Table of Contents

- 1 Asymptotic notations
- 2 Recursion
  - 2.1 Description
  - 2.2 Example
- 3 Binary Search
  - 3.1 Description
  - 3.2 Example
  - 3.3 Time complexity
  - 3.4 Code example
- 4 Selection Sort
  - 4.1 Description
  - 4.2 Example
  - 4.3 Time complexity
  - 4.4 Code example
- 5 Bubble Sort
  - 5.1 Description
  - 5.2 Example
  - 5.3 Time complexity
  - 5.4 Code example
- 6 Merge sort
  - 6.1 Description
  - 6.2 Example
  - 6.3 Time complexity
  - 6.4 Code example
- 7 Quick Sort
  - 7.1 Description
  - 7.2 Example
  - 7.3 Time complexity
  - 7.4 Code example
- 8 Resources

The goal of this notebook is to document the learning process of sorting and searching algorithms. I am focusing here on algorithms analyze and time complexity.

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
```

1 Asymptotic notations

> **Algorithm** - is a set of steps to accomplish the task (c) Khan Academy

What makes good algorithm:

- Correctness

• Efficiency - is how well you are using your computer resources to get a particular job done.

Complexity – is a measure of algorithm efficiency in terms of time usage. Running time - is how long it takes to run an algorithm (steps, seconds, iterations since it does not matter what unit of measure).
Ways to approach tasks:

- Divide-and-conquer
- Greedy method - the algorithm checks the problem one step at the time and focuses just on this step (often as a part of optimization).

Asymptotic notations are the mathematical notations used to describe the running time of algorithms when the input tends towards a particular value or a limiting value. It helps to make comparison between different algorithms, especially on large inputs.
High level idea: to suppress constant factors (like system dependency) and lower terms (irrelevant for large inputs). So we simplify function in order to drop the less significant parts.
For example, if the function: $an^2 * bn + c$, we can say that our algorithm takes n^2 , so we are dropping:
1) coefficient because it does not matter what coefficient used,
2) last part $bn + c$ because it is relatively small compare to first part n^2

Also it doesn't matter what base of the logarithm we use in asymptotic notation.

> The main idea of asymptotic analysis is to see how algorithms behave on tail end, when input gets large. (c) Back To Back SWE. YouTube.

There are three main types of asymptotic notations:

```
- big-O - upper bound of an algorithm (the worst case)
- big-Θ or big-Theta - "tight" or "exact" bound. It is a combination of Big
(average case)
- big-Ω or big–Omega - lower bound of an algorithm (best case) or the faste
st the algorithm can go.
```

Examples of different complexity of big-O:
Constant complexity O(1) – the same time no matter how big input. Example:

In [2]:

```python
def sum(a, b, c):
    return a + b + c
sum(3, 2, 1)
# takes the same time as:
sum(1000, 300, 111)
```

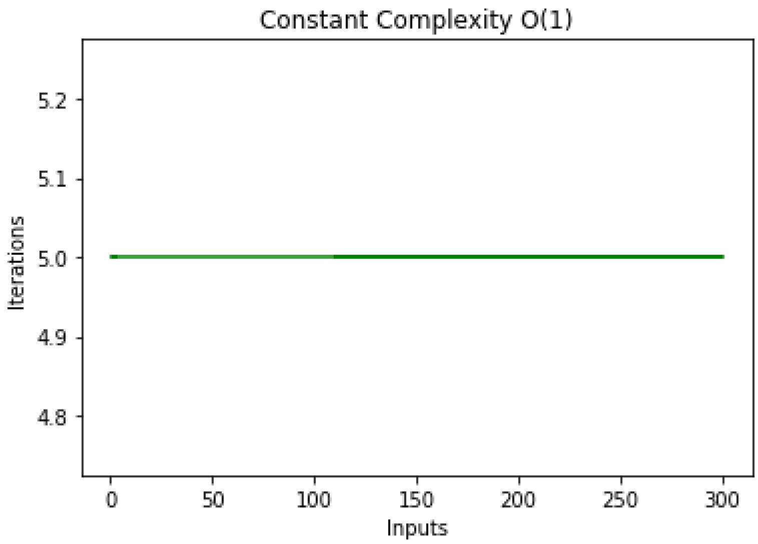
Out[2]:

```
1411
```

In [3]:

```
x = [3, 2, 1, 8, 300, 111] #input(n)
y = [5, 5, 5, 5, 5, 5] #number or iterations

plt.plot(x, y, 'g')
plt.title('Constant Complexity O(1)')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.show()
```



Linear complexity O(n) – the number of operations grows in the same rate as number of inputs. Example (linear search):

In [4]:

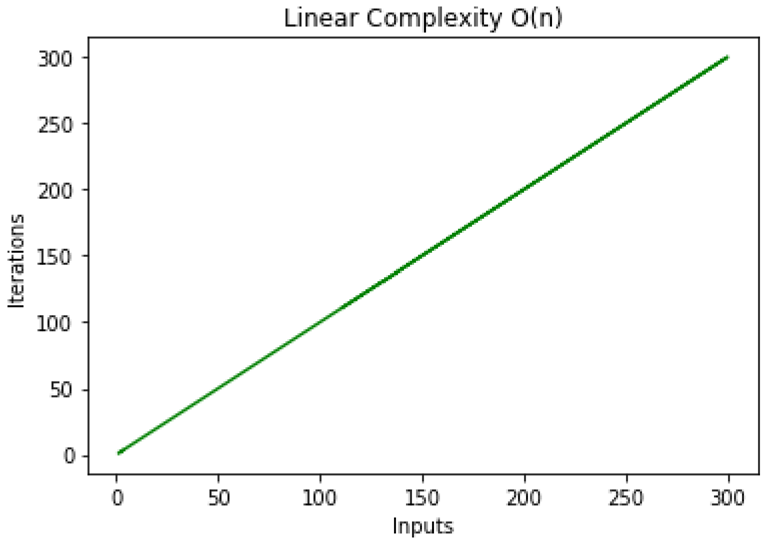
```
foodArray = ['pizza', 'burger', 'sushi', 'curry']
def findFoodInTheApp(arrayOfFood, desiredFood):
    for food in arrayOfFood:
        if food == desiredFood:
            return f'Here is your {food}'
print(findFoodInTheApp(foodArray, 'curry')) #To find a "curry" is taking O(n)
```

