Linear and binary search

```
from timeit import default_timer
n=[]

for i in range(1,1001):
    n.append(i)

def linear(target):
  start=default_timer()
  for x in n:
      if x==target:
        duration=default_timer()-start
        print("found in",duration,"seconds")

def binary(target):
    start=default_timer()
    l=0
    u=len(n)
    while(l<=u):
       mid=(l+u)//2
       if n[mid]==target:
        duration=default_timer()-start
        print("found in",duration,"seconds")
        break;
       elif n[mid]<target:
          l=mid+1
       else:
          u=mid-1

while True:
    target=int(input("target number: "))
    choice=input("enter 1 for linear or 2 for binary search: ")
    if choice=="1":
        linear(target)
    elif choice=="2":
        binary(target)
```

linear search

```
from timeit import default_timer
pos=-1
def search(list,n):
    i=0
    for i in range(len(list)):
        if list[i]==n:
            globals()["pos"]=i
            return True
    return False
start = default_timer()
list=[3,5,1,7,9]
n=1
if search(list,n):
    print("found at",pos+1)
    duration = default_timer() - start
    print("found in", duration, "seconds")
else:
    print("not found")
```

binary search

```
from timeit import default_timer
pos=-1
def search(list,n):
  l=0
  u=len(list)
  while l<=u:
    mid=(l+u)//2
    if list[mid]==n:
       globals()["pos"]=mid
       return True
    else:
        if list[mid]<n:
            l=mid+1;
        else:
            u=mid-1;
start = default_timer()
list=[1,2,3,4,5,6]
n=2
if search(list,n):
 print("found at",pos)
 duration = default_timer() - start
 print("found in", duration, "seconds")
else:
 print("not found")
```

Bubble sort

```
from timeit import default_timer
def sort(n):
    start=default_timer()
    for i in range(len(n)-1,0,-1):
        for j in range(i):
            if n[j]>n[j+1]:
                temp=n[j]
                n[j]=n[j+1]
                n[j+1]=temp
    duration = default_timer() - start
    print("found in", duration, "seconds")

n=[5,2,4,1,6,7]
sort(n)
print(n)
```

selection sort

```
from timeit import default_timer
def sort(n):
    start=default_timer()
    for i in range(len(n)):
        minpos=i
        for j in range(i,len(n)):
            if n[j]<n[minpos]:
                minpos=j
                temp=n[i]
                n[i]=n[minpos]
                n[minpos]=temp
    duration=default_timer()-start
    print("found in",duration,"seconds")

n=[3,9,6,4,2,1]
sort(n)
print(n)
```

insertion sort

```
from timeit import default_timer
def sort(n):
    start=default_timer()
    for i in range(1,len(n)):
        key=n[i]
        j=i-1
        while j>=0 and key<n[j]:
            n[j+1]=n[j]
            j=j-1
            n[j+1]=key
    duration=default_timer()-start
    print("found in",duration,"seconds")

n=[5,1,7,8,3,2]
sort(n)
print(n)
```

merge sort

```
from timeit import default_timer
def sort(list):
    if len(list)>1:
        left_list=list[:len(list)//2]
        right_list=list[len(list)// 2:]

        sort(left_list)
        sort(right_list)

        i=0
        j=0
        k=0
        while i<len(left_list) and j<len(right_list):
            if left_list[i]<right_list[j]:
               list[k]=left_list[i]
               i=i+1
               k=k+1
            else:
               list[k]=right_list[j]
               j=j+1
               k=k+1
        while i<len(left_list):
               list[k]=left_list[i]
               i=i+1
               k=k+1
        while j<len(right_list):
               list[k]=right_list[j]
               j=j+1
               k=k+1


list_test=[8,4,9,2,6,1]
start=default_timer()
sort(list_test)
duration=default_timer()-start
print("found in",duration,"seconds")
print(list_test)
```

quick sort

```
from timeit import default_timer
def quicksort(l, r, nums):
    if len(nums)==1:
        return nums
    if l<r:
        pi=partition(l,r,nums)
```

```
        quicksort(l,pi - 1,nums)
        quicksort(pi+1,r,nums)
    return nums
def partition(l,r,nums):
    pivot,ptr=nums[r],l
    for i in range(l,r):
        if nums[i]<=pivot:
            nums[i],nums[ptr]=nums[ptr],nums[i]
            ptr+=1
    nums[ptr],nums[r]=nums[r],nums[ptr]
    return ptr


start = default_timer()
nums= [1,3,9,8,2,7,5]
duration = default_timer() - start
print("found in", duration, "seconds")
print(quicksort(0, len(nums) - 1, nums))
```

DLL

```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
        self.prev=None
class Doubly:
    def __init__(self):
        self.head=None
        self.tail=None

    def print1(self,):
        if self.head is None:
            print("Linked is empty")
        else:
            n=self.head
            while n is not None:
                print(n.data,end=" ")
                n=n.next
    def empty(self,data):
        new_node=Node(data)
        if self.head is None:
            self.head=new_node
        else:
            print("Linked is not empty")

    def beg(self,data):
        new_node=Node(data)
        if self.head is None:
            self.head=new_node
        else:
            new_node.next=self.head
            self.head.prev=new_node
            self.head=new_node
    def end(self,data):
        new_node=Node(data)
        if self.head is None:
            self.head=new_node
        else:
            n=self.head
            while n.next is not None:
                n=n.next
            n.next=new_node
            new_node.prev=n

dll=Doubly()
dll.empty(4)
```

```
dll.beg(5)
dll.end(10)
dll.print1()
```

