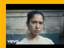
Internet and Big Data

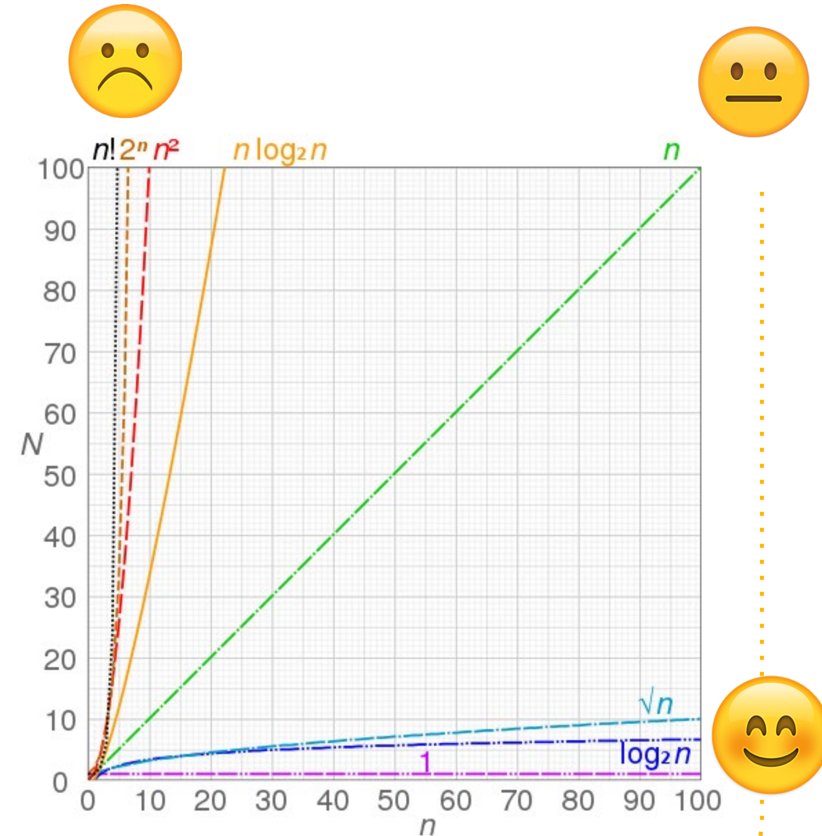


Complexity

Algorithms we know

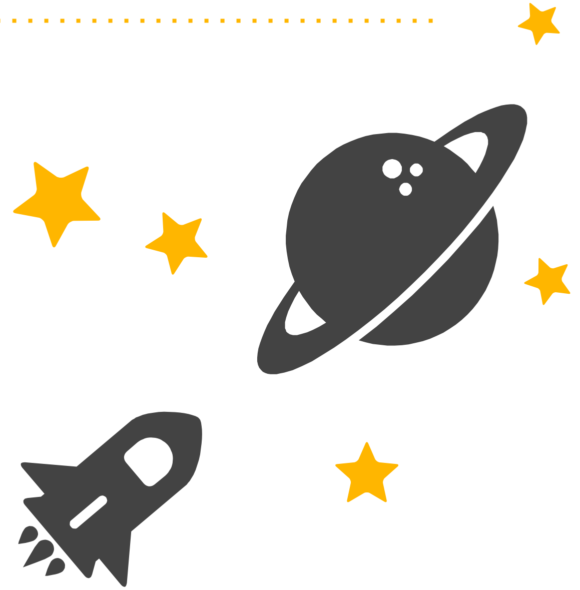
- Linear Search, Binary Search
- Selection Sort, (will see Merge Sort)

- **Algorithms** can be **ranked** in terms of **time** (or space) **efficiency**
- **n** is # elements we have.
- **N** is # operations our algorithm does.
- Rate algorithm by a **reference function** which closely represents how N grows as n gets larger.



