Contest 1 Editorial

COMP4128 25T2

June 19, 2025

A Ores

Algorithm

We begin by noticing that it is always in the players' best interest to take as many diamond ores as possible first before taking iron/gold ores. This is because only the diamond ores are "shared" between the two players, while there is no competition for iron or gold.

Formally, suppose Steve (without loss of generality) takes an iron ore while there are still diamonds available.

- If Steve takes a diamond in the future, we could exchange these two selections without changing the result.
- If Steve does not take a diamond in the future, we could change this selection from iron to diamond and not constrain any of Steve's future moves, so Steve is once again no worse off. Indeed he might now be able to continue the game for one more round than before.

Therefore Steve may as well take a diamond so long as diamonds are available, and the same is true for Alex.

So both players take diamonds until there are none left, and take iron or gold thereafter until they run out of moves.

Method 1

We can explicitly count the number of ores each player takes.

- Steve will take $\lceil \frac{k}{2} \rceil$ diamonds and n iron before running out of moves.
- Alex will take $\lfloor \frac{k}{2} \rfloor$ diamonds and m gold before running out of moves.

Whichever player has more moves available will win. If the number of moves is equal, then Alex wins because Steve had to take first.

The counts and comparison can be performed in constant time.

Method 2

We can skip ahead to the end of the diamonds.

- If the number of diamonds is even, then it is equivalent to a game with only iron and gold.
- If the number of diamonds is odd, then it is equivalent to a game with only iron and gold *but with Alex to play first.*

We can compare the number of iron and gold ores to see who runs out of moves first, with a tie again going to the "second" player.

The counts and comparison can be performed in constant time.

Method 3

We can simulate the entire process of taking one ore at a time. However, this takes $O(n+m+k)$ time, and the input integers are up to 10^9, so this will come very close to or exceed the time limit.

Implementation Notes

General

- 32-bit signed integers (`int`) are sufficient for this problem, as they can store values up to $2^{31}-1 \approx 2 \times 10^9$, and the largest intermediate value we store in any of the methods above is $\approx 1.5 \times 10^9$. However, the same problem could have been posed with inputs up to 10^{18}, in which case we need `long long` for 64-bit integers.
- Even and odd numbers have remainder 0 and 1 respectively modulo 2.

Method 1

- In C/C++, the `/` operator performs integer division, i.e. `a/b` yields $\lfloor \frac{a}{b} \rfloor$. For the ceiling, we can do `(a+b-1)/b` instead! This works because:
  - If a is a multiple of b, then $\lceil \frac{a}{b} \rceil = \lfloor \frac{a}{b} \rfloor$. Adding $b-1$ to a does not reach the next multiple of b, so the quotient is unaffected.
  - If a is *not* a multiple of b, $\lceil \frac{a}{b} \rceil = 1 + \lfloor \frac{a}{b} \rfloor$. Adding $b-1$ to a is guaranteed to reach the next multiple of b, so the quotient is increased by one as required.

Method 2

- The description above gives two cases. We can instead unite these two cases by noticing that if the number of diamonds is odd, then Steve's last diamond is equivalent to an extra iron.

Reference Solution(s)

```cpp
// Solution by Raveen, counting moves
#include <iostream>
using namespace std;

int main (void) {
    int iron, gold, diam;
    cin >> iron >> gold >> diam;

    // number of moves available to each player
    int steve = (diam+1)/2 + iron, alex = diam/2 + gold;
    cout << (steve <= alex ? "Alex" : "Steve") << '\n';
}
```

```cpp
// Solution by Yiheng, using the parity of the diamonds
#include <iostream>
using namespace std;

int main() {
    int n, m, k;
    cin >> n >> m >> k;

    // having an odd number of diamonds is equivalent to
    // having no diamonds and one additional iron
    if (k % 2 == 1) { n++; } // or equivalently, n += (k % 2);

    // if there at least as many gold as iron, then Alex will win, otherwise Steve
    cout << (n <= m ? "Alex" : "Steve") << "\n";
}
```

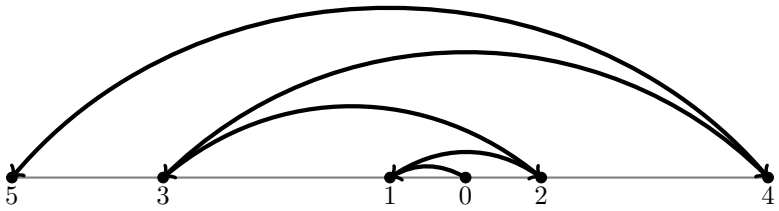
B Eki Stamps

Algorithm

We are asked if there is a permutation of the array such that the absolute difference between adjacent elements is non-decreasing, and if yes, print the permutation.

Such a permutation always exists, which we can demonstrate this with the following construction. First, sort the locations of stamps. One type of valid construction is an oscillation centered at the middle stamp location in this ordering, and at each step expanding one further location alternately to the left and right .

An example path taken using this strategy for $n = 6$ is provided below, where the points represent the stamp locations and the number under them represents the order in which they are visited (0-indexed).



We see that every subsequent journey will cover at least as much distance as the previous journey (simply because we go over the previous path once again; equality is achieved when stamps coincide), fulfilling the required property in the problem statement!

The main catch is that there are many reasonable-seeming schemes to pick the *first* journey (e.g. the closest pair of locations), but the more productive direction is the opposite; the most natural choice for the *last* journey is between the leftmost and rightmost locations, and the rest of the solution follows.

