

COMP1001

Week 11

What is recursion? What makes a function recursive?

What is recursion? What makes a function recursive?

- A recursive function is able to call itself

What is recursion? What makes a function recursive?

- A recursive function is able to call itself
- Rather than iterating with a loop, a recursive function usually calls itself with a smaller or broken-down version of the input until it reaches an “answer”

What are the two key parts of a recursive function?

What are the two key parts of a recursive function?

- Recursive case
- Base case

What are the two key parts of a recursive function?

- Recursive case:
 - the function calls itself, with a different (smaller/broken-down) input
- Base case:
 - the function has reached the smallest/simplest version of the problem and stops recursing
 - (doesn't call itself further)

In what cases is recursion useful? Where should it be used with caution?

In what cases is recursion useful? Where should it be used with caution?

- Useful when an iterative solution requires nesting of loops proportional to the size of the input

In what cases is recursion useful? Where should it be used with caution?

- Useful when an iterative solution requires nesting of loops proportional to the size of the input
- Otherwise there's usually an equally elegant iterative solution!
 - Function calls are computationally expensive so we would select the iterative function in this case

Here's a function that finds the nth Fibonacci number:

```
def fib(n):  
    if n in [0,1]:  
        return n  
    return fib(n-1) + fib(n-2)
```

Visualise it!

<https://recursion.vercel.app/>

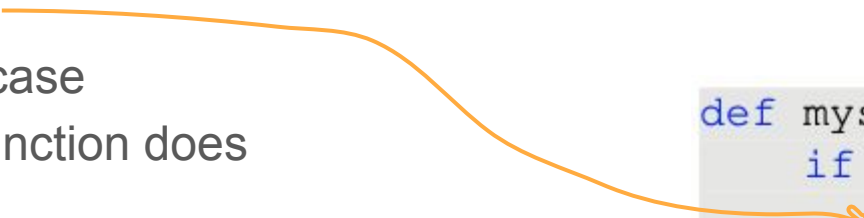
Identify the parts of the function!

- Base case
- Recursive case
- What the function does

```
def mystery(x):  
    if len(x) == 1:  
        return x[0]  
    else:  
        y = mystery(x[1:])  
        if x[0] > y:  
            return x[0]  
        else:  
            return y
```

Identify the parts of the function!

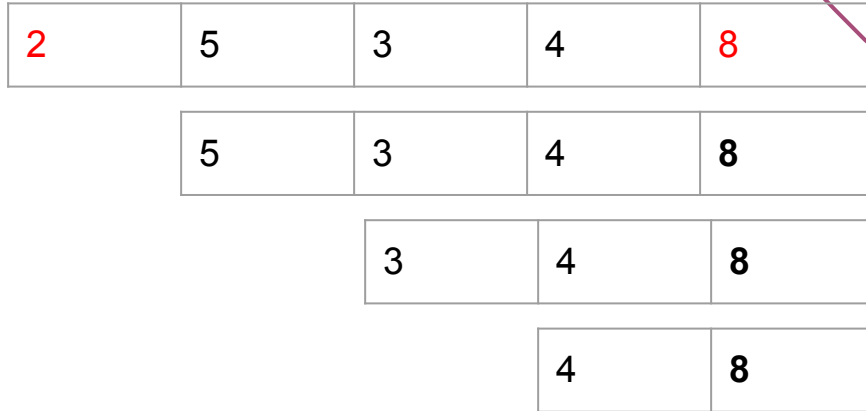
- Base case
- Recursive case
- What the function does



```
def mystery(x):  
    if len(x) == 1:  
        return x[0]  
    else:  
        y = mystery(x[1:])  
        if x[0] > y:  
            return x[0]  
        else:  
            return y
```

Identify the parts of the function!