https://en.wikipedia.org/wiki/Big_O_notation

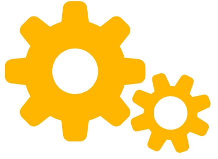
Big O Notation



To describe an algorithm's efficiency 👍

http://interactivepython.org/courselib/static/pythonds/AlgorithmAnalysis/BigONotation.html

This lesson



- Complexity and Big O Notation
- Analysis of brief algorithms
- Analysis of Linear Search
- Analysis of Binary Search
- Analysis of Selection Sort
- Analysis of Merge Sort

Category	Reference Function
Constant	1
Logarithmic	$\log_2(n)$
Linear	n
nlogn	$n\log_2(n)$
Quadratic	n^2
Cubic	n^3
Exponential	a^n, $a>1$

How do we calculate time complexity?

- **Count** the number of times a **critical operation** is executed
  - Usually seen in a loop
- **Disregard "constants"**
  - Usually seen as unindented lines, and independent of input size
- **Disregard "lower exponent terms"**
  (e.g., for $\mathbf{3\ n^2 + 15n + 35}$; ignore the **+15n**, ignore the **+35**).

Let's see some examples!

Calculating Time Complexity - Example 1

```
1  x = 0
2  y = 10
3  x += 1
4  for i in range(n):
5      x += y
6      y += 1000
```

Recall:

- Count the number of times a critical operation is executed
- Disregard “constants”
- Disregard “lower exponent terms”

- Critical operations depending on n?
  - addition (lines 5&6)
- What are the constants?
  - Lines 1-3
- How many addition/assignments that matters are executed?
  - 2*n (lines 5&6 repeated n times)
- What is the time complexity?
  - O(n)

Calculating Time Complexity - Example 2

```
1    count = 0
2    for i in range(n):
3        count = count + 10
4    for j in range(n):
5        count = count + j
```

Recall:

- Count the number of times a critical operation is executed
- Disregard "constants"
- Disregard "lower exponent terms"

- Critical operations depending on n?
  - addition (lines 3, 5)
- What are the constants?
  - Line 1
- How many additions that matter are executed?
  - n + n = 2n (line 3 repeated n times, so is line 5)
- What is the time complexity?
  - O(n)

Calculating Time Complexity - Example 3

```
1    count = 0
2    for i in range(n):
3        for j in range(n):
4            count = count + 10
```

Recall:

- Count the number of times a critical operation is executed
- Disregard "constants"
- Disregard "lower exponent terms"

- Critical operations depending on n?
  - addition (line 4)
- What are the constants?
  - Line 1
- How many additions that matters are executed?
  - $n * n = n^2$ (line 4 repeated n times in j loop, which repeated n times in i loop)
- What is the time complexity?
  - $O(n^2)$

Calculating Time Complexity - Example 4

```
1    count = 0
2    for i in range(n):
3        for j in range(n):
4            count = count + 10
5        for j in range(n):
6            count += 2
```

Recall:

- Count the number of times a critical operation is executed
- Disregard "constants"
- Disregard "lower exponent terms"

- Critical operations depending on n?
  - addition (lines 4&6)
- What are the constants?
  - Line 1
- How many addition/assignments that matters are executed?
  - $(n + n) * n = 2n^2$ (line 4 repeated n times in j loop, so is line 6, both loops repeated n times in i loop)
- What is the time complexity?
  - $O(n^2)$ (can omit the coefficient if it is a number)

Best case, worst case, average case

- When the running time depends on the data (and not a variable already in the program), we can calculate best case, worst case, and average case scenarios
- For example, suppose we are performing a Linear Search…
  - **Best case scenario** happens when the first element we check is what we are looking for (no need to go through the list)
  - **Worse case scenario** happens when the last element we check is what we are looking for, or it doesn't exist (have to go through the list to know)
  - **Average case scenario** happens when the element we look for has an equal chance to be anywhere in the list

Linear search complexity

```python
# Input: List of numbers, number to search for
# Output: The first index where number can be found, -1 if not in list
def linear_search_index(input_list, item):
    for i in range(len(input_list)):
        if input_list[i] == item:
            return i
    return -1
```

