1. Python Iterators

Iterators are objects that can be iterated upon. In this tutorial, you will learn how iterator works and how you can build your own iterator using __iter__ and __next__ methods.

What are iterators in Python?

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.
Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
Technically speaking, Python **iterator object** must implement two special methods, __iter__() and __next__(), collectively called the **iterator protocol**.
An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.
The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterating Through an Iterator in Python
We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise StopIteration. Following is an example.

```
# define a list

my_list = [4, 7, 0, 3]

# get an iterator using iter()

my_iter = iter(my_list)

## iterate through it using next()

#prints 4

print(next(my_iter))

#prints 7

print(next(my_iter))

## next(obj) is same as obj.__next__()

#prints 0

print(my_iter.__next__())

#prints 3

print(my_iter.__next__())

## This will raise error, no items left

next(my_iter)
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>>for element inmy_list:
...     print(element)
...
4
7
0
3
```

How for loop actually works?

As we see in the above example, the for loop was able to iterate automatically through the list.
In fact the for loop can iterate over any iterable. Let's take a closer look at how the for loop is actually implemented in Python.

```
for element initerable:
# do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)

# infinite loop
whileTrue:
try:
# get the next item
        element = next(iter_obj)
# do something with element
exceptStopIteration:
# if StopIteration is raised, break from loop
break
```

So internally, the for loop creates an iterator object, iter_obj by calling iter() on the iterable.
Ironically, this for loop is actually an infinite while loop.
Inside the loop, it calls next() to get the next element and executes the body of the for loop with this value. After all the items exhaust, StopIteration is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Building Your Own Iterator in Python

Building an iterator from scratch is easy in Python. We just have to implement the methods __iter__() and __next__().
The __iter__() method returns the iterator object itself. If required, some initialization can be performed.
The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

def __init__(self, max = 0):
self.max = max

def __iter__(self):
self.n = 0
        return self

def __next__(self):
        if self.n<= self.max:
            result = 2 ** self.n
self.n += 1
            return result
        else:
            raise StopIteration
```

2 Python Recursion

What is recursion in Python?

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.

Python Recursive Function
We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Following is an example of 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.

```
# An example of a recursive function to

# find the factorial of a number


defcalc_factorial(x):

    """This is a recursive function

    to find the factorial of an integer"""


    if x == 1:

        return 1

    else:

        return (x * calc_factorial(x-1))


num = 4

print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, calc_factorial() is a recursive functions as it calls itself.

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

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

```
1. calc_factorial(4)            # 1st call with 4
2. 4 * calc_factorial(3)        # 2nd call with 3
3. 4 * 3 * calc_factorial(2)    # 3rd call with 2
4. 4 * 3 * 2 * calc_factorial(1)  # 4th call with 1
5. 4 * 3 * 2 * 1# return from 4th call as number=1
6. 4 * 3 * 2# return from 3rd call
7. 4 * 6# return from 2nd call
8. 24# return from 1st call
```

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.

Advantages of Recursion

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

Disadvantages of Recursion

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

3. Python Program to Display Fibonacci sequence Using Recursion

In this program, you'll learn to display Fibonacci sequence using a recursive function.

To understand this example, you should have the knowledge of following Python programming topics:

- Python for Loop
- Python Functions
- Python Recursion

A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8....

The first two terms are 0 and 1. All other terms are obtained by adding the preceding two terms.This means to say the nth term is the sum of $(n-1)^{th}$ and $(n-2)^{th}$ term.

Source Code

```
# Python program to display the Fibonacci sequence

defrecur_fibo(n):
if n <= 1:
return n
else:
return(recur_fibo(n-1) + recur_fibo(n-2))

nterms = 10

# check if the number of terms is valid
ifnterms<= 0:
print("Plese enter a positive integer")
else:
print("Fibonacci sequence:")
foriin range(nterms):
print(recur_fibo(i))
```

