Algorithm Analysis and Midterm Details

Dr. Isaac Griffith Idaho State University

Inspiration

Idaho State University | Computer Science

"The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct." – Donald Knuth

Outline

The lecture if structured as follows:

- Big-O Notation
- Complexity of Algorithms
- Midterm Exam Details

Big-O Notation

CS 1187

Big-O Notation

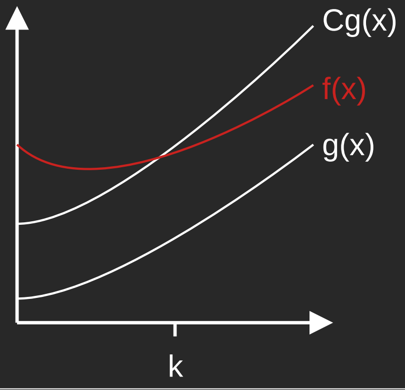
- **Big-O Notation:** provides the ability to estimate the growth of a function without worrying about constant multipliers or smaller order terms
  - Simplifies the analysis of an algorithm

- **Definition:** Let *f* and *g* be functions from $\mathbb{R}$ or $\mathbb{Z}$ to the set $\mathbb{R}$, we say that $f(x)$ is $O(g(x))$ if there are constants *C* and *k* such that:

$$|f(x)| \leq C|g(x)| \text{ whenever } x > k$$

  - That is $f(x)$ grows slower than some fixed multiple of $g(x)$ as x grows without bound

- *C* and *k* are called **witnesses** to the relationship $f(x)$ is $O(g(x))$
  - we only need one pair of witnesses to show this.



Working with Big-O

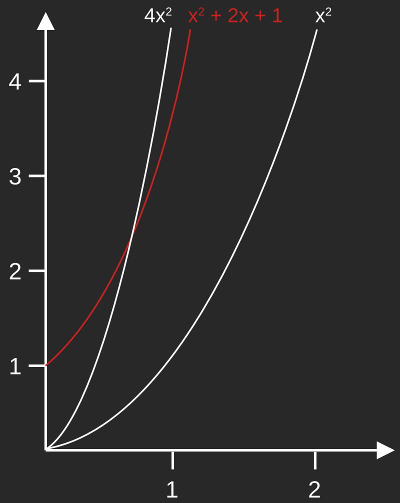
- Finding a pair of witnesses
  1. Find a k for which the size of $|f(x)|$ can be readily estimated when $x > k$
  2. Use this to find a value for C for which $|f(x)| \leq C|g(x)|$ for $x > k$

- *Example:* Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$

  - estimate size of $f(x)$ when $x > 1$
  - because $x < x^2$ and $1 < x^2$ when $x > 1$
  - then $0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$ when $x > 1$
  - **witnesses:** $k = 1,\ C = 4$

  - we could also use $x > 2$
  - for which $2x \leq x^2$ and $1 \leq x^2$, if $x > 2$
  - we then have: $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + 3x^2$
  - **witnesses:** $k = 2, C = 3$

Working with Big-O



In the example, we had two functions

$$\begin{aligned} f(x) &= x^2 + 2x + 1 \\ g(x) &= x^2 \end{aligned}$$

We showed that $f(x)$ is $O(g(x))$, but we could also prove that $g(x)$ is $O(f(x))$ because both functions are of the **same order**

Working with Big-O

- If $f(x)$ is $O(g(x))$, and $h(x)$ is a function with sufficiently larger value for x than $g(x)$ it follows that $f(x)$ is $O(h(x))$ as well.

- We can replace $g(x)$ with $h(x)$ in $f(x)$ is $O(g(x))$ iff
  - $|f(x)| \leq C|g(x)|$ if $x > k$, and
  - $|h(x)| > |g(x)|$ for all $x > k$, then
  - $|f(x)| \leq C|h(x)|$ if $x > k$

- i.e., if $f(x)$ is $O(x^2)$ it is also $O(x^3)$, $O(x^4)$, $O(x^5)$, $\ldots$

- However, we typically want to find the smallest (or tightest) growth rate functions for use with Big-O

