Code the below question in c++

Given two integer array A and B of size N each.
A sum combination is made by adding one element from array A and another element of array B.
Return the maximum K valid sum combinations from all the possible sum combinations.

Note : Output array must be sorted in non-increasing order.

Example 1:

Input:
N = 2
K = 2
A [ ] = {3, 2}
B [ ] = {1, 4}
Output: {7, 6}
Explanation:
7 -> (A : 3) + (B : 4)
6 -> (A : 2) + (B : 4)
Example 2:

Input:
N = 4
K = 3
A [ ] = {1, 4, 2, 3}
B [ ] = {2, 5, 1, 6}
Output: {10, 9, 9}
Explanation:
10 -> (A : 4) + (B : 6)
9 -> (A : 4) + (B :
9 -> (A : 3) + (B : 6)
Your Task:
You don't need to read input or print anything. Your task is to complete the function maxCombinations() which takes the interger N,integer K and two integer arrays A [ ] and B [ ] as parameters and returns the maximum K valid distinct sum combinations .

Expected Time Complexity: O(Nlog(N))
Expected Auxiliary Space: O(N)

Constraints:
1 ≤ N ≤ 105
1 ≤ K ≤ N
1 ≤ A [i] , B [i] ≤ 104

To solve this problem, we can use the following approach:

Approach:

1. **Sort both arrays A and B**: This will allow us to efficiently generate the largest sum combinations, as the largest sums will come from the largest elements of both arrays.
2. **Use a Max-Heap (Priority Queue)**: The goal is to find the maximum `K` sums. A max-heap will allow us to quickly extract the largest sums. In C++, the `priority_queue` is a max-heap by default.
3. **Generate sum combinations**: Start by adding the largest element of `A` with the largest element of `B`. After that, we will generate possible sum combinations by moving one element from `A` and one element from `B` while ensuring that the combinations are distinct.
4. **Store the results in a heap**: We will maintain a max-heap to keep track of the largest combinations. As we generate combinations, we insert them into the heap and extract the top `K` largest combinations.

Steps:

1. **Sort arrays A and B** in descending order.
2. **Push the sum of the largest elements** of A and B into the heap.
3. For each combination, push the next possible combinations into the heap, and track the already visited indices to avoid duplicates.
4. Extract the top `K` sums from the heap.

Code Implementation:

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>

using namespace std;

vector<int> maxCombinations(int N, int K, vector<int>& A, vector<int>& B) {
    // Sort both arrays in descending order
    sort(A.begin(), A.end(), greater<int>());
    sort(B.begin(), B.end(), greater<int>());

    // Max-Heap to store the combinations
    priority_queue<pair<int, pair<int, int>>> maxHeap;

    // Set to track visited pairs of indices
    set<pair<int, int>> visited;

    // Insert the first combination (A[0] + B[0]) into the max-heap
    maxHeap.push({A[0] + B[0], {0, 0}});
    visited.insert({0, 0});

    vector<int> result;

    // Generate the top K combinations
    while (K--) {
        // Get the current max combination
        auto top = maxHeap.top();
        maxHeap.pop();

        int sum = top.first;
        int i = top.second.first;
        int j = top.second.second;

        // Add the sum to the result
        result.push_back(sum);

        // Try to add the next element from A or B to the current sum combination
        if (i + 1 < N && visited.find({i + 1, j}) == visited.end()) {
            maxHeap.push({A[i + 1] + B[j], {i + 1, j}});
            visited.insert({i + 1, j});
        }

        if (j + 1 < N && visited.find({i, j + 1}) == visited.end()) {
            maxHeap.push({A[i] + B[j + 1], {i, j + 1}});
            visited.insert({i, j + 1});
        }
    }

    return result;
}

int main() {
    // Example 1
    vector<int> A1 = {3, 2};
    vector<int> B1 = {1, 4};
```

```
    int N1 = 2;
    int K1 = 2;
    vector<int> result1 = maxCombinations(N1, K1, A1, B1);
    for (int val : result1) {
        cout << val << " ";
    }
    cout << endl;  // Output: 7 6

    // Example 2
    vector<int> A2 = {1, 4, 2, 3};
    vector<int> B2 = {2, 5, 1, 6};
    int N2 = 4;
    int K2 = 3;
    vector<int> result2 = maxCombinations(N2, K2, A2, B2);
    for (int val : result2) {
        cout << val << " ";
    }
    cout << endl;  // Output: 10 9 9

    return 0;
}
```

Explanation:

1. **Sorting**: Both `A` and `B` are sorted in descending order so that the largest elements are considered first when generating combinations.
2. **Max-Heap**: We use a max-heap (priority queue) to store the sum of elements from `A[i] + B[j]`, along with the indices `(i, j)` to track which elements from `A` and `B` were used to form the sum. This allows efficient extraction of the maximum sum combination.
3. **Visited Set**: A `set` is used to ensure that we don't revisit the same index pair `(i, j)` multiple times, avoiding duplicate combinations.
4. **Main Logic**:
   - Initially, we push the largest combination `(A[0] + B[0])` into the heap.
   - Then we generate new sums by incrementing either the index in `A` or in `B` and push these combinations into the heap.
   - The top `K` largest sums are extracted from the heap and returned as the result.

Time Complexity:

- Sorting both arrays takes `O(N log N)`.
- The priority queue operations (insertions and deletions) take `O(log N)`, and we perform at most `K` such operations.
- Overall, the time complexity is dominated by `O(N log N)` due to the sorting step.

Space Complexity:

- We use a priority queue and a set to store combinations, so the space complexity is `O(N)`.

Edge Cases:

- If `K = 1`, we should return just the single largest combination.
- If `N = 1`, only one sum combination will be possible.