- Base case
- Recursive case
- What the function does

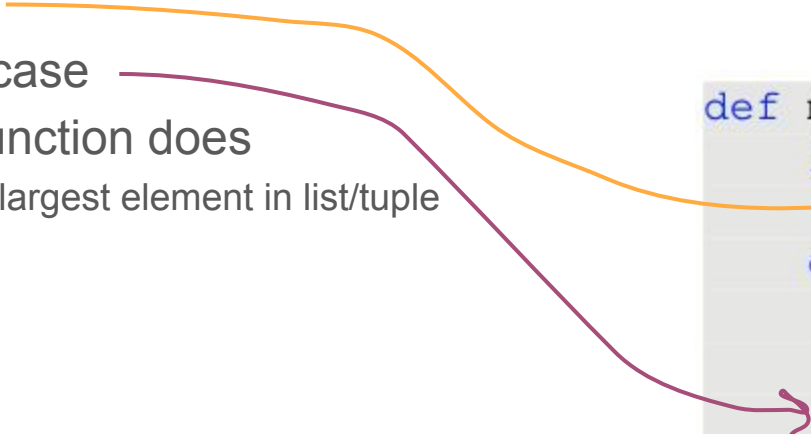


```
def mystery(x):  
    if len(x) == 1:  
        return x[0]  
    else:  
        y = mystery(x[1:])  
        if x[0] > y:  
            return x[0]  
        else:  
            return y
```

Identify the parts of the function!

- Base case
- Recursive case
- What the function does
 - Returns largest element in list/tuple

```
def mystery(x):  
    if len(x) == 1:  
        return x[0]  
    else:  
        y = mystery(x[1:])  
        if x[0] > y:  
            return x[0]  
        else:  
            return y
```



Identify the parts of the function!


- Base case
- Recursive case
- What the function does

```
def mistero(x):  
    a = len(x)  
    if a == 1:  
        return x[0]  
    else:  
        y = mistero(x[a//2:])  
        z = mistero(x[:a//2])  
        if z > y:  
            return z  
        else:  
            return y
```


Identify the parts of the function!

- Base case
- Recursive case
- What the function does

```
def mistero(x):  
    a = len(x)  
    if a == 1:  
        return x[0]  
    else:  
        y = mistero(x[a//2:])  
        z = mistero(x[:a//2])  
        if z > y:  
            return z  
        else:  
            return y
```



Identify the parts of the function!

- Base case
- Recursive case
- What the function does

2	5	3	4	8
---	---	---	---	---

2	5
---	---

3	4	8
---	---	---

4	8
---	---

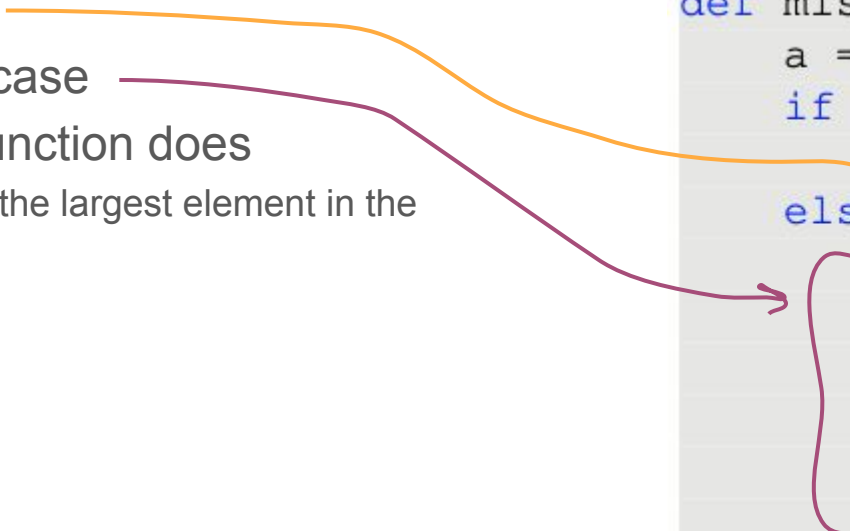
4	8
---	---

```
def mistero(x):  
    a = len(x)  
    if a == 1:  
        return x[0]  
    else:  
        y = mistero(x[a//2:])  
        z = mistero(x[:a//2])  
        if z > y:  
            return z  
        else:  
            return y
```

Identify the parts of the function!

