MIPT, spring camp 2016, day #2
Theme: sqrt-decomposition

March 28, Sergey Kopeliovich

Contents

1. Sqrt decomposition on tree

Consider a tree of $n \leq 10^5$ vertices. We have to perform two types of queries:

1. `add(v,x)` – all neighbours of the vertex `v` will get `+x`.
2. `get(v)` – what is value of the vertex `v`?

It's easy to solve the problem in time $\mathcal{O}(\sqrt{n})$ per query. Lets call vertex *light* if its degree is less than $\sqrt{n}$, in other case vertex is *heavy*. How to maintain `add(v,x)`? If `v` is light, it's easy. If `v` is heavy, lets store `x` into `bonus[v]` – the value we have to add to all the neighbours of `v`. How to maintain `get(v)`? Lets take value of the vertex and add `bonus[i]` for every heavy neighbour `i` of the `v`.
The trick is "*there are no more than* $2\sqrt{n}$ *heavy vertices in any tree*".

• **Exercise.** Calculate number of triangles in directed graph in time $\mathcal{O}(E\sqrt{V})$.

2. Sqrt decomposition on strings

Consider searching substrings $s_1, s_2, \ldots, s_k$ of equal length l in the string t. For each i we are interested, if s_i is a substring of t. We can solve this problem in $\mathcal{O}(lk + |t|)$ time, using hash table of polynomial hashes of the strings s_i. Here you may ask, why don't we use Aho-Corasic? Lets imagine, we just do not know this algorithm, but already heard about hashes and hash tables.

Let we have strings of arbitrary lengths. How to use our solution for previous problem? The idea of sqrt decomposition helps. Lets denote summary length of all strings s_i as L then we may iterate all small lengths (less than $\sqrt{L}$) in time $\mathcal{O}(L + |t|\sqrt{L})$. We have at most $\mathcal{O}(\sqrt{L})$ big strings of length at least $\sqrt{L}$, so summary working time of the solution "*group the strings by its length and perform each length in linear time*" is also $\mathcal{O}(L + |t|\sqrt{L})$.

• **Exercise.** You are given text t and dictionary D. Check, if text is a concatenation of words from the dictionary. If $L = \max(|t|, \sum_{w \in D} |w|)$ then there is solution in time $\mathcal{O}(L\sqrt{L})$.

3. Sqrt decomposition on array

Consider an abstract problem "*we have an array «a» and we want to perform many different hard queries on its subsegments*". Lets start with the simplest problem. Query #1: sum on the range. Query #2: change $a[i]$. Lets denote as $[f(n), g(n)]$ data structure which can perform the first type queries in time $f(n)$ and can perform queries of the second type in time $g(n)$. For example, range tree as well as Fenwick tree is $[\mathcal{O}(\log n), \mathcal{O}(\log n)]$ data structure. Below we'll describe $[\mathcal{O}(1), \mathcal{O}(\sqrt{n})]$ and $[\mathcal{O}(\sqrt{n}), \mathcal{O}(1)]$ data structures. Denote $k = \lfloor \sqrt{n} \rfloor$.

Solution #0. Note, we may calculate partial sums of initial array `sum[i+1]=sum[i]+a[i]`, sum on the range $[l, r]$ is equal to `sum[r+1]-sum[l]`, and to change any $a[i]$ we recalculate all the array `sum[]`. Another way is do not precompute anything, we may calculate the sum of the range in linear time. These are solutions in $[\mathcal{O}(n), \mathcal{O}(1)]$ and $[\mathcal{O}(1), \mathcal{O}(n)]$.

Solution #1 in $[\mathcal{O}(k), \mathcal{O}(1)]$. Lets maintain array s, $s[i]$ is equal to sum of $a[j]$ for $j \in [ki..k(i+1))$. Query "*sum on the range*": the range can be viewed as head + body + tail, where body consists of big parts whose sums we already know. Head and tail are not longer than k.
Query "*change* $a[i]$": `set(i,x) { s[i/k] += x-a[i]; a[i] = x; }`

Solution #2 in $[\mathcal{O}(1), \mathcal{O}(k)]$. Lets maintain the same array s, and partial sums on it. Lets also maintain partial sums for each range $[ki..k(i+1))$. To change $a[i]$ we have to fully recalculate two arrays of partial sums (sums of small part, sums of s). To get sum on the range, we may view it as head + body + tail. For each of three parts we may get the sum in $\mathcal{O}(1)$ time.

