An Algorithmic Approach to Knapsack Problems

Shengyuan Wang

Macalester College

May 28, 2023

Abstract

The knapsack problem is a traditional combinatorial optimization problem that aims to maximize the payload without exceeding the capacity of the bag. When one of the problem variables which are "the capacity" or "the types/number of materials" increases, the complexity of the problem size increases significantly. Thus, it has been an area of interest for researchers and has been approached using various algorithms. In this paper, we present how to use algorithms to solve different variations of the knapsack problem, including the $0-1$ knapsack problem and the fractional knapsack problem. We also present the multidimensional knapsack problem. Furthermore, we show how these algorithms can be applied in real-life practice.

1 Overview

1.1 What Are Knapsack Problems?

Suppose we are planning a hiking trip and are therefore interested in filling a knapsack with items considered necessary for the trip. There are N different item types that are deemed desirable, such as a bottle of water, an apple, an orange, or a sandwich. Each item type has a given set of two attributes: a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type of item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is figuring out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value [6].

The problem discussed so far represents a variety of knapsack-type problems in which a set of entities is given, each with an associated value and size. The goal is to select one or more disjoint subsets such that the sum of the sizes in each subset does not exceed a given bound and the sum of the selected values is maximized.

The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884-1956) and refers to the common problem of packing the most valuable or useful items without exceeding the luggage's weight or volume capacity.

1.2 Mathematical Formulation

The knapsack problem can be mathematically formulated [4] by numbering the objects from 1 to n and introducing a vector of binary variables x_j $(j = 1, \ldots, n)$ having the following meaning:

$$x_j = \begin{cases} 1 & \text{if object } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

Then, if p_j is a measure of the comfort given by object j, w_j its size and c the size of the knapsack, our problem will be to select, from among all binary vectors x satisfying the constraint

$$\sum_{j=1}^{n} w_j x_j \leq c,$$

the one which maximizes the objective function

$$\sum_{j=1}^{n} p_j x_j.$$

1.3 Terminology

The objects considered in the previous section will generally be called *items* and their number be indicated by n. The value and size associated with the jth item will be called *profit* and *weight*, respectively, and denoted by p_j and w_j $(j \in \{1, \ldots, n\})$.

It is always assumed, as is usual in the literature, that profits, weights and capacities are positive values. The results obtained, however, can easily be extended to the case of real values and, in the majority of cases, to that of nonpositive values.

The prototype problem of the previous section,

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j = 0 \text{ or } 1, \\ & j \in \{1, \ldots, n\} \end{aligned}$$

is known as the $0-1$ *Knapsack Problem* and will be analyzed in section 2.

In section 3 we consider the generalization arising when we no longer demand a solution of the knapsack problem to be integral but allow for fractional values, then we get the *Fractional Knapsack Problem*. Another important generalization of the $0-1$ knapsack problem, in section 4, is the *Multidimensional Knapsack Problem*, arising when there is more than one constraint of given capacities available.

2 $0-1$ Knapsack Problem

2.1 Problem Statement

The $0-1$ *Knapsack Problem* (KP) is [5]: given a set of n *items* and a *knapsack*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_j &= \textit{weight} \text{ of item } j, \\ c &= \textit{capacity} \text{ of the knapsack}, \end{aligned}$$

select a subset of the items so as to

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j = 0 \text{ or } 1, \quad j \in \{1, \ldots, n\} \end{aligned}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

2.2 Algorithms

There are several algorithms that can be used to solve the $0-1$ knapsack problem, including dynamic programming, approximate, and greedy algorithms. Dynamic programming algorithms involve breaking the problem down into smaller subproblems and solving these subproblems recursively. Approximate algorithms provide a solution that is close to the optimal solution, but may not be the optimal solution itself. Greedy algorithms make a locally optimal choice at each step in the hope of finding a globally optimal solution.