- Base case
- Recursive case
- What the function does
 - Returns the largest element in the list/tuple

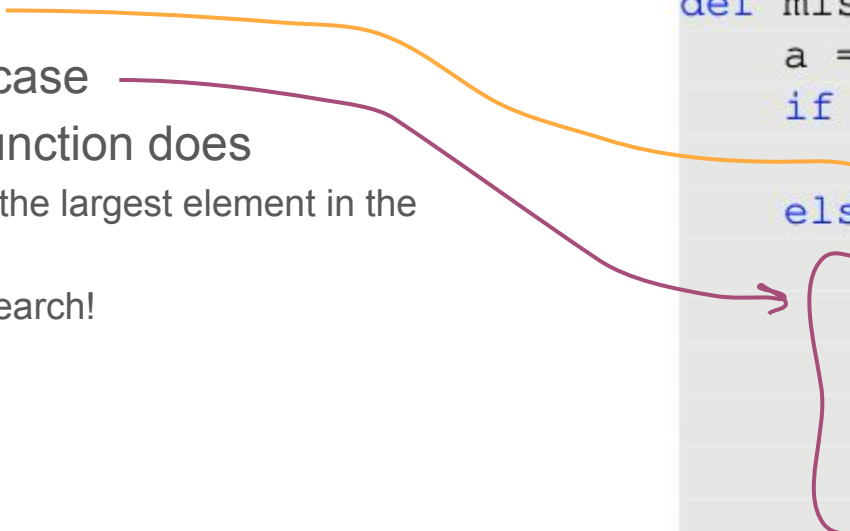
```
def mistero(x):  
    a = len(x)  
    if a == 1:  
        return x[0]  
    else:  
        y = mistero(x[a//2:])  
        z = mistero(x[:a//2])  
        if z > y:  
            return z  
        else:  
            return y
```



Identify the parts of the function!

- Base case
- Recursive case
- What the function does
 - Returns the largest element in the list/tuple
 - Binary search!

```
def mistero(x):  
    a = len(x)  
    if a == 1:  
        return x[0]  
    else:  
        y = mistero(x[a//2:])  
        z = mistero(x[:a//2])  
        if z > y:  
            return z  
        else:  
            return y
```



Fill in the blanks!

Is it possible to make some amount from a given selection of coin values, possibly using multiple coins of the same value?

```
def can_make_change(amount, coins):  
    # base case: success  
      
    return True  
  
    # base case: failure  
    if amount < 0 or len(coins) == 0:  
          
  
    # recursive case: handle two possibilities, either:  
    # 1. another of this coin value gets used, or  
    # 2. we don't need another coin of this value  
    coin = coins[-1]  
    return (can_make_change(, coins)  
            or can_make_change(amount, coins[:-1]))
```

Fill in the blanks!

Is it possible to make some amount from a given selection of coin values, possibly using multiple coins of the same value?

E.g.1 If my amount is 23 and I have coins with values [3,6,8] then I can make 23: 3+6+6+8

```
def can_make_change(amount, coins):  
    # base case: success  
      
    return True  
  
    # base case: failure  
    if amount < 0 or len(coins) == 0:  
          
  
    # recursive case: handle two possibilities, either:  
    # 1. another of this coin value gets used, or  
    # 2. we don't need another coin of this value  
    coin = coins[-1]  
    return (can_make_change(, coins)  
            or can_make_change(amount, coins[:-1]))
```

Fill in the blanks!

Is it possible to make some amount from a given selection of coin values, possibly using multiple coins of the same value?

E.g.1 If my amount is 23 and I have coins with values [3,6,8] then I can make 23: 3+6+6+8

E.g.2 If my amount is 11 and I have coins with values [2,8,6] then I can't make 11!

```
def can_make_change(amount, coins):  
    # base case: success  
      
    return True  
  
    # base case: failure  
    if amount < 0 or len(coins) == 0:  
          
  
    # recursive case: handle two possibilities, either:  
    # 1. another of this coin value gets used, or  
    # 2. we don't need another coin of this value  
    coin = coins[-1]  
    return (can_make_change(, coins)  
            or can_make_change(amount, coins[:-1]))
```

Fill in the blanks!