Example

- Show $f(n) = 5n^3 + 2n^2 + 22n + 6$ is $O(n^3)$

- **Proof:**
  Let $C = 6$, we want to find the smallest n such that

$$
\begin{aligned}
6n^3 &> 5n^3 + 2n^2 + 22n + 6 \\
n^3 &> (2n^2 + 22n + 6)
\end{aligned}
$$

$$
\begin{array}{ll}
n = 1 & 1 < 30 \\
n = 2 & 8 < 126 \\
\vdots & \\
n = 5 & 125 < 126 \\
n = 6 & 216 > 210 \\
n = 7 & 343 > 258 \\
\vdots &
\end{array}
$$

Witnesses: $C = 6,\ k = 6$

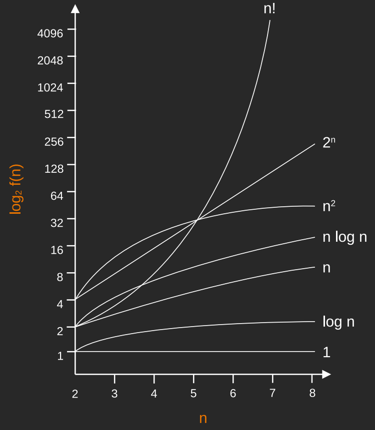
Therefore, $f(n)$ is $O(n^3)$

Big-O Estimates

- Polynomials often can be used to estimate the growth of functions
  - Rather than analyzing the growth of polynomials each time they occur we want a generalizable result
- The following theorem does just that
- **Theorem:** Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n \in \mathbb{R}$. Then $f(x)$ is $O(x^n)$
  - The leading term of a polynomial dominates its growth, thus a polynomial of degree n is $O(x^n)$
- *Example:* $1 + 2 + \ldots + n$
  - $1 + 2 + \ldots + n \leq n + n + n + \ldots + n = n^2$
  - $\therefore\ 1 + 2 + \ldots + n = O(n^2),\ C = 1,\ k = 1$
- *Example:* $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$
  - $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \leq n \cdot n \cdot n \cdot \ldots \cdot n = n^n$
  - $\therefore\ n! = O(n^n),\ C = 1,\ k = 1$
- Also note:
  - $\log n < n$, $\log n$ is $O(n)$
  - $\log n! < \log n^n = n \log n$
    - $\log n!$ is $O(n \log n)$, $C = 1$, $k = 1$

Big-O Estimates

- Some important Big-O properties
  - If $d > c > 1$, then n^c is $O(n^d)$, but n^d is not $O(n^c)$
  - Whenever $b > 1$ and c and d are positive $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O(\log_b n)^c$
  - Whenever d is positive and $b > 1$: n^d is $O(b^n)$, but b^n is not $O(n^d)$
  - When $c > b > 1$, then b^n is $O(c^n)$, but c^n is not $O(b^n)$
  - If $C > 1$, then c^n is $O(n!)$, but $n!$ is not $O(c^n)$



Growth of functions commonly used in Big-O estimates.

Function Combinations

- Often algorithms are made up of two or more separate procedures
  - Thus, the number of steps needed for computation is the sum of the steps from all the procedures
  - A Big-O estimate is then the Big-O estimate for the combination
    - This requires we take care during the combination.

- **Theorem:** Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then $(f_1 + f_2)(x)$ is $O(g(x))$, where $g(x) = (\max(|g_1(x), |g_2(x)|))$ for all x.
  - **Corollary:** Suppose that $f_1(x)$ and $f_2(x)$ are both $O(g(x))$. Then, $(f_1 + f_2)(x)$ is $O(g(x))$

- **Theorem:** Suppose that $f_1(x)$ is $O(g_1(x))$ and that $f_2(x)$ is $O(g_2(x))$. Then, $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$

Function Combinations

- *Example:* Give a Big-O estimate for $f(x) = (x+1)\log(x^2+1) + 3x^2$

$$\begin{array}{rclcl} f(x) & = & (x+1)\log(x^2+1) & + & 3x^2 \\ & & O(x\log x^2) & & O(x^2) \\ & & O(x^2) & & \end{array}$$

- *Example:* Give a Big-O estimate for $f(n) = 3n\log(n!) + (n^3+3)\log n$

$$\begin{array}{rclcl} f(n) & = & 3n\log(n!) & + & (n^3+3)\log n \\ & & O(nlogn) & & O(n^3\log n) \\ & & O(n^3logn) & & \end{array}$$

Big-Ω and Big-Θ Notation

Idaho State University | Computer Science

- Big-O is useful, however it only provides an *upper bound* and does not provide any insight about the *lower bound* of a function
  - For lower bounds we use **Big-Ω notation**
  - For an exact (upper and lower bound) we use **Big-Θ notation**

