Algorithm Design

Aims

Understand the principles of algorithm design with examples

Understand how to merge data from 2 files

Understand how to search a list

Understand recursion and its application to lists and to string reversal

Merging Data

Query File

205607_s_at
41329_at
220840_s_at
208438_s_at
213800_at
215388_s_at
210267_at
214579_at
205996_s_at
208967_s_at
203417_at
212175_s_at
209839_at
218223_s_at
203925_at
220295_x_at
212101_at
202194_at
212102_s_at
212103_at
219159_s_at
215776_at
212147_at
…

Data File (could contain 000's of entries)

208644_at	ENSG00000143799
221921_s_at	ENSG00000162706
215388_s_at	ENSG00000244414
208270_s_at	ENSG00000176393
206446_s_at	ENSG00000142615
202194_at	ENSG00000117500
204418_x_at	ENSG00000213366
208500_x_at	ENSG00000187140
211050_x_at	ENSG00000240618
215566_x_at	ENSG00000011009
221831_at	ENSG00000169641
201235_s_at	ENSG00000159388
203872_at	ENSG00000143632
203561_at	ENSG00000244682
215728_s_at	ENSG00000097021
217365_at	ENSG00000232423
215388_s_at	ENSG00000000971
200910_at	ENSG00000163468
209839_at	ENSG00000197959
218917_s_at	ENSG00000117713
203417_at	ENSG00000117122
212147_at	ENSG00000117122
210923_at	ENSG00000162383
…	

Parsing Files – Possible Solution 1

```
out_file = open("id_matches.txt", "w")

with open("affy_ids.txt") as file:

    for id in file:

        with open("affy_genes.txt") as file2:

            for line in file2:

                gene_line = line.split()

                if id.rstrip() == gene_line[0]:

                    out_file.write(line)
```

Solution 1 - Problems

File “affy_genes.txt” has to be opened for each affy ID in the "affy_ids.txt“

This could require the file to be opened and closed thousands of times

The “affy_genes.txt” file itself could contain hundreds of thousands of lines

Affy ids may be associated with more than one gene

Genes can also match more than one affy ID

So can’t exit “affy_genes.txt” file as soon as first match made

Have to parse entire file each time

Solution 2

```
ids = {}

with open("affy_ids.txt") as file:
        for line in file:
                ids[line.rstrip()] = 1

out_file = open("id_matches.txt", "w")

with open("affy_genes.txt") as file:
        for line in file:
                list = line.split()
                if list[0] in ids:
                        out_file.write(line)
```

Both files are now only opened and parsed once

Searching a List

Consider a function that searches a list for a particular entry and returns the index position, if it is present

```
def search(num, vals):
    # Variable num is the entry to be searched for
    # and values is a list of values to be searched
    # The function will return the index position
```

Example use:

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

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

Searching Function

```
def search(num, vals):

    for i in range(len(vals)):
        if vals[i] == num:  # item found, return the index value
            return i

    return -1          # loop finished, item was not in list
```

Linear Search

The Python “in” and “index” functions, as well as the “search” function all use linear searches

This means the function starts at index position 0 and works through the list until a match is found

