Data Structures and Algorithms

XMUT-COMP 103 - 2025 T1

Recursion and Algorithm Complexity

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

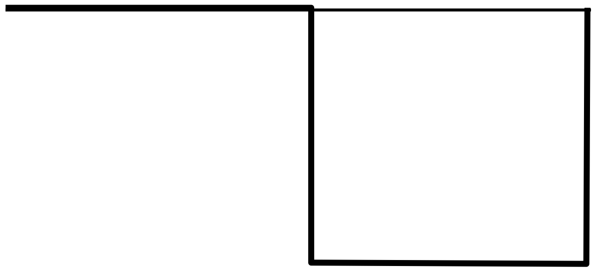
© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

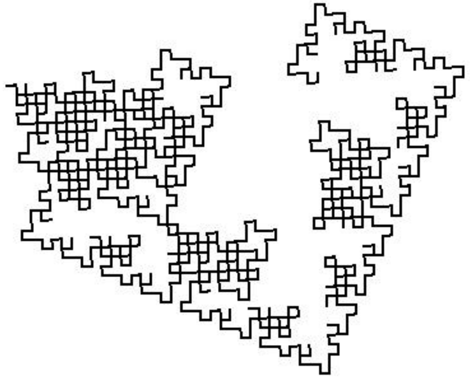
Recursion and Fractals

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

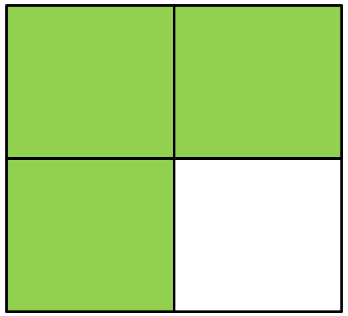
Simple examples:

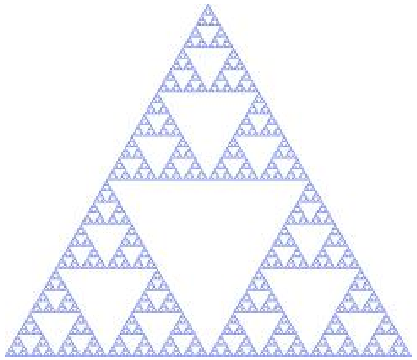
- Fractal Line





- Sierpinski Triangle

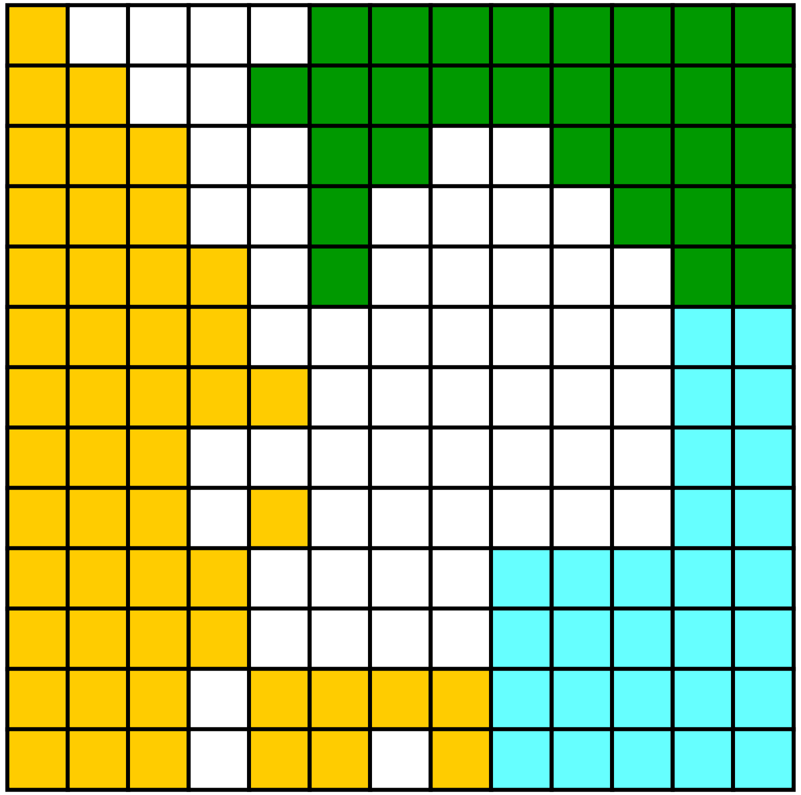




Sierpinski triangle image is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license (https://creativecommons.org/licenses/by-sa/3.0/deed.en).
Attribution: Beojan Stanislaus
© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

