MIPT, ЗКШ, февраль 2015
Лекция про структуры данных

Копелиович Сергей Собрано 27 февраля 2015 г. в 09:35

Содержание

1. STL **1**
- 1.1 `vector` и range check error . . . 1
- 1.2 `set` и `map` . . . 2
- 1.3 k-й элемент в множестве . . . 3
- 1.4 Переопределяем аллокатор . . . 3

2. Функции на отрезках **4**
- 2.1 Частичные суммы . . . 4
- 2.2 Дерево отрезков . . . 4
- 2.3 Дерево Фенвика . . . 4
- 2.4 Persistent Scanline . . . 4
- 2.5 Sparse Table и его модификации . . . 5
- 2.6 Алгоритм Фараха-Колтона-Бендера . . . 6
- 2.7 Аналог Sparse Table для суммы . . . 6

3. Количество различных чисел на отрезке **7**
- 3.1 Формулировка задачи . . . 7
- 3.2 Решение задачи . . . 7

4. k-я порядковая статистика **8**
- 4.1 Общие идеи . . . 8
- 4.2 Решение за логарифм n для неменяющегося массива . . . 8
- 4.3 А теперь массив меняется . . . 8
- 4.4 Применение новой структуры . . . 9

STL

Да прибудет с вами сила STL

Напутствие перед контетом

1.1. vector и range check error

vector – массив переменной длины с range check-ами и прочими плюшками.

Основные, полезные нам функции:

```
vector<int> a(n, 1); // вектор длины n, заполненный единицами
vector<int> b; b.reserve(n); // вектор длина 0, уже выделена память на n ячеек
b.push_back(1); // мы уверены, что не произойдёт перевыделение памяти
b.size(); // размер
b[i]; // обращение как с обычным массивом
b.at(i); // то же, что и выше, но с проверкой выхода за пределы 0..size()
b.clear(); // размер вектора теперь 0, зарезервированная память не освобождена
b.resize(0); // то же, что clear
```

Автоматический отлов выходов за пределы массива. Пусть в вашем коде были `int a[N]` и `vector<int> b(n)`. Пусть вы к ним иногда обращались. И естественным образом, однажды случайно обратившись к `a[-1]`, получали `undefined behavior` (по-русски: дальше может случиться, что угодно). Если вы сталкивались с ситуацией "закомментировал debug-вывод, не работает, раскомментировал обратно, заработало", это были последствия того самого `undefined behavior`, который в `C++` проще всего заработать, обратившись к чужой памяти (например, `a[-1]`).

Есть, конечно, более надёжный способ: `a[i] = a[i++ + 1];`. Но так, надеюсь, никто из вас не пишет =). Ни массив, ни обычный вектор не обязаны ловить выходы за пределы массива. У вектора есть метод `at(i)` – обращение к i-му элементу с проверкой границ, но повсеместно его используя, мы получем менее красивый (читабельный) код. Для автоматической ловли ошибок есть ещё и такой подход:

```
template <class T>
struct MyVector : vector<T> {
  MyVector() : vector<T>() { }
  MyVector( int n ) : vector<T>(n) { }
  T &operator [] ( int i ) {
    assert(0 <= i && i < (int)vector<T>::size());
    return vector<T>::operator[](i);
  }
};
```

И везде (вместо всех массивов и векторов) использовать `MyVector`. Пример:

```
MyVector<int> b(2); // size = 2
b.push_back(0); // size += 1
b[2];
b[3]; // сработает assert
```

1.2. set и map

Упорядоченные множества

```
#include <set>
#include <map>
std::set<int> s; // внутри живёт красно-чёрное дерево, все операции за O(logn)
std::map<int, int> m; // внутри живёт set<pair<int,int>>
```

`map` иногда используют как обычный массив с широким диапазоном индексов. В таких случаях почти всегда уместнее использовать `unordered_map`. `set` иногда используют как именно упорядоченное множество: (`s.lower_bound(x)`), иногда как кучу: (min = `s.begin()`, max = `s.rbegin()`), а иногда как множество, для быстрых проверок "лежит ли элемент в множестве" в последнем случае уместнее `unordered_set`.

