

CS1020E: DATA STRUCTURES AND ALGORITHMS I

Tutorial 7 – Recursion

(Week 9, starting 10 October 2016)

1. Simple Recursion

The greatest common divisor (GCD) of two positive integers is the largest positive integer that divides the numbers without a remainder. For example, the GCD of 18 and 42 is 6.

Write a **recursive** function `gcd(int a, int b)`, which prints how Euclid's algorithm is used to derive the answer. For example, `gcd(5, 8)` returns 1, but prints out:

```
(5, 8) 5 = 0 * 8 + 5
(8, 5) 8 = 1 * 5 + 3
(5, 3) 5 = 1 * 3 + 2
(3, 2) 3 = 1 * 2 + 1
(2, 1) 2 = 2 * 1 + 0
```

Answer

Notice:

- Base case – We **only stop** when `b == 0`, returning `a`
- Winding step – Print based on `a` and `b`
- Recursive call – Same problem with different values of `a` and `b`
- Unwinding step merely returns the last value of `a` all the way up

```
int gcd(int a, int b) {
    if (b == 0) return a;
    cout << "(" << a << ", " << b << ") " << a << " = "
         << (a/b) << " * " << b << " + " << (a%b) << endl;
    return gcd(b, a%b);
}
```

Do you see how you can visualize the execution of the recursive algorithm by:

- Drawing the recursive tree (list in this case)?
- Viewing the problem as a method "pasted" within another?
- Identifying the relationship between the bigger problem and the smaller one(s)?

2. Application of Recursion

You have a 2D int array, in which each element contains a colour - 0 for white, 1 for light gray, 2 for dark gray. You have to implement the paint bucket **fill** functionality. If you don't know what this is, open paint and play around with the paint bucket tool 🎨.


If you fill one element in the 2D array with a colour **c**, the surrounding cells are coloured till it reaches the border of the original colour. The colour spreads from the target cell to the 4 neighbouring cells (top left right bottom), the neighbours of the neighbours, and so on.

Implement the `fill(int** arr, ...)` function below. The **main algorithm must be recursive**. The user will only call the second function. Your second function can make use of the first function if you need to add more parameters.

```
void fill(int** arr, int numRows, int numCols,
         int currRow, int currCol, int newColour /*,one more param? */) {
    // can use this method to recurse if necessary
}
void fill(int** arr, int numRows, int numCols,
         int startRow, int startCol, int newColour) {
    // client will use this method
}
```

As an example, the PaintBucket tool is here being applied to row index 2, column index 3:

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

If you are banned from using recursion, what data structure(s) can help you accomplish something similar?

Answer

The first function does the recursive work, simplifying the entire algorithm. The second function prevents the client from wrongly providing initial values of things the client should not be concerned with. Also, checks that only need to be made before / after the recursion can be moved out to the second function.

```
void fill(int** arr, int nRows, int nCols,
         int currRow, int currCol, int newColo, int firstColo) {

    if (currRow < 0 || currRow >= nRows // row out of bounds
        || currCol < 0 || currCol >= nCols // col out of bounds
        || firstColo != arr[currRow][currCol]) // past colour border
        return;

    arr[currRow][currCol] = newColo;

    fill(arr, nRows, nCols, currRow - 1, currCol, newColo, firstColo);
    fill(arr, nRows, nCols, currRow + 1, currCol, newColo, firstColo);
    fill(arr, nRows, nCols, currRow, currCol - 1, newColo, firstColo);
    fill(arr, nRows, nCols, currRow, currCol + 1, newColo, firstColo);
}

void fill(int** arr, int numRows, int numCols,
         int startRow, int startCol, int newColour) {

    int firstColour = arr[startRow][startCol];
    if (firstColour == newColour)
        return; // nothing to fill

    fill(arr, numRows, numCols,
         startRow, startCol, newColour, firstColour);
}
```

Recursion uses the call stack, so we can convert this into an iterative function by storing "current (row, col) indexes" in a `stack<pair<int, int> >`. If properly implemented, the effect will be identical, and execution time will often be faster.

We could also use a `queue<pair<int, int> >` to solve this problem, but the order in which the array elements change colour is different. Are you able to guess the order in which the array elements are filled in each solution, based on your understanding of the behaviour of queues and stacks?

Design algo before coding
Find repeatable patterns
Draw out the recursive tree



If time is still permitting, Lab TA will discuss some of the Midterm Test questions.