This problem describes a simulation of a "Orbs" machine that processes a collection of weighted orbs. The machine operates in rounds, and in each round, it selects two orbs based on their sorted weights, removes them, and introduces two new orbs with derived weights. The goal is to determine the weights of the orbs remaining after a specified number of rounds.

Here's a breakdown of the problem and a C++ approach to solve it:

**Problem Description:**

* **Initial State:** N orbs are in a rail, each with an initial weight w\_i.
* **Machine Operation (L rounds):**
  1. **Selection:** In each round, the machine selects two orbs from the rail based on their weights when sorted from least to greatest: the orb at the *a*-th position and the orb at the *b*-th position (where $a \< b$ always). Let their weights be

x and y respectively (so xley since $a \< b$).

* 1. **Removal:** These two selected orbs are removed from the rail.
  2. **Creation and Insertion:** Two new orbs are created with weights y−x and lfloorfracx+y2rfloor. These new orbs are added to the rightmost end of the rail, first

y−x, then lfloorfracx+y2rfloor.

* **Output:** After L rounds, output the weights of all remaining orbs, sorted from least to greatest.

**Constraints:**

* 2leNle2,000,000
* 1leLle2,000,000
* $1 \\le a \< b \\le N$
* 1lew\_ile1,000,000,000
* Time limit: 2.5 seconds
* Memory limit: 512 MB

**Key Challenges and Data Structures:**

The main challenge lies in efficiently selecting the *a*-th and *b*-th smallest elements and then inserting new elements while maintaining the ability to quickly sort and re-select in subsequent rounds. Since N and L can be up to 2times106, a naive approach of sorting the entire list of orbs in each round would be too slow (O(NlogN) per round, leading to O(LNlogN) total, which is too slow).

To optimize this, we need a data structure that can:

1. **Efficiently find the *k*-th smallest element:** This is required to select the *a*-th and *b*-th orbs.
2. **Efficiently remove elements:** After selection, the two orbs are removed.
3. **Efficiently insert elements:** The two new orbs are added.
4. **Maintain sorted order (or approximate sorted order) for selection:** The data structure should facilitate finding the *k*-th smallest without a full sort every time.

A std::multiset in C++ is a good candidate for this. Here's why:

* It stores elements in sorted order.
* insert() takes O(logS) time, where S is the current number of elements.
* erase() takes O(logS) time.
* Finding the *k*-th element, however, is not direct. Iterating to the *k*-th element takes O(k) time. For large *k*, this can be slow.

A more efficient approach for finding the *k*-th element, insertion, and deletion is using a **balanced binary search tree (BBST)** or a **Fenwick tree (BIT)** / **segment tree** combined with coordinate compression if the value range is too large. However, given w\_ile109, coordinate compression isn't directly applicable for the weights themselves unless we consider their ranks.

Given the constraints, std::multiset might be too slow for finding the *a*-th and *b*-th elements if a or b are large, as iterating through a multiset is linear to the count of elements being iterated. However, N remains constant, so the number of elements in the multiset will always be N.

Let's re-evaluate the number of elements. In each round, 2 elements are removed and 2 elements are added, so the number of elements in the rail always stays

N.

A std::multiset in C++ uses a self-balancing binary search tree. Iterating to the *k*-th element would be O(k) which is problematic if k is close to N. We need a way to access the *k*-th element in O(logN) time. This can be achieved with a policy-based data structure from GNU extension (like tree from <ext/pb\_ds/assoc\_container.hpp>) which supports find\_by\_order.

If policy-based data structures are not allowed or preferred, we can simulate the multiset with two std::multisets or priority queues (one min-heap, one max-heap) to keep a "sliding window" or "median-finding" like structure. However, the problem isn't asking for median but specific *k*-th elements.

Let's assume we can use std::multiset and iterate. N=2times106, L=2times106. If we iterate to a-th and b-th element, it's O(N) for each selection, resulting in O(LcdotN) total, which is 4times1012, definitely too slow.

The correct data structure for this problem, given the need for efficient *k*-th element lookup, insertion, and deletion, is a **std::set of pairs (weight, ID)** or a **std::multiset and careful iteration/counting for the *k*-th element, or a std::map<int, int> to store counts of each weight** which would allow for finding the k-th smallest element.

**Using std::map<long long, int> (Weight -> Count):**

This approach can be efficient for finding the *k*-th smallest element.

