Midterm Review Notes Ed 2.0

Preamble

In the second edition (2.0) of my review notes you will find a further in depth perspective (of my original perspective) on the material presented during the University of Waterloo's summer semester offering of CS 136. These notes are not aimed to replace lectures, tutorials, labs, office hours with your professor or TA, Piazza, or any other source directly provided by the course. It is solely aimed as a study aid that you can refer to for peer perspective on the course material.

These notes were written during the University of Waterloo's summer semester offering of CS 136 in 2013. It covers only midterm material (modules 1 - 6) and was written pre-midterm as a review for the midterm itself. The material that influenced the contents these notes are as follows:

- Lecture slides designed by the professors.
- "C Programming a Modern Approach – Second Edition", textbook for the course.
- Class averages on assignments (publicly posted on Piazza).
- Other posts on the CS 136 Piazza class.
- Student perspectives.

To preserve the chronological order that the material was presented during lecture, these review notes will follow a similar structure. However, some material in these review notes may be introduced earlier or later. This will only occur if we feel as if material flows better in a separate section than it was originally introduced in.

Practice exercises can be found scattered throughout each section of the review notes. There is a special section at the end dedicated to sample midterm problems based on emphasized course material, struggling assignment questions and other factors.

Prerequisites

Student: Can you give me some pointers for the CS 136 midterm?
Professor: 0xff98120f, 0x000000ff, 0x89ff7192, ...

It is expected that before you begin reading these notes that you have attended every lecture, completed assignments 1 through 5 (and possibly 6) and that you have already been introduced to a majority the material presented in these notes.

Disclaimer

Special Notices

These review notes would not be possible without the offering of CS 136 at the University of Waterloo and all the hard work the professors, TAs and behind the scenes administration put into this course. This is a truly fantastic course and every student involved in creating these review notes are absolutely loving it. Special mentions to the following students:

- Jacob Pollack, author (**jpollack@uwaterloo.ca**).
- Jacob Willemsma, co-author.
- Nimesh Das, pilot student.

Acknowledgements

Some further acknowledgments to the authors of the CS 136 material and other influencing figures.

- Professors and Teaching Assistants for the summer semester of 2013: Dave Tompkins, Olga Zorin and Patrick Nicholson.
- Design inspired by ”Linear Algebra Course Notes” by Dan Wolczuk (with permission).

Table of Contents

Preamble . . . i
Disclaimer . . . ii
Special Notices . . . ii

1 Introduction: Module 01 1
1.1 Full Racket . . . 3
1.2 Binary Search Trees . . . 9
1.3 Documentation (Design Recipe) . . . 11

2 Modularization: Module 02 12
2.1 The Interface . . . 14
2.2 Using Racket Modules . . . 16

3 Functional C: Module 03 17

4 C Memory Model: Module 04 18

5 Imperative C: Module 05 19

6 Efficiency: Module 06 20
6.1 Efficiency . . . 22
6.2 Input Size . . . 23
6.3 Ordered Runtime: Big O . . . 24

1 Introduction: Module 01

The official catalog entry for CS 136 at the University of Waterloo is as follows:

DEFINITION
CS 136

This course builds on the techniques and patterns learned in CS 135 while making the transition to use of an imperative language. It introduces the design and analysis of algorithms, the management of information, and the programming mechanisms and methodologies required in implementations. Topics discussed include iterative and recursive sorting algorithms; lists, stacks, queues, trees, and their application; abstract data types and their implementations.

In the prerequisite course(s) – CS 135 (and CS 115/116), the focus was programming in a functional paradigm in Racket (and in CS 116, Python). However in CS 136, as I hope you are now aware, the focus has shifted to programming in an imperative paradigm in C (C99). Let's look into what it means to program in both paradigms.

DEFINITION
Functional Paradigm

The process of evaluating code as a sequence of Mathematical functions, avoiding mutation and the manipulation of state. It has an emphasis on function definition, function application and recursion.

DEFINITION
Imperative Paradigm

The process of evaluating statements that change a programs state. This is done through mutation and control flow.

There is a third paradigm, declarative programming, however we are not introduced this in the course and hence it will not be referenced again throughout these review notes.

From looking at these paradigms it is fair to conclude that one is the opposite of the other. But do not take my word for it, let's see an example.

EXAMPLE 1

Consider the following two functions that return the sum from 0...n of a positive integer integer n.

```
int sum_recursive( int const n ) {
  return ( 0 == n ) ? 0 : ( n + sum( n - 1 ) );
}
```

Remark that this is the functional approach to handling this problem. Now consider the following.

```
int sum_iterative( int const n ) {
  int acc_sum = 0;

  for( int i = 0; i <= n; i++ ) {
    acc_sum += i;
  }

  return acc_sum;
}
```

Remark that this is the imperative approach to handling this problem. It is important we understand both paradigms as it is likely that we will be asked to program in a specific paradigm. These review notes will touch base on this in module 05, for now consider only the functional approach to problems.

1.1 Full Racket

It is true! We are finally old enough (as Dave says) to use full Racket. However there are a few things we will need to keep in mind in order to use it properly. Recall that to enable full Racket you must have the following definition at the top of every Racket file.

```
#lang racket
```

Functions

Keep in mind there is some funky terminology for Racket. We **apply** functions which **consume** arguments and **produce** a value. Consider the following example.

EXAMPLE 1 The following function, f consumes two integers, x and y and produces the sum of x and y.

```
(define (f x y)
  (+ x y))
```

Constants

Recall how to define a constant in Racket.

```
(define age 19)
```

When you are defining a constant in Racket it is an immutable identifier. We will see later on how to mutate an immutable identifier in Racket, however not now. Consider the following example.

EXAMPLE 2 The following function, *get_pennies* consumes one integer, n and produces the value of n pennies.

```
(define penny 1)

(define (get_pennies n)
  (* n pennies))
```

Constants are quite useful to improve **readability** and ease of **maintainability**. What if the value of a penny changed from 1 cent to 2 cents and we have over 100 different functions using the value of a penny. All we would need to do is change one constant as oppose to changing the value of a penny in over 100 separate places.

Functions without Parameters

This is indeed possible. However, you will find this feature not very useful until we introduce **side effects**. For now, just be aware of what they are. Consider the following example.

EXAMPLE 3 The following function, *get_lucky_number* consumes nothing. It produces a lucky number.

```
(define (get_lucky_number)
  9)
```

But how do we define a number as lucky? This is up to interpretation however a lucky number in my opinion is 9 (trick question).

Top-Level Expressions

These are rather interesting. If you recall, Racket files will be executed line by line. What would happen if you were to place an expression above a function? Consider the following example.

EXAMPLE 4 First consider what the following Racket file will do. Then open Dr Racket and discover for yourself what it really does.

```
#lang racket

; A top-level expression.
(+ 3 4) ; => 7

; A function definition.
(define (f x y)
  (+ x y))

; Some more top-level expressions.
(f 3 4) ; => 7
(f (f 3 4) (f 3 4)) ; => 14
(* 2 8) ; => 10
```

You should notice that it outputs 7, 7, 14 and 10 to the console. These are are **top-level expressions** that get executed and output to the console.

Logic in Racket

Recall booleans, symbols, strings, characters, logic operators (and ... or) and conditional statements from CS 135 (or CS 115). These are still very important for the Racket portion of CS 136. If you remember the syntax for a conditional statement, we are now introduced to a newer, more compact syntax. Consider the following.

EXAMPLE 5 The following function will consume a symbol and output $\#t$ or $\#f$ if the symbol is my favorite color.

```
(define favorite-color #\b) ; Char b for blue.
(define favorite-color "blue") ; Blue as a string.
(define favorite-color 'blue) ; Blue as a symbol.

; Older syntax.
(define (guess-color color)
  (cond
    [(symbol=? color favorite-color) #t]
    [else #f]))

; Newer syntax.
(define (guess-color color)
  (if (symbol=? color favorite-color) #t #f))

; .. or solely for this specific function.
(define (guess-color color)
  (symbol=? color favorite-color))
```

Keep in mind that you would think 0 is considered as $\#f$ in Racket, however you are wrong. Anything that is not explicitly $\#f$ is considered true in Racket, including 0.

```
(equal? #f 0) ; => False
(equal? #f 1) ; => False
(equal? #t 0) ; => True
(equal? #t 1) ; => True
```

Structures

No, we are not getting rid of these bad boys anytime soon. In full Racket the syntax for structures become more compact with one little tedious, but important detail. Consider the following.

EXAMPLE 6 Carefully examine the definition of how the *posn* structure is defined in full Racket. Do you spot anything new?

```
(struct posn (x y) #:transparent)
```

If you noticed, it is no longer $define - struct$, it is now just $struct$. In addition, we must add the $\# : transparent$ keyword at the end of the structure.

EXERCISE Create a structure in Racket without the $\# : transparent$ keyword then define a random *posn* and observe what happens when you execute the Racket file.

Lists

From CS 135 (or CS 115) we should be comfortable using $cons$, $list$, $empty$, $first$, $rest$, $list - ref$, $length$, $append$, $reverse$, $last$ and $drop - right$. Consider the following.

EXAMPLE 7 Examine the use of *cons* and *list*. The function *is_equal* will compare two lists and you can assume it works fine.

```
(define lst1 (cons 1 (cons 2 (cons 3 empty))))
(define lst2 (list 1 2 3))

(define (is_equal lstx lsty)
  (cond
    [(and (empty? lstx) (empty? lsty)) #t]
    [(or (empty? lstx) (empty? lsty)) #f]
    [(equal? (first lstx) (first lsty)) (and #t (is_equal (rest lstx)
                                                          (rest lsty)))]
    [else #f]))

(equal? #t (is_equal lst1 lst2)) ; => True
lst1 ; => '(1 2 3)
lst2 ; => '(1 2 3)
```

If you recall even further, *member* will check whether an element is a memory of a list. In full Racket *member* now produces the tail of the list if true, otherwise $\#f$. Keep in mind only $\#f$ is false in Racket, meaning the tail of a list is still considered true.

Abstract List Functions

Recall what an **abstract list** function is? They are built-in functions to Racket that perform useful operations on lists. The important feature for abstract list functions is that we can pass parameters to them.

We should be familiar with all the following abstract list functions: $filter$, $build - list$, map, $foldl$ and $foldr$. Consider the following examples.

EXAMPLE 8 Examine the use of $filter$, $build - list$, map, $foldl$ and $foldr$ in the following top level expressions.

```
(define lst1 (build-list 5 values)) ; => (list 1 2 3 4 5)
(define lst2 (build-list 5 values)) ; => (list 1 2 3 4 5)

(filter odd? lst) ; => (list 2 4)
(map + lst1 lst2) ; => (list 2 4 6 8 10)
(foldl + 0 lst1) ; => 15
(foldr + 0 lst1) ; => 15
```

Lambda

The use of *lambda* in Racket is probably one of my favorite language features. It is so powerful, if you do not know what it is your mind will be blown! Consider the problem that, we want to add a small bit of functionality that will only be used once and is exclusive to a specific task. We could create a helper function, however it will get messy. What about an anonymous function? Consider the following example.

EXAMPLE 9 Observe the use of *lambda* as an anonymous function to add functionality that would otherwise require the use of a helper function.

```
(define lst1 (build-list 10 values)) ; => (list 1 ... 10 )

(filter (lambda (x) (and (even? x) (> x 5))) lst1) ; => (list 6 8 10)
```

The power of *lambda* in Racket is endless.

EXERCISE Use the abstract list function $build - list$ to create a list of squares from 0 ... 10. Hint, use *lambda*.

Implicit Locals

We no longer need to explicitly state *local* while we are writing local constants or helper functions. This feature can still however be used but it is not needed nor recommended in full Racket. The *local* special form is now implicit.

1.2 Binary Search Trees

The most useful tree we have seen in CS 135 (and CS 115) which we will continue to use is the **Binary Search Tree** (BST). Here let's make an important definition describing what a valid BST is.

DEFINITION
Binary Search Tree

A node-based binary tree data structure, sometimes referred to as a sorted binary tree, which has the following properties.

- The left subtree of a node contains only keys less than the node's key.
- The right subtree of a node contains only keys greater than the node's key.
- The left and right subtree must also be valid BSTs.
- There cannot be any duplicates.

These properties are also referred to as the **ordering property**.

Remark that an empty BST can be represented by the keyword *empty*, however any sentinel value (such as $\#f$) is accepted. I prefer to stick with *empty* and hence it will be used in any further examples.

In this course, a BST is defined as follows.

```
(struct bst-node (key val left right) #:transparent)
```

Questions where we are supposed to create functions that perform some operation on a BST will 99% of the time have some form of recursion. A BST is a recursive data structure and hence recursion is more often than not used to access nodes. Consider the following examples to jog your memory and improve your understanding.

EXAMPLE 1

Write a function *key_exists?* that will consume two arguments, a key and a valid BST and will produce $\#t$ if the key exists, otherwise $\#f$.

```
(struct bst-node (key val left right) #:transparent)

(define (key_exists? key abst)
  (cond
    [(empty? abst) #f]
    [(equal? key (bst-node-key abst)) #t]
    [else (or (key_exists? key (bst-node-left abst))
              (key_exists? key (bst-node-right abst)))]))
```

EXAMPLE 2 Write a function *insert_node* that will consume three arguments, a key, a value and a valid BST and will produce the new BST with the key and value added to the BST. If the key already exists, it will overwrite the value of that node.

```
(struct bst-node (key val left right) #:transparent)

(define (insert_node key val abst)
  (cond
    [(empty? abst) (struct key val empty empty)]
    [(equal? key (bst-node-key abst)) (struct key val
                                         (bst-node-left abst)
                                         (bst-node-right abst))]
    [(> key (bst-node-key abst)) (struct (bst-node-key abst)
                                    (bst-node-val abst)
                                    (bst-node-left abst)
                                    (insert_node key val
                                                (bst-node-right abst)))]
    [else (struct (bst-node-key abst)
                  (bst-node-val abst)
                  (insert_node key val (bst-node-left abst))
                  (bst-node-right abst))]))
```

EXAMPLE 3 Write a function *sum_vals* that will consume one argument, a valid BST and will produce the sum of all the values in the BST. You can assume all values are integers.

```
(struct bst-node (key val left right) #:transparent)

(define (sum_vals abst)
  (cond
    [(empty? abst) 0]
    [else (+ (+ (bst-node-val abst) (sum_vals (bst-node-left abst)))
             (bst-node-right abst))]))
```

EXERCISE Write a function *get_values* that consumes two arguments, a non-empty list of valid keys and a valid BST. It will produce a list of values from smallest key's value to greatest key's value.

1.3 Documentation (Design Recipe)

This is a very tedious but necessary aspect of designing successful software. You may have noticed that not all the questions on the assignments given out by the professors are hand marked and you may be thinking, why bother to include a design recipe? Regardless of hand marking or not, it is required as it does two important things:

- To help us design new functions from scratch.
- To aid in the communication of our function to other developers.

This may not seem useful now, however as we proceed into module 02 it will become more apparent with respect to modularization. Recall from CS 135 (and CS 115) that there were quite a few elements to the design recipe. Let's define what the design recipe is in CS 136.

DEFINITION
Design Recipe

This is the process of communicating information about our function to a client, another developer or our self through smart documentation. It should communicate the following information:

- The contract of the function (ie what it consumes and produces).
- The purpose of the function.
- The pre and post conditions of the function (ie what are the input and output restraints).

The design recipe should be commented out above the function it is describing. Consider the following example.

EXAMPLE 1

Write the design recipe for the Racket equivalent of the *sum* function on page 1 of these review notes.

```
; (sum n): Int -> Int
; Purpose: Consumes an integer, n and produces a value greater than
;          or equal to 0. This value will be the sum from 0 ... n.
; PRE: n >= 0
; POST: produces an integer >= 0
(define (sum n)
  ...)
```

Well documented code is one of the most important features to have in your projects. Documentation must be written for all functions and helper functions (including locally defined helper functions).

2 Modularization: Module 02

One of the struggles in modern day computing is how do we collectively work a project. Not only that but for a large program with a large collection of functions, how do we break it down into smaller parts. This is where the push for modularization comes in.

DEFINITION
Modulariza-tion

Is the process of separating functionality from a program into smaller, independent and interchangeable modules such that each module has a well defined purpose. Each module should have low coupling and high cohesion.

Without modularization it would be extremely difficult to work as a collective towards a common goal on a project. There are three key pushes for modularizing a project, **re-usability**, **maintainability** and **abstraction**.

DEFINITION
Re-usability

The process of writing modules that can be taken from one project and applying them to future projects. For example, creating a module that handles MySQL database interaction can be applied to multiple projects that require the ability to access a MySQL database.

DEFINITION
Maintainabil-ity

The process of separating the project into multiple modules with a specific purpose, enabling multiple programmers to work on different aspects of project in parallel. In particular, if one module needs an update then a programming can simply extract that module, update it, then plug it back into the project without impeeding on other development on the project.

DEFINITION
Abstraction

The process of using a module designed by another company or programmer without actually knowing how it works.

Some of these terms are better understood with an example, consider the following.

EXAMPLE 1

Going into the garage to repair your car radio. Instead of heading to your local Ford dealer and buy a new car, you simply extract the radio from the car and replace it with a new radio. This is an example of maintainability.

EXAMPLE 2

Putting triple A batteries into your flashlight before you head out camping. This is an example of abstraction.

EXAMPLE 3 Taking notes in your highschool calculus course and then realizing that Math 137 will spend a lot of time reviewing what you learned in highschool. So instead of taking extensive new notes, you use your old notes from highschool! This is an example of re-usability, something I did not do :(

2.1 The Interface

One of the most important aspect of modularization is developing the interface for a module.

DEFINITION
Interface

Consists of a collection of functions (function definitions) that are accessible outside of the module (public), as well as the appropriate documentation for those given functions. In short, everything a client would need in order to use our module.

But how do we know what a successful modular design should ahieve. What standards we should meet in our design. If you recall from the definition of modularization the terms low coupling and high cohesion were used. In successful modular design we aim to achieve both low coupling and high cohesion.

DEFINITION
Low Coupling

There are as few modular inter-dependencies as possible.

EXAMPLE 1

If you have a module referencing many functions from another module, you will need to copy both modules over if you plan to re-use any of the functionality. This is an example of high coupling since you cannot extract modules without extracting other modules it depends on.

DEFINITION
High Cohesion

All the functions within a given module are working collectively towards a common goal or purpose.

EXAMPLE 2

If there was a module designed to make craft dinner and there was a function that ordered a coke zero from the vending machine, it would not contribute towards the common goal of the module and hence would lead to low cohesion.

Information Hiding and Documentation

It can be important to hide information about your module from the client. Consider you were contracted by TD Canada Trust (a Canadian bank) to implement a module that would handle deposits and withdrawls from an ATM. You will want to hide some of the implementation of your module to avoid letting the client access functions that could alter the intention of your program. This is commonly known as **security**.

In Racket, when we implement a function it is automatically hidden from the client. To allow the client to have access to a given function in our module, we must *provide* the function. Consider the following.

EXAMPLE 3 Implement a module that will let a client guess your favorite color.

```racket
#lang racket ; fav-color.rkt

; Providing the appropriate functions to the client.
(provide fav-color?)

; (fav-color? color): Symbol -> Boolean
; Purpose: Consumes a symbol, color and produces #t if the color
;          guessed is my favorite color, otherwise #f.
; PRE: true
; POST: returns a boolean

; ==================================================================

; Declaring my favorite color (private).
(define favorite-color 'blue)

; (fav-color?/helper): Symbol -> Boolean (private)
; Purpose: Consumes a symbol, color and produces #t if the color
;          guessed is my favorite color, otherwise #f.
; PRE: true
; POST: returns a boolean
(define (fav-color?/helper color)
  (symbol=? favorite-color color))

; See interface above (public).
(define (fav-color? color)
  (fav-color?/helper color))
```

Observe that the documentation for the functions that we want the client to use are placed at the top of the file followed by some delimiter. When documenting interfaces in Racket all of the design recipes for functions that we want to *provide* to the client should be at the top. Documentation for other functions, such as helper functions, should be placed respectively with the function definition. You can document multiple functions in the interface by separating their design recipe with a new line.

2.2 Using Racket Modules

To use a Racket module you must use the *require* special form. When it reaches a line with *require*, it will stop executing your code and begin executing the code for the module you are including. Once this is done it will return to your code and continue executing where it left off. Consider the following client module for our favorite color module.

```
#lang Racket ; client.rkt

; Requiring the favorite color module.
(require "fav-color.rkt")

; Guessing my favorite color.
(fav-color? 'green) ; => #f
(fav-color? 'red) ; => #f
(fav-color? 'blue) ; => #t
(fav-color? 'orange) ; => #false

; ERRORs
(fav-color?/helper 'blue) ; => Error!
```

Testing

In full Racket there is no such function called $check - expect$. The way we will create a testing module is by checking if the function with desired inputs is equal to a desired output, similar to $check - expect$.

EXAMPLE 1 Design a testing module for my favorite color module.

```
#lang Racket ; test-client.rkt

; Requiring the favorite color module.
(require "fav-color.rkt")

; Performs some testing.
(equal? (fav-color? 'red) #f) ; => Test passes.
(equal? (fav-color? 'blue) #t) ; => Test passes.
```

3 Functional C: Module 03

To be continued...

4 C Memory Model: Module 04

To be continued...

5 Imperative C: Module 05

To be continued...

6 Efficiency: Module 06

This section goes briefly into efficiency. Unfortunately in CS 136 we are only introduced to some of the basics and questions such as ”why” may be left unanswered.

Consider efficiency. What is the first word that comes to mind?

DEFINITION
Algorithm

Is a step-by-step description on how to solve a problem.

Algorithms are not restricted to computing. You have devised hundreds of subconcious algorithms that you perform daily. Consider the following.

EXAMPLE 1

What is my algorithm for eating cake?

Solution: My algorithm can be written in the following steps.

1. Grab a slice of cake.
2. Grab a fork and knife.
3. Eat it as fast as possible.
4. As I approach licking my plate, get in line for seconds.

EXERCISE

What is the algorithm you use for brushing your teeth?

Consider the following problem.

EXAMPLE 2

Write a Racket function *nodes* >? that will count all of the non-empty nodes in a valid BST and determine whether there are more than k nodes.

Solution: Consider the following algorithms that solve this problem.

1. Calculate the total number of nodes and compare that number to k.
2. Recurse through the BST. At each node recurse to $k - 1$ until it reaches empty or k becomes negative.

Both algorithms solve the same problem however how do we determine which one is better suited to the task? Define better? How do we **compare** algorithms?

The implementation of both algorithms is as follows.

```
; Algorithm 1
(define (m1/nodes>? abst k)
  (define (height-bst abst)
    (cond
      [(empty? abst) 0]
      [else (+ 1 (+ (height-bst (bst-node-left abst))
                    (height-bst (bst-node-right abst))))]))
  (> (height-bst abst) k))

; Algorithm 2
(define (m2/node>? abst k)
  (cond
    [(empty? abst) #f]
    [(< k (bst-node-key abst)) #t]
    [else (or (m2/node>? (bst-node-left abst) (- k (bst-node-key abst)))
              (m2/node>? (bst-node-right abst) (- k (bst-node-key abst))))]))
```

EXERCISE Think of alternative algorithms and implement them. Then follow the same analysis we take with respect to the provided algorithms.

6.1 Efficiency

The two most common measures of efficiency are **time efficiency** and **space efficiency**.

DEFINITION
Time

The worst case scenario for how much time it takes for an algorithm to solve a given problem.

DEFINITION
Space

The worse case scenario for how much space is required for an algorithm to solve a given problem.

The efficiency of an algorithm may depend on its implementation. All algorithms are measured on their worst case, this will be properly addressed shortly. Unless otherwise specified, the professors will always be referring to time efficiency.

It is important to quantify efficiency. It may seem trivial, use seconds, however you are wrong. Seconds are not useful to a programmer because there are so many factors that could impact seconds such as, was this 1980 or 2040? Was it on a quantum computer or an iMac? What was the operating system and chip manufacturer? ... with respect to Racket we will quantify efficiency as the amount of substitution steps it takes to solve a given problem.

Revisiting our second algorithm in example 2.

```
(define bst-1 (bst-node 5 "" (bst-node 3 "" empty empty)
                             (bst-node 7 "" empty empty)))

(m2/node>? bst-1 10) ; => 47 steps
```

In C one measure may be how many machine instructions are executed. The problem is that the machine instruction count vary from machine to machine and would be an unreliable source of information. To quantify efficiency in C we will count the number of operations executed.

```
sum = 0;                  // 1
i = 0;                    // 1

while ( i < 5 ) {         // 6
  sum = sum + i;          // 2 * 5 = 10
  i = i + 1;              // 2 * 5 = 10
}
```

Note that the expression in the loop is executed 6 times. The 6th execution of the expression is to verify that i is now equal to 5.

6.2 Input Size

Recall our second algorithm in example 2. Did you notice that the number of steps depended on the input size? If there are n nodes in the BST, it will require $14n + 2$ steps to solve the problem. From now on we are always interested in measuring the running time based on the size of the input.

DEFINITION
Running Time

The number of steps or operations with respect to the input, n, that an algorithm requires to solve a problem.

We will denote the running time of a function, T as $T(n)$ where n is the input size. Keep in mind there may also be another **attribute** of the input that is important in addition to size.

Analysis

If you recall we mentioned that time and space effieciency are measured with respect to their worst case. Consider the following input to both algorithms.

```
(define bst-1 (bst-node 10 "" (bst-node 8 "" empty empty) empty))

(m1/node>? bst-1 6) ; T(9n + 4) => 22 steps
(m2/node>? bst-1 6) ; T(15n + 5) => 5 steps
```

The **best case** is when only the first node of the BST is visited and the **worst case** is when all of the nodes are smaller than k and hence all the nodes are visited. How should we decide which one is more efficient?

DEFINITION
Worst Case Analysis

Typically, we want to be conservative (pessimistic) and use the worst case. This is the process of determing the efficiency of an algorithm by comparing worst case scenarios.

Comparing the worst case, we see that $T(9n + 4)$ is more efficient than $T(15n + 5)$. It may also be important to know the **average running time** however we will not be touching this is not touched upon in CS 136 as it requires further analysis of the algorithm with lot's of data.

6.3 Ordered Runtime: Big O

In practice we are not concerned about the difference between the run times of $T(9n+4)$ and $T(15n+5)$. We are interested in the **order** of a running time.

DEFINITION
Dominant Term

The term that grows the largest as n approaches infinity.

DEFINITION
Order

The dominant term in the running time without any constant coefficients. It is also known as the growth rate.

The dominant term in both $T(9n+4)$ and $T(15n+5)$ is n and hence it is of order n, denoted as $O(n)$ in **Big O notation**. The orders that we will need to know:

$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3)$ and $O(2^n)$.

EXAMPLE 1

Consider the following orders.

- 1994 is $O(1)$.
- $1994 + n$ is $O(n)$.
- $10 + n^2 + n \log n + 1994$ is $O(n^2)$.
- $9 + 2^n + n^3$ is $O(2^n)$.

Pay attention to the fact that the dominant term is the order.

To tie order into efficiency, the algorithm with the lowest order is the most efficient. If we were to compare two different implementations $O(n)$ and $O(1)$, $O(1)$ would be the more efficient implementation.

EXERCISE

Compare the graphs of all of the orders we are responsible for knowing.

Big O Arithmetic

When adding two orders, the larger of the two orders will be the result. Consider the following.

EXAMPLE 2

What is the sum of $O(n)$ and $O(n^3)$?

Solution: Notice that the sum is the $max(O(n), O(n^3))$ which equals $O(n^3)$.

When multiplying two orders, the result is the distribution of both orders. Consider the following.

EXAMPLE 3

What is the product of $O(n)$ and $O(n^2)$?

Solution: $O(n) \times O(n^2)$ equals $O(n^3)$.

EXERCISE

Go through all of your assignments and determine the order of each function.