2.2.1 Dynamic Programming Algorithm

Algorithm 1 Dynamic Programming Knapsack Algorithm [3]

Require: Nonnegative integers $n, c_1, \cdots, c_n, w_1, \cdots, w_n, W$

```
C ← Σ_{j=1}^{n} c_j
x(0,0) ← 0
x(0,k) ← ∞ for k = 1, ··· , C
for j ← 1 to n do
  for k ← 0 to C do
    s(j,k) ← 0
    x(j,k) ← x(j−1,k)
  end for
  for k ← c_j to C do
    if x(j−1,k−c_j) + w_j ≤ min(W, x(j,k)) then
      x(j,k) ← x(j−1,k−c_j) + w_j
      s(j,k) ← 1
    end if
  end for
end for
```

2.2.2 Approximation Algorithm

Algorithm 2 Knapsack Approximation Scheme [3]

Require: Nonnegative integers$n, c_1, \cdots, c_n, w_1, \cdots, w_n$ *and* W. A number $\epsilon > 0$.

{The output will be A subset $S \subset \{1, \cdots, n\}$ such that $\sum_{j \in S} w_j \leq W$ and $c(S) \geq \frac{1}{1+\sigma} c(S')$ for all $S' \subset \{1, \cdots, n\}$ with $\sum_{j \in S'} w_j \leq W$.}

Let $c_1, \cdots, c_n, w_1, \cdots, w_n$ *and* W be nonnegative integers with $w_j \leq W$ for $j = 1, \cdots, n, \sum_{i=1}^{n} w_i > W,$ and $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$

```
S_1 ← min{ j ∈ {1, ··· , n : Σ_{i=1}^{j} w_i > W} }
if c(S_1) = 0 then
  S ← S_1
  stop
end if
t ← max{ 1, ε c(S_1) / n }
for j ← 1, ··· , n do
  c_j ← ⌊ c_j / t ⌋
end for
Apply the Dynamic Programming Knapsack Algorithm to the instance
(n, c_1, ··· , c_n, w_1, ··· , w_n, W) set C ← 2c(S_1) / t
if c(S_1) > c(S_2) then
  S ← S_1
else
  S ← S_2
end if
```

Theorem 1. *Greedy Algorithm Redux is a 2-approximation for the knapsack problem.*

Proof. In this case, we used a greedy algorithm, which means that if our solution is sub-optimal, there must be some leftover space $B - S$ at the end. Since our algorithm was able to take a fraction of an item. Then, we can add $\frac{B-S}{s_{k+1}} P_{k+1}$ to our knapsack value to either match or exceed the optimal value (OPT). Therefore, either $\sum_{k=1}^{i-1} pi \geq \frac{1}{2}\text{OPT}$ or $p_{k+1} \geq \frac{B-S}{s_{k+1}} p_{k+1} \geq \frac{1}{2}\text{OPT}$. □

3 Fractional Knapsack Problem

3.1 Problem Statement

Relaxing the $0-1$ constraint on the variables, the *Fractional Knapsack Problem* (FKP) is [5]: given a set of n *items* and a *knapsack*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_j &= \textit{weight} \text{ of item } j, \\ c &= \textit{capacity} \text{ of the knapsack,} \end{aligned}$$

select a subset of the items so as to

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & 0 \leq x_j \leq 1, \\ & j \in \{1, \ldots, n\}. \end{aligned}$$

3.2 Investigation

In contrast to the $0-1$ knapsack problem, the fractional knapsack problem can be solved using a simple and efficient greedy algorithm. Informally, the algorithm works as follows: consider the items in decreasing value-to-weight ratio and add whole items to the knapsack one at a time until we reach an item whose addition would cause the knapsack's capacity W to be exceeded. Then, add the largest fraction of that item that fits into the knapsack and stop. This is expressed in pseudocode below:

Algorithm 3 Fractional Knapsack Algorithm [3]

