# Fundamental Data Structures and Algorithms 03 - Sorting and Recursion

# Unit 2: Basic Algorithms

## 2.1 Sorting

Previously, we mentioned about dictionaries and how they have to sorted alphabetically for it to be meaningful. We shall now revisit sorting in greater detail.

In computer science, **sorting** usually refers to bringing a set of items into some well-defined order. To do this, we first specify the order on which the items are to be sorted it. For example, for numbers we can use the usual numerical order, such as ascending or descending, and for strings, the so-called lexicographic or alphabetic order, which is the one often used in dictionaries and even 'contacts' in various social media applications.

Sorting is important because having the items in order makes it much easier to find a given item, such as finding the most popular product in an e-commerce platform or a file corresponding to a particular client at a bank. If the sorting can be done prior to being used as input to the program, this enables faster access to the required item. This is important because many programs requires large datasets to be searched in real-time.

We will explore 2 common sorting algorithms in computer science namely, *bubble sort* and *insertion sort*. You are encouraged to visualize the following sorting algorithms by visiting here.

### 2.1.1 Bubble Sort

Assume we have array `arr` of size $n$ that we wish to sort in ascending order. **Bubble sort** starts by comparing `arr[0]` with `arr[1]` and swaps them if they are in the wrong order. It then compares `arr[1]` and `arr[2]` and swaps those if required, and so on. This means that once it reaches `arr[n-1]` and `arr[n]`, the largest entry will be in the correct place. It then repeats from the start , but leaving the $n^{th}$ entry alone (which is known to be correct). After it has reached the front again, the second-largest entry will be in place. The entire process is repeated until there is no more comparisons to make. At this point, the array has been sorted in ascending order.

It is worth introducing a simple test-case of size $n = 4$ to demonstrate how the sorting algorithms work:

> Given an array $arr$ with the elements $\{9, 0, 6, 4\}$ that requires to be sorted in ascending order, bubble sort starts by comparing $arr[0] = 9$ with $arr[1] = 0$. Since $arr[0] > arr[1]$, it swaps them, giving $\{0, 9, 6, 4\}$. It then compares $arr[1] = 9$ with $arr[2] = 6$ and again having to swap. We get $\{0, 6, 9, 4\}$. Then it compares $arr[2] = 9$ with $arr[3] = 4$ thus, giving $\{0, 6, 4, 9\}$. At this point, the largest entry it at the correct position within the array and we have reached the end of the array. This entire process is considered as *"1 repetition"*. We then repeat this entire repetition, however this time we stop at 1 index lesser than the previous repetition. This mean, $a[0]$ is compared with $a[1]$ - no change. Then $arr[1]$ is compared with $arr[2]$, this time swapping since $arr[1] > arr[2]$. Finally, this gives us $\{0, 4, 6, 9\}$. While although, this is already sorted, the algorithm will still continue for one final repetition to compare $arr[0]$ and $arr[1]$, at which the sorting process has completed.

