будем поддерживать сумму на текущем подотрезке длины k. Для этого при перемещении «окна» на один элемент вправо нужно прибавить к сумме добавленный элемент и вычесть из суммы удаленный элемент.

Для каждого отрезка длины k вычислим произведение минимума и суммы, после чего обновим ответ.

Реализация основной части решения на языке C++ приведена ниже:

```
min_queue q;
```

```
ll sum = 0;
ll ans = 0;
for (int i = 0; i < k - 1; ++i) {
    q.push(a[i]);
    sum += a[i];
}
for (int i = k - 1; i < n; ++i) {
    q.push(a[i]);
    sum += a[i];
    ans = max(ans, q.get() * sum);
    q.pop();
    sum -= a[i - k + 1];
}
```

Задача J. Прыгающий робот

Автор задачи: Елена Андреева

Подзадача 1

Для решения первой подзадачи можно воспользоваться полным перебором. Заметим, что значения $a > \max\{d_i\}$ нет смысла перебирать.

Зафиксируем первую платформу и значение a. Проверим, подходит ли данное значение.

Асимптотика решения $O(n^2 \max\{d_i\})$.

Подзадача 2

Избавимся от перебора значений a. Пусть мы зафиксировали начальную платформу. Найдем минимальное значение a, которое подходит. Инициализируем a как 0. Пробежимся по всем платформам, если текущей ловкости не хватает, чтобы перепрыгнуть на следующую, мы знаем, на какую величину ее надо увеличить. При этом увеличение ловкости не влияет на возможность совершать прыжки между рассмотренными ранее платформам. Получаем решение за $O(n^2)$.

Подзадача 3

Здесь нам задана начальная платформа, поэтому можно не реализовывать ее перебор. Применим решение предыдущей подзадачи и найдем подходящее начальное значение ловкости.

Подзадача 4

Рассмотрим еще раз внимательно процесс поиска начальной ловкости после фиксирования стартовой платформы i:

```
int a = 0;
for (int k = 0; k < n; k++) {
    if (d[(i + k) % n] > a) {
        a = d[(i + k) % n];
    }
    a++;
}
a -= n;
```

Изменим немного этот фрагмент, избавимся от операции `a++`, включив накопленное значение в сравнение:

```
int a = 0;
for (int k = 0; k < n; k++) {
    if (d[(i + k) % n] > a + k) {
        a = d[(i + k) % n] - k;
    }
}
```

Перенося `+ k` на другую сторону неравенства, получаем окончательно код

```
int a = 0;
for (int k = 0; k < n; k++) {
    if (d[(i + k) % n] - k > a) {
        a = d[(i + k) % n] - k;
    }
}
```

в котором легко узнать поиск максимума в массиве $d[(i+k)\%n]-k$.

Заменим массив d на $d'[k] = d[k]-k$ и удвоим его: $d'[k+n] = d'[k]+n$. Тогда, чтобы учесть сдвиг индекса на i, надо взять максимум значений этого массива на полуинтервале $[i, i+n)$ и прибавить к нему значение i.

Таким образом мы свели задачу к задаче поиска максимума на отрезке. Эта задача широко известна и в этой подзадаче можно применить любую из известных структур данных, например дерево отрезков или разреженную таблицу.

Подзадача 5

Это «учебная» подзадача, которая проверяет, что решение участников умеет генерировать массив по формуле для $f = 2$. Поскольку перебирать начальную платформу не надо, решение за линейное время с поиском максимума в массиве работает.

Поздадача 6

В этой подзадаче размер массива 10^7, поэтому разреженные таблицы не помещаются в память, а дерево отрезков сверху не проходит по времени. Можно либо использовать очень оптимизированное дерево отрезков в реализации снизу, дерево Фенвика, либо воспользоваться структурой данных для «максимума в окне».

Есть несколько разных реализаций этой структуры данных, одна из наиболее простых следующая. Нам нужно найти максимумы на полуинтервалах $[0, n)$, $[1, n+1)$, $\ldots$, $[n; 2n)$. Посчитаем суффиксные максимумы на полуинтервалах $[0, n)$, $[1, n)$, $[2, n)$, и так далее, а также префиксные максимумы на полуинтервалах $[n, n+1)$, $[n, n+2)$, и так далее. Заметим, что максимум на нужном нам полуинтервале это максимум из двух предподсчитанных значений.