0 1 Knapsack Pattern:

Introduction
Given the weights and profits of ‘N’ items, we are asked to put these items in a knapsack which has a capacity ‘C’. The goal is to get the maximum profit from the items in the knapsack. Each item can only be selected once, as we don’t have multiple quantities of any item.

Let’s take the example of Merry, who wants to carry some fruits in the knapsack to get maximum profit. Here are the weights and profits of the fruits:

Items: { Apple, Orange, Banana, Melon }
Weights: { 2, 3, 1, 4 }
Profits: { 4, 5, 3, 7 }
Knapsack capacity: 5

Let’s try to put different combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit
Apple + Banana (total weight 3) => 7 profit
Orange + Banana (total weight 4) => 8 profit
Banana + Melon (total weight 5) => 10 profit

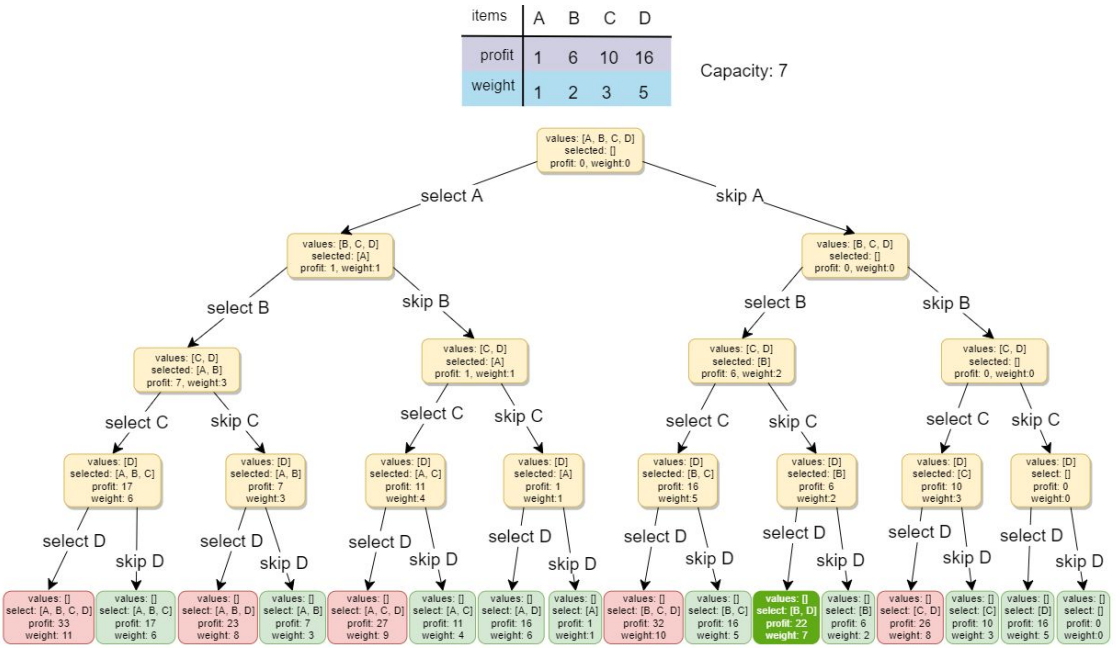
This shows that Banana + Melon is the best combination, as it gives us the maximum profit and the total weight does not exceed the capacity.

Problem Statement
Given two integer arrays to represent weights and profits of ‘N’ items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number ‘C’. Each item can only be selected once, which means either we put an item in the knapsack or we skip it.

Basic Solution
A basic brute-force solution could be to try all combinations of the given items (as we did above), allowing us to choose the one with maximum profit and a weight that doesn’t exceed ‘C’. Take the example of four items (A, B, C, and D), as shown in the diagram below. To try all the combinations, our algorithm will look like:

```
for each item 'i'
  create a new set which INCLUDES item 'i' if the total weight does not exceed the capacity, and
     recursively process the remaining capacity and items
  create a new set WITHOUT item 'i', and recursively process the remaining items
return the set from the above two sets with higher profit
```

Here is a visual representation of our algorithm:



All green boxes have a total weight that is less than or equal to the capacity (7), and all the red ones have a weight that is more than 7. The best solution we have is with items [B, D] having a total profit of 22 and a total weight of 7.

