COM661 Full Stack Techniques and Development

Practical A4: Recursive Techniques

Aims

- To introduce recursion as a programming technique for the specification of simplified solutions to complex problems
- To provide practice in designing recursive solutions
- To identify the problem of circular definitions in recursive solutions
- To introduce iterative relaxation as an approximation technique
- To identify the structure and operation of Google's *PageRank* algorithm
- To analyse algorithm performance with respect to future load
- To implement *PageRank* in Python
- To investigate the integration of *PageRank* with the Web Crawler and Web Content Scraper introduced earlier

Contents

A4.1 Recursion

Recursion in computer science is a method where the solution to a problem is expressed in terms of a solution to a smaller version of the problem. In programming, it is implemented as defining a function that calls itself in the course of its execution.

Recursive functions typically consist of 2 sections

- a **base case** which defines the exit condition for the function
- a **recursive call** that defines the function in terms of itself

For example, examine the recursive implementation of the Fibonacci sequence generator presented below. The Fibonacci series begins 0, 1, … and continues with each successive number expressed as the sum of the previous 2 values.

File: A4/fibonacci.py

```
def fibonacci(n):
   if n <= 2:
      return n - 1
   else:
      return fibonacci(n - 1) + fibonacci(n - 2)

for i in range(1, 11):
   print(fibonacci(i))
```

Here, we express our base case as being the first 2 elements in the sequence (values 0 and 1). If the value of the parameter `n` is less than or equal to 2, then the base case is satisfied, and the function simply returns 0 or 1 as required. However, for values of `n` greater than 2, then the Fibonaccci element is defined as the sum of the previous 2 Fibonacci elements.

For example, consider a call for **`fibonacci(4)`** and trace the recursive calls made

fibonacci(4) = fibonacci(3) + fiboinacci(2)
= (fibonacci(2) + fibonacci(1)) + (fibonacci(1) + fibonacci(0))
= ((fibonacci(1)+ fibonacci(0))+ fibonacci(1)) + (1 + 0)
= 1 + 0 + 1 + 1 + 0
= 3

Do it now!	Run *fibonacci.py* and verify that the sequence illustrated in Figure A4.1 is generated.

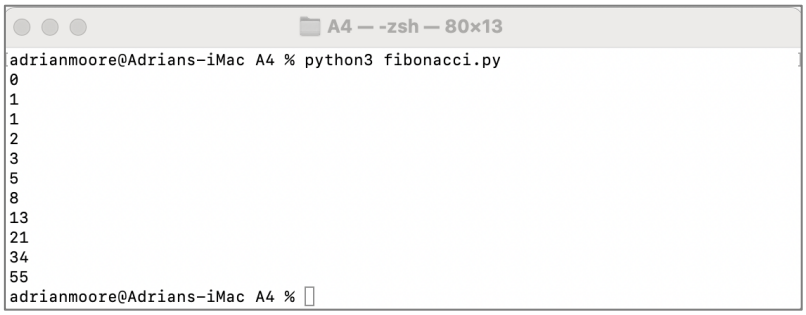

Figure A4.1 Recursive Fibonacci generation

Try it now!	Modify *fibonacci.py* so that it prints out the Fibonacci sequence on a single line (e.g. 0+1+1+2+3+5+8+…) as illustrated in Figure A4.2.

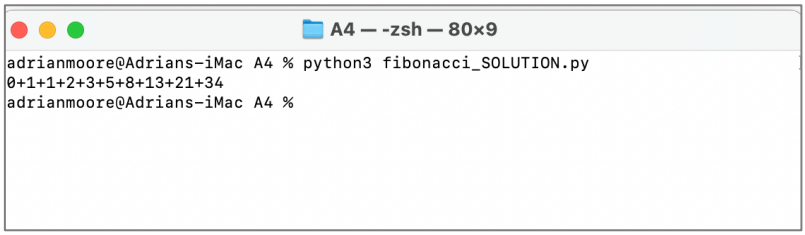

Figure A4.2 Fibonacci sequence

Another example commonly used to illustrate recursion is a test to check if a string provided as input is a palindrome. Again, this is an ideal candidate to be expressed recursively as follows:

Base cases:

- If the string is empty, then it is a palindrome
- If the string consists of a single letter, then it is a palindrome
- If the first and last letters of the string are different, then it is not a palindrome

Recursive Case:

- If the first and last letters of the string are the same, then it is a palindrome if the characters between the first and last letters represent a palindrome.

