COMP10001 Foundations of Computing
Semester 1, 2019

Tutorial Questions: Week 12

— VERSION: 1474, DATE: MAY 23, 2019 —

Discussion

1. What is an "algorithm"? Why are algorithms a large area of Computer Science?

   A: *An algorithm is a set of steps for solving an instance of a particular problem type. They are important in Computing because every time we solve a problem with programming, we have written an algorithm to do so. As we deal with more and more data, the efficiency of our algorithms becomes important and hence we study how to write them well.*
   You can think of programming and algorithms as literacy and literature: programming is simply learning how to write code, how to structure the grammar of our programming languages to write statements which the computer can execute. Studying algorithms is learning about good code, elegant ways to solving problems and how to use more advanced vocabulary to write more powerful coding sentences.

2. What are the two criteria with which we can judge algorithms?

   A: *Correctness judges whether an algorithm produces the correct output for each input (eg. Grok green diamonds)*
   Efficiency judges, basically, how good an algorithm is. Takes into account aspects such as how quickly it runs, how much storage it demands and how much processing power it requires. We haven't looked at this measure in COMP10001, but it's important as you learn more about algorithms.
   You could have, for example, an algorithm which calculates a correct answer but takes 150 years to do so, or one which finishes in seconds but may not produce the most correct result in all cases.

3. What is the difference between exact and approximate approaches to designing an algorithm? Why might an approximate approach be necessary?

   A: *An exact approach calculates a solution with a guarantee of correctness. Approximate approaches estimate the solution so may not be as correct.*
   If a problem is too complex to calculate with full completeness (ie. the efficiency of a correct algorithm is very low), it might be worth using an approximate approach to make it feasible to run the algorithm.

4. Identify the following as belonging to the exact or approximate approaches to algorithms, and discuss how they approach solving problems with some examples:
   Brute-Force (Generate and Test), Heuristic Search, Simulation, Divide and Conquer

   A: *Brute-Force approach: exact approach. Finds the solution by enumerating every possible answer and testing them one-by-one. This requires answers to be within a scale that allows the program to run to completion: if there are billions of options, it may take years to do this, rendering this approach useless to that problem. Examples include Linear search and Problem 1 of this tutorial sheet.*
   Heuristic Search: approximate approach. Finds a solution by using a more efficient approximate method than one which is completely correct. The answer will not be optimal but often "close enough" especially when the alternative is a significantly slower algorithm. Examples include finding the closest distance or shortest path to a destination: finding the definitive solution would require processing many possibilities but by looking at the most likely solutions, we can find a rough answer in much less time.
   Simulation: approximate approach. Finds a solution by generating a lot of data to predict an overall trend. One example is simulating playing a game of chance to test whether it's worth playing. Many rounds of results 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. Other examples include modelling movement of objects by using physics formulas and the prediction of weather patterns by running simulations of models based on past observations.
   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 of the overall problem. Examples include binary search and a lot of sorting algorithms. The idea behind Divide and Conquer based sorting algorithms is that they may, for example, sort each half of a list separately before combining them, leading to a faster solution than sorting the whole list at once.

   Now try Exercise 1

5. What are some of the bases we can store numbers with? How can we convert between bases in Python?

   A: *As humans, we usually use decimal numbers (base 10). Other bases which we can store numbers with include base 2 (binary numbers); base 8 (octal numbers); and base 16 (hexadecimal numbers). We can convert to binary, octal and hexadecimal by using the* `bin()`, `oct()` *and* `hex()` *functions respectively. Converting to a decimal number can be done using the* `int(num, base)` *function, where* `num` *is the number to be converted to decimal and* `base` *is the base to be converted from. Writing binary, octal and hexadecimal numbers directly into code can be achieved by putting* `0b`, `0o` *and* `0x` *respectively before the number.*

6. Why are binary and hexadecimal convenient for computers? Why is decimal more difficult?

A: *Binary numbers are good for computers because they store data in a binary format and processors handle binary values to make calculations. That's why computer storage is measured in "bits" or "bytes". So binary is a computer's "native" type like decimal is ours. Hexadecimal is good since one hex character can be stored with exactly four bits* ($0000_2 = 0_{16}, 0001_2 = 1_{16} ... 1110_2 = E_{16}, 1111_2 = F_{16}$)*. Decimal is more difficult because to use decimal values, computers must convert them to and from a binary representation and sometimes that conversion isn't exact, resulting in numbers which contain rounding errors or other inconsistencies.*

Now try Exercise 2

Exercises

1. Search the following sorted lists for the number 8, using **(a)** Linear search (Brute-Force approach) and **(b)** Binary search (Divide and Conquer approach)

   Think about the best, worst and average case scenarios of these algorithms. For example, can the best case scenario of a Brute-Force algorithm be faster than running the same task with a more clever algorithm?

(a)	1	2	4	5	8	9	10	12	15	19	21	23	25
(b)	8	9	11	15	16	17	22	24	27	28	29	32	33
(c)	2	4	5	6	7	9	11	12	13	15	19	22	25

   A: *Linear search iterates through the list from start to end, testing whether each item of the list is the one being searched. This is an example of Brute-Force because there is no logic to the order of the search, it will simply work its way through each possibility to find the answer.*
   Binary search starts at the middle of the list and proceeds to search its upper half or lower half depending on whether the middle item is smaller or larger respectively than the value being searched for. This is an example of Divide and Conquer approach as the area to be searched is divided in half each iteration.
   Linear search will have its best case scenario in list (b) because the item being searched is first in the list, therefore the first item that linear search will check. It will be faster than binary search in this case, as the latter has to start in the middle and move through the list until it gets to the start.
   Linear search has its worst case scenario in list (c) because the item being searched for is not in the list and therefore linear search will iterate through each item before it comes to that conclusion. Binary search is much faster as it finds the position where the number would be, and when it's not there returns that result immediately.
   Binary search is much more reliable than linear search since there isn't much variance in its worst case and best case scenarios: it will always take a fairly short amount of time to run. Linear search may be faster in its best case, but will be much slower in its worst and you can never depend on a best case scenario!
   Also note that a list must be sorted to use binary search: not so for linear search as it doesn't have any logic which relies on the order of the elements in the list.