- Ω: Let f and g be functions from $\mathbb{R}$ or $\mathbb{Z}$ to $\mathbb{R}$. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k with C positive such that:

$$|f(x)| \geq C|g(x)| \text{ whenever } x > k$$

  - **Note:** $f(x)$ is $\Omega(g(x))$ iff $g(x)$ is $O(f(x))$

Big-Ω and Big-Θ Notation

Idaho State University | Computer Science

- Θ: Let f and g be functions from $\mathbb{R}$ or $\mathbb{Z}$ to $\mathbb{R}$. We say that $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$. That is $f(x)$ is $\Theta(g(x))$ iff there are positive real numbers C_1 and C_2 and a positive real number k, such that:

$$C_1|g(x)| \leq f(x) \leq C_2|g(x)| \text{ whenever } x > k$$

  - **Note:** We also say that if $f(x) is \Theta(g(x))$ then $f(x)$ is *order* $g(x)$

- *Example:* Let $f(n) = 1 + 2 + 3 + \ldots + n$. Since we know $f(n)$ is $O(n^2)$, to show that $f(n)$ is order n^2, we need a positive constant C such that $f(n) > Cn^2$
  - To obtain the lower bound, we can ignore the first half of the terms, summing only terms greater than $\lceil n/2 \rceil$

$$\begin{aligned} 1 + 2 + \ldots + n &\geq \lceil \tfrac{n}{2} \rceil + (\lceil \tfrac{n}{2} \rceil + 1) + \ldots + n \\ &\geq \lceil \tfrac{n}{2} \rceil + \lceil \tfrac{n}{2} \rceil + \ldots + \lceil \tfrac{n}{2} \rceil \\ &= (n - \lceil \tfrac{n}{2} \rceil + 1) \lceil \tfrac{n}{2} \rceil \\ &\geq (\tfrac{n}{2})(\tfrac{n}{2}) \\ &= \frac{n^2}{4} \end{aligned}$$

- Thus $f(n)$ is $\Omega(n^2)$.
- Because $f(n)$ is $\Omega(n^2)$ and is $O(n^2)$, then it is order n^2 or $\Theta(n^2)$

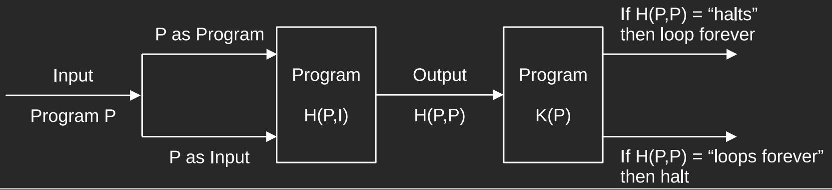
The Halting Problem

- In computing there are some problems which are impossible to solve, one of the most famous is the Halting Problem.

- **Halting Problem:** Is there a procedure that takes as input a program and input to the program and determines whether the procedure will eventually stop when run with this input.
- Alan Turing, showed that this problem is unsolvable by using a proof by contradiction:
  - Assume there is a solution, a procedure called $H(P, I)$ which takes
    - a program P and its input I, as input
    - H produces the string “Halt” as output if P halts on input I
    - H produces the string “Loops forever” otherwise
  - Now a procedure can be represented as a string, which can be interpreted as a sequence of bits. Thus the program itself may be used as data.
    - H can take P as both of its inputs
    - H should then be able to determine if P will halt given itself as input

The Halting Problem

- To show that H cannot exist, we create a simple procedure $K(P)$
  - Takes the output of $H(P, P)$ as input
  - Does the opposite of what the output of $H(P, P)$ specifies
- However, if we provide K as the input to K
  - Note: if the output of $H(K, K)$ is “Loop forever”, then K Halts
  - Thus, the output of $H(K, K)$ would be “Halt”, *A Contradiction*
  - If the output of $H(K, K)$ is “Halts”, then K would loop forever, *A Contradiction*
- This means H cannot always give the correct answer, hence no procedure solves the Halting problem



Complexity of Algorithms

CS 1187

ROAR

Complexity of Algorithms

- **Computational Complexity:** a measure of how costly it is to evaluate a given function
  - Typically measured in either the computational time required to solve the problem, *Time Complexity*, or
  - In the amount of computer memory required to implement the algorithm, *Space Complexity*

- In this course we will limit our discussion to *time complexity* and leave the discussion of *space complexity* to Computational Theory.

