Chapter 24
The Knapsack Problem

24.1 Types of Knapsack Problems

The **knapsack problem** is an important computational problem that revolves around the following task: given a knapsack of capacity C and a collection of n items, where each item i has weight w_i and value v_i, find the maximum value you can store in your knapsack without exceeding its capacity. The following is an example of a knapsack problem:

> You are given a knapsack with weight capacity 11, as well as the following 5 items. Return the maximum value you can store in your knapsack without going over its capacity.
>
> | **Item** | 1 | 2 | 3 | 4 | 5 |
> |---|---|---|---|---|---|
> | **Weight** | 1 | 2 | 5 | 6 | 7 |
> | **Value** | 1 | 6 | 18 | 22 | 28 |

There are several variations of the knapsack problem that you may encounter, each with slightly different rules on how items can be selected. The following list summarizes the most common of these variations:

- **0-1 Knapsack**: Each of the n items are unique, and you must either take an item or leave it behind (hence the name 0-1). This is the standard variation of the knapsack problem, and it is assumed if the variation is not specified otherwise.
- **Fractional Knapsack**: Same as the 0-1 knapsack problem, but a fractional amount of each item can be taken.
- **Bounded Knapsack**: Each of the n items is not guaranteed to be unique, and may have a bounded quantity available (each item i also has a quantity q_i that identifies how many of that item exists).
- **Unbounded Knapsack**: Same as the bounded knapsack, but you have an unbounded (infinite) quantity of each item type.

Since each of these variants presents the problem in a different way, the best algorithm to use depends on the type of knapsack problem we are trying to solve. In this chapter, we will look at the four types of knapsack problems listed above and use our knowledge of algorithm families to develop an efficient solution for each.

24.2 0-1 Knapsack

※ 24.2.1 *Solving 0-1 Knapsack Using Brute Force*

The **0-1 knapsack problem** is the standard variation of the knapsack problem, where you are given a knapsack of capacity C and a collection of n *unique* and *unbreakable* items with value v_i and weight w_i. Your goal is to find the subset of items S that maximizes total value without exceeding the the capacity of the knapsack:

$$\text{maximize} \left(\text{value}(S) = \sum_{i \in S} v_i \right) \text{ such that } \left(\sum_{i \in S} w_i \leq C \right)$$

The simplest solution to the knapsack problem is to use brute force and generate the entire solution space, and then find the viable solution within this solution space with the largest overall value. In a 0-1 knapsack problem, each of the n items can either be *included* or *excluded* from our solution; since there are two possibilities for each item, there are a total of 2^n subsets that need to be generated. An example of the brute force approach is applied below:

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28

Knapsack Capacity: 11

Subset No.	**1**	**2**	**3**	**4**	**5**	**Weight**	**Value**	**Viable?**
1	0	0	0	0	0	0	0	✓
2	1	0	0	0	0	1	1	✓
3	0	1	0	0	0	2	6	✓
4	1	1	0	0	0	3	7	✓
5	0	0	1	0	0	5	18	✓
6	1	0	1	0	0	6	19	✓
7	0	0	0	1	0	6	22	✓
8	1	0	0	1	0	7	23	✓
9	0	1	1	0	0	7	24	✓
10	1	1	1	0	0	8	25	✓
11	0	0	0	0	1	7	28	✓
12	0	1	0	1	0	8	28	✓
13	1	0	0	0	1	8	29	✓
14	1	1	0	1	0	9	29	✓
15	0	1	0	0	1	9	34	✓
16	1	1	0	0	1	10	35	✓
17	**0**	**0**	**1**	**1**	**0**	**11**	**40**	✓
18	1	0	1	1	0	12	41	✗
19	0	0	1	0	1	12	46	✗
20	0	1	1	1	0	13	46	✗
21	1	0	1	0	1	13	47	✗
22	1	1	1	1	0	14	47	✗
23	0	0	0	1	1	13	50	✗
24	1	0	0	1	1	14	51	✗
25	0	1	1	0	1	14	52	✗
26	1	1	1	0	1	15	53	✗
27	0	1	0	1	1	15	56	✗
28	1	1	0	1	1	16	57	✗
29	0	0	1	1	1	18	68	✗
30	1	0	1	1	1	19	69	✗
31	0	1	1	1	1	20	74	✗
32	1	1	1	1	1	21	75	✗

Here, the optimal solution would be to take items 3 and 4 for a total value of $18 + 22 = 40$ and a weight of $5 + 6 = 11$. However, we had to enumerate over 2^n possible subsets and sum up the weight and value of each subset to get our solution — since each summation takes $\Theta(n)$ time, the overall time complexity of the brute force approach is $\Theta(n2^n)$. Is there a way to do better?

※ 24.2.2 *Why Greedy Fails for 0-1 Knapsack*

An intuitive strategy would be to try a greedy approach first, since it is simpler and often hard to beat if it works. Since we want to maximize the total value of our items while minimizing total weight, there are two greedy strategies that immediately jump out:

1. Greedily select the items with the *highest value* first.
2. Greedily select the items with the *lowest weight* first.

