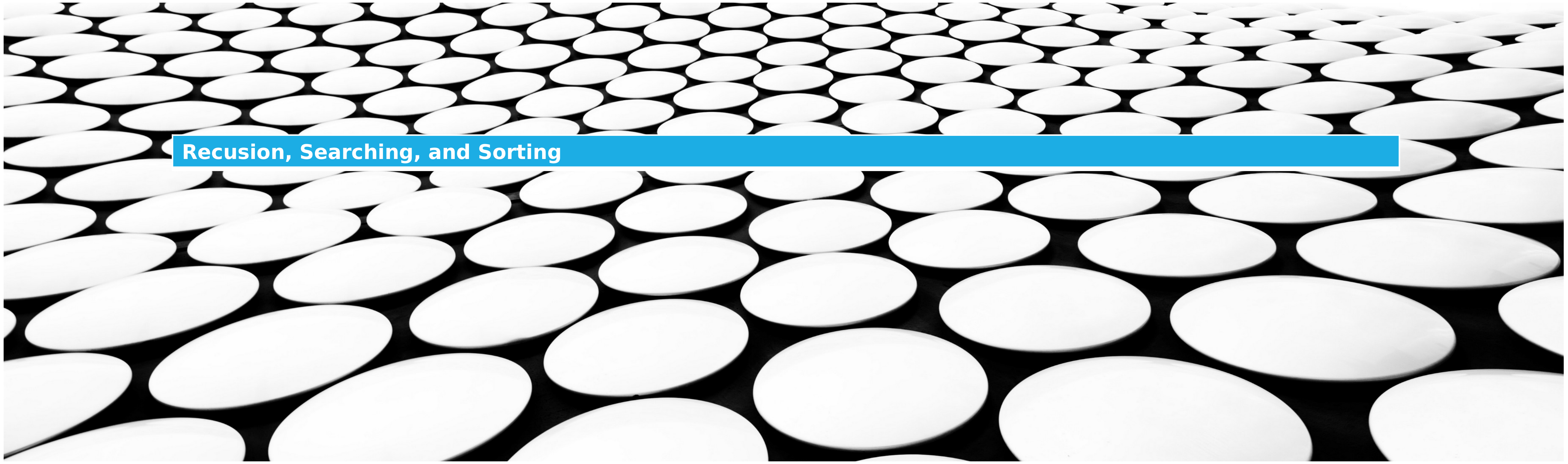
DAS 456 – WEEK 3

RECURSION, SEARCHING AND SORTING



THE RUN-TIME STACK

- The run-time sack is basically the way your programs sore and handle your local non-static variables.
- Think of the runtime sack as a sack of plates. With each function call, a new "plate" is placed onto the sacks. local variables and parameters are placed onto this plate.
- Variables from the calling function are no longer accessible (because they are on the plate below).
- When a function terminates, the local variables and parameters are removed from the stack.
- Control is then given back to the calling function. Understanding the workings of the run-time sack is key to understanding how and why recursion works

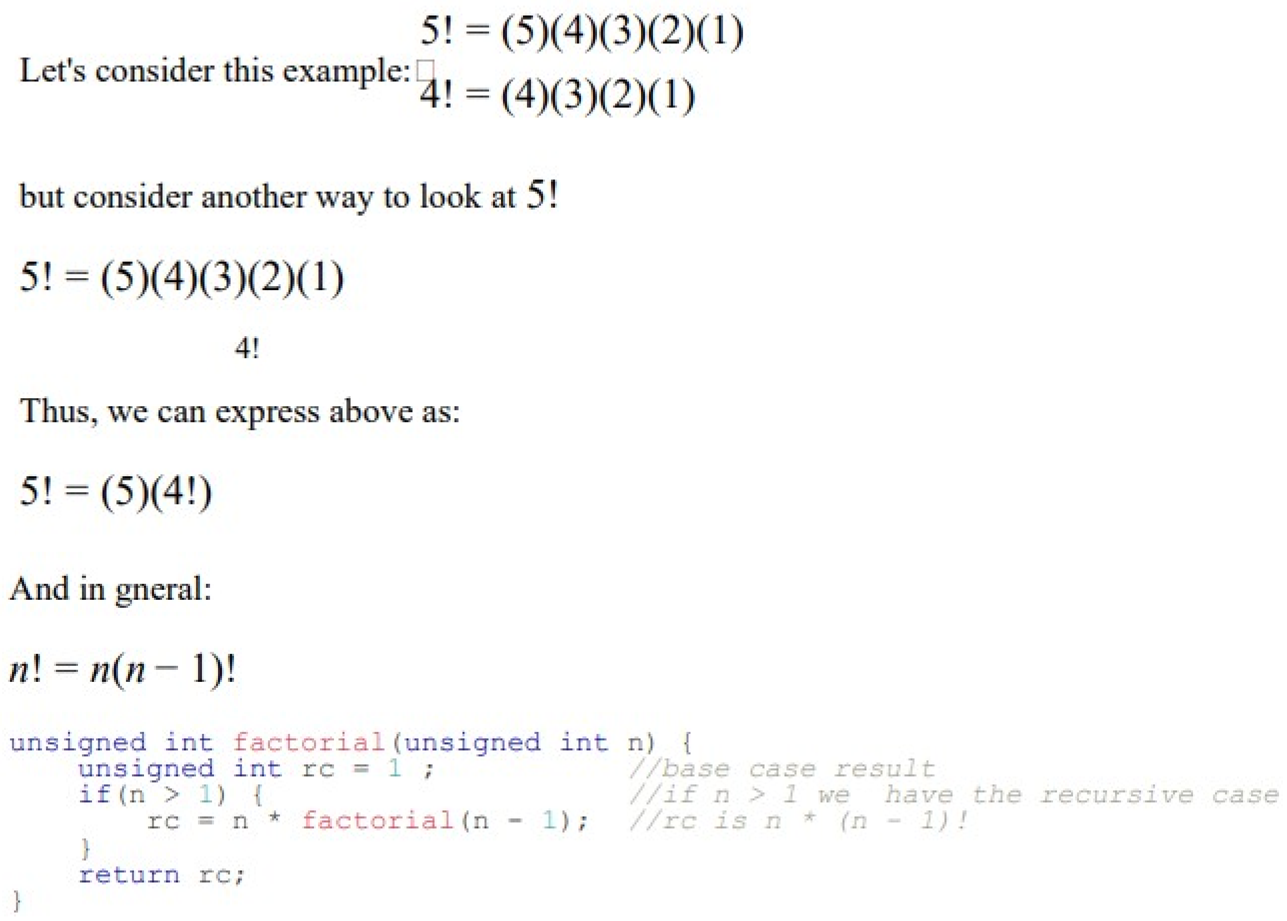
WRITING RECURSIVE FUNCTIONS

- Recursive functions are sometimes hard to write because we are not used to thinking about problems recursively. However, if we try to keep the following in mind, it will make writing recursive functions a lot simpler.
- There are two things you should do:
  1. State the base case (aka easy case). what argument values will lead to a solution so simple that you can simply return that value (or do a very simple calculation and return that value?
  2. State the recursive case. if you are given something other than the base case how can you use the function itself to get to the base case
  3. Recursion is about stating a problem in terms of itself. The solution to the current problem consists of taking a small step and restating the problem.

EXAMPLE: THE FACTORIAL FUNCTION

- Write a function that is given a single argument n and returns n!. n! = n*(n-1)*(n-2) ...2* 1.
- For example 4! = 4 * 3 * 2 * 1 , by definition, 0! is 1.
- How would you do it recursively?
- Lets start with base case.
- What value of n is it easy to come up with the result of n!? 0 is easy.. by definition 0! is 1. 1 is also easy because 1! = 1
- Thus, we can establish that any value 1 or less is a base case with result of 1. Now, the recursive case... how can we state factorial in terms of itself?

EXAMPLE: THE FACTORIAL FUNCTION CONT.



Let's consider this example: $5! = (5)(4)(3)(2)(1)$

$4! = (4)(3)(2)(1)$

but consider another way to look at $5!$

$$5! = (5)(4)(3)(2)(1)$$

$4!$

Thus, we can express above as:

$$5! = (5)(4!)$$

And in gneral:

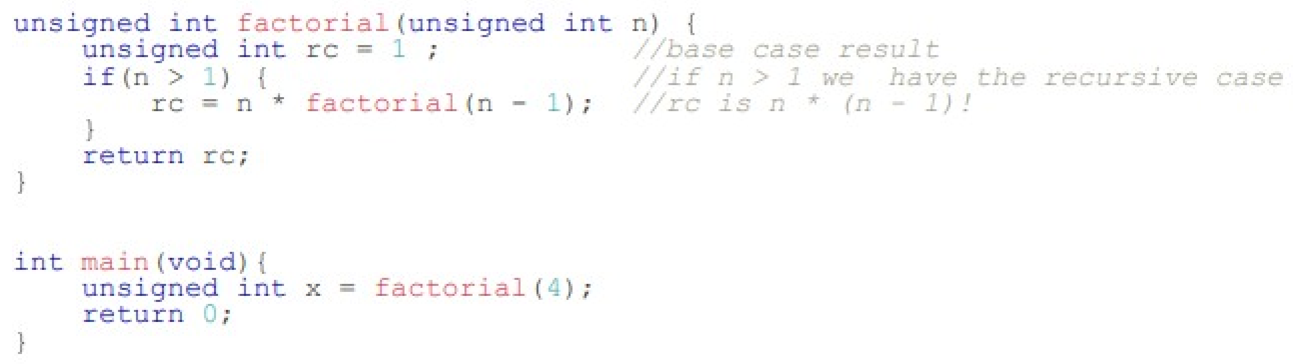
$$n! = n(n-1)!$$

```
unsigned int factorial(unsigned int n) {
    unsigned int rc = 1 ;           //base case result
    if(n > 1) {                     //if n > 1 we  have the recursive case
        rc = n * factorial(n - 1);  //rc is n * (n - 1)!
    }
    return rc;
}
```

HOW DOES RECURSION WORK?

