added: 11 Jun 2008 10:54
edited: 14 Jan 2013 17:58

Sqrt-decomposition

Sqrt-decomposition is a method or data structure that allows you to perform some typical operations (summation of the elements of the subarray, finding the minimum / maximum, etc.) for $O(\sqrt{n})$, which is much faster than $O(n)$ for the trivial algorithm.

First, we will describe the data structure for one of the simplest applications of this idea, then we show how to generalize it for solving some other problems, and, finally, consider a slightly different application of this idea: splitting input requests into sqrt-blocks.

Contents [hide]

- Sqrt-decomposition
  - Data structure based on sqrt-decomposition
    - Description
    - Implementation
    - Other tasks
  - Sqrt-decomposition of input requests
    - Example task: adding on a line
    - Task example: disjoint-set-union with partition
  - Offline tasks for queries on sub-drills of the array and the universal sqrt-heuristics for them

Data structure based on sqrt-decomposition

Let's put the problem . Given an array $a[0 \dots n-1]$. It is required to implement a data structure that can find the sum of elements $a[l \dots r]$ for arbitrary l and r for $O(\sqrt{n})$ operations.

Description

The main idea of sqrt-decomposition is that we do the following **pre-requisite** : we divide the array a into blocks of length approximately $\sqrt{n}$, and in each block we i pre-calculate the sum of the $b[i]$ elements in it.

We can assume that the length of one block and the number of blocks are equal to the same number - the root of n, rounded up:

$$s = \lceil \sqrt{n} \rceil ,$$

then the array is $a[]$ divided into blocks in this way:

$$\underbrace{a[0]\ a[1]\ \dots\ a[s-1]}_{b[0]}\ \ \underbrace{a[s]\ a[s+1]\ \dots\ a[2 \cdot s-1]}_{b[1]}\ \ \dots\ \ \underbrace{a[(s-1) \cdot s]\ \dots\ a[n]}_{b[s-1]}.$$

Although the last block can contain less than s, elements (if n not divided by s), this is not a matter of principle.

Thus, for each block, k we know the amount on it $b[k]$:

$$b[k] = \sum_{i=k \cdot s}^{\min(n-1,(k+1) \cdot s-1)} a[i].$$

So, let these values be b_k preliminarily calculated (for this we obviously need $O(n)$ operations). What can they give when calculating the answer to the next request (l, r)? Note that if the segment is $[l; r]$ long, then it will contain several blocks entirely, and for such blocks we can find out the amount on them in one operation. As a result, $[l; r]$ only two blocks remain from the whole segment , falling into it only partially, and on these pieces we will have to sum up the trivial algorithm.

Illustration (here through k denotes the number of the block in which lies l, and after p - the number of the block in which it lies r):

$$\dots\ \overbrace{a[l]\ \dots\ a[(k+1) \cdot s-1]\ \underbrace{a[(k+1) \cdot s]\ \dots\ a[(k+2) \cdot s-1]}_{b[k+1]}\ \dots\ \underbrace{a[(p-1) \cdot s]\ \dots\ a[p \cdot s-1]}_{b[p]}\ a[p \cdot s]\ \dots a_r}^{sum=?}\ \dots$$

This figure shows that in order to calculate the amount in a segment $[l \dots r]$, it is necessary to sum the elements in only two "tails": $[l \dots (k+1) \cdot s-1]$ and $[p \cdot s \dots r]$, and sum the values $b[i]$ in all blocks, starting with $k+1$ and ending with $p-1$:

$$\sum_{i=l}^{r} a[i] = \sum_{i=l}^{(k+1) \cdot s-1} a[i] + \sum_{i=k+1}^{p-1} b[i] + \sum_{i=p \cdot s}^{r} a[i]$$

(Note: this formula is wrong, when $k = p$: in this case, some of the elements to be added together twice, in which case you just have to sum elements l of r)

Thus, we save a significant number of transactions. Indeed, the size of each of the "tails" obviously does not exceed the length of the block s, and the number of blocks also does not exceed s. Since s we chose $\approx \sqrt{n}$, we only $[l \dots r]$ need $O(\sqrt{n})$ operations to calculate the sum on a segment .