Неупорядоченные множества

```
// -std=c++11
#include <unorderd_set>
#include <unorderd_map>
std::unorderd_set<int> hs; // внутри живёт хеш-таблица, все операции за O(1)
std::unorderd_map<int, int> hm; // внутри живёт хеш-таблица, все операции за O(1)
```

Как заставить стандартный unorderd_set<int> работать быстро?

```
// -std=c++11
#include <unorderd_set>
#include <unorderd_map>
const int N = 1e6;
// максимальное число элементов, которое мы собираемся класть в хеш-таблицу
std::unorderd_set<int> hs(N);
hs.rehash(N);
```

Пара приёмов использования

```
// Пример #1
unordered_set<int> visited;
void go( int state ) { // перебор с запоминанием
  if (visited.insert(state).second) // попробовали добавить
    return; // если уже был, вышли
  ...
}
// Пример #2
unordered_map<int, int> f;
void go( int state ) { // перебор с запоминанием
  int &result = f[state];
  if (result != 0) // уже были здесь
    return;
  ...
  return result = ...; // обещаем, что result != 0
}
```

Есть случай, когда лучше set, чем unordered_set

```
  const int N = 1e6;
  set<int> s[N]; // быстрее
  unordered_set<int> hs[N]; // медленнее
```

1.3. k-й элемент в множестве

Пусть мы умеем написать дерево поиска (например, декартово) и в каждой вершине дерева поддерживать размер. Тогда мы за $\mathcal{O}(\log n)$ умеем отвечать на запросы "элемент по номеру", "номер по элементу". Тоже самое можно делать средствами gnu-сного расширения стандартной библиотеки:

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> s;
s.insert(x);
s.erase(x);
s.count(x);
s.find_by_order(k); // k-й по величине элемент в множестве
s.order_of_key(x); // сколько элементов в множестве меньше x
```

Внутри `tree<int,...>` живёт красно-чёрное дерево, все операции делаются за $\mathcal{O}(\log n)$. `tree<int,...>` умеет всё тоже, что и `set<int>` плюс кое-что ещё.

1.4. Переопределяем аллокатор

Есть общее средство ускорить `STL`. На олимпиаде раза в полтора-два ускорить код, пожертвовав "правильным освобождением памяти" – весьма ценно. Итак, у `STL`-контейнеров уйма времени уходит на работу с памятью.

Давайте переопределим аллокатор

```
const int MAX_MEM = 1e8;
int mpos = 0;
char mem[MAX_MEM];
inline void * operator new ( size_t n ) {
  char *res = mem + mpos;
  mpos += n;
  assert(mpos <= MAX_MEM);
  return (void *)res;
}
inline void operator delete ( void * ) { }
```

Функции на отрезках

Даёшь всё за логарифм!

Революционный призыв

2.1. Частичные суммы

С помощью предподсчитанных сумм на префиксах мы можем считать значение обратимой функции на отрезке. Например, сумму чисел $sum(l, r) = sum(r) - sum(l{-}1)$. Также можно считать произведение не нулей, композицию перестановок, произведение обратимых матриц.

Запомним: предподсчитали простую функцию на всех префиксах; изменение делать нельзя; чтобы считать функцию на отрезке, функция должна быть обратимой.

2.2. Дерево отрезков

Позволяет считать функцию на отрезке и делать изменение в точке. Чтобы посчитать функцию на отрезке, отрезок разбивается на не более чем $2\log_2 n$ вершин дерева отрезков, для которых значение функции уже посчитано.

Например, можно в каждой вершине дерева отрезков хранить сумму на отрезке, тогда мы можем считать сумму на отрезке и делать изменение в точке. Обе операции за $\mathcal{O}(\log n)$. А можно в каждой вершине дерева отрезков хранить дерево поиска (например, декартово дерево), тогда мы можем считать “количество $i\colon l \le i \le r, x \le a_i \le y$” и делать изменение в точке. Обе операции за $\mathcal{O}(\log^2 n)$.

Какие ещё функции можно считать деревом отрезков? min, max, gcd, композиция перестановок, произведение по модулю (даже не обязательно простому), произведение матриц $2{\times}2$, сумма парабол $(a_i x^2{+}b_i x{+}c_i)$, ... Любую ассоциативную функцию.

Запомним: предподсчитали функцию на $\mathcal{O}(n)$ отрезках, используя $\mathcal{O}(n)$ памяти, любой отрезок $[l..r]$ можем разбить на $\mathcal{O}(\log n)$ непересекающихся отрезков, на которых функция уже посчитана. Таким образом научились считать любую ассоциативную функцию на отрезке.

2.3. Дерево Фенвика

Позволяет считать функцию на префиксе и делать изменение в точке. Плюсы по сравнению с деревом отрезков: меньше кода, меньше памяти, меньше константа в оценке времени работы.

Запомним: всё тоже, что и у дерева отрезков, но функция должна быть обратимой.

2.4. Persistent Scanline

Персистентное дерево отрезков: делая изменение дерева отрезков за $\mathcal{O}(\log n)$, мы получаем новое дерево отрезков, при этом у нас остаётся возможность пользоваться старым, то есть каждая операция изменения порождает новое дерево отрезков, после n операций у нас n деревьев отрезков, которые в сумме занимают $\mathcal{O}(n \log n)$ памяти.

Например, если мы хотим отвечать на запрос “количество $i\colon l \le i \le r, x \le a_i \le y$”, то мы можем для каждого префикса $[1..r]$ насчитать дерево отрезков $tree_r$ с операцией сумма на отрезке, которое умеет отвечать на запрос $get(x, y)$ “количество $i\colon i \le r, x \le a_i \le y$”. $tree_{r+1}$ получается из $tree_r$ изменением в точке. Тогда ответ на исходный запрос “количество $i\colon l \le i \le r, x \le a_i \le y$” равен $tree_r(x, y) - tree_{l-1}(x, y)$, что считается за $\mathcal{O}(\log n)$.

Запомним: предподсчитали **сложную** функцию на всех префиксах; изменение делать нельзя; чтобы считать функцию на отрезке, функция должна быть обратимой.

2.5. Sparse Table и его модификации

Обычно изучается в контексте “структура, которая позволяет посчитать минимум на отрезке за $\mathcal{O}(1)$”.

Предподсчёт