- To understand how recursion works, we need to look at the behavior of the run time stack as we make the function calls.
- The runtime stack is a structure that keeps track of function calls and local variables as the program runs. When a program begins, the main() function is placed on the run time stack along with all variables local to main().
- Each time a function is called, it gets added to the top of the runtime stack along with variables and parameters local to that function. Variables below it become inaccessible.
- When a function returns, the function along with its local variables are popped of the stack allowing access to its caller and its callers variables.

HOW DOES RECURSION WORK? CONT.

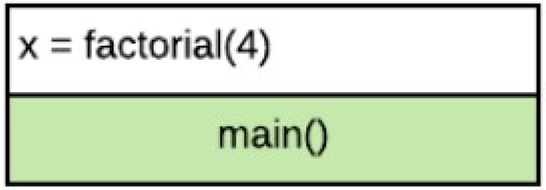


```
unsigned int factorial(unsigned int n) {
    unsigned int rc = 1 ;           //base case result
    if(n > 1) {                     //if n > 1 we  have the recursive case
        rc = n * factorial(n - 1);  //rc is n * (n - 1)!
    }
    return rc;
}


int main(void){
    unsigned int x = factorial(4);
    return 0;
}
```

HOW DOES RECURSION WORK? CONT.

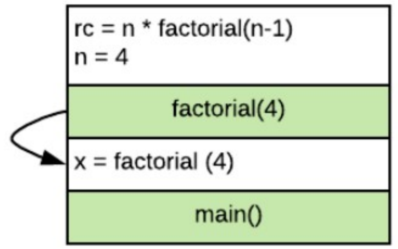
Lets trace what happens to it with respect to the run time stack:

Program begins, main() function is placed on stack along with local variable x.

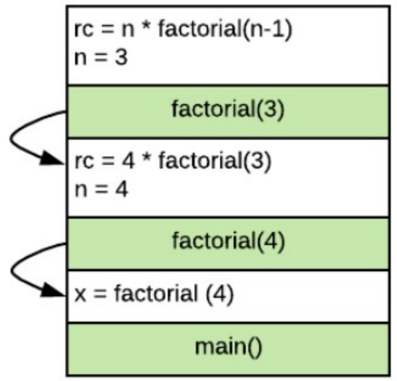


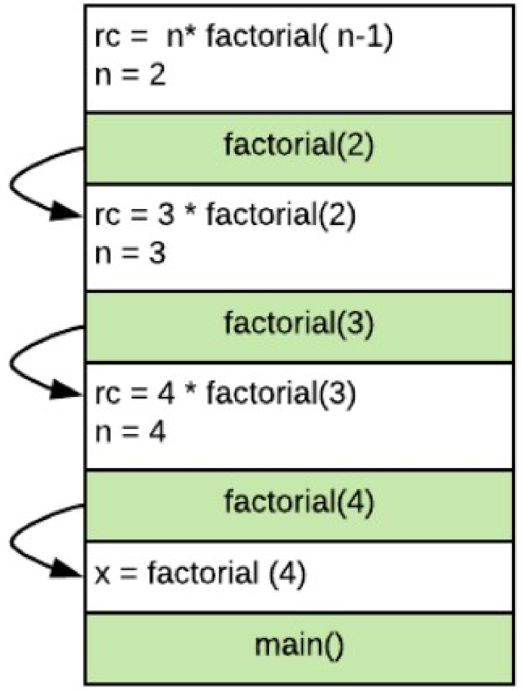
factorial(4) is called, so we push factorial(4) onto the stack along with local variables n and rc.

HOW DOES RECURSION WORK? CONT.



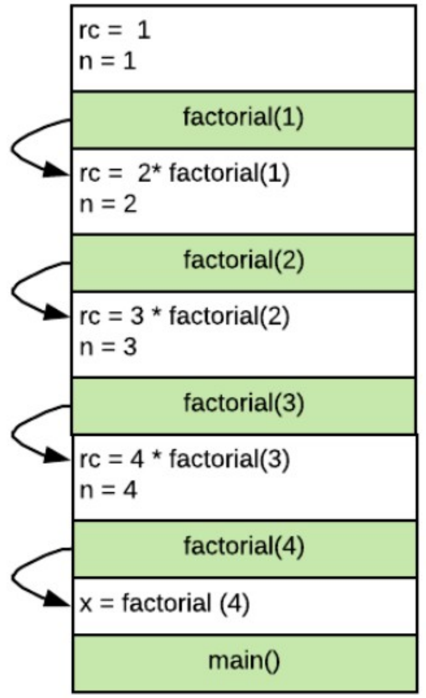
n is 4, and thus the if statement in line 3 is true, thus we need to call factorial(3) to complete line 4.



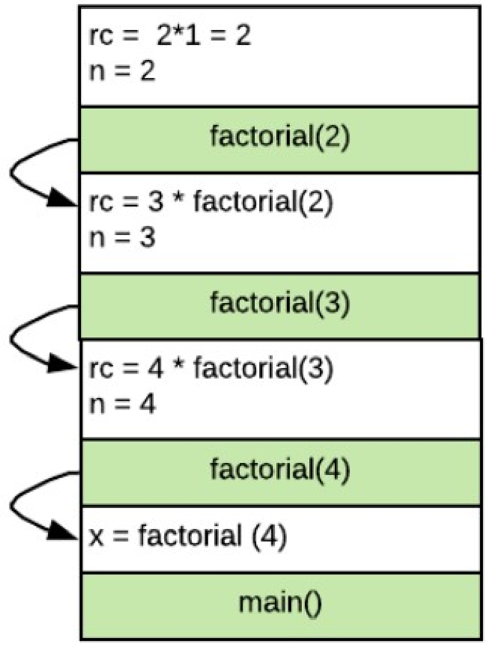


Once again, the if statement is true, and thus, we need to call factorial(1).

HOW DOES RECURSION WORK? CONT.

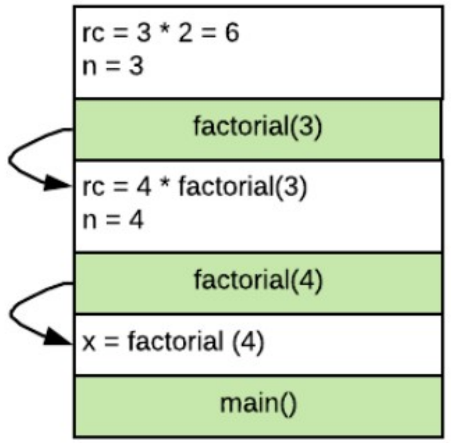


Once we make this call though, our if statment is false. Thus, we return rc popping the stack

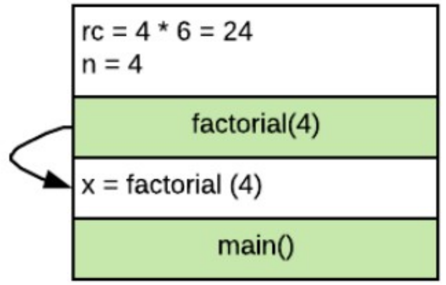


Once it is returned we can complete our calculation for rc = 2. This is returned, popping the stack

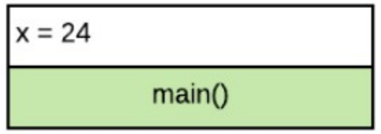
HOW DOES RECURSION WORK? CONT.



Once it is returned we can complete our calculation for rc = 6. This is returned, popping the stack



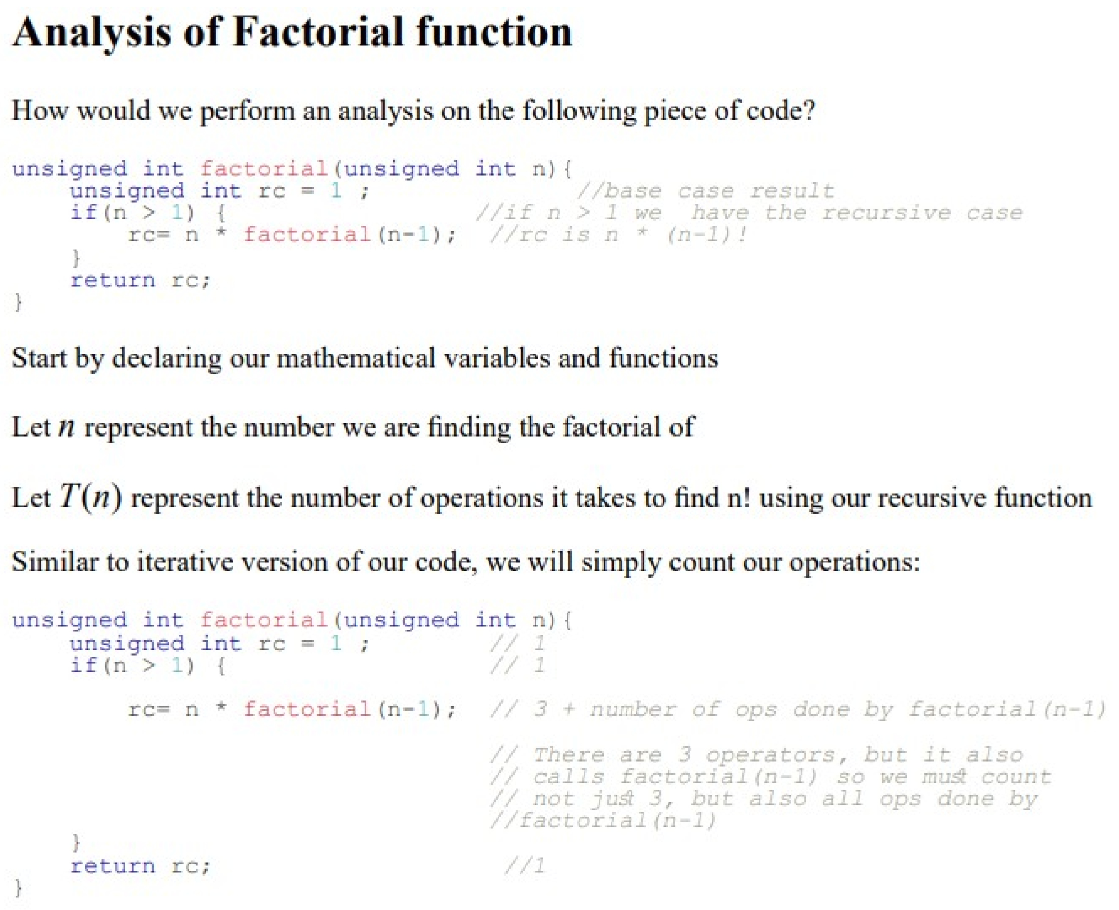
Once it is returned we can complete our calculation for rc = 24. This is returned, popping the stack



ANALYSIS OF A RECURSIVE FUNCTION

- Performing an analysis of a recursive function is not all that different from performing an analysis of a non-recursive function.
- We are sill going to use the same methodology to find a formula that will represent the number of operations required for a given data size.
- We will then use that formula to find a curve that best describes the resource usage growth rate.

