Сортировка событий (Sweep Line)

Denis Bakin

План лекции

1. Лямбда-функции и кастомная сортировка в C++
2. Метод сканирующей прямой — идея
3. Задача: посетители сайта
4. Задача: кассы на вокзале
5. Задача: объединение отрезков
6. Задача: точки и отрезки
7. Советы по реализации

Лямбда-функции в C++

Лямбда — анонимная функция, определяемая “на месте”

Синтаксис: `[captures](params) { body }`

```
auto func = [](int x) { return 2 * x; };

std::cout << func(15);  // 30
std::cout << func(-3);  // -6
```

- `[]` — захват переменных из внешней области
- `(params)` — параметры функции
- `{ body }` — тело функции

Лямбды как компараторы

```cpp
std::vector<int> data = {4, 1, 5, 3, 2};

// По возрастанию (по умолчанию)
std::sort(data.begin(), data.end());
// [1, 2, 3, 4, 5]

// По убыванию - с лямбдой
std::sort(data.begin(), data.end(), [](int a, int b) {
    return a > b;  // true, если a должен быть раньше b
});
// [5, 4, 3, 2, 1]
```

Компаратор возвращает `true`, если первый аргумент должен идти **раньше** второго

Лексикографическое сравнение

`std::pair` и `std::tuple` сравниваются **лексикографически**:

- сначала по первому элементу
- при равенстве — по второму
- и так далее...

$$\{1, 2\} < \{1, 3\} < \{2, 2\} < \{2, 3\}$$

Аналогия: сортировка слов в словаре

Сложная сортировка пар

Задача: отсортировать по второму элементу, затем по первому

```
std::vector<std::pair<int, int>> data = {{1, 2}, {5, 1}, {3, 1}, {1, 3}};

std::sort(data.begin(), data.end(), [](const auto& a, const auto& b) {
    if (a.second != b.second) {
        return a.second < b.second;  // сначала по второму
    }
    return a.first < b.first;        // затем по первому
});
// [{3, 1}, {5, 1}, {1, 2}, {1, 3}]
```

Сортировка: возрастание + убывание

Задача: по возрастанию первого, по убыванию второго

```
std::sort(data.begin(), data.end(), [](const auto& a, const auto& b) {
    if (a.first != b.first) {
        return a.first < b.first;   // возрастание первого
    }
    return a.second > b.second;     // убывание второго
});
// [{1, 3}, {1, 2}, {3, 1}, {5, 1}]
```

Трюк: для убывания меняем знак сравнения (> вместо <)

Метод сканирующей прямой

Sweep Line (сканирующая прямая) — алгоритмическая техника:

- преобразуем данные в **события**
- **сортируем** события по координате/времени
- **проходим** слева направо, обновляя состояние

Метод сканирующей прямой

Sweep Line (сканирующая прямая) — алгоритмическая техника:

- преобразуем данные в **события**
- **сортируем** события по координате/времени
- **проходим** слева направо, обновляя состояние

Применения:

- подсчёт пересечений отрезков
- нахождение максимума/минимума
- вычисление площадей и длин
- геометрические задачи

Задача: посетители сайта

Для каждого пользователя известно время входа l_i и выхода r_i.

Найти: максимальное количество одновременных посетителей

Задача: посетители сайта

Для каждого пользователя известно время входа l_i и выхода r_i.

Найти: максимальное количество одновременных посетителей

Формально: найти точку m, принадлежащую максимальному числу отрезков

$$m_{max} = \max_{m} \sum_{i=1}^{n} \mathbb{1}\{m \in [l_i, r_i]\}$$

Наивный подход

Идея: перебрать все моменты времени, для каждого посчитать посетителей

```
for m = min_l to max_r:
    cnt = 0
    for each segment [l, r]:
        if l <= m <= r:
            cnt++
    max_cnt = max(max_cnt, cnt)
```

Наивный подход

Идея: перебрать все моменты времени, для каждого посчитать посетителей

```
for m = min_l to max_r:
    cnt = 0
    for each segment [l, r]:
        if l <= m <= r:
            cnt++
    max_cnt = max(max_cnt, cnt)
```

Сложность: $O\left((\max r_i - \min l_i) \cdot n\right)$

Проблемы:

- зависит от диапазона времени, а не от количества данных
- при смене единиц измерения (минуты $\rightarrow$ секунды) — в 60 раз медленнее

Ключевое наблюдение

Количество посетителей меняется только в **моменты событий**!