Code

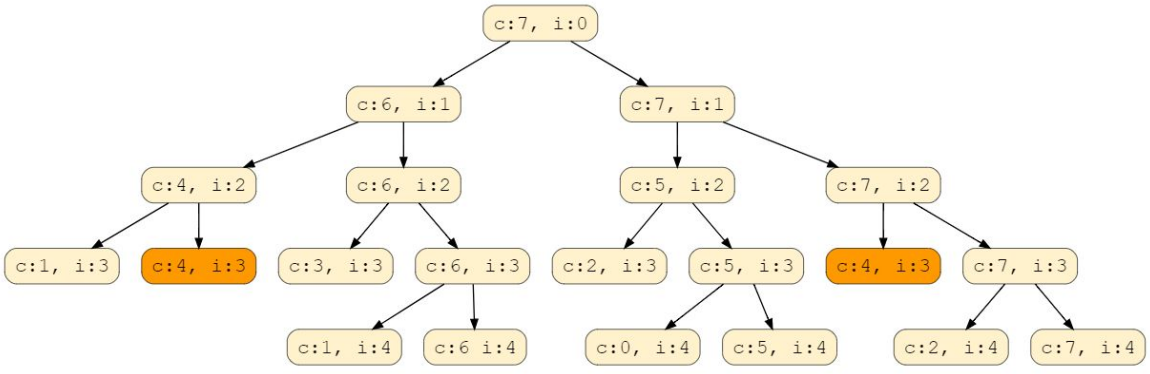
Here is the code for the brute-force solution:

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
public:
  int solveKnapsack(const vector<int> &profits, const vector<int> &weights,
int capacity) {
    return this->knapsackRecursive(profits, weights, capacity, 0);
  }

private:
  int knapsackRecursive(const vector<int> &profits, const vector<int>
&weights, int capacity,
                        int currentIndex) {
    // base checks
    if (capacity <= 0 || currentIndex >= profits.size()) {
      return 0;
    }

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we
shouldn't process this
    int profit1 = 0;
```

```
    if (weights[currentIndex] <= capacity) {
      profit1 =
          profits[currentIndex] +
          knapsackRecursive(profits, weights, capacity -
weights[currentIndex], currentIndex + 1);
    }

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(profits, weights, capacity, currentIndex
+ 1);

    return max(profit1, profit2);
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

The time complexity of the above algorithm is exponential O(2^n), where ‘n’ represents the total number of items. This can also be confirmed from the above recursion tree. As we can see that we will have a total of ‘31’ recursive calls – calculated through (2^n) + (2^n) - 1, which is asymptotically equivalent to O(2^n).
The space complexity is O(n). This space will be used to store the recursion stack. Since our recursive algorithm works in a depth-first fashion, which means, we can’t have more than ‘n’ recursive calls on the call stack at any time.

Let’s visually draw the recursive calls to see if there are any overlapping sub-problems. As we can see, in each recursive call, profits and weights arrays remain constant, and only capacity and currentIndex change. For simplicity, let’s denote capacity with ‘c’ and currentIndex with ‘i’:



We can clearly see that ‘c:4, i=3’ has been called twice, hence we have an overlapping sub-problems pattern. As we discussed above, overlapping sub-problems can be solved through Memoization.

Top-down Dynamic Programming with Memoization

We can use memoization to overcome the overlapping sub-problems. To reiterate, memoization is when we store the results of all the previously solved sub-problems and return the results from memory if we encounter a problem that has already been solved.

Since we have two changing values (capacity and currentIndex) in our recursive function knapsackRecursive(), we can use a two-dimensional array to store the results of all the solved sub-problems. As mentioned above, we need to store results for every sub-array (i.e. for every possible index 'i') and for every possible capacity 'c'.

