1 Introduction Recursion

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

- complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Syntax:

```
def func(): <--
              |
              | (recursive call)
              |
    func() ----
```

Python Algorithm:

Algorithm is **a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output**.

Recursion: (definition) noun. See recursion. Recursion: Formal Definition: An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task. • Recursive solutions involve two major parts: 1. Base case(s), in which the problem is simple enough to be solved directly. 2. Recursive case(s). A recursive case has three components: (a) Divide the problem into one or more simpler or smaller parts of the problems, (b) Invoke the function (recursively) on each part, and (c) Combine the solutions of the parts into a solution for the problem. • Depending on the problem, any of these may be trivial or complex.

Example: Sum A non-recursive example:

```
def it_sum(a_list):
   result = 0
   for x in a_list:
      result += x
   return result
```

We say that the above function iterates over the values in the variable a list, and returns their sum. Recursion is similar to iteration, such that the operation being performed is defined (partly) in terms of itself. Such an operation is said to be recursive. Here is a recursive definition of the sum() function:

```
def rec_sum(a_list):
   if a_list == []:
      return 0
   else:
      return a_list[0] + rec_sum(a_list[1:])
```

rec sum computes the same exact thing as it sum, but in a different way. The first thing to note is that it does not use a for-loop. The second thing to note is that the rec sum function calls itself. That is to say, rec sum() is defined in terms of itself; it is recursive. How does it work? Let's go through the parts of recursion mentioned at the introduction to this handout.

1. Base Case: What is the base case of rec sum?

2. Recursive case: (a) How do we divide the problem? (b) Where do we invoke the function recursively? (c) Finally, where do we combine the solutions?

rec_sum([1, 2, 3])

= 1 + rec_sum([2, 3])

= 1 + (2 + rec_sum([3]))

= 1 + (2 + (3 + rec_sum([]))

= 1 + (2 + (3 + 0))

= 1 + (2 + 3)

= 1 + 5

= 6

Note that our base case is when the list is empty. That is the recursive call to rec sum([]), which evaluates to 0. A base case is very important - it is the stopping point for recursion. The recursive case is demonstrated by calls to rec sum where the argument is a non-empty list. During a recursive case, we make incremental progress towards solving the problem, and also make a recursive call to the function with a smaller input space.

2.Search

Searching is a very basic necessity when you store data. The simplest approach is to go across every element and match it with the value you are searching for.

2.1 Binary search:

A binary search is an algorithm to find a particular element in the list. Suppose we have a list of thousand elements, and we need to get an index position of a particular element. We can find the element's index position very fast using the binary search algorithm.

There are many searching algorithms but the binary search is most popular among them.

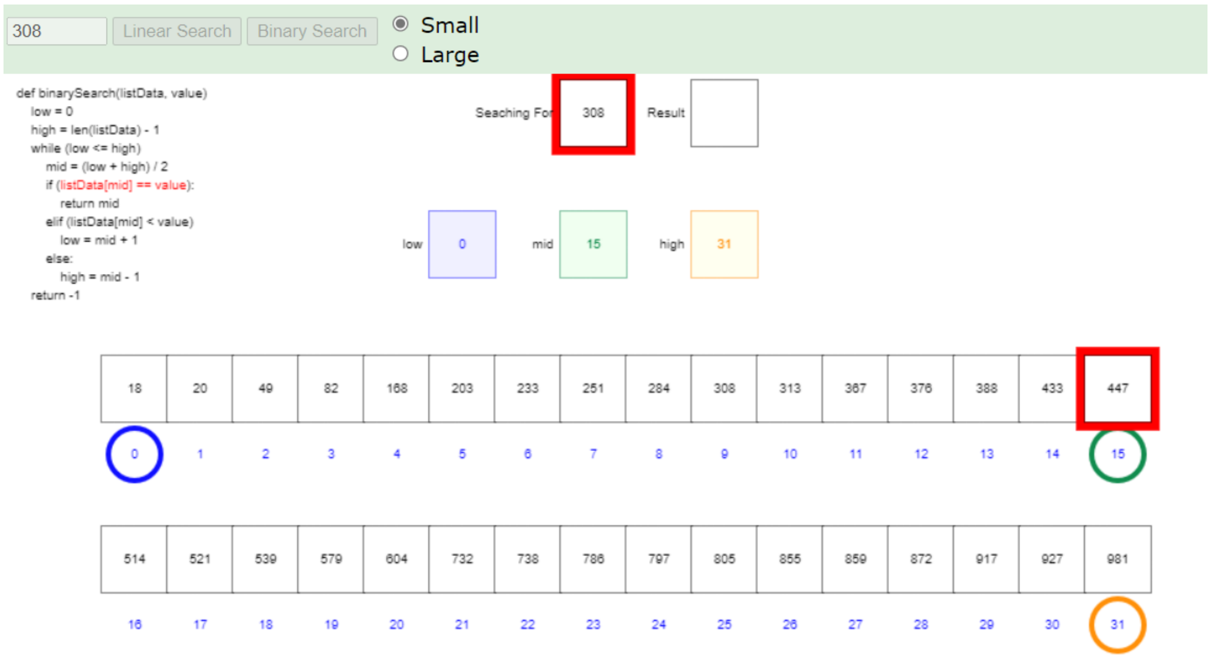
The elements in the list must be sorted to apply the binary search algorithm. If elements are not sorted then sort them first.

