Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

Algorithms: recursion

Mohammad Nekooei

School of Engineering and Computer Science

Victoria University of Wellington

Problem Solving / Algorithm Design

A Key principle of problem solving:

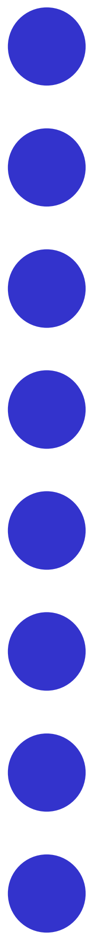
- Break problems up into smaller chunks to solve independently

EG: Iteration:

- To do something to lots of items:
  - work out how to do it to a “typical” item
  - put it in a loop

Algorithm design using iteration

```
public void drawBubbles(double x, double y, int n){
    for (int i = 0; i<n; i++ ) {
        this.drawBubble(x, y, 15);
        y = y - 20;
    }
}

public void drawBubble(double x, double y, double size){
    UI.setColor(Color.blue);
    UI.fillOval(x, y, size, size);
}
```



Algorithm Design with Recursion

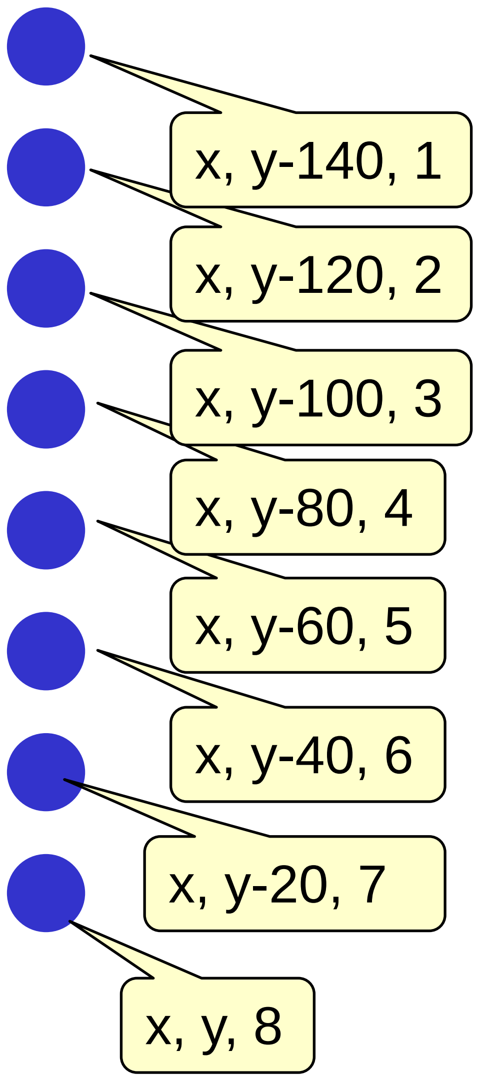
Break up a problem into “the first” and “the rest”

- where “the rest” is a smaller version of the same problem.
  - ➔ can use the same method:

```
public void drawBubbles(double x, double y, int n){

    // draw one bubble
    this.drawBubble(x, y, 15);

    // if there are any remaining bubbles
    if ( n > 1 )  {
        // draw them

        this.drawBubbles(x, y-20, n-1);

    }
}
```

Must have condition to prevent infinite recursion:
Need a “Base case” with no recursive call

