Paths of 0-1 knapsack In C++

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

struct Pair {
    int i;
    int j;
    string psf;

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

void printPaths(vector<vector<int>>& dp,
vector<int>& vals, vector<int>& wts, int i,
int j, string psf, deque<Pair>& que) {
    while (!que.empty()) {
        Pair rem = que.front();
        que.pop_front();

        if (rem.i == 0 || rem.j == 0) {
            cout << rem.psf << endl;
        } else {
            int exc = dp[rem.i - 1][rem.j];

            if (rem.j >= wts[rem.i - 1]) {
                int inc = dp[rem.i - 1][rem.j -
wts[rem.i - 1]] + vals[rem.i - 1];

                if (dp[rem.i][rem.j] == inc) {
                    que.push_back(Pair(rem.i - 1,
rem.j - wts[rem.i - 1], to_string(rem.i - 1) +
" " + rem.psf));
                }
            }

            if (dp[rem.i][rem.j] == exc) {
                que.push_back(Pair(rem.i - 1,
rem.j, rem.psf));
            }
        }
    }
}

void knapsackPaths(vector<int>& vals,
vector<int>& wts, int cap) {
    int n = vals.size();
    vector<vector<int>> dp(n + 1,
```

Knapsack Problem Explanation:

- **Items**: There are 5 items with associated values and weights:
  - Item 1: Value = 15, Weight = 2
  - Item 2: Value = 14, Weight = 5
  - Item 3: Value = 10, Weight = 1
  - Item 4: Value = 45, Weight = 3
  - Item 5: Value = 30, Weight = 4
- **Knapsack Capacity**: The knapsack has a capacity of 7 units.

Dynamic Programming Table (dp):

The dynamic programming table is built to calculate the maximum value that can be achieved for each capacity using the first i items. The size of the table is (n+1) x (cap+1) where n is the number of items and cap is the knapsack capacity.

DP table construction:

The DP table dp[i][j] represents the maximum value achievable with the first i items and a knapsack capacity j. The recurrence relation is as follows:

1. If item i is not included: dp[i][j] = dp[i-1][j]
2. If item i is included (i.e., if the weight of the item is less than or equal to the remaining capacity): dp[i][j] = max(dp[i][j], dp[i-1][j-wts[i-1]] + vals[i-1])

Here's how the DP table looks after filling:

Items/ Capacity	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1 (15, 2)	0	0	15	15	15	15	15	15
2 (14, 5)	0	0	15	15	15	14	15	15
3 (10, 1)	0	10	15	15	25	25	25	25
4 (45, 3)	0	10	15	45	45	45	55	55
5 (30, 4)	0	10	15	45	45	45	55	75

- The maximum value achievable with a capacity of 7 is 75, which occurs by including the items 3 and 4 (corresponding to the values 10 and 45, respectively).

Path Finding:

```
vector<int>(cap + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= cap; j++) {
            dp[i][j] = dp[i - 1][j];

            if (j >= wts[i - 1]) {
                dp[i][j] = max(dp[i][j], dp[i - 1][j -
wts[i - 1]] + vals[i - 1]);
            }
        }
    }

    int ans = dp[n][cap];
    cout << "Maximum value: " << ans <<
endl;

    deque<Pair> que;
    que.push_back(Pair(n, cap, ""));

    printPaths(dp, vals, wts, n, cap, "", que);
}

int main() {
    vector<int> vals = {15, 14, 10, 45, 30};
    vector<int> wts = {2, 5, 1, 3, 4};
    int cap = 7;

    knapsackPaths(vals, wts, cap);

    return 0;
}
```

Once the DP table is filled, the program uses a breadth-first search (BFS) approach to backtrack and find all the possible paths that lead to the maximum value. The paths are stored in a deque, and for each item, the program checks whether the current value is achieved by including or excluding the item.

The function printPaths recursively finds all paths that lead to the maximum value and prints them. The path output is based on the indices of the items included in the optimal knapsack configuration.

Output:

- The **maximum value** for the knapsack is 75.
- The path "3 4" indicates that items 3 and 4 were included in the optimal solution, which leads to the maximum value.

Output:-

```
Maximum value: 75
3 4
```