Implementation Notes

General

- A construction is required by the problem, hence we need to preserve the initial index of each stamp location after the sort. This can be done by storing stamp locations as a `pair<int, int>` – the first value being the location, second value being the initial index. This technique was seen in the example problem *Sodor Census* in Lecture 2 (*Getting Started*).
- It turns out that if we store our stamp locations with 0-indexing, starting from index $\lfloor \frac{n}{2} \rfloor$ (or equivalently just $n/2$ in C++) and making the first move leftwards will always work. This eliminates annoying casework of determining initial stamp location and direction for n even or odd.
- An alternative way to approach this construction is to start from either side and work inwards, then reverse the sequence before printing at the end.
- The initial sort complexity is $O(n \log n)$, and regardless of approach the latter construction complexity is $O(n)$. This results in a total of $O(n \log n)$ time complexity, which suffices to pass for the given constraints of $n \leq 10^5$.

Reference Solution(s)

```cpp
// Solution by Yiheng
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int n;
    cin >> n;
```

```
    vector<pair<int, int>> a(n); // (location, index)
    for (int i = 0; i < n; i++) {
        cin >> a[i].first;
        a[i].second = i;
    }

    sort(a.begin(), a.end());

    // start from the middle and take increasing jump sizes
    // to the left and right until no stamp locations are left
    int pos = n / 2, gap = 1, dir = -1;
    while (0 <= pos && pos < n) {
        cout << a[pos].second << " ";
        pos += gap * dir;
        gap++;
        dir *= -1;
    }
    cout << "\n";
}
```

C Snakes and Ladders

Algorithm

We can simply iterate through the dice rolls and note which square it takes us to, finally returning the final square.

Since $2 \leq n \leq 100$, an algorithm which simulates every turn will pass easily. Small optimisations are particularly futile in this setting; we can happily accept a poor constant factor if it allows for easy implementation.

Implementation Notes

General

- It is important to store the "special" squares where the snakes and ladders occur, and encode the origins and destinations of the resulting teleportations.
  - This could be done using a `std::map<int,int>`, mapping origins to destinations.
  - Alternatively, since the number of snakes and ladders is only 52 in total, we could just check every snake and every ladder each time we roll the die.
  - Finally, since the number of squares is only $10,000$, we could instead treat every square as the origin of a teleport. The head of a snake takes you to the tail of that snake, the base of a ladder takes you to the top of that ladder, and every other square takes you to *itself*.
- We must correctly identify snakes and ladders.
  - The `isupper()` and `islower()` functions from C's `string.h` library are provided in the `cstring` header.
  - Since ASCII codes for letters are consecutive, we can infer *which* snake or ladder has been encountered by subtracting `'a'` for snakes (`ch - 'a'`) or `'A'` for ladders (`ch - 'A'`).
- Since the grid numbers start from the bottom and change direction every other row, we must handle this appropriately. We can
  - Parse the snakes and ladders on the board from bottom up, and reverse the direction we read each row (from left to right TO right to left) every 2nd row.
  - Compute a mapping from row and column to square number.
  - Rectify the input, using `reverse()` from the `algorithm` header to flip the right-to-left rows and even reverse the order of rows.

Reference Solution(s)

```
// Solution by Raveen
#include <iostream>
#include <cstring>
using namespace std;

const int N = 110;
char board[N][N];
int dest[N*N+1]; // if you roll to this square, where do you actually get sent
const int T = 30;
int s[T][2]; // snakes: (tail, head)
int l[T][2]; // ladders: (base, top)

int main (void) {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < n; i++) {
        cin >> board[i];
    }

    int id = 0; // square number
    for (int i = n-1; i >= 0; i--) { // bottom to top
```

```
        int j;
        for (int k = 0; k < n; k++) {
            if (i % 2 == 1) { j = k; } // left to right
            else { j = n-1-k; } // right to left
            id++;
            if (islower(board[i][j])) { // snake
                int t = board[i][j] - 'a'; // which snake
                if (s[t][0]) { s[t][1] = id; } // if tail already found, this is head
                else         { s[t][0] = id; } // otherwise this is tail
            } else if (isupper(board[i][j])) { // ladder
                int t = board[i][j] - 'A'; // which ladder
                if (l[t][0]) { l[t][1] = id; } // if base already found, this is top
                else         { l[t][0] = id; } // otherwise this is base
            }
            dest[id] = id;
        }
    }

    for (int t = 0; t < T; t++) {
        dest[s[t][1]] = s[t][0]; // landing on snake head takes you to tail
        dest[l[t][0]] = l[t][1]; // landing on ladder base takes you to top
    } // if some snake or ladder never occured, we just set dest[0] = 0

    int ans = 1, d;
    for (int k = 0; k < m; k++) {
        cin >> d; // die roll
        if (ans + d >= n*n) { // win condition
            cout << n*n << '\n';
            return 0;
        }
        ans = dest[ans+d]; // advance d squares
                           // and follow a snake or ladder if applicable
    }
    cout << ans << '\n';
}
```

D Find the Word

Algorithm

We are asked to find all "locations" of a word occurring within a grid. A very simple algorithm to consider is: what if we try to start s at every single cell in the grid, trying all eight directions that it could be oriented and incrementing our answer by 1 every time s is found.

What is the time complexity of this algorithm? Well for every cell in the grid, we try all 8 directions and "walk" in that direction for at most the length of s giving the complexity of $O(n^2|s|)$.