Try it now!	Write the script *palindrome.py* that implements a recursive function that takes a single string as a parameter and returns **`True`** if the parameter is a palindrome, and **`False`** otherwise. Test your function on a range of input.

A4.2 Unbounded Recursion

A4.2.1 Circular definitions

Sometimes a precise answer is impossible to compute and the best we can do is to make a guess and progressively refine it. Consider the following situation where we have a collection of people represented as a graph that reflects who follows who on a social network site.

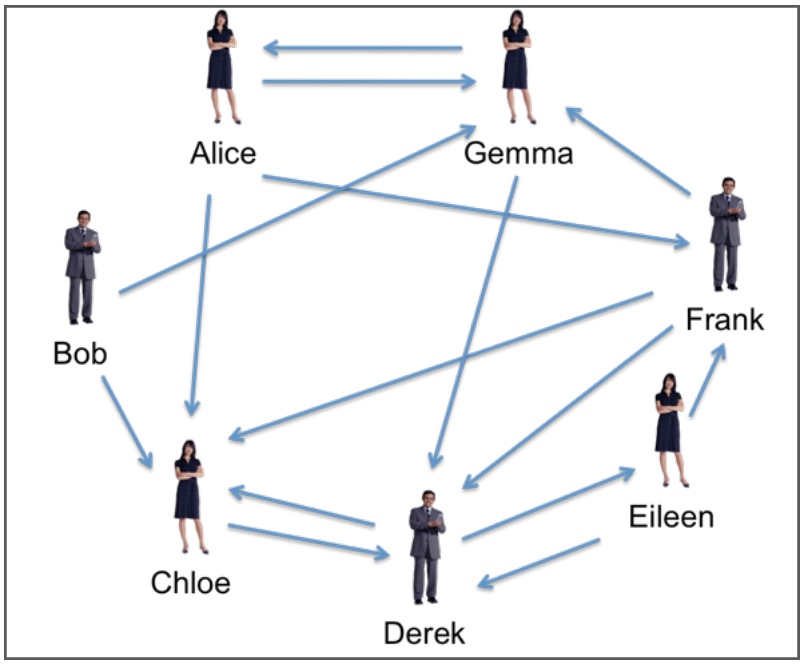

Figure A4.3. Network of friends

In this world, friendship is defined as a one-way link between a pair of individuals. For example, in the scenario depicted here, Alice is friends with Chloe, Frank and Gemma; while Bob is friends with Chloe and Gemma. This network can be represented in Python as a dictionary as shown in the following code fragment – with the person's name as the key and the list of their friends' names as the value.

File: A4/friends.py

```
friends = { 'Alice'  : ['Chloe','Frank','Gemma'],
            'Bob'    : ['Chloe','Gemma'],
            'Chloe'  : ['Derek'],
            'Derek'  : ['Chloe','Eileen'],
            'Eileen' : ['Derek','Frank'],
            'Frank'  : ['Chloe','Derek','Gemma'],
            'Gemma'  : ['Alice','Derek']
          }
```

From this network, determining the most popular individual seems to be a trivial task – simply find the person who has most incoming "friend" links.

Try it now!	Write the script *simple_friends.py* that implements a function that takes a friendship graph such as that shown above as a parameter, and returns a list containing the names of all those who have the most incoming 'friend' links. **Hint** – for the graph depicted above, the function should return the list `['Chloe', 'Derek']` as both are equally popular, with 4 incoming 'friend' links

This is a simple definition of popularity, but one that does not stand up to scrutiny, as it does not take into account the popularity of an individual's friends. For example, in the graph above, Alice and Eileen each have one incoming friend link, but Eileen's link is from the joint most popular person Derek, whereas Alice's link is from the slightly less popular Gemma. By association, Eileen could be said to be more popular than Alice, as she is connected to more people by virtue of her friendship with Derek.

