CS 5012: Foundations of Computer Science

UNIVERSITY OF VIRGINIA
DATA SCIENCE INSTITUTE

Exam 1 Study Guide

Topics
Propositional Logic
Predicate Logic
Sets and set operations
Functions and relations
Big-O notation; Selection/order statistics
Sorting, Searching, Hashing and Indexing
Abstract Data Types – Priority Queues
Trees

The readings provided below are in addition to class material and in-class exercises

Readings

- MSD textbook (on Collab)
  - Ch. 5 (not 5.5) [Algorithms]
  - Ch. 6 (especially: 6.1, 6.2, and 6.4) [Data Structures / ADT]
  - Ch. 7 (not 7.5) [Trees / Binary Trees / BST / Heaps]
- Introduction to Algorithms (3rd edition) by Cormen et al.
  - Ch. 1, Ch. 2, Ch. 3 [Algorithms]
  - Ch. 6 (6.5) [Priority Queues]
  - Ch. 7 (not 7.3) [Quicksort]
  - Ch.8.4 (just this section) [Bucketsort]
  - Ch. 11 (specifically 11.1, 11.2) [Hashing]
- *Additional textbook*: Discrete Mathematics and its Applications (7th edition) by Rosen (See Resources on Collab)
  - Ch. 1.1-1.5 [Propositional Logic]
  - Ch. 2.1-2.3 [Sets and Functions]
  - Ch. 3 [Algorithms]

- Ch. 4.5 (read only section on hashing functions) [Hashing]
- Ch. 9.1 [Relations]
- Ch. 11 [Trees / Tree Traversals]

[Note: The majority of the questions will come from the last four (4) topic areas: Big-O to Trees]

Propositional Logic

- Syntax (symbols and rules) vs. semantics (meaning and interpretation)
- Tautology
- Contradiction
- Logically equivalent
- Valid vs. meaningful statements - doesn't have to be meaningful to be valid
- Logical Equivalence
- Implication Law
- What is a proposition? What aren't propositions? Understand and recognize the difference
- Logical Connectives/Operators
  - Not: inclusive or vs exclusive or
- Atomic propositions vs. Compound propositions
- Read / understand / create truth tables
- De Morgan's Laws
- Knowledge of converse/contrapositive/inverse – not tested directly on these
- Rules of Inference

Predicate Logic

- Propositional logic vs. Predicate logic
- What do we gain? A significant increase in expressive power
- What is a predicate?
- What are quantifiers?
- What is the universe of discourse?
- Symbolic elements
  - Variables
  - Predicates
  - Quantifiers
    - ∀ - universal quantification ("for all")
    - ∃ - existential quantification ("there exists")
  - Logical connectives
- Properties of quantifiers – order of quantifiers (does switching make a difference?)
- Predicate sentences and formulae
  - Constant symbols
  - Function symbols
  - Predicate symbols
  - Use of connectives and quantifiers
- Convert logical formulas to and from English sentences

Set Theory

- Some special sets
    - Universal set, empty set, whole numbers, natural numbers, integers, real numbers, prime numbers, rational numbers
- Basic terms and syntax/symbols
    - Set, subset, proper subset, superset, proper superset, elements
    - Union, intersection, difference, equality, compliment, element of
    - Cartesian product, disjoint, member, partition, cardinality
- Set-builder notation
    - Interpret and write
- Describe and explain sets and set-elements
- Venn Diagrams
- De Morgan's Laws

Relations and Functions

- What is a relation?
- What is an ordered pair?
- Domain (& codomain) and range
- Relations expressed as a set – (relations between one or more sets)
- Types of Relations
    - Symmetric
    - Transitive
    - Reflexive
    - Equivalence
    - Identity
    - Asymmetric
- Operations on relations
    - Union
    - Intersection
    - Difference
    - Inverse
    - Composition
- What is a function?
- A function is a special kind of relation – special property
- NOT: Types of functions
    - Linear
    - Quadratic
    - Power
    - Exponential
    - Logarithmic
    - Sine
    - Cosine
- Operations on functions
- Identify and analyze functions and relations

Analysis of Algorithms (Big-O notation; Selection/order statistics)

- What is an algorithm
- Properties of algorithms
- What does efficiency mean? What is being measured?
    - In general, the resource that interests us the most is time
- Why not just time algorithms?
    - Independent of computer (processor time), programming language, programmer, implementation details
    - Dependent on the size of input
    - Often dependent on the nature of the input
- Asymptotic analysis
    - Algorithmic complexity
    - Interested in scalability
    - The slower the asymptotic growth rate the better the algorithm
    - Asymptotic – how do things change as the input size *n* gets larger?
- Comparison of Growth rates
- "Big-O"
    - Highest order term
    - Fastest growing term
    - Ignore constants and lower order terms
    - As n gets large, highest order terms becomes most significant
    - Upper-bound on how inefficient an algorithm can be
        - E.g. Algorithm A is $O(n^2)$ means Algorithm A's efficiency grows like a quadratic algorithm **or** grows more slowly (as good or better)
- Order Classes
    - O(1) – constant time
    - O(lg n) – logarithmic time
    - O(n) – linear time
    - O(n lg n) – log-linear time
    - $O(n^2)$ – quadratic time
    - $O(n^3)$ – cubic time
    - …
    - $O(2^n)$ – exponential time
- Big-O is a good estimate
- Asymptotically superior algorithms
    - Once the problem size becomes sufficiently large, the asymptotically superior algorithm always executes more quickly
- Be able to give examples of the kinds of problems that fall under the order classes

Sorting

- A common and useful operation
- What are some of the broader issues of sorting
- Sorting stability
    - What does this mean?
    - Why is this important?
    - In what circumstances would stable sorts be useful?
- Basic operation is comparison of keys, swapping items can be expensive too

