MIPT, camp 2014, day #3

Theme: data structures sqrt-decomposition, semiplanes intersection

November 14, Sergey Kopeliovich

1 Sqrt decomposition

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 linear time, in $\mathcal{O}(lk + |t|)$ time, using hash table of polynomial hashes of the strings s. Here you may ask, why don't we use Aho-Corasic? Let 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 upgrade our solution to previous problem? The idea of sqrt decomposition helps. Let denote summary length of all strings s_i as L then we may iterate lengths less than $\sqrt{L}$ in time $\mathcal{O}(L + |t|\sqrt{L})$. Moreover we have at most $\mathcal{O}(\sqrt{L})$ strings of length at least $\sqrt{L}$, so summary work time of solution "group string by length and proceed each length in linear time" is also $\mathcal{O}(L + |t|\sqrt{L})$.

You may use this approach not only string problems. For example, consider problems about trees. Let we have tree of n vertices. It may contain a lot of vertices of degree less than $\sqrt{n}$, but there are at most $\sqrt{n}$ vertices of degree $\sqrt{n}$.

2 Sqrt decomposition on array

Consider an abstract problem "we have an array a and want to proceed many different hard queries on its subsegments". Let start with the simplest problem. Query #1: sum on the segment. Query #2: change $a[i]$. Let denote data structure which can proceed the first type queries in time $f(n)$ and can proceed queries of the second type in time $g(n)$ as $[f(n), g(n)]$. For example, segment 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 segment $[l, r]$ is equal to `sum[r+1]-sum[l]`, and during changing any $a[i]$ we may simply recalculate all the array `sum[]`. Another way is do not precompute anything, we may calculate sum on the segment in linear time. These are solutions in $[\mathcal{O}(n), \mathcal{O}(1)]$ and $[\mathcal{O}(1), \mathcal{O}(n)]$.

Solution #1 in $[\mathcal{O}(k), \mathcal{O}(1)]$. Let maintain array s, $s[i]$ is equal to sum of $a[j]$ for $j \in [ki..k(i+1))$. Query "sum on the segment": the segment can be viewed as head + body + tail, where body consist 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 $[\mathcal{O}(1), \mathcal{O}(k)]$. Let maintain the same array s, and partial sums on it. Moreover let maintain partial sums for each segment $[ki..k(i+1))$. 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 segment, 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 $[\mathcal{O}(k), \mathcal{O}(k)]$. Let 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`. Let maintain property $|Changes| \le k$. Query "sum on the segment": sum on the segment $[l, r]$ in initial array was equal to `sum[r+1]-sum[l]`, in $\mathcal{O}(|Changes|)$ time we may calculate, how much it was changed. Query "change $a[i]$": to set $a[i] := x$, let add to *Changes* the pair $\langle i, x - a[i] \rangle$, and put x into $a[i]$. If now $|Changes| > k$ then build partial sums of current version of the array in time $\mathcal{O}(n)$, and clear the list *Changes*. Let denote this operation *Rebuild*. Note we'll call *Rebuild* not so often. One time per k queries. So amortized time of proceeding one query is $\mathcal{O}(\frac{n}{k}) = \mathcal{O}(\sqrt{n})$.

Last approach we'll call *delayed operations*. This approach has no advantages solving this task. But it will be useful later.

3 Sqrt decomposition on array: split & rebuild

Let 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 segment $[l, r]$.
4. *Reverse(l,r)* – reverse the segment $[l, r]$.

To solve this problem, at first, let answer to query *Sum(l,r,x)* (the only query with some answer), applied to all the array, ie query *Sum(0,n-1,x)*. Let we have no other types of queries, only the queries of type *Sum*.

So we can easily maintain sorted array and partial sums on it. To get answer to query let do binary search and get sum on the suffix. The time to build the structure is $\mathcal{O}(n \log n)$, the time to answer one query is $\mathcal{O}(\log n)$.

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

```
vector<Segment*> T;
int Split( int i ) { ... }
```

```
void Insert(i, x) {
  a[n++] = x;
  int j = Split(i);
  T.insert(T.begin() + j, new Segment(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 proceed queries of type `Reverse`, we need to store additional flag "is the segment 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;
}
```

We have the solution, which starts with $k = \sqrt{n}$ segments, proceeding of each query may increase k by at most 2. After k queries it may happens, that $k \geq 3\sqrt{n}$ (three times bigger), at this moment let rebuild all 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 proceed any query in amortized time $\mathcal{O}(\sqrt{n} \log n)$.

We may speed up it. Note, *Split(i)* may be done in linear time, more precisely $\mathcal{O}(k + \frac{n}{k})$, rebuild may be also done in liner time, in $\mathcal{O}(n)$. So let number of segments k is equal to $\sqrt{n/\log n}$, amortized time to proceed 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.

4 Semiplanes intersection

Let solve one complicated problem acm.timus.ru:1390. The statement is: we stay on the plane at point $(0, 0)$. From time to time walls appear on the plane. The walls are segments do not containing $(0, 0)$. The walls may intersect each other. From time to time we launch the bullet. The bullet flies in a straight line, while does not touch a wall. Answer to launch-bullet-query (type = "bullet") is distance, which buller has flied (possibly infinity).

In total we'll learn to proceed n queries of arbitrary type in time $\mathcal{O}(n \log^2 n)$. But at first let try to understand how to approach this problem. Problem is complicated. Queries of different types in arbitrary order, walls may intersect... Let's try to simplify the task. Let there are some walls, but no new walls will appear. Moreover let all the walls are straight infinite lines.