Sort the items so that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$
$s \leftarrow 0$
$k \leftarrow 1$
while $k \leq n$ and $s_{w_k} \leq W$ **do**
  $x_k \leftarrow 1$
  $s \leftarrow s + w_k$
  $k \leftarrow k + 1$
end while
if $k \leq n$ **then**
  $x_k \leftarrow \frac{W-s}{w_k}$
  for $t \leftarrow k + 1, n$ **do**
    $x_t \leftarrow 0$
  end for
end if

This algorithm provides an approximate solution to the fractional knapsack problem that is guaranteed to be at least $\frac{1}{2}$ of the optimal solution. This will be proven later in the paper.

3.3 Proof of Optimality

Without loss of generality, we can assume that the items are indexed in the order in which they are sorted by the algorithm. Thus,

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

If $\sum_{t=1}^{n} w_t \leq W$, then clearly the optimal knapsack will be $(1, 1, \cdots, 1)$, and this is what the algorithm returns in the case. And in the rest of the proof, we will assume that $\sum_{t=1}^{n} w_i > W$.

Lemma 2. *Let $S = (x_1, \cdots, x_n)$ be a knapsack, and i, j be items such that $i < j, x_i < 1$,and $x_j > 0$. Then there is a knapsack $S' = (x'_1, \cdots, x'_n)$ such that $V(S') \geq V(S)$, S' is identical to S in all items except i and j, and either $x'_i = 1$ or $x'_j = 0$.*

Proof. Intuitively, we can construct S' by transferring some of the weight occupied by item j to item i, which is at least as valuable per unit of weight. There are two exhaustive cases to consider, depending on whether we can take all of the weight occupied by item j and use it to steal more of item i (in which case $x'_j = 0$), or we can take only part of the weight occupied by item j and use it to steal all of the rest of item i (in which case $x'_i = 1$). □

Case 1. *$w_j x_j \leq w_i(1 - x_i)$ (i.e., we can transfer all of the weight occupied by item j). Then let $S' = (x'_1, \cdots, x'_n)$ where*

$$\begin{aligned} x'_t &= x_t, \text{ for all } t \neq i, j \\ x'_i &= x_i + \frac{w_j}{w_i} x_j \\ x'_j &= 0 \end{aligned}$$

First we verify that this is indeed a knapsack, i.e., it satisfied the two feasibility constraints (a) and (b). For (a), it is obvious that $0 \leq x'_t \leq 1$ for all $t \neq i$, and that $x'_i \geq 0$. Furthermore, by the definition of x'_i and the hypothesis of Case 1, we have

$$x'_i x_i + \frac{w_j}{w_i} x_j \leq x_i + w_i \frac{1 - x_i}{w_i} = 1$$

so $x'_i \leq 1$, as wanted. And for (b), by the definition of x'_i and x'_j, we have

$$\sum_{t=1}^{n} w_t x'_t = \left(\sum_{t \neq i,j} w_t x'_t \right) + w_i x'_i + w_j x'_j = \left(\sum_{t \neq i,j} w_t x_t \right) + w_i (x_i + \frac{w_j}{w_i} x_j) = \sum_{t=1}^{n} w_t x_t \leq W$$

as wanted. It remains to prove that $V(S') \geq V(S)$. Indeed, using the definition of x'_i and x'_j we have

$$V(S') - V(S) = v_i x'_i + v_j x'_j - v_i x_i - v_j x_j = v_i (x_i + \frac{w_j}{w_i} x_j - x_i) - v_j xj = \frac{v_i}{w_i} w_j x_j - v_j x_j \geq 0$$

where the last inequality follows from the fact that $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ by multiplying both sides by $w_j x_j$. Therefore $V(S') \geq V(S)$.

