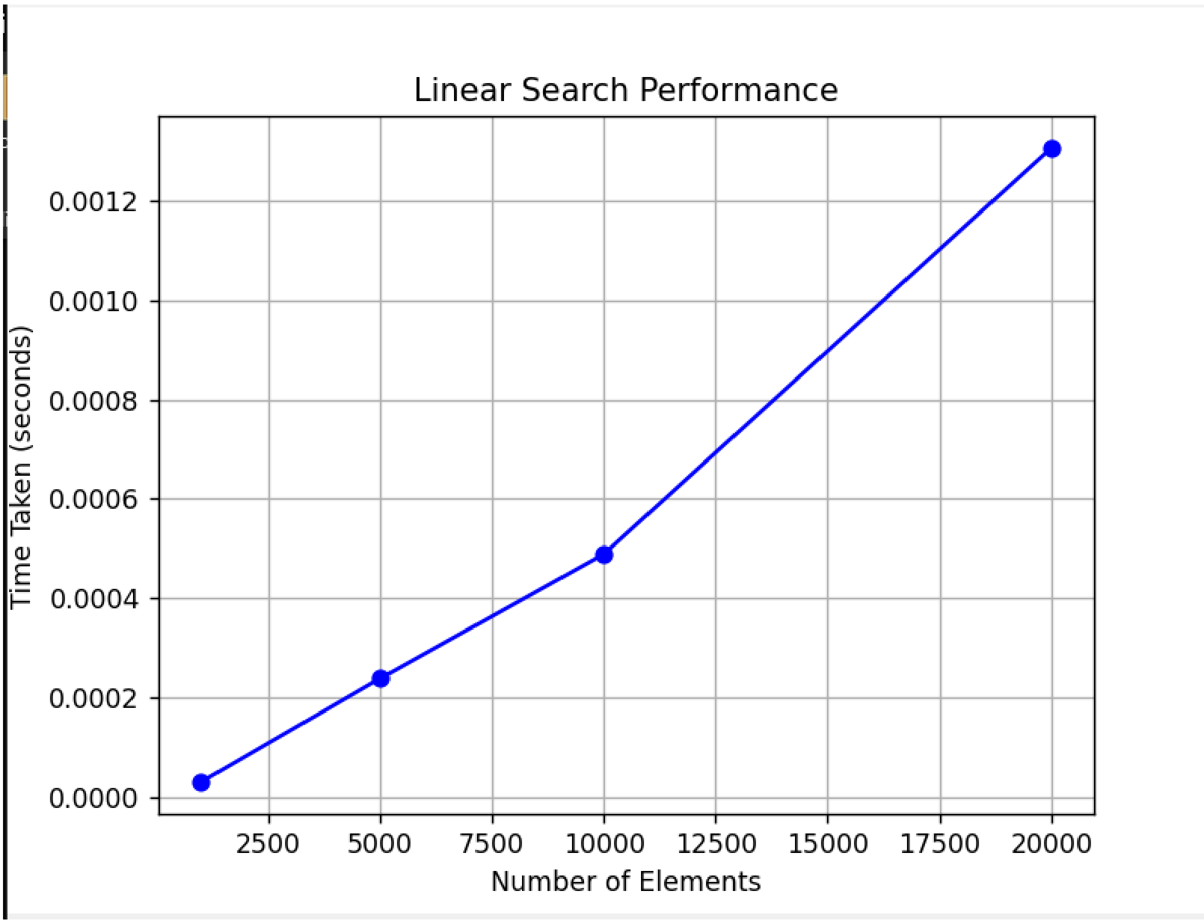
1.Linear Search

```
import time
import random
import matplotlib.pyplot as plt
def linear_search(arr, key):
    return arr.index(key) if key in arr else -1
sizes = [1000, 5000, 10000, 20000]
times = []
for n in sizes:
    arr = sorted(random.sample(range(1, 1000000), n))
    key = arr[-1]
    print(f"Array size: {n}")
    print(f"Sample elements: {arr[:10]} ...")
    print(f"Target element: {key}\n")
    start_time = time.perf_counter()
    position = linear_search(arr, key)
    end_time = time.perf_counter()
    times.append(end_time - start_time)
    print(f"Search element found at position: {position + 1}" if position != -1 else "Search element not found.")
    print(f"Time taken: {times[-1]:.6f} seconds\n")
plt.plot(sizes, times, marker='o', linestyle='-', color='b')
plt.xlabel("Number of Elements")
plt.ylabel("Time Taken (seconds)")
plt.title("Linear Search Performance")
plt.grid()
plt.show()
```

```
========================== RESTART: D:/Download/eee.py ==========================
Array size: 1000
Sample elements: [650, 1562, 3532, 4552, 7604, 8170, 8933, 9923, 12070, 12400] .
..
Target element: 999740

Search element found at position: 1000
Time taken: 0.000030 seconds

Array size: 5000
Sample elements: [296, 329, 586, 621, 1016, 1179, 1450, 1458, 1515, 2078] ...
Target element: 999945

Search element found at position: 5000
Time taken: 0.000238 seconds

Array size: 10000
Sample elements: [212, 363, 499, 618, 685, 687, 961, 1039, 1207, 1257] ...
Target element: 999862

Search element found at position: 10000
Time taken: 0.000489 seconds

Array size: 20000
Sample elements: [177, 206, 296, 439, 506, 550, 598, 608, 615, 626] ...
Target element: 999948

Search element found at position: 20000
Time taken: 0.001306 seconds
```



