AVS Tree

This problem was worth 2000 points.
The author of this problem is Aryan V S.

Note: GitHub does not support LaTex in Markdown. If you want a more readable version of the problem, download the PDF file instead.

Statement

Aryan was trying to create a fun but hard problem on AVL trees so that he could call the problem modification AVS Tree, and propose the problem idea to Dhruv and include it in the contest. It turned out that the problem is too hard (or maybe he's just terrible at problem solving). He'd like to believe the former and that AVL tree problem modifications are simply too hard to solve, and so he dropped the idea of having a problem on the same 😔

Instead, he decided to create a new data structure (still called AVS tree) that could efficiently process the following queries on an array a of n elements:

- "$neg\ L\ R$": perform $a_i = -a_i$ for $L \leq i \leq R$
- "$set\ X\ Y$": perform the operation $a_X = Y$
- "$sum\ L\ R$": output the value of $\sum_{i=L}^{R} a_i$

Aryan does not know how to implement such a data structure. It is easier to just decide to create a data structure than to implement it, right? Help Aryan implement such a data structure that efficiently processes q such queries.

Input Format

The first line contains two space separated integers n and q - the number of elements and number of queries.

The second line contains n space separated integers a_i - the initial elements of the array a.

The following q lines contain queries as described above.

Output Format

For each query of the type "$sum\ L\ R$", output a single integer - the property described in the statement.

Constraints

$1 \leq n, q \leq 2 \cdot 10^5$

$-10^9 \leq a_i,\ Y \leq 10^9$

$1 \leq X \leq n$

$1 \le L \le R \le n$

Sample Tests

Sample Test 1

Input

```
5 5
1 2 3 4 5
sum 1 5
neg 2 4
set 3 1
neg 1 5
sum 1 4
```

Output

```
15
4
```

Explanation

Initially the array is $[1,\ 2,\ 3,\ 4,\ 5]$.

For the first operation, we output the sum of values $[a_1,\ \ldots,\ a_5] = 15$.

After the second operation, the array becomes $[1,\ -2,\ -3,\ -4,\ 5]$.

After the third operation, the array becomes $[1,\ -2,\ 1,\ -4,\ 5]$.

After the fourth operation, the array becomes $[-1,\ 2,\ -1,\ 4,\ -5]$.

For the fifth operation, we output the sum of values $[a_1,\ \ldots,\ a_4] = 4$.

Sample Test 2

Input

```
8 10
8 -3 -9 6 -8 -10 8 0
sum 6 7
neg 6 6
sum 8 8
set 5 6
neg 6 8
neg 1 3
sum 1 8
sum 5 8
sum 3 6
set 4 1
```

Output

```
-2
0
-2
-12
11
```

Solution

Time Complexity: $O(n \cdot log(n))$

Memory Complexity: $O(n)$

The brute force solution is quite obvious and we just implement what is asked in the statement. Let's see how we could optimise it.

The solution that is about to be described requires the knowledge of a data structure called [Segment Tree](). This data structure provides an efficient way to solve the following kinds of query problems:

- Point query, point update
- Range query, point update
- Point query, range update (may or may not require lazy propagation technique depending on problem)
- Range query, range update (may or may not require lazy propagation technique depending on problem)

These are some kinds of queries Segment Trees can be used for but applications are not limited to these. The lazy propagation technique is a method that allows us to perform updates on a range of values very efficiently.

Each tree node in a Segment tree holds some precomputed value for a range of indices. Let's say we wanted to find the sum of values in a range, we could use a Segment tree and be able to compute it in only $O(log(n))$ time instead of iterating through and calculating the sum in $O(n)$ time. The link provided above takes a deep dive into segment trees. I advise you to read through and understand their use before trying to understand this solution.

This particular solution requires the use of the lazy propagation technique. We need to support point updates, range updates and range queries. To support the range updates, let's define our node structure to hold two values: integer *sum* and boolean *lazy*. *sum* is the sum of values in the range $[l, r]$ for a particular node (based on what range that node covers) while *lazy* is to denote whether a node needs to push an update of negating values or not.

It is obvious that if we perform two negate operations on the same range of indices, then, the sum does not change. If *lazy* is true, it means that the values in a range are to be negated i.e. $a_i = -a_i$ must be done for all indices i in that particular range (the range can be represented by some subset of tree nodes where the subset size is bounded by $O(log(n))$). If *lazy* is false, it means that the values in that range do not need to be negated.

