	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Profesor(a): Jorge Alberto Solano Gálvez

Asignatura: Estructuras de Datos y Algoritmos II

Grupo: 2

No de Práctica(s): 4

Integrante(s): Valenzuela Ascencio Gustavo

No. de Equipo de cómputo empleado: -

Semestre: 2024-1

Fecha de entrega: 22 de septiembre del 2023

Observaciones: ______

CALIFICACIÓN: __________________

Searching Algorithms I.

Objective: Learn about the structure of the key comparison based searching algorithms

Activities:

- Implement the recursive and iterative algorithm of linear search in python language for locating a key or value in a data set.
- Implement the recursive and iterative algorithm of binary search in python language for locating a key or value in a data set.

Instructions:

- Implement sequential search in Python, both iteratively and recursively, to find a node.
- Implement binary search in the Python language, both iteratively and recursively, to find a node. Before performing the search, the set must be sorted using a direct sorting method (n^2) and a logarithmic sorting method (n * log(n)).
- Determine the algorithmic complexity for each implementation (sequential search, binary search with direct sorting, and binary search with logarithmic sorting).
- Obtain the empirical performance of the algorithms for the best, worst, and average-case scenarios for each implementation (sequential search, binary search with direct sorting, and binary search with logarithmic sorting).

The practice must be done individually.

The practice is checked during the laboratory session and must be uploaded with all the code and the report in a compressed file, once qualified, to the SiCCAAD platform.

Linear search

Linear search is an algorithm to find a value in every kind of set, based and iterating and comparing a key to find, using brute force approach.

Linear search implementation and analysis using RAM model

```python
# Linear search analysis TIME || SPACE

def linear_search(key_to_find,nodes):

    for i in range (len(nodes)): # 4(n+1) || 1

        if nodes[i].key == key_to_find: # 5(n) || 1

            return i # 1 || 1

    return None # 1 || 1


# Linear search TIME polynomial: 9n+6 = O(n)

# Linear search SPACE polynomial: 4 = O(1)



# Linear search (Recursive) analysis TIME || SPACE

def recursive_linear_search(nodes, key, index=0):

    if index >= len(nodes): # 6 || 1

        return None  # 1 || 1

    if nodes[index].key == key: # 6 || 1

        return index  # 1 || 1

    else: # 1 || 1

            return recursive_linear_search(nodes, key, index + 1)  # n recursive calls, multiplies all the function  5(n) || n

# Linear search (Recursive) TIME polynomial: 20n = O(n)

# Linear search (Recursive) SPACE polynomial: 5n = O(n)
```

```
#       Worst Case       ||       Average case        ||       Best case
```

```
def maxHeap(arr, n):

    for i in range(n // 2 - 1, -1, -1): #4 * (n/2)

        heapify(arr, n, i) #45 (log n) * (n/2)

    # maxHeap polyomial: 45 (n log n) + 4 (n/2) = O(n log n)



def heapify(arr, n, i):

    largest = i # 3

    left_child = 2 * i + 1 # 5

    right_child = 2 * i + 2 # 5


    if left_child < n and arr[left_child] > arr[largest]: # 8

        largest = left_child # 3


    if right_child < n and arr[right_child] > arr[largest]: # 8

        largest = right_child # 3


     # After this we've been completed a subtree transversal, dividing by
two the set, making the recursive call

    # logarithmic


    if largest != i: # 4

        arr[i], arr[largest] = arr[largest], arr[i]  # 6

          heapify(arr, n, largest)  # Recursive call, we go down the tree
until we get to the leaves (log n)
```

```
    # Heapify polynomial: 45 (log n) = O(log n)


def heapSort(arr):

    n = len(arr) # 4


    maxHeap(arr,n) # 45(n log n) + 4(n/2)


    for i in range(n - 1, 0, -1): # 5(n+1)

        arr[0], arr[i] = arr[i], arr[0]  # 6(n)

        heapify(arr, i, 0)  # 45(log n) * (n)


    # heapSort polyomial: 90 (n log n) + 11(n) + 4(n/2) + 9 = O(n log n)


# Example usage

arr = [12, 11, 13, 5, 6, 7, 10, 15, 90, 117, 95, 80, 77, -100, 0]

heapSort(arr)

print("Sorted array:", arr)
```

Linear search all cases complexity explanation.

Best case: The first element is the element to search. $\Omega(1)$

Average case: A random element is the element to search. $\Theta(n)$

Worst case: The element isn’t in the set. $O(n)$

Binary search.

Binary search is an algorithm to find a value in a sorted set, based and iterating and comparing a key to find, using the divide and conquer approach.

Binary Search implementation and analysis using RAM Model

```
# Binary search analysis TIME || SPACE

def binary_search(key_to_find,nodes):

    left, right = 0, len(nodes) - 1 # 6 || 2

    while left <= right: #  4 (log n + 1) || 1

        mid = (left + right) // 2 # 5 (log n) || 1

        if nodes[mid].key == key_to_find: # 6 (log n)|| 1

            return mid # 1 || 1

        elif nodes[mid].key < key_to_find: # 6 (log n) || 1

            left = mid + 1 # Constantly dividing by two 4(log n) || 1

        else:

                right = mid - 1 # # Constantly dividing by two (by 4 in
accumnulate) 8(log n) || 1

    return None # 1 || 1


# Binary search TIME polynomial: 33(log n) + 12 = O(log n)

# Binary search SPACE polynomial: 8 = O(1)
```

```
# Binary search (Recursive) analysis TIME || SPACE

def recursive_binary_search(nodes, key, left=0, right=None):

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1
```

```
    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_search(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_search(nodes, key, left, mid - 1)

         # log n recursive calls, multiplies all the function  12(log n) ||
2 log n


# Binary search TIME polynomial: 48 log n = O(log n)

# Binary search SPACE polynomial: 10 log n = O(log n)
```

Binary search all cases complexity explanation.

Best case: The middle element is the element to search. $\Omega(1)$

Average case: A random element is the element to search. $\Theta(log\ n)$

Worst case: The element isn't in the set. $O(log\ n)$

Binary search all cases complexity explanation (with inefficient sorting algorithm).

Best case: The middle element is the element to search. $\Omega(1)$ + Bubble sort complexity $\Omega(n^2)$

Average case: A random element is the element to search. $\Theta(log\, n)$ Bubble sort complexity $\Theta(n^2)$

Worst case: The element isn't in the set. $O(log\, n)$ + Bubble sort complexity $O(n^2)$

So, for any case complexity will be $O(n^2)$.

Binary search all cases complexity explanation (with efficient sorting algorithm).

Best case: The middle element is the element to search. $\Omega(1)$ + Heap sort complexity $\Omega(nlogn)$

Average case: A random element is the element to search. $\Theta(log\, n)$ Heap sort complexity $\Theta(nlogn)$

Worst case: The element isn't in the set. $O(log\, n)$ + Heap sort complexity $O(nlogn)$

So, for any case complexity will be O(nlogn).

