# SEARCHING AND SORTING ALGORITHMS

**EXP NO 1** **LINEAR SEARCH IMPLEMENTATION**
**DATE : 06-02-23**

## AIM:

To implement Linear Search and determine the time required to search for an element. To repeat the experiment for different values of n, the number of elements in the list to be searched and to plot a graph of the time taken versus n.

## ALGORITHM:

1) Read the number of elements of the list as n.
2) Generate the n array elements using random.randint() function.
3) Initialize two lists
    - x : for storing the n values
    - t : for storing the time taken values corresponding to the respective n values
4) Read the key element.
5) Start the timer.
6) Call the function linear(list,key) to perform linear search on the key element and store the return value in a variable called pos.
7) End the timer after the function call.
8) IF pos = -1
    PRINT "Item is not in the list"
  ELSE
    PRINT the pos value
9) Compute time_taken = end time - start time and add to the t list.
10) Add the value of n to the x list.
11) Ask for choice 1. Continue 2. Exit
12) IF choice = 1 then
    Repeat steps 1 to 11
  ELSE
    End
13) Plot a graph for x_list vs t_list which

linear(list,key)

1) SET I = 0
1) Repeat steps 1,2,3 until I < length(list)
2) IF list[I] = key then
    return I + 1
3) SET I = I + 1
4)Return -1

## PROGRAM:

```
def linear(item, a):
```

```
    for i in range(len(a)):
        time.sleep(0.01)
        if a[i] == item:
            return i + 1
    return -1
n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 999)) for i in range(n)]
print("Elements are: ")
print(*a)
t=[]
x=[]
x.append(n)

while True:
    item = int(input("Enter the key element: "))
    start = time.time()
    pos = linear(item,a)
    end = time.time()
    if pos == -1:
        print("Item not found")
    else:
        print("Item found at the position", pos)
    time_taken = end - start
    t.append(time_taken)
    print("Time taken =", time_taken)
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the array size: "))
        x.append(n)
        a = [int(random.randint(0, 999999)) for i in range(n)]
        print("Elements are: ")
        print(*a)

    else:
        break

print("The time taken",t)
print("The n values",x)
plt.figure(figsize=(9,7))
plt.plot(xx,tt,marker='o',markersize=8)
plt.title('Linear search time complexity')
plt.xlim([0,70])
plt.ylim([0,1.2])
```

**OUTPUT:**

```
Enter the number of elements of the list: 0
```

Elements are:

Enter the key element: 1
Item not found
Time taken = 0.0

1. Continue
2. Exit
Enter your option: 1

Enter the array size: 10
Elements are:
85 113 399 608 42 824 667 67 663 247
Enter the key element: 345355
Item not found
Time taken = 0.16099143028259277

1. Continue
2. Exit
Enter your option: 1

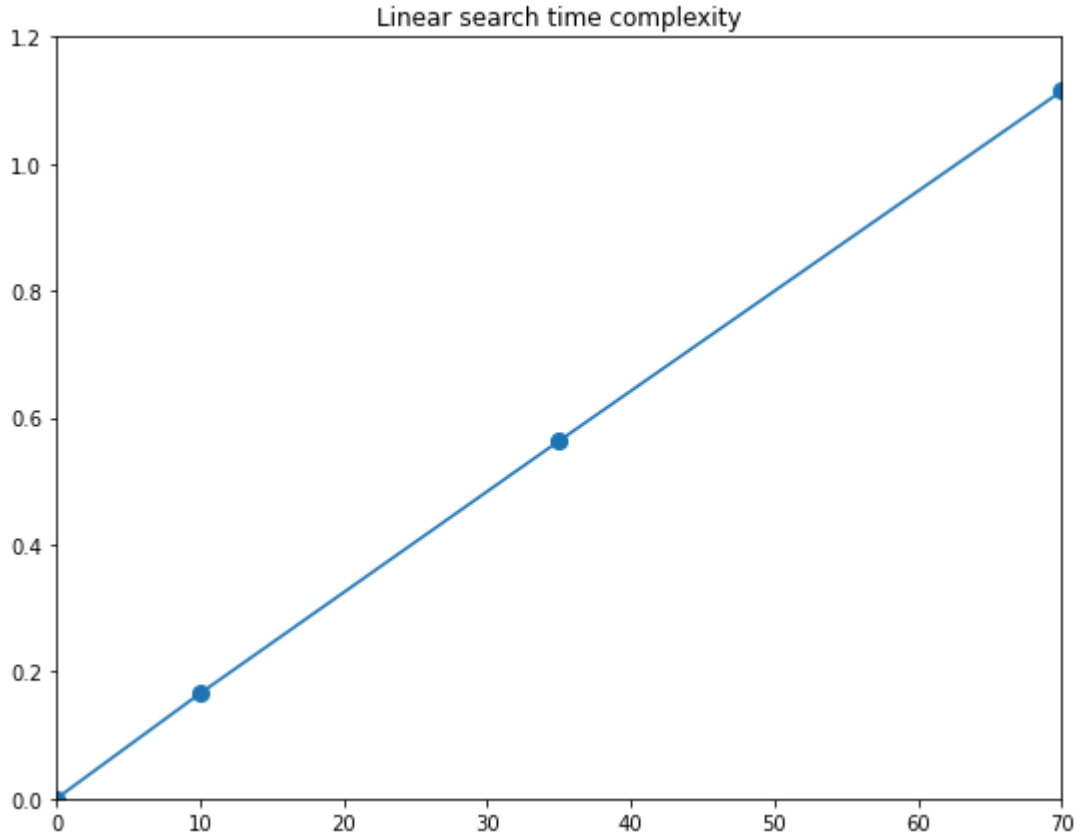
Enter the array size: 35
Elements are:
136 0 636 791 554 598 175 109 476 976 82 711 291 65 656 576 281 886 578 778 571 642 636 170 194 287 853 522 915 773 893 602 203 86 656
Enter the key element: 2121
Item not found
Time taken = 0.5872187614440918

1. Continue
2. Exit
Enter your option: 1

Enter the array size: 70
Elements are:
613 510 770 30 867 230 892 37 700 284 168 80 45 656 98 900 354 152 945 521 8 250 16 23 64 189 43 35 780 527 624 866 385 258 42 730 831 790 451 530 604 218 794 796 160 584 0 829 472 380 41 752 274 896 937 564 478 844 309 765 372 832 902 906 311 449 620 346 612 820
Enter the key element: 1333
Item not found
Time taken = 1.1071064472198486

1. Continue
2. Exit
Enter your option: 2

The time taken [0.0, 0.16550683975219727, 0.5632953643798828, 1.1143748760223389]
The n values [0, 10, 35, 70]




**RESULT:**

Hence successfully implemented Linear Search and determined the time required to search for an element and repeated the experiment for different values of n, the number of elements in the list to be searched and plotted a graph of the time taken versus n.

**EXP NO 2 RECURSIVE BINARY SEARCH IMPLEMENTATION**
**DATE : 06-02-23**

## AIM:

To implement Recursive Binary Search and determine the time required to search for an element. To repeat the experiment for different values of n, the number of elements in the list to be searched and to plot a graph of the time taken versus n.