Ключевое наблюдение

Количество посетителей меняется только в **моменты событий**!

Между событиями — значение константно

$$\underbrace{0}_{\text{до } t_1} \xrightarrow{+1} \underbrace{1}_{[t_1,t_2)} \xrightarrow{-1} \underbrace{0}_{[t_2,t_3)} \xrightarrow{+1} \underbrace{1}_{[t_3,\infty)}$$

События и их типы

Каждый отрезок $[l_i, r_i]$ порождает **2 события**:

Событие	Кортеж	Действие
Вход пользователя	$(l_i, 0)$	`cnt++`
Выход пользователя	$(r_i, 1)$	`cnt--`

События и их типы

Каждый отрезок $[l_i, r_i]$ порождает **2 события**:

Событие	Кортеж	Действие
Вход пользователя	$(l_i, 0)$	`cnt++`
Выход пользователя	$(r_i, 1)$	`cnt--`

Зачем второй элемент?

При одинаковом времени: сначала входы, потом выходы

$(t, 0) < (t, 1)$ — лексикографически

Алгоритм

1. Создать список событий из всех отрезков
2. **Отсортировать** события лексикографически
3. Пройти по событиям, обновляя счётчик:
   - вход: `cnt++`, обновить максимум
   - выход: `cnt--`

Алгоритм

1. Создать список событий из всех отрезков
2. **Отсортировать** события лексикографически
3. Пройти по событиям, обновляя счётчик:
   - вход: `cnt++`, обновить максимум
   - выход: `cnt--`

Сложность:

- n отрезков $\to 2n$ событий
- сортировка: $O(n \log n)$
- обход: $O(n)$
- **Итого:** $O(n \log n)$

Пример работы алгоритма

Отрезки: $[1, 5]$, $[2, 4]$, $[3, 6]$

Время	Тип	Событие	cnt	max
1	0	вход	1	1
2	0	вход	2	2
3	0	вход	3	**3**
4	1	выход	2	3
5	1	выход	1	3
6	1	выход	0	3

Ответ: 3 посетителя одновременно

Структура события в C++

```cpp
enum EventType {
    kOpen = 0,   // приоритет выше (обрабатывается раньше)
    kClose = 1
};

struct Event {
    int time;
    EventType type;

    bool operator<(const Event& other) const {
        if (time != other.time) {
            return time < other.time;
        }
        return type < other.type;  // kOpen < kClose
    }
};
```

Реализация: создание событий

```
int n;
std::cin >> n;

std::vector<Event> events;
events.reserve(2 * n);  // оптимизация памяти

for (int i = 0; i < n; ++i) {
    int start, stop;
    std::cin >> start >> stop;
    events.push_back({start, kOpen});
    events.push_back({stop, kClose});
}

std::sort(events.begin(), events.end());
```

Реализация: обработка событий

```cpp
int cnt = 0;
int max_cnt = 0;

for (const auto& event : events) {
    if (event.type == kOpen) {
        ++cnt;
        max_cnt = std::max(max_cnt, cnt);
    } else {
        --cnt;
    }
}

std::cout << max_cnt << "\n";
```

Три типа событий

Задача: выводить количество посетителей по запросу

События с приоритетами:

Событие	Приоритет	Действие
Вход	0	`cnt++`
Запрос	1	вывести `cnt`
Выход	2	`cnt--`

Три типа событий

Задача: выводить количество посетителей по запросу

События с приоритетами:

Событие	Приоритет	Действие
Вход	0	`cnt++`
Запрос	1	вывести `cnt`
Выход	2	`cnt--`

При одном времени: сначала входы $\rightarrow$ запрос $\rightarrow$ выходы

Три типа событий: структура

```
enum EventType {
    kOpenEvent = 0,
    kRequestEvent = 1,
    kCloseEvent = 2
};

struct Event {
    int time;
    EventType type;

    bool operator<(const Event& other) const {
        if (time != other.time) return time < other.time;
        return type < other.type;
    }
};
```

Три типа событий: обработка

```
for (const auto& event : events) {
    if (event.type == kOpenEvent) {
        ++cnt;
    } else if (event.type == kRequestEvent) {
        std::cout << cnt << "\n";
    } else if (event.type == kCloseEvent) {
        --cnt;
    }
}
```