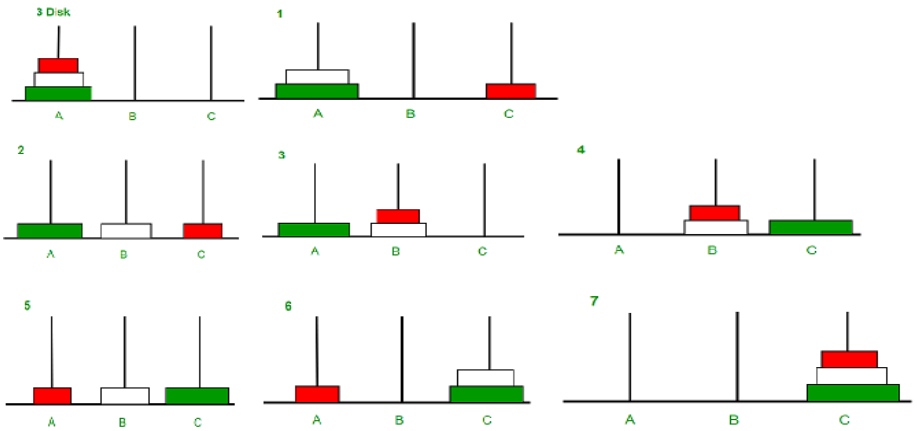
Output

```
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
```

Note: To test the program, change the value of nterms.
In this program, we store the number of terms to be displayed in nterms.
A recursive function recur_fibo() is used to calculate the nth term of the sequence. We use a for loop to iterate and calculate each term recursively.
Visit here to know more about recursion in Python.

4. Python Program for Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.



```
# Recursive Python function to solve tower of hanoi

defTowerOfHanoi(n , from_rod, to_rod, aux_rod):
    ifn ==1:
        print"Move disk 1 from rod",from_rod,"to rod",to_rod
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print"Move disk",n,"fromrod",from_rod,"to rod",to_rod
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

# Driver code
n =4
TowerOfHanoi(n, \'A\', \'C\', \'B\')
# A, C, B are the name of rods
```

Output:

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Sorting, searching and algorithm analysis

Introduction

We have learned that in order to write a computer program which performs some task we must construct a suitable algorithm. However, whatever algorithm we construct is unlikely to be unique – there are likely to be many possible algorithms which can perform the same task. Are some of these algorithms in some sense better than others? Algorithm analysis is the study of this question.

In this chapter we will analyse four algorithms; two for each of the following common tasks:

- *sorting*: ordering a list of values
- *searching*: finding the position of a value within a list

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.

Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:

- *time complexity*: how the number of steps required depends on the size of the input
- *space complexity*: how the amount of extra memory or storage required depends on the size of the input

Sorting algorithms

The sorting of a list of values is a common computational task which has been studied extensively. The classic description of the task is as follows:

Given a *list of values* and a function that *compares two values*, order the values in the list from smallest to largest.

The values might be integers, or strings or even other kinds of objects. We will examine two algorithms:

- *Selection sort*, which relies on repeated *selection* of the next smallest item
- *Merge sort*, which relies on repeated *merging* of sections of the list that are already sorted

Other well-known algorithms for sorting lists are *insertion sort*, *bubble sort*, *heap sort*, *quicksort* and *shell sort*.

There are also various algorithms which perform the sorting task for restricted kinds of values, for example:

- *Counting sort*, which relies on the values belonging to a small set of items
- *Bucket sort*, which relies on the ability to map each value to one of a small set of items
- *Radix sort*, which relies on the values being sequences of digits

If we restrict the task, we can enlarge the set of algorithms that can perform it. Among these new algorithms may be ones that have desirable properties. For example, *Radix sort* uses fewer steps than any generic sorting algorithm.

Algorithm	**Best case**	**Expected**	**Worst case**
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Merge sort	O(N log N)	O(N log N)	O(N log N)
Linear search	O(1)	O(N)	O(N)
Binary search	O(1)	O(log N)	O(log N)

What does O(1) mean? It means that the cost of an algorithm is *constant*, no matter what the value of N is. For both these search algorithms, the best case scenario happens when the first element to be tested is the correct element – then we only have to perform a single operation to find it.

In the previous table, big O notation has been used to describe the *time complexity* of algorithms. It can also be used to describe their *space complexity* – in which case the cost function represents the number of units of space required for storage rather than the required number of operations. Here are the space complexities of the algorithms above (for the worst case, and excluding the space required to store the input):

Algorithm	Space complexity
Selection sort	O(1)
Merge sort	O(N)
Linear search	O(1)
Binary search	O(1)

5. Python | Merge list elements

Sometimes, we require to merge some of the elements as single element in the list. This is usually with the cases with character to string conversion. This type of task is usually required in the development domain to merge the names into one element. Let's discuss certain ways in which this can be performed.

