1/0 Knapsack Problem using Dynamic Programming:

Dynamic programming solves the 0/1 knapsack problem by storing solutions to subproblems and building up to the overall problem.

Algorithm:

1. Create a 2D array **dp[n+1][capacity+1]** where **n** is the number of items and **capacity** is the knapsack capacity.
2. Initialize the array with zeros.
3. Iterate through items and capacities, filling up the **dp** table with optimal values based on whether the item can be included or not.

Code:

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

int knapsackDP(int values[], int weights[], int n, int capacity) {
   int dp[n + 1][capacity + 1];
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (weights[i - 1] <= w)
            dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
         else
            dp[i][w] = dp[i - 1][w];
      }
   }
   return dp[n][capacity];
}

int main() {
   int values[] = {60, 100, 120};
   int weights[] = {10, 20, 30};
   int n = sizeof(values) / sizeof(values[0]);
   int capacity = 50;
   cout << "Maximum value in Knapsack: " << knapsackDP(values, weights, n, capacity);
   return 0;
}
```

Output:

```
Maximum value in Knapsack: 220
```

Space complexity: _____

Justification:_____
_____

Time complexity:

Best case time complexity: _____

Justification:_____
_____

Worst case time complexity: _____

Justification:_____