БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СБОРНИК ЗАДАЧ

ПО ТЕОРИИ АЛГОРИТМОВ.

Структуры данных.
Часть. Струтуры данных для решения задач на интервалах.

МИНСК
2020

УДК 510.51(075.8)
ББК 22.12я73-1
С23

А в т о р ы :
С. А. Соболь, К. Ю. Вильчевский, В. М. Котов,
Е. П. Соболевская

Р е ц е н з е н т ы :
кафедра информатики и методики преподавания информатики
физико-математического факультета Белорусского государственного
педагогического университета им. М. Танка
(заведующий кафедрой, кандидат педагогических наук,
доцент *С. В. Вабищевич*);
профессор кафедры информационных технологий в культуре
Белорусского государственного университета культуры и искусства,
кандидат физико-математических наук, доцент *П. В. Гляков*

ISBN 978-985-518-412-4

Часть 1

СТРУКТУРЫ ДАННЫХ ДЛЯ РЕШЕНИЯ ЗАДАЧ НА ИНТЕРВАЛАХ

Когда на практике требуется хранить в памяти последовательность элементов и обращаться к одиночным элементам этой последовательности по индексу, обычно используют массив фиксированного размера или динамический массив. Эта структура данных проста и эффективна.

Если нужно выполнить какую-либо произвольную операцию над k элементами (например, проверить, сколько из них удовлетворяют некоторому свойству), то в программе очевидным образом появляется цикл, пробегающий по k индексам и обращающийся к каждому элементу за $O(1)$, в итоге такая операция работает суммарно за время $\Omega(k)$ — требует выполнения не менее чем k обращений, плюс ещё время обработки самих элементов. В самом деле, это фундаментальное ограничение: не имея какой-то дополнительной информации, мы вынуждены просмотреть все k элементов и обработать каждый из них в отдельности, чтобы полностью выполнить операцию в общем случае.

Однако для конкретных задач иногда возможно разработать специальные структуры данных, которые позволяют выполнять групповые операции быстрее, т. е. обрабатывать один запрос, который сразу затрагивает k элементов, за время меньшее, чем $\Omega(k)$. В этой части мы рассмотрим примеры таких структур данных.

1.1. ПОСТАНОВКА ЗАДАЧИ О СУММЕ

Для простоты ограничимся случаем, когда k элементов, участвующих в операции, стоят в исходной последовательности рядом, т. е. образуют подпоследовательность подряд идущих элементов. Для обозначения

такого объекта будем использовать понятие *интервала* (англ. *interval*). Интервал задаётся двумя индексами его концов (в зависимости от типа интервала концы включаются или не включаются). Длина интервала может быть произвольной: от пустого интервала до интервала, захватывающего всю последовательность.

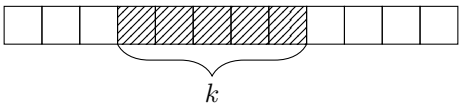

Рис. 1.1. Интервал длины k

Отметим, что вокруг понятия интервала в математике есть заметная терминологическая путаница (как в русскоязычной, так и в иностранной литературе). Некоторые авторы под интервалом подразумевают только открытый интервал (без включения концов). Другие авторы используют термин «промежуток» вместо слова «интервал». Кроме того, в некоторых книгах вводится понятие «отрезка» или «сегмента» для обозначения замкнутого интервала (концы включены). Чтобы не возникало неоднозначности, говоря об элементах массива, мы будем использовать понятие «интервал» и явно уточнять, включаются ли его концы в рассмотрение. Термин «отрезок» применять не будем, так как он имеет скорее геометрический оттенок.

Часто при программировании удобно работать с полуинтервалами, у которых левый конец включается, а правый исключается. Будем использовать следующее обозначение:

$$[l, r) = \{l, l + 1, ..., r - 1\}.$$

В отличие от ранее рассмотренных популярных структур, структуры данных для решения задач на интервалах обычно сильно завязаны на предметную область, плохо обобщаются и потому не представлены в стандартных библиотеках. Это создаёт дополнительные трудности для программиста. Раз использовать готовые реализации не получается, очень важно изучить подходы, идеи и алгоритмы, лежащие в основе построения таких структур данных, и уметь применить их при необходимости в конкретном практическом случае.

Рассмотрим следующую модельную задачу. Изначально дана последовательность чисел A длины n (индексация с нуля):

$$a_0, a_1, a_2, \ldots, a_{n-1}.$$

Поступают запросы двух типов.

• *Запрос модификации.* Задан индекс i и число x. Нужно прибавить к i-му элементу число x.

• *Запрос суммы.* Задана пара индексов l и r. Нужно вычислить сумму элементов на полуинтервале $[l, r)$, т. е. $a_l + a_{l+1} + \ldots + a_{r-1}$, и вернуть результат.