Method #1 :

Using join()+ListSlicing

The join function can be coupled with list slicing which can perform the task of joining each character in a range picked by the list slicing functionality.

```
# Python3 code to demonstrate

# merging list elements

# using join() + list slicing


# initializing list

test_list =['I', 'L', 'O', 'V', 'E', 'G', 'F', 'G']


# printing original list

print("The original list is : "+str(test_list))


# using join() + list slicing

# merging list elements
```

```
test_list[5: 8] =[''.join(test_list[5: 8])]


# printing result
print("The list after merging elements : "+  str(test_list))
```

Output:

```
The original list is : ['I', 'L', 'O', 'V', 'E', 'G', 'F', 'G']
The list after merging elements : ['I', 'L', 'O', 'V', 'E', 'GFG']
```

Method #2 : Using reduce() + lambda + list slicing
The task of joining each element in a range is performed by reduce function and lambda. reduce function performs the task for each element in the range which is defined by the lambda function. It works with Python2 only

```
# Python code to demonstrate
# merging list elements
# using reduce() + lambda + list slicing

# initializing list
test_list =['I', 'L', 'O', 'V', 'E', 'G', 'F', 'G']

# printing original list
print("The original list is : "+str(test_list))

# using reduce() + lambda + list slicing
# merging list elements
test_list[5: 8] =[reduce(lambdai, j: i +j, test_list[5: 8])]

# printing result
print("The list after merging elements : "+  str(test_list))
```

Output:

```
The original list is : ['I', 'L', 'O', 'V', 'E', 'G', 'F', 'G']
The list after merging elements : ['I', 'L', 'O', 'V', 'E', 'GFG']
```

6. Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

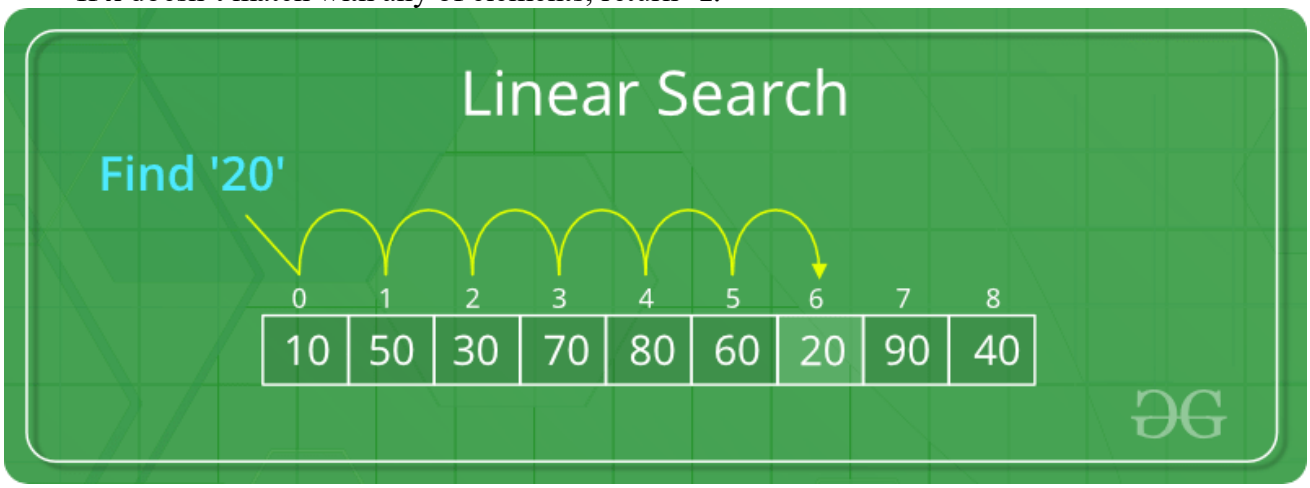
Examples :

Input :arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Input :arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 175;
Output : -1
Element x is not present in arr[].

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Example:

```
# Python3 code to linearly search x in arr[].
# If x is present then return its location,
```

```
# otherwise return -1

defsearch(arr, n, x):

    fori inrange(0, n):
        if(arr[i] ==x):
            returni;
    return-1;

# Driver Code
arr =[ 2, 3, 4, 10, 40];
x =10;
n =len(arr);
result =search(arr, n, x)
if(result ==-1):
    print("Element is not present in array")
else:
    print("Element is present at index", result);
```

Output:

```
Element is present at index 3
```

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

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.