Implementation Notes

General

- How can we iterate through all eight directions?
  - We could write eight loops, each iterating through one of the directions. However this is obviously tedious, but more importantly prone to bugs (due to the significant repetition).
  - A better approach is to write one function that executes this loop, with the direction encoded in parameters, and call it eight times.
  - The best of all is to hard-code the row and column offsets corresponding to each direction in an array, so we can try each of them in turn using one outer loop.
- When checking each direction, we must avoid reading off the edge of the board.
  - Optimisations here (e.g. only checking in directions where the word can fit) will only save a constant factor.
  - Just bounds check every square *before* you read its value.
  - If the board size n is stored in a global variable, you can avoid passing it to the bounds checking function each time.
  - Small optimisation: declaring the function as `inline` instructs the compiler not to make a new stack frame when it is called, instead substituting the necessary code into the calling function. You can't always do this (e.g. recursive functions) and there is a tradeoff with executable size.
- This problem was meant to get you thinking about edge cases.
  - How many ways could you circle the word, as in the explanation of the first sample case?
  - What if s is a palindrome, i.e. the same forwards and backwards? Our method will double-count it – each "location" of the word will be counted both forwards and backwards. Therefore if the string is a palindrome (easily checked by comparing the string to its reverse, or character-by-character), we must divide the answer by two.
  - Actually there is one more type of edge case! If $|s| = 1$, the above solution will output four times the answer (counting each of the eight directions, and then halving to account for the palindrome). To fix this, we can treat $|s| = 1$ as a special case.
- If ever you are unsure about the interpretation of a problem, please submit a question via the Communication tab of CMS.
  - You can let us make the decision of whether or not it is too direct to be answered.

Reference Solution(s)

```
// Solution by Raveen, checking each direction
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

int n;
```

```
string word;
const int N = 110;
char grid[N][N];

inline bool onboard (int i, int j) { // bounds check
    return i >= 0 && i < n && j >= 0 && j < n;
}

int main (void) {
    cin >> n >> word;
    int len = word.length();
    for (int i = 0; i < n; i++) { cin >> grid[i]; } // scan row-by-row

    int ans = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int _i, _j, l;

            // NW
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i-l, j-l) && grid[i-l][j-l] == word[l]) { l++; }
            if (l == len) { ans++; }

            // N
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i-l, j  ) && grid[i-l][j  ] == word[l]) { l++; }
            if (l == len) { ans++; }

            // NE
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i-l, j+l) && grid[i-l][j+l] == word[l]) { l++; }
            if (l == len) { ans++; }

            // E
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i  , j+l) && grid[i  ][j+l] == word[l]) { l++; }
            if (l == len) { ans++; }

            // SE
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i+l, j+l) && grid[i+l][j+l] == word[l]) { l++; }
            if (l == len) { ans++; }

            // S
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i+l, j  ) && grid[i+l][j  ] == word[l]) { l++; }
            if (l == len) { ans++; }

            // SW
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i+l, j-l) && grid[i+l][j-l] == word[l]) { l++; }
            if (l == len) { ans++; }

            // W
            _i = i; _j = j; l = 0;
            while (l < len && onboard(i  , j-l) && grid[i  ][j-l] == word[l]) { l++; }
            if (l == len) { ans++; }
        }
    }

    // palindrome check
    string rev = word;
    reverse(rev.begin(), rev.end());

    if (word.length() == 1) { ans /= 8; }
    else if (word == rev) { ans /= 2; }

    cout << ans << '\n';
}
```

```
// Solution by Raveen, using a function to check a given direction
#include <algorithm>
#include <cstring>
```

```cpp
#include <iostream>
using namespace std;

int n;
string word;
const int N = 110;
char grid[N][N];

inline bool onboard (int i, int j) { // bounds check
    return i >= 0 && i < n && j >= 0 && j < n;
}

bool check (int i, int j, int di, int dj) { // starting row/col, offset row/col
    for (int l = 0; l < word.length(); l++) {
        i += di; j += dj;
        if (!onboard(i, j) || grid[i][j] != word[l]) { return false; }
    }
    return true;
}

int main (void) {
    cin >> n >> word;
    for (int i = 0; i < n; i++) { cin >> grid[i]; } // scan row-by-row

    int ans = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int di : {-1, 0, 1}) {
                for (int dj : {-1, 0, 1}) {
                    if (di || dj) { // exclude (0,0) direction
                        ans += check(i, j, di, dj);
                    }
                }
            }
        }
    }

    // palindrome check
    string rev = word;
    reverse(rev.begin(), rev.end());

    if (word.length() == 1) { ans /= 8; }
    else if (word == rev) { ans /= 2; }

    cout << ans << '\n';
}
```

```cpp
// Solution by Raveen, using "di/dj trick"
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

int n;
string word;
const int N = 110;
char grid[N][N];
const int di[8] = {-1,-1,-1, 0, 1, 1, 1, 0}; // row offset
const int dj[8] = {-1, 0, 1, 1, 1, 0,-1,-1}; // column offset
                  // NW  N NE  E SE  S SW W

bool onboard (int i, int j) { // bounds check
    return i >= 0 && i < n && j >= 0 && j < n;
}

int main (void) {
    cin >> n >> word;
    for (int i = 0; i < n; i++) { cin >> grid[i]; } // scan row-by-row

    int ans = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < 8; k++) { // for each direction
```

```
                int _i = i, _j = j;
                int l = 0; // match length
                while (l < word.length() && onboard(_i, _j) && grid[_i][_j] == word[l])
                    {
                    _i += di[k];
                    _j += dj[k];
                    l++;
                }
                if (l == word.length()) { ans++; }
            }
        }
    }

    // palindrome check
    string rev = word;
    reverse(rev.begin(), rev.end());

    if (word.length() == 1) { ans /= 8; }
    else if (word == rev) { ans /= 2; }

    cout << ans << '\n';
}
```

E Duels

This problem was hard! Well done to those who solved it.