Задача: кассы на вокзале

n касс с графиком работы $[t_{open}, t_{close}]$

Найти: общее время, когда работают **все** кассы

Задача: кассы на вокзале

n касс с графиком работы $[t_{open}, t_{close}]$

Найти: общее время, когда работают **все** кассы

Особые случаи:

Условие	Интерпретация	События
$t_{open} < t_{close}$	обычный режим	2
$t_{open} > t_{close}$	через полночь	4
$t_{open} = t_{close}$	круглосуточно	0

Кассы: алгоритм

1. Обработать особые случаи (круглосуточные)
2. Сгенерировать события открытия/закрытия
3. Отсортировать
4. Пройти, накапливая время когда `cnt == n`

```
if (cnt == n) {
    total_time += event.time - last_time;
}
```

Задача: объединение отрезков

Дано n отрезков $[l_i, r_i]$

Найти: суммарную длину их объединения

Объединение: алгоритм

Идея: отслеживаем “глубину покрытия”

- глубина $> 0 \rightarrow$ накапливаем длину
- глубина $= 0 \rightarrow$ пропускаем

Событие	Действие
Открытие	если `depth == 0`: запомнить `start`; `depth++`
Закрытие	`depth--`; если `depth == 0`: добавить `coord - start`

Объединение: пример

Координата	Тип	depth	Добавлено	Сумма
1	open	1	—	0
2	open	2	—	0
4	close	1	—	0
6	close	0	6 - 1 = 5	5
8	open	1	—	5
10	close	0	10 - 8 = 2	**7**

Задача: точки и отрезки

Дано n отрезков и m точек-запросов

Для каждой точки: сколько отрезков её содержат?

Задача: точки и отрезки

Дано n отрезков и m точек-запросов

Для каждой точки: сколько отрезков её содержат?

Идея: объединить точки и концы отрезков в один поток

Событие	Приоритет
Начало отрезка	0
Точка-запрос	1
Конец отрезка	2

Точки и отрезки: пример

Отрезки: $[1, 5]$, $[2, 4]$ | Точки: 3, 6

Координата	Тип	depth	Ответ
1	open	1	
2	open	2	
3	query	2	**2**
4	close	1	
5	close	0	
6	query	0	**0**

Точки и отрезки: структура события

```
struct Event {
    int coord;
    EventType type;
    int query_id;  // только для запросов

    bool operator<(const Event& other) const {
        if (coord != other.coord) return coord < other.coord;
        return type < other.type;
    }
};

// Сохраняем ответы в массив
std::vector<int> answers(m);
// ...
case kQuery:
    answers[event.query_id] = depth;
```

Приоритеты: почему важны?

Вопрос: точка на границе отрезка — внутри или снаружи?

Приоритеты	Точка на l	Точка на r
open $<$ query $<$ close	внутри	внутри
open $<$ close $<$ query	внутри	снаружи
query $<$ open $<$ close	снаружи	снаружи

Приоритеты: почему важны?

Вопрос: точка на границе отрезка — внутри или снаружи?

Приоритеты	Точка на l	Точка на r
open $<$ query $<$ close	внутри	внутри
open $<$ close $<$ query	внутри	снаружи
query $<$ open $<$ close	снаружи	снаружи

Вывод: приоритеты определяют семантику границ!

Общий шаблон решения

```
// 1. Определить типы событий и приоритеты
enum EventType { kType1 = 0, kType2 = 1, ... };

// 2. Структура события
struct Event { ... };

// 3. Создать события
std::vector<Event> events;
for (...) { events.push_back(...); }

// 4. Отсортировать
std::sort(events.begin(), events.end());

// 5. Обработать
for (const auto& e : events) {
    switch (e.type) { ... }
}
```

Советы по реализации

1. **Чётко определите приоритеты** — что обрабатывается первым при равных координатах?
2. **Используйте `enum`** для типов событий — читаемость и защита от ошибок
3. **Перегружайте `operator<`** в структуре события — стандартная сортировка “из коробки”
4. **`reserve()` для вектора** — если знаете размер заранее
5. **Отладка:** выводите отсортированные события и состояние счётчика

Итоги

- **Сортировка событий** — мощная техника оптимизации
- Ключевая идея: обрабатывать только **моменты изменений**
- Сложность: обычно $O(n \log n)$ вместо наивного $O(n \cdot T)$