Solution: 0/1 Knapsack

Let's solve the 0/1 Knapsack problem using the Dynamic Programming pattern.

We'll cover the following

Statement

You are given n items whose weights and values are known, as well as a knapsack to carry these items. The knapsack cannot carry more than a certain maximum weight, known as its **capacity**.

You need to maximize the total value of the items in your knapsack, while ensuring that the sum of the weights of the selected items does not exceed the capacity of the knapsack.

If there is no combination of weights whose sum is within the capacity constraint, return 0.

> **Notes:**
>
> 1. An item may not be broken up to fit into the knapsack, i.e., an item either goes into the knapsack in its entirety or not at all.
> 2. We may not add an item more than once to the knapsack.

Constraints:

- $1 \leq$ `capacity` $\leq 10^4$
- $1 \leq$ `values.length` $\leq 10^3$
- `weights.length` $==$ `values.length`
- $1 \leq$ `values[i]` $\leq 10^4$
- $1 \leq$ `weights[i]` $\leq$ `capacity`

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

A naive approach would be to generate all combinations of weights and calculate the profit of each combination. We would then choose the combination that yields the highest profit from among those that don't exceed the knapsack capacity.

For example, suppose we're given a knapsack with a capacity of 5, and the following list of values and weights:

- **values:** $[3, 5, 2, 7]$
- **weights:** $[3, 1, 2, 4]$

To find the maximum profit, we try all possible valid combinations, that is, whose weight does not exceed 5:

Weights	Values	Total Weight	Total Value
$3, 1$	$3, 5$	4	8
$3, 2$	$3, 2$	5	5
$1, 2$	$5, 2$	3	7
$1, 4$	$5, 7$	5	12

This means that we need some way to generate all possible valid combinations of weights.

The time complexity of the naive approach is $O(2^n)$, where n is the total number of values. The space complexity of this naive approach is $O(n)$.

Optimized approach using dynamic programming

Let's start solving this problem by implementing the naive solution.

> **Note:** In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Naive recursive solution

Here's how the naive recursive algorithm works:

- **Base case:** If there are no items left to add or the maximum capacity of the knapsack has been reached, we return 0.
- **Recursive case 1:** If the current item has a weight less than or equal to the remaining capacity of the knapsack, it can be added to the knapsack. At this point, we make two recursive calls to solve two sub-problems:
  - Find the maximum value of items we can include in the knapsack, while *including* the current item.
  - Find the maximum value of items we can include in the knapsack, while *excluding* the current item.

  Of the two options, we choose the one that yields the higher value.
- **Recursive case 2:** On the other hand, if the weight of the item is greater than the remaining capacity of the knapsack, the item cannot be added to the knapsack. Therefore, we use a recursive call to move on to the next item, *without* adding this item to the knapsack.

Java

```
1  class KnapsackProfit {
2
```

```
    public static int findMaxKnapsackProfitHelper(int capacity, int [] weights, int [] values, int n) {
        // Base cases
        if (n == 0 || capacity == 0) {
            return 0;
        }

        // Recursive cases
        // If the weight of the nth item is less than capacity, then:
        if (weights[n - 1] <= capacity) {
            // We either include the item and deduct the weight of the item from the knapsack capacity (t
            // or we don't include the item at all. We pick the option that yields the highest value.
            return Math.max(
                    values[n - 1] + findMaxKnapsackProfitHelper(capacity - weights[n - 1], weights, value
                    findMaxKnapsackProfitHelper(capacity, weights, values, n - 1));
        } else {
            // Item can't be added to our knapsack if its weight is greater than the capacity
            return findMaxKnapsackProfitHelper(capacity, weights, values, n - 1);
        }
    }

    public static int findMaxKnapsackProfit(int capacity, int [] weights, int [] values) {
        int n = weights.length;
        return findMaxKnapsackProfitHelper(capacity, weights, values, n);
    }

    // Driver code
```

0/1 Knapsack

> **Note:** Please observe that if you include the large test case currently commented out in the driver code, the solution is likely to time out. Try to solve this larger problem using the dynamic programming solutions provided below and see the difference.

