Recursion & Performance

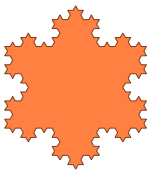
Part 7

Recursion

The best way to learn recursion…
is to, first, learn recursion!

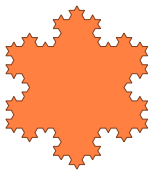
Recursion

- *Recursion* occurs when a function directly or indirectly calls *itself*
- This results in a loop
- However, it doesn't use iterative structures such as For or While loops



Recursion

- This can greatly simply programming tasks
- Commonly used to traverse a graph, tree, or run complex calculations
- … but can also create pitfalls



Breaking a Problem Down

- Recursion allows a problem to be broken down into smaller instances of themselves
- Each call will represent a smaller, simpler, version of the same problem
- Eventually, it will reach a "base case" which will not require any more recursive calls

Where Recursion Shines

- When the program can be broken into smaller pieces, recursion is a great solution
- Examples:
  - graph traversal – searching, etc.…
  - state machines
  - sorting
  - many math problems

Recursion Overhead

- Recursion, while powerful, is costly on computer resources
- Each time a function calls another function an *activation record* is placed on the *stack*
- Huh? You will learn about that in CSC 130

Okay, What Happens?

- When a function *A* calls *B*…
  - *A* is temporarily suspended
  - information about *A* – local variables, position, etc… is stored for later
  - the computer then transfers control to *B*
- When function *B* completes…
  - the information saved about function *A* is read
  - function *A* continues from where it left off

Danger: Never Ending

- If you break down a task into smaller parts… at some point, it should become a single value
- If not, the function will never end and will recurse forever – *at least until the computer runs out of resources*

Danger: Accidental Recursion

- Accidental recursion is a common mistake my beginner programmers
- It might be done directly or indirectly
  - for example: A calls B, B calls C, C calls A
  - so, it is important to organize your code carefully

Results of These Dangers…

- Either will crash your program
  - function will recurse *forever*
  - eventually all memory is exhausted
- You will see either…
  - "stack overflow" error
  - "heap exhaustion" error

Example: To infinity… but not beyond

```
void toInfinity()
{
   System.out.println("To infinity!");
   toInfinity();
   System.out.println("and beyond!");
}
```

We never get here!

Designing a Recursive Function

- Does the problem lend itself to recursion?
  - can the problem be broken down into smaller instances of itself?
  - is there a iterative version that is better
- Is there a base case?
  - is there a case where recursion will stop?
  - remember: ALWAYS have a stopping point!

Example: Factorials

- Factorials are classic mathematical problem that lends itself easily to recursion
- If you don't remember, a factorial of *n* is defined as the value of n multiplied by all lesser integers ≥ 1
- For example: 5! → 5×4×3×2×1 → 120

Example: Factorials

- It should be easy to observe that *n!* can be defined as *n × (n – 1)!*
- So, n! can be computed by multiplying n by the factorial of one less than it
- 4! → 4 × 3! → 4 × 3 × 2! → 4 × 3 × 2 × 1

Example: Factorials

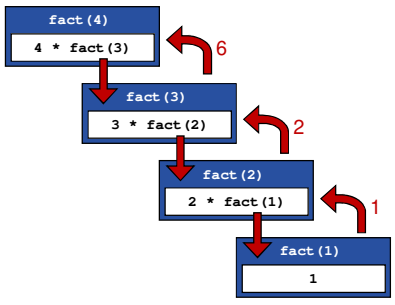
```
int fact(int n) {

   if (n == 1) {
     return 1;
   } else {
     return n * fact(n - 1);
   }
}
```

base case

Recursion

Example Factorial



Iteration vs. Recursion

- Any program that can be expressed using recursion, can be done through iteration
- The recursive solution will often be far simpler – more "eloquent" to read
- … but is never more efficient due to the overhead of calling functions

Historical Perspective

Some really cool solutions!