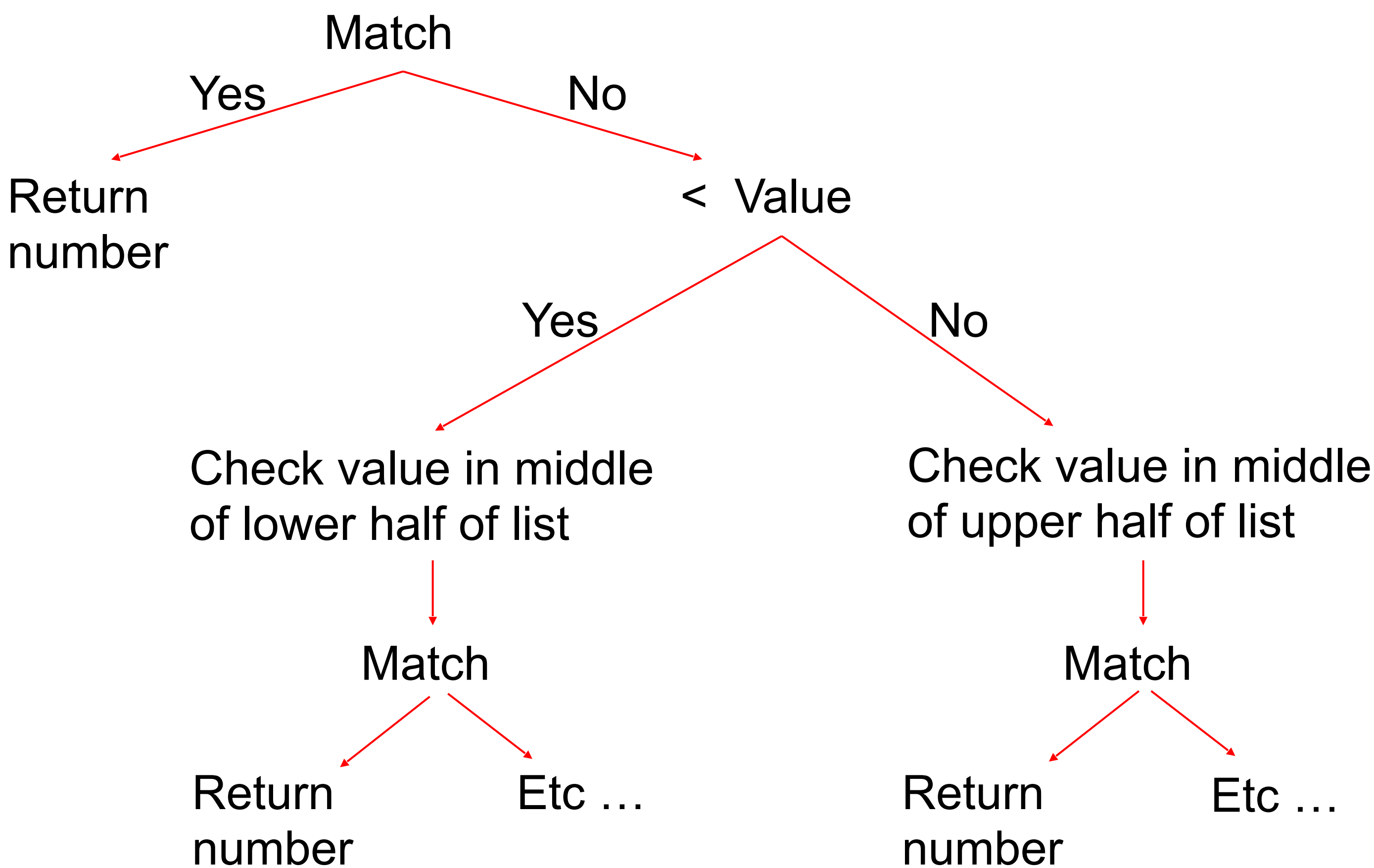
In most instances this search strategy is adequate but with very large lists the efficiency starts to deteriorate

A better option for large lists would be a binary search

Binary Search

Requires an ordered list

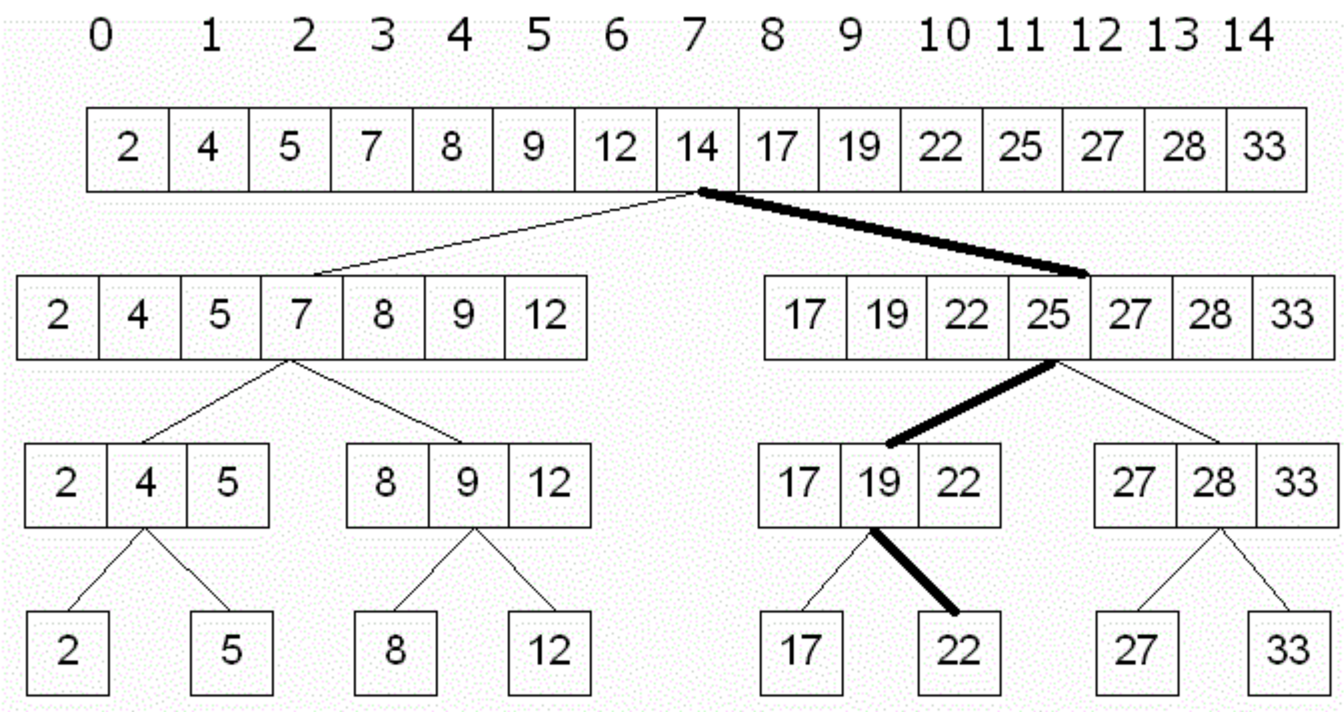
Strategy is to start by checking the value in the middle of the list:



Binary Search Example

List = [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33)

Search for ‘22’:



Maximum 4 steps to search for an entry

Python Binary Search Algorithm

Need two variables to keep track of position in the list – the low and high points

Initially low is set to the first index position and high to the last

Compare search value with value in the middle of the list

If lower then set high point to middle index position

If higher set low point to middle index

Repeat until value is found or there are no more indices to search

Python Binary Search Algorithm

```
def search(num, vals):
    low = 0
    high = len(vals) – 1

    while low <= high:                      # There is a range to search
        mid = (low + high) / 2              # Position of middle item
        item = vals[mid]
        if num == item:                     # Found it! Return the index
            return mid
        elif num < item:                    # x is in lower half of range
            high = mid - 1                  #  move top marker down
        else:                               # x is in upper half of range
            low = mid + 1                   #  move bottom marker up
    return -1                               # No range left to search,
                                            # x is not there
```

Python Binary Search Efficiency

Binary search is far more efficient than the linear search but do need to sort list first

Once sorted the best case performance is one comparison and the worst-case (value not in list) is $\log_2 N$, where N is the size of the array

List Size	Linear (N)	Binary ($\mathrm{Log}_2 N$)
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Recursion

Recursion simply means applying a function as a part of the definition of that same function.

Essentially the function calls itself.

The key to making it work is that there must be a terminating condition such that the function branches to a non-recursive solution at some point.

Consider a function to calculate a factorial:

$$1! = 1$$
$$2! = 1 \times 2 = 2$$
$$3! = 1 \times 2 \times 3 = 2! \times 3 = 6$$
$$N! = 1 \times 2 \times 3 \times .... (N-2) \times (N-1) \times N = (N-1)! \times N$$

Factorial Function

```
def factorial(n):
        if n == 1:
                return 1
        else:
                return n * factorial(n-1)
```

Now because we decrement N each time and we test for N equal to 1 the function must complete.

There is a small bug in this definition however, if you try to call it with a number less than 1 it goes into an infinite loop! To fix that change the test to use "<=" instead of "==".