```
Here is your curry
```

In [5]:

```
x = [3, 2, 1, 8, 300, 111] #inputs(n)
y = [3, 2, 1, 8, 300, 111] #number or iterations

plt.plot(x, y, 'g')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.title('Linear Complexity O(n)')
plt.show()
```

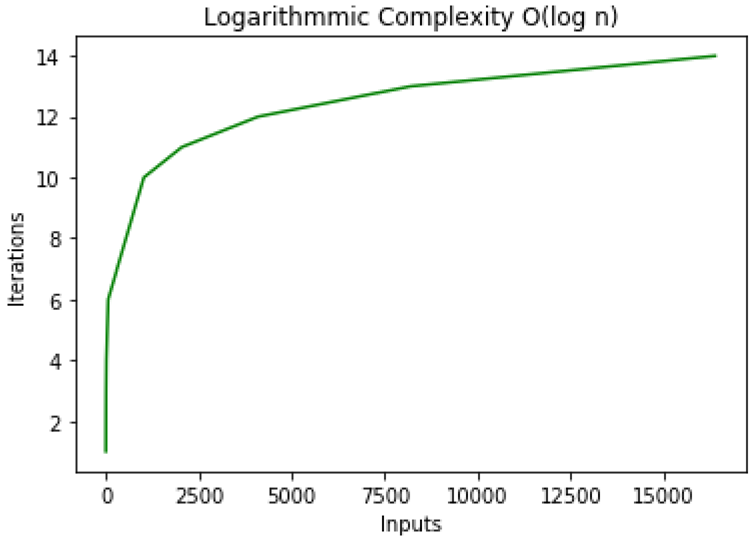


Logarithmic complexity O(log n) – the number of operation grows slower than the inputs. More efficient for larger inputs. Example: [binary search (algorithms/searching/binary_search.py)](algorithms/searching/binary_search.py)

In [6]:

```
x = [1, 4, 8, 16, 64, 1024, 2048, 4096, 8192, 16384] #inputs grow faster
y = [1, 2, 3, 4, 6, 10, 11, 12, 13, 14] #number or iterations

plt.plot( x, y, 'g')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.title('Logarithmmic Complexity O(log n)')
plt.show()
```

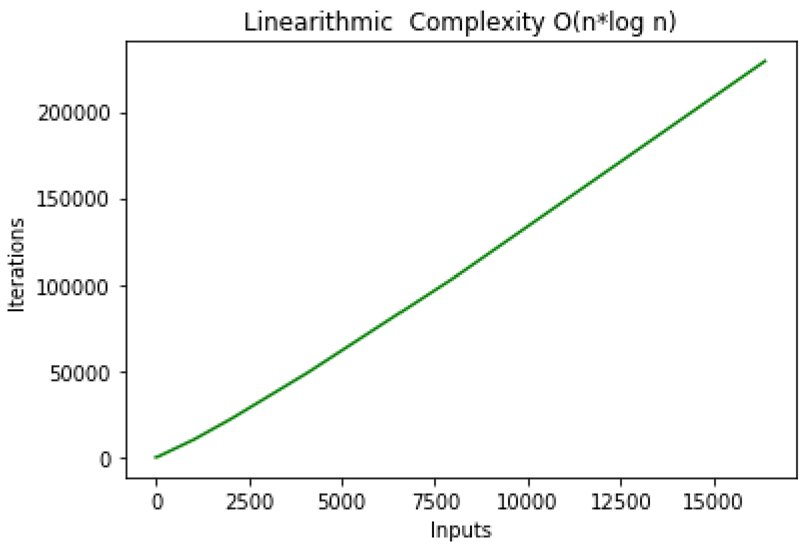


Linearithmic complexity O(n*log(n)) - slower then linear, but better then quadratic. Examples (merge sort, quick sort)

In [7]:

```
x = [1, 4, 8, 16, 64, 1024, 2048, 4096, 8192, 16384] #input(n)
y = [1, 4*2, 8*3, 16*4, 64*6, 1024*10, 2048*11, 4096*12, 8192*13, 16384*14] #number

plt.plot( x, y, 'g')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.title('Linearithmic  Complexity O(n*log n)')
plt.show()
```



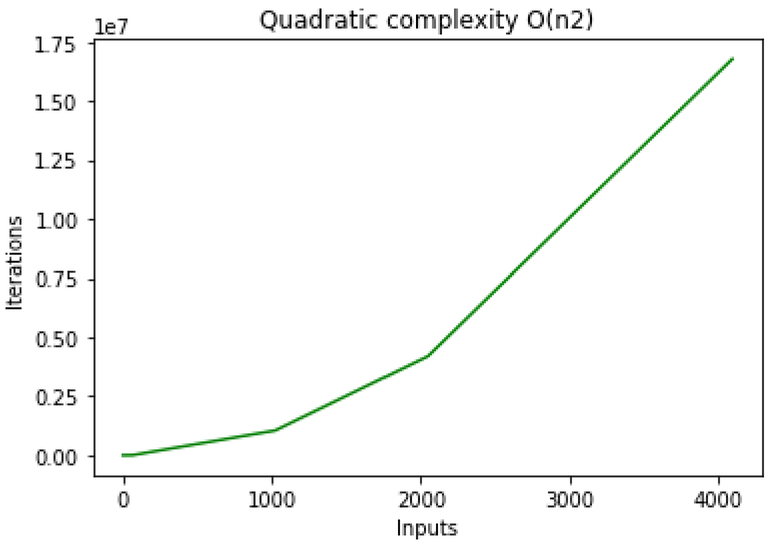
Quadratic complexity O(n^2) - operations grow as a square of the number of inputs. When we have nested iterations. Ex. bubble sort, selection sort, insertion sort

In [8]:

```
x = [1, 4, 8, 16, 64, 1024, 2048, 4096] #input(n)

y = [1, 16, 64, 256, 4096, 1048576, 4194304, 16777216] #number or iterations

plt.plot(x, y, 'g')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.title('Quadratic complexity O(n2)')
plt.show()
```



Exponential complexity O(2n) The algorithm takes twice the number of previous operations for every new element added. Example: recursive calculation of Fibonacci numbers.

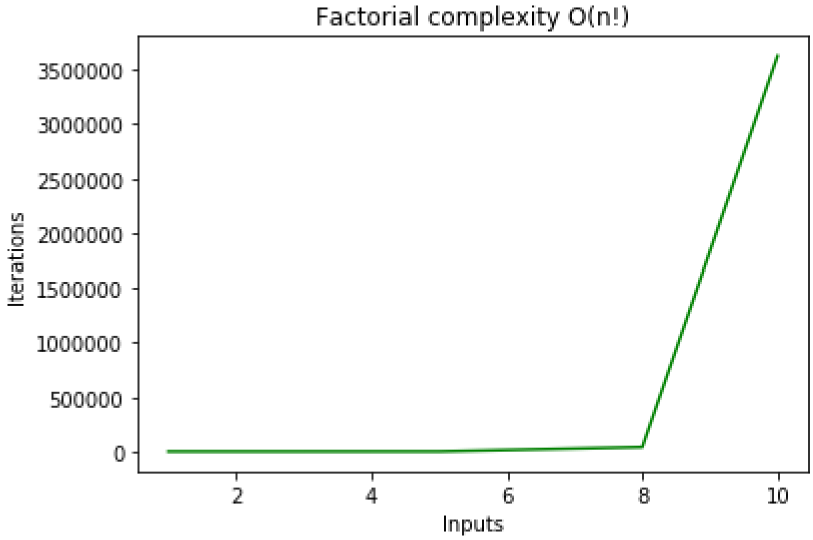
Factorial complexity O(n!) – very slow. Ex. traveling salesman

In [9]:

```
x = [1, 2, 3, 4, 5, 8, 10 ] #input(n)

y = [1, 2, 6, 24, 120, 40320, 3628800 ] #number or iterations

plt.plot(x, y, 'g')
plt.xlabel('Inputs')
plt.ylabel('Iterations')
plt.title('Factorial complexity O(n!)')
plt.show()
```



---

2 Recursion

2.1 Description

Recursion is a method of solving problems that relies on the capability of a function to continue calling itself until it satisfies a particular condition.

> “Recursion is when function calls itself until it doesn't” (c)🙈 Fun Fun Functions

Recursion function consists of two main points:

1. Base case (exit condition, otherwise infinite loop)
2. The rule how to move through function (alter input parameter)

Recursion is similar to while loop, because we also keep going through some piece of code again and again till exit or break condition (when we need to stop).

2.2 Example

Find n!:
Base case: If n = 0, then n! = 1
The rule: If n > 0, then n! = n*(n-1)!

So factorial of 3 or 3! is

In [10]:

```
def factorial(n):
    if n == 0: #base case or when we need to stop
        return 1
    else:
        return n * factorial(n - 1) #how to move

