Linear search

Algorithm

The algorithm for linear search can be specified as follows.

Input to algorithm: A list and an element to be searched.

Output: Index of the element if the element is present. Otherwise,-1.

1. Start from index 0 of the list.
2. Check if the element is present at the current position.
3. If yes, return the current index. Goto 8.
4. Check if the current element is the last element of the list.
5. If yes, return -1. Goto 8. Otherwise, goto 6.
6. Move to the next index of the list.
7. Goto 2.
8. Stop.
9. **Code:**

```
import array as mya
class linear_search:
  def ls(self,a,n,k):
    for i in range(0,n):
      if a[i]==k:
        return ("The element is present at index "+str(i)+" in the list")
    return ("The key element you are searching for is not present in the list")
l1=[]
a=mya.array('b',l1)
lim=int(input("Enter the limit3"))
for i in range(lim):
  el=int(input("Enter the element"))
  a.append(el)
  lim=lim-1
k=int(input("Enter key value"))
n=len(a)
obj=linear_search()
obj.ls(a,n,k)
```

```
Enter the limit4
Enter the element3
Enter the element5
Enter the element7
Enter the element9
Enter key value5
'The element is present at index 1 in the list'
```

Binary search
Algorithm
- We write a function that takes two arguments, the first of which is the list and the second of which is the objective to be found.
- We declare two variables start and end, which point to the list's start (0) and end (length – 1), respectively.
- Since the algorithm would not accept items outside of this range, these two variables are responsible for removing items from the quest.
- The next loop will continue to locate and delete items as long as the start is less than or equal to the end, since the only time the start exceeds the end is if the item is not on the list.
- We find the integer value of the mean of start and end within the loop and use it as the list's middle object.
- **Code:**

```
class bis:
  def bs(self,l1,low,high):
    while low<=high:
      mid=(low+high)//2
      if l1[mid]<k:
        low=mid+1
      elif l1[mid]>k:
        high=mid-1
      else:
        return mid
    return -1
ob1=bis()
l1 = []
lim=int(input("Enter the limit"))
for i in range(lim):
   el=int(input("Enter the element"))
   l1.append(el)
   lim=lim-1
k=int(input("Enter a key value"))
leng=len(l1)
res=ob1.bs(l1,0,leng-1)
if res == -1:
   print("element is not present")
else:
   print("element is at position ",res)
```

```
Enter the limit5
Enter the element9
Enter the element5
Enter the element7
Enter the element2
Enter the element4
Enter a key value6
element is not present
```

Merge sort

Algorithm

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

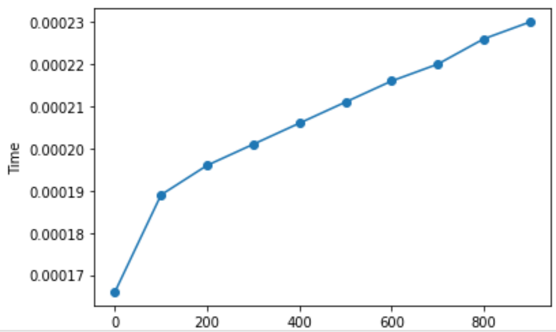
step 4: Stop

Code:

```
import random
import time
import timeit
import matplotlib.pyplot as plt
# start = timeit.default_timer()
def merge(a1,a2):
   c=[]
   x=0
   y=0
   while(x<len(a1) and y<len(a2)):
      if(a1[x]<a2[y]):
         c.append(a1[x])
         x+=1
      else:
         c.append(a2[y])
         y+=1
   while(x<len(a1)):
      c.append(a1[x])
      print(c)
      x+=1
   while(y<len(a2)):
      c.append(a2[y])
      print(c)
      y+=1
   return c
def mergesort(array):
   if(len(array)==1):
      return array
   mid=(len(array))//2
   a1=mergesort(array[:mid])
   a2=mergesort(array[mid:])
   return merge(a1,a2)
```

```
array=[]
x_coordinate = []
y_coordinate = []
start=time.time()
for i in range(0,10):
  n=random.randint(10,5000)
  array.append(n)
  x_coordinate.append(i*100)
  y_coordinate.append(round(time.time()-start,6))
print(mergesort(array))
plt.plot(x_coordinate, y_coordinate, marker="o")
plt.xlabel("Size")
plt.ylabel("Time")
plt.show()
```

```
[3126, 3423]
[1909, 3000]
[1350, 1909]
[1350, 1909, 3000]
[1350, 1909, 3000, 3126]
[1350, 1909, 3000, 3126, 3423]
[1684, 4288]
[1032, 1803]
[1032, 1803, 2855]
[1032, 1684, 1803, 2855, 4288]
[1032, 1350, 1684, 1803, 1909, 2855, 3000, 3126, 3423, 4288]
[1032, 1350, 1684, 1803, 1909, 2855, 3000, 3126, 3423, 4288]
```