В этом случае запрос суммы называют *интервальным*, поскольку он затрагивает целый интервал значений.

1.2. НАИВНЫЙ ПОДХОД. ПОДСЧЁТ ПРЕФИКСНЫХ СУММ

Для хранения элементов будем использовать обычный массив. Запрос модификации выполняется за время $O(1)$, а вот запрос суммы требует линейного прохода по элементам. Поскольку длина интервала ограничена лишь размером массива, время выполнения запроса можно оценить как $O(n)$.

Практическая применимость такого решения обуславливается особенностями конкретной задачи. Если запросов первого типа много, а запросы второго типа редки, наивный подход будет работать хорошо.

Если посмотреть на проблему с другой стороны, можно получить альтернативное решение, которое даёт нам быстрое суммирование, но медленную модификацию.

Введём понятие *частичной суммы*, или *суммы на префиксе длины* k:

$$s_k = \sum_{i=0}^{k-1} a_i = a_0 + a_1 + \ldots + a_{k-1}.$$

По данной последовательности A массив префиксных сумм S можно построить за время $O(n)$, используя нехитрое рекуррентное соотношение:

$$\begin{aligned} s_0 &:= 0, \\ s_i &:= s_{i-1} + a_{i-1}, \quad i = 1, \ldots, n. \end{aligned}$$

Нетрудно видеть, что, исходя из свойств алгебраической операции сложения, сумма элементов массива a на полуинтервале $[l, r)$ равна разности двух префиксных сумм s_r и s_l.

Пример 1.1. Рассмотрим последовательность $(2, 1, 9, 7, 5, 2, 6)$.

a_i	2	1	9	7	5	2	6	
	0	1	2	3	4	5	6	7
s_i	0	2	3	12	19	24	26	32

Для суммы на полуинтервале $[2, 6)$ справедливы равенства:

$$\Sigma_{[2,6)} = a_2 + a_3 + a_4 + a_5 = 9 + 7 + 5 + 2 = 23;$$

$$\Sigma_{[2,6)} = s_6 - s_2 = 26 - 3 = 23.$$

Таким образом, при наличии массива S выполнение любого запроса суммы на полуинтервале сводится к вычислению разности двух чисел — это $\mathrm{O}(1)$. Запрос модификации выполняется сложнее: раз i-й элемент участвует в префиксных суммах s_{i+1}, s_{i+2}, …, s_n, то к каждой из этих сумм придётся прибавить число x. Значит, такой запрос выполняется за время $\mathrm{O}(n)$. Отметим, что такое простое решение прекрасно работает в случае, когда запросов модификации мало или нет вовсе.

Итак, используя лишь структуру данных «массив», мы можем решать задачу таким образом, что одна из операций выполняется за константу, а другая — за линейное время. Возникает логичный вопрос: можно ли разработать алгоритм, который даёт время лучше линейного сразу для обеих операций? Первоначально кажется, что линейного прохода не избежать, но на самом деле это не так, ответ на вопрос положительный. Мы далее рассмотрим две интересные идеи решения поставленной задачи. Идеального метода не существует, каждое решение предполагает определённый компромисс между разными аспектами (см. табл. 1.1).

Подход	Время на запрос модификации	Время на запрос суммы	Время на предпросчёт	Объём доп. памяти
Обычный массив	$\mathrm{O}(1)$	$\mathrm{O}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
Префиксные суммы	$\mathrm{O}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(n)$	$\mathrm{O}(1)$
Sqrt-декомпозиция	$\mathrm{O}(1)$	$\mathrm{O}(\sqrt{n})$	$\mathrm{O}(n)$	$\mathrm{O}(\sqrt{n})$
Дерево отрезков	$\mathrm{O}(\log n)$	$\mathrm{O}(\log n)$	$\mathrm{O}(n)$	$\mathrm{O}(n)$

Таблица 1.1. Сравнение различных методов решения

1.3. SQRT-ДЕКОМПОЗИЦИЯ

В наивном подходе узким местом является операция подсчёта суммы: выполняется проход по всем элементам интервала. Возникает такая идея: можно разбить весь массив на блоки, посчитать заранее суммы для целых блоков; затем при суммировании в цикле можно будет «перепрыгивать» блоки, сумма для которых уже известна (рис. 1.2).