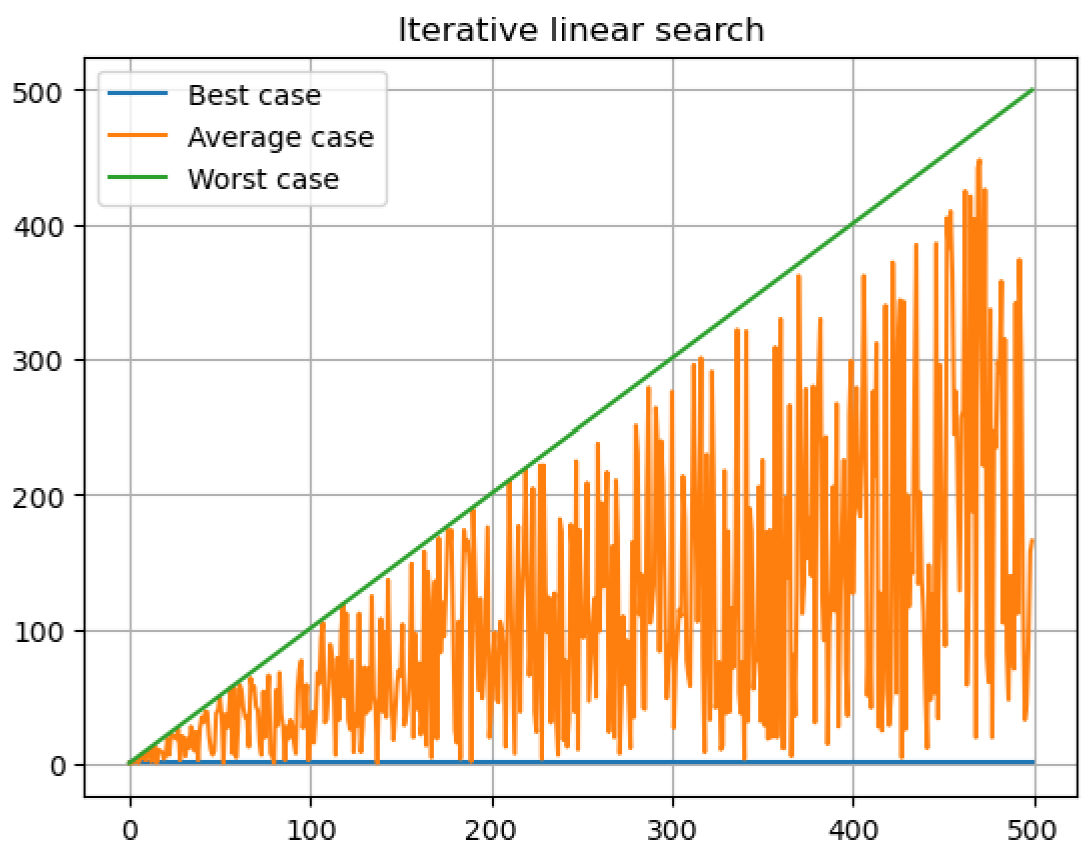
All cases graph and graph code.

```
import random

import matplotlib.pyplot as plt


def generate_random_string(length=16):

    characters = string.ascii_letters + string.digits

        random_string = ''.join(random.choice(characters) for _ in range(length))

    return random_string
```

```
class Node:

    def __init__(self,key):

        self.key = key

        self.value = generate_random_string()


def linear_search(key_to_find,nodes):

    times = 0

    for i in range (len(nodes)): # 4(n+1) || 1

        times += 1

        if nodes[i].key == key_to_find: # 5(n) || 1

            return times # 1 || 1

    return times # 1 || 1



if __name__ == "__main__":

    x = []

    y = []

    l = []

    n = 500

    for i in range(n):

        x.append(i)

        temp = Node (random.randint(-500,500))

        l.append(temp)

        y.append(linear_search(l[0].key,l))
```

```
u = []

v = []

n = 500

l = []

for i in range(n):

    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    v.append(linear_search(averageCase.key,l))


a = []

b = []

n = 500

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    b.append(linear_search(1000,l))


plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")
```

```
    plt.legend()

    plt.title('Iterative linear search')

    plt.grid(True)

    plt.show()
```




```
import random

import matplotlib.pyplot as plt


times = 0

space = 0
```

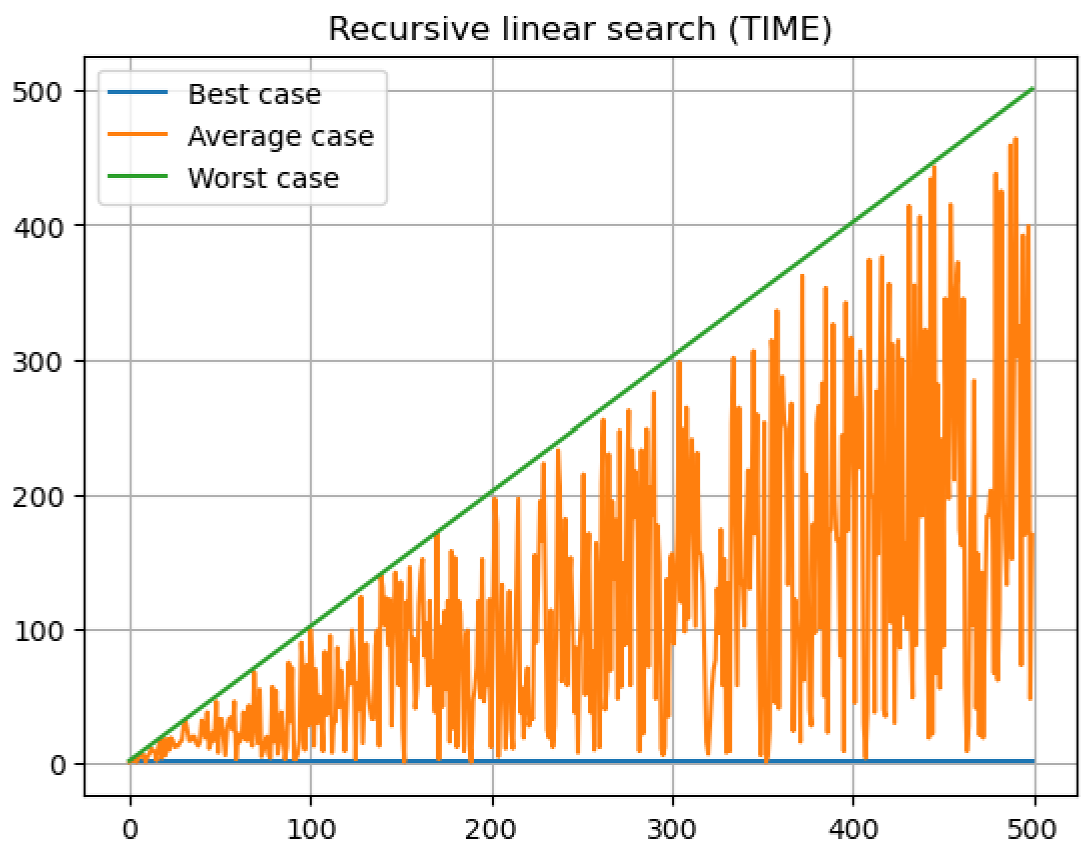
```python
def generate_random_string(length=16):
    characters = string.ascii_letters + string.digits
        random_string = ''.join(random.choice(characters) for _ in range(length))
    return random_string


class Node:
    def __init__(self,key):
        self.key = key
        self.value = generate_random_string()


def recursive_linear_search(nodes, key, index=0):
    global times
    times += 1
    if index >= len(nodes): # 6 || 1
        return None  # 1 || 1
    if nodes[index].key == key: # 6 || 1
        return index  # 1 || 1
    else: # 1 || 1
        return recursive_linear_search(nodes, key, index + 1)


def recursive_linear_searchS(nodes, key, index=0):
    global space
    space += 1
```