ANALYSIS OF A RECURSIVE FUNCTION CONT.



Analysis of Factorial function

How would we perform an analysis on the following piece of code?

```
unsigned int factorial(unsigned int n){
    unsigned int rc = 1 ;            //base case result
    if(n > 1) {                   //if n > 1 we  have the recursive case
        rc= n * factorial(n-1);  //rc is n * (n-1)!
    }
    return rc;
}
```

Start by declaring our mathematical variables and functions

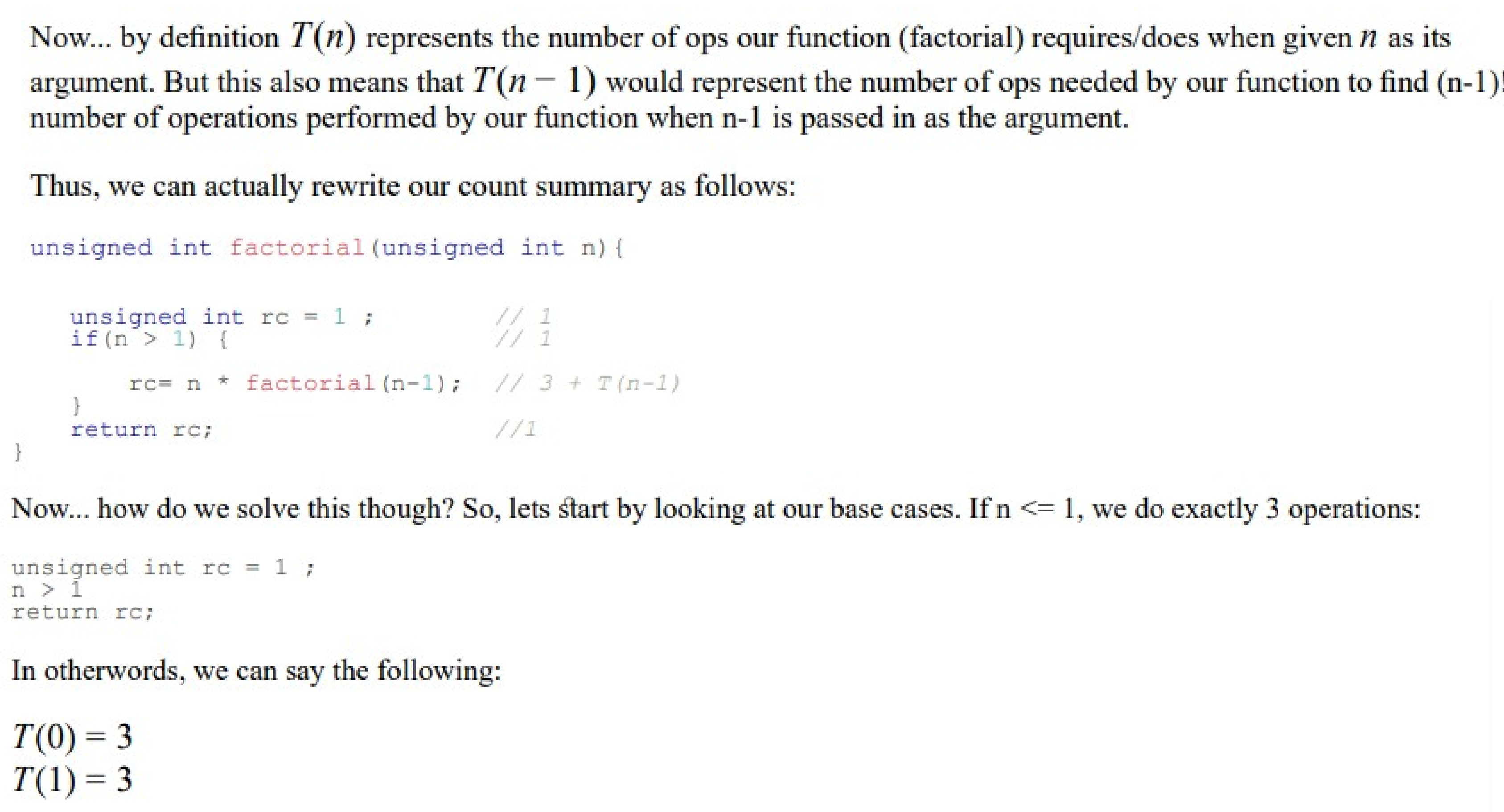
Let n represent the number we are finding the factorial of

Let $T(n)$ represent the number of operations it takes to find n! using our recursive function

Similar to iterative version of our code, we will simply count our operations:

```
unsigned int factorial(unsigned int n){
    unsigned int rc = 1 ;         // 1
    if(n > 1) {                   // 1

        rc= n * factorial(n-1);   // 3 + number of ops done by factorial(n-1)

                                  // There are 3 operators, but it also
                                  // calls factorial(n-1) so we must count
                                  // not just 3, but also all ops done by
                                  //factorial(n-1)
    }
    return rc;                     //1
}
```

ANALYSIS OF A RECURSIVE FUNCTION CONT.



Now... by definition $T(n)$ represents the number of ops our function (factorial) requires/does when given n as its argument. But this also means that $T(n-1)$ would represent the number of ops needed by our function to find (n-1)! (ie number of operations performed by our function when n-1 is passed in as the argument.

Thus, we can actually rewrite our count summary as follows:

```
unsigned int factorial(unsigned int n){

   unsigned int rc = 1 ;          // 1
   if(n > 1) {                    // 1

      rc= n * factorial(n-1);     // 3 + T(n-1)
   }
   return rc;                     //1
}
```

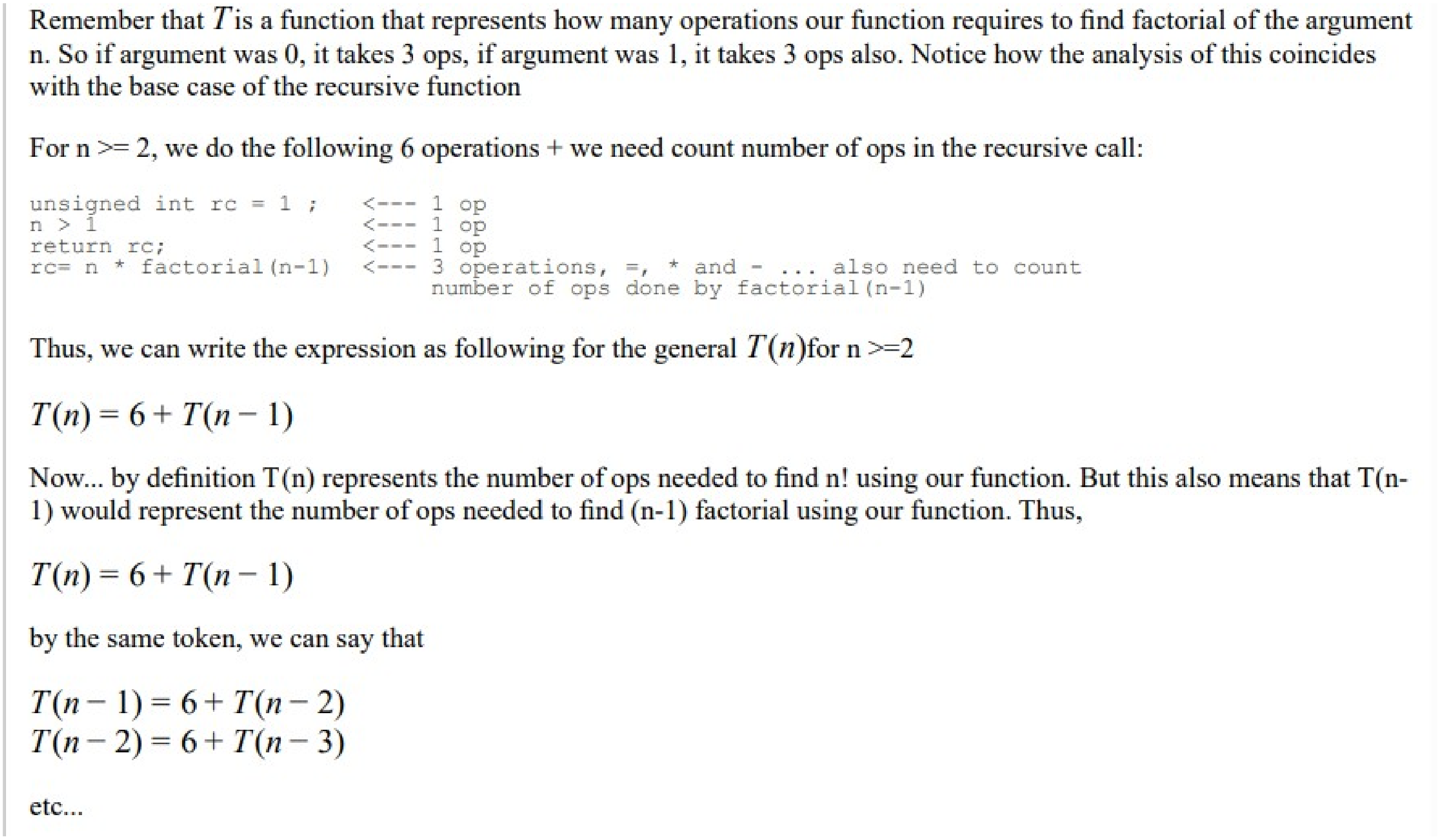
Now... how do we solve this though? So, lets start by looking at our base cases. If n <= 1, we do exactly 3 operations:

```
unsigned int rc = 1 ;
n > 1
return rc;
```

In otherwords, we can say the following:

$$T(0) = 3$$
$$T(1) = 3$$

ANALYSIS OF A RECURSIVE FUNCTION CONT.



