Functional Programming

Prof Matthew Fricke

Recap: Programming Paradigms

- Procedural programming:
  - Each step in the program is laid out one after the other in a straightforward way

```
# Code that sums ints
start = 5
end = 20
total = 0
current_num = start
while current_num <= end:
    total=total + current_num
    current_num=current_num + 1
print(total)
```

Recap: Programming Paradigms

- Object Oriented Programming:
  - Behaviour and state are packaged together.

```
class Square():

    # Size

    def __init__(self, s ):
        self.size = s
        self.turtle = turtle.Turtle()
        self.colour = "blue"

    def getArea(self):
        return self.size**2
```

Notice that inside the methods the program is still procedural.
We are just changing the way we think about the code.

Functional Programming

- This is another programming style or paradigm.
- Recall that we have been trying throughout the semester to limit how much we have to think about at any one time.
- We used objects and classes to do that. There we just think about what an object can do and not so much about how it does it.
- We called that encapsulation, information hiding, etc.

- Functional programming uses functions to encapsulate solutions to problems.
- Each function takes arguments and returns a solution.

- We have already done that.
- In functional programming we are going to think about how to put functions together each of which solve a small problem, and use them all together to solve bigger problems.

Functional Programming

- To make everything easier to understand we are going to forbid side effects.
- A side effect is when a function changes variables that were defined outside itself.

```
x = 2

def side_effect( first, second ):
    global x
    x = first * second

print(x)

side_effect(3,4)

print(x)
```

```
PS H:\Teaching\CS151-2020\code> python3 .\side_effect.py
2
12
```

See how the value of x changes even though x doesn't belong to side_effect's namespace.

Side effects make code hard to understand and unpredictable. That's why python makes you use the global keyword to make sure it's what you want.

Side effects are also called mutations and functional languages such as Haskell, Scheme, and ML tend to forbid them.

Functional Programming: Recursion

- We have seen functions call other functions before. Recall Lecture 6, slide 83:

```
def for_mult(start, end):
    total = 1
    for x in range(start, end+1):
        total = total * x

    return total

# Factorial function
def factorial(x):
    return for_mult(1, x)
```

We used the for_mult function inside the factorial function because we already had for_mult, and for_mult already solved a problem sort of like factorial.

So we used constructive laziness to just use our existing solution.

Recursion just means a function that calls itself to solve a problem.

Functional Programming: Recursion

- Recursive programs tend to follow a consistent pattern.*
- They have:
- One or more base cases.
- One or more recursive steps.

We think about how to break problems into smaller pieces, over and over again, until we get a problem so small it is easy.

Then we put all the easy answers together to solve a big problem.

*If you know proof my mathematical induction this is exactly the same idea.

Functional Programming

- Recursive programs tend to follow a consistent pattern.*
- They have:
- One or more base cases.
  A base case is a problem that is very easy to solve. In fact is it the easiest possible version of the problem we want to solve.
- One or more recursive steps.

*If you know proof my mathematical induction this is exactly the same idea.

Functional Programming

- Recursive programs tend to follow a consistent pattern.*
- They have:
- One or more base cases.
  A base case is a problem that is very easy to solve. In fact is it the easiest possible version of the problem we want to solve.
- One or more recursive steps.
  The recursive step is where we code up how to take solutions to slightly simpler problems to solve a slightly bigger problem.

*For those of you who know proof my mathematical induction, this is the same idea.

Recursion Example

- Let's think about how to multiply a range of integers like functional instead of procedural programmers.
- Problem to solve: We want a program that multiplies integers between a start and end value.
- As procedural programmers we iteratively multiply the current product by the next number in a list:

```
total = 1
for x in range(start, end+1):
    total = total * x
```

Recursion Example

- Let's think about how to multiply a range of integers like functional instead of procedural programmers.
- Problem to solve: We want a program that multiplies integers between a start and end value.
- As functional programmers we think what is the easiest version of this problem?
- The easiest version (base case) is when there are only 2 numbers to multiply. That is, when the value of end is only one more than start*
- So solve that problem we just multiply end by start.

```
end * start
```

*Let's ignore the even easier case where end = start now.