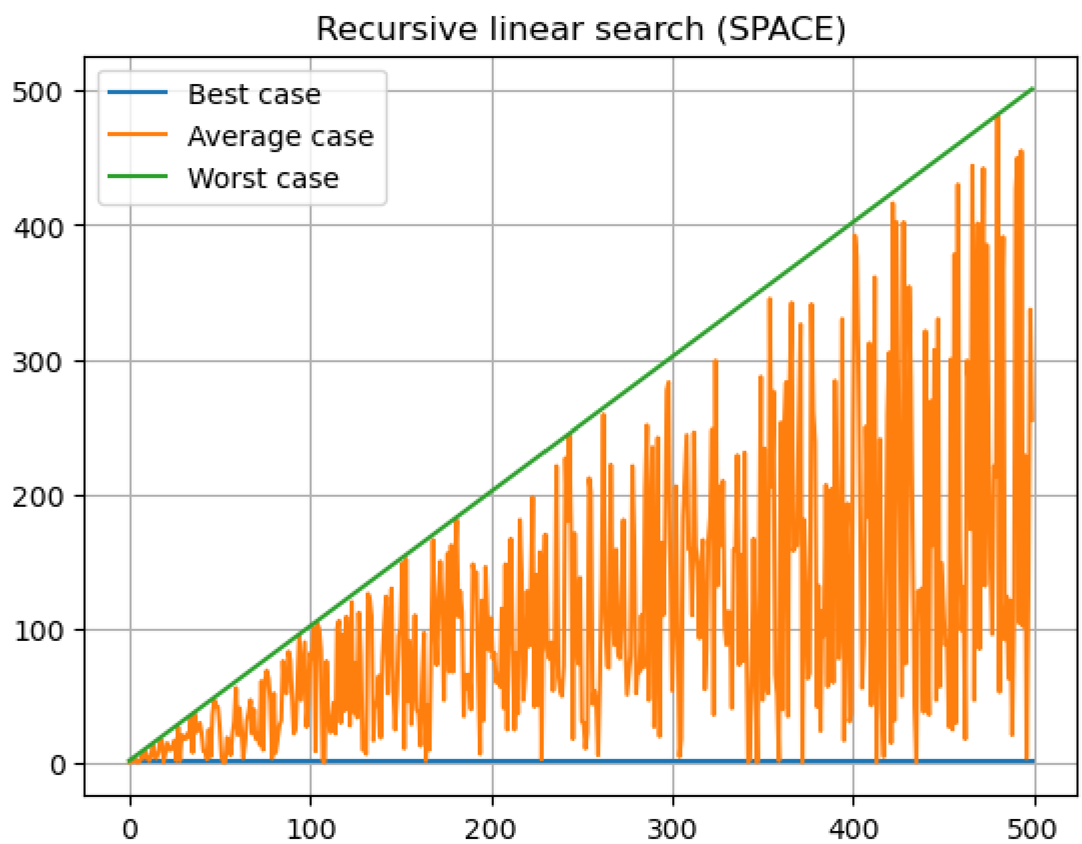
```
    if index >= len(nodes): # 6 || 1
        return None  # 1 || 1
    if nodes[index].key == key: # 6 || 1
        return index  # 1 || 1
    else: # 1 || 1
        return recursive_linear_searchS(nodes, key, index + 1)


if __name__ == '__main__':
    x = []
    y = []
    l = []
    n = 500
    for i in range(n):
        x.append(i)
        temp = Node (random.randint(-500,500))
        l.append(temp)
        recursive_linear_search(l,l[0].key)
        y.append(times)
        times = 0


    u = []
    v = []
    n = 500
    l = []
    for i in range(n):
```

```
    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    recursive_linear_search(l,averageCase.key)

    v.append(times)

    times = 0


a = []

b = []

n = 500

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    recursive_linear_search(l,1000)

    b.append(times)

    times = 0


plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()
```

```
plt.title('Recursive linear search (TIME)')

plt.grid(True)

plt.show()


x = []

y = []

l = []

n = 500

for i in range(n):

    x.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    recursive_linear_searchS(l,l[0].key)

    y.append(space)

    space = 0


u = []

v = []

n = 500

l = []

for i in range(n):

    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    index = random.randint(0,len(l)-1)
```

```
    averageCase = l[index]

    recursive_linear_searchS(l,averageCase.key)

    v.append(space)

    space = 0


a = []

b = []

n = 500

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    recursive_linear_searchS(l,1000)

    b.append(space)

    space = 0


plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()

plt.title('Recursive linear search (SPACE)')

plt.grid(True)

plt.show()
```


Recursive linear search (TIME)
Best case
Average case
Worst case
500
400
300
200
100
0
0
100
200
300
400
500