Time Complexity

- Can be expressed in terms of the number of operations used by an algorithm when the input is of a particular size
  - This provides a general unit of measure, which is agnostic of the particular hardware upon which the implementation will run

Example: What is the time complexity of the *max* algorithm?

Algorithm:

1: **procedure** MAX(A)
2: $\quad max := A_1$
3: $\quad$ **for** $i := 2$ **to** n **do**
4: $\quad\quad$ **if** $max < A_i$ **then** $max := A_i$
5: $\quad$ **return** max

Evaluation:

2.) 1 operation

3-4.) 2 comparisons for $n - 1$ iterations + 1 to exit $\rightarrow 2(n-1) + 1 = 2n - 1$ operations

5.) 1 operation

Total: $2n + 1$ which is $\Theta(n)$ time complexity

Time Complexity

Example: What is the time complexity of linear search

Algorithm:

1: **procedure** LINEARSEARCH(A, x)
2: $\quad i := 1$
3: $\quad$ **while** $i \leq n$ **and** $x \neq A_i$ **do**
4: $\quad\quad i := i + 1$
5: $\quad$ **if** $i \leq n$ **then** $location := i$
6: $\quad$ **else** $location := 0$
7: $\quad$ **return** $location$

Evaluation

2.) 1 operation

3-4.) 2 comparisons + 1 assignment for each iteration

5-6.) 2 operations

7.) 1 operation

Total: $1 + 2(n + 1) + 2 + 1 = 2n + 6 \rightarrow \Theta(n)$ in the worst case

ROAR

Worst-Case Complexity

- **Worst-Case Analysis:** Evaluating an algorithm for the largest number of operations that would be required to solve a given problem using the algorithm on an input of a specified size (typically *n* where *n* is some very large number).

- This type of analysis tells us how many operations an algorithm requires to guarantee that it will produce a solution.

Worst-Case Complexity

Example: What is the worst case complexity of binary search?

Algorithm:

procedure BINSEARCH(A, x)
$i := 1$
$j := n$
while $i < j$ **do**
$m := \lfloor (i + j)/2 \rfloor$
if $x > A_m$ **then** $i := m + 1$
else $j := m$
if $x = A_j$ **then** $location := i$
else $location := 0$
return $location$

Evaluation:

2.) 1 operation
3.) 1 operation
4-7.) At most $2 \log n + 2$ comparisons
8-9.) 1 comparison + 1 assignment
10.) 1 operation
Total:
$1+1+(2 \log n+2)+2+1 = 2 \log n+7 = \Theta(\log n)$
in the worst case

ROAR

Average-Case Complexity

Idaho State University | Computer Science

- **Average Case Analysis:** Analysis to find the average number of operations used to solve the problem over all possible inputs of a given size. Typically much more complicated than worst-case analysis

Example: Linear Search in terms of average number of comparisons used, x is in the list, and it is equally likely that x is in any position.

Algorithm:

```
procedure LINEARSEARCH(A, x)
    i := 1
    while i ≤ n and x ≠ A_i do
        i := i + 1
    if i ≤ n then location := i
    else location := 0
    return location
```

Evaluation:

- if x is in position $1 \to 3$ comparisons
- if x is in position $2 \to 5$ comparisons
- if x is in position $i \to (2i+1)$ comparisons

$$\begin{aligned}
\text{Avg Comparisons} &= \frac{3+5+7+\ldots+(2n+1)}{n} \\
&= \frac{2(1+2+3+\ldots+n)+n}{n} \\
&= \frac{2\left(\frac{n(n+1)}{2}\right)}{n} \\
&= n+2 = \Theta(n)
\end{aligned}$$

Analyzing Insertion Sort

Example: Worst-case complexity of insertion sort in terms of comparisons made:

procedure SORT(A)
  for $j := 2$ **to** n **do**
    $i := 1$
    while $A_j > A_i$ **do**
      $i := i + 1$
    $m := A_j$
    for $k := 0$ **to** $j - i - 1$ **do**
      $A_{j-k} := A_{j-k-1}$
    $A_i := m$

- j comparisons are required to insert the j^{th} element into the correct position
- Thus, the total number of comparisons needed to sort a list of n elements is $2 + 3 + \ldots + n = \frac{n(n+1)}{2} - 1$
- Thus, worst-case complexity is $\Theta(n^2)$