This leads to a more robust definition of popularity as a value derived from the popularity of one's friends. In other words, we can define the popularity of an individual as the sum of the popularity of all of that individual's friends.

$$popularity(p) = \Sigma\ popularity(f)$$

where `f` is a friend of `p`

Therefore, the popularity of (for example) Alice, is defined as the popularity of Chloe plus the popularity of Frank plus the popularity of Gemma.

This definition is fine in theory, but turns out to have a significant limitation when we try to represent it in code. Consider the following worked example where we attempt to calculate the popularity of Alice

Step 1: (find popularity of Alice)
popularity(Alice) = popularity(Chloe) + popularity(Frank) + popularity(Gemma)

Step 2: (find popularity of Chloe)
popularity(Chloe) = popularity(Derek)

Step 3: Find popularity of Derek

popularity(Derek) = popularity(Chloe) + popularity(Eileen)

We now hit a serious problem, as our next step would be to calculate the popularity of Chloe, but this is already one of the unknown elements involved in our calculation for the popularity of Alice. Our program code would be trapped endlessly defining the popularity of Chloe and Derek as functions of each other and would never return a value. We have created a **circular definition** in which elements are defined in terms of each other.

A4.2.2 Relaxation

Relaxation methods are iterative techniques for solving problems with circular definitions, where we make an initial approximation of the answer and iteratively refine it by repeated calculation. Once a pre-determined number of iterations have been completed, we accept the current answer as being 'close enough' to the answer required.

The central technique is based around a recursive call to the calculation function that returns a definite value after a set number of iterations has been completed. The following pair of statements describe the algorithm for **`popularity(person, timestamp)`**, which defines the popularity of a person at time **`t`**

`popularity(person, 0) = 1`

`popularity(person, t) = Σ popularity(f, t-1)` – where `f` is a friend of `person`

In other words, the popularity of a person at time **`t`** is equal to the sum of their friend's popularities at time **`t-1`**.

The definition for **`popularity(person, 0)`** gives us a base case that allows the recursive algorithm to exit after a set number of attempts have been made.

File: A4/friends.py

```
friends = {'Alice'  : ['Chloe','Frank','Gemma'],
           ...
          }
LOOP_COUNT = 10

def popularity(friends, person, timestamp):
   if timestamp == 0:
      return 1
   else:
      pop = 0
      for f in friends[person]:
         pop = pop + popularity(friends, f, timestamp - 1)
      return pop

def get_popularity_values(friends):
   popularities = {}
   for friend in friends:
      pop_value = popularity(friends, friend, LOOP_COUNT)
      popularities[friend] = pop_value
   return popularities
```

Here we have a pair of functions **`popularity()`**, that recursively generates a popularity value for an individual, and **`get_popularity_values()`**, that builds a dictionary **`popularities`** containing the popularity rating of each person in the group.

Examine the code for **`popularity()`** and see how it matches the algorithm presented above – iterating over our definition for popularity and terminating once a pre-determined number of iterations have been completed.

The code for **`get_popularity_values()`** declares an empty **dictionary** called **`popularities`**. It then iterates across each entry in the **`friends`** dictionary, calling **`popularity()`** for that person, and creating an entry in the **`popularities`** dictionary that associates that person with their popularity figure.

Do it now!	Run *friends.py* in the Python interpreter and see how the popularity values presented in Figure A4.4 are generated. Carefully examine the structure of the code and ensure that you are comfortable with the algorithm before proceeding.

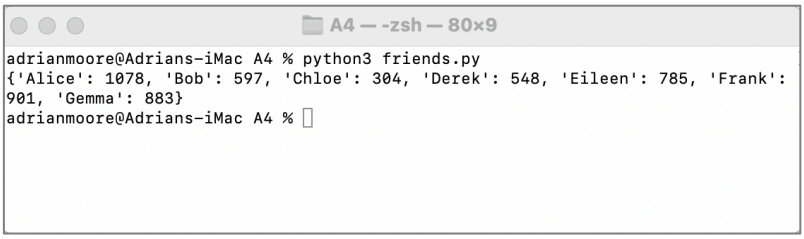

Figure A4.4. Relaxation technique

In the example presented above, we iterate 10 times before accepting the popularity value as an acceptable approximation, but what is the optimum number of iterations? A simple experiment can help determine this.

