Published in codeburst

Meet Zaveri Follow

Mar 10, 2018 · 11 min read · Listen

An intro to Algorithms: Searching and Sorting algorithms

One of the seemingly most-overused words in tech is “algorithm”. From the apps on your phone to the sensors in your wearables and how posts appear in your Facebook News Feed, you’ll be pushed to find a service that isn’t powered by some form of algorithm. I am too fascinated how algorithms made an impact in our day-to-day lives

> An algorithm is a finite sequence of precise instructions for performing a computation or for solving a problem.

For example, here is an algorithm for singing that annoying song “99 Bottles of Beer on the Wall”, for arbitrary values of 99:

```
BOTTLESOFBEER(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```


As Donald Knuth wrote in The Art of Computer Programming– which could be described as the encyclopedia of algorithms and data structures– it’s a confusing word

1. **Input** -An algorithm has input values from a specified set.
2. **Output** -From each set of input values an algorithm produces output values from a specified set. The output values are the solution to the problem.
3. **Definiteness** -The steps of an algorithm must be defined precisely.
4. **Correctness** -An algorithm should produce the correct output values for each set of input values.
5. **Finiteness** -An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.
6. **Effectiveness** -It must be possible to perform each step of an algorithm exactly and in a finite amount of time.
7. **Generality** -The procedure should be applicable for all problems of the desired form, not just for a particular set of input values

Now before heading up to main topic, I want to share the basics of analysis of the algorithms including time complexity and space complexity.

Time Complexity and Space Complexity in algorithms

Always a question arises -

> When does an algorithm provide a satisfactory solution to a problem?

- One measure of efficiency is the time used by a computer to solve a problem using the algorithm, when input values are of a specified size
- second measure is the amount of computer memory required to implement the algorithm when input values are of a specified size


There are three types of time complexity — **Best, average and worst case**.

In simple words for an algorithm, if we could perform and get what we want in just one(eg. on first instance) **computational** approach, then that is said as **O(1) i.e. Time complexity here falls into "Best case" category.**

Say for example, same algorithm results into many **iterations/recursions** or say `n` times it had to perform to get the result, then the example used for this algorithm describes it's **worst case time complexity.**

Below are some common time complexities with simple definitions. Feel free to check out Wikipedia, though, for more in-depth definitions.

- ***Constant time*** has an order of growth `1`, for example: `a = b + c`.
- ***Logarithmic time*** has an order of growth `LogN`, it usually occurs when you're dividing something in half (binary search, trees, even loops), or multiplying something in same way.
- ***Linear***, order of growth is `N`, for example

```
int p = 0;
for (int i = 1; i < N; i++)
  p = p + 2;
```

- ***Linearithmic***, order of growth is `n*logN`, usually occurs in divide and conquer algorithms.
- ***Cubic***, order of growth `N^3`, classic example is a triple loop where you check all triplets:

```
int x = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
          x = x + 2
```

- ***Exponential***, order of growth `2^N`, usually occurs when you do exhaustive search, for example check subsets of some set.

https://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm

Simple example with code -

