# 8.1 Recursive definitions

## Recursive algorithms

An ***algorithm*** is a sequence of steps, including at least 1 terminating step, for solving a problem. A ***recursive algorithm*** is an algorithm that breaks the problem into smaller subproblems and applies the algorithm itself to solve the smaller subproblems.



Because a problem cannot be endlessly divided into smaller subproblems, a recursive algorithm must have a ***base case***: A case where a recursive algorithm completes without applying itself to a smaller subproblem.

**PARTICIPATION ACTIVITY** 8.1.1: Recursive factorial algorithm for positive numbers.

### Animation captions:

1. A recursive algorithm to compute N factorial has 2 parts: the base case and the non-base case.
2. N is assumed to be a positive integer. N = 1 is the base case, wherein a result of 1 is returned.
3. The non-base case computes the result by multiplying N by (N - 1) factorial.
4. The algorithm applying itself to a smaller subproblem is what makes the algorithm recursive.

**PARTICIPATION ACTIVITY** 8.1.2: Recursive algorithms.

1) A recursive algorithm applies itself to a smaller subproblem in all cases.
   - ○ True
   - ○ False

2) The base case is what ensures that a recursive algorithm eventually terminates.
   - ○ True
   - ○ False



3) The presence of a base case is what identifies an algorithm as being recursive.
   - ○

True

False

## Recursive functions

A ***recursive function*** is a function that calls itself. Recursive functions are commonly used to implement recursive algorithms.



Table 8.1.1: Sample recursive functions: Factorial, CumulativeSum, and ReverseList.

```
Factorial(N) {
   if (N == 1)
      return 1
   else
      return N * Factorial(N - 1)
}
```

```
CumulativeSum(N) {
   if (N == 0)
      return 0
   else
      return N + CumulativeSum(N - 1)
}
```

```
ReverseList(list, startIndex, endIndex) {
   if (startIndex >= endIndex)
      return
   else {
      Swap elements at startIndex and endIndex
      ReverseList(list, startIndex + 1, endIndex - 1)
   }
}
```

**PARTICIPATION ACTIVITY** 8.1.3: CumulativeSum recursive function.

1) What is the condition for the base case in the CumulativeSum function?

   N equals 0

   N does not equal 0

2) If Factorial(6) is called, how many additional calls are made to Factorial to compute the result of 720?

   7

   5

   3

3) Suppose ReverseList is called on a list of size 3, a start index of 0, and an out-of-bounds end index of 3. The base case ensures that the function still properly reverses the list.

- True
- False



# 8.2 Recursive algorithms

## Fibonacci numbers

The ***Fibonacci sequence*** is a numerical sequence where each term is the sum of the previous 2 terms in the sequence, except the first 2 terms, which are 0 and 1. A recursive function can be used to calculate a ***Fibonacci number***: A term in the Fibonacci sequence.

Figure 8.2.1: FibonacciNumber recursive function.

```
FibonacciNumber(termIndex) {
   if (termIndex == 0)
      return 0
   else if (termIndex == 1)
      return 1
   else
      return FibonacciNumber(termIndex - 1) + FibonacciNumber(termIndex - 2)
}
```

**PARTICIPATION ACTIVITY** 8.2.1: FibonacciNumber recursive function.

1) What does FibonacciNumber(2) return?

Check Show answer



2) What does FibonacciNumber(4) return?

Check Show answer

3) What does FibonacciNumber(8) return?

Check Show answer

## Recursive binary search



***Binary search*** is an algorithm that searches a sorted list for a key by first comparing the key to the middle element in the list and recursively searching half of the remaining list so long as the key is not found.

Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the index. If the search key is not found, the algorithm recursively searches the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

Figure 8.2.2: BinarySearch recursive algorithm.

```
BinarySearch(numbers, low, high, key) {
   if (low > high)
      return -1

   mid = (low + high) / 2
   if (numbers[mid] < key) {
      return BinarySearch(numbers, mid + 1, high, key)
   }
   else if (numbers[mid] > key) {
      return BinarySearch(numbers, low, mid - 1, key)
   }
   return mid
}
```

**PARTICIPATION ACTIVITY** 8.2.2: Recursive binary search.

Suppose BinarySearch(numbers, 0, 6, 42) is used to search the list (14, 26, 42, 59, 71, 88, 92) for key 42.



1) What is the first middle element that is compared against 42?

- 42
- 59
- 71

2) What will the low and high argument values be for the first recursive call?

- low = 0
  high = 2
- low = 0
  high = 3
- low = 4
  high = 6



3) How many calls to BinarySearch will be made by the time 42 is found?

- 2
- 3
- 4

**PARTICIPATION ACTIVITY** 8.2.3: Recursive binary search base case.

1) Which does not describe a base case for BinarySearch?

- The low argument is greater than the high argument.
- The list element at index `mid` equals the key.
- The list element at index `mid` is less than the key.