CLL

```
class Node(object):
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None

    def push(self, data, temp=None):
        if self.head == None:
            node = Node(data)
            self.head = node
            node.next = self.head
            return
        if temp == None:
            temp = self.head
        if temp.next == self.head:
            node = Node(data)
            node.next = self.head
            temp.next = node
            return
        self.push(data, temp.next)

    def traverse(self, temp=None):
        if temp == None:
            temp = self.head
        if temp.next == self.head:
            print(temp.data, end="\n")
            return
        print(temp.data, end=" ")
        self.traverse(temp.next)

if __name__ == "__main__":
    clist = CircularLinkedList()
    clist.push(2)
    clist.push(3)
    clist.push(7)
    clist.push(5)
    print(" Circular Linked List: ")
    clist.traverse()
```

stack

```
stack=[]
def push():
    element=input("enter the number")
    stack.append(element)
    print(stack)
def pop_element():
    if not stack:
        print("stack is empty:")
    else:
        e=stack.pop()
        print("removed element:",e)
while True:
    print("select the operation 1.push 2.pop 3.quit")
    choice=int(input())
    if choice==1:
        push()
    elif choice==2:
```

```
        pop_element()
    elif choice==3:
        break
    else:
        print("enter the correct operation ")
```

queue

```
from qqueue import Queue
q=qqueue(maxsize=3)
def qqueue():
    print("queue full ")
print(q.qqsize())
q.put('a')
q.put('b')
q.put('c')
print("\nfull:",q.full())
print("/n element dequeued from the queue ")
print(q.get())
print(q.get())
print(q.get())
n=int(input("limit of queue"))
```

```
queue=[]
def enqueued():
   elements=input("enter the number")
   queue.append(elements)
   print(queue)
def dequeued():
  if not queue:
     print("queue is empty")
  else:
     e=queue.pop()
     print("removed element:",e)
     print(queue)
while True:
 print("select the operation 1.enqueued 2.dequeued 3.quit")
 choice=int(input())
 if choice==1:
    enqueued()
 elif choice==2:
    dequeued()
 elif choice==3:
    break;
 else:
    print("enter the correct operation")
```

factorial

```
import sys
sys.setrecursionlimit(10**6)
def fact(n):
  if n<0 or int(n)!=n:
      return "!not defined"
  if (n==0 or n==1):
     return n
  else:
     return n*fact(n-1)
n=int(input("enter the number:"))
print("factorial of given number:", fact(n))
```

fibonaciii

```
def fib(n):
    if (n<0 or int(n)!=n):
       return "!not defined"
    elif n==0 or n==1:
       return n
    else:
       return fib(n-1) + fib(n-2)

n=int(input("enter the number:\n"))
print("Fibonacci series:",end=" ")
for i  in range(0, n):
   print(fib(i), end=" ")
```

tower of hanoi

```
def t_h(disks, target, source, auxiliary):

    if (disks==1):
        print('move disk 1 from rod{} to rod{}.'.format(source,target))
        return
    t_h(disks-1, target, source, auxiliary)
    print('move disks{} from rod{} to rod{}.'.format(disks, source, target))
    t_h(disks-1, target, source, auxiliary)

disks=int(input("enter the number of disks:"))
t_h(disks,'A','B','C')
```

BFS

```
graph={
'5':['3','7'],
'3':['2','4'],
'7':['8'],
'2':[ ],
'4':['8'],
'8':[ ]
 }
visited=[]
queue=[]
def bfs(visited,graph,node):
   visited.append(node)
   queue.append(node)
   while queue:
      m=queue.pop(0)
      print(m, end="")
      for neighbour in graph[m]:
        if neighbour not in visited:
          visited.append(neighbour)
          queue.append(neighbour)
print("following is the Breadth-First search")
bfs(visited,graph,'5')
```

DFS

```
graph = {
 '5' : ['3','7'],
 '3' : ['2', '4'],
 '7' : ['8'],
 '2' : [],
 '4' : ['8'],
```

```
 '8' : []
}
visited = set()
def dfs(visited, graph, node):
    if node not in visited:
       print (node)
       visited.add(node)
       for neighbour in graph[node]:
        dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

hashing

```
class HashTable:
    def __init__(self,size):
       self.size = size
       self.hash_table = self.create_buckets()

    def create_buckets(self):
        return[[]for _ in range(self.size)]

    def set_Val(self, key, Val):
        hashed_key = hash(key) % self.size
        bucket = self.hash_table[hashed_key]
        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_Val = record
            if record_key == key:
               found_key = True
               break
        if found_key:
            bucket[index] = (key, Val)
        else:
            bucket.append((key, Val))

    def get_Val(self, key):
        hashed_key = hash(key) % self.size
        bucket = self.hash_table[hashed_key]
        found_key = False
        for index, record in enumerate(bucket):
           record_key, record_Val = record
           if record_key == key:
               found_key=True
               break
        if found_key:
            return record_Val
        else:
            return "No record found"

    def delete_Val(self, key):
        hashed_key = hash(key) % self.size
        bucket = self.hash_table[hashed_key]
        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_Val = record
            if record_key == key:
               found_key = True
               break
        if found_key:
            bucket.pop(index)
        return

    def __str__(self):
       return "".join(str(item) for item in self.hash_table)

hash_table = HashTable(10)
```

```
hash_table.set_Val(1, 'mon')
print(hash_table)
hash_table.set_Val(4, 'thur')
print(hash_table)
hash_table.set_Val(7, 'sun')
print(hash_table)
print("search result: ")
print(hash_table.get_Val(1))
hash_table.delete_Val(1)
print("After deleting item from hash table:")
print( hash_table)
```