School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Problem Solving and Algorithmic Strategies

Review

- Data can be stored in many ways and the way you store it can make a difference
  - These different forms of storage can be accessed in many different ways and this also makes a difference
  - Think of a tree or a linked list

- Being a Computer Scientist revolves around determining which approach to take and understanding why for a problem.

- Deciding on the data structure and the algorithm
  -> two pieces of one puzzle

Problem Solving

- How do we solve a problem?
  - We, human-beings, are excellent problem solvers.
  - Some solutions just leap into our minds.
  - We think about the ways we can solve problems. Our first solution is often not our best solution.
  - The more problems we solve, the better we are at solving problems

- How do machines solve a problem?
  - They must be provided with an algorithm that solves the problem
  - Or at least an algorithm that tells them how to learn to solve a problem

Algorithms

- An algorithm is:
  "A sequence of unambiguous instructions for solving a problem." (Levitin)
- Algorithmics, the study of algorithms, is the core of Computer Science.

- Families of algorithms
  - Algorithms can usually be grouped based on some criteria.
  - Here, we're going to group them by a high-level strategy that you choose for designing the algorithm

Algorithmic Strategies

- Brute force (exhaustive search)
- Backtracking
- Branch and bound
- Divide-and-Conquer
- Transform-and-Conquer
- Dynamic Programming
- Greedy Algorithms
- Heuristic Algorithms

Brute Force

- This is the simplest design strategy for discrete spaces.
  - Usually directly based on the problem statement and concepts involved, a straightforward approach to solve the problem.
- Checks all possible solutions systematically. It works!!
  - Search space usually grows exponentially with the input size
  - You use large amounts of computing power to do the job, rather than using your brain to reduce the amount of work that is required to do.
  - Unless we have a better algorithm for locating the best solution, we have to generate AND inspect each solution in turn (exhaustively) until we have looked at all of them and found which one was the best.
- Examples
  - Search in a sorted list, primality testing, 8 queens puzzle, etc

Backtracking

- Still a systematic search on the search space, but smarter than brute force!
- Consider the 8 queens puzzle
  - Consider the search tree of it!
  - When the first two queens are threatening each other, then you don't need to inspect different configurations of other queens, as the rest of the solution.
- Usually for Constraint satisfaction problems
- Builds a solution incrementally
  - Backtracks when a violation happens in the partial solution
  - Omits large parts of the search space like this

Branch and Bound

- Still a systematic search on the search space, but smarter than brute force!
- The problem is to minimize or maximize an objective function
- Search regions of the search space (branches of a tree)
- Finds lower and/or upper bounds of quality of the solutions (objective function) of regions of the search space, and based on that, decides to abandon that region or not.
  - Omits large parts of the search space like this (pruning step)

Divide-and-Conquer

- Look at a larger problem and break it up into smaller problems of roughly the same size as each other
- Solve these smaller problems (often recursively)
- Combine these smaller solutions to get the larger solution

- Examples?
  - Merge sort
  - Quick sort
  - Maximum subsequence sum problem

Transform and Conquer

- Some problems cannot be immediately solved.
- Hence we may choose to transform it to a form that is easier to solve, then we conquer it. There are three major variations:
  - Transform to a simpler or more convenient instance of the same problem
    - For example: check whether the elements of an array are unique
      - Transform to a sorted list which is easier to check for redundancy
  - Change the representation
    - For example, BST -> AVL
  - Transform to a different problem that we can solve!
    - For example, find the LCM of m and n
      - You find g=GCD(m,n) then LCM= mn/g

Transform and Conquer

- Representation change and problem reduction are generally more difficult to carry out
- Representation change requires a problem where the original form has a useful alternative representation
- Problem reduction requires you to reduce to an equivalent but simpler problem, which requires mathematical rigorous proof, usually.

Dynamic Programming

- What do we do in dynamic programming?
  - Again, break the problem down to some smaller subproblems
  - Solving each of them once
  - Storing the solutions into some data structure (usually a table)
  - Using the stored values, find answers to larger sub-problems

- Examples
  - Fibonacci
  - Counting coins

Greedy and Heuristic approaches

- Greedy algorithms choose the branch that optimizes over all choices
  - Minimizing the number of coins: choose the largest coin
  - Minimum spanning tree: choose the smallest edge

- Heuristic algorithms use knowledge about properties of the data to rank the search choices.
  - Chess: avoid moves that expose the King, moves that capture opposing Queen are high value
  - Noughts and crosses: choose middle square (maximizes possible winning moves)

Fundamentals

- Understand the problem
- Determine the computing resources you have
- Choose between exact and approximate solving
- Decide on a data structure
- Choose a general approach (Strategy)
- Specify your algorithm
- Prove its correctness
- Analyze for efficiency
  - Time/Space
  - Simplicity
  - Generality
- Implementation
- Testing


THE UNIVERSITY
of ADELAIDE