**Algorithms:**

**Data structures and algorithms is a very important part in the data science while building any machine learning models. It helps in optimizing our task and models.**

**Question: Why should we learn it in context of data science?**

Answer: data science is all about building business models and using computer science we should be able to solve our customers problem. To understand it more intuitively we will discuss the amazon example and these fundamental blocks are used everywhere in day-to-day life.

Suppose you are data scientist at Amazon and your job is to recommend good product to customers and when you recommending these products there should be high probability that the customers will purchase it. Here we need to understand that customers want results very fast and quick plus we need to increase our sales too. Suppose we made a model using data science and its giving recommendation in one or two minutes then its of no use because here customer want results in seconds. So, we need to optimize our model and for that DSA plays very crucial role because it teaches us how we can reduce time or space complexity by understanding the fundamentals of algorithms.

**We will be looking into the following algorithms in detail**:

1. Linear Search
2. Binary Search
3. Bubble Sort
4. Selection Sort
5. Insertion Sort
6. Merge Sort

**Let’s get started:**

1. **Linear Search:** In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **linear search** or **sequential search** is a method for finding an element within a [list](https://en.wikipedia.org/wiki/List_(computing)). It sequentially checks each element of the list until a match is found or the whole list has been searched.

**Ques: How to write code for linear search?**

**Ans: We will be following the below steps:**

1. **Understanding the logic behind it:** In linear search we are searching whether the element is present in the list or not one by one.
2. **Coding part:** Since we are searching the element so it’s like we are going through each and every element of the list and in python we can use loops for it. Right!
3. **What should be the answer! 🡪** if the element is present in the list then we need to return the index of that element and if element is not present in the list then return -1.

Example ->

INPUT ->

a = [4,5,8,10,11,6,2,1]

target = 10

OUTPUT -> 3 (index value)

|  |  |
| --- | --- |
| **Linear search** | |
| **Class** | [Search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) |
| [**Worst-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(*n*)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Best-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(1)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Average**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(*n*)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Worst-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**space complexity**](https://en.wikipedia.org/wiki/Space_complexity) | [*O*(1)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) iterative |

1. **Binary Search:** In [computer science](https://en.wikipedia.org/wiki/Computer_science), binary search, also known as half-interval search, logarithmic search, or binary chop, is a [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) that finds the position of a target value within a [sorted array](https://en.wikipedia.org/wiki/Sorted_array). Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

**Key points:**

1. Our list should be sorted in order for binary search to work.
2. Here we are comparing the target value with the middle value of the array.
3. If the target value is not equal to the middle value and less than the middle value then we discard the right side of the list.
4. If the target value is not equal to the middle value and greater than the middle value then we discard the left side of the list.
5. We keep on discarding the left side or right side until we reach the position where target value is equal to middle element, if not then we can say that target value is not present in the list.

**Ques**: Why we always comparing the middle element i.e., we discarding 50 percent of our list at each iteration? Why not 20 percent or some other value?

**Answer**: Lets take an example of 20 percent- Suppose our element is not present in this 20 percent of our list then we can only discard 20 percent of the list so clearly it will take more time to get the index of our target element hence we take 50 percent.

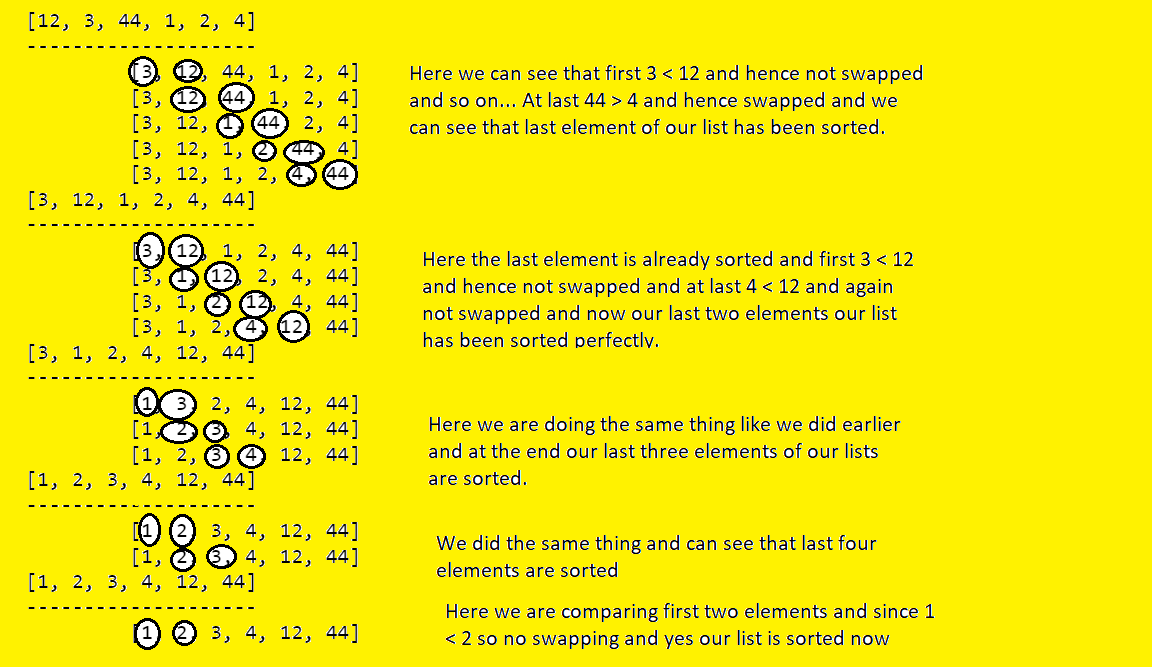
**Ques: How to code for binary search?**

