**DATA STRUCTURES WITH PYTHON**

**UNIT II**

**Algorithm Analysis:** Asymptotic Analysis and Big O notation **Recursion:** What is recursion, examples [Factorial functions, Fibonacci series]. **Array-Based Sequences:** Python Sequence types, low-level arrays, dynamic arrays, the efficiency of python’s sequences, using array-based sequences. **Searching:** Sequential Search, binary search and algorithmic analysis. **Sorting:** Insertion sort, selection sort, bubble sort.

Learning Outcomes:

After completion of this unit, the student will be able to

● summarize various ways of representing data (L2)

● explain the working of linear and binary search algorithms (L2)

● compare data representations and sorting algorithms (L6)

**What is an Algorithm?**

An algorithm is a set of well-defined instructions in sequence to solve a problem.

**Qualities of a good algorithm**

* Input and output should be defined precisely.
* Each step in the algorithm should be clear and unambiguous.
* Algorithms should be most effective among many different ways to solve a problem.
* An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

**Examples Of Algorithms In Programming**

**Write an algorithm to add two numbers entered by the user.**

Step 1: Start

Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

sum←num1+num2

Step 5: Display sum

Step 6: Stop

**Write an algorithm to find the largest among three different numbers entered by the user.**

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a > b

If a > c

Display a is the largest number.

Else

Display c is the largest number.

Else

If b > c

Display b is the largest number.

Else

Display c is the greatest number.

Step 5: Stop

**Write an algorithm to find all roots of a quadratic equation ax2+bx+c=0.**

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D ← b2-4ac

Step 4: If D ≥ 0

r1 ← (-b+√D)/2a

r2 ← (-b-√D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp ← -b/2a

ip ← √(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

For example, an algorithm to solve the problem of factorials might look something like this:

Problem: Find the factorial of n

Initialize fact = 1

For every value v in range 1 to n:

Multiply the fact by v

fact contains the factorial of n

Here, the algorithm is written in English. If it was written in a programming language, we would call it to code instead. Here is a code for finding the factorial of a number in C++.

**int factorial(int n) {**

**int fact = 1;**

**for (int v = 1; v <= n; v++) {**

**fact = fact \* v;**

**}**

**return fact;**

**}**

Programming is all about data structures and algorithms. Data structures are used to hold data while algorithms are used to solve the problem using that data.

Data structures and algorithms (DSA) goes through solutions to standard problems in detail and gives you an insight into how efficient it is to use each one of them. It also teaches you the science of evaluating the efficiency of an algorithm. This enables you to choose the best of various choices.

**Use of Data Structures and Algorithms to Make Your Code Scalable**

**Time is precious.**

Suppose, Alice and Bob are trying to solve a simple problem of finding the sum of the first 1011 natural numbers. While Bob was writing the algorithm, Alice implemented it, proving that it is as simple as criticising Donald Trump.

**Algorithm (by Bob)**

Initialize sum = 0

for every natural number n in range 1 to 1011 (inclusive):

add n to sum

sum is your answer

**Code (by Alice)**

int findSum() {

int sum = 0;

for (int v = 1; v <= 100000000000; v++) {

sum += v;

}

return sum;

}

Alice and Bob are feeling euphoric of themselves that they could build something of their own in almost no time. Let's sneak into their workspace and listen to their conversation.

Alice: Let's run this code and find out the sum.

Bob: I ran this code a few minutes back, but it's still not showing the output. What's wrong with it?

Oops, something went wrong! A computer is the most deterministic machine. Going back and trying to run it again won't help. So let's analyse what's wrong with this simple code.

Two of the most valuable resources for a computer program are time and memory.

**The time taken by the computer to run code is:**

Time to run code = number of instructions \* time to execute each instruction.

The number of instructions depends on the code you used, and the time taken to execute each code depends on your machine and compiler.

In this case, the total number of instructions executed (let's say x) are x = 1 + (1011 + 1) + (1011) + 1, which is x = 2 \* 1011 + 3

Let us assume that a computer can execute y = 108 instructions in one second (it can vary subject to machine configuration). The time taken to run above code is

Time taken to run code = x/y (greater than 16 minutes)

Is it possible to optimise the algorithm so that Alice and Bob do not have to wait for 16 minutes every time they run this code?

I am sure that you already guessed the right method. The sum of first N natural numbers is given by the formula:

Sum = N \* (N + 1) / 2

Converting it into code will look something like this:

**int sum(int N) {**

**return N \* (N + 1) / 2;**

**}**

This code executes in just one instruction and gets the task done no matter what the value is. Let it be greater than the total number of atoms in the universe. It will find the result in no time.

The time taken to solve the problem, in this case, is 1/y (which is ten nanoseconds). By the way, the fusion reaction of a hydrogen bomb takes 40-50 ns, which means your program will complete successfully even if someone throws a hydrogen bomb on your computer at the same time you ran your code. :)

**Asymptotic Analysis**

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

**Asymptotic Notations**

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear, i.e. the best case.

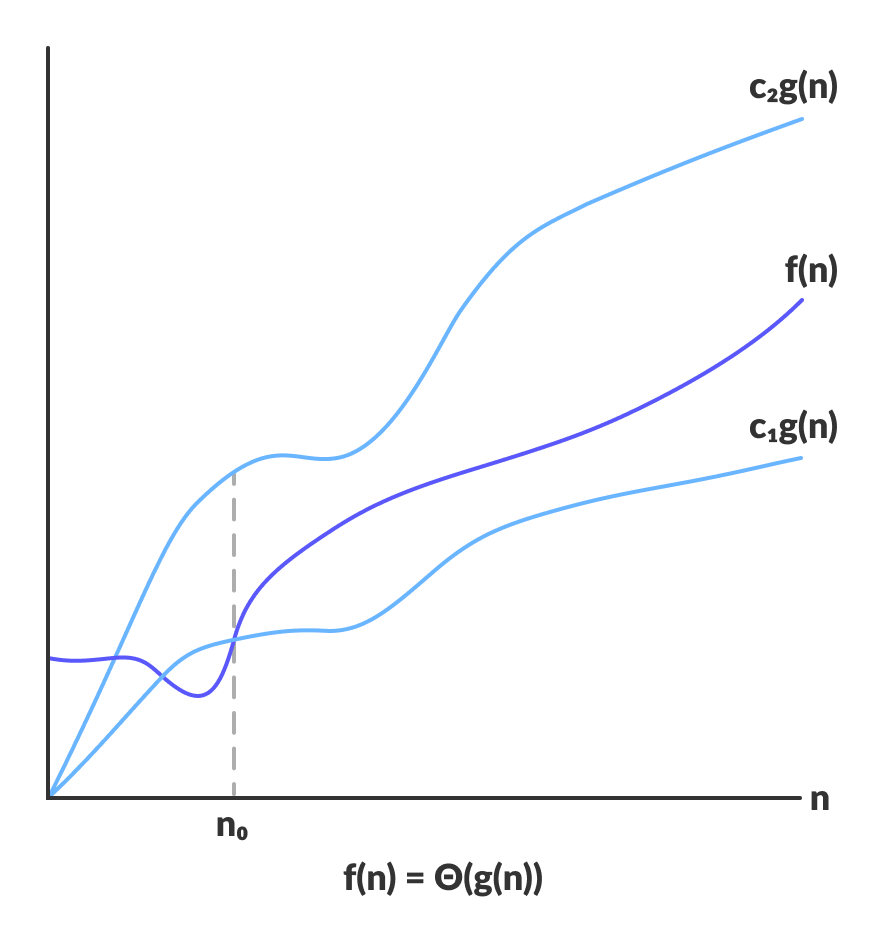
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements, i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

**Theta Notation (Θ-notation)**

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the average-case complexity of an algorithm.



