

COMP10001 Foundations of Computing

Semester 2, 2022

Tutorial Solutions: Week 11

— VERSION: 1478, DATE: OCTOBER 10, 2022 —

Discussion

1. What is HTML? What is it used for?

A: *HTML (Hypertext Markup Language) is a markup language (not a programming language like Python) used to take a document composed of text and other media (images, video, hyperlinks...) and communicate how it is to be rendered for display. It is used extensively on the web: an internet browser displays a webpage by downloading and presenting the website's HTML.*

2. Revise the HTML tags you've learned about. How do we use them to format our document?

A: *Tags are `<angle bracket delimited>` commands which give instruction about how a document is to be formatted. Often there will be an "opening" `<tag>` and "closing" `</tag>` pair of tags where the second includes a slash to show it is closing the first. This communicates that text between the tags should be displayed or interpreted in a particular way. Tags which we've covered include text formatting `` (bold text), `<i>` (italic text) and `<u>` (underline); structural tags such as `<html>` (covers an entire html document), `<head>` (header section) and `<body>` (the body content of a document); lists and tables such as `` (unordered bullet-point list), `` (ordered list), `` (denoting a single list item), `<table>` (holding a table) which contains `<tr>` (table rows) and `<td>` (table cells); and media tags `<a>` (hyperlink to a URL), `` (an image), `<audio>` (some audio) and `<video>` (a video).*

3. What is an HTML entity? Why are they needed?

A: *An entity is a special character which must be written with a special `&entity;` syntax. An example of entities are the angle bracket characters which form part of HTML syntax and would therefore be impossible to include as simple characters without the use of entities (`<` is `<` and `>` is `>`).*

4. What is the difference between a static and dynamic HTML page?

A: *A static page is one which does not change each time it is sent to a user who requests it. A dynamic HTML page is one where its contents are generated by a computer each time it is requested, allowing them to be tailored to the user requesting it. An example of a static HTML page is an information page for a product which has no need to be personalised as it should be the same for any customer who looks at it. A dynamic HTML page could be Grok, since your name is displayed in the top corner and your code will be included, making the website different for every person loading it: Grok's system inserts your information before it sends you the HTML document.*

Now try Exercises 1 & 2

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

A: *An algorithm is a sequence of steps for solving an instance of a particular problem type. They are important in Computing because every time we solve a problem with code, 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 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 vocabulary to write more powerful coding sentences.*

6. 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 how "good" an algorithm is. This 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.*

7. 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.*

8. 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: exact approach. Finds the solution by enumerating every possible answer and testing them one-by-one. This requires answers to be within a calculable range that allows the program to run to completion: if there are billions of options, it may take years to do this, rendering a brute-force approach useless to that problem. Examples include Linear search and Problem 2 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 may not be optimal but is 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 the play of 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 climate 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 many 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 3

Exercises

1. Try visiting your favourite website and viewing the HTML source. How many tags can you recognise? Try editing the source to change the appearance of the webpage (this change will only occur locally on your computer). With knowledge of HTML, you can do a lot more on the web!
2. The function `customise_html` takes three arguments, `template_file` (the filename of a html template, see next page for an example), `context` (a dictionary of strings), `outfile` (a string filename for the output). `customise_html` should find variables within the html template denoted by double curly brackets surrounding a variable name, then replace it with the variable's value from the context dictionary. For example, if the template contained the snippet `{{ shape }}`, with `context = {"shape": "triangle"}`, then the output file would contain `triangle`.

