THE UNIVERSITY OF MELBOURNE

COMP10001

Foundations of Computing

Semester 1, 2021

Tutorial 10

Andrew Naughton

andrew.naughton@unimelb.edu.au

THE UNIVERSITY OF MELBOURNE

Outline

- Recursion
- Algorithms
- Exact vs. Approximate
- Brute Force
- Heuristic Search
- Simulation
- Divide and Conquer
- Exercises

THE UNIVERSITY OF MELBOURNE

Recursion

- A method of solving a problem where the solution depends on solutions to smaller instances of the same problem
- In other words, when a function calls itself (with a subset of the original argument that was passed into it)

THE UNIVERSITY OF MELBOURNE

Recursion

- Made up of two parts:
  - Base case: The simplest version of the problem (this will be the stopping condition for recursion)
  - Recursive case: Where the function calls itself, breaking the problem down into a smaller version

```
def excludes(nums, target):
    # base case #1
    if len(nums) == 0:
        return True

    # base case #2
    head, rest = nums[0], nums[1:]
    if head == target:
        return False

    # recursive case
    return excludes(rest, target)
```

THE UNIVERSITY OF MELBOURNE

Algorithms

- A set of steps for solving an instance of a particular problem type
- As the size of the our inputs increase, the efficiency of our algorithms becomes important and hence we study how to write them well

- Programming is simply learning how to write code:
  - how to use correct grammar and structure when writing in a programming language so that a computer can understand what we intend to communicate.
- Studying algorithms is learning about good code:
  - Elegant ways of solving problems and how to use more advanced techniques to write more powerful programs

THE UNIVERSITY OF MELBOURNE

Algorithms

- We judge algorithms on 2 factors:
  - Correctness
    - Does the algorithm produce the right answer for all inputs?
  - Efficiency
    - How much time and memory does it take/consume?

THE UNIVERSITY OF MELBOURNE

Exact vs. Approximate

- Two distinct approaches to solving problems:
  - Exact
    - Calculates a solution with a guarantee of correctness
  - Approximate
    - Estimates the solution, so may not be as correct
- When might you prefer an approximate approach?
  - If a problem is too complex to calculate precisely, it may be more practical to use an approximate approach
  - E.g. Geo-location

THE UNIVERSITY OF MELBOURNE

Brute Force

- Also known as Generate and Test
- Exact approach

- Finds the solution by enumerating every possible answer and testing them one-by-one
- If there are a billion options, this method for solving could take years, which is infeasible

- E.g. Password cracking by generating all possible combinations of characters

THE UNIVERSITY OF MELBOURNE

Heuristic Search

- Also known as Greedy Search
- Approximate approach

- Finds a solution by using a more efficient (approximate) method than one which is completely correct. The result may not be correct/optimal, but an approximate solution will be found efficiently, which is preferred if the alternative is a significantly slower algorithm.

- E.g. Finding shortest path from source to target location

THE UNIVERSITY OF MELBOURNE

Simulation

- Approximate approach
- Finds a solution by generating a lot of data to predict an overall trend
- E.g. Simulating the play of a game of chance to test whether it's worth playing. Many rounds can be generated and combined to find whether, on average, money was won or lost. In the real world, the outcome may be different, but by running simulations we can find the most likely outcome

THE UNIVERSITY OF MELBOURNE

Divide and Conquer

- Exact approach
- Finds a solution by dividing the problem into a set of smaller sub-problems which can be more easily solved, then combining the solutions of those sub-problems to find the answer to the overall problem
- E.g. Binary search, Sorting algorithms (e.g. Merge-sort, Quick-sort)

THE UNIVERSITY OF
MELBOURNE

Exercises