Recursive call



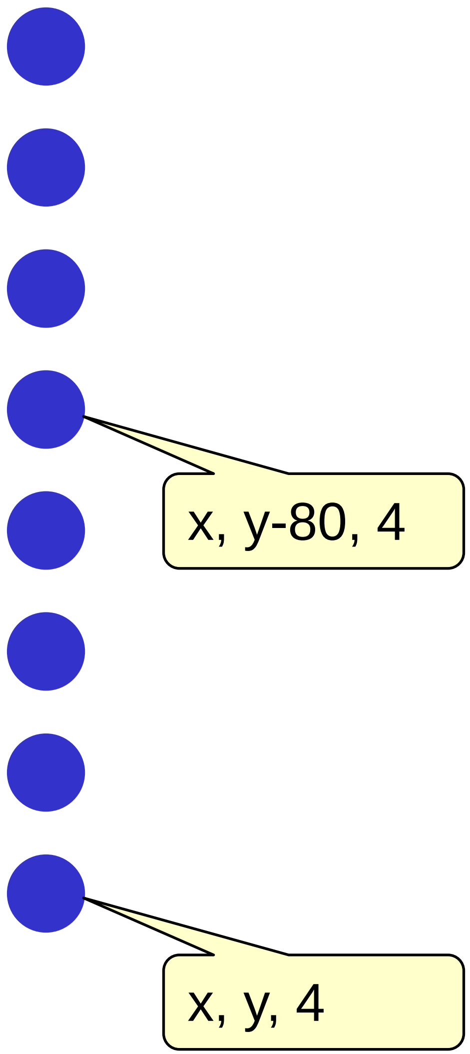
Algorithm Design with Recursion

Break up a problem into “first half” and “second half”

- where each half is a smaller version of the same problem.
  - ➔ can use the same method:

```
public void drawBubbles(double x, double y, int n){
    if ( n == 1 )   {
        this.drawBubble(x, y, 15);
    }
    else if ( n > 1 )   {
        this.drawBubbles(x, y, n/2);
        this.drawBubbles(x, y-n/2*20, (n – n/2));
    }
}
```

x, y, 8



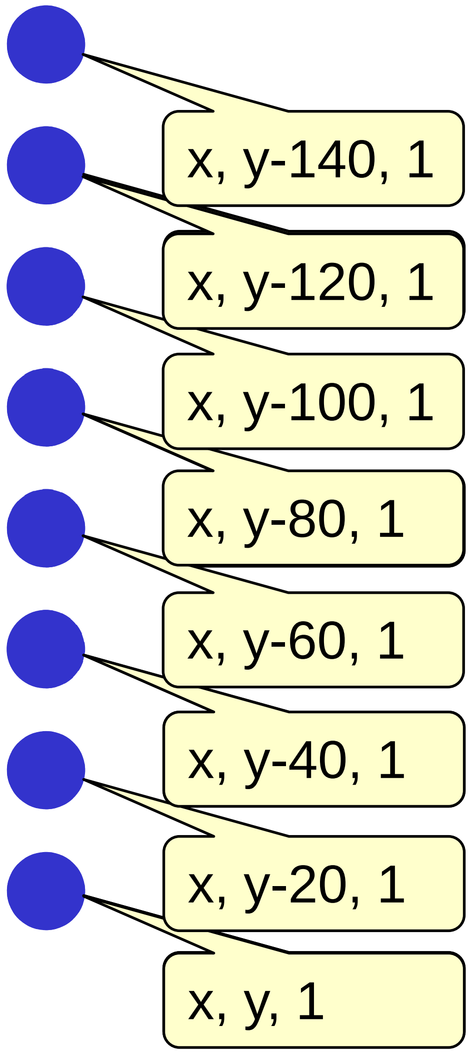
Algorithm Design with Recursion

Break up a problem into “first half” and “second half”

- where each half is a smaller version of the same problem.
  - ➔ can use the same method:

```
public void drawBubbles(double x, double y, int n){
    if ( n == 1 )   {
        this.drawBubble(x, y, 15);
    }
    else if ( n > 1 )   {
        this.drawBubbles(x, y, n/2);
        this.drawBubbles(x, y-n/2*20, (n – n/2));
    }
}
```

x, y, 8



Recursion *vs* Iteration

- Iteration:
  - break problem into sequence of the typical case
  - identify the typical case (body)
  - identify the increment to step to the next case
  - identify the keep-going or stopping condition
  - identify the initialisation
- Recursion: (simple)
  - break problem into first and rest
  - identify the first case
  - identify the recursive call for the rest
    - work out the arguments for the rest
  - identify when you should do the recursive call.
    - make sure there is a stopping case (base case)
  - may need a wrapper method to initialise

```
public int fact(int n){

    int result = 1;
    for (int  i = 1; i<=n; i++ ) {
        result *= i;
    }
    return result;
}
```

```
public int fact(int n){

    if (n == 1) return 1;

    return n * fact(n-1);
}
```