Recursion Example

- Let's think about how to multiply a range of integers like functional instead of procedural programmers.
- The next step is to thing of a way to use that to solve the next biggest problem.
- The next biggest problem is when there are three integers to multiply.
- How can we solve the case with 3 integers? We already solved the case with 2 integers and 3 integers is just one integer more.

*Let's ignore the even easier case where end = start now.

Recursion Example

- Let's think about how to multiply a range of integers like functional instead of procedural programmers.

Let's think a bit abstractly: Let's call our three integers A, B, and C.

We already solved the two integer case: integer A times integer B. To solve the 3 integers case we just take the solution to the two integer case and do it again. After all the solution to the two integer case is just a single integer*.

(integer A times integer B) times integer C. (again reducing the size of the problem*)

We can do this repeatedly for any number of integers.

The base case is (integer A times integer B)

The recursive step is (solution to smaller problem) times (the next integer)

*Important: any recursive algorithm will tend to reduce the size of the problem remaining at each step.

Recursion Example

Let's code it up:


```
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

Recursion Example

Let's code it up:

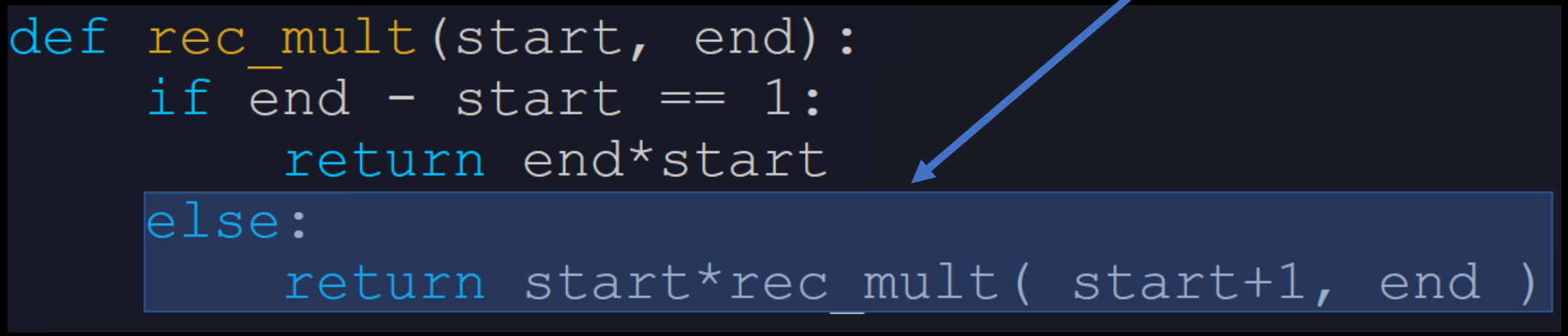
The base case where end is just 1 away from start. Just multiply the two.

```
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

Recursion Example

Let's code it up:

The inductive step.
Multiply start by whatever the solution is to the smaller problem of multiplying integers between start+1 and end.



```
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

The key is to recognize that rec_mult(start+1, end) has one less integer to multiply to get the solution than rec_mult(start, end) does.

Recursion Example

Let's code it up:

```
                    3     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7)

Recursion Example

Let's code it up:

```
                    3     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                  3  ×                  4       7
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7)

Recursion Example

Let's code it up:

```
                    4     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                 4  ×                  5      7
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7)

Recursion Example

Let's code it up:

```
                    5     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                 5  ×                  6      7
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × rec_mult(6,7)

Recursion Example

Let's code it up:

```
                    6     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:          6    7
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)

Recursion Example

Let's code it up:

```
                    6     7
def rec_mult(start, end):
    if end - start == 1:
      42 return end*start
    else:          6    7
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)
= 42

Recursion Example

Let's code it up:

```
                    5     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                 5  ×   42
    210 return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)

= 5 × 42

Recursion Example

Let's code it up:

```
                    4     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                 4  ×   210
    840 return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)

= 4 × 5 × 42

Recursion Example

Let's code it up:

```
                    3     7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:                 3  ×   840
   2250 return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)