Case 2. *$w_j x_j > w_i(1 - x_i)$(i.e. we can only transfer part of the weight occupied by item j before exceeding the remaining weight of item i). Then let $S' = (x'_1, \cdots, x'_n)$ where*

$$\begin{aligned} x'_t &= x_t, \textit{for allt} \neq i, j \\ x'_i &= 1, \\ x'_j &= x_j + (\frac{w_i}{w_j})(1 - x_i) \end{aligned} \tag{3}$$

Using algebra as in Case i, we can verify that S' is a knapsack, and that $V(S') \geq V(S)$. The next lemma states something obvious: An optional knapsack cannot have any unused capacity.

Lemma 4. *If $S = (x_1, \cdots, x_n)$ is an optional knapsack then $\sum_{t=1}^{n} w_t x_t = W$*

Proof. We prove the contrapositive: Suppose $\sum_{t=1}^{n} w_t x_t \neq W$. So S has some surplus capacity $c = W - \sum_{t=1}^{n} w_t x_t > 0$. Since $\sum_{t=1}^{n} w_t > W$, it cannot be that $x_t = 1$for all items t, so there is some item k such that $x_k < 1$. By increasing x_k to $x_k + \min(\frac{c}{w_k}, 1 - x_k)$(i.e. using up the surplus capacity c to steal more of item k, upto stealing all of it) we obtain a knapsack of greater value than S. So S is not optimal. □

Let us define an anomaly in a knapsack $(x_1, \cdots, x_n)$ to be a pair of items i and j such that $i < j$, $x_i < 1$ and $x_j > 0$. The next lemma states that the knapsack returned by the greedy algorithm has no anomalies and no unsed capacity.

Lemma 5. *The knapsack $G = (x_1^g, \cdots, x_n^g)$ returned by the greedy algorithm satisfied the following: (a) There is some item k such that $x_t^g = 1$ for all items $t < k$, and $x_t^g = 0$ for all items $t > k$. (b) $\sum_{t=1}^{n} w_t x_t^g = W$*

Proof. From the algorithm's description (Algorithm 3) it is clear that, for some k, the algorithm assigns $x_k = 1$ to the first $k - 1$ items in line 4, assigns some part of item k's weight to x_k in line 9. and assigns 0 to all remaining items in line 11. Thus, part (a) of the lemma holds. A straightforward induction shows that, at the end of the $i-$th iteration of the loop in lines 4-7, $s = \sum_{j=1}^{i} w_j$. Since, by assumption, $\sum_{i=1}^{n} w_i > W$, the algorithm exits the while loop with $i \leq n$. So, by the assignments between lines 8 and 12, $\sum_{i=1}^{n} w_i x_i = W$. □

There is only one knapsack with no anomalies and no unused capacity (i.e., the knapsack is filled to its maximum capacity). If there were two different knapsacks with these properties, the 1s in one would be a proper prefix of the 1s in the other, which implies that the first has unused capacity. However, by Lemma 3, the knapsack returned by the greedy algorithm, G, has no unused capacity. Therefore, there is only one knapsack with no anomalies and no unused capacity, which is G.

Furthermore, by Lemmas 1 and 2, there exists an optimal knapsack with no anomalies and no unused capacity. Since there is only one such knapsack, namely G, it follows that G is also optimal. This means that the greedy algorithm returns the optimal solution to the $0 - 1$ knapsack problem [2].

3.4 Greedy Algorithm

We can consider the following greedy algorithm for the knapsack problem which will refer to as greedy knapsack. We sort all the items by the ratio of their profits to their sizes so that $\frac{p_1}{s_1} \geq \frac{p_2}{s_2} \geq \frac{p_3}{s_3} \geq \cdots \geq \frac{p_n}{s_n}$. Then, we greedily take items in this order as long as adding an te to our collection does not exceed the capacity of the knapsack. It turns out that the algorithm can be arbitrarily bad. Suppose we only have two items in N. Let $s_1 = 1$, $p_1 = 2$, $s_2 = B$, and $p_2 = B$. The algorithm will only take item 1, but taking only item 2 would be a better solution. As it turns out, we can easily modify this algorithm to

a 2-approximation by simply taking the best of solution or the most profitable item. We will call the new algorithm modified greedy.

