CS 250 Final Review

mguiney

December 2018

Contents

1 Exam 1 Review **4**
- 1.1 Section 1.1: Proof Primer Logic . . . 4
- 1.2 Section 1.2: Sets and Set Operators . . . 4
- 1.3 Section 1.4: Graphs and Trees . . . 4
- 1.4 Section 2.1: Functions, Definitions, and Examples . . . 4
- 1.5 Section 2.2: Composition of Functions . . . 4
- 1.6 Section 2.3: Properties and Applications . . . 4

2 Section 2.4: Countability **5**
- 2.1 Cardinality Notation . . . 5
- 2.2 General Concepts . . . 5
- 2.3 Cardinality Proof Example . . . 5
- 2.4 Cantor Diagonalization Proof Method . . . 6
- 2.5 Cardinality of Integers Proof . . . 6
- 2.6 Some Countability Results . . . 6
- 2.7 Countability Fun Facts (wow so fun) . . . 6

3 Section 3.1: Inductively-defined Sets **8**
- 3.1 Inductive Definition Components . . . 8
  - 3.1.1 Inductively-defined Set Example #1 . . . 8
  - 3.1.2 Inductively-defined Set Example #2 . . . 8
  - 3.1.3 Example of Inductively Defined Sets of Strings . . . 9
  - 3.1.4 Example of Inductively Defined Sets of Lists . . . 9
- 3.2 Infix Notation for Lists and Cons . . . 9
  - 3.2.1 Infix notation Example #1: . . . 9
  - 3.2.2 Infix Notation Example #2 (A more confusing case) . . . 9
- 3.3 Notation for Binary Trees . . . 10
  - 3.3.1 Binary Tree Notation Example . . . 10

4 Section 3.2: Recursively Defined Functions and Procedures **11**
- 4.1 Recursive definitions and Procedures . . . 11
- 4.2 Recursively defining functions and procedures: . . . 11
  - 4.2.1 Recursive definition of functions example . . . 11
- 4.3 Strings and Languages Review, String Recursion . . . 12
  - 4.3.1 Recusively-defined Strings Example . . . 13
- 4.4 Lists Review, ists recursion . . . 13
- 4.5 Graph Traversals . . . 14
- 4.6 Traversing Binary Trees . . . 14
- 4.7 Infinite Sequences . . . 15

5 Section 5.1: Analyzing Algorithms **16**
- 5.1 Definitions and Properties of an Algorithm . . . 16
  - 5.1.1 Properties of algorithms: . . . 16
- 5.2 Analyzing Algorithms . . . 16

5.2.1 Algorithm Efficiency Metrics 16
5.3 Summation Notation Basics 17
5.4 Analysis 17

6 Section 5.6 Comparing Rates of Growth 18
6.1 Complexity 18
6.2 Comparing rates of growth 18
6.3 Big Oh 18
6.3.1 Big Oh Example #1 19
6.3.2 Big Oh example #2 19
6.3.3 What is Big-Oh used for? 20
6.4 Math Review Break! 20
6.5 Big Omega 20
6.5.1 What is Big Omega used for? 21
6.6 Big Theta 21
6.6.1 What is Big Theta Used for? 21

1 Exam 1 Review

Note: we won't have questions from this material, but we may need to use it for questions from later sections, so I'm circling back to it once finished with more relevent sections.

1.1 Section 1.1: Proof Primer Logic

Content coming soon

1.2 Section 1.2: Sets and Set Operators

Content coming soon

1.3 Section 1.4: Graphs and Trees

Content coming soon

1.4 Section 2.1: Functions, Definitions, and Examples

Content coming soon

1.5 Section 2.2: Composition of Functions

Content coming soon

1.6 Section 2.3: Properties and Applications

Content coming soon

2 Section 2.4: Countability

2.1 Cardinality Notation

Given a set "A", A's cardinality is denoted as |A |

The expression |A |= |B |indicates that there is bijection between A and B.

The expression |A |<= |B |indicates that there is injection from A to B

2.2 General Concepts

Sets "A" and "B" have the same cardinality IFF there is a one-to-one correspondance (i.e. a Bijection) from A to B.

IF a set is finite or has the same cardinality as **N**, then it is called **countable**, otherwise it is **uncountable**

In mathematics, the **cardinality** of a set is a measure of the number of elements of the set. The cardinality of the natural numbers **N** is denoted aleph-null

If the cardinality of some set "S" is equal to that of the set of natural numbers. then the set S is called **countably infinite**.

2.3 Cardinality Proof Example

Prove: that the set of positive rational numbers is countable.

$$E = \{n \mid n \bmod 2 = 0 \}$$

Let us define: f: **N** –> E as f:(n) = 2n.