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
        - …

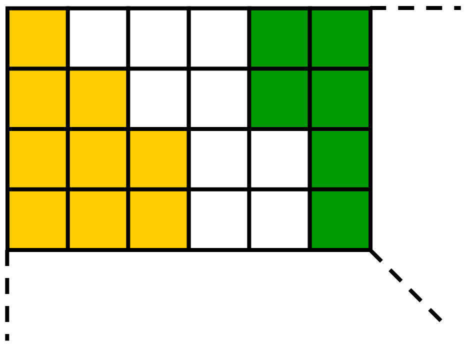


© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

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);
}
```



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

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)

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

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);
        }
    }
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

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
        }
}
```

investment(1000, 0.1, 4) 1464.1

investment(1000, 0.1, 3) * 1.1

1331

investment(1000, 0.1, 2) * 1.1

1210

investment(1000, 0.1, 1) * 1.1

1100

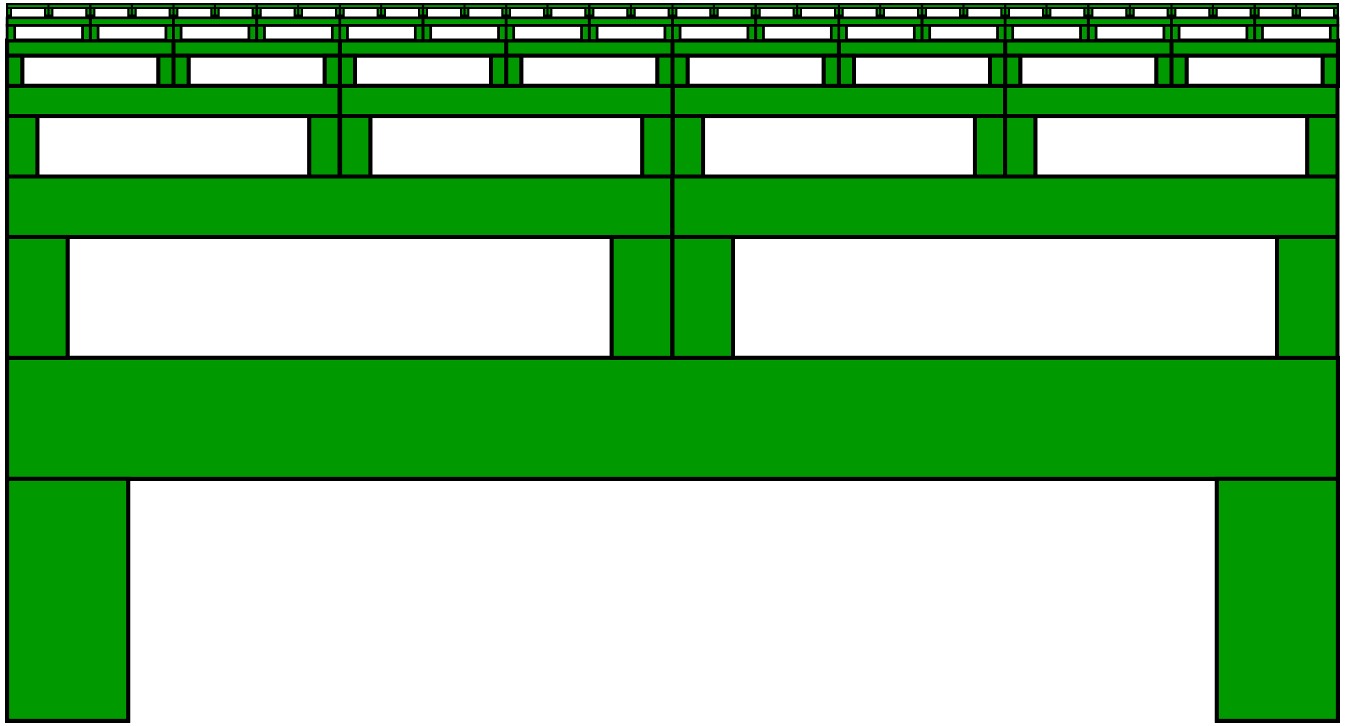
investment(1000, 0.1, 0) * 1.1

1000

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Multiple Recursion

- Draw a recursive arch-wall:
  - Consists of an arch with two half size arch-walls on top of it.