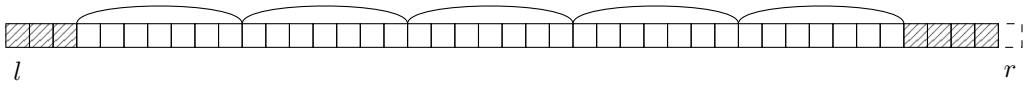

Рис. 1.2. Обработка запроса суммы на $[l, r)$

Пусть исходный массив A имеет размер n, индексы начинаются с нуля. Этот массив естественным образом разбивается на $\lceil n/k \rceil$ блоков размера k (последний блок, возможно, будет неполным). Блоки также удобно нумеровать с нуля, при этом элемент i массива относится к блоку с индексом $\lfloor i/k \rfloor$. Для хранения сумм по блокам нам понадобится дополнительный массив B. Элемент b_j вычисляется так:

$$b_j = a_{j \cdot k} + a_{j \cdot k+1} + \ldots + a_{(j+1) \cdot k-1} \tag{1.1}$$

(если какой-либо индекс выходит за границы массива A, полагаем слагаемое равным нулю).

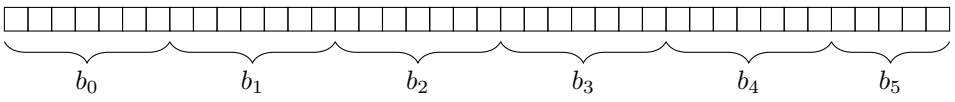

Рис. 1.3. Расчёт сумм по блокам при $n = 40$ и $k = 7$

Изначально, когда структура данных инициализируется, значения массива B вычисляются по формуле (1.1) исходя из значений в массиве A.

Затем, когда приходит запрос модификации и нужно прибавить к i-му элементу число x, мы делаем две операции: увеличиваем a_i на x, увеличиваем $b_{\lfloor i/k \rfloor}$ также на x. Таким образом поддерживается согласованность массивов A и B. Операции выполняются за время $O(1)$.

Запрос суммы на полуинтервале $[l, r)$ можно обрабатывать так.

Для начала определим индексы блоков, к которым соответствуют границы интервала:

$$j_l := \lfloor l/k \rfloor,$$
$$j_r := \lfloor r/k \rfloor.$$

Если эти числа оказались равными, то обе границы попали в один блок, а значит, сумму можно вычислить напрямую, используя только массив A. На это будет затрачено время $\mathrm{O}(k)$. Если же $j_l < j_r$, то итоговая сумма будет складываться из трёх частей.

1. «Хвост» j_l-го блока: слагаемые от a_l (включительно) до $a_{(j_l+1)\cdot k}$ (не включительно). Вычисляется циклом по массиву A за время $\mathrm{O}(k)$.

2. Промежуточные блоки, которые попадают в интервал целиком: слагаемые от b_{j_l+1} (включительно) до b_{j_r} (не включительно). Вычисляется циклом по массиву B за время $\mathrm{O}(n/k)$.

3. «Голова» j_r-го блока: слагаемые от $a_{j_r \cdot k}$ (включительно) до a_r (не включительно). Вычисляется циклом по массиву A за время $\mathrm{O}(k)$.

Итого общее время выполнения такого запроса будет равно

$$T(n) = \mathrm{O}(k) + \mathrm{O}\left(\frac{n}{k}\right) + \mathrm{O}(k) = \mathrm{O}\left(k + \frac{n}{k}\right).$$

Каким лучше выбрать размер блока k? Точный ответ зависит от конкретных констант, скрытых в асимптотике, но можно попробовать исследовать функцию, стоящую под O, на экстремум средствами математического анализа.

Действительно, у функции

$$f(k) = k + \frac{n}{k}$$

производная имеет вид

$$f'(k) = 1 - \frac{n}{k^2}.$$

Приравнивая это выражение к нулю, получаем, что в точке $k = \sqrt{n}$ достигается минимум, равный $2\sqrt{n}$.

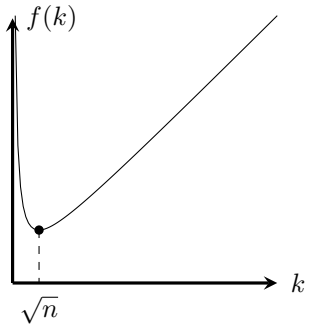

Рис. 1.4. График функции

Таким образом, оптимально разбивать на блоки размера примерно по корню квадратному из n каждый. При этом время обработки запроса тоже будет величиной порядка $\sqrt{n}$. Поэтому такой приём называют *Sqrt-декомпозицией*, или *корневой эвристикой*.

