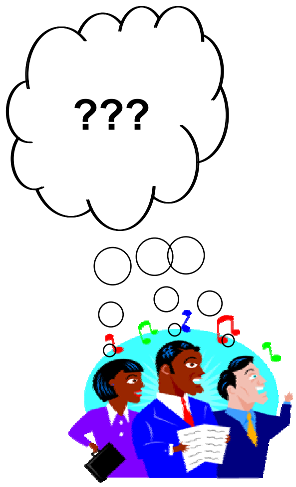
Introduction To CPSC 331

James Tam

Administrative Information For James Tam

- Contact Information
  - Office: ICT 707
  - Phone: 210-9455
  - Email: tamj@cpsc.ucalgary.ca
- Office hours
  - Office hours: MW (12:00 – 12:50)
  - Email: (any time)
  - Appointment: phone or call
  - Drop by for urgent requests (but no guarantee that I will be in!)

Feedback



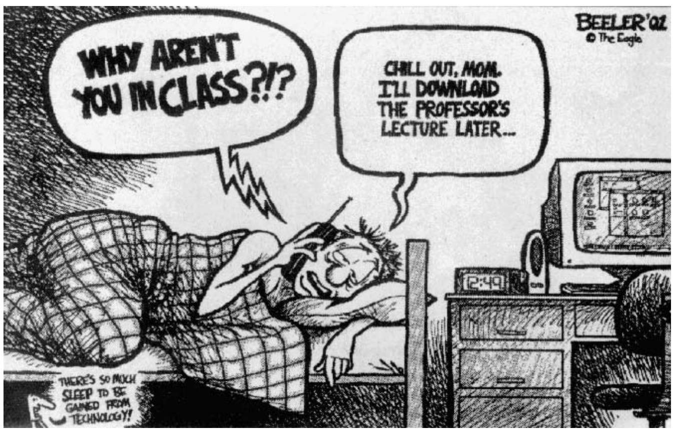
Course Resources

- Course website:
  - http://pages.cpsc.ucalgary.ca/~tamj/331
- Required course text book:
  - Data Structures and Algorithms in Java by Adam Drozdek
- Another good resource (previous version of the course)
  - http://pages.cpsc.ucalgary.ca/~marina/331/

How To Use The Course Resources

•They are provided to support and supplement the class.

•Neither the course notes nor the text book are meant as a substitute for regular attendance to lecture and lab



What This Course Is About

•Data Structures

•Algorithms

Data Structure

- A composite type that has a set of basic operations that may be performed on instances of that type:
  - The type may be a built-in part of the programming language
    - e.g., arrays are a basic part of the Pascal language
    - Some basic operations on arrays include: adding, deleting or modifying array elements.
  - The type may also be defined by the programmer inside a program
    - e.g. linked lists must be defined by the programmer when writing Pascal programs
    - Some basic list operations include: creating and initializing a new list, adding, deleting, modifying and searching the nodes on the list.

In this course you will learn how to define additional types of data structures

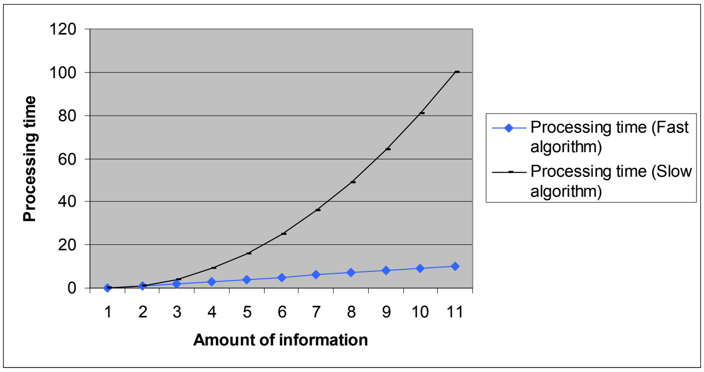
Algorithms

- "*An algorithm is clearly specified set of instructions to be followed to solve a problem.*" (Weiss)
- It is one the factors affecting the speed of software
  1. Type of inputs to the program e.g., Memory vs. disk access
  2. Type of program code e.g., High vs. low level languages
  3. Processor speed
  4. The complexity of an algorithm

The focus of this course will be on the last point

Algorithm Complexity

- This is the "cost" of an algorithm is typically measured in terms of computational time.