Code

Here is the code with memoization (see the changes in the highlighted lines):

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
public:
  int solveKnapsack(const vector<int> &profits, const vector<int> &weights,
int capacity) {
    vector<vector<int>> dp(profits.size(), vector<int>(capacity + 1, -1));
    return this->knapsackRecursive(dp, profits, weights, capacity, 0);
  }

private:
  int knapsackRecursive(vector<vector<int>> &dp, const vector<int> &profits,
                        const vector<int> &weights, int capacity, int
currentIndex) {
    // base checks
    if (capacity <= 0 || currentIndex >= profits.size()) {
      return 0;
    }

    // if we have already solved a similar problem, return the result from
memory
    if (dp[currentIndex][capacity] != -1) {
      return dp[currentIndex][capacity];
    }

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we
shouldn't process this
    int profit1 = 0;
    if (weights[currentIndex] <= capacity) {
      profit1 = profits[currentIndex] + knapsackRecursive(dp, profits,
weights,
                                                          capacity -
weights[currentIndex],
```

```
                                                    currentIndex + 1);
    }

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(dp, profits, weights, capacity,
currentIndex + 1);

    dp[currentIndex][capacity] = max(profit1, profit2);
    return dp[currentIndex][capacity];
  }
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

What is the time and space complexity of the above solution? Since our memoization array dp[profits.length][capacity+1] stores the results for all the subproblems, we can conclude that we will not have more than N*C subproblems (where 'N' is the number of items and 'C' is the knapsack capacity). This means that our time complexity will be O(N*C).

The above algorithm will be using O(N*C) space for the memoization array. Other than that we will use O(N) space for the recursion call-stack. So the total space complexity will be O(N*C + N), which is asymptotically equivalent to O(N*C).

Bottom-up Dynamic Programming
Let's try to populate our dp[][] array from the above solution, working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and for every possible capacity. This means, dp[i][c] will represent the maximum knapsack profit for capacity 'c' calculated from the first 'i' items.

So, for each item at index 'i' (0 <= i < items.length) and capacity 'c' (0 <= c <= capacity), we have two options:

Exclude the item at index 'i'. In this case, we will take whatever profit we get from the sub-array excluding this item => dp[i-1][c]
Include the item at index 'i' if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the remaining capacity and from remaining items => profit[i] + dp[i-1][c-weight[i]]
Finally, our optimal solution will be maximum of the above two values:

dp[i][c] = max (dp[i-1][c], profit[i] + dp[i-1][c-weight[i]])

Code

Here is the code for our bottom-up dynamic programming approach:

```
using namespace std;

#include <iostream>
#include <vector>

class Knapsack {
public:
  int solveKnapsack(const vector<int> &profits, const vector<int> &weights,
int capacity) {
    // basic checks
    if (capacity <= 0 || profits.empty() || weights.size() != profits.size())
{
      return 0;
    }

    int n = profits.size();
    vector<vector<int>> dp(n, vector<int>(capacity + 1));

    // populate the capacity=0 columns, with '0' capacity we have '0' profit
    for (int i = 0; i < n; i++) {
      dp[i][0] = 0;
    }

    // if we have only one weight, we will take it if it is not more than the
capacity
    for (int c = 0; c <= capacity; c++) {
      if (weights[0] <= c) {
        dp[0][c] = profits[0];
      }
    }

    // process all sub-arrays for all the capacities
    for (int i = 1; i < n; i++) {
      for (int c = 1; c <= capacity; c++) {
        int profit1 = 0, profit2 = 0;
        // include the item, if it is not more than the capacity
        if (weights[i] <= c) {
          profit1 = profits[i] + dp[i - 1][c - weights[i]];
        }
        // exclude the item
        profit2 = dp[i - 1][c];
        // take maximum
        dp[i][c] = max(profit1, profit2);
      }
    }

    // maximum profit will be at the bottom-right corner.
    return dp[n - 1][capacity];
  }
```

```
};

int main(int argc, char *argv[]) {
  Knapsack ks;
  vector<int> profits = {1, 6, 10, 16};
  vector<int> weights = {1, 2, 3, 5};
  int maxProfit = ks.solveKnapsack(profits, weights, 6);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
  maxProfit = ks.solveKnapsack(profits, weights, 7);
  cout << "Total knapsack profit ---> " << maxProfit << endl;
}
```

More Problems on 0 1 Knapsack:

Problem Statement 1:

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets is equal. Each element has to be added in either one of the two sets.