<u>**Best case**</u> : Returned position `i` is `0` → **O(1)**

<u>**Worst case**</u> : Two possible situations

Last item: Returned position `i` is `n-1`

Not found: Returned position `i = -1`

O(n)

Let's analyze this algorithm

```
1   sum = 0
2   i = n
3   while i >= 1:
4       sum = sum + 1
5       i = i/2
```

Not as simple because it's not n times

Recall:

- Count the number of times a critical operation is executed
- Disregard "constants"
- Disregard "lower exponent terms"

- Critical operations depending on n?
  - addition (lines 4&5)
- What are the constants?
  - Lines 1&2
- How many addition/assignments that matters are executed?
  - 2*k (lines 4&5 repeated k times)
- What is the time complexity?
  - O(k) ← not the answer yet as it should be a function of n

Let's analyze this algorithm

```
1   sum = 0
2   i = n
3   while i >= 1:
5       sum = sum + 1
6       i = i/2
```

Divides in half **k** times

Recall:

- Count the number of times a critical operation is executed
- Disregard "constants"
- Disregard "lower exponent terms"

- **Observation**: instead of decreasing 1 in each iteration, **i decreases by half**
- Then the question becomes: "How many times can the while-loop **divide n by 2** until it becomes 1 (as it ends in the next iteration)?"
- Suppose **n** is a power of **2** (e.g., 2^{10})

  We can do that because if not, n is going to be between two such numbers (e.g., $2^6 < 100 < 2^7$)
- Let $\mathbf{n = 2^k}$. Then, what is k?

The concept of logarithm

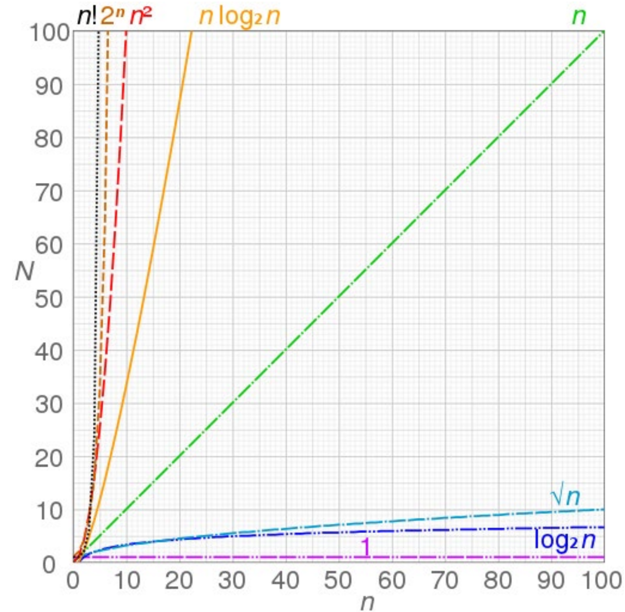
If the number of elements is **n**, and **n = 2k**
And we would like to know what **k** is, how can we solve for **k**?
We can use logarithm:

$$\log_2 n = k$$

- So, if n is 2^{10}, then k is 10 → while-loop needs to divide 10 times; and if n is 2^k, then while-loop needs to divide k times
- Therefore, our algorithm has a **O(logn)** time complexity