Remember that T is a function that represents how many operations our function requires to find factorial of the argument n. So if argument was 0, it takes 3 ops, if argument was 1, it takes 3 ops also. Notice how the analysis of this coincides with the base case of the recursive function

For n >= 2, we do the following 6 operations + we need count number of ops in the recursive call:

```
unsigned int rc = 1 ;    <--- 1 op
n > 1                    <--- 1 op
return rc;               <--- 1 op
rc= n * factorial(n-1)   <--- 3 operations, =, * and - ... also need to count
                              number of ops done by factorial(n-1)
```

Thus, we can write the expression as following for the general $T(n)$ for n >=2

$$T(n) = 6 + T(n-1)$$

Now... by definition T(n) represents the number of ops needed to find n! using our function. But this also means that T(n-1) would represent the number of ops needed to find (n-1) factorial using our function. Thus,

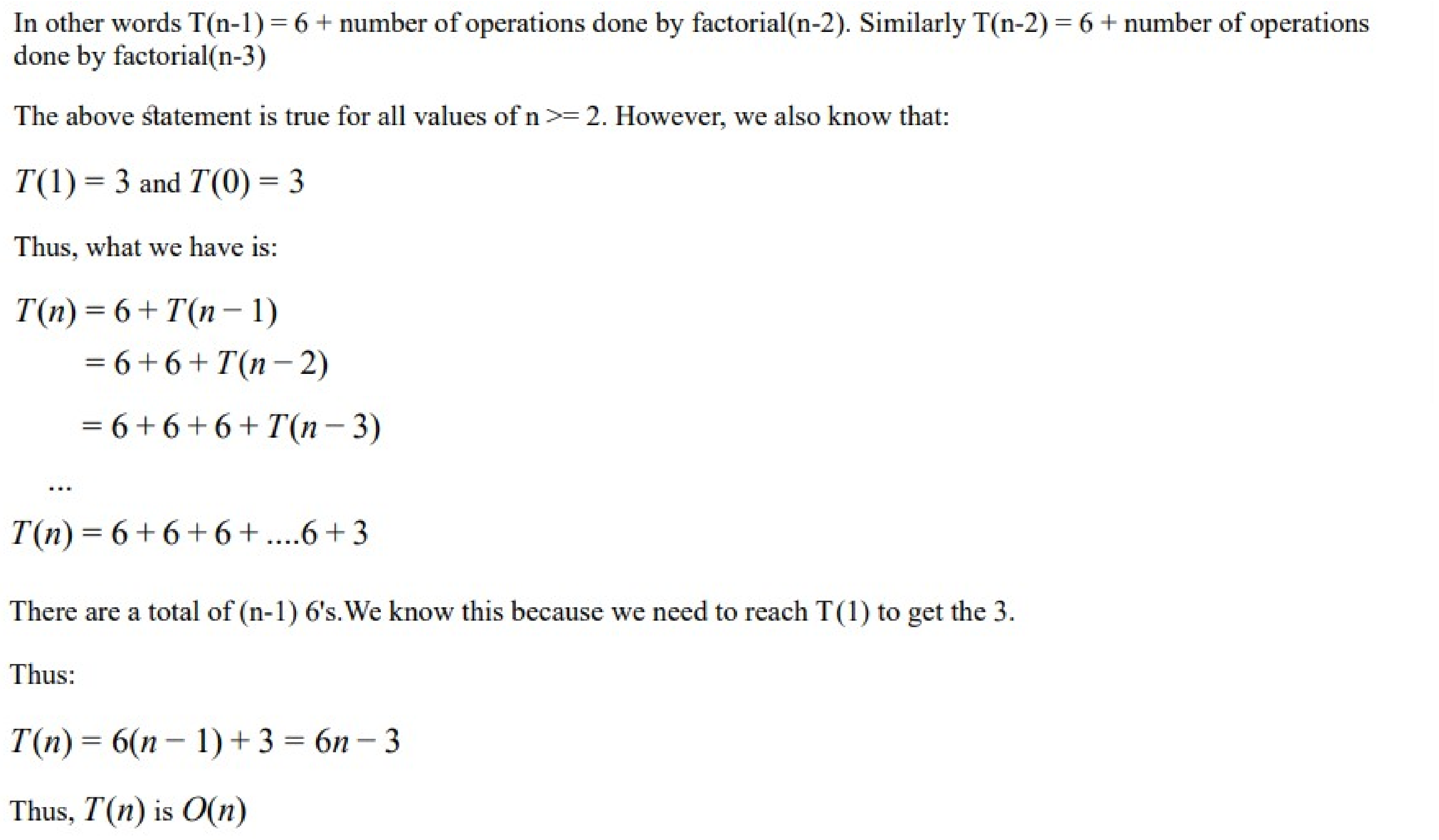
$$T(n) = 6 + T(n-1)$$

by the same token, we can say that

$$T(n-1) = 6 + T(n-2)$$
$$T(n-2) = 6 + T(n-3)$$

etc...

ANALYSIS OF A RECURSIVE FUNCTION CONT.



In other words T(n-1) = 6 + number of operations done by factorial(n-2). Similarly T(n-2) = 6 + number of operations done by factorial(n-3)

The above statement is true for all values of n >= 2. However, we also know that:

$T(1) = 3$ and $T(0) = 3$

Thus, what we have is:

$$
\begin{aligned}
T(n) &= 6 + T(n-1) \\
&= 6 + 6 + T(n-2) \\
&= 6 + 6 + 6 + T(n-3) \\
&\ldots
\end{aligned}
$$

$$T(n) = 6 + 6 + 6 + \ldots.6 + 3$$

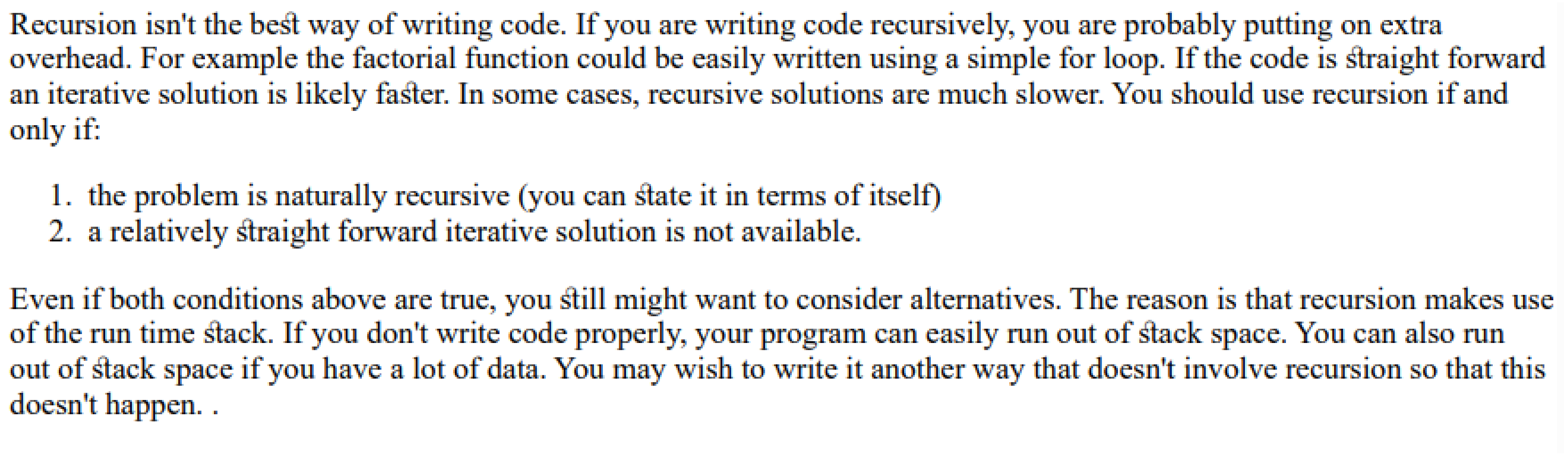
There are a total of (n-1) 6's.We know this because we need to reach T(1) to get the 3.

Thus:

$$T(n) = 6(n-1) + 3 = 6n - 3$$

Thus, $T(n)$ is $O(n)$

DRAWBACKS OF RECURSION AND CAUTION



Recursion isn't the best way of writing code. If you are writing code recursively, you are probably putting on extra overhead. For example the factorial function could be easily written using a simple for loop. If the code is straight forward an iterative solution is likely faster. In some cases, recursive solutions are much slower. You should use recursion if and only if:

1. the problem is naturally recursive (you can state it in terms of itself)
2. a relatively straight forward iterative solution is not available.

Even if both conditions above are true, you still might want to consider alternatives. The reason is that recursion makes use of the run time stack. If you don't write code properly, your program can easily run out of stack space. You can also run out of stack space if you have a lot of data. You may wish to write it another way that doesn't involve recursion so that this doesn't happen. .

MORE RECURSION EXAMPLES

Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2 .