```
1  var invalidChar = 'b';
2  var arr = [ 'a', 'b', 'b', 'd', 'e' ];
3  var ptr = 0, n = arr.Length;
4  for (int i = 0; i < N; i++){
```

ts.js hosted with ❤ by GitHub view raw

So scenario on time complexity for this above given example would be -

int i=0;	This will be executed only once. The time is actually calculated to i=0 and not the declaration.
i<N;	This will be executed N+1 times
i++ ;	This will be executed N times
if(arr[i]!=invalidChar)	This will be executed N times
arr[ptr]=arr[i];	This will be executed N times (in worst case scenario)
ptr++;	This will be executed N times (in worst case scenario)

Asymptotic Notations

Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms:

Big Oh (O)

Big Oh is often used to describe the worst-case of an algorithm by taking the highest order of a polynomial function and ignoring all the constants value since they aren't too influential for sufficiently large input.

Big Omega (Ω)

Big Omega is the opposite of Big Oh, if Big Oh was used to describe the upper bound (worst-case) of a asymptotic function, Big Omega is used to describe the lower bound of a asymptotic function. In analysis algorithm, this notation is usually used to describe the complexity of an algorithm in the best-case, which means the algorithm will not be better than its **best-case**.

Big Theta (Θ)

When an algorithm has a complexity with lower bound = upper bound, say that an algorithm has a complexity O(n log n) and Ω(n log n), it's actually has the complexity Θ(n log n), which means the running time of that algorithm always falls in n log n in the best-case and worst-case

Algorithms in plain English: time complexity and Big-O notation

Every good developer has time on their mind. They want to give their users more of it, so they can do all those things...

medium.freecodecamp.org

or look at Shilpa Jain 's article —

The Ultimate Beginners Guide To Analysis of Algorithm

The other day, I came across a post on StackOverflow which read "Is theoretical computer science(TCS) useful?". I was...

codeburst.io

Space Complexity