Intuition of an O(log n) algorithm

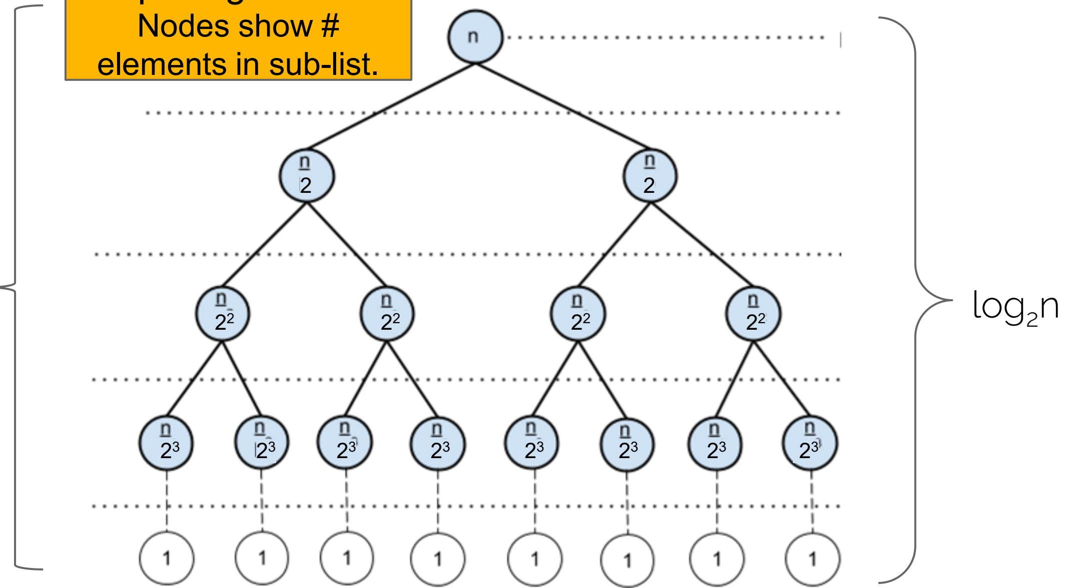
In each stage the algorithm processes half of the previous stage. Which algorithms have we seen that process half of the previous stage each time?



Binary Search **analysis**

One path from the top to the bottom is a full search for a value.

The height of this tree is equal to the number of comparisons needed to find the item in the worst case.



The height of the tree is the answer to the following question: **How many times do we divide the problem of size n by 2** until we get down to a problem size of 1?

Credit: https://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly/13093274

```
1   # Input:  An unsorted list of numbers
2   # Output: Returns a sorted list of numbers
3   #         (Input list is unchanged)
4   def selection_sort(data_original):
5       # Make a copy so we don't change the original
6       data = data_original[:]
7
8       # For each spot in the list, find the next smallest number
9       # in the remaining sublist of our numbers
10      for i in range(len(data)):
11          # Start by assuming smallest element is first in sublist
12          min_number = data[i]
13          min_idx = i
14
15          # Look through rest of sub-list for smallest element
16          for j in range(i + 1, len(data)):
17              if data[j] < min_number:
18                  min_number = data[j]
19                  min_idx = j
20
21          # Swap the current element with the next smallest element
22          temp = data[i]
23          data[i] = data[min_idx]
24          data[min_idx] = temp
25
26      return data
```

Selection Sort

A reminder from last time!

```
28  test = [5,3,6,2,1]
29  print(selection_sort(test))
30  print(test)
```

Complexity of Selection Sort

Selection Sort looks for the smallest element multiple times, by going through n elements, then n-1 elements, then n-2 elements.

The total number of operations (comparisons) is therefore:

$$= n + (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

$$= n(n+1)/2$$

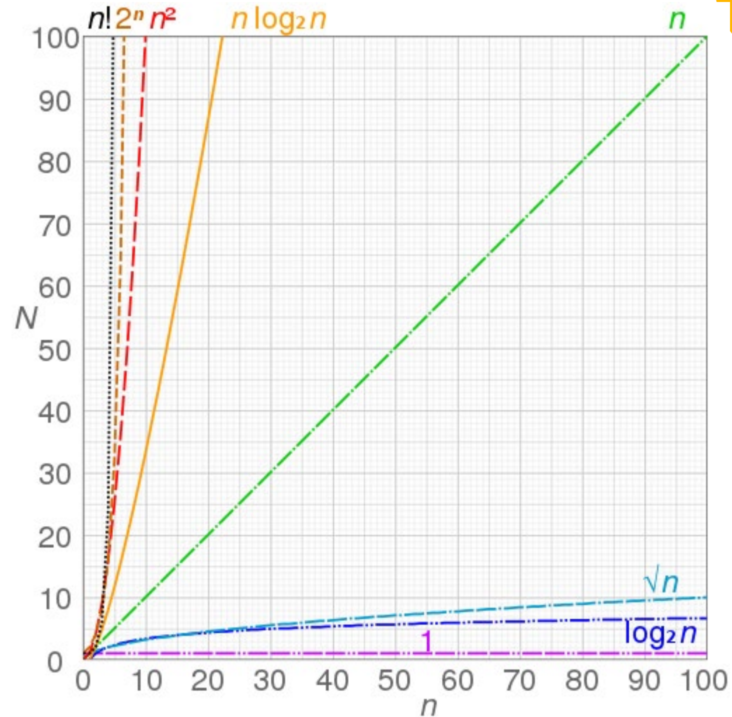
$$= n^2/2 + n/2$$

Selection sort is **quadratic, $O(n^2)$.** This is true for both the **best** and **worst case (why?)**

Now we have analyzed 3 algorithms!

We used **Big O** Notation to describe these algorithms' **worst case** runtime:

- **O(n)** Linear search
- **O(logn)** Binary search
- **O(n²)** Selection sort



https://en.wikipedia.org/wiki/Big_O_notation

http://www.cs.sfu.ca/CourseCentral/120/ggbaker/guide/parts/guide06

Is sorting worth it?

Binary search **requires** the data to be **sorted**. Sorting can be **expensive**, e.g. **$O(n^2)$**

Even if **binary search** is **O(logn)**, wouldn't the exponential cost of sorting make this approach worse than **linear search O(n)**?

Think: In what cases would it be worth it to sort, then perform binary search?

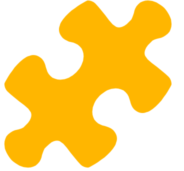
http://interactivepython.org/courselib/static/pythonds/SortSearch/TheBinarySearch.html

input size, e.g. list to search/sort

What happens when n gets larger?

n	Ex: Get first element in list **O(1)** Always constant	Ex: Linear search **O(n)**	Ex: Selection sort **O(n^2)**	Ex: Binary search **O(log n)**
10	1	10	100	3.32
100	1	100	10,000	6.64
1000	1	1000	1,000,000	9.96
10000	1	10000	100,000,000	13.28

Introducing Merge Sort

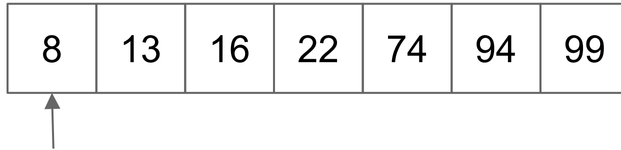


- **Mergesort** is much faster than Selection Sort (along with shellsort, quicksort, etc.)

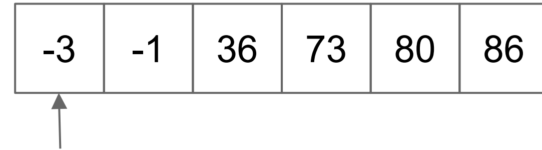
https://visualgo.net/bn/sorting

How do you merge 2 sorted lists?