Observations

For this problem, we need to make a number of observations:

1. If we have two arrays of equal length (call them `A` and `B`), and we want to check whether there exists an arrangement of `A` and `B` such that every value in `A` beats (is greater than) every value in `B`, then we only need to check whether `A` beats `B` when both arrays are sorted.

   Intuitively, we need the smallest value in `A` to beat the smallest value in `B`, second smallest in `A` to beat the second smallest in `B`, and so on. If at some point, the k-th smallest value in `A` does not beat the k-th smallest value in `B`, then it will not beat any other value in `B`, so there is no valid arrangement. You can prove this more formally using any greedy proof.

2. Let the brothers be `A` and `B`, and CP of their i-th Pokémon be `A`$[i]$ and `B`$[i]$ respectively. A tied pairing is possible if there exist equal values in `A` and `B`, i.e., there exist indices i and j such that `A`$[i] =$ `B`$[j]$. Dealing with ties is nontrivial, so we will consider them in a separate case.

   (a) First, consider the case when a tie is impossible. In this case, we can pair the lower half of `A` with the higher half of `B`, and vice versa, to check if a valid arrangement exists.

   We do this by first sorting `A` and `B`, then splitting them into halves `AL`, `AH`, `BL` and `BH`. We then compare `AL` with `BH`, and `AH` with `BL`, and check if this is valid.

   - To prove that this works, consider any winning arrangement of `A` and `B`. Since the twins both get the same number of points, we need exactly half of `A` to be losing, and the other half to be winning.
   - Suppose x and y are winning and losing elements of `A` respectively, but that $x < y$. Then x defeats some w in `B`, and y is defeated by some z in `B`, where $w < x < y < z$. We could swap x and y's opponents, so now y defeats w and x is defeated by z, without changing any of the results. Repeating this, we can enforce that the upper half of `A`'s monsters (by combat power) are winning and the lower half are losing. Denote these `AH` (for `hi`) and `AL` (for `lo`) respectively.
   - Similarly, we need all of `B`'s weaker half `BL` to lose, and all of `BH` to win.
   - `B`'s losers must lose to `A`'s winners and vice versa, and now we can use observation 1 to pair up elements of `AL` with `BH`, and `AH` with `BL`.

   This is a natural subtask to the problem, and is much easier to solve. As a general rule, it's helpful to go through examples or try to solve 'simplified' versions of a problem. In future contests, you will often earn partial marks for solving given subtasks.

   (b) Finally, we can account for ties by testing whether arrays `A` and `B` become valid after we remove some of the ties. Formally, suppose that some integer t appears at least c times in each array. Then we check whether the array is valid after removing each of $2, 4, 6, \ldots, c$ occurrences of t from both `A` and `B`. This represents pairing off some number of tied matches in the `AL` vs `BH` group, and an equal number in the `BL` vs `AH` group. If any number of removals makes the two arrays a valid arrangement, then the answer is `YES`. Otherwise, we have a `NO` case.

   - It seems natural to again insist that `A`'s upper half `AH` should be paired against `BL` and vice versa. However, any claim that narrows the search space requires a proof, even if it seems natural, *because it might not be true!*

     Consider for example the test case

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

     where equal points can only be achieved by making two draws (with a `2` from `AL` paired against a `2` from `BL`), violating the claim!

Proof that only one tied value is required in a valid matching

Consider this simple example of two tied values a and b (WLOG $a < b$) with matched pairs (a, a) and (b, b). This provides `A` with one point and `B` with one point, constituting a valid matching. However if we instead pair (a, b) and (b, a), the players still earn one point each.

In general when given two pairs of ties with different values, we can "cross" the pairings without changing either player's points total.

Therefore we can always reduce the number of ties so long as there are two different tying values. We can repeat this until we are only left with at most 1 tied value.

Algorithm

We begin by sorting both arrays. We then find all the elements in the intersection of the two arrays and store them in pairs of the form (v, f), where v is the value of the element, and f is its frequency. For each (v, f) pair, we check whether there exists an $0 \leq i \leq f$ such that arrays `A` and `B` can form a valid pairing after removing i occurrences of v from both arrays. If such an i exists for any pair, then we return `YES`, otherwise we return `NO`.

The intersection logic can be done in $O(n)$ time, and for each pair (v_i, f_i), we need to test f_i possible arrangements. To test a possible arrangement, we need to first create the two modified arrays, and then check whether they form a valid pairing. Creating the array can be done in $O(n)$ or $O(n \log n)$ time, depending on the approach (see implementation notes), while checking the arrays takes $O(n)$ time. The sum of all frequencies is bounded above by the total number of elements, so this becomes $O(n \cdot \sum_i f_i) = O(n^2)$ time or $O(n^2 \log n)$ time. Overall, this solution can squeeze into the time limit with an optimised $O(n^2 \log n)$ solution or comfortably pass in $O(n^2)$ time.

An alternative student submission passed by rotating array `B` (wrapping around the end) and testing every offset of `B` against `A` (both sorted, as before). There are n offsets to test, and each offset takes $O(n)$ time to check, giving a time complexity of $O(n^2)$. This apparently works, we have no idea why.