Space complexity deals with finding out how much (extra)space would be required by the algorithm with change in the input size. For e.g. it considers criteria of a data structure used in algorithm as ***Array*** or ***linked list.***

How to calculate space complexity of an algorithm — https://www.quora.com/How-do-we-calculate-space-time-complexity-of-an-algorithm

I'll cover up at least 2 practically used algorithms in each category based on searching and sorting. I had written pseudocode and explanation in my personal notes(images here).

Searching Algorithms

Search algorithms form an important part of many programs. Some searches involve looking for an entry in a database, such as looking up your record in the IRS database.


The general searching problem can be described as follows: Locate an element `x` in a list of distinct elements `a1,a2,...an` or determine that it is not in the list. The solution to this search problem is the location of the term in the list that equals `x` and is 0 if `x` is not in the list.

Linear Search

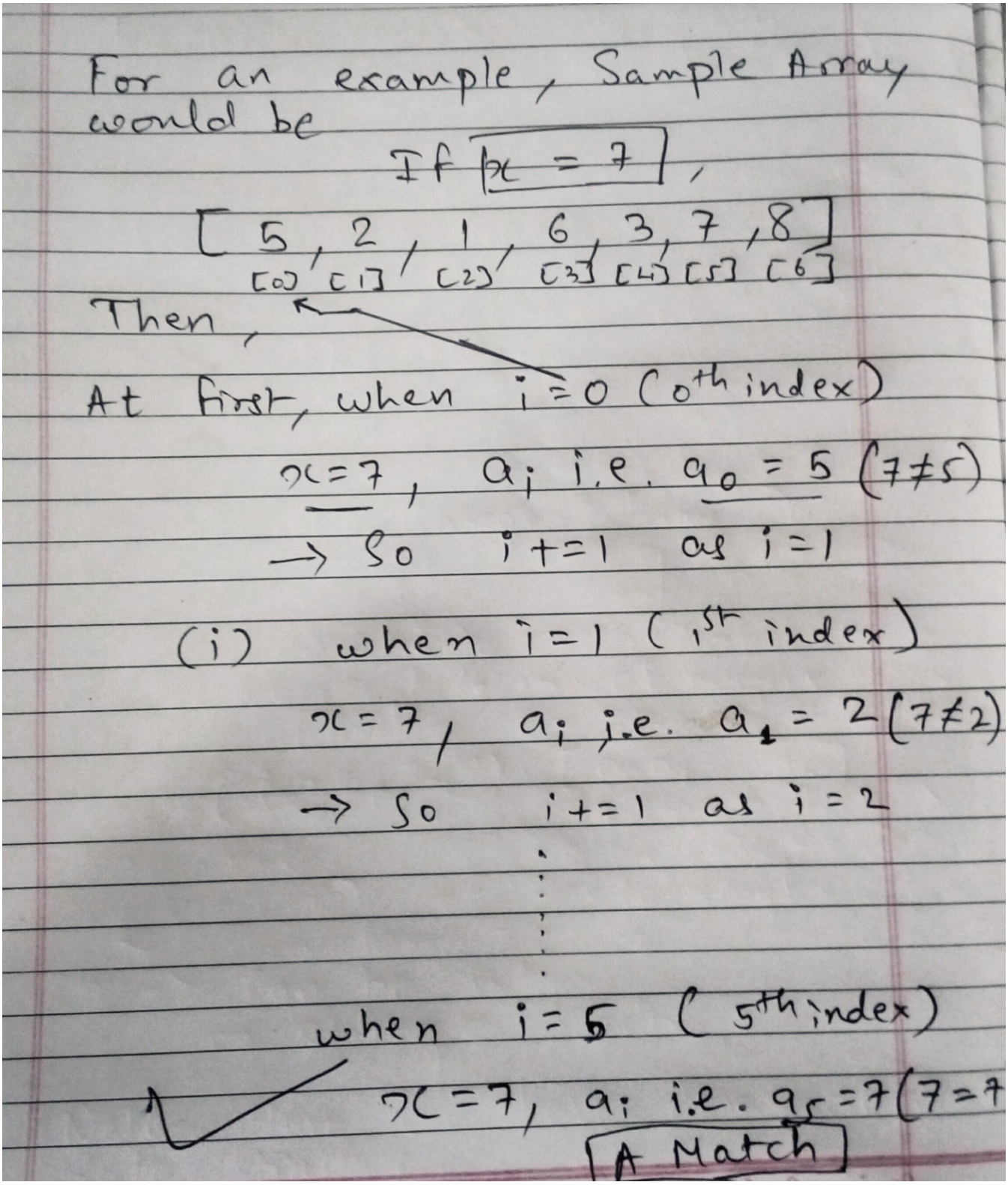
The linear search is the algorithm of choice for short lists, because it's simple and requires minimal code to implement. The linear search algorithm looks at the first list item to see whether you are searching for it and, if so, you are finished. If not, it looks at the next item and on through each entry in the list.

How does it work ?

Linear search is the basic search algorithm used in data structures. It is also called as sequential search. Linear search is used to find a particular element in an array. It is not compulsory to arrange an array in any order (Ascending or Descending) as in the case of binary search.

Into Linear Search

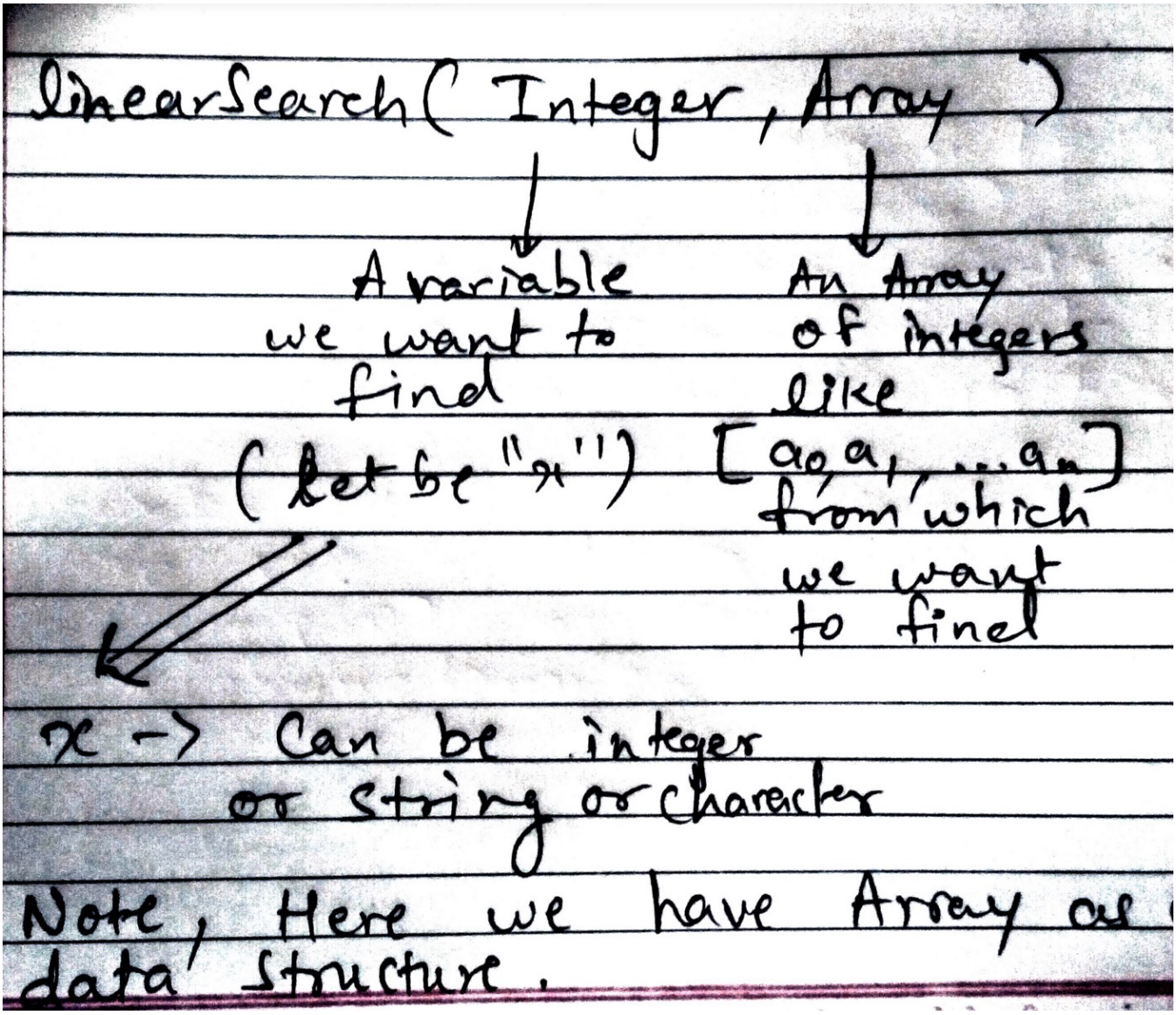
Given a sample array as shown in figure and value we want to search for is 7, then it'll traverse in linear way.

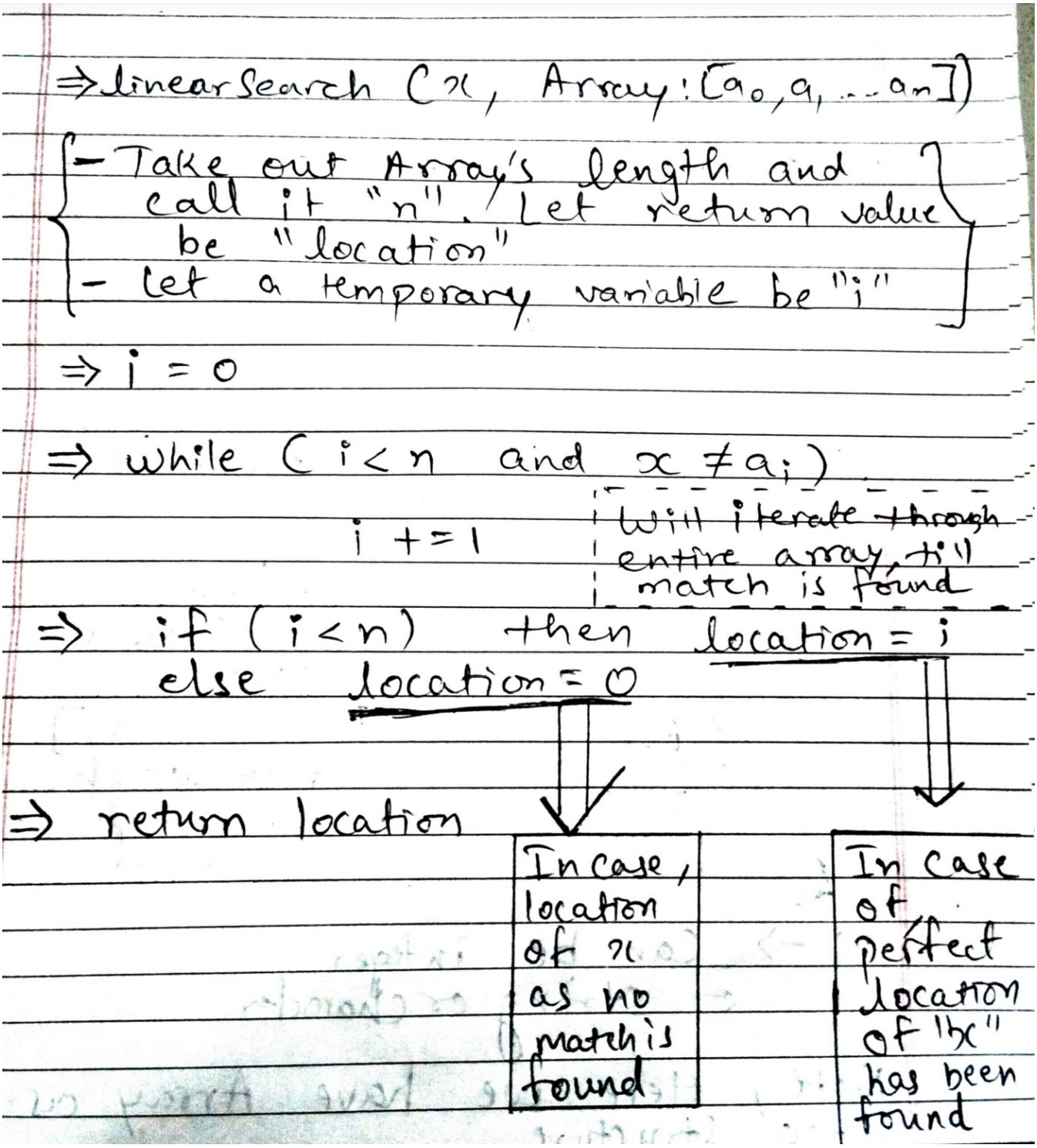



P.S. My handwriting could be sometimes unreadable

Pseudocode

Here is the pseudocode as divided into two images -





Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Time complexity

The time complexity of above algorithm is O(n).