- Discuss pros and cons
- Efficiencies of the sorting algorithms
- Sorting algorithms
    - Bucket sort
    - Bubble sort
    - Insertion sort
    - Merge sort

Searching

- Another common and useful operation
- What is a sentinel value?
- What are some questions to answer before searching?
- What are the best/worst/average case scenarios for these algorithms?
- Discuss pros and cons
- Brute-force search
- Heuristics
    - What is a heuristic?
    - Why are heuristics useful? What do we gain by using heuristics?
    - When are heuristics used?
    - Given a problem, suggest (a) heuristic(s) that can be used
    - Accepting sub-optimal solutions
    - Reducing the search space
- Sequential (“Linear”) search
- Binary search
    - What is the pre-requisite?
    - Comparison between linear vs. binary search
- “Informed Search” Algorithms: Best-First / Greedy Search
    - Evaluation function
    - Heuristics (in this context)
    - Locally optimal vs globally optimal choice
- “Informed Search” Algorithms: A* Search
    - Evaluation function
    - Heuristics (in this context) “admissible heuristic”
    - Optimality of this algorithm
- Understand that there are many search algorithms

Indexing and Hashing

Hashing

- Hash table (also called “hash map”)
- Maps keys to values
- Hash function
- Hash index
- Why consider hashing/hash tables?
- Relative efficiency of hash tables compared to other structures (better)
- Uniform hash
- Applications of hash tables and hash functions

- What makes a good hash function
- What are hash collisions
    - Why do they happen?
    - Can they be avoided?
- Collision Resolution techniques
    - Chaining
    - Linear Probing
- Manual calculation of hash code vs. coding approach
- Direct-address table
    - Do not make use of hash function
    - Small set of keys, each key directly address the indices of the table (each slot corresponds to a key)

Indexing

- Facilitates search
- Indexing used for search engines
    - Indexing used in Google search
        - Crawling
        - Indexing
        - Delivering (and relevancy)
- Indexing and databases (Ordered indices)
    - Improving the speed of data retrieval (especially with large size)
    - Primary vs. secondary indices
    - Extra-level of indirection (secondary indices)
    - Understanding overhead
    - Dense vs. sparse indices
        - A primary index that is dense on an ordered table is redundant!
        - Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions
    - How much indexing to do?
        - Read heavy DBs / Write-heavy DBs / Write-only DBs
    - Two-level sparse index
    - Updating indices when deleting and inserting

Abstract Data Types – Priority Queues

- A priority queue (PQ) is an Abstract Data Type (ADT)
    - An abstract data type (ADT) is a computational model for data structures that have similarity in behavior
    - An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects of those operations for performance efficiency
    - ADTs are purely theoretical entities used to
        - Simplify the description of abstract algorithms
        - Classify and evaluate data structures
    - An ADT may be implemented by specific data structures

    - An “abstraction” to hide implementing details (e.g. microwave oven)
- The most important element deserving priority can be Min or Max depending on the nature of application
- In a PQ, an element with “high priority” is served before an element with “low priority”
- If two elements have the same priority, they are served according to their order in the queue
- A queue has ‘FIFO’ structure
- (Whereas a stack has ‘LIFO’ structure)
- Reasons one might jump a (standard) queue – why break the normal priority of a queue
    - Operating system scheduling
        - FCFS (*non pre-emptive*)
        - SJF (*non pre-emptive*)
        - SRTF (pre-emptive version of SJF)
- Priority Queue Operations
- Priority Queue – several ways to implement it
    - Heap data structure is the preferred option
    - (vs. array implementation)
        - Sorted or unordered array
        - Insertion and deletion
    - (vs. list-based implementation)
        - Sorted or unordered list
        - Insertion and deletion
- Binary Heap structure
    - Special version of a binary tree with special structure and heap-order properties
- Tree Data Structure
    - Basic vocabulary and tree elements
    - Trees are also used for sorting and searching different types of data. The insert/ delete operations can be done faster in tress than in arrays
    - Arrays are generally fixed size unless re-dimentioned at runtime. A tree naturally grows to hold an arbitrary, unlimited number of objects to go with the needs of the user
- Binary heap
    - Balanced binary tree
    - Heap must be a complete tree
    - Arrays are generally fixed size unless re-dimentioned at runtime
    - A tree naturally grows to hold an arbitrary, unlimited number of objects to go with the needs of the user
    - A binary heap is a heap data structure created using a binary tree
    - It can be seen as a binary tree with two additional constraints:
        - Shape property
        - Order (heap) property
    - Minheap vs. maxheap
        - Minheap used
        - A ‘min’ heap example will be used where the key with the lowest value is at the root
    - Binary heap inserting and retrieving the smallest value

- Heaps as 1-D arrays
  - Level order
  - Heapform
- Inserting a node into a Heap (minheap)
- Deleting a node from a Heap – successor nodes (and where to find them!)

Data Structures: Trees

- General definition and terminology
  - Recursive definition
- Tree Traversals
  - What is it
  - Why might this operation be useful/needed
  - The three common tree traversals for binary trees
    - Pre-order traversal
      - Prefix expression
    - In-order traversal
      - Sorts values from smallest to largest
      - Infix expression
    - Post-order traversal
      - Depth-first search
      - Postfix expression
    - Understand these traversal methods are applied recursively
  - Given a tree, know how to perform each of these three traversals
- Traversal applications
- Binary Search Trees (BST)
  - Binary search tree property
  - Finding and inserting in BST
  - How to traverse a BST so that the nodes are visited in sorted order?
  - Deleting from a BST – not covered