Implementation

First we give the simplest realization:

```
// входные данные
int n;
vector<int> a (n);

// предпосчёт
int len = (int) sqrt (n + .0) + 1; // и размер блока, и количество блоков
vector<int> b (len);
for (int i=0; i<n; ++i)
        b[i / len] += a[i];

// ответ на запросы
for (;;) {
        int l, r; // считываем входные данные - очередной запрос
        int sum = 0;
        for (int i=l; i<=r; )
```

```
                if (i % len == 0 && i + len - 1 <= r) {
                        // если i указывает на начало блока, целиком лежащего в [l;r]
                        sum += b[i / len];
                        i += len;
                }
                else {
                        sum += a[i];
                        ++i;
                }
}
```

The disadvantage of this implementation is that it is unreasonable for many division operations (which, as is known, are performed much more slowly than other operations). Instead, you can count the block numbers c_l and c_r, in which the boundaries lie l, r respectively, and then loop through the blocks with $c_l + 1$ by $c_r - 1$ separately processing the "tails" in blocks c_l and c_r. In addition, with this implementation, the case $c_l = c_r$ becomes special and requires separate processing:

```
int sum = 0;
int c_l = l / len,   c_r = r / len;
if (c_l == c_r)
        for (int i=l; i<=r; ++i)
                sum += a[i];
else {
        for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)
                sum += a[i];
        for (int i=c_l+1; i<=c_r-1; ++i)
                sum += b[i];
        for (int i=c_r*len; i<=r; ++i)
                sum += a[i];
}
```

Other tasks

We considered the problem of finding the sum of the elements of an array in some sort of sub-trip. This task can be slightly extended: we also allow to **change the** individual elements of the array A. Indeed, if an element changes a_i, then it is enough to update the value $b[k]$ in the block in which this element is located ($k = i/len$):

$$b[k] += a[i] - old\_a[i].$$

On the other hand, instead of the problem of the sum, we can similarly solve the problem of the **minimal, maximal** elements in an interval. If you allow changes in the individual elements in these tasks, you will also have to recalculate the value of b_k the block to which the element you want to change, but recalculate completely, by passing all the elements of the block for $O(len) = O(\sqrt{n})$ operations.

Similarly, sqrt-decomposition can be used for many **other** similar tasks: finding the number of zero elements, the first non-zero element, counting the number of certain elements, etc.

There is also a whole class of tasks when there are **changes in the elements on the whole subsegment** : the addition or assignment of elements on some subsegment of the array A.

For example, you need to perform the following two types of queries: add a $[l; r]$ value to all elements of a certain interval δ, and learn the value of a single element a_i. Then, in quality, b_k let us set the value that should be added to all the elements of the k-th block (for example, everything from the beginning $b_k = 0$); then when you run the "add" query, you will need to add to all the elements of the a_i "tails", and then perform the addition to all elements b_i for the blocks that lie entirely in the segment $[l \dots r]$. And the answer to the second request will obviously be just $a_i + b_k$ where $k = i/len$. Thus, the addition on the segment will be performed for $O(\sqrt{n})$, and the request for a single element will be performed $O(1)$.

Finally, you can combine both types of tasks: changing elements on a line and responding to queries also on a line. Both types of operations will be performed for $O(\sqrt{n})$. To do this, you will have to do two "block" arrays b and c: one - to ensure changes on the segment, the other - to respond to requests.

You can give an example of other tasks, to which you can apply sqrt-decomposition. For example, you can solve the problem of **maintaining a set of numbers** with the possibility of adding / deleting numbers, checking the number for belonging to the set, searching for the k-th order number. To solve this problem, you must store the numbers in sorted order, divided into several blocks by the $\sqrt{n}$ numbers in each. When adding or removing a number, it will be necessary to "rebalance" the blocks, moving the numbers from the beginning / end of one block to the beginning / end of the neighboring blocks.