```
# can be simply done using 'in' operator
# Example:
# if x in arr:
#   print arr.index(x)

# If you want to implement Linear Search in python

# Linearly search x in arr[]
# If x is present then return its location
# else return -1

def search(arr, x):

    for i in range(len(arr)):

        if arr[i] == x:
            return i

    return -1
```

linearsearch.py hosted with ❤ by GitHub view raw

Binary Search

Binary Search is one of the most fundamental and useful algorithms in Computer Science. It describes the process of searching for a specific value in an ordered collection.

Binary search is a popular algorithm for large databases with records ordered by numerical key. Example candidates include the IRS database keyed by social security number and the DMV records keyed by driver's license numbers. The algorithm starts at the middle of the database — if your target number is greater than the middle number, the search will continue with the upper half of the database. If your target number is smaller than the middle number, the search will continue with the lower half of the database. It keeps repeating this process, cutting the database in half each time until it finds the record. This search is more complicated than the linear search but for large databases it's much faster than a linear search.

1. **Pre-processing** — Sort if collection is unsorted.
2. **Binary Search** — Using a loop or recursion to divide search space in half after each comparison.
3. **Post-processing** — Determine viable candidates in the remaining space.

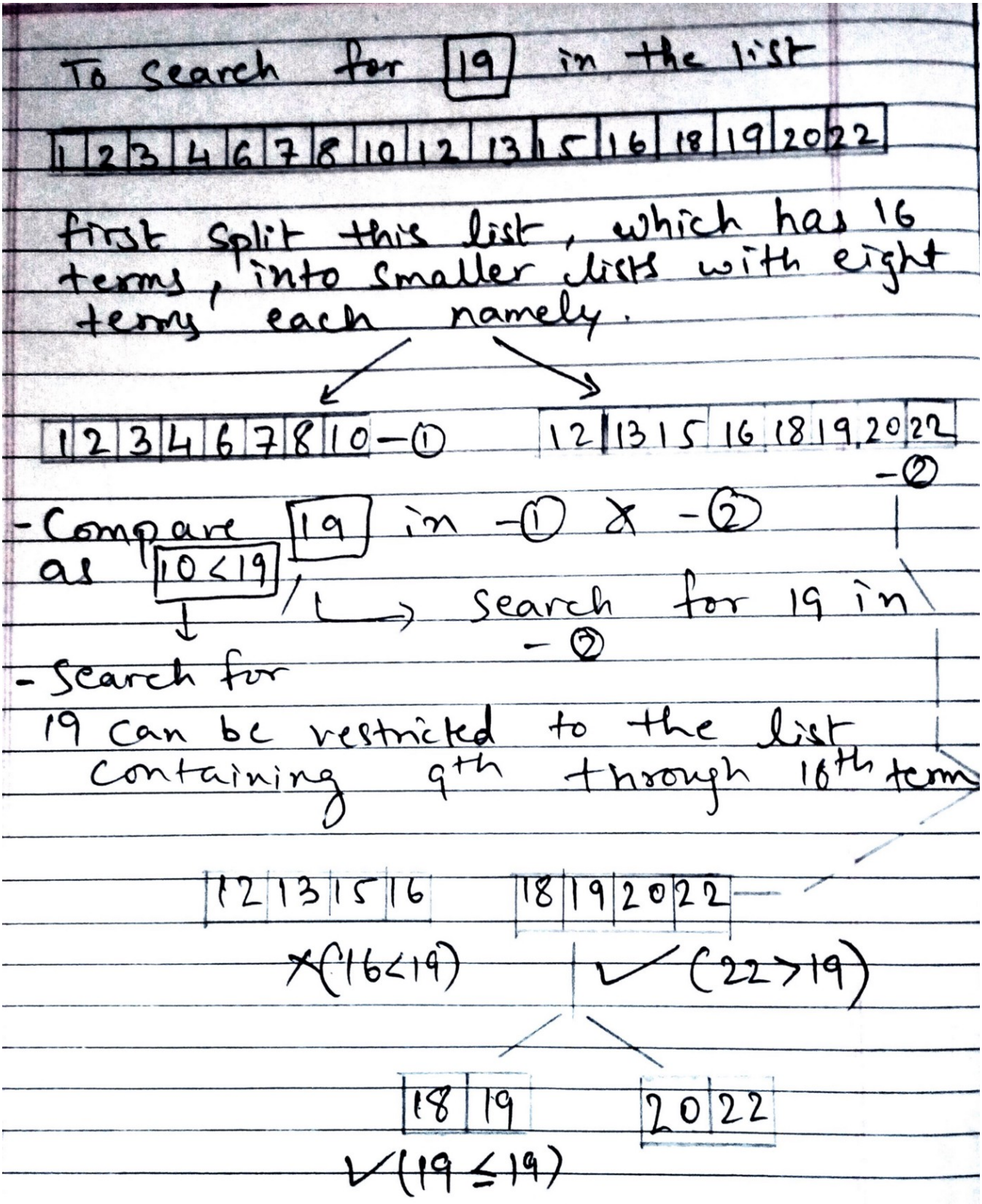
How does it work ?

In its simplest form, Binary Search operates on a contiguous sequence with a specified left and right index. This is called the Search Space. Binary Search maintains the left, right, and middle indices of the search space and compares the search target or applies the search condition to the middle value of the collection; if the condition is unsatisfied or values unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with an empty half, the condition cannot be fulfilled and target is not found.

Into Binary Search

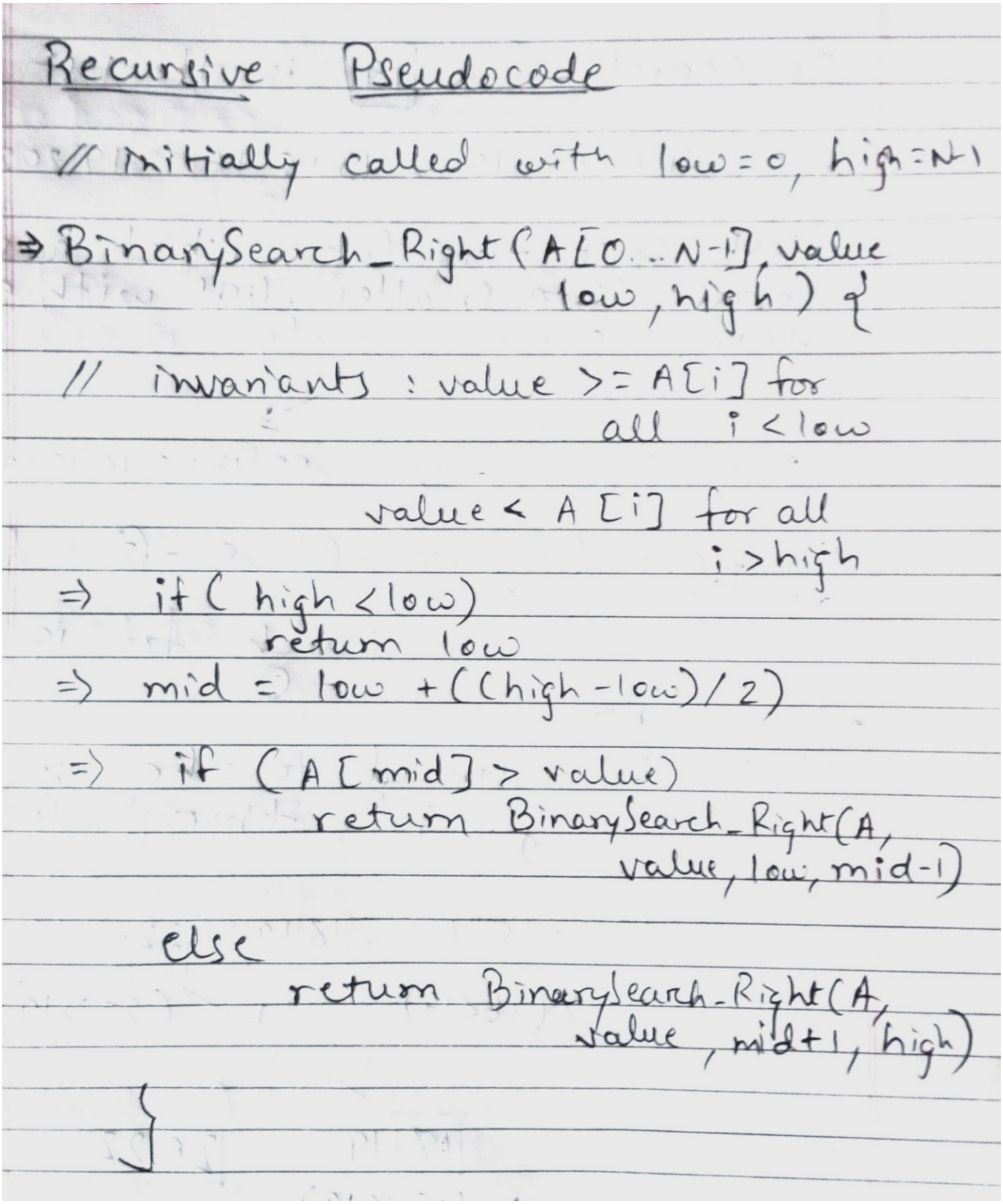
Given a sample array, first we find out midpoint and split it out. If **midpoint** is the search value, then it's game over. So **O(1)** time complexity is achieved.

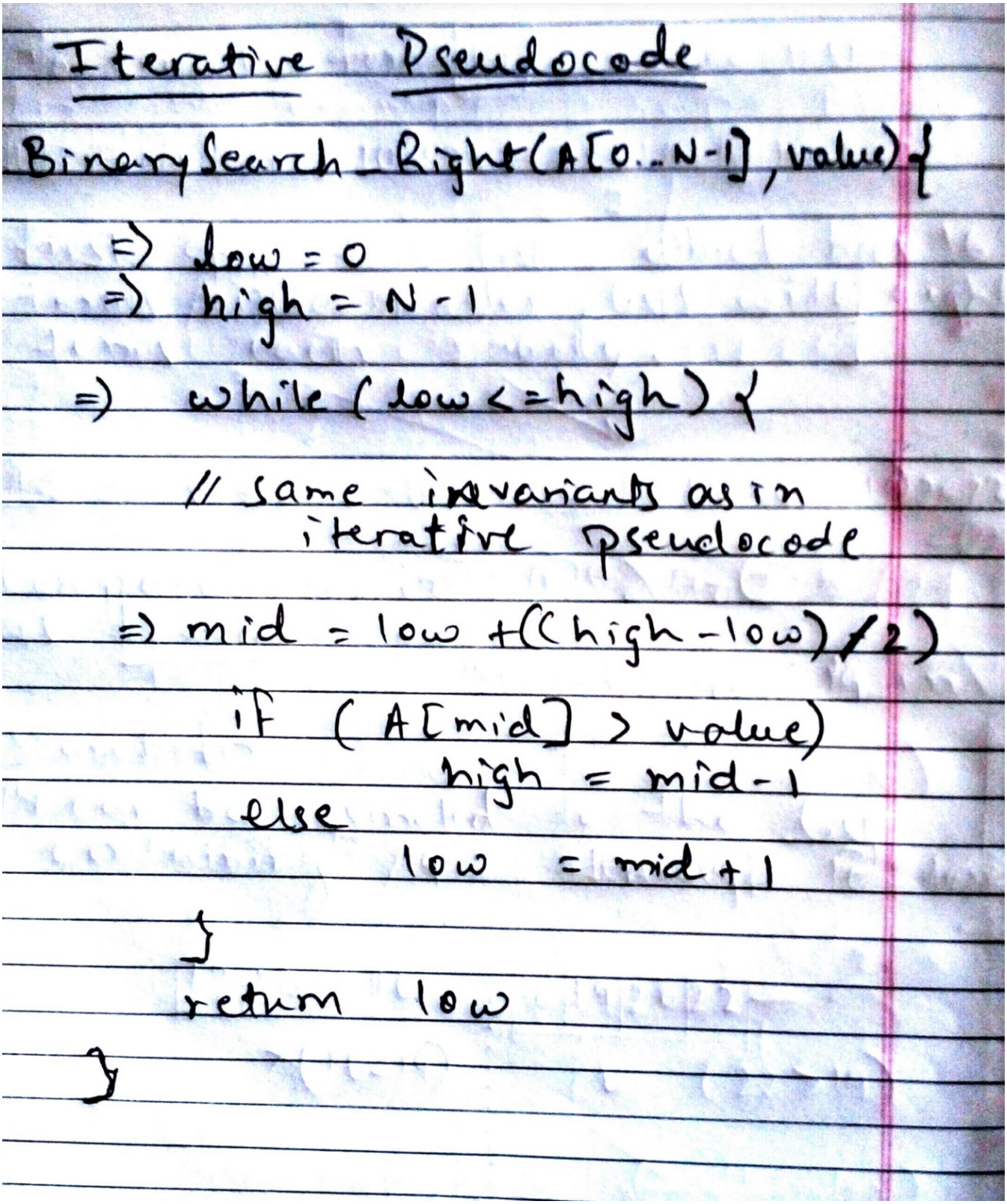
But if it's not the midpoint's value, then we have to go on an enchanted search for the value in divided halves. Because of this, now we can achieve time complexity in order of `log(n)` or `n` i.e. **O(logn) or O(n).**

You can see here in above example that `19` was found after so much of divisions of a single array(lists in python).

You can find difference between iteration and recursion as part of debates in reddit or stackoverflow .





Time complexity

The time complexity of Binary Search can be written as