## ALGORITHM:

1) Read the number of elements of the list as n.
2) Generate the n array elements using random.randint() function.
3) Initialize two lists
   - x : for storing the n values
   - t : for storing the time taken values corresponding to the respective n values
4) Read the key element.
6) Sort the list.
5) Start the timer.
6) Call the function recursive_binary_search(list,key,0,n-1) to perform linear search on the key element and store the return value in a variable called pos.
7) End the timer after the function call.
8) IF pos = -1
   PRINT "Item is not in the list"
   ELSE
   PRINT the pos value
9) Compute time_taken = end time - start time and add to the t list.
10) Add the value of n to the x list.
11) Ask for choice 1. Continue 2. Exit
12) IF choice = 1 then
   Repeat steps 1 to 11
   ELSE
   Go to step 13
13) Plot a graph for x_list vs t_list
14) End

recursive _binary_search(list,key,start,end)

1) IF start > end then
   return -1
2) Compute mid as mid = (start + end)/2
3) IF list[mid] = key
   return mid + 1
   ELSE IF list[mid] < key:
   return recursive_binary_search(arr, target, mid + 1, end)
   ELSE

```
        return recursive_binary_search(arr, target, start, mid - 1)
```

**PROGRAM:**

```
import time
import random
def recursive_binary_search(arr, target,start,end):

    if start > end:
        return -1

    mid = (start + end) // 2
    time.sleep(0.01)
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return recursive_binary_search(arr, target, mid + 1, end)
    else:
        return recursive_binary_search(arr, target, start, mid - 1)

def sort(a):
    n = len(a)
    for i in range(n-1):
        for j in range(i+1, n):
            if a[i] > a[j]:
                a[i], a[j] = a[j], a[i]

n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 999)) for i in range(n)]
print("Elements are: ")
print(*a)
t=[]
x=[]
x.append(n)
sort(a)
print("Sorted list: ")
print(*a)
while True:
    item = int(input("Enter the key element: "))
    start = time.time()
    pos = recursive_binary_search(a,item,0,len(a)-1)
    end = time.time()
    if pos == -1:
        print("Item not found")
    else:
        print("Item found at the position", pos)
    time_taken = end - start
    t.append(time_taken)
    print("Time taken =", time_taken)
```

```
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the array size: "))
        x.append(n)
        a = [int(random.randint(0, 999)) for i in range(n)]
        print("Elements are: ")
        print(*a)
        sort(a)
        print("Sorted list: ")
        print(*a)
    else:
        break
print("The time taken ",t)
print('The n values ',x)
import matplotlib.pyplot as plt
plt.figure(figsize=(9,7))
plt.plot(x,t,marker='o',markersize=8)
plt.title('Binary search time complexity')
plt.xlim([0,70])
plt.ylim([0,1.2])
```

**OUTPUT:**

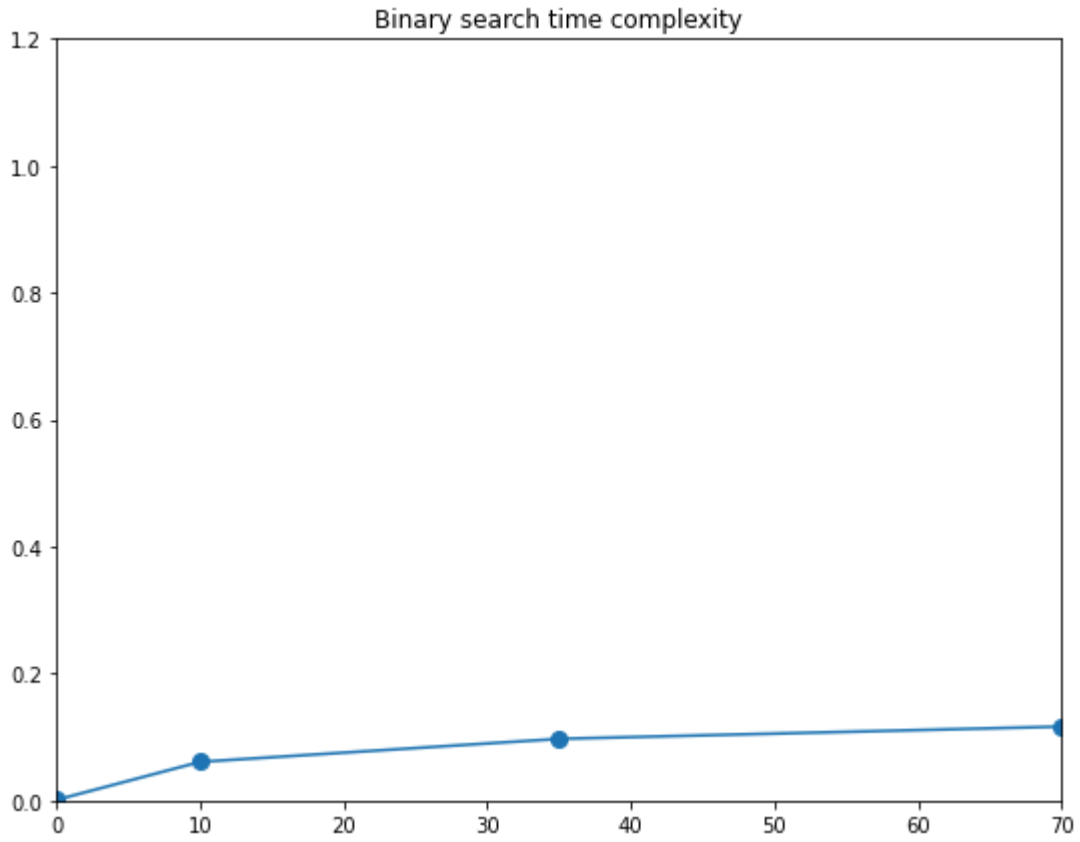
```
Enter the number of elements of the list: 0
Elements are:

Sorted list:

Enter the key element: 12
Item not found
Time taken = 0.0009992122650146484

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 10
Elements are:
260 656 845 639 546 17 74 998 267 671
Sorted list:
17 74 260 267 546 639 656 671 845 998
Enter the key element: 1233
Item not found
Time taken = 0.06089186668395996

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 35
Elements are:
```

316 145 659 918 133 909 939 88 285 19 206 863 370 326 295 722 797 166 859 117 158 818 441 946 826 311 386 869 220 668 860 794 45 353 412
Sorted list:
19 45 88 117 133 145 158 166 206 220 285 295 311 316 326 353 370 386 412 441 659 668 722 794 797 818 826 859 860 863 869 909 918 939 946
Enter the key element: 1233
Item not found
Time taken = 0.09713506698608398

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 70
Elements are:
883 887 706 254 340 520 478 277 177 331 920 217 643 798 643 871 903 830 12 188 105 697 635 430 782 299 774 252 159 948 777 526 146 16 291 824 510 737 955 108 322 88 710 357 892 586 650 842 497 581 292 698 680 13 990 994 655 871 275 801 706 498 534 260 13 728 961 70 252 101
Sorted list:
12 13 13 16 70 88 101 105 108 146 159 177 188 217 252 252 254 260 275 277 291 292 299 322 331 340 357 430 478 497 498 510 520 526 534 581 586 635 643 643 650 655 680 697 698 706 706 710 728 737 774 777 782 798 801 824 830 842 871 871 883 887 892 903 920 948 955 961 990 994
Enter the key element: 1222
Item not found
Time taken = 0.1164863109588623

1. Continue
2. Exit
Enter your option: 2

The time taken  [0.0009992122650146484, 0.06089186668395996, 0.09713506698608398, 0.1164863109588623]
The n values  [0, 10, 35, 70]




**RESULT:**

Hence successfully implemented Recursive Binary Search and determined the time required to search for an element and repeated the experiment for different values of n, the number of elements in the list to be searched and plotted a graph of the time taken versus n.