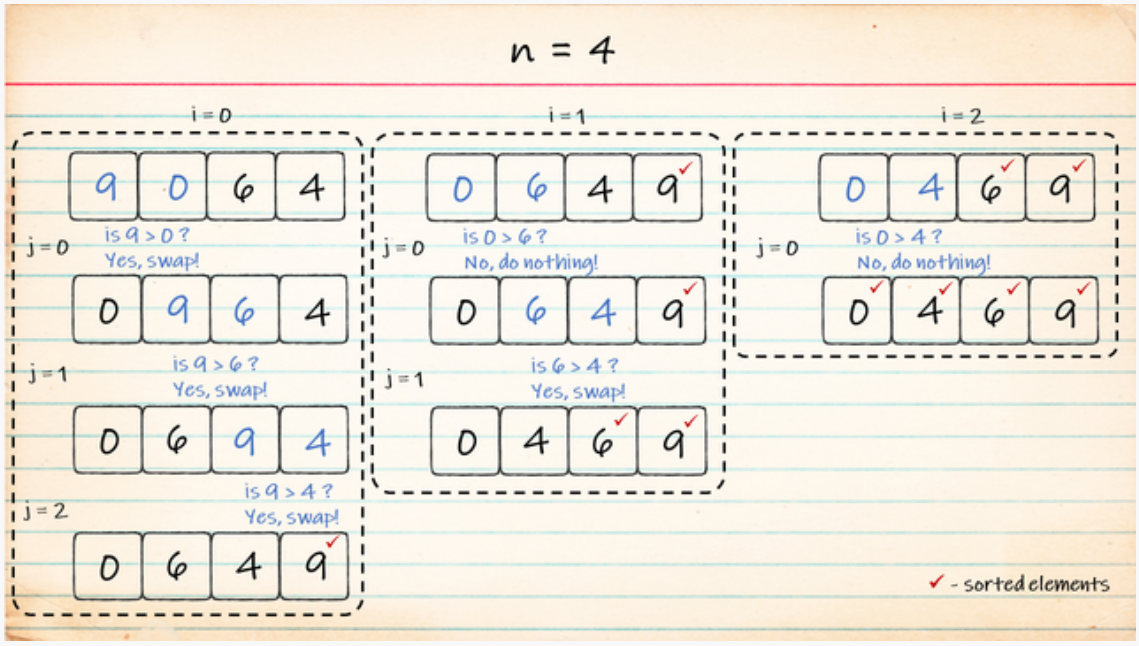


Fig 2.1. Bubble sort of on a simple test case

This entire algorithm can be implemented as follows:

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n-1):            # where 'i' is the number of repetitions
        for j in range(n-i-1):      # and 'j' is the index
            if arr[j] > arr[j+1] :
                # statement to swap 2 values in an array
                arr[j], arr[j+1] = arr[j+1], arr[j]

arr = [9, 0, 6, 4]
bubbleSort(arr)
print ("Sorted array is:", arr)
```

As is usual for comparison-based sorting algorithms, the time complexity will be measured by counting the number of comparisons that are being made. The outer loop is carried out $(n-1)$ times. while the inner loop is carried out $(n-i-1)$ times. Thus, the worst case and average case number of comparisons are both proportional to $n^2$, and hence the average and worst case time complexities are $O(n^2)$.

Alternatively, we can see in Fig 2.1. that the number of comparisons it has to make is a sum of the first $n-1$ numbers which is:

$$\frac{n(n-1)}{2} - n = \frac{1}{2}n^2 + \frac{1}{2}n$$

Using big O notation, we will drop the constant factors and all lower-order terms (i.e. $+\frac{1}{2}n$) , leaving us with $O(n^2)$.

Although bubble sort is very easy to implement, it tends to be particularly slow to run.

### 2.1.2 Insertion Sort

**Insertion sort** works the way many people sort a hand of playing cards. We start with an empty left hand and the deck of cards face down on the table. We then remove one card at a time from the deck and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from left to right.

The algorithm starts by temporarily assigning the first entry `arr[0]` as an already sorted array, then checks the second entry `arr[1]` and compares it with the first. If they are in the wrong order, it swaps the two. That leaves `arr[0]` and `arr[1]` sorted. Then it takes the third entry and positions it in the right place, leaving `arr[0]`, `arr[1]` and `arr[2]` sorted, and so on. More generally, at the beginning of the $i^{th}$ stage, insertion sort has the entries `arr[0]`, ... , `arr[i-1]` sorted and inserts `arr[i]`, giving sorted entries.

The implementation of insertion sort is given below:

```
def insertionSort(arr):
    for i in range(1, len(arr)):
        temp = arr[i]
        j = i-1
        while (j >= 0) and (temp < arr[j]) :
            arr[j+1] = arr[j]
            j -= 1
        arr[j + 1] = temp
```

We can test the `insertionSort` function using the following test case:

```
arr = [7, 8, 4, 1, 2, 6, 9, 5, 3]
insertionSort(arr)
print ("Sorted array is:", arr)
```

The computational complexity is again taken to be the number of comparisons performed. The outer loop is always carried out $(n-1)$ times while the inner loop depends on the items being sorted. Similar to bubble sort, insertion sort has an average and worse-case complexity of $O(n^2)$.

### 2.1.3 Python's Built-In Sorting Functions

Python provides two built-in ways to sort data. The first is the `sort` method of the list class. As an example, suppose that we define the following list:

`colors = [ red , green , blue , cyan , magenta , yellow ]`

That method has the effect of reordering the elements of the list into order, as defined by the natural meaning of the `<` operator for those elements. In the above example, within elements that are strings, the natural order is defined alphabetically. Therefore, after a call to `colors.sort( )`, the order of the list would become:

`[ blue , cyan , green , magenta , red , yellow ]`

Python also supports a built-in function, named `sorted`, that can be used to produce a new ordered list containing the elements of any existing iterable container. Going back to our original example, the syntax `sorted(colors)` would return a new list of those colors, in alphabetical order, while leaving the contents of the original list unchanged. This second form is more general because it can be applied to any iterable object as a parameter; for example, `sorted('green')` returns:

`['e', 'e', 'g', 'n', 'r']`.

## 2.2 Recursion

Any procedure that involve at least one step that invokes (or calls) the procedure itself is known as a **recursion**.

A recursive function has 2 basic properties:

1. A terminating or base condition - which, in the example above, is $n > 1$. This condition is critical otherwise the function may fall into an infinite loop or never terminate.
2. A recurrence relation - in this case, the function itself. We should define each possible recursive call so that it makes progress towards a base case.

Let's look at the following example:

```
def numberPrint(n):
    for num in range(n):
        print(num + 1)
```

This is a simple function to print all numbers from $1$ to $n$, inclusive. However, this function can also be written recursively using the follow

```
def recursiveNumberPrint(n):
    if (n > 1):
        recursiveNumberPrint(n-1)
    print(n)
```

Using a simple test case of `n = 4`, it is interesting to see that the recursive function will be called 4 times before it even starts to print and even does it in a 'unwinding' manner.

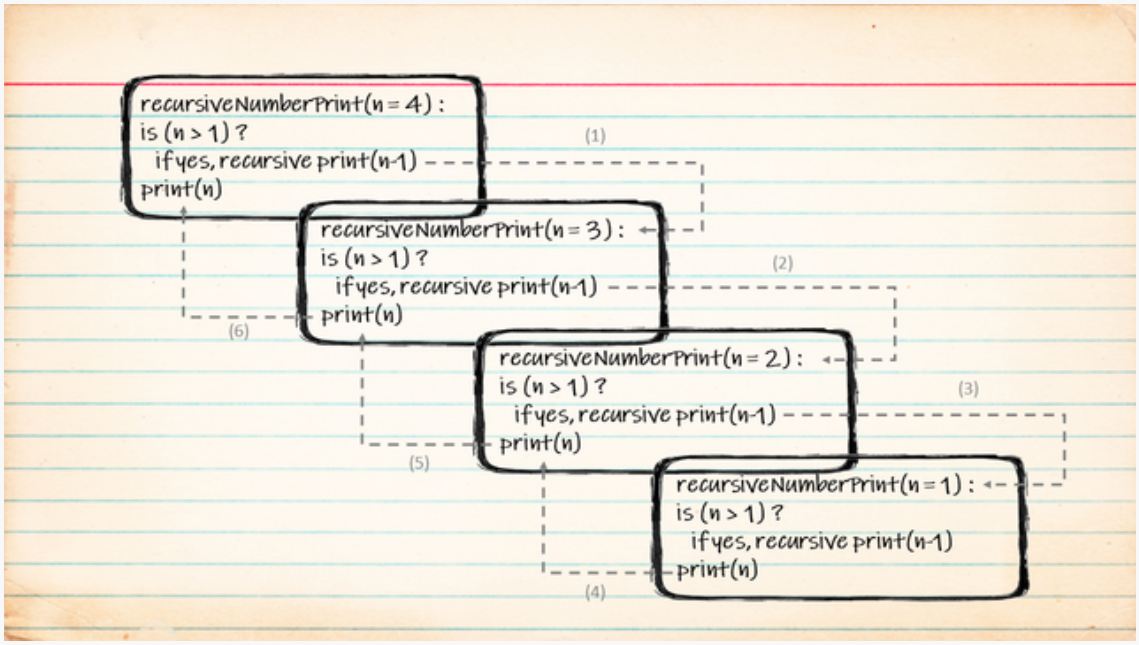


Fig 2.2. Recursion trace to print consecutive numbers

When developing or evaluating a recursive function, we typically use a *recursive call tree* or *recursion trace* such as the one illustrated in Fig 2.2. The diagram consists of small boxes and directed edges between the boxes. Each box represents a function call and is labeled with the name of the function and the actual arguments passed to the function when it was invoked. The directed arrows between the boxes indicate the flow of execution. Arrows 1 to 3 indicate the recursive function calls while arrows 4 to 6 indicate function returns.

While it may seem like an unnecessary way of solving a problem that can otherwise be solved with simple iterations such as `for` loops and `while` loops, recursive functions may provide an elegant solution to some other problems. Later, we will see that recursive functions are used in many data structures and algorithms.

### 2.2.1 Recursion Example 1: Factorial

Let's extend the previous example to solve the factorial of an positive integer. The factorial of a positive integer $n$ can be used to calculate the number of permutations of $n$ elements. The function is defined as:

$$n! = n \times (n-1) \times (n-2) \times \ \ldots \ \times 1$$

with the special case of $0! = 1$. This problem can be solved easily using an iterative implementation that loops through the individual values $[1 \ldots n]$ and computes a product of those values. But it can also be solved with a recursive solution and provides a simple example of recursion. Consider the factorial function on different integer values:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 2 \times 1 \\ 3! &= 3 \times 2 \times 1 \\ 4! &= 4 \times 3 \times 2 \times 1 \end{aligned}$$