2. Convert the binary number 1110101 to hexadecimal by filling in the Xs of the following diagram:

$$\begin{array}{c}
1110101_2 \\
\Downarrow \\
\begin{array}{cc} 1X1_2 & 01XX_2 \\ \Downarrow & \Downarrow \\ X1X1_2 & 01XX_2 \\ \Downarrow & \Downarrow \\ X_{16} & X_{16} \end{array} \\
\Downarrow \\
XX_{16}
\end{array}$$

A:

$$\begin{array}{c}
1110101_2 \\
\Downarrow \\
\begin{array}{cc} 111_2 & 0101_2 \\ \Downarrow & \Downarrow \\ 0111_2 & 0101_2 \\ \Downarrow & \Downarrow \\ 7_{16} & 5_{16} \end{array} \\
\Downarrow \\
75_{16}
\end{array}$$

Step 1: separate into 4-bit sequences
Step 2: Add leading zeroes to make them all four bits long
Step 3: Directly convert binary numbers (0000-1111) into hexadecimal numbers (0-F)
Step 4: Combine hexadecimal numbers together, retaining place value of the original binary sequence

Problems

1. Write a Brute-Force algorithm to solve the following problem:

   The length 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?

   A:

```python
# Conditions:
# (1) length == int(length) (length of ship is an integer)
# (2) children >= 4 (has multiple sons and daughters)
# (3) children < age < 100 (age greater than num children, less than 100)
# (4) length * children * age = 32118 (product is 32118)

# Iterates through all possibilities, checks conditions and returns result
for children in range(4, 99):  # From (2)
    for age in range(children + 1, 100):  # From (3)
        length = (children * age)  # From (4)
        if 32118 % length == 0:  # From (1)
            print("Found answer")
            print("Children:", children)
            print("Age:", age)
            print("Length", length)
            break  # No need to continue checking once found
```

2. Implement linear search and binary search in Python. For an extra challenge, write a recursive version of binary search.

A:

```python
def linear_search(my_list, item):
    """Returns first index of `item` in `my_list`
    or None if it doesn't appear"""

    for i in range(len(my_list)):
        if my_list[i] == item:
            return i
    # Haven't found it after iterating completely
    return None
```

There are several different approaches to implementing binary search: this one slices the list to reduce it on each repetition. You can also keep track of a beginning and an end index which means more variables but you don't have to worry about adding an offset for finds in the second half of a list. Check the lecture slides for a recursive implementation of this approach.

```python
def binary_search_rec(my_list, item):
    """Recursively finds & returns the index of an instance of `item` in
    sorted list `my_list` using the binary search algorithm.
    Returns None if there is no such instance."""

    # Base case: if there's only one element left,
    # either it's the 0th elemnent or not there - return None
    if len(my_list) == 1:
        if my_list[0] == item:
            return 0
        else:
            return None

    mid = len(my_list) // 2
    # Midpoint is too big: recurse over first half
    if my_list[mid] > item:
        return binary_search_rec(my_list[:mid], item)

    # Midpoint is too small: recurse over second half
    elif my_list[mid] < item:
        result = binary_search_rec(my_list[mid:], item)
        # If found in second half, we need to add on the midpoint value
        # since the returned index refers only to the position in the sub-list
        if result is not None:  # Only add mid if item was found
            result += mid
        return result

    # Base case: found the item
    else:
        return mid
```

A:

```python
def binary_search_iter(my_list, item):
    """Iteratively finds & returns the index of an instance of `item` in
    sorted list `my_list` using the binary search algorithm.
    Returns None if there's no such instance."""

    offset = 0
    # `my_list` will be halved each loop: while there's more than one item
    # left in it, there's more to search
    while len(my_list) > 1:
        mid = len(my_list) // 2

        # Shortens `my_list` to first half to continue searching
        if my_list[mid] > item:
            my_list = my_list[:mid]

        # Shortens `my_list` to second half, adds mid to offset
        elif my_list[mid] < item:
            my_list = my_list[mid:]
            offset += mid

        # Found: return
        else:
            return mid + offset

    # `my_list` exhausted: item either in 0th position or not there.
    if my_list[0] == item:
        return offset
    else:
        return None
```

3. Write a function which takes a string and returns it as a sequence of binary ASCII values, stored as a string in which each character should take up 8 spaces. Write another function which takes this string of 1s and 0s and converts it back into the original string of text. The `ord()`, `chr()` and `int(_, 2)` functions should be useful.

   A:

```python
def convert_number(text):
    """Converts string `text` to a string of 8-bit binary encoded characters"""
    binary = ""
    for char in text:
        o = ord(char)
        # F-string formats binary conversion of `o` to take 8 characters
        # of space, padded with zeroes
        binary += f"{o:08b}"
    return binary

def interpret(binary):
    """Converts to text a string made of 8-bit binary encoded characters"""
    text = ""
    # Iterates through indices which finish an 8-character string of bits
    for i in range(8, len(binary) + 1, 8):
        num = binary[i - 8:i]
        # Converts num (interpreted as a binary number) back into an integer,
        # followed by conversion to a character and addition to text string
        text += chr(int(num, 2))
    return text
```