Theorem 6. *Modified Greedy has an approximation ratio of $\frac{1}{2}$ for the knapsack problem [1].*

Proof. Let k be the index of the first item that is not accepted by greedy knapsack. To begin, we claim that $p_1 + p_2 + \cdots + p_k \geq \text{OPT}$. In fact, $p_1 + p_2 + \cdots + \alpha p_k \geq \text{OPT}$, where $\alpha = \frac{B-(s_1+s_2+\cdots+s_{k-1})}{s_k}$ is the fraction of item k that can fit in the knapsack after packing the first $k-1$ items. Then the proof of Theorem 6 follows immediately from the claim. Particularly, either $p_1 + p_2 + \cdots + p_{k-1}$ or p_k must be at least OPT/2.

Consider an LP relaxation of the knapsack problem as follows:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} s_i x_i \leq B,$$

where $x_i \in [0, 1]$ denotes the fraction of item i packed in the knapsack.

Then, let OPT' be the optimal value of the objective function in this LP problem. Since any solution to Knapsack is a feasible solution to the LP, and both problems share the same objective function, so $\text{OPT}' \geq \text{OPT}$. If we set $x_1 = x_2 = \cdots = x_{k-1} = 1$, $x_k = \alpha$, and $x_i = 0$ for all $i > k$, then we obtain a feasible solution to the LP that cannot be improved by changing any one tight constraint, as we sorted the items. Therefore, $p_1 + p_2 + \cdots + \alpha p_k = \text{OPT}' \geq \text{OPT}$. □

4 Multidimensional Knapsack Problem

4.1 Problem Statement

The *Multidimensional Knapsack Problem* (MDKP) is [5]: given a set of n *items* and a set of m *knapsacks*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_j &= \textit{weight} \text{ of item } j, \\ c_i &= \textit{capacity} \text{ of knapsack i}, \end{aligned}$$

the goal is to find a subset of objects that maximizes the total profit while satisfying

some resource constraints. Formally,

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i \in \{1, \ldots, m\} \\ & x_j = 0 \text{ or } 1, \quad j \in \{1, \ldots, n\} \end{aligned}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

When $m = 1$, MDKP reduces to the $0-1$ knapsack problem considered in Section 2.

5 Work Example

To illustrate how knapsack problems can be applied in real-life situations, we will use two different knapsack problem algorithms in two different settings. This will show how these algorithms can be used to make better and more rational decisions in practical scenarios.

Setting 1

During Black Friday, almost every product has a discount. As a clever customer, we need to maximize the discounts within a given budget. To do this, we collect data from Amazon and design a problem about how to select the products we want from a total of 30 products to maximize the discount while not exceeding the $500 budget. In this setting, we introduce two algorithms: the $0-1$ Knapsack algorithm and the Approximation algorithm. Here table (1) is the first 5 items of the 30 products, the full product list is shown in appendix as Table (6).

Item	**Discounted_Price**	**Original_Price**	**Discount**
Formal shoe	$23	$36.99	$14
Tablet	$30	$49.99	$20
Headphone	$60	$149.99	$90
Printer	$20	$59.99	$40
Speaker	$45	$59.99	$15

Table 1: First 5 Products Information

5.1 $0-1$ Knapsack Algorithm

We utilize a dynamic programming method here to implement the $0-1$ Knapsack algorithm. To use dynamic programming to find the optimal result for picking the products, we will need to

- Create a matrix representing all subsets of the items, with rows representing items and columns representing the remaining budget.
- Loop through the matrix and calculate the discount Loop through the matrix and calculate the discount that can be obtained by each combination of items at each stage of the bag's capacity.
- Examine the completed matrix to determine which products to buy in order to maximize the total discount.