factorial_3 = factorial(3)
print(factorial_3) #6
```

6

Steps:

1. $factorial(3)$
2. $3 * factorial(2)$
3. $3 * 2 * factorial(1)$
4. $3 * 2 * 1$
5. 6

We can see step by step logic of recursive function. We are not multiplying till we dig dipper and hit base case - factorial(1), then we move forward and multiply.

---

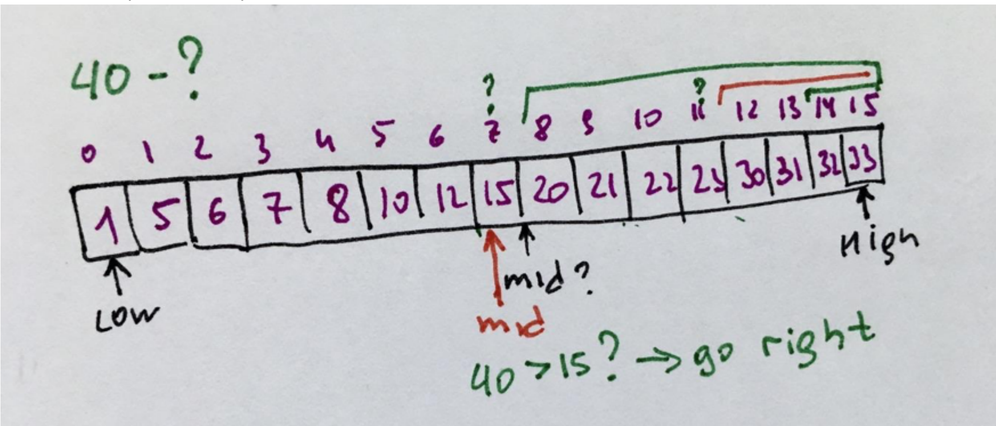
3 Binary Search

3.1 Description

> The goal of **binary search** is finding an item from a **sorted** list of items. It works by repeatedly dividing in half the length of the list that could contain the item, until it has narrowed down the possible locations to just one.(c) Binary search. Khan Academy

3.2 Example

Searching for number 40 (Picture 1).



Picture 1. Binary Search example

Steps (see the python code example below):

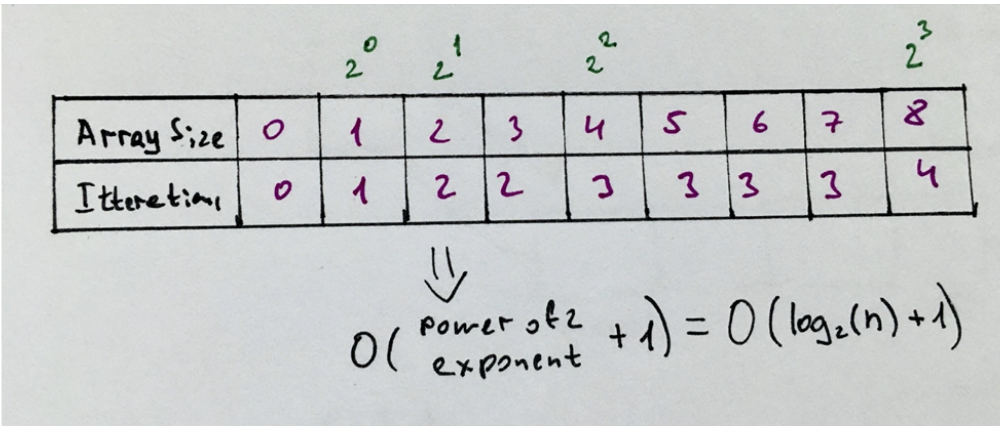
1. the lowest number = 1 and the highest number = 33
2. middle in array with length = 16 so the middle is 8, and the element with index (8-1) is 15.
3. 15 became the new lowest number and the highest number is the same = 33. 15 < 33
4. 40 is bigger then 15 so we move to the right part of the array and repeat same steps.
5. In the end we compare the number to 32 and the last one 33.
6. Then we can conclude that there is no such a number in the current sorted array.

Termination condition: while the lowest number smaller or equal to the highest number.

Criteria that algorithms is correct: The input array must be ordered either in increasing or in decreasing order according to the comparison operator above. It takes an array and element which has to be found and returns the index of the element (in case it is in the array). It returns index of the element.

3.3 Time complexity

Efficiency is not going to be as big as Linear O(n). Let’s consider how many iterations we have with different sizes of array (assumption from Udacity example).



Picture 2. Finding a pattern with binary search from Data Structures & Algorithms in Python. Udacity.

Current example (Picture 2) shows how many elements in the input array and how many iterations has to be done. Also we know that logarithms are the inverse of exponentials, so that if log(n)=x then $n = 2^x$.

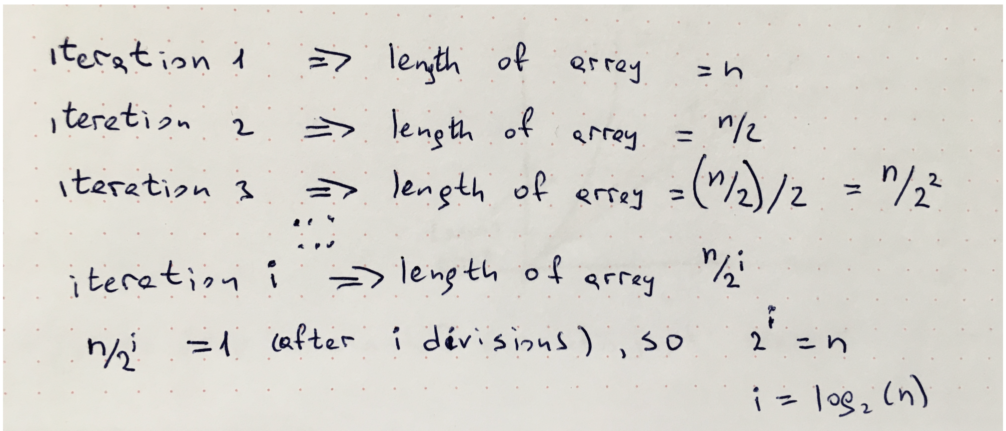
Back to example, we can see patterns behind, first of all, as array size grows the number iterations are growing much slower. And there is even bigger difference with huge inputs size. Second, we can see that number of iterations most of the time could be found as follows: if we represent array size as power of two, then number of iterations is its exponent+1, which we can furthermore represent by logarithm, $log(n) + 1$ where n - is input size. That happens because at every iteration (Picture 1), the searching space decreases by half, that means that the time needed to analyze it also decreases.

It follows that efficiency of the binary search could be:

$O(log(n) + 1$. From above explanation of asymptotic notation, we can conclude that adding one does not change efficiency much, so we get

$$O(log(n))$$

The same result I achieved, following the steps of divisions for each iteration (Picture 3)



Picture 3. Analysing iterations of binary search

Average case of binary search is $O(log(n))$. To prove this we will follow the same logic of halving array on each iteration.

Best case of binary search is $O(1)$ (when searched element is in the middle position)