- f maps N to E
- the defined function f is a one-to-one correspondence

Therefore: f is countable.

2.4 Cantor Diagonalization Proof Method

Prove: that the set of positive rational numbers is countable.

Q = { m/n — m, n ∈ N}

RETURNING TO THIS BIT LATER BECAUSE I HATE FORMATTING THIS PROOF

2.5 Cardinality of Integers Proof

The set of **Z** is countable, so it has the same cardinality as **N**.

Prove: there is a one-to-one correspondence from **N** to the Integers.

f(n) = n/2 if **N** is even and -(n-1)/2 if odd.

- f maps N to E
- the defined function f is a one-to-one correspondence

Therefore: the function f is countable, as f maps the set of integers.

2.6 Some Countability Results

1. Subsets and images of countable sets are also countable.
2. Set "S" is countable IFF |S |<= |**N** |

3. **N** x **N** is countable. This can be proved by using Cantor's Bijection to associate (x,y) with $((x+y)^2 + 3x + y)/2$

2.7 Countability Fun Facts (wow so fun)

- The set **R** of real numbers is not countable.
- **Cantor's Result:** |A |< |power(A) |for any set A.
  - Power(**N**) is uncountable because |**N** |< |power(**N**)|.
  - The power set of **N** has the same cardinality as **R**.
- Most of the other sets that we have used thus far are countable:
  - **N** is a subset of **R**, but **R** is not a subset at **N**

- **Q** is a subset of **R**, but **R** is not a subset at **Q**
- **N**, **Z**, and **Q** are all the same, which is to say aleph-null
- Every finite set "A" of elements from N is a subset of N, so |A |< |N |

• The cardinality of the set of real numbers (**R**) js not the same as that of **N**

3 Section 3.1: Inductively-defined Sets

3.1 Inductive Definition Components

An inductively-defined set "S" has three primary components:

1. **Basis:** Specify one of more elements of S (having more than one of these is fine).

2. **Induction:** Specify one or more rules to construct elements of S from existing elements of S.

3. **Closure:** Specify that no other elements are in S (this step is always implicit)

The basis elements and the induction rules are called **constructors**.

3.1.1 Inductively-defined Set Example #1

Problem: Find an inductive definition for S = {3, 16, 29, 42...}
Solution:

1. **Basis:** $3 \in S$

2. **Induction:** If $x \in 3, then x + 13 \in 3$. The constructors are 3, and the operation of adding 13.

3.1.2 Inductively-defined Set Example #2

Problem: Find an inductive definition for S = {3, 16, 29, 42, ...}
Solution: To simplify, we might try the method of D̈ivide and Conquer; by writing S as the union of more familiar sets, like this:

$$S = \{3, 4, 5, 8, 9, 12, 16, 17, 20, 24, 33, ...\} \cup \{4, 8, 12, 16, 20, 24, ...\}$$

1. **Basis:** 3, $4 \in S$

2. **Induction:**

```
If x $\in$ S then:
 if x is odd:
     2x-1 $\in$ S
 else:
     x + 4 $\in$
```

3.1.3 Example of Inductively Defined Sets of Strings

Problem: Find an Inductive definition for S = { λ, ac, aacc,aaaccc, ...} = / $a^n c^n | n \in \mathbf{N}$

Solution:

1. **Basis:** $\lambda \in$ S.
2. **Induction:** if x $\in$ S then axc $\in$ S.

NOTE: For strings, we start with a *middle* element, and inductive build outwards.

3.1.4 Example of Inductively Defined Sets of Lists

Problem: Describe the set S defined by:

1. **Basis:** $\langle 0 \rangle \in$ S
2. **Induction:** x $\in$ S implies cons(1, x) $\in$ S.

Solution: S = $\{\langle 0\rangle, \langle 1, 0\rangle, \langle 1, 1, 0\rangle, ...\}$

3.2 Infix Notation for Lists and Cons

Notation: cons(h,t) = h :: t. Associate to the right. This means that "x :: y :: z = x :: (y :: z)" is equivalent to cons(x, const(y, z)).

3.2.1 Infix notation Example #1:

Problem: Find an inductive definition for S = { $\langle\rangle, \langle a, b\rangle, \langle a, b, a, b\rangle, ...$}
Solution:
Basis: $\langle\rangle \in$ S.
Induction: x $\in$ S implies a :: b :: x $\in$ S (or cons(a, cons(b, x))), if you prefer.

