-CS 315-
Project Report

Done by:
Dana AL-Duayji 421202227
Emtenan AL-fozan 421215099
Shrooq AL-Qaied 4212202213

Supervised by Ms. Norah Alosily

Table of Contents

03

MERGE SORT ALGORITHM

by Dana AL-Duayji

-Implementation

```
[351] import timeit
      import numpy
      import pandas as pd
      import plotly.express as px
```

```
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
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)
```

04

MERGE SORT ALGORITHM

creating an array of random numbers and call merge sort

```
[353] def randomarray(siz):
        numarray = numpy.random.randint(1,100,siz)
        n = len(numarray)
        print(numarray)
        mergeSort(numarray,0,n-1)
        print(numarray)
```

creating an array of ascending numbers and call merge sort

```
[354] def ascendingarray(siz):
        numarray = numpy.random.randint(1,100,siz)
        n = len(numarray)
        sortednumarray=sorted(numarray)
        print(sortednumarray)
        mergeSort(sortednumarray,0,n-1)
        print(sortednumarray)
```

creating an array of ascending numbers and call merge sort

```
[355] def descendingarray(siz):
        numarray = numpy.random.randint(1,100,siz)
        n = len(numarray)
        desnumarray = sorted(numarray , reverse = True)
        print(desnumarray)
        mergeSort(desnumarray,0,n-1)
        print(desnumarray)
```

array of 100

```
[356] st1 = timeit.default_timer()
      randomarray(100)
      end1= (timeit.default_timer()-st1)*1000
      print("time taken for sorting is " , (end1) , "milliseconds " )


      st2 = timeit.default_timer()
      ascendingarray(100)
      end2= (timeit.default_timer()-st2)*1000
      print("time taken for sorting is " , (end2) , "milliseconds " )

      st3 = timeit.default_timer()
      descendingarray(100)
      end3= (timeit.default_timer()-st3)*1000
      print("time taken for sorting is " , (end3) , "milliseconds ")
```

05

MERGE SORT ALGORITHM

array of 1000

```
[357] stt1 = timeit.default_timer()
      randomarray(1000)
      endd1= (timeit.default_timer()-stt1)*1000
      print("time taken for sorting is " , (endd1) , "milliseconds " )

      stt2 = timeit.default_timer()
      ascendingarray(1000)
      endd2= (timeit.default_timer()-stt2)*1000
      print("time taken for sorting is " , (endd2) , "milliseconds " )

      stt3 = timeit.default_timer()
      descendingarray(1000)
      endd3= (timeit.default_timer()-stt3)*1000
      print("time taken for sorting is " , (endd3) , "milliseconds " )
```

array of 10000

```
      sttt1 = timeit.default_timer()
      randomarray(10000)
      enddd1= (timeit.default_timer()-sttt1)*1000
      print("time taken for sorting is " , (enddd1) , "milliseconds " )

      sttt2 = timeit.default_timer()
      ascendingarray(10000)
      enddd2= (timeit.default_timer()-sttt2)*1000
      print("time taken for sorting is " , (enddd2) , "milliseconds " )

      sttt3 = timeit.default_timer()
      descendingarray(10000)
      enddd3= (timeit.default_timer()-sttt3)*1000
      print("time taken for sorting is " , (enddd3) , "milliseconds " )
```

06

MERGE SORT ALGORITHM

result table

```
data = {'n': [100, 100, 100, 1000, 1000, 1000, 10000, 10000, 10000],
        'case': ['random',  'ascending', 'descending', 'random',  'ascending', 'descending', 'random',  'ascending', 'descending'],
        'time': [end1, end2, end3, endd1, endd2, endd3, enddd3, enddd2, enddd3]}


df = pd.DataFrame(data)

df
```

result chart

```
[ ]  fig = px.line(data, x='n', y='time', color='case')
     fig.update_layout(xaxis_type='category')
     fig.update_traces(mode='markers+lines')
     fig.show()
```