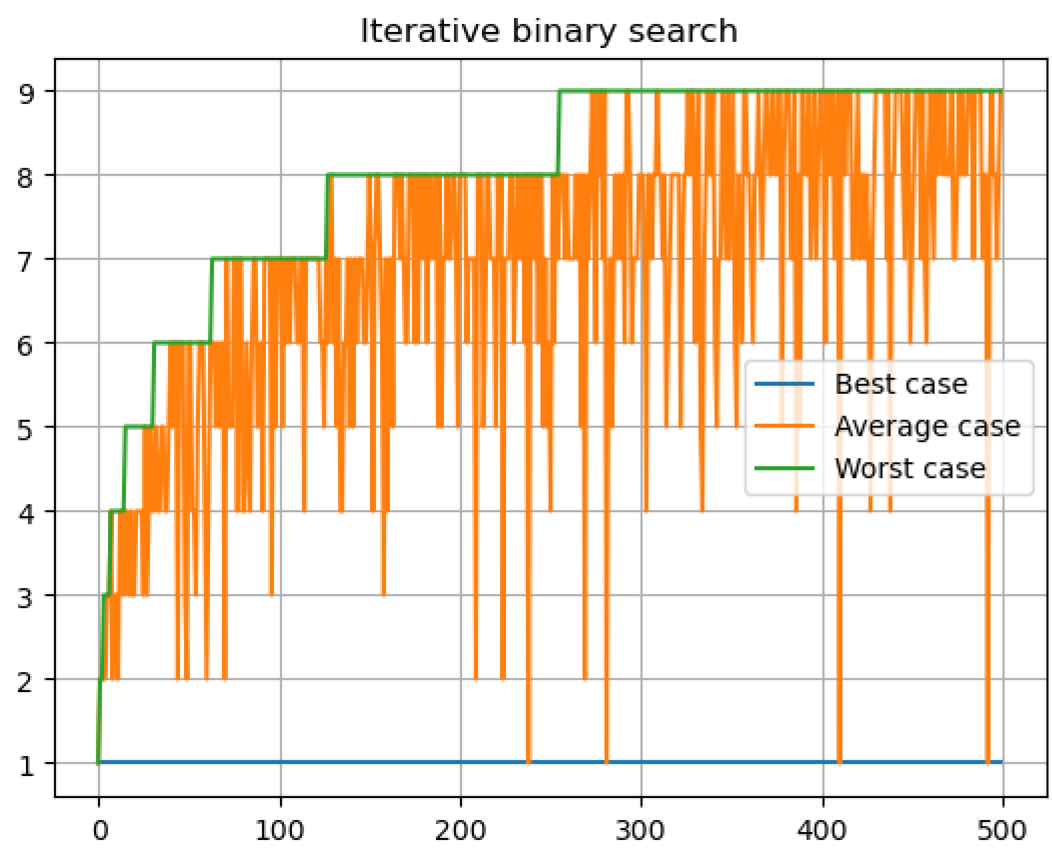



```
import random

import string

import matplotlib.pyplot as plt


def generate_random_string(length=16):

    characters = string.ascii_letters + string.digits

        random_string   =   ''.join(random.choice(characters)   for   _   in
range(length))

    return random_string


class Node:

    def __init__(self,key):
```

```
        self.key = key

        self.value = generate_random_string()


def binary_search(key_to_find,nodes):

    times = 0

    left, right = 0, len(nodes) - 1 # 6 || 2

    while left <= right: #  4 (log n + 1) || 1

        times += 1

        mid = (left + right) // 2 # 5 (log n) || 1

        if nodes[mid].key == key_to_find: # 6 (log n)|| 1

            return times # 1 || 1

        elif nodes[mid].key < key_to_find: # 6 (log n) || 1

            left = mid + 1 # Constantly dividing by two 4(log n) || 1

        else:

                right = mid - 1 # # Constantly dividing by two (by 4 in accumnulate) 8(log n) || 1

    return times # 1 || 1


if __name__ == "__main__":

    x = []

    y = []

    l = []

    n = 500

    for i in range(n):

        x.append(i)
```

```
    temp = Node(i)

    l.append(temp)

    y.append(binary_search(l[(len(l)-1)//2].key,l))


u = []

v = []

n = 500

l = []

for i in range(n):

    u.append(i)

    temp = Node (i)

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    v.append(binary_search(averageCase.key,l))


a = []

b = []

n = 500

l = []

for i in range(n):

    a.append(i)

    temp = Node (i)

    l.append(temp)

    b.append(binary_search(1000,l))
```

```
plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()

plt.title('Iterative binary search')

plt.grid(True)

plt.show()
```




```
import random
```

```
import string
import matplotlib.pyplot as plt


times = 0
space = 0


def generate_random_string(length=16):
    characters = string.ascii_letters + string.digits
        random_string = ''.join(random.choice(characters) for _ in
range(length))
    return random_string


class Node:
    def __init__(self,key):
        self.key = key
        self.value = generate_random_string()


def recursive_binary_search(nodes, key, left=0, right=None):
    global times
    times += 1
    if right is None: # 3 || 1
        right = len(nodes) - 1 # 4 || 1

    if left > right: # 4 || 1
        return None # 1 || 1
```

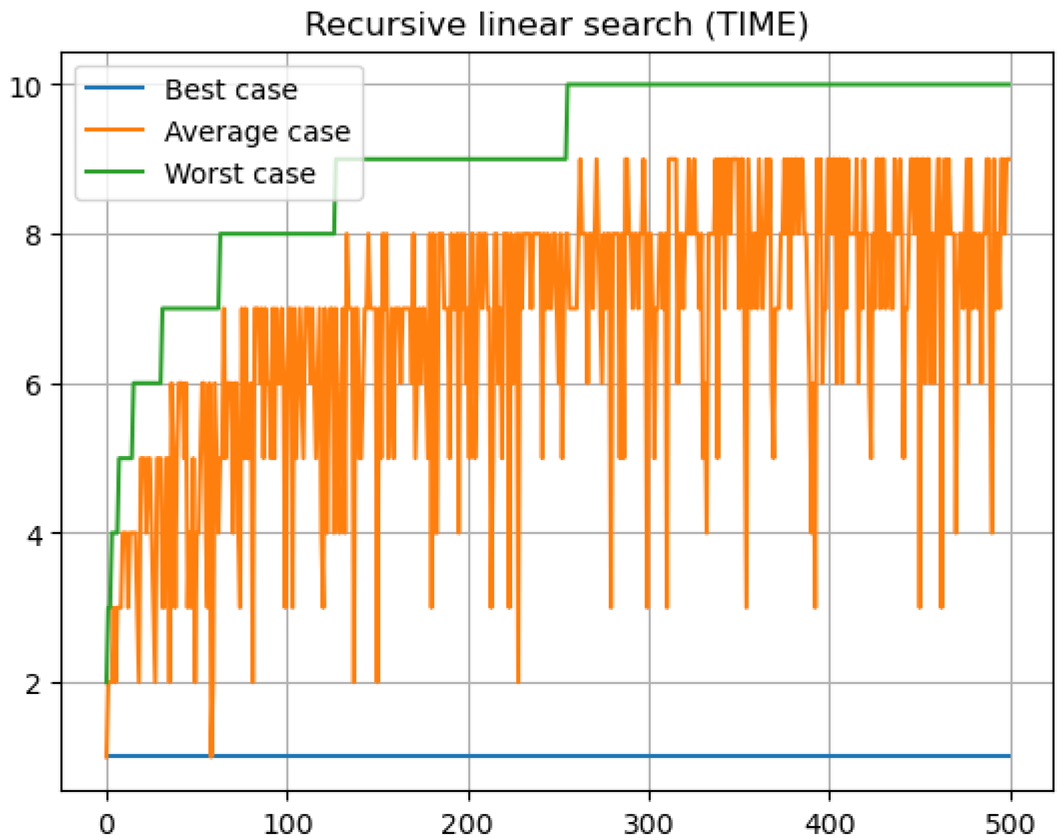
```
    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_search(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_search(nodes, key, left, mid - 1)


def recursive_binary_searchS(nodes, key, left=0, right=None):

    global space

    space += 1

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1


    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_searchS(nodes, key, mid + 1, right)
```