= 3 × 4 × 5 × 42

Recursion Example

Let's code it up:

```
     2250          3      7
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

You can think about unrolling the recursion with an example (start=3, end=7) to understand it better:

rec_mult(3, 7) = 3 × rec_mult(4,7) × rec_mult(5,7) × (6 × 7)

= 3 × 4 × 5 × 42

= 2520

Recursion Example

Let's code it up:

```
def rec_mult(start, end):
    if end - start == 1:
        return end*start
    else:
        return start*rec_mult( start+1, end )
```

Notice there is no explicit assignment of values to variables "=".

1. Values that define the problem move through function arguments.
2. Values that represent the solution (or partial solution) are returned by functions.

Recursion Example (slightly different solution)

Let's code it up:

```
def rec_mult(start, end):
    if end == start:
        return end
    else:
        return start*rec_mult( start+1, end )
```

Like everything else we have seen so far there are usually many solutions to a problem.
Here is another recursive solution (this one is slightly more elegant).

The base case here is where start and end are the same number (the simplest subproblem is just one integer)
The recursive step it the same.
The only difference is that rec_mult is recursively called one more time than previously.

Linear Recursion

Every problem that can be solved with iteration can be straightforwardly solved with recursion.

Not every problem that can be solved with recursion can be solved with iteration.

The kind of recursion we just saw is called tail recursion. Tail recursive functions have a single recursive call at the end of the function. They are related to iteration.

Linear Recursion

Recall Palindrome Example from the Exam 1 review. Here is a procedural style solution.

```
def reverse( text ):
    reversed_text = ""
    length = len(text)
    for t in range(1, length+1):
        reversed_text += text[length-t]
    return reversed_text

def palindrome( text ):
    return reverse(text) == text
```

Linear Recursion

Let’s design a functional style solution:

We need to think of

1. the base case,

2. the recursive step,

3. and how to reduce the size of the problem at each step.

Linear Recursion

Let's design a functional style solution:

We need to think of

1. the base case,
   The smallest subproblem would be a string of length 2.
   Check if the first character is equal to the second character.

2. the recursive step,
   If the characters at the end of the current sting are equal, and all the subproblems are palindromes then the whole string is a palindrome.

3. and how to reduce the size of the problem at each step.
   Remove the end characters and call palindrome on the smaller string.

Recursive Palindrome

```
def palindrome( text):
    if len(text) < 2:
        return True
    else:
        return text[0]==text[-1] and palindrome(text[1:-1])
```

We have one base case. When a string has less than two characters, we say it is a palindrome (i.e. one or no characters), i.e. return True.

We have one recursive step. We check to see if the outermost characters are the same and whether the next smallest subproblem is a palindrome.

Recall that both arguments to and have to be true for the statement to be true.

Recursive Palindrome

```
def palindrome( text):
    if len(text) < 2:
        return True
    else:
        return text[0]==text[-1] and palindrome(text[1:-1])
```

