CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 8 – Recursion

(Week 10, starting 21 March 2016)

1. Recursion on Stacks

Restrictions: You are NOT allowed to use any additional data structure aside from what is given. Remember that you are supposed to view **stack** as an **ADT**, and can only use its `isEmpty()`, `push()`, `pop()` and `peek()` methods.

Design and implement **recursive methods** to solve the following problems.

a) Return the sum of all the integers in a given stack. Ensure that at the end of the operation, you do not corrupt the data in the stack.

```
int sum(Stack<Integer> stackInt)
```

b) A stack contains integers in a sorted ascending sequence, i.e. the largest integer is at the top. Push `val` into the correct position in the given stack, such that the stack remains sorted after insertion.

```
void insert(Stack<Integer> stackInt, int val)
```

c) Given a stack containing integer elements, sort its elements in ascending order, i.e. largest integer at the top. You may use the `insert()` method in (b). You are NOT allowed to use the Java API Collections.sort(), as that defeats the objective of this exercise.

```
sort(Stack<Integer> stackInt)
```

Answers

(a)

Given any stack of integers: sum(stack's elements) = pop() stack + sum(remaining elements)

e.g. sum([10, 6, 2, 3, 4]top) = **4** + sum([10, 6, 2, 3]top)	
= 4 + (**3** + sum([10, 6, 2]top))	
= 4 + (3 + (**2** + sum([10, 6]top)))	
= 4 + (3 + (2 + (**6** + sum([10]top))))	
= 4 + (3 + (2 + (6 + (**10** + sum([])))))	push back popped elm
= 4 + (3 + (2 + (6 + (**10** + 0))))	[**10**]top
= 4 + (3 + (2 + (**6** + 10)))	[10, **6**]top
= 4 + (3 + (**2** + 16))	[10, 6, **2**]top
= 4 + (**3** + 18)	[10, 6, 2, **3**]top
= **4** + 21 = 25	[10, 6, 2, 3, **4**]top

Ask yourself:

- Where are the **local variables** and parameters in each method call stored?
- In what order is code that appears **before** the recursive call executed?
- In what order is code that appears **after** the recursive call executed?

```
public static int sum(Stack<Integer> stackInt) {
     if (stackInt.isEmpty())
          return 0;
     Integer popped = stackInt.pop();
     int sum = sum(stackInt);
     stackInt.push(popped);
     return sum + popped;
}
```

(b)

As in part (a), we need a second stack to store the contents of the first. Since we are not allowed to use additional data structure, we can make use of Java's call stack to store the popped elements.

```
public static void insert(Stack<Integer> stackInt, int val) {
     if (stackInt.isEmpty() || stackInt.peek() <= val) {
          stackInt.push(val);
          return;
     }
     Integer popped = stackInt.pop();
     insert(stackInt, val);
     stackInt.push(popped);
}
```

(c)

sort() shows a recursive algorithm that is quite similar to insertion sort[1]. An empty stack is a sorted stack. Recursively pop() every element from the stack, and then insert each popped element into the correct place using (b).

```
public static void sort(Stack<Integer> stackInt) {
     if (!stackInt.isEmpty()) {
          Integer popped = stackInt.pop();
          sort(stackInt);
          insert(stackInt, popped);
     } // what is the base case here?
}
```

When designing the recursive algorithm, it may be useful to:

- Define the problem recursively, and check that the algorithm fits the definition
- Trace through the sequence of recursive calls made by drawing out the recursive list/tree
- Watch how the parameters, local variables and return values change in each recursive call

As you get better, you may only need to define the problem recursively. However, the second and third technique allows you to convince yourself that the algorithm works, and identify mistakes in your algorithm.

[1] Insertion sort works on a list, but this algorithm works on a stack.

2. *Power Set*

Dylan, the boss of InfoLogistics, a huge logistics business, faces many logistical problems daily with regards to his warehouses in Singapore. In particular, he faces the problem of picking the **best set** of **a small group of different items** to store so that his warehouse will be fully utilized.