2.Binary Search

```
import time
import random
import matplotlib.pyplot as plt
def binary_search(arr, key):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1
sizes = [1000, 5000, 10000, 20000]
times = []
for n in sizes:
    arr = sorted(random.sample(range(1, 1000000), n))
    key = arr[(0+n)//2]
    print(f"Array size: {n}")
    print(f"Sample elements: {arr[:10]} ...")
    print(f"Target element: {key}\n")
    start_time = time.perf_counter()
    position = binary_search(arr, key)
    end_time = time.perf_counter()
    times.append(end_time - start_time)
    print(f"Search element found at position: {position + 1}" if position != -1 else "Search element not found.")
    print(f"Time taken: {times[-1]:.6f} seconds\n")
plt.plot(sizes, times, marker='o', linestyle='-', color='g')
```

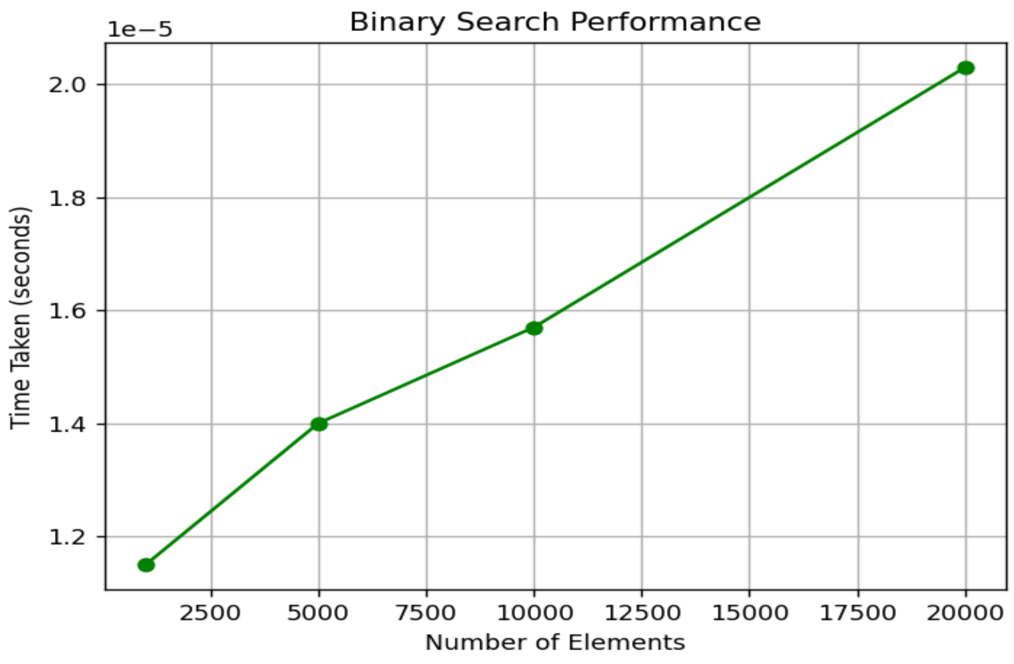
plt.xlabel("Number of Elements")

