Multiprocessing

Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system. A multiprocessing system can have: multiprocessor, i.e. a computer with more than one central processor. multi-core processor, i.e. a single computing component with two or more independent actual processing units (called “cores”).

In Python, the multiprocessing module includes a very simple and intuitive API for dividing work between multiple processes.

In [ ]:

```
# importing the multiprocessing module
import multiprocessing
def print_cube(num):
    """
    function to print cube of given num
    """
    print("Cube: {}".format(num * num * num))

def print_square(num):
    """
    function to print square of given num
    """
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating processes
    p1 = multiprocessing.Process(target=print_square, args=(10, ))
    p2 = multiprocessing.Process(target=print_cube, args=(10, ))

    # starting process 1
    p1.start()
    # starting process 2
    p2.start()

    # wait until process 1 is finished
    p1.join()
    # wait until process 2 is finished
    p2.join()

    # both processes finished
    print("Done!")
```

In [ ]:

```
Output:
Square: 100
Cube: 1000
Done!
```

Once the processes start, the current program also keeps on executing. In order to stop execution of current program until a process is complete, we use join method.

As a result, the current program will first wait for the completion of p1 and then p2. Once, they are completed, the next statements of current program are executed.

In [ ]:

```
import multiprocessing
import os

def worker1():
    # printing process id
    print("ID of process running worker1: {}".format(os.getpid()))

def worker2():
    # printing process id
    print("ID of process running worker2: {}".format(os.getpid()))

if __name__ == "__main__":
    # printing main program process id
    print("ID of main process: {}".format(os.getpid()))
    # creating processes
    p1 = multiprocessing.Process(target=worker1)
    p2 = multiprocessing.Process(target=worker2)
    # starting processes
    p1.start()
    p2.start()
    # process IDs
    print("ID of process p1: {}".format(p1.pid))
    print("ID of process p2: {}".format(p2.pid))
    # wait until processes are finished
    p1.join()
    p2.join()
    # both processes finished
    print("Both processes finished execution!")
    # check if processes are alive
    print("Process p1 is alive: {}".format(p1.is_alive()))
    print("Process p2 is alive: {}".format(p2.is_alive()))
```

In [ ]:

```
Output:
ID of main process: 28628
ID of process running worker1: 29305
ID of process running worker2: 29306
ID of process p1: 29305
ID of process p2: 29306
Both processes finished execution!
Process p1 is alive: False
Process p2 is alive: False
```

Shared memory

In [ ]:

```
Shared memory : multiprocessing module provides Array and Value objects to share data b
etween processes.
Array: a ctypes array allocated from shared memory.
Value: a ctypes object allocated from shared memory.
```

In [ ]:

```
import multiprocessing

def square_list(mylist, result, square_sum):
    # append squares of mylist to result array
    for idx, num in enumerate(mylist):
        result[idx] = num * num
    # square_sum value
    square_sum.value = sum(result)
    # print result Array
    print("Result(in process p1): {}".format(result[:]))
    # print square_sum Value
    print("Sum of squares(in process p1): {}".format(square_sum.value))

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]
    # creating Array of int data type with space for 4 integers
    result = multiprocessing.Array('i', 4)
    # creating Value of int data type
    square_sum = multiprocessing.Value('i')
    # creating new process
    p1 = multiprocessing.Process(target=square_list, args=(mylist, result, square_sum))
    # starting process
    p1.start()
    # wait until process is finished
    p1.join()
    # print result array
    print("Result(in main program): {}".format(result[:]))
    # print square_sum Value
    print("Sum of squares(in main program): {}".format(square_sum.value))
```

In [ ]:

```
Output:
Result(in process p1): [1, 4, 9, 16]
Sum of squares(in process p1): 30
Result(in main program): [1, 4, 9, 16]
Sum of squares(in main program): 30
```

Server process

In [ ]:

```
Server process : Whenever a python program starts, a server process is also started. Fr
om there on, whenever a new process is needed, the parent process connects to the serve
r and
requests it to fork a new process.
A server process can hold Python objects and allows other processes to manipulate them
using proxies.
multiprocessing module provides a Manager class which controls a server process. Hence,
managers provide a way to create data which can be shared between different processes.

Server process managers are more flexible than using shared memory objects because they
can be made to support arbitrary object types like
lists, dictionaries, Queue, Value, Array, etc. Also, a single manager can be shared by
processes on different computers over a network. They are,
however, slower than using shared memory.
```