Solution #3 in $[\mathcal{O}(k), \mathcal{O}(k)]$. Lets maintain partial sums `sum[i+1]=sum[i]+a[i]` and array of changes, which have been applied to array since partial sums were calculated. Denote with array as `Changes`. One change is pair $\langle i, x \rangle$. It denotes operation `a[i]+=x`. Lets maintain property $|\texttt{Changes}| \le k$. Query "*sum on the range*": sum on the range $[l, r]$ in initial array was equal to `sum[r+1]-sum[l]`, in $\mathcal{O}(|\texttt{Changes}|)$ time we may calculate, how much it was changed. Query "*change* $a[i]$": to set $a[i] := x$, lets add to `Changes` the pair $\langle i, x - a[i] \rangle$, and put x into $a[i]$. If now $|\texttt{Changes}| > k$ then build partial sums of current version of the array in time $\mathcal{O}(n)$, and clear the list `Changes`. Lets denote this operation *rebuild*. Note we'll call *rebuild* not so often. One time per k queries. So amortized time of performing one query is $\mathcal{O}(\frac{n}{k}) = \mathcal{O}(\sqrt{n})$.

Last approach we'll call "*delayed operations*" or "*sqrt decomposition on queries*". This approach has no advantages solving this task. But it will be useful later.

- **Exercise.** Queries: `set(l,r,x)` – set all the range. `sum(l,r)` – sum on the range.

4. Sqrt decomposition on array: split & rebuild

Lets solve more complicated task: we have four operations with the array

1. *Insert(i, x)* – insert x on i-th position.
2. *Erase(i)* – erase i-th element of the array.
3. *Sum(l,r,x)* – calculate sum of elements greater than x on the range $[l, r]$.
4. *Reverse(l,r)* – reverse the range $[l, r]$.

At first, imagine, we have only queries *Sum(0,n-1,x)*. Notice, it is not query on the range, it is query on whole array. Then we will maintain sorted array and partial sums on it. To answer the query lets do binary search and get sum on the suffix. The time to build the structure is $\mathcal{O}(n \log n)$ (do sort), the time to answer one query is $\mathcal{O}(\log n)$ (do binary search).

Lets solve full version of the problem. Main idea: in any moment we store our array, splitted into some parts. For every part the structure described above is built. To do anything on range $[l, r]$, lets do *Split(r+1)*, *Split(l)*. After it range $[l, r]$ is union of some parts. If amount of parts became extermly big, rebuild whole our structure.

We have the array $a[0..n)$. We will maintain some partition of the array into ranges $T = [A_1, A_2, \ldots, A_m]$. For each range A_i we store two versions – initial array and sorted array with partial sums. We will maintain two properties: $\forall i\colon |A_i| \le \sqrt{n}$ and $m < 3\sqrt{n}$. Initially lets split the array into $k = \sqrt{n}$ ranges of length $\sqrt{n}$. For each of k ranges we'll call operation *build* which sorts the range and calculates partial sums. The time for each range is $\mathcal{O}(\sqrt{n} \log n)$, the total time is $\mathcal{O}(n \log n)$. Now lets describe the main operation *Split(i)*, which returns such j, that i-the element is the start of j-th range. If i is not a start of a range, find $A_j = [l, r)$, such that $l < i < r$, and split it into two ranges $B = [l, i)$ and $C = [i, r)$. For ranges B and C call *build*, we've got new partition of the array into ranges: $T' = [A_1, A_2, \ldots, A_{j-1}, B, C, A_{j+1}, \ldots, A_m]$. Time to perform one *Split(i)* is $\mathcal{O}(k) + \mathcal{O}(\texttt{build}(\frac{n}{k}))$, means if $k = \sqrt{n}$ time is $\mathcal{O}(\sqrt{n} \log n)$. Here we assume T stores not the ranges, but links to it, so to "*copy*" one range we need only $\mathcal{O}(1)$ of time. We have *Split(i)*. Now lets express all other operations in terms of *Split(i)*.

```
vector<Range*> T;
int Split( int i ) { ... }
void Insert(i, x) {
  a[n++] = x;
  int j = Split(i);
  T.insert(T.begin() + j, new Range(n-1, n));
}
void Erase(i) {
  int j = Split(i);
  split(i + 1);
  T.erase(T.begin() + j);
}
int Sum(l, r, x) { // [l, r]
```

```
  l = split(l), r = split(r + 1); // [l, r)
  int res = 0;
  while (l <= r)
    res += T[l++].get(x); // binary search and use partial sums
  return res;
}
```

To perform queries of type `Reverse`, we need to store additional flag "*is the range reversed*", implementation of *Split(i)* becomes bit more complicated, all other parts stay the same.

```
void Reverse(l, r) {
  l = split(l), r = split(r + 1);
  reverse(T + l, T + r)
  while (l <= r)
    T[l++].reversed ^= 1; // this range might be already "reversed"
}
```

We have the solution, which starts with $k = \sqrt{n}$ ranges, perform of each query may increase k by at most 2. After k queries will have at most $3\sqrt{n}$ parts, at this moment lets rebuild whole the structure in $\mathcal{O}(n \log n)$ time. Amortized time of one rebuild is $\frac{n \log n}{k} = \sqrt{n} \log n$. In total our solution can perform any query in amortized time $\mathcal{O}(\sqrt{n} \log n)$.