Now the task is following: we have straight lines, which cut out convex polygon, containing point $(0,0)$, we need to proceed queries kind of by given ray from $(0,0)$ find intersection point of the ray and the border of the polygon. Solution to the task: let intersect all semiplanes in $\mathcal{O}(n \log n)$, we'll get an convex polygon. Then each query can made by binary search in $\mathcal{O}(\log n)$ time. Describe the solution more precisely. Let start from binary search. Take angle of any polygon's vertex and denote it by α. For each other vertex take angle from $[\alpha, \alpha + 2\pi)$, write down this numbers in ascending order, we'll get an array of angles, denote it a. To know, where the bullet flying into (x, y) will stop, let calculate its angle $\beta \in [\alpha, \alpha + 2\pi)$ and find (using binary search) such minimal i: $a[i] \le \beta < a[i+1]$. Finally we simply intersect the segment, formed by points i, $i+1$ with the ray (trace of the bullet). Time is $\mathcal{O}(\log n)$. Now let talk about semiplanes intersection. At first, let add semiplanes $-M \le x, x \le M, -M \le y, y \le M$ for some big M. Now we are sure, intersection is finite convex polygon. Sort all lines by its angle (angle is from 0 to 2π). If several lines have the same angle, let leave only one among them – the nearest to $(0,0)$. Then we'll add lines in such order into the stack. Before addition of new line, some top lines in stack may be popped out. Let two top lines on the stack are l_1 and l_2. The top line should be popped out, if point $p = intersect(l_1, l_2)$ lies in opposite side from new line than $(0,0)$. In the stack we obtain the answer, the polygon. To obtain whole polygon, let add all the lines twice in the same order. Now contents of the stack is head + the answer + tail. To cut off the tails consider points of intersection of adjacent lines on the stack and take any point, which meets twice. Between these two positions the answer lies. The time to build the structure is $\mathcal{O}(sort) + \mathcal{O}(n)$.

Complicate the problem: let some new walls (lines) appear. We may use approach of "delayed operations" and proceed each query of type "bullet" in time $O(\sqrt{n})$. If number of delayed operation becomes greater than $\sqrt{n}$, we call the function rebuild, which works in $\mathcal{O}(n)$, because old walls (lines) are already sorted, and new $\sqrt{n}$ items we may add in time $\mathcal{O}(\sqrt{n} \log n + n)$. In total time to proceed n queries is $\mathcal{O}(n\sqrt{n})$.

We may do it in other way. Let store vertices of the polygon in balanced search tree (before we stored it in sorted array). To add new line with direction vector $\vec{v}$ and normal $\vec{n}$ (direction of normal is out of $(0,0)$), we need find the vertex $\vec{p}$ of the polygon such that scalar product $\langle \vec{n}, \vec{p} \rangle$ is maximal possible. It can be done by binary search by sorted array or, if we have BST (and we have), by descent throw BST tree from root to leaves If the point $\vec{p}$ lies at the same side from the new line, as point $(0,0)$ does, we should do nothing. Else we have to erase point $\vec{p}$ and, possibly, some other adjacent points. Finally we have to add two new points instead of all deleted data. Let delete vertices in loop, until ordinary point is at the same side from the line as point $(0,0)$ does. Finally we add intersection of the line and the segment (process of deletion stops, it gives us two points – the last deleted, the first not deleted). So the time to add new line is $\mathcal{O}(\log n + k \log n)$, where k – number of already deleted points. Each point will be deleted only once, so in total we can proceed n queries in $\mathcal{O}(n \log n)$. In terms of speed this solution is much better than previous one. In terms of simplicity of implementation the last one is much shorter and simply.

Complicate the problem one more time: now we have not lines, but segments. At fist solve the task in offline (all queries are known in advance). Then we know set of all possible angles of all ends of all segments. Sort it and build on the obtained array segment tree. Each vertex corresponds to a certain range of angles. When we add new wall, the new wall is splitted into $\mathcal{O}(\log n)$ parts by our segment tree. In each of these vertices the wall lies exactly from left border two right border, means behaves like a line. In total: in each vertex of the segment tree we'll maintain semiplanes intersection. Note, to build semiplanes intersection inside one vertex of the segment tree, we need to build a polyline, not a polygon. This polyline passes from left ray to right one (we may assume, difference of angles is not greater than $\frac{\pi}{4}$). So procedure of building of semiplanes intersect can be much simplified in this case – it's enough to add each line only once. Also we do not need to give off the cycle. So. We have the segment tree. In each vertex of the segment tree semiplanes intersection lies. Query of type "bullet": angle of the bullet lies in $\log n$ ranges (vertices of the segment tree). In each of these vertices we'll ask inner structure in time $\mathcal{O}(\log n)$. Summary time is $\mathcal{O}(\log^2 n)$. Query of time "add new wall": split the wall into $\log n$ vertices of the segment tree. Add the "line" into each of the vertices. Summary working time is $\mathcal{O}(\log^2 n)$. Amount of used memory is $O(n \log n)$.

Now to obtain online solution, we make from our segment tree "dynamic segment tree" Initially segment tree consist of the only vertex – its root. Then during the query to segment tree, which is proceeding downwards, if we go into nonexistent vertex, we just create it. The process of downning stops, when length of the segment is less than ε.

We may also use "delayed operations" here. Then time to proceed query would be $\mathcal{O}(k + \log^2 n)$. Where k – number of delayed operations. The structure can be built in time $\mathcal{O}(n \log n)$: sort all the walls one time, then in each vertex of the segment tree intersect semiplanes in linear time. Optimal k is equal to $\sqrt{n \log n}$. In total summary time to proceed n queries is $(n \log n)^{3/2}$.