## EXP NO 3 PATTERN MATCHING
## DATE : 09-02-23

**AIM:**

To develop a program to find the number of occurrences of a pattern in the given text.

**ALGORITHM:**

1) Read the text as t.
2) Read the pattern as p.
3) Store the lengths of text and pattern into the variables n and m respectively.
4) SET J = 0, count = 0.
5) Repeat steps 6,7 until J < n - m
6) IF p[0…m] == t[J….J+m]
   THEN SET count = count + 1
7) SET J = J+1
8) PRINT count
9) END

**PROGRAM:**

```
t=list(input("Enter the text:"))
p=list(input("Enter the pattern:"))

m = len(p)
n = len(t)
c=0

for i in range(0,n-m):
    if p[:m]==t[i:i+m]:
        c+=1

print("Pattern occurs",c,"times in the given text")
```

**OUTPUT:**

```
Enter the text:aaaaaaaaaa
Enter the pattern:aa
Pattern occurs 8 times in the given text
```

**RESULT:**

Hence, successfully implemented a pattern matching algorithm.

## EXP NO 4 INSERTION SORT AND HEAP SORT IMPLEMENTATION
## DATE : 13-02-23

### AIM:

To sort a given set of elements using the Insertion sort and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n

### INSERTION SORT ALGORITHM:

1) Read the number of elements of the list as n.
2) Generate the n array elements using random.randint() function.
3) Initialize two lists
   - r : for storing the n values
   - s : for storing the time taken values corresponding to the respective n values
4) Start the timer.
5) Call the function insertion_sort(list) to perform quick sort on the list
6) End the timer after the function call.
7) PRINT the sorted list
8) Compute time_taken = end time - start time and add to the s list.
9) Add the value of n to the r list.
10) Ask for choice 1. Continue 2. Exit
11) IF choice = 1 then
        Repeat steps 1 to 11
    ELSE
        Go to step 12
12) Plot a graph for r_list vs s_list
13) End

insertion_sort(list)

1) Read the element one by one from the list.
2) Insert the element into its correct position in its leftmost subarray.
3) Repeat steps 1,2 until the last element is inserted in its correct position.

### PROGRAM:

```
import random
import time

def insertion_sort(a):
    for i in range(1, len(a)):
        item = a[i]
        j = i - 1
```

```
        while j >= 0 and item < a[j]:
            a[j+1] = a[j]
            j -= 1
        a[j+1] = item
n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 999)) for i in range(n)]

s=[]
r=[]
r.append(n)
while True:
    start = time.time()
    insertion_sort(a)
    end = time.time()
    time_taken = end - start
    s.append(time_taken)
    print(f"Time taken ={time_taken:.6f}")
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the number of elements of the list: "))
        r.append(n)
        a = [int(random.randint(0, 9999)) for i in range(n)]
    else:
        break

print("The n values",r)
print("The time taken",s)
plt.title("Insertion sort")
plt.plot(r,s)
```

**OUTPUT:**

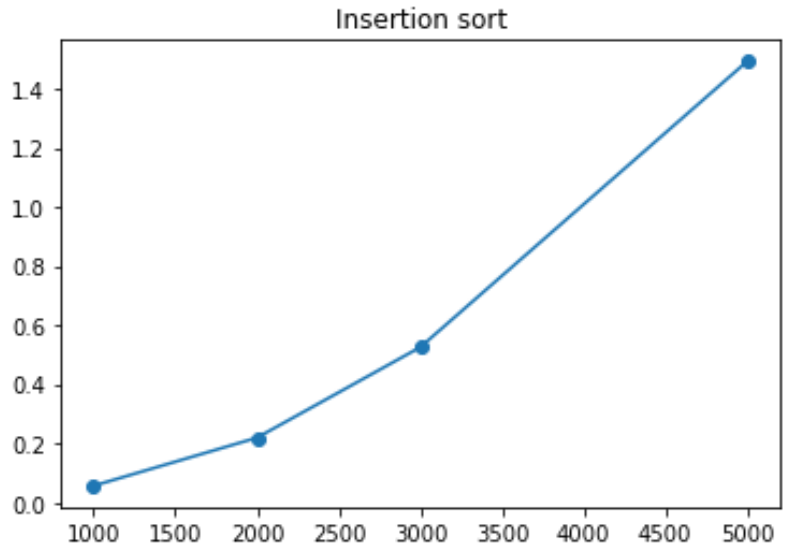
```
Enter the number of elements of the list: 1000
Time taken =0.073828

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 2000
Time taken =0.291579

1. Continue
2. Exit
Enter your option: 1
```

Enter the number of elements of the list: 3000
Time taken =0.703787

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 5000
Time taken =1.844883

1. Continue
2. Exit
Enter your option: 2

The n values [1000, 2000, 3000, 5000]
The time taken [0.07382798194885254, 0.2915792465209961, 0.703787088394165, 1.8448834419250488]




**HEAP SORT ALGORITHM:**

1. Read the number of elements as n.
2. Initialize two lists
    - x list : for storing n values.
    - t list : for storing time taken.
3. Construct a **Binary Tree** with a given list of Elements.
4. Transform the Binary Tree into **Min Heap.**
5. Delete the root element from Min Heap using **Heapify** method.

6. Put the deleted element into the Sorted list.
7. Repeat the same until Min Heap becomes empty.
8. Repeat for different values of n and note the time taken.
9. Plot a graph for x list vs t list.
10. End

**PROGRAM:**

```
import time
import random
import matplotlib.pyplot as plt

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[i] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

# Function to measure the time required to sort elements using heap sort
def measure_heap_sort_time(a):
    # Generate a random list of n elements
    start_time = time.time()
    heapSort(a)
    end_time = time.time()
    return end_time - start_time
```

