Оглавление

Глава 1

Префиксные суммы

1.1 Постановка задачи

Дан массив $a_0, a_1, ...a_{n-1}$, состоящий из n целых чисел, и q запросов. Для каждого запроса, характеризующегося двумя целыми числами l и r $(0 \leq l \leq r \leq n-1)$, требуется найти сумму $a_l + a_{l+1} + ... + a_r$.

1.2 Наивное решение: линейный поиск

Простейшее решение — для каждого запроса пройтись по массиву, начиная с позиции l и заканчивая позицией r. Алгоритм достаточно прост в реализации, но неэффективен для множества запросов на достаточно больших массивах.

Листинг 1.1: Линейный поиск суммы для каждого запроса

```
vector<int> a;

int Find(int l, int r) {
    int S = 0;
    for (int i = l; i <= r; i++) {
        S += a[i];
    }
    return S;
}

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

    int q; cin >> q;
    for (int i = 0; i < q; i++) {
        int l, r; cin >> l >> r;
        cout << Find(l, r) << '\n';
    }
}
```

- **Временная сложность**: $O(n)$ на запрос
- **Пространственная сложность**: $O(1)$

1.3 Оптимальное решение: префиксные суммы

Префиксные суммы — это структура данных, позволяющая быстро вычислять сумму элементов массива на отрезке. Для массива $a[0..n-1]$ префиксным массивом называется массив $p[0..n-1]$, где: p[i] $= \sum_{k=0}^{i} a_k$

Теперь для ответа на запрос суммы элементов на позициях с l до r нужно лишь посмотреть на значение:

- $a_l + ... + a_r = a_0 + ... + a_r = p_r$, если $l = 0$.
- $a_l + ... + a_r = (a_0 + ... + a_r) - (a_0 + ... + a_{l-1}) = p_r - p_{l-1}$, если $l \neq 0$.

Листинг 1.2: Поиск суммы с помощью префиксных сумм

```
vector<int> a;
vector<int> p;

int Find(int l, int r) {
    if (l == 0) {
        return p[r];
    }
    else {
        return p[r] - p[l - 1];
    }
}

void Solve() {
    int n; cin >> n;

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

    p.resize(n); int S = 0;
    for (int i = 0; i < n; i++) {
        S += a[i];
        p[i] = S;
    }

    int q; cin >> q;
    for (int i = 0; i < q; i++) {
        int l, r; cin >> l >> r;
        cout << Find(l, r) << '\n';
    }
}
```

- **Временная сложность**: $O(1)$ на запрос и $O(n)$ на предпосчет
- **Пространственная сложность**: $O(1)$ на запрос и $O(n)$ на предпосчет

1.4 Задачи

1.4.1 Задача 1: Префиксный XOR

Условие: Дан массив $a_0, a_1, ..., a_{n-1}$ из n целых чисел. Требуется находить XOR на отрезке $[l, r]$.

Решение: Аналогично обычным префиксным суммам, построим массив префиксных XOR:

$$p_i = \begin{cases} a_i & \text{при } i = 0 \\ p_{i-1} \oplus a_i & \text{при } i > 0 \end{cases}$$

Тогда XOR на отрезке $[l, r]$ вычисляется как: p_r, если $l = 0$ или $p_r \oplus p_{l-1}$, если $l \neq 0$.

1.4.2 Задача 2: 2D префиксные суммы

Условие: Дана матрица $n \times m$. Требуется отвечать на запросы суммы элементов в прямоугольнике с верхним левым углом $(x1, y1)$ и правым нижнем $(x2, y2)$.

Решение: Строим матрицу префиксных сумм:

$$p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + a[i][j]$$

Сумма в прямоугольнике:

$$sum = p[x2][y2] - p[x1-1][y2] - p[x2][y1-1] + p[x1-1][y1-1]$$

1.4.3 Задача 3: Поиск подотрезка с заданной суммой

Условие: Дан массив целых чисел и число S. Найти любой подотрезок с суммой равной S (или определить, что такого нет).

Решение: Используем префиксные суммы и хэш-таблицу: вычисляем префиксные суммы $p[0...n-1]$ и для каждого p_i ищем в хэш-таблице значение $p_i - S$. Если находим, то отрезок между этими индексами даёт сумму S.

1.4.4 Задача 4: Количество подотрезков с заданной суммой

Условие: Дан массив целых чисел и число S. Найти количество подотрезков с суммой равной S.

Решение: Модификация предыдущего подхода - вместо хранения последней позиции для каждой суммы, храним количество вхождений каждой суммы.

