CPSC 320 Learning Goals

Course-level Learning Goals

At the end of the course, a student will be able to:

1. Recognize which algorithm design technique(s), such as divide and conquer, prune and search, greedy strategies, or dynamic programming was used in a given algorithm.

2. Select and judge several promising paradigms and/or data structures (possibly slightly modified) for a given problem by analyzing the problem's properties.

3. Implement a solution to a problem using a specified algorithm design paradigm, given sufficient information about the form of that problem's solution.

4. Select, judge and apply promising mathematical techniques (such as asymptotic notations, recurrence relations, amortized analysis and decision trees) to establish reasonably tight upper and lower bounds on the running time of algorithms.

5. Recognize similarities between a new problem and some of the problems they have encountered, and judge whether or not these similarities can be leveraged towards designing an algorithm for the new problem.

Topics-level Learning Goals[1]

At the end of the course, a student will be able to:

- Asymptotic analysis:

  ASA.1 Apply the definitions of O, Ω, and Θ to derive upper and lower bounds on the worst-case running times of algorithms that use loops and conditionals.[4]

  ASA.2 Use decision trees to prove lower bounds on the worst-case running time of comparison-based algorithms for problems related to queries about, and ordering of, elements.[4]

  ASA.3 Use limits to determine the asymptotic relationship between two functions (O, o, Ω, ω or Θ).[4]

- Divide and conquer algorithms:

  DC.1 Analyze a recursive algorithm to derive a recurrence relation whose solution describes the algorithm's worst-case running time.[4]

[1]The superscripts at the end of each learning goal refer to the course-level learning goal(s) it helps achieve.

- DC.2 Prove a given upper or lower bound on the solution of a recurrence relation, using mathematical induction.[4]
- DC.3 Draw a recursion tree that corresponds to the execution of a recursive algorithm, and derive from that recursion tree a summation whose solution is the algorithm's worst-case running time.[4]
- DC.4 Apply the Master Theorem to obtain quickly the solution to most recurrence relations that arise from divide and conquer algorithms.[4]
- DC.5 Recognize algorithms that implement the divide and conquer paradigm.[1]
- DC.6 Design a divide and conquer algorithm for a problem in which the merge step is reasonably straightforward.[3]
- DC.7 Judge whether or not the divide and conquer paradigm might be appropriate for solving a problem.[2]

• Randomization:

- R.1 Explain the usefulness of randomization when constructing a data structure or designing an algorithm.[2,4]
- R.2 Compare and contrast randomized and deterministic algorithms for a same problem, and analyze advantages and disadvantages of each one.[2,5]
- R.3 For at least one type of randomized data structure D_r (for instance, skip lists, randomized binary search trees or treaps) show how each of the supported operations is performed on D_r.[2]
- R.4 Derive the expected running time of the operations on a randomized data structure, or of the execution of a randomized algorithm, as a function of the number of elements involved.[4]

• Greedy algorithms:

- GA.1 Determine whether or not an algorithm is greedy.[1]
- GA.2 Sketch a proof showing that a greedy algorithm returns the correct solution, and complete the proof in cases where this proof is similar to a proof the student has already seen.[2,4]
- GA.3 Develop greedy algorithms for problems where a reasonably straightforward greedy strategy can be applied successfully.[1,3]

• Dynamic programming:

- DP.1 Recognize that an algorithm uses dynamic programming.[1]
- DP.2 Determine the parameters that characterize instances of a given optimization problem.[2,3]

DP.3 Select and judge promising types of recurrence relation among the variants seen in class and on the assignments, by looking at the parameters that characterize instances of an optimization problem, and use one of them to express the solution to a problem.[2,3,5]

DP.4 Transform a recurrence relation for a problem into a dynamic programming algorithm to solve this problem.[3]

- Amortized analysis:

AMA.1 Give examples where amortized analysis can be used to obtain a tighter upper bound on the running time algorithm than a more straightforward analysis.[4]

AMA.2 Use progress on an invariant to obtain a tight upper bound on the running time of an algorithm using one or more loops.[4]

AMA.3 Explain the purpose of the *potential function* and how we should interpret its value.[4]

AMA.4 Use the potential method to obtain a tight upper bound on the worst-case running time of a sequence of operations on a simple data structure.[4]

AMA.5 For at least one type of data structure D_a whose analysis must be amortized (for instance, splay trees, Fibonacci heaps or Merge-Find sets), construct the data structure resulting from a sequence of operations.[2]

AMA.6 Explain how amortized analysis is critical in order to determine the running time of a sequence of operations on D_a.[2,4]

- NP-completeness:

NP.1 Explain the difference between the classes P and NP.[4]

NP.2 Write the decision problem corresponding to a given optimization problem, and vice-versa.[4]

NP.3 Describe the steps involved in proving that a problem is NP-complete.[4,5]

NP.4 Explain the implications of NP-completeness on the design of an algorithm to solve a given problem.[2,4,5]