sorted list 1



sorted list 2

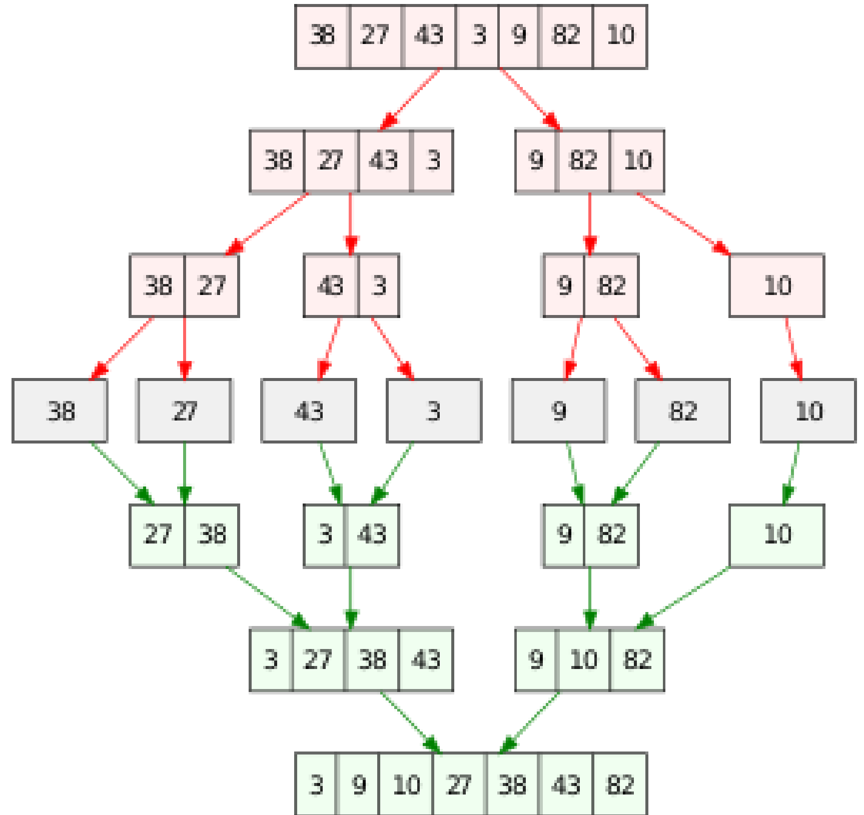


result

Start from the beginning of each list. Pick the smaller item and append it to the result list, repeat until both lists are empty

Merge sort

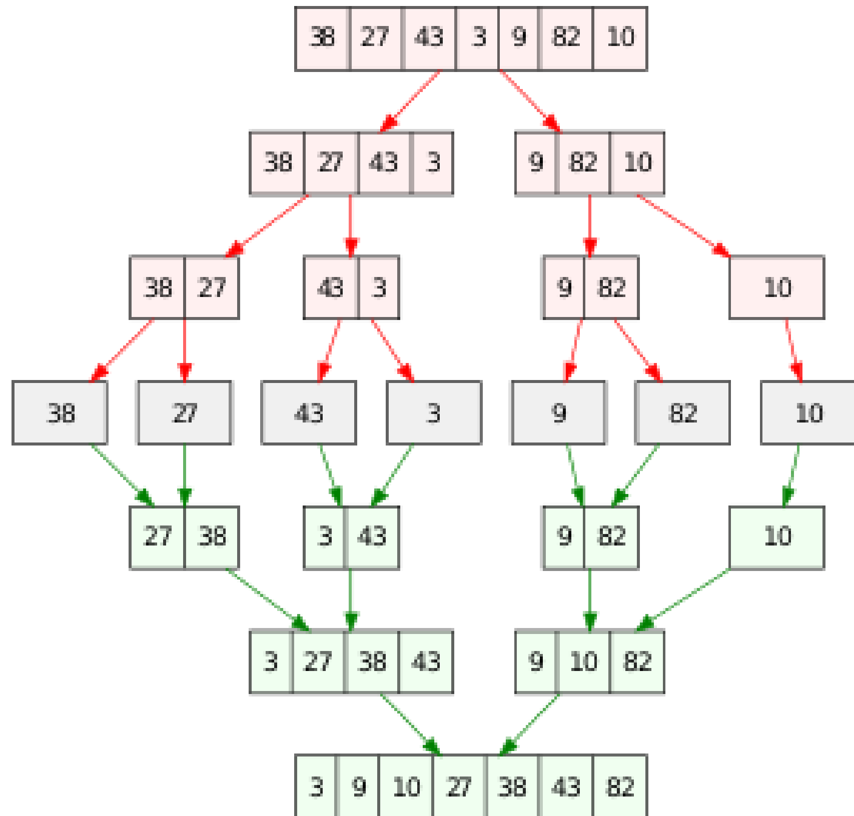
- Split the list until there is one element
- Progressively recombine neighbouring sorted lists using the merge function we defined earlier



