COMP10001 Foundations of Computing
Algorithms

Semester 2, 2016
Chris Leckie

October 2, 2016

Lecture Agenda

- Last lecture:
    - Encoding
- This lecture:
    - Properties and families of algorithms

What is an Algorithm?

- Definition: An algorithm is a set of steps for solving an instance of a particular problem type
- Computational desiderata of algorithms:
    - **Correctness**
        - an algorithm should terminate for every input with the correct output
        - incorrect algorithms can either: (a) terminate with the wrong output; or (b) not terminate
    - **Efficiency**
        - *runtime*: run as fast as possible
        - *storage*: require as little storage as possible

Example: Searching

- Search
  - looking for (the first instance of) particular value in a “collection”
  - specification:

```
def search(value, numbers):
    '''Find value in list numbers
       and output its position,
       or None if not present.'''
```

  - Examples:

```
>>> search(4, [1,3,4,7,8])
2
>>> search(9, [1,3,4,7,8])
None
```

Linear Search: Algorithm

- Algorithm idea:
  1. Initialise the index to the first element of the list
  2. While the index points to a list element:
     (a) If the value at the current list index is equal to the required value, terminate and return the index
     (b) Else increment the index
  3. If the index has run off the end of the list, return `None`

Algorithmic Analysis: Linear Search

- Is it correct? How do we know?
- Is it run-time efficient? How efficient is it (best case vs. worst case vs. average)?
- It is storage efficient? How efficient is it (best case vs. worst case vs. average)?

Binary Search: Algorithm

- Algorithm idea:
  1. Initialise a sub-list to the full list, and our index to the mid-point of the list
  2. While the sub-list is non-empty:
     (a) If the value at the current list index is smaller than the one wanted, continue to search over the right half of the current sub-list
     (b) Else if the value at the current list index is *larger* than the one wanted, continue to search over the *left* half of the current sub-list
     (c) Else if the value at the current list index is equal to the required value, terminate the search and return the current index
  3. If the list has been exhausted without finding the value, return `None`

Algorithmic Analysis: Binary Search

- Is it correct? How do we know?
- Is it run-time efficient? How efficient is it (best case vs. worst case vs. average)?
- It is storage efficient? How efficient is it (best case vs. worst case vs. average)?

Exact vs. Approximate Methods

- Exact approach: calculate the solution (set), with a guarantee of correctness, e.g.:
  - brute force
  - divide and conquer
- Approximate approaches: estimate the solution (set), ideally with an additional estimate of how "close" this is to the exact solution (set), e.g.
  - simulation
  - heuristic search
- Always use exact approaches where possible

Brute-Force (aka "Generate and Test")

- Assumptions
  - A candidate answer is easy to test
  - The set of candidate answers is ordered or can be generated exhaustively
- Strategy
  - Generate candidate answers and test them one by one until a solution is found
- Examples:
  - Linear search
  - Test whether a number is prime
  - Maze solving

Generate and Test: Example I

- How old is the captain?
  The length (in metres) of a ship is an integer. The captain has sons and daughters. His age is greater than the number of his children, but less than 100. How old is the captain, how many children does he have, and what is the length of the ship if the product of these numbers is 32118?
- Constraints:

```
length * children * age = 32118
children < age < 100
children >= 4
```

Generate and Test: Example II

```
for children in range(4, 99):
  for age in range(children + 1, 100):
    if (32118 % (children * age) == 0):
      length = 32118 / (children * age)
      print("{} children, \
      the captain's age is {} \
      and the ship's length is {}".format(\
      children, age, length))
```

Divide and Conquer

- Strategy:
  - Solve a smaller sub-problem
  - Extend the sub-solution to create the solution of the original problem
- Examples:
  - Binary search
  - Fibonacci number generation
- Divide and conquer relates closely to recursion (although the actual solution may be in iterative terms)

Divide & Conquer and Memoisation I

- The calculation of Fibonacci numbers naturally lends itself to divide-and-conquer approaches (as each element is defined relative to earlier elements in the sequence)
- Through the advent of "memoisation" (i.e. storing the value for each element used, to avoid having to recalculate it), it is possible to achieve significant gains in efficiency

Divide & Conquer and Memoisation II

```python
fib = {}

def fib_fast(n):
    global fib
    if n < 2:
        return 1
    else:
        if n-1 not in fib:
            fib[n-1] = fib_fast(n-1)

        if n-2 not in fib:
            fib[n-2] = fib_fast(n-2)

        return fib[n-1] + fib[n-2]
```

More Divide & Conquer I

- A more interesting example of divide & conquer:

  Given a list of integers, calculate the maximum sum of a contiguous sublist of elements in the list

  ```
  >>> lst = [5, 3, -1]
  >>> maxsubsum(lst)
  8
  ```

- The brute-force solution simply calculates the sum of each (non-empty) sublist, and calculates the maximum among them

More Divide & Conquer II

- The divide-and-conquer approach work as follows:
  - Assume `maxsubsum(i-1)` is the maximum sum for the sublist `lst[:i]`
  - The maximum sum for the sublist `lst[:i+1]` is `max(lst[i],lst[i]+maxsubsum(i-1))`
- The recursive version will have issues with the limit on recursion depth, so implement iteratively

More Divide & Conquer III

```
def dc_maxsubsum(lst):
    assert len(lst) > 0
    max_sum_i = [lst[0]]
    for i in range(1,len(lst)):
        max_sum_i.append(max( \
          max_sum_i[-1]+lst[i],lst[i]))
    return max(max_sum_i)
```

More Divide & Conquer IV

- How run-time efficient are the respective implementations (brute-force vs. divide-and-conquer)? (best case vs. worst case vs. average)?
- How storage efficient are the respective implementations (brute-force vs. divide-and-conquer)? (best case vs. worst case vs. average)?

Simulation

- Strategy:
    - Randomly generate a large amount of data to predict an overall trend
    - Use multiple runs to verify the stability of an answer
    - Used in applications where it is possible to describe individual properties of a system, but hard/impossible to capture the interactions between them
- Applications:
    - Weather forecasting
    - Movement of planets
    - Prediction of share markets

Simulation: A Game of Chance

- Gambling game:
  - You bet \$1, and roll two dice
  - If the total is between 8 and 11, you win \$2
  - If the total is 12, you win \$6
  - Otherwise, you lose
- Is it worth playing?
  - Start with a \$5 float and play to \$0 or \$20
  - How many games do you win on average?

Monte Carlo Simulation

- Method:
  - iteratively test a model using random numbers as inputs
  - problem is complex and/or involves uncertain parameters
  - a simulation typically has at least 10,000 evaluations
  - approximate solution to problem that is not (readily) analytically solvable
- Game of chance:
  - should a casino offer this game?

Heuristic Search

- Strategy:
  - Search via a cheap, approximate solution which works *reasonably* well *most* of the time ... but where there is no proof of how close to optimal the proposed solution is
- Examples:
  - Play your lowest (valid) card every play in Whistful Hearts

Lecture Summary

- What is an algorithm?
- What computational desiderata are associated with algorithms?
- What are "exact" and "approximate" methods? What are common examples of each?
- What are each of: brute-force, divide and conquer, simulation and heuristic search?