"first" might be split in multiple parts

- Example: Print an "onion" : ((((((( )))))))

```
public void onion (int layers){
    UI.print( " (" );
    if (layers > 1)  {  this.onion(layers-1); }
    UI.print( " )" );
}
```

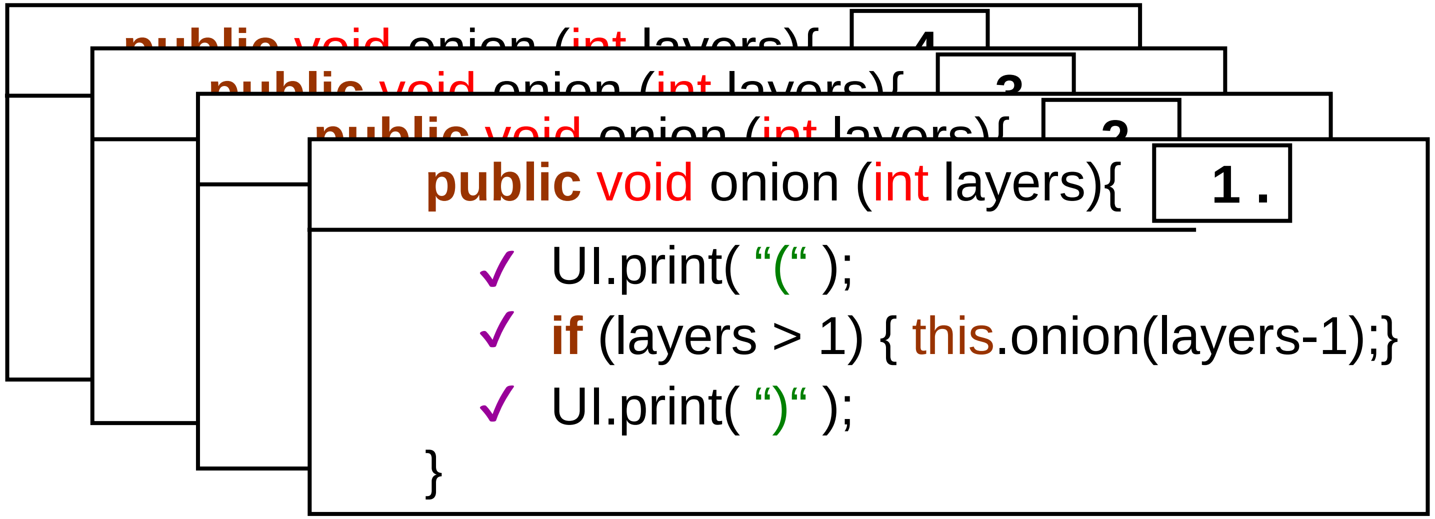
open

do the inside

close

Recursion at work

onion(4) ⇒ (((( ))))



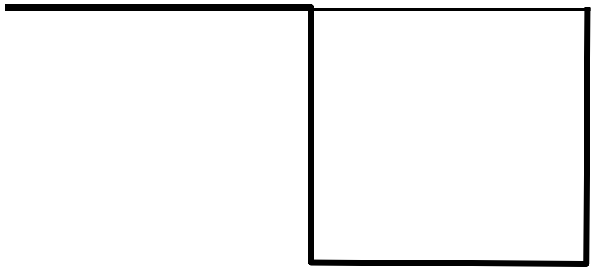

```
public void onion (int layers){
    UI.print( “(“ );
    if (layers > 1)  { this.onion(layers-1); }
    UI.print( “)“ );
}
```