For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

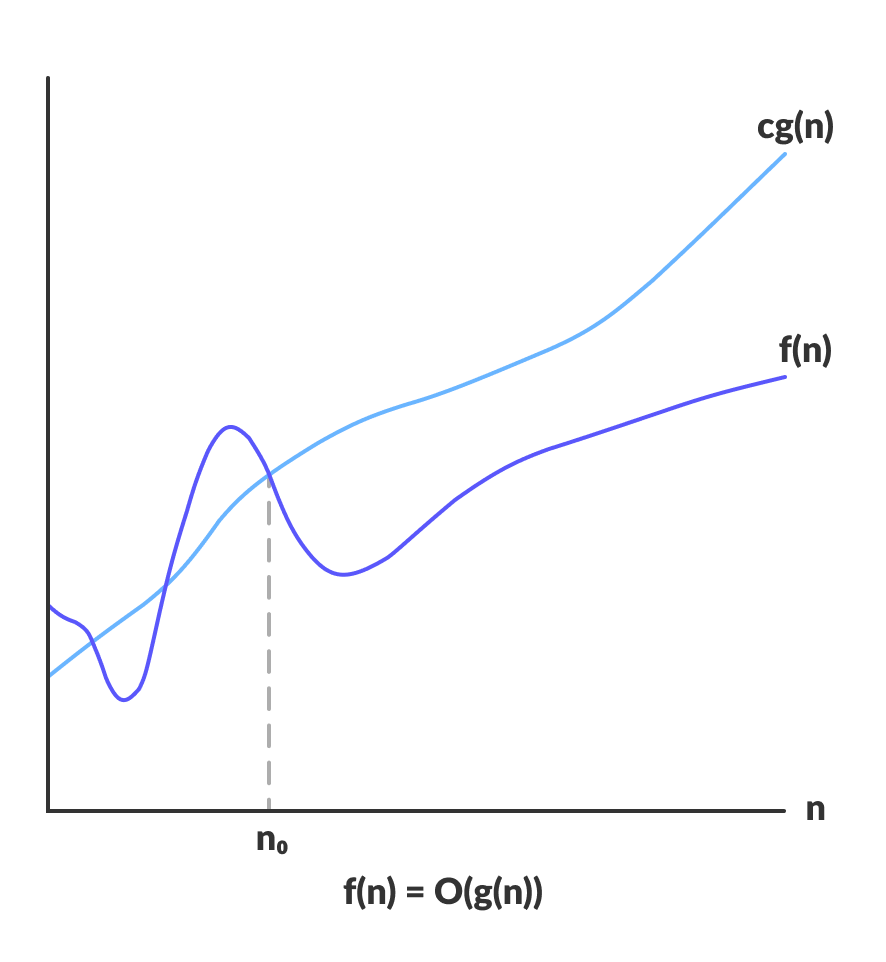
The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched

between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2 > g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

**Big-O Notation (O-notation)**

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

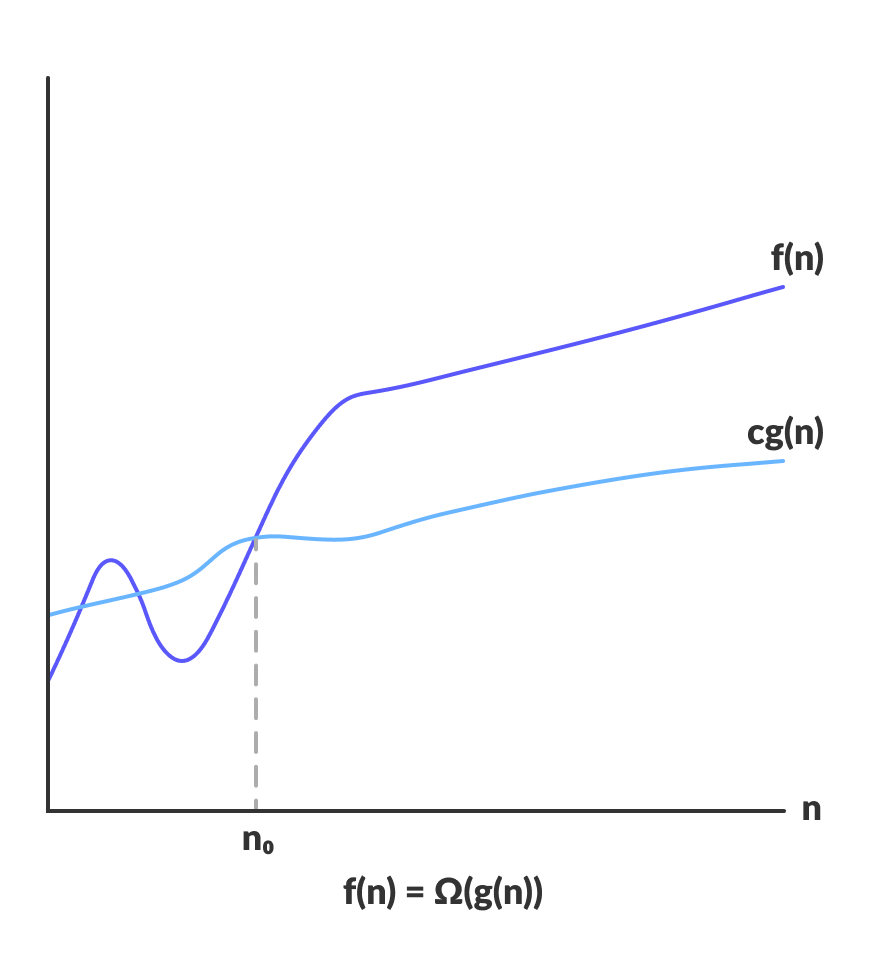
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst case scenario.

**Omega Notation (Ω-notation)**

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Ω(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n))

**Recursion**

**What is recursion?**

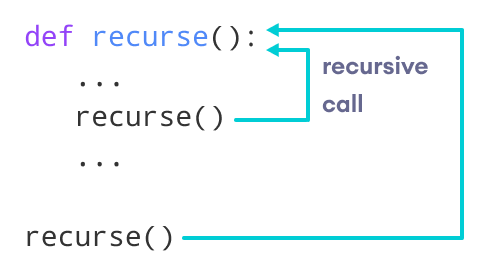
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

**Recursive Function**

In Python, we know that a [function](https://www.programiz.com/python-programming/function) can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called recurse.



Recursive Function in Python

Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1\*2\*3\*4\*5\*6 = 720.

Example of a recursive function

**def factorial(x):**

**"""This is a recursive function**

**to find the factorial of an integer"""**

**if x == 1:**

**return 1**

**else:**

**return (x \* factorial(x-1))**

**num = 3**

**print("The factorial of", num, "is", factorial(num))**

Output

The factorial of 3 is 6

In the above example, factorial() is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one.

This recursive call can be explained in the following steps.

**factorial(3) # 1st call with 3**

**3 \* factorial(2) # 2nd call with 2**

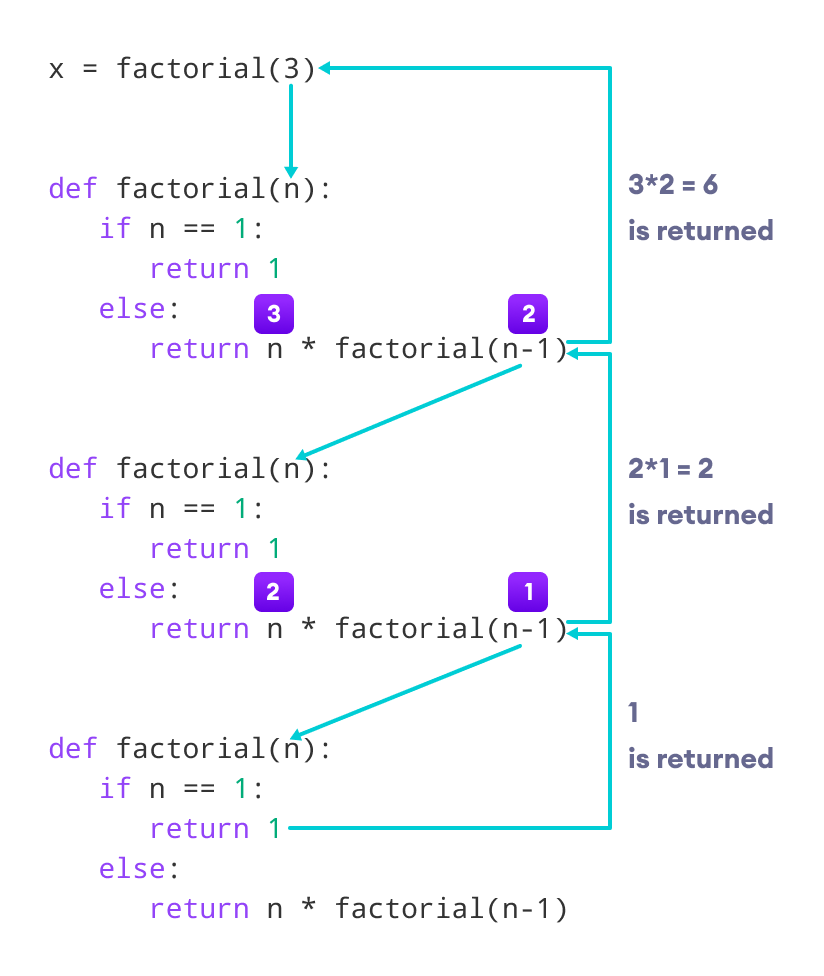
**3 \* 2 \* factorial(1) # 3rd call with 1**