# 8.3 Analyzing the time complexity of recursive algorithms



## Recurrence relations

The runtime complexity T(N) of a recursive function will have function T on both sides of the equation. Ex: Binary search performs constant time operations, then a recursive call that operates on half of the input, making the runtime complexity T(N) = O(1) + T(N / 2). Such a function is known as a ***recurrence relation***: A function f(N) that is defined in terms of the same function operating on a value < N.

Using O-notation to express runtime complexity of a recursive function requires solving the recurrence relation. For simpler recursive functions such as binary search, runtime complexity can be determined by expressing the number of function calls as a function of N.

**PARTICIPATION ACTIVITY** 8.3.1: Worst case binary search runtime complexity.



### Animation content:

`undefined`

### Animation captions:

1. In the non-base case, BinarySearch does some O(1) operations plus a recursive call on half the input list.
2. The maximum number of recursive calls can be computed for any known input size. For size 1, 1 recursive call is made.
3. Additional entries in the table can be filled. A list of size 32 is split in half 6 times before encountering the base case.
4. By analyzing the pattern, the total number of function calls can be expressed as a function of N.
5. The number of function calls corresponds to the runtime complexity.

**PARTICIPATION ACTIVITY** 8.3.2: Binary search and recurrence relations.

1) When the low and high arguments are equal, BinarySearch does not make a recursive call.
   - ○ True
   - ○ False

2) Suppose BinarySearch is used to search for a key within a list with 64 numbers. If the key is not found, how many recursive calls to BinarySearch are made?
   - ○ 1
   - ○ 7
   - ○ 64



3) Which function is a recurrence relation?
   - ○

$T(N) = N^2 + 6N + 2$

- $T(N) = 6N + T(N/4)$
- $T(N) = log_2 N$

## Recursion trees



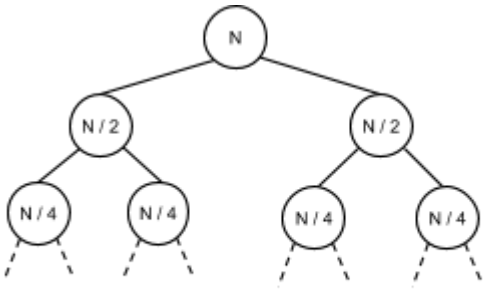
The runtime complexity of any recursive function can be split into 2 parts: operations done directly by the function and operations done by recursive calls made by the function. Ex: For binary search's T(N) = O(1) + T(N / 2), O(1) represents operations directly done by the function and T(N / 2) represents operation done by a recursive call. A useful tool for solving recurrences is a ***recursion tree***: A visual diagram of an operation done by a recursive function, that separates operations done directly by the function and operations done by recursive calls.

**PARTICIPATION ACTIVITY** 8.3.3: Recursion trees.

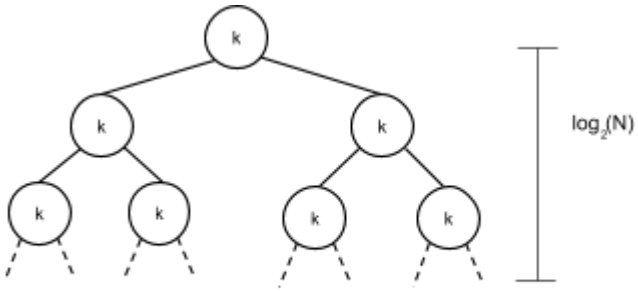
### Animation captions:

1. An algorithm like binary search does a constant number of operations, k, followed by a recursive call on half the list.
2. The root node in the recursion tree represents k operations inside the first function call.
3. Recursive operations are represented below the node. The first recursive call also does k operations.
4. The tree's height corresponds to the number of recursive calls. Splitting the input in half each time results in $log_2 N$ recursive calls. O($log_2 N$) = O($log N$).
5. Another algorithm may perform N operations then 2 recursive calls, each on N / 2 items. The root node represents N operations.
6. The initial call makes 2 recursive calls, each of which has a local N value of the initial N value / 2.
7. N operations are done per level.
8. The tree has O($log_2 N$) levels. O($N * log_2 N$) = O($N log N$) operations are done in total.

**PARTICIPATION ACTIVITY** 8.3.4: Matching recursion trees with runtime complexities.

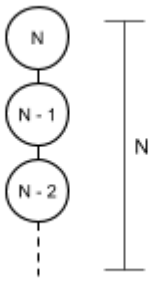


Tree 1

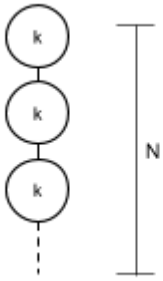


Tree 2





Tree 3



Tree 4

Tree 2 | Tree 4 | Tree 1 | Tree 3

| | |
|---|---|
| | T(N) = k + T(N / 2) + T(N / 2) |
| | T(N) = k + T(N - 1) |
| | T(N) = N + T(N - 1) |
| | T(N) = N + T(N / 2) + T(N / 2) |

Reset

**PARTICIPATION ACTIVITY** 8.3.5: Recursion trees.

Suppose a recursive function's runtime is $T(N) = 7 + T(N - 1)$.