Source Code: Click Here

07

MERGE SORT ALGORITHM

-Execution Time

SIZE=100	time taken for sorting random array is 1.1328909999974712 milliseconds time taken for sorting ascending array is 0.5772740000793419 milliseconds time taken for sorting descending is 0.5054270000073302 milliseconds
SIZE=1000	time taken for sorting random array is 7.800822999797674 milliseconds time taken for sorting ascending array is 3.968220999922778 milliseconds time taken for sorting descending aaray is 4.098275000160356 milliseconds
SIZE=10000	time taken for sorting random array is 74.12565999993603 milliseconds time taken for sorting ascending array is 50.66710500022964 milliseconds time taken for sorting descending array is 55.013711000356125 milliseconds

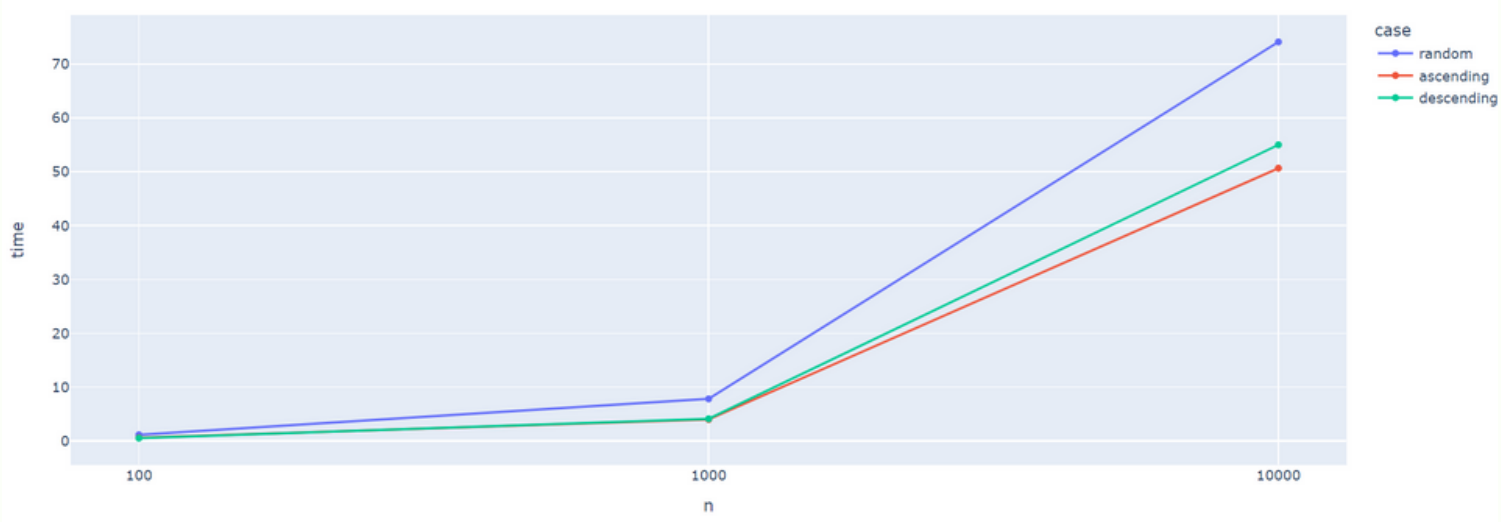
08

MERGE SORT ALGORITHM

-Result Table

	n	case	time
0	100	random	1.132891
1	100	ascending	0.577274
2	100	descending	0.505427
3	1000	random	7.800823
4	1000	ascending	3.968221
5	1000	descending	4.098275
6	10000	random	74.125660
7	10000	ascending	50.667105
8	10000	descending	55.013711

-Result Chart



09

MERGE SORT ALGORITHM

-Analyzation

To conclude, merge sort is a comparison-based sorting algorithm that divides an input array into smaller sub-arrays, compares them and then sorts those sub-arrays, and then merges them back together to produce a sorted array.

The time complexity of merge sort is O(n log n) in all cases, making it efficient for sorting large datasets. This O(n log n) complexity is the same for best, average, and worst-case scenarios. Here's why:

- Merge sort follows a divide-and-conquer strategy, where the array is repeatedly divided into two halves until you have sub-arrays of size 1. The time complexity for this division step is O(log n), where "n" is the size of the initial array. Then, it merges these sub-arrays in a sorted manner. So , regardless of the initial order of the array, this process will continue with the same time complexity.