Try it now!	Write the Python application *friends_experiment.py* that repeats the relaxation algorithm for an increasing number of iterations and prints the results for each. For example, repeating for iterations ranging from 1-10 would generate the output illustrated in Figure A4.5 below.

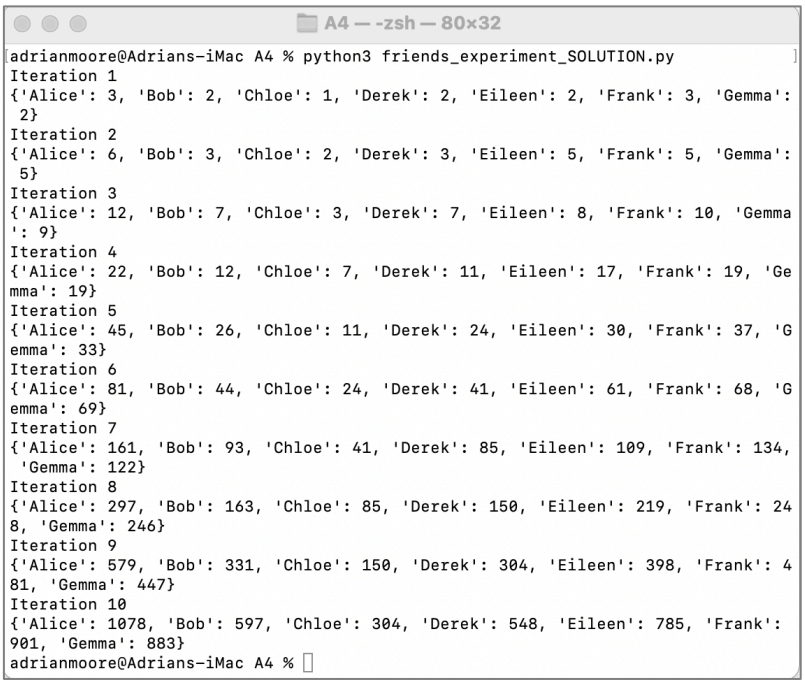

Figure A4.5. Popularity scores after various iterations

Here, we can see that after 5 iterations (the first iteration in which all of the individuals have a unique popularity rating), Alice is the most popular person, followed by Frank, Gemma, Eileen, Bob, Derek and finally Chloe. After each subsequent iteration, the ranking order stays the same. In general, the number of iterations required depends upon the size of the data set, but this simple experiment can be used to determine the optimal number in each case.

A4.3 Ranking Web Pages

A4.3.1 The *PageRank* algorithm

The friendship example presented above can also be used to obtain a comparative ranking for web pages. Where we have represented friendship as a directed graph, we can have the same relationship between one webpage that provides a hyperlink to another. When we make a request of a search engine and multiple pages contain the search term, the pages are presented to us in order of their "popularity" rating – calculated in much the same way as for our example above.

This ranking technique is known as ***PageRank*** and was developed and published by **Larry Page** and **Sergi Brin**, who developed the Google search engine around this algorithm. To apply the "relaxing popularity" technique to ranking web pages, we need to make 2 adjustments to the basic algorithm presented above.

(i) *PageRank* values are relative to the number of pages being indexed.

As can be seen by the previous experiment, as the number of iterations is increased, so the magnitude of the popularity values increases. With our network of 7 friends, 10 iterations result in values of 1000 or more. For a network of 45+ billion web pages, it is obvious that some form of moderation is required

Page and Brin's scaling technique is two-fold. First, the base case of the recursive function is modified so that instead of returning 1, it returns 1 divided by the number of pages being indexed. Secondly, they determine that a link from a page containing a small number of links should carry more importance than one from a page with many links. This is implemented by dividing a page's popularity score by the number of outward facing links on that page.

(ii) *PageRank* values should model the behaviour of a "random surfer"

One feature of the basic popularity index is that people with no incoming friendship links have a popularity index of zero. This might be acceptable in a "friends" context, but a web page with no incoming links should still be considered as the potential best match for a search query.

