Computer Science 2XC3 - Graded Lab II

In this day and age, it is very easy to generate automated solutions to any problem, not necessarily becuase of AI, but because of vast online communities that exist to find solution to popular/common problems. Whether or not that solution is correct and applicable to our context, can be assessed only if we understand the concepts and can critically evaluate them. The goal of this lab is to motivate you to not only produce the correct solution to problems, but also to reflect about why, how and when your solution will likely succeed/fail.

In this lab you will design experiments with sorting and search algorithms. Please read all instructions carefully. Seek the help of TA's if you need clarifications on the task. Do not hard code any results.

In [18]:
```
import random
import time
import timeit
import matplotlib.pyplot as plt
import numpy as np
```

Part A

A1. Implement three classes with the following sorting algorithms:

- Bubble Sort
- Insertion Sort
- Selection Sort

In [44]:
```
class BubbleSort:
    def __init__(self, items_to_sort):
        self.items = items_to_sort
        self.sorted_items=[]

      ### your implementation for bubble sort goes here
    def bubble_sort(data):
        no_more_swap =True
        while no_more_swap:
            no_more_swap = False
```

```
        for iteration in range(len(data)-1, 0, -1):
            for index in range(iteration):
                if data[index] > data[index+1]:
                    data[index],data[index+1]=data[index+1],data[index]
        return data

    def get_sorted(self,):
        return self.sorted_items
```

In [45]:
```
class InsertionSort:
    def __init__(self, items_to_sort):
        self.items = items_to_sort
        self.sorted_items=[]

    def insertion_sort(data):
        for index in range(1, len(data)):
            while index > 0 and data[index] < data[index-1]:
                data[index],data[index-1]=data[index-1],data[index]
                index -= 1
        return data

      ### your implementation for insertion sort goes here

    def get_sorted(self,):
        return self.sorted_items
```

In [46]:
```
class SelectionSort:
    def __init__(self, items_to_sort):
        self.items = items_to_sort
        self.sorted_items=[]

      ### your implementation for selection sort goes here
    def selection_sort(data):
        for index in range(len(data)):
            min_index = index

            for j in range(index + 1, len(data)):
                if data[j] < data[min_index]:
                    min_index = j

            (data[index], data[min_index]) = (data[min_index], data[index])
        return data

    def get_sorted(self,):
        return self.sorted_items
```

A2. Compute the performance of above 3 algorithms on a single list of real numbers.

First generate a custom random list using function `create_custom_list()`. Execute each of the above algorithm for N trials (select N >= 75) on the list and plot the timing of each execution on a bar chart. Also calculate the average execution time for the entire batch of N trials (you can either display it on the chart or simply `print()` it). For larger values of N, consider breaking N trials into mini batches of n executions and plotting execution times for each mini batch. For instance, if you select N=1000, to plot execution timings for 1000 trials, you may break them into mini batch of n=10 trials and display average of each mini batch. This will reduce clutter in your bar charts while still enabling you to perform extensive testing with higher N.

Execute each of the above algorithm on the same set of integers. The outcome of your code should be 3 charts for each algorithm run on your list N times. Few utility functions are given below. You do not have to necessarily use the `draw_plot()` function. You can plot your timings using an excel sheet and paste the image of your timings here. Refer to Markdown Guide on how to add images in the jupyter notebook or ask your TA.

In [5]:
```
def create_custom_list(length, max_value, item=None, item_index=None):
    random_list = [random.randint(0,max_value) for i in range(length)]
    if item!= None:
        random_list.insert(item_index,item)
    return random_list
```

In [6]:
```
def draw_plot(run_arr):
    x = np.arange(0, len(run_arr),1)
    fig=plt.figure(figsize=(12,8))
    plt.bar(x,run_arr)
    plt.axhline(np.mean(run_arr),color="red",linestyle="--",label="Avg")
    plt.xlabel("Iterations")
    plt.ylabel("Run time in ms order of 1e-6")
    plt.title("Run time for retrieval")
    plt.show()
```