Is it possible to make some amount from a given selection of coin values, possibly using multiple coins of the same value?

E.g.1 If my amount is 23 and I have coins with values [3,6,8] then I can make 23: 3+6+6+8

E.g.2 If my amount is 11 and I have coins with values [2,8,6] then I can't make 11!

```
def can_make_change(amount, coins):  
    # base case: success  
    if amount == 0:  
        return True  
  
    # base case: failure  
    if amount < 0 or len(coins) == 0:  
        return False  
  
    # recursive case: handle two possibilities, either:  
    # 1. another of this coin value gets used, or  
    # 2. we don't need another coin of this value  
    coin = coins[-1]  
    return (can_make_change(amount - coin, coins)  
            or can_make_change(amount, coins[:-1]))
```


Practice programming

When handling csv files, there are a couple of ways we can get the data out of the csv file and into our program: `csv.reader` and `csv.DictReader`. You'll have used `csv.reader` in the Grok worksheets, so this time try to use `csv.DictReader`.

Write a function `count_sales(csv_filename)`, that takes a string `csv_filename`, and returns a dictionary that counts the frequency of products sold. On the example file shown below, it should return:

```
{'Toy Car': 2, 'Comic Book': 1}
```

```
Date,Product,Customer
2024-03-21,Toy Car,Bluey
2024-04-12,Comic Book,Bingo
2024-05-07,Toy Car,Rusty
```

```
import csv

PRODUCT = "Product"
READ_MODE = "r"

def count_sales(csv_filename):
    product_count = {}

    with open(csv_filename, READ_MODE) as file:

        for row in csv.DictReader(file):
            product = row[PRODUCT]

            if product in product_count:
                product_count[product] += 1
            else:
                product_count[product] = 1

    return product_count
```

```
import random
```

```
# come up with a large example ...
```

```
all_subjects = ["A", "B", "C", "D", "E"]
```

```
all_zbinis = [(random.randint(0,255), random.sample(all_subjects,  
random.randint(1,3))) for i in range(1000)]
```

Challenge: You've found a secret message:

```
secret_message.txt
```

```
erkbvl ur kbvd tlmexr:  
gxoxk zhggz zbox rhn ni  
gxoxk zhggz exm rhn whpg  
gxoxk zhggz kng tkhngw tgw wxlxkm rhn  
gxoxk zhggz ftdx rhn vkr  
gxoxk zhggz ltr zhhwurx  
gxoxk zhggz mxee t ebx tgw ankm rhn
```

All that you know about the message is that it is encrypted by a basic shift cipher (also known as a Caesar cipher, where each letter is shifted by some constant number of places in the alphabet), any alphabetic character in the message is lowercase, and that it contains the string segment `'desert'`.

Write a function to decrypt the message that takes an `infilename`, `outfilename` and `segment` (all strings, and you can assume that all files exist). You can use a brute-force approach (try all possible values) to guess the number to shift by. You might find the functions `ord(character)` and `chr(number)` useful!

What is an “algorithm”? Why are algorithms a large area of Computer Science

What is an “algorithm”? Why are algorithms a large area of Computer Science

- Algorithm: sequence of steps for solving an instance of a particular problem type

What is an “algorithm”? Why are algorithms a large area of Computer Science

- Algorithm: sequence of steps for solving an instance of a particular problem type
- We have been solving problems with algorithms all semester!
- As we deal with more and more data, the efficiency of our algorithms becomes increasingly important
- Think of “programming” as “literacy” and “algorithms” as “literature”
 - Programming: learning rules of grammar and structure so a computer can understand what we intend to communicate
 - Algorithms: learning good code, elegant ways of solving problems and how to use more advanced vocabulary!