Реализуем в псевдокоде описанную структуру данных.

```
class Summator:
    def __init__(self, a):
        self.a = a
        self.k = floor(sqrt(len(a)))

        self.b = []
        for i in range(0, len(a), self.k):
            bsum = sum(self.a[i:(i + self.k)])
            self.b.append(bsum)

    def Add(self, i, x):
        self.a[i] += x
        self.b[i // self.k] += x

    def FindSum(self, l, r):
        jl = l // self.k
        jr = r // self.k
        if jl == jr:  # same block
            return sum(self.a[l:r])
        else:
            return (
                sum(self.a[l:((jl + 1) * self.k)]) +
                sum(self.b[(jl + 1):jr]) +
                sum(self.a[(jr * self.k):r])
            )
```

1.4. ДЕРЕВО ОТРЕЗКОВ

Дальнейшим развитием идеи разбиения на блоки будет такая идея: можно использовать блоки разной длины и организовать их в виде бинарного дерева.

Пусть каждая вершина дерева представляет некоторый отрезок (или, для удобства, полуинтервал) элементов исходного массива A. Корень дерева соответствует всему массиву — полуинтервалу $[0, n)$. Далее, если какая-либо вершина связана с полуинтервалом $[t_l, t_r)$, причём $t_r - t_l > 1$, то у этой вершины будет два потомка: один отвечает за часть $[t_l, m)$, второй — за часть $[m, t_r)$, где $m = \lfloor (t_l + t_r)/2 \rfloor$ (рис. 1.5). Листьями дерева будут вершины, для которых полуинтервал имеет вид $[x, x + 1)$ и содержит всего один элемент.

Такое дерево и называют *деревом отрезков*. В англоязычной традиции нет устоявшегося термина для этой структуры данных. Можно встретить вариант *segment tree*, но в литературе это словосочетание

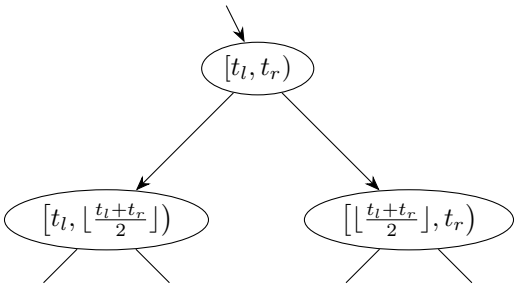

Рис. 1.5. Фрагмент дерева отрезков

часто употребляется в другом смысле: для обозначения структуры данных, которая работает с геометрическими отрезками, а вовсе не с отрезками массива, и позволяет для точки определять, какие отрезки её содержат.

Существует также понятие «дерево интервалов» (англ. *interval tree*): обычно оно относится к совсем другой структуре данных, которая здесь рассматриваться не будет.

1.4.1. Организация

Пример дерева отрезков показан на рис. 1.6. Из рисунка нетрудно видеть, что дерево отрезков в общем случае не является полным (уровни могут быть заполнены не целиком). Однако, тем не менее, высота этого дерева имеет порядок $\mathrm{O}(\log n)$, число листьев равно n, каждый лист однозначно соответствует элементу исходного массива. Более того, общее число вершин в дереве есть величина порядка $\mathrm{O}(n)$.

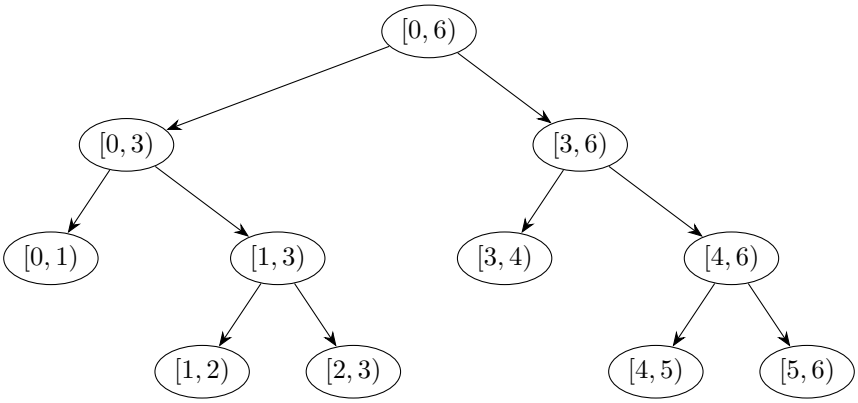

Рис. 1.6. Дерево отрезков, построенное для шести элементов

Теорема 1.1. *В дереве отрезков, которое построено для последовательности из n элементов, число вершин равно $2n - 1$.*

Доказательство. Применим метод математической индукции.

Если $n = 1$, то по формуле получим $2n - 1 = 1$. Действительно, дерево отрезков состоит из одной вершины, поэтому утверждение верно.

Если $n > 1$, то полуинтервал $[0, n)$ разобьётся на два подинтервала $[0, m)$ и $[m, n)$. Для каждого подинтервала будет независимо построено дерево отрезков, затем будет добавлена вершина для $[0, n)$, объединяющая два этих дерева. Общее число вершин будет равно

$$(2m - 1) + (2(n - m) - 1) + 1 = 2n - 1.$$

Итого, в дереве ровно n листьев и $n - 1$ внутренняя вершина. □

В каждой вершине дерева отрезков содержится сумма элементов исходного массива, индексы которых принадлежат соответствующему отрезку. Так, в корне дерева лежит общая сумма всех элементов массива. В листьях дерева находятся сами элементы.