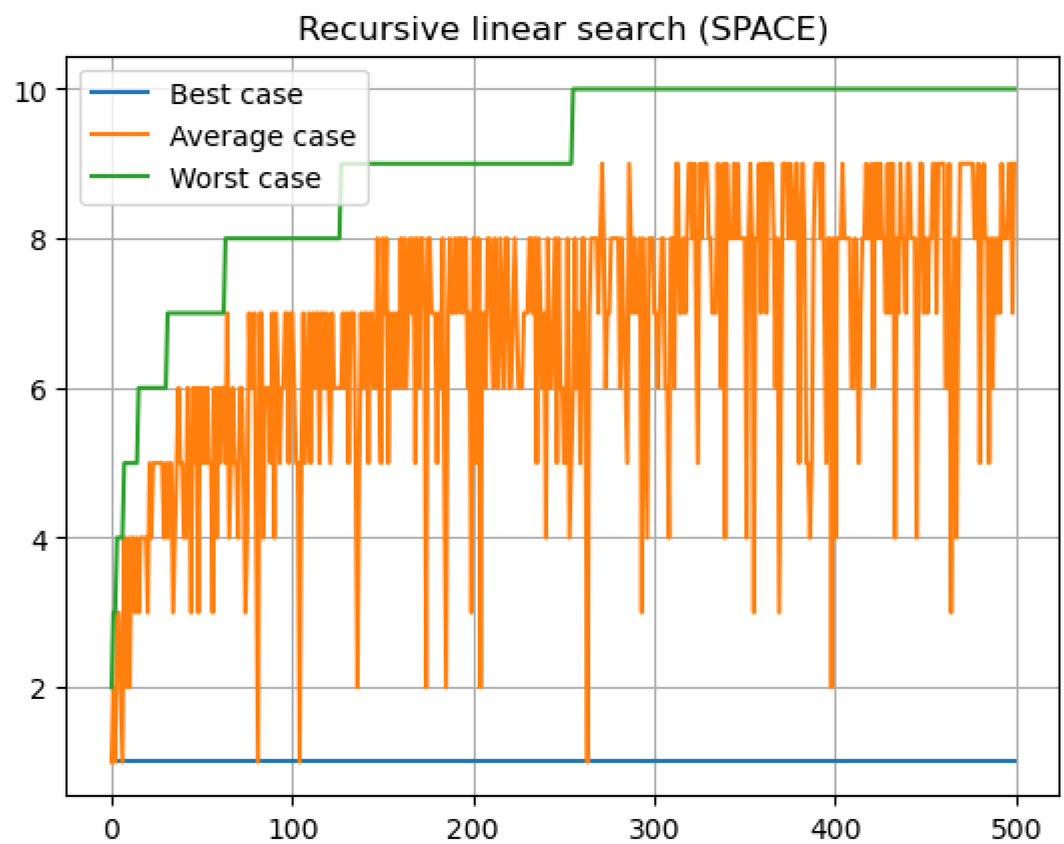
```
        # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_searchS(nodes, key, left, mid - 1)



if __name__ == '__main__':

    x = []

    y = []

    l = []

    n = 500

    for i in range(n):

        x.append(i)

        temp = Node (i)

        l.append(temp)

        recursive_binary_search(l,l[(len(l)-1)//2].key)

        y.append(times)

        times = 0


    u = []

    v = []

    n = 500

    l = []

    for i in range(n):

        u.append(i)
```

```
    temp = Node (i)

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    recursive_binary_search(l,averageCase.key)

    v.append(times)

    times = 0


a = []

b = []

n = 500

l = []

for i in range(n):

    a.append(i)

    temp = Node (i)

    l.append(temp)

    recursive_binary_search(l,1000)

    b.append(times)

    times = 0


plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()

plt.title('Recursive linear search (TIME)')
```

```
plt.grid(True)

plt.show()


x = []

y = []

l = []

n = 500

for i in range(n):

    x.append(i)

    temp = Node (i)

    l.append(temp)

    recursive_binary_searchS(l,l[(len(l)-1)//2].key)

    y.append(space)

    space = 0


u = []

v = []

n = 500

l = []

for i in range(n):

    u.append(i)

    temp = Node (i)

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]
```

```
    recursive_binary_searchS(l,averageCase.key)
    v.append(space)
    space = 0


a = []
b = []
n = 500
l = []
for i in range(n):
    a.append(i)
    temp = Node (i)
    l.append(temp)
    recursive_binary_searchS(l,1000)
    b.append(space)
    space = 0


plt.plot(x,y,label = "Best case")
plt.plot(u,v,label = "Average case")
plt.plot(a,b, label = "Worst case")
plt.legend()
plt.title('Recursive linear search (SPACE)')
plt.grid(True)
plt.show()
```


Recursive linear search (TIME)
Best case
Average case
Worst case
10
8
6
4
2
0
100
200
300
400
500