Implementation Notes

General

- How do we split and compare `A` and `B`?
  - While we could write a function which takes two `std::vector<int>`s and compares their elements one by one, you might notice here that for every index $0 \leq i < n/2$ in one array, we want the corresponding element to lose to the $(i + n/2)$-th element in the other array. That is, for each $0 \leq i < n/2$, we just need to verify that $\texttt{A}[i] < \texttt{B}[i + n/2]$ and $\texttt{B}[i] < \texttt{A}[i + n/2]$.
  - You can also reuse this code (as a function) to check for winning instances when we deal with ties.
  - This check can early exit whenever one of the comparisons fail.
    - Early exiting generally won't make the difference between a passing and failing submission. In this case, $n^2 \log n \approx 3 \times 10^8$ which is *just* more than 2×10^8 so early exiting brings the time complexity down just enough. The nature of this problem is such that even an $O(nM)$ (where M is the maximum value) solution can pass with early exit, although such solutions could easily be brought down to $O(n^2)$ with coordinate compression.
- Given a pair (`val`, `num`), how do we modify arrays `A` and `B` so that they have `num` fewer occurrences of `val`?
  - It's probably easiest to create two new arrays `A'` and `B'`, and run them through the checking function. It's also much easier if you have a frequency table to store the number of occurrences of each value in each array.
  - One simple way to do this is to add all non-`val` elements from `A` and `B` into their modified arrays, then add `val` to the modified arrays `freq[val] - num` times, where `freq[val]` is the frequency of `val` in the corresponding array. We then sort the array to put it back in sorted order.

– This idea can be improved by keeping in mind that the original arrays are already sorted. We can iterate through an array, keeping track of the number of times we've seen `val`. We ignore the first `num` occurrences, and then start adding `val` to our modified array as normal.

Reference Solution(s)

```cpp
// O(n^2) solution by Yiheng
#include <bits/stdc++.h>

using namespace std;

signed main() {
    int n; cin >> n;
    vector<int> a(n), b(n), cnta(100005, 0), cntb(100005, 0);

    // keep track of the number of occurences of each value in array 'a' and 'b'
    // in 'cnta' and 'cntb'
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        cnta[a[i]]++;
    }
    for (int i = 0; i < n; i++) {
        cin >> b[i];
        cntb[b[i]]++;
    }

    // the maximum number of ties 'i' can be within is min(cnta[i], cntb[i])
    // let's store these all of these possibilities
    vector<pair<int, int>> dups; // (frequency, value) pairs
    for (int i = 0; i < 100005; i++) if (min(cnta[i], cntb[i]) > 0)
        dups.emplace_back(min(cnta[i], cntb[i]), i);

    sort(a.begin(), a.end());
    sort(b.begin(), b.end());

    // function to check whether a valid matching exists AFTER the singular tied
    // value is "removed"
    auto valid = [&](vector<int> &x, vector<int> &y) -> bool {
        bool ret = true;
        for (int i = 0; i < (int) x.size() / 2; i++) {
            if (x[i] >= y[i + (int) x.size() / 2]) ret = false;
            if (y[i] >= x[i + (int) x.size() / 2]) ret = false;
        }
        return ret;
    };

    // let's suppose no ties occur
    // if there exists a valid matching then we can print "Yes" immediately
    if (valid(a, b)) {
        cout << "Yes\n";
        return 0;
    }

    // let's account for ties now
    for (auto [cnt, tie] : dups) {
        vector<int> x, y;
        for (int i = 0; i <= cnt; i++) {
            x.clear(); y.clear();

            // let's ignore the first 'i' occurences of 'tie' in 'a'
            // this is effectively the same as tying 'tie' a total of 'i' times
            int tmp = 0;
            for (int j = 0; j < n; j++) {
                if (a[j] != tie) x.push_back(a[j]);
                else {
                    if (tmp >= i) x.push_back(a[j]);
                    tmp++;
                }
            }

            // same for 'b'
            tmp = 0;
```

```
            for (int j = 0; j < n; j++) {
                if (b[j] != tie) y.push_back(b[j]);
                else {
                    if (tmp >= i) y.push_back(b[j]);
                    tmp++;
                }
            }

            // note that 'x' and 'y' remain in non-decreasing order just like 'a'
            // and 'b'

            // if the number of elements left after removal of ties are not even
            // then the answer is always "No" so we don't have to check at all
            if ((int) x.size() % 2 == 1) continue;

            // all we need to do is check whether the remaining 'a' and 'b' arrays
            // are valid and if so print "Yes"
            if (valid(x, y)) {
                cout << "Yes\n";
                return 0;
            }
        }
    }

    // if we have not found a valid matching after trying all tied values the answer
    // must be "No"
    cout << "No\n";
}
```

```
// O(n^2logn) solution by Yiheng
// the logn factor comes from sorting the array at each step.
// important to note that this squeezes by due to early exits
#include <bits/stdc++.h>

using namespace std;

signed main() {
    int n; cin >> n;

    // keep track of the number of occurences of each value in array 'a' and 'b'
    // in 'cnta' and 'cntb'
    vector<int> a(n), b(n), cnta(100005, 0), cntb(100005, 0);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        cnta[a[i]]++;
    }
    for (int i = 0; i < n; i++) {
        cin >> b[i];
        cntb[b[i]]++;
    }

    // the maximum number of ties 'i' can be within is min(cnta[i], cntb[i])
    // let's store these all of these possibilities
    vector<pair<int, int>> dups; // (frequency, value) pairs
    for (int i = 0; i < 100005; i++) if (min(cnta[i], cntb[i]) > 0)
        dups.emplace_back(min(cnta[i], cntb[i]), i);

    sort(a.begin(), a.end());
    sort(b.begin(), b.end());

    // function to check whether a valid matching exists AFTER the singular tied
    // value is "removed"
    auto valid = [&](vector<int> &x, vector<int> &y) -> bool {
        bool ret = true;
        for (int i = 0; i < (int) x.size() / 2; i++) {
            if (x[i] >= y[i + (int) x.size() / 2]) ret = false;
            if (y[i] >= x[i + (int) x.size() / 2]) ret = false;
        }
        return ret;
    };

    // let's suppose no ties occur
    // if there exists a valid matching then we can print "Yes" immediately
```

```
    if (valid(a, b)) {
        cout << "Yes\n";
        return 0;
    }

    // let's account for ties now
    for (auto [cnt, tie] : dups) {
        vector<int> x, y;
        for (int i = 0; i <= cnt; i++) {
            x.clear(); y.clear();

            // let's ignore 'i' occurences of 'tie' in 'a'
            // this is effectively the same as tying 'tie' a total of 'i' times
            for (int j = 0; j < n; j++) if (a[j] != tie) x.push_back(a[j]);
            for (int j = 0; j < cnta[tie] - i; j++) x.push_back(tie);

            // same for 'b'
            for (int j = 0; j < n; j++) if (b[j] != tie) y.push_back(b[j]);
            for (int j = 0; j < cntb[tie] - i; j++) y.push_back(tie);

            // if the number of elements left after removal of ties are not even
            // then the answer is always "No" so we don't have to check it at all
            if ((int) x.size() % 2 == 1) continue;

            // note that 'x' and 'y' do not remain in non-decreasing order so we
            // must sort them
            // this creates an additional O(logn) factor in time complexity
            sort(x.begin(), x.end());
            sort(y.begin(), y.end());

            // all we need to do is check whether the remaining 'a' and 'b' arrays
            // are valid and if so print "Yes"
            if (valid(x, y)) {
                cout << "Yes\n";
                return 0;
            }
        }
    }

    // if we have not found a valid matching after trying all tied values the answer
    // must be "No"
    cout << "No\n";
}
```

