Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

Comprehensive Assignment (32%)
CSI 2120 Programming Paradigm

Winter 2021

Part 1 due on Januaray 25 at 23:00

Part 2 due on February 22 at 23:00

Part 3 & 4 dues on April 14 at 23:00

Problem description

This comprehensive assignment asks you to solve the Knapsack problem using the programming paradigm seen in class.

This problem can be expressed as follows: given a set of items, each of them having a weight and a value and given a knapsack (i.e. a container for items) having a maximal weight capacity, find the subset of items to be added to the knapsack that maximizes the total value without exceeding the weight capacity. Each item can be inserted just once and cannot be subdivided. Note that we will solve here the problem for which the item weights are expressed in integer units. Mathematically the problem to solve is the following:

Given two sets of n positive integers

$< v_0, v_1, v_2, \dots, v_{n-1} >$ *and* $< w_0, w_1, w_2, \dots, w_{n-1} >$

and a number W > 0, find the subset $K \subset \{0,1,\dots,n\text{-}1\}$ *that maximizes*

$$\sum_{k \in K} v_k$$

subject to

$$\sum_{k \in K} w_k \leq W$$

Example:

Consider the following example with a knapsack of capacity 7:

	Item A	Item B	Item C	Item D
Value	1	6	10	15
Weight	1	2	3	5

The solution to this problem is to select items B and D for a total value of 21 and a total weight of 7.

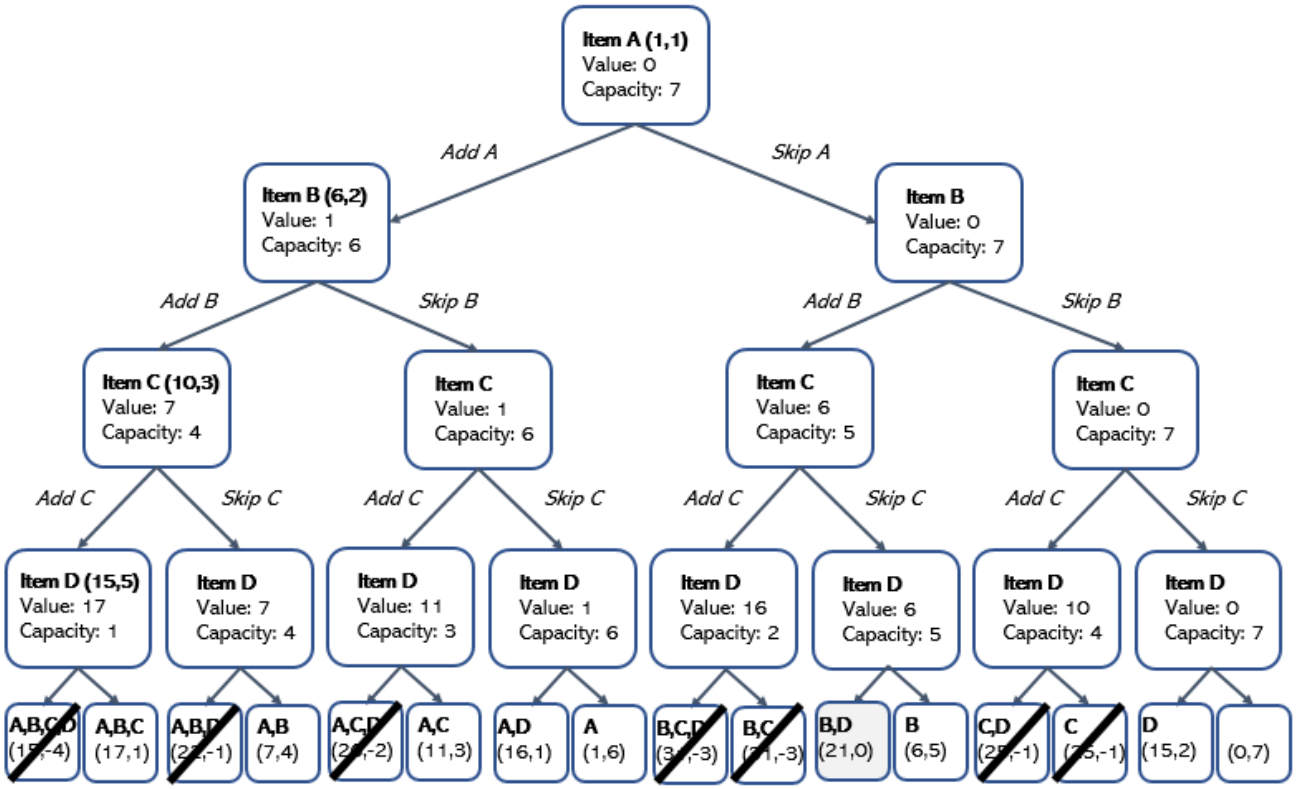
Solving the problem

There are two possible strategies to solve this problem: a brute-force solution and a dynamic programming solution. We will ask you to compare both in this assignment.