Глава 2

Разностный массив

2.1 Постановка задачи

Дан массив $a_0, a_1, ...a_{n-1}$, состоящий из n целых чисел, и q запросов. Для каждого запроса, характеризующегося тремя целыми числами l, r и x ($0 \leq l \leq r \leq n-1$), нужно прибавить x ко всем элементам: $a_l, a_{l+1}, ...a_r$. После всех операций необходимо вывести конечный массив.

2.2 Наивное решение: линейные обновления

Простейшее решение — для запроса пройтись по массиву, начиная с позиции l и заканчивая позицией r, прибавляя к каждому элементу число x. Алгоритм достаточно прост в реализации, но неэффективен для множества запросов на достаточно больших массивах.

Листинг 2.1: Линейные обновления для каждого запроса

```
vector<int> a;

int Upd(int l, int r, int x) {
    for (int i = l; i <= r; i++) {
        a[i] += x;
    }
}

void Solve() {
    int n; cin >> n;

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

    int q; cin >> q;
    for (int i = 0; i < q; i++) {
        int l, r, x; cin >> l >> r >> x;
        Upd(l, r, x);
    }

    for (auto e : a) {
        cout << e << ' ';
    }
}
```

- **Временная сложность**: $O(n)$ на запрос
- **Пространственная сложность**: $O(1)$

2.3 Оптимальное решение: разностный массив

Разностный массив — это структура данных, обратная префиксным суммам, которая позволяет эффективно выполнять операции прибавления на отрезке.

Для массива $a[0..n-1]$ разностным массивом называется массив $d[0..n]$, где:

$$d[i] = \begin{cases} a[0], & i = 0 \\ a[i] - a[i-1], & 1 \le i < n \\ -a[n-1], & i = n \end{cases}$$

Создаем разностный массив d. Для каждого запроса (l, r, x):

- Прибавляем x к $d[l]$ - все элементы начиная с l будут увеличены на x.
- Вычитаем x из $d[r+1]$ - отменяется прибавление для элементов после r.

На каждом обновлении только элементы с l по r получат прибавку x. Осталось лишь восстановить ans с помощью разностного массива (см. Глава 3).

Листинг 2.2: Обработка изменений с помощью разностного массива

```
vector<int> a;
vector<int> d;

void Upd(int l, int r, int x) {
    d[l] += x;
    d[r + 1] -= x;
}

void Solve() {
    int n; cin >> n;

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

    d.resize(n + 1);
    d[0] = a[0];
    for (int i = 1; i < n; i++) {
        d[i] = a[i] - a[i - 1];
    }
    d[n] = -a[n - 1];

    int q; cin >> q;
    for (int i = 0; i < q; i++) {
        int l, r, x; cin >> l >> r >> x;
        Upd(l, r, x);
    }

    vector<int> ans(n);
    ans[0] = d[0];
    for (int i = 1; i < n; i++) {
        ans[i] = ans[i - 1] + d[i];
    }

    for (auto e : ans) {
        cout << e << ' ';
    }
}
```

- **Временная сложность**: $O(1)$ на запрос и $O(n)$ на предпосчет
- **Пространственная сложность**: $O(1)$ на запрос и $O(n)$ на предпосчет

2.4 Задачи

2.4.1 Задача 1: Прибавление арифметической прогрессии

Условие: Дан массив размера n, изначально заполненный нулями. Дано q запросов вида "прибавить арифметическую прогрессию $u, u+v, u+2v, \ldots$ на отрезке $[l, r]$". После всех операций вывести итоговый массив.

Решение: Создаем разностный массив d. Для запроса (l, r, u, v): $d[l] += u$, $d[l+1] += v$, $d[l+2] += v$, ..., $d[r+1] -= u + (r-l) \cdot v$. Применив разностный массив d' к d все эти операции можно обработать за $O(q)$. После чего восстанавливаем итоговый массив.

2.4.2 Задача 2: Прибавление на прямоугольнике

Условие: Дана матрица $n \times m$, изначально заполненная нулями. Дано q запросов вида "прибавить x на прямоугольнике $(x1, y1) - (x2, y2)$". После всех операций вывести итоговую матрицу.