3.2.2 Infix Notation Example #2 (A more confusing case)

Problem: Find and inductive definition for S = { $\langle\rangle, \langle\langle\rangle\rangle, \langle\langle\langle\rangle\rangle\rangle, ...$}
Solution:
Basis: $\langle\ \rangle \in$ S.
Induction: x $\in$ S implies x :: $\langle\rangle \in$ S.

3.3 Notation for Binary Trees

Let:

- t(L, x, R) be the tree with root x, left subtree L, and right subtree R
- $\langle\ \rangle$ denote the empty binary tree.

If T = t(L, x, R), then root(T) = x, left(T) = L, and right(T) = R.

3.3.1 Binary Tree Notation Example

Problem: Describe the set "S" defined inductively as follows:

1. **Basis:** t($\langle\ \rangle$, ·, $\langle\ \rangle$) ∈ S.
2. **Induction:** T ∈ S implies t(T, ·, t($\langle\ \rangle$, ·, $\langle\ \rangle$)) ∈ S.

4 Section 3.2: Recursively Defined Functions and Procedures

4.1 Recursive definitions and Procedures

In a **recursive definition**, an object is defined in terms of itself. Sequences, sets, and functions can all be defined recursively.

A function "f" is **recursively defined** if at least one value f(x) is defined in terms of another value f(y), where x != y. **Recursive procedures** are much the same, but define things in terms of P rather than f.

4.2 Recursively defining functions and procedures:

There are two steps to the technique used in the creation of recursive definitions:

1. Specify a value f(x) or action P(x), for each basis element x of S.

2. Specify rules that, for each inductively-defined element x in S, define f(x) or P(x) in terms of previously defined values of f or P.

4.2.1 Recursive definition of functions example

Problem: Find a recursive definition of for the function f:$\mathbf{N} \rightarrow \mathbf{N}$, as defined by:

$$f(n) = 0 + 3 + 6 + ... + 3n$$

Solution: **N** is an inducively defined set, so we need to give f(0) a value in **N**, and we need to define f(n+1) in terms of f(n). Let's iterate through a few of our terms and see what we find.

1. f(0) = 0

2. f(n+1) = (0 + 3 + 6 + ... + 3n) + 3(n+1)

3. f(n+1) = (0 + 3 + 6 + ... + 3n) + 3(n+1) + 3(n+2)

Using this, we can piece together the recursive definition for f:
f(0) = 0
f(n+1) = f(n) + 3(n+1)

4.3 Strings and Languages Review, String Recursion

- An **alphabet** "A" is a given finite set of symbols.
- A **string** "w' over alphabet A is a finite-length sequence of elements of A.
  - The concatenation of strings "x" and "y" is denoted "xy"
  - Concatenation of x n-many times is x^n
- The length of a given string "w" is |w |.
- The empty string is Λ, and $|\Lambda| = 0$.
- The set of all strings over A is $A*$ (includes Λ)
- a **language** "L" is a subset of $A*$, which is to say it is a set of strings.
- The **lexicographic ordering** of strings is the same as the dictionary ordering. In **short-lex**, the shorter strings precede the longer ones.

Given a set "A" and a string "a":

- A^n != a^n
  - A^2 = A × A
  - a^2 = aa
- ∅, Λ, and { Λ } are all different
- |∅ |= 0
- |∅∗ |= |Λ |= 1
- Concatenation rules:
  - (xy)z = x(yz)
  - xΛ = Λx = x
  - $a^n a^n = a^m + n$

4.3.1 Recusively-defined Strings Example

Problem: Find a recursive definition for cat: $A* \times A* \rightarrow A*$.
Solution:

1. $A*$ is inductively defined: $\Lambda \in A*$; a ∈ A and x ∈ $A*$ imply that ax ∈ $A*$, where ax denotes the string version of cons.

2. Define **cat** recursively using the first arg. The definition is:
   - cat(Λ, t) = Λt = t
   - recursive component: cat(ax, t) = axt = a(xt) = a(cat(x,t))

This gives us our final recursive string definitions:

- cat(Λ, t) = t
- cat(ax, t) = a(cat(x, t))

4.4 Lists Review, ists recursion

- Operations on lists:
  - head(⟨a, b, a, c ⟩) = a
  - tail(⟨a, b, a, c ⟩) = ⟨ b, a, c ⟩
  - cons(⟨a, ⟨ b, a, c ⟩ ⟩) = ⟨a, b, a, c⟩
- The set of lists whose elements are in A is denoted by lists(A)

4.5 Graph Traversals

• A **graph traversal** starts at some vertex "v" and visits all yet-unvisited vertices on the paths that start at v.