Analyzing Matrix Multiplication


```
procedure MatrixMult(A, B)
    for i := 1 to m do
        for j := 1 to n do
            C_ij := 0
            for q := 1 to k do
                C_ij := A_iq · B_qj
    return C
```

- Since there are n^2 entries in the product of **A** and **B**. To find each entry requires a total of n multiplications and $n-1$ additions
- Thus, a total of n^3 multiplications and $n^2(n-1)$ additions are needed.
- Therefore, $O(n^3)$
- **Note:** two $n \times n$ matrices can be multiplied in $O(n^{\sqrt{7}})$ multiplications and additions

Algorithmic Paradigms

- **Algorithmic Paradigm (or Algorithmic Design Strategy):** is a general approach based on a particular concept that can be used to construct algorithms for solving a variety of problems:
  - Serve as the basis for constructing algorithms for solving a range of problems.
- Well know algorithmic paradigms include:
  - **Divide-and-Conquer**
  - **Dynamic Programming**
  - **Backtracking**
  - **Greedy Algorithms**
  - **Brute-Force Algorithms**
  - Transform-and-Conquer
  - Branch-and-Bound
  - Probabilistic Algorithms
  - Randomized Algorithms
  - Linear Programming
- There are many other paradigms beyond what is listed.

Brute-Force Algorithms

- **Brute-Force Algorithm:** An algorithm which solves a problem in the most straight-forward manner based on the problem statement and the definition of terms.
  - Typically designed without regard to computing resources required

- These are typically naive approaches which
  - Do not take advantage of special structures in the problem
  - Do not utilize clever ideas

- Though useful, they are often inefficient, however
  - Can serve as a baseline for comparison to more efficient algorithms

Brute-Force Algorithms

Example: Finding closed pair of points

procedure CLOSESTPAIRS($(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$: pairs of real numbers)
  $min := \infty$
  for $i := 2$ **to** n **do**
    for $j := 1$ **to** $i - 1$ **do**
      if $(x_j - x_i)^2 + (y_j - y_i)^2 < min$ **then**
        $min := (x_j - x_i)^2 + (y_j - y_i)^2$
        $closestPair := ((x_i, y_i), (x_j, y_j))$
  return $closestPair$

- In terms of additions and comparisons this algorithm is $\Theta(n^2)$

Understanding Algorithmic Complexity

Idaho State University | Computer Science

- Commonly used terminolgoy for the complexity of algorithms:
  - *Constant Complexity:* $\Theta(1)$
  - *Logarithmic Complexity:* $\Theta(\log n)$
  - *Linear Complexity:* $\Theta(n)$
  - *Linearithmic Complexity:* $\Theta(n \log n)$
  - *Polynomial Complexity:* $\Theta(n^b)$
  - *Exponential Complexity:* $\Theta(b^n)$, where $b > 1$
  - *Factorial Complexity:* $\Theta(n!)$

Tractability

Idaho State University | Computer Science

- **Tractable:** a problem that is solvable using an algorithm with polynomial (or better) worst-case complexity
  - such an algorithm will produce a solution to the problem a reasonably sized input in a relatively short time
- **Intractable:** a problem that cannot be solved using an algorithm with worst-case polynomial time complexity
  - usually an extremely large amount of time is required to solve such problems, even on small inputs
  - however, many important problems from industry thought to be intractable, can be practically solved for all real-world data sets.
- **Unsolvable:** Some problems, i.e. the halting problem, exists for which it can be show no algorithm exists for solving them.

P vs. NP

- **Class P:** the class of problems which are tractable
- **Class NP:** the class of problems that have the following property
  No algorithm with polynomial worst-case complexity can solve them, but a solution, if known can be checked in polynomial time
  - *Note:* **NP** stands for *nondeterministic polynomial time*
- **NP-Complete Problems:** Problems with the property that if any of these problems are solved by a polynomial worst-case time algorithm, then all problems in the class NP can be solved by a polynomial worst-case time algorithm.
  - *Note:* all problems in the class NP are reducible to those problems in the class NP-Complete
- **P vs. NP Problem:** asks whether, the class NP = P or not. Currently, there is no solution to this problem, and it is assumed that NP $\neq$ P.

Practical Considerations

- *Note:* Time complexity (i.e., $\Omega()$) expresses how the time to solve a problem increases as the input increases in size, it cannot be directly translated into actual computational time.