In [ ]:

```
import multiprocessing
def print_records(records):
    """
    function to print record(tuples) in records(list)
    """
    for record in records:
        print("Name: {0}\nScore: {1}\n".format(record[0], record[1]))

def insert_record(record, records):
    """
    function to add a new record to records(list)
    """
    records.append(record)
    print("New record added!\n")

if __name__ == '__main__':
    with multiprocessing.Manager() as manager:
        # creating a list in server process memory
        records = manager.list([('Sam', 10), ('Adam', 9), ('Kevin',9)])
        # new record to be inserted in records
        new_record = ('Jeff', 8)
        # creating new processes
        p1 = multiprocessing.Process(target=insert_record, args=(new_record, records))
        p2 = multiprocessing.Process(target=print_records, args=(records,))
        # running process p1 to insert new record
        p1.start()
        p1.join()
        # running process p2 to print records
        p2.start()
        p2.join()
```

In [ ]:

```
Output:
New record added!
Name: Sam
Score: 10
Name: Adam
Score: 9
Name: Kevin
Score: 9
Name: Jeff
Score: 8
```

Communication between processes

In [ ]:

```
Effective use of multiple processes usually requires some communication between them, s
o that work can be divided and results can be aggregated.
multiprocessing supports two types of communication channel between processes:
    Queue
    Pipe
```

Queue

In [ ]:

```
Queue : A simple way to communicate between process with multiprocessing is to use a Qu
eue to pass messages back and forth. Any Python object can pass through a Queue.
```

In [ ]:

```
import multiprocessing

def square_list(mylist, q):
    """
    function to square a given list
    """
    # append squares of mylist to queue
    for num in mylist:
        q.put(num * num)

def print_queue(q):
    """
    function to print queue elements
    """
    print("Queue elements:")
    while not q.empty():
        print(q.get())
    print("Queue is now empty!")

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]
    # creating multiprocessing Queue
    q = multiprocessing.Queue()
    # creating new processes
    p1 = multiprocessing.Process(target=square_list, args=(mylist, q))
    p2 = multiprocessing.Process(target=print_queue, args=(q,))
    # running process p1 to square list
    p1.start()
    p1.join()
    # running process p2 to get queue elements
    p2.start()
    p2.join()
```

In [ ]:

```
Output:
Queue elements:
1
4
9
16
Queue is now empty!
```

Pipe

In [ ]:

```
Pipes : A pipe can have only two endpoints. Hence, it is preferred over queue when only
two-way communication is required.
multiprocessing module provides Pipe() function which returns a pair of connection obje
cts connected by a pipe. The two connection objects returned by Pipe()
represent the two ends of the pipe.
Each connection object has send() and recv() methods (among others).
```

In [ ]:

```
import multiprocessing
def sender(conn, msgs):
    """
    function to send messages to other end of pipe
    """
    for msg in msgs:
        conn.send(msg)
        print("Sent the message: {}".format(msg))
    conn.close()

def receiver(conn):
    """
    function to print the messages received from other
    end of pipe
    """
    while 1:
        msg = conn.recv()
        if msg == "END":
            break
        print("Received the message: {}".format(msg))

if __name__ == "__main__":
    # messages to be sent
    msgs = ["hello", "hey", "hru?", "END"]
    # creating a pipe
    parent_conn, child_conn = multiprocessing.Pipe()
    # creating new processes
    p1 = multiprocessing.Process(target=sender, args=(parent_conn,msgs))
    p2 = multiprocessing.Process(target=receiver, args=(child_conn,))
    # running processes
    p1.start()
    p2.start()
    # wait until processes finish
    p1.join()
    p2.join()
```

In [ ]:

```
Output:
Sent the message: hello
Sent the message: hey
Sent the message: hru?
Received the message: hello
Sent the message: END
Received the message: hey
Received the message: hru?
```