Top-down memoization technique

Now, let's store overlapping subproblems through the memoization technique. It avoids repeatedly computing the solutions to the same subproblems by storing the solution of each subproblem the first time and looking up the solution when the subproblem recurs.

In the naive recursive solution, we observe that two variables change in each recursive call:

- `capacity`: The capacity of the knapsack.
- `n`: The number of items to consider.

We will use a 2D table with the above two indexes to uniquely identify a subproblem and store its solution. At any later time, when we encounter the same subproblem, we can fetch the stored result from the table with a $O(1)$ lookup instead of recalculating that subproblem.

Java

```
class KnapsackProfit {

    public static int findMaxKnapsackProfitHelper(int capacity, int[]  weights, int[]  values, int n, int
        // Base case
        if (n == 0 || capacity == 0) {
            return 0;
        }

        // If we have already solved this subproblem, fetch the result from memory
        if (dp[n][capacity] != -1) {
            return dp[n][capacity];
        }

        // Otherwise, we solve it and save the result in our lookup table
```

```
        if (weights[n - 1] <= capacity) {
            dp[n][capacity] = Math.max(
                    values[n - 1] + findMaxKnapsackProfitHelper(capacity - weights[n - 1], weights, value
                    findMaxKnapsackProfitHelper(capacity, weights, values, n - 1, dp));
            return dp[n][capacity];
        }

        dp[n][capacity] = findMaxKnapsackProfitHelper(capacity, weights, values, n - 1, dp);
        return dp[n][capacity];
    }

    public static int findMaxKnapsackProfit(int capacity, int[]  weights, int[]  values) {
        int n = weights.length;
        // Set up the dp table to store solutions to subproblems
```

0/1 Knapsack

> **Note:** Please observe that if you now uncomment the large test case in the driver code, the dynamic programming solution will still work.

Bottom-up tabulation technique

In the approach above, we need to first check whether a subproblem has already been solved before solving it. This adds extra overhead and slows down the algorithm. By nature, a recursive algorithm uses up space on the call stack. We can overcome both problems through the use of tabulation, which is an iterative, bottom-up dynamic programming approach, in which we first solve the smallest subproblems and then combine their solutions to solve larger subproblems until we have solved the overall problem.

First, we set up a 2D table, `dp`, of size (`n + 1`) $\times$ (`capacity + 1`). Each row corresponds to an item, and each column represents the remaining capacity of the knapsack at any given point. Therefore, `dp[i][j]` stores the value of the combination such that:

1. the combination consists of one or more of the first i items,
2. its weight does not exceed j, and,
3. there is no combination of these items that yields a higher value.

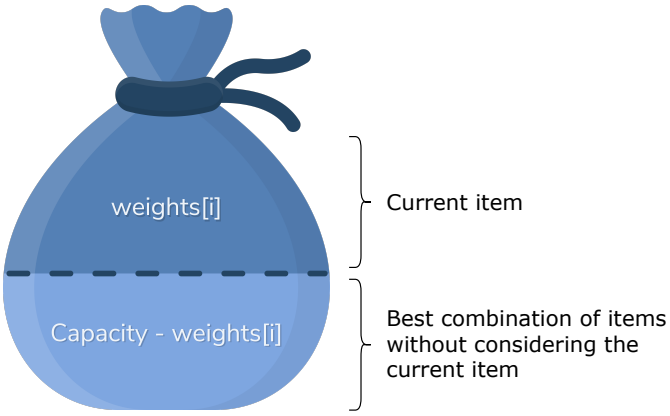
Here's how the algorithm works:

- First, we fill the first column with 0, since the value of the knapsack is 0 when the capacity is 0, i.e., none of the items fit. We also fill the first row with 0, to represent the base case, where none of the items has been added to the knapsack.
- For the remaining rows and columns, we iterate through them from top-left to bottom-right. For each entry in the table, we check if the weight of the current item is less than or equal to the current (available) capacity.
    - If it is, then we have two options, either to include the current item or to exclude it:
        1. To include the item, we first need to make space for it. We subtract the weight of the item, `weights[i]`, from the current capacity, `j`, to get the remaining capacity of the knapsack. To find the highest-value combination of items for the remaining capacity, we look up the solution to this subproblem from the `dp` table.