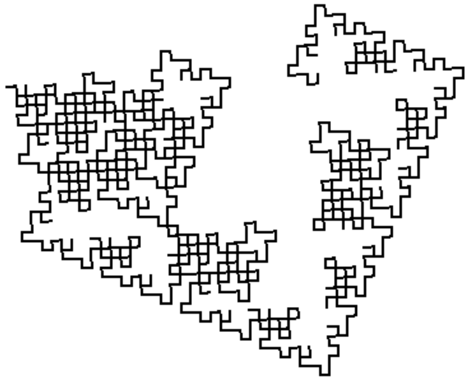
Recursion and Fractals

- Fractals are geometric patterns with repeated structure at multiple levels:

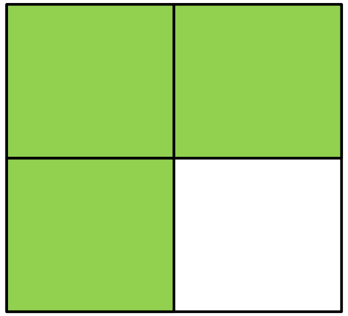
Simple examples:

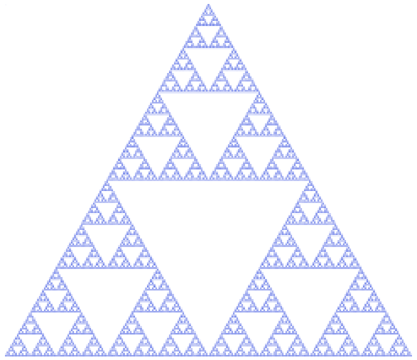
- Fractal Line





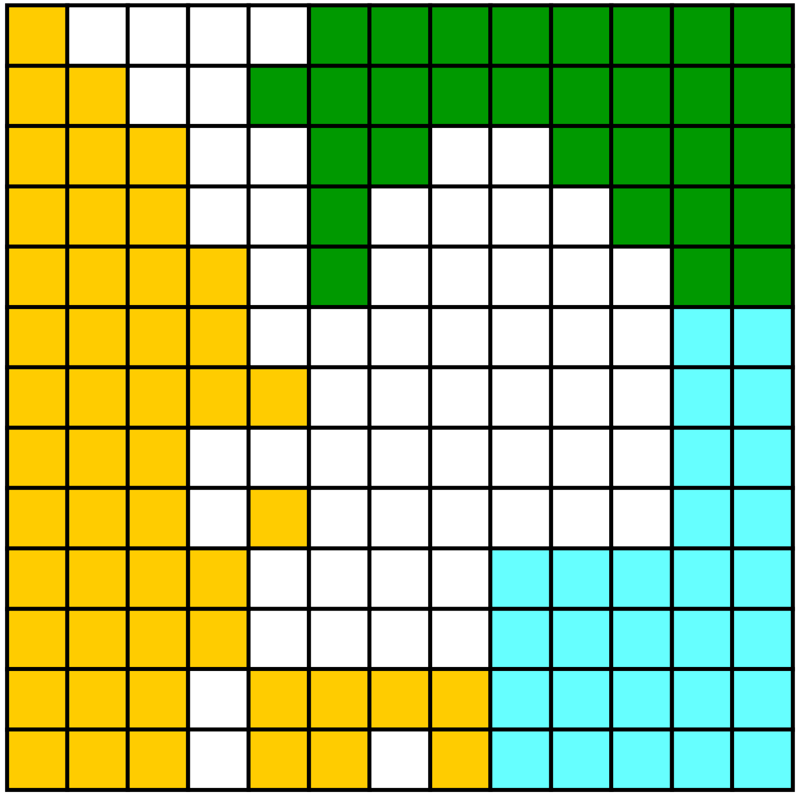
- Sierpinski Triangle





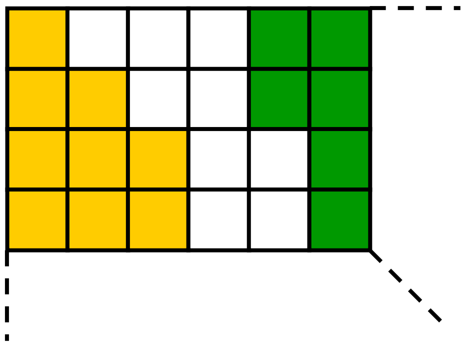
Multiple Recursion

- “Pouring” Paint in a painting program:
  - colour this pixel
  - spread to each of the neighbour pixels
    - colour the pixel
    - spread to its neighbours
      - colour the pixel
      - spread to its neighbours
        - …