**3 \* 2 \* 1 # return from 3rd call as number=1**

**3 \* 2 # return from 2nd call**

**6 # return from 1st call**

Let's look at an image that shows a step-by-step process of what is going on:



Working of a recursive factorial function

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in a RecursionError.

Let's look at one such condition.

**def recursor():**

**recursor()**

**recursor()**

Output

Traceback (most recent call last):

File "<string>", line 3, in <module>

File "<string>", line 2, in a

File "<string>", line 2, in a

File "<string>", line 2, in a

[Previous line repeated 996 more times]

RecursionError: maximum recursion depth exceeded

**Advantages of Recursion**

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* Sequence generation is easier with recursion than using some nested iteration.

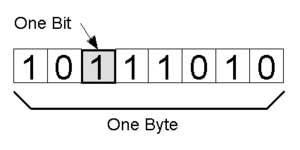
**Disadvantages of Recursion**

* Sometimes the logic behind recursion is hard to follow through.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
* Recursive functions are hard to debug.

# **Array Based Sequences**

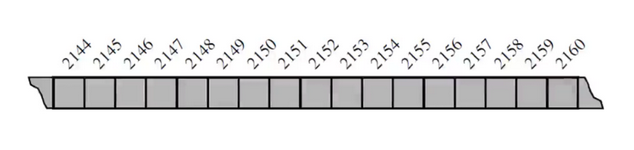
An array is a fundamental data structure available in most programming languages and it has a wide range of uses across different algorithms. There are more low-level concept and they were used in python to implement built-in data structure like lists, tuples and strings.

To know this data structure in details first we need to understand array in low level. The primary memory of a computer is composed of bits of information and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a byte, which is equivalent to 8 bits.



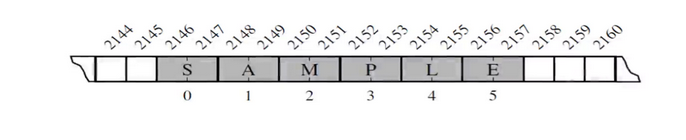
8 Bits = 1 Byte

A computer system will have a huge number of bytes and memory and to keep track of what information is stored and what byte the computer uses in abstraction known as a memory address and in effect each byte of memory is associated with a unique number that serves as its address.



This is usually how we represent a low-level computer memory. It’s individual bytes with consecutive addresses. So, despite the sequential nature of the numbering system loops the computer hardware is designed in theory so that any byte of the main memory can be efficiently accessed based upon its memory address in this sense we can say that a computer’s main memory performs as a random-access memory. So, each individual byte of memory can be stored or retrieved in order one-time O(1).

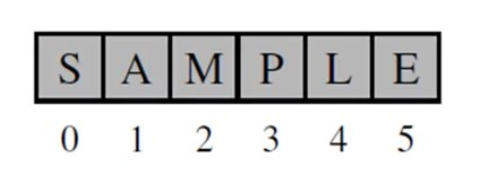
In general a programming language keeps track of the association between an identifier and the memory address in which the associated value is stored. A text string “SAMPLE” stored as an ordered sequence of individual characters and something to note here is that Python internally represents each Unicode character with 16 bits or two bytes for each character. If we actually diagram it out it looks like below.



Each cell of an array uses the same number of bytes. Allow any cell to be accessed in constant time. Appropriate memory address location can calculate by

start+(cell size)\*(index)

In higher level of abstraction, we simple think of array as sequence of same type of data type stored in sequence order as below

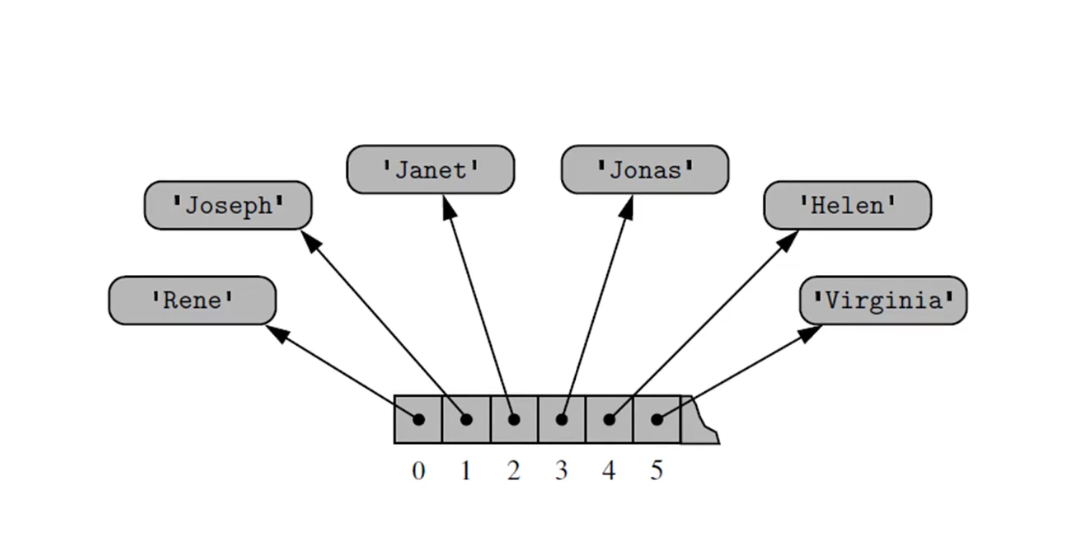


Besides array has it own disadvantages as below

* The number of elements to be stored in an array should be known in advance.
* An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased.
* Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
* Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

To overcome this python uses referential arrays for example to store all the employees name in the organization with they IDs each employee naturally have different length so Python represents a list or tuple instance using an internal storage mechanism of an array of object references that is Referential array.

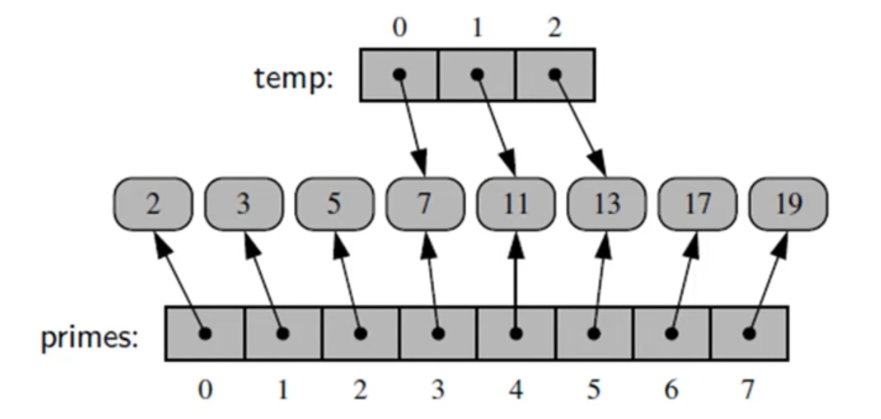
To store all the employees name in an array what we will do is we will initialize an array but instead of the values or char in it we will reference object of the employees names in each cell as show in below figure.