```
for (int i = 0; i < n; i++)
  f[0][i] = a[i];
for (int k = 0; (1 << (k + 1)) < n; k++)
  for (int i = 0; i < n; i++)
    f[k+1,i] = min(f[k][i], f[k][i + (1 << k)]);
for (int i = 2; i < n; i++)
  maxK[i] = maxK[i >> 1] + 1;
```

Использование

```
get(l, r) {
  int k = maxK[r - l + 1]; // максимальная степень двойки не более длины отрезке
  return min(f[k][l], f[k][k][r - (1 << k) + 1]);
}
```

Вопрос: можем ли мы посчитать сумму чисел на отрезке с помощью той же идеи? Нет, не можем. Отрезки перекрываются, некоторые числа мы учтём в сумме несколько раз.

Вопрос: можем ли мы посчитать gcd чисел на отрезке с помощью той же идеи? Можем. Потому что также как и $\min(a, a) = a$ и $\gcd(a, a) = a$ (идемпотентность). Ещё мы пользуемся коммутативностью и ассоциативностью. Грубо говоря, мне нужно, чтобы $f(a, b, c, b, c, d) = f(f(a, b, c), f(b, c, d))$. Здесь $f(a) = a, f(a, b, \ldots) = f(a, f(b, \ldots))$. Чтобы это было так, достаточно раскрыть скобки, поменять местами слагаемые и воспользоваться идемпотентностью: $f(b, b) = b, f(c, c) = c$.

Улучшаем время работы. Сейчас Sparse Table сохраняет предподсчитанную функцию для $\mathcal{O}(n \log n)$ отрезков и разбивает любой отрезок $[L..R]$ на $2 = \mathcal{O}(1)$ возможно пересекающихся отрезка. Сделаем предподсчёт для $\mathcal{O}(n \log \log n)$ отрезков и будем разбивать любой отрезок $[L..R]$ на $4 = \mathcal{O}(1)$ возможно пересекающихся отрезка. Пусть $k = \lceil \log_2 n \rceil$. Обозначим за s_i отрезок $[ki..k(i+1))$. $b[i] = \min\limits_{j \in s_i} a[j]$. Длина массива b равна $\mathcal{O}(\frac{n}{\log n})$. Можно построить на b Sparse Table. Также насчитаем для каждого отрезка s_i минимумы на всех префиксах и суффиксах. Как ответить на запрос min на $[l..r]$? Если отрезок пересекает хотя бы одну границу, точку $k{\cdot}i$, то он разбивается на префикс + запрос к Sparse Table + суффикс. Иначе он целиком лежит в каком-то из s_i. Давайте на каждом отрезке s_i создадим свой маленький Sparse

Table размера $k \log k = \log n \log\log n$. Суммарный размер структуры $n \log\log n$ отрезков, на которых мы предподсчитали функцию. Ответ на запрос работает за $\mathcal{O}(1)$.

Можно ещё улучшить. Мы получили 2-уровневый Sparse Table, можно сделать многоуровневый и получить предподсчёт на $\mathcal{O}(n)$ отрезках и разбиение отрезка $[l..r]$ на $\mathcal{O}(\log^* n)$ возможно перекрывающихся отрезков.

Замечание. Новой структурой можно считать значение любой функции, которую мы умели считать с помощью обычного Sparse Table. Например, gcd.