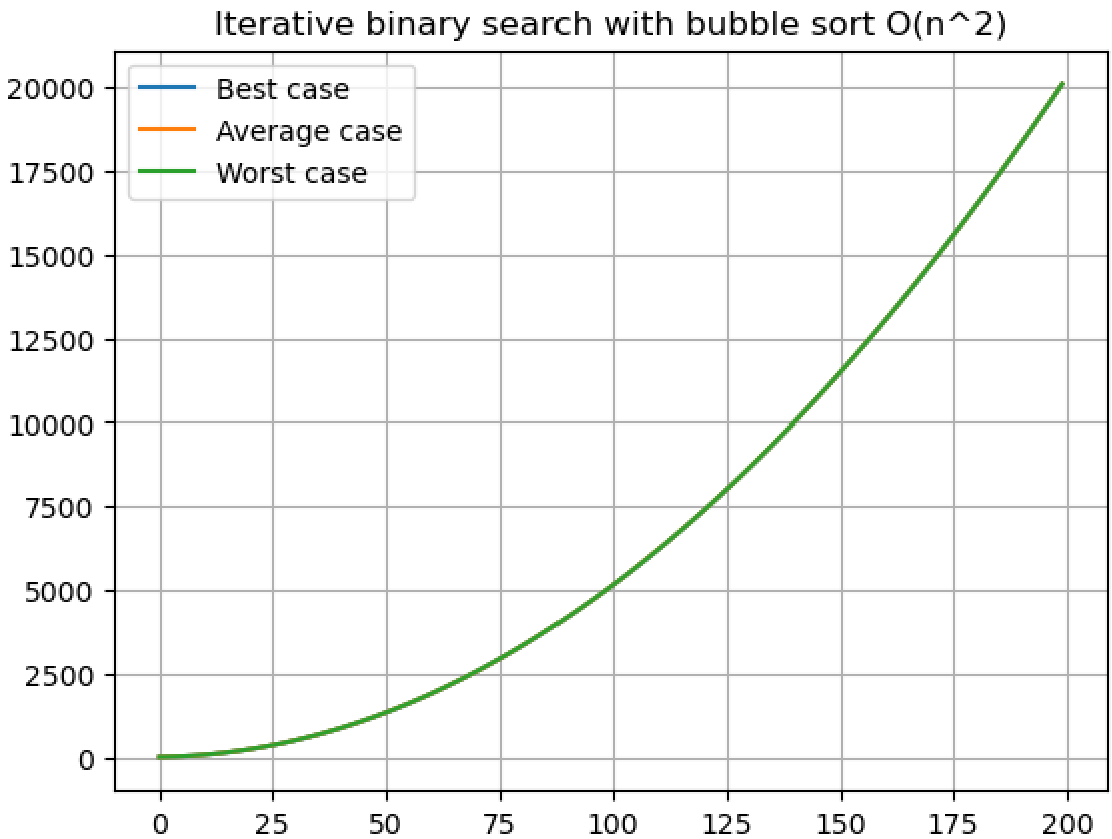



```
import random

import string

import matplotlib.pyplot as plt


def generate_random_string(length=16):

    characters = string.ascii_letters + string.digits

        random_string = ''.join(random.choice(characters) for _ in range(length))

    return random_string


class Node:

    def __init__(self,key):
```

```
        self.key = key

        self.value = generate_random_string()


def bubbleSortNodes(node_list):

    times = 0

    n = len(node_list)

    i = 0

    while i < n:

        times +=1

        j = 0

        while j < n - i - 1:

            times += 1

            if node_list[j].key > node_list[j + 1].key:

                node_list[j], node_list[j + 1] = node_list[j + 1], node_list[j]

            j += 1

        i += 1

    return times


def binary_search(key_to_find,nodes):

    times = 0

    left, right = 0, len(nodes) - 1 # 6 || 2

    while left <= right: #  4 (log n + 1) || 1

        times += 1

        mid = (left + right) // 2 # 5 (log n) || 1
```

```
        if nodes[mid].key == key_to_find: # 6 (log n)|| 1
            return times # 1 || 1
        elif nodes[mid].key < key_to_find: # 6 (log n) || 1
            left = mid + 1 # Constantly dividing by two 4(log n) || 1
        else:
                right = mid - 1 # # Constantly dividing by two (by 4 in
accumnulate) 8(log n) || 1
    return times # 1 || 1


if __name__ == "__main__":
    x = []
    y = []
    l = []
    n = 200
    for i in range(n):
        x.append(i)
        temp = Node(random.randint(-500,500))
        l.append(temp)
                                            y.append(bubbleSortNodes(l)     +
binary_search(l[(len(l)-1)//2].key,l))


    u = []
    v = []
    n = 200
    l = []
    for i in range(n):
```

```
    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    v.append(bubbleSortNodes(l)+binary_search(averageCase.key,l))


a = []

b = []

n = 200

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    b.append(bubbleSortNodes(l) + binary_search(1000,l))


plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()

plt.title('Iterative binary search with bubble sort O(n^2)')

plt.grid(True)

plt.show()
```




```
import random

import string

import matplotlib.pyplot as plt


times = 0

space = 0


def generate_random_string(length=16):

    characters = string.ascii_letters + string.digits

        random_string  =  ''.join(random.choice(characters)  for  _  in
range(length))

    return random_string
```

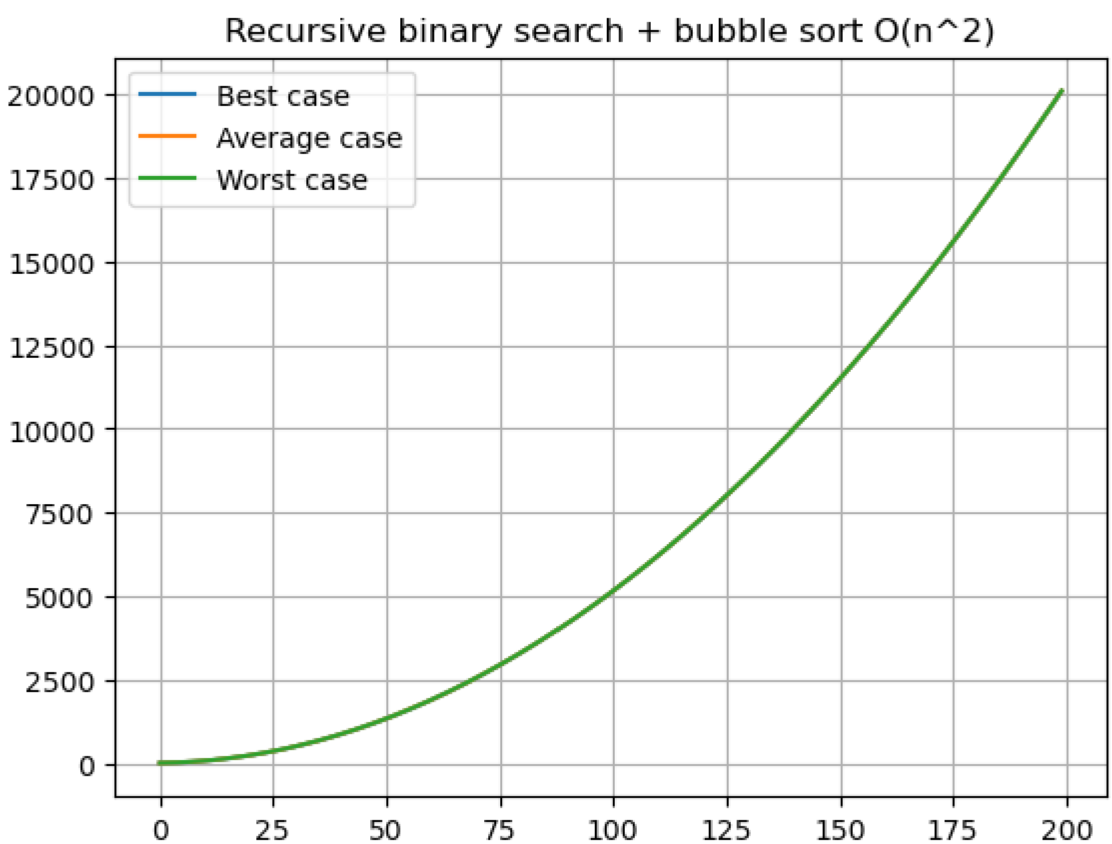
```python
class Node:

    def __init__(self,key):

        self.key = key

        self.value = generate_random_string()


def recursive_binary_search(nodes, key, left=0, right=None):

    global times

    times += 1

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1


    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_search(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_search(nodes, key, left, mid - 1)
```

```
def recursive_binary_searchS(nodes, key, left=0, right=None):

    global space

    space += 1

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1


    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_searchS(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_searchS(nodes, key, left, mid - 1)


def bubbleSortNodes(node_list):

    time = 0

    n = len(node_list)

    i = 0

    while i < n:

        time +=1
```

```
        j = 0
        while j < n - i - 1:
            time += 1
            if node_list[j].key > node_list[j + 1].key:
                node_list[j], node_list[j + 1] = node_list[j + 1],
node_list[j]
            j += 1
        i += 1
    return time



if __name__ == '__main__':
    x = []
    y = []
    l = []
    n = 200
    for i in range(n):
        x.append(i)
        temp = Node (random.randint(-500,500))
        l.append(temp)
        bubbleSortNodes(l)
        recursive_binary_search(l,l[(len(l)-1)//2].key)
        y.append(times + bubbleSortNodes(l))
        times = 0
```

```
u = []
v = []
n = 200
l = []
for i in range(n):
    u.append(i)
    temp = Node (random.randint(-500,500))
    l.append(temp)
    index = random.randint(0,len(l)-1)
    averageCase = l[index]
    bubbleSortNodes(l)
    recursive_binary_search(l,averageCase.key)
    v.append(times + bubbleSortNodes(l))
    times = 0


a = []
b = []
n = 200
l = []
for i in range(n):
    a.append(i)
    temp = Node (random.randint(-500,500))
    l.append(temp)
    bubbleSortNodes(l)
    recursive_binary_search(l,1000)
```

```
        b.append(times + bubbleSortNodes(l))

        times = 0


    plt.plot(x,y,label = "Best case")

    plt.plot(u,v,label = "Average case")

    plt.plot(a,b, label = "Worst case")

    plt.legend()

    plt.title('Recursive binary search + bubble sort O(n^2)')

    plt.grid(True)

    plt.show()
```




```
import random
```