A single instance may have multiple reference to the same object as elements in the list and single object can be the element of two or more lists pretty confusing right lets look at below image

primes = [2, 3, 5, 7, 11, 13, 17, 19]

temp = primes[3:6] # deep copy



temp = primes[3:6]

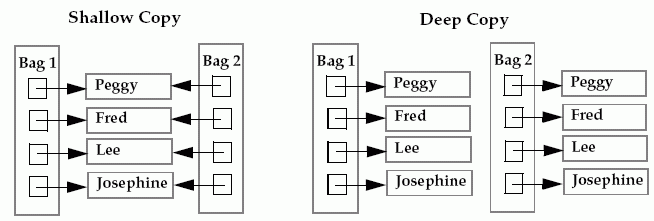
Now lets keep aside deep copy and shallow copy and focus on following things 1) A single instance may have multiple reference to the same object as elements in the list.

2) Single object can be the element of two or more.

consider element 7 in primes lists (i.e primes[3]) element 7 has both reference from primes[3] and from temp[0].

Now lets talk about deep copy and shallow copy.

On the abstraction level shallow copy means both the lists will have reference to the same values where as deep copy mean both lists have they own separate values and different memory allocation as show in below.

****

# **Shallow Copy of List**

Bag1 = [“peggy”, “Fred”, “Lee”, “Josephine” ]

Bag2 = Bag1 # it will a shallow copy as show in above image

# 

# **Deep Copy of List**

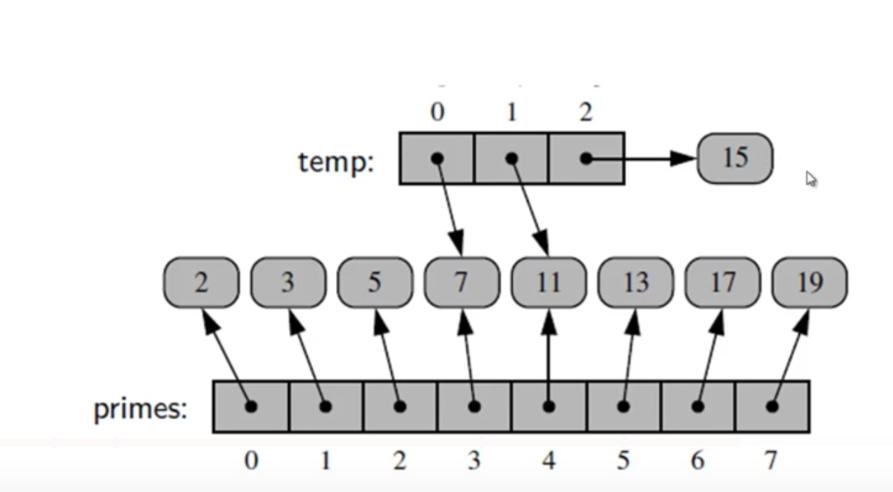
Bag1 = [“peggy”, “Fred”, “Lee”, “Josephine” ]

Bag2 = Bag2[:] # By slicing it will create a deep copy

This is OK but what about temp = primes[3:6] if it deep copy then why does it make one new reference as above Bag2 in deep copy as we are slicing it?

In deep copy python internal implementation new list has reference to the same elements that are in original list but when we do reassignment of the deep copy list considering the elements are mutable if we change

temp[2] = 15

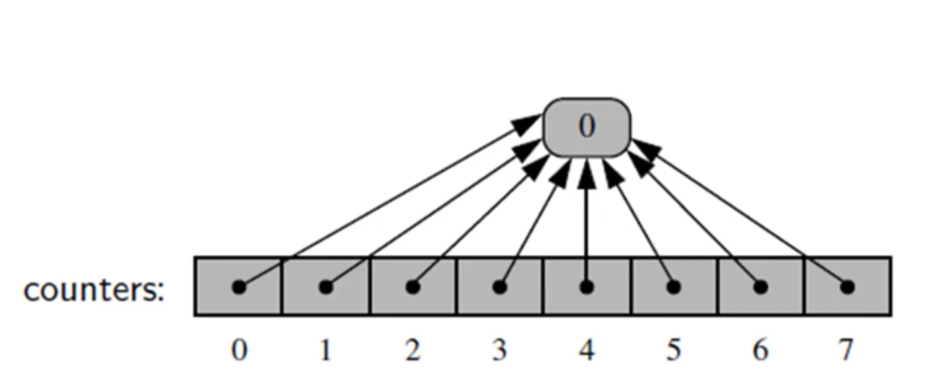


It will create a new reference object 15 it will not change the old reference object as done in shallow copy

Lets see some more examples

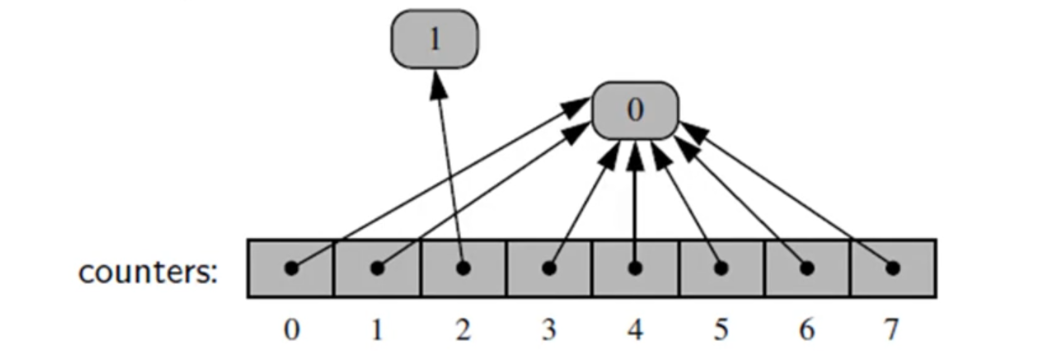
counters = [0]\*8

In this above list counters all eights cell reference the same object



If we change any value in the counters element it wont change the element that it is refers to instead it will create a new elements and change its reference to the new element as shown below

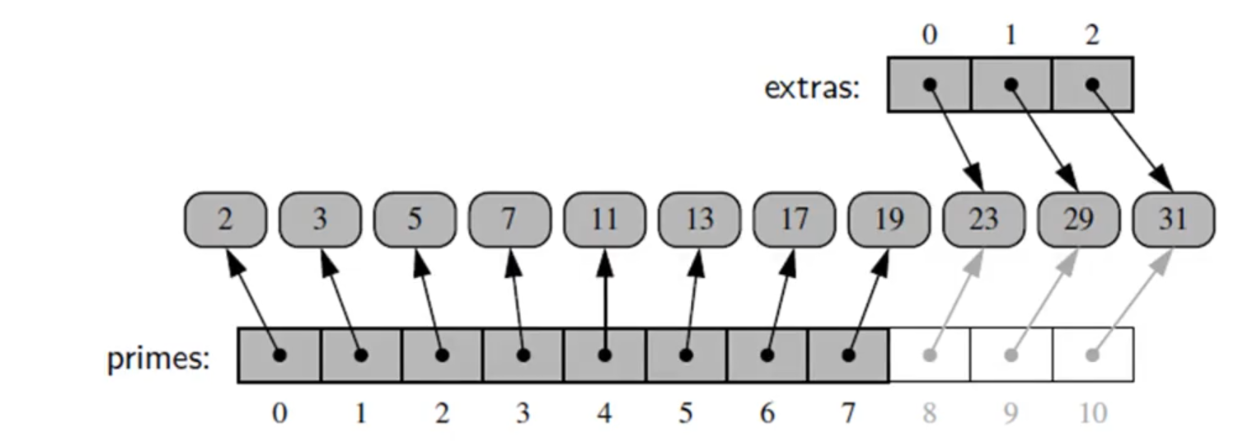
counters[2]+=1



If you are performing extend operation on a list in abstract level it show like the elements are getting appended to the old list its just referencing the new list as below

extras = [23, 29, 31]

primes.extend(extras)