1. **Store weights and their counts:** Use std::map<long long, int> weights\_count; where the key is the weight and the value is its count.
2. **Finding *k*-th smallest:** To find the *k*-th smallest element:
   * Iterate through the map. Since std::map is sorted by key, this effectively iterates through weights in increasing order.
   * Maintain a running count\_seen.
   * When count\_seen + current\_weight\_count >= k, the current weight is the *k*-th smallest.
   * This still requires iteration, but on the number of *distinct* weights, not the total number of orbs. In the worst case, all N orbs could have distinct weights, leading to O(N) iteration. Still problematic.

**Correct Approach: Policy-Based Data Structures (if available)**

The most efficient solution typically involves a policy-based data structure from GNU's libstdc++ extensions, specifically tree from <ext/pb\_ds/assoc\_container.hpp>. This provides an ordered set that supports find\_by\_order(k) (returns an iterator to the k-th element, 0-indexed) and order\_of\_key(key) (returns the number of elements strictly less than key).

C++

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

// Define an ordered\_multiset

template<typename T>

using ordered\_multiset = \_\_gnu\_pbds::tree<

T,

\_\_gnu\_pbds::null\_type,

std::less\_equal<T>, // For multiset behavior

\_\_gnu\_pbds::rb\_tree\_tag,

\_\_gnu\_pbds::tree\_order\_statistics\_node\_update>;

The std::less\_equal<T> in ordered\_multiset is important for handling duplicate keys, effectively making it a multiset. However, it's generally better to store std::pair<long long, int> (weight, unique\_id) to handle duplicates correctly with std::less.

**Alternative without Policy-Based Data Structures: Two Priority Queues or Segment Tree**

For the given constraints, a segment tree on compressed coordinates or a custom balanced BST might be needed if policy-based data structures are not allowed. Given the competitive programming context, usually pb\_ds is allowed if it's necessary.

Let's assume pb\_ds is available.

**Algorithm using ordered\_multiset (or ordered\_set with pairs):**

1. **Initialization:**
   * Read

N, L, a, b.

* + Create an ordered\_multiset<long long> (or ordered\_set<std::pair<long long, int>> if std::less\_equal is not good enough for duplicates). Let's use ordered\_set<std::pair<long long, int>> to handle duplicates explicitly by assigning unique IDs.
  + Read the initial N weights w\_i. Insert each w\_i into the ordered\_set along with a unique ID (e.g., its original index). The pair would be

(weight, id).

1. **L Rounds Simulation:**
   * Loop

L times:

* + - **Select orbs:**
      * Get the *a*-th smallest element: auto it\_a = orbs.find\_by\_order(a - 1); (0-indexed). The weight is it\_a->first.
      * Get the *b*-th smallest element: auto it\_b = orbs.find\_by\_order(b - 1); (0-indexed). The weight is it\_b->first.
      * Let

x=it\_a−first and y=it\_b−first.

* + - **Remove orbs:**
      * Erase \*it\_a and \*it\_b from the ordered\_set.
      * **Crucial:** If x=y (i.e., a-th and b-th weights are the same), and we are using ordered\_set<pair<long long, int>>, then it\_a and it\_b will point to different unique (weight, id) pairs. If we used ordered\_multiset<long long>, simply erasing x and y might remove *any* instance of x and y, which is fine if we are certain about the specific elements. However, ordered\_multiset with std::less\_equal typically behaves like multiset, so find\_by\_order would point to correct iterators.
    - **Create and Insert new orbs:**
      * Calculate new weights:

w\_1′=y−x and w\_2′=lfloorfracx+y2rfloor.

* + - * Generate new unique IDs for these orbs. A simple global counter next\_id can work.
      * Insert

(w\_1', next\_id++) and (w\_2', next\_id++) into the ordered\_set.

1. **Final Output:**
   * After L rounds, iterate through the ordered\_set.
   * Print the

first element (weight) of each pair, separated by spaces.

**Example Walkthrough (from problem description for Example 1):**

Initial: N=5, L=3, a=1, b=3. Weights: [40, 20, 10, 30, 50]

Using ordered\_set<pair<long long, int>> (weight, original\_index) for distinctness. Initial orbs: {(10,2), (20,1), (30,3), (40,0), (50,4)} (sorted by weight).

**Round 1:** Sorted weights: 10, 20, 30, 40, 50

a=1 (0-indexed): weight 10 ((10,2)) -> x=10

b=3 (0-indexed): weight 30 ((30,3)) -> y=30

Remove (10,2) and (30,3). New weights:

y−x=30−10=20, lfloorfrac10+302rfloor=lfloor20rfloor=20.