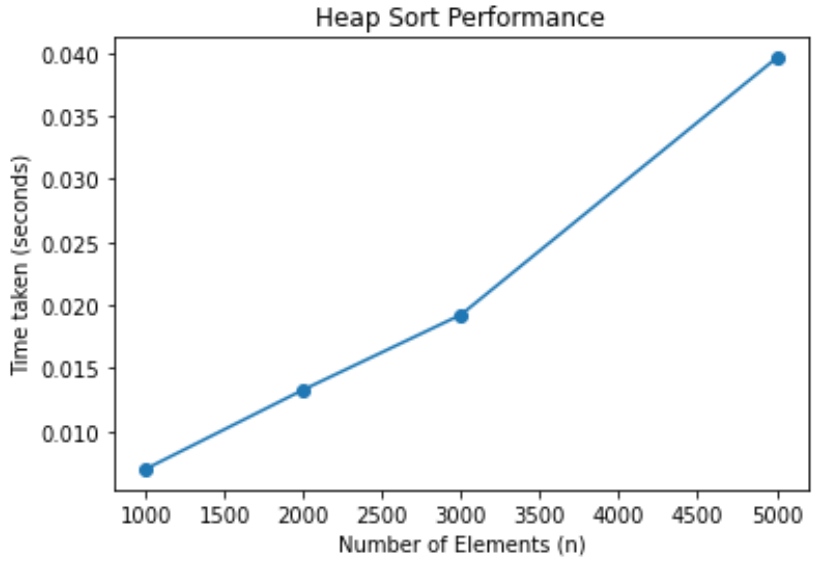
```
# Main code
n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 999)) for i in range(n)]

t=[]
x=[]
x.append(n)

while True:
    time_taken=measure_heap_sort_time(a)
    t.append(time_taken)
    print(f"Time taken ={time_taken:.6f}")
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the number of elements of the list: "))
        x.append(n)
        a = [int(random.randint(0, 9999)) for i in range(n)]
    else:
        break


plt.plot(x, t, marker='o')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time taken (seconds)')
plt.title('Heap Sort Performance')
plt.show()
```

**OUTPUT:**

```
Enter the number of elements of the list: 1000
Time taken =0.007022

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 2000
Time taken =0.013304

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 3000
Time taken =0.019247
```

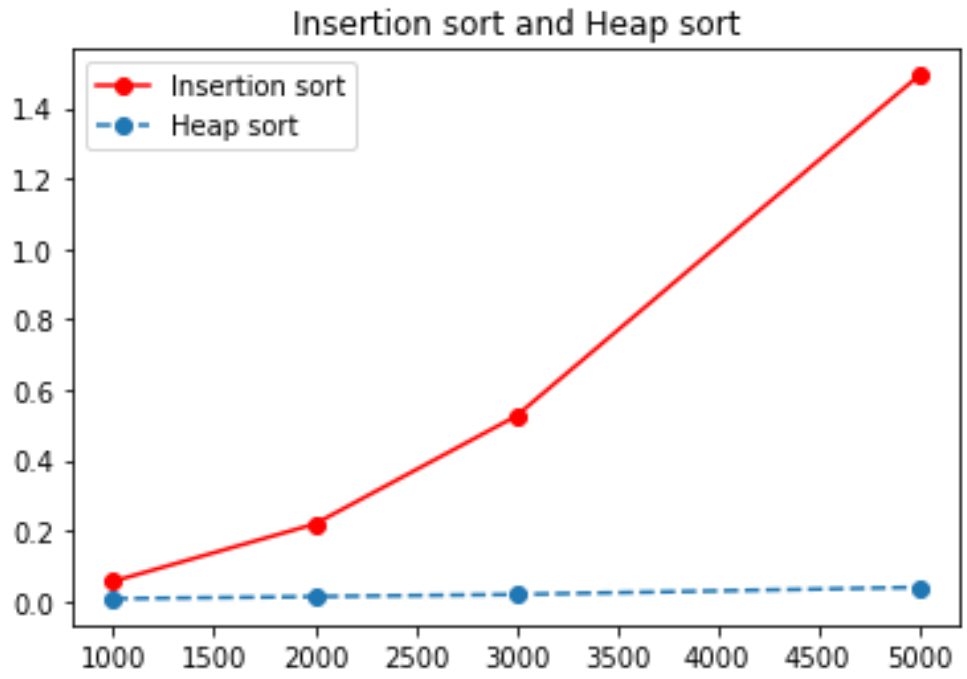
1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 5000
Time taken =0.039577

1. Continue
2. Exit
Enter your option: 2




**INSERTION SORT AND HEAP SORT GRAPH:**

```
plt.title("Insertion sort and Heap sort")
plt.plot(r,s,'r',marker='o')
plt.plot(x,t,'--',marker='o')
plt.legend(['Insertion sort','Heap sort'])
```




**RESULT:**

Hence, successfully implemented Insertion sort and Heap sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

# GRAPH ALGORITHMS

## EXP NO: 1 GRAPH TRAVERSAL TECHNIQUES

## DATE: 20-02-23 BREADTH FIRST SEARCH

**AIM:**

To implement the Breadth First search graph traversal strategy to find the goal state in a state space tree.

**ALGORITHM:**

- **Step 1:** Consider the graph you want to navigate.
- **Step 2:** Select any vertex in your graph (say **v1**), from which you want to traverse the graph.
- **Step 3:** Utilize the following two data structures for traversing the graph.
- Visited array(size of the graph)
- Queue data structure
- **Step 4:** Add the starting vertex to the visited array, and afterward, you add v1's adjacent vertices to the queue data structure.
- **Step 5:** Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.
- **Step 6:** Repeat step 5 until the queue is not empty and no vertex is left to be visited.

**PROGRAM:**

```
graph = {
   'A': ['B', 'C', 'D'],
   'B': ['A'],
   'C': ['A', 'D'],
   'D': ['A', 'C', 'E'],
   'E': ['D'],
}


def bfs(node):

     visited = [False] * (len(graph))

      queue = []

   visited.append(node)
   queue.append(node)

   while queue:
           v = queue.pop(0)
           print(v, end=" ")
```

```
            for neigh in graph[v]:
                if neigh not in visited:
                        visited.append(neigh)
                        queue.append(neigh)


    bfs('A')
```

**OUTPUT**:

A B C D E

**RESULT:**

Hence, successfully implemented graph traversal using bfs.

## EXP NO: 2 DEPTH FIRST SEARCH
## DATE: 20-02-23

**AIM:**

To implement depth first search.

**ALGORITHM:**

**Step 1**: Consider the graph you want to navigate.

**Step 2**: Select any vertex in our graph, say v1,from which you want to begin traversing the graph.

**Step 3**: Examine any two data structures for traversing the graph.

Visited array(size of the graph)

Stack data structure

**Step 4**: Insert v1 into the array's first block and push all the adjacent nodes or vertices of vertex v1 into the stack.

**Step 5**: Now, using the FIFO principle , pop the topmost element and put it into the visited array, pushing all of the popped element's nearby nodes into it.

**Step 6**: If the topmost element of the stack is already present in the array, discard it instead of inserting it into the visited array.

**Step 7**:Repeat step 6 until the stack data structure isn't empty.

**PROGRAM:**

```
def dfs(node, graph, visited, component):
   component.append(node)  # Store answer
   visited[node] = True  # Mark visited

     for child in graph[node]:
       if not visited[child]:
           dfs(child, graph, visited, component)


     graph = {
       0: [2],
       1: [2, 3],
       2: [0, 1, 4],
       3: [1, 4],
       4: [2, 3]
   }
   node = 0
   visited = [False]*len(graph)  # Make all nodes to False initially
   component = []
```

```
    dfs(node, graph, visited, component)
    print(f"Following is the Depth-first search: {component}")
```

**OUTPUT**:

Following is the Depth-first search: [0, 2, 1, 3, 4]

**RESULT:**

Hence, successfully implemented graph traversal using dfs.