- It compares and combines two sorted sub-arrays in a way that ensures the resulting merged array is sorted. This merge operation takes O(n) time for any input, which is independent of the order of the elements.

BST ALGORITHM

-Implementation

```
[199] import time
      import random
      import numpy as np
      import matplotlib.pyplot as plt
      import pandas as pd
      import plotly.express as px
```

```
[200] class Node:
          def __init__(self, key):
              self.left = None
              self.right = None
              self.val = key
```

Definition of the BinarySearchTree class,operation

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if node is None:
            return Node(key)

        if key < node.val:
            node.left = self._insert(node.left, key)
        else:
            node.right = self._insert(node.right, key)

        return node

    def search(self, key):
        return self._search(self.root, key)

    def _search(self, node, key):
        if node is None or node.val == key:
            return node
        if key < node.val:
            return self._search(node.left, key)
        return self._search(node.right, key)

    def inorder_traversal(self):
        if self.root:
            self._inorder_traversal(self.root)

    def _inorder_traversal(self, root):
```

```
        if root:
            self._inorder_traversal(root.left)
            print(root.val, end=" ")
            self._inorder_traversal(root.right)

    def postorder_traversal(self):
        if self.root:
            self._postorder_traversal(self.root)

    def _postorder_traversal(self, root):
        if root:
            self._postorder_traversal(root.left)
            self._postorder_traversal(root.right)
            print(root.val, end=" ")

    def delete(self, key):
        if self.root is None:
            return self.root
        if key < self.root.val:
            self.root.left = self.delete(self.root.left, key)
        elif key > self.root.val:
            self.root.right = self.delete(self.root.right, key)
        else:
            if self.root.left is None:
                return self.root.right
            elif self.root.right is None:
                return self.root.left
            self.root.val = self.find_min(self.root.right)
            self.root.right = self.delete(self.root.right, self.root.val)
        return self.root

    def find_min(self, root):
        while root.left:
            root = root.left
        return root.val
```

Function to measure the runtime of BST search

```
[202] def search_runtime(bst, number):
          start = time.time()
          bst.search(number)
          end = time.time()
          return (end - start) * 1000
```

Function to convert a sorted array to a balanced BST

```
[203] def sortedArrayToBST(arr):
          if not arr:
```

```
            return None

        mid = len(arr) // 2
        root = Node(arr[mid])
        root.left = sortedArrayToBST(arr[:mid])
        root.right = sortedArrayToBST(arr[mid+1:])
        return root
```

Function to insert numbers to a BST

```
[204] def add_numbers_to_bst(numbers, bst):
          for num in numbers:
              bst.insert(num)
```

Function to calculate average runtime over multiple runs

```
[205] def calc_avg_runtime(case_runtime):
          runs = 50
          total_time = 0
          for i in range(runs):
              total_time += case_runtime()
          return total_time / runs
```

Generate random data for each size

```
[206] size1 = 50
      size2 = 150
      size3 = 500

      data1 = np.random.randint(1, 1000, size=size1)
      data2 = np.random.randint(1, 1000, size=size2)
      data3 = np.random.randint(1, 1000, size=size3)
```

Creating a best-case BST where the tree is balanced

```
[207] print(data1)
      #print(data2)
      #print(data3)

      bst_balanced = BinarySearchTree()
      start = time.time()
      bst_balanced.root = sortedArrayToBST(sorted(data1))
      end = time.time()
      best_case_50=(end - start) * 1000
```

Connected to Python 3 Google Compute Engine backend

BST ALGORITHM

```
bst_balanced3 = BinarySearchTree()
start = time.time()
bst_balanced3.root = sortedArrayToBST(sorted(data3))
end = time.time()
best_case_500=(end - start) * 1000

print("Elements in-order for bst_balanced:")
bst_balanced.inorder_traversal()
print()
print("Elements post-order for bst_balanced:")
bst_balanced.postorder_traversal()
print()
```

