The blue path

Algorithms, algorithms and more algorithms...

Monday, September 03, 2012

Counting inversions in an array using Binary Indexed Tree

Introduction

Given an *array* A of N integers, an *inversion* of the *array* is *defined* as any pair of indexes (i, j) such that $i \leqslant j$ and $A[i] \geqslant A[j]$.

For example, the array $a = \{2, 3, 1, 5, 4\}$ has three inversions: $(1, 3)$, $(2, 3)$, $(4, 5)$, for the pairs of entries $(2, 1)$, $(3, 1)$, $(5, 4)$.

Traditionally the problem of counting the inversions in an array is solved by using a modified version of Merge Sort. In this article we are going to explain another approach using **B**inary **I**ndexed **T**ree (BIT, also known as Fenwick Tree). The benefit of this method is that once you understand its mechanics, can be easily extended to many other problems.

Prerequisite

This article assume that you have some basic knowledge of Binary Indexed Trees, if not please first refer to this tutorial.

Replacing the values of the array with indexes

Usually when we are implementing a BIT is necessarily to map the original values of the array to a new range with values between $[1, N]$, where N is the size of the array. This is due to the following reasons:

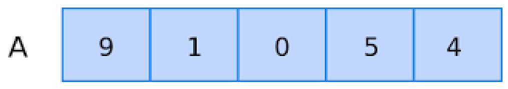
(1) The values in one or more $A[i]$ entry are too high or too low.
(e.g. 10^{12} or 10^{-12}).

For example imagine that we are given an array of 3 integers:
$\{1, 10^{12}, 5\}$
This means that if we want to construct a frequency table for our BIT data structure, we are going to need at least an array of 10^{12} elements. Believe me not a good idea...

(2) The values in one or more $A[i]$ entry are negative.
Because we are using arrays it's not possible to handle in our BIT frequency of negative values (e.g. we are not able to do $freq[-12]$).

A simple way to deal with this issues is to replace the original values of the target array for indexes that maintain its relative order.

For example, given the following array:



The first step is to make a copy of the original array A let's call it B. Then we proceed to sort B in non-descending order as follow:

Labels

ad hoc (2) algorithms (2) arithmetic progression (1) arithmetic sequence (1) bfs (1) binary indexed trees (2) binary search (2) BIT (2) brute force (8) C++ (2) codeforces (11) codejam (1) combinatorics (3) connected components (1) counting inversions (1) dfs (5) disjoint sets (4) divisors (1) dynamic programming (7) euler's formula (1) fenwick tree (1) fibonacci numbers (1) gcd (3) geometry (2) google (1) graph theory (5) greedy (7) implementation (6) infinite geometric series (1) lcm (3) math (19) number theory (7) perfect squares (1) planar graph (1) prime factors (3) probability (1) programming (1) programming contest (2) recursion (5) Segment tree (1) sieve of eratosthenes (2) simulation (2) sorting (3) strings (5) topcoder (2) tree (1) triangular numbers (2) trie (1) tries (1) two pointers (2) uva (12)

Blog Archive

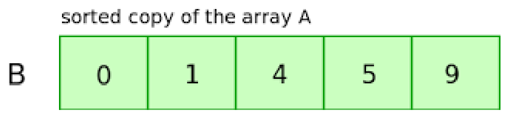
► 2013 (4)
▼ 2012 (29)
► November (2)
► October (2)
▼ September (2)
TopCoder SRM 556
Counting inversions in an array using Binary Index...
► August (3)
► July (11)
► June (7)
► May (2)
► 2011 (1)

About Me

Pavel Simo

View my complete profile

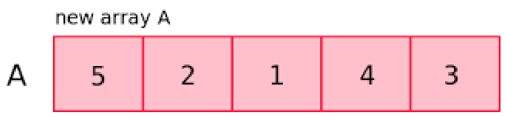
Followers



Using binary search over the array B we are going to seek for every element in the array A, and stored the resulting position indexes (1-based) in the new array A.