The logarithm function grows very slowly. So O(log n) is much faster than O(n), but it gets a lot faster as the list of items grows.

Worse case	Average case	Best case
$(Olog(n))$	$\Theta(log(n))$	Ω(1)

3.4 Code example

[binary_search.py (algorithms/searching/binary_search.py)](algorithms/searching/binary_search.py)

---

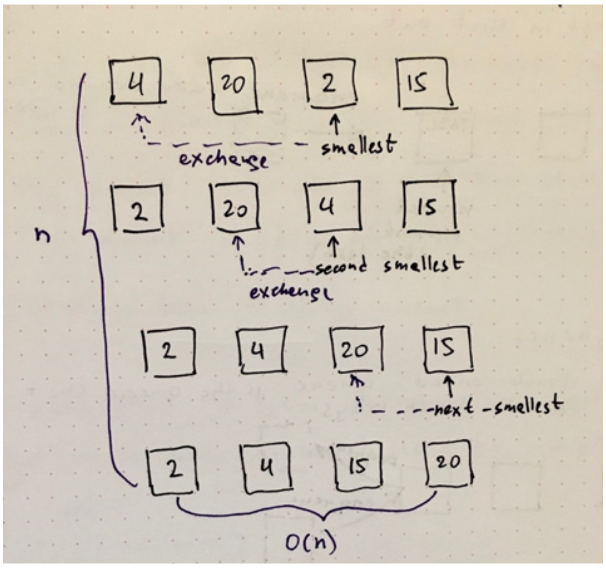
4 Selection Sort

4.1 Description

Selection sort – is the simplest sorting algorithm when we select the next smallest element and exchange it into place again and again. This algorithm sorts an array or list by repeatedly finding the minimum value (in case we are sorting in ascending order) from the array and placing it at the beginning of the list.

4.2 Example

Sort an array [4, 20, 2, 15]



Picture 4. Selection sort example.

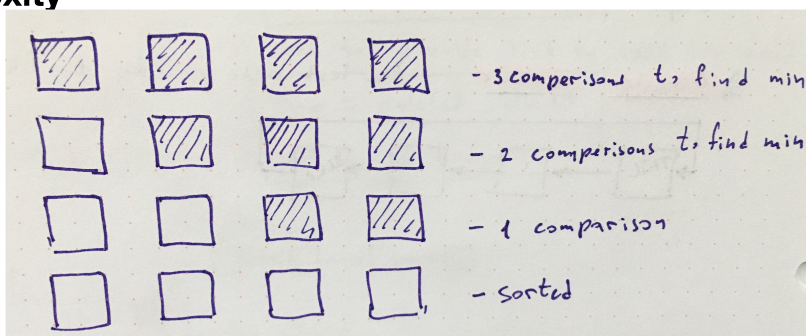
Steps following Picture 4 (see the python code example bellow):

1. Find the smallest number which is 2 and exchange it with the first number so we get [2, 20, 4, 15].
2. Find the second-smallest number (4) and exchange it with the second number of the array, so we get [2, 4, 20, 15]
3. Do over and over again finding the next-smallest number, and exchanging it into the correct position until the array is sorted.

Termination condition: If the next-smallest is the smallest element of the array.

Criteria that algorithms is correct: Input: array of n elements. Output is the same length array sorted in increasing or decreasing order.

4.3 Time complexity



Picture 5. Selection sort comparisons.

We have n steps and on each step we do the number comparisons one less then number of unsorted elements (Picture 5). In our example we did (4-1), (4-2), (4-3) comparisons untill we get sorted array. Let's say the number of iterations n-1, then n-2, and so on, which is following summation notation evaluates to $(n+1)(n/2)$ or $n^2/2 + n/2$. (Analysis of selection sort (article). Khan Academy. [https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort (https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort)](https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort)). Summation notation (Gauss formula) is explained in "Back to Back" episode

Following asymptotic notation, we dropped constant and low-order term. So the time complexity in the worst case and average case (if we get nearly sorted array) is

$$O(n^2)$$

Even if we get sorted array, we still need to go through each level that's why the best case is $O(n^2)$.

Worse case	Average case	Best case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$

That is, no matter how many elements we sort so the space complexity is O(1).

4.4 Code example

[selection_sort.py (algorithms/sorting/selection_sort.py)](algorithms/sorting/selection_sort.py)

---

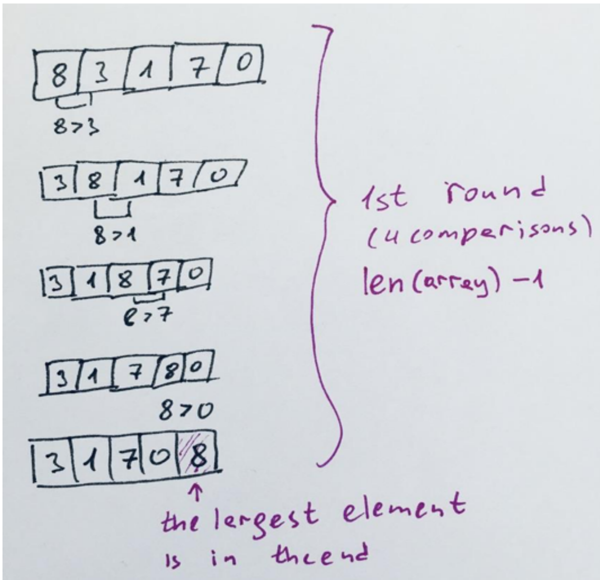
5 Bubble Sort

5.1 Description

Bubble sort – is an approach when in each iteration the largest element in the array will bubble up to the top. The goal of the algorithm is by checking two items at a time, rearrange those not already in ascending order (in the same array) from left to right.

5.2 Example

Sort an array [8, 3, 1, 7, 0] The array = [8, 3, 1, 7, 0] has length(n) equal 5, so n= 5 (Picture 6)



Picture 6. Bubble sort example.

Steps:

1. First round took 4 comparisons and 3 swaps, we get [3, 1, 7, 0, 8]