Being a good friend of Dylan, you decide to help him enumerate all possible sets of items he can select from a given list of items, so that he does not miss out any choice that could be made. The items in the given list are already sorted ascending lexicographically.

```
// This is the method that Dylan will call.
// Precond : items non-null, lexicographically sorted ASC.
// Postcond: items not modified, all storage options printed
public void enumerateOptions(List<String> items) {
     enumerateOptions(items, 0, new ArrayList<String>());
}

private void enumerateOptions(List<String> items, int currIdx,
     ArrayList<String> selectedItems) {
     // Complete this recursive method
}
```

Design and implement a simple recursive solution that prints all the different possible sets, with an empty line between sets of items.

For example, given a list of items:

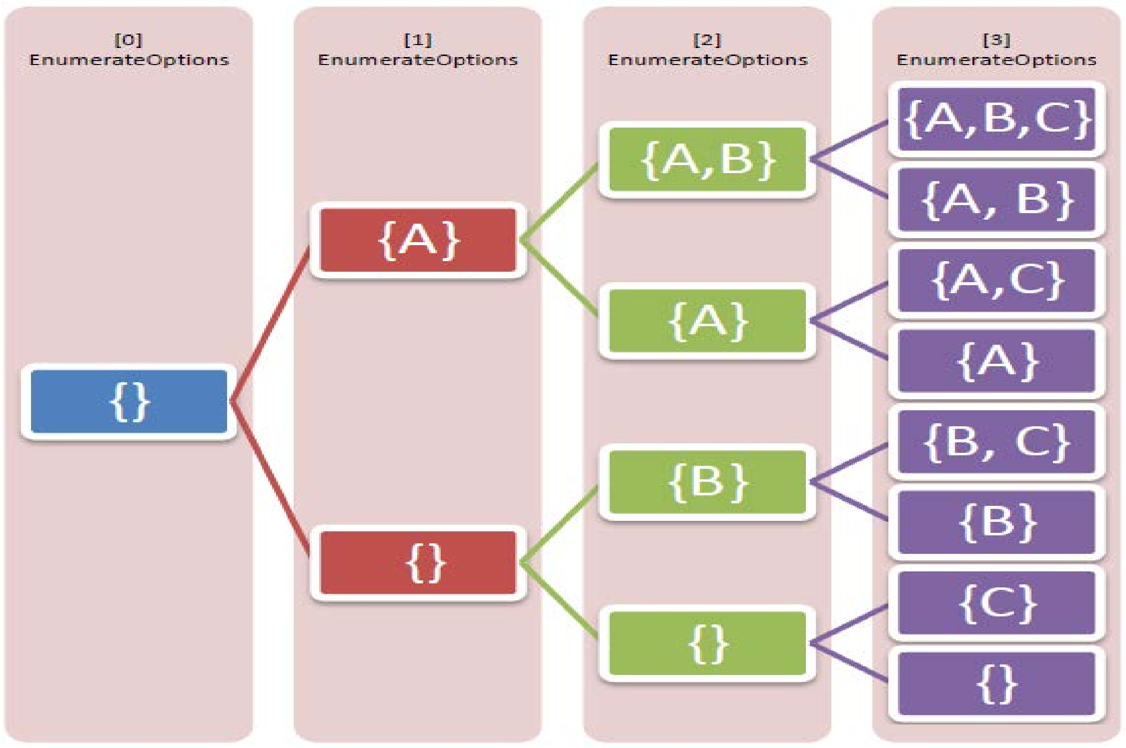
0. Apple Macbook Pro
1. Lenovo Thinkpad
2. Mircosoft Surface Book

Your solution should print out the 8 possible sets of items in the following order:

```
Apple Macbook Pro
Lenovo Thinkpad
Mircosoft Surface Book

Apple Macbook Pro
Lenovo Thinkpad

Apple Macbook Pro
Mircosoft Surface Book

Apple Macbook Pro

Lenovo Thinkpad
Mircosoft Surface Book

Lenovo Thinkpad

Mircosoft Surface Book

[Store Nothing]
```

Note that "[Store Nothing]" itself is a valid choice which should be generated for Dylan. You do not want him to forget that he should rest, and can choose not to do any work.

Answer



One elegant solution is to recursively generate different storage options. A storage option will be printed when we have determined whether each item is to be selected or not.

