

Recursion / Searching

Week 13

Recursion

- Recursion is the process where a module calls itself
- Recursion runs the risk of creating infinite loops – since the function could continuously call itself over and over
- To prevent this – create a base case – something that doesn't call itself and will be eventually satisfied

Recursion

- Creating a recursive function requires two steps:
 - Write the base cases – this case returns a value. This ends the recursion
 - There may be multiple base cases provided
 - Write the recursive case
- When you call your function, the following steps happen:
 - The code is executed until it reaches the recursive call or the base case – if it reaches the base case, then it is done
 - If it does not reach the base case, it is paused and the code is once again called until it reaches the base case or the recursive call
 - This process continues until it reached the base case
 - Once the base case is reached – it provides a value to the last called recursive function which provides a value to the previous function, etc.

Program execution

- A program's main module sits in a memory location called the **stack**
- When a call to another module is made, the main module is suspended and the new module occupies an unused area of memory.
 - This is known as being pushed onto the stack
- If the new module calls another module, the same process happens, the previous module is suspended, the new module placed in unused memory on the stack and runs
- This process repeats until the module called can finish running – once it has – it is removed from memory
 - This is known as **popped** from the stack

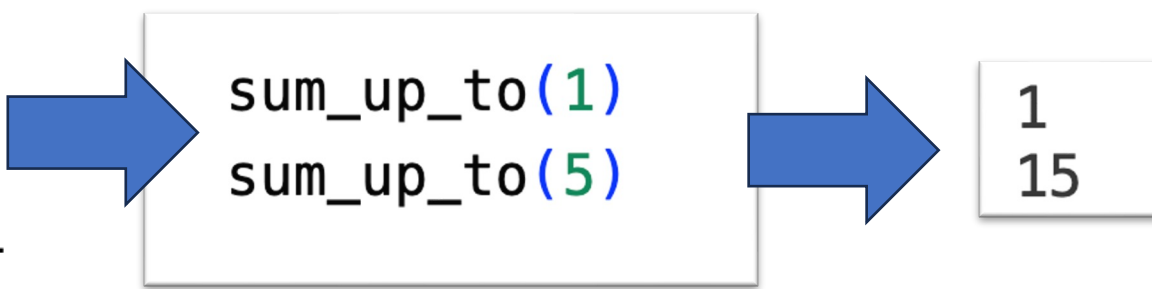
Program Execution

- When a module is popped from the stack, the memory it was using is relinquished which means that any variables, etc. are no longer accessible
 - Once the module completes its run, nothing exists anymore
- Continuously adding modules (having modules call new modules after new modules) without every reaching the end, can mean that you use up all available memory to the stack
 - This condition where no memory remains is called a **stack overflow**

Recursion

- Create a program that will decide the sum of all numbers up to a specific number
- For example, the sum of all numbers up to 5:

```
def sum_up_to(number):  
    for i in range(number):  
        number = number + i  
    print(number)
```



```
sum_up_to(1)  
sum_up_to(5)
```

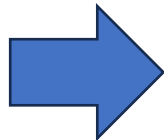
```
1  
15
```

Recursion

- This could be written recursively:

```
def rsum_up_to(number):  
    if number == 1:  
        return number  
    else:  
        return number + rsum_up_to(number - 1)
```

```
print(rsum_up_to(1))  
print(rsum_up_to(5))
```



```
1  
15
```

Recursion

- How this works -> Start with 5
- $5 + \text{rsum_up_to}(4)$ -> Which calls itself to get
 $4 + \text{rsum_up_to}(3)$ -> which calls itself to get
 $3 + \text{rsum_up_to}(2)$ -> which calls itself to get
 $2 + \text{rsum_up_to}(1)$ -> which returns 3
 $\text{rsum_up_to}(1)$ returns 1 -> $1 + 2 = 3$
- This means that $3 + \text{rsum_up_to}(2)$ is -> $3 + 3 = 6$
- This means that $4 + \text{rsum_up_to}(3)$ is -> $6 + 4 = 10$
- This means that $5 + \text{rsum_up_to}(4)$ is -> $10 + 5 = 15$

Sorting

- Python has the `sort()` and `sorted()` to place items in ascending or descending order
 - When applied to a list of numbers, it places them in numerical order
 - When applied to a list of words/characters, it places them in alphabetical order
- When people sort, we take a look at the whole dataset to see what goes where
- Programs cannot do that – they need to compare two items at a time

