COMP 3170 – Analysis of Algorithms and Data Structures

Course Description

Calendar entry

Fundamental algorithms for sorting, searching, storage management, graphs, databases and computational geometry. Correctness and Analysis of those Algorithms using specific data structures. An introduction to lower bounds and intractability. Prerequisites: COMP 2080 and COMP 2140.

General Course Description

COMP 3170 is the third course in theoretical computer science. After learning fundamental data structures and techniques for algorithm design in COMP 2140 and COMP 2080, students will gain a deeper understanding of design and analysis of efficient algorithms. Students will learn new techniques for solving specific problems more efficiently and for analyzing space and time requirements, including showing whether a problem is unlikely to be solvable exactly by any efficient algorithm, designing algorithms that provide approximate solutions, and analyzing amortized and expected costs.

Detailed Prerequisites

Before entering this course, a student should be able to:

- Implement common data structures and their associated operations, including a binary search tree, a heap, and a hash table and analyze each operation's running time.
- Implement the common abstract data types: stack, queue, and priority queue, along with their associated operations, and analyze each operation's running time.
- Implement common sorting algorithms, and analyze their running times, including merge sort and quicksort.
- Be familiar with topics in discrete mathematics, including logical equivalence, logical implication, quantifiers, proofs, mathematical induction, introductory set theory, introductory graph theory.
- Analyze the worst-case running times of iterative and recursive deterministic algorithms.
- Apply big Oh, Omega, Theta, little oh, and little omega notation correctly in algorithm analysis.
- Understand how to apply each case of the Master Theorem to solve a recurrence relation and explain when the Master Theorem is not applicable.
- Design a divide-and-conquer algorithm to solve a given problem.
- Design a greedy algorithm to solve a given problem.
- Design a dynamic programming algorithm to solve a given problem.

Course Goals

By the end of this course students will:

- Implement a linear-time selection algorithm (e.g., quickSelect).
- Implement advanced data structures and their associated operations, including a balanced binary search tree, skip lists, and binomial heaps, and analyze the worst-case and/or expected space and time costs of each data structure and operation.
- Augment a data structure to support additional operations efficiently by writing code to extend a binary search tree to support the rank and select operations efficiently.
- Analyze the amortized cost of a sequence of operations using the aggregate method, the accounting method, or the potential method.
- Implement the disjoint set ADT, along with its associated operations, using a disjoint set forests data structure and applying the techniques of union by rank and path compression.
- Prove a lower bound on the worst-case running time of any algorithm that solves a given problem by using a decision tree argument.
- Show that any algorithm that solves a given problem requires at least as much time as another given problem in the worst case by using a polynomial-time reduction.
- Be introduced to the concept of complexity classes, including the class NP, and the distinction between NP-hard and NP-complete.
- Show that a given problem is NP-hard by using a polynomial-time reduction.
- Express a bound on the quality of the approximation factor guaranteed by an algorithm relative to an optimal solution.
- Implement an approximation algorithm for a given problem and analyze its approximation factor and its space and time costs.

Learning Outcomes

Selection

Students should be able to:

1. Implement the quickSelect algorithm.
2. Analyze the expected-time and worst-case time for quickSelect.
3. Explain the O(n) worst-case time selection algorithm (median of medians algorithm) and analyze its worst-case time.

Balanced Binary Search Trees (AVL or Red-black trees)

Students should be able to:

1. Understand the differences between weight-balanced trees and height-balanced trees. Express the balancing factor of a tree and determine whether a given tree is balanced or unbalanced.
2. Derive an upper bound on the height of a balanced tree as a function of the number of nodes.

3. Explain tree rotations, when a double rotation is necessary, the cost of a rotation, and which rotations are necessary after insertion and deletion into a balanced binary search tree.
4. Analyze the worst-case cost of insertion and deletion in a balanced binary search tree.

Augmenting Data Structures

Students should be able to:

1. Explain when and why a data structure could be augmented.
2. Explain how to augment a data structure to support additional operations efficiently.
3. Write code to extend a binary search tree class to support the rank and select operations efficiently.
4. Analyze the worst-case time for the rank and select operations as a function of the tree height and the number of nodes.

Skip Lists

Students should be able to:

1. Implement a skip list, including the operations search, insert, delete and predecessor.
2. Explain how node heights are assigned at random according to a geometric distribution.
3. Analyze the expected space and time costs of skip lists and each operation.

Binomial Heaps

Students should be able to:

1. Understand the differences in the operations supported by a mergeable heap as compared to a traditional heap.
2. Implement a binomial heap, including the operations insert, extractMax, and union.
3. Analyze the worst-case time for these operations.

Amortized Analysis

Students should be able to:

1. Explain the differences between worst-case time, expect time, and amortized time.
2. Bound the worst-case total time for a sequence of operations for specific examples (e.g., bit counter, resizable array).
3. Analyze the amortized per operation using the aggregate method, the accounting method, or the potential method.

Disjoint Sets

Students should be able to:

1. Implement the disjoint set ADT using a disjoint set forests data structure, including the operations makeSet, find, and union.
2. Explain the techniques of union by rank and path compression.
3. Understand the rate of growth of the inverse Ackermann function.

4. Express the amortized cost per operation for the disjoint set forest data structure using union by rank and path compression.

Lower Bounds

Students should be able to:

1. Using a decision tree argument, prove a lower bound on the worst-case running time of any algorithm that solves a given problem, e.g., comparison-based sorting.
2. Explain how a lower bound on the worst-case time for solving one problem can imply a lower bound on the worst-case time for solving another problem by using a reduction to show that the second problem is at least as hard to solve as the first problem.
3. Give a lower bound argument for a given problem using a reduction.

Computational Complexity

Students should be able to:

1. Understand the concept of a complexity class.
2. Explain the complexity classes P and NP.
3. Explain what it means for a problem to be NP-hard vs. NP-complete. Understand the concept of complexity classes, including the class NP.
4. Using a polynomial-time reduction, show that a given problem is NP-hard.

Approximation Algorithms

Students should be able to:

1. Explain when an approximation algorithm provides a good solution to a problem.
2. Express a bound on the quality of the approximation factor guaranteed by an algorithm relative to an optimal solution.
3. Implement an approximation algorithm for a given problem and analyze its approximation factor and its space and time costs.
4. Explain the definition of a PTAS.
5. Explain the class APX-hard.
6. Using a polynomial-time reduction, show that a given problem is APX-hard.