we are setting two pointers in our list. One pointer is used to denote the smaller value called **low** and the second pointer is used to denote the highest value called **high**.

Next, we calculate the value of the **middle** element in the array.

```
1. Here, the low is 0 and the high is 31.
2. if low <= high:
3. mid = (low+high)/2
   mid = (0+31)/2
   mid = 15 (Integer)
4. if  list1[mid] == n:
           return mid
   elif list1[mid] > n:
           high = mid - 1
   elif list1[mid] < n:
           low = mid + 1
```

n=308

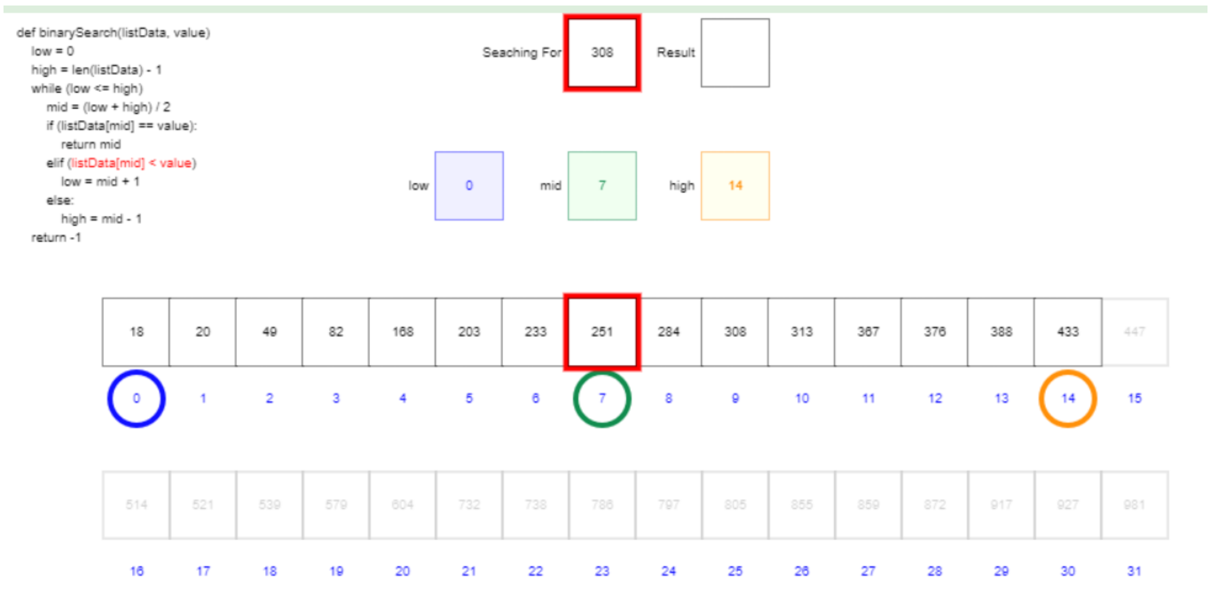


```
def binarySearch(listData, value)
  low = 0
  high = len(listData) - 1
  while (low <= high)
    mid = (low + high) / 2
    if (listData[mid] == value):
      return mid
    elif (listData[mid] < value)
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Seaching For 308 Result