However, if we apply either of these greedy strategies to the example, we will see that they do not work. If we greedily select items with higher value first, we would take item 5 first, and then item 2 (since this is the best we can do without going over capacity after selecting item 5), and then item 1. The total value of this solution is $28 + 6 + 1 = 35$, which is not optimal. Similarly, if we greedily select items with lower weight first, we would take item 1 first, then item 2, then item 3. The value of this solution is $1 + 6 + 18 = 25$, which is also not optimal.

In general, the approach of selecting higher value items first may not work if large items have high value (such as item 5), which forces us to incur a large weight cost after greedily selecting a high-value item. On the other hand, the approach of selecting lower weight items first could fail if small items have low value (such as item 1), which causes us to gain little value for the weight we take on. Therefore, to find a valid greedy approach (if it exists), we would need to consider both the weight and value of each item in tandem.

This leads us to a third approach: greedily selecting items with the highest *value density* (or *value-weight ratio*) first. The value density of an item is its value divided by its weight (e.g., value/weight). This allows us to maximize the value we get from each unit of weight we take on, instead of blindly taking on valuable or light items without considering the other variable. However, if we try this strategy on the example, it also does not produce an optimal solution: we would end up taking item 5 first, then 2, then 1 for a total value of 35.

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28
Ratio	1	3	3.6	3.67	4

Unfortunately, this greedy strategy could still fail for the 0-1 knapsack problem if large items have a high value density (such as item 5). By taking item 5, we were prevented from taking the next best item (item 4) because our knapsack could not support it.

If considering the value-weight ratio failed to find a valid greedy solution, is there even a way to do better? It actually turns out that the answer is no: *there exists no greedy strategy that guarantees an optimal solution for the 0-1 knapsack problem*! If we wanted to solve 0-1 knapsack, we would have to rely on a slightly more complicated algorithm family.

※ 24.2.3 *Solving 0-1 Knapsack Using Dynamic Programming*

One approach that we can use to solve the 0-1 knapsack problem is dynamic programming, as it is perfectly follows the decision making problem structure discussed in the previous chapter. To solve 0-1 knapsack using dynamic programming, we will iterate over each item and use the solutions of previously computed subproblems to determine we should take the item or leave it behind. To illustrate this, let's define $F(i,c)$ as the optimal value attainable from considering the first i items with a knapsack of capacity c, with w_i and v_i as the weight and value of the i^{th} item. We can express $F(i,c)$ recursively by noticing that there are only two possibilities for each item i: it can either be taken or left behind.

- If item i is *left behind*, the maximum value attainable must be equal to $F(i-1,c)$, or the optimal solution of the first $i-1$ items with a knapsack capacity c. This is because, if the i^{th} item is not included, the best you can do is equal to the solution of the subproblem that does not consider this item.
- If item i is *taken*, the maximum value attainable must be equal to $F(i-1,c-w_i)+v_i$, or the optimal solution of the first $i-1$ items with a knapsack capacity $c-w_i$, plus the value of the i^{th} item. This is because, if you want to include the i^{th} item in your knapsack, the earlier items can only take up at most $c-w_i$ weight so you don't go over capacity.

Therefore, to compute $F(i,c)$, we simply have to take the better of these two values. The base case of this problem occurs when $i=0$ (no items to choose from), which implies that the best value attainable is also 0 (since you cannot take any items). Putting this all together, we end up with the following recurrence relation:

$$F(i,c) = \begin{cases} 0, & \text{if } i = 0 \\ \max(F(i-1,c),\ F(i-1,c-w_i)+v_i), & \text{if } i > 0,\ c > 0 \end{cases}$$

A top-down solution is shown below — in this solution, we simply make the recursive calls reflected in the above recurrence relation and write our solutions to a memo, where memo[i][c] stores the solution to $F(i,c)$.

```
1   struct Item {
2     int32_t weight;
3     int32_t value;
4   };
5
6   int32_t knapsack_01(int32_t c, const std::vector<Item>& items) {
7     const size_t n = items.size();
8     std::vector<std::vector<int32_t>> memo(n + 1, std::vector<int32_t>(c + 1, -1));
9     return knapsack_helper(c, items, n - 1, memo);
10  } // knapsack_01()
11
12  int32_t knapsack_helper(int32_t c, const std::vector<Item>& items, int32_t i,
13                          std::vector<std::vector<int32_t>>& memo) {
14    if (i < 0) {
15      return 0;
16    } // if
17    if (memo[i][c] != -1) {
18      return memo[i][c];
19    } // if
20    int val_excluded = knapsack_helper(c, items, i - 1, memo);
21    // optimization: if weight of item already over capacity, it must be excluded
22    if (items[i].weight > c) {
23      return memo[i][c] = val_excluded;
24    } // if
25    int val_included = knapsack_helper(c - items[i].weight, items, i - 1, memo) + items[i].value;
26    return memo[i][c] = std::max(val_excluded, val_included);
27  } // knapsack_helper()
```