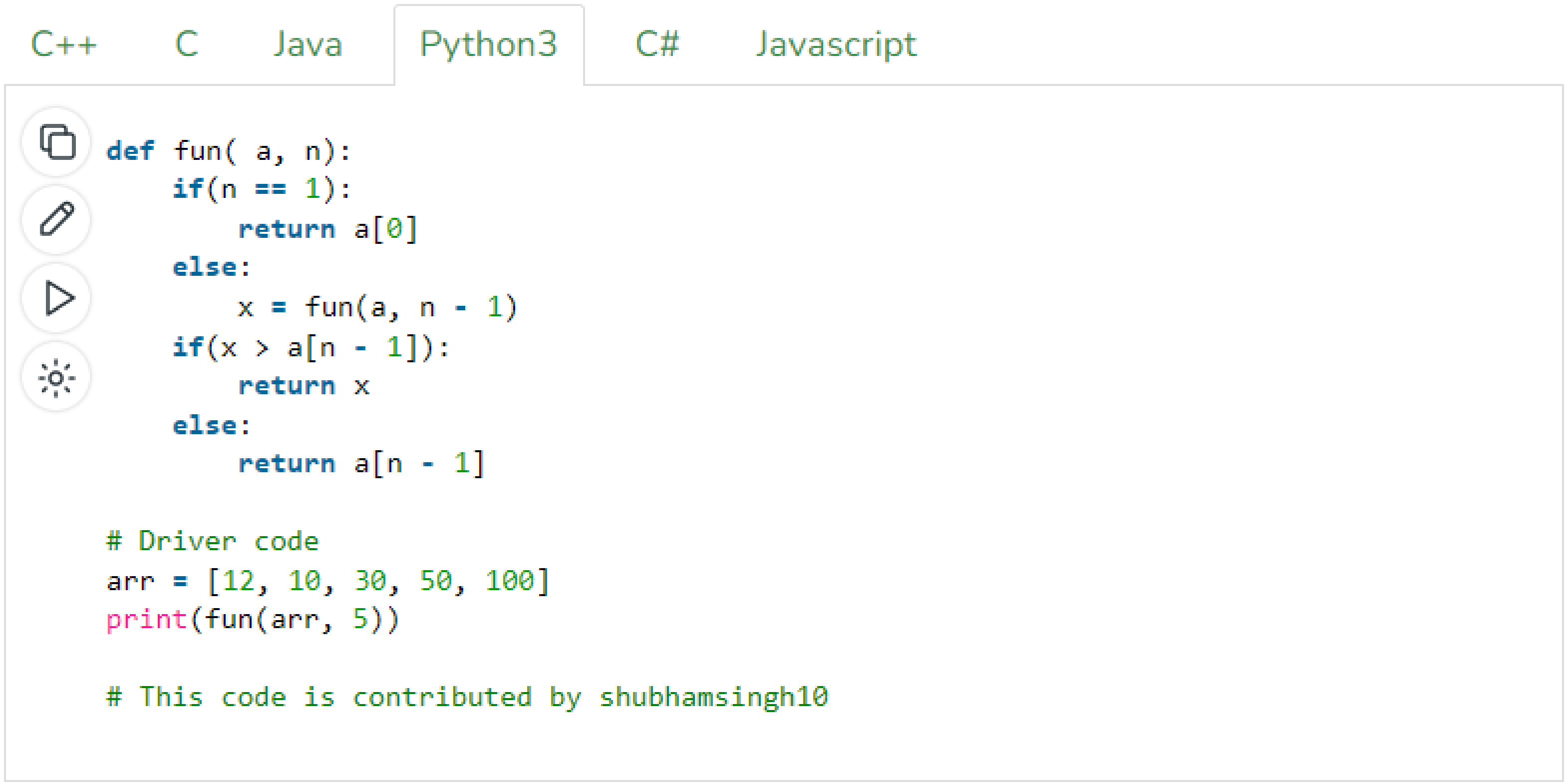
Mathematical Equation:

```
n if n == 0, n == 1;
fib(n) = fib(n-1) + fib(n-2) otherwise;
```

MORE RECURSION EXAMPLES

Question 2

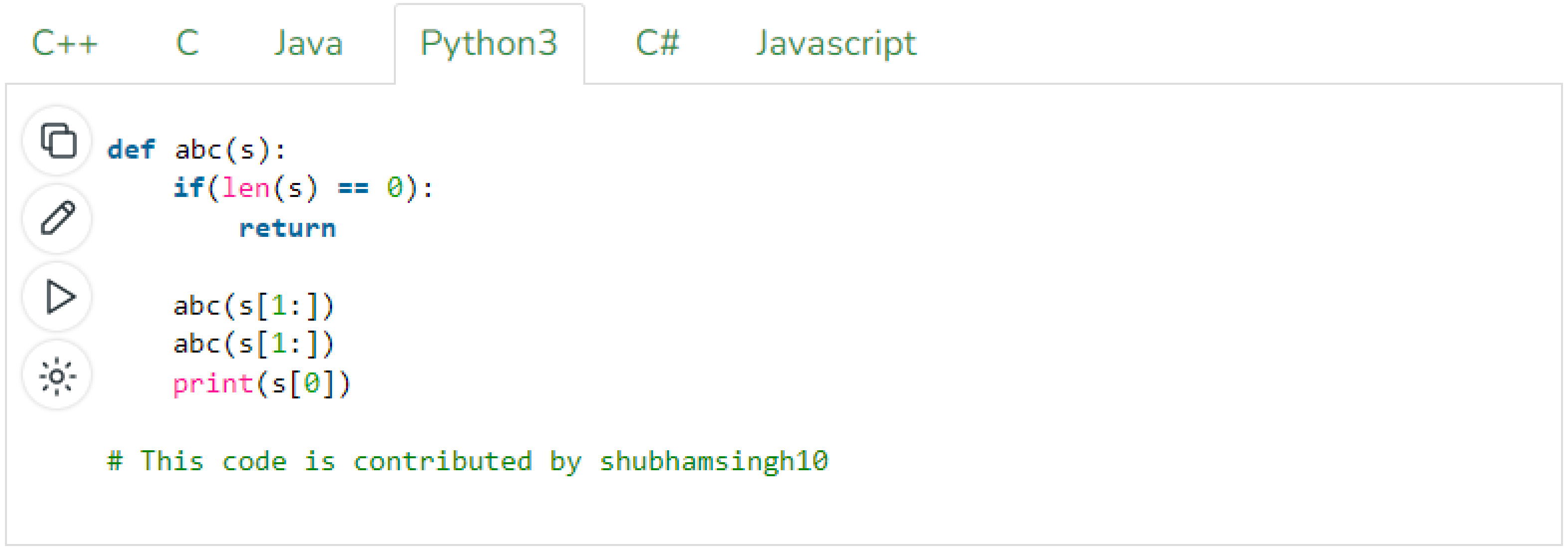
Predict the output of the following program. What does the following fun() do in general?



```python
def fun( a, n):
    if(n == 1):
        return a[0]
    else:
        x = fun(a, n - 1)
    if(x > a[n - 1]):
        return x
    else:
        return a[n - 1]

# Driver code
arr = [12, 10, 30, 50, 100]
print(fun(arr, 5))

# This code is contributed by shubhamsingh10
```

MORE RECURSION EXAMPLES

Consider the following recursive C function. Let *len* be the length of the string s and *num* be the number of characters printed on the screen. Give the relation between *num* and *len* where *len* is always greater than 0.



```python
def abc(s):
    if(len(s) == 0):
        return

    abc(s[1:])
    abc(s[1:])
    print(s[0])

# This code is contributed by shubhamsingh10
```

SEARCHING AND SORTING

- Searching is the process of finding particular piece of data (the key) within a data structure.
- Sorting is the process of taking a set of data and creating an ordering based on some criteria.

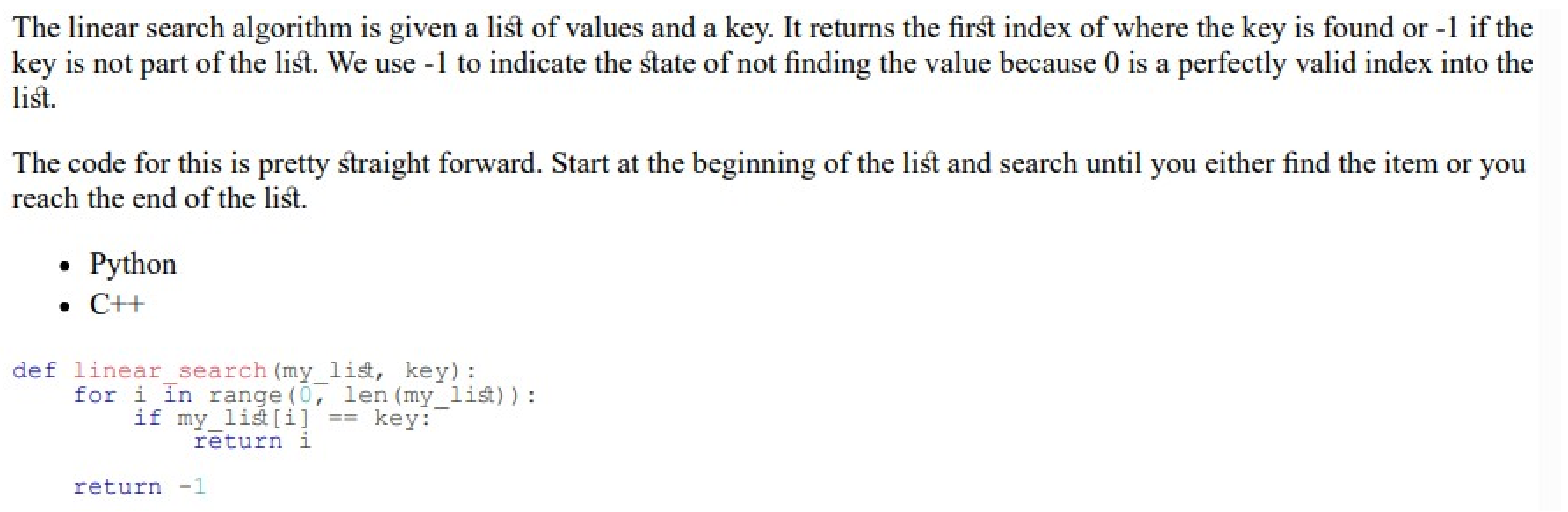
SEARCHING

- Searching is the process of finding particular piece of data (the key) within a data structure. Some data structures will support faster searching by the way that the data is sorted and organized
- In this section of the notes we are concerned with how to search a list. In particular we are looking at lists that are array like. For example, Python lists are "array like", as are vectors from the C++ standard library. The list from C++ standard library however is not array based so this is not the type of list we are considering in this chapter. What we want are data structures that provide fast random access to any element given its index

