

0-1 Knapsack Problem

Introduction

This program is an implementation of the **0-1 Knapsack Problem** by using **Dynamic Programming**. By running [Zero_One_Knapsack.cpp](#) in Visual Studio, the program will read the weight limit and the items' information from the input file [input.txt](#), and then generate an optimal solution with total revenue and items selected, which would be saved in [output.txt](#).

*Note that all the files have to be placed under the same directory.

Concept

1. Read Input File

- (1) Read the first line of [input.txt](#) to get the number of items n .
- (2) Read the second line to get the weight limit [max_weight](#).
- (3) Read n consecutive lines from the file. Each line contains the value and weight of an item, and store them into two vector arrays, [value](#) and [weight](#), respectively.

2. Solve the 0-1 Knapsack Problem

Apply **Dynamic-Programming method**:

(1) Optimal Substructure:

Let O be an optimal subset of all n items with weight limit K .

- i. If O does not contain item n :
→ O is an optimal subset of the first $n - 1$ items.
- ii. If O does contain item n :
→ $O - \{n\}$ is a solution to the problem instance that includes the first $n - 1$ items, and a weight limit $K - w_n$, where w_n is the weight of item n .

(2) Recursive Formulation:

$$c[n, w] = \begin{cases} 0 & , \text{if } n = 0 \\ c[n - 1, w] & , \text{if } n > 0 \text{ and } w - \text{weight}[n] < 0 \\ \max\{c[n - 1, w], c[n - 1, w - \text{weight}[n]] + \text{value}[n]\} & , \text{if } n > 0 \text{ and } w - \text{weight}[n] \geq 0 \end{cases}$$

$c[n, w]$: the maximum revenue on the first n items with a weight limit of w .

$\text{value}[n]$: the value of item n

$w[n]$: the weight of item n

(3) Compute Optimal Value:

Zero_One_Knapsack function: A **bottom-up** algorithm based on the recurrence.

- i. The array *revenue* is constructed to record the current largest revenue for each possible weight limit.
- ii. For each item *i*, starting from $j = \text{max_weight}$, we examine the revenue for each possible weight limit *j*, where *j* is greater than the weight of item *i*.
- iii. Calculate the new revenue if item *i* is included (i.e. “the revenue without item *i*” + “the value of item *i*”).
- iv. Compare the new revenue with the revenue if it is not included at this particular weight limit. If the former is larger, it means that selecting item *i* yields a greater revenue than not selecting it, so we update the revenue for the particular weight limit *j*.
- v. Continue iterating through *i* and *j*.
- vi. Occasionally, after all iterations are done, we can obtain the maximum total revenue from *revenue[max_weight]*.

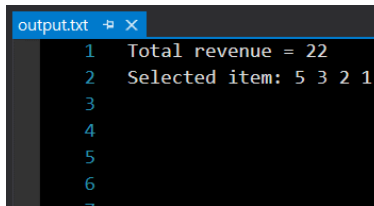
*Note that the iteration of weight limit *j* starts from *max_weight*, and is decremented by 1 after each iteration. We perform this iteration in reverse order, so that instead of using a 2D array to represent the maximum revenue *c* mentioned in the recurrence, we could just use a 1D array to store the results.

(4) Construct an Optimal Solution:

- i. The boolean array *choice* is constructed in order to record whether an item is selected or not at each possible weight limit.
- ii. Initially, all the elements are set as *false*, meaning that no items are selected at the beginning.
- iii. During the process of (3) iv., we will mark *choice[i][j]* as true when the new revenue is greater, meaning that item *i* is selected at this weight limit *j*.
- iv. After all computations are done, we can obtain the chosen items by examining *choice*. (Explained in the next paragraph)

3. Write Output File

- (1) An optimal solution generated from the above process is written in *output.txt* in the following format:



```
output.txt
1 Total revenue = 22
2 Selected item: 5 3 2 1
3
4
5
6
7
```

- (2) Obtain the selected items:
- Since we iterate in reverse order in the *Zero_One_Knapsack* function, we will have to look through *choice* in the same manner.
 - Start from the last item $n-1$ and check if it is selected when weight limit= max_weight (i.e. $choice[n-1][max_weight] == true$).
 - If false:
→Continue checking for the previous item.
 - If true:
→Write this item into *output.txt*. Note that the index is incremented by 1 to get the real index(1~n).
→Subtract its weight from the weight limit.
→Go on to the previous item.
 - Continue until all items are checked, and we will occasionally obtain an optimal solution.

. Divide the n elements of the input array into $b_n = 5c$ groups of 5 elements each and at most one group made up of the remaining $n \bmod 5$ elements. 2. Find the median of each of the $d_n = 5e$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements. 3. Use SELECT recursively to find the median x of the $d_n = 5e$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.) 4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the k th smallest element and there are n_k elements on the high side of the partition. 5. If $i \leq k$, then return x . Otherwise, use SELECT recursively to find the i th smallest element on the low side if $i \leq k$.