1. Brute force

Here the idea is simply to consider all possible solutions and select the one giving the highest value. Indeed, the solution space for this problem can be represented by a binary tree in which each level is associated to a specific item. For each item, we have to take a binary decision: should we include it in the knapsack or not? This binary decision is represented by the two branches below each node of the binary tree.

For example, considering the case given in the table at the top of this page, we can first consider item A. We have two options: if we add it to the knapsack, then we have a current value of 1 and a residual capacity of 6 (because item A has a weight of 1 and our knapsack has a capacity of 7). Now, let's consider the next item, that is item B (this is the second level of our solution tree). The first node of this level corresponds to the case where we already have item A in the knapsack. If we also choose to add item B then the new value is 7 and the residual capacity is 4. If we do not add item B, then we still have current value at 1 and capacity at 6. The second node of this level is for the case where we have not added item A to the knapsack. In that case if we add item B, then we have a current value of 6 and a residual capacity of 5. Finally, if we choose to not add these two items, our capacity is still at 7 and the value of this empty knapsack is obviously at 0. The full solution tree is given on the next page, and the inspection of the tree leaves gives us the best solution.



Few observations about the knapsack solution tree:

- *this is a perfect tree with a height equals to the number of items;*
- *some nodes of the tree can correspond to infeasible solutions when the knapsack capacity becomes negative, in that case there is no need to explore the subtree below;*
- *the number of nodes of this tree is 2^{n+1}-1 which means that the complexity of the brute algorithm for n items is $O(2^n)$. Consequently, this approach becomes rapidly impracticable when the number of items becomes large.*

We therefore need a better algorithm to solve this problem in the general case.

2. Dynamic programming

In this approach we will iteratively solve our knapsack problem by solving each sub-problem until we obtain the full solution. That is if the capacity of the knapsack is W then we will solve the problem for knapsacks of capacity 0, 1, 2, … up to W with numbers of items going from 0 to n. We will proceed by building a table that will list all these subproblems.

Let's first consider the obvious cases: that is when the knapsack has a capacity of 0 and also the ones where we have 0 item to add to the knapsack. We use here the example presented earlier. The columns of this table correspond to knapsacks of capacity 0 to n (here n=7). The rows correspond to the case where we consider 0 to n items (i.e. first row is for 0 item, second row is for item A only, third row is for the sub-problems where we only have items A and B, etc). The cells of this table give us the solution

(maximum value) for each corresponding sub-problem. The solution to the problem we want to solve will be found in the bottom right cell.

As it can be seen in the first column of the table below, the solution for knapsacks of capacity 0 has always a value of 0 (obviously) no matter how many items we have. Same for the first row where the solution is also 0 when there is no item to put in the knapsack.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0							
Items A,**B**	0							
Items A,B,**C**	0							
Items A,B,C,**D**	0							?

Now let's consider the sub-problems where we have only item A to put in our knapsack. This item has a weight of 1 so it fits in all knapsack with a capacity >0. The value of item A is 1, so we can populate the second row of our table with 1s.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0							
Items A,B,C	0							
Items A,B,C,D	0							?

Now we consider the case where we have 2 items: A and B (this is the third row). Item B has a weight of 2 and a value of 6. For a knapsack of capacity 1, item B does not fit so we have to keep item A in it. For a capacity of 2, we can now insert item B in it. This item has a higher value than item A; so the optimal solution would be 6. When the capacity of the knapsack has a capacity greater or equal to 3, the two items can be added to it; the maximum value becomes then 7 (the sum of the values of A and B).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0	1	6	7	7	7	7	7
Items A,B,C	0							
Items A,B,C,D	0							?

These were easy cases, we need a more general strategy in order to fill all the rows of this table. Let's now move to the case where we have 3 items (A, B and C). Item C has a weight of 3 and a value of 10. For all knapsack with a capacity less than 3, the optimal solution is the same as the ones in the previous rows since the new item cannot fit in these knapsacks.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0	1	6	7	7	7	7	7
Items A,B,C	0	1	6					
Items A,B,C,D	0							?

Now let's consider the case of the knapsack of capacity 3. We have the option to include item C or not. If we do not include item C, then the optimal solution becomes the same than the one of the previous row (which is 7). If we include item C, then the value of the knapsack will be 10 (value of C) which is better than the previous value without C. With this single item, the knapsack is full so this is our optimal solution.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0	**1**	6	**7**	7	7	7	7
Items A,B,C	0	1	6	**10**				
Items A,B,C,D	0							?