Linear Search

- A linear search is a search algorithm that starts at the beginning of a list and checks each element in order until one of two conditions is met
 - The searched item is found
 - The end of the list is reached

Linear Search

```
letters = ['a', 'b', 'c']  
num_searches = 0  
look_for = 'a'
```

Found! It took 1
a was not found after 1 search(es).

```
for letter in letters:  
    num_searches += 1  
    if letter == look_for:  
        print(f'Found! It took {num_searches}')        break
```

f was not found after 3 search(es).

```
print(f'{look_for} was not found after {num_searches} search(es).')
```

Algorithm Runtime

- An algorithm's runtime is the time it takes to complete the execution of its code.
- For a linear search, the runtime is directly related to the number of items it has to search to find – or the total number of elements in the list
 - For example – if each search in the previous code took 1s, it would take 1s to find the letter a, 2s to find the letter b, etc.
 - Not finding it would take 3s
- This can be described as **a list with N elements requires at most N comparisons to complete**
 - Where **N** is the total number of elements in the list

Binary Search

- Linear search will progress in a linear fashion
- If we were looking through an array of letters a-z, and we were looking for z – it would take us 26 comparisons to find the letter z.
- Since this list is ordered – there is a quicker way
- A binary search, rather than starting at the beginning of a list will start in the middle

Binary Search

```
start = 0
end = len(letters) - 1
num_searches = 1
while end >= start:
    mid = (end + start) // 2
    if letters[mid] < look_for:
        num_searches += 1
        start = mid + 1
    elif letters[mid] > look_for:
        num_searches += 1
        end = mid - 1
    else:
        print(f'Found the letter {look_for} after {num_searches} search(es).')
        break
```

Check if the midpoint comes before the item you were looking for, if it does, then this becomes the new **starting** point for searching

Check if the midpoint comes after the item you were looking for, if it does, then this becomes the new **end** point for searching

Binary search is efficient when searching a sorted list
Each iteration reduces the items to be searched by half.

Binary Search

Linear Search – Found the letter v! It took 22
Binary Search – Found the letter v it took 5 search(es).

Linear Search – Found the letter w! It took 23
Binary Search – Found the letter w it took 3 search(es).

Linear Search – Found the letter f! It took 6
Binary Search – Found the letter f it took 2 search(es).

Linear Search – Found the letter a! It took 1
Binary Search – Found the letter a it took 4 search(es).

Big O Notation

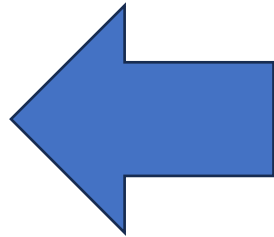
- Big-O Notations is a way to describe running time
 - How long it takes for a block of code to complete
- Constant time – this is way of describing an operation that takes the same amount of time to execute
 - This is $O(1)$ in big-O notation
 - For example – looking up the **first** item in a list that has 1 item or 1000 takes the same amount of time

Constant Time

```
numbers = [1,2,3,4,5,6,7,8,9]
```

```
def print_first(list):  
    print(list[0])
```

```
print_first(numbers)  
print_first(letters)
```



Whether the list is 9 elements long or 100,
it has no effect on the running time

Big O Notation

- Linear search- looking through every item from start to finish, increases proportionally to the number of items in the list
 - If it takes 10 units of time to look through 10 items, then it would take 100 units of time to look through 100 items
 - $O(N)$ is the notation for this linear time operation
 - The size of the input N also is the amount of time the process takes.
 - There is a direct, linear relationship between the input size and the time it takes to search

Linear Time

```
def print_all(list):  
    for element in list:  
        print(element)
```

This takes as long as the number of elements that are in the list.
More elements, longer time.
Less elements, less time

```
print_all(numbers)
```

This involves 9 elements

```
print_all(letters)
```

This involves 26 elements

Quadratic Time

- Also known as n squared
- This describes an algorithm that takes time proportional to the square of the input size
- Referred to as $O(n^2)$

Quadratic Time

- Prints all the items for every item

```
def repeat_for_every_item(list):  
    for i in range(len(list)):  
        print(i)  
        for element in list:  
            print(element)
```

