Last update: August 16, 2024 Original

Knapsack Problem

Prerequisite knowledge: Introduction to Dynamic Programming

Introduction

Consider the following example:

[USACO07 Dec] Charm Bracelet

There are n distinct items and a knapsack of capacity W. Each item has 2 attributes, weight (w_i) and value (v_i). You have to select a subset of items to put into the knapsack such that the total weight does not exceed the capacity W and the total value is maximized.

In the example above, each object has only two possible states (taken or not taken), corresponding to binary 0 and 1. Thus, this type of problem is called "0-1 knapsack problem".

0-1 Knapsack

Explanation

In the example above, the input to the problem is the following: the weight of i^{th} item w_i, the value of i^{th} item v_i, and the total capacity of the knapsack W.

Let $f_{i,j}$ be the dynamic programming state holding the maximum total value the knapsack can carry with capacity j, when only the first i items are considered.

Assuming that all states of the first $i-1$ items have been processed, what are the options for the i^{th} item?

- When it is not put into the knapsack, the remaining capacity remains unchanged and total value does not change. Therefore, the maximum value in this case is $f_{i-1,j}$
- When it is put into the knapsack, the remaining capacity decreases by w_i and the total value increases by v_i, so the maximum value in this case is $f_{i-1,j-w_i} + v_i$

From this we can derive the dp transition equation:

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

Further, as f_i is only dependent on f_{i-1}, we can remove the first dimension. We obtain the transition rule

$$f_j \leftarrow \max(f_j, f_{j-w_i} + v_i)$$

that should be executed in the **decreasing** order of j (so that f_{j-w_i} implicitly corresponds to $f_{i-1,j-w_i}$ and not $f_{i,j-w_i}$).

It is important to understand this transition rule, because most of the transitions for knapsack problems are derived in a similar way.

Implementation

The algorithm described can be implemented in $O(nW)$ as:

```
for (int i = 1; i <= n; i++)
  for (int j = W; j >= w[i]; j--)
    f[j] = max(f[j], f[j - w[i]] + v[i]);
```

Again, note the order of execution. It should be strictly followed to ensure the following invariant: Right before the pair (i, j) is processed, f_k corresponds to $f_{i,k}$ for $k > j$, but to $f_{i-1,k}$ for $k < j$. This ensures that f_{j-w_i} is taken from the $(i-1)$-th step, rather than from the i-th one.

Complete Knapsack

The complete knapsack model is similar to the 0-1 knapsack, the only difference from the 0-1 knapsack is that an item can be selected an unlimited number of times instead of only once.

We can refer to the idea of 0-1 knapsack to define the state: $f_{i,j}$, the maximum value the knapsack can obtain using the first i items with maximum capacity j.

It should be noted that although the state definition is similar to that of a 0-1 knapsack, its transition rule is different from that of a 0-1 knapsack.

Explanation

The trivial approach is, for the first i items, enumerate how many times each item is to be taken. The time complexity of this is $O(n^2W)$.

This yields the following transition equation:

$$f_{i,j} = \max_{k=0}^{\infty}(f_{i-1,j-k\cdot w_i} + k \cdot v_i)$$

At the same time, it simplifies into a "flat" equation:

$$f_{i,j} = \max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

The reason this works is that $f_{i,j-w_i}$ has already been updated by $f_{i,j-2\cdot w_i}$ and so on.

Similar to the 0-1 knapsack, we can remove the first dimension to optimize the space complexity. This gives us the same transition rule as 0-1 knapsack.

$$f_j \leftarrow \max(f_j, f_{j-w_i} + v_i)$$

Implementation

The algorithm described can be implemented in $O(nW)$ as:

```
for (int i = 1; i <= n; i++)
  for (int j = w[i]; j <= W; j++)
    f[j] = max(f[j], f[j - w[i]] + v[i]);
```

Despite having the same transition rule, the code above is incorrect for 0-1 knapsack.

Observing the code carefully, we see that for the currently processed item i and the current state $f_{i,j}$, when $j \geqslant w_i$, $f_{i,j}$ will be affected by $f_{i,j-w_i}$. This is equivalent to being able to put item i into the backpack multiple times, which is consistent with the complete knapsack problem and not the 0-1 knapsack problem.