Example 1:
Input: {1, 2, 3, 4}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 4} & {2, 3}
Example 2:
Input: {1, 1, 3, 4, 7}
Output: True
Explanation: The given set can be partitioned into two subsets with equal sum: {1, 3, 4} & {1, 7}
Example 3:
Input: {2, 3, 4, 6}
Output: False
Explanation: The given set cannot be partitioned into two subsets with equal sum.
Example 4:
Input: {2, 4, 9, 1}
Output: False

Basic Solution

This problem follows the 0/1 Knapsack pattern. A basic brute-force solution could be to try all combinations of partitioning the given numbers into two sets to see if any pair of sets has an equal sum.

Assume if S represents the total sum of all the given numbers, then the two equal subsets must have a sum equal to S/2. This essentially transforms our problem to: "Find a subset of the given numbers that has a total sum of S/2".

So our brute-force algorithm will look like:

for each number 'i'
  create a new set which INCLUDES number 'i' if it does not exceed 'S/2', and recursively
    process the remaining numbers
  create a new set WITHOUT number 'i', and recursively process the remaining items
return true if any of the above sets has a sum equal to 'S/2', otherwise return false

Code
Here is the code for the brute-force solution:

```
using namespace std;

#include <iostream>
#include <vector>

class PartitionSet {
public:
  bool canPartition(const vector<int> &num) {
    int sum = 0;
    for (int i = 0; i < num.size(); i++) {
      sum += num[i];
    }

    // if 'sum' is a an odd number, we can't have two subsets with equal sum
    if (sum % 2 != 0) {
      return false;
    }

    return this->canPartitionRecursive(num, sum / 2, 0);
  }

private:
  bool canPartitionRecursive(const vector<int> &num, int sum, int
currentIndex) {
    // base check
    if (sum == 0) {
      return true;
    }

    if (num.empty() || currentIndex >= num.size()) {
      return false;
```

```
    }

    // recursive call after choosing the number at the currentIndex
    // if the number at currentIndex exceeds the sum, we shouldn't process
this
    if (num[currentIndex] <= sum) {
      if (canPartitionRecursive(num, sum - num[currentIndex], currentIndex +
1)) {
        return true;
      }
    }

    // recursive call after excluding the number at the currentIndex
    return canPartitionRecursive(num, sum, currentIndex + 1);
  }
};

int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 4};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 1, 3, 4, 7};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{2, 3, 4, 6};
  cout << ps.canPartition(num) << endl;
}
```

The time complexity of the above algorithm is exponential O(2^n), where ‘n’ represents the total number. The space complexity is O(n), this memory which will be used to store the recursion stack.

Top-down Dynamic Programming with Memoization
We can use memoization to overcome the overlapping sub-problems. As stated in previous lessons, memoization is when we store the results of all the previously solved sub-problems return the results from memory if we encounter a problem that has already been solved.

Since we need to store the results for every subset and for every possible sum, therefore we will be using a two-dimensional array to store the results of the solved sub-problems. The first dimension of the array will represent different subsets and the second dimension will represent different ‘sums’ that we can calculate from each subset. These two dimensions of the array can also be inferred from the two changing values (sum and currentIndex) in our recursive function canPartitionRecursive().