2. Second round took 4 comparisons and 2 swaps, we get [1, 3, 0, 7, 8]
3. Third round took 4 comparisons and 1 swap we get [1, 0, 3, 7, 8]
4. Forth round took 4 comparisons and 1 swap, we get the sorted array [0, 1, 3, 7, 8]

Termination condition: No more swaps needed - array is sorted.

Criteria that algorithms is correct: Input: array of n elements. Output is the same length sorted array.

5.3 Time complexity

If n is size of array, in the worst case we always do the swap so it takes n-1 rounds to sort the array, each round takes n-1 comparisons. Means that in the worst case we have the same number of swaps and comparisons, which is (n-1). Then, following the summation notation (Gauss formula), we inject (n-1) in Gauss formula and get $(n-1)((n-1)+1)/2$= $((n-1)*n)/2$

According asymptotic notation we can simplify it to n^2 , so we get the running time of bubble sorting for the worst case is:

$$O(n^2)$$

The above example (Picture 6) represents average case - sometimes I do swaps, sometimes not and we do not know how many swaps we need.

Best case of bubble sort is $O(n)$. For example, when array is already sorted (or there is only one element in the array), we need to make comparisons, but we do not do swaps.

Worse case	Average case	Best case
$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$

Space complexity is O(1) – everything we do in the same array.

5.4 Code example

[bubble_sort.py (algorithms/sorting/bubble_sort.py)](algorithms/sorting/bubble_sort.py)

---

6 Merge sort

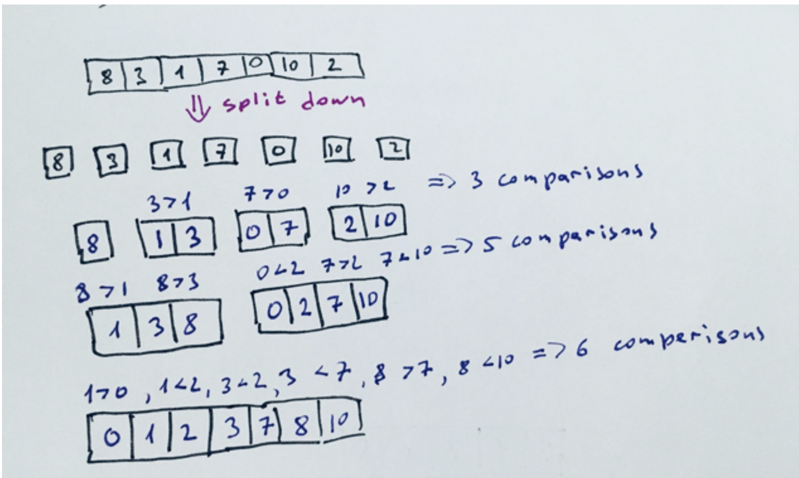
6.1 Description

Merge sort is an algorithm when we split the array down as much as possible till base case (base case is one item) and build it back up (merging) doing the comparisons and sorting on each step. Divide and Conquer principles here.

6.2 Example

Example 1.

Sort an array [8, 3, 1, 7, 0, 10, 2] The array = [8, 3, 1, 7, 0, 10, 2] has length(n) equal 7, so n= 7 (Picture 7)



Picture 7. Merge sort example.

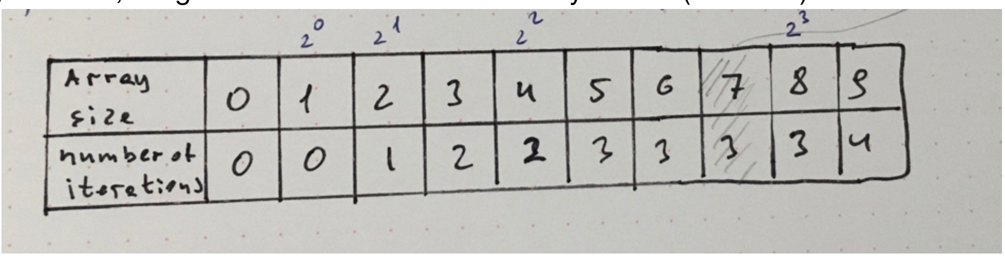
Steps:

1. Split array
2. Start building back, making comparisons which one is smaller

Termination condition: it is recursive algorithm so we have the base case which is one.

If n is the size of the array that we are building, the number of comparisons is going to be n - 1. In our example we can say that approximately we are going to do 7 comparisons at each round (it is very approximately!). Now we need to find how many rounds we do. For array of 7 element, I have done 3 rounds.

If we draw array of result, we get the similar table as for binary search (Picture 8).



Array size	0	1	2	3	4	5	6	7	8	9
number of iterations	0	0	1	2	2	3	3	3	3	4

Picture 8. Merge sort iterations.

6.3 Time complexity

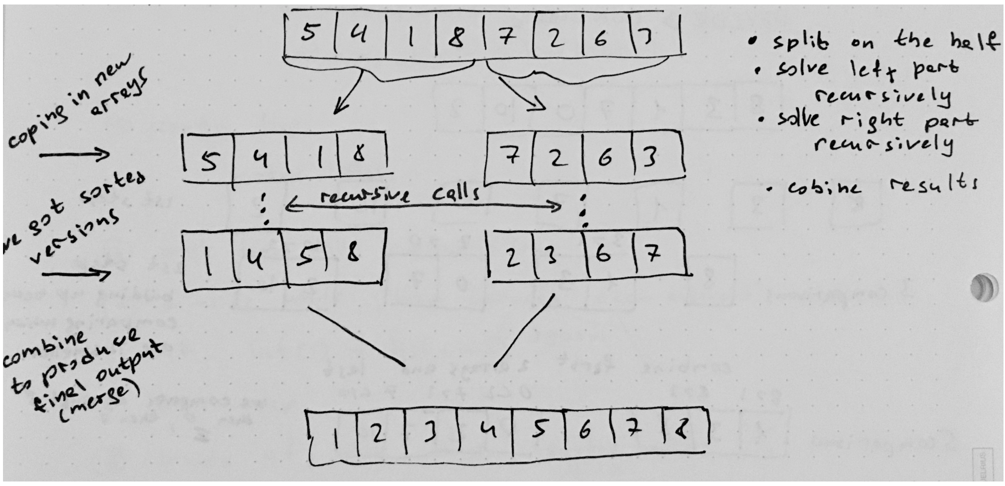
We get approx. $log(n)$ for number of rounds, and n comparisons on the each round, so the efficiency is:

$$O(n * log(n))$$

Let's analyze algorithm.

Merge sort is recursive algorithm.

Example 2:



Picture 9. Merge sort algorithm.

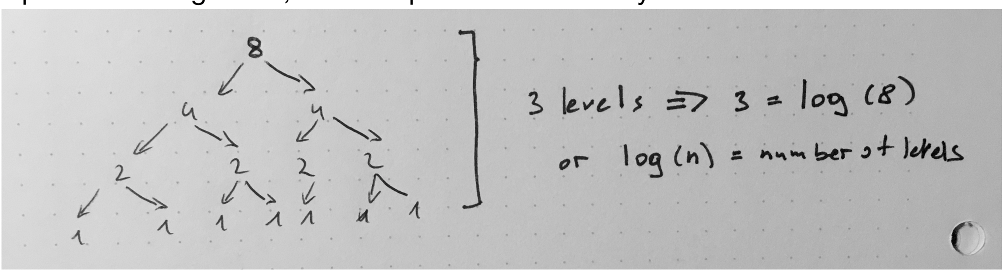
Steps (Picture 9):

1. Split array on the half
2. Recursively sort 1st part of the array
3. Recursively sort 2nd part of the array
4. Merge two sorted sublists into one array.

We have two routines:

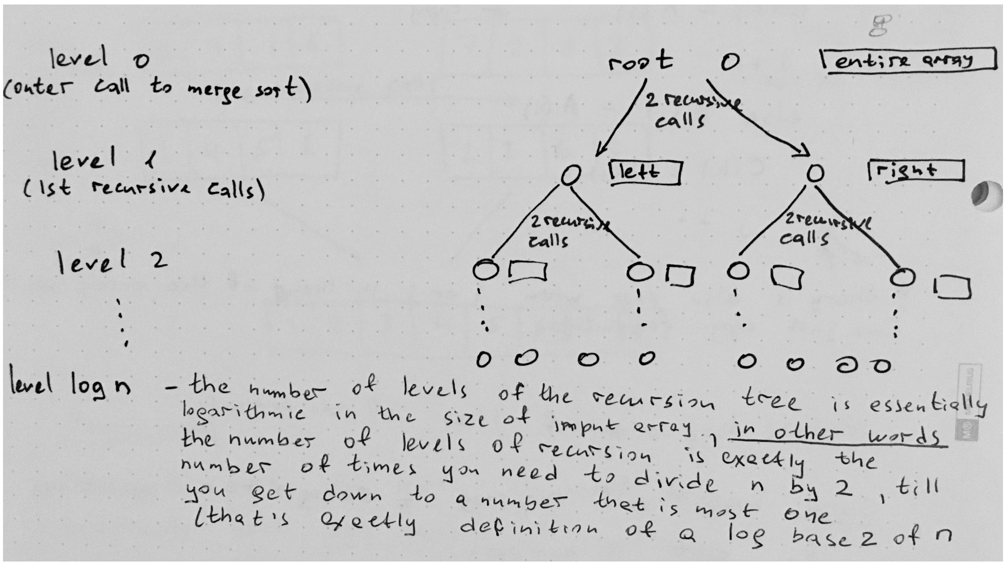
1. split
2. merge

So let's see the split routine in general, for example if we have array of 8 elements:



Picture 10. Number of levels for split routine.

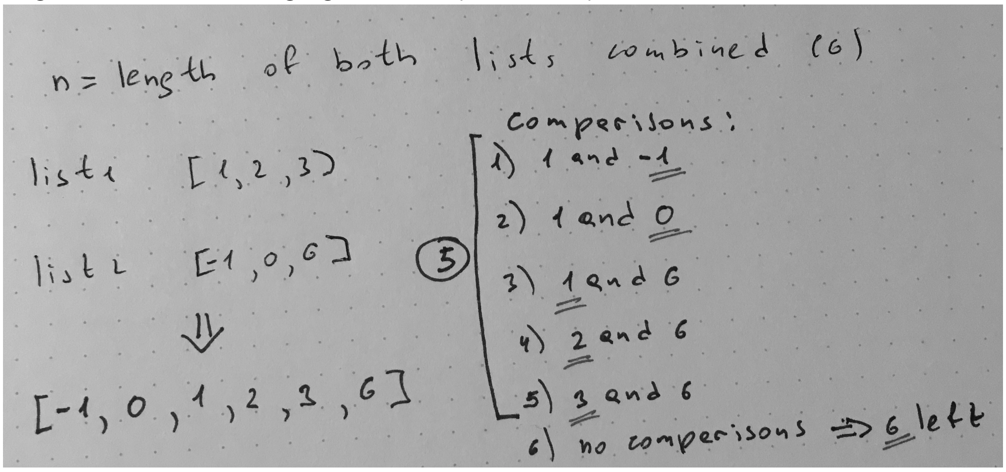
As we can see on Picture 10, for array n=8, we have got 3 levels of splitting which is log(8) and since n = 8, number of levels is log(n). Now, we need to find how many basic cases we have? If we count all basic cases, it is equal to 8 or number of basic cases = n.



Picture 11. Merge sort algorithm split routine and number of levels.

Conclusion: how many split steps can happen? Since we are halving array every step - the amount of splittings is log(n) (Picture 11).

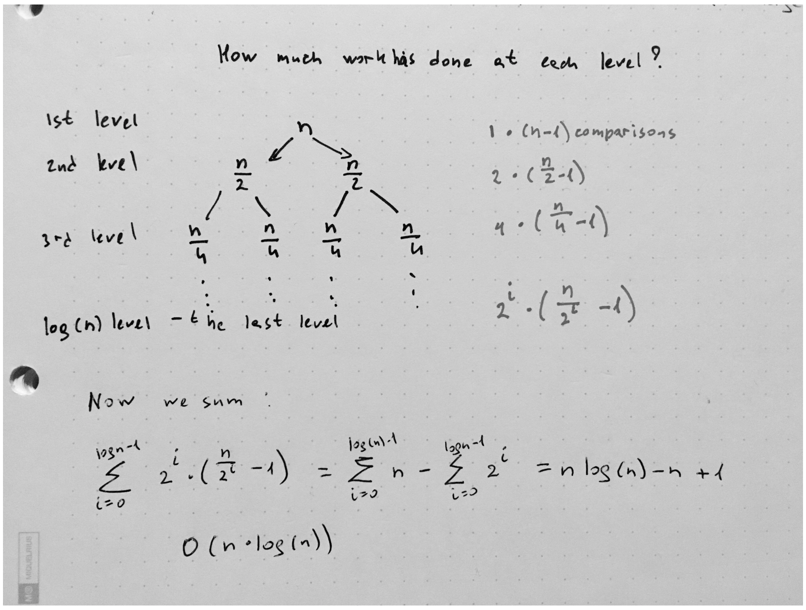
Now let's see merge routine. I am merging two lists (Picture 12)



Picture 12. Merge routine.

As we can see on the picture 12, we have 5 comparisons for n = 6, which means number of comparisons is (n-1)

Let's continue with concluding how much work have done at each level in terms of n (Picture 13).



Picture 13. Recoursive tree and amount of work at each level.

We can see from pictures 11 and 13 that at each level i = 0, 1, 2, ...log(n), there are 2^i subproblems, each subproblem of size $n/2^i$. It means, for example, if we have array of n = 8, then at level 2, there are 2^2 subproblems, each of size $8/2^2$. We concluded that the amount of work at any level i is going to be $2^i(n/2^i - 1)$. How to find all work for the worst case? We need to calculate sum of work at each level from level 0 to level $log(n) - 1$, since the last level creates 0 comparisons (Picture 10). After calculation sum with Gauss Formula (summation formula), we will get (Picture 13):

$$O(n * log(n))$$

Time complexity is the same in all 3 cases (worst, average and best) as we always need to divide the array and then merge.

Worse case	Average case	Best case
$O(n * log(n))$	$\Theta(n * log(n))$	$\Omega(n * log(n))$

Space complexity is O(n) – on each step we need to create new arrays and copy into it.

Merge sort is often used for sorting a linked list.

6.4 Code example

[merge_sort.py (algorithms/sorting/merge_sort.py)](algorithms/sorting/merge_sort.py)

---

7 Quick Sort

7.1 Description

Quick sort is one of the most efficient sorting algorithms. Quick sort characteristics:

- Divide and conquer - splits the array into smaller arrays until it ends up with an empty array, or one that has only one element, before recursively sorting the larger arrays.

- In-place - doesn't create any copies of the array.

The goal of the algorithm is to use divide and conquer method to gain the same advantages as the merge sort, but without creating new arrays. But in case the list is not divided in half, it would take more time for algorithm.

We have two subroutines:

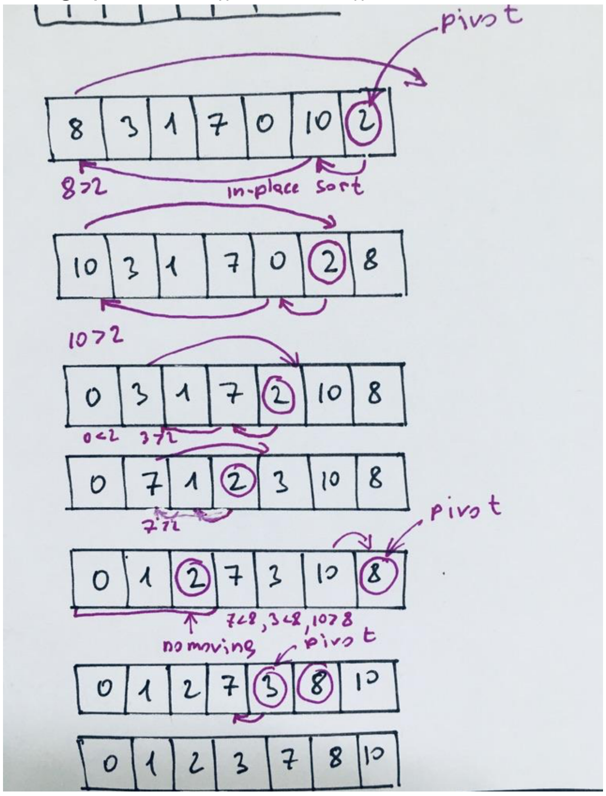
- split function, which splits the input
- partitioning subroutine - the goal is to choose the pivot

> Pivot - the value within the partitioning space that I want to find the position for. (c) Back To Back SWE. YouTube. The Quicksort Sorting Algorithm: Pick A Pivot, Partition, & Recurse

7.2 Example

Example 1. General overview how algorithm works.

Sort the list [8, 3, 1, 7, 0, 10, 2] using quick sort ((Picture 14))



Picture 14. Quick sort example.

Steps:

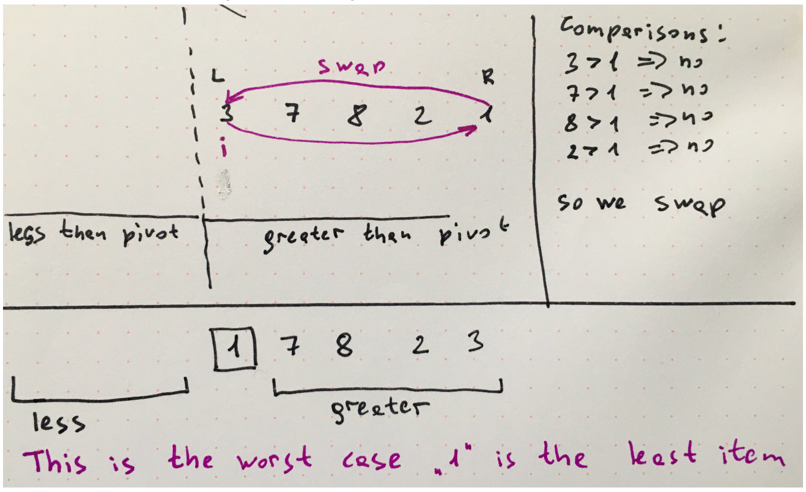
1. pick one of the values randomly (pivot is the random number)
2. move all values larger than it above it
3. move all values lower than it below it