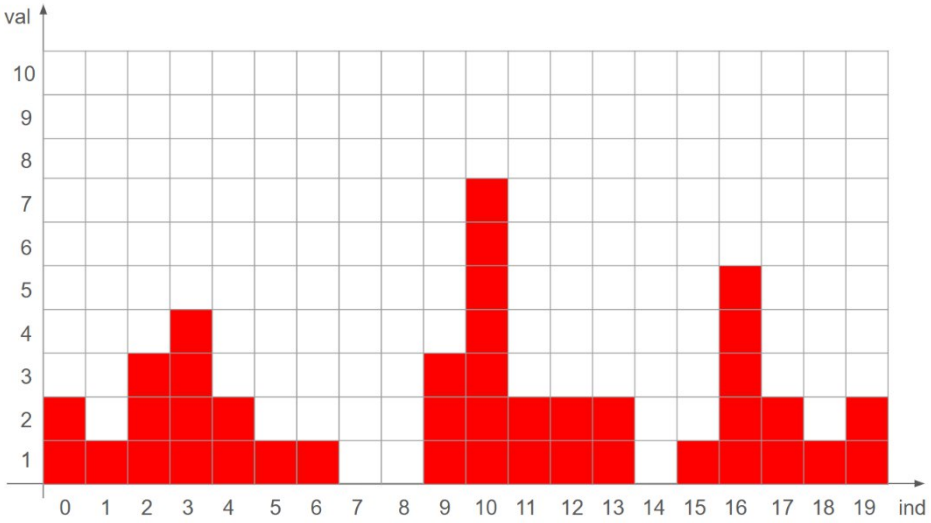
Решение: Создаем 2D разностный массив d размера $(n+1) \times (m+1)$. Для запроса $(x1, y1, x2, y2, x)$: $d[x1][y1] += x$, $d[x2+1][y1] -= x$, $d[x1][y2+1] -= x$, $d[x2+1][y2+1] += x$

Восстанавливаем матрицу: $ans[i][j] = ans[i-1][j] + ans[i][j-1] - ans[i-1][j-1] + d[i][j]$

Глава 3

Связь префиксных сумм и разностного массива

Отметим, что разностный массив и префиксные суммы являются взаимно обратными операциями: применение разностного массива к префиксным суммам дает исходный массив (ведь $a_i = p_i - p_{i-1}$); применение префиксных сумм к разностному массиву также восстанавливает исходный массив, даже после изменений.



Представим произвольный массив a как диаграмму, где номера столбцов представляют собой индексы его элементов, а номера строк — значения. Так, например, $a_4 = 2$, ведь в соответствующем столбце закрашены 2 клетки. a_7, a_8, a_{14} можно считать равными нулю. Составим разностный массив:

2	-1	2	1	-2	-1	0	-1	0	3	4	-5	0	0	-2	1	4	-3	-1	1	-2

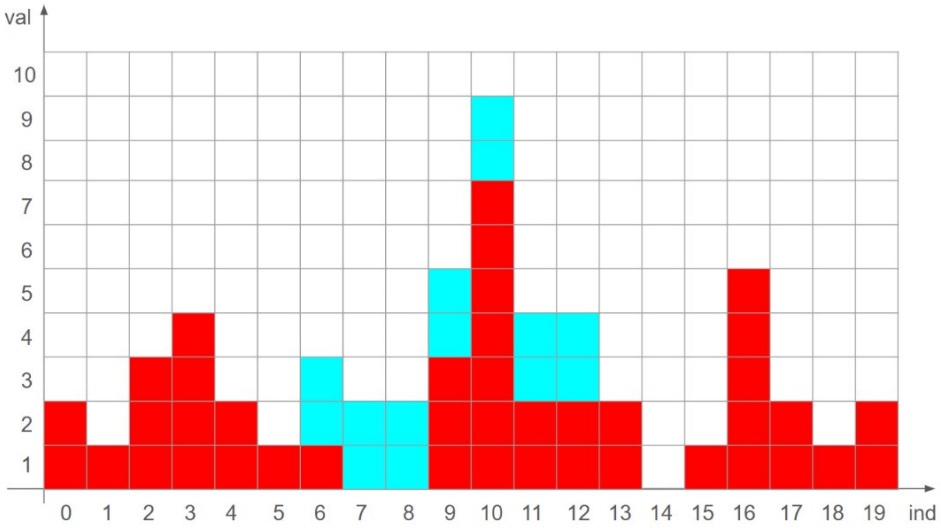
Легко заметить, что:

- $a_0 = d_0$
- $a_1 = a_0 + d_1$ (ведь d_1 обозначает то, насколько a_1 отличается от a_0) $= d_0 + d_1$
- $a_2 = a_1 + d_2 = a_0 + d_1 + d_2 = d_0 + d_1 + d_2$

  ...
- $a_{n-1} = d_0 + d_1 + ... + d_{n-1}$

А значит, применение префиксных сумм к разностному массиву дает исходный массив a.