## EXP NO 3 DIJIKSTRA'S ALGORITHM

## DATE 20-03-23

### AIM:

To develop a program to find the shortest paths from a given vertex to other vertices using Dijkstra's algorithm.

### ALGORITHM:

1. Mark all nodes as unvisited.
2. Mark the picked starting node with a current distance of 0 and the rest nodes with infinity.
3. Now, fix the starting node as the current node.
4. For the current node, analyze all of its unvisited neighbors and measure their distances by adding the current distance of the current node to the weight of the edge that connects the neighbor node and current node.
5. Compare the recently measured distance with the current distance assigned to the neighboring node and if it is smaller then make it as the new current distance of the neighboring node otherwise do nothing.
6. After that, consider all of the unvisited neighbors of the current node, mark the current node as visited.
7. If the destination node has been marked visited then stops, the algorithm ends.
8. Else, choose the unvisited node that is marked with the least distance, fix it as the new current node, and repeat the process again from step 4.

### PROGRAM:

```
# takes the graph and the starting node
# returns a list of distances from the starting node to every other node
from numpy import Inf
def dijkstra(graph, start):
    l = len(graph)
    # initialize all node distances as infinite
    dist = [Inf for x in range(l)]

    # set the distance of starting node as 0
    dist[start] = 0

    # create a list that indicates if a node is visited or not
    vis = [False for x in range(l)]
```

```
    # iterate over all the nodes
    for i in range(l):

        # set u=-1 to indicate a current starting node
        u = -1

        for x in range(l):

            if not vis[x] and (u == -1 or dist[x] < dist[u]):
                u = x

        if dist[u] == Inf:
            break

        vis[u] = True

        for v, d in graph[u]:
            if dist[u] + d < dist[v]:
                dist[v] = dist[u] + d

     return dist

graph = {
    0: [(1, 1)],
    1: [(0, 1), (2, 2), (3, 3)],
    2: [(1, 2), (3, 1), (4, 5)],
    3: [(1, 3), (2, 1), (4, 1)],
    4: [(2, 5), (3, 1)]
}
print(dijkstra(graph,0))
```

**OUTPUT:**

```
[0, 1, 3, 4, 5]
```

**RESULT:**

Hence, successfully implemented Dijikstra's algorithm to find the shortest path from a given node to all the other nodes in the graph.

## EXP NO 4 PRIM'S ALGORITHM
## DATE : 27-03-23

**AIM:**

To develop a program to find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

**ALGORITHM:**

1. Mark all the vertices as unvisited.
2. Mark the starting vertex alone as visited with a current distance of 0 while others as infinity.
3. Let V be the number of vertices and initially no_of_edges be 0.
4. For the current node, analyze all of its unvisited neighbors and measure their distances which is the weight of the edge that connects the neighbor node and current node.
5. Compare the recently measured distance with the current distance assigned to the neighboring node and if it is smaller then make it as the new current distance of the neighboring node otherwise do nothing.
6. After that mark the current node as visited and print.
7. Set no_of_edges = no_of_edges + 1
8. Repeat steps 4 to 7 until the no_of_edges becomes V-1.

**PROGRAM:**

```
INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
    [19, 0, 5, 9, 2],
    [5, 5, 0, 1, 6],
    [0, 9, 1, 0, 1],
    [0, 2, 6, 1, 0]]

selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True
```

```
# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):

    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

**OUTPUT:**

```
Edge    : Weight

0 - 2    :    5
2 - 3    :    1
3 - 4    :    1
4 - 1    :    2
```

**RESULT:**

Hence, successfully implemented the Prim's algorithm to find the minimum spanning tree of the given graph.

# EXP NO 5 FLOYD'S ALGORITHM FOR ALL-PAIR-SHORTEST
# DATE : 30-03-23 PATHS PROBLEM

## AIM:

To develop a program to implement Floyd's algorithm for the all-pair shortest path problem.

## ALGORITHM:

**Step 1:** Initialize the shortest paths between any 2 vertices with their edge weights otherwise with Infinity.

**Step 2:** Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.

**Step 3:** Minimize the shortest paths between any 2 pairs in the previous operation.

**Step 4:** For any 2 vertices (i,j) ,minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices

dist[k][j] represents the shortest path between the pair k,j

As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

**Step 5:** This process continues until we use all N vertices as intermediate nodes.

**Step 6:** After that , the algorithm ends.

## PROGRAM:

```
nV = 4

INF = 999
print("The all-pair shortest paths")
# Algorithm implementation
def floyd_warshall(G):
    distance = list(map(lambda i: list(map(lambda j: j, i)), G))

    # Adding vertices individually
    for k in range(nV):
        for i in range(nV):
            for j in range(nV):
                distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])
    print_solution(distance)
```

```
# Printing the solution
def print_solution(distance):
    for i in range(nV):
        for j in range(nV):
            if(distance[i][j] == INF):
                print("INF", end=" ")
            else:
                print(distance[i][j], end="  ")
        print(" ")


G = [[0, 3, INF, 5],
        [2, 0, INF, 4],
        [INF, 1, 0, INF],
        [INF, INF, 2, 0]]
floyd_warshall(G)
```

**OUTPUT:**

The all-pair shortest paths

```
0  3  7  5
2  0  6  4
3  1  0  5
5  3  2  0
```

**RESULT:**

Hence, successfully implemented the all-pair-shortest-paths problem using Floyd's algorithm.

## EXP NO 6 TRANSITIVE CLOSURE OF A DIRECTED GRAPH

## DATE : 03-04-23 USING WARSHALL'S ALGORITHM

### AIM:

To develop a program to compute the transitive closure of a given directed graph using Warshall's algorithm.

### ALGORITHM:

1. Input the number of vertices, n, in the graph.
2. Create an adjacency matrix, a, of size n x n and initialize it with the graph's connectivity information.
3. Apply the Warshall's algorithm:
    a. Iterate over a variable k from 0 to n-1, representing intermediate vertices.
    b. For each vertex pair (i, j), check if there exists a path from i to j through k. If a[i][k] is 1, set a[i][j] to 1.
    c. Repeat the above step for all vertex pairs.
4. Output the transitive closure matrix, a, which represents the connectivity information for all pairs of vertices.
5. End the algorithm.

### PROGRAM:

```
def warshall(a, n):
    for k in range(n):
        for i in range(n):
            if a[i][k] == 1:
                for j in range(n):
                    a[i][j] = a[i][j] or a[k][j]

n = int(input("Enter number of vertices: "))
a = []
```

```
print("Enter adjacency matrix:")
for i in range(n):
    temp = []
    temp = list(map(int,input().split()))
    a.append(temp)

warshall(a, n)

print("The transitive closure is:")
for i in range(n):
    for j in range(n):
        print(a[i][j], end="\t")
    print()
```

**OUTPUT:**

