All paths minimum jumps in C++

```
#include <iostream>
#include <climits>
#include <queue>
using namespace std;

class Pair {
public:
    int i, s, j;
    string psf;

    Pair(int i, int s, int j, string psf) {
        this->i = i;
        this->s = s;
        this->j = j;
        this->psf = psf;
    }
};

void solution(const int arr[], int n) {
    int dp[n];
    fill_n(dp, n, INT_MAX);
    dp[n - 1] = 0;

    for (int i = n - 2; i >= 0; i--) {
        int steps = arr[i];
        int min_steps = INT_MAX;

        for (int j = 1; j <= steps && i + j < n; j++) {
            if (dp[i + j] != INT_MAX && dp[i + j] <
min_steps) {
                min_steps = dp[i + j];
            }
        }

        if (min_steps != INT_MAX) {
            dp[i] = min_steps + 1;
        }
    }

    cout << dp[0] << endl;

    queue<Pair> q;
    q.emplace(0, arr[0], dp[0], "0");

    while (!q.empty()) {
        Pair rem = q.front();
        q.pop();

        if (rem.j == 0) {
            cout << rem.psf << "." << endl;
        }

        for (int j = 1; j <= rem.s && rem.i + j < n;
j++) {
            int ci = rem.i + j;
            if (dp[ci] != INT_MAX && dp[ci] ==
rem.j - 1) {
                q.emplace(ci, arr[ci], dp[ci], rem.psf
+ "->" + to_string(ci));
            }
```

Dry Run:

Step 1: Calculate the dp array (minimum jumps to reach the end from each index)

The dp array keeps track of the minimum number of jumps required to reach the last index from any given index. Let's calculate the dp array starting from the last index (since we know that dp[n-1] = 0 as no jumps are needed from the last index):

- dp[9] = 0 (since we're already at the last index).
- dp[8] = INT_MAX (can't reach the last index from index 8, because there are no valid jumps).
- dp[7] = 1 (one jump to index 9, because arr[7] = 2 allows jumping to index 9).
- dp[6] = 1 (one jump to index 9, because arr[6] = 4 allows jumping to index 9).
- dp[5] = 2 (minimum of dp[6] + 1 and dp[7] + 1, so min(1+1, 1+1) = 2).
- dp[4] = 2 (minimum of dp[5] + 1 and dp[6] + 1, so min(2+1, 1+1) = 2).
- dp[3] = 2 (minimum of dp[4] + 1 and dp[5] + 1, so min(2+1, 2+1) = 2).
- dp[2] = 3 (can't jump to a valid position from here).
- dp[1] = 3 (same as above, can't jump to a valid position).
- dp[0] = 4 (minimum of dp[1] + 1, dp[2] + 1, and dp[3] + 1, so min(3+1, 3+1, 2+1) = 4).

Thus, the dp array will look like this:

dp = {4, 3, 3, 2, 2, 2, 1, 1, INT_MAX, 0}

Step 2: Generate paths using BFS

Next, we use BFS to generate all valid paths from the start (index 0) to the end (index 9) using the minimum number of jumps (dp[0] = 4).

We initialize the queue with the first index 0 and process each index in the queue, exploring all possible jumps from that index:

1. Start from index 0, jump to index 3 (because dp[3] = 2 and dp[0] = dp[3] + 1).
2. From index 3, jump to index 5 (because dp[5] = 2 and dp[3] = dp[5] + 1).
3. From index 5, jump to index 6 (because dp[6] = 1 and dp[5] = dp[6] + 1).
4. From index 6, jump to index 9 (because dp[9] = 0 and dp[6] = dp[9] + 1).

This gives the path: 0 -> 3 -> 5 -> 6 -> 9.

```
            }
        }
    }

    int main() {
        const int arr[] = {3, 3, 0, 2, 1, 2, 4, 2, 0, 0};
        int n = sizeof(arr) / sizeof(arr[0]);
        solution(arr, n);
        return 0;
    }
```

Similarly, another valid path is:

1. Start from index 0, jump to index 3.
2. From index 3, jump to index 5.
3. From index 5, jump to index 7 (because dp[7] = 1 and dp[5] = dp[7] + 1).
4. From index 7, jump to index 9 (because dp[9] = 0).

This gives the path: 0 -> 3 -> 5 -> 7 -> 9.

Step 3: Final Output

The correct output should be:

4
0->3->5->6->9.
0->3->5->7->9.

Output:-
4
0->3->5->6->9.
0->3->5->7->9.