Some Well-known Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing

Some Well-known Problems

- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Fibonacci Numbers

- Rabbits tend to reproduce like… well… rabbits
- Mathematician Fibonacci analyzed this situation and created a mathematical system to predict this phenomena
- It is used today in finance, simulation, and several computer science algorithms

Fibonacci Numbers

- The problem:
  - start with a pair of rabbits
  - at month #2, the rabbits begin to reproduce
  - the female gives birth to a new pair of rabbits: one male and one female
  - babies mature at the same rate and will have babies
- Fibonacci number sequences predict the total pairs after *n* months

Fibonacci Numbers

- The problem:
  - start with a pair of rabbits
  - at month #2, the rabbits begin to reproduce
  - the female gives birth to a new pair of rabbits: one male and one female
  - babies mature at the same rate and will have babies
- Fibonacci number sequences predict the total pairs after *n* months

Fibonacci Numbers

- After two months, the female gives birth creating a new pair…. then they get pregnant again!
- This continues forever…..
- Sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
if n == 1 then Fib(n) = 1
if n == 2 then Fib(n) = 1
if n > 2  then Fib(n) = Fib(n-2) + Fib(n-1)
```


Example: Fibonacci Numbers

```
int fib(int n) {

  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fib(n-2) + fib(n-1);
  }
}
```

Recursion

Greatest Common Denominator

- A common problem in computer science is finding the greatest (or least) common denominator for two integers
- For example, the GCD of 64 and 40 is 8
- Euclid (of geometry fame) created an ingenious algorithm for finding the greatest common divisor

Euclid's Algorithm

- Euclid's algorithm is recursive
- You reapply the expression below until the second value of gcd(m,n) is zero.
- In this case, m will be the CGD