```
Enter number of vertices: 4
Enter adjacency matrix:
0       1       0       1
0       0       1       0
0       0       0       1
0       1       0       0
The transitive closure is:
0       1       1       1
0       1       1       1
0       1       1       1
0       1       1       1
```

**RESULT:**

Hence, successfully implemented Warshall's algorithm for finding transitive closure of a given directed graph.

# ALGORITHM DESIGN TECHNIQUES

## EXP NO 1 MINIMUM AND MAXIMUM OF AN ARRAY USING DIVIDE AND CONQUER

## DATE 10-04-23

### AIM:

To develop a program to find minimum and maximum in an array using divide and conquer technique.

### ALGORITHM:

- Divide array by calculating mid index i.e. mid = l + (r — l)/2
- Recursively find the maximum and minimum of left part by calling the same function i.e. leftMinMax[2] = minMax(X, l, mid)
- Recursively find the maximum and minimum of right part by calling the same function i.e. rightMinMax[2] = minMax(X, mid + 1, r)
- Finally, get the overall maximum and minimum by comparing the min and max of both halves.

### PROGRAM:

```
# Python program of above implementation
def getMinMax(low, high, arr):
    arr_max = arr[low]
    arr_min = arr[low]
    # If there is only one element
    if low == high:
        arr_max = arr[low]
        arr_min = arr[low]
        return (arr_max, arr_min)
    # If there is only two element
    elif high == low + 1:
        if arr[low] > arr[high]:
```

```
                arr_max = arr[low]
                arr_min = arr[high]
        else:
                arr_max = arr[high]
                arr_min = arr[low]
        return (arr_max, arr_min)
    else:
        # If there are more than 2 elements
        mid = int((low + high) / 2)
        arr_max1, arr_min1 = getMinMax(low, mid, arr)
        arr_max2, arr_min2 = getMinMax(mid + 1, high, arr)
    return (max(arr_max1, arr_max2), min(arr_min1, arr_min2))

arr = [1000, 11, 445, 1, 330, 3000]
high = len(arr) - 1
low = 0
arr_max, arr_min = getMinMax(low, high, arr)
print('Minimum element is ', arr_min)
print('\nMaximum element is ', arr_max)
```

**OUTPUT:**

```
Minimum element is  1

Maximum element is  3000
```

**RESULT:**

Hence, successfully implemented finding maximum and minimum elements using divide and conquer technique.

**EXP NO 2 MERGE SORT AND QUICK SORT IMPLEMENTATION**
**DATE :12-04-23**

**AIM:**

To implement Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

**ALGORITHM:**

1) Read the number of elements of the list as n.
2) Generate the n array elements using random.randint() function.
3) Initialize two lists
   - x : for storing the n values
   - t : for storing the time taken values corresponding to the respective n values
4) Start the timer.
5) Call the function mergeSort(list,0,n-1) to perform merge sort on the list
6) End the timer after the function call.
7) PRINT the sorted list
8) Compute time_taken = end time - start time and add to the t list.
9) Add the value of n to the x list.
10) Ask for choice 1. Continue 2. Exit
11) IF choice = 1 then
    Repeat steps 1 to 11
    ELSE
    Go to step 12
12) Plot a graph for x_list vs t_list
13) End

mergeSort(list,l,r)

1) IF l < r means divide the list into smaller parts until each part has one element
   - Compute the mid value m=l+(r-l)/2
   - Call mergeSort(list,l,m)
   - Call mergeSort(list,m+1,r)
   - Call merge(list,l,m,r) to merge the parts by sorting them recursively.

**PROGRAM:**

```
import random
import time
def mergeSort(arr, l, r):
    if l < r:
        m = l+(r-l)//2
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    L = [0] * (n1)
    R = [0] * (n2)


    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    i = 0
    j = 0
    k = l

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1



    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1



    while j < n2:
        arr[k] = R[j]
```

```
        j += 1
        k+=1
n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 999)) for i in range(n)]
b=a

t=[]
x=[]

x.append(n)

while True:

    start = time.time()
    mergeSort(a,0,n-1)
    end = float(time.time())
    time_taken = float(end - start)
    print(f"Time taken ={time_taken:.6f}")
    t.append(time_taken)
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the number of elements of the list: "))
        x.append(n)
        a = [int(random.randint(0, 9999)) for i in range(n)]


    else:
        break


print("The n values",x)
print("The time taken list",t)

import matplotlib.pyplot as plt
plt.title("MERGE SORT")
plt.plot(x,t)
```

**OUTPUT:**

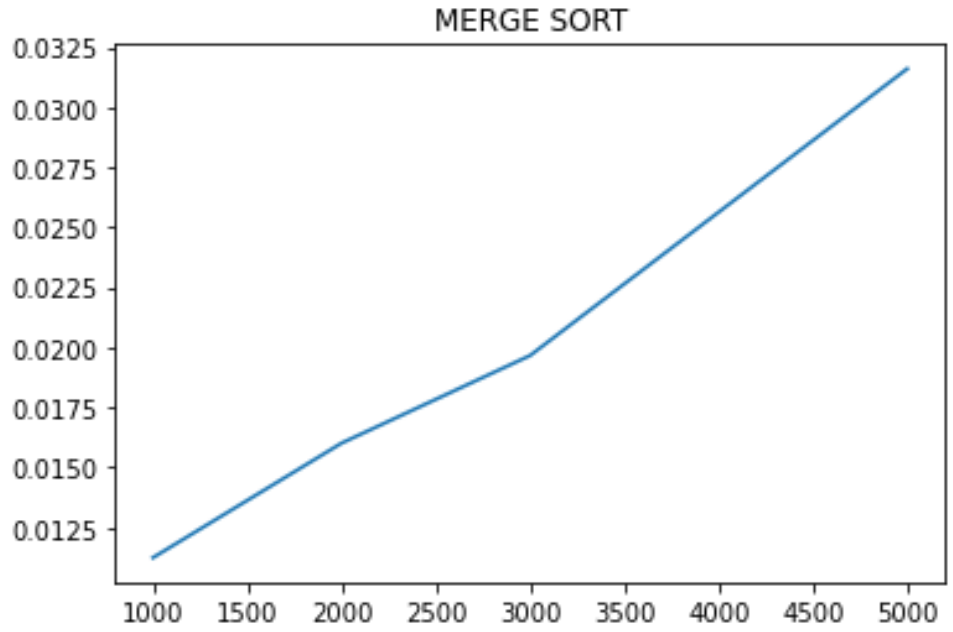
```
Enter the number of elements of the list: 1000
Time taken =0.011242