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

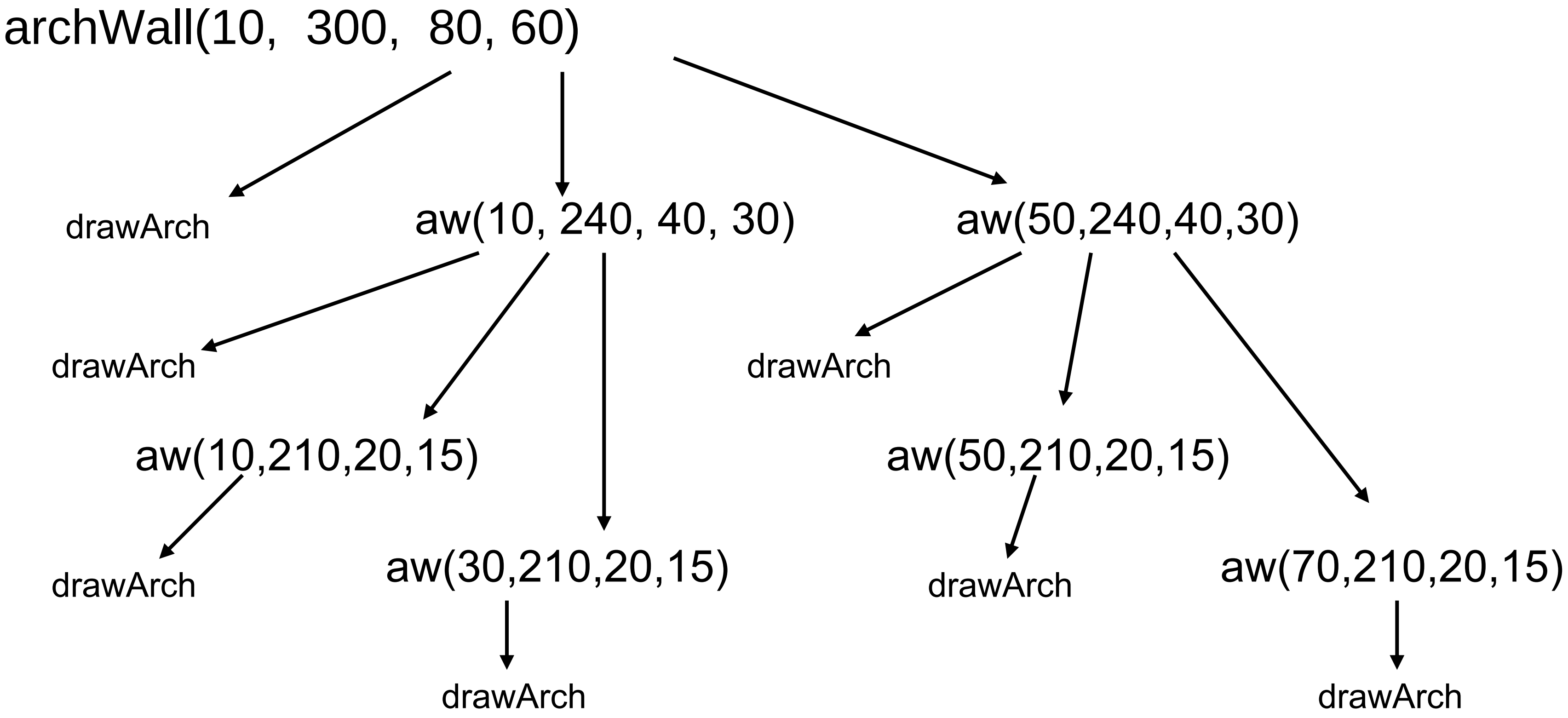
Multiple Recursion: ArchWall

- to draw an ArchWall of given base size (wd,ht):
  - draw a base arch of size (wd,ht)
  - if wd is not too small
    - draw a half size archwall on the left
    - draw a half size archwall on the right

```
public void archWall (int left, int base, int wd, int ht){
    this.drawArch(left, base, wd, ht);
    if ( wd > 20 ) {
        int w = wd/2;            // width of  smaller arch walls
        int h = ht/2;            // height of smaller arch walls
        int mid = left+w;        // x pos of right arch wall
        int top = base-ht;       // base of smaller arch walls
        this.archWall(left,  top, w, h);     // left half
        this.archWall(mid, top, w, h);       // right half
    }
}
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Tracing the execution:



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Analysing Costs (in general)

How can we determine the costs of a program?

- **Time:**
  - Run the **program** and count the milliseconds/minutes/days.
  - Count number of steps/operations the **algorithm** will take.
- **Space:**
  - Measure the amount of memory the **program** occupies.
  - Count the number of elementary data items the **algorithm** stores.
- Applies to Programs or Algorithms? *Both.*
  - programs ! “benchmarking”
  - algorithms ! “analysis”

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

What is a good algorithm?

Obviously needs to do what is expected consistently. However most problems can be solved in many ways. What is most important?

- Clarity - easy to read/implement
- Efficiency - the cost of running it

Clarity is relatively simple to measure. Find somebody else to read you code.

But how do we measure efficiency of an algorithm?