1.4.2. Хранение в памяти

Дерево отрезков является частным случаем бинарного дерева, поэтому можно организовать хранение дерева отрезков в памяти компьютера с помощью указателей или ссылок. Однако более рационально хранить все элементы дерева отрезков в массиве и задавать структуру дерева неявно через индексы, как это делается для бинарной кучи (см. раздел **??**): индексация в массиве начинается с единицы, у вершины с индексом v левый сын имеет индекс $2v$, правый сын — $2v + 1$. Таким образом, каждая вершина хранится в виде одного числа.

В случае, когда n не степень двойки, бинарное дерево не является полным, из-за чего в массиве могут быть неиспользуемые ячейки. Чтобы места гарантированно хватило, можно выставить размер массива равным $4n$, или же оценить нужный размер более точными формулами.

Пример 1.2. Для массива из $n = 6$ элементов $(9, 2, 5, 3, 6, 1)$ дерево отрезков, условно показанное на рис. 1.7, в виде массива будет храниться так:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
26	16	10	9	7	3	7	—	—	2	5	—	—	6	1

Обратите внимание, что количество чисел равно 11, т. е. $2n - 1$.

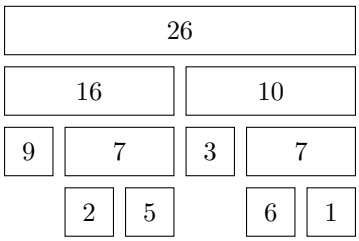

Рис. 1.7. Дерево отрезков для массива $(9, 2, 5, 3, 6, 1)$

1.4.3. Построение дерева отрезков

Процесс построения дерева отрезков по заданному массиву A удобно описывать рекурсивно.

Пусть мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$. Если вершина является листом, т. е. $t_r = t_l + 1$, то в вершину дерева заносим сам элемент массива $A[t_l]$. Иначе определяем границу m, вызываем функцию рекурсивно на $[t_l, m)$ для левого поддерева (его корень имеет индекс $2v$) и на $[m, t_r)$ для правого поддерева (его корень с индексом $2v+1$), агрегируем суммы потомков и получаем финальное значение для текущей вершины.

```
def DoBuild(a, t, v, tl, tr):
    if tr - tl == 1:
        t[v] = a[tl]
    else:
        m = (tl + tr) // 2
        DoBuild(a, t, v=2*v, tl=tl, tr=m)
        DoBuild(a, t, v=2*v+1, tl=m, tr=tr)
        t[v] = t[2*v] + t[2*v+1]
```

Процедура запускается от корня дерева следующим образом:

```
def Build(a, n):
    t = [None] * 4*n  # array of size 4n
    DoBuild(a, t, v=1, tl=0, tr=n)
    return t
```

Теорема 1.2. *По массиву из n элементов дерево отрезков строится за $O(n)$.*

Доказательство. Действительно, число вершин в дереве линейное, благодаря стеку рекурсии значения вычисляются снизу вверх, для каждой вершины значение определяется за $O(1)$ на основании значений вершин-потомков. □

1.4.4. Запрос модификации

Реализуем рекурсивную функцию, которая будет выполнять прибавление числа x к элементу с индексом i. Предположим, что мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$.

Если текущая вершина — лист, то достаточно прибавить x к числу, которое хранится в этой вершине. Иначе определяем границы полуинтервалов, которые соответствуют двум потомкам текущей вершины: $[t_l, m)$ и $[m, t_r)$. Если $i < m$, то выполняем рекурсивный вызов для левого сына, иначе — для правого сына. Наконец, пересчитаем значение суммы в текущей вершине точно таким же образом, как делали это при первоначальном построении дерева отрезков.

```
def DoAdd(t, v, tl, tr, i, x):
    if tr - tl == 1:
        t[v] += x
        return
    m = (tl + tr) // 2
    if i < m:
        DoAdd(t, v=2*v, tl=tl, tr=m, i=i, x=x)
    else:
        DoAdd(t, v=2*v+1, tl=m, tr=tr, i=i, x=x)
    t[v] = t[2*v] + t[2*v+1]
```

Стартовый вызов рекурсивной функции делаем так:

```
def Add(t, n, i, x):
    DoAdd(t, v=1, tl=0, tr=n, i=i, x=x)
```

Теорема 1.3. *Время работы операции модификации одиночного элемента в дереве отрезков есть* $\mathrm{O}(\log n)$.