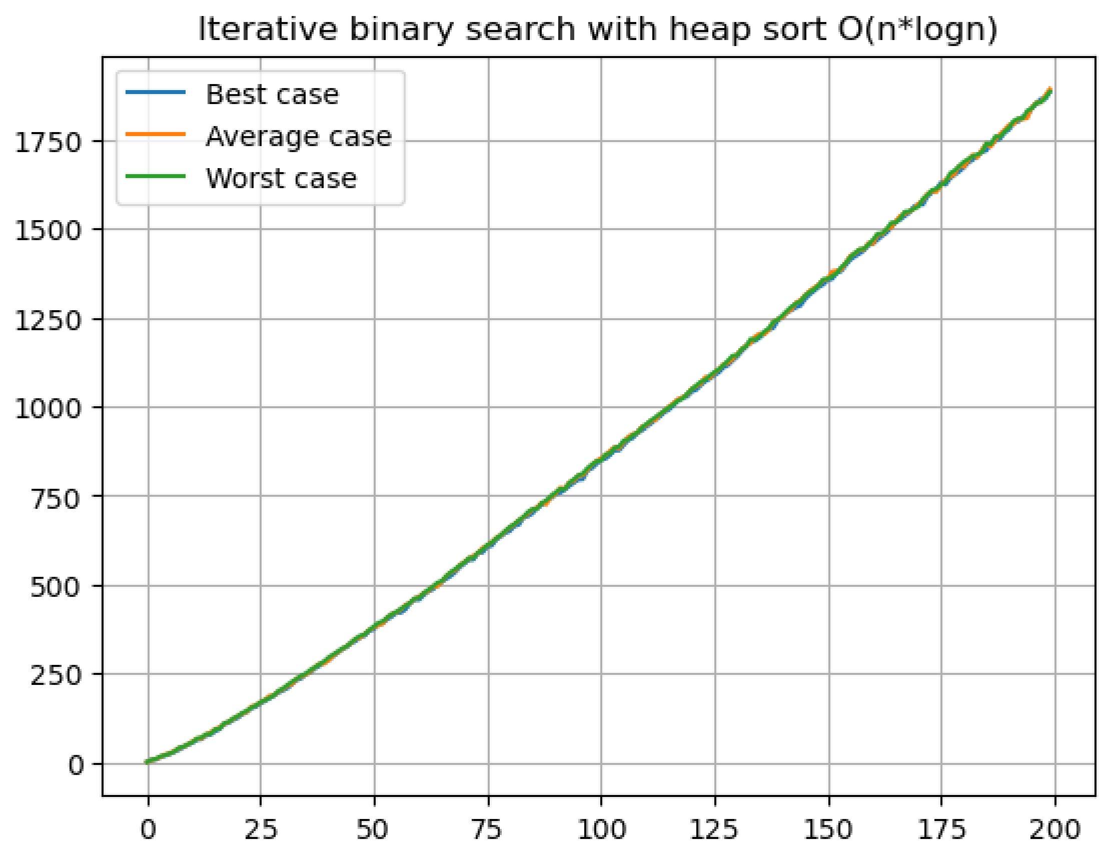
```
import string
import matplotlib.pyplot as plt

time = 0

def generate_random_string(length=16):
    characters = string.ascii_letters + string.digits
        random_string = ''.join(random.choice(characters) for _ in
range(length))
    return random_string

class Node:
    def __init__(self,key):
        self.key = key
        self.value = generate_random_string()

def maxHeapNodes(arr, n):
    global time
    for i in range(n // 2 - 1, -1, -1):
        time +=1
        heapifyNodes(arr, n, i)

def heapifyNodes(arr, n, i):
    global time
    time +=1
```

```
    largest = i

    left_child = 2 * i + 1

    right_child = 2 * i + 2


    if left_child < n and arr[left_child].key > arr[largest].key:

        largest = left_child


    if right_child < n and arr[right_child].key > arr[largest].key:

        largest = right_child


    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        heapifyNodes(arr, n, largest)


def heapSortNodes(arr):

    global time

    n = len(arr)


    maxHeapNodes(arr, n)


    for i in range(n - 1, 0, -1):

        time +=1

        arr[0], arr[i] = arr[i], arr[0]

        heapifyNodes(arr, i, 0)
```

```
def binary_search(key_to_find,nodes):
    times = 0
    left, right = 0, len(nodes) - 1 # 6 || 2
    while left <= right: #  4 (log n + 1) || 1
        times += 1
        mid = (left + right) // 2 # 5 (log n) || 1
        if nodes[mid].key == key_to_find: # 6 (log n)|| 1
            return times # 1 || 1
        elif nodes[mid].key < key_to_find: # 6 (log n) || 1
            left = mid + 1 # Constantly dividing by two 4(log n) || 1
        else:
            right = mid - 1 # # Constantly dividing by two (by 4 in accumnulate) 8(log n) || 1
    return times # 1 || 1


if __name__ == "__main__":
    x = []
    y = []
    l = []
    n = 200
    for i in range(n):
        x.append(i)
        temp = Node(random.randint(-500,500))
        l.append(temp)
```

```
    heapSortNodes(l)

    y.append(time + binary_search(l[(len(l)-1)//2].key,l))

    time = 0


u = []

v = []

n = 200

l = []

for i in range(n):

    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    heapSortNodes(l)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    v.append(time+binary_search(averageCase.key,l))

    time = 0


a = []

b = []

n = 200

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))
```

```
        l.append(temp)

        heapSortNodes(l)

        b.append(time + binary_search(1000,l))

        time = 0


    plt.plot(x,y,label = "Best case")

    plt.plot(u,v,label = "Average case")

    plt.plot(a,b, label = "Worst case")

    plt.legend()

    plt.title('Iterative binary search with heap sort O(n*logn)')

    plt.grid(True)

    plt.show()
```




```
import random

import string

import matplotlib.pyplot as plt


times = 0

space = 0

time = 0


def generate_random_string(length=16):

    characters = string.ascii_letters + string.digits

        random_string   =   ''.join(random.choice(characters)   for   _   in
range(length))
```

```
    return random_string


class Node:

    def __init__(self,key):

        self.key = key

        self.value = generate_random_string()


def recursive_binary_search(nodes, key, left=0, right=None):

    global times

    times += 1

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1


    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_search(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_search(nodes, key, left, mid - 1)
```