**Answer:** Before starting coding for binary search, we should we clear about what we are doing end to end. We should know how can we complete the logic with coding. If we are using python then we should be aware about the programming language and should know how to use it.

So let’s start coding

|  |  |
| --- | --- |
| **Binary search algorithm** | |
| Visualization of the binary search algorithm where 7 is the target value | |
| **Class** | [Search algorithm](https://en.wikipedia.org/wiki/Search_algorithm) |
| **Data structure** | [Array](https://en.wikipedia.org/wiki/Array_data_structure) |
| [**Worst-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(log *n*)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Best-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(1)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Average**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**performance**](https://en.wikipedia.org/wiki/Time_complexity) | [*O*(log *n*)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
| [**Worst-case**](https://en.wikipedia.org/wiki/Best,_worst_and_average_case)[**space complexity**](https://en.wikipedia.org/wiki/Space_complexity) | [*O*(1)](https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions) |
|  |  |

1. **Bubble Sort: Bubble sort**, sometimes referred to as **sinking sort**, is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that repeatedly steps through the input list element by element, comparing the current element with the one after it, [swapping](https://en.wikipedia.org/wiki/Swap_(computer_science)) their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort), is named for the way the larger elements "bubble" up to the top of the list.

**Example:** We will understand bubble sort with the following example where we taking an unsorted list: a = [12,3,44,1,2,4] and we trying to understand the intuition of bubble sort visually.

Now the big question is we understood the logic for the bubble sort but how to code guys?

1. We are comparing each element with the following element and if preceding element is larger then we are swapping. To go through each and every element we can use loops right!
2. While comparing the elements we need to put the pointer (lets j) and we comparing j with j+1 and doing the swapping operation.

4 . Selection sort: In [computer science](https://en.wikipedia.org/wiki/Computer_science), **selection sort** is an [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) [comparison](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm). It has an [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2) [time complexity](https://en.wikipedia.org/wiki/Time_complexity), which makes it inefficient on large lists, and generally performs worse than the similar [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where [auxiliary memory](https://en.wikipedia.org/wiki/Auxiliary_memory) is limited.

Understanding the logic:

We’ll be taking the following example to see how selection sort works and will try to understand the intuition behind it.

We will set pointer called min\_index and will compare this with all the elements in our list and if any element is less than the min\_index then we will swap the positions.

[22, 4, 8, 66, 7, 1, 11, 2]

Iteration 1: Here min\_index = 22 we will compare as follows.

|  |  |
| --- | --- |
| 22 > 4 (Yes) /current item | min\_index = 4 |
| 4 > 8 (No) | min\_index = 4 |
| 4 > 66 (No) | min\_index = 4 |
| 4 > 7 (No) | min\_index = 4 |
| 4 > 1 (Yes) | min\_index = 1 |
| 1 > 11 (No) | min\_index = 1 |
| 1 > 2 (No) | min\_index = 1 |

We have compared all of our elements and now our real minimum element 1 and we will swap it with our current item i.e., 22. And we will keep doing the same logic and will get the sorted list in the end.

[1, 4, 8, 66, 7, 22, 11, 2]

--------------------

[1, 2, 8, 66, 7, 22, 11, 4]

--------------------

[1, 2, 4, 66, 7, 22, 11, 8]

--------------------

[1, 2, 4, 7, 66, 22, 11, 8]

--------------------

[1, 2, 4, 7, 8, 22, 11, 66]

--------------------

[1, 2, 4, 7, 8, 11, 22, 66]