CS2040 Tutorial 4 Suggested Solution

Week 6, starting 12 Sep 2022

Q1 Simple Recursion – Sum of Natural Numbers

Write a **recursive** method `int sum(int n)`, which returns the sum of the natural numbers from 1 to **n**, where n ≥ 1. Do NOT use the formula for arithmetic series in your solution

For example, Sum(4) returns **10**, but prints out:

```
Sum(4) = 4 + Sum(3)
Sum(3) = 3 + Sum(2)
Sum(2) = 2 + Sum(1)
Sum(1) = 1
```

Answer

Notice:

- Base case – We **only stop** when **n** == 1, returning **1**
- While winding – Print based on **n**
- Recursive call – Same problem with smaller value of **n**
- While unwinding, returns **n** added to the result of the smaller problem where input is **n**-1

```
int sum(int n) {
   if (n == 1) {
      System.out.println("Sum(1) = 1");
      return 1;
   }
   System.out.println("Sum(" + n + ") = " + n +
      " + Sum(" + (n - 1) + ")");
   return n + sum(n - 1);
}
```

You should be aware that Java maintains a call stack to store, for each recursive call:

- the values of local variables and parameters for that call
- which statement to continue executing in that call, once the next call returns

among other things

Q2 Paint Bucket

You have a 2-D array of Integers, named `colorMatrix`**.** Each cell denotes a number, 0, 1, or 2 which represents a colour (here white, light gray and dark gray respectively). You have to implement the Paint Bucket tool. If you fill one cell in the 2-D array with a colour **c**, the surrounding cells are coloured till it reaches the border of the original colour. The paint gets filled by spreading to the neighbouring cells (each cell has 4 neighbours: left, top, right and bottom), the neighbours' neighbours, and so on…

Implement a **recursive** paintBucketFill() method that takes in these 4 parameters:

- the 2D array
- the row and column index of the cell where the filling starts
- 0, 1, or 2, representing the intended fill colour

You may write a private paintBucketFill() overloaded method that calls the recursive method, to avoid repeated computation, and to allow the recursive problem to take additional parameters without letting the user be concerned with the value of those parameters

1	1	1	1	2	2	2	1
1	1	1	0	0	0	0	2
2	1	0	0	0	0	2	2
1	0	0	0	0	0	2	2
2	0	0	2	2	2	1	1
2	0	0	2	2	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	2

1	1	1	1	2	2	2	1
1	1	1	2	2	2	2	2
2	1	2	2	2	2	2	2
1	2	2	2	2	2	2	2
2	2	2	2	2	2	1	1
2	2	2	2	2	0	0	0
1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	2

Use the skeleton provided to help you test your program:

```
public static void printMatrix(int[][] matrix){
   for (int[] row : matrix) {
      for (int col : row) {
         System.out.print(col + " ");
      } // not particular about extra trailing space here
      System.out.println();
   }
}

public static void main(String[] args) {
   int[][] colorMatrix = {
      {1,1,1,1,2,2,2,1},
      {1,1,1,0,0,0,0,2},
      {2,1,0,0,0,0,2,2},
      {1,0,0,0,0,0,2,2},
      {2,0,0,2,2,2,1,1},
      {2,0,0,2,2,0,0,0},
      {1,1,1,0,0,0,0,0},
      {0,0,0,0,0,0,0,2}
   };
   System.out.println("Before fill...");
   printMatrix(colorMatrix);
   System.out.println();
   paintBucketFill(colorMatrix, 2, 3, 2); /* TODO : Implement that */
   System.out.println("After fill...");
   printMatrix(colorMatrix);
}
```

Answer

There are 3 colours we are concerned with in this problem, the initial colour of the first cell to be filled, the colour to be changed to, as well as the current cell colour

For every cell we are at, expand the fill to all 4 neighbouring cells. Do nothing when we are out of bounds, or have crossed the colour border (initial colour differs from current cell colour). Therefore, to find the border correctly, the initial colour of the first cell to be filled must also be passed to every recursive call of the function

To prevent infinite recursion, we need to ensure the problem shrinks with each recursive call. By filling the current cell before filling its neighbours recursively, the problem shrinks as the current cell has now crossed to the other side of the colour border and will never be filled again

```
public static void paintBucketFill(int[][] matrix,
   int row, int col, int newColor){ // calls recursive private method
   if (newColor != matrix[row][col]) // recurse only if change in colour
      paintBucketFill(matrix, row, col, newColor, matrix[row][col]);
}
private static void paintBucketFill(int[][] matrix,
   int row, int col, int newColor, int initColor){

   if (row < 0 || row >= matrix.length
      || col < 0 || col >= matrix[0].length
      || initColor != matrix[row][col])
      return;

   matrix[row][col] = newColor; // shrink problem
   paintBucketFill(matrix, row-1, col, newColor, initColor);
   paintBucketFill(matrix, row+1, col, newColor, initColor);
   paintBucketFill(matrix, row, col-1, newColor, initColor);
   paintBucketFill(matrix, row, col+1, newColor, initColor);
}
```