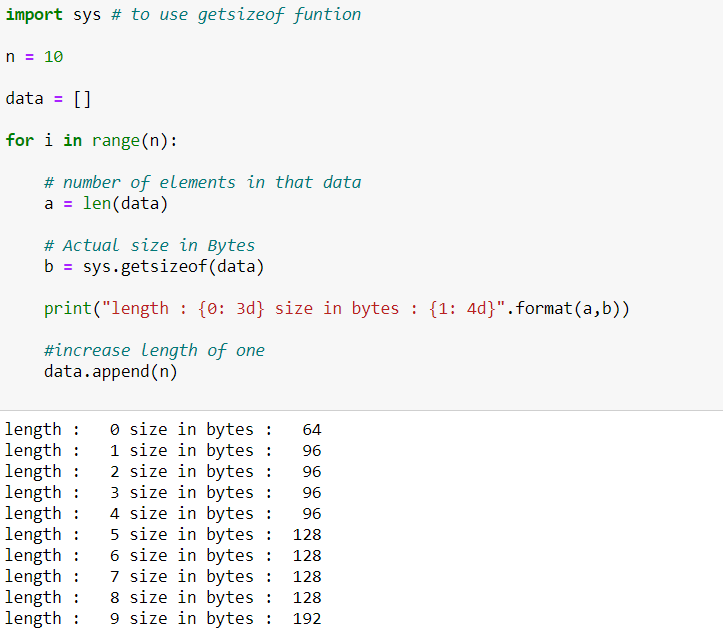


# 

# **Dynamic Arrays :**

when we are working with python lists we do not need to specific the size of lists as in array we can keep constantly adding to it. How lists does that?

A list instance often has greater capacity than current length. If elements keep getting appended, eventually this extra space runs out. below is demonstrating who dynamic array in lists works by creating our own dynamic array.



So you notice we begin a full length of 0 and we have size in bytes is 64 bytes are being used. Now here’s where the interesting part happens when we increase length to 1 the size in bytes gets increased to 96.

But you notice as we keep increasing length all the way up to 4 the size in bytes doesn’t actually change So Python has done is it’s actually set a number of bytes larger than what it needs to hold the current elements in the list. While it’s appending new elements to the array. So to get a better idea of this we’re going to actually put in a much larger end.

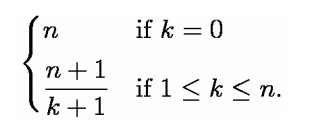
**Linear Search**

In computer science, linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

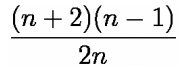
Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

**Analysis**

For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed. If the value being sought occurs k times in the list, and all orderings of the list are equally likely, the expected number of comparisons is



For example, if the value being sought occurs once in the list, and all orderings of the list are equally likely, the expected number of comparisons is (n+1)/2 . However, if it is known that it occurs once, then at most n - 1 comparisons are needed, and the expected number of comparisons is



(for example, for n = 2 this is 1, corresponding to a single if-then-else construct). Either way, asymptotically the worst-case cost and the expected cost of linear search are both O(n).

**Pseudocode**

**Forward iteration**:

This pseudocode describes a typical variant of linear search, where the result of the search is supposed to be either the location of the list item where the desired value was found; or an invalid location Λ, to indicate that the desired element does not occur in the list.

for each item in the list:

if that item has the desired value,

stop the search and return the item's location.

return Λ.

In this pseudocode, the last line is executed only after all list items have been examined with none matching.

If the list is stored as an array data structure, the location may be the index of the item found (usually between 1 and n, or 0 and n−1). In that case the invalid location Λ can be any index before the first element (such as 0 or −1, respectively) or after the last one (n+1 or n, respectively).

If the list is a simply linked list, then the item's location is its reference, and Λ is usually the null pointer. R

**Recursive version**

Linear search can also be described as a recursive algorithm:

LinearSearch(value, list)

if the list is empty, return Λ;

else

if the first item of the list has the desired value,

return its location;

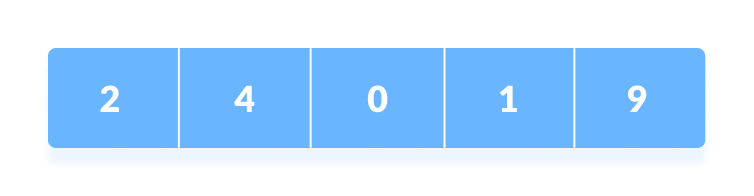
else:

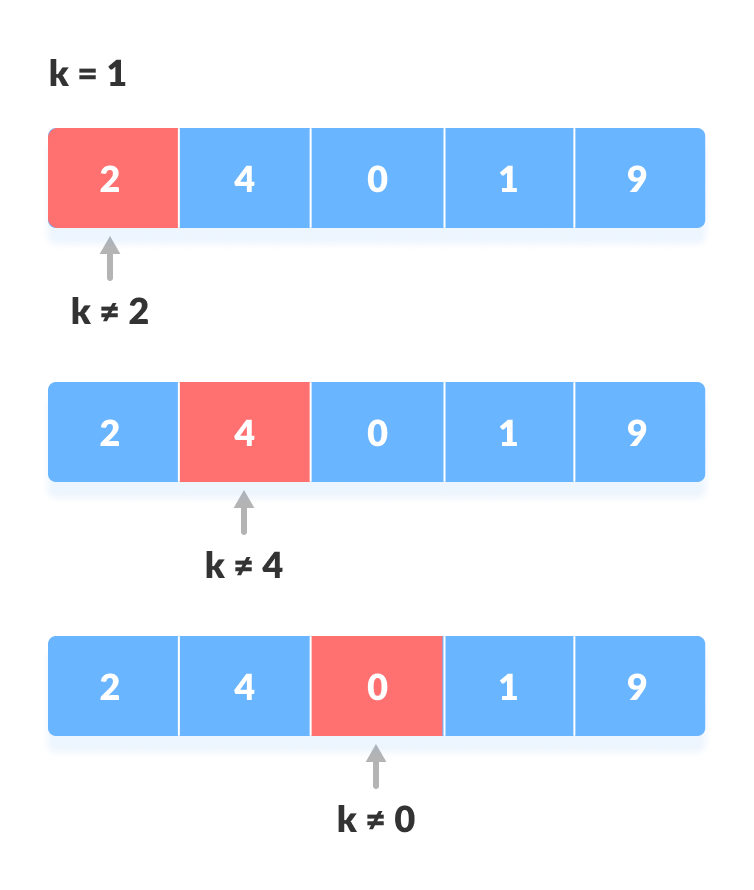
return LinearSearch(value, remainder of the list)

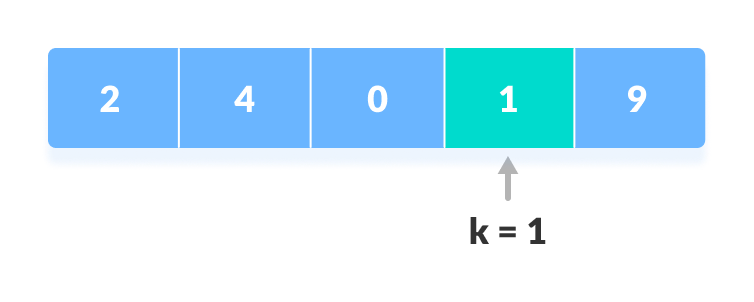
Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

**How Linear Search Works?**

The following steps are followed to search for an element k = 1 in the list below.

Array to be searched for

1. Start from the first element, compare k with each element x.Compare with each element

2. If x == k, return the index.Element found

3. Else, return not found.

**Linear Search Algorithm**

LinearSearch(array, key)

for each item in the array

if item == value

return its index

**# Linear Search in Python**

**def linearSearch(array, n, x):**

**# Going through array sequencially**

**for i in range(0, n):**

**if (array[i] == x):**

**return i**

**return -1**

**array = [2, 4, 0, 1, 9]**

**x = 1**

**n = len(array)**

**result = linearSearch(array, n, x)**

**if(result == -1):**

**print("Element not found")**

**else:**

**print("Element found at index: ", result)**

**Linear Search Complexities**

Time Complexity: O(n)

Space Complexity: O(1)

**Linear Search Applications**

For searching operations in smaller arrays (<100 items).

**Binary Search Algorithm:**

In computer science, a binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

**Overview**

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.