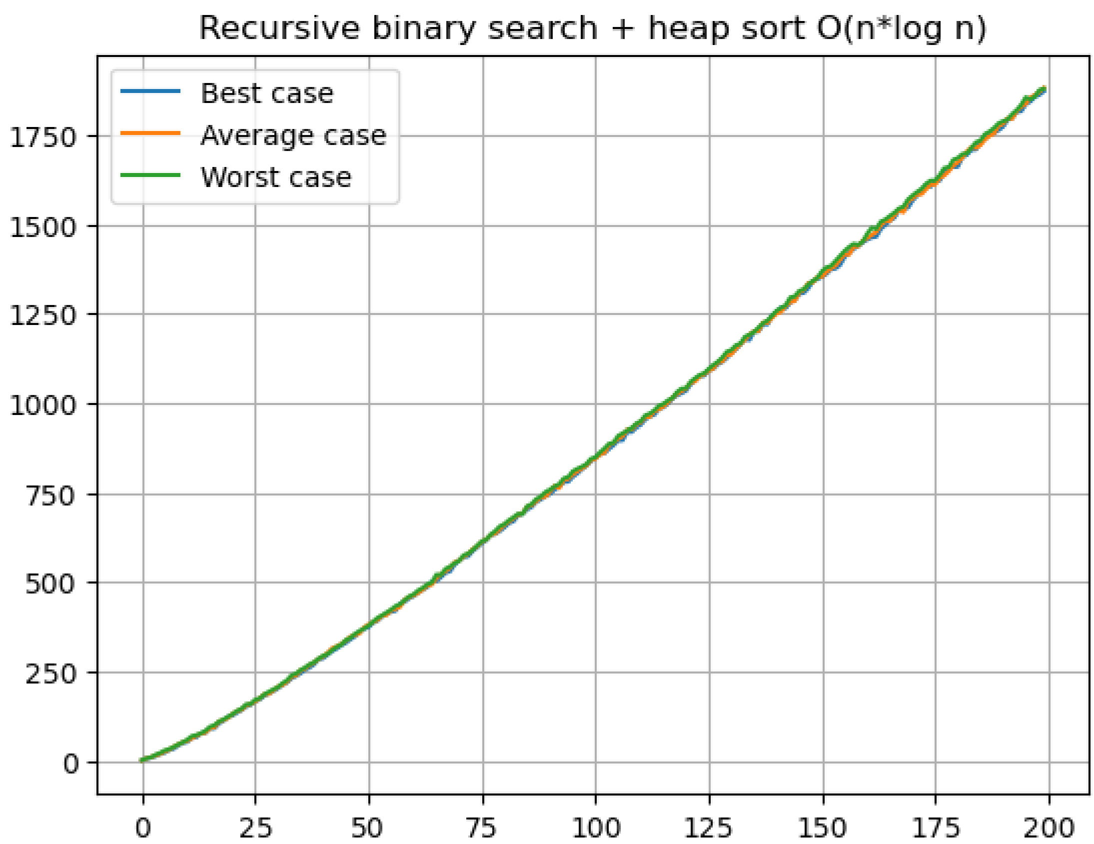
```
def recursive_binary_searchS(nodes, key, left=0, right=None):

    global space

    space += 1

    if right is None: # 3 || 1

        right = len(nodes) - 1 # 4 || 1


    if left > right: # 4 || 1

        return None # 1 || 1


    mid = (left + right) // 2 # 7 || 1

    if nodes[mid].key == key: # 5 || 1

        return mid # 1 || 1

    elif nodes[mid].key < key: # 5 || 1

        return recursive_binary_searchS(nodes, key, mid + 1, right)

         # log n recursive calls, multiplies all the function  6(log n) ||
log n

    else: # 1 || 1

        return recursive_binary_searchS(nodes, key, left, mid - 1)


def maxHeapNodes(arr, n):

    global time

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

        time +=1

        heapifyNodes(arr, n, i)
```

```
def heapifyNodes(arr, n, i):
    global time
    time +=1
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < n and arr[left_child].key > arr[largest].key:
        largest = left_child

    if right_child < n and arr[right_child].key > arr[largest].key:
        largest = right_child

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

def heapSortNodes(arr):
    global time
    n = len(arr)

    maxHeapNodes(arr, n)

    for i in range(n - 1, 0, -1):
```

```
        time +=1
        arr[0], arr[i] = arr[i], arr[0]
        heapifyNodes(arr, i, 0)



if __name__ == '__main__':
    x = []
    y = []
    l = []
    n = 200
    for i in range(n):
        x.append(i)
        temp = Node (random.randint(-500,500))
        l.append(temp)
        heapSortNodes(l)
        recursive_binary_search(l,l[(len(l)-1)//2].key)
        y.append(times + time)
        times = 0
        time = 0

    u = []
    v = []
    n = 200
    l = []
    for i in range(n):
```

```
    u.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    index = random.randint(0,len(l)-1)

    averageCase = l[index]

    heapSortNodes(l)

    recursive_binary_search(l,averageCase.key)

    v.append(times + time)

    times = 0

    time = 0


a = []

b = []

n = 200

l = []

for i in range(n):

    a.append(i)

    temp = Node (random.randint(-500,500))

    l.append(temp)

    heapSortNodes(l)

    recursive_binary_search(l,1000)

    b.append(times + time)

    times = 0

    time = 0
```

```
plt.plot(x,y,label = "Best case")

plt.plot(u,v,label = "Average case")

plt.plot(a,b, label = "Worst case")

plt.legend()

plt.title('Recursive binary search + heap sort O(n*log n)')

plt.grid(True)

plt.show()
```



Conclusion: Both searches are very useful depending on the context of the problem we're solving, the difficulty of this practice is evaluating all sorting + binary search cases for determining which algorithm we should use.

As a competitive programmer, I use binary search very frequently, because the input is often sorted, and for handling queries, binary search is better than linear, but in a real world problem we have to choose wisely the algorithm to use.