Доказательство. Заметим, что раз отрезки на каждом уровне не пересекаются (по построению), то i-й элемент входит в $\mathrm{O}(\log n)$ отрезков (не более чем в один отрезок на уровне). Поэтому для выполнения запроса модификации требуется обновить не более чем $\mathrm{O}(\log n)$ значений в дереве отрезков. □

1.4.5. Запрос суммы на интервале

Эта задача более сложная. Реализуем рекурсивную функцию, которая будет её решать. Пусть нужно вычислить сумму на полуинтервале $[l, r)$. Мы находимся в вершине дерева отрезков с индексом v, она соответствует некоторому полуинтервалу $[t_l, t_r)$.

Если $[l, r) = [t_l, t_r)$, то в качестве ответа сразу возвращаем предварительно вычисленное значение суммы, записанное в дереве.

В противном случае определяем границы полуинтервалов, соответствующих двум потомкам: $[t_l, m)$ и $[m, t_r)$. Если $[l, r) \subseteq [t_l, m)$, то рекурсивно вызываем функцию для левого потомка (эта вершина дерева имеет индекс $2v$), не изменяя границы запроса. Аналогичную проверку $[l, r) \subseteq [m, t_r)$ делаем для правого потомка (он имеет индекс $2v + 1$).

Если же запрос $[l, r)$ пересекается с полуинтервалами обоих потомков, то нет другого выхода, кроме как перейти в левого потомка и вычислить ответ для фрагмента $[l, m)$, перейти в правого потомка и вычислить ответ для фрагмента $[m, r)$, затем просуммировать оба ответа.

```
def DoFindSum(t, v, tl, tr, l, r):
    if l == tl and r == tr:
        return t[v]
    m = (tl + tr) // 2
    if r <= m:
        return DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=r)
    if m <= l:
        return DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=l, r=r)
    return (
        DoFindSum(t, v=2*v, tl=tl, tr=m, l=l, r=m) +
        DoFindSum(t, v=2*v+1, tl=m, tr=tr, l=m, r=r)
    )
```

Таким образом, алгоритм представляет собой спуск по дереву отрезков, который посещает нужные ветви и для быстрой работы использует уже посчитанные суммы, хранящиеся в дереве.

Чтобы вычислить ответ для полуинтервала $[l, r)$, нужно начать путь из корня дерева, вызывая функцию `DoFindSum` с правильными аргументами:

```
def FindSum(t, n, l, r):
    return DoFindSum(t, v=1, tl=0, tr=n, l=l, r=r)
```

Возникает вопрос о том, каково время работы описанного алгоритма. Если каждый раз рекурсия будет уходить в обе ветви, то общее время будет линейным. Однако на практике этого не происходит, и вот почему.

Теорема 1.4. *Время работы операции нахождения суммы на интервале есть* $O(\log n)$.

Доказательство. Предположим вначале, что левая граница полуинтервала, на котором нужно искать сумму, равна нулю, т. е. полуин-

тервал захватывает начало массива и имеет вид $[0, r)$.

На каждом шаге алгоритма возможна одна из трёх конфигураций, показанных на рис. 1.8.

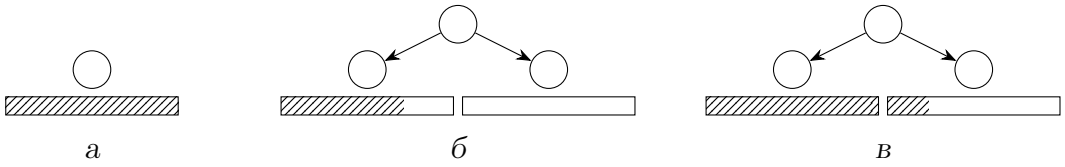

Рис. 1.8. Возможные ситуации на очередном шаге рекурсии для $[0, r)$: *а* — остановка; *б* — спуск только влево; *в* — спуск влево и вправо

В случае *а* отрезок, который представляется текущей вершиной, полностью совпадает с искомым, поэтому алгоритм прекращает работу и возвращает значение, записанное в вершине дерева для этого отрезка, за $O(1)$.

В случае *б* выполняется рекурсивный спуск только в левое поддерево, поскольку в правом поддереве нет интересующих нас элементов.

Наибольший интерес представляет случай *в*. Здесь рекурсия разветвляется, делаются два рекурсивных вызова. Однако левая рекурсивная цепочка остановится на следующем же шаге из-за того, что придёт к случаю *а*. Продолжит выполняться только правая цепочка.

Таким образом, можно заметить, что активно работает только одна ветвь рекурсии на каждом уровне. Раз число уровней имеет порядок логарифма, то общее время обработки запроса есть $O(\log n)$.