```python
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it
        # can only be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present
        # in right subarray
        else:
            return binarySearch(arr, mid+1, r, x)

    else:
        # Element is not present in the array
        return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"
```


Ordering the elements of a list is a problem that occurs in many contexts. For example, to produce a telephone directory it is necessary to alphabetize the names of subscribers. Similarly, producing a directory of songs available for downloading requires that their titles be put in alphabetic order.

Putting addresses in order in an e-mail mailing list can determine whether there are duplicated addresses. Creating a useful dictionary requires that words be put in alphabetical order. Similarly, generating a parts list requires that we order them according to increasing part number.

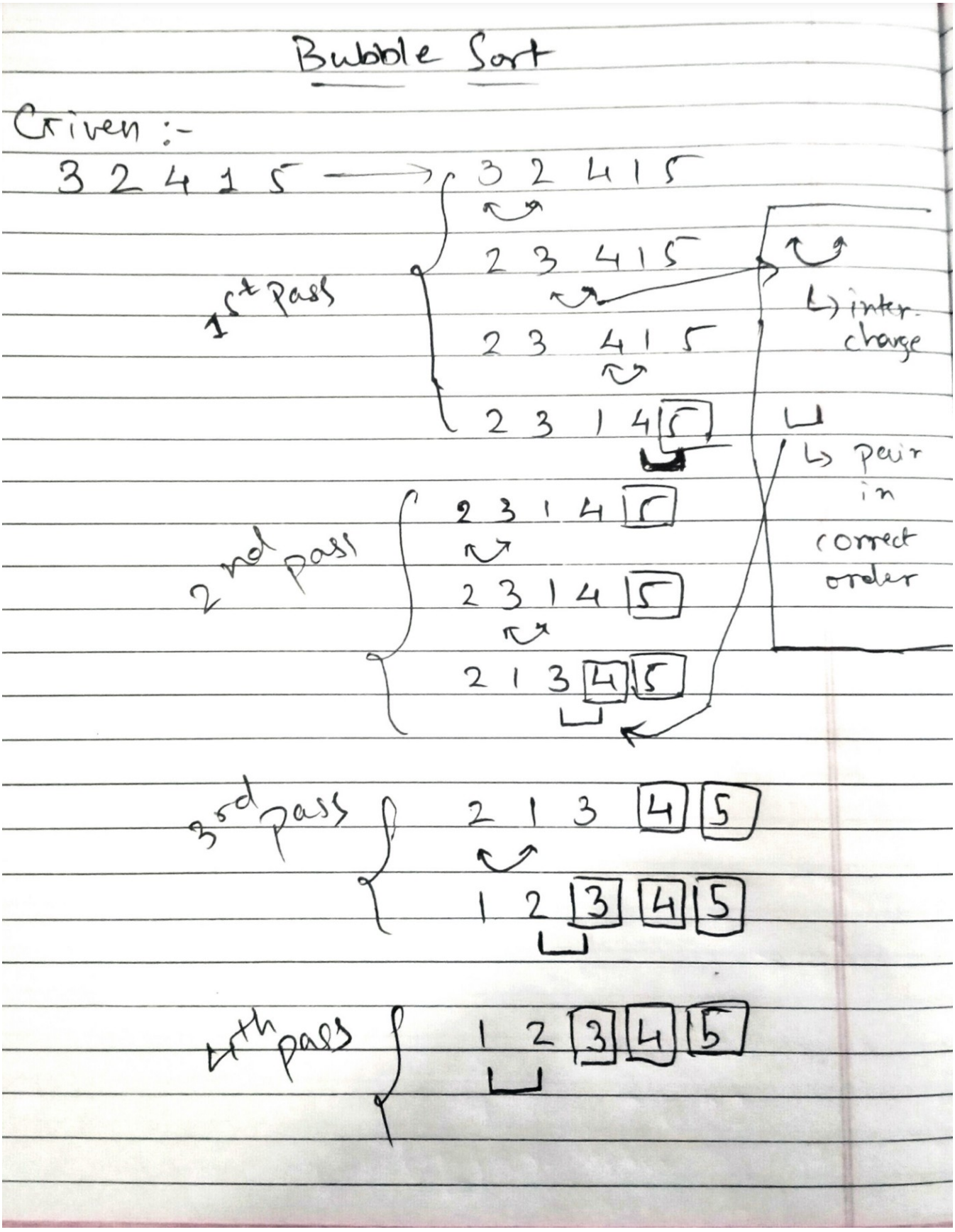
The *Art of Computer Programming*, **Donald Knuth** devotes close to 400 pages to sorting, covering around 15 different sorting algorithms in depth! More than 100 sorting algorithms have been devised, and it is surprising how often new sorting algorithms are developed.

Bubble Sort

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete.

It is one of the most inefficient sorting algorithms because of how simple it is. While asymptotically equivalent to the other algorithms, it will require $\mathbf{O(n^2)}$ swaps in the worst-case.



Pseudocode (Source — Wikipedia)

```
swapped = false
for j from 0 to N - 1
   if a[j] > a[j + 1]
      swap( a[j], a[j + 1] )
      swapped = true
```

Time Complexity

- **Worst and Average Case Time Complexity:** O(n*n). Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

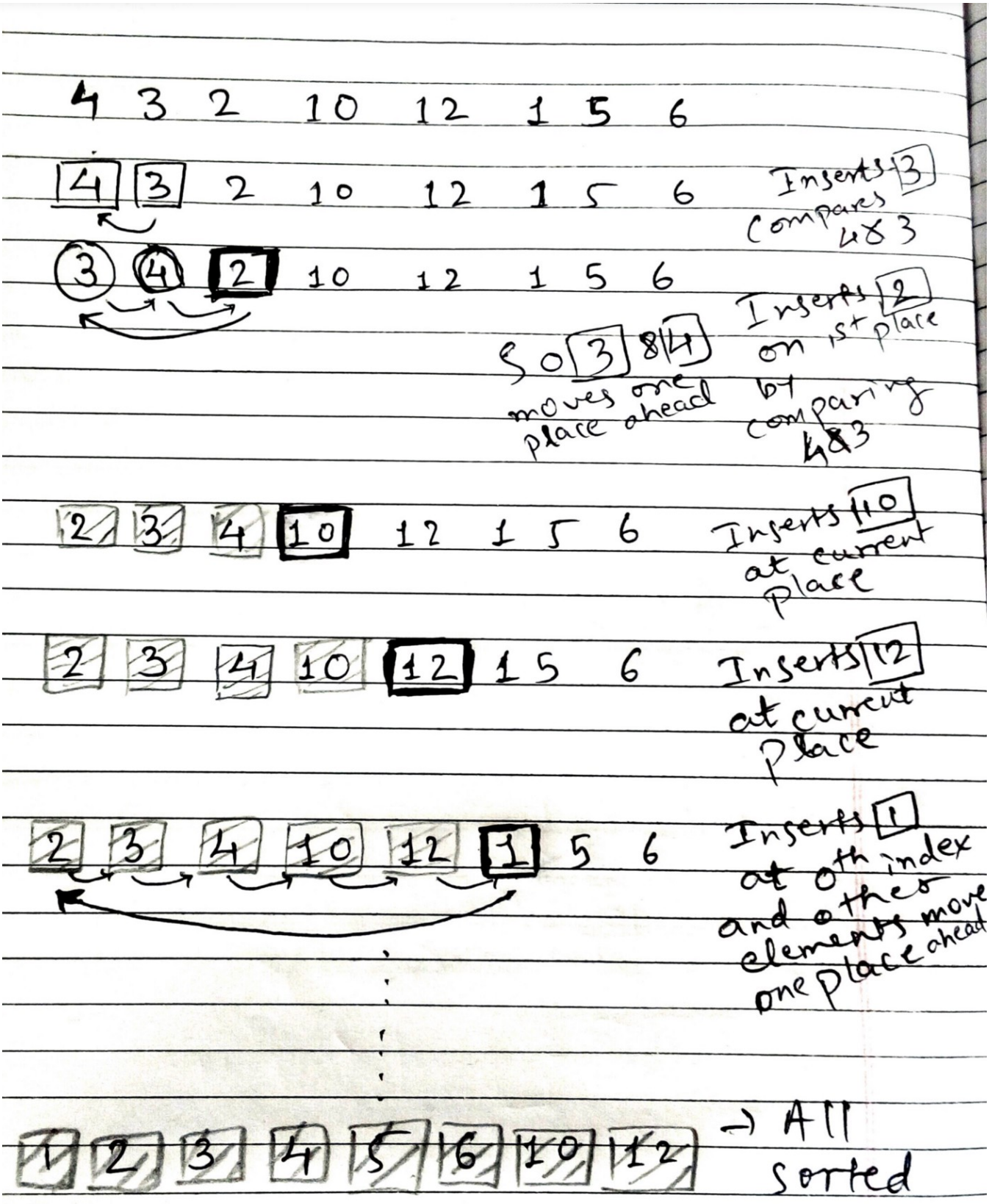
Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

How it works

To sort a list with `n` elements, the insertion sort begins with the second element. The insertion sort compares this second element with the first element and inserts it before the first element if it does not exceed the first element and after the first element if it exceeds the first element. At this point, the first two elements are in the correct order. The third element is then compared with the first element, and if it is larger than the first element, it is compared with the second element; it is inserted into the correct position among the first three elements.

Into Insertion Sort

Pseudocode