Multiple Knapsack

Multiple knapsack is also a variant of 0-1 knapsack. The main difference is that there are k_i of each item instead of just 1.

Explanation

A very simple idea is: "choose each item k_i times" is equivalent to "k_i of the same item is selected one by one". Thus converting it to a 0-1 knapsack model, which can be described by the transition

function:

$$f_{i,j} = \max_{k=0}^{k_i}(f_{i-1,j-k\cdot w_i} + k \cdot v_i)$$

The time complexity of this process is $O(W \sum_{i=1}^{n} k_i)$

Binary Grouping Optimization

We still consider converting the multiple knapsack model into a 0-1 knapsack model for optimization. The time complexity $O(Wn)$ can not be further optimized with the approach above, so we focus on $O(\sum k_i)$ component.

Let $A_{i,j}$ denote the j^{th} item split from the i^{th} item. In the trivial approach discussed above, $A_{i,j}$ represents the same item for all $j \leq k_i$. The main reason for our low efficiency is that we are doing a lot of repetetive work. For example, consider selecting $\{A_{i,1}, A_{i,2}\}$, and selecting $\{A_{i,2}, A_{i,3}\}$. These two situations are completely equivalent. Thus optimizing the splitting method will greatly reduce the time complexity.

The grouping is made more efficient by using binary grouping.

Specifically, $A_{i,j}$ holds 2^j individual items ($j \in [0, \lfloor \log_2(k_i + 1) \rfloor - 1]$).If $k_i + 1$ is not an integer power of 2, another bundle of size $k_i - 2^{\lfloor \log_2(k_i+1) \rfloor - 1}$ is used to make up for it.

Through the above splitting method, it is possible to obtain any sum of $\leq k_i$ items by selecting a few $A_{i,j}$'s. After splitting each item in the described way, it is sufficient to use 0-1 knapsack method to solve the new formulation of the problem.

This optimization gives us a time complexity of $O(W \sum_{i=1}^{n} \log k_i)$.

Implementation

```
index = 0;
for (int i = 1; i <= n; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
```

```
  list[index].v = h * k;
}
```

Monotone Queue Optimization

In this optimization, we aim to convert the knapsack problem into a maximum queue one.

For convenience of description, let $g_{x,y} = f_{i,x\cdot w_i+y},\ g'_{x,y} = f_{i-1,x\cdot w_i+y}$. Then the transition rule can be written as:

$$g_{x,y} = \max_{k=0}^{k_i}(g'_{x-k,y} + v_i \cdot k)$$

Further, let $G_{x,y} = g'_{x,y} - v_i \cdot x$. Then the transition rule can be expressed as:

$$g_{x,y} \leftarrow \max_{k=0}^{k_i}(G_{x-k,y}) + v_i \cdot x$$

This transforms into a classic monotone queue optimization form. $G_{x,y}$ can be calculated in $O(1)$, so for a fixed y, we can calculate $g_{x,y}$ in $O(\lfloor \frac{W}{w_i} \rfloor)$ time. Therefore, the complexity of finding all $g_{x,y}$ is $O(\lfloor \frac{W}{w_i} \rfloor) \times O(w_i) = O(W)$. In this way, the total complexity of the algorithm is reduced to $O(nW)$.

Mixed Knapsack

The mixed knapsack problem involves a combination of the three problems described above. That is, some items can only be taken once, some can be taken infinitely, and some can be taken atmost k times.

The problem may seem daunting, but as long as you understand the core ideas of the previous knapsack problems and combine them together, you can do it. The pseudo code for the solution is as:

```
for (each item) {
  if (0-1 knapsack)
    Apply 0-1 knapsack code;
  else if (complete knapsack)
    Apply complete knapsack code;
  else if (multiple knapsack)
    Apply multiple knapsack code;
}
```

Practise Problems

- Atcoder: Knapsack-1
- Atcoder: Knapsack-2
- LeetCode - 494. Target Sum
- LeetCode - 416. Partition Equal Subset Sum
- CSES: Book Shop II
- DMOJ: Knapsack-3
- DMOJ: Knapsack-4

Contributors:

OverRancid (98.37%) Ahmed-Elshitehi (1.09%) adamant-pwn (0.54%)