Запомним: предподсчитали функцию на $\mathcal{O}(n)$ отрезках, используя $\mathcal{O}(n)$ памяти, любой отрезок $[l..r]$ можем разбить на $\mathcal{O}(\log^* n)$ возможно пересекающихся отрезков, на которых функция уже посчитана. Изменение делать нельзя. Таким образом мы научились считать любую ассоциативную коммутативную идемпотентную функцию на отрезке.

2.6. Алгоритм Фараха-Колтона-Бендера

Позволяет за $\mathcal{O}(n)$ сделать сведение `RMQ` $\to$ `LCA` $\to$ `RMQ±` и последнюю задачу решить за $\mathcal{O}(n)$ предподсчёта и $\mathcal{O}(1)$ на запрос. Подходит только для операции "минимум на отрезке". В дальнейшем нам не понадобится.

2.7. Аналог Sparse Table для суммы

Научимся предподсчитывать функцию на некоторых $\mathcal{O}(n \log n)$ отрезках таким образом, чтобы любой отрезок $[l..r]$ разбивался на два непересекающихся отрезка, на которых функция уже посчитана.

Идея. Пусть отрезок $[l..r]$ содержит точку $m = \frac{n}{2}$, тогда $[l..r] = [l..m]$+$(m..i]$. Давайте, предподсчитаем функцию на отрезках $\forall i \colon [i..m], (m..r]$. Таких отрезков ровно n. Как обработать отрезки, которые целиком справа/слева от точки m? Рекурсивно для отрезков $[1..m]$ и $(m..n]$ построить такую же структуру. Глубина рекурсии $\log n$, общее количество отрезков $\mathcal{O}(n \log n)$. Любой отрезок $[l..r]$ представляется в виде объединения двух отрезков.

Замечание #1. Можно использовать такое же улучшение, как и в Sparse Table и получить $\mathcal{O}(n \log\log n)$ отрезков и умение любой отрезок $[l..r]$ разбивать на $\mathcal{O}(1)$ непересекающихся, или $\mathcal{O}(n)$ отрезков и умение любой отрезок $[l..r]$ разбивать на $\mathcal{O}(\log^* n)$ непересекающихся.

Замечание #2. Структура, как и Sparse Table не допускает изменений исходного массива.

Зачем это нужно? Сумму мы и так умели считать с помощью частичных сумм. Но это только потому что сумма – обратимая функция. Минимум мы умели считать с помощью Sparse Table, потому что минимум – идемпотентная функция. Но бывают не обратимые не идемпотентные функции! Например, произведение по непростому модулю.

Количество различных чисел на отрезке

3.1. Формулировка задачи

Поступают запросы $[l..r]$, нужно говорить для каждого, сколько на отрезке $[l..r]$ исходного массива различных чисел.

3.2. Решение задачи

Для каждой ячейки массива i есть ближайшее справа число с таким же значеним: `next[i] > i`, `a[next[i]] == a[i]`. Чтобы посчитать количество различных чисел на отрезке $[l..r]$, нам нужно посчитать количество таких i: $l \le i \le r$ **и** $next_i > r$.

<u>Решение в offline:</u> переберём r в порядке возрастания, будем поддерживать множество таких i: $i \le r < next_i$

