[TAP:DSQEV] Big-O

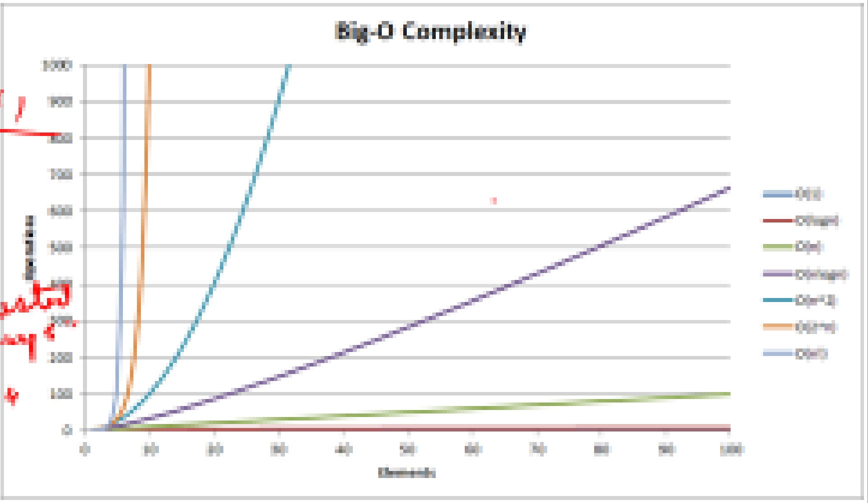
```
for (int i=0; i < arr.length; i++){
  for (int j=0; j < arr.length; j++){
    for (int k=0; k < arr.length; k++)
      System.out.println("digits: "
                          +arr[i]+arr[j]+arr[k]);
    for (int k=0; k < arr.length; k++)
      System.out.println("digits: "+arr[k]);
  }
}
```

- What is the time complexity of the code above?
  - A. O(n)
  - B. O(n^2)
  - C. O(n^3)
  - D. O(n^4)
  - E. Whatever

Asymptotic Analysis (Big-O Analysis)

- "How scalable is the algorithm?" as input size ↑
- Commonly split into the following *classes*:
  - O(1) : "constant" = "no loop through the input"
  - O(log n) : "logarithmic" or "log n"
  - O(n) : "linear" = "1 loop" eg. indexOf()
  - O(n log n) : "n log n"
  - O(n^c) : "polynomial"
    - O(n^2) : "quadratic" "2 nested loops"
    - O(n^3) : "cubic" "3 nested loops"
  - O(c^n) : "exponential"

"good"