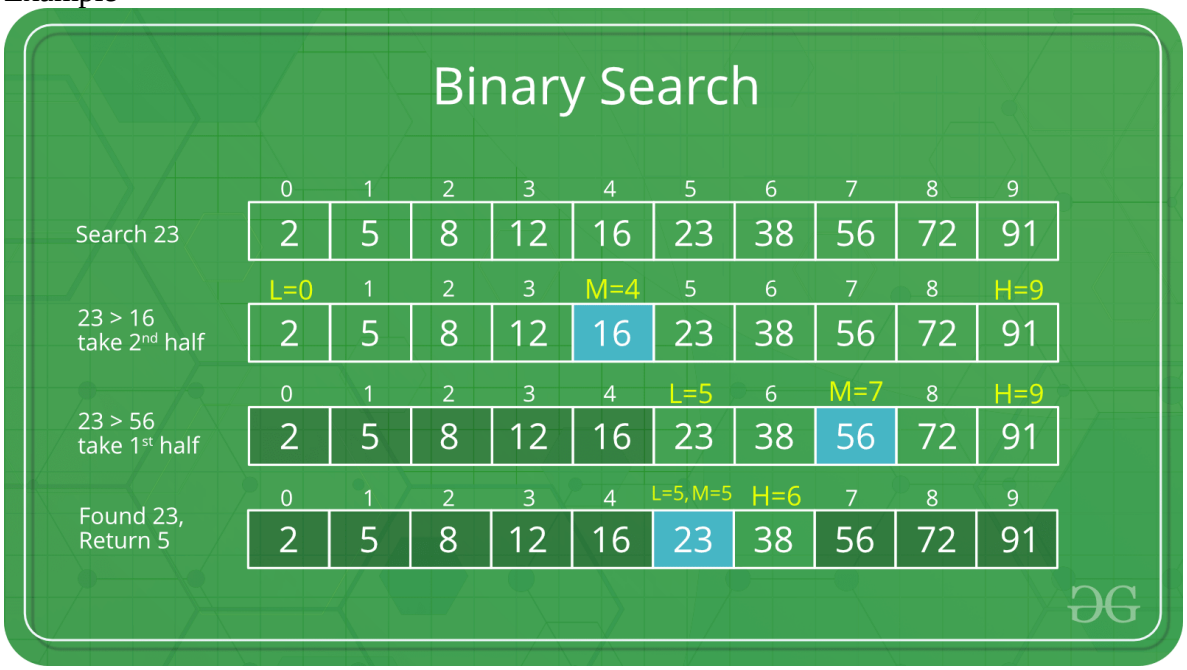
7. Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search.**The time complexity of above algorithm is O(n). Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example :



The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Recursive implementation of Binary Search

```
# Python Program for recursive binary search.

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

    # Check base case
    ifr >=l:

        mid =l +(r -l)/2

        # If element is present at the middle itself
        ifarr[mid] ==x:
            returnmid

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

```
        # Else the element can only be present
        # in right subarray
        else:
            returnbinarySearch(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)

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

Output :

```
Element is present at index 3
```

Iterative implementation of Binary Search

```
# Python code to implement iterative Binary
# Search.

# It returns location of x in given array arr
# if present, else returns -1
defbinarySearch(arr, l, r, x):

    whilel <=r:

        mid =l +(r -l)/2;

        # Check if x is present at mid
        ifarr[mid] ==x:
            returnmid

        # If x is greater, ignore left half
        elifarr[mid] < x:
            l =mid +1

        # If x is smaller, ignore right half
        else:
            r =mid -1
```

```
    # If we reach here, then the element
    # was not present
    return-1


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

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

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

Output :