```
fill(pos, pos + M, -1); // число от 0 до M - 1
for (r = 0; r < n; r++) {
  if (pos[a[r]] != -1)
    delete(pos[a[r]]);
  add(r);
  pos[a[r]] = r;
  // в этот момент, чтобы ответить на запрос [l..r],
  // нужно для l посчитать количество элементов на суффиксе
  // это можно проще всего сделать деревом Фенвика
}
```

Получили решение за $\mathcal{O}((n+m)\log n)$, где n – длина массива, m – количество запросов.

<u>Решение в online:</u> в offline мы решили задачу сканирующей прямой с деревом Фенвика (или деревом отрезков). Сделаем дерево Фенвика (дерево отрезков) персистентным. Сохраним все версии. Ответ на запрос $[l..r]$ равен $\texttt{tree}_r\texttt{.get(l)}$.

Получили решение за $\mathcal{O}(n \log n)$ времени и памяти на предподсчёт и $\mathcal{O}(\log n)$ на запрос.

k-я порядковая статистика

> Говорите, была уже задача, где просили посчитать [...] на отрезке, да? А давайте тогда попросим посчитать k-е [...] на отрезке!
>
> Как придумывают задачи на структуры данных

4.1. Общие идеи

Общая идея поиска k-й порядковой статистики – бинарный поиск по ответу. Внутри бинарного поиска нужно быстро отвечать на запрос "количество $i: l \le i \le r$ и $a_i \le x$". Мы умеем отвечать на такой запрос за $\mathcal{O}(\log^2 n)$ в online деревом отрезков сортированных массивов, умеем отвечать за $\mathcal{O}(\log n)$ структурой данных, получаемой проходом сканирующей прямой с персистентным деревом отрезков. Первое решение сразу даёт решение задачи за $\mathcal{O}(\log^3 n)$, второе решение решает задачу за $\mathcal{O}(\log^2 n)$

4.2. Решение за логарифм n для неменяющегося массива

Очень кратко. Запустили сканирующую прямую с персистентным деревом отрезков, получили набор деревьев. $tree_r$ – дерево отрезков с операцией сумма, которое умеет за $\mathcal{O}(\log n)$ отвечать на запрос $get(x, y)$ "количество $i \le r: x \le a_i \le y$". Чтобы ответить на запрос "k-я порядковая статистика на отрезке $[l..r]$", берём $a = tree_r$, $b = tree_{l-1}$ и начинаем (вместо бинарного поиска!) параллельный спуск по этим двум деревьям. Пусть диапазон значений $[0..m]$, тогда $z = (a.l.value - b.l.value)$ — количество чисел на отрезке $[l..r]$ со значением в $[0..\frac{m}{2}]$. Если это число хотя бы k, делаем переход $a \to a.l; b \to b.l$, иначе уменьшаем k на z и делаем переход $a \to a.r; b \to b.r$

4.3. А теперь массив меняется

Идея с персистентным деревом и сканирующей прямой не обощается, так как эта структура не допускает изменений.

Зато идея с деревом отрезков сортированных массивов отлично обобщается. Давайте сортированный массив заменим на деркартово дерево.

Решение за $\mathcal{O}(\log^3 n)$: бинарный поиск по ответу, а внутри запрос к дереву отрезков декартовых деревьев. Заметим, что в данном случае вместо декартова дерево можно использовать `tree<int,...>`.

Решение за $\mathcal{O}(\log^2 n)$: можно бинарный поиска заменить на параллельный спуск по $\mathcal{O}(\log n)$ деревьям. Дерево отрезков разделило отрезок $[l..r]$ на $\mathcal{O}(\log n)$ вершин дерева отрезков. В каждой у нас хранится декартово дерево... давайте, вместо декартова дерева использовать "динамическое дерево отрезков" (дерево отрезков по диапазону $[0..M]$, которое использует на $\mathcal{O}(M)$ памяти, а $\mathcal{O}(n \log M)$, где n – количество элементов внутри дерева. По деревьям отрезков мы умеем спускаться параллельно! Делается также, как в предыдущей главе, только теперь деревьев не 2, а $\mathcal{O}(\log n)$.

Получили структуру данных, которая использует $\mathcal{O}(n \log n \log M)$ памяти и отвечает на запрос за $\mathcal{O}(\log n \log M)$.

4.4. Применение новой структуры

Только что мы отвечали на запрос "k-я статистика на отрезке" деревом отрезков, в каждой вершине которого хранится другое дерево. Мы теперь умеем разбивать отрезок $[l..r]$ на 2 непересекающихся куска. Если использовать эту идею, и для каждого из $\mathcal{O}(n \log n)$ отрезков предподсчитать "динамическое дерево отрезков по значаниям с операцией сумма", то мы получим структуру, использующую $\mathcal{O}(n \log M)$ памяти, и отвечающую за $\mathcal{O}(\log M)$ на запрос.

КОНЕЦ