$O(n \log n)$ solution to Duels

Since $n \leq 5000$, $O(n^2)$ is fast enough to solve Duels. However, it is possible to achieve a faster time complexity. The solution we present in this section is non-constructive, meaning that it answers the yes/no question without directly telling us how to organise the duels.

Preliminaries

For simplicity, we call the two brothers A and B and say that the monsters have CP represented by A_i and B_i respectively (rather than j_i and t_i in the original problem).

We use (A_i, B_j) as shorthand to mean that monsters A_i and B_j are paired in a duel.

We define the *score* of a duel to be $+1$ if A wins (which we simply call a *win*), -1 if B wins (which we call a *loss*), and 0 for a draw. The total score is the sum of the scores of all n duels.

The question can be rephrased as: "Is it possible to achieve a total score of 0?".

Solution Overview

Our solution has 3 steps:

1. Find the maximum possible total score (i.e. the best possible scenario for player A). Call this `max_score`.
2. Find the minimum possible total score (i.e. the best possible scenario for player B). Call this `min_score`.
3. The answer is `Yes` if and only if `max_score` ≥ 0 and `min_score` ≤ 0.

Steps 1 and 2 are essentially the same problem with arguments interchanged, so we only describe how to find `max_score`. The proof of the third step is complicated and is deferred to the end of this editorial.

Finding `max_score`

Our algorithm to find `max_score` is constructive, meaning we find a way to organise the n duels to give the best possible outcome for player A. We assume both arrays are sorted, meaning that $A_1 \leq A_2 \leq \ldots \leq A_n$ and $B_1 \leq B_2 \leq \ldots \leq B_n$.

Our algorithm is greedy and considers 3 cases.

Case 1: $A_n > B_n$

If $A_n > B_n$, then A_n can beat any of B's monsters. To maximise the points that A's remaining monsters can get, we might as well pair A_n with B_n (this can be proven formally using an *exchange argument*).

Case 2: $A_1 > B_1$

If $A_1 > B_1$, then B_1 will lose to any of B's monsters. To maximise the points that A's remaining monsters can get, we might as well pair A_1 with B_1 (this can also be proven formally using an *exchange argument*).

Case 3: $A_n = B_n$ **and** $A_1 = B_1$

We have two further subcases.

Case 3.1: $A_1 = A_n$**.** Then, $A_1 = B_1 = A_2 = B_2 = \ldots = A_n = B_n$ (i.e. everything is the same). In this case, it doesn't matter how we pair the monsters because everything will be a draw regardless.

Case 3.2: $A_1 \neq A_n$**.** Since A is sorted, we know that $A_1 < A_n$. In this case, we pair monster A_1 with monster B_n, which will result in a loss for player A. We prove that this is optimal using an exchange argument.