Q3 Knapsack Problem

A storage facility provides box storage services to hostel students who are moving out temporarily. Being a poor student, you can only afford to store one large box of items. The given box width is **W** cm and can take at most **K** kg weight. You have **N** items whose lengths and height are exactly the same as the lengths and height of the box

To fully utilize the box space in the box and the weight allowed, you want to:

- Prioritize taking up as much space in the box as possible
- If there are multiple ways to occupy the most space, clear as much total weight as possible

You don't really care what items you should place in the box, you just want to get rid of 'as much clutter' as possible (at least according to the thought process spelled out above)

For example, if you have items with (width, weight) pairs (4,8), (3,2), (2,3), (1,3), (4,3) as well as box limits **W** = 10cm and **K** = 10 kg, the optimal answer will be Result(Width=9, Weight=8) by selecting these 3 pairs: {(3, 2), (2, 3), (4, 3)}

You realize that this problem can be solved brute-force with "choose / don't choose"-styled recursion. Each item can either be chosen, or not chosen, in order to arrive at the best result

(a) Complete the compareTo() method so that the better of two results (disregarding the width and weight limit of the box) will compare greater:

```
class Result implements Comparable<Result> {
   int totalWidth;
   int totalWeight;
   Result(int wd, int wt) { totalWidth = wd; totalWeight = wt; }
   public String toString() {
      return "Width=" + totalWidth + ", Weight=" + totalWeight;
   }
   public int compareTo(Result other) { ... }
}
class Item {
   int width, weight;
   Item(int wd, int wt) { width = wd; weight = wt; }
}
```

(b) Implement the algorithm to find the best result recursively:

```
public Result findMaxSpaceWeight(Item[] items, int W, int K) {
   ...
} // recurse using an overloaded private method if it helps you
```

Answer

(a)

```
public int compareTo(Result other) {
   if (totalWidth < other.totalWidth) return -1;
   if (totalWidth > other.totalWidth) return 1;
   if (totalWeight < other. totalWeight) return -1;
   if (totalWeight > other. totalWeight) return 1;
   return 0;
}
```

(b)

The best result will come from the better of 2 results:

- The best result given that the current item is chosen
- The best result given that the current item is NOT chosen

When we find that there are **no solutions** to the current branch of the problem, we should return the 'worst' result, so that this result is almost never chosen. We have a **valid solution** when we have wound past the last item, and the space and weight limit have not yet been exceeded. One of these solutions form the best result

While unwinding, we just **compare** which of the 2 results is better and return it

```
public Result findMaxSpaceWeight(Item[] items, int W, int K) {
   return findMaxSpaceWeight(items, W, K, 0, 0, 0);
}
private Result findMaxSpaceWeight(Item[] items, int W, int K,
      int currIdx, int selWidth, int selWeight) {

   if (selWidth > W || selWeight > K) // Base case: invalid solution
      return new Result(0, 0);

   if (currIdx == items.length) // Base case: done choosing items
      return new Result(selWidth, selWeight);

   Result toTake = findMaxSpaceWeight( // Best result for 'choose'
      items, W, K, currIdx + 1,
      selWidth + items[currIdx].width,
      selWeight + items[currIdx].weight);

   Result notToTake = findMaxSpaceWeight( // Best result for 'don't'
      items, W, K, currIdx + 1,
      selWidth, selWeight);

   if (toTake.compareTo(notToTake) > 0)
      return toTake;
   return notToTake;
}
```

Question 4 (Online Discussion) – Can Walk?

You are given an **N** x **N** `char[][] grid`, where '-' is a footpath and '#' are bushes. If you can only walk on footpaths and move left/right/up/down, find if there is a way to get from one given cell to another

e.g. `canWalk(grid, 1,1, 3,4)` returns true but `canWalk(grid, 1,1, 0,3)` returns false

```
--#--
--###
-#---
---#-
#----
```

Use recursion to solve this problem, worry less about efficiency. What is the time complexity of your algorithm? What do you think the time complexity of the best algorithm to solve this problem is?