Synchronization and Pooling of processes in Python

In [ ]:

```
Process synchronization is defined as a mechanism which ensures that two or more concur
rent processes do not simultaneously
execute some particular program segment known as critical section.Critical section refe
rs to the parts of the program where the shared resource is accessed.

A race condition occurs when two or more processes can access shared data and they try
to change it at the same time. As a result, the values of variables may be
unpredictable and vary depending on the timings of context switches of the processes.
```

Race Condition

In [ ]:

```
# Python program to illustrate
# the concept of race condition
# in multiprocessing
import multiprocessing
# function to withdraw from account
def withdraw(balance):
    for _ in range(10000):
        balance.value = balance.value - 1
# function to deposit to account
def deposit(balance):
    for _ in range(10000):
        balance.value = balance.value + 1

def perform_transactions():
    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 100)
    # creating new processes
    p1 = multiprocessing.Process(target=withdraw, args=(balance,))
    p2 = multiprocessing.Process(target=deposit, args=(balance,))
    # starting processes
    p1.start()
    p2.start()
    # wait until processes are finished
    p1.join()
    p2.join()
    # print final balance
    print("Final balance = {}".format(balance.value))

if __name__ == "__main__":
    for _ in range(10):
        # perform same transaction process 10 times
        perform_transactions()
```

In [ ]:

```
Output:
Final balance = 1311
Final balance = 199
Final balance = 558
Final balance = -2265
Final balance = 1371
Final balance = 1158
Final balance = -577
Final balance = -1300
Final balance = -341
Final balance = 157
```

In [ ]:

```
In above program, 10000 withdraw and 10000 deposit transactions are carried out with in
itial balance as 100. The expected final balance is 100
but what we get in 10 iterations of perform_transactions function is some different val
ues.

This happens due to concurrent access of processes to the shared data balance. This unp
redictability in balance value is nothing but race condition.
```

Using Locks

In [ ]:

```
multiprocessing module provides a Lock class to deal with the race conditions. Lock is
implemented using a Semaphore object provided by the Operating System.
    A semaphore is a synchronization object that controls access by multiple processes
to a common resource in a parallel programming environment.
    It is simply a value in a designated place in operating system (or kernel) storage
that each process can check and then change.
    Depending on the value that is found, the process can use the resource or will find
that it is already in use and must wait for some period before trying again.
    Semaphores can be binary (0 or 1) or can have additional values. Typically, a proce
ss using semaphores checks the value and then, if it using the resource,
    changes the value to reflect this so that subsequent semaphore users will know to w
ait.
```

In [ ]:

```
import multiprocessing
# function to withdraw from account
def withdraw(balance, lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()
# function to deposit to account
def deposit(balance, lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()
def perform_transactions():
    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 100)
    # creating a lock object
    lock = multiprocessing.Lock()
    # creating new processes
    p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
    p2 = multiprocessing.Process(target=deposit, args=(balance,lock))
    # starting processes
    p1.start()
    p2.start()
    # wait until processes are finished
    p1.join()
    p2.join()
    # print final balance
    print("Final balance = {}".format(balance.value))

if __name__ == "__main__":
    for _ in range(10):
        # perform same transaction process 10 times
        perform_transactions()
```

In [ ]:

```
Output:
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
Final balance = 100
```

Pooling between processes

In [ ]:

```
 Python program to find
# squares of numbers in a given list
def square(n):
    return (n*n)
if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]
    # empty list to store result
    result = []
    for num in mylist:
        result.append(square(num))
    print(result)

Only one of the cores is used for program execution and it’s quite possible that othe
r cores remain idle.
In order to utilize all the cores, multiprocessing module provides a Pool class. The
Pool class represents a pool of worker processes.
It has methods which allows tasks to be offloaded to the worker processes in a few di
fferent ways.
Here, the task is offloaded/distributed among the cores/processes automatically by Po
ol object. User doesn’t need to worry about creating processes explicitly.
```

In [ ]:

```
import multiprocessing
import os

def square(n):
    print("Worker process id for {0}: {1}".format(n, os.getpid()))
    return (n*n)
if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]
    # creating a pool object
    p = multiprocessing.Pool()
    # map list to target function
    result = p.map(square, mylist)
    print(result)
```