If the list to be searched contains more than a few items (a dozen, say) a binary search will require far fewer comparisons than a linear search, but it imposes the requirement that the list be sorted. Similarly, a hash search can be faster than a binary search but imposes still greater requirements. If the contents of the array are modified between searches, maintaining these requirements may even take more time than the searches. And if it is known that some items will be searched for much more often than others, and it can be arranged that these items are at the start of the list, then a linear search may be the best. More generally, an algorithm allows searching over the argument of any monotonic function for a point, at which function reaches the arbitrary value (enclosed between minimum and maximum at the given range).

Examples Example:

L = 1 3 4 6 8 9 11. X = 4.

Compare X to 6. It's smaller. Repeat with L = 1 3 4.

Compare X to 3. It's bigger. Repeat with L = 4.

Compare X to 4. It's equal. We're done, we found X.

Each iteration of (1)-(4) the length of the list we are looking in gets cut in half.

Therefore, the total number of iterations cannot be greater than logN.

**Number guessing game**

This rather simple game begins with something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'Higher', 'Lower', or 'Yes!' as might be the case." Supposing that N is the number of possible values (here, twenty-one as "inclusive" was stated), then at most ⌊Log2N⌋ questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.

Even if the number to guess can be arbitrarily large, in which case there is no upper bound N, the number can be found in at most 2⌊Log2 K⌋ + 1 steps (where k is the (unknown) selected number) by first finding an upper bound by repeated doubling.

For example, if the number were 11, the following sequence of guesses could be used to find it: 1 (Higher), 2 (Higher), 4 (Higher), 8 (Higher), 16 (Lower), 12 (Lower), 10 (Higher). Now we know that the number must be 11 because it is higher than 10 and lower than 12. One could also extend the method to include negative numbers; for example the following guesses could be used to find −13: 0, −1, −2, −4, −8, −16, −12, −14. Now we know that the number must be −13 because it is lower than −12 and higher than −14.

**Recursive**

A straightforward implementation of binary search is recursive. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray. Each of the calls is tail recursive, so a compiler need not make a new stack frame for each call. The variables *imin* and *imax* are the lowest and highest inclusive indices that are searched.



It is invoked with initial imin and imax values of 0 and N-1 for a zero based array of length N.

The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as "(imin + imax) / 2". In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result "(imin + imax)" might overflow. With limited numbers, the midpoint can be calculated correctly as "imin + ((imax - imin) / 2)".

**Binary Search** is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

**Binary Search Working**

Binary Search Algorithm can be implemented in two ways which are discussed below.

Iterative Method

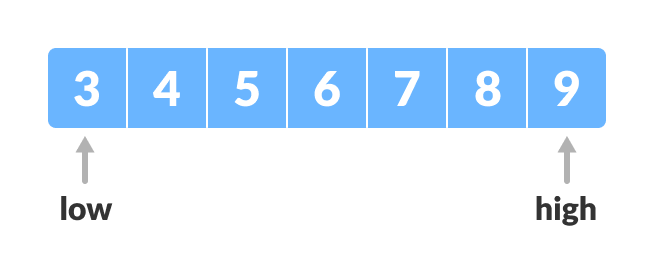
Recursive Method

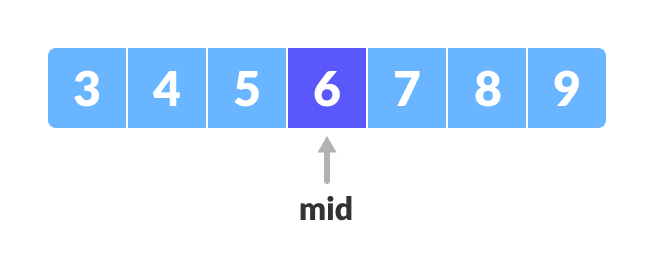
The recursive method follows [the divide and conquer](https://www.programiz.com/dsa/divide-and-conquer) approach.

The general steps for both methods are discussed below.

The array in which searching is to be performed is:Initial array

Let x = 4 be the element to be searched.

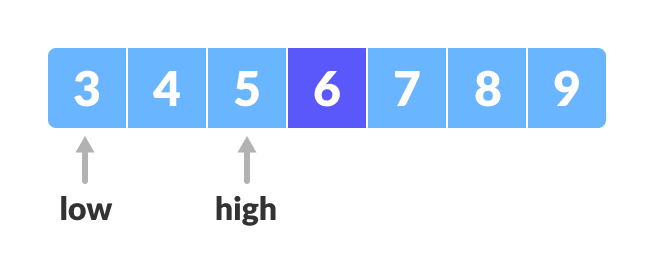
Set two pointers low and high at the lowest and the highest positions respectively.Setting pointers

Find the middle element mid of the array ie. (arr[low + high]) / 2 = 6.Mid element

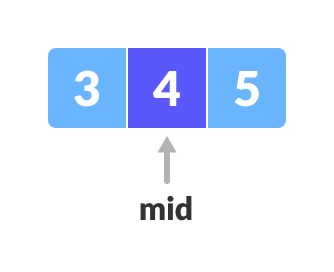
If x == mid, then return mid.Else, compare the element to be searched with m.

If x > mid, compare x with the middle element of the elements on the right side of mid. This is done by setting low to low = mid + 1.

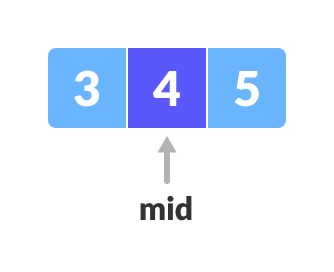
Else, compare x with the middle element of the elements on the left side of mid. This is done by setting high to high = mid - 1.

Finding mid element

Repeat steps 3 to 6 until low meets high.

Mid element

x = 4 is found.



Found

**Iteration Method**

do until the pointers low and high meet each other.

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > A[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid - 1

Recursive Method

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

return mid

else if x < data[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the right side

return binarySearch(arr, x, low, mid - 1)

**# Binary Search in python**

**Example-1:**

**def binarySearch(array, x, low, high):**

**# Repeat until the pointers low and high meet each other**

**while low <= high:**

**mid = low + (high - low)//2**

**if array[mid] == x:**

**return mid**

**elif array[mid] < x:**

**low = mid + 1**

**else:**

**high = mid - 1**

**return -1**

**array = [3, 4, 5, 6, 7, 8, 9]**

**x = 4**

**result = binarySearch(array, x, 0, len(array)-1)**

**if result != -1:**

**print("Element is present at index " + str(result))**

**else:**

**print("Not found")**

**Example-2:**

**# Binary Search in python**

def binarySearch(array, x, low, high):

if high >= low:

mid = low + (high - low)//2

# If found at mid, then return it

if array[mid] == x:

return mid

# Search the left half

elif array[mid] > x:

return binarySearch(array, x, low, mid-1)

# Search the right half

else:

return binarySearch(array, x, mid + 1, high)

else:

return -1

array = [3, 4, 5, 6, 7, 8, 9]

x = 4

result = binarySearch(array, x, 0, len(array)-1)

if result != -1:

print("Element is present at index " + str(result))

else:

print("Not found")

**Binary Search Complexity**

Time Complexities

Best case complexity: O(1)

Average case complexity: O(log n)

Worst case complexity: O(log n)

**Space Complexity**

The space complexity of the binary search is O(n).

**Binary Search Applications**

In libraries of Java, .Net, C++ STL

While debugging, the binary search is used to pinpoint the place where the error happens.

**Sorting**

Sorting is the process of arranging or ordering a collection of items such that each item and its successor satisfy a prescribed relationship. The items can be simple values, such as integers and reals, or more complex types, such as student records or dictionary entries. In either case, the ordering of the items is based on the value of a sort key. The key is the value itself when sorting simple types or it can be a specific component or a combination of components when sorting complex types.

**Insertion sort**

This is an in-place comparison-based sorting algorithm. A sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element that is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Algorithm**

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.

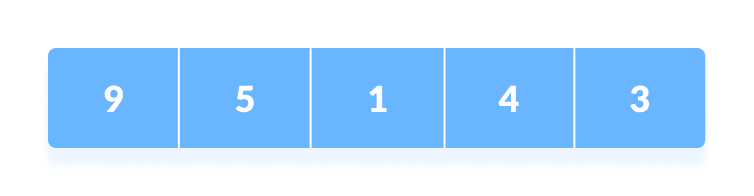
In the same way, other unsorted cards are taken and put at their right place.