- vertices are not revisted
- Any traversal of a connected graph will visit all of its vertices

Graph-traversal Methods

- Breadth-first search
  - Search across levels.
  - Breadth-first algo
    1. Start at root
    2. Explore all neighboring nodes until all nodes have been visited.
- Depth-first Search
  - Search to each leaf (unvisited graph node) as fast as possible
  - Depth-first algo
    1. Start with some arbitrary node as your root.
    2. traverse graph until you hit all of the nodes with no children
    3. backtrack along the path from the original root node.

4.6 Traversing Binary Trees

There are 3 types of Binary tree traversal that are often used:

- Preorder Traversal
  - preorder(T): if T != ⟨ ⟩, then visit root(T); preorder(left(T)); preorder(right(t)); fi
  - process node data, traverse left, traverse right
  - List each vertex the first time it is encountered.
- Inorder Traversal
  - inorder(T): if T != ⟨ ⟩ then inorder(left(T)); visit root(T); inorder(right(t)); fi
  - traverse left, process node data, traverse right
  - List each leaf when initially encountered, and all other nodes the second time it is passed (i.e. when we process its data)

- Postorder Traversal
  - postorder(T): if t != $\langle\,\rangle$, then postorder(left(T)); postorder(right(T)); visit root(T); fi
  - traverse left, traverse right, process node data.
  - List each vertex the last time it is encountered.

4.7 Infinite Sequences

We can construct recursive definitions for infinite sequences by defining a value f(x) in terms of x and f(y) for some value y in the sequence.

NOTE: I'll circle back and add an example here if i have time, but it is pg 81 in the slides

5 Section 5.1: Analyzing Algorithms

5.1 Definitions and Properties of an Algorithm

An **algorithm** is a finite set of precise instructions for performing a computation or solving a problem.

5.1.1 Properties of algorithms:

- Input from a specified set
- Output from a specified set (solution set)
- Definiteness of every step in computation
- Correctness of output for every possible input
- Finiteness of each calculation step
- Effectiveness of each calculation step
- Generality for a class of problems

5.2 Analyzing Algorithms

Given some problem "P" and an algorithm "A" that is designed to solve P:

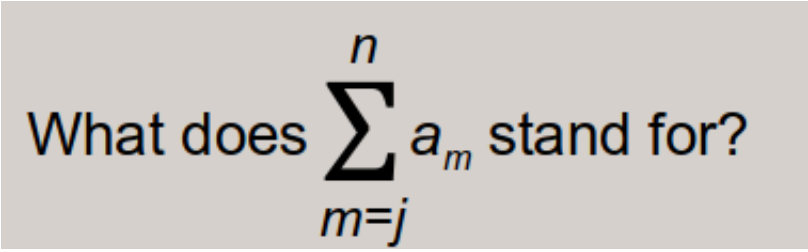
- The runtime of A can be analyzed by counting the number of certain operations that are performed during its execution
- this count is generally dependent on the size of the input.
- A **worst-case input** of size n is an input that causes A to execute the largest number of operations.

5.2.1 Algorithm Efficiency Metrics

Algorithmic efficiency is generally measured by:

- time complexity (number of elementary computations)
- space complexity (number of memory cells that the algorithm requires)

5.3 Summation Notation Basics



This graphic is a representation of the sum:

$$a_m + a_{m+1} + a_{m+1} + ... + a_n$$

The variable **j** is called the **index of summation**, which runs from m to its upper limit n.
A summation's **"closed form"** is a more readable, intuitively calculable approximation of a summation series

5.4 Analysis

count(n) is the number of := statements in a code snippet

I FEEL LIKE THERE NEEDS TO BE MORE HERE. WILL FILL IN FROM BOOK

6 Section 5.6 Comparing Rates of Growth

6.1 Complexity

When measuring complexity, we are mostly interested in how the algorithm performs in cases where the input dataset is very very large, so it is where the efficiency of the algotithm matters the most.

Exponential algorithms, for example, pretty much always cost too much time, as if (as per the example in the slides) you have a program that runs in time 2^n, and you have 1280 inputs, then the time the program will use will be greater than the number of atoms in the universe!

6.2 Comparing rates of growth

We often compare functions f and g to see whether f(n) and g(n) are about the same or whether one grows faster as n increases.

The functions might represent might represent running times of algorithms that we need to compare.

In order to compare the runtime efficiency of various algorithms, We need to use one of a few types of notation:

6.3 Big Oh

Definition: The growth rate of function "f" is bounded above by the growth rate of g such that if there are some constants d > 0 and m such that:

$$|f(n)| \leq d\ |g(n)| \text{for } n \geq m$$

Basically, this is saying that after some number m, the function g(n) on all inputs larger than m will always have a value larger or equal to the value of f(n).