plt.ylabel("Time Taken (seconds)")

plt.title("Binary Search Performance")

plt.show)

```
========================= RESTART: D:/Download/binary.py ========================
Array size: 1000
Sample elements: [945, 1283, 1532, 1538, 2065, 6201, 9128, 9196, 11243, 11819] .
..
Target element: 521774

Search element found at position: 501
Time taken: 0.000011 seconds

Array size: 5000
Sample elements: [125, 354, 474, 486, 556, 748, 785, 788, 952, 1369] ...
Target element: 503617

Search element found at position: 2501
Time taken: 0.000014 seconds

Array size: 10000
Sample elements: [45, 63, 66, 161, 205, 815, 897, 905, 970, 1047] ...
Target element: 495485

Search element found at position: 5001
Time taken: 0.000016 seconds

Array size: 20000
Sample elements: [26, 84, 89, 111, 127, 149, 180, 356, 361, 398] ...
Target element: 498692

Search element found at position: 10001
Time taken: 0.000020 seconds
```



3.Naive pattern search

```
def search(pat, txt):
    n = len(txt)
    m = len(pat)
    found = False
    for i in range(n - m + 1):
        if txt[i:i + m] == pat:
            print(f"Pattern found at index {i}")
            found = True
    if not found:
        print("Pattern not found")
txt = input("Enter the text: ")
pat = input("Enter the pattern: ")
search(pat, txt)
```

```
= RESTART: D:/Download/naivepattern.py
Enter the text: AABAACAADAABAAABAA
Enter the pattern: AABA
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

3.Insertionsort vs Heapsort

```
import time
import random
import matplotlib.pyplot as plt
import heapq
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key, j = arr[i], i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
def heap_sort(arr):
    heapq.heapify(arr)
    return [heapq.heappop(arr) for _ in range(len(arr))]
sizes = [100, 200, 300]
insertion_times, heap_times = [], []
for n in sizes:
    arr = random.sample(range(1, 1000000), n)
    start = time.perf_counter()
    insertion_sort(arr[:])
    insertion_times.append(time.perf_counter() - start)
    start = time.perf_counter()
    heap_sort(arr[:])
    heap_times.append(time.perf_counter() - start)
    print(f"Size: {n} | Insertion Sort: {insertion_times[-1]:.6f}s | Heap Sort: {heap_times[-1]:.6f}s")
plt.plot(sizes, insertion_times, 'bo-', label="Insertion Sort")
plt.plot(sizes, heap_times, 'ro-', label="Heap Sort")
plt.xlabel("Number of Elements"), plt.ylabel("Time (s)"), plt.title("Sort Performance")
plt.legend(), plt.grid(), plt.show()
```

```
= RESTART: D:/Download/insertionheap.py
Size: 100 | Insertion Sort: 0.000190s | Heap Sort: 0.000033s
Size: 200 | Insertion Sort: 0.000996s | Heap Sort: 0.000092s
Size: 300 | Insertion Sort: 0.001619s | Heap Sort: 0.000094s
```

5.DFS