Аналогичные рассуждения можно провести для случая, когда интервал захватывает конец массива и имеет вид $[l, n)$.

Пусть теперь интервал, на котором ищется сумма, произвольный — $[l, r)$. Начиная рекурсивный спуск от корня, алгоритм всякий раз приходит к одной из конфигураций, показанных на рис. 1.9.

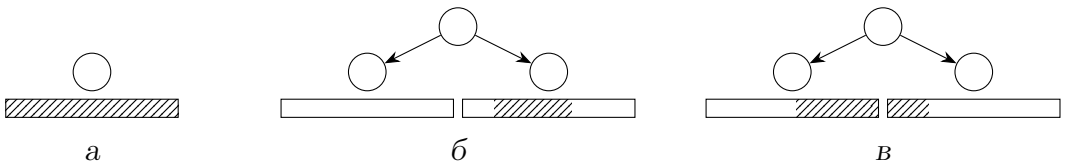

Рис. 1.9. Возможные ситуации на очередном шаге рекурсии для $[l, r)$: *а* — остановка; *б* — спуск в одну сторону; *в* — спуск в обе стороны

В случае *а*, как и ранее, работа прекращается. В случае *б* спуск продолжается только по одной из ветвей. Наконец, если через какое-то

время (не более чем через $O(\log n)$ шагов) достигнута конфигурация $в$, то дальнейшая задача сводится к двум независимым подзадачам описанного ранее вида. Эти подзадачи решаются двумя активно работающими рекурсивными вызовами. □

1.4.6. Обобщения

Можно также построить нерекурсивные реализации описанных операций. На практике код без рекурсии обычно работает быстрее. Кроме того, можно выполнять операции не «сверху вниз», а «снизу вверх», как описано в [5].

Выше рассматривалась задача, когда запрос модификации затрагивает единственный элемент массива. На самом деле дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время $O(\log n)$.

Концепция интервальных запросов очень гибкая и может быть распространена на многие другие задачи.

1.5. ЗАДАЧА RMQ. SPARSE TABLE

Операцию сложения в разделе 1.1 можно заменить на какую-либо иную операцию: взятие минимума, максимума и пр. Отдельное название в литературе получила задача поиска минимального значения на интервале — RMQ (англ. *range mininum query*) — в связи с её практической важностью. К этой задаче при определённых условиях сводится, например, задача поиска наименьшего общего предка в дереве — LCA (англ. *lowest common ancestor*).

Нетрудно заметить, что для решения задачи RMQ можно применить корневую эвристику или дерево отрезков.

1.5.1. Статическая версия RMQ

Предположим, что запросов модификации нет, все данные статичны. Есть некоторый постоянный массив A, состоящий из n чисел. Поставим задачу следующим образом: поступают запросы вида $[l, r)$, ответом на запрос является минимальное значение среди элементов массива $a_l, a_{l+1}, \ldots, a_{r-1}$. Нужно уметь отвечать на такие запросы за $O(1)$.

Простой префиксный метод здесь не работает: если известен минимум в массиве на полуинтервале $[0, l)$ и минимум на полуинтервале

$[0, r)$, о минимуме на $[l, r)$ в общем случае ничего сказать нельзя.

Очевидное решение — посчитать заранее ответы на всевозможные запросы и сохранить их в двумерном массиве, чтобы затем извлекать за $O(1)$. Полная таблица T, где элемент $t_{i,j}$ содержит минимум на полуинтервале $[i, j)$, строится за время $\Theta(n^2)$ и занимает $\Theta(n^2)$ памяти. Эти расходы слишком велики. Оказывается, можно построить таблицу меньшего размера — «разрежённую» — так, что она будет занимать $O(n \log n)$ памяти, а время ответа на каждый запрос будет по-прежнему константным.

1.5.2. Sparse table

Sparse table (русск. *разрежённая таблица*, но этот термин не является устоявшимся) представляет собой двумерную структуру, в которой хранятся минимумы на всех интервалах, длины у которых являются некоторыми степенями двойки. Другими словами, для каждой позиции i массива A будут подсчитаны минимумы на всех интервалах длины $1, 2, 4, \ldots$ вправо от i. Элемент $t_{i,k}$ таблицы определяется формулой

$$t_{i,k} = \min\{a_i, a_{i+1}, \ldots, a_{i+2^k-1}\}.$$

Нетрудно заметить, что нет смысла рассматривать значения k больше, чем $\lfloor \log_2 n \rfloor$. Для фиксированного k заполняются лишь ячейки, для которых $i + 2^k \leqslant n$. Тем не менее общий размер таблицы — $O(n \log n)$.