Page and Brin accomplish this by introducing a damping constant (**d**) that models the behaviour of a random web surfer. In this model, a web page has a **d** percent chance of

being visited as a result of a user clicking on a hyperlink, and a **(100-d)** percent change of being visited as a result of directly entering the URL in the address bar of the browser. Page and Brin suggest a value of **0.85** as the damping constant – representing an 85% chance that a page is visited as the result of clicking on a hyperlink on another page, and a 15% chance that its URL is directly entered in the browser's address bar.

A4.3.2 Implementing *PageRank*

The basic algorithm for *PageRank* is similar to that for the popularity index, and is described by the following pair of rules

rank(url, 0) = 1/nPages
where **nPages** is the number of pages being indexed

rank(url, t) = ((1-d)/nPages) + Σ d * rank(t-1, page) / numOutLinks

where **d** is the damping constant,
page is each webpage that links to **url**,
and **numOutLinks** is the number of hyperlinks on **page**

In other words, the base case (or default value) is 1 divided by the number of pages in the index (i.e. the change of randomly visiting this page), and the rank is progressively improved by combining the chance that this page was directly visited **[(1-d)/nPages]** with the sum of the ranks of pages that link to this one – each divided by the number of links on the page.

rank (url, t) = ((1-d)/nPages) + Σ d * rank(t-1, page) / numOutLinks

Chance of directly entering the URL

Sum of PageRanks of pages that link to this URL, multiplied by the chance of clicking on a link to the page, and divided by the number of links available

One final simplification recognises that we only need to keep track of the *current* and *previous* values for the ranks, so we can replace the **`rank(t)`** and **`rank(t-1)`** terms above by **`new_ranks`** and **`ranks`**, respectively – as long as we update **`ranks`** with the value of **`new_ranks`** after each iteration.

This gives our final equation:

$$\mathbf{newRanks[url] = (1-d)/nPages + \Sigma\ d * ranks[page] / numOutlinks}$$

where **`d`** is the damping constant, **`page`** is each webpage that links to **`url`**, and **`numOutLinks`** is the number of hyperlinks on **`page`**

The full *PageRank* algorithm can be expressed by a Python function that takes as a parameter a graph representing a set of web pages and the links between them. The graph is organised as a dictionary where the keys are URLs and the value for each is a list of URLs that are linked to by the key (i.e. the set of outward facing hyperlinks in the key).

For example, if our web site consists of 3 pages linked as illustrated in Figure A4.6,

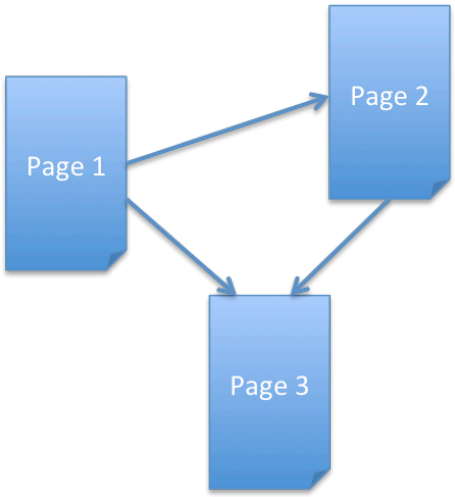

Figure A4.6. Small sample website

we would represent the graph by the Python declaration.

```
graph = { 'page1.html' : ['page2.html', 'page3.html'],
          'page2.html' : ['page3.html'],
          'page3.html' : []
        }
```

We can now demonstrate this by presenting a Python application (overleaf) that specifies an input graph representing the organization of a website and generates the PageRank values. The code begins by setting each page's *PageRank* value to the default of 1 divided by the number of pages. Next, the main loop iterates over the required number of iterations, each time calculating the *PageRank* value as a function of the value in the previous iteration of the loop. Following each iteration, the current value of **ranks** is updated to the value of **newRanks** in preparation for the next time through the loop. Figure A4.7 shows the final PageRank values generated for our small website of 3 linked pages.