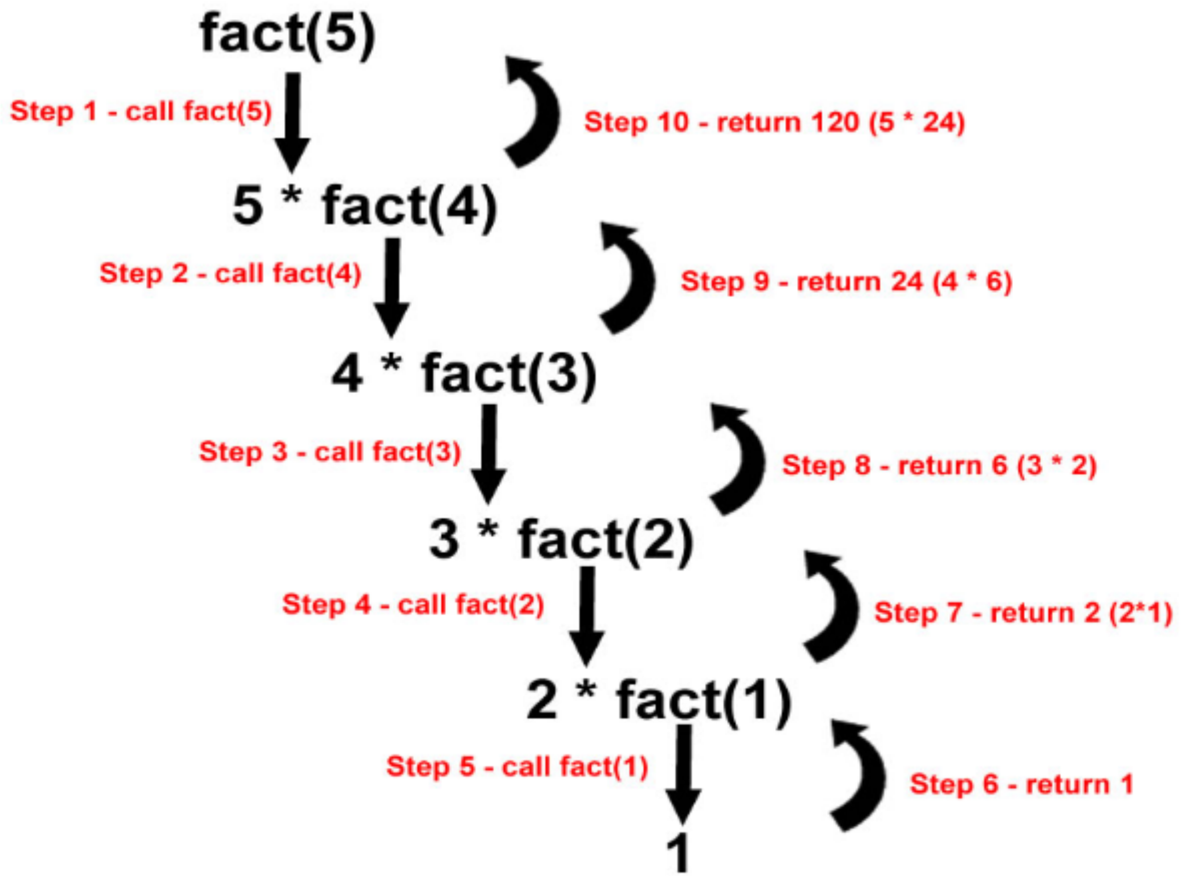
This goes to show how easy it is to make mistakes with terminating conditions, this is probably the single most common cause of bugs in recursive functions.

Make sure you test all the values around your terminating point to ensure correct operation.

Factorial Function

How does the factorial function work?

The return statement returns n * (the result of the next factorial call) so the code is effectively doing:



Reversing a String

Have already seen one method for this:

```
s = “abcde”
s = s[::-1]
```

Strings can also be reversed with a recursive function:

```
def reverse(s):
        if s == "":
                return s
        else:
                return reverse(s[1:]) + s[0]
```

Recursion and Lists

Can use recursion to print a list

```
def printList(L):
        if L:
                print (L[0])      # Print the first item in the list
        printList(L[1:])          # Uses slice to call printList on the
                                  # remainder of the list

nums = (1, 2, 3, 4, 5)
printList(nums)
```

Outputs:

```
1
2
3
4
5
```

Recursion and Lists

Recursion is of more use if printing a ***list*** that contains one or more ***lists***.

This function uses the Python ***type()*** function to determine if an item is a ***list***:

```
def printList(L):
        # if its empty do nothing
        if not L:
                return

        # if it's a list call printList on 1st element
        if type(L[0]) == type([]):
                printList(L[0])
        else: # no list so just print
                print (L[0])

        # now process the rest of L
        printList(L[1:])
```

Towers of Hanoi

There is an old legend about a temple with three poles, one of them filled with 64 gold disks.

The disks are of different sizes and they are put on top of each other such that each disk on the pole a little smaller than the one beneath it.