low 0 mid 15 high 31

18	20	49	82	168	203	233	251	284	308	313	367	376	388	433	447
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
514	521	539	579	604	732	738	786	797	805	855	859	872	917	927	981
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

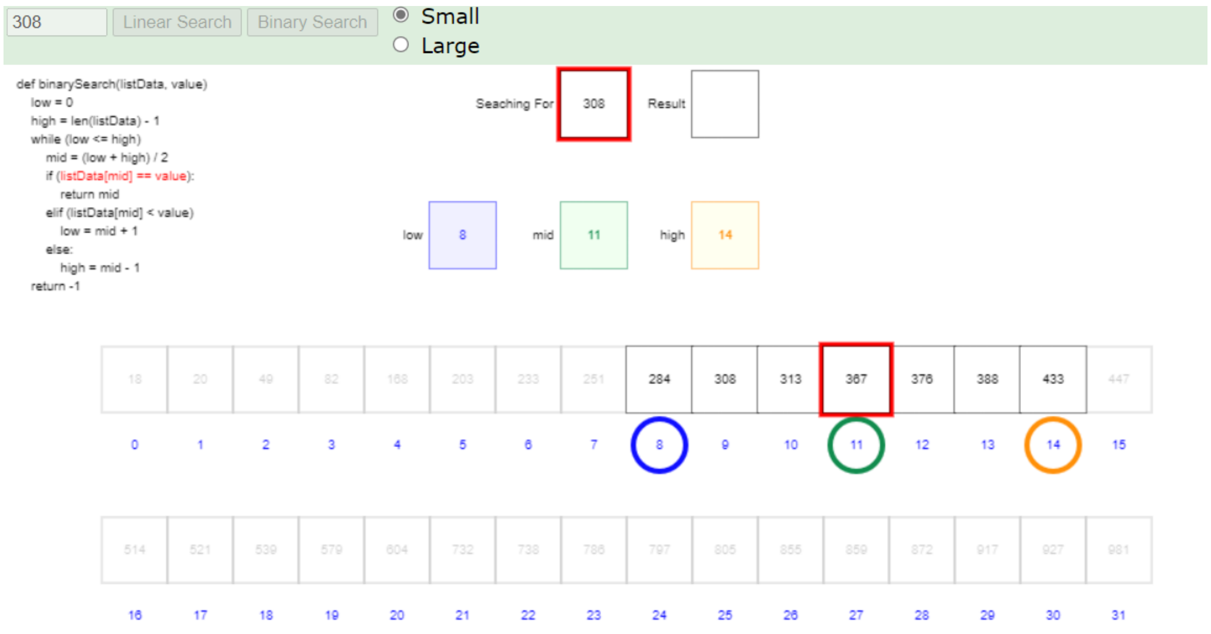


```
def binarySearch(listData, value)
  low = 0
  high = len(listData) - 1
  while (low <= high)
    mid = (low + high) / 2
    if (listData[mid] == value):
      return mid
    elif (listData[mid] < value)
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Seaching For 308 Result

low 0 mid 7 high 14

18	20	49	82	168	203	233	251	284	308	313	367	376	388	433	447
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
514	521	539	579	604	732	738	786	797	805	855	859	872	917	927	981
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

