Дерево Фенвика: одномерное, двумерное

Антонов Кирилл
Б01-411

Курс «Алгоритмы и структуры данных»

Мотивация: Real-time аналитика

Проблема

- Массив продаж[регионы][дни] — 100 регионов × 365 дней
- Типовой запрос: $\sum_{\text{регион}=r_1}^{r_2} \sum_{\text{день}=d_1}^{d_2}$ продажи[регион][день]
  Требуется: мгновенные отчеты + real-time обновления

Базовые подходы

Метод	Запрос	Обновление	Примечание
Без предподсчёта	O(N²)	O(1)	Полный перебор
С предподсчётом	O(1)	O(N²)	Префиксные суммы
Дерево Фенвика	**O(log N)**	**O(log N)**	Оптимальное решение

Вывод

Для реальных данных (36 500 элементов) нужна структура, эффективная и для запросов, и для обновлений → **дерево Фенвика**

Математическая формулировка

Одномерный случай

Дан массив $A[0..n-1]$

- update$(i, \Delta) : A[i] \leftarrow A[i] + \Delta$
- prefix_sum$(k) : \sum_{j=0}^{k} A[j]$
- range_sum(l, r) : prefix_sum(r) − prefix_sum$(l-1)$

Двумерный случай

Дан массив $A[0..n-1][0..m-1]$

- update$(x, y, \Delta) : A[x][y] \leftarrow A[x][y] + \Delta$
- rectangle_sum(x_1, y_1, x_2, y_2) : $\sum_{i=x_1}^{x_2} \sum_{j=y_1}^{y_2} A[i][j]$

Принцип работы 1D

Ключевая идея

- Используем двоичное представление индексов
- fenw[i] хранит сумму $A[i - 2^k + 1 \dots i]$, т.ч. $2^k = i\& - i$

Распределение блоков для n=8

Индекс	Двоично	2^k	Диапазон
1	001	1	[1,1]
2	010	2	[1,2]
3	011	1	[3,3]
4	100	4	[1,4]
5	101	1	[5,5]
6	110	2	[5,6]
7	111	1	[7,7]
8	1000	8	[1,8]

Операции 1D: реализация

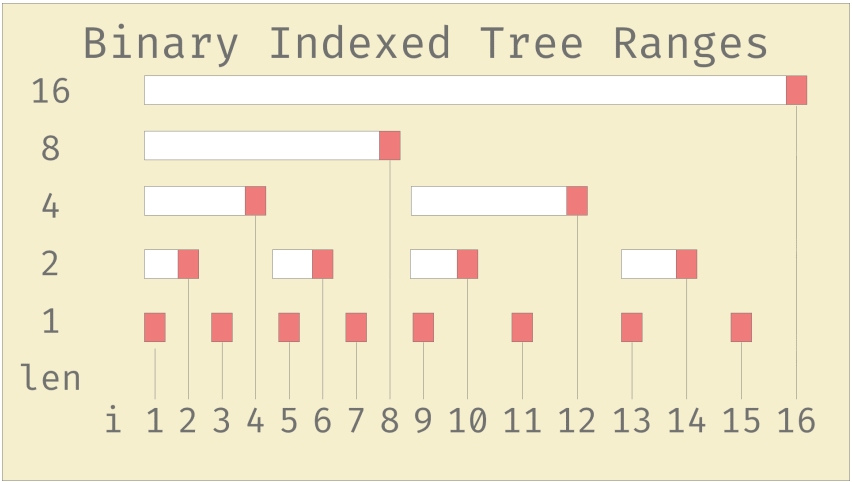
```
int lsb(int x){
    return x & -x;
}

int prefix_sum(int i, int fenw[]){
    int sum = 0;
    for (; i > 0; i -= lsb(i))
        sum += fenw[i];
    return sum;
}

void update(int i, int delta, int fenw[], int n){
    for (; i <= n; i += lsb(i))
        fenw[i] += delta;
}

int range_sum(int l, int r, int fenw[]){
    return prefix_sum(r, fenw) - prefix_sum(l-1, fenw);
}
```

Визуализация: запрос суммы до 16



Запрос sum(15): блоки 15 + 14 + 12 + 8 = [1..15]

Обобщение на 2D случай

Принцип «дерево деревьев»

- fenw$[i][j]$ хранит сумму прямоугольника $[i - 2^k + 1..i] \times [j - 2^l + 1..j]$
- Операции: два вложенных цикла

Сложность операций

Размерность	Запрос	Обновление
1D	O(log n)	O(log n)
2D	O(log n $\times$ log m)	O(log n $\times$ log m)
3D	O(log n $\times$ log m $\times$ log k)	O(log n $\times$ log m $\times$ log k)