TWO MAIN TYPES OF SEARCHING

- Linear Search
- Binary Search

LINEAR SEARCH



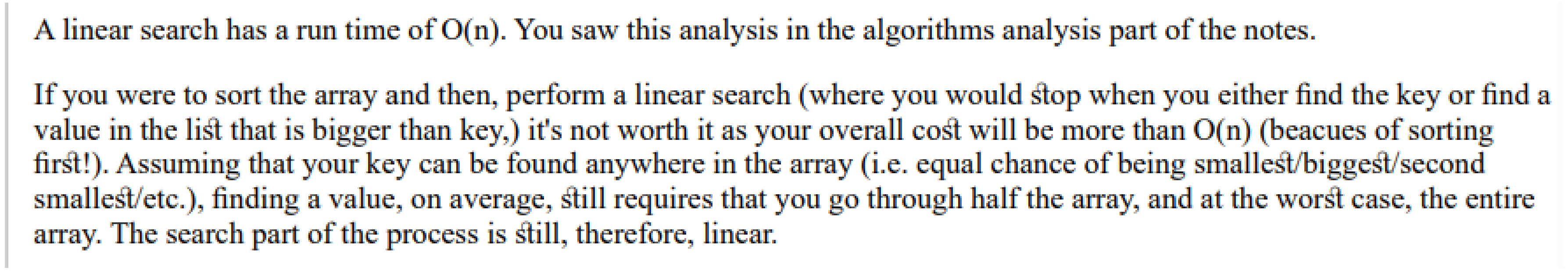
The linear search algorithm is given a list of values and a key. It returns the first index of where the key is found or -1 if the key is not part of the list. We use -1 to indicate the state of not finding the value because 0 is a perfectly valid index into the list.

The code for this is pretty straight forward. Start at the beginning of the list and search until you either find the item or you reach the end of the list.

- Python
- C++

```
def linear_search(my_list, key):
    for i in range(0, len(my_list)):
        if my_list[i] == key:
            return i

    return -1
```

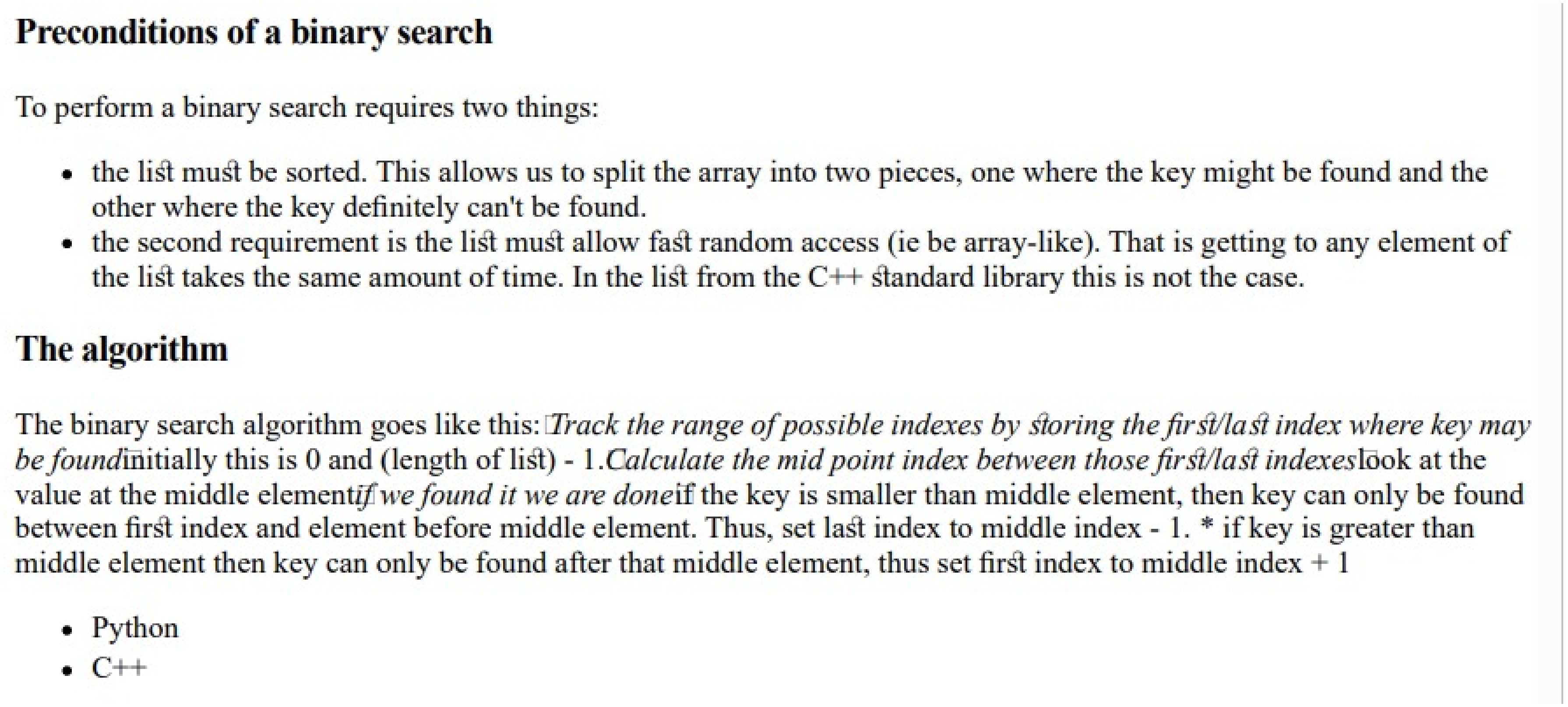
PERFORMANCE OF A LINEAR SEARCH



A linear search has a run time of O(n). You saw this analysis in the algorithms analysis part of the notes.

If you were to sort the array and then, perform a linear search (where you would stop when you either find the key or find a value in the list that is bigger than key,) it's not worth it as your overall cost will be more than O(n) (beacues of sorting first!). Assuming that your key can be found anywhere in the array (i.e. equal chance of being smallest/biggest/second smallest/etc.), finding a value, on average, still requires that you go through half the array, and at the worst case, the entire array. The search part of the process is still, therefore, linear.

BINARY SEARCH



Preconditions of a binary search

To perform a binary search requires two things:

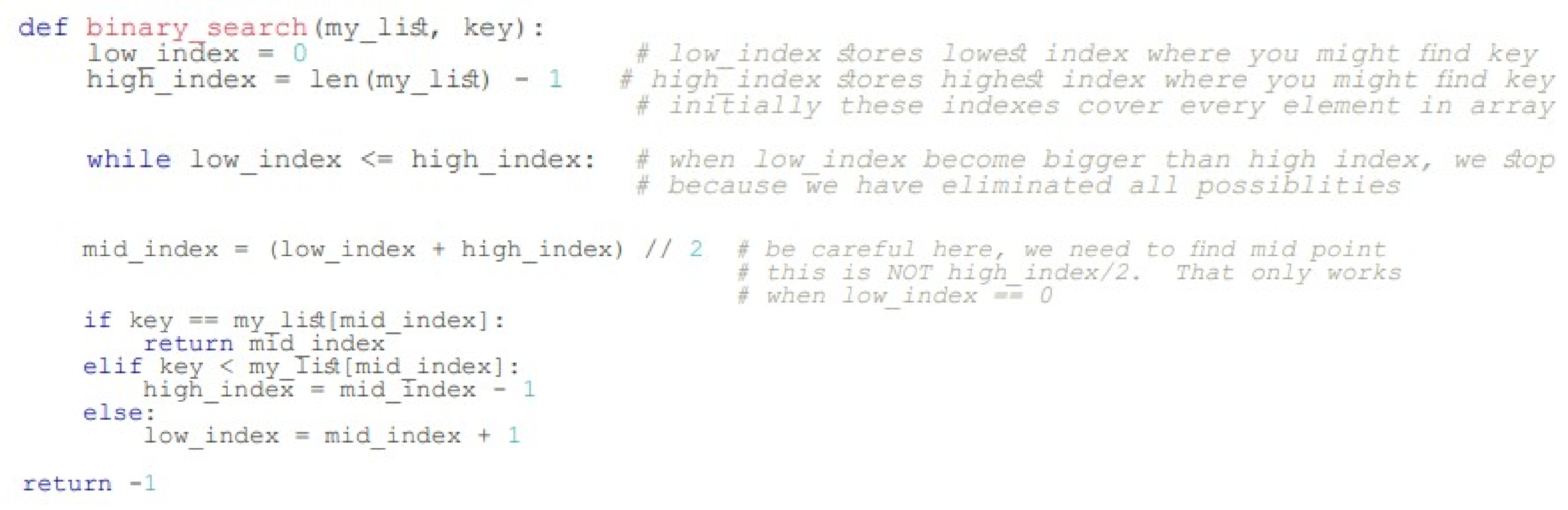
- the list must be sorted. This allows us to split the array into two pieces, one where the key might be found and the other where the key definitely can't be found.
- the second requirement is the list must allow fast random access (ie be array-like). That is getting to any element of the list takes the same amount of time. In the list from the C++ standard library this is not the case.