Теперь попробуем прибавить произвольное число x на случайном подотрезке $[l, r]$. Пусть $x = 2$, $[l, r] = [6, 12]$. Посмотрим на изменения:



2	-1	2	1	-2	-1	2	-1	0	3	4	-5	0	-2	-2	1	4	-3	-1	1	-2

Заметим, что теперь разностный массив корректно задает новый массив a. А значит, применение к нему префиксных сумм (как в прошлом случае) поможет восстановить a, учитывая все изменения.

Глава 4

Стек рекордов

4.1 Постановка задачи

Дан массив $a_0, a_1, ...a_{n-1}$, состоящий из n целых чисел. Нужно для каждой позиции i найти индекс ближайшего элемента слева, строго меньшего текущего a_i.

4.2 Наивное решение: линейный поиск для каждой позиции

Простейшее решение — для каждой позиции i пройтись назад, пока текущий элемент не будет строго меньше a_i. Алгоритм достаточно прост в реализации, но неэффективен для больших массивов.

Листинг 4.1: Линейный поиск для каждой позиции

```
void go(int pos, vector<int>& a) {
    for (int i = pos; i >= 0; i--) {
        if (a[i] < a[pos]) {
            cout << i << ' ';
            return;
        }
    }
    cout << -1 << ' ';
}

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

    for (int i = 0; i < n; i++) {
        go(i, a);
    }
}
```

- **Временная сложность**: $O(n^2)$
- **Пространственная сложность**: $O(n)$

4.3 Оптимальное решение: стек рекордов

Заведём стек, в котором будем хранить индексы i_1, i_2, ..., i_k, поддерживая монотонность элементов a_{i_1}, a_{i_2}, ..., a_{i_k} по возрастанию. Другими словами для каждой новой позиции:

- Удаляем из стека все индексы j (они будут находиться на вершине стека), при которых $a_j \geq a_i$.
- Если стек не пуст - его вершина и будет ближайшим меньшим элементом слева.
- Добавляем текущий индекс i в стек.

Листинг 4.2: Стек рекордов для поиска ближайшего меньшего слева

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

    vector<int> res(n);
    vector<int> st;

    for (int i = 0; i < n; i++) {
        while (st.size() > 0 && a[st[st.size() - 1]] >= a[i]) {
            st.pop_back();
        }

        if (!st.empty()) {
            res[i] = st.top();
        }
        else {
            res[i] = -1;
        }

        st.push_back(i);
    }

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

- **Временная сложность**: $O(n)$
- **Пространственная сложность**: $O(n)$

4.4 Задачи

4.4.1 Задача 1: Число подотрезков, где текущий элемент — минимум

Условие: Дан массив a размера n. Для каждой позиции i найдите количество подотрезков, где $a[i]$ — минимальный элемент.

Решение: Найдем ближайшие меньшие элементы слева и справа. Все отрезки между ними (пересекающие позицию i) содержат $a[i]$ как минимум.

4.4.2 Задача 2: Прямоугольник в гистограмме

Условие: Дан массив высот $h_0, ..., h_{n-1}$. Найти максимальную площадь прямоугольника, который можно вырезать в гистограмме.

Решение: Для каждой высоты найдем ближайшие меньшие элементы слева и справа — это определяет ширину прямоугольника. Чтобы реализовать это, используем стек минимумов с обеих сторон.

4.4.3 Задача 3: Монотонная очередь

Условие: Дан массив a размера n и число k. Для каждого окна (подотрезка исходного массива) длины k найти максимум.

Решение: Используем двустороннюю очередь, в которой поддерживаем элементы в убывающем порядке. Удаляем устаревшие индексы и те, что меньше текущего значения.

Глава 5

Метод двух указателей

5.1 Постановка задачи

Дан **упорядоченный** массив из n целых чисел и целое число x. Требуется проверить, существуют ли два элемента массива, сумма которых равна x.

5.2 Наивное решение: полный перебор

Простейшее решение - проверить все возможные пары элементов. Алгоритм достаточно прост в реализации, но неэффективен для множества запросов на достаточно больших массивах.