Реализация 2D Фенвика

```
void update_2d(int x, int y, int delta,
  vector<vector<int>> &fenw, int n, int m){
    for (int i = x; i <= n; i += lsb(i))
        for (int j = y; j <= m; j += lsb(j))
            fenw[i][j] += delta;
}

int prefix_sum_2d(int x, int y,
  vector<vector<int>> &fenw){
    int sum = 0;
    for (int i = x; i > 0; i -= lsb(i))
        for (int j = y; j > 0; j -= lsb(j))
            sum += fenw[i][j];
    return sum;
}

int rectangle_sum(int x1, int y1, int x2, int y2,
  vector<vector<int>> &fenw){
    return prefix_sum_2d(x2, y2, fenw)
         - prefix_sum_2d(x1-1, y2, fenw)
         - prefix_sum_2d(x2, y1-1, fenw)
         + prefix_sum_2d(x1-1, y1-1, fenw);
}
```

Бенчмарки: Фенвик vs Наивные подходы

Тестовая среда

- Размер массива: 100,000 элементов
- Количество операций: 50,000
- Процессор: Intel i5, 16GB RAM
- ОС: Manjaro Linux, ядро 6.12.48-1-MANJARO
- Компилятор: g++ (GCC) 15.2.1 20250813

Результаты (миллисекунды)

Метод	100% запросы	100% обновления	50/50 смешанно
Фенвик	**9 мс**	**3 мс**	**4 мс**
Наивный (запрос)	5,6 с	1 мс	2,7 с
Наивный (префикс)	1 мс	22,7 с	11 с

Вывод

В смешанной нагрузке Фенвик **в 700-2800 раз** быстрее!

Сложность по времени: 1D случай

Операции

- `update(i, delta)`: $O(\log n)$
- `prefix_sum(i)`: $O(\log n)$
- `range_sum(l, r)`: $O(\log n)$

Доказательство для `prefix_sum`

- На каждой итерации: $i = i - \text{LSB}(i)$
- LSB(i) ≥ 1, поэтому итераций $\leq$ количества бит в n
- Количество бит $= \lceil \log_2 n \rceil = O(\log n)$

Доказательство для `update`

- Аналогично: $i = i + \text{LSB}(i)$
- Максимальное значение LSB(i) $= 2^{\lfloor \log_2 n \rfloor}$
- Итераций $\leq \log_2 n$

Сложность по времени: 2D случай

Операции

- `update_2d(x, y, delta)`: $O(\log n \times \log m)$
- `prefix_sum_2d(x, y)`: $O(\log n \times \log m)$
- `rectangle_sum(x1, y1, x2, y2)`: $O(\log n \times \log m)$

Доказательство

- Внешний цикл: $O(\log n)$ итераций
- Внутренний цикл: $O(\log m)$ итераций
- Итого: $O(\log n) \times O(\log m) = O(\log n \times \log m)$

Пример

Для $n = m = 1024$: $\log_2 1024 = 10$,
$10 \times 10 = 100$ операций vs $1024^2 = 1,048,576$

Сложность по памяти

1D случай

- Требуется массив размера n
- Память: $O(n)$
- **Доказательство**: Храним ровно n элементов

2D случай

- Требуется матрица размера $n \times m$
- Память: $O(n \times m)$
- **Доказательство**: Храним $n \times m$ элементов

Сравнение с деревом отрезков

- **Фенвик**: $O(n)$ vs **Дерево отрезков**: $O(2n)$
- В 2D: $O(nm)$ vs $O(4nm)$
- Выигрыш в памяти: в 2-4 раза!

Амортизационный анализ

Константы в O-нотации

- `update`: $\approx 2 \log n$ операций
- `prefix_sum`: $\approx 2 \log n$ операций
- На практике: очень маленькие константы

Сравнение операций

Операция	Фенвик	Дерево отрезков	Выигрыш
Сложение	1 такт	3-5 тактов	3-5x
Обращение к памяти	$\log n$	$2 \log n$	2x

Практическая эффективность

- Лучшая локальность данных
- Меньше branching (условных переходов)
- Быстрее на реальных процессорах

Доказательство корректности

Инвариант

Для любого $i > 0$, fenw$[i]$ содержит сумму элементов $A[i - 2^k + 1 \dots i]$, где $2^k = \text{LSB}(i)$

Доказательство по индукции

- **База**: Для $n = 1$: fenw$[1] = A[1]$ ✓
- **Переход**: Предположим верно для $n - 1$ элементов
- update(n) обновляет все fenw$[i]$ где $i = n + 2^m$
- Каждый такой i покрывает корректный диапазон

Корректность `prefix_sum`

- Любое число n можно представить как сумму степеней 2
- $n = 2^{k_1} + 2^{k_2} + \dots + 2^{k_m}$
- Алгоритм собирает именно эти блоки

Оптимальность

Нижняя оценка

- Любая структура для динамического RSQ
- Должна тратить $\Omega(\log n)$ на операцию
- Фенвик достигает этой границы!

Теорема

Не существует структуры данных, которая поддерживает `update` и `prefix_sum` быстрее чем $O(\log n)$ в модели сравнений.

Почему Фенвик оптимален?

- Достигает теоретического минимума
- Константы близки к оптимальным
- Использует минимально возможную память