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Benchmarking: program cost

Measure:

- actual programs, on real machines, with specific input
- measure elapsed time
  - System.currentTimeMillis ()

    → time from the system clock in milliseconds
- measure real memory usage

Problems:

- what input? ⇒ use large data sets

  don't include user input
- other users/processes? ⇒ minimise

  average over many runs
- which computer? ⇒ specify details

- how to compare cross-platform? ⇒ measure cost at an abstract level

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Analysis: Algorithm “complexity”

- Abstract away from the details of
  - the hardware, the operating system
  - the programming language, the compiler
  - the specific input
- Measure number of “steps” as a function of the data size
  - best case (easy, but not interesting)
  - worst case (usually easy)
  - average case (harder)
- The precise number of steps is not required
  - $3.47\ n^2 - 67n + 53$ steps
  - $3n \log(n) + 5n - 3$ steps
- Rather, we are interested in how the cost grows with data size on large data

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Big-O Notation

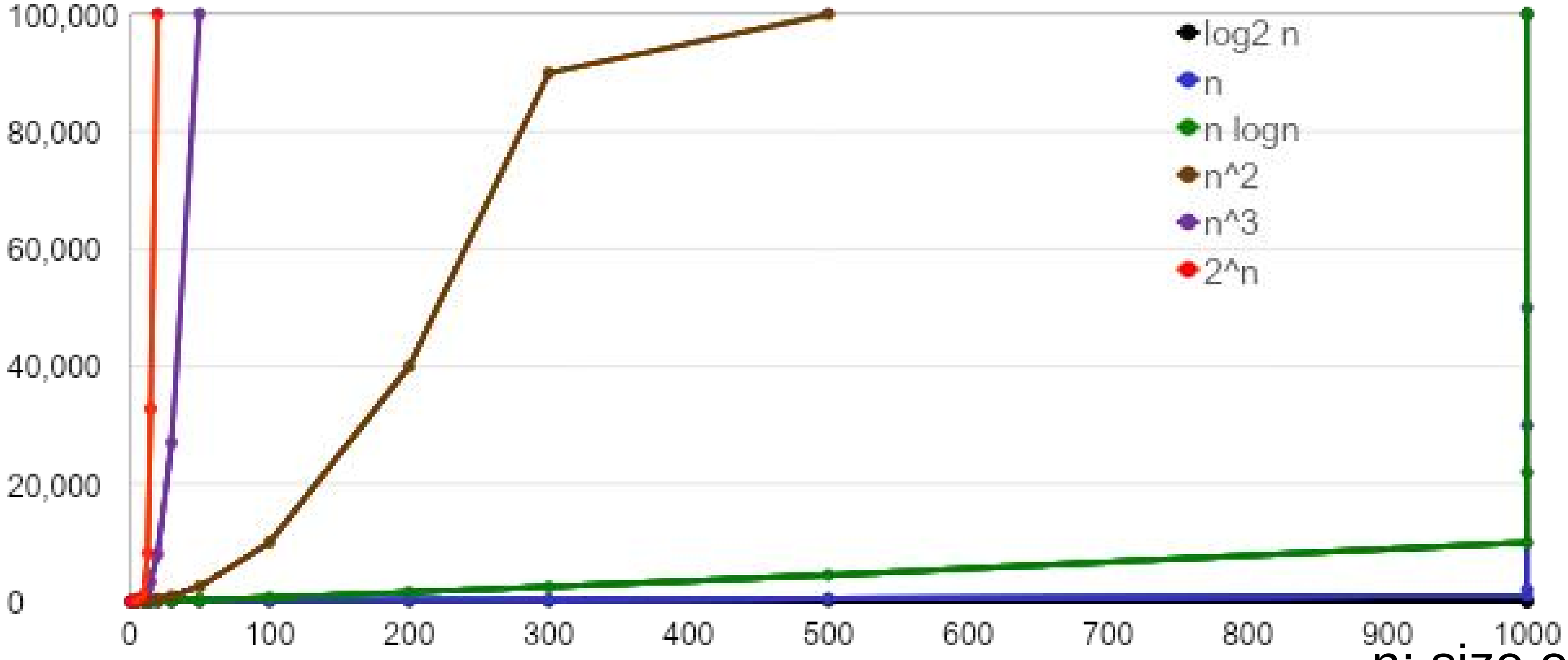
- “Asymptotic cost”, or “big-O” cost describes how cost grows with large input size
- Only care about large input sets
  - Lower-order terms become insignificant for large n
- We care about how cost grows with input size
  - Don’t care about constant factors
  - Multiplication factors (3, 102, 3 and 12 below) don’t tell us how things “scale up”
  - Lower-order terms become insignificant for large n

$3.47\ n^2 + 102n + 10064$ steps ! $O(n^2)$

$3n \log n + 12n$ steps ! $O(n \log n)$

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

How the different costs grow



n: size of input

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei