Algorithmic Complexity

Christopher Godley

CSCI 2270 Data Structures

September 25th, 2019

Algorithms

- In any computer program, there is a specific set of instructions that tells the computer what to do.

- This set of instructions is similar to a recipe in that there is an objective to accomplish (problem to solve), and a set of steps in a specified order to accomplish the objective.

- These instructions are also known as an algorithm: a defined set of steps that are followed to solve a problem.

Algorithms

As an example, an algorithm that puts the following sequence of numbers in ascending order

<54, 34, 23, 45, 56, 90>

would produce an output of

<23, 34, 45, 54, 56, 90>.

Algorithms

- **Pre-Condition**
  - conditions that must be true prior to the algorithm's execution in order for it to work as defined.
  - Pre-conditions can include the inputs to the algorithm and the restrictions on the types and range of values on those inputs.
  - Pre-conditions can also include other dependencies, such as other algorithms that need to execute first.
- **Post-Condition**
  - The expected changes, or the return value, after the algorithm executes.
  - For example, a function to calculate the factorial of a particular number could look like:

Evaluation

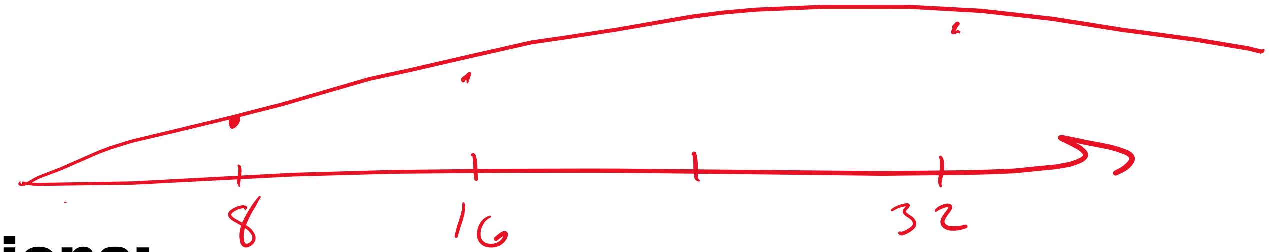
- **Correctness**
  - The algorithm returns the desired result or performs the desired action appropriately
- **Cost**
  - Various ways to measure, application specific
  - Two standard metrics:
    - ***Runtime (follow n)***
    - ***Memory usage***

Evaluation: Algorithm Analysis

Common Functions:

- **Constant Function**
  - $f(n) = c$
  - This function has a constant runtime, such that the output is not dependent on the value or size of the input *n*
  - Ex:
    - ***Variable assignment***
    - ***Inserting to the front of a linked list***
    - ***Overwriting element in an array***
    - ***Accessing element in hash table***

Evaluation: Algorithm Analysis



Common Functions:

- **Logarithmic Function**
  - f(n) = log(n)
  - The logarithmic runtime will frequently be base 2 (thanks binary!)
  - log(n) vs lg(n) vs log_b(n)
  - Ex:
    - ***Minimum height of BST***
    - ***Searching in a BST***

Evaluation: Algorithm Analysis

Common Functions:

- **Linear Function:**
  - $f(n) = n$
  - The value and size of *n* directly correlates to the runtime
  - What about $f(n) = 2n$? Or $f(n) = 1{,}000{,}000n$?
    - ***Constants do not detract from linear runtime***
  - Ex:
    - ***Traversing elements in linked list***
    - ***Traversing elements of a 1D array***
    - ***Shifting elements in a 1D array***

Evaluation: Algorithm Analysis

Common Functions:

- **N-Log-N Function:**
  - $f(n) = n\log(n)$
  - Log(n) repeated n times
  - Ex:
    - ***N searches on BST***
    - ***Merge sort***

Evaluation: Algorithm Analysis

Common Functions:

- **Quadratic Function:**
  - $f(n) = n^2$
  - Ex:
    - ***Traversing a 2D matrix with n rows and n columns***
    - ***Algorithms with nested for loops***
    - ***Bubble Sort***

Evaluation: Algorithm Analysis

Common Functions:

- **Polynomial Function:**
  - f(n) = n^k
- **Exponential Function:**
  - f(n) = b^n
  - b is some constant we call the base, most commonly 2 (thanks binary!)
  - f(n) = 2^n

Visual Comparison

Big O
worst case

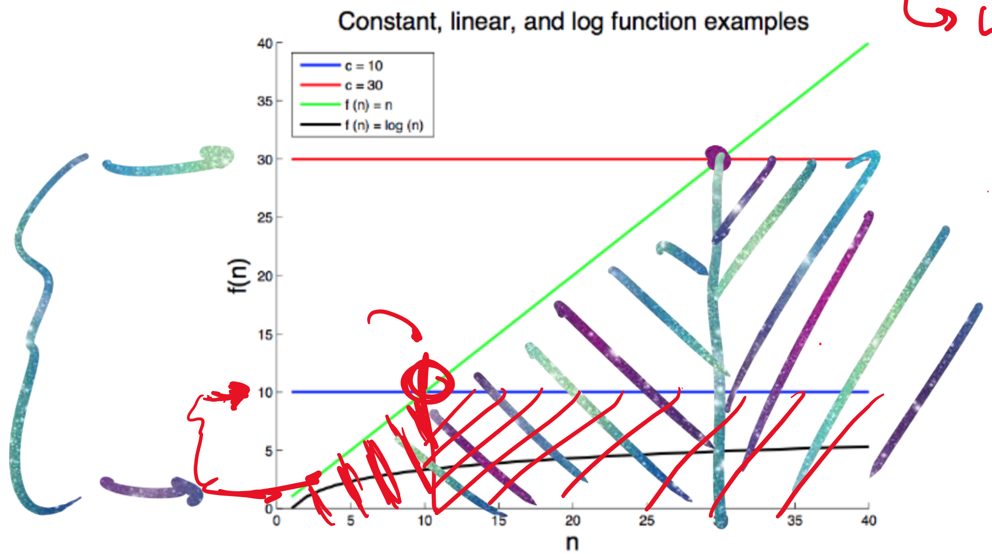


Figure 1. Comparison of growth rates for a constant, linear, and log(n) function for input size n. The linear function grows faster than the log function.

Visual Comparison

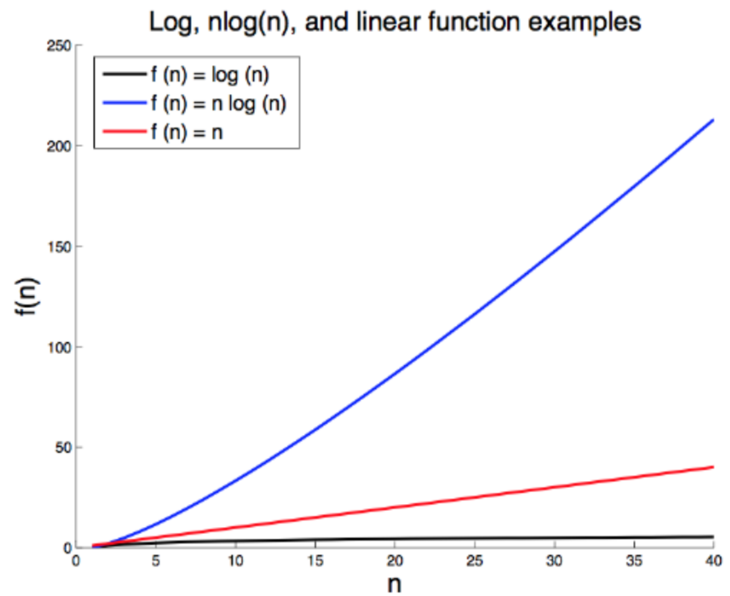


Figure 2. Comparison of growth rates for log(n), nlog(n), and linear functions for a given input size n. The nlog(n) function grows the fastest of the three functions.

Visual Comparison

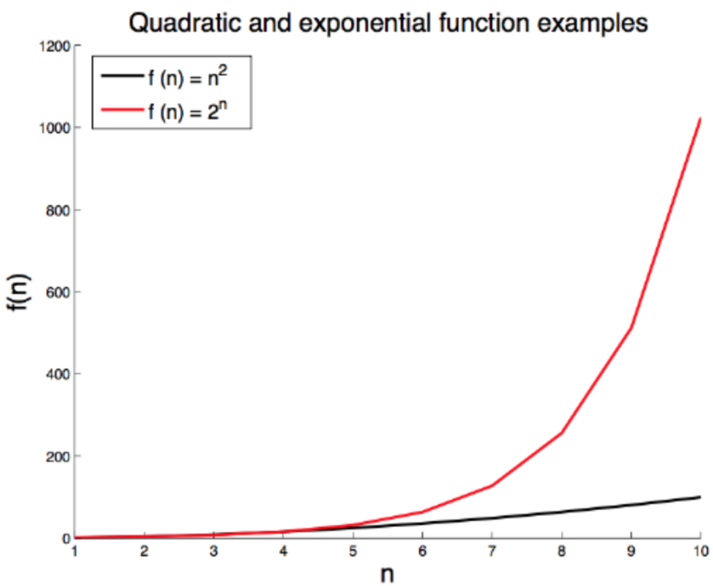


Figure 4. Comparison of growth rates for quadratic and exponential functions for a given input size n. Exponential growth rate typically equals very bad performance for large n.

Code Analysis

- **When looking at code, look for loops!**
  - ex) *`for i to n:`* // this is an ‘n’ time loop
- **Sequential loops? (Add the runtime!)**
  - ex) *`for i to n:`* // this is an ‘n’ time loop
  - *`for i to n:`* // also an ‘n’ time loop, after the first loop
  - **Runtime? 2n**
- **Nested loops? (Multiply the runtime!)**
  - ex) *`for i to n:`* // this is an ‘n’ time loop
  - *`for j to n:`* // also an ‘n’ time loop, inside the ‘i’ loop
  - **Runtime? n^2**