Quick Sort

quickSort(arr[], low, high) {

    if (low < high) {

        pi = partition(arr, low, high);

        quickSort(arr, low, pi – 1); // Before pi

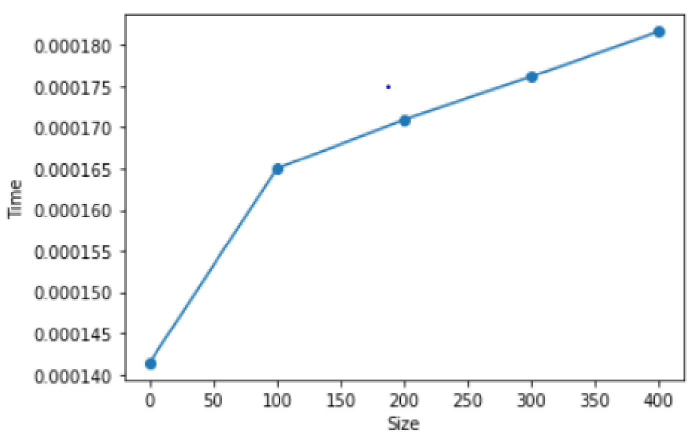
        quickSort(arr, pi + 1, high); // After pi

    }

```
}

partition (arr[], low, high)

{

pivot = arr[high];

 i = (low – 1)

for (j = low; j <= high- 1; j++){

if (arr[j] < pivot){

i++;

 swap arr[i] and arr[j]

     }

 }

    swap arr[i + 1] and arr[high])

return (i + 1)

}
```

```python
import random
def partition(my_arr, start, end):
  pivot = my_arr[end]
  i = start-1
  for j in range(start, end):
    if my_arr[j]<=pivot:
      i=i+1
      my_arr[i], my_arr[j] = my_arr[j], my_arr[i]
    print(my_arr)
  my_arr[i+1], my_arr[end] = my_arr[end], my_arr[i+1]
  return i+1
def quicksort(my_arr, start, end):
  if start<end:
    q = partition(my_arr, start, end)
    quicksort(my_arr, start, q-1)
    quicksort(my_arr, q+1, end)
my_arr = []
x_coordinate = []
y_coordinate = []
start=time.time()
for i in range(0,5):
  n=random.randint(100,500)
  my_arr.append(n)
  x_coordinate.append(i*100)
  y_coordinate.append(round(time.time()-start,8))
quicksort(my_arr, 0, 4)
print(my_arr)
print('Time: ', time.time()- start)
plt.plot(x_coordinate, y_coordinate, marker="o")
plt.xlabel("Size")
plt.ylabel("Time")
plt.show()
```

```
[254, 174, 234, 227, 435]
[254, 174, 234, 227, 435]
[254, 174, 234, 227, 435]
[254, 174, 234, 227, 435]
[254, 174, 234, 227, 435]
[174, 254, 234, 227, 435]
[174, 254, 234, 227, 435]
[174, 227, 234, 254, 435]
[174, 227, 234, 254, 435]
Time:  0.012327909469604492
```



Finding max and min using divide and conquer method

```
DAC(a, i, j)

{

  if(small(a, i, j))

    return(Solution(a, i, j))

  else

    m = divide(a, i, j)           // f1(n)

    b = DAC(a, i, mid)            // T(n/2)

    c = DAC(a, mid+1, j)          // T(n/2)

    d = combine(b, c)             // f2(n)

 return(d)

}
```

Code

```python
import random
def DAC_Max(a, index, l):
    max = -1
    if (index >= l - 2):
        if (a[index] > a[index + 1]):
            return a[index]
        else:
            return a[index + 1];
    max = DAC_Max(a, index + 1, l)
    if (a[index] > max):
        return a[index]
    else:
        return max
def DAC_Min(a, index, l):
    min = 0
    if (index >= l - 2):
        if (a[index] < a[index + 1]):
            return a[index]
        else:
            return a[index + 1]
    min = DAC_Min(a, index + 1, l)
    if (a[index] < min):
        return a[index]
    else:
        return min;
a=[]
for i in range(0,10000):
  n=random.randint(0,100000)
  a.append(n)
l1=len(a)-1
max = DAC_Max(a, 0, 11)
min = DAC_Min(a, 0, 11)
print("The minimum number in a given array is : ", min);
print("The maximum number in a given array is : ", max);
```

```
The minimum number in a given array is :  680
The maximum number in a given array is :  84864
```

Kruskal's algorithm

- Sort all the edges of the graph from low weight to high.
- Take the edge of the lowest weight and add it to the required spanning tree. If adding this edge creates a cycle in the graph, then reject this edge.
- Repeat this process until all the vertices are covered with the edges.