```
gcd(m,n) → gcd(n, m mod n)
```


Euclid's Algorithm Examples

- 60 and 24
  - gcd(60, 24) → gcd(24,12) → gcd(12, 0)
  - the result is 12
- 84 and 20
  - gcd(84, 20) → gcd(20, 4) → gcd(4, 0)
  - result is 4
- These might seem trivial, but it can find HUGE numbers quite easily

Order of Growth

Uh, "O"

Order of Growth

- One property of functions that we are interested in its rate of growth
- *Rate of growth* doesn't simply mean the "slope" of the line associated with a function
- Instead, it is more like the curvature of the line

Order of Growth

- What is important is how an algorithm's time grows as $n \rightarrow \infty$
- In computer science several types of growth occur

Order of Growth

- Algorithms will fall into one of these categories for worst-case, best-case, and average-case
- Examples:
  - how faster will it run on computer that is twice as fast?
  - how long does it take with double the input size?

Several Growth Functions

- There are several functions
- In increasing order of growth, they are:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - Log Linear ≈ n log n
  - Quadratic ≈ n^2
  - Exponential ≈ 2^n

Growth Rates Compared

n =	1	2	4	8	16
1	1	1	1	1	1
log n	0	1	2	3	4
n	1	2	4	8	16
n log n	0	2	8	24	64
n^2	1	4	16	64	256
n^3	1	8	64	512	4096
2^n	2	4	16	256	65536

Classifications

- Using the known growth rates…
  - algorithms are classified using three notations
  - these allows you to see, quickly, the advantages/disadvantages of an algorithm
- Major notations:
  - Big-O
  - Big-Theta
  - Big-Omega

Order of Growth

Notation	Name	Meaning
O (*n*)	Big-O	class of functions f(n) that grow no faster than n
Θ (*n*)	Big-Theta	class of functions f(n) that grow at same rate as n
Ω (*n*)	Big-Omega	class of functions f(n) that grow at least as fast as n

Big-O

- So, Big-O notation gives an *upper bound* on growth of an algorithm
- We will use Big-O almost exclusively rather than the other two

Big-O

- The following means that the growth rate of f(n) is no more than the growth rate of n
- This is one of the classifications mentioned earlier

$f(n)$ *is* $O(n)$

O(1)

- Represents a constant algorithm
- It does not increase / decrease depending on the size of n
- Examples
  - appending to a linked list (with an end pointer)
  - array element access
  - practically all simple statements

O(log n)

- Represents logarithmic growth
- These increase with n, but the rate of growth diminishes
- For example: for base 2 logs, the growth only increases by one each time n doubles

O(log n) Examples

- Searching for an item on a sorted array – (e.g. a binary search)
- Traversing a sorted tree

O(n)

- Represents an algorithm that grows linearly with n
- Very common in programming – for iteration
- Examples:
  - finding an item in a linked list
  - merging two sorted arrays

O(n log n)

- Represents an algorithm that has "log linear" growth
- These algorithms grow based on both n and n's log value

O(n log n) Examples

- Quick Sort
- Heap Sort
- Merge Sort
- Fourier transformation

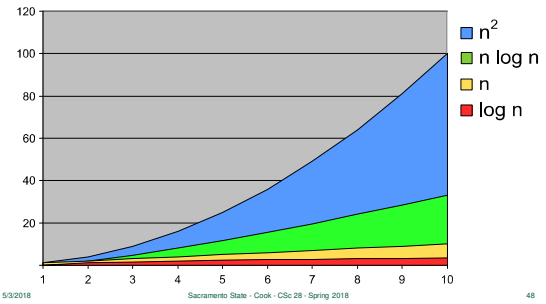
$O(n^2)$

- Represents an algorithm that has "exponential" growth
- These algorithms grow dramatically fast depending on the size of n
- Avoid for large values of n !

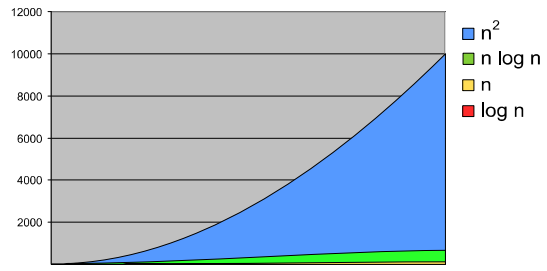
$O(n^2)$ Examples

- Bubble Sort, Selection Sort, etc….
- matrix multiplication
- merging unsorted arrays

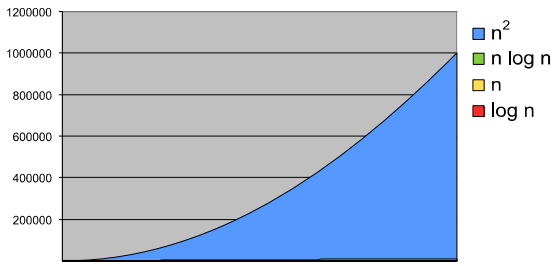
Growth: 1 to 10



Growth: 1 to 100



Growth: 1 to 1000



Time Given a 1 Microsecond Op

n	O(log n)	O(n)	O(n log n)	O(n^2)
10	0.000003	0.000010	0.000033	0.000100
100	0.000007	0.000100	0.000664	0.010000
1,000	0.000010	0.001000	0.009966	1.000000
10,000	0.000013	0.010000	0.132877	100.000000
100,000	0.000017	0.100000	1.660964	6.94 days
1,000,000	0.000020	1.000000	19.931569	1.9 years
10,000,000	0.000023	10.000000	232.534966	190.2 years

Why it is O-some!

- These classes make it is easy to…
  - compare algorithms for efficiency
  - making decisions on which algorithm to use
  - determining the scalability of an algorithm
- So, if two algorithms are the same class…
  - they have the same rate of growth
  - both are equally valid solutions

Big-O Math

Time for a "Big-O" headache

Asymptotic Analysis

- Any algorithm can be analyzed and its complexity/growth can be written an a simple mathematical expression
- *Asymptotic analysis* of an algorithm determines the running time in big-O notation

Asymptotic Analysis

1. Find the worst-case number of primitive operations executed as a function of the input size
2. Eliminate meaningless values the base rate in found

Asymptotic Analysis

- Example:
  - If we analyze an algorithm and find it executes 12 * n – 1
  - constant factors and lower-order terms dropped
  - they become meaningless for large values of n
  - remember, this is a *growth rate*
  - it will be "O(n)"

Examples

$3000n + 7$ is $O(n)$

$2n^5 + 3n^3 + 5$ is $O(n^5)$

$7n^3 - 2n + 3$ is $O(n^3)$

Test Your Might…

```
for(i = 0; i < 100; i++)
{
    total += values[i];
}
```

O(1)

Test Your Might…

```
for(x = 0; x < n; x++)
{
     sum += score[x];
}

for(x = 0; x < n; x++)
{
     sum -= score[x];
}
```

O(n)

Test Your Might…

```
for (x = 0; x < n; x++)
{
  for (y = 0; y < x; y++)
  {
    sum += x – y;
  }
}
```

O(n^2)