```
n = int(input("Enter the number of Vertices: "))
graph = {i: [] for i in range(n)}
for _ in range(int(input("Enter the number of Edges: "))):
    u, v = map(int, input("Edge (u v): ").split())
    graph[u].append(v)
    graph[v].append(u)
def dfs(node, visited):
    if node in visited:
        return
    print(node, end=" ")
    visited.add(node)
    for neighbor in graph[node]:
        dfs(neighbor, visited)
visited = set()
dfs(int(input("Start vertex: ")), visited)
```

```
= RESTART: D:/Download/dfs.py
Enter the number of Vertices: 5
Enter the number of Edges: 4
Edge (u v): 0 1
Edge (u v): 0 2
Edge (u v): 1 3
Edge (u v): 1 4
Start vertex: 0
0 1 3 4 2
```

6.BFS

```
from collections import deque
n = int(input("Enter the number of Vertices: "))
graph = {i: set() for i in range(n)}
for _ in range(int(input("Enter the number of Edges: "))):
    u, v = map(int, input("Edge (u v): ").split())
    graph[u].add(v)
    graph[v].add(u)
def bfs(start):
    visited, queue = set(), deque([start])
    while queue:
        node = queue.popleft()
        if node in visited: continue
        print(node, end=" ")
        visited.add(node)
        queue.extend(graph[node] - visited)
print("\nBFS Traversal:")
bfs(int(input("Start vertex: ")))
```

```
=================================
Enter the number of Vertices: 5
Enter the number of Edges: 4
Edge (u v): 0 1
Edge (u v): 0 2
Edge (u v): 1 3
Edge (u v): 1 4

BFS Traversal:
Start vertex: 0
0 1 2 3 4
```

7.Dijikstra

```
V = int(input("Enter the number of Vertices: "))
print("Enter the adjacency matrix:")
graph = [list(map(int, input().split())) for _ in range(V)]
def dijkstra(graph, src):
    INF = float('inf')
    dist, visited = [INF] * V, set()
    dist[src] = 0
    for _ in range(V - 1):
        u = min((d, v) for v, d in enumerate(dist) if v not in visited)[1]
        visited.add(u)
        for v in range(V):
            if v not in visited and graph[u][v] and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]
    print("\nVertex \t Distance from Source")
    for i, d in enumerate(dist):
        print(f"{i} \t {d}")
dijkstra(graph, int(input("Source vertex: ")))
```

```
Enter the number of Vertices: 4
Enter the adjacency matrix:
0 10 6 5
10 0 15 0
6 15 0 4
5 0 4 0
Source vertex: 0

Vertex   Distance from Source
0        0
1        10
2        6
3        5
```

8.Prims's

```
V = int(input("Enter the number of Vertices: "))
print("Enter the adjacency matrix:")
graph = [list(map(int, input().split())) for _ in range(V)]
selected, edges, total_cost = [False] * V, 0, 0
selected[0] = True
print("\nMST Edges (u - v : weight):")
while edges < V - 1:
    min_weight, u, v = float('inf'), -1, -1
    for i in range(V):
        if selected[i]:
            for j in range(V):
                if not selected[j] and 0 < graph[i][j] < min_weight:
                    min_weight, u, v = graph[i][j], i, j
    print(f"{u} - {v} : {min_weight}")
    selected[v], total_cost, edges = True, total_cost + min_weight, edges + 1
print("\nTotal MST Cost:", total_cost)
```

```
========================== RESTART: D:/Dow
Enter the number of Vertices: 4
Enter the adjacency matrix:
0 10 6 5
10 0 15 0
6 15 0 4
5 0 4 0