$binary\_search(B, 9) = 4$	found at position 4 of the array B
$binary\_search(B, 1) = 1$	found at position 1 of the array B
$binary\_search(B, 0) = 0$	found at position 0 of the array B
$binary\_search(B, 5) = 3$	found at position 3 of the array B
$binary\_search(B, 4) = 2$	found at position 2 of the array B

The resulting array after increment each position by one is the following:



The following C++ code fragment illustrate the ideas previously explained:

```cpp
for(int i = 0; i < N; ++i)
   B[i] = A[i]; // copy the content of array A to array B

sort(B, B + N); // sort array B

for(int i = 0; i < N; ++i) {
   int ind = int(lower_bound(B, B + N, A[i]) - B);
   A[i] = ind + 1;
}
```

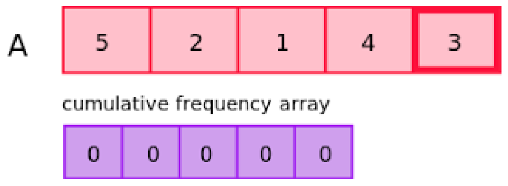
Counting inversions with the accumulate frequency

The idea to count the inversions with BIT is not to complicate, we start iterating our target array in reverse order, at each point we should ask ourselves the following question **"How many numbers less than $A[i]$ have already occurred in the array so far?"** this number corresponds to the total number of inversions beginning at some given index. For example consider the following array $\{3, 2, 1\}$ when we reach the element 3 we already seen two terms that are less than the number 3, which are 2 and 1. This means that the total number of inversions beginning at the term 3 is two.

Having this ideas in mind let's see how we can applied BIT to answer the previous question:

1. $\mathbf{read}(idx)$ - accumulate frequency from index 1 to idx
2. $\mathbf{update}(idx, val)$ - update the accumulate frequency at point idx and update the tree.
3. **cumulative frequency array** - this array represents the cumulative frequencies (e.g. $c[3] = f[1] + f[2] + f[3]$) , as a note to the reader this array is not used for the BIT, in this article we used as a way of illustrating the inner workings of this data structure.

Step 1: Initially the cumulative frequency table is empty, we start the process with the element 3, the last one in our array.

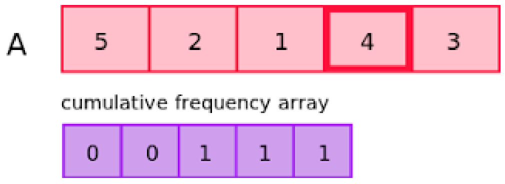


how many numbers less than 3 have we seen so far
$x = read(3 - 1)$ = 0
$inv\_counter = inv\_counter + x$

update the count of 3's so far

$update(3, +1)$
$inv\_counter = 0$

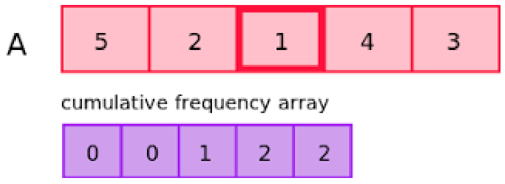
Step 2: The cumulative frequency of value 3 was increased in the previous step, this is why the $read(4-1)$ count the inversion $(4, 3)$.



how many numbers less than 4 have we seen so far
$x = read(4-1) = 1$
$inv\_counter = inv\_counter + x$

update the count of 4's so far
$update(4, +1)$
$inv\_counter = 1$

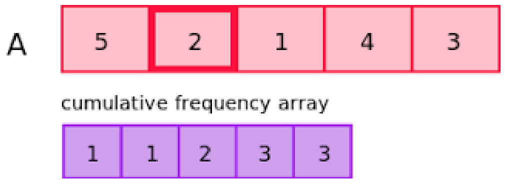
Step 3: The term 1 is the lowest in our array, this is why there is no inversions beginning at 1.



how many numbers less than 1 have we seen so far
$x = read(1-1) = 0$
$inv\_counter = inv\_counter + x$

update the count of 1's so far
$update(1, +1)$
$inv\_counter = 1$

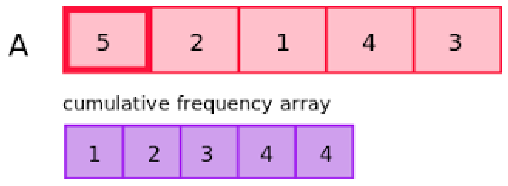
Step 4: Theres is only one inversion involving the value 2 and 1.



how many numbers less than 2 have we seen so far
$x = read(2-1) = 1$
$inv\_counter = inv\_counter + x$

update the count of 2's so far
$update(2, +1)$
$inv\_counter = 2$

Step 5: There are 4 inversions involving the term 5: $(5, 2)$, $(5, 1)$, $(5, 4)$ and $(5, 3)$.



how many numbers less than 5 have we seen so far
$x = read(5-1) = 4$
$inv\_counter = inv\_counter + x$

update the count of 5's so far
$update(5, +1)$
$inv\_counter = 6$

The total number of inversion in the array is 6.