```
[785  78 426 501 944 639  15 475 374 460 899 876 458 878 741 112   2 208
 963 739 266 296 346 388 237 253 206  90 330 920 677 931 581 169 348 991
 188 204 190 348 909 672 378 553 390 348 511 568 899 442]
Elements in-order for bst_balanced:
2 15 78 90 112 169 188 190 204 206 208 237 253 266 296 330 346 348 348 348 374 378 388 390 426 442 458 460 475 501 511 553 568 581 639 672 677 739 741 785 876 878 899 899 909 920 931 944 963 991
Elements post-order for bst_balanced:
2 78 15 112 169 90 190 204 208 237 206 188 266 330 296 348 348 346 374 378 390 426 388 348 253 458 475 460 511 553 501 581 639 677 739 672 568 785 876 899 899 878 920 931 963 991 944 909 741 442
```

Creating an average-case BST where the tree has random input

```
[208] bst_avg1 = BinarySearchTree()
      start = time.time()
      add_numbers_to_bst(data1, bst_avg1)
      end = time.time()
      Average_case50 =(end - start) * 1000

      bst_avg2 = BinarySearchTree()
      start = time.time()
      add_numbers_to_bst(data2, bst_avg2)
      end = time.time()
      Average_case150 =(end - start) * 1000

      bst_avg3 = BinarySearchTree()
      start = time.time()
      add_numbers_to_bst(data3, bst_avg3)
      end = time.time()
      Average_case500 =(end - start) * 1000
```

Creating a worst-case BST where the tree is unbalanced

```
[209] bst_worst1 = BinarySearchTree()
      data1.sort()
      start = time.time()
      add_numbers_to_bst(data1, bst_worst1)
      end = time.time()
```

```
      Worst_case50 =(end - start) * 1000
239]
      bst_worst2 = BinarySearchTree()
      data2.sort()
      start = time.time()
      add_numbers_to_bst(data2, bst_worst2)
      end = time.time()
      Worst_case150 =(end - start) * 1000

      bst_worst3 = BinarySearchTree()
      data3.sort()
      start = time.time()
      add_numbers_to_bst(data3, bst_worst3)
      end = time.time()
      Worst_case500 =(end - start) * 1000
```

Calculating average runtimes for best, average, and worst cases

```
240] best_case_avg_runtime50 = calc_avg_runtime(lambda: best_case_50)
     best_case_avg_runtime150 = calc_avg_runtime(lambda: best_case_150)
     best_case_avg_runtime500 = calc_avg_runtime(lambda: best_case_500)

     average_case_avg_runtime50 = calc_avg_runtime(lambda: Average_case50)
     average_case_avg_runtime150 = calc_avg_runtime(lambda: Average_case150)
     average_case_avg_runtime500 = calc_avg_runtime(lambda: Average_case500)

     worst_case_avg_runtime50 = calc_avg_runtime(lambda: Worst_case50)
     worst_case_avg_runtime150 = calc_avg_runtime(lambda: Worst_case150)
     worst_case_avg_runtime500 = calc_avg_runtime(lambda: Worst_case500)
```