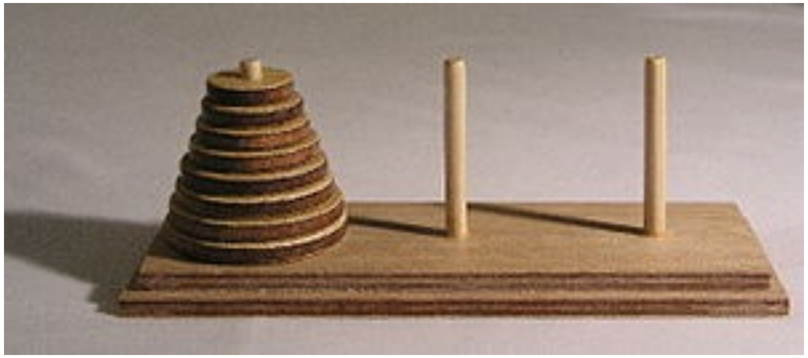
The priests have to move this stack from one of the three poles to another one, but a large disk can never be placed on top of a smaller one.

The legend says that when they have finished their work the temple would crumble into dust, and the world would end.

However, it would require $2^{64} - 1$ moves (18,446,744,073,709,551,615) to complete the task.

The legend and the game of towers of Hanoi was actually created in 1883 by the French mathematician Edouard Lucas.

Towers of Hanoi



Why Towers of Hanoi?

The factorial solution is the “hello word” example of a recursive function

Along with the list examples, they all demonstrate recursion but can also be solved using iterative code, even if often less efficiently

The towers of Hanoi is a problem that cannot easily be solved iteratively but can be solved with a relatively simple recursive function

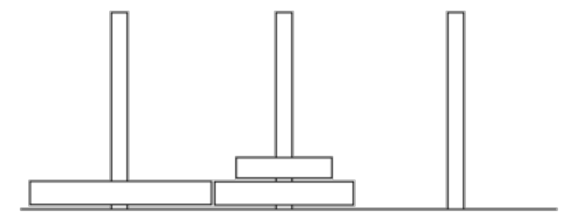
To solve any problem recursively you need to first define the rules for solving the problem

Towers of Hanoi Rules

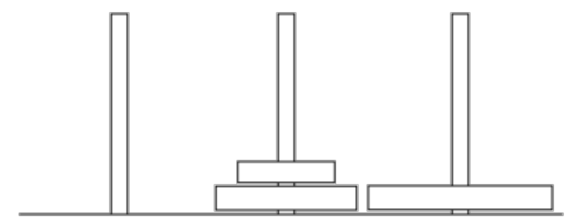
The 3 pegs can be labelled “start”, “middle” and “end”.

The rules for the solution are then:

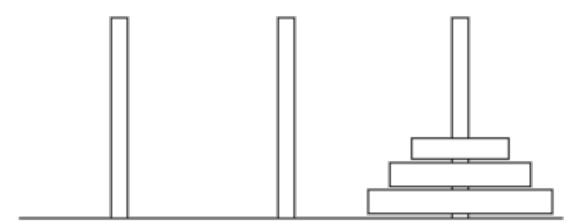
1: Move N-1 disks from the start peg to the middle peg



2: Move the Nth (largest) disk from the start peg to the end peg



3: Move N-1 disks from the middle peg to the final peg



Rule 3 calls rules 1 and 2 and the pegs change designation accordingly.
This is followed recursively until the disks are all moved to the end peg.

Towers of Hanoi Solution

```
def hanoi(disk, start, middle, end):

        if disk > 0:
                # 1: Move N-1 disks from the start pole to the middle peg

                hanoi(disk - 1, start, end, middle)

                print('Move disk' + str(disk) + ' from ' + start + ' to ' + end)

                hanoi(disk - 1, middle, start, end)


# Call function with specified number of starting disks
hanoi(3, “Start”, “Middle”, “End”)
```