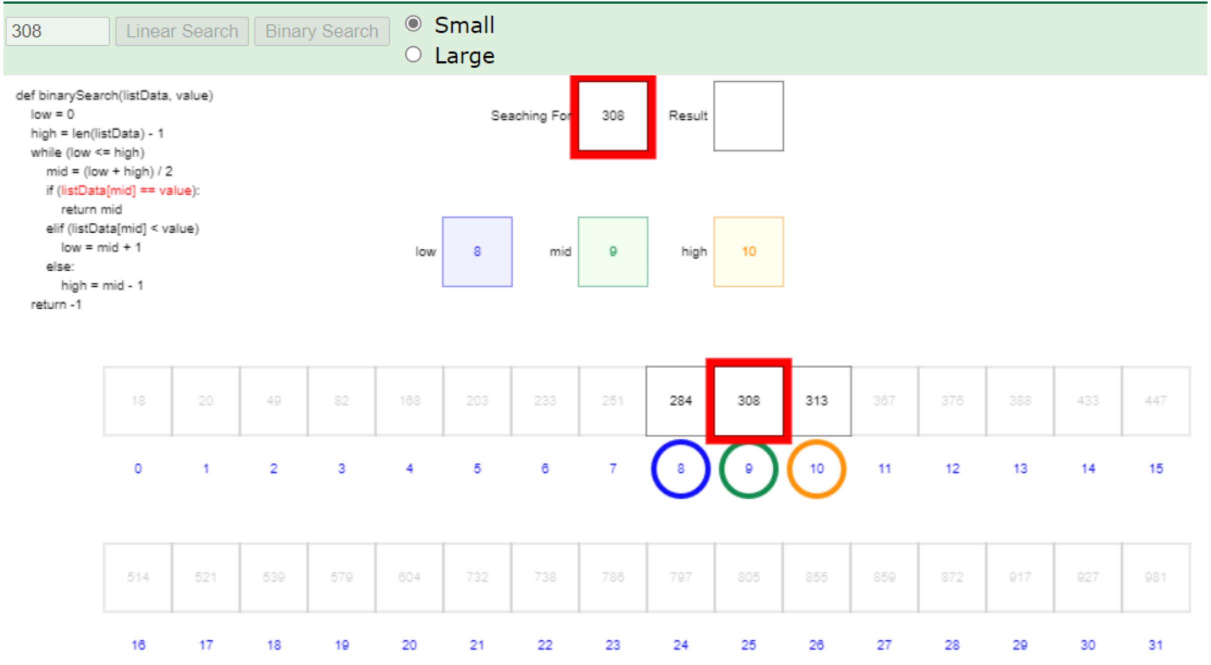


```
def binarySearch(listData, value)
  low = 0
  high = len(listData) - 1
  while (low <= high)
    mid = (low + high) / 2
    if (listData[mid] == value):
      return mid
    elif (listData[mid] < value)
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Seaching For 308 Result

low 8 mid 11 high 14

18	20	49	82	168	203	233	251	284	308	313	367	376	388	433	447
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
514	521	539	579	604	732	738	786	797	805	855	859	872	917	927	981
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

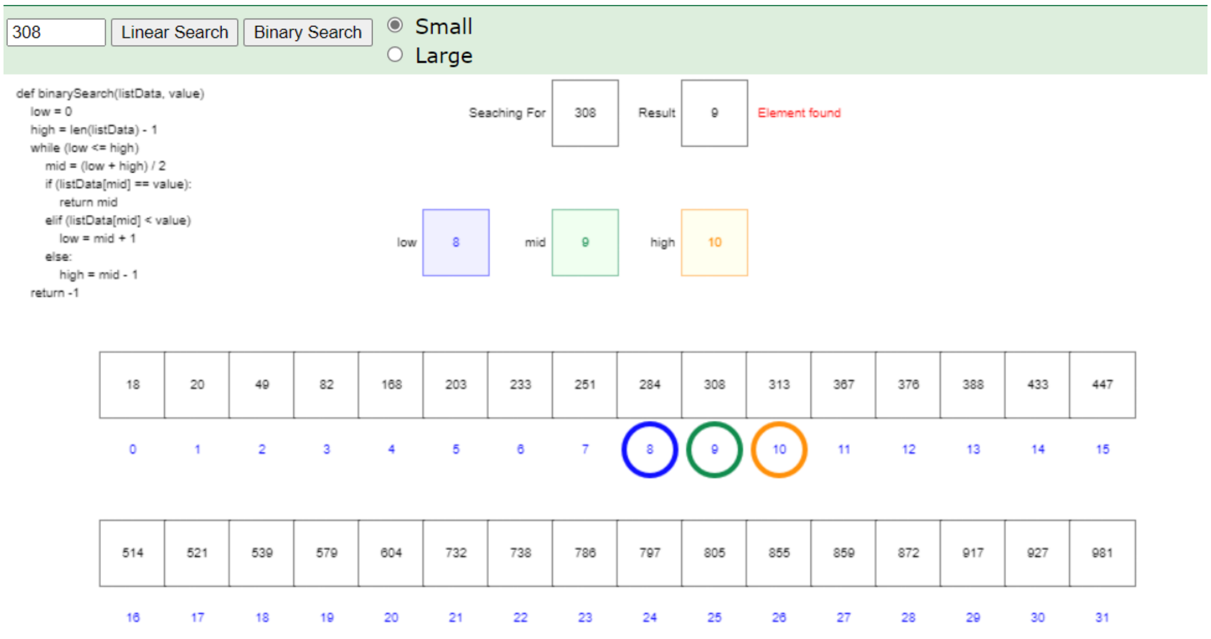


