COMP 2080 – Algorithms: Design and Implementation

Course Description

Calendar entry

Techniques for algorithm design: divide-and-conquer, greedy, dynamic programming, and randomization. Analysis of recursive algorithms through recurrence relations. The design and implementation of common algorithms such as sorting and selection. STAT 1000 or STAT 1001 or STAT 1150 is a recommended prerequisite.

Prerequisites: COMP 2140 and [one of MATH 1240, MATH 1241, or the former COMP 2130].

General Course Description

This course introduces common algorithmic techniques used in the design of algorithms that solve a variety of problems, as well as techniques for analyzing the efficiency of algorithms in terms of their costs, such as running time and memory usage.

Students can expect to implement sorting and selection algorithms, as well as algorithms that make use of a variety of techniques such as divide-and-conquer, greedy, dynamic programming, and randomization.

The material covered here provides the foundation upon which Computer Science depends. Regardless of the third- or fourth-year course, the design and analysis of algorithms is a part of everything we do.

Detailed Prerequisites

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

- Design and implement iterative and recursive algorithms.
- Design and implement standard algorithms that manipulate arrays, linked lists, and binary trees.
- Implement abstract data types such as stacks, queues, balanced search trees, dictionaries, and priority queues using data structures such as arrays, linked lists, and/or binary trees.
- Express the worst-case cost of iterative algorithms as a function of input size.
- Apply common techniques in discrete mathematics, including logical equivalence, logical implication, quantifiers, evaluating finite summations, proofs, mathematical induction, introductory set theory, basic counting of permutations and combinations, introductory graph theory.

Course Goals

By the end of this course students will:

- Express the worst-case cost of a recursive algorithm using a recurrence relation.
- Solve certain types of recurrence relations, simplify the solution using asymptotic notation, and prove correctness using a proof by induction.
- Apply divide-and-conquer, greedy, randomized, and dynamic programming algorithmic design techniques.
- Analyze the worst-case running time of typical divide-and-conquer, greedy, randomized, and dynamic programming algorithms.
- Prove whether common properties of greedy algorithms and dynamic programming hold.
- Implement common sorting and selection algorithms.

Learning Outcomes

Recurrence Relations

Students should be able to:

1. Express the worst-case cost of a recursive algorithm using a recurrence relation.
2. Differentiate between the recursive expression for a recurrence relation and its closed-form expression.
3. Solve certain types of recurrence relations using the substitution method.
4. Simplify a recurrence relation's closed-form solution using asymptotic notation.
5. Prove the correctness of a recurrence relation's closed-form expression using a proof by induction.
6. Apply the Master Theorem to solve a recurrence relation.

Divide-and-Conquer Algorithms

Students should be able to:

1. Explain the structure of divide-and-conquer algorithms.
2. Apply divide-and-conquer as a technique in algorithm design.
3. Analyze the worst-case running time of typical divide-and-conquer algorithms using recurrence relations.

Sorting Algorithms

Students should be able to:

1. Write code that implements quicksort and merge sort, recursively, for the array data structure.

2. Explain the strategy of divide-and-conquer algorithms in merge sort and quicksort.
3. Explain the strategy of randomization in quicksort.
4. Analyze the worst-case runtimes of merge sort and quicksort.
5. Explain why the worst-case time of quicksort differs from its expected-case time.
6. Write code that implements heapsort using a heap data structure.
7. Implement a counting sort and analyze its running time.

Greedy Algorithms

Students should be able to:

8. Explain the structure of greedy algorithms, including greedy choice and growing a solution incrementally.
9. Apply greedy choice as a technique in algorithm design.
10. Analyze the worst-case running time of typical greedy algorithms.
11. Prove whether the two properties of greedy algorithms hold: the optimal substructure property and the greedy-choice property.

Dynamic Programming Algorithms

Students should be able to:

1. Explain the structure of dynamic programming algorithms.
2. Apply dynamic programming as a technique in algorithm design.
3. Analyze the worst-case running time of typical dynamic programming algorithms.
4. Prove whether the two properties of dynamic programming algorithms hold: the optimal substructure property and the overlapping subproblems property.
5. Differentiate between memoization and tabulation.
6. Implement a dynamic programming algorithm that computes both the size of a solution and returns the solution.

Randomized Algorithms

Students should be able to:

1. Explain the structure of randomized algorithms.
2. Explain the differences between Las Vegas and Monte Carlo randomized algorithms.
3. Apply randomization as a technique in algorithm design.
4. Analyze the expected running time of typical randomized algorithms.
5. Differentiate between expected running time and worst-case time.
6. Analyze the probability that a randomized algorithm returns a correct solution.

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.

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.

Algorithm Correctness

Students should be able to:

1. Define preconditions and postconditions for simple code.
2. Define loop invariants and loop measures for simple iterative code.
3. Prove partial correctness, termination, and full correctness for simple code.