Longest Bitonic Subseq In C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
int LongestBitonicSubseq(int arr[], int n) {
    vector<int> lis(n, 1); // lis[i] will store the length of LIS ending at index i
    vector<int> lds(n, 1); // lds[i] will store the length of LDS starting at index i

    // Computing LIS
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (arr[j] <= arr[i]) {
                lis[i] = max(lis[i], lis[j] + 1);
            }
        }
    }

    // Computing LDS
    for (int i = n - 2; i >= 0; i--) {
        for (int j = n - 1; j > i; j--) {
            if (arr[j] <= arr[i]) {
                lds[i] = max(lds[i], lds[j] + 1);
            }
        }
    }

    int omax = 0; // To store the overall maximum length of LBS

// Finding the length of the Longest Bitonic Subsequence
    for (int i = 0; i < n; i++) {
        omax = max(omax, lis[i] + lds[i] - 1);
    }
 return omax;
}

int main() {
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << LongestBitonicSubseq(arr, n) << endl;

    return 0;
}
```

Step 1: Compute lis (Longest Increasing Subsequence)

The lis logic in your code checks every previous index j for every current index i (j < i) and ensures:

```cpp
if (arr[j] <= arr[i]) {
    lis[i] = max(lis[i], lis[j] + 1);
}
```

This means:

- It allows increasing subsequences.
- It also includes elements that are **equal** (since arr[j] <= arr[i]).

Step 2: Compute lds (Longest Decreasing Subsequence)

The lds logic in your code checks every later index j for every current index i (j > i) and ensures:

```cpp
if (arr[j] <= arr[i]) {
    lds[i] = max(lds[i], lds[j] + 1);
}
```

This means:

- It allows decreasing subsequences.
- It also includes elements that are **equal** (since arr[j] <= arr[i]).

Step 3: Compute omax (Longest Bitonic Subsequence)

The total length of the Longest Bitonic Subsequence is computed as:

```cpp
omax = max(omax, lis[i] + lds[i] - 1);
```

This combines lis[i] and lds[i] for every index i, but subtracts 1 to avoid double-counting the pivot element.

Test Input

The array is:

arr = {10, 22, 9, 33, 21, 50, 41, 60, 80, 3}

Let's compute lis, lds, and omax step-by-step **exactly as per your code**.

Step 1: Compute lis

Index (i)	Value (arr[i])	LIS (lis[i]) Calculation
0	10	lis[0] = 1 (initial)
1	22	lis[1] = 2 (10 → 22)
2	9	lis[2] = 1 (no increase)
3	33	lis[3] = 3 (10 → 22 → 33)
4	21	lis[4] = 2 (10 → 21)
5	50	lis[5] = 4 (10 → 22 → 33 → 50)
6	41	lis[6] = 4 (10 → 22 → 33 → 41)
7	60	lis[7] = 5 (10 → 22 → 33 → 50 → 60)
8	80	lis[8] = 6 (10 → 22 → 33 → 50 → 60 → 80)
9	3	lis[9] = 1 (no increase)

LIS Array: {1, 2, 1, 3, 2, 4, 4, 5, 6, 1}

Step 2: Compute lds

Index (i)	Value (arr[i])	LDS (lds[i]) Calculation
9	3	lds[9] = 1 (initial)
8	80	lds[8] = 2 (80 → 3)
7	60	lds[7] = 3 (60 → 3)
6	41	lds[6] = 4 (41 → 3)
5	50	lds[5] = 5 (50 → 41 → 3)
4	21	lds[4] = 2 (21 → 3)

Index (i)	Value (arr[i])	LDS (lds[i]) Calculation
3	33	lds[3] = 4 (33 → 21 → 3)
2	9	lds[2] = 2 (9 → 3)
1	22	lds[1] = 3 (22 → 9 → 3)
0	10	lds[0] = 3 (10 → 9 → 3)

LDS Array: {3, 3, 2, 4, 2, 5, 4, 3, 2, 1}

Step 3: Compute omax

LBS[i]=LIS[i]+LDS[i]−1LBS[i] = LIS[i] + LDS[i] - 1LBS[i]=LIS[i]+LDS[i]−1

Index (i)	LIS (lis[i])	LDS (lds[i])	LBS (lis[i] + lds[i] - 1)
0	1	3	3
1	2	3	4
2	1	2	2
3	3	4	6
4	2	2	3
5	4	5	8
6	4	4	7
7	5	3	7
8	6	2	7
9	1	1	1

Maximum LBS: 7

Correct Output: 7

Output:-7