If we carefully inspect of these equations, it becomes obvious that each of the successive equations, for $n > 1$, can be rewritten in terms of the previous equation:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \times (1-1)! \\ 2! &= 2 \times (2-1)! \\ 3! &= 3 \times (3-1)! \\ 4! &= 4 \times (4-1)! \end{aligned}$$

Since the function is defined in terms of itself and contains a base case, a recursive definition can be produced for the factorial function as shown here:

$$n! = \begin{cases} 1, & if \quad n = 0 \\ n \times (n-1)!, & if \quad n > 0 \end{cases}$$

Here, we provide the recursive implementation of the factorial function:

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

We illustrate the execution of a recursive function using a recursion trace of `fact(4)`

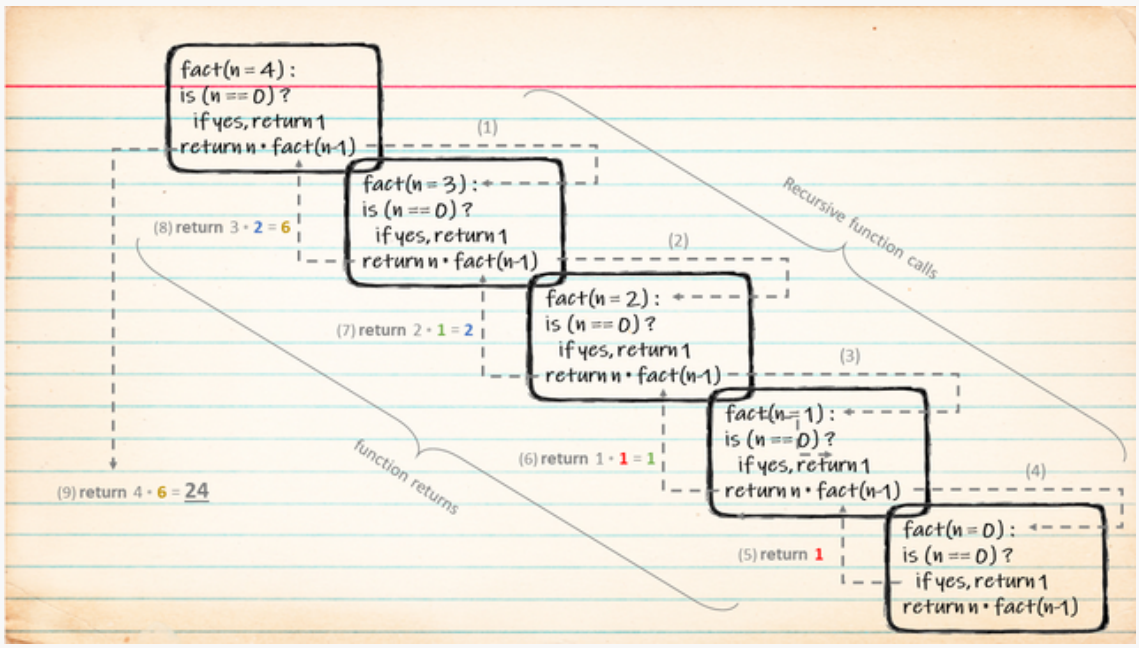


Fig 2.3. Recursion trace for `fact(4)`

Each entry of the trace corresponds to a recursive `fact` function call. Each new recursive `fact` function call is indicated by a downward arrow to a new invocation. When the function returns, an upward arrow showing this return is drawn and the return value is indicated alongside this arrow. Finally, the last return called is by the very first invocation of the recursive `fact` function to give us the answer $4! = 24$

### 2.2.2 Recursion Example 2: Merge Sort

**Merge sort** is a sorting algorithm that takes advantage of recursion. Merge sort divides the $n$-sized array into two smaller arrays. It then calls itself to sort the 2 smaller arrays until it reaches the *terminating condition* where each array is of length 1. Finally, merging the 2 sub-arrays to produce the sorted array.

The mechanisms to merge sort may seem simple, but implementing in code is much more challenging than say *bubble sort* or *insertion sort*.

Let's take a look at the implementation:

```
def mergeSort(arr):
    n = len(arr)
    if n > 1:
        # Finding the mid of the array
        mid = n // 2
        # 1. Dividing the array elements into 2 halves
        print('dividing ', arr)
        left = arr[:mid]
        right = arr[mid:]
        # 2. recursive calls to mergeSort for left and right sub arrays
        mergeSort(left)
        mergeSort(right)

        # initalizes pointers for left (i), right (j) and output array (k)
        i = j = k = 0

        # 3. Traverse and merges the sorted arrays
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i+= 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1
        # Checking if any element was left
        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1
    print('merging ', arr)
    return(arr)
```

Using the following test case,

```
arr = [3,5,6,4,2,7,9,1,8]
mergeSort(arr)
```

We run this program for the following results:

```
dividing  [8, 4, 7, 1, 5, 6, 3, 2]
dividing  [8, 4, 7, 1]
dividing  [8, 4]
merging  [8]
merging  [4]
merging  [4, 8]
dividing  [7, 1]
merging  [7]
merging  [1]
merging  [1, 7]
merging  [1, 4, 7, 8]
dividing  [5, 6, 3, 2]
dividing  [5, 6]
merging  [5]
merging  [6]
merging  [5, 6]
dividing  [3, 2]
merging  [3]
merging  [2]
merging  [2, 3]
merging  [2, 3, 5, 6]
merging  [1, 2, 3, 4, 5, 6, 7, 8]
```

Here we see a visualization of the above example.

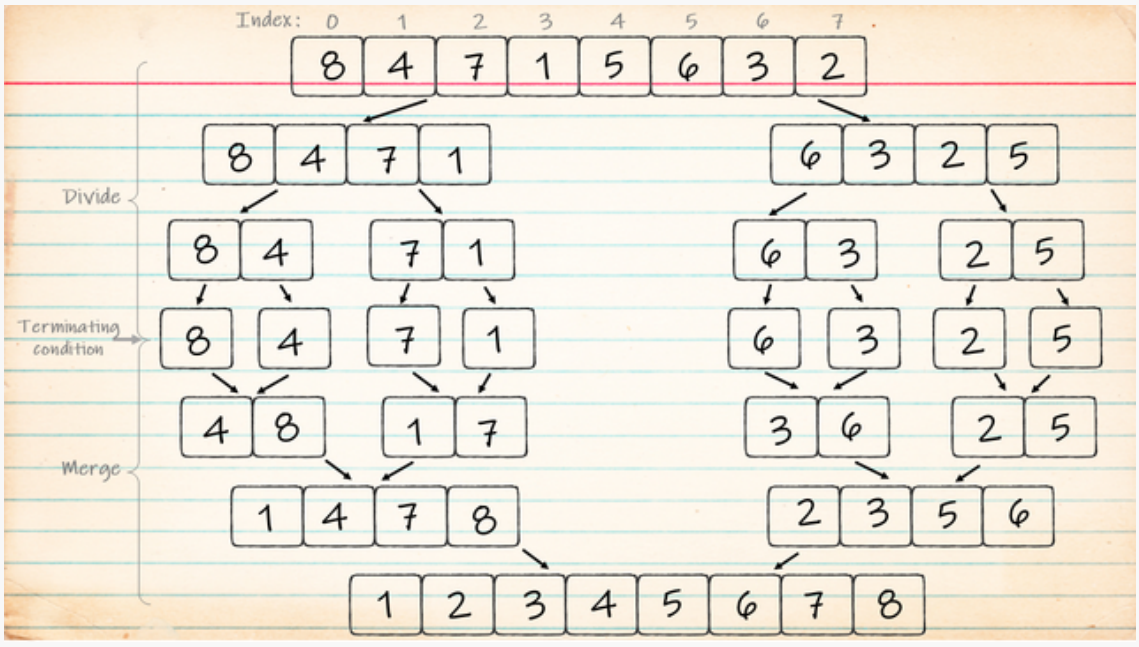


Fig 2.4. Visualization of merge sort algorithm

Interestingly, the computational time complexity of merge sort is $O(n \log n)$. We will now look at the reason behind this. You are encouraged to visualize and analyze the problem using this.

#### 2.2.2.1 Analysis of Merge Sort Algorithm

Let's do a bottom-up approach and analyze the *merge* step first. We assume that we are merging a total of $n$ elements in the entire array. We saw three part to merging:

1. After finding the midpoint, we copy each element in the `arr` into either `left` or `right`.
2. As long as some elements are untaken in both `left` and right, compare the first two untaken elements and copy the smaller one back into `arr`.
3. Once either `left` or `right` has had all its elements copied back into `arr`, copy each remaining untaken element from the other temporary array back into `arr`.

How many lines of code do we need to execute for each of these steps? It's a constant number *per element*. Each element is copied from `arr` into either `left` or `right` exactly one time in step 1. Each comparison in step 2 takes constant time, since it compares just two elements, and each element "wins" a comparison at most one time. Each element is copied back into `arr` exactly one time in steps 2 and 3 combined. Since we execute each line of code a constant number of times per element and that the subarray `arr` contains $n$ elements, the running time for merging is indeed $O(n)$.