Consider any solution which does not pair A_1 with B_n. Assume that A_1 is paired with B_j and B_n is paired with A_i. A cannot possibly win either of these duels (because $A_1 \leq B_j$ for all j and $B_n \geq A_i$ for all i). We prove that an alternate pairing of (A_1, B_n) and (A_i, B_j) is no worse:

- If (A_1, B_j) and (A_i, B_n) were both draws, then the new pairing (A_1, B_n) will be a loss and (A_i, B_j) will be a win, giving the same score. This is because $B_j = A_1 < B_n = A_i$.
- If (A_1, B_j) was a loss and (A_i, B_n) was a draw, then (A_i, B_j) will either be a draw or a win, giving at least the same score. This is because $A_i = B_n$, and therefore $A_i \geq B_j$ for all j.
- If (A_1, B_j) was a draw and (A_i, B_n) was a loss, then (A_i, B_j) will either be a draw or a win, giving at least the same score. This is because $A_1 = B_j$, and therefore $B_j \leq A_i$ for all i.
- If (A_1, B_j) and (A_i, B_n) were both losses, then rearranging the pairs cannot make the score any worse.

Our algorithm for finding `max_score`

Based on the above cases, the following greedy algorithm can be used to find `max_score`:

1. If $A_n > B_n$, then pair (A_n, B_n).
2. Otherwise, if $A_1 > B_1$, then pair (A_1, B_1).
3. Otherwise, pair (A_1, B_n).

We remove the paired monsters from the lists, and repeat the greedy algorithm. This is done until all monsters have been paired up.

Since the greedy requires us to sort the lists and remove from either end, we use a double-ended queue (`deque`), which supports random access and insertion/removal at either end in constant time. Sorting A and B takes $O(n \log n)$, and executing the greedy takes only linear time thereafter.

Proof that the answer is `Yes` if and only if `max_score` ≥ 0 and `min_score` ≤ 0

Before presenting the proof, we first give some reasons why this claim is not as simple as it may seem.

Firstly, if n is odd, then the claim is false. Consider the input

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

If the monsters are paired as $(2, 1)$, $(4, 3)$, $(6, 5)$, then the score is 3 (which is `max_score`). If the monsters are paired as $(6, 1)$, $(2, 3)$, $(4, 5)$, then the score is -1 (which is `min_score`). However, it is impossible to achieve a score of 0. Specifically, because all 6 CPs are distinct, draws are impossible. Therefore, since n is odd, it is impossible for there to be an equal number of wins and losses and so a score of 0 is impossible. Because of this, our proof **must** rely on the fact that n is even.

Secondly, it is not necessarily possible to achieve **every** possible score between `min_score` and `max_score`. For example, if n is even and all values are distinct, then only even scores are possible. Our proof must utilise some reason why 0 is "special" and is always possible to achieve (the reason is that 0 is even).

In the examples above, `max_score` and `min_score` both have the same parity as n. This is not true in general, for example in the input

```
4
2 3 3 4
1 2 4 4
```

where `max_score` is 1 ((2, 1), (3, 2), (3, 4), (4, 4)) and `min_score` is −1 ((3, 4), (3, 4), (4, 1), (2, 2)).

We first prove for the case where `max_score` is even, and then adapt our proof to the case where `max_score` is odd.

In our proof, we assume `min_score` < 0 and `max_score` > 0, because if either value was equal to 0 then no further proof is necessary.

Proof when `max_score` is even

For simplicity, we reorder A and B so that $(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)$ is a pairing with score `max_score`.

In general, we define a pairing using a permutation σ, where monster A_i is paired with $B_{\sigma(i)}$. Define a *fixed point* of σ as an index i where $\sigma(i) = i$.

Using this notation, the identity pairing (denoted id, with $\mathrm{id}(i) = i$) has score `max_score` and has n fixed points.

Assume we have some pairing σ where $\sigma \neq \mathrm{id}$ (i.e. where not every index is a fixed point). We can gradually transform σ into id as follows, using an operation which we will call a *swap*:

- Let i be an index which is not a fixed point, i.e. where $\sigma(i) \neq i$.
- Let j be the index where $\sigma(j) = i$. We "swap" $\sigma(i)$ and $\sigma(j)$, creating a new pairing σ' where:

$$\sigma'(k) = \sigma(k) \text{ if } k \neq i \text{ and } k \neq j,$$
$$\sigma'(i) = \sigma(j)\,(= i)\,, \text{ and } \sigma'(j) = \sigma(i).$$

Any fixed points of σ are retained in σ', and a new fixed point is created at i. Therefore σ' has strictly more fixed points than σ, and so after at most n swaps we will transform any pairing σ into id.

The outline of our proof is the following: we start with any pairing σ with a negative score (at least one such pairing must exist because `min_score` is negative). We slowly transform σ into id, and show that if we carefully choose our swaps, at least one of the intermediate pairings will have a score of 0.

Specifically, we will show that each swap changes the score by at most 2, meaning that the only way for no intermediate pairing to have a score of 0 is if the score "jumps" from -1 to 1. We call such a swap a *bad swap*. Our goal is prove that there is a sequence of swaps that avoids a bad swap.

We choose our swaps as follows, where σ is the current pairing:

- Option 1: If there is an index i where $A_i = B_i$ but $\sigma(i) \neq i$, then we do a swap using index i.
- Option 2: Otherwise, if there is an index i with $A_i = B_{\sigma(i)}$ but $\sigma(i) \neq i$, then we do a swap using index i.
- Option 3: Otherwise, we do any swap.

We prove two facts:

- Any swap changes the score by at most 2.
- Options 1 and 2 change the score by at most 1.

Together, these two facts are sufficient to complete our proof. If both facts are true, then only option 3 can lead to a bad swap. However, when option 3 occurs, we know that $A_i = B_i$ if and only if $A_i = B_{\sigma(i)}$, meaning that the pairings σ and id have the same number of draws. Because `max_score` is even, it must have an even number of draws, and so the score of σ must also be even. Therefore a bad swap never occurs.

