C++ Laboratory Examination Practice

axc18@ic.ac.uk

Problem Description

The knapsack problem or rucksack problem is a problem in combinatorial optimization:

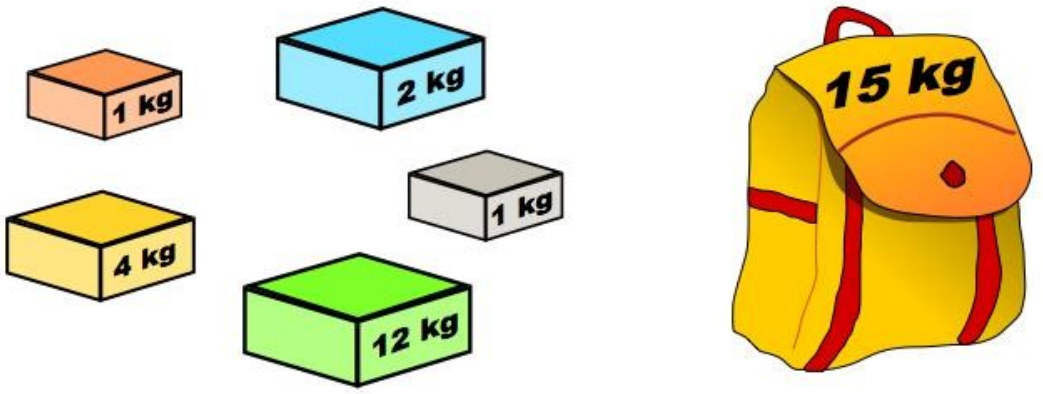

Figure 1: An example of a Knapsack problem

Given a set of items, each with a weight, find the combination of items that results in the least wasted space in the knapsack. For the example in Figure 1, the optimal solution would be: **1 x 12kg, 1 x 2kg, 1 x 1kg = 15kg**

Pre-Supplied Files

To get you started, you are supplied with function prototypes in **knapsack.h**, in which the implementations to should be in the file **knapsack.cpp**

You are also provided with an `items.txt` file which contains items and their corresponding weights:

```
A 4
B 21
C 14
```

In this example, Item **A** has weight **4**, Item **B** has weight **21**, etc.

Specific Tasks

1. Write a function `get_items(filename, items)` which reads the contents of `filename` and produces an output string `items` containing the items listed in the file. If the given file cannot be opened, the function returns false, true otherwise.

   For example, the code:

   ```
   char items[512];
   get_items("items.txt", items);
   ```

   Should return `true` and `items` containing the string `ABCDE`.

2. Write a function `item_weight(filename, item)` which retrieves from `filename` the weight of a given `item`. The function should return 0 if it cannot be found in the file, or if the file does not exist.

   For example, the code:

   ```
   cout << item_weight(“items.txt”, ‘A’);
   cout << item_weight(“items.txt”, ‘Z’);
   ```

   Should return `4` and `0` respectively.

3. Write a function `insert_knapsack(capacity, item, filename)` which attempts to insert an `item` (whose weight is to be retrieved from `filename`) into a knapsack of a given `capacity`. The function should return the new capacity of the knapsack after inserting the item, or -1 if it cannot be inserted (item is too big).

   For example, the code:

   ```
   cout << insert_knapsack(20, 'A', "items.txt");
   cout << insert_knapsack(20, 'B', "items.txt");
   cout << insert_knapsack(20, 'C', "items.txt");
   ```

   Should return `16, -1, 6` respectively.

4. Write a function `fill_knapsack(capacity, items, filename)` which determines the ***minimum*** possible space left unfilled in the knapsack given a list of items, `items`, and returns this value.

   You should assume that each item may only be used once.

   - `capacity` is an integer indicating the capacity of the knapsack
   - `items` is char array of items (as returned by the `get_items` function)
   - `filename` is read-only string that is used to look up the weights of the items

   For example, the code:

```
char items[512];
get_items("items.txt", items);
cout << fill_knapsack(10, items, "items.txt");
cout << fill_knapsack(20, items, "items.txt");
cout << fill_knapsack(30, items, "items.txt");
cout << fill_knapsack(40, items, "items.txt");
```

   Should return `3, 1, 0, 0` respectively.

   (A knapsack of 10 can be filled with D, leaving 3)
   (A knapsack of 20 can be filled with DE, leaving 1)
   (A knapsack of 30 can be filled with ACE, leaving 0)
   (A knapsack of 40 can be filled with BDE, leaving 0)

   For full credit, your solution should be recursive and use pointer arithmetic. However, partial credit will be awarded for a working iterative solution.

Bonus Challenge

Using the `fill_knapsack` function created in Question 4, write a function:

```
int bonus_fill_knapsack(
     int capacity,
     char * items,
     std::string & knapsack,
     const char * filename
);
```

That does the same thing as `fill_knapsack`, but also returns (by reference) a string, `knapsack`, containing the items that were used to achieve the optimal configuration. **This function should be fully recursive.**

For example the code:

```
string knapsack;
space = bonus_fill_knapsack(20, items, knapsack, "items.txt");
cout << "Knapsack of capacity 20 can be filled with ‘"
cout << knapsack;
cout << "’ to achieve leftover space " << space;
```

Should print:

```
> Knapsack of capacity 20 can be filled with ‘DE’ to
achieve leftover space 1
```

Hints

1. Feel free to define any auxiliary functions which would help to make your code more elegant.

2. When attempting Question 4, consider how this may be solve recursively. Since the order of the items does not matter, the problem may be reduced to whether a given item is either (a) taken or (b) skipped.

   Capacity = 10; Items = ‘A, B, C’

   Take A:
   Capacity = 6; Items = ‘B, C’
   Leave A:
   Capacity = 10; Items = ‘B, C’