A minor increase in data size, results in a much larger increase in the amount of time

```
repeat_for_every_item(numbers)
```

```
repeat_for_every_item(letters)|
```

Selection Sort

- Selection sort classifies input data into two categories:
 - Sorted
 - Unsorted
- Decides which next value in the unsorted, needs to be moved to the sorted
- Selection sort can use a large number of comparisons
 - $O(N^2)$

Selection Sort

```
import random

letters = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']
random.shuffle(letters)
print(letters)

for index in range(len(letters)-1):

    divider = index

    for unsorted_start in range(divider+1, len(letters)):

        print(f'Divider letter: {letters[divider]}')
        print(f'Unsorted letter: {letters[unsorted_start]}')

        if letters[unsorted_start] < letters[divider]:
            print('Moving...')
            divider = unsorted_start

    temp = letters[index]
    print(f'Letter to move:{temp}')
    letters[index] = letters[divider]
    letters[divider] = temp

print(letters)
```

```
u, u, v, p, g, l, q, n, j, s, k, n, u, x, d, z, w, c, t, e, y, c, o, c, z, m, j
Divider letter: d
Unsorted letter: u
Divider letter: d
Unsorted letter: v
Divider letter: d
Unsorted letter: p
Divider letter: d
Unsorted letter: a
```

Insertion Sort

- Like a selection sort, instead of selecting it is inserting
- Nested loops that are repeatedly inserting the value from an unsorted section into the correct location in the sorted part
- This also contains a large number of comparisons – $O(N^2)$
 - Runtime for nearly sorted lists is $O(N)$

Insertion Sort

```
for index in range(1, len(letters)):
    unsorted_index = index

    while unsorted_index > 0 and letters[unsorted_index] < letters[unsorted_index - 1]:

        temp = letters[unsorted_index]
        print(f'Letter to move:{temp}')
        letters[unsorted_index] = letters[unsorted_index - 1]
        letters[unsorted_index - 1] = temp
        print(letters)
        unsorted_index -= 1

print(letters)
```

```
['p', 't', 'u', 'k', 'c', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:k
['p', 't', 'k', 'u', 'c', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:k
['p', 'k', 't', 'u', 'c', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:k
['k', 'p', 't', 'u', 'c', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:c
['k', 'p', 't', 'c', 'u', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:c
['k', 'p', 'c', 't', 'u', 'y', 'w', 'd', 'o', 'v', 'l', 'q', 'm', 'b', 'z', 'i', 'e', 'n', 'f', 'a', 's', 'x', 'g', 'j', 'h', 'r']
Letter to move:c
```

Logarithmic Time

- An algorithm that can reduce the number of comparisons by half each time will have logarithmic time complexity
- As input data increases, the time to perform the search increases at a slower rate
- This is $O(\log n)$ in big O Notation

Binary Search

```
start = 0
end = len(letters) - 1
num_searches = 1
while end >= start:
    mid = (end + start) // 2
    if letters[mid] < look_for:
        num_searches += 1
        start = mid + 1
    elif letters[mid] > look_for:
        num_searches += 1
        end = mid - 1
    else:
        print(f'Binary Search - Found the letter {look_for} it took {num_searches} search(es).')
        break
```

Quicksort

- Quicksort partitions the input into low and high parts
- It has a pivot point – that divides the array into three parts or partitions
 - Elements that are less than the pivot (still unsorted)
 - The pivot
 - Elements that are greater than the pivot (still unsorted)

A recursive sorting process then sorts the low and high partitions until each partition has one or zero elements – meaning it is sorted

Quick Sort - Partitioning

```
def quick_sort(collection_of_numbers):  
    if len(collection_of_numbers) <= 1:  
        return collection_of_numbers  
    else:  
        print("Iteration", i)  
        pivot_index = len(collection_of_numbers) // 2  
        pivot = collection_of_numbers[pivot_index]  
        print("Pivot:", pivot)  
  
        less = [x for x in collection_of_numbers[:pivot_index] + collection_of_numbers[pivot_index + 1:] if x <= pivot]  
        print("Less:", less)  
  
        greater = [x for x in collection_of_numbers[:pivot_index] + collection_of_numbers[pivot_index + 1:] if x > pivot]  
        print("Greater:", greater)  
  
        return quick_sort(less) + [pivot] + quick_sort(greater)  
  
print("Given list:", numbers)  
  
quick_sort(numbers)
```