Lemma 1. *Each swap changes the score by at most* 2.

Proof. Assume that a swap changes the score by more than 2. Since a swap only impacts two duels, this can only occur in one of the following scenarios:

- Two wins were turned into two losses (or vice-versa)

- Two wins were turned into a draw and a loss (or vice-versa)
- Two losses were turned into a draw and a win (or vice-versa)

However, all of these options are impossible: if both duels were wins, then after a swap at least one must remain a win (the most powerful of the two winning monsters will stay winning). Similarly, if both duels were losses, then after a swap at least one must remain a loss (the least powerful of the two losing monsters will stay losing). □

Lemma 2. *The swaps in option 1 and 2 change the score by at most 1.*

Proof. In option 1 and 2, both swaps involve at least one pair which is a draw. Consider two cases:

- If both pairs were draws before the swap, then after the swap they are either both still draws or there is a win and a loss. Either way, the score does not change.
- If one pair was a draw and the other was a win, after the swap there will still be at least one win, meaning the score changes by at most 1.
- If one pair was a draw and the other was a loss, after the swap there will still be at least one loss, meaning the score changes by at most 1.

□

Proof when `max_score` is odd

If `min_score` is even, we can apply the same proof.

In fact, if there exists any pairing σ with an even score, we can apply the same proof by making this pairing the target (instead of id). We therefore just need to prove that there exists at least one pairing with an even score.

Lemma 3. *When n is even and `min_score` $\neq$ `max_score`, there always exists at least one pairing σ with an even score.*

Proof. Assume for contradiction that the lemma is false. Consider any pairing σ, which by our assumption must have an odd score. We will show that either:

- It is possible to make a swap to create a new pairing σ' with an even score (a contradiction), or
- All pairings have the same score (a contradiction because this implies that `min_score` = `max_score`).

Since the score of σ is odd and n is even, σ must contain at least one draw. Further, if there exist draws with two different values (i.e. $A_i = B_{\sigma(i)}$ and $A_j = B_{\sigma(j)}$ but $A_i \neq A_j$), we can swap these two pairs and this will not change the score. Because of this, we assume that all draws are at a single value of combat power. Call this value k – that is, if $A_i = B_{\sigma(i)}$, then $A_i = B_{\sigma(i)} = k$. Let i be the index (in A) of one of these draws.

If there exists another index j where $A_j \neq k$ and $B_{\sigma(j)} \neq k$, then we create a new pairing σ' by swapping $\sigma(i)$ and $\sigma(j)$. In this new pairing, we know that $A_i \neq B_{\sigma'(i)}$ and $A_j \neq B_{\sigma'(j)}$, meaning that σ' has exactly one fewer draw than σ. It follows that σ' must have an even score, which is a contradiction.

Now assume that it is impossible to achieve this contradiction. That is, for every j, either $A_j = k$ or $B_{\sigma(j)} = k$ (or both). Two important facts follow from this:

1. Of the $2n$ combat power values, k occurs at least $n+1$ times. This is because there is at least one draw at CP k (accounting for two occurrences) and each of the other $n-1$ pairs has at least one occurrence.
2. Every monster that player A has with $A_j > k$ must be winning (because it is paired to a monster with CP k). Similarly, every monster with $A_j < k$ is losing. The same is true for player B: monsters with $B_j > k$ are winning and $B_j < k$ are losing. Putting this together, the score of the pairing is

$$\underbrace{[\#j : A_j > k]}_{\text{A's winners}} + \underbrace{[\#j : B_j < k]}_{\text{B's losers}} - \underbrace{[\#j : A_j < k]}_{\text{A's losers}} - \underbrace{[\#j : B_j > k]}_{\text{B's winners}},$$

noting that no duel is double-counted because any monster with CP other than k has an opponent with CP exactly k.

Because we assume that it is impossible to achieve the first contradiction for **any** pairing σ, the above two facts are true for every pairing. Because of the first fact, we know that every pairing has the same value of k, and because of the second fact we know that every pairing has the same score. This contradicts the assumption that `min_score` $\neq$ `max_score`, completing the proof. □

Reference Solution

```cpp
#include <algorithm>
#include <deque>
#include <iostream>
using namespace std;

int n;

// we use std::deque because it allows us to do both pop_back and pop_front in O(1)
int getMaxScore(deque<int> A, deque<int> B) { // pass by copy
    sort(A.begin(), A.end());
    sort(B.begin(), B.end());
    int score = 0;
    for (int i = 0; i < n; i++) {
        int chosen_A, chosen_B;
        if (A.back() > B.back()) {
            chosen_A = A.back(); A.pop_back();
            chosen_B = B.back(); B.pop_back();
        } else if (A.front() > B.front()){
            chosen_A = A.front(); A.pop_front();
            chosen_B = B.front(); B.pop_front();
        } else {
            chosen_A = A.front(); A.pop_front();
            chosen_B = B.back(); B.pop_back();
        }
        if (chosen_A > chosen_B) { score++; }
        else if (chosen_A < chosen_B) { score--; }
    }
    return score;
}

int main (void) {
    cin >> n;
    deque<int> A(n), B(n);
    for (int i = 0; i < n; i++) { cin >> A[i]; }
    for (int i = 0; i < n; i++) { cin >> B[i]; }
    int max_score = getMaxScore(A, B);
    int min_score = -getMaxScore(B, A);
    cout << (max_score >= 0 && min_score <= 0 ? "Yes" : "No") << '\n';
}
```