Given that the *merge* step runs in $O(n)$, time when merging $n$ elements, how do we get to showing that `mergeSort` runs in $O(n \log n)$ time? Consider sorting a total of $n$ elements in the entire array:

1. The *divide* step takes constant time, regardless of the subarray size. After all, the *divide* step just computes the midpoint. we indicate constant time by $O(1)$.
2. The *recursive* step sorts two subarrays of approximately $\frac{n}{2}$ elements each, takes some amount of time, but we'll account for that time when we consider the sub problems.
3. The *merge* step merges a total of $n$ elements, taking $O(n)$ time.

If we think about the *divide* and *merge* steps together, the $O(1)$ running time for the *divide* step is a low-order term when compared with the $O(n)$ running time of the *merge* step. So, recalling the big O notation definition, we can think of the *divide* and *merge* steps together as taking $O(n)$ time. To make things more concrete, let's say that the *divide* and *merge* steps together take $cn$ time for some constant $c$.

To keep things reasonably simple, let's assume that if $n > 1$, then $n$ is always even, so that when we need to think about $\frac{n}{2}$, it's an integer. (Accounting for the case in which $n$ is odd will not change the result in terms of big O notation.) So now we can think of the running time of `mergeSort` on an $n$-element subarray as being the sum of twice the running time of `mergeSort` on an $\frac{n}{2}$-element subarray (for the *recursive* step) plus $cn$ (for the *divide* and *merge* steps).

Now we have to figure out the running time of two recursive calls on $\frac{n}{2}$ elements. Each of these two recursive calls takes twice of the running time of `mergeSort` on an $\frac{n}{4}$-element subarray (because we have to halve $\frac{n}{2}$) plus $\frac{cn}{2}$ to merge. We have two sub problems of size $\frac{n}{2}$, and each takes $\frac{cn}{2}$ time to merge, and so the total time we spend merging for sub problems of size $\frac{n}{2}$ is $(2 \cdot \frac{cn}{2}) = cn$.

To understand the running time performance of recursion, that is, what is the rate of growth in the time it takes for the algorithm to complete relative to the size of $n$, we can map each recursive call onto a a branch-like structure, also known as a *tree structure* (more details will be covered during the *Tree* chapter. Each node in the tree is a *recursive call* working on progressively smaller sub problems:

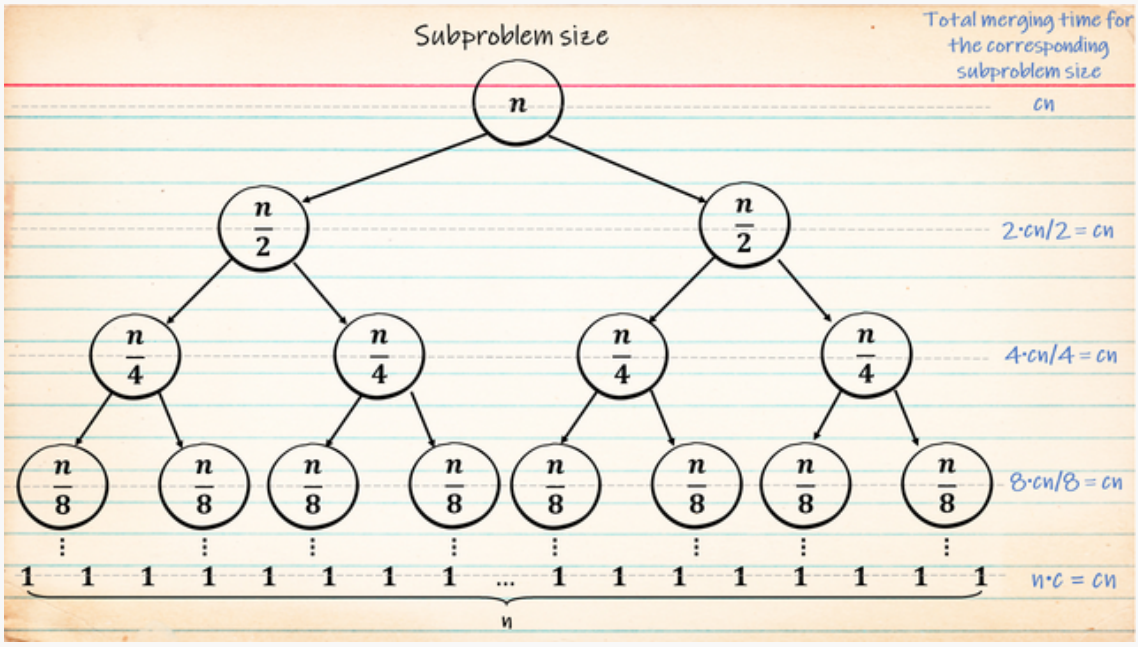


Fig 2.5. Merging time for all sub problems visualized as a tree