Spreading Paint

```
private int ROWS = 25;
private int COLS = 35;
private Color[ ][ ] grid = new Color[ROWS][COLS];    // the grid of colours,

/** Spread new colour in place of oldColour on this cell and all its adjacent cells*/
public void spread(int row, int col, Color newColour, Color oldColour){
    if (row<0 || row>=ROWS || col<0 || col >=COLS) { return; }
    if ( ! grid[row][col].equals(oldColour) ) { return; }
    setPixel(row, col, newColour);
    spread(row-1,  col,      oldColour, newColor);
    spread(row+1, col,      oldColour, newColor);
    spread(row,      col-1,  oldColour, newColor);
    spread(row,      col+1,  oldColour, newColor);
}
```



Recursion that returns a value.

- What if the method returns a value?

  ⇒ get value from recursive call, then do something with it

  typically, perform computation on value, then return answer.

- Compound interest
  - value at end of *n* th year =

    value at end of previous year * (1 + interest).

value(deposit, year) = deposit [if year is 0]

= value(deposit, year-1) * (1+rate)

Recursion returning a value

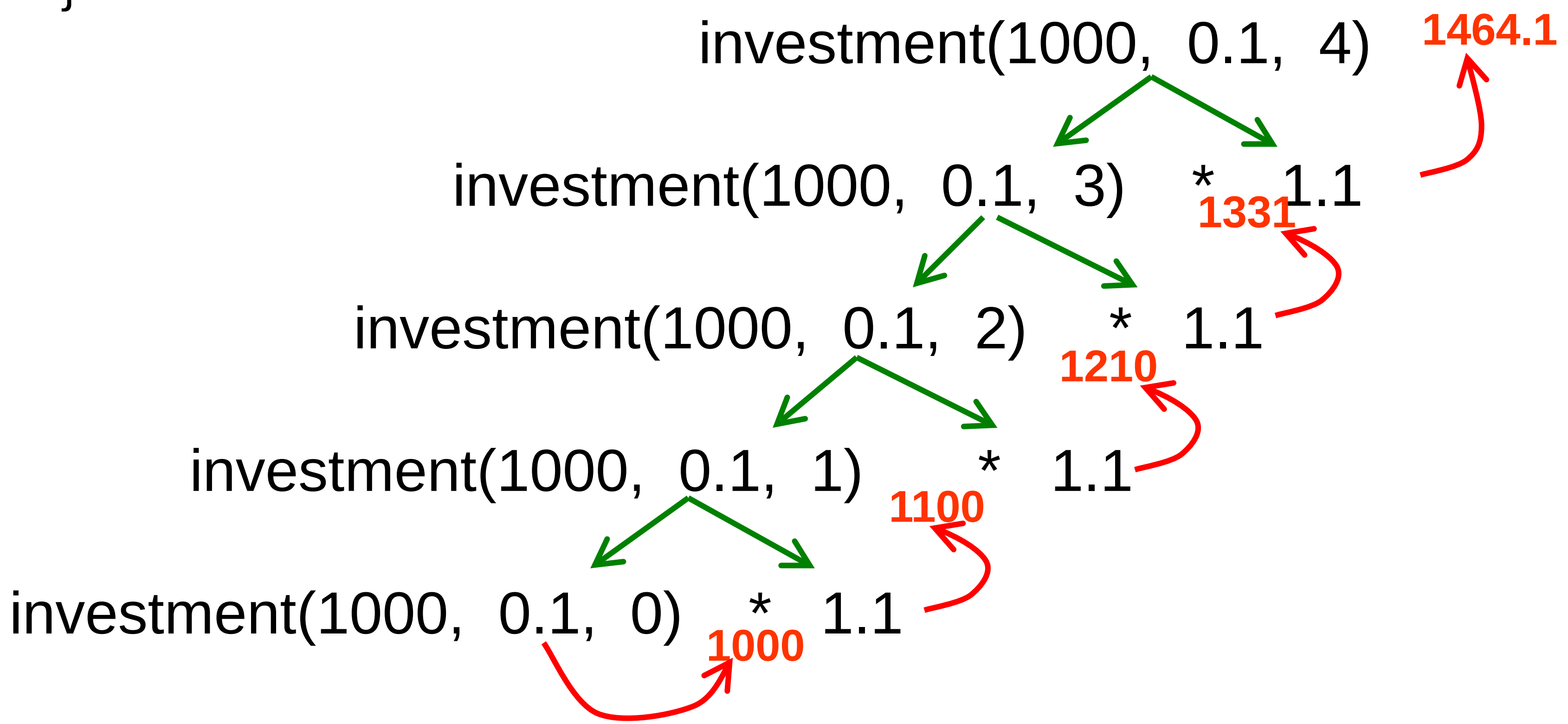
```
/** Compute compound interest of a deposit */
    public double compound(double deposit, double rate, int years){
        if (years == 0)
            return deposit;
        else
            return  ( this.compound(deposit, rate, years-1) * (1 + rate) );
    }
```