In Python, we can create this matrix by:

```
dp = [[0 for _ in range(col)] for _ in range(row)]
```

We have padded the rows and columns by 1 so that the indices match the item and weight numbers. Now that we have created our matrix, we will fill it by looping over the rows and columns.

For each element, we will calculate its worth value. We do this by considering the following: if the item at the index matching the current row fits with the weight capacity represented by the current column, take the maximum of either:

- The total worth of the items already in the bag, or
- The total worth of all the items in the bag except the item at the previous row index, plus the new item's worth.

In Python, we can implement the method by:

```
for item in range(row):
  for budget in range(col):
    if price[item] > budget:
      dp[item][budget] = dp[item-1][budget]
      continue
    previous_best_discount = dp[item-1][budget]
    new_best_discount = discount[item] + dp[item-1][budget-price[item]]
    dp[item][budget] = max(previous_best_discount, new_best_discount)
```

In other words, as our algorithm considers each item, we are asked to decide whether adding this item to the bag would produce a higher total worth than the last item added to the bag, given the bag's current total weight. If the current item is a better choice, we include it in the bag; if not, we leave it out.

Result

The result derived from the Dynamic Programming Method is shown in the following Table (2). The total price is $499, and we save $464 through this purchasing match.

Item	Discounted_Price	Discount
Formal shoe	$23	$13
Headphone	$59	$90
Printer	$19	$40
Sneaker	$55	$34
Wireless speaker	$99	$100
Bluetooth headphone	$78	$51
Skateboard	$36	$22
PS controller	$38	$25
Shirt	$27	$12
Sunglass	$34	$55
Swimsuit	$31	$22

Table 2: Result from Dynamtic Programming Method

5.2 Approximate Algorithm

We can also employ an approximate algorithm here to approximate the optimal solution for this problem setting.

We need to

- Sort the items by calculating for each item how much discount we can get using one dollar.
- Loop through the sorted items and see if adding the item will exceed our budget.
- If it does, we will compare which will create more discount: buying just the item or all the other items picked before.

In Python, we can implement the method by:

```
def approx_alg(w, v, W):
    ordered = []
    num = 1
    for vi, wi in zip(v, w):
        ordered.append([vi/wi, vi, wi, num])
        num += 1
    ordered.sort(reverse=True)
    S1 = []
```

```
    weight = 0
    val = 0
    for i in range(len(ordered)):
        vi, wi, itemNum = ordered[i][1], ordered[i][2], ordered[i][3]
        if weight + wi <= W:
            S1.append([itemNum, vi, wi])
            weight += wi
            val += vi
        else:
            S2 = [[itemNum, vi, wi]]
            val2 = vi
            if val > val2:
                return S1, val
            else:
                return S2, val2
```

Result

The result derived from the Approximate Method is shown in the following Table (3). The total price is $476, and we save $459 through this purchasing match. From both Table (2) and Table (3), we find that the optimal result derived from Approximate Method is different from Dynamic Programming Method. However, the result does not vary too much with one being $464, the other being $459.

Item	Discounted_Price	Discount
Printer	$19	$40
Sunglass	$34	$55
Headphone	$59	$90
Wireless speaker	$99	$100
Swimsuit	$31	$22
Tablet	$29	$20
PS controller	$38	$25
Bluetooth headphone	$78	$51
Sneaker	$55	$34
Skateboard	$36	$22

Table 3: Result from Approximate Method

5.3 Setting 2

In the magic potion shop, we can purchase different kinds of potions, including the Elixir of Power, Wisdom, Healing, Magic, Invisibility, Strength, and Agility. Our aim is to

bring them back to our hometown to sell in order to make a profit. However, considering the limited capacity of our backpack, which is 200, and the varying profit for each different potion, we need to find the best matching of potions. The weight and profit we can get from each kind of potion is shown in Table (4)

Potion Type	**Power**	**Wisdom**	**Healing**	**Magic**	**Invisibility**	**Strength**	**Agility**
Weight	50	60	30	80	50	20	70
Profit	$340	$100	$180	$120	$220	$300	$60

Table 4: Potion Table For Setting 2

Here, we introduce the fractional knapsack algorithm to solve this problem. To do this, we need to:

- Sort the potions according to the profit earned for one unit of capacity.
- Handle whole items first, and then partial items.
- Finally, we will get the optimal solution.