Sqrt-decomposition of input requests

Consider now a completely different application of the idea of sqrt-decomposition.

Suppose that we have a task in which we are given some input data, and then we receive k commands / requests, each of which we have to give to handle and return a response. We consider the case when requests are requesting (not changing the state of the system, but only requesting some information), and modifying (that is, affecting the state of the system, originally specified by the input data).

The specific task can be quite complex, and its "honest" solution (which reads one request, processes it, changes the state of the system, and returns the answer) may be technically difficult or even not at the strength of the decisive one. On the other hand, the solution to the "offline" version of the problem, i.e. When there are no modifying operations, and there are only requesting queries, it often turns out to be much simpler. Suppose that we are **able to solve the "off-line" version of the** problem, i.e. build over time $B(n)$ some kind of data structure that can respond to requests, but does not know how to handle modifying queries.

Then **we break the input requests into blocks** (for how long - until we specify, denote this length through s). At the beginning of the processing of each block, we will $B(n)$ build a data structure for the "offline" version of the task based on the state of the data at the beginning of this block.

Now we take turns taking requests from the current block and processing each of them. If the current request is modifying, then skip it. If the current request is a requestor, then we turn to the data structure for the offline version of the task, but beforehand **taking into account all the modifying queries in the current block** . Such a consideration of modifying queries is not always possible, and it should happen quickly enough - for a time $O(s)$ or a little worse; denote this time by $Q(s)$.

Thus, if all of our m requests, then their processing will take $B(m)\frac{m}{s} + mQ(s)$ time. The value s should be chosen based on the specific type of function $B()$ u $Q()$. For example, if $B(m) = O(m)$ u $Q(s) = O(s)$, then the optimal choice is $s \approx \sqrt{m}$, and the final asymptotics will be obtained $O(m\sqrt{m})$.

Since the above reasoning is too abstract, here are some examples of tasks to which such sqrt-decomposition is applicable.

Example task: adding on a line

Condition of the problem: an array of numbers is given $a[1 \dots n]$, and there are two types of queries: find the value in the i-th element of the array, and add some number $x$to all elements of the array in a certain interval $a[l \dots r]$.

Although this task can be solved without this reception with the partitioning of queries into blocks, we will bring it here - as a simple and visual application of this method.

So, let's break the input requests into blocks by $\sqrt{m}$(where $m$is the number of requests). At the beginning of the first block of queries, we do not need to build any structures, just store the array $a[]$. We now go on the requests of the first block. If the current query is an add request, we skip it. If the current query is a request to read the value in a certain position i, then at the beginning, simply take the value as the answer $a[i]$. Then, let's go through all the requests of addition added in this block, and for those of them that fall into i, we apply their increases to the current answer.

Thus, we learned to respond to requesting requests in time $O(\sqrt{m})$.

It remains only to note that at the end of each block of requests we must apply all modifying requests of this block to the array $a[]$. But this is easy to do for $O(n)$- enough for each addition request (l, r, x)note in the sub-array at the point $l$number $x$and the point $r + 1$- a number $-x$, and then walk through the array, adding a running total of the array $a[]$.

Thus, the final asymptotics of the solution is $O(\sqrt{m}(n + m))$.

Task example: disjoint-set-union with partition

There is an undirected graph with $n$vertices and $m$edges. There are three kinds of requests: add an edge (x_i, y_i), remove an edge (x_i, y_i), and check whether or not the vertex $x_i$and the $y_i$path are connected .

If there were no delete requests, then the solution of the problem would be a known data structure disjoint-set-union (a system of disjoint sets) . However, in the presence of deletions, the task becomes much more complicated.

We proceed as follows. At the beginning of each block of queries, let's see which edges in this block will be deleted, and immediately **remove** them from the graph. Now we construct a system of disjoint sets (dsu) on the resulting graph.

How do we now respond to the next request from the current block? Our system of disjoint sets "knows" about all edges except those that are added / deleted in the current block. However, we do not need to delete from dsu - we removed all such edges from the graph in advance. Thus, all that can be is additional, added edges, which can be a maximum of $\sqrt{m}$pieces.