> Let's consider a concrete example: we are considering a knapsack of capacity 5, and the current item is the 4^{th} item with a weight of 2. Now, we need to find the solution to the

> following subproblem: "Given a knapsack of $capacity = 5 - 2 = 3$ and the items $\{1, 2, 3\}$, which combination of these items yields the highest value?"
>
> Since this is the highest possible value obtained through a combination of the items *prior* to considering the current item, we will find it in the previous row of the `dp` table, at the *remaining* capacity. In our example, we will look up `dp[3][3]`.
>
> We add this value to the value of our current item, `values[i]`, to obtain the highest possible value from a combination of items that *includes* our current item.

Below is a representation of the knapsack, showing the current item included.



The total value of this knapsack is: value of i^{th} item + value of best combination of items that fit in (Capacity - weight of i^{th} item)

2. To exclude the item, we simply look up the value of the knapsack at the *current* capacity from the previous row of the `dp` table. This is the highest possible value obtained at the current capacity without considering the current item.

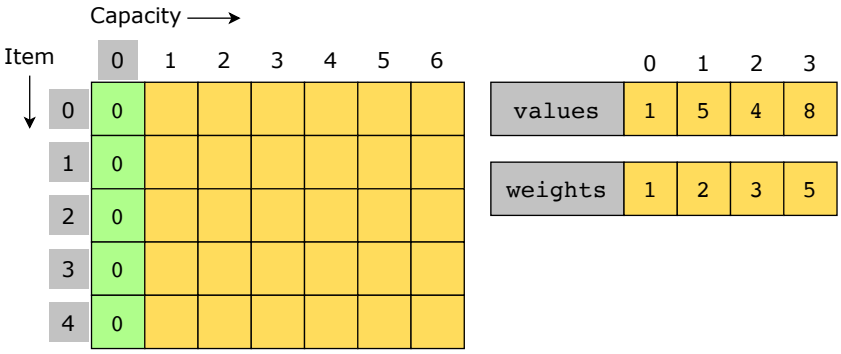
We choose the option that yields the maximum value and update `dp[i][j]` accordingly.

> **Note:** In general, the rules to update the cell in row `i`, column `j` may be coded as follows:
>
> ```
> dp[i][j] = Math.max(values[i-1]+ dp[i-1][j - weights[i-1]], dp[i-1][j])
> ```

- Otherwise, since the weight of the current item is greater than the current capacity, we cannot include it in the knapsack. Therefore, we reuse the solution to the subproblem `dp[i-1][j]`, i.e., the maximum value that can be obtained at the current capacity, without the current item.

- We repeat the steps above until all the entries of the `dp` table have been filled.
- We return the last entry, `dp[n][capacity]`, which contains the maximum profit for the given capacity.

The slides below illustrate how the algorithm runs:



For the base case, we make the first column equal to 0, since the knapsack can not have a value greater than 0 when its capacity is 0 itself.

Let's implement the algorithm as discussed above:

Java

```java
class KnapsackProfit {

    public static int findMaxKnapsackProfit(int capacity, int [] weights, int [] values) {
        // Create a table to hold intermediate values
        int n = weights.length;
        int[][] dp = new int[n + 1][capacity + 1];
        for(int[] row:dp) {
            Arrays.fill(row, 0);
        }

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= capacity; j++) {
                // Check if the weight of the current item is less than the current capacity
                if (weights[i - 1] <= j) {
                    dp[i][j] = Math.max(values[i - 1] + dp[i - 1][j - weights[i - 1]],
                            dp[i - 1][j]);
                }
                // We don't include the item if its weight is greater than the current capacity
                else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][capacity];
    }

    // Driver code
```

> **Note:** Please observe that if you now uncomment the large test case in the driver code, the dynamic programming solution will still work.