Листинг 5.1: Полный перебор

```
bool Find(vector<int>& a, int x) {
    for (int i = 0; i < a.size(); i++) {
        for (int j = 0; j < a.size(); j++) {
            if (i != j && a[i] + a[j] == x) {
                return true;
            }
        }
    }
    return false;
}
```

- **Временная сложность**: $O(n^2)$
- **Пространственная сложность**: $O(1)$

5.3 Оптимальное решение: два указателя

Метод двух указателей — это эффективный алгоритмический подход для решения задач, где нужно обрабатывать последовательности (массивы, строки) за линейное время $O(n)$ с постоянной дополнительной памятью $O(1)$. Суть метода заключается в использовании двух указателей (индексов), которые перемещаются по структуре данных в одном или противоположных направлениях, что позволяет избежать полного перебора.

- Пусть указатель l начинает с первого элемента (индекс 0), а r с последнего (индекс $n-1$)
- Если `a[l] + a[r] < x`: увеличиваем l (берем больший элемент)
- Если `a[l] + a[r] > x`: уменьшаем r (берем меньший элемент)

Алгоритм работает за $O(n)$ (на каждом шаге сдвигается хотя бы один из указателей) и не пропускает возможные пары, так как:

- Если $a[l] + a[r] < x$, то пара $(a[l], a[k])$ для любого $k < r$ тоже даст сумму меньше x (из-за отсортированности). Поэтому l можно увеличивать.
- Если $a[l] + a[r] > x$, то пара $(a[l], a[k])$ для любого $k > r$ тоже даст сумму больше x (из-за отсортированности). Поэтому r можно уменьшать.

Листинг 5.2: Два указателя

```cpp
bool Find(vector<int>& a, int x) {
    int l = 0;
    int r = a.size() - 1;

    while (l < r) {
        if (a[l] + a[r] < x) {
            ++l;
        }
        else if (a[l] + a[r] > x) {
            --r;
        }
        else {
            return true;
        }
    }

    return false;
}
```

- **Временная сложность**: $O(n)$
- **Пространственная сложность**: $O(1)$

5.4 Задачи

5.4.1 Задача 1: Максимальная разница

Условие: Дан массив A размера n, состоящий из целых чисел. Найти максимальную разницу $j - i$, такую что $i < j$ и $A[i] < A[j]$.

Решение: Создаем массив $LMin$, где $LMin[i]$ — минимальный элемент слева от i. Создаем массив $RMax$, где $RMax[j]$ — максимальный элемент справа от j.

Используем два указателя: если $LMin[i] < RMax[j]$, обновляем максимум и двигаем j вправо, иначе двигаем i вправо. Если максимум не найден, возвращаем -1.

5.4.2 Задача 2: Подотрезок с максимальной суммой

Условие: Дан массив и число k. Найти подотрезок длины k с максимальной суммой.

Решение: Вычисляем сумму первых k элементов, а затем на каждом шаге добавляем новый элемент в сумму и вычитаем тот, что вышел из границ подвинутого отрезка.

Глава 6

Бинарный поиск

6.1 Постановка задачи

Дан **упорядоченный** массив из n целых чисел. Требуется проверить, присутствует ли в нём заданный элемент x.

6.2 Наивное решение: линейный поиск

Простейшее решение — проверить каждый элемент массива по порядку. Алгоритм достаточно прост в реализации, но неэффективен для множества запросов на достаточно больших массивах.

Листинг 6.1: Линейный поиск

```
bool Find(vector<int>& a, int x) {
    for (auto p : a) {
        if (p == x) {
            return true;
        }
    }
    return false;
}
```

- **Временная сложность**: $O(n)$ на запрос
- **Пространственная сложность**: $O(1)$

6.3 Оптимальное решение: бинарный поиск

Основная идея алгоритма заключается в **использовании свойства отсортированности** массива для исключения половины элементов на каждом шаге. Если искомый элемент x находится в интервале (l, r), то:

- При $a\left[\dfrac{l+r}{2}\right] > x$ продолжаем поиск в левой половине, ведь всё, что находится в правой, гарантированно будет больше x: двигаем правую границу $r = \dfrac{l+r}{2}$
- При $a\left[\dfrac{l+r}{2}\right] < x$ продолжаем поиск в правой половине, ведь всё, что находится в левой, гарантированно будет меньше x: двигаем левую границу $l = \dfrac{l+r}{2}$

Каждая итерация цикла уменьшает диапазон поиска вдвое, поэтому максимальное число итераций равно $\lceil \log_2 n \rceil$.