- Examine current item if there is such an item
- Enumerate all storage options that choose the current item first
- Enumerate all storage options that do not choose the current item last
- If there is no such item, then we are at the end of the list. Therefore, print out this option

```
private void enumerateOptions(List<String> items, int currIdx,
   ArrayList<String> selectedItems) {
   if (currIdx >= items.size()) { // Base case: No more items
      for (String item : selectedItems)
         System.out.println(item);
      if (selectedItems.isEmpty())
         System.out.println(MSG_STORE_NOTHING);
      System.out.println();
      return;
   }
   ArrayList<String> selectedCopy =
      new ArrayList<String>(selectedItems);
   selectedCopy.add(items.get(currIdx));
   enumerateOptions(items, currIdx+1, selectedCopy); // Pick curr
   enumerateOptions(items, currIdx+1, selectedItems); // Ignore curr
}
```

3. *Knapsack*

Now that you have helped Dylan enumerate all possible storage options, his friend Camilla points out that your program is not very useful as a warehouse has limited capacity. No warehouse will be able to store everything in this world.

His other friend, Benjamin, also raised the fact that Dylan should be picking items based on how much he can earn or profit from, rather than on what he fancies, so that he can make the most value out of his time and effort in his work.

Therefore you decide to improve your previous work from Question 2, by choosing only the best storage option! The **best option** has the **highest total value**, and the space taken up by all items must fall **within the capacity** Dylan provides.

Some classes in this patch have already been created, as shown in the UML diagrams below. You now expect Dylan to provide more information. For each item, we need to know the **space required** and the **value the item would bring** him by storing it. We also need to know the **capacity** of the warehouse.

```
Item
-name : String
-space : int
-value : int
+Item(pName:String,
  pSpace:int, pValue:int)
+toString() : String
+getName() : String
+getSpace() : int
+getValue() : int
```

```
ItemList
-items : ArrayList<Item>
-totalSpace : int
-totalValue : int
+ItemList(anotherList:ItemList)
+toString() : String
+addItem(pItem:Item) : void
+getTotalSpace() : int
+getTotalValue() : int
+isEmpty() : boolean
```

```java
// This is the method that Dylan will call.
// Preconditions  : items non-null, in lexi ASC. order of names
// Postconditions : items not modified, ONLY best option printed
public void enumerateOptions(List<Item> items, int capacity) {
     ItemList selection = enumerateOptions(items, capacity, 0,
          new ItemList(null)); // Creates empty ItemList - See code!
     if (selection.isEmpty()) System.out.println(MSG_STORE_NOTHING);
     else System.out.println(selection.toString());
}
private ItemList enumerateOptions(List<Item> items, int capacity,
     int currIdx, ItemList selection) {
     // Complete the recursive method
}
```

After designing and implementing your algorithm, ask yourself:

- How are ties broken?
- How is this algorithm similar to what you have done in Question 2?
- How is this algorithm different from what you have done in Question 2?

Answer

```
private ItemList enumerateOptions(List<Item> items, int capacity,
   int currIdx, ItemList selection) {

   if (currIdx >= items.size()) // Base case: No more items
      return selection;

   if (capacity < selection.getTotalSpace() +
      items.get(currIdx).getSpace()) // Curr item over capacity
     return enumerateOptions(items, capacity, currIdx+1, selection);

   ItemList copy = new ItemList(selection);
   copy.addItem(items.get(currIdx)); // Immutable, need not clone

   // Find "best" selection both ways
   ItemList bestWith = // Pick curr item
      enumerateOptions(items, capacity, currIdx+1, copy);
   ItemList bestWithout = // Ignore curr item
      enumerateOptions(items, capacity, currIdx+1, selection);

   // Compare selection values
   if (bestWith.getTotalValue() > bestWithout.getTotalValue())
      return bestWith;
   if (bestWith.getTotalValue() < bestWithout.getTotalValue())
      return bestWithout;
   if (bestWith.getTotalSpace() <= bestWithout.getTotalSpace())
      return bestWith;
   return bestWithout;
}
```

In this solution, if there is a tie in total value of the selected items, the set that takes the least space is chosen. If there is a tie in the total space of the selected items, we have arbitrarily chosen the set in which the first differing element has a smaller name, lexicographically. You may use a different criterion to break ties.

=O