For the `template.html` file on the next page, the function should be able to be called like this to generate an output html file which can be opened in your browser:

```
context = {"name": "Dragon",
           "desc": "A legendary creature. Breathes fire!",
           "power": "100",
           "picture_file": "https://upload.wikimedia.org/wikipedia/commons/
                           /thumb/6/6b/Easy_origami_dragon_for_beginners-
                           _how_to_paint_a_dragon_for_beginners.jpg/320px"
                           "-Easy_origami_dragon_for_beginners-_how_to_pai"
                           "nt_a_dragon_for_beginners.jpg"}
customise_html("template.html", context, "out.html")
```

All the required lines of the function are available below, out of order. Put each line number in the correct order, and introduce appropriate indentation.

```
1 out.write("".join(template))
2 start = idx + TOKEN_LEN
3 key = "".join(template[start:end]).strip()
4 if template[idx:idx + TOKEN_LEN] == ["{", "{"]:
5 with open(template_file, 'r') as file, open(outfile, 'w') as out:
6 template = list(file.read()); idx = 0
7 def customise_html(template_file, context, outfile):
8 idx += 1
9 template[idx:end + TOKEN_LEN] = escape(context.get(key, ""))
10 from html import escape
11 while idx < len(template):
12 TOKEN_LEN = 2
13 end = start + template[idx + TOKEN_LEN:].index("}")
```

template.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Your new card!</title>
    <nav class="navbar bg-primary">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">Card deck</a>
      </div>
    </nav>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css"
      rel="stylesheet" crossorigin="anonymous"
      integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi">
    </head>
    <body>
      <div class="container-fluid">
        <div class="card" style="width: 18rem;">
          
          <div class="card-body">
            <h5 class="card-title">{{ name }}</h5>
            <p class="card-text">{{ desc }}</p>
            <a href="#" class="btn btn-primary">Play</a>
          </div>
          <div class="card-footer">
            Magic level: {{ power }}
          </div>
        </div>
      </div>
      <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js"
        integrity="sha384-OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3">
      </script>
    </body>
  </html>
```

A:

```
from html import escape
TOKEN_LEN = 2
def customise_html(template_file, context, outfile):
    with open(template_file, 'r') as file, open(outfile, 'w') as out:
        template = list(file.read()); idx = 0
        while idx < len(template):
            if template[idx:idx + TOKEN_LEN] == ["{", "{"]:
                start = idx + TOKEN_LEN
                end = start + template[idx + TOKEN_LEN:].index("}")
                key = "".join(template[start:end]).strip()
                template[idx:end + TOKEN_LEN] = escape(context.get(key, ""))
                idx += 1
            out.write("".join(template))
```

3. 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 for. 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.

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:

```
# 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 = 32118 / (children * age) # From (4)
        if length == int(length): # 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. Each search function should work as follows: `search([1, 4, 6, 8, 9], 4)` should return `True`.

A:

```
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.

```
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 element 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:

```
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 list of lists containing cell values and optionally two lists specifying row and column headings, and formats the cell values into a HTML table, returning the HTML text as a string. `list_to_table([[1, 2, 3], [4, 5, 6]], ['r1', 'r2'], ['c1', 'c2', 'c3'])` should return a string equivalent to the following HTML.

```
<table>
  <tr>
    <td>&nbsp;</td>
    <th>c1</th><th>c2</th><th>c3</th>
  </tr>
  <tr>
    <th>r1</th><td>1</td><td>2</td><td>3</td>
  </tr>
  <tr>
    <th>r2</th><td>4</td><td>5</td><td>6</td>
  </tr>
</table>
```

A:

```
def generate_cell(value, tag="td"):
    return f"<{tag}>{value}</{tag}>"

def generate_row(row, tag="tr", first_cell=""):
    row_out = f"<{tag}>{first_cell}"
    for element in row:
        row_out += generate_cell(element, tag)
    return row_out + "</tr>"

def list_to_table(lst, row_headings=None, col_headings=None):
    out = "<table>"
    if col_headings:
        first_cell = ''
        if row_headings:
            assert len(row_headings) == len(lst)
            first_cell = generate_cell("&nbsp;")
        out += generate_row(col_headings, tag="th", first_cell=first_cell)
    for i in range(len(lst)):
        first_cell = ''
        if row_headings:
            first_cell = generate_cell(row_headings[i], tag="th")
        out += generate_row(lst[i], first_cell=first_cell)
    return out + "</table>"
```