With a large data set the algorithm chosen can be the most important factor in determining the speed of a program

Algorithm Speeds

- Measured in terms of time needed "t" for a given amount of data "n"
- Time will be some function of n.
- T equals some function with n as input

How Many Times Do The Following Loops Run?

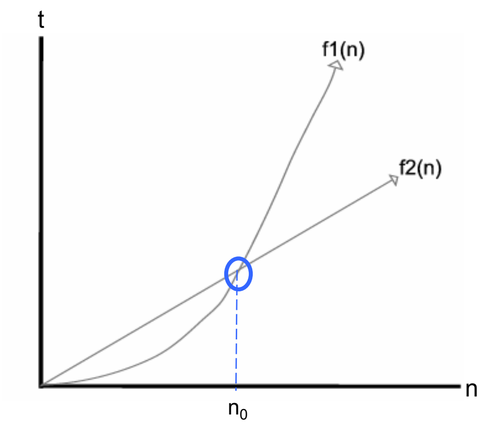
```
Example 1
for (int i = 1; i <= 5; i++)
      :           :

Example 2
for (int i = 1; i <= n; i++)
      :           :

Example 3
for (int i = 1; i <= 3; i++)
   for (int j = 1; j <= 5; j++)
            :           :
```


Common Algorithm Speeds

N	T = $Log_2 N$	T = N	T = $N*Log_2 N$	T = N^2	T = N^3	T = N^M M >3	T= 2^N	T= N!
1	0	1	0	1	1	1	2	1
2	1	2	2	4	8	16	4	2
4	2	4	8	16	64	256	16	24
8	3	8	24	64	512	4096	256	40320
16	4	16	64	256	4096	65536	65536	~ 20 trillion
32	5	32	160	1024	32768	~ One Million	~ Four Billion	REAL BIG!

Algorithm Analysis Focuses On Large Data Sets

- Notice that some of the slower algorithms run quicker than faster algorithms when the n values are smaller e.g., n! vs. n^3
- Algorithm analysis focuses **asymptotic efficiency of algorithms**: "after some (unspecified) point [n_0]" how fast will the algorithm run.



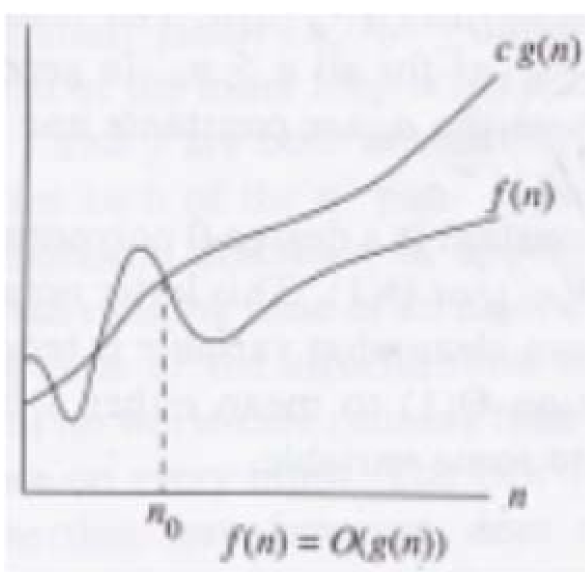
Algorithm Analysis Focuses On The Largest Term

- We are interested in analyzing performance for large data sets (i.e., when n is large).
- Therefore the slowest (largest) part of a function will be used to determine the speed of that function
- e.g., $t = n^2 + 10n + \log_2 n$

O-Notation

- The big *O*-notation describes a function g(n) that acts as the upper bound for the algorithm that we are trying to analyze f(n).
- If after a given number of inputs n_0, the values of f(n) are equal to or smaller than the values of g(n) then f(n) is in Big *O* of g(n).
- O(g(n)) = {f(n) : there exists positive constants c and n_0 such that 0 <= f(n) <= c*g(n) for all n >= n_0}.
  - n: The number of inputs to the function
  - c: A constant that accounts for factors such as machine speed, disk accesses, the number of program statements etc.
- Important concept for this course