Therefore, when answering the current requesting query, we can simply start a crawl in width along the connectivity components dsu, which will work for $O(\sqrt{m})$, since we only have $O(\sqrt{m})$edges in our consideration .

Offline tasks for queries on sub-drills of the array and the universal sqrt-heuristics for them

Consider another interesting variation of the idea of sqrt-decomposition.

Let's have some problem, in which there is an array of numbers, and request queries that have the form (l, r)- learn something about the sub-trip $a[l \dots r]$. We believe that requests are not modifying, and are known to us in advance, i.e. the task is offline.

Finally, we introduce the last **restriction** : we believe that we are able to quickly recalculate the response to a query when the left or right boundary changes by one. Those. if we knew the answer to the query (l, r), we can quickly calculate the answer to the query $(l + 1, r)$or $(l - 1, r)$or $(l, r + 1)$or $(l, r - 1)$.

We now describe the **universal heuristic** for all such problems. Sort the requests pair: $(l \operatorname{div} \sqrt{n}, r)$. Those. we sorted queries by the number of the sqrt-block, in which the left end lies, and at equality - on the right end.

Рассмотрим теперь группу запросов с одинаковым значением $l \operatorname{div} \sqrt{n}$ и будем обрабатывать все запросы этой группы. Ответ на первый запрос посчитаем тривиальным образом. Каждый следующий запрос будем считать на основе предыдущего ответа: т.е. двигать левую и правую границы предыдущего запроса к границам следующего запроса, поддерживая при этом текущий ответ. Оценим асимптотику: левая граница каждый раз могла двигаться на не более $\sqrt{n}$ раз, а правая — не более n раз в сумме по всем запросам текущей группы. Итого, если текущая группа состояла из k запросов, в сумме будет совершено не более $n + k \cdot \sqrt{n}$ пересчётов. В сумме по всему алгоритму получится — $O((n + m) \cdot \sqrt{n})$ пересчётов.

A simple **example of** this heuristic is the task: to find out the number of different numbers in a segment of an array $[l; r]$.

A slightly more complicated version of this problem is the task from one of the Codeforces rounds .

так как тут не указаны примеры задач дам одну ссылку с тимуса.
http://acm.timus.ru/problem...
6 • Ответить • Поделиться ›

Rostislav Kislenko ➚ ibra • 3 года назад
решается сортировкой, вообще не в тему
1 • Ответить • Поделиться ›

Stramp ➚ ibra • 6 лет назад
+1, присоединяюсь к вопросу :)
• Ответить • Поделиться ›

Андрей Меньков • 3 года назад
На этом изображении ошибка:

последний выделенный блок должен быть b[p - 1], а не b[p].
2 • Ответить • Поделиться ›

Prestige • 6 лет назад
Что-то я не пойму как использовать sqrt-декомпозицию в задаче 1613 на тимусе, расскажите как
2 • Ответить • Поделиться ›

ibra ➚ Prestige • 6 лет назад
Ну просто делим весь массив на блоки. Каждый блок сортируем. Теперь чтобы узнать есть ли на промежутке число, в блоках которые полностью попали в промежутки используем стандартную функцию binary_search, а для блоков попавших частично, используем линейный поиск в попавшей части.
• Ответить • Поделиться ›

programmer ➚ ibra • 6 лет назад
Можете код написать, чтоб понятней было
• Ответить • Поделиться ›

Dauren ➚ Prestige • 6 лет назад
Делим массив на sqrt(N) блоков, и в каждом блоке храним hash_table.
• Ответить • Поделиться ›

Helena • 5 лет назад
Расскажите, пожалуйста, подробнее, как можно реализовать запрос присвоения отрезка с одновременной реазизацией запроса суммы на отрезке? Пыталась сделать на основе прибавления, но не получилось.
Воникает вопрос, что должен хранить массив с и как его содержимое связать с начальным массивом a? И как должен измениться массив с при изменении одиночного элемента?

