CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Problem Solving and Algorithmic Strategies

adelaide.edu.au

seek LIGHT

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 knowing when and why to take a certain approach.

- Deciding on the data structure and the algorithm -> 2 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 should 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

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
- Analyse for efficiency
  - Time/Space
  - Simplicity
  - Generality
- Implementation
- Testing

Understanding

- What is the problem that you're trying to solve?
- Solving the wrong problem is unlikely to give you the right answer - unless you're very lucky.
- Can you describe it to someone else?
- Can you write it down as a concise and precise statement?

Computing Resources

- There's an old saying "When all you have is a hammer, every problem looks like a nail."

- Which computing resources do you have?
  - Time?
  - Hardware (including memory)?
  - OS?
  - API?
  - Architecture?

Exact or Approximated Solutions

- What does your answer have to look like?
- Consider the question “how much rain has there been today?”
  - Hard (too costly) to find the exact answer
  - An answer within 90% -110% of the exact answer is acceptable
- If you’re looking for an optimal answer, must it be the best, or just not too bad?
- Other example: find the shortest path to visit some cities
- Approximation-algorithms
  - Provably produce an answer x that is r-approximate, meaning that x<=r*opt, where opt is the optimal solution for a minimization problem (or x>=opt/r for a maximization problem)

Representation

- Some problems are easier to solve in a queue, some are better expressed as trees.
- The underlying data representation makes a big difference - choose wisely.
- Consider space and time performance, ease of understanding, level of support and your own degree of familiarity.

General Approach

- How are you planning to solve this problem?
- Which algorithmic approach is the most suitable?
- Can you try a loose approach on a prototype to see if it works?
- What's your next choice?
- You should be able to explain "WHY?".

- We will talk about it in a few minutes!

Specify the algorithm

- You should clearly detail your algorithm and, even in pseudo-code, comment it clearly.

Check for correctness

- Technically, you should prove your algorithm's correctness for the range of inputs and outputs you are expecting.
- At this stage, you should be checking that, for a reasonable set of test cases, your algorithm will give the right answer.
- If you can, do mathematical verification.

Handling “No Answer”

- How do we send a “No Answer” message?
- When a mathematical solution could return an integer in the range 0..n, is returning -1 a valid “No Answer”?

- This function will fail when b==0.

```
int badCode(int a, int b) {
  return a/b;
}
```

```
int badCode(int a, int b) {
  if (b!=0)
    return a/b;
  else return 0;
}
```

Efficiency Analysis

- Your algorithm may work, but how efficient is it?
- You need to know:
  - The performance of your algorithm in realistic situations
  - The resources that you will have in those situations
- Efficiency Checklist
  - Is your algorithm capable of running in the resources that you have for the most likely range of inputs? (Space/Time efficiency)
  - Have you chosen an approach that is understandable? (Simplicity)
  - Could you reuse this somewhere else without too much work? (Generality)
- Improve your algorithm before coding

Review - 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
- Analyse for efficiency
  - Time/Space
  - Simplicity
  - Generality
- Implementation
- Testing

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.
  - Brute force/exhaustive search algorithms are 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 grows usually 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, Is Prime or not, 8 queen puzzle, etc

Backtracking

- Still a systematic search on the search space, but smarter than brute force!
- Consider the 8 queen 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

- In divide-and-conquer, we:
- 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
      - Think of a Brute force strategy first
      - Transform to a sorted list which is easier to check for redundancy
  - Change the representation
    - For example, you may decide to make an AVL tree to work on
  - Transform to a different problem that we can solve!
    - For example, you want to find the lcm of m and n
      - You find g=gcd(m,n); 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 Algorithms

- Remember our counting coins algorithm?
- The change-making problem is generally defined as "How do I give the right amount of change using the smallest number of coins?"
- In our dynamic programming approach, we just made change, with no extra rules. We found the number of possible solutions.
- Now we would like an optimal solution (minimum number of coins).