The algorithm

The binary search algorithm goes like this: *Track the range of possible indexes by storing the first/last index where key may be found* initially this is 0 and (length of list) - 1. *Calculate the mid point index between those first/last indexes* look at the value at the middle element *if we found it we are done* if the key is smaller than middle element, then key can only be found between first index and element before middle element. Thus, set last index to middle index - 1. * if key is greater than middle element then key can only be found after that middle element, thus set first index to middle index + 1

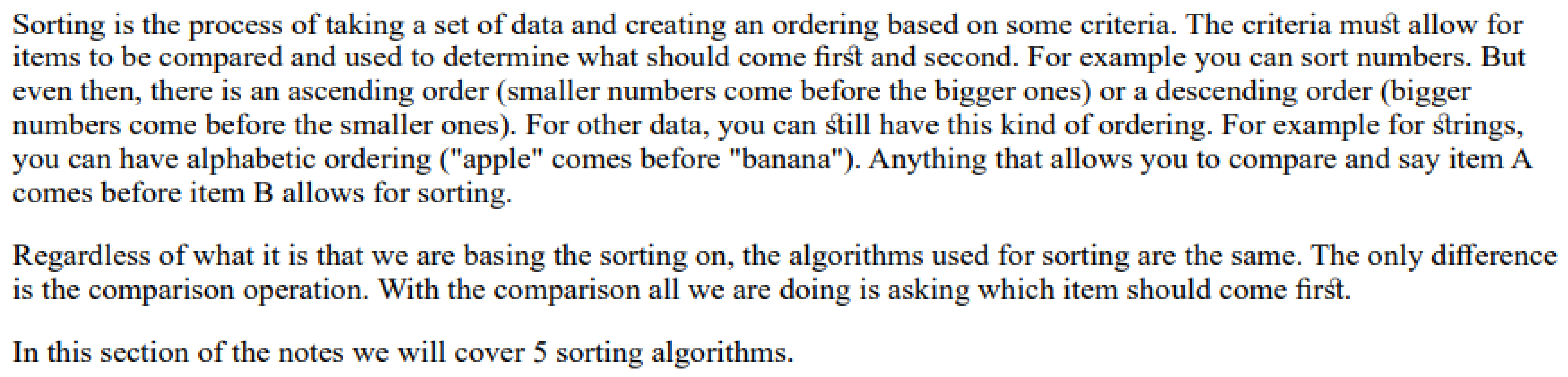
- Python
- C++

BINARY SEARCH



```
def binary_search(my_list, key):
    low_index = 0                       # low_index stores lowest index where you might find key
    high_index = len(my_list) - 1      # high_index stores highest index where you might find key
                                        # initially these indexes cover every element in array

    while low_index <= high_index:      # when low_index become bigger than high_index, we stop
                                        # because we have eliminated all possiblities

    mid_index = (low_index + high_index) // 2   # be careful here, we need to find mid point
                                                # this is NOT high_index/2.  That only works
                                                # when low_index == 0

    if key == my_list[mid_index]:
        return mid_index
    elif key < my_list[mid_index]:
        high_index = mid_index - 1
    else:
        low_index = mid_index + 1

return -1
```

SORTING



Sorting is the process of taking a set of data and creating an ordering based on some criteria. The criteria must allow for items to be compared and used to determine what should come first and second. For example you can sort numbers. But even then, there is an ascending order (smaller numbers come before the bigger ones) or a descending order (bigger numbers come before the smaller ones). For other data, you can still have this kind of ordering. For example for strings, you can have alphabetic ordering ("apple" comes before "banana"). Anything that allows you to compare and say item A comes before item B allows for sorting.

Regardless of what it is that we are basing the sorting on, the algorithms used for sorting are the same. The only difference is the comparison operation. With the comparison all we are doing is asking which item should come first.

In this section of the notes we will cover 5 sorting algorithms.

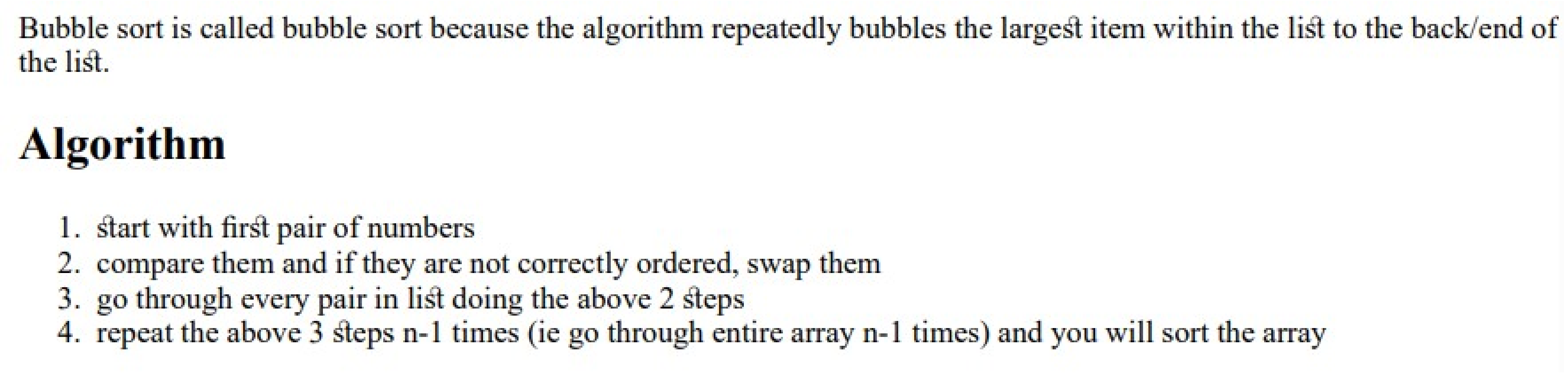
Simple sorts

- Bubble sort
- Selection sort
- Insertion sort

More complex sorts

- Merge sort
- Quick sort

BUBBLE SORT $O(N^2)$



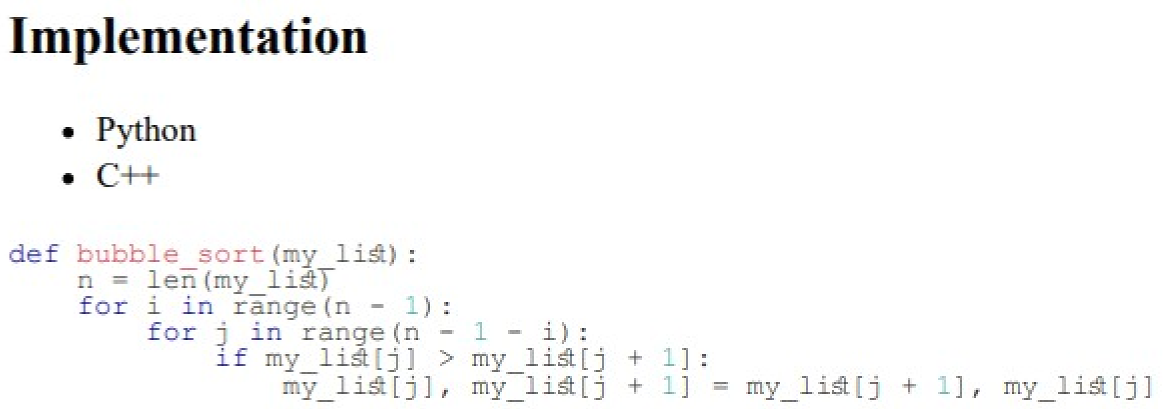
Bubble sort is called bubble sort because the algorithm repeatedly bubbles the largest item within the list to the back/end of the list.

Algorithm

1. start with first pair of numbers
2. compare them and if they are not correctly ordered, swap them
3. go through every pair in list doing the above 2 steps
4. repeat the above 3 steps n-1 times (ie go through entire array n-1 times) and you will sort the array

Data Structure Animations using Processing.js by cathyatseneca

BUBBLE SORT

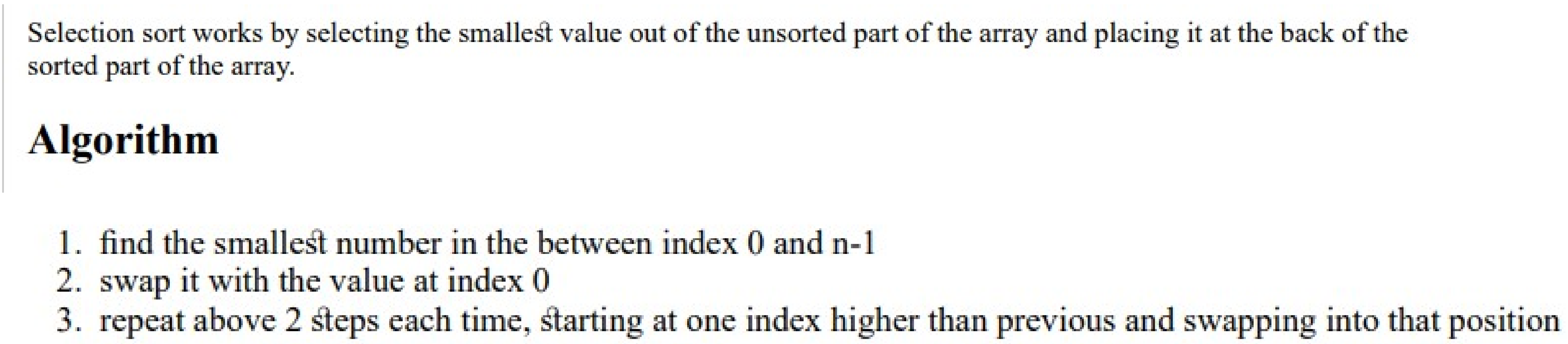


Implementation

- Python
- C++

```
def bubble_sort(my_list):
    n = len(my_list)
    for i in range(n - 1):
        for j in range(n - 1 - i):
            if my_list[j] > my_list[j + 1]:
                my_list[j], my_list[j + 1] = my_list[j + 1], my_list[j]
```