A similar approach is used by insertion sort.

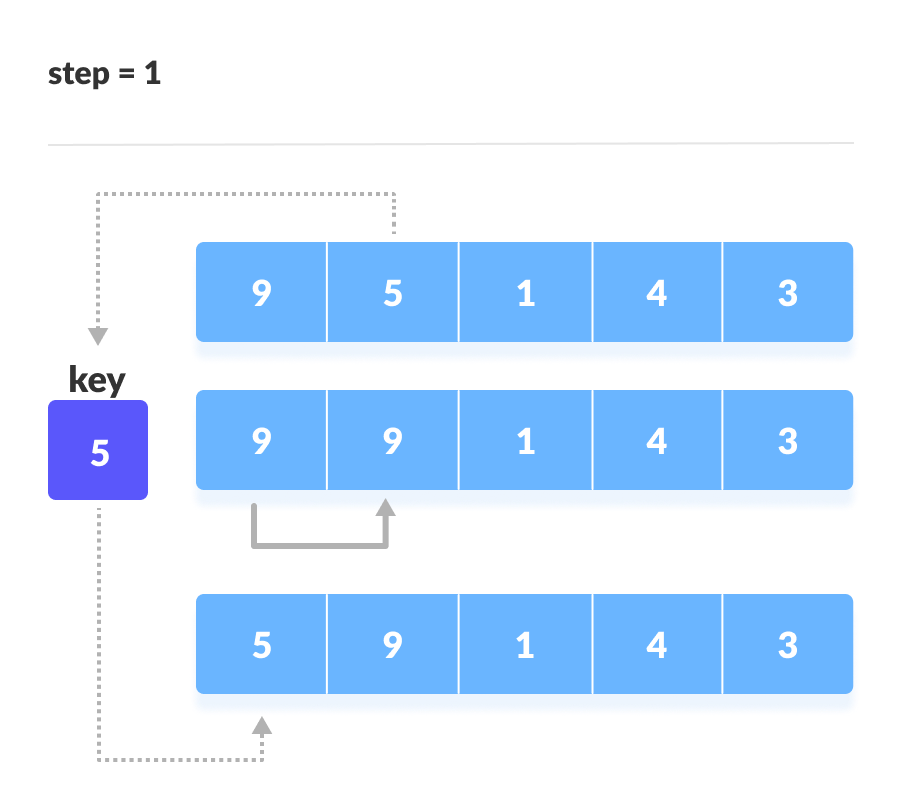
Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