```
240]
```

Printing average runtimes for different cases

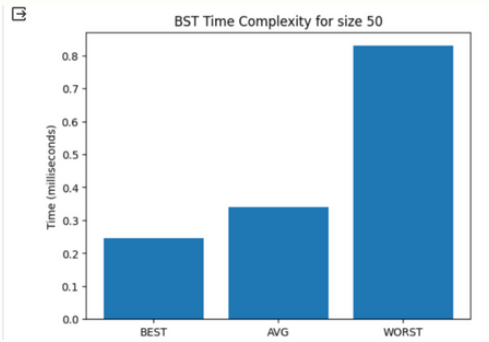
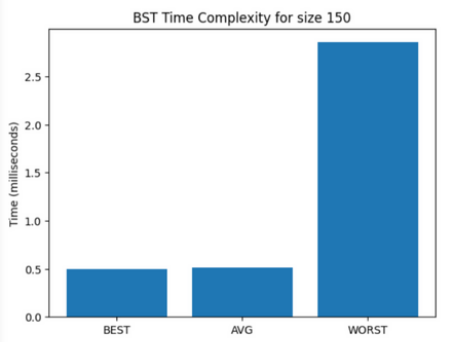
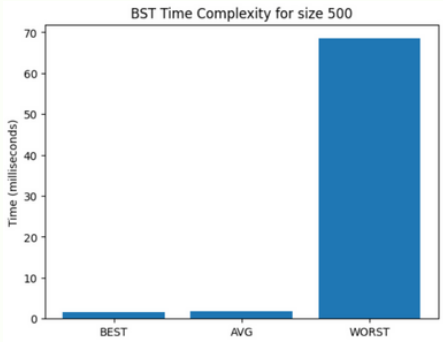
```
print("Best Case average Runtime for size=50:", best_case_avg_runtime50)
print("Average Case average Runtime for size=50 :", average_case_avg_runtime50)
print("Worst Case average Runtime for size=50:", worst_case_avg_runtime50)

print("Best Case average Runtime for size=150:", best_case_avg_runtime150)
print("Average Case average Runtime for size=150:", average_case_avg_runtime150)
print("Worst Case average Runtime for size=150:", worst_case_avg_runtime150)

print("Best Case average Runtime for size=500:", best_case_avg_runtime500)
print("Average Case average Runtime for size=500:", average_case_avg_runtime500)
print("Worst Case average Runtime for size=500:", worst_case_avg_runtime500)
```

```
Best Case average Runtime for size=50: 0.141143798828125
Average Case average Runtime for size=50 : 0.21147727966308594
Worst Case average Runtime for size=50: 0.37169456481933594
Best Case average Runtime for size=150: 0.40459632873535156
Average Case average Runtime for size=150: 0.4661083221435547
Worst Case average Runtime for size=150: 4.796504974365234
Best Case average Runtime for size=500: 0.8156299591064453
Average Case average Runtime for size=500: 1.7011165618896484
Worst Case average Runtime for size=500: 33.669233322143555
```

Source Code: Click here

BST ALGORITHM

-Execution Time

SIZE=50	Best Case average Runtime for size 50: 0.24461746215820312 Average Case average Runtime for size 50: 0.3409385681152344 Worst Case average Runtime for size 50: 0.8289813995361328 
SIZE=150	Best Case average Runtime for size 150: 0.4947185516357422 Average Case average Runtime for size 150: 0.5099773406982422 Worst Case average Runtime for size 150: 2.854585647583008 
SIZE=500	Best Case average Runtime for size 500: 1.6121864318847656 Average Case average Runtime for size 500: 1.814126968383789 Worst Case average Runtime for size 500: 68.47238540649414 

BST ALGORITHM

-Result Table

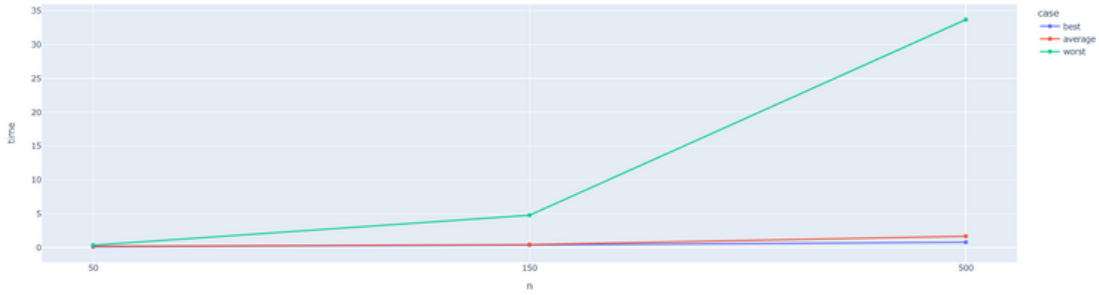
```
data = {'n': [50, 50, 50, 150, 150, 150, 500, 500, 500],
        'case': ['best',  'average', 'worst', 'best',  'average', 'worst', 'best',  'average', 'worst'],
        'time': [best_case_avg_runtime50, average_case_avg_runtime50, worst_case_avg_runtime50, best_case_avg_runtime150, average_case_avg_runtime150, worst_case_avg_runtime150, best_case_avg_runtime500, average_case