SELECTION SORT $O(N^2)$



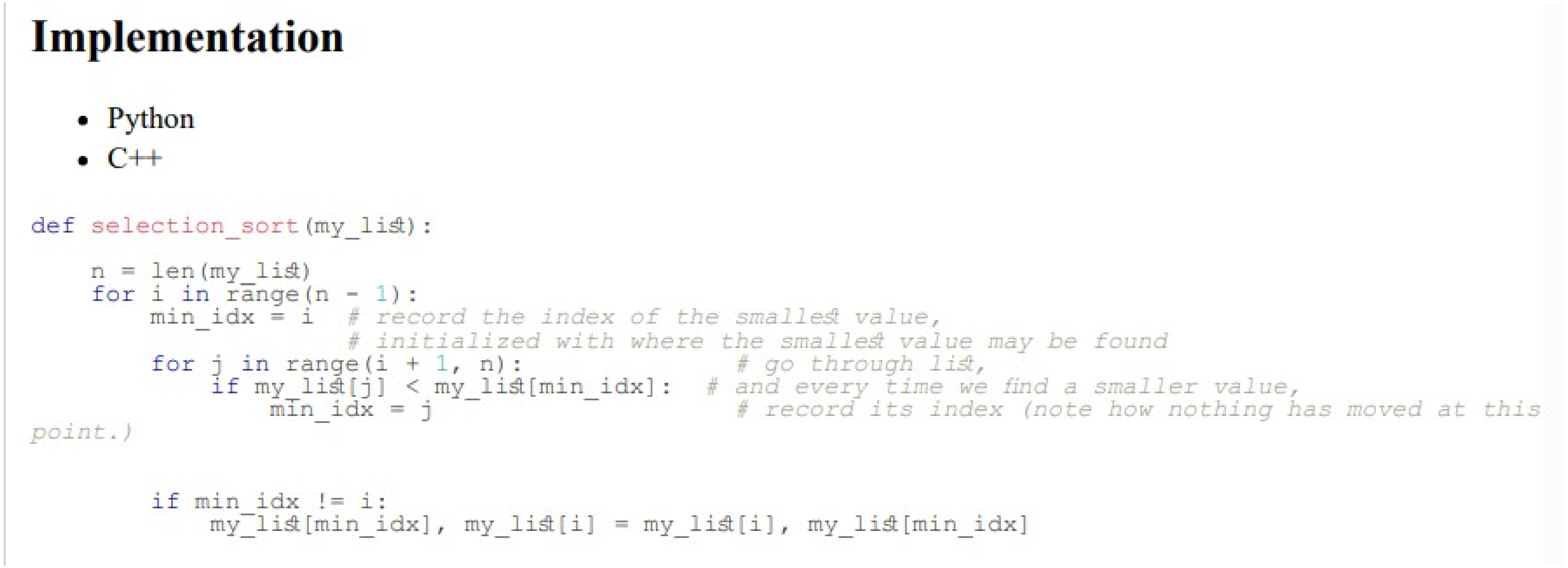
Selection sort works by selecting the smallest value out of the unsorted part of the array and placing it at the back of the sorted part of the array.

Algorithm

1. find the smallest number in the between index 0 and n-1
2. swap it with the value at index 0
3. repeat above 2 steps each time, starting at one index higher than previous and swapping into that position

Data Structure Animations using Processing.js by cathyatseneca

SELECTION SORT

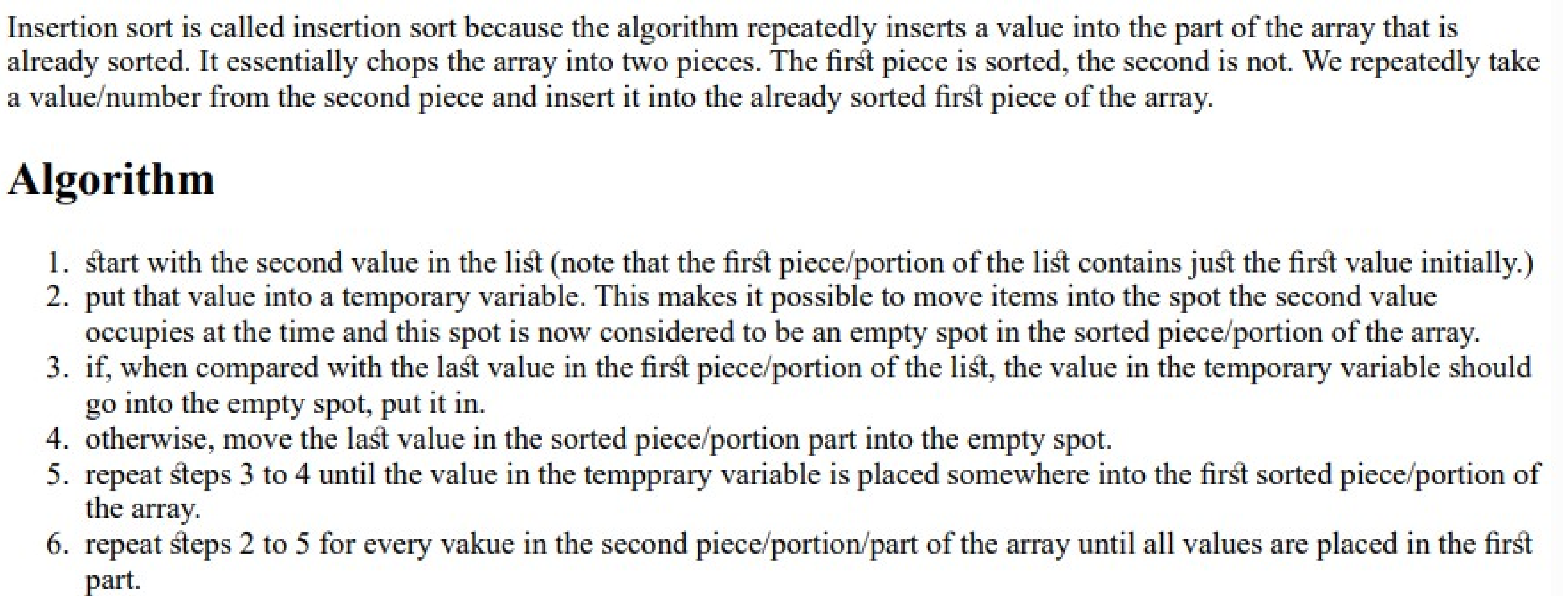


Implementation

- Python
- C++

```
def selection_sort(my_list):

    n = len(my_list)
    for i in range(n - 1):
        min_idx = i  # record the index of the smallest value,
                     # initialized with where the smallest value may be found
        for j in range(i + 1, n):                # go through list,
            if my_list[j] < my_list[min_idx]:  # and every time we find a smaller value,
                min_idx = j                      # record its index (note how nothing has moved at this
point.)


        if min_idx != i:
            my_list[min_idx], my_list[i] = my_list[i], my_list[min_idx]
```

INSERTION SORT $O(N^2)$



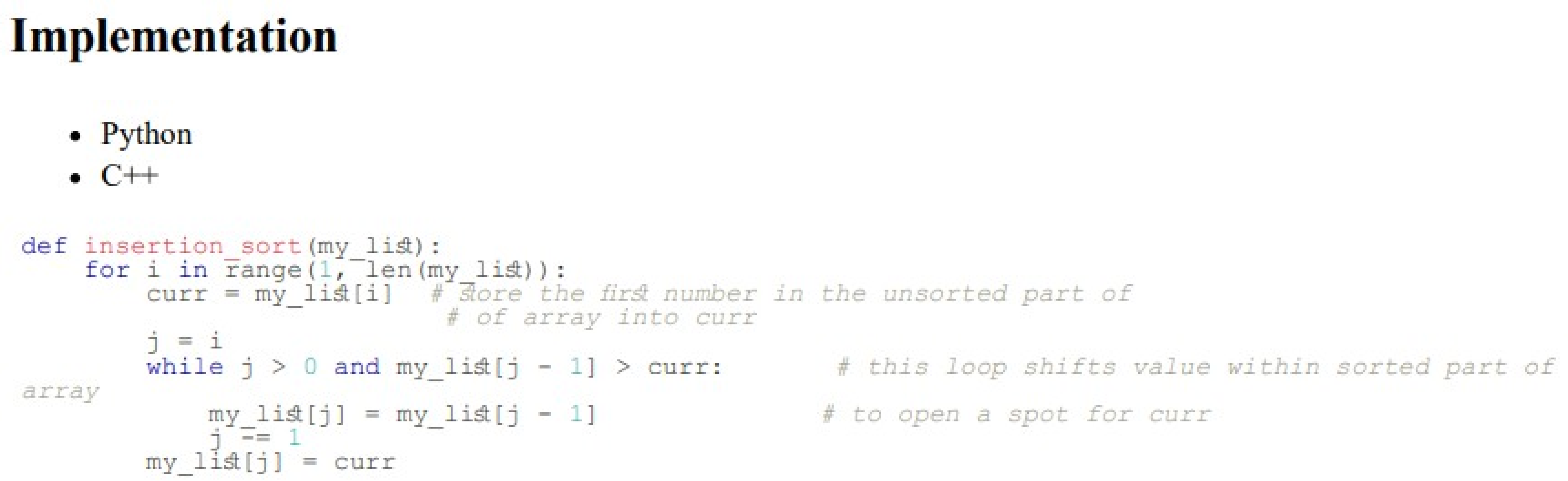
Insertion sort is called insertion sort because the algorithm repeatedly inserts a value into the part of the array that is already sorted. It essentially chops the array into two pieces. The first piece is sorted, the second is not. We repeatedly take a value/number from the second piece and insert it into the already sorted first piece of the array.

Algorithm

1. start with the second value in the list (note that the first piece/portion of the list contains just the first value initially.)
2. put that value into a temporary variable. This makes it possible to move items into the spot the second value occupies at the time and this spot is now considered to be an empty spot in the sorted piece/portion of the array.
3. if, when compared with the last value in the first piece/portion of the list, the value in the temporary variable should go into the empty spot, put it in.
4. otherwise, move the last value in the sorted piece/portion part into the empty spot.
5. repeat steps 3 to 4 until the value in the tempprary variable is placed somewhere into the first sorted piece/portion of the array.
6. repeat steps 2 to 5 for every vakue in the second piece/portion/part of the array until all values are placed in the first part.

Data Structure Animations using Processing.js by cathyatseneca

INSERTION SORT



Implementation

- Python
- C++

```
def insertion_sort(my_list):
    for i in range(1, len(my_list)):
        curr = my_list[i]   # store the first number in the unsorted part of
                             # of array into curr
        j = i
        while j > 0 and my_list[j - 1] > curr:       # this loop shifts value within sorted part of
array
            my_list[j] = my_list[j - 1]              # to open a spot for curr
            j -= 1
        my_list[j] = curr
```

END OF WEEK3

- Don’t forget to work on your lab 2 and assignment 1