Space-optimized tabulation technique

You must have noticed in the above algorithm that to fill up a row, we only require the previous rows's values, that is, for filling the row against the i^{th} element, we require the values from the previous row representing the $(i-1)^{th}$ element. Therefore, there is no point in storing all the previous $(i-2)$ rows.

Here's how we can optimize our algorithm:

- We declare two arrays of size $(capacity + 1)$:
  - `dp`: This array represents the subproblems up to the previous, that is, the $(i-1)^{th}$ item. We will use the solutions of these subproblems to compute the solutions up to the i^{th} item.
  - `temp`: This array represents the subproblems up to the current, that is, the i^{th} item.
- We use nested loops to fill each entry of the `temp` array, using a modified version of the formula discussed above:

```
if (weights[i - 1] <= j) {
        temp[j] = Math.max(values[i - 1] + dp[j - weights[i - 1]], dp[j]);
}
else {
    temp[j] = dp[j];
}
```

- When the inner loop ends, all the subproblems up to the i^{th} item have been solved. Therefore, we set the `dp` array equal to the `temp` array, and move to the next item.
- We repeat the above process till we have processed all the items. The last entry, `dp[capacity]`, contains the highest possible value obtained by considering all items, so we return `dp[capacity]`.

Java

```java
class KnapsackProfit {

    public static int findMaxKnapsackProfit(int capacity, int[] weights, int[] values) {
        int n = weights.length;

        // previous (i-1)th row which will be used to fill up the current ith row
        int[] dp = new int[capacity + 1];
        Arrays.fill(dp, 0);

        // current ith row that will use the values of the previous (i-1)th row to fill itself
        int[] temp = new int[capacity + 1];
        Arrays.fill(temp, 0);

        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= capacity; j++) {
                if (weights[i - 1] <= j) {
                    temp[j] = Math.max(values[i - 1] + dp[j - weights[i - 1]], dp[j]);
                }
                else {
                    temp[j] = dp[j];
                }
            }

            // Setting the (i-1)th row equal to the ith row
            dp = temp.clone();
        }

        return dp[capacity];
```

0/1 Knapsack

> **Note:** Please observe that if you now uncomment the large test case in the driver code, the dynamic programming solution will still work.

Can we do even better?

The above solution uses two arrays for the $(i-1)^{th}$, and i^{th} rows, respectively. In every iteration of the outer loop, we pay the cost of copying the current temporary row. Let's see if we can use the same array throughout all the iterations.

We make two observations:

1. When computing cell values in, for example, the second row, we are looking up the value in the same column in the first row. This is true for both the cases considered in our logic (as seen by the presence of the `dp[j]` term):

Current item weighs *less* than the current capacity	Current item weighs *more* than the current capacity
`temp[j] = Math.max(values[i-1] + dp[j - weights[i-1]], dp[j])`	`temp[j] = dp[j]`

   This is true in the general case of the i^{th} row.

   Interestingly, in the single-array solution, the values of the previous row have not yet been overwritten, and instead of needing to look them up from a copy of this row, we can simply use the value of the cell itself. This means that we could rewrite these lines as follows:

Current item weighs *less* than the current capacity	Current item weighs *more* than the current capacity
`temp[j] = Math.max(values[i-1] + dp[j - weights[i-1]], temp[j])`	`temp[j] = temp[j]`

   Immediately, we see that when the current item weighs more than the current capacity, we don't actually need to do anything.
2. Secondly, we notice that when computing the solutions for i items, we need to look up solutions for $i-1$ items at various capacity levels. This means that, if we can, somehow, avoid overwriting the solutions for $i-1$ items *before* they're needed, we don't need to store them in a separate array.