In [ ]:

```
Output:
Worker process id for 2: 4152
Worker process id for 1: 4151
Worker process id for 4: 4151
Worker process id for 3: 4153
Worker process id for 5: 4152
[1, 4, 9, 16, 25]
```

MultiThreading

In [ ]:

```
Just like multiprocessing, multithreading is a way of achieving multitasking. In multit
hreading, the concept of threads is used.
A thread is an entity within a process that can be scheduled for execution. Also, it is
the smallest unit of processing that can be performed in an OS (Operating System).
```

In [ ]:

```
A thread contains all this information in a Thread Control Block (TCB):
    Thread Identifier: Unique id (TID) is assigned to every new thread
    Stack pointer: Points to thread’s stack in the process. Stack contains the local va
riables under thread’s scope.
    Program counter: a register which stores the address of the instruction currently b
eing executed by thread.
    Thread state: can be running, ready, waiting, start or done.
    Thread’s register set: registers assigned to thread for computations.
    Parent process Pointer: A pointer to the Process control block (PCB) of the process
that the thread lives on.
```

In [ ]:

```
Multiple threads can exist within one process where:

Each thread contains its own register set and local variables (stored in stack).
All thread of a process share global variables (stored in heap) and the program code.
```

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

In [ ]:

```
in a simple, single-core CPU, it is achieved using frequent switching between threads.
This is termed as context switching. In context switching,
the state of a thread is saved and state of another thread is loaded whenever any inter
rupt (due to I/O or manually set) takes place.
Context switching takes place so frequently that all the threads appear to be running p
arallely (this is termed as multitasking).
# Python program to illustrate the concept
# of threading
# importing the threading module
import threading
def print_cube(num):
    """
    function to print cube of given num
    """
    print("Cube: {}".format(num * num * num))

def print_square(num):
    """
    function to print square of given num
    """
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))
    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()
    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
    # both threads completely executed
    print("Done!")
```

In [ ]:

```
# Python program to illustrate the concept
# of threading
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))

if __name__ == "__main__":
    # print ID of current process
    print("ID of process running main program: {}".format(os.getpid()))
    # print name of main thread
    print("Main thread name: {}".format(threading.main_thread().name))
    # creating threads
    t1 = threading.Thread(target=task1, name='t1')
    t2 = threading.Thread(target=task2, name='t2')
    # starting threads
    t1.start()
    t2.start()
    # wait until all threads finish
    t1.join()
    t2.join()
```

In [ ]:

```
Output:
ID of process running main program: 11758
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 11758
Task 2 assigned to thread: t2
ID of process running task 2: 11758
```

Using Locks

In [ ]:

```python
import threading

# global variable x
x = 0

def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1

def thread_task(lock):
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()

def main_task():
    global x
    # setting global variable x as 0
    x = 0
    # creating a lock
    lock = threading.Lock()
    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))
    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()

if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```

In [ ]:

```
Output:
Iteration 0: x = 200000
Iteration 1: x = 200000
Iteration 2: x = 200000
Iteration 3: x = 200000
Iteration 4: x = 200000
Iteration 5: x = 200000
Iteration 6: x = 200000
Iteration 7: x = 200000
Iteration 8: x = 200000
Iteration 9: x = 200000
```

Data Structure

Stack

In [ ]:

```
Implementation 1:

Stack works on the principle of “Last-in, first-out”. Also, the inbuilt functions in Py
thon make the code short and simple.
To add an item to the top of the list, i.e., to push an item, we use append() function
and to pop out an element we use pop() function.
These functions work quiet efficiently and fast in end operations.

# Python code to demonstrate Implementing
# stack using list
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack)
```

In [ ]:

```
Implementation 2:

class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)

s=Stack()

print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

Queue

In [ ]:

```
Implementing queue is a bit different. Queue works on the principle of "First-in, first
-out".Due to the properties of list, which is fast at the end operations but
slow at the beginning operations, as all other elements have to be shifted one by one.
So, we prefer the use of collections.deque over list,
which was specially designed to have fast appends and pops from both the front and back
end.
```

In [ ]:

```
Implementation 1:

# Python code to demonstrate Implementing
# Queue using deque and list
from collections import deque
queue = deque(["Ram", "Tarun", "Asif", "John"])
print(queue)
queue.append("Akbar")
print(queue)
queue.append("Birbal")
print(queue)
print(queue.popleft())
print(queue.popleft())
print(queue)
```