```
def binarySearch(listData, value)
  low = 0
  high = len(listData) - 1
  while (low <= high)
    mid = (low + high) / 2
    if (listData[mid] == value):
      return mid
    elif (listData[mid] < value)
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

Seaching For 308 Result

low 8 mid 9 high 10

18	20	49	82	168	203	233	251	284	308	313	367	376	388	433	447
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
514	521	539	579	604	732	738	786	797	805	855	859	872	917	927	981
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31



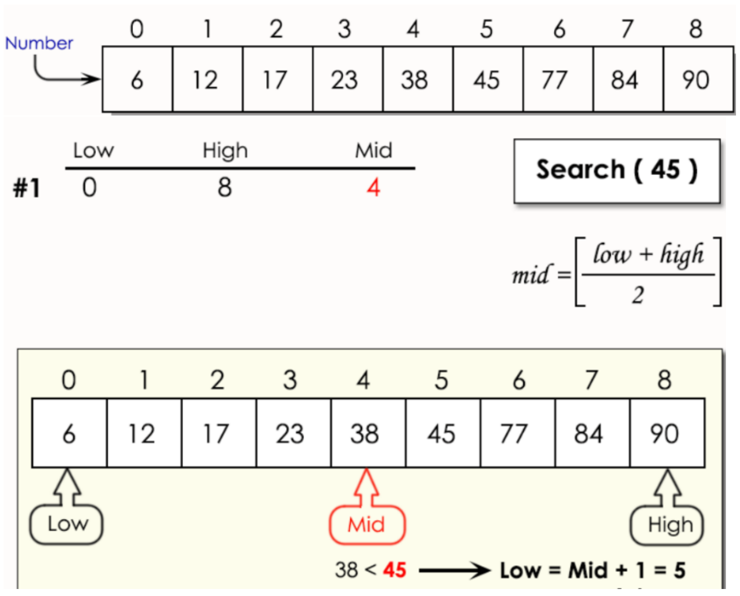
```
def binarySearch(listData, value)
   low = 0
   high = len(listData) - 1
   while (low <= high)
      mid = (low + high) / 2
      if (listData[mid] == value):
         return mid
      elif (listData[mid] < value)
         low = mid + 1
      else:
         high = mid - 1
   return -1
```

Seaching For 308 Result 9 Element found

low 8 mid 9 high 10

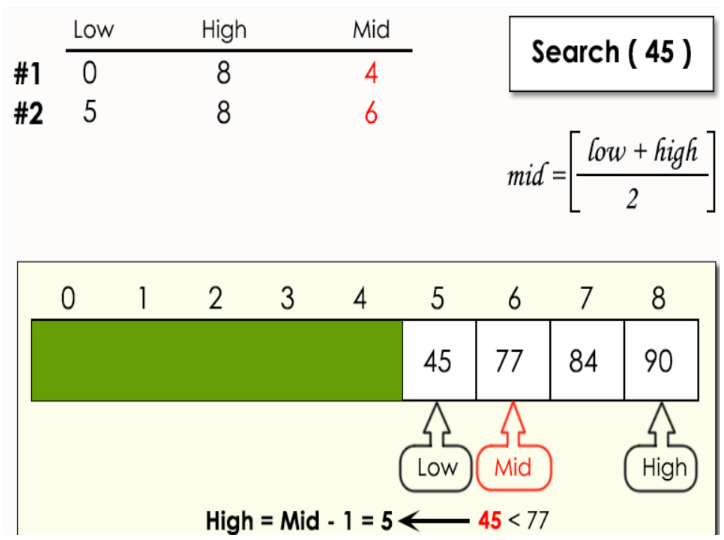
18	20	49	82	168	203	233	251	284	308	313	367	376	388	433	447
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
514	521	539	579	604	732	738	786	797	805	855	859	872	917	927	981
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Extra:



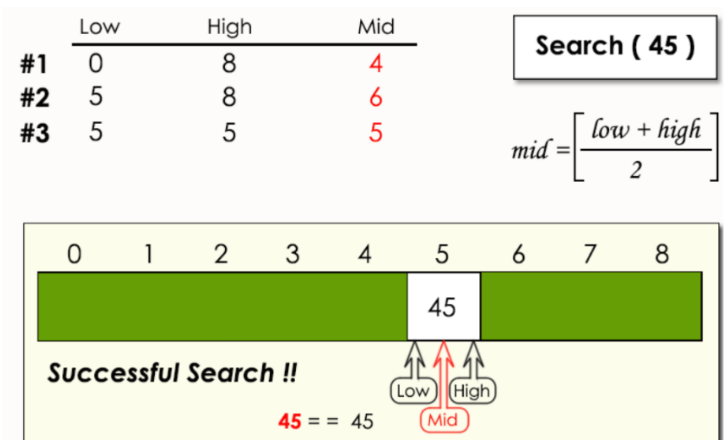
0	1	2	3	4	5	6	7	8
6	12	17	23	38	45	77	84	90

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



	Low	High	Mid
#1	0	8	4
#2	5	8	6

Search (45)
$mid = \left[\frac{low + high}{2}\right]$
0 1 2 3 4 5 6 7 8
45 77 84 90
Low
Mid
High
High = Mid - 1 = 5 ← 45 < 77

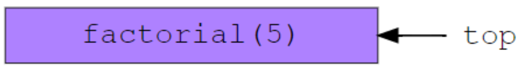


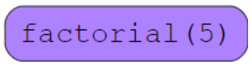
	Low	High	Mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

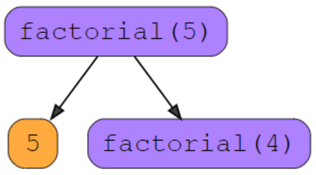
Program:

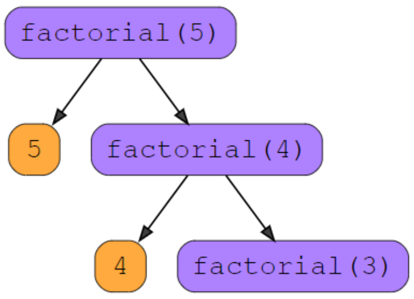
```
def binary_search(list1, low, high, n):

    # Check base case for the recursive function

    if low <= high:

        mid = (low + high) // 2

        # If element is available at the middle itself then return the its index

        if list1[mid] == n:

            return mid

        # If the element is smaller than mid value, then search moves

        # left sublist1
    elif list1[mid] > n:
      return binary_search(list1, low, mid - 1, n)
    # Else the search moves to the right sublist1
    else:
      return binary_search(list1, mid + 1, high, n)
  else:
    # Element is not available in the list1
    return -1
#main
list1 = [6,12,17,23,38,45,77,84,90]
n = 45
# Function call
res = binary_search(list1, 0, len(list1)-1, n)
if res != -1:
  print("Element is present at index", res)
else:
  print("Element is not present in list1")
```

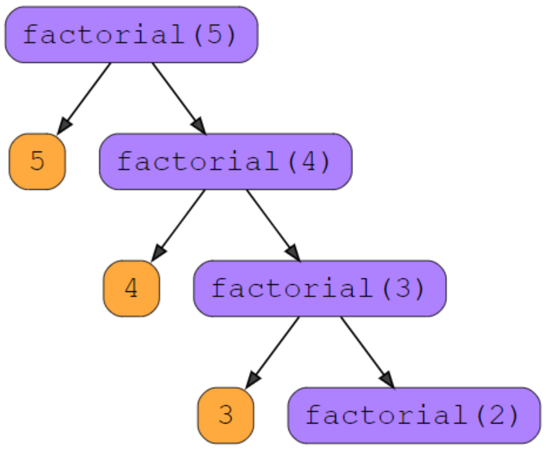
Recursive factorial:

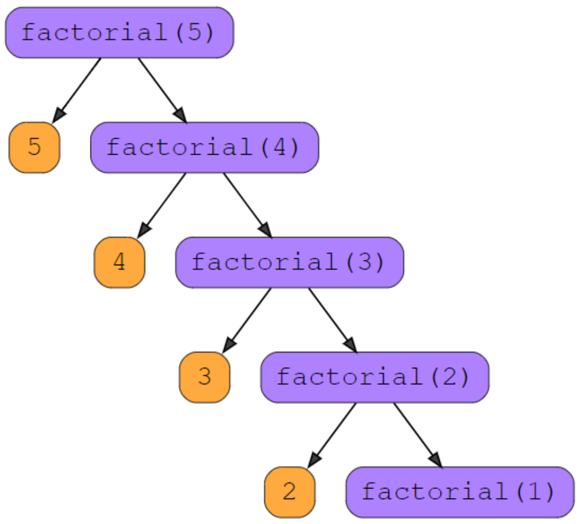


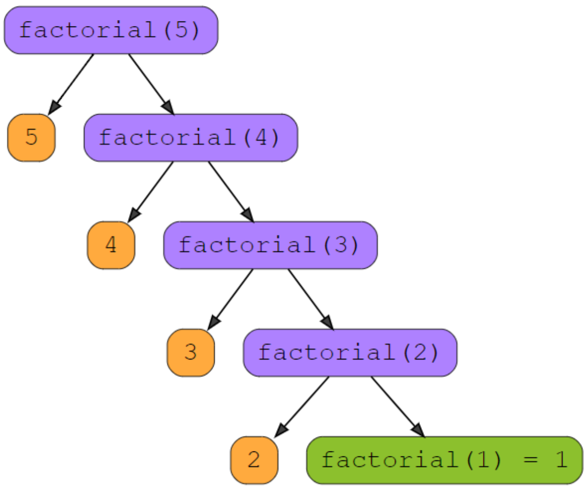


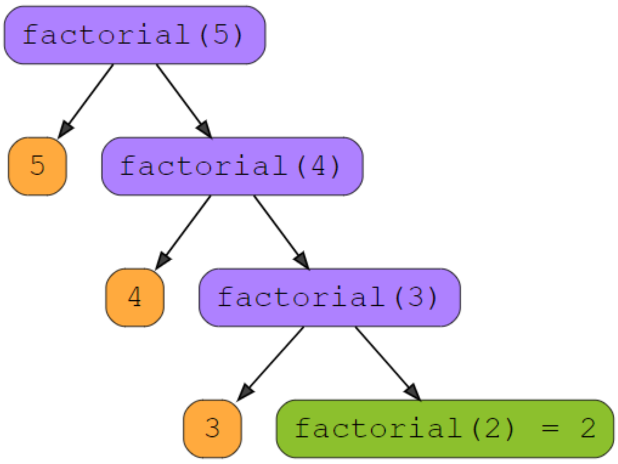


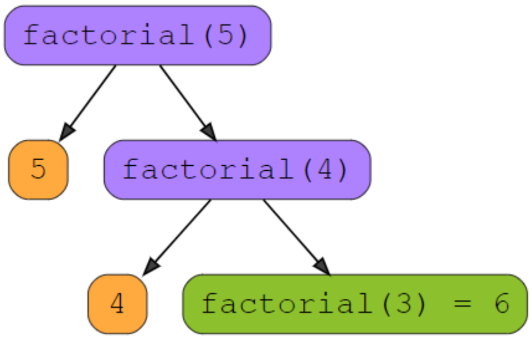


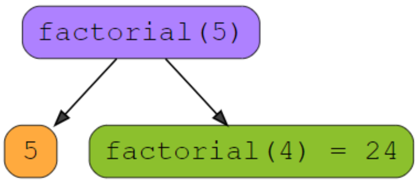


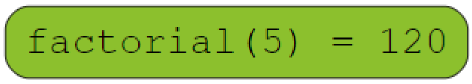



factorial(5)
5
factorial(4)
4
factorial(3)
3
factorial(2)
2
factorial(1) = 1


factorial(5)
5
factorial(4)
4
factorial(3)
3
factorial(2) = 2


factorial(5)
5
factorial(4)
4
factorial(3) = 6


factorial(5)
5
factorial(4) = 24


factorial(5) = 120

Program:

```
def factorial(num):
    if num == 1:
        return 1
    return num*factorial(num-1)
print(factorial(5))  #120
#factorial(4) = 4*factorial(3) = 4*3*factorial(2) = 4*3*2*factorial(1) = 4*3*2*1 = 24
```