alternative :

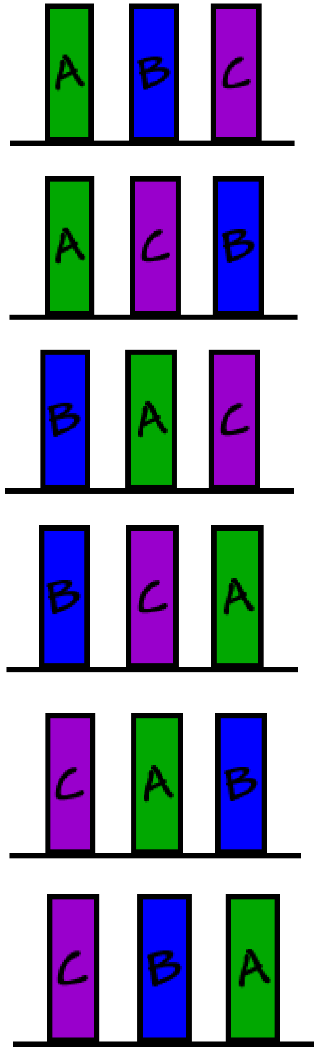
```
    public double compound(double deposit, double rate, int years){
        if (years == 0)
            return deposit;
        else {
            double prev = this.compound(deposit, rate, years-1);
            return  prev * (1 + rate);
        }
    }
```

Recursion with return: execution

```
public  double investment(double deposit, double rate, int year){
    if (year == 0)  {   return deposit;   }
    else {
        double prev = this.investment(deposit, rate, year-1);        ← step 1
        return  prev * (1 + rate);                                   ← step 2
        }
}
```



Recursion – An Example



- How many ways are there to arrange n books in a line?
- This number is called *n factorial* and is usually written as **`n!`**
- Example: 3! = 3 * 2 * 1 = 6
- For any positive integer **n** it is defined as the product of all integers from 1 to n inclusive:

```
n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

- This definition can also be expressed recursively:

```
1! = 1
n! = n * (n-1)!
```

- That is, a factorial is defined in terms of another (smaller) factorial until the base case of 1! is reached
- Note: some mathematical formulas have a very elegant recursive definition

Small exercise

- Can you find a way to calculate n! using recursion?
  - n * (n-1) * (n-2) * ….. * 3 * 2 * 1 = n!
- Tip:
  - Use the example as a template to solve it
  - When do you know the answer without any further call (base solution)?
  - What is the calculation of the current known value and the result of “the rest”

Factorial – Using Iteration

```
public int fact(int n){

    int result = 1;
    for (int  i = 1; i<=n; i++ ) {
        result *= i;
    }
    return result;
}

UI.println(fact(5));
```

120

Factorial – Using Recursion

```
public int fact(int n){

    if (n == 1) return 1;

    return n * fact(n-1);
}


//The runtime system creates a stack of results
```