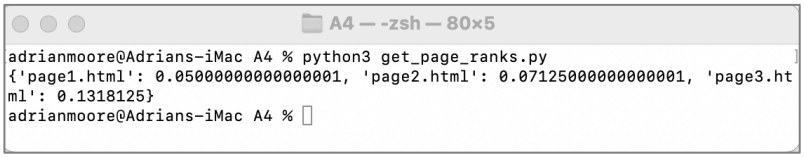

Figure A4.7 Calculate PageRank Values

File: A4/get_page_ranks.py

```
def compute_ranks(graph):
   d = 0.85
   num_loops = 10
   ranks = {}
   npages = len(graph)

   for page in graph:
      ranks[page] = 1 / npages

   for i in range(0, num_loops):
      new_ranks = {}
      for page in graph:
         new_rank = (1 - d) / npages
         for node in graph:
            if page in graph[node]:
               new_rank = new_rank + d * (ranks[node] /
                                          len(graph[node]))
         new_ranks[page] = new_rank
      ranks = new_ranks
   return ranks

graph = { 'page1.html' : ['page2.html', 'page3.html'],
          'page2.html' : ['page3.html'],
          'page3.html' : []
        }
ranks = compute_ranks(graph)
print(ranks)
```

Do it now!	Run *get_page_ranks.py* in the Python interpreter and see how the PageRank values presented in Figure A4.7 are generated. Try changing the structure of the website by adding additional pages and links to the graph to see how your changes affect the PageRank values obtained.

A4.3.3 Incorporating *PageRank* into *POODLE*

The **compute_ranks()** function stands alone within the architecture of the search engine – as long as we can provide it with the graph that is required as input. As far as our ***POODLE*** architecture is concerned, we need to modify our Web Crawler to return the graph of links as well as the list of crawled pages. ***This can be easily achieved and is left for you as an exercise***.

Note:	The graph can actually be returned **instead of** the list of crawled pages as the keys of the graph are the elements of the list.

Try it now!	Revisit your Web Crawler application from Practical A3 and modify it so that it generates a graph representing the organization of the web pages visited as well as the list of pages.

We have now implemented all of the components of a search engine:

- A ***Web Crawler*** that takes a seed URL and returns a list of pages visited and a graph (organised as a dictionary) that stores all of the URLs linked from each page visited.
- A ***Web Scraper*** that returns all of the content keywords found at a URL and (in conjunction with the Web Crawler) builds an index (organised as a dictionary) that associates keywords with the URLs of the pages at which they are found.
- A ***PageRank*** value generator for each page visited.

The search algorithm accepts a keyword from the user and uses it as a lookup into the index returned by the ***Web Crawler*** and ***Web Scraper***. From the list of URLs where the keyword is located, the URL with the greatest ***PageRank*** value is presented to the user.

Try it now! **ADVANCED CHALLENGE**	Combine the components you have implemented during Section A of the module to build *poodle.py*, a small web indexing and search engine. In particular, you will need the following: (i) The modified **Web Crawler** from the "Try It Now!" exercise above that generates the graph of web pages. (ii) The enhanced **Web Scraper** from Practical A3 that visits the pages listed in the graph and builds the **Index** of content, structured as a **dictionary**. (iii) The **`compute_ranks()`** function from this practical to generate the **PageRank** values for each page in the graph of web pages (iv) Your ***basic_search.py*** solution from Practical A3 that prompts the user for a search term and returns the list of pages on which the term is found. Extend this application so that the PageRank of each URL is also displayed. You should use the small web site at http://adrianmoore.net/com661/test_index.html as the seed for the Web Crawler.

A4.4 Further Information

- http://en.wikipedia.org/wiki/Recursion
  Wikipedia definition of recursion

- http://stackoverflow.com/questions/937000/python-recursion-and-return-statements
  Python recursion and return statements

- http://www.openbookproject.net/thinkcs/python/english2e/ch11.html
  Recursion and exceptions – How to think like a computer scientist

- https://en.wikipedia.org/wiki/Relaxation_(approximation)
  Relaxation algorithms

- http://en.wikipedia.org/wiki/PageRank
  PageRank definition from Wikipedia

- http://searchengineland.com/what-is-google-pagerank-a-guide-for-searchers-webmasters-11068
  What is Google PageRank? – a guide for searchers and webmasters

- http://infolab.stanford.edu/~backrub/google.html
  The Anatomy of a Large-Scale Hypertextual Web Search Engine – Page & Brin's original description of the Google technique

- http://pr.efactory.de/e-pagerank-algorithm.shtml
  The PageRank algorithm

- http://www.python.org/doc/essays/graphs/
  Implementing graphs in Python