With a knapsack of capacity 4, we also have the option to not add item C in which case the optimal solution is 7 (solution given by the cell above). But if we insert item C which has a weight of 3, then the knapsack has still room for 1 weight unit; we therefore have to find the optimal solution for the case of the two remaining items (A and B) with a knapsack of capacity 1. The solution to this subproblem is 1 as found in the previous row, second column. So the total value for keeping item C is 10 + 1= 11 which is better than 7 (the solution without C in the knapsack). Then for a knapsack of capacity 5, the optimal solution is either 7 (without C) or 10 plus the optimal solution for a knapsack of capacity 2 with 2 items, 10+6=16. For a capacity of 6, it is 10+7=17. Same for the knapsack of capacity 7.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0	1	6	7	7	7	7	7
Items A,B,C	0	1	6	10	11	16	17	17
Items A,B,C,D	0							?

The rule for filling one cell is therefore as follows: the solution is the maximum between the value just above (the case where we do no select the current item) and the value of the current item (which is D for the last row, with a value of 15 and a weight of 5) plus the solution to the subproblem of the previous row with a capacity of the current knapsack minus the weight of the current item. Using this rule, it becomes easy to fill the rest of the table.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
No item	0	0	0	0	0	0	0	0
Item A	0	1	1	1	1	1	1	1
Items A,B	0	1	6	7	7	7	7	7
Items A,B,C	0	1	6	10	11	16	17	17
Items A,B,C,D	0	1	6	10	11	16	17	**21**

Formally, the cell at row i and column j in our table T is given by the following formula:

```
if (wi > j) // new item at row i (having weight wi)
            // does not fit into knapsack of capacity j
   T(i,j)= T(i-1,j)
else
   T(i,j)= max ( T(i-1,j) , vi + T(i-1,j-wi) )
```

In order to fill this table, we therefore have to iterate over the columns, row by row. We have a total of n+1 rows and W+1 columns, the complexity of this algorithm is therefore O(nW). Note that in the case of this assignment, we will create a table with columns for each weight units (so W+1 columns) but in practice, you just need to find the greatest common divisor across all the items' weights and use it as increment for the table columns.

Programming

You have to write programs under different paradigms that solves the knapsack problem. Each program takes as input the name of the file containing the problem description. This file has this simple format: number of items, name value and weight of each item, capacity of the knapsack.

```
4
A  1  1
B  6  2
C  10 3
D  15  5
7
```

Your program must save the solution in a file formatted as follows (same name as input file but with extension .sol):

```
21
B  D
```

Part 2: concurrent programming (GO) **[8 marks]**

For the concurrent programming paradigm, you will only implement the brute force solution of the knapsack problem. A Go version of this solution is given to you in its regular recursive form (with a single thread). Your mission is to use concurrency in order to accelerate the resolution of this problem.

Since the brute force approach consists in exploring the binary solution tree, a first naïve approach would be to subdivide the problem into two parts, the left sub-tree and the right sub-tree, and to create two threads in order to concurrently solve these subproblems. Once these two threads completed, only the best solution is kept.

Following this idea, instead of solving the problem this way:

```
KnapSack(W, weights , values)
```

You could proceed as follows:

```
last := len(weights)-1
// best solution with the last item
go KnapSack(W - weights [last], weights [:last], values [:last])
// best solution without the last item
go KnapSack(W, weights [:last], values [:last])

// code here to synchronize and then determine which one is the best
solution
```

You can even go one step further and solve the two sub-sub-problems of each two sub-problems above using 4 concurrent threads. You can also proceed similarly for the next level (8 sub-problems) and so on. This sounds like a good approach except that at some point the sub-problems will become so simple (i.e. having so few items) that there won't be any value in splitting them into concurrent threads. Moreover, we know that the number of nodes in the solution tree is $O(2^n)$*; consequently, the number of threads to create in order to have all sub-problems concurrently solved would be prohibitive. Even if the go threads are very lightweight they still require some resources such that the total amount of memory that this exhaustive approach would require could crash your machine.*

Consequently if our objective is to make our program to run faster, there should be an optimal choice in terms of number of threads to create. That is to say that that a sub-problem should be solved concurrently only if the number of items in this sub-problem is greater than a given threshold.

You are then asked to solve the knapsack problem following the brute force strategy and using concurrent programming. You have to experimentally determine the optimal number of threads that should be created in order to obtain a solution as quickly as possible.

Your program must read the input data from a file et display the name of the chosen items, their total value and the execution runtime. Note that the go program given to you does not meet these requirements and must therefore be modified.

In addition to the source code of your go program, you must submit a document describing how you implemented your concurrent solution and what experiments you performed in order to find the optimal number of threads to create. A graph showing runtime versus number of threads for different configurations should be included.

Each program will be marked as follows:

Program produces the correct value	*[1.5 points]*
Program produces the correct set of items	*[1.5 points]*
Adherence to programming paradigm	*[3 points]*
Quality of programming (structures, organisation, etc)	*[1 point]*
Quality of documentation (comments and documents)	*[1 point]*

All your files must include a header showing student name and number. These files must be submitted in a zip file.