Code:

```
[ ] from collections import defaultdict
    class Graph:
        def __init__(self, vertices):
            self.V = vertices
            self.graph = []
        def addEdge(self, u, v, w):
            self.graph.append([u, v, w])
        def find(self, parent, i):
            if parent[i] == i:
                return i
            return self.find(parent, parent[i])
        def union(self, parent, rank, x, y):
            xroot = self.find(parent, x)
            yroot = self.find(parent, y)
            if rank[xroot] < rank[yroot]:
                parent[xroot] = yroot
            elif rank[xroot] > rank[yroot]:
                parent[yroot] = xroot
            else:
                parent[yroot] = xroot
                rank[xroot] += 1
        def KruskalMST(self):
            result = []
            i = 0
            e = 0
            self.graph = sorted(self.graph,
                                key=lambda item: item[2])
            parent = []
            rank = []
            for node in range(self.V):
                parent.append(node)
                rank.append(0)
```

```
[ ]         while e < self.V - 1:
                u, v, w = self.graph[i]
                i = i + 1
                x = self.find(parent, u)
                y = self.find(parent, v)
                if x != y:
                    e = e + 1
                    result.append([u, v, w])
                    self.union(parent, rank, x, y)
            minimumCost = 0
            print ("Edges in the constructed MST")
            for u, v, weight in result:
                minimumCost += weight
                print("%d - %d = %d" % (u, v, weight))
            print("Minimum Spanning Tree" , minimumCost)
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)
    g.KruskalMST()
```

```
Edges in the constructed MST
2 - 3 = 4
0 - 3 = 5
0 - 1 = 10
Minimum Spanning Tree 19
```

Prims algorithm

- Take any vertex as the source and set its weight to 0. Set the weights of all other vertices to infinity.
- For every adjacent vertices, if the current weight is more than that of the current edge, then we replace it with the weight of the current edge.
- Then, we mark the current vertex as visited.
- Repeat these steps for all the given vertices in ascending order of weight.

Code:

```
def primsAlgorithm(vertices):
    adjacencyMatrix = [[0 for column in range(vertices)]
                       for row in range(vertices)]
    mstMatrix = [[0 for column in range(vertices)]
                       for row in range(vertices)]
    for i in range(0,vertices):
        for j in range(0+i,vertices):
            adjacencyMatrix[i][j] = int(input('Enter the path weight between the vertices: '))
            adjacencyMatrix[j][i] = adjacencyMatrix[i][j]
    positiveInf = float('inf')
    selectedVertices = [False for vertex in range(vertices)]
    while(False in selectedVertices):
        minimum = positiveInf
        start = 0
        end = 0
        for i in range(0,vertices):
            if selectedVertices[i]:
                for j in range(0+i,vertices):
                    if (not selectedVertices[j] and adjacencyMatrix[i][j]>0):
                        if adjacencyMatrix[i][j] < minimum:
                            minimum = adjacencyMatrix[i][j]
                            start, end = i, j
        selectedVertices[end] = True
        mstMatrix[start][end] = minimum

        if minimum == positiveInf:
            mstMatrix[start][end] = 0

        mstMatrix[end][start] = mstMatrix[start][end]
    print(mstMatrix)
primsAlgorithm(int(input('Enter the vertices number: ')))
```

```
Enter the vertices number: 3
Enter the path weight between the vertices: 0
Enter the path weight between the vertices: 2
Enter the path weight between the vertices: 9
Enter the path weight between the vertices: 5
Enter the path weight between the vertices: 11
Enter the path weight between the vertices: 7
[[0, 2, 9], [2, 0, 0], [9, 0, 0]]
```

job sequencing with deadlines

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

D(0) := J(0) := 0

k := 1

J(1) := 1   // means first job is selected

for i = 2 … n do

  r := k

  while D(J(r)) > D(i) and D(J(r)) ≠ r do

    r := r – 1

  if D(J(r)) ≤ D(i) and D(i) > r then

    for l = k … r + 1 by -1 do

      J(l + 1) := J(l)

