Similar Pair

by idlecool

All topics

Depth First Search

Depth First Search (popularly abbreviated as DFS) is a recursive graph searching technique.

The Algorithm
While doing a DFS, we maintain a set of visited nodes. Initially this set is empty.
When DFS is called on any vertex (say u), first that vertex is marked as visited and then for every edge going out of that vertex, (u, v), such that v is unvisited, we call DFS on v.
Finally, we return when we have exhausted all the edges going out from u.
Hence, if a graph is represented in the standard adjacency list form (a vector of vectors in C++), the C++ implementation follows:

```
void dfs(int node)  {
  seen[node] = true;
  for(int next: graph[node])  {
    if(not seen[next])  {
      dfs(next);
    }
  }
}
```

One may note that recursive calls imitate stack push and pop. Hence, DFS can also be implemented using stacks. In cases where the system stack size (which is used for recursion) is very limited, one might use a user-created stack to perform DFS within the memory limits.

The Complexity and Correctness
The above code provides a good angle to view at the complexity. Under no circumstance, DFS will be called twice on a node. For every node, we have iterations equal to the degree of that node. Hence the time complexity is simply sum of degrees of all the nodes which is $O(V + E)$.
If DFS is called on a node u, and there exists a path $u \rightarrow v$ then DFS will definitiely visit v.
This can be proven as an induction on the path length of $u \rightarrow v$ by observing that DFS indeed visits all nodes which have a direct edge from the first node and hence establishing the base case i.e. for path length = 1.

Applications
DFS is probably the most used graph-algorithm in programming contests. Many of the standard applications include - searching, counting connected components and their sizes, providing a useful way to number the vertices of a tree, finding bridges, finding euler path/tour and many others.
We'll discuss about numbering the vertices of a tree.
Assume that you have a rooted tree and called a DFS on the root. Now, consider any arbitrary call the DFS makes, say its on node u. Now, you can notice that DFS first visits all the nodes in u's subtree and only then visits any other node. Hence, if we number the nodes in the order they are visited, all the nodes in a particular subtree will come under a contigous range.
This kind of numbering is useful for questions which ask to modify all nodes in a particular node's subtree. Doing this converts the problem to a range modification one which are pretty standard.

Related challenge for **Depth First Search**

Cut the tree

Success Rate: 66.35% Max Score: 50 Difficulty:

Solve Challenge

Scroll to Top

Binary Indexed Tree

Binary Indexed Tree (BIT), in its vanilla form, is a data-structure which helps us to answer the following types of queries, on an array $A[1..n]$, efficiently:

1. $Set(i, val)$ - This adds val to $A[i]$
2. $Get(i)$ - This returns $\sum_{j=1}^{i} A[i]$

How to do that?
Well, lets forget about the updates first. Now, the Get can be efficiently handled with a simple prefix sum implementation. But if we have updates, we might need to change about $O(n)$ entries in our prefix sum, which is not desirable. The problem is that using prefix sum, we get the answer in $O(1)$ time, basically, each answer is calculated separately. Instead, if each answer depended on say some non-constant number of quantities (which preferably is sublinear in number denoted by k for now), we might aim to get the answer to each query in $O(k)$ time and update each quantity in $O(1)$ resulting in an $O(k)$ update time.

Turns out, we can achieve $k = O(log(n))$ using BIT.

For that, lets think about the binary representation of any number. Lets say our number is 101101 in binary.

Now, $101101 = 100000 + 1000 + 100 + 1$.

Hence, we can divide the interval $[1, 101101]$ into 4 parts of length 100000, 1000, 100 and 1 respectively.

Now, we maintain an array $B[1..n]$ where in the i^{th} element stores $\sum_{j=0}^{j<2^l} A[i-j]$ where l is the position of the right-most set bit of i where the unit's place position is considered to be 0.

For example:
$B[101101] = A[101101]$
$B[101100] = A[101100] + A[101011] + A[101010] + A[101001]$
$B[101000] = A[101000] + A[100111] + A[100110] + A[100101] + A[100100] + A[100011] + A[100010] + A[100001]$
and so on.

You can see that we have effectively done the division that we talked about.
Observe that now,
$Get(101101) = B[101101] + B[101100] + B[101000] + B[100000]$
We basically sum all the $B[j]$'s where j's are obtained by removing the rightmost set bit of i one by one.
Hence, we've reduced each query to a sum of at most $log_2(n)$ entities. If we can find a way to keep up with the updates, we'll have a functional data structure!
Now, consider we need to update the value of $A[101011]$. Which all entities of B have $A[101011]$'s information?
By construction, those will be the indices - $101011, 101100, 110000, 1000000$.
Basically, informally speaking, we are erasing a contigous series of 1's from the right and replacing it with a new 1 on the left of the leftmost 1 erased.
This is basically just the reverse of the query, and it should be no surprise since we need to access those indices j for which i appears as in element in the query of j.
So, again, we can have at-most $log_2(n)$ entities to change.
We have therefore found out a way to deal with the required operations in a really good complexity i.e. $O(log(n))$ for both update and query. :)

Implementation
First lets talk about queries. The big question here is that given an i, how do you efficiently find out the required indices to sum?
Let's say $i = 110010$.
We have $-i = 001101 + 1 = 001110$ i.e., the rightmost set bit remains intact while every other bit to the left of that bit is altered and every bit ot the right of it becomes 0.
Hence if we do $i\&(-i)$, we'll effectively get just the rightmost set bit of i.
So, $i\&(-i)$ for $i = 1001110$ will be 0000010, for $i = 11111$ will be 00001 and so on.
Hence, if we subtract this from i, we are effectively removing all the set bits of i starting from the rightmost set bit, one by one, which is exactly what we need to do. Hence , the code is simply:

```
// get cumulative sum up to and including i
int Get(int i) {
  int res = 0;
  while(i) {
    res += B[i];
    i -= (i & -i);
  }
  return res;
}
```

Coming to the update part, its just the reverse of the query. So the code will be:

```
// add val to value at i
void Set(int i, int val) {
  while(i <= N) {
    B[i] += val;
    i += (i & -i);
    }
  }
```

That's about it! Well if you are wondering why is it a tree? Then think that each index represents a node and if for some i, we get j by erasing its rightmost set bit, then j is the parent of i. Hence, we are effectively summing the path from a node to the root while querieng. :)

So, here's a standard question, you are given multiple updates and queries as described above, in addition to that, there is a second type of query - for some given value x, find the largest i such that $Get(i) \leq x$.
Also assume that $A[i] \geq 0$ for all i throughout the process.

To solve this, notice that $Get(i)$ is a non-decreasing function w.r.t i. Which means that we can do a binary search to get the largest i with a complexity of $O(log^2(n))$.