Contents

1 О структурах данных

Каждая структура данных характеризуется следующими параметрами:

- Запросы, на которые она умеет отвечать
- Время построения
- Время ответа на запрос
- Объём занимаемой памяти

Чтобы коротко записывать параметры 2-4 введём обозначения
$\langle O(build), O(querry), O(memory)\rangle$. Иногда будем опускать количество потребляемой памяти.

2 Частичные суммы

Пусть нам необходима структура, которая умеет отвечать ровно на один запрос:

- Найти сумму на отрезке $[l;r]$ в массиве a

Тогда на помощь придут частичные (или же префиксные) суммы:

Пусть $p_k = \sum\limits_{i=1}^{k} a_i$. То есть p_k есть сумма на префиксе длины k. Сам массив p считается за один проход: $p_k = p_{k-1} + a_i$. Далее не сложно заметить, что вычисление суммы на отрезке $[l;r]$ можно сделать через массив p:

$$\sum_{i=l}^{r} a_i = \left[\textit{Добавим и вычтем } \sum_{i=1}^{l-1} a_i\right] = \sum_{i=1}^{r} a_i - \sum_{i=1}^{l-1} a_i = p_r - p_{l-1}$$

Значит, получили алгоритм по времени и памяти: $\langle O(n), O(1), O(n)\rangle$. Последнее $O(n)$ по памяти не обязательно и можно строить префиксные суммы прямо в массиве a.

3 Дерево отрезков

3.1 Объяснение

О нём очень хорошо написано на алгоритмике, дублировать хороший текст не вижу смысла.

3.2 Реализация

Единственный минус - их реализация просто ужасна, ниже я опишу свою для суммы на отрезке, но для начала немного ликбеза по C++:

В функции можно передавать параметры по умолчанию, то есть если при вызове не указывать этот параметр, то он будет равен тому, что мы указываем в качестве аргумента по умолчанию. Обратите внимание, что можно задавать параметры по умолчанию только для некоторого суффикса переменных при объявлении функции [1]. Синтаксис:

```
int dfs(int v, int p = -1){
  //code
}
```

Так, если мы вызовем $dfs(1)$, то он запустится с параметрами $dfs(1, -1)$, а ещё мы можем вызвать dfs сразу с двумя параметрами: $dfs(1, 1)$.

В C++ есть так называемые макросы. В промышленном программировании вас за них убьют, но в олимпиадном их можно использовать.

```
#define sqr(a) ((a) * (a))
```

Теперь если вы напишете $sqr(2)$, то вам вернется 4. Макрос - это что-то типа функции, только она "подменяет" код на этапе препомпиляции и вместо явной функции, которая возведёт a в квадрат в месте где вы это написали совершит замену с $sqr(2)$ на $((2) * (2))$. В работе с макросами нужно не збывать **про порядок действий**, так как в момент такой подмены может произойти то, что вы совсем не ожидает. Например, если бы скобок не стояло, то при вызове $sqr(2+2)$ вы бы получили $2+2*2+2 = 6$ вместо 16, поэтому в самом макросе следует писать скобки везде, где только можно.

Собственно код ДО:

```
int n; // initialize !!
int a[N]; // initialize !!

int t[4 * N];
#define left (2 * v + 1)
#define right (2 * v + 2)
#define mid ((r + l) >> 1)

void recalc(int v) {
  t[v] = t[left] + t[right];
}

void build(int v = 0, int l = 0, int r = n) {
  if (r - l == 1)
    t[v] = a[l];
  else {
    int m = mid;
    build(left, l, m)
```

[1] Подумаете, почему?

```
    build(right, m, r);
    recalc(v);
  }
}

void update(int pos, int x, int v = 0, int l = 0, int r = n) {
  if (r - l == 1)
    t[v] = x;
  else {
    int m = mid;
    if (pos < m)
      update(pos, x, left, l, m);
    else
      update(pos, x, right, m, r);
    recalc(v);
  }
}


int get(int ql, int qr, int v = 0, int l = 0, int r = n) {
  if (qr <= l || r <= ql) // [ql; qr) and [l; r) doesn't intersect
    return 0;
  else if (ql <= l && r <= qr) // [l; r) into [ql; qr)
    return t[v];
  else {
    int m = mid;
    return get(ql, qr, left, l, m) + get(ql, qr, right, m, r);
}
```

Небольшое пояснение:

- В данной реализации я придерживаюсь подхода полуинтервалов, а не отрезков — это помогает избежать большого количества ± 1 в коде
- left и right - магия define'ов — как и макросы, они подменяются на этапе компиляции на выражения $2 * v + 1$ и $2 * v + 2$ соответственно
- Почему сыновья именно $2v + 1$ и $2v + 2$? Потому[1].
- Выполнять запросы нужно так:

```
//initialize n, a;
build(); // segment tree building
get(l, r); // count sum on half-interval [l; r)
update(pos, x); // set a[pos] = x
```

По коду очевидно, что затраты по времени и памяти следующие: $\langle O(n), O(\log n), O(n) \rangle$

3.3 Какие функции можно вычислять с помощью ДО?