A bottom-up approach follows the same idea, but starts with the base cases and builds upwards toward the final solution. To implemement a bottom-up solution, we will need a double nested loop: an outer loop that iterates over all the items, and an inner loop that iterates over all knapsacks from capacity 0 to c. A bottom-up implementation is shown below:

```
1   struct Item {
2     int32_t weight;
3     int32_t value;
4   };
5
6   int32_t knapsack_01(int32_t c, const std::vector<Item>& items) {
7     const size_t n = items.size();
8     std::vector<std::vector<int32_t>> memo(n + 1, std::vector<int32_t>(c + 1, 0));
9     for (size_t i = 0; i < n; ++i) {
10      for (size_t j = 0; j < c + 1; ++j) {
11        if (j < items[i].weight) {
12          // weight of item already over capacity, exclude
13          memo[i + 1][j] = memo[i][j];
14        } // if
15        else {
16          memo[i + 1][j] = std::max(memo[i][j], memo[i][j - items[i].weight] + items[i].value);
17        } // else
18      } // for j
19    } // for i
20    return memo[n][c];
21  } // knapsack_01()
```

To illustrate how this works, we will use the example provided. We start off by initiating a memo of size $(n+1)\times(c+1)$ to store the solutions of our subproblems. Since any subproblem where $n = 0$ has a solution of 0, row 0 of our memo must be entirely filled out with 0s.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1												
2												
3												
4												
5												

We will then consider row 1, which considers all items up to item 1. Item 1 has a weight of 1, which does not fit in a knapsack with capacity 0, so memo[1][0] gets set to 0.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2												
3												
4												
5												

However, if our knapsack capacity increases to 1, we would be able to add item 1 to our knapsack. This increases our value to 1, which is assigned to memo[1][1].

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	**1**										
2												
3												
4												
5												

Since row 1 involves subproblems that only consider up to item 1, the optimal solution will always stay at 1 for these subproblems even as capacity increases, as you only have one item (of value 1) to pick from. The remainder of this row is filled out as follows:

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2												
3												
4												
5												

Next, we move on to row 2, which brings item 2 (of weight 2 and value 6) into consideration. Similar to before, if our knapsack capacity is 0, then we cannot add any items, and our value is consequently 0. Therefore, memo[2][0] gets assigned to a value of 0. Notice that this gets handled by the `if` check on lines 11-13 of the bottom-up code; since the capacity (0) is less than the weight of item 2 (6), we just assign memo[2][0] to the value of memo[1][0].

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	**0**											
3												
4												
5												

This also applies to the solution of memo[2][1]. Since item 2 does not fit if our knapsack capacity is 1, we simply assign memo[2][1] to the value of the cell directly above, or memo[1][1], which is the optimal value without considering item 2.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	**1**										
3												
4												
5												

When we move to a knapsack capacity of 2, however, item 2 is able to fit in our knapsack. Thus, we are left with a choice: we can either exclude item 2 or include it in our knapsack. If we exclude item 2, then the best we can do is the value of the cell directly above, or memo[1][2]. This is because memo[1][2] stores the optimal solution of considering only items up to item 1 with a knapsack of size 2 (i.e., the same problem, but with item 2 omitted).

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	**1**	1	1	1	1	1	1	1	1	1
2	0	1										
3												
4												
5												

On the other hand, if we include item 2, then the best we can do is the value of item 2, plus the best value of choosing the remaining items with the capacity left over. In this case, the capacity left over after adding item 2 is 0 (which we know from memo[1][0]), so we will add the value of item 2 to memo[1][0] to get the best value attainable from including item 2.

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	**6**	18	22	28

+

Capacity

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	0	0
1	**0**	1	1	1	1	1	1	1	1	1	1	1
2	0	1										
3												
4												
5												

Item

If we include item 2, the best value we can get is 6, which is better than the best value of 1 if we exclude item 2. Therefore, the value of memo[2][2] is determined to be 6.

Capacity

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	**6**									
3												
4												
5												

Item

This process is repeated for the rest of the table. For example, here are the two values we need to reference to solve memo[5][11]: the best value of excluding item 5 is 40 (memo[4][11]), and the best value of including item 5 is 7 + 28 = 35 (value[5] + memo[4][11 - weight[5]]). Since 40 is better than 35, it is the solution for memo[5][11] (and also our entire problem).

Capacity

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	29	29	**40**
5	0	1	6	7	7	18	22	28	29	34	35	

Item

The best value attainable from excluding item 5 is 40.

Capacity

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	**7**	18	22	23	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	

Item

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	**28**

+

The best value attainable from including item 5 is 7 + 28 = 35.

Capacity

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Item

40 is better than 35, so we exclude item 5 to get our solution of 40.

Given a knapsack of capacity C and n items to choose from, a dynamic programming solution to the 0-1 knapsack problem takes $\Theta(nC)$ time, since there are at most $\Theta(nC)$ subproblems that need to be solved once, each taking $\Theta(1)$ time. The auxiliary space used by the above solution is also $\Theta(nC)$, since a memo of this size is declared. However, if you are using a bottom-up approach, you can reduce this space complexity to $\Theta(C)$ using the memo optimization strategy covered in the previous chapter (since you only need to keep track of the two most recent rows to solve any subproblem).