- Even worse, we often only have a big-O upper bound on the worst-case, but not a lower bound

- All of this aside it is often important to have an estimate of the approximate time an algorithm will take to complete

Practical Considerations

Problem Size	Bit Operations Used					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

- **Note:**
  - A "*" indicates times of $> 10^{100}$*years*
- As technology has increased processor speed and memory have increased
  - Additionally, we can decrease time needed to solve problems using *parallel processing*

Proving Recursive Algs Correct

Idaho State University | Computer Science

- Both Mathematical and Strong induction can be used to prove a recursive algorithm is correct

Example:

Algorithm:

procedure POWER(a, n)
  if $n = 0$ **then return** 1
  elsereturn $a \cdot$ POWER$(a, n-1)$

Proof:

Basis Step: if $n = 0$, $power(a, 0) = 1$, this is correct since $a^0 = 1$ for every nonzero real number a.

Inductive Step: inductive hypothesis: $power(a, k) = a^k$ for all $a \neq 0$ and an arbitrary k is correct.

Assuming the inductive hypothesis is correct, then by the inductive hypothesis

$$\begin{aligned} power(a, k+1) &= a \cdot power(a, k) \\ &= a \cdot a^k \\ &= a^{k+1} \end{aligned}$$

$\therefore$ we can conclude the algorithm is correct

Cost of Recursion

- Recursion can create expensive computations. A famous example is Ackerman's Function

```
ack 0 y = y + 1
ack x 0 = ack (x - 1) 1
ack x y = ack (x - 1) (ack x (y - 1))
```

  - This function works fine on small inputs but grows extremely quickly as *x* and *y* increase

- *Note:* Often an iterative implementation of a recursively defined function or sequence will require less computation

ROAR

State

- A function, such as in Haskell, always returns the same result, given the same arguments. This phenomenon is known as **side-effect free**

- However, some computations (such as those in imperative languages like Python and Java) do not have this property
  - i.e., a function which returns the current date.

- These functions require the use of and manipulation of *state*

- **State:** the entire set of circumstances that can affect the results of a computation

- In order to reason about these types of computations, or even to include them in languages like Haskell, we could introduce the *state* as an argument to the functions
  - However, for large programs or complicated functions, this would become overwhelming and cumbersome
  - This is why in imperative languages, they forgo the use of this *explicit state* for the easier to work with *implicit state* (hence variable assignments, etc. As for Haskell, we can work with state using *Monads* and `do` expressions.

Midterm Exam Details

CS 1187

ROAR

Midterm Exam

- **Exam will Open on Monday April 4th at 8:00 am and will close on Wednesday April 6th at 11:00 pm**
- Exam will be online on Moodle
- You will have 50 minutes to complete it
- It will range between 15 and 25 questions
  - Questions will be a combination of multiple choice, true/false, essay, matching, and short answer
- The exam is open book and open notes.
- **You may <u>NOT</u> consult the internet, other class members, or your friends**

Things to Study

- Logic - Lectures 4, 5, 6
  - Propositional Logic
  - Predicate Logic
  - Truth Tables and Reasoning with them
  - Laws of Propositional and Predicate Logic

Things to Study

- Equational Reasoning - Lectures 3, 5, 6, 7, 8
  - Boolean Algebra
  - Function Proofs
  - Recursive Proofs
  - Sets

Things to Study

- Set Theory - Lecture 7
  - Important Sets
  - Set Notation (especially Set Comprehensions)
  - Venn Diagrams
  - Cartesian Products
  - Set Laws
  - Membership Tables and Proofs Using them

Things to Study

- Recursion - Lecture 8
  - Ideas of Recursively Defined Data Structures (i.e., Trees and Lists)
  - Binary Trees
- Algorithms - Lecture 8
  - Properties of Algorithms
  - Concept of Greed Algorithms
  - Concept of Divide-and-Conquer

Things to Study

- Functions - Lecture 9
  - Domain and Codomain
  - Image and Range
  - Idea of Inductively defined Functions
  - One-to-One (Injective)
  - Onto (Surjective)
  - One-to-One and Onto (Bijective)
  - Inverse Functions

Things to Study

- Sequences and Summations - Lecture 9
  - Geometric Progression
  - Arithmetic Progression
  - Strings
  - Recurrence Relations
  - Fibonacci Sequence
  - Summation Notation
  - Useful Summation Formulae
  - Countability of Sets

Are there any questions?