Главный критерий — это умение "разделять" запрос на 2 части, а потом, зная результат этих двух частей, быстро их "склеивать", то есть вычислять результат всего отрезка.

[1] Нарисуйте дерево с такими номерами и всё поймёте

Тогда "хорошими функциями" можно называть сумму на отрезке, минимум, НОД, произведение по произвольному модулю и многие другие. "Плохими" будет, например, количество различных чисел на отрезке, так как зная количество различных чисел в одном множестве и в другом нельзя сказать сколько различных в иъ объединении.

Задача 1. Дана скобочная последовательность из открывающих и закрывающих скобок одного типа. Необходимо уметь выполнять два запроса:

- Изменить тип скобки на позиции i
- Проверить, верно ли, что подотрезок $[l; r]$ является правильной скобочной последовательностью

Задача 2. Дан массив. Нужно уметь искать k-ый элемент на отрезке $[l; r]$, если бы все элементы на нём были бы отсортированы. Это называется k-й порядковой статистикой на отрезке.

4 Sparse table

4.1 Задача

Задача 3. Минимум на отрезке.

Подходов для этой задачи очень много. Одним из таких подходов является более известное Дерево Отрезков, которое занимает $O(n)$ времени на построение и $O(\log n)$ на запрос.

Мы рассмотрим статичную структуру, которая умеет строиться за $O(n \log n)$ [1] и отвечать на запросы за $O(1)$.

Для начала заметим, что функция минимума, ровно как и функция максимума, НОДа, побитового или, обладает свойством идемпотентности, что на русско-крестьянском означает следующее:

$$f(a, a) = a$$

Это свойство будет играть ключевую роль в нашем алгоритме. Мы получим структуру, работающую для любой идемпотентной операции [2], однако далеко не все операции такие: например, сумма, исключающее или не идемпотентны.

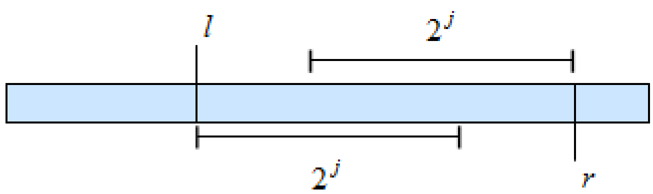
4.2 Алгоритм

Алгоритм будет крайне простым и состоять из двух шагов.

1. **Предпросчёт.** Вычислим минимум на всех отрезках вида $[l; l + 2^k]$. Вычислять можно эффективно, используя уже посчитанные минимумы:

```
for (int k = 0; k < LOGN; ++k) {
  for (int i = 0; i + (1 << k) < N; ++i) {
    if (k == 0)
      mn[i][k] = a[i];
    else
      mn[i][k] = min(mn[i][k - 1], mn[i + (1 << (k - 1))][k - 1]);
  }
}
```

2. **Ответ на запрос.** Каждый запрос минимума $[l; r]$ можно представить в виде объединения двух отрезков вида: $[l; l + 2^j] \cup [r - 2^j; r]$.



Осталось аккуратно выбрать j: выберем такое минимальное $j : 2^{j+1} > r - l + 1$. Для этого я рекомендую предпосчитать массив двоичных логарифмов всех чисел от 1 до n по очень простой рекурсивной формуле:

$$deg2_1 = 0, deg2_i = deg2_{i/2} + 1$$

Теперь минимум на отрезке $[l; r]$ считается как

[1] На самом деле эта структура более продвинутая и её модификации могут строиться за $O(n \log \log n)$ и быстрее, но сегодня это не наша тема

[2] Опять же, есть модификация Disjoint Sparse table, которая позволяют избежать этого ограничения

```
int get_min(int l, int r) {
  int sz = deg2[r - l + 1];
  return min(mn[l][sz], mn[r - (1 << sz)][sz]);
}
```

Очевидно, что алгоритм занимает $O(n \log n)$ построения и памяти и $O(1)$ на запрос.