If we wanted to find the actual set of items we should take instead of just the optimal value, we can simply work backward using our completed memo. To reconstruct the knapsack items, we start at the solution cell and consider whether each item should be taken or left behind by referencing the subproblems where the item is and isn't taken. We then use this decision to walk through our memo, all the way back to a base case. For instance, to identify which items we should take for the provided example, we would start at memo[5][11] and first determine if item 5 should be taken. The maximum value attainable from including item 5 is 35 (memo$[i-1][c-w_i] + v_i$ = memo[4][4] + 28 = 35), and the maximum value attainable from excluding item 5 is 40 (memo$[i-1][c]$ = memo[4][11] = 40), so item 5 should be excluded.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	29	29	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Item	1	2	3	4	5
Taken?					✗

Now that we have determined that item 5 should not be taken, we will now need to determine if item 4 should be included. If item 4 is excluded, then the best value we can still obtain is 25, from the value of memo[3][11]. However, if item 4 is included, the best value we can obtain is memo[3][5] + 22 = 18 + 22 = 40 (this is because taking item 4 would leave us with a remaining capacity of 11 - 6 = 5, and we would gain 22 units of value). Therefore, item 4 should be included in our knapsack.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	**18**	←19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	~~29~~	~~29~~	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Item	1	2	3	4	5
Taken?				✓	✗

We now move on to item 3. If we exclude item 3, then the best value we can attain is 7, from memo[2][5]. If include item 3, then the best value we can attain is memo[2][0] + 18 = 18. Therefore, item 3 should be included in our knapsack.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	**0**	←1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	~~7~~	~~7~~	**18**	←19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	~~29~~	~~29~~	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Item	1	2	3	4	5
Taken?			✓	✓	✗

Our knapsack is out of space, so we can immediately conclude that items 1 and 2 must not be included. The final solution would therefore be to take items 3 and 4 to obtain our optimal value of 40.

Capacity

Item	0	1	2	3	4	5	6	7	8	9	10	11
0	**0**	↖0	0	0	0	0	0	0	0	0	0	0
1	**0**	↖1	1	1	1	1	1	1	1	1	1	1
2	**0**	←1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	~~7~~	~~7~~	**18**	←19	24	25	25	25	25
4	0	1	6	7	7	18	22	23	28	~~29~~	~~29~~	**40**
5	0	1	6	7	7	18	22	28	29	34	35	**40**

Item	1	2	3	4	5
Taken?	✗	✗	✓	✓	✗

The implementation of the knapsack reconstruction process is shown below.

```
struct Item {
  int32_t weight;
  int32_t value;
};

std::vector<bool> reconstruct_knapsack(int32_t c, const std::vector<Item>& items) {
  const size_t n = items.size();
  size_t rem_cap = c; // remaining capacity
  std::vector<bool> taken(n, false);
  for (int32_t i = n - 1; i >= 0; --i) {
    if (items[i].weight <= rem_cap) {
      if (memo[i][rem_cap - items[i].weight] + items[i].value >= memo[i][rem_cap]) {
        taken[i] = true;
        rem_cap -= items[i].weight;
      } // if
    } // if
  } // for i
  return taken;
} // reconstruct_knapsack()
```

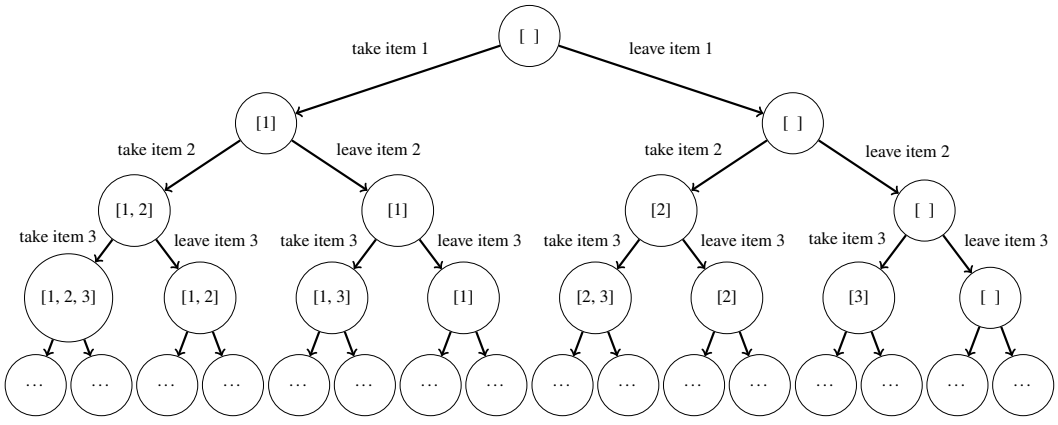
Since reconstructing the knapsack from a completed memo only requires a single iteration over all the items, this process has a time complexity of $\Theta(n)$. This doesn't affect the overall time complexity of solving the entire 0-1 knapsack problem, however, since you still have to complete the memo before you can reconstruct the solution.

※ 24.2.4 *Solving 0-1 Knapsack Using Branch and Bound*

The knapsack problem is an example of an optimization problem, so branch and bound can also be used. Since knapsack is a maximization problem, we will need an upper bound that identifies an optimistic estimate for each partial solution and a lower bound that identifies the best complete solution encountered so far. The components needed to solve 0-1 knapsack using branch and bound are summarized below:

- The `promising()` function should return `true` if the collection of items chosen in a partial solution does not overfill the knapsack (i.e., total weight $< C$).
- The lower bound should be initialized to the *highest possible underestimate of the optimal solution* and then updated if a better solution is found. After branch and bound runs to completion, the value of the lower bound is the optimal solution.
  - We want to start with an underestimate because we don't want to prune off the actual best solution — if we end up with a starting estimate that is too high, then our algorithm would think that the optimal solution isn't good enough and would therefore overprune. On the other hand, we still want this underestimate to be as high as reasonably possible (without sacrificing performance to obtain this estimate) since a closer bound allows us to prune faster.
  - A good strategy is to use the greedy 0-1 knapsack solution (by value density) as the initial lower bound. Although the greedy approach does not guarantee an optimal solution for 0-1 knapsack, it can be used to produce a "close enough" underestimate relatively quickly.
- The upper bound for any partial solution should be calculated by adding the sum of the existing items' values with an *overestimate* of the value that can fit in the remaining capacity.
  - We want an overestimate because the upper bound serves as a limit on the best outcome from extending a partial solution. If the upper bound is less than the lower bound (which is the best solution we know so far), then the partial solution *cannot* be optimal and thus can be pruned.
  - A good strategy is to use the greedy *fractional* knapsack solution of the remaining items (by value density) as the upper bound. (A fractional knapsack allows you to break items apart and take only a part of an item, where the proportion of weight taken reflects the proportion of value obtained.) Because the fractional knapsack problem gives you the ability to break items apart for partial value, its solution cannot be worse than that of the 0-1 knapsack problem (and would thus be a good bound on the value of continuing a partial solution).

It should be noted that, unlike TSP, there is no additional check we need to make to determine if a promising solution is also a valid solution (since any promising solution can be a valid solution). Furthermore, we only need to generate all *combinations* of items rather than permutations, as the ordering of items in our knapsack does not matter. The state-space tree of the 0-1 knapsack problem is shown below, where each branch represents a decision to include (left branch) or exclude (right branch) the next item. This tree represents the solution space that we are exploring while running our branch-and-bound algorithm.



Each of the nodes of the tree represents the "state" of a partial solution that our algorithm may encounter. Since there are two choices we can make at each state, the total number of states in the tree is 2^n when given n items to choose from. Furthermore, if we assume that a greedy fractional knapsack problem is solved at each state to estimate an upper bound, the overall worst-case time complexity of the branch-and-bound approach is bounded by $O(n2^n)$, since each fractional knapsack problem takes $O(n)$ time to solve.[1] However, as mentioned in earlier chapters, this worst-case complexity bound isn't reflective of the actual performance of branch-and-bound on a 0-1 knapsack; this is because the worst case only occurs if we get unlucky while pruning and end up exploring most of the search space, a scenario that almost never happens in practice.

24.3 Fractional Knapsack

The **fractional knapsack problem** is a variation of the 0-1 knapsack problem that allows you to take a portion of an item for partial value. For instance, if you had a knapsack of capacity 5 and an item with weight 15, you could break that item up and only take the portion that fits (for $5/15 = 1/3^{\text{rd}}$ of the value). To illustrate this, consider the same set of items provided earlier:

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28

Knapsack Capacity: 11

Previously, the optimal 0-1 knapsack solution was to take items 3 and 4 for a combined value of 40. However, if we allow items to be broken up, the optimal solution would then be to take item 5 and $2/3^{\text{rd}}$ of item 4 for a total value of $28 + (2/3)(22) = 42\ ^2/_3$.

Unlike the 0-1 knapsack problem, greedily selecting items with the highest value density for the fractional knapsack problem actually *guarantees an optimal solution*! Consider the example above: if we compute the value density of each of the items, we would first add item 5 to our knapsack, and then item 4 (until we run out of capacity): this gives us the optimal solution of $42\ ^2/_3$. In fact, we can prove that the greedy approach (by value density) works for all cases of the fractional knapsack problem:

Frame the problem as one in which only one subproblem remains after a greedy choice is made.

In this case, the greedy choice would be to consider the item with the highest value density and add as much of it as we can to our knapsack without going over capacity. After making this greedy choice, we are left with the subproblem of solving the fractional knapsack problem using the remaining items with the capacity left over.

Show that the greedy-choice property is satisfied, such that at least one optimal solution to the original problem includes the first greedy choice made by the greedy algorithm.

This was the step that failed for the 0-1 knapsack problem, as there was no guarantee that the first greedy choice was optimal. However, things are a bit different if we allow items to be broken up for partial value. We can use a proof by contradiction to show that the greedy choice for the fractional knapsack problem is included in at least one optimal solution.

Let's assume that some item i has the highest value density $\frac{v_i}{w_i}$ among all the items available. Since the greedy approach takes as much of item i as possible, we first assume that the optimal solution does not take as much of item i as possible (this also implies the knapsack is completely filled, since you can add more of item i to get a better solution if the knapsack were not full). Since item i has the highest value density, the optimal solution must have taken some other item j such that $\frac{v_j}{w_j} \leq \frac{v_i}{w_i}$. As a result, we can take out any x weight of item j in

[1] This assumes a standard implementation of the fractional knapsack algorithm where the items have already been sorted by value density. If the items are not already sorted by value density, then they should be sorted, which takes $\Theta(n\log(n))$ time. However, this sorting step is only done once at the beginning of the algorithm, so the $\Theta(n\log(n))$ work is a lower-order term and does not contribute to the overall time complexity class of the problem. (If you are reading this and don't know what fractional knapsack even is, don't worry; it will be covered in the next section.)

the knapsack and replace it with x weight of item i; this changes the value of our knapsack by $x\left(\frac{v_i}{w_i} - \frac{v_j}{w_j}\right)$, which is ≥ 0. This produces a contradiction, as we can replace the contents of a knapsack without item i with item i to get a total value that is no worse than our original solution! As a result, we have proven that there exists an optimal solution that includes the item with the highest value density, and that the greedy-choice property holds.