Keeping these two observations in mind, an ingenious solution is to run the inner loop, not from 0 to the maximum capacity of the knapsack, but in the reverse direction. This is what the code will look like now:

```
public static int findMaxKnapsackProfit(int capacity, int [] weights, int [] values) {
    int n = weights.length;
    int[] dp = new int[capacity + 1];

    for (int i = 0; i <= capacity; i++) {
        dp[i] = 0;
    }

    for (int i = 0; i < n; i++) {
        for (int j = capacity; j >= 0; j--) {
            dp[j] = Math.max(values[i] + dp[j - weights[i]], dp[j]);
        }
```

```
    }

    return dp[capacity];
}
```

Notice that since there's nothing to do when the current item weighs more than the current capacity, we can eliminate `else` in the inner loop.

We can do one final optimization. Instead of running the inner loop all the way to 0 and checking every time to see whether the current item weighs more than the current capacity, we can simply stop the inner loop as soon as this condition fails.

With these changes, our fully optimized solution will be as follows:

Java

```java
class KnapsackProfit {

    public static int findMaxKnapsackProfit(int capacity, int [] weights, int [] values) {
        int n = weights.length;
        int[] dp = new int[capacity + 1];
        for (int i = 0; i <= capacity; i++) {
            dp[i] = 0;
        }

        for (int i = 0; i < n; i++) {
            System.out.println("\nComputing solutions for " + (i + 1) + " item(s):");
            for (int j = capacity; j >= weights[i]; j--) {
                System.out.println("\tat capacity " + j + ", max of: " + dp[j] + " and " + (values[i] + d
                dp[j] = Math.max(values[i] + dp[j - weights[i]], dp[j]);
            }

            System.out.println("Solutions for " + (i + 1) + " item(s): " + Arrays.toString(dp));
        }

        return dp[capacity];
    }

    // Driver code
    public static void main(String[] args) {

        int[][] weights = {
            { 1, 2, 3, 5 },
            { 4 }
```

0/1 Knapsack

> **Note:** Please observe that if you now uncomment the large test case in the driver code, the dynamic programming solution will still work.

Just the code

Here's the complete solution to this problem:

Java

```java
class KnapsackProfit {

    public static int findMaxKnapsackProfit(int capacity, int [] weights, int [] values) {
        int n = weights.length;
        int[] dp = new int[capacity + 1];
        for (int i = 0; i <= capacity; i++) {
            dp[i] = 0;
        }
```

```
9
10          for (int i = 0; i < n; i++) {
11              for (int j = capacity; j >= weights[i]; j--) {
12                  dp[j] = Math.max(values[i] + dp[j - weights[i]], dp[j]);
13              }
14          }
15
16          return dp[capacity];
17      }
18
19      // Driver code
20      public static void main(String[] args) {
21
22          int[][] weights = {
23              { 1, 2, 3, 5 },
24              { 4 },
25              { 2 },
```

0/1 Knapsack

Solution summary

To recap, the solution to this problem can be divided into the following steps:

1. Initialize an array of size c, where c is the total capacity of the knapsack. Set all its values to 0. After considering all the items, each entry in the array will store the highest value yielded at that capacity level.
2. Use an outer loop that runs n number of times, where n is the number of items.
3. Use an inner loop to update the entries of the array.
    I. Traverse the array in reverse order.
    II. The inner loop ends when the weight of the current item exceeds the current capacity.
    III. For each capacity level, find the value yielded by including and excluding the current item. Choose the higher of the two values.
4. Repeat steps 2–3 for all the items.
5. Return the last entry of the array.

Time complexity

The time complexity of this solution is $O(n \times c)$, where n is the number of items, and c is the total capacity of the knapsack.

Explanation:

- The outer loop runs in $O(n)$ time.
- In the worst case where all the items have a weight of 1, the inner loop runs in $O(c)$ time.

Space complexity