Листинг 6.2: Бинарный поиск

```
bool Find(vector<int>& a, int x) {
    int l = -1;
    int r = a.size();

    while (l + 1 != r) {
        int mid = (l + r) / 2;

        if (a[mid] > x) {
            r = mid;
        }
        else if (a[mid] < x) {
            l = mid;
        }
        else {
            return true;
        }
    }

    return false;
}
```

- **Временная сложность**: $O(\log n)$ на запрос
- **Пространственная сложность**: $O(1)$

Глава 7

Двоичный поиск по ответу

Двоичный поиск по ответу применяется, когда нужно найти максимальное или минимальное значение параметра N при условии:

Когда ищем минимальное значение:

1. Если решение работает для N, то оно гарантированно работает и для всех **больших** значений $(N+1, N+2, \dots)$
2. Если решение **не** работает для N, то оно гарантированно не работает и для всех **меньших** значений $(N-1, N-2, \dots)$

Когда ищем максимальное значение:

1. Если решение работает для N, то оно гарантированно работает и для всех **меньших** значений $(N-1, N-2, \dots)$
2. Если решение **не** работает для N, то оно гарантированно не работает и для всех **больших** значений $(N+1, N+2, \dots)$

7.1 Алгоритм

1. Определяем границы поиска:
   - $left$ - минимально возможное значение - 1
   - $right$ - максимально возможное значение + 1
2. Пока $left + 1 \neq right$:
   - Вычисляем среднее значение: $mid = \left\lfloor \frac{left+right}{2} \right\rfloor$
   - Если условие для mid выполняется:
     - Для минимума: $right = mid$
     - Для максимума: $left = mid$
   - Если условие для mid не выполняется:
     - Для минимума: $left = mid$
     - Для максимума: $right = mid$
3. Возвращаем результат:
   - Для минимума: $left$
   - Для максимума: $right$

Примечание: Основные понятия

- **Предикат** - это функция, которая принимает некоторое значение и возвращает `true` или `false`. В бинарном поиске предикат определяет, удовлетворяет ли текущее значение искомому условию.
- **Диапазон поиска** - это интервал значений $[left, right]$, в котором гарантированно содержится искомый ответ. На каждой итерации диапазон сужается вдвое.

7.2 Задачи

7.2.1 Задача 1: Монетки

Условие: Даны n стопок монет, в i-й стопке a_i монет. За один ход можно взять k монет из любой стопки. Найти минимальное k, чтобы убрать все монеты за $\leq m$ ходов.

Решение:

- Предикат: $P(k) = \sum_{i=1}^{n} \lceil a_i/k \rceil \leq m$.
- Диапазон поиска: $k \in [1, \max(a_i)]$.

7.2.2 Задача 2: Оптимальное размещение станций

Условие: Даны n точек на прямой. Разместить k станций так, чтобы максимальное расстояние от точки до ближайшей станции было минимально.

Решение:

- Предикат: $P(d) =$ можно покрыть все точки станциями с радиусом d.
- Диапазон поиска: $d \in [0, \max(\text{координаты})]$.

7.2.3 Задача 3: Максимальная медиана подмассива

Условие: Дан массив a длины n. Найти подмассив длины $\geq k$ с максимальной медианой.

Решение:

- Предикат: $P(x) =$ существует подмассив длины $\geq k$ с медианой $\geq x$.
- Диапазон поиска: $x \in [\min(a_i), \max(a_i)]$.

7.2.4 Задача 4: Вычисление корня

Условие: Для заданного целого числа $A > 0$ вычислите $\sqrt{A}$ с точностью до 10^{-6}.

Решение:

- Предикат: $P(x) = x^2 <= A$.
- Диапазон поиска: $a \in [0, A]$.

Примечание: чтобы решение работало, нужно сделать порядка 100 итераций бинарного поиска - это обеспечит нужную точность.

Глава 8

Тернарный поиск

Тернарный поиск — это алгоритм поиска экстремума минимума (или максимума) унимодальной функции на отрезке. Функция называется унимодальной, если на заданном отрезке она сначала строго убывает, затем строго возрастает (или наоборот), имея ровно одну точку экстремума. Алгоритм работает путем последовательного деления отрезка поиска на три части и исключения одной из них на каждом шаге.