Show that the problem exhibits an optimal substructure, where the optimal solution of the remaining subproblem can be combined with the greedy choice to obtain an optimal solution for the original problem.

Let S represent the original fractional knapsack problem we are trying to solve, which has optimal solution V for weight W. If the greedy choice selects item i, an optimal substructure would imply that the solution to the subproblem $S' = S - \{i\}$ with knapsack capacity $W' = W - W_i$ is equal to $V' = V - V_i$, where W_i and V_i represent the weight and value of item i.

We will now use a proof by contradiction to show that an optimal substructure must exist. Let's assume that V' is not the optimal solution of S' and that another solution V'' exists that is better than V'. If that were the case, we could simply take the items that give us a value of V'' and then add item i to get a value of $V'' + V_i$ for our original problem S, which is better than $V' + V_i$. However, we know that the optimal solution to S is $V' + V_i = V$, which is a contradiction! As a result, we can conclude that the optimal solution of S' must be $V - V_i$ … otherwise, we would be able to construct a solution for S that is better than the optimal solution of V. We have successfully proven that the fractional knapsack problem exhibits an optimal substructure.

Because the strategy of greedily selecting items with the highest value density satisfies the greedy-choice property and exhibits optimal substructure, we can conclude that it guarantees an optimal solution for any fractional knapsack problem.

Similar to other greedy algorithms, the greedy solution to fractional knapsack boils down to the following steps:

- Sort the items in order of value density (this allows us to quickly determine which item we should consider at each step).
- Greedily select the available item with the largest value density and add as much of it to the knapsack without going over capacity. Repeat this until the knapsack is full.

The code for a greedy implementation is shown below:

```
1   struct Item {
2     int32_t weight;
3     int32_t value;
4   };
5
6   bool compare(const Item& i1, const Item& i2) {
7     double ratio1 = static_cast<double>(i1.value) / i1.weight;
8     double ratio2 = static_cast<double>(i2.value) / i2.weight;
9     return ratio1 > ratio2;
10  }; // compare()
11
12  double fractional_knapsack(int32_t c, std::vector<Item>& items) {
13    // sort items in decreasing order of value density
14    std::sort(items.begin(), items.end(), compare);
15    int32_t curr_weight = 0;
16    double curr_value = 0.0;
17    for (size_t i = 0; i < items.size(); ++i) {
18      // add item completely if possible
19      if (curr_weight + items[i].weight <= c) {
20        curr_weight += items[i].weight;
21        curr_value += items[i].value;
22      } // if
23      // otherwise, add fractional part of it
24      else {
25        double partial_weight = c - current_weight;
26        curr_value += (items[i].value * partial_weight / items[i].weight);
27        break;
28      } // else
29    } // for i
30    return curr_value;
31  } // fractional_knapsack()
```

Given n items to choose from, the sorting step takes $\Theta(n\log(n))$ time and the item selection step takes $\Theta(n)$ time. Combining these two steps together, we get an overall time complexity of $\Theta(n\log(n))$, since the sorting step is the bottleneck of the algorithm (everything else is a lower order term). However, if the items are already presorted, then the time complexity would drop to $\Theta(n)$, as a single linear pass would be sufficient for computing the optimal solution.

24.4 Bounded Knapsack (⁎)

The standard 0-1 knapsack problem we discussed earlier involved only a *single* instance of each item. However, what if *identical copies* of each item are available? This leads us to a variation of the standard 0-1 knapsack problem known as the **bounded knapsack problem**, which restricts the number of copies of each item i to some non-negative integer q_i. The following is an example of a bounded knapsack problem, where you are given 8 copies of item 1, 7 copies of item 2, 3 copies of item 3, 2 copies of item 4, 6 copies of item 5, and a knapsack capacity of 45.

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28
Quantity	8	7	3	2	6

Knapsack Capacity: 45

The optimal solution for this knapsack problem can be attained by taking 5 copies of item 5 (for a total value of $28 \times 5 = 140$ using weight $7 \times 5 = 35$) and then taking 2 copies of item 3 (for a total value of $18 \times 2 = 36$ using weight $5 \times 2 = 10$), which results in an overall value of $140 + 36 = 176$. However, this solution isn't inherently obvious, and it cannot be attained using the greedy by value density approach (which would give us a suboptimal solution of 175). How can we devise an algorithm that can efficiently solve the bounded knapsack problem?

One simple yet effective approach is to "refactor" the bounded knapsack problem into a standard 0-1 knapsack problem by considering each of the duplicate items as its own separate item. Instead of evaluating the above example as a collection of 5 items to choose from, we will instead treat the input as a collection of 26 items: 8 of which have weight 1 and value 1, 7 of which have weight 2 and value 6, 3 of which have weight 5 and value 18, and so on.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…	26
Weight	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	5	5	5	6	…	7
Value	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	18	18	18	22	…	28

However, notice that the number of subproblems we need to consider (and thus the size of the memo) can get quite large as a consequence of this strategy. Previously, the time complexity of the standard 0-1 knapsack problem using dynamic programming was $\Theta(nC)$, where n is the number of items we are given and C is the capacity of the knapsack. However, if we convert the bounded knapsack problem into a 0-1 knapsack problem, the overall time complexity would become $\Theta(NC)$, where $N = \sum_{i=1}^{n} q_i$.