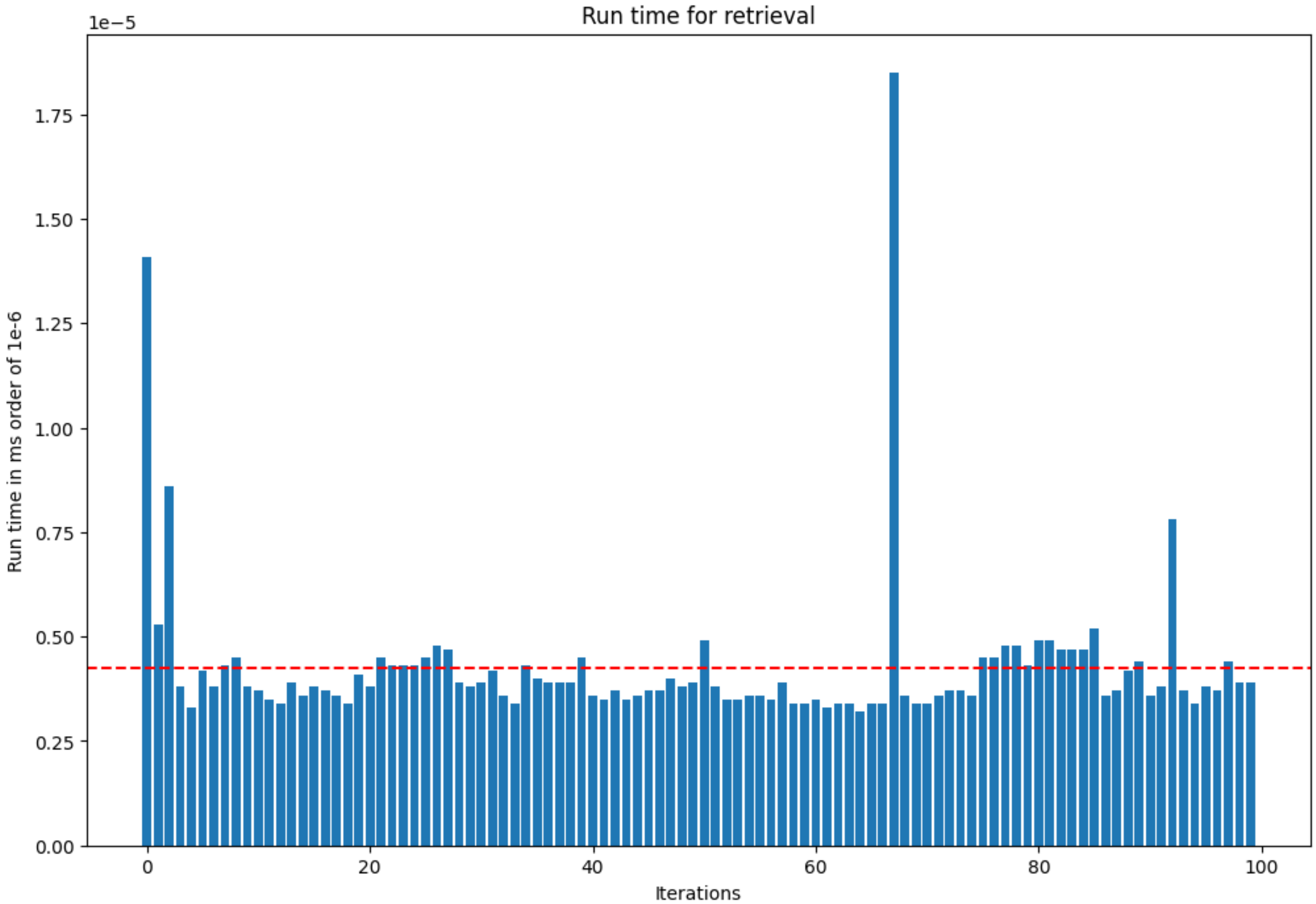
In [7]:
```
my_list = create_custom_list(10, 2000)
list1 = my_list.copy()
list2 = my_list.copy()
list3 = my_list.copy()
print(my_list)
```

```
[41, 1470, 1082, 602, 1445, 1100, 92, 1059, 73, 1122]
```

In [8]:
```
### Bubble sort experiment code goes here
runs = 100
run_times = []
#list1 = my_list
```