The overall time complexity of this solution is $O(NlogN)$, the following code corresponds to a complete implementation of the ideas explained in this tutorial:

```
#include <algorithm>
#include <cstdio>
#include <cstring>

using namespace std;

typedef long long llong;

const int MAXN = 500020;
llong tree[MAXN], A[MAXN], B[MAXN];

llong read(int idx){
 llong sum = 0;
 while (idx > 0){
  sum += tree[idx];
  idx -= (idx & -idx);
 }
 return sum;
}

void update(int idx ,llong val){
 while (idx <= MAXN){
  tree[idx] += val;
  idx += (idx & -idx);
 }
}

int main(int argc, char *argv[]) {
   int N;
   while(1 == scanf("%d",&N)) {
      if(!N) break;
      memset(tree, 0, sizeof(tree));
      for(int i = 0; i < N; ++i) {
         scanf("%lld",&A[i]);
         B[i] = A[i];
      }
      sort(B, B + N);
      for(int i = 0; i < N; ++i) {
         int rank = int(lower_bound(B, B + N, A[i]) - B);
         A[i] = rank + 1;
      }
      llong inv_count = 0;
      for(int i = N - 1; i >= 0; --i) {
         llong x = read(A[i] - 1);
         inv_count += x;
         update(A[i], 1);
      }
      printf("%lld\n",inv_count);
   }
   return 0;
}
```

Related problems

UVa 10810 Ultra-QuickSort
UVa 11495 Bubbles and Buckets

SPOJ 6256 Inversion Count
Codeforces Beta Round #57 E. Enemy is weak

Posted by Pavel Simo at 11:02 PM

G+1 +2 Recommend this on Google

Labels: binary indexed trees, BIT, counting inversions, fenwick tree, sorting

10 comments:

arpit said...

Hi Povel,

Thanks for the wonderful Blog. I was trying the problem Coder ratings that u have mentioned in your blog.

I see there is a solution for it using a 2-d Bit . Is there any solution for it using a 1- d bit ?

11:20 PM

Juan Pablo said...

This comment has been removed by the author.

7:49 PM

Juan Pablo said...

Hey man thank you a lot! a beauty technique with a good explanation! Can you post something about BIT for query ranges and updates ranges? thanks! excelent post!

7:53 PM

Ganesh Ponraju said...

This comment has been removed by the author.

4:12 AM

Lee Wei said...

What if elements of A[] are identical? Will your lower_bound approach still work to convert element values to their respective array indices?

7:43 AM

Pavel Simo said...

@Lee Wei:
It should not be a problem.

For example given the array:
8 4 9 9 9 1 14

after sorting we have:

1 2 3 4 4 4 5 id
----
1 4 8 9 9 9 14 A[]

replacing the values by the id we get:
3 2 4 4 4 1 5

9:46 PM

zinj wang said...

can you please help me to solve below::
He had also asked you to answer M of his questions. Each question sounds like: "How many inversions will the array A contain, if we swap the elements at the i-th and the j-th positions?".
The inversion is such a pair of integers (i, j) that i < j and Ai > Aj.

like
Input:

```
6 3
1 4 3 3 2 5
1 1
1 3
2 5
```

Output:

```
5
6
0
```

3:00 PM

ChandraSekhar said...

awesome tutorial...thank you...really helped me...))

7:22 AM

bradd123 said...

Thank you for this excellent tutorial

1:29 AM

murdocc007 said...

Shouldn't the number of inversions be (idx-1)-read(idx-1) as read(idx-1) only gives the pairs such that i<=j and A[i]<=A[j]?

9:27 PM

Post a Comment

Newer Post Home Older Post