Computer scientists draw trees upside-down from how actual trees grow. Each circle in the tree is called a *node*. The *root node* is on top. In Fig 2.5, the *root* is labeled with the $n$ subarray size for the original $n$-element array. Below the *root* are two *child nodes*, each labeled $\frac{n}{2}$ to represent the subarray sizes for the two sub problems of size $\frac{n}{2}$.

Each of the sub problems of size $\frac{n}{2}$ recursively sorts two subarrays of size $\frac{(\frac{n}{2})}{2}$ ,or $\frac{n}{4}$. Because there are two sub problems of size , $\frac{n}{2}$ here are four sub problems of size $\frac{n}{4}$. Each of these four sub problems merges a total of $\frac{n}{4}$ elements, and so the merging time for each of the four sub problems is $\frac{cn}{2}$. Summed up over the four sub problems, we see that the total merging time for all sub problems of size $\frac{n}{4}$ is $(4 \cdot \frac{cn}{4}) = cn$. And subsequently, there will be eight of them with a total merging time of $8 \cdot \frac{cn}{8} = cn$ and so on.

As the sub problems get smaller, the number of sub problems doubles at each level of the recursion, but the merging time halves. The doubling and halving cancel each other out, and so the total merging time is $cn$ at each level of recursion. Eventually, we get down to sub problems of size $1$, i.e., the base case. We have to spend $O(1)$ time to sort subarrays of size $1$, because we have to test whether `left[i] < right[j]` and this test takes time. How many subarrays of size $1$ are there? Since we started with $n$ elements, there must be $n$ of them. Since each base case takes $O(1)$ time, let's say that altogether, the base cases take $cn$ time.

Now we know how long merging takes for each sub problem size. The total time for `mergeSort` is the sum of the merging times for all the levels. If there are $l$ levels in the tree, then the total merging time is $(l \cdot cn)$. So what is $l$? We start with sub problems of size $n$ and repeatedly halve until we get down to sub problems of size 1. We saw this characteristic when we analyzed binary search, and the answer is $l = (\log_2 n + 1)$. For example, if $n = 8$, then $(\log_2 n + 1) = 4$, and sure enough, the tree has four levels: n$= 8, 4, 2, 1$. The total time for `mergeSort`, then, is $cn(\log_2 n + 1)$. Thus, when we use the big O notation to describe this running time, we can discard the low-order term (i.e.. $+1$) and the constant coefficient ($c$), giving us a running time of $O(n \log_2 n)$, or simply $O(n \log n)$.

However, calling recursive functions adds a lot of unwanted overhead to the *runtime memory stack*. We will cover more about *Stacks* later, but to get you started on understanding, you can watch this video. As such, it is imperative that we use recursive functions sparingly.