Towers of Hanoi Solution 1 Output

Move disk1 from Start to End

Move disk2 from Start to Middle

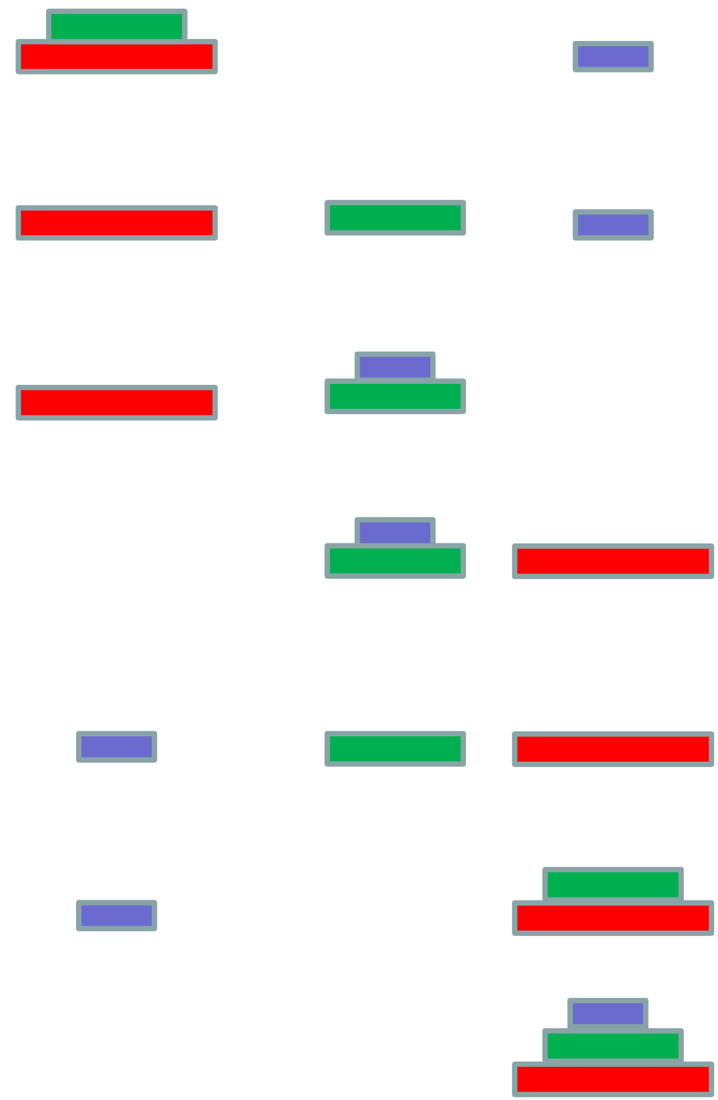
Move disk1 from End to Middle

Move disk3 from Start to End

Move disk1 from Middle to Start

Move disk2 from Middle to End

Move disk1 from Start to End



Code Explanation

```
# Set the terminating condition. No disks left so function just returns and
# cascades back

if disk > 0:

        # 1: Move N-1 disks from the start pole to the middle peg
        # The middle peg is now the target (end) and the end becomes the
        # intermediary (middle)

        hanoi(disk - 1, start, end, middle)

        # 2: Move the Nth (largest) disk from the start pole to the end peg

        print('Move disk' + str(disk) + ' from ' + start + ' to ' + end)

        # 3: Move N-1 disks from the middle peg to the final pole
        # The middle is now the starting position (start) and the start is the
        # intermediary (middle)
        hanoi(disk - 1, middle, start, end)
```

Exponentiation

The iterative way to calculate a^n for any integer n is to multiply a by itself n times

This can be achieved with a simple loop:

```
def calcPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans


print (calcPower(2, 8))
```

This method requires n calculations, in this case 8.

Recursive Exponentiation

The basis of solving the problem using recursion is the principle of divide and conquer

The laws of exponents mean that $2^8 = 2^4(2^4)$

Therefore if 2^4 is known then 2^8 just needs one calculation multiplication.

Furthermore, $2^4 = 2^2(2^2)$ and $2^2 = 2(2)$

$$2(2) = 4,\ 4(4) = 16,\ 16(16) = 256 = 2^8$$

The result is that 2^8 can be calculated with only 3 multiplications, 2^{16} in 4 etc

For an even number $a^n = a^{n//2}(a^{n//2})$

For an odd number $a^n = a^{n//2}(a^{n//2})\ (a)$

Recursive Exponentiation Solution

```
def calcPower(a, n):

        # Set the terminating condition.
        if n ==  0:
                return 1

        else:
                factor = calcPower(a, n//2)

                if n%2 == 0:    # Even
                        return factor * factor

                else:       # Odd

                        return factor * factor * a


print (calcPower(2, 8))
```

Recursion

There are similarities between looping and recursion

Anything that can be done with a loop can also be done with a recursive function

Some problems that are difficult to solve with loops (e.g. tree traversal) are relatively simple to solve with recursion

Be aware when recursion can be used as it may improve the efficiency of your code

However, a loop may still be the best solution

Conclusions

Understanding how to program is just the first stage in writing code

Algorithm design is the core component of program design

When designing code consider the efficiency of the code

Remember that generally anything that can be done with a loop can be done with a simple recursive function

Unless the code is only going to be a short script then design it first