```
print(palindrome("wasitacaroracatisaw"*))
```

True

Let’s trace through an example:
palindrome(“racecar”)
Does “r” equal “r” and (Does “a” equal “a” and (Does “c” equal “c” and True)) which is True.

*https://www.youtube.com/watch?v=JUQDzj6R3p4

Tree Recursion

Functions that use tree recursion* have multiple calls to themselves at each recursive step. This leads to a branching pattern like the branches of a tree.

Recall the Fibonacci sequence from Lecture 8, slides 1 though 17.

*Also called multiple recursion.

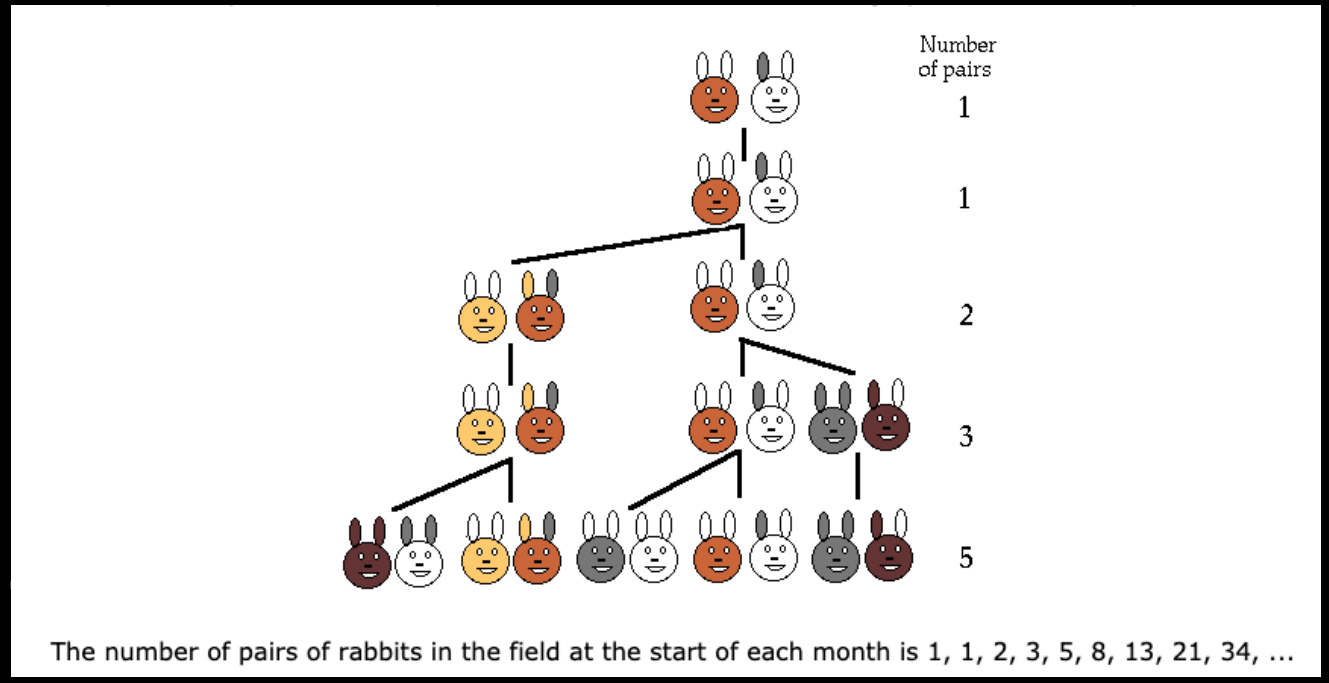
Tree Recursion

Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}$$

Notice that mathematically the sequence is defined recursively (in terms of itself).

…and that the figure already takes on a (upside-down) tree like structure.



The number of pairs of rabbits in the field at the start of each month is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
def fibonacci( N ):
    fib = [0] * N
    fib[0] = 1
    fib[1] = 1
    for i in range(2,N):
        fib[i] = fib[i-1] + fib[i-2]
    return fib
```