      J(r + 1) := i

      k := k + 1

Code:

```
[ ] def jswd(arr,t):
      n=len(arr)
      for i in range(n):
        for j in range(n-1-i):
            if(arr[j][2]<arr[j+1][2]):
                arr[j],arr[j+1]=arr[j+1],arr[j]
      result = [False]*t
      job = ['-1']*t
      for i in range(len(arr)):
          for j in range(min(t-1,arr[i][1]-1),-1,-1):
            if(result[j] is False):
              result[j] = True
              job[j] = arr[i][0]
              break
      print(job)
    arr = [['J3', 20, 2], ['J1', 60, 3], ['J2', 100, 1],  ['J4', 40, 2], ['J5', 20, 1]]
    print("Following is the maximum profit sequence of jobs")
    jswd(arr,3)
```

```
Following is the maximum profit sequence of jobs
['J4', 'J3', 'J1']
```

All pairs shortest path floyd's algorithm

```
Algorithm FLOYD_APSP ( L)
// L is the matrix of size n  n representing original graph
// D is the distance matrix

D ← L
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j]^k ← min ( D[i, j]^k-1, D[i, k]^k-1 + D[k, j]^k-1 )
        end
    end
end
return D
```

Code:

```python
V = 4
INF = 99999
def floydWarshall(graph):
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
    for k in range(V):
        for i in range(V):
            for j in range(V):
                dist[i][j] = min(dist[i][j],
                                 dist[i][k] + dist[k][j]
                                 )
    printSolution(dist)
def printSolution(dist):
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print ("%7s" % ("INF"),end=" ")
            else:
                print ("%7d\t" % (dist[i][j]),end=' ')
            if j == V-1:
                print ()
graph =  [
          [  0,   1, 3, INF],
          [  1,   0, 1, INF],
          [  3,   1, 0,   2],
          [INF, INF, 2,   0]
          ]
floydWarshall(graph)
```

```
      0        1                2                4
      1        0                1                3
      2        1                0                2
      4        3                2                0
```

Dijkstra's algorithm

- We will receive a weighted graph and an initial node.
- Start with the initial node. Check the adjacent nodes.
- Find the node with the minimum edge value.
- Repeat this process until the destination node is visited.
- At the end of the function, we return the shortest path weight for each node and the path as well.

Code:

```
[ ]
import sys
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])
    def minDistance(self, dist, sptSet):
        min = sys.maxsize
        for u in range(self.V):
            if dist[u] < min and sptSet[u] == False:
                min = dist[u]
                min_index = u
        return min_index
    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V
        for cout in range(self.V):
            x = self.minDistance(dist, sptSet)
            sptSet[x] = True
            for y in range(self.V):
                if self.graph[x][y] > 0 and sptSet[y] == False and \
                dist[y] > dist[x] + self.graph[x][y]:
                        dist[y] = dist[x] + self.graph[x][y]
        self.printSolution(dist)
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]]
g.dijkstra(0);
```

```
Vertex  Distance from Source
0        0
1        4
2        12
3        19
4        21
5        11
6        9
7        8
8        14
```

Sum of subsets

```
initialize Col with empty subset
loop over all elements x in S
    loop over all subsets, L, that are already in Col
        if sum(L) + x = t, take this subset as result and break
        else if sum(L) + x < t, add the subset L + x to Col
of the subsets that are in Col, find the subset L' with the largest sum(L')
```

Code:

```python
def printAllSubsetsRec(arr, n, v, sum) :
    if (sum == 0) :
        for value in v :
            print(value, end=" ")
        print()
        return
    #print("No sub sets found")
    if (n == 0):
        return
    printAllSubsetsRec(arr, n - 1, v, sum)
    v1 = [] + v
    v1.append(arr[n - 1])
    printAllSubsetsRec(arr, n - 1, v1, sum - arr[n - 1])
def printAllSubsets(arr, n, sum):
    v = []
    printAllSubsetsRec(arr, n, v, sum)
arr = [1,2,3,4,5,6]
sum = 9
n = len(arr)
printAllSubsets(arr, n, sum)
```

```
4 3 2
5 3 1
5 4
6 2 1
6 3
```

BFS

1. Start with a root node and push it to the queue.
2. Mark the root node as visited and print it
3. Continue a loop until the queue is empty:
 3.1. Pop the front node from the queue
 3.2. Push the child/neighbor nodes of the front node to the queue
 3.3 Mark them as visited and print

Code:

```
[ ]  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')

     Following is the Breadth-First Search
     5 3 7 2 4 8
```