```
INSERTION-SORT(A)
1.   for j = 2 to n
2.       key ← A [j]
3.       // Insert A[j] into the sorted sequence A[1..j-1]
4.       j ← i – 1
5.       while i > 0 and A[i] > key
6.             A[i+1] ← A[i]
7.             i ← i – 1
8.       A[j+1] ← key
```

Time Complexity: O(n*n)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Implementation of Insertion Sort in python -

```python
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
```


```
20
21  # Driver code to test above
22  arr = [12, 11, 13, 5, 6]
23  insertionSort(arr)
24  print ("Sorted array is:")
25  for i in range(len(arr)):
26      print ("%d" %arr[i])
27
28  # This code is contributed by Mohit Kumra
```

Conclusion

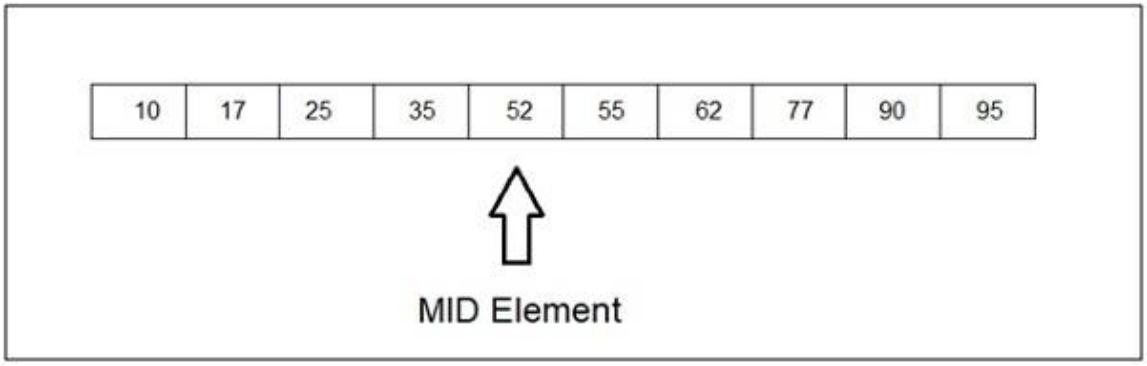
So, these were some basic algorithms that will provide a glimpse before diving deep into advanced and complex algorithms. As I titled it “I” part, I think I’ll be making part 2 covering more greedy and dynamic algorithms in future.

Also If I have been missing some topic to cover or maybe fault in my notes please put suggestions in comment.

Thank you for spending your time to read this article. Keep coding !

Binary search works on a sorted array. The value is compared with the middle element of the array. If equality is not found, then the half part is eliminated in which the value is not there. In the same way, the other half part is searched.

Here is the mid element in our array. Let's say we need to find 62, then the left part would be eliminated and the right part is then searched −



These are the complexities of a binary search −

Worst-case performance	O(log n)
Best-case performance	O(1)
Average performance	O(log n)
Worst-case space complexity	O(1)

Example

Let us see the method to implement the binary search −

```
public static object BinarySearchDisplay(int[] arr, int key) {
  int minNum = 0;
  int maxNum = arr.Length - 1;

  while (minNum <=maxNum) {
    int mid = (minNum + maxNum) / 2;
    if (key == arr[mid]) {
      return ++mid;
    } else if (key < arr[mid]) {
      max = mid - 1;
    }else {
      min = mid + 1;
    }
  }
  return "None";
}
```

Bubble sort is a simple sorting algorithm. This sorting algorithm is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

Let's say our int has 5 elements –

```
int[] arr = { 78, 55, 45, 98, 13 };
```

Now, let us perform Bubble Sort.

Start with the first two elements 78 and 55. 55 is smaller than 78, so swap both of them. Now the list is –

```
55, 78,45,98, 13
```

Now 45 is less than 78, so swap it.

```
55, 45, 78, 98, 3
```

Now 98 is greater than 78, so keep as it is.

3 is less than 98, so swap it. Now the list looks like –

```
55, 45, 78, 3, 98
```

This was the first iteration. After performing all the iterations, e will get our sorted array using Bubble Sort –

```
3, 45, 55, 78, 93
```

Example

Let us see an example with 10 elements in an array and sort it.

Live Demo

```
using System;
namespace BubbleSort {
  class MySort {
    static void Main(string[] args) {
      int[] arr = { 78, 55, 45, 98, 13 };
      int temp;
      for (int j = 0; j <= arr.Length - 2; j++) {
        for (int i = 0; i <= arr.Length - 2; i++) {
          if (arr[i] > arr[i + 1]) {
            temp= arr[i + 1];
            arr[i + 1] = arr[i];
            arr[i] = temp;
          }
```

```
        }
      }
      Console.WriteLine("Sorted:");
      foreach (int p in arr)
        Console.Write(p + " ");
      Console.Read();
    }
  }
}
```

Output

```
Sorted:
13 45 55 78 98
```

Insertion Sort is a sorting algorithm that takes an element at a time and inserts it in its correct position in the array. This process is continued until the array is sorted.

A program that demonstrates insertion sort in C# is given as follows.

Example

Live Demo

```
using System;
namespace InsertionSortDemo {
  class Example {
    static void Main(string[] args) {
      int[] arr = new int[10] { 23, 9, 85, 12, 99, 34, 60, 15, 100, 1 };
      int n = 10, i, j, val, flag;
      Console.WriteLine("Insertion Sort");
      Console.Write("Initial array is: ");
      for (i = 0; i < n; i++) {
        Console.Write(arr[i] + " ");
      }
      for (i = 1; i < n; i++) {
        val = arr[i];
        flag = 0;
        for (j = i - 1; j >= 0 && flag != 1; ) {
          if (val < arr[j]) {
            arr[j + 1] = arr[j];
            j--;
            arr[j + 1] = val;
          }
          else flag = 1;
        }
      }
      Console.Write("\nSorted Array is: ");
      for (i = 0; i < n; i++) {
        Console.Write(arr[i] + " ");
      }
    }
  }
}
```

Output

The output of the above program is as follows.