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

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

- Correctness
- Efficiency

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

- Correctness
 - Does the algorithm give the correct output?
- Efficiency
 - How “good” is the algorithm?
 - Speed, storage, processing power, etc.
 - We haven’t looked at this in COMP10001! But it’s important to think about as you learn more about algorithms

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

- Correctness
 - Does the algorithm give the correct output?
- Efficiency
 - How “good” is the algorithm?
 - Speed, storage, processing power, etc.
 - We haven’t looked at this in COMP10001! But it’s important to think about as you learn more about algorithms
- Which would you prefer? An algorithm guaranteed to calculate the correct answer that takes 150 years to finish, or an algorithm that takes seconds but may not always produce the correct result

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

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

- Exact approach: gives correct solution
- Approximate approach: gives almost the correct solution, through estimation, simulation, etc.

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

- Exact approach: gives correct solution
- Approximate approach: gives almost the correct solution, through estimation, simulation, etc.
- If a problem is too complex to calculate with full completeness, an approximate approach might be more useful!

Are these algorithm types exact or approximate?

- Brute force (generate and test)
- Heuristic search
- Simulation
- Divide and conquer

Are these algorithm types exact or approximate?

- Brute force (generate and test)
- Heuristic search
- Simulation
- Divide and conquer

Are these algorithm types exact or approximate?

- Brute force (generate and test)
 - Exact. Finds every possible answer and tests it. Requires set of possible answers to be finite to guarantee completion. E.g. linear search
- Heuristic search
- Simulation
- Divide and conquer

Are these algorithm types exact or approximate?

- Brute force (generate and test)
 - Exact. Finds every possible answer and tests it. Requires set of possible answers to be finite to guarantee completion. E.g. linear search
- Heuristic search
 - Approximate. E.g. finding the shortest path to a destination. Finding the definitive solution would require processing many possibilities - we can find an approximate search very quickly. (Ranks alternatives to prioritise possible paths to look at first) May find the exact solution!
- Simulation
- Divide and conquer

Are these algorithm types exact or approximate?

- Brute force (generate and test)
 - Exact. Finds every possible answer and tests it. Requires set of possible answers to be finite to guarantee completion. E.g. linear search
- Heuristic search
 - Approximate. E.g. finding the shortest path to a destination. Finding the definitive solution would require processing many possibilities - we can find an approximate search very quickly. (Ranks alternatives to prioritise possible paths to look at first) May find the exact solution!
- Simulation
 - Approximate. Finds a solution by generating lots of data to predict an overall trend. E.g. simulate play of a game to find out if it's worth playing
- Divide and conquer

Are these algorithm types exact or approximate?

- Brute force (generate and test)
 - Exact. Finds every possible answer and tests it. Requires set of possible answers to be finite to guarantee completion. E.g. linear search
- Heuristic search
 - Approximate. E.g. finding the shortest path to a destination. Finding the definitive solution would require processing many possibilities - we can find an approximate search very quickly. (Ranks alternatives to prioritise possible paths to look at first) May find the exact solution!
- Simulation
 - Approximate. Finds a solution by generating lots of data to predict an overall trend. E.g. simulate play of a game to find out if it's worth playing
- Divide and conquer
 - Exact. Divides problem into sub-problems which can be more easily solved. E.g. binary search

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
---	---	---	---	---	---	----	----	----	----	----	----	----

Write a recursive function which takes an integer n and calculates the n^{th} fibonacci number. The 0^{th} fibonacci number is 0, the 1^{st} fibonacci number is 1 and all following fibonacci numbers are defined as the sum of the preceding two fibonacci numbers. `fib(10)` should return 55

Write a recursive function which takes an integer n and calculates the n^{th} fibonacci number. The 0^{th} fibonacci number is 0, the 1^{st} fibonacci number is 1 and all following fibonacci numbers are defined as the sum of the preceding two fibonacci numbers. `fib(10)` should return 55

```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n - 1) + fib(n - 2)
```

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?

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?

```
# 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
```