If this is the case, we can say that f(n) = O(g(n)), which is to say that f(n) is big oh of g(n). Informally, we can also say that after that point, g(n) is always **bigger** than f(n)

NOTE: "=" is, here, an unusual one-way equality in that what we are really saying when we say f(n) = O(g(n)) is that f(n) $\in$ O(g(n)).

Simple Examples of Big Oh notations:

- f(n) = O(f(n))
- 200n = O((1/100)n)

- n = O(n^2)

6.3.1 Big Oh Example #1

Definition: f(n) = O(g(n)) if there exist constants d and m such that $0 \leq$ f(n) $\leq$ d*g(n) for all n $\geq$ m.
Problem: Prove that $65n^3 + 100n^2 = O(10n^4)$
This means that $65n^3 + 100n^2$ $d * 10n^4$ after some value k, we can do the following simplifications:

1. $12n^3 + 20n^2 \leq d * 2n^4$ (divided original inequality by 5)

2. 13n $+20 \leq d * 2n^2$ (divided step 1 by n^2)

If we define d=7 (aka $\lceil 13/2 \rceil$), **then** $13n + 20 <$14n^2.

We can then find "m" by testing 1, 2, 3, etc. m=2 works, because 13*2 + 20 $\leq$ 14*4.

This gives us our solution:
d=7 and m=2 show that $65n^3 + 100n^2 = O(10n^4)$

6.3.2 Big Oh example #2

Problem: Show that $16n^2 = O(2^n)$, where:

$$f(n) = 3n^2 + n^2 + 12$$

Solution:

1. $3n^2 + n^2 + 12 \leq dn^2 = 4n^2 + 12 \leq dn^2$

2. **Let:** d = 12

3. **Substitute in d:** $4n^2 + 12 \leq 12n^2$

4. **Divide by 12 to simplify:** $1/3n^2 + 1 \leq n^2$

5. **Find some value of m (n, in the specific case) such that all following values make the inequality true**

   - **n=1:** $4/3 > 1$ (does not make inequality true)
   - **n=2:** $7/3 \leq 4$ (makes inequality true)
   - **n=3:** $4 \leq 9$ (makes inequality true)

This gives us our solution:
f(n) = O(n^2) where d=12 and m=2.

6.3.3 What is Big-Oh used for?

Big-Oh is used to time-bound the worst-case of algorithms

Note: When we do time-complexity calculations to determine big oh, we use the character "c" to indicate a const.

6.4 Math Review Break!

We may need to use some of the following formulae in our calculations:

- n = 2^k IFF k = log(n)
- a = $b^{log_b a}$
- $log_c(ab) = log_c a + log_c b$
- $log_b(c/a) = log_b c$ - $log_b a$
- $log_b(1/a) = -log_b a$
- $c * log_b a = log_b a^c$
- $log^k n = log(n)^k$, but that is not the same as $log_b a^c$
- we use base 2 written lg n
- $n^{a+b} = n^a n^b$
- $n^{2a} = (n^a)^2$
- $n^0 = 1$
- n^1 = n
- $n^{-1} = 1/n$

6.5 Big Omega

Definition: The growth rate of f is bounded below by the growth rate of g if there are constants c > 0 and m such that

$$c|g(n)\ | \leq |f(n)\ | \text{for } n \geq m$$

If this is the case, we can say that f(n) = Ω(g(n)), or that f(n) is big omega of g(n). another way to state this is:

$$cg(n) \leq f(n) \text{ for } n \geq m$$

Simple examples of Big Omega Notation:

- f(n) = Ω(f(n))
- (1/100)n = Ω(200n)
- n^2 = Ω(n)

6.5.1 What is Big Omega used for?

Big Omega is used to indicate a lower-bound for an optimal solution.

6.6 Big Theta

Definition: f has the same growth rate as g if there are constants c > 0, d > 0, and m such that:

$$c\ |g(n)\ | \leq |f(n)\ | \leq d\ |g(n)\ | \text{for } n \geq m$$

Or, if g(n) != 0:

$$c \leq |f(n)/g(n)\ | \leq d \text{ for } n \geq m.$$

Note: in the above definitions, keep in mind that c and d are different. If this is the case, we can say f(n) = Θ(g(n)), or that f(n) is big theta of g(n). Another way to represent this is to say:

$$cg(n) \leq f(n) \leq dg(n) \text{ for } n \geq m.$$

Simple example of Big Theta Notation:

- n(n+1)/2 = $\Theta(n^2)$

6.6.1 What is Big Theta Used for?

Big theta is used for comparing asymptotic efficiency of algorithms.