```
Element is present at index 3
```

Time Complexity:

The time complexity of Binary Search can be written as

$T(n) = T(n/2) + c$

The above recurrence can be solved either using Recurrence T ree method or Master method.

It falls in case II of Master Method and solution of the recurrence is .

Auxiliary Space: O(1) in case of iterative implementation. In case of recursive implementation, O(Logn) recursion call stack space.

8. Selection Sort Algorithm

In this tutorial, you will learn how selection sort works. Also, you will find working examples of selection sort in C, C++, Java and Python.

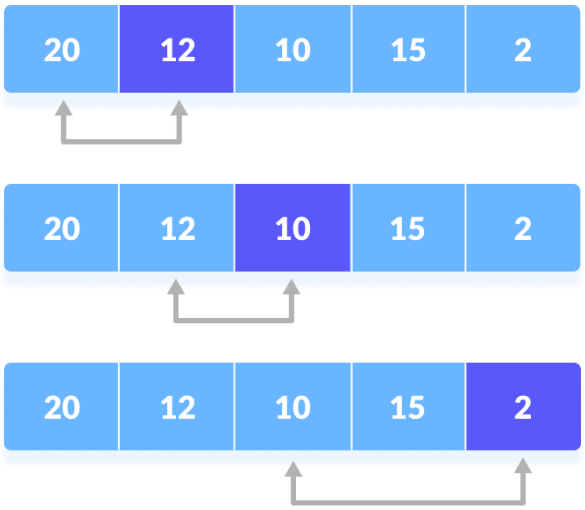
Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

How Selection Sort Works?

1. Set the first element as minimum.

2. Compare minimum with the second element. If the second element is smaller than minimum, assign second element as minimum.

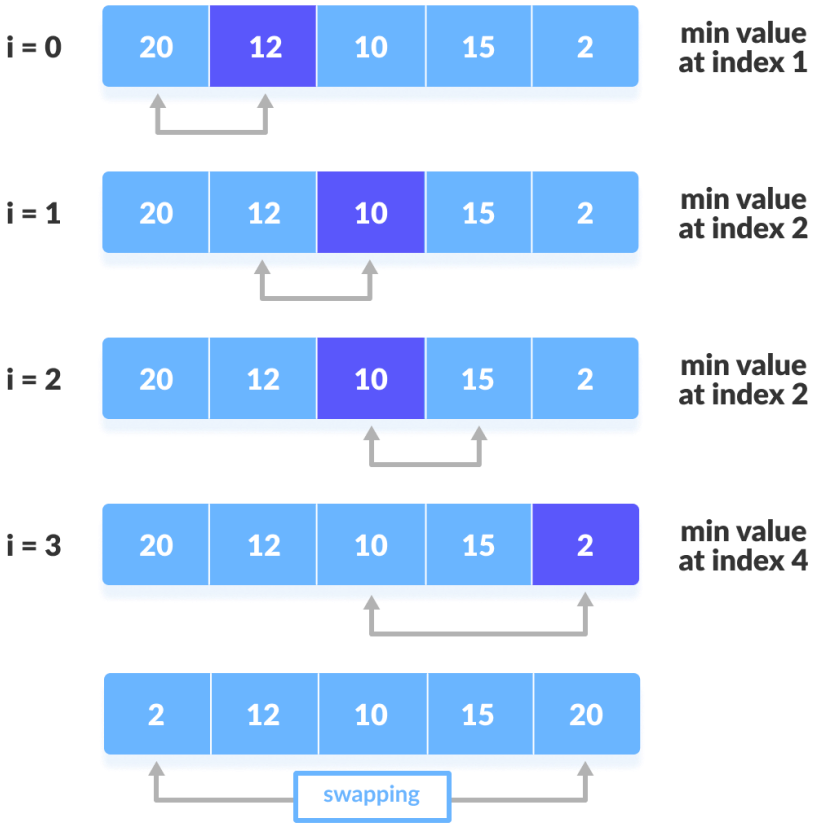
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



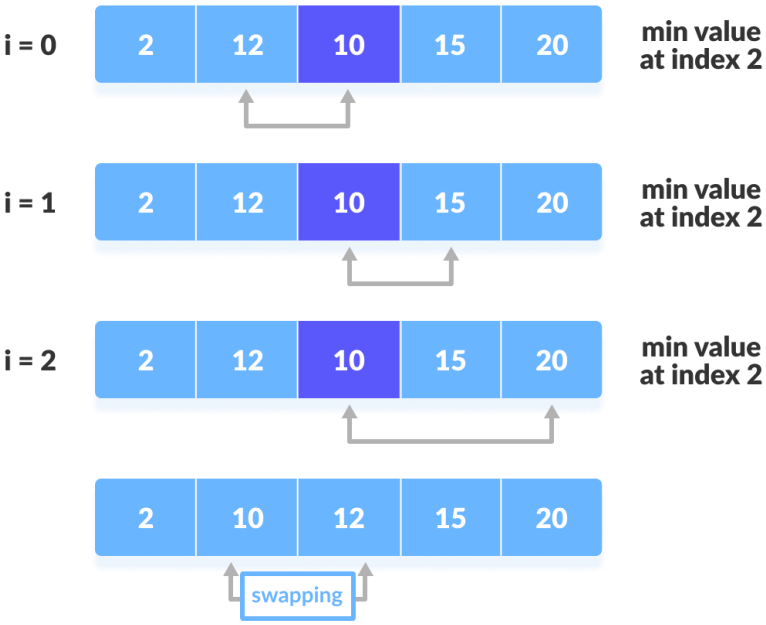
3. After each iteration, minimum is placed in the front of the unsorted list.

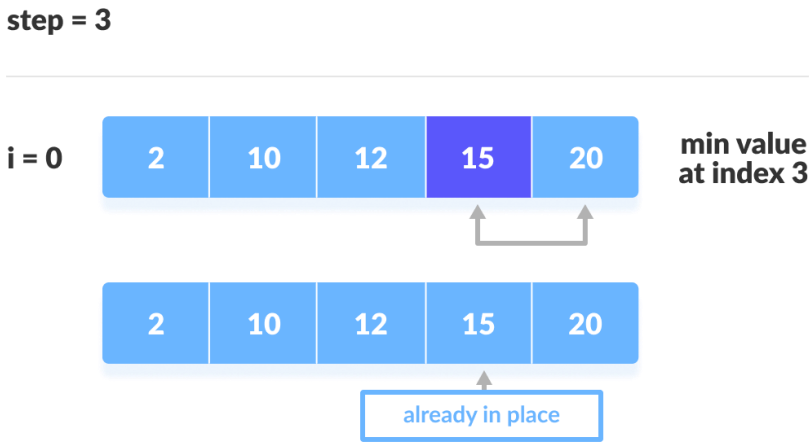
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

step = 0


i = 0
20 12 10 15 2
min value at index 1
i = 1
20 12 10 15 2
min value at index 2
i = 2
20 12 10 15 2
min value at index 2
i = 3
20 12 10 15 2
min value at index 4
2 12 10 15 20
swapping

step = 1


i = 0
2 12 10 15 20
min value at index 2
i = 1
2 12 10 15 20
min value at index 2
i = 2
2 12 10 15 20
min value at index 2
2 10 12 15 20
swapping



Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
set the first unsorted element as the minimum
for each of the unsorted elements
if element <currentMinimum
set element asnew minimum
  swap minimum with first unsorted position
endselectionSort
```