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 2000
Time taken =0.016011
```

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 3000
Time taken =0.019660

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 5000
Time taken =0.031603

1. Continue
2. Exit
Enter your option: 2

The n values [1000, 2000, 3000, 5000]
The time taken list [0.011242151260375977, 0.016011476516723633, 0.019660472869873047, 0.03160262107849121]




**QUICK SORT ALGORITHM:**

1) Read the number of elements of the list as n.
2) Generate the n array elements using random.randint() function.
3) Initialize two lists
   - x : for storing the n values
   - t : for storing the time taken values corresponding to the respective n values

4) Start the timer.
5) Call the function quickSort(list,0,n-1) to perform quick sort on the list
6) End the timer after the function call.
7) PRINT the sorted list
8) Compute time_taken = end time - start time and add to the t list.
9) Add the value of n to the x list.
10) Ask for choice 1. Continue 2. Exit
11) IF choice = 1 then
        Repeat steps 1 to 11
    ELSE
        Go to step 12
12) Plot a graph for x_list vs t_list
13) End

quickSort(list,low,high)

1) IF low < high means divide the list into smaller parts until each part has one element
    - Call partition(list,low,high) to find the index of the pivot element after sorting it in its correct position in the list.
    - Call quickSort(list,low,pi-1) to perform quicksort on the left side of the pivot element
    - Call quickSort(list,pi+1,high) to perform quicksort on the right side of the pivot element

partition(list,low,high)

1) Set the last element as pivot.
2) Compare all the elements with the pivot element.
3) Shift the elements that are smaller than the pivot to its left and greater than it to its right.
4) Return the position of the pivot (sorted).

**PROGRAM:**

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1
```

```
def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)

        quickSort(array, low, pi - 1)

        quickSort(array, pi + 1, high)

n = int(input("Enter the number of elements of the list: "))
a = [int(random.randint(0, 9999)) for i in range(n)]
tt=[]
xx=[]
xx.append(n)
while True:
    start = time.time()
    quickSort(a,0,n-1)
    end = time.time()
    time_taken = end - start
    tt.append(time_taken)
    print(f"Time taken ={time_taken:.6f}")
    print("\n1. Continue\n2. Exit")
    opt = int(input("Enter your option: "))
    if opt == 1:
        n = int(input("Enter the number of elements of the list: "))
        xx.append(n)
        a = [int(random.randint(0, 9999)) for i in range(n)]



    else:
        break

print("The n values",xx)
print("The time taken",tt)
plt.title("Quick sort")
plt.plot(xx,tt,'r')
```

**OUTPUT:**

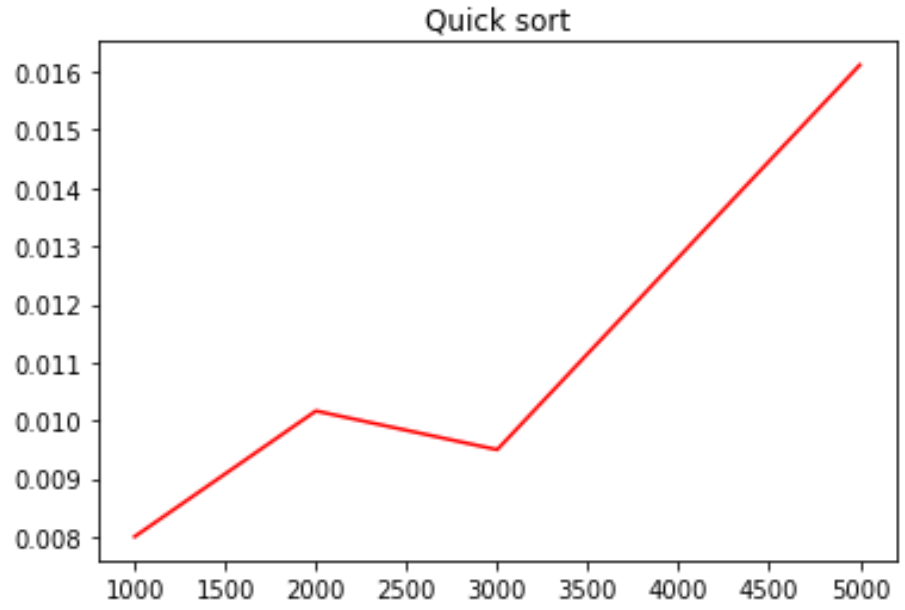
```
Enter the number of elements of the list: 1000
Time taken =0.007999

1. Continue
```

2. Exit
Enter your option: 1
Enter the number of elements of the list: 2000
Time taken =0.010163

1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 3000
Time taken =0.009496

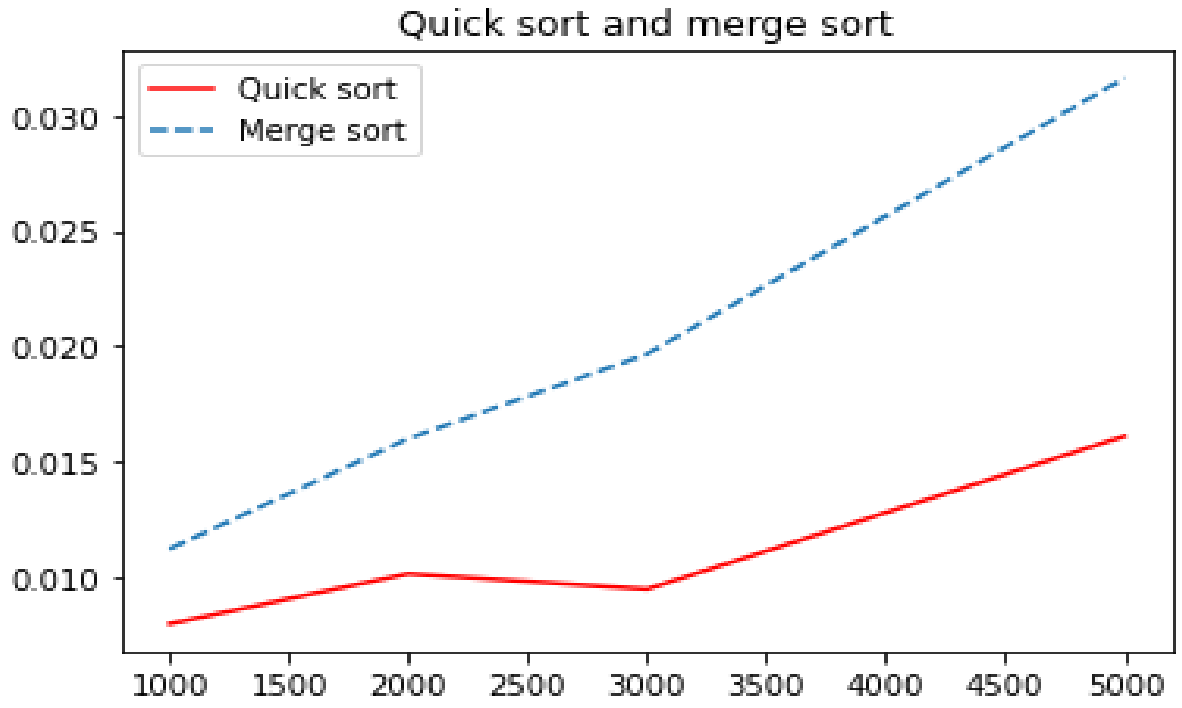
1. Continue
2. Exit
Enter your option: 1
Enter the number of elements of the list: 5000
Time taken =0.016119

1. Continue
2. Exit
Enter your option: 2
The n values [1000, 2000, 3000, 5000]
The time taken [0.007999181747436523, 0.010163068771362305, 0.009496212005615234, 0.01611924171447754]



**QUICK SORT AND MERGE SORT GRAPH:**

```
plt.title("Quick sort and merge sort")
plt.plot(xx,tt,'r')
plt.plot(x,t,'--')
plt.legend(['Quick sort','Merge sort'])
```




**RESULT:**

Hence, successfully implemented Merge sort and Quick sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

# STATE SPACE SEARCH ALGORITHMS

## EXP NO 1 N QUEENS PROBLEM
## DATE : 17-04-23

### AIM:

To develop a program to implement the N-Queens problem using Backtracking.