Tree Recursion

Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}$$

Index: 0, 1, 2, 3, 4, 5…

Value: 0, 1, 1, 2, 3, 5…

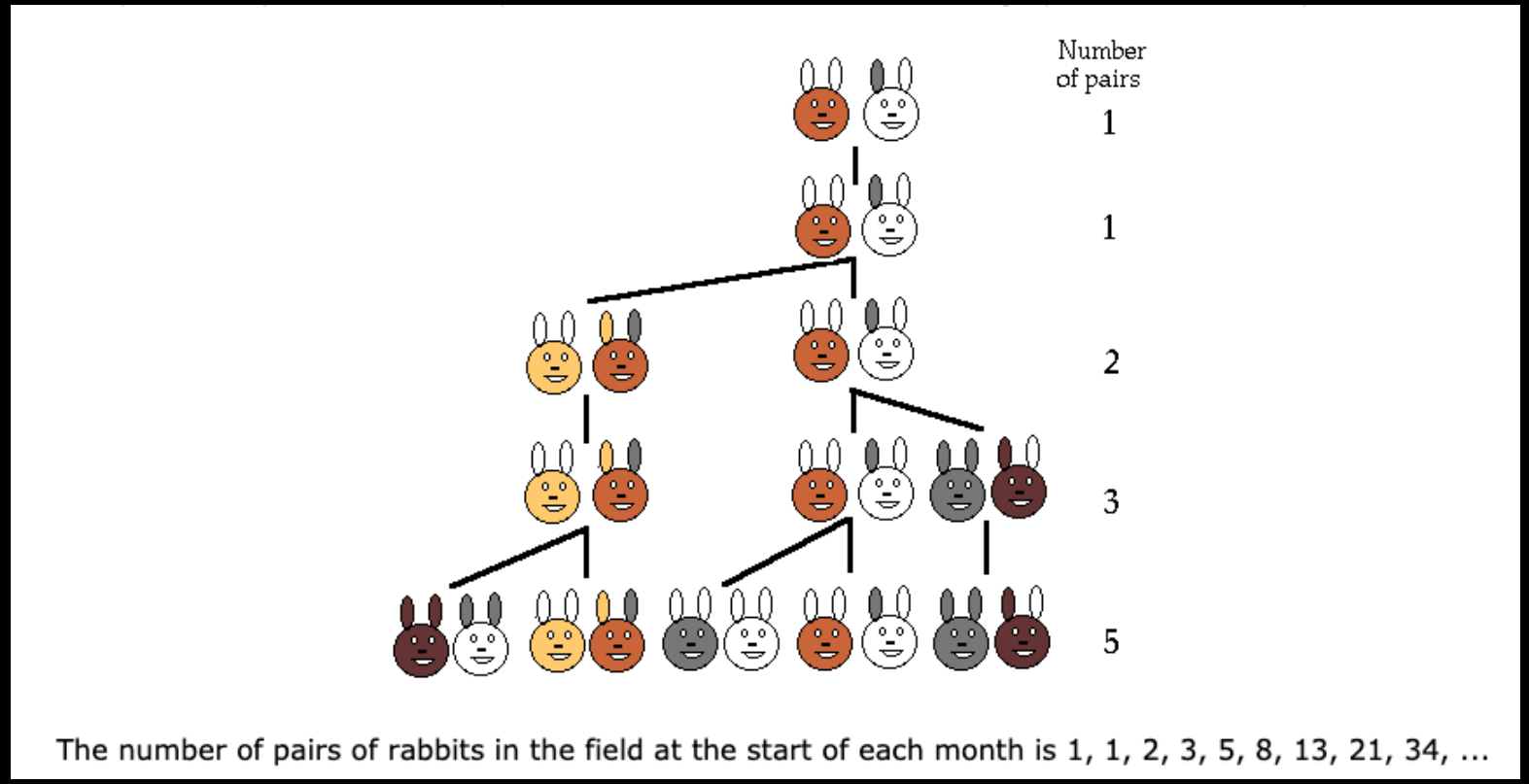
Base cases:

F(0) = 0

F(1) = 1

Recursive step:

F(N-1) + F(N-2)