- Python

```python
# Selection sort in Python

defselectionSort(array, size):
for step in range(size):
min_idx = step
foriinrange(step + 1, size):

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

if array[i] < array[min_idx]:
min_idx = i

        (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:\n')
print(data)
```

Complexity

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

Number of comparisons: (n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2 nearly equals to n^2

Complexity = $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$.

Time Complexities:

WorstCaseComplexity: $O(n^2)$

If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **BestCaseComplexity:** $O(n^2)$

  It occurs when the the array is already sorted

- **AverageCaseComplexity:** $O(n^2)$

  It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of 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:

- 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)

9. Python Program for Merge Sort

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge () function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one

MergeSort(arr[], l, r)
If r > l
1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2
2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge () function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

 # Python program for implementation of MergeSort

```
# Merges two subarrays of arr[].

# First subarray is arr[l..m]

# Second subarray is arr[m+1..r]

defmerge(arr, l, m, r):

    n1 =m -l +1

    n2 =r-m


    # create temp arrays

    L =[0] *(n1)

    R =[0] *(n2)


    # Copy data to temp arrays L[] and R[]

    fori inrange(0, n1):

        L[i] =arr[l +i]


    forj inrange(0, n2):

        R[j] =arr[m +1+j]


    # Merge the temp arrays back into arr[l..r]

    i =0     # Initial index of first subarray

    j =0     # Initial index of second subarray
```

```
    k =l     # Initial index of merged subarray

    whilei< n1 andj < n2 :
        ifL[i] <=R[j]:
            arr[k] =L[i]
            i +=1
        else:
            arr[k] =R[j]
            j +=1
        k +=1

    # Copy the remaining elements of L[], if there
    # are any
    whilei< n1:
        arr[k] =L[i]
        i +=1
        k +=1
    # Copy the remaining elements of R[], if there
    # are any
    whilej < n2:
        arr[k] =R[j]
        j +=1
        k +=1
# l is for left index and r is right index of the
# sub-array of arr to be sorted
defmergeSort(arr,l,r):
    ifl < r:
        # Same as (l+r)/2, but avoids overflow for
        # large l and h
        m =(l+(r-1))/2
```

```
        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
# Driver code to test above
arr =[12, 11, 13, 5, 6, 7]
n =len(arr)
print("Given array is")
fori inrange(n):
    print("%d"%arr[i]),

mergeSort(arr,0,n-1)
print("\n\nSorted array is")
fori inrange(n):
    print("%d"%arr[i]),

# This code is contributed by MohitKumra
```

Output:

```
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```