```
Insertion Sort
Initial array is: 23 9 85 12 99 34 60 15 100 1
Sorted Array is: 1 9 12 15 23 34 60 85 99 100
```

Now, let us understand the above program.

First the array is initialized and its value is printed using a for loop. This can be seen in the following code snippet –

```
int[] arr = new int[10] { 23, 9, 85, 12, 99, 34, 60, 15, 100, 1 };
int n = 10, i, j, val, flag;
Console.WriteLine("Insertion Sort");
Console.Write("Initial array is: ");
for (i = 0; i < n; i++) {
  Console.Write(arr[i] + " ");
}
```

A nested for loop is used for the actual sorting process. In each pass of the outer for loop, the current element is inserted into its correct position in the array. This process continues until the array is sorted. This can be seen in the following code snippet.

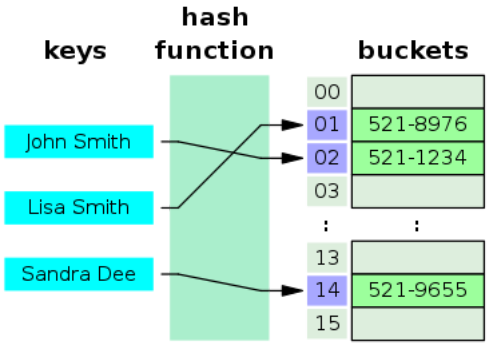
```
for (i = 1; i < n; i++) {
  val = arr[i];
  flag = 0;
  for (j = i - 1; j >= 0 && flag != 1; ) {
    if (val < arr[j]) {
      arr[j + 1] = arr[j];
      j--;
      arr[j + 1] = val;
    } else flag = 1;
  }
}
```

Finally, the sorted array is displayed. This can be seen in the following code snippet.

```
Console.Write("\nSorted Array is: ");
for (i = 0; i < n; i++) {
  Console.Write(arr[i] + " ");
}
```

Search algorithm

In computer science, a **search algorithm** is an algorithm (if more than one, algorithms [1]) designed to solves a search problem. Search algorithms work to retrieve information stored within particular data structure, or calculated in the search space of a problem domain, with either discrete or continuous values.



Visual representation of a hash table, a data structure that allows for fast retrieval of information.

While the search problems described above and web search are both problems in information retrieval, they are generally studied as separate subfields and are solved and evaluated differently. Web search problems are generally focused on filtering and finding documents highly relevant to human queries. Classic search algorithms are evaluated on how fast they can find a solution, and whether the solution found is optimal. Though information retrieval algorithms must be fast, the quality of ranking, and whether good results have been left out and bad results included, is more important.

The appropriate search algorithm often depends on the data structure being searched, and may also include prior knowledge about the data. Search algorithms can be made faster or more efficient by specially constructed database structures, such as search trees, hash maps, and database indexes. [2][3]

Search algorithms can be classified based on their mechanism of searching into three types of algorithms: linear, binary, and hashing. Linear search algorithms check every record for the one associated with a target key in a linear fashion.[4] Binary, or half-interval, searches repeatedly target the center of the search structure and divide the search space in half. Comparison search algorithms improve on linear searching by successively eliminating records based on comparisons of the keys until the target record is found, and can be applied on data structures with a defined order.[5] Digital search algorithms work based on the properties of digits in data structures by using numerical keys.[6] Finally, hashing directly maps keys to records based on a hash function.[7]

Algorithms are often evaluated by their computational complexity, or maximum theoretical run time. Binary search functions, for example, have a maximum complexity of $O(\log n)$, or logarithmic time. In simple terms, the maximum number of operations needed to find the search target is a logarithmic function of the size of the search space.

Contents

Applications of search algorithms

Specific applications of search algorithms include:

- Problems in combinatorial optimization, such as:
  - The vehicle routing problem, a form of shortest path problem
  - The knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
  - The nurse scheduling problem
- Problems in constraint satisfaction, such as:
  - The map coloring problem
  - Filling in a sudoku or crossword puzzle
- In game theory and especially combinatorial game theory, choosing the best move to make next (such as with the minmax algorithm)
- Finding a combination or password from the whole set of possibilities
- Factoring an integer (an important problem in cryptography)
- Optimizing an industrial process, such as a chemical reaction, by changing the parameters of the process (like temperature, pressure, and pH)
- Retrieving a record from a database
- Finding the maximum or minimum value in a list or array
- Checking to see if a given value is present in a set of values

Classes

For virtual search spaces

Algorithms for searching virtual spaces are used in the constraint satisfaction problem, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations / equalities. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables. Algorithms for these problems include the basic brute-force search (also called "naïve" or "uninformed" search), and a variety of heuristics that try to exploit partial knowledge about the structure of this space, such as linear relaxation, constraint generation, and constraint propagation.

An important subclass are the local search methods, that view the elements of the search space as the vertices of a graph, with edges defined by a set of heuristics applicable to the case; and scan the space by moving from item to item along the edges, for example according to the steepest descent

or best-first criterion, or in a stochastic search. This category includes a great variety of general metaheuristic methods, such as simulated annealing, tabu search, A-teams, and genetic programming, that combine arbitrary heuristics in specific ways. The opposite of local search would be global search methods. This method is applicable when the search space is not limited and all aspects of the given network are available to the entity running the search algorithm.[8]

This class also includes various tree search algorithms, that view the elements as vertices of a tree, and traverse that tree in some special order. Examples of the latter include the exhaustive methods such as depth-first search and breadth-first search, as well as various heuristic-based search tree pruning methods such as backtracking and branch and bound. Unlike general metaheuristics, which at best work only in a probabilistic sense, many of these tree-search methods are guaranteed to find the exact or optimal solution, if given enough time. This is called "completeness".

Another important sub-class consists of algorithms for exploring the game tree of multiple-player games, such as chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, such as in robot guidance or in marketing, financial, or military strategy planning. This kind of problem — combinatorial search — has been extensively studied in the context of artificial intelligence. Examples of algorithms for this class are the minimax algorithm, alpha–beta pruning, and the A* algorithm and its variants.

For sub-structures of a given structure

The name "combinatorial search" is generally used for algorithms that look for a specific sub-structure of a given discrete structure, such as a graph, a string, a finite group, and so on. The term combinatorial optimization is typically used when the goal is to find a sub-structure with a maximum (or minimum) value of some parameter. (Since the sub-structure is usually represented in the computer by a set of integer variables with constraints, these problems can be viewed as special cases of constraint satisfaction or discrete optimization; but they are usually formulated and solved in a more abstract setting where the internal representation is not explicitly mentioned.)

An important and extensively studied subclass are the graph algorithms, in particular graph traversal algorithms, for finding specific sub-structures in a given graph — such as subgraphs, paths, circuits, and so on. Examples include Dijkstra's algorithm, Kruskal's algorithm, the nearest neighbour algorithm, and Prim's algorithm.

Another important subclass of this category are the string searching algorithms, that search for patterns within strings. Two famous examples are the Boyer–Moore and Knuth–Morris–Pratt algorithms, and several algorithms based on the suffix tree data structure.

Search for the maximum of a function

In 1953, American statistician Jack Kiefer devised Fibonacci search which can be used to find the maximum of a unimodal function and has many other applications in computer science.

For quantum computers