Code
Here is the code for Top-down DP with memoization:

```
using namespace std;

#include <iostream>
#include <vector>

class PartitionSet {
```

```
public:
  bool canPartition(const vector<int> &num) {
    int sum = 0;
    for (int i = 0; i < num.size(); i++) {
      sum += num[i];
    }

    // if 'sum' is a an odd number, we can't have two subsets with equal sum
    if (sum % 2 != 0) {
      return false;
    }

    vector<vector<int>> dp(num.size(), vector<int>(sum / 2 + 1, -1));
    return this->canPartitionRecursive(dp, num, sum / 2, 0);
  }

private:
  bool canPartitionRecursive(vector<vector<int>> &dp, const vector<int> &num,
int sum,
                             int currentIndex) {
    // base check
    if (sum == 0) {
      return true;
    }

    if (num.empty() || currentIndex >= num.size()) {
      return false;
    }

    // if we have not already processed a similar problem
    if (dp[currentIndex][sum] == -1) {
      // recursive call after choosing the number at the currentIndex
      // if the number at currentIndex exceeds the sum, we shouldn't process
this
      if (num[currentIndex] <= sum) {
        if (canPartitionRecursive(dp, num, sum - num[currentIndex],
currentIndex + 1)) {
          dp[currentIndex][sum] = 1;
          return true;
        }
      }

      // recursive call after excluding the number at the currentIndex
      dp[currentIndex][sum] = canPartitionRecursive(dp, num, sum,
currentIndex + 1) ? 1 : 0;
    }

    return dp[currentIndex][sum] == 1 ? true : false;
  }
};
```

```
int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 4};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 1, 3, 4, 7};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{2, 3, 4, 6};
  cout << ps.canPartition(num) << endl;
}
```

The above algorithm has time and space complexity of O(N*S), where 'N' represents total numbers and 'S' is the total sum of all the numbers.

Bottom-up Dynamic Programming

Let's try to populate our dp[][] array from the above solution, working in a bottom-up fashion. Essentially, we want to find if we can make all possible sums with every subset. This means, dp[i][s] will be 'true' if we can make sum 's' from the first 'i' numbers.

So, for each number at index 'i' (0 <= i < num.length) and sum 's' (0 <= s <= S/2), we have two options:

1. Exclude the number. In this case, we will see if we can get 's' from the subset excluding this number: dp[i-1][s]
2. Include the number if its value is not more than 's'. In this case, we will see if we can find a subset to get the remaining sum: dp[i-1][s-num[i]]

If either of the two above scenarios is true, we can find a subset of numbers with a sum equal to 's'.

Code

Here is the code for our bottom-up dynamic programming approach:

```
using namespace std;

#include <iostream>
#include <vector>

class PartitionSet {
public:
  bool canPartition(const vector<int> &num) {
    int n = num.size();
    // find the total sum
    int sum = 0;
    for (int i = 0; i < n; i++) {
      sum += num[i];
    }

    // if 'sum' is a an odd number, we can't have two subsets with same total
    if (sum % 2 != 0) {
      return false;
    }
```

```
    // we are trying to find a subset of given numbers that has a total sum
of 'sum/2'.
    sum /= 2;

    vector<vector<bool>> dp(n, vector<bool>(sum + 1));

    // populate the sum=0 columns, as we can always for '0' sum with an empty
set
    for (int i = 0; i < n; i++) {
      dp[i][0] = true;
    }

    // with only one number, we can form a subset only when the required sum
is
    // equal to its value
    for (int s = 1; s <= sum; s++) {
      dp[0][s] = (num[0] == s ? true : false);
    }

    // process all subsets for all sums
    for (int i = 1; i < n; i++) {
      for (int s = 1; s <= sum; s++) {
        // if we can get the sum 's' without the number at index 'i'
        if (dp[i - 1][s]) {
          dp[i][s] = dp[i - 1][s];
        } else if (s >= num[i]) { // else if we can find a subset to get the
remaining sum
          dp[i][s] = dp[i - 1][s - num[i]];
        }
      }
    }

    // the bottom-right corner will have our answer.
    return dp[n - 1][sum];
  }
};

int main(int argc, char *argv[]) {
  PartitionSet ps;
  vector<int> num = {1, 2, 3, 4};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{1, 1, 3, 4, 7};
  cout << ps.canPartition(num) << endl;
  num = vector<int>{2, 3, 4, 6};
  cout << ps.canPartition(num) << endl;
}
```

The above solution has time and space complexity of O(N*S), where 'N' represents total numbers and 'S' is the total sum of all the numbers.