8.1 Алгоритм

- Для поиска **минимума** (случай для поиска максимума рассматривается аналогично) унимодальной функции f: f сначала убывает, а затем возрастает, на каждом шаге выбираем две точки m_1 и m_2 ($m_1 < m_2$) внутри текущего отрезка $[l, r]$ и сравниваем значения $f(m_1)$ и $f(m_2)$:
  - Если $f(m_1) < f(m_2) \to$ новый отрезок $[l, m_2]$, так как минимум не может находиться правее m_2, ведь:
    - Если обе точки m_1 и m_2 слева от экстремума, то функция на $[m_1, m_2]$ убывает по определению, и должно выполняться $f(m_1) > f(m_2)$ (что противоречит условию).
    - Если m_1 слева, а m_2 справа от экстремума, то минимум между ними. (см. рис. 1)
    - Если обе точки m_1 и m_2 справа от экстремума, то функция возрастает на $[m_1, m_2]$, и условие $f(m_1) < f(m_2)$ означает, что минимум левее m_2. (см. рис. 2)
  - Если $f(m_1) \geq f(m_2) \to$ новый отрезок $[m_1, r]$, так как минимум не может находиться левее m_1, аналогично рассуждениям первого случая.

На каждой итерации длина отрезка уменьшается в $\frac{2}{3}$ раза. После n итераций длина отрезка будет:

$$(r - l) \cdot \left(\frac{2}{3}\right)^n$$

Рекомендуется продолжать алгоритм, пока $r - l > 5$, а значения в оставшихся точках получить явно (подставляя каждое $x \in [l, r]$ в f).

8.2 Сравнение с бинарным поиском

Критерий	Тернарный поиск	Бинарный поиск
Тип задачи	Унимодальные функции	Монотонные функции
Скорость сходимости	$\log_{3/2} n$	$\log_2 n$
Память	$O(1)$	$O(1)$

Листинг 8.1: Шаблон для поиска минимума унимодальной функции f

```
void Solve() {
    int l = L; // the minimum possible value (without -1 !!!)
    int r = R; // the maximum possible value (without +1 !!!)

    while (r - l > 5) {
        int m1 = l + (r - l) / 3;
        int m2 = r - (r - l) / 3;

        if (F(m1) < F(m2)) {
            r = m2;
        }
        else {
            l = m1;
        }
    }

    int ans = 1e18;

    for (int i = l; i <= r; i++) { // if l or r didn't move, we will find their value
        ↪ to calculate total answer
        ans = min(ans, F(i));
    }
}
```

