Data Structure I

More about The Competition

TLE? 10^8 integer operations, or 10^7 floating point operations, in a for loop, run in around one second for a typical desktop PC. Therefore,

- if $n = 20$, then exponential algorithm is safe;
- if $n = 100$, then $O(n^3)$ is safe;
- if $n = 1000$, then $O(n^2)$ is safe;
- if $n = 10^4$, then $n \cdot \text{polylog}(n)$ and $n\sqrt{n}$ are safe.
- if $n = 10^5$, then $n \log(n)$ is safe.
- if $n = 10^6$, then $O(n)$ is safe.
- if $n = 10^9$, then $O(\log(n)$ is safe.

No MLE in onsite contests. In C++, it is possible that memory error turns out to be WA, instead of RE!

Try to master a modern IDE that is mostly available in the contests. Eclipse is recommended. VS is not since Linux is the mainstream in the contests.

Suggested data structure problem sets

- `http://acm.timus.ru/problemset.aspx?space=1&tag=structure`.
- `https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=13&page=show_contest&contest=278`.
- `http://www.cnphp6.com/archives/36008` (Original by notonlysuccess).
- SPOJ GSS series.
- SPOJ QTREE seires.

Section 1

BST, learn to use STL

Learn to Use STL

The STL implements most basic data structures and algorithms. The reference can be found in the Internet, e.g. `http://en.cppreference.com/`.

The following is particularly useful.

- vector. Can generally be used as a (dynamic) array, which supports enlarging dynamically.
- priority_queue. As the name suggests, included in queue header file.
- sort. Reliable and fast implementation of sorting algorithm. included in algorithm header.
- binary_search, next_permutation, lower_bound, upper_bound etc. In algorithm header.
- set, map, multiset, multimap. Balanced search tree based set and map.

Binary Search Tree as Dictionary

The basic usage of BST is as dictionary[1]. In other words, it supports insertion, deletion and searching.

We can use STL map. It is a good place to learn the iterator, the [] operator of map.

[1] http://poj.org/problem?id=2418

Binary Search Tree: lower_bound, upper_bound

Question:
`http://acm.timus.ru/problem.aspx?space=1&num=1613`.

We are given n integers ($n < 70000$), $a_1, a_2, \ldots, a_n$. There are q queries ($q < 70000$), in each query, we are asked to report whether there is a number x apears in $\{a_i : l \leq i \leq r\}$, for the given l, r in the query.

We consider pairs (i, a_i), which also records the index for the given integers. We then define a partial ordering for (i, a_i)'s.
$(i, a_i) < (j, a_j)$, if and only if $a_i < a_j$, or $a_i = a_j$ but $i < j$.

We put pairs $\{(i, a_i)\}_i$ in a STL set. Then for a query (l, r, x), we find whether there is an element (i, a_i) in the set such that $a_i = x$, and $l \leq i \leq r$.

This can be done by finding the smallest element in the set that is at least (l, x), and then test if it satisfies the constraints. The lower_bound method in STL set is exactly what is needed here.

Section 2

Monotone Queue

Introduction

A queue is a linear list, such that elements can be inserted at one end only, and can be deleted at the other end only. We extend the queue to support the query of the minimum element in it.

We present a data structure that supports each of the insertion, deletion and minimum query operation in amortized $O(1)$ time[2].

[2] `http://poj.org/problem?id=2823`

Observation

We suppose each insertion is defined as a pair (i, k_i), where i is the index and k_i is the value (keyword).
Suppose now we use a normal queue Q', but look at how the minimum behaves.

Suppose we insert an element (n, k_n). We note that before deleting (n, k_n), the minimum cannot be less than k_n. Therefore, it is safe *not* to store $(n-1, k_{n-1}), (n-2, k_{n-2}), \ldots, (n-i, k_{n-i})$, such that $k_j \geq k_n$ for $1 \leq j \leq i$.

Hence, if we discard the non-necessary in this way, the keywords of elements in the queue is increasing, in terms of indices, and it is an important implication that the head element is the minimum.

Implementation

The implementation is very straightforward. We use an auxiliary queue Q as the backend.

If it is insertion (i, k_i), we keep on discarding the tail element, until

1. Q is empty, or
2. the tail element is smaller than k_i.

Then we append k_i to the tail.