We may speed up the solution. Note, *Split(i)* may be done in linear time, more precisely $\mathcal{O}(k + \frac{n}{k})$, rebuild may be also done in linear time, in $\mathcal{O}(n)$. Let the number of ranges k is equal to $\sqrt{n / \log n}$, amortized time to perform one query of type *Sum* is $\mathcal{O}(S + G + R)$, where S – time of *Split(i)*, is equal to $\mathcal{O}(\frac{n}{k} + k)$, G – time of inner for-loop of function *Sum*, is equal to $\mathcal{O}(k \log n)$, B – amortized time per one rebuild, is equal to $\mathcal{O}(\frac{n}{k})$. In total we get $\mathcal{O}(\sqrt{nlogn})$ per query.

• **Exercise.** Queries:

`reverse(l,r)` – reverse the subrange.
`increaseUpTo(l,r,x)` – for all i in $[l, r]$ perform `a[i] = max(a[i], x)`.

5. Sqrt decomposition on queries

Example: extendable sorted array. The task is to maintain set of integers and process queries

1. `add(x)` – add number x to the set.
2. `count(l,r)` – count number of elements with value from l to r

If we had only `count(l,r)`, the solution is to sort array and then "`count(l,r) = upper_bound(r) - lower_bound(l)`". How to add new elements? Lets store these new elements outside of long sorted array, in special small sorted "*array of new elements*". Lets also merge these to parts each $\sqrt{n}$ time. Merging we may do in $\mathcal{O}(n + \sqrt{n}\log n)$ (sort the small part, merge parts). Our result: `time(add)` $= \mathcal{O}(\sqrt{n})$ amortized time. `add(x)` inserts into small part and one time per $\sqrt{n}$ queries merge parts in $\mathcal{O}(n)$; `time(count)` $= \mathcal{O}(\log n + \log n)$. `count(l,r)` does binary search by the big sorted array and by the small sorted array.

This solution has some variations:

1. Small array is not sorted. Then `time(add)` = `time(count)` $= \mathcal{O}(\sqrt{n})$.
2. Do not merge arrays, just resort all data each $\sqrt{n\log n}$ time.
   Then `time(add)` = `time(count)` $= \mathcal{O}(\sqrt{n\log n})$.

The second variation is worse, why we talk about? Because this result may be used for any structure `T` with interface `T.build()`, `T.get()`. Let $f(n) =$ `time(T.build)` $\geq n$, lets store new elements in unsorted array and each $\sqrt{f(n)}$ time. Then summary time of all builds is $\mathcal{O}(f(n)^{3/2})$ and each get operation works in time $\mathcal{O}(\sqrt{f(n)} +$ `time(T.get)`$)$.

• **Dynamic connectivity**

We have an undirected graph. We have to perform queries of three types:

1. Add edge
2. Delete edge
3. Check, if two vertices are connected.

Let number of queries is n, number of vertices is no more than n, than all the queries can be easily answered in $\mathcal{O}(n\sqrt{n})$ time in offline mode. How to process pack of $\sqrt{n}$ queries in $\mathcal{O}(n)$? Each query fixes set of edges and forms a graph.

1. Lets fix all the edges that are contained in all $\sqrt{n}$ graphs.
2. `dfs`: find components of the graph, formed by selected edges. For each vertex v we know $c[v]$ – number of its component.
3. Answer query(a,b): check, if $c[a]$ and $c[b]$ are in one component of graph formed, by new $\mathcal{O}(\sqrt{n})$ edges. Of course, we need $\mathcal{O}(\sqrt{n})$ of time to check it.

To check if a and b are in one component, you may use `dfs`. As usual. But `DSU` is faster `;-)`

• **Exercise.** Let we have a set of points on the plane. You have to add new points and answer `get(a,b)` $= \max_i(ax_i + by_i)$, in other words, the fartherst point along direction $\langle a, b\rangle$.

• **Exercise.** Queries on tree: minimum on path $[a, b]$, add new leaf with parent p.