In Python, we can implement the method by:

```
items_list.sort(reverse = True)
for item in items_list:
    if capacity - item.weight >= 0:
        capacity -= item.weight
        knapsack_value += item.value
        knapsack.append(item.item)
        knapsack_weights.append(item.weight)
    else:
        fraction = capacity / item.weight
        knapsack_value += item_value * fraction
        knapsack.append(item.item)
        knapsack_weights.append(round(fraction, 2))
        capacity -= int(item.weight * fraction)
        break
```

Result

The result derived from Greedy Algorithm is shown in the following Table (5). The total profit we earn $1123.33 from potions through adding 200 weight to our bag.

Potion Type	Power	Wisdom	Healing	Magic	Invisibility	Strength	Agility
Weight	50	50	30	80	50	20	0
Profit	$340	$83.33	$180	$120	$220	$300	$0

Table 5: Result from Greedy Algorithm

6 Conclusion and Future Work

In this paper, we focus on introducing dynamic programming, approximate, and greedy algorithms in the $0-1$ knapsack problem and fractional knapsack problem. We explore the proof of optimality of these algorithms and how they can be applied in real-life practice. We find that these algorithms help us make better and more rational decisions.

For future work, we plan to analyze the multidimensional knapsack problem and explore more algorithms, such as genetic algorithms, applied in knapsack problems. In addition, we may also conduct a comparative analysis of the performance of different algorithms.

Appendix

Item	Discounted_Price	Original_Price	Discount
Formal shoe	$23	$36.99	$13
Tablet	$30	$49.99	$20
Headphone	$60	$149.99	$90
Printer	$20	$59.99	$40
Speaker	$45	$59.99	$15
Sneaker	$55	$89.95	$35
Running Shoe	$66	$85.00	$19
Watch	$75	$91.75	$17
Wireless speaker	$99	$199.00	$100
Jogger	$41	$54.00	$14
Arsenal kit	$84	$90.00	$6
Gym shoe	$57	$69.95	$13
Laptop bag	$98	$113.97	$16
Bluetooth headphone	$78	$129.99	$52
Skateboard	$37	$59.99	$23
Sweater	$26	$27.82	$2
PS controller	$39	$64.99	$26
GTA V game	$85	$99.99	$15
Bike helmet	$30	$33.96	$4
Hair dryer	$39	$59.99	$21
Bed sheet	$32	$39.78	$8
Shirt	$28	$39.99	$12
Power bank	$40	$46.99	$7
Sunglass	$35	$89.99	$55
Swimsuit	$32	$54.00	$22
Aquarium tank	$28	$32.89	$5
Blazer	$86	$99.99	$14
Kitchen mixer	$37	$39.67	$2
Microwave bowl	$16	$17.99	$2
Protein powder	$24	$28.99	$5

Table 6: Product Information For Setting 1

References

[1] Chandra Chekuri. The knapsack problem, Jan 2009.

[2] Vassos Hadzilacos. Fractional knapsack, 2022.

[3] Bernhard Korte and Jens Vygen. *Combinatorial optimization: Theory and algorithms*. Springer Berlin Heidelberg, 2000.

[4] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and computer implementations*. UMI Books on Demand, 2006.

[5] Hayat Abdullah Shamakhai. The 0-1 multiple knapsack problem, 2017.

[6] Ben Hamida Yazid. Quantum-inspired genetic algorithm for a class of combinatorial optimization. Master's thesis, M'Sila, Algeria, 2016.