Для эффективного вычисления $t_{i,k}$ можно использовать следующее рекуррентное соотношение:

$$\begin{aligned} &t_{i,0} := a_i; \\ &t_{i,k} = \min\{t_{i,k-1}, t_{i+2^{k-1},k-1}\}, \quad k \geqslant 1. \end{aligned}$$

Пример 1.3. Для массива $(2, 9, 1, 9, 6, 7, 5, 2)$ из восьми элементов разрежённая таблица имеет следующий вид (для наглядности каждому k соответствует одна строка, каждому i — столбец):

k	2^k	0	1	2	3	4	5	6	7
0	1	2	9	1	9	6	7	5	2
1	2	2	1	1	6	6	5	2	
2	4	1	1	1	5	2			
3	8	1							

Допустим, таблица сформирована, тогда ответ на запрос минимума в массиве на полуинтервале $[l, r)$ будет вычисляться следующим образом.

Найдём максимальную степень двойки p, которая не превосходит длины заданного полуинтервала:

$$2^p \leqslant r - l < 2^{p+1}.$$

Для этого можно использовать следующую формулу:

$$p = \lfloor \log_2(r - l) \rfloor.$$

На практике, чтобы избежать операций с плавающей точкой при вычислении логарифма, можно заранее просчитать значения p для всевозможных длин при помощи простого рекуррентного соотношения и сохранить их в таблицу. Так или иначе, число p можно найти за $\mathrm{O}(1)$.

Пример 1.4. Полуинтервал $[3, 8)$ имеет длину 5, значит, $p = 2$.

Далее, нетрудно показать, что полуинтервал $[l, r)$ может быть покрыт не более чем двумя полуинтервалами, длина которых равна 2^p. Возможно, эти интервалы будут пересекаться.

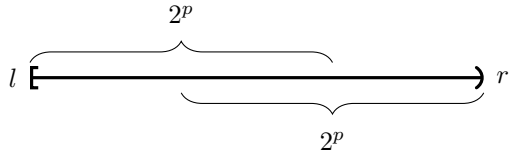

Рис. 1.10. Покрытие интервала $[l, r)$ двумя интервалами длины 2^p

Искомый минимум на $[l, r)$ равен минимуму из двух минимумов: на $[l, l + 2^p)$ и $[r - 2^p, r)$. Эти два минимума посчитаны заранее и хранятся в sparse table, значит, можно получить ответ на запрос за время $\mathrm{O}(1)$:

$$\min\{t_{l,p}, t_{r-2^p,p}\}.$$

Отметим, что описанный подход можно применить к операции максимума, но нельзя, например, к операции сложения, так как сумма сумм на двух перекрывающихся подинтервалах не равна сумме на целом интервале.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — М. : Вильямс, 2005. — 1296 с.

2. *Котов В. М.*, *Мельников О. И.* Информатика. Методы алгоритмизации : учеб. пособие для 10 – 11 кл. общеобразоват. шк. с углубл. изучением информатики. — Минск : Нар. асвета, 2000. — 221 с.

3. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. — Минск : БГУ, 2011. — 267 с. — (Классическое университетское издание).

4. Сборник задач по теории алгоритмов : учеб.-метод. пособие/ В. М. Котов [и др.]. — Минск : БГУ, 2017. — 183 с.

5. Теория алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.

6. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета // Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26–27 февр. 2019 г. / Белорус. гос. ун-т ; редкол.: А. Д. Король (пред.) [и др.]. — Минск : БГУ, 2019. — С. 263-267.

7. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRunner // Судьбы классического университета: национальный контекст и мировые тренды [Электронный ресурс] : материалы XIII Респ. междисциплинар. науч.-теорет. семинара «Инновационные стратегии в современной социальной философии» и междисциплинар. летней школы молодых ученых «Экология культуры», Минск, 9 апр. 2019 г. / Белорус. гос. ун-т ; сост.: В. В. Анохина, В. С. Сайганова ; редкол.: А. И. Зеленков (отв. ред.) [и др.] — С. 346–355.

СОДЕРЖАНИЕ

Учебное издание

Соболь Сергей Александрович
Вильчевский Константин Юрьевич
Котов Владимир Михайлович и др.

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Редактор *Х. Х. XXXXXXX*
Художник обложки *С. А. Соболь*
Технический редактор *Х. Х. XXXXXXX*
Компьютерная вёрстка *С. А. Соболя*
Корректор *Х. Х. XXXXXX*

Подписано в печать 29.02.2020. Формат 60×84/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10,69. Уч.-изд. л. 9,6.
Тираж 150 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/270 от 03.04.2014.
Пр. Независимости, 4, 220030, Минск.

Республиканское унитарное предприятие
«Издательский центр Белорусского государственного университета».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №2/63 от 19.03.2014.
Ул. Красноармейская, 6, 220030, Минск.