The number of pairs of rabbits in the field at the start of each month is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
def fibonacci( N ):
    if N == 0:
        return 0
    elif N == 1:
        return 1
    else:
        return fibonacci(N-1) + fibonacci(N-2)
```

Binary Search

Imagine you are looking for page 400 in a long book.

You would probably not do it by starting at page 1 and then checking each consecutive page to see if it is page 400.

You wouldn't start at the end and work backwards either.

You are more likely to start looking somewhere in the middle of the book. If the page number you find is too high you will look earlier in the book, if too low you look later in the book.

In other words you are dividing the problem into smaller subproblems. You are using a divide-and-conquer algorithm.

This is called binary search and comes up a lot in computer science. It has nothing to do with binary numbers. It is called binary search because we will divide the problem in two (binary) each time.

That's a lot like the recursive algorithms we saw so far. Only there we reduced the problem by one or two elements at a time. When we can divide the problem in two at every step we can solve it very fast.

Computer Science (Preview)

Computer science is concerned with what kind of problems can be solved, and how fast can we solve them.

All kinds of algorithms have been classified by how long they take to solve. This is extremely important because it turns out some important algorithms take a very long time to solve for large problems.

A famous example is the Travelling Salesperson problem:

Imaging you want to know the shortest path through some number of cities, where the path only goes through each city once. Sounds pretty easy, especially when we are used to GPS navigation.

Let's look at how long it takes to solve the problem given the fastest known algorithm:

Num cities	**Number of possible paths**	**Time to find the solution* (1 *μs/path*)**
5	12	12 *μs*
10	181, 440	0.18 *s*
15	4.359×10^{10}	12 *h*
20	6.082×10^{16}	1, 928 years
61	4.160×10^{81}	13.19×10^{67} years

For comparison, the age of the universe is currently estimated to be around 13×10^{9} years and the number of atoms in the observed universe is around 10^{80}.

One of the reasons it takes so long to solve this problem is there is no divide and conquer strategy.

* Table reproduced from Swarm Problem-Solving, Antoine Dutot, Damien Olivier, in Agent-based Spatial Simulation with NetLogo, Volume 2, 2017

Binary Search

The binary search problem is not like the travelling salesperson problem. We can use recursion and divide and conquer to solve it very fast.

From now on we will not only think about how to solve problems but how long our solutions take.

* Table reproduced from Swarm Problem-Solving, Antoine Dutot, Damien Olivier, in Agent-based Spatial Simulation with NetLogo, Volume 2, 2017

Binary Search

The problem setup: We want to find the position of an item in a list when the list is sorted from smallest to largest or return False if the item is not in the list.

Examples,
Search for 3 in the following list: [-4, -2, 1, 2, 3, 6, 7, 9, 10] should return 4
Search for 8 in the following list: [0, 3, 900, 2000, 96543] should return False

Let's use the usual recursive problem-solving strategy:

1) Base case
   The smallest problem size is a list with a single element. If we have a single element we can just check if it is the item for which we are looking.

2) Recursive step
   Look at the item in the middle of the current list. If it is bigger than the item we are looking for search the half of the list before that item, and latter half if it's smaller. Otherwise check the single item list.

3) How do we reduce the problem size at each step.
   We divide the problem size in half every step and eliminate half the list from consideration.

Binary Search: Coding it up

This is enough to return whether the item is in the list.

```
def binary_search(item, L ):
    if len(L) == 1:
        return item == L[0]
    else:
        midpoint = len(L)//2
        if L[midpoint] > item:
            return binary_search( item, L[:midpoint] )
        elif L[midpoint] < item:
            return binary_search( item, L[midpoint:] )
        else:
            return binary_search(item, [L[midpoint]] )
```

Binary Search: Coding it up

To return the position in the list where the item was found we need a bit more.

```
def binary_search(item, L, position ):
    if len(L) == 1:
        if item == L[0]:
            return position
        else:
            return False
    else:
        midpoint = len(L)//2
        next_midpoint = midpoint//2
        if next_midpoint == 0:
            next_midpoint = 1
        if L[midpoint] > item:
            return binary_search( item, L[:midpoint], position-next_midpoint )
        elif L[midpoint] < item:
            return binary_search( item, L[midpoint:], position+next_midpoint )
        else:
            return binary_search(item, [L[midpoint]], position )
```

Binary Search: Coding it up

To return the position in the list where the item was found we need a bit more.

```
def binary_search(item, L):
    midpoint = len(L)//2

    if L[midpoint] == item:
        return midpoint
    elif len(L) == 1:
        return False
    else:
        if L[midpoint] < item:
            val = binary_search(item, L[midpoint:])
            if not val:
                return False
            else:
                return midpoint + val
        else:
            return binary_search(item, L[:midpoint])
```

The highlighted areas keep track of where the item was found.

We have to be careful to check for False. Since midpoint + False would be executed as though False were just 0.

Binary Search: Coding it up

To return the position in the list where the item was found we need a bit more.

```
def binary_search(item, L):
    midpoint = len(L)//2

    if L[midpoint] == item:
        return midpoint
    elif len(L) == 1:
        return False
    else:
        if L[midpoint] < item:
            val = binary_search(item, L[midpoint:])
            if not val:
                return False
            else:
                return midpoint + val
        else:
            return binary_search(item, L[:midpoint])
```