Exactly how bad can this get? Even though the quantity of any item q_i can take on any non-negative integer in the bounded knapsack problem, it practically has a maximum value, since no more than $\lfloor c/w_i \rfloor$ copies of item i can fit in the knapsack.

$$q_i \leq \lfloor \tfrac{c}{w_i} \rfloor,\ i = 1, \ldots, n$$

If q_i were somehow larger than $\lfloor c/w_i \rfloor$ for some item i, we can simply replace q_i with $\lfloor c/w_i \rfloor$ without changing the solution, since any additional copies would have no chance of fitting in the knapsack in the first place (and therefore can be ignored). If we consider this cap on the quantity of items available, we can conclude that

$$N \leq \lfloor \tfrac{c}{w_1} \rfloor + \lfloor \tfrac{c}{w_2} \rfloor + \ldots + \lfloor \tfrac{c}{w_n} \rfloor < nC$$

In the worst case, $N = O(nC)$. Therefore, the worst-case time complexity of solving the bounded knapsack problem, if you directly convert it into a 0-1 knapsack problem by considering each item individually, is $O(NC) = O(nC \times C) = O(nC^2)$. However, it turns out there is a more efficient way to convert the bounded knapsack problem into a 0-1 knapsack problem.

To come up with this better method, note that every number can be written as the sum of distinct powers of two:

$$\begin{aligned}
1 &= 2^0 \\
2 &= 2^1 \\
3 &= 2^1 + 2^0 \\
4 &= 2^2 \\
5 &= 2^2 + 2^0 \\
6 &= 2^2 + 2^1 \\
7 &= 2^2 + 2^1 + 2^0 \\
8 &= 2^3
\end{aligned}$$

Since the inclusion or exclusion of each power of two is a binary decision for each sum, we can consider our items in groups of *powers of two* rather than individually. This gives us the same behavior as considering every item individually, since any value we want can be simulated by combining these powers of two. For instance, we are given 8 copies of item 1 in the previous example. The naïve approach would consider all 8 copies individually as its own item.

Item	1	2	3	4	5	6	7	8
Weight	1	1	1	1	1	1	1	1
Value	1	1	1	1	1	1	1	1

However, the smarter approach would aggregate the items into groups of powers of two (combining any remaining items left over as its own item at the very end):

Item	1	2	3	4	5	6	7	8
Weight	1	1	1	1	1	1	1	1
Value	1	1	1	1	1	1	1	1

Item	1	2	3	4
Weight	1	2	4	1
Value	1	2	4	1

This still allows us to take any valid amount of the item that we want, just without having to increase our table size to $\sum_{i=1}^{n} q_i$. For instance, if we wanted to take 6 copies of the original item, we can replicate this behavior using the new table by taking the aggregated items of weights 2 and 4 (instead of considering each of the 6 items as separate and distinct, which we did before).

Using this strategy, we can convert the previous table of 26 columns into one with 14, halving a dimension of our problem!

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Weight	1	2	4	1	2	4	8	5	10	6	6	7	14	21
Value	1	2	4	1	6	12	24	18	36	22	22	28	56	84

What is the worst-case time complexity of this modified strategy? Since we are grouping each item using powers of two, the total number of item groups N we may end up with is at most $\sum_{i=1}^{n} \left(\lfloor \log_2(q_i) \rfloor + 1 \right)$ instead of $\sum_{i=1}^{n} q_i$ (since each item i can be split into $\log_2(q_i)$ groups, plus one for any amount left over after the largest possible power of two). Using the same rules as before and applying some logarithm rules, we can see that N is at worst $O(n\log(C))$ using this approach, giving our converted 0-1 knapsack problem a worst-case time complexity of $O(NC) = O(n\log(C) \times C) = O(nC\log(C))$. This is better than the $O(nC^2)$ time we obtained earlier from considering every copy as a separate item! An implementation of this solution is shown below (reusing the `knapsack_01()` method implemented in section 24.2.3):

```
1   struct Item {
2     int32_t weight;
3     int32_t value;
4   };
5
6   struct ItemWithQuantity {
7     int32_t weight;
8     int32_t value;
9     int32_t quantity;
10  };
11
12  int32_t bounded_knapsack(int32_t c, const std::vector<ItemWithQuantity>& items) {
13    std::vector<Item> translated_items = translate_to_01(items);
14    return knapsack_01(c, translated_items);
15  } // bounded_knapsack()
16
17  std::vector<Item> translate_to_01(const std::vector<ItemWithQuantity>& items) {
18    std::vector<Item> items_01;
19    for (const ItemWithQuantity& item : items) {
20      int32_t rem_quantity = item.quantity;
21      int32_t curr_power_of_two = 1;
22      while (rem_quantity > 0) {
23        int32_t quantity_to_aggregate = std::min(rem_quantity, curr_power_of_two);
24        items_01.push_back(Item{.weight = quantity_to_aggregate * item.weight,
25                                .value  = quantity_to_aggregate * item.value});
26        rem_quantity -= quantity_to_aggregate;
27        curr_power_of_two *= 2;
28      } // while
29    } // for item
30    return items_01;
31  } // translate_to_01()
32
33  // This function is identical to the bottom-up 0-1 solution provided earlier
34  int32_t knapsack_01(int32_t c, const std::vector<Item>& items) {
35    const size_t n = items.size();
36    std::vector<std::vector<int32_t>> memo(n + 1, std::vector<int32_t>(c + 1, 0));
37    for (size_t i = 0; i < n; ++i) {
38      for (size_t j = 0; j < c + 1; ++j) {
39        if (j < items[i].weight) {
40          memo[i + 1][j] = memo[i][j];
41        } // if
42        else {
43          memo[i + 1][j] = std::max(memo[i][j], memo[i][j - items[i].weight] + items[i].value);
44        } // else
45      } // for j
46    } // for i
47    return memo[n][c];
48  } // knapsack_01()
```