If it is deletion the i-th element, then we discard the head element if i is equal to the index of the head element.

The minimum is always the head element.

Section 3

Block List

Block List

Suppose we are to maintain a list of at most n elements, and support the following operations.

1. Insert an element e after index i.
2. Delete an element at index i.
3. Apply a simple operation to elements from index i to index j. The simple operation can be (and not limited to) adding a number to the elements, or querying the sum / minimum / maximum of the elements.

The last operation is a range operation, which is non-trivial for a standard linear list such as arrays and linked lists.
We will introduce a data structure that works in time $O(\sqrt{n})$ for each of the operations.

Block List

For simplicity, we assume there is no insertion / deletion, and we focus on the range query[3]. We assume we are to support

1. Change all elements in an interval $[i, j]$ to be x.
2. Query the sum of elements in an interval $[i, j]$.

We partition the indices into $O(\sqrt{n})$ segments, with each segment of length L, where $L := \lceil \sqrt{n} \rceil$. For example, if $n = 20$, we partition the indices to $[1, 5]$, $[6, 10]$, $[11, 15]$ and $[16, 20]$.

[3] `http://poj.org/problem?id=3468`

Framework

The idea is to maintain segments individually, and when an operation on interval $[i, j]$ comes, we update / summarize the information in each segments that *intersect* $[i, j]$.
For each segment, we maintain several information.

1. The left and right endpoint for the segment, namely l and r.
2. The value of individual elements. Both array and linked list work.
3. The sum s of all the elements in the segment. This can be calculated in $O(L)$ time.
4. A boolean variable b that denotes whether the *whole* interval is to be updated, and if the variable is true, we also record y to be the common element to which all elements in the segment will be changed.

The last piece of information is important for efficiency. We will elaborate more.

Maintain Individual Segments

We implement the change and query operation for a segment.

Change. Suppose we are to change $[i, j]$ to x. If $i = l, j = r$, then we set the b to be true, and record $y := x$, $s := x * (r - l) + 1$. Otherwise, if b is true, then we use a brute force to set every element to be y and set b to be false, and then (whatever b is) we use brute force to change $[i, j]$ to be x, and calculate s.

Query. Suppose we are to answer query $[i, j]$. If $i = l, j = r$, then return s. Otherwise, if b is true then we first use a brute force to set every element to be y and set b to be false, and then (whatever b is) we use brute force to calculate the sum of $[i, j]$.

We note that both change and query operation is $O(1)$ is $i = l$ and $j = r$, and $O(L)$ otherwise.

Perform The Operations

According to our construction, there are at most two segments that can intersect $[i, j]$, and others represent sub-intervals of $[i, j]$ for any fixed $[i, j]$. For example, $[3, 17]$ partially intersects $[1, 5]$ and $[16, 20]$, and contains $[6, 10]$ and $[11, 15]$ as sub-intervals.

Suppose $S_1, S_2, \ldots, S_k$ are the consecutive segments that intersect $[i, j]$. To perform the operation on $[i, j]$ we perform the corresponding operation on S_i's. Observe that apart from S_1 and S_k, all other segments denote a subset of $[i, j]$, and the operation can be performed in $O(1)$ for each of them. Moreover, we can perform the operation on S_1 and S_k in time $O(L)$.

Therefore, the operation can be performed in time $O(1) \cdot O(\sqrt{n}) + O(L) = O(\sqrt{n})$.

Handle Insertion and Deletion

Now we turn to insertion and deletion[4].

We maintain the invariant that either each segment contains more than L elements and less than $3L$ elements, or there is only one segment and the segment contains less than $3L$ elements.

[4] http://poj.org/problem?id=3580

Insertion

We first locate the segment S where we insert the element. This takes $O(\sqrt{n})$.

We use brute force insertion on S, which takes $O(L)$. If the size of S reaches $3L$, then we break it into two segments evenly (both with $\frac{3}{2}L$ elements).

Deletion

We locate the segment S where we delete the element. This takes $O(\sqrt{n})$.

We use brute force deletion on S, which takes $O(L)$. If the size of S reaches L, and if S has an adjacent segment, we combine the two to form a new segment S'. The size of S' is more than $L + L = 2L$, and less than $L + 3L = 4L$. We then break S' evenly to two new segments, and they each contain more than L elements and less than $2L$ elements.