This boolean property allows us to handle the range updates efficiently. If we need to make a negate update to some range $[l, r]$, we simply change *lazy* to true if it was false and false if it was true. While traversing the segment tree for a point update or range query, we must make sure to push down updates to the children nodes of the current node if that range is to be negated (which can be determined by checking the *lazy* property of the current node).

Solution in C++:

```
/* Arrow */

#ifdef LOST_IN_SPACE
#  if   __cplusplus > 201703LL
#    include "lost_pch1.h" // C++20
#  elif __cplusplus > 201402LL
#    include "lost_pch2.h" // C++17
#  else
#    include "lost_pch3.h" // C++14
#  endif
#else
#  include <bits/stdc++.h>
#endif

constexpr bool test_cases = false;

void solve () {
  int n, q;
  std::cin >> n >> q;

  std::vector <int64_t> a (n);
  for (int i = 0; i < n; ++i)
    std::cin >> a[i];

  struct node {
    int64_t value = 0;
    bool lazy = false;
  };

  std::vector <node> tree (4 * n);

  auto build = [&] (auto self, int v, int tl, int tr) -> void {
    if (tl == tr) {
      tree[v].value = a[tl];
      tree[v].lazy = false;
      return;
    }

    int tm = (tl + tr) / 2;
    self(self, 2 * v + 1, tl, tm);
    self(self, 2 * v + 2, tm + 1, tr);
    tree[v].value = tree[2 * v + 1].value + tree[2 * v + 2].value;
  };

  auto push = [&] (int v) {
    tree[2 * v + 1].value = -tree[2 * v + 1].value;
    tree[2 * v + 2].value = -tree[2 * v + 2].value;
    tree[2 * v + 1].lazy ^= tree[v].lazy;
    tree[2 * v + 2].lazy ^= tree[v].lazy;
    tree[v].lazy = false;
  };

  auto update = [&] (auto self, int v, int tl, int tr, int x, int64_t y) -> void {
    if (tl == tr) {
      tree[v].value = y;
      return;
```

```
    }

    if (tree[v].lazy)
      push(v);

    int tm = (tl + tr) / 2;
    if (x <= tm)
      self(self, 2 * v + 1, tl, tm, x, y);
    else
      self(self, 2 * v + 2, tm + 1, tr, x, y);
    tree[v].value = tree[2 * v + 1].value + tree[2 * v + 2].value;
  };

  auto negate = [&] (auto self, int v, int tl, int tr, int l, int r) -> void {
    if (l > r)
      return;

    if (tl == l and tr == r) {
      tree[v].value = -tree[v].value;
      tree[v].lazy ^= true;
      return;
    }

    if (tree[v].lazy)
      push(v);

    int tm = (tl + tr) / 2;
    self(self, 2 * v + 1, tl, tm, l, std::min(r, tm));
    self(self, 2 * v + 2, tm + 1, tr, std::max(l, tm + 1), r);
    tree[v].value = tree[2 * v + 1].value + tree[2 * v + 2].value;
  };

  auto query = [&] (auto self, int v, int tl, int tr, int l, int r) -> int64_t {
    if (l > r)
      return 0;

    if (tl == l and tr == r)
      return tree[v].value;

    if (tree[v].lazy)
      push(v);

    int tm = (tl + tr) / 2;
    auto lnode = self(self, 2 * v + 1, tl, tm, l, std::min(r, tm));
    auto rnode = self(self, 2 * v + 2, tm + 1, tr, std::max(l, tm + 1), r);
    return lnode + rnode;
  };

  build(build, 0, 0, n - 1);

  std::string type;
  int l, r, x;
  int64_t y;

  while (q--) {
    std::cin >> type;
```

```cpp

    if (type == "neg") {
      std::cin >> l >> r;
      --l; --r;
      negate(negate, 0, 0, n - 1, l, r);
    }
    else if (type == "set") {
      std::cin >> x >> y;
      --x;
      update(update, 0, 0, n - 1, x, y);
    }
    else if (type == "sum") {
      std::cin >> l >> r;
      --l; --r;
      std::cout << query(query, 0, 0, n - 1, l, r) << '\n';
    }
    else
      throw std::runtime_error("invalid query");
  }
}

int main () {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  std::cout.precision(10);
  std::cerr.precision(10);
  std::cout << std::fixed << std::boolalpha;
  std::cerr << std::fixed << std::boolalpha;

  int cases = 1;
  if (test_cases)
    std::cin >> cases;
  while (cases--)
    solve();

  return 0;
}
```

Solution in Python:

I've never implemented a Segment tree in Python as it isn't my primary programming language. If any participant would like to implement the solution for this problem and send it over, I'll be glad to include it here.