Graphically Illustrating Big *O*-Notation

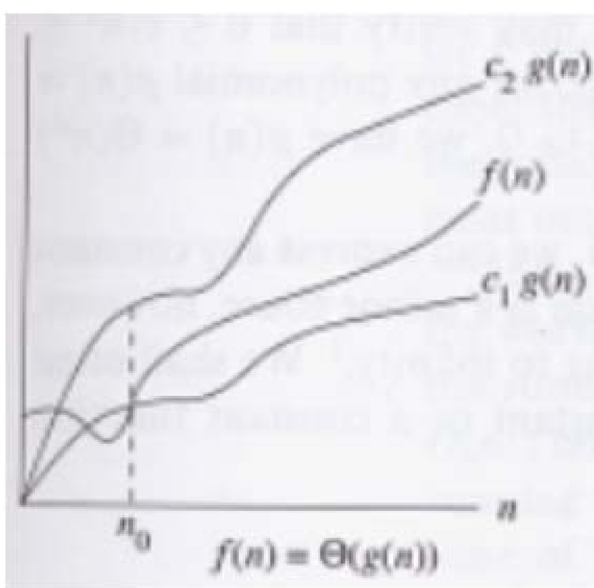


From “Introduction to Algorithms” by Cormen, Leirserson and Rivest

Θ-Notation

- The theta-notation describes a function g(n) that acts as an upper and lower bound for the algorithm that we are trying to analyze f(n).
- If after a given number of inputs n_0, there exists constants c_1 and c_2 such that the values of f(n) are "sandwiched" between c_1*g(n) and c_2*g(n), then f(n) is in Θ of g(n).
- Θ(g(n)) = {f(n) : there exits positive constants c_1, c_2 and n_0 such that 0 <= c_1*g(n) <= f(n) <= c_2*g(n) for all n >= n_0}.
- This means that f(n) is equal to g(n) within a constant factor.
- F(n) must be non-negative whenever n is sufficiently large so that g(n) must also be non-negative.

Graphically Illustrating The Θ-Notation



From "Introduction to Algorithms" by Cormen, Leirserson and Rivest

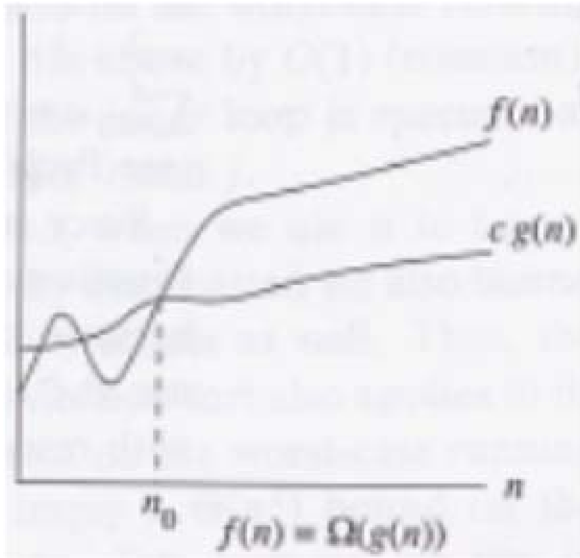
Big-*O* And Θ

- Theta is the more restrictive version of Big-*O* (*Asymptotically tight*)
- Theta is a subset of Big *O*

Ω-Notation

- The Omega-notation describes a function g(n) that acts as the lower bound for the algorithm that we are trying to analyze f(n).
- If after a given number of inputs n_0, the values of f(n) are greater than or equal to than the values of g(n) then f(n) is in Ω of g(n).
- $\Omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 <= c*g(n) <= f(n) <$ for all $n >= n_0\}$.

Graphically Illustrating The Ω-Notation



From "Introduction to Algorithms" by Cormen, Leirserson and Rivest

Determining The Running Time Of Code: Big-*O*

Assume that each statement takes up an equal amount of time.

1. Loops
   - `Running time of a loop = (running time of statements) * (loop iterations)`
   - Example

```
for (i = 1;          O(1)
    i <= n;          O(n+1)
    i++)             O(n)
      num += i * i;  O(1+1) – excludes loop
```

Loop running time = O(1) + O(n+1) + O(n)

= O(2n+2)

Body running time = O(2 statements * n repetitions) = O(2n)

TOTAL = Loop running time + Body running time

= O(2n + 2) + O(2n) = O(4n + 2)

= O(n)

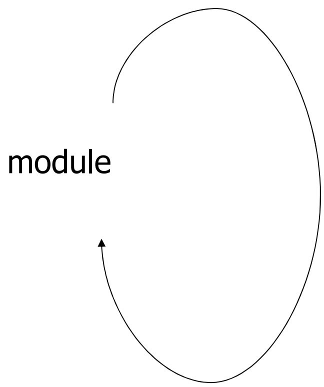
Determining The Running Time Of Code: Big-*O* (2)

2. Nested loops
   - Analyze it inside out:
   - `running time = (running time of statements) * (product of the sizes of all the loops)`
   - **This is the product rule**
3. Consecutive statements
   - Add the total number of executions
4. Decision making constructs
   - It's the running time of the test plus the running time of the largest of the conditions (**sum rule**)

Recursion In Programming

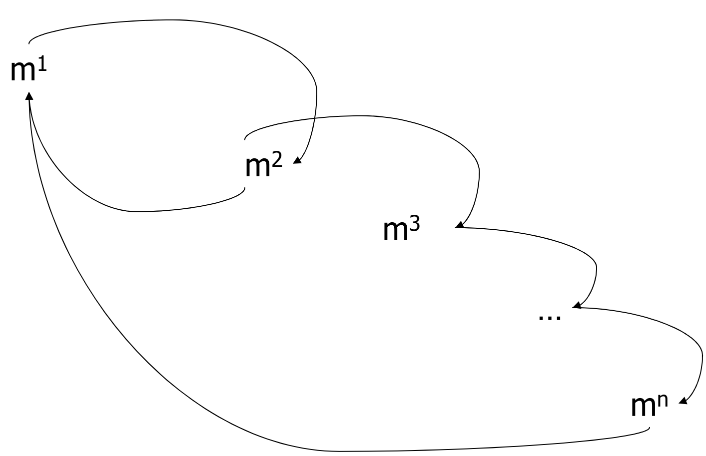
- "A programming technique whereby a function or procedure (called a method in Java) calls itself either directly or indirectly."

Direct Call




```
aMethod ()
{
    :
  aMethod ();
    :
}
```


Indirect Call



Indirect Call (2)

```
methodOne ()
{
    :
    methodTwo();
}

methodTwo ()
{
    :
   methodOne();
}
```


Determining The Running Time Of Simple Recursive Programs

Example:

The full example can be found in the directory: /home/331/tamj/examples/intro/simpleRecursion

(Given a positive number "num" the program will count from that number down to one).

```
class DriverRecursive
{
    public static void main (String [] args)
    {
        final int NUM = 5;
        RecursiveTail r1 = new RecursiveTail ();
        r1.count(NUM);
    }
}
```

This is a modified version of an example from "Data Structures and Abstractions with Java" by Frank M. Carrano and Walter Savitch

Determining The Running Time Of Simple Recursive Programs (2)

```
public class RecursiveTail
{
    public void count (int num)
    {
        System.out.println(num);
        if (num > 1)
            count(num-1);
        return;
    }
}
```


Solving The Recursive Example

- First determine what is the general recurrence relation
  - $t(n) = 1 + t(n-1)$ for $n >= 1$ and $t(1) = 1$

- Solve the relation for a value of n (e.g., n = 5)

$t(5) = 1 + t(4)$

$t(4) = 1 + t(3)$

$t(3) = 1 + t(2)$

$t(2) = 1 + t(1)$

$t(1) = 1$

This recursive function is in $O(n)$

Tail Recursion

- The last action is the method (aside from a return statement) is a recursive call

```
public void count (int num)
{
    if (num > 1)
        count(num-1);
    return;
}
```

- Tail recursion can be implemented easily as a loop.

```
public static void main (String [] args)
{
    int i;
    for (i = 5; i >=1; i--)
        System.out.println(i);
}
```


