COMP10001 Foundations of Computing
Algorithms

Semester 1, 2019
Tim Baldwin, Nic Geard, Farah Khan, and Marion Zalk

THE UNIVERSITY OF MELBOURNE

— VERSION: 1449, DATE: MARCH 1, 2019 —

Lecture Agenda

- Last lecture:
  - The Internet and HTML
- This lecture:
  - Properties and families of algorithms

Lecture Outline

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):
# Inputs: Numbers = list of numbers
#         value = number
# Output: position of value in Numbers
# or None if value is not in Numbers
```

  - Examples:

```
>>> search(4, [3,1,4,2,5])
2
>>> search(7, [3,1,4,2,5])
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

Initialise a sub-list to the full list, and our index to the mid-point of the list
While the sub-list is non-empty:

- 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
- 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
- Else if the value at the current list index is equal to the required value, return the current index

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)?

Lecture Outline

1. Algorithm Fundamentals
2. Algorithm Families

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
  - Our solution for Coins and Denominations using loops

Coins: A Bruteforce Approach

```python
'''Count the number of combinations of
   (1,2,5,10,20) that sum to N
'''
answer = 0
for a in range(N+1):
  for b in range(N//2+1):
    for c in range(N//5+1):
      for d in range(N//10+1):
        for e in range(N//20+1):
          if a+2*b+5*c+10*d+20*e == N:
            answer += 1
```


Divide and Conquer

- Strategy:
  - Solve a smaller sub-problem
  - Extend the sub-solution to create the solution of the original problem
  - (Sounds like recursion, but can also be iterative.)
- Examples:
  - Binary search
  - Many sorting algorithms

Binary Search: Divide and Conquer

```python
def bsearch(val,nlist):
    return bs_rec(val,nlist,0,len(nlist)-1)

def bs_rec(val,nlist,start,end):
    if start > end:
        return None
    mid = start+(end-start)//2
    if nlist[mid] == val:
        return mid
    elif nlist[mid] < val:
        return bs_rec(val,nlist,mid+1,end)
    else:
        return bs_rec(val,nlist,start,mid-1)
```


Divide & Conquer and Memoisation

- Naive implementations of divide and conquer can lead to many repeated, identical function calls
- For example, the calculation of Fibonacci numbers

```python
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- F(9) calls F(8) & F(7); F(8) calls F(7) & F(6); ...
- "Memoisation" (i.e. storing the value for each element used, to avoid recalculating it), can lead to more efficient algorithms.

Divide & Conquer and Memoisation

```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

- 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

  ```python
  >>> 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

```
def sublist_sum_bf(lst):
  """Brute force."""
  max_so_far = (lst[0], 0, 0)

  for s in range(len(lst)):
    for e in range(s, len(lst)):
        subsum = sum(lst[s:e+1])
        if subsum > max_so_far[0]:
            max_so_far = (subsum, s, e)
  return(max_so_far)

print(sublist_sum_bf([3,-1,-2,2]))
```


More Divide & Conquer

- The divide-and-conquer approach work as follows:
  - Assume `maxsubsum(i-1)` is the maximum sum for the sublist ending at $i-1$ (inclusive)
  - 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

```
def dc_maxsubsum(lst):
  assert len(lst) > 0
  max_sum_i = [lst[0]]
  # base case

  for i in range(1,len(lst)):
    c1 = lst[i]
    # start new

    c2 = max_sum_i[-1]+lst[i]
    # or extend

    max_sum_i.append(max(c1, c2))
    # now take the max of all these

  return max(max_sum_i)
```


More Divide & Conquer

- 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:
  - For finding a closest neighbor many Location-based Services use Euclidean distances as they are easy to compute
  - There is no guarantee they are equal to road-network distances though
  - So whatever you find, is just "possibly" a good solution

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?