In [ ]:

```
Implementation 2:
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Linkedlist

In [ ]:

```
Simple Creation of Node and linkedlist:

# Node class
class Node:
    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize
                          # next as null
# Linked List class
class LinkedList:
    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

In [ ]:

```
# A simple Python program to introduce a linked list

# Node class
class Node:
    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null

# Linked List class contains a Node object
class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None


# Code execution starts here
if __name__=='__main__':
    # Start with the empty list
    llist = LinkedList()
    llist.head  = Node(1)
    second = Node(2)
    third  = Node(3)

    '''
    Three nodes have been created.
    We have references to these three blocks as first,
    second and third

    llist.head        second              third
         |                |                  |
         |                |                  |
    +----+------+     +----+------+     +----+------+
    | 1  | None |     | 2  | None |     |  3 | None |
    +----+------+     +----+------+     +----+------+
    '''

    llist.head.next = second; # Link first node with second

    '''
    Now next of first Node refers to second.  So they
    both are linked.
        llist.head        second              third
         |                |                  |
         |                |                  |
    +----+------+     +----+------+     +----+------+
    | 1  |  o-------->| 2  | null |     |  3 | null |
    +----+------+     +----+------+     +----+------+
    '''

    second.next = third; # Link second node with the third node

    '''
    Now next of second Node refers to third.  So all three
    nodes are linked.

    llist.head        second              third
         |                |                  |
         |                |                  |
```

```
    +----+------+     +----+------+     +----+------+
    | 1  |  o-------->| 2  |  o-------->|  3 | null |
    +----+------+     +----+------+     +----+------+
    '''
```

Transverse

In [ ]:

```
# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null

# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next
# Code execution starts here
if __name__=='__main__':
    # Start with the empty list
    llist = LinkedList()
    llist.head  = Node(1)
    second = Node(2)
    third  = Node(3)
    llist.head.next = second; # Link first node with second
    second.next = third; # Link second node with the third node
    llist.printList()
```

Algorithms

Linear Search

In [ ]:

```
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.
# Python3 code to linearly search x in arr[].
# If x is present then return its location,
# otherwise return -1
Implementation 1:
def search(arr, n, x):
    for i in range (0, n):
        if (arr[i] == x):
            return i;
    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)
```

In [ ]:

```
Implementation 2:
list_ = list(map(int,input().split()))
variable = int(input())
index =  [x for x,y in enumerate(list_) if y == variable]
print(index[0])
```

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.

Binary Search

In [ ]:

```
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, nar
row 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.
```

In [24]:

```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1

    while first<=last:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
                return midpoint+1
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1
    return False
testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 42))
print(binarySearch(testlist, 13))
```

```
9
5
```

the binary search is O(logn).

Sorting

Bubble Sort

In [ ]:

```
Bubble Sort
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the a
djacent elements if they are in wrong order.
Example:
First Pass:
( 5 1 4 2 8 ) –> ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and
swaps since 5 > 1.
( 1 5 4 2 8 ) –>  ( 1 4 5 2 8 ), Swap since 5 > 4
( 1 4 5 2 8 ) –>  ( 1 4 2 5 8 ), Swap since 5 > 2
( 1 4 2 5 8 ) –> ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5
), algorithm does not swap them.

Second Pass:
( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )
( 1 4 2 5 8 ) –> ( 1 2 4 5 8 ), Swap since 4 > 2
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )
( 1 2 4 5 8 ) –>  ( 1 2 4 5 8 )
Now, the array is already sorted, but our algorithm does not know if it is completed.
The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )
```

In [29]:

```python
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

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.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Selection Sort

In [ ]:

```
The selection sort improves on the bubble sort by making only one exchange for every pa
ss through the list. In order to do this, a selection sort looks
for the largest value as it makes a pass and, after completing the pass, places it in t
he proper location.
```

In [ ]:

```
import sys
A = [64, 25, 12, 22, 11]
# Traverse through all array elements
for i in range(len(A)):
    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j
    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]
# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

Time Complexity: O(n2) as there are two nested loops.

In [ ]:

```

```