Non-Tail Recursion

- The last action (excluding the return statement) is not a recursive call.
- It is harder to convert to an iterative equivalent.
- (Given a number "num" the program will count up from one to that number).

```
public class RecursiveNonTail
{
    public void count (int num)
    {
        if (num > 1)
            count (num-1);
        System.out.println(num);
        return;
    }
}
```

An Example Of Inappropriate Use Of Recursion

- Fibonacci sequence is modeled on wabbit reproduction

Month zero

Month one 

Month two 

Month three  

Month four   

Image from www.rabbit.org

General Definition Of The Fibonacci Sequence

$$\text{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{otherwise} \end{cases}$$

Recursive Solution

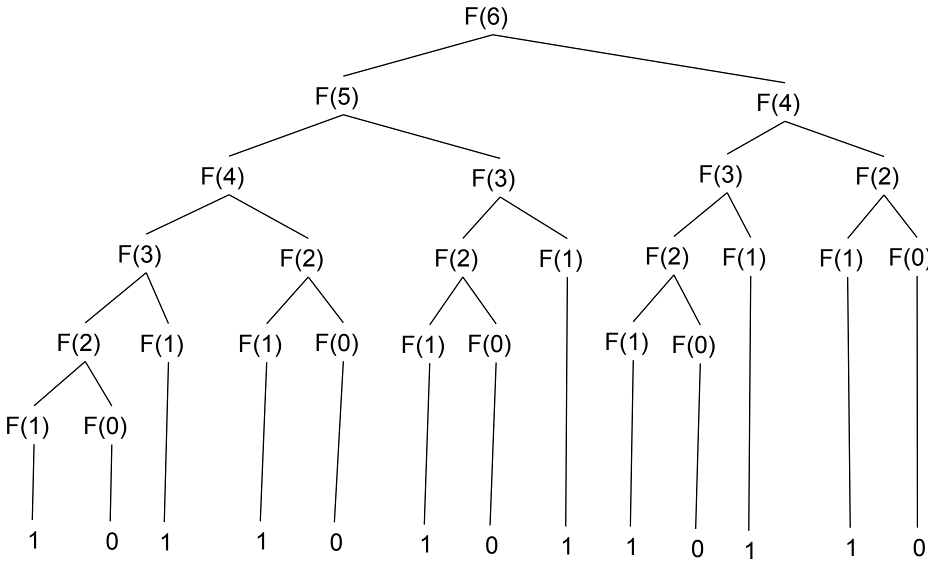
The full example can be found in the directory:
/home/331/tamj/examples/intro/fibonacciRecursion

```
class DriverFib
{
    public static void main (String [] args)
    {
        int num;
        Fib f = new Fib ();;
        System.out.print("Enter the no of fibonacci numbers to
           calculate: ");
        num = Console.in.readInt();
        System.out.println("Fib. of " + num + " = " +
           f.calculate(num));
    }
}
```


Recursive Solution (2)

```
public class Fib
{
    public int calculate (int num)
    {
        if (num < 2)
        {
            return num;
        }
        else
        {
            return (calculate(num-1) + calculate(num-2));
        }
    }
}
```

The Recursive Solution Is Redundant



You Should Now Know

- What is a data structure.
- What is an algorithm.
- What are some of the factors that effect the speed of a program.
- What is meant by algorithm complexity.
- What are some common algorithm speeds and how they rank vs. each other in terms of speed.
- What asymptotic notations mean in terms of algorithm analysis:
  - Big-O
  - Theta
  - Omega
- How to determine the worse case running time an algorithm (Big *O*-Notation)

You Should Now Know (2)

- Recursion
  - How to determine the running time of simple recursive programs.
  - What is tail recursion and how it differs from non-tail recursion.

Sources Of Material

- *Data Structures and Algorithms in Java* by Adam Drozdek
- *Data Structures and Algorithm Analysis in C++* by Mark Allen Weiss
- *Introducing Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
- *Data Structures and Abstractions with Java* by Frank M. Carrano and Walter Savitch
- CPSC 331 course notes by Marina L. Gavrilova http://pages.cpsc.ucalgary.ca/~marina/331/