2 • Ответить • Поделиться ›

shweta singh • 2 года назад
Не совсем понятно по примеру disjoint-set-union с разделением. Если мы сразу, в начале блока, удалили ребра, то на момент запроса на связность dsu может состоять из большего числа компонент, чем должно быть на самом деле, и как тогда обход в ширину сможет связать две вершины. То есть допустим, в блоке сперва идет запрос на связность (u, v), а потом запрос удаления и, причем, это удаление приводит к несвязности (u, v). Применив это удаление на входе в блок, мы получим в dsu две разные компоненты связности для этих вершин. Как обход в ширину здесь учтет это ?
• Ответить • Поделиться ›

Aleksei Preobrazhenskii ➚ shweta singh • год назад
Рассмотрим блок из q запросов, пусть G - состояние графа связности на момент начала блока запросов, назовем ребро графа G - "фантомным", если оно затрагивается в блоке запросов, рассмотрим граф G \ F, где F - множество "фантомных" ребер, построим dsu(G \ F) за O(n + m) (например, пойдем по всем запросам предыдущих блоков в обратном порядке, исполняя только операции добавления, если до этого не встретили операцию удаления для этого ребра, можем использовать хэш-таблицу для пометки удаленных ребер), теперь важное наблюдение, что реберная структура подможеств dsu(G \ F) не затрагивается ни одним запросом из блока, то есть мы можем рассматривать подмножества как мета-вершины, между которыми могут быть добавляться или удаляться ребра, количество которых в худшем случае

равно q. Ну и теперь просто просимулируем q запросов, добавление ребра (u, v) - тривиально добавляем ребро в списки инцидентности вершин meta(u) и meta(v) за O(1), удаление ребра - тривиальное удаление из списков за O(q) (потому что в списках находится O(q) ребер), проверка (u, v) на принадлежность одной компоненте - либо meta(u) == meta(v), либо существует путь от meta(u) до meta(v) по ребрам блока, проверка существования пути осуществляется за O(q) поиском в ширину. То есть для блока из q запросов мы тратим O(n + m) на инициализацию и потом на каждый запрос мы тратим O(q), то есть суммарно мы тратим O(n + m + q * q) на блок из q запросов, таких блоков примерно k = m / q, то есть суммарная сложность O((m / q) * (n + m + q * q)), при выборе q = sqrt(m), сложность будет O((n + m) * sqrt(m)).

Ответить • Поделиться ›

Nedoluzhko Andrey • 3 года назад

Можно ли каким-то образом реализовать запрос максимума на отрезке с одновременным прибавлением на отрезке с помощью корневой декомпозиции (с асимптотикой N*sqrt(N), естественно), или же без дерева отрезков не обойтись?

Ответить • Поделиться ›

Сергей Егоров ➚ Nedoluzhko Andrey • 3 года назад

Можно и sqrt-декомпозицией, и деревом отрезков. Только асимптотика будет разная.

Ответить • Поделиться ›

Kassido • 4 года назад

please can you write about reverse massive from possition l to r with sqrt-decomposition
i know russian language just i have mistake in grammer, sorry for english:)

Ответить • Поделиться ›

Anton • 5 лет назад

Не совсем понятно по примеру disjoint-set-union с разделением. Если мы сразу, в начале блока, удалили ребра, то на момент запроса на связность dsu может состоять из большего числа компонент, чем должно быть на самом деле, и как тогда обход в ширину сможет связать две вершины. То есть допустим, в блоке сперва идет запрос на связность (u, v), а потом запрос удаления и, причем, это удаление приводит к несвязности (u, v). Применив это удаление на входе в блок, мы получим в dsu две разные компоненты связности для этих вершин. Как обход в ширину здесь учтет это ?

Ответить • Поделиться ›

shweta singh ➚ Anton • 2 года назад

я тоже не понимаю эту проблему. Вы получите ответ ?? то пожалуйста, помогите мне тоже

Ответить • Поделиться ›