df = pd.DataFrame(data)
df
```

	n	case	time
0	50	best	0.141144
1	50	average	0.211477
2	50	worst	0.371695
3	150	best	0.404596
4	150	average	0.466108
5	150	worst	4.796505
6	500	best	0.815630
7	500	average	1.701117
8	500	worst	33.669233

-Result Chart

chart

```
[243] fig = px.line(data, x='n', y='time', color='case')
      fig.update_layout(xaxis_type='category')
      fig.update_traces(mode='markers+lines')
      fig.show()
```



-Analyzation

In conclusion, the analysis of the Binary Search Tree (BST) algorithm provides valuable insights into its performance across different scenarios.

The algorithm illustrates efficient and reliable behavior in **best-case scenarios**, where the tree is balanced, with a complexity of O(log n).

The **average-case scenario** observed with random input data, also maintains a logarithmic time complexity of O(log n).

The **worst-case scenario** happens if the tree becomes unbalanced(sorted data) with a complexity O(n).

I measured the average run time of each case by different data sizes [50,150,500], and I concluded that the best, average cases are quietly close to each other in their average runtime.

I have provided many operations of BST and I have tested the inorder, post-order traversal.

Overall, this analysis serves as a valuable foundation for understanding the practical importance and trade-offs associated with the Binary Search Tree algorithm.

HASH TABLE

By shrooq AL-Qaied

-Implementation

```
[ ]  import time
     import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
```

```
[ ]  class HashTable:
         def __init__(self, size):
             self.size = size
             self.table = [None] * size

         def _hash(self, key):
             return hash(key) % self.size

         def insert(self, key, value):
             index = self._hash(key)
             if self.table[index] is None:
                 self.table[index] = []
             for item in self.table[index]:
                 if item[0] == key:
                     item[1] = value
                     return
             self.table[index].append([key, value])

         def get(self, key):
             index = self._hash(key)
             if self.table[index] is not None:
                 for item in self.table[index]:
                     if item[0] == key:
                         return item[1]
             raise KeyError(f"Key '{key}' not found")
```

HASH TABLE ALGORITHM

```
[ ]  def insert_best_case(hash_table, size):
         for i in range(size):
             key = i * 2
             value = f"value {i}"
             hash_table.insert(key, value)
```

```
[ ]  def insert_average_case(hash_table, size):
         for i in range(size):
             key = np.random.randint(0, size)
             value = f"value {i}"
             hash_table.insert(key, value)
```

```
[ ]  def insert_worst_case(hash_table, size):
         for i in range(size):
             key = i * size
             value = f"value {i}"
             hash_table.insert(key, value)
```

```
[ ]  def measure_runtime(function, *args, num_iterations=100):

         total_time = 0

         for _ in range(num_iterations):
             start_time = time.time()
             function(*args)
             end_time = time.time()
             elapsed_time = (end_time - start_time) * 1000
             total_time += elapsed_time

         average_time = total_time / num_iterations
         return average_time
```

HASH TABLE ALGORITHM

```
[ ]  if __name__ == "__main__":

        sizes = [1000, 5000, 10000]

        data = []

        for size in sizes:
            best_case_ht = HashTable(size)
            average_case_ht = HashTable(size)
            worst_case_ht = HashTable(size)

            average_time_best_case = measure_runtime(insert_best_case, best_case_ht, size)
            average_time_average_case = measure_runtime(insert_average_case, average_case_ht, size)
            average_time_worst_case = measure_runtime(insert_worst_case, worst_case_ht, size)

            data.append([size, 'Best Case', average_time_best_case])
            data.append([size, 'Average Case', average_time_average_case])
            data.append([size, 'Worst Case', average_time_worst_case])

    df = pd.DataFrame(data, columns=['Hash Table Size', 'Case', 'Average Search Time (seconds)'])
    df
```

