Square Root Decomposition

Square root decomposition is a special data structure that allows for O($\sqrt{n}$) interval query and O(1) update operations (update operation complexity depends on the problem!) of a given manner on a 1-D array. It works by 'splitting' the given array into $\sqrt{n}$ sized chunks.

Suppose we have the following array of numbers (indices start at 0):

```
1 4 2 10 99 60 13 24 8 9 55 69 96 12 14 18 43 7 8 22 42
```

Now also suppose we would like to be able to perform an interval query `Q(L,R)` where we return the sum of all numbers in the array between the two indices `L` and `R` inclusive, as well as an update operation `U(V,N)` where we change the `V`th index in the array to `N`. While this *is* a segment tree problem, we can also use square root decomposition to do it.

There are 21 numbers in our array, so we will split it into chunks of $\lfloor\sqrt{21}\rfloor = 4$ size (we will cut off the last chunk if it is too small). We then end up with:

```
1 4 2 10 | 99 60 13 24 | 8 9 55 69 | 96 12 14 18 | 43 7 8 22 | 42
```

We will now compute the sum of each chunk:

```
   17          196          141           140           80       42
1 4 2 10 | 99 60 13 24 | 8 9 55 69 | 96 12 14 18 | 43 7 8 22 | 42
```

Suppose we attempt to execute `Q(0,12)`. A naive implementation would simply add each number to get `1+4+2+10+99+60+13+24+8+9+55+69+96 = 450`. With square root decomposition, however, many parts of this sum are precomputed. We can simply add the first, second, and third chunks as well as the first number of the fourth chunk to compute the same sum, which comes out to a much less intensive operation of `17+196+141+96 = 450`. The interval query operation is performed in O($\sqrt{n}$) time, since there are $\sqrt{n}$ chunks and each chunk is at most $\sqrt{n}$ size.

To do update operations, only the element in the array and the sum of its chunk need to be changed. For example, performing `U(2,8)` means the 17 in the first chunk is changed to 23 and the 2 in the first chunk is changed to an 8. This is clearly an O(1) operation, and thus square root decomposition is illustrated with this example. A code implementation will be shown below.

Square root decomposition should be used when a segment tree is not possible (since segment trees tend to have better time complexities). Some updates are simply complex in a manner such that a segment tree is not possible. A good way to check if square root decomposition

should be used is to look at the problem bounds; if an O($Q\sqrt{n}$) time complexity is acceptable, and the 'form' of the problem is updates and queries, then it is time to think about using square root decomposition.

Input for this example code is as so: on the first line is N and Q, representing array size and number of operations respectively, followed by a line containing N integers representing the array and then Q lines of 3 integers *a,b,c* representing every operation. If *a* is 1, then the program will print out `Q(b,c)`, and if *a* is 2 then the program will perform `U(b,c)`.

```
#include <bits/stdc++.h>
using namespace std;

int arr[100000],sqd[320]; //arr is the initial array, and sqd will store the sums of 'chunks'
int N,Q,cs; //N is the size of the array, Q is the number of queries, cs is sqrt(N)

int query(int L, int R){
    int tot = 0;
    while(L % cs != 0 && L <= R){ //adding on the parts of the first chunk
        tot += arr[L];
        L++;
    }
    while(L + cs <= R){ //adding on all the middle chunks
        tot += sqd[L/cs];
        L += cs;
    }
    while(L <= R){ //adding on the parts of the last chunk
        tot += arr[L];
        L++;
    }
    return tot;
}

void update(int A, int B){
    sqd[A/cs] += B-arr[A];
    arr[A] = B;
    return;
}

int main(){
    freopen("input.txt","r",stdin);
    cin >> N >> Q;
    cs = sqrt(N); //since cs is an int, sqrt(N) will automagically round down
    for(int i = 0; i < N; i++){
        cin >> arr[i];
        sqd[i/cs] += arr[i]; //constructing chunk sums
    }
    for(int i = 0; i < Q; i++){
        int a,b,c;
        cin >> a >> b >> c;
        if(a == 1) cout << query(b,c) << endl;
        else update(b,c);
    }
}
```