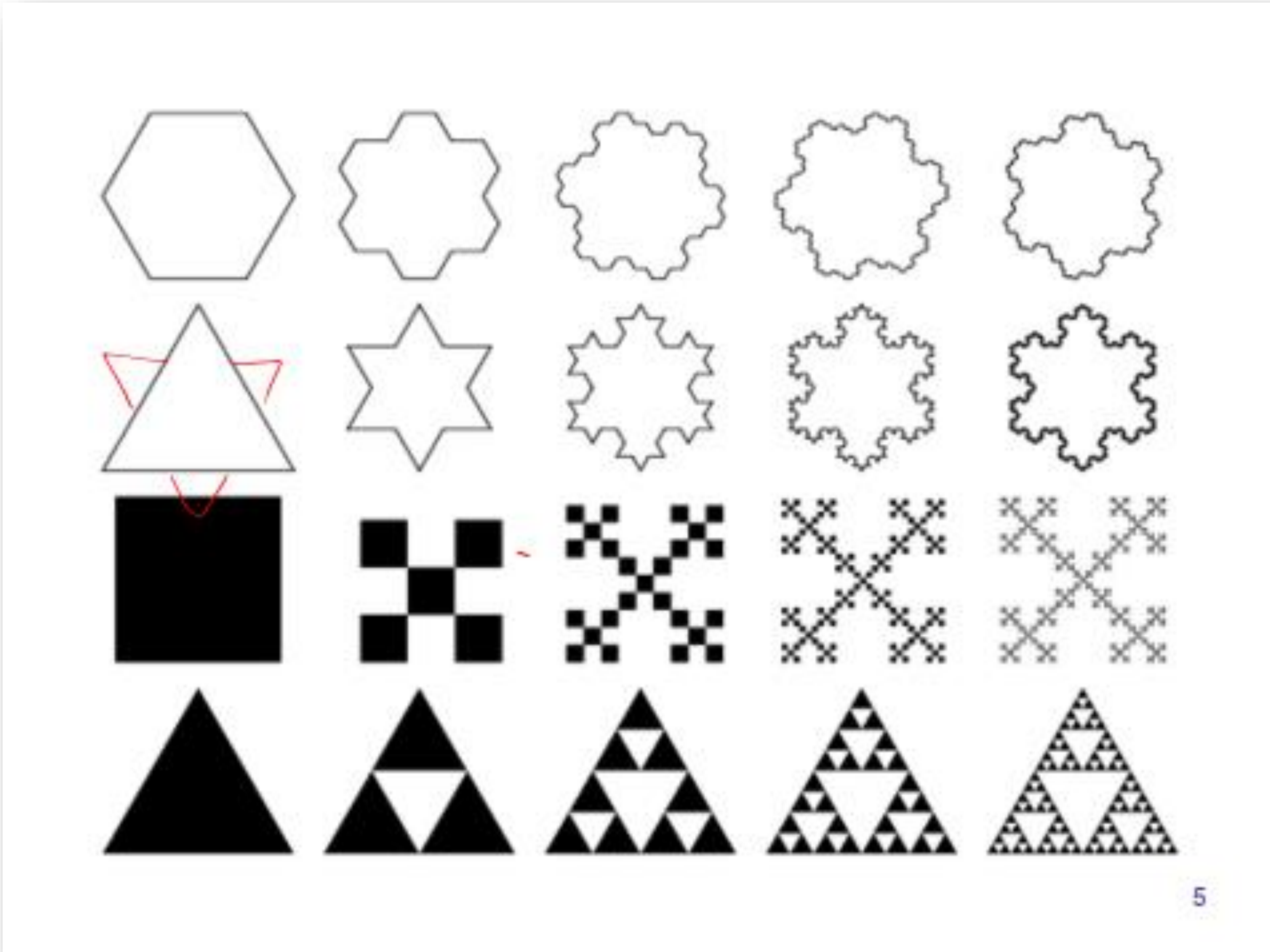


Administrative Details

- Lab 3
  - This is a partner lab; you get to work in groups of 2.
  - Please complete PRE-LAB before lab
    - **Submit to the google form, please!**

Agenda

- Recursion



Factorial

iterative

$n! = n \cdot n-1 \cdot n-2 \cdot \ldots \cdot 1$

recursive

$n! = n \cdot (n-1)!$

$0! = 1$

```
fact(int n){
    if (n==0)
        return 1;
    return n·fact(n-1);
}
```

base case

recursive case

```
fact(n){
    return n·fact(n-1);
}

fact(0){
    return 1;
}
```

Recursion

- In recursion, we always use the same basic approach/structure
  - base case ex) $n = 0$
  - recursive case ex) $n > 0$

Fibonacci Numbers

0   1   2   3   4   5   . . .

1 , 1 , 2 , 3 , 5 , 8 , . . .

$F_0 = 1$

$F_1 = 1$

for $n > 1$

$F_n = F_{n-1} + F_{n-2}$

fib()

```
//pre: n >= 0
//post: nth Fibonacci # is returned
public static int fib(int n){
    assert n>=0;
    //base case
    if (n==0)
        return 1;

    if (n==1)
        return 1;
    //recursive
    return fib(n-1)+fib(n-2);

}
```

```
if((n==0)||(n==1))
    return 1;
```

bad contains()

```
//Pre: nums != null
public static boolean contains(int[] nums, int x)
    if (nums.length == 0)
        return false;

    if (nums[0] == x)
        return true;

    int[] remaining = new int[nums.length - 1];
    for (int i = 0; i < remaining.length; i++)
        remaining[i] = nums[i+1];
    return contains(remaining, x);
}
```

inefficient!

contains()

```
public static boolean contains(int[] nums, int x){
    return containsHelper(nums, x, 0);
}

private static boolean containsHelper(int[] nums, int x,
                                      int curIdx)
    if (curIdx >= nums.length)
        return false;

    return nums[curIdx]==x ||
                    containsHelper(nums, x, curIdx+1);
}
```

canMakeSum()

{3, 5, 8} 12

{3 5 8}
{3 5 }
{3 8 }
{3 }
{ 5 8 }
{ 5 }
{ 8 }
{ }

$2^3 = 8$

```
Helper(int[] set,
       int targetSum,
       int index) {
   ...
   return Helper(set, targetSum - set[index],
                 index+1)
      || Helper(set, targetSum, index+1);
}
```

"include the current element in the subset"

"do not"

Recursion Tradeoffs

- Advantages
  - Code is usually cleaner
  - Some problems do not have obvious non-recursive solutions
- Disadvantages
  - Overhead of recursive calls
    - Can use lots of memory (need to store state for each recursive call until base case is reached)

assert

- Pre- and post-condition comments are useful as a programmer, but it not enforced.
- *assert* throws an error if the condition is not met!
- assert syntax
  - assert boolean_expression;
  - assert boolean_expression: String expression;

run java with "-ea"

ex) java -ea Recursion