It should be noted that there do exist other algorithms that can be used to solve the bounded knapsack problem in $\Theta(nC)$ time, where n is the original number of items given without double-counting identical copies. These algorithms won't be discussed in this chapter, however, as they are a bit too complicated for the scope of this class.

24.5 Unbounded Knapsack

The final variant of the 0-1 knapsack problem we will discuss in this chapter is the **unbounded knapsack problem**, which provides an unlimited number of copies for each item. Although this variant may seem complicated at first, it is actually simpler than the other variants we have discussed so far! Previously, we had a constraint on the number of times each item can be used, which required us to consider the costs of including and excluding each item. However, since the unbounded knapsack problem removes this constraint, we no longer have to keep track of which items have already been used. Therefore, we can reduce our subproblems to only one dimension: the capacity of our knapsack.

If we define $F(c)$ as the maximum profit achievable with a knapsack of capacity c, we can recursively compute $F(c)$ by iterating over all items smaller than c and finding the ones we should add to maximize profit. To illustrate how this works, consider the previous example, but this time with an unlimited quantity of each item:

Item	1	2	3	4	5
Weight	1	2	5	6	7
Value	1	6	18	22	28
Quantity	∞	∞	∞	∞	∞

Knapsack Capacity: 45

Since we no longer have to worry about which items have already been included, there are only five subproblems that our solution depends on (given $C = 45$), of which we take the largest:

- The best solution for a knapsack of capacity 44 + value of item 1 ($F(c-w_1)+v_1$, or $F(44)+1$ in this example)
- The best solution for a knapsack of capacity 43 + value of item 2 ($F(c-w_2)+v_2$, or $F(43)+6$ in this example)
- The best solution for a knapsack of capacity 40 + value of item 3 ($F(c-w_3)+v_3$, or $F(40)+18$ in this example)
- The best solution for a knapsack of capacity 39 + value of item 4 ($F(c-w_4)+v_4$, or $F(39)+22$ in this example)
- The best solution for a knapsack of capacity 38 + value of item 5 ($F(c-w_5)+v_5$, or $F(38)+28$ in this example)

In general, the solution to $F(c)$ for any c can be expressed using the following recurrence relation:

$$F(c) = \begin{cases} 0, & \text{if } c = 0 \\ \max(\{F(c-w_i)+v_i \text{ for all } 1 \le i \le n \text{ where } w_i \le c\}), & \text{if } i > 0,\ c > 0 \end{cases}$$

A top-down solution for the unbounded knapsack problem is shown below:

```
1   struct Item {
2     int32_t weight;
3     int32_t value;
4   };
5
6   int32_t unbounded_knapsack(int32_t c, const std::vector<Item>& items) {
7     std::vector<int32_t> memo(c + 1, -1);
8     return knapsack_helper(c, items, memo);
9   } // unbounded_knapsack()
10
11  int32_t knapsack_helper(int32_t c, const std::vector<Item>& items, std::vector<int32_t>& memo) {
12    if (c == 0) {
13      return 0;
14    } // if
15    if (memo[c] != -1) {
16      return memo[c];
17    } // if
18    int32_t best = 0;
19    for (size_t i = 0; i < items.size(); ++i) {
20      if (items[i].weight <= c) {
21        best = std::max(best, knapsack_helper(c - items[i].weight, items, memo) + items[i].value);
22      } // if
23    } // for i
24    return memo[c] = best;
25  } // knapsack_helper()
```

A bottom-up solution is shown below:

```
1   int32_t unbounded_knapsack(int32_t c, const std::vector<Item>& items) {
2     std::vector<int32_t> memo(c + 1);
3     for (size_t j = 0; j <= c; ++j) {
4       for (size_t i = 0; i <= items.size(); ++i) {
5         if (items[i].weight <= j) {
6           memo[j] = std::max(memo[j], memo[j - items[i].weight] + items[i].value);
7         } // if
8       } // for i
9     } // for j
10    return memo[c];
11  } // unbounded_knapsack()
```

There are a total of C subproblems, each taking $\Theta(n)$ time to solve (since we have to loop over all items to determine which we should add to our knapsack). Using dynamic programming, we only need to solve each subproblem once, which gives us a $\Theta(nC)$ time complexity overall. In addition, since we declare a memo of size $C+1$ in our solution, the auxiliary space usage is $\Theta(C)$.