Insert (20, new\_id1), (20, new\_id2). orbs becomes (conceptual, after re-sorting and new IDs): {(20,1), (40,0), (50,4), (20,new\_id1), (20,new\_id2)} Sorted

orbs (with new unique IDs): {(20,1), (20,new\_id1), (20,new\_id2), (40,0), (50,4)}

**Round 2:** Sorted weights: 20, 20, 20, 40, 50

a=1 (0-indexed): weight 20 ((20,1)) -> x=20

b=3 (0-indexed): weight 20 ((20,new\_id2)) -> y=20

Remove (20,1) and (20,new\_id2). New weights:

y−x=20−20=0, lfloorfrac20+202rfloor=lfloor20rfloor=20.

Insert (0, new\_id3), (20, new\_id4). Sorted

orbs: {(0,new\_id3), (20,new\_id4), (20,new\_id1), (40,0), (50,4)}

**Round 3:** Sorted weights: 0, 20, 20, 40, 50

a=1 (0-indexed): weight 0 ((0,new\_id3)) -> x=0

b=3 (0-indexed): weight 20 ((20,new\_id1)) -> y=20

Remove (0,new\_id3) and (20,new\_id1). New weights:

y−x=20−0=20, lfloorfrac0+202rfloor=lfloor10rfloor=10.

Insert (20, new\_id5), (10, new\_id6). Sorted

orbs: {(10,new\_id6), (20,new\_id4), (20,new\_id5), (40,0), (50,4)}

After 3 rounds, remaining weights (sorted): 10, 20, 20, 40, 50. This matches the example output.

**C++ Program Structure:**

C++

#include <iostream>

#include <vector>

#include <algorithm> // For std::sort if needed, but ordered\_set keeps sorted.

#include <cmath> // For std::floor, std::abs

// For policy-based data structures

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

// Define an ordered\_set that behaves like a multiset by storing pairs (value, unique\_id)

// This is necessary because std::less\_equal can be tricky with tree.

// std::less is generally safer and unique\_id ensures distinct nodes for find\_by\_order

template<typename T>

using ordered\_set\_multiset\_behavior = \_\_gnu\_pbds::tree<

std::pair<T, int>, // Store pair<value, unique\_id>

\_\_gnu\_pbds::null\_type,

std::less<std::pair<T, int>>, // Use std::less for pairs

\_\_gnu\_pbds::rb\_tree\_tag,

\_\_gnu\_pbds::tree\_order\_statistics\_node\_update>;

int main() {

// Optimize C++ streams for faster input/output

std::ios\_base::sync\_with\_stdio(false);

std::cin.tie(NULL);

int N; // Number of orbs

int L; // Number of rounds

std::cin >> N >> L;

int a, b; // Parameters for orb selection

std::cin >> a >> b;

ordered\_set\_multiset\_behavior<long long> orbs;

int current\_id = 0; // Unique ID counter for new orbs

// Read initial orb weights

for (int i = 0; i < N; ++i) {

long long w;

std::cin >> w;

orbs.insert({w, current\_id++});

}

// Simulate L rounds

for (int round = 0; round < L; ++round) {

// Step 1: Select the a-th and b-th orbs

// find\_by\_order is 0-indexed, so we use (a-1) and (b-1)

auto it\_a = orbs.find\_by\_order(a - 1);

auto it\_b = orbs.find\_by\_order(b - 1);

long long x = it\_a->first; // Weight of a-th orb

long long y = it\_b->first; // Weight of b-th orb

// Step 2: Remove the selected orbs

orbs.erase(it\_a);

orbs.erase(it\_b);

// Step 3: Create and insert new orbs

long long new\_weight1 = y - x;

long long new\_weight2 = static\_cast<long long>(std::floor(static\_cast<double>(x + y) / 2.0));

// Note: The problem statement uses floor( (x+y)/2 ). For integers, (x+y)/2 in integer division

// naturally floors for positive numbers. But using std::floor and double conversion is safer

// to match the exact mathematical definition, especially if x+y could be negative (though weights are positive here).

// For positive integers, (x+y)/2 (integer division) is equivalent to floor((x+y)/2.0).

orbs.insert({new\_weight1, current\_id++});

orbs.insert({new\_weight2, current\_id++});

}

// Output the remaining orb weights in sorted order

bool first = true;

for (const auto& orb\_pair : orbs) {

if (!first) {

std::cout << " ";

}

std::cout << orb\_pair.first;

first = false;

}

std::cout << std::endl;

return 0;

}