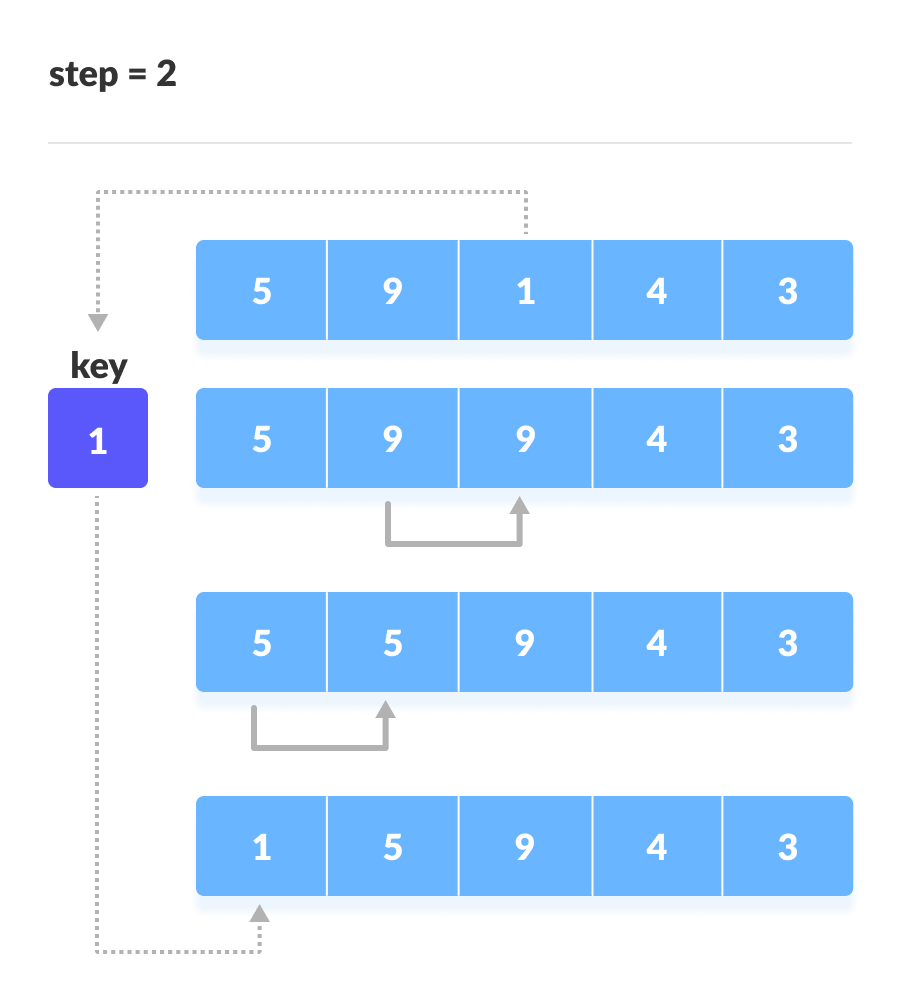
**How Insertion Sort Works?**

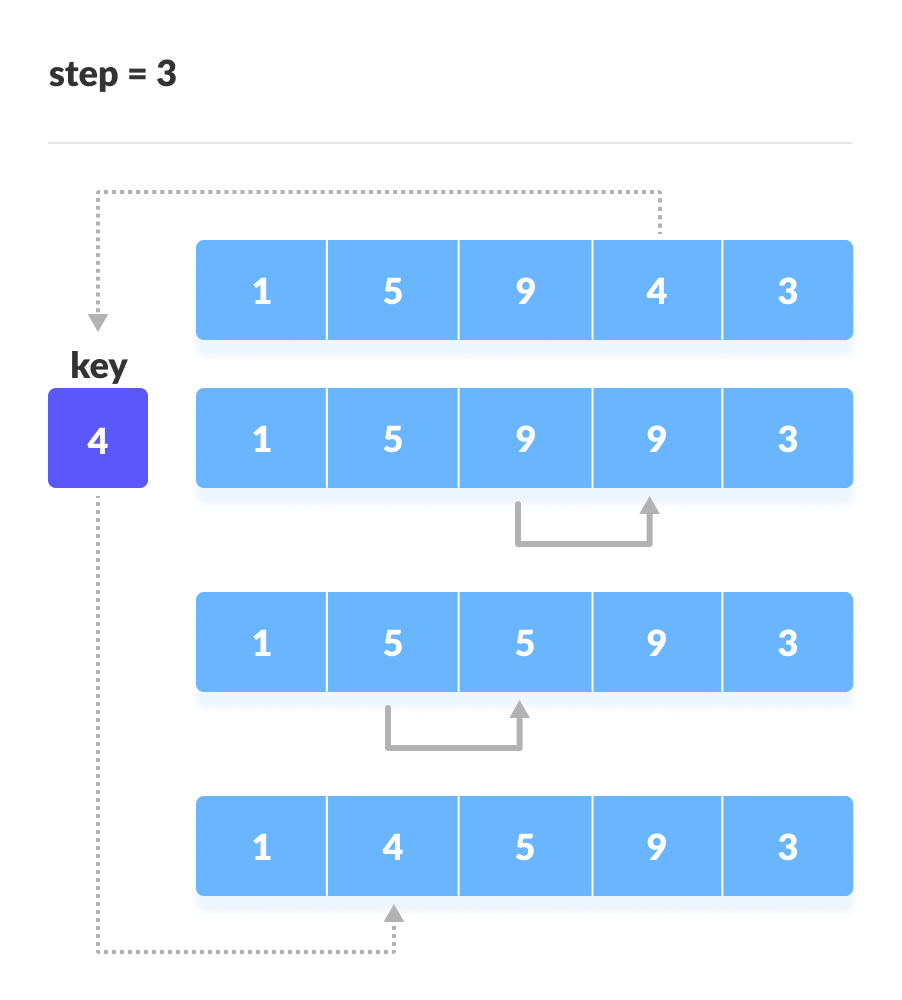
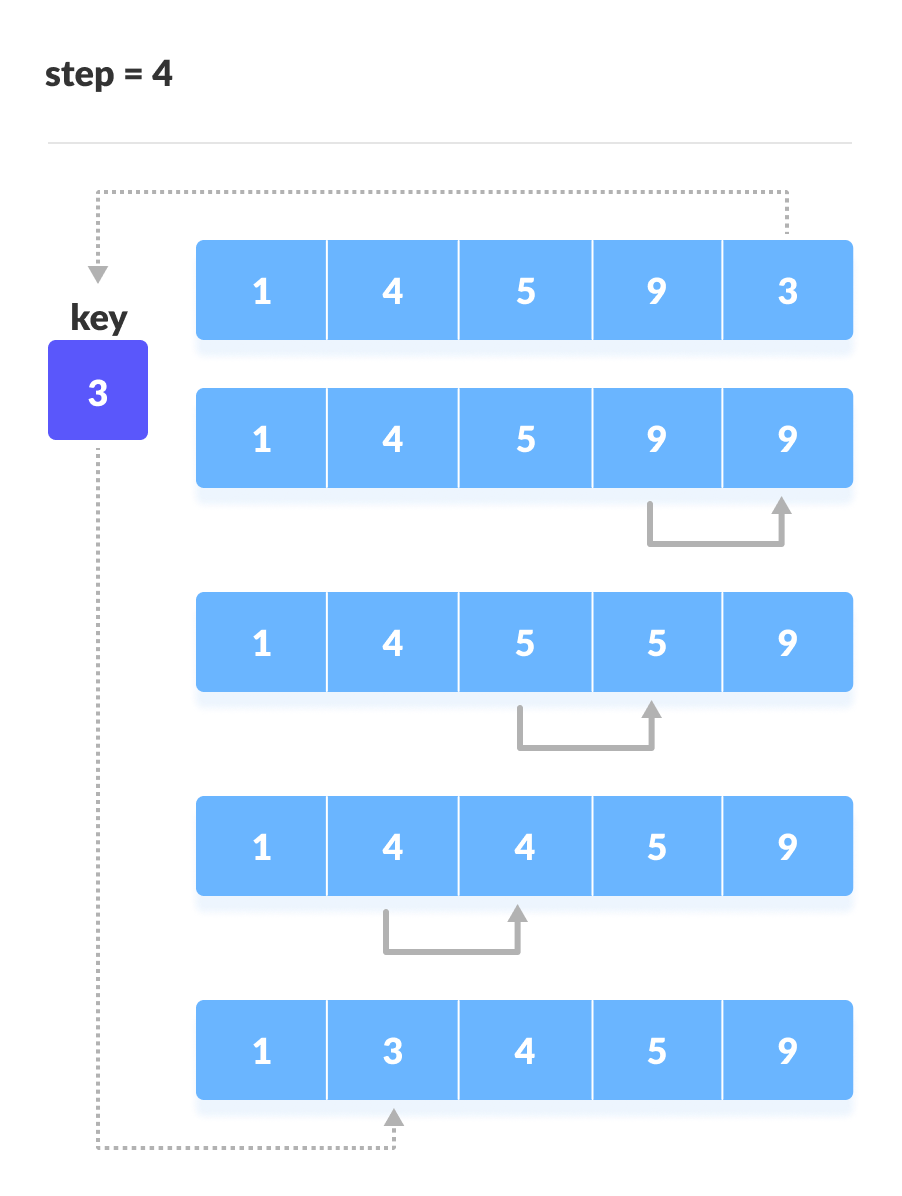
Suppose we need to sort the following array.

Initial array

The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.If the first element is greater than key, then key is placed in front of the first element.

Now, the first two elements are sorted.  
  
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.Place 1 at the beginning

Similarly, place every unsorted element at its correct position.Place 4 behind 1 Place 3 behind 1 and the array is sorted

**Insertion Sort Algorithm**

insertionSort(array)

mark first element as sorted

for each unsorted element X

'extract' the element X

for j <- lastSortedIndex down to 0

if current element j > X

move sorted element to the right by 1

break loop and insert X here

end insertionSort

**# Insertion sort in Python**

def insertionSort(array):

for step in range(1, len(array)):

key = array[step]

j = step - 1

# Compare key with each element on the left of it until an element smaller than it is found

# For descending order, change key<array[j] to key>array[j].

while j >= 0 and key < array[j]:

array[j + 1] = array[j]

j = j - 1

# Place key at after the element just smaller than it.

array[j + 1] = key

data = [9, 5, 1, 4, 3]

insertionSort(data)

print('Sorted Array in Ascending Order:')

print(data)

**Complexity**

Time Complexities

**Worst Case Complexity: O(n2)**

Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.  
  
Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.  
  
Thus, the total number of comparisons = n\*(n-1) ~ n2

**Best Case Complexity: O(n)**When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

**Average Case Complexity: O(n2)**It occurs when the elements of an array are in jumbled order (neither ascending nor descending).

**Space Complexity**

Space complexity is O(1) because an extra variable key is used.

**Insertion Sort Applications**

The insertion sort is used when:

the array is has a small number of elements

there are only a few elements left to be sorted

**Selection Sort**

Selection sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving an unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

**Algorithm**

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

**# Selection sort in Python**

**def selectionSort(array, size):**

**for step in range(size):**

**min\_idx = step**

**for i in range(step + 1, size):**

**# to sort in descending order, change > to < in this line**

**# select the minimum element in each loop**

**if array[i] < array[min\_idx]:**

**min\_idx = i**

**# put min at the correct position**

**(array[step], array[min\_idx]) = (array[min\_idx], array[step])**

**data = [-2, 45, 0, 11, -9]**

**size = len(data)**

**selectionSort(data, size)**

**print('Sorted Array in Ascending Order:')**

**print(data)**

**Time Complexities:**

* **Worst Case Complexity: O(n2)**If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
* **Best Case Complexity: O(n2)**It occurs when the array is already sorted
* **Average Case Complexity: O(n2)**It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place. The minimum element is not known until the end of the array is not reached.

**Space Complexity:**

Space complexity is O(1) because an extra variable temp is used.

## **Selection Sort Applications**

**The selection sort is used when:**

* **a small list is to be sorted**
* **cost of swapping does not matter**
* **checking of all the elements is compulsory**
* **cost of writing to a memory matters like in flash memory (number of writes/swaps is O(n) as compared to O(n2) of bubble sort)**

**Bubble Sort**

Bubble 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. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with the very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

**Algorithm**

We assume the list is an array of **n** elements. We further assume that the swap function swaps the values of the given array elements.

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**# Bubble sort in Python**

**def bubbleSort(array):**

**# run loops two times: one for walking through the array**

**# and the other for comparison**

**for i in range(len(array)):**

**for j in range(0, len(array) - i - 1):**

**# To sort in descending order, change > to < in this line.**

**if array[j] > array[j + 1]:**

**# swap if greater is at the rear position**

**(array[j], array[j + 1]) = (array[j + 1], array[j])**

**data = [-2, 45, 0, 11, -9]**

**bubbleSort(data)**

**print('Sorted Array in Ascending Order:')**

**print(data)**

**Time Complexities:**

* **Worst Case Complexity:O(n2)**If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
* **Best Case Complexity:O(n)**If the array is already sorted, then there is no need for sorting.
* **Average Case Complexity:O(n2)**It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

**Space Complexity:**

Space complexity is O(1) because an extra variable temp is used for swapping.

In the optimized algorithm, the variable swapped adds to the space complexity thus, making it O(2).

## **Bubble Sort Applications**

Bubble sort is used in the following cases where

1. the complexity of the code does not matter.
2. a short code is preferred.