Merge sort

Mergesort can be implemented as a recursive algorithm:

- It has a base case: stop when there is just one item to sort
- It has a recursive case: give itself a sublist to sort
- It changes the input towards the base case: the sublist is a smaller list



Mergesort – Pseudocode

```
1 mergeSort(alist):
2   if (alist has 2 or more elements):
3       sortedLeft = mergeSort(left half of alist)
4       sortedRight = mergeSort(right half of alist)
5       result = merge(sortedLeft, sortedRight)
6   else:
7       result = alist  # list is already sorted, 1 element only
8   return result
```

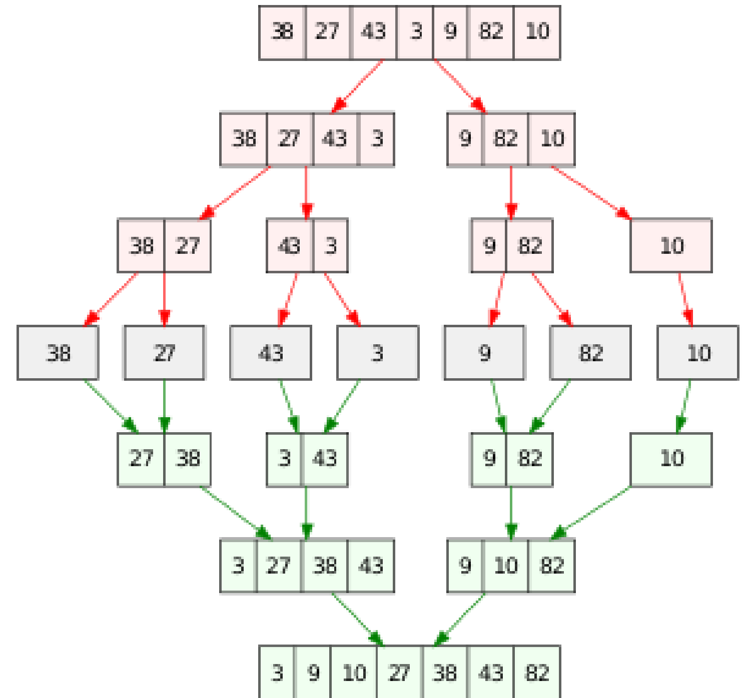
mergeSort() itself is recursive as it calls itself with a smaller list

The function **merge()** is a non-recursive helper function that creates a new sorted list by merging the two sorted lists (see pg. 6)

One way to think of it is instead of tracing how the recursion unfolds, focus on the current step:
It recursively calls itself with 2 halves (lines 3&4), and "magically" gets the halves back sorted.
The merge them into one sorted list (line 5)

Complexity of mergesort?

- Height of the tree (just as in binary search) ⇒ **log n**
- Merge operation for all the elements ⇒ **n** operations



Complexity of mergesort?

O(nlogn) (n times log n)

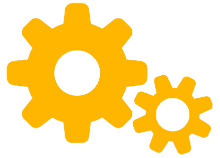
Intuitively, considering various operations:

- Height of the tree (just as in binary search) ⇒ **log n**
- Merge operation for all the elements ⇒ **n** operations

Note: Extra space is needed to create the merged lists at each level. This may be problematic for large n.

Other sorting algorithms do all the operations in place, e.g., swapping with only 1 temp variable needed.

Review

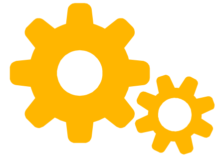


Give the worst case time complexity for each of the algorithms below.

- Linear search
- Binary search
- Selection sort
- Merge sort

Options: $O(n!)$, $O(1)$, $O(n^2)$,$O(n)$, $O(n\log n)$, $O(\log n)$, $O(n^3)$

Review



Assume that a problem can be solved with two different algorithms, and you need to decide which algorithm to implement based on their time complexity.

Their time complexities are **O(n)** and **O(nlogn)**.

Which algorithm would you choose, if you had a very large dataset?