MST Edges (u - v : weight):
0 - 3 : 5
3 - 2 : 4
0 - 1 : 10

Total MST Cost: 19
```

9.Floyd

```
INF = float('inf')
V = int(input("Enter the number of vertices: "))
print(f"Enter the adjacency matrix ({V}x{V}) row by row:")
graph = []
for i in range(V):
    row = list(map(int, input().split()))
    graph.append([INF if x == 0 and i != j else x for j, x in enumerate(row)])
dist = [row[:] for row in graph]
for k in range(V):
    for i in range(V):
        for j in range(V):
            if dist[i][k] != INF and dist[k][j] != INF:
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
print("\nShortest distances between every pair of vertices:")
for row in dist:
    for val in row:
        print("INF" if val == INF else val, end="\t")
    print()
```

```
Enter the number of vertices: 5
Enter the adjacency matrix (5x5) row by row:
0 4 0 0 0
4 0 8 0 0
0 8 0 7 0
0 0 7 0 9
0 0 0 9 0

Shortest distances between every pair of vertices:
0	4	12	19	28
4	0	8	15	24
12	8	0	7	16
19	15	7	0	9
28	24	16	9	0
```

10.Warshall

```
n = int(input("Enter the number of vertices: "))


graph = []
print("Enter the adjacency matrix row by row:")
for _ in range(n):
    graph.append(list(map(int, input().split())))


for k in range(n):
    for i in range(n):


        for j in range(n):
            graph[i][j] = graph[i][j] or (graph[i][k] and graph[k][j])
print("Transitive Closure:")
for row in graph:
    print(" ".join(map(str, row)))
```

```
= RESTART: D:/python/warshall.py
Enter the number of vertices: 4
Enter the adjacency matrix row by row:
0 1 0 0
1 0 1 0
1 0 0 1
0 1 1 0
Transitive Closure:
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
```

11.Minmax

```
def find_max_min(arr):
    if len(arr) == 1:
        return arr[0], arr[0]

    if len(arr) == 2:
        return (arr[0], arr[1]) if arr[0] > arr[1] else (arr[1], arr[0])

    mid = len(arr) // 2
    left_max, left_min = find_max_min(arr[:mid])
    right_max, right_min = find_max_min(arr[mid:])

    return max(left_max, right_max), min(left_min, right_min)

n = int(input("Enter the number of elements: "))
arr = list(map(int, input(f"Enter {n} elements separated by spaces: ").split()))

if len(arr) != n:
    print("Error: Number of elements entered does not match the expected count.")
else:
    max_num, min_num = find_max_min(arr)
    print("Maximum number:", max_num)
    print("Minimum number:", min_num)
```

```
============== RESTART: D:/python/maxmin.py ==
===========
Enter the number of elements: 6
Enter 6 elements separated by spaces: 23 5 1 0
2 122
Maximum number: 122
Minimum number: 0
```

12.MergeVS Quick

```
import time
import random
import matplotlib.pyplot as plt
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left, right = arr[:mid], arr[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
```

```
        j += 1
        k += 1
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
sizes = [100, 200, 300]
merge_times, quick_times = [], []
for n in sizes:
    arr = random.sample(range(1, 1000000), n)
    start = time.perf_counter()
    merge_sort(arr[:])
    merge_times.append(time.perf_counter() - start)
    start = time.perf_counter()
    quick_sort(arr[:])
    quick_times.append(time.perf_counter() - start)
    print(f"Size: {n} | Merge Sort: {merge_times[-1]:.6f}s | Quick Sort: {quick_times[-1]:.6f}s")
plt.plot(sizes, merge_times, 'bo-', label="Merge Sort")
plt.plot(sizes, quick_times, 'ro-', label="Quick Sort")
plt.xlabel("Number of Elements"), plt.ylabel("Time (s)"), plt.title("Sort Performance")
plt.legend(), plt.grid(), plt.show()
```

```
====================== RESTART: D:/Download/quickmerge.py ====
Size: 100 | Merge Sort: 0.000144s | Quick Sort: 0.000170s
Size: 200 | Merge Sort: 0.000291s | Quick Sort: 0.000347s
Size: 300 | Merge Sort: 0.000604s | Quick Sort: 0.000664s
```