4. continue on recursively picking a pivot in the upper and lower sections. The main goal is cutting array on the two spaces (less then pivot and more then pivot)
5. sort them till whole array is sorted

Termination condition: the size of the array we are sorting recursively is one (the base case). Criteria that algoritm is correct: output is sorted array the same length as input

7.3 Time complexity

The worst case is when pivot is the greatest item or the least item.

The best case is when pivot is median of partition space.



Picture 15. Worst case scenario by Back To Back SWE "The Quicksort Sorting Algorithm: Pick A Pivot, Partition, & Recurse".

In the worst case, when we've gotten array which near sorted for example [3, 7, 8, 2, 1] (Picture 15) or sorted [12, 10, 3, 2, 1], we need to compare each element in the array again and again, the same like in Bubble sort algorithm. To compare each item on the first level n takes (n-1) comparisons, then ((n-1)-1).....and so on until we get down to one item. But we are not halving our array. Using math and Guss formula, we need to sum the work done on each level from 1 to (n-1), which is $(n-1)((n-1)+1)/2$ and can conclude that it takes in the worst scenario $(n-1)*n/2$ comparisons If we drop constants, the worst case time complexity is $O(n^2)$.

In the best and average case, when pivot is moved to the middle (the pivot is median of partitioning space). It means that in this case we will halve the array every time and we know that when we are halving array, it takes log(n) levels. And again we need to sum the work on each level. Explanation of complexity of work done at each level the same as for merge routine. (Picture 13.)

The result and logic we see is similar to merge, because we do the same: splitting and working on subproblems recursively. Therefore, the time complexity in the worst case is

$$O(n * log(n))$$

> Quicksort is faster in practice because it hits the average case way more often than the worst case. (c) Grokking Algorithms.

Worse case	Average case	Best case
$O(n^2)$	$\Theta(n * log(n))$	$\Omega(n * log(n))$

Quicksort is used by Chrome (V8 engine) javascript sorting.

7.4 Code example

quick_sort.py (algorithms/sorting/quick_sort.py)

8 Resources

1. Back To Back SWE. YouTube. https://www.youtube.com/c/BackToBackSWE (https://www.youtube.com/c/BackToBackSWE)
2. Data Structures & Algorithms in Python. Udacity. https://classroom.udacity.com/courses/ud513 (https://classroom.udacity.com/courses/ud513)
3. Algorithms. Khan Academy. https://www.khanacademy.org/computing/computer-science/algorithms (https://www.khanacademy.org/computing/computer-science/algorithms)
4. Bhargava, A. (2016). Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People (1st ed.). Manning Publications.
5. Divide and Conquer, Sorting and Searching, and Randomized Algorithms(Week 1). Coursera. https://www.coursera.org/learn/algorithms-divide-conquer (https://www.coursera.org/learn/algorithms-divide-conquer)
6. Woltmann, S. (2021, January 13). Selection Sort – Algorithm, Source Code, Time Complexity. https://www.happycoders.eu/algorithms/selection-sort/ (https://www.happycoders.eu/algorithms/selection-sort/)
7. Fun Fun Function. (2015, August 24). Recursion - Part 7 of Functional Programming in JavaScript. YouTube. https://www.youtube.com/watch?v=k7-N8R0-KY4 (https://www.youtube.com/watch?v=k7-N8R0-KY4)