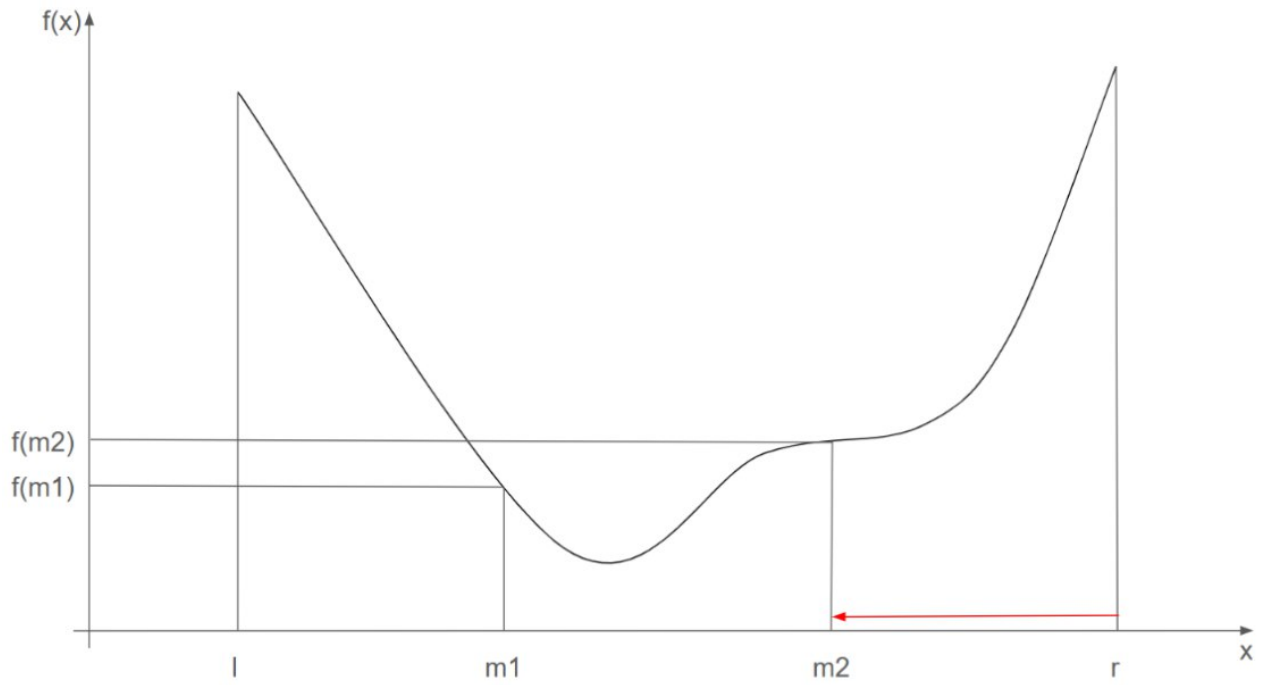


Рис. 8.1: ext between

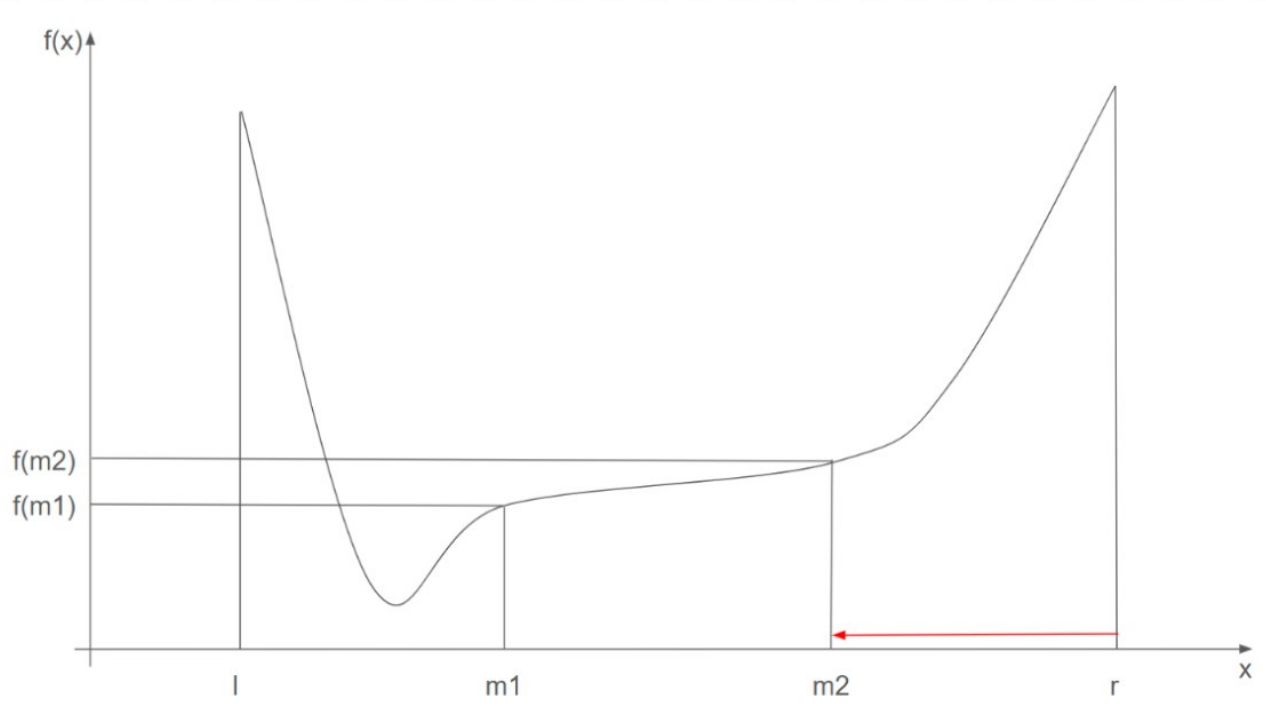


Рис. 8.2: ext left

8.3 Задачи

8.3.1 Задача 1: Последняя битва

Условие: Вот уже как сто лет идет война между "разноцветными эльфами"и "хитрыми гномами". Вы - генерал армии эльфов. Конечно, это большая ответственность - нужно как можно лучше распределить силы своих воинов. Для этого вы можете создать несколько отрядов.

Когда два эльфа одного цвета оказываются в разных отрядах, к силе армии прибавляется некая константа b. При этом разделение на большое число отрядов понижает сплоченность войска, поэтому каждый новый отряд (кроме первого) уменьшает силу армии на некую константу x. Найдите наибольшую возможную силу армии, которую можно получить.

Решение: Используем тернарный поиск по количеству отрядов. Поиск наибольшей силы армии при заданном количестве отрядов можно искать, жадно распределяя воинов (чтобы эльфов одного цвета было "примерно поровну"во всех группах).