```
[ ]  pivot_df = df.pivot(index='Hash Table Size', columns='Case', values='Average Search Time (seconds)')
     pivot_df.plot(kind='bar', figsize=(10, 6))
     plt.xlabel('Hash Table Size')
     plt.ylabel('Average Search Time (seconds)')
     plt.title('Average Search Time in Hash Table for Different Cases and Sizes')
     plt.legend(title='Case')

     plt.show()
```

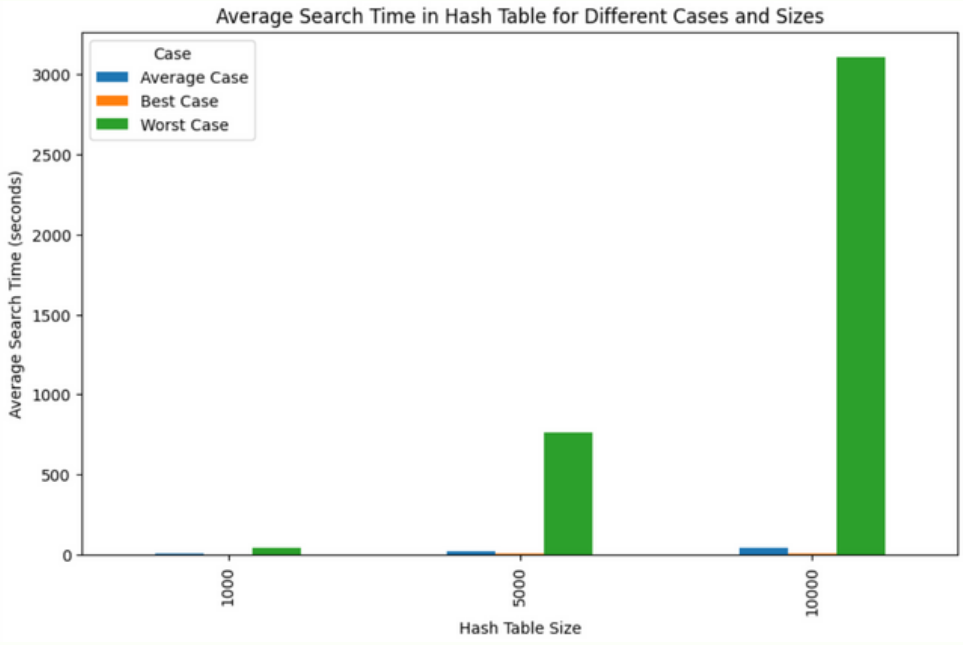
Source Code: Click here

HASH TABLE ALGORITHM

-RESULT TABLE

	Hash Table Size	Case	Average Search Time (seconds)
0	1000	Best Case	0.773826
1	1000	Average Case	3.981521
2	1000	Worst Case	42.209091
3	5000	Best Case	3.878448
4	5000	Average Case	20.675890
5	5000	Worst Case	762.954707
6	10000	Best Case	7.715747
7	10000	Average Case	43.334467
8	10000	Worst Case	3108.881736

-RESULT CHART



HASH TABLE ALGORITHM

-Analyzation

In conclusion,
The hash table is a data structure that allows for efficient key-value pair **storage** and **recall**.
It operates on the principle of using a hash function to map keys to specific locations in an array, providing fast access to values associated with those keys.
for the best, average and wort cases i can conclusion that:

BEST CASE:

- **In the best-case scenario,** where keys are well-distributed and there are no collisions OR there are equal number of collisions, which lead to highly efficient performance.
- The search and insertion operations have a time complexity of O(1).

AVERAGE CASE:

- **In the average-case scenario,** where the keys are randomly distributed and there are random numbers of collisions, which lead to a good performance.
- The search and insertion operations also have a time complexity of O(1), with some little differences due to the number of collisions.

WORST CASE:

- **In the worst-case scenario,** where all the keys are inserted into the same index, which lead to poor performance.
- The search and insertion operations have a time complexity of O(n), where "n" is the number of item in the collision chain.