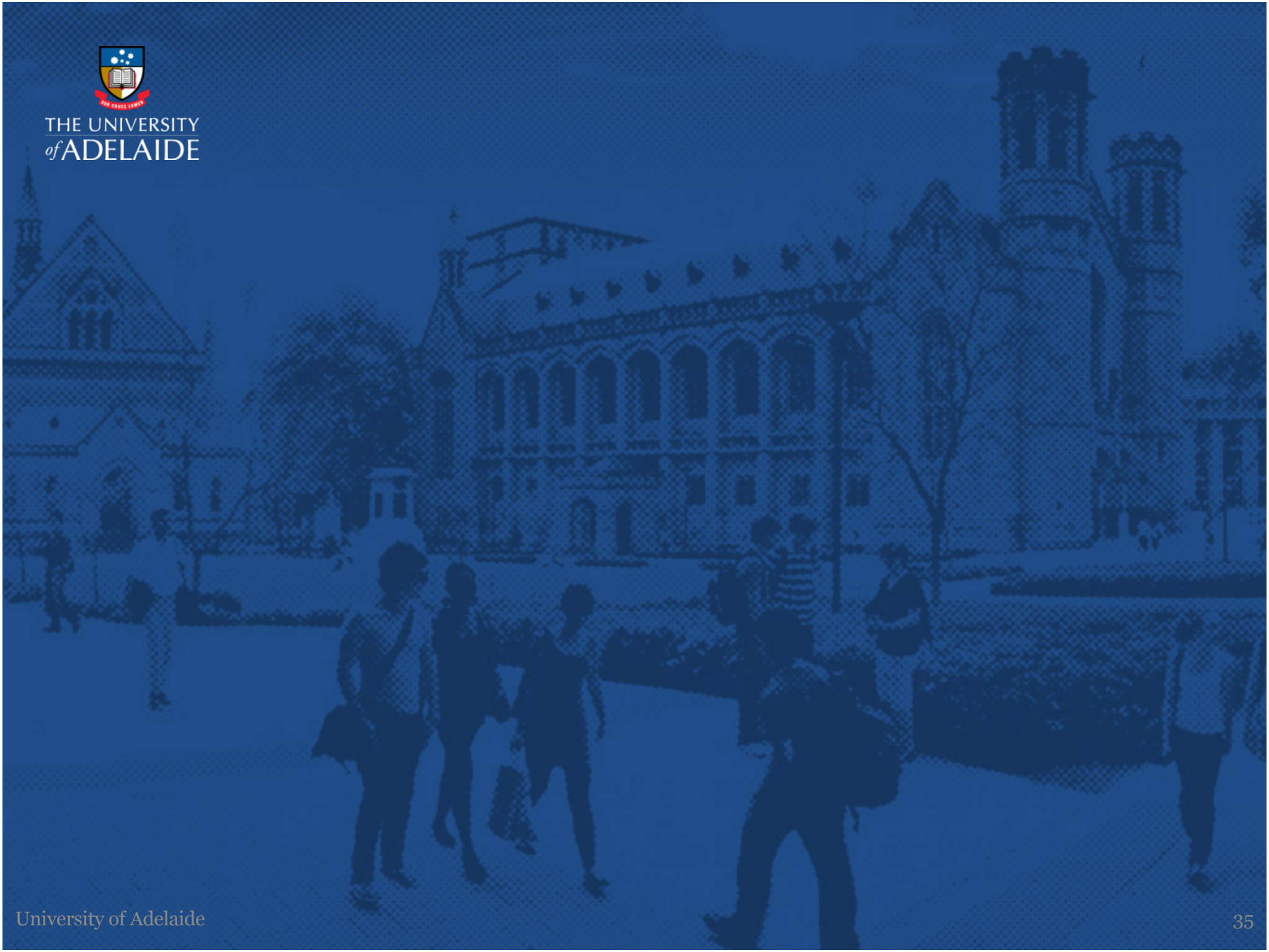
Greedy Algorithms

- We can obtain local optimal solutions to optimisation problems by constructing a solution through a set of steps, where:
  - Each choice is feasible and satisfies our requirements
  - Each choice is the best choice to make from all possible choices FOR THAT STEP
  - We cannot change the choice we make here, as we make more choices further on.

Heuristic Algorithms

- Rule of thumb!
- No guarantee on finding the best solution
- Can reduce complexity of finding an acceptable solution

- Example:
  - Visit a number of cities in different states of Australia
  - You feel that it is more rational to visit all cities of one state and then move to another state…


THE UNIVERSITY
of ADELAIDE