1) How many levels will the recursion tree have?

- 7
- $log_2 N$
- $N$

2) What is the runtime complexity of the function using O notation?

- O(1)
- O($logN$)
- O($N$)



**PARTICIPATION ACTIVITY** 8.3.6: Recursion trees.

Suppose a recursive function's runtime is $T(N) = N + T(N - 1)$.

1) How many levels will the recursion tree have?

- $log_2 N$
- $N$
- $N^2$

2) The runtime can be expressed by the series N + (N - 1) + (N - 2) + ... + 3 + 2 + 1. Which expression is mathematically equivalent?

- $N * log_2 N$
- $(N/2) * (N + 1)$

3) What is the runtime complexity of the function using O notation?

- O($N$)
- O($N^2$)

# 8.4 Linked lists: Recursion



## Forward traversal

Forward traversal through a linked list can be implemented using a recursive function that takes a node as an argument. If non-null, the node is visited first. Then, a recursive call is made on the node's next pointer, to traverse the remainder of the list.

The ListTraverse function takes a list as an argument, and searches the entire list by calling ListTraverseRecursive on the list's head.

**PARTICIPATION ACTIVITY** 8.4.1: Recursive forward traversal.

### Animation content:

undefined

### Animation captions:



1. ListTraverse begins traversal by calling the recursive function, ListTraverseRecursive, on the list's head.
2. The recursive function visits the node and calls itself for the next node.
3. Nodes 19 is visited and an additional recursive call visits node 41. The last recursive call encounters a null node and stops.

**PARTICIPATION ACTIVITY** 8.4.2: Forward traversal in a linked list with 10 nodes.

1) If ListTraverse is called to traverse a list with 10 nodes, how many calls to ListTraverseRecursive are made?
   - ○ 9
   - ○ 10
   - ○ 11

**PARTICIPATION ACTIVITY** 8.4.3: Forward traversal concepts.

1) ListTraverseRecursive works for both singly-linked and doubly-linked lists.
   - ○ True
   - ○ False
2) ListTraverseRecursive works for an empty list.
   - ○ True
   - ○ False



## Searching

A recursive linked list search is implemented similar to forward traversal. Each call examines 1 node. If the node is null, then null is returned. Otherwise, the node's data is compared to the search key. If a match occurs, the node is returned, otherwise the remainder of the list is searched recursively.

### Figure 8.4.1: ListSearch and ListSearchRecursive functions.



```
ListSearch(list, key) {
   return ListSearchRecursive(key, list→head)
}

ListSearchRecursive(key, node) {
   if (node is not null) {
      if (node→data == key) {
         return node
      }
      return ListSearchRecursive(key, node→next)
   }
   return null
}
```

**PARTICIPATION ACTIVITY** 8.4.4: Searching a linked list with 10 nodes.

Suppose a linked list has 10 nodes.

1) When more than 1 of the list's nodes contains the search key, ListSearch returns _____ node containing the key.
   - the first
   - the last
   - a random

2) Calling ListSearch results in a minimum of _____ calls to ListSearchRecursive.
   - 1
   - 2
   - 10
   - 11



3) When the key is not found, ListSearch returns _____.
   - the list's head

- the list's tail
- null

## Reverse traversal

Forward traversal visits a node first, then recursively traverses the remainder of the list. If the order is swapped, such that the recursive call is made first, the list is traversed in reverse order.



**PARTICIPATION ACTIVITY** 8.4.5: Recursive reverse traversal.

### Animation content:

undefined

### Animation captions:

1. ListTraverseReverse is called to traverse the list. Much like a forward traversal, ListTraverseReverseRecursive is called for the list's head.
2. The recursive call on node 19 is made before visiting node 23.
3. Similarly, the recursive call on node 41 is made before visiting node 19, and the recursive call on null is made before visiting node 41.
4. The recursive call with the null node argument takes no action and is the first to return.
5. Execution returns to the line after the ListTraverseReverseRecursive(null) call. The node argument then points to node 41, which is the first node visited.
6. As the recursive calls complete, the remaining nodes are visited in reverse order. The last ListTraverseReverseRecursive call returns to ListTraverseReverse.
7. The entire list has been visited in reverse order.

**PARTICIPATION ACTIVITY** 8.4.6: Reverse traversal concepts.

Suppose ListTraverseReverse is called on the following list.




1) ListTraverseReverse passes _____ as the argument to ListTraverseReverseRecursive.

- node 67
-

node 18

- null

2) ListTraverseReverseRecursive has been called for each of the list's nodes by the time the tail node is visited.

- True
- False



3) If ListTraverseReverseRecursive were called directly on node 91, the nodes visited would be: _____.

- node 91 and node 67
- node 18, node 46, and node 91
- node 18, node 46, node 91, and node 67