### ALGORITHM:

**Step 1 -** Place the queen row-wise, starting from the left-most cell.

**Step 2 -** If all queens are placed then return true and print the solution matrix.

**Step 3 -** Else try all columns in the current row.

- **Condition 1 -** Check if the queen can be placed safely in this column then mark the current cell [Row, Column] in the solution matrix as 1 and try to check the rest of the problem recursively by placing the queen here leads to a solution or not.
- **Condition 2 -** If placing the queen [Row, Column] can lead to the solution return true and print the solution for each queen's position.
- **Condition 3 -** If placing the queen cannot lead to the solution then unmark this [row, column] in the solution matrix as 0, BACKTRACK, and go back to condition 1 to try other rows.

**Step 4 -** If all the rows have been tried and nothing worked, return false to trigger backtracking.

### PROGRAM:

```
def place(m, k):
    for i in range(k):
        if m[i] == m[k] or abs(m[i] - m[k]) == abs(i - k):
            return False
    return True

def display(m, n):
    s = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        s[i][m[i]] = 1
    for i in range(n):
        for j in range(n):
            if s[i][j]:
                print("Q", end="\t")
            else:
                print("x", end="\t")
        print()
```

```
    exit(1)

def main():
    n = int(input("Enter number of Queens: "))
    print("\nThe solution for the problem is:\n")
    n -= 1
    m = [0] * 50
    k = 0
    while k >= 0:
        m[k] += 1
        while m[k] <= n and not place(m, k):
            m[k] += 1
        if m[k] <= n:
            if k == n:
                display(m, n + 1)
                break
            else:
                k += 1
                m[k] = -1
        else:
            k -= 1

if __name__ == "__main__":
    main()
```

**OUTPUT:**

Enter number of Queens: 4

The solution for the problem is:

x Q x x
x x x Q
Q x x x
x x Q x

**RESULT:**

Hence, successfully implemented the N Queens problem using backtracking.

# APPROXIMATION AND RANDOMISED ALGORITHMS

## EXP NO 1 TRAVELLING SALESMAN PROBLEM
## DATE : 24-04-23

### AIM:

To develop a program to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

### ALGORITHM:

**Step 1** − Choose any vertex of the given graph randomly as the starting and ending point.

**Step 2** − Construct different paths by using itertools.permutatioms() function.

**Step 3** − Compute the cost of each path and keep track of the minimum cost path.

**Step 4** − The minimum cost path obtained in step 3 is the exact path cost.

**Step 5** – Compute an approximate path cost whose value is close to the exact path cost but greater than it.

**Step 6** – Print the exact path and its cost.

**Step 7** – Print the approximate path and its cost.

**Step 8** – Calculate the error in approximation as approx_cost/exact_cost - 1 and print the same.

### PROGRAM:

```
import itertools
# function to calculate the cost of a path
def calculate_cost(path, graph):
    cost = 0
    for i in range(len(path) - 1):
        cost += graph[path[i]][path[i+1]]
    cost += graph[path[-1]][path[0]]
    return cost

# exact algorithm using brute-force approach
def tsp_exact(graph):
    n = len(graph)
    cities = list(range(n))
    best_path = None
    best_cost = float('inf')
```

```
    for path in itertools.permutations(cities):
        cost = calculate_cost(path, graph)
        if cost < best_cost:
            best_path = path
            best_cost = cost
    return best_path, best_cost

# approximation algorithm using nearest-neighbor approach
def tsp_approx(graph):
    n = len(graph)
    current_city = 0
    unvisited_cities = set(range(1, n))
    path = [current_city]
    cost = 0
    while unvisited_cities:
        next_city = min(unvisited_cities, key=lambda city: graph[current_city][city])
        unvisited_cities.remove(next_city)
        path.append(next_city)
        cost += graph[current_city][next_city]
        current_city = next_city
    cost += graph[path[-1]][path[0]]
    return path, cost

graph = [[0, 2, 9, 10],
         [1, 0, 6, 4],
         [15, 7, 0, 8],
         [6, 3, 12, 0]]

exact_path, exact_cost = tsp_exact(graph)
print("Exact solution: path={}, cost={}".format(exact_path, exact_cost))
approx_path, approx_cost = tsp_approx(graph)
print("Approximation: path={}, cost={}".format(approx_path, approx_cost))
print("Error in approximation: {}".format(approx_cost / exact_cost - 1))
```

**OUTPUT:**

```
 Exact solution: path=(0, 2, 3, 1), cost=21
Approximation: path=[0, 1, 3, 2], cost=33
Error in approximation: 0.5714285714285714
```

**RESULT:**

Hence, successfully implemented the travelling salesman problem using approximation algorithm.

**EXP NO 2** **RANDOMISED ALGORITHM FOR FINDING KTH**
**DATE : 03-05-23** **SMALLEST NUMBER**

**AIM:**

To develop a program to find the kth smallest number using a randomized algorithm.

**ALGORITHM:**

- Select a random element from an array as a pivot.
- Then partition to the array around the pivot, its help to all the smaller elements were placed before in the pivot and all greater elements are placed after the pivot.
- Then Check the position of the pivot. If it is the kth element then return it.
- If it is less than the kth element then repeat the process of the subarray.
- If it is greater than the kth element then repeat the process of the left subarray.
- Print the kth smallest element that has returned from the function.
- End.

**PROGRAM:**

```
import random
def kthSmallest(arr, l, r, k):
    if (k > 0 and k <= r - l + 1):
        pos = randomPartition(arr, l, r)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return kthSmallest(arr, l, pos - 1, k)
        return kthSmallest(arr, pos + 1, r,
                           k - pos + l - 1)
    return 999999999999


def swap(arr, a, b):
    temp = arr[a]
```

```
    arr[a] = arr[b]
    arr[b] = temp

def partition(arr, l, r):
    x = arr[r]
    i = l
    for j in range(l, r):
        if (arr[j] <= x):
            swap(arr, i, j)
            i += 1
    swap(arr, i, r)
    return i

def randomPartition(arr, l, r):
    n = r - l + 1
    pivot = int(random.random() * n)
    swap(arr, l + pivot, r)
    return partition(arr, l, r)

arr = []
print("Enter No. of elements in the array:")
n = int(input())
print("Enter the elements:")
for i in range(0, n):
        num = int(input())
        arr.append(num)
n = len(arr)
print("Enter the k value: ")
k=int(input())
```

```
print("K'th smallest element is",kthSmallest(arr, 0, n - 1, k))
```

**OUTPUT:**

```
Enter No. of elements in the array:
5
Enter the elements:
12
45
14
33
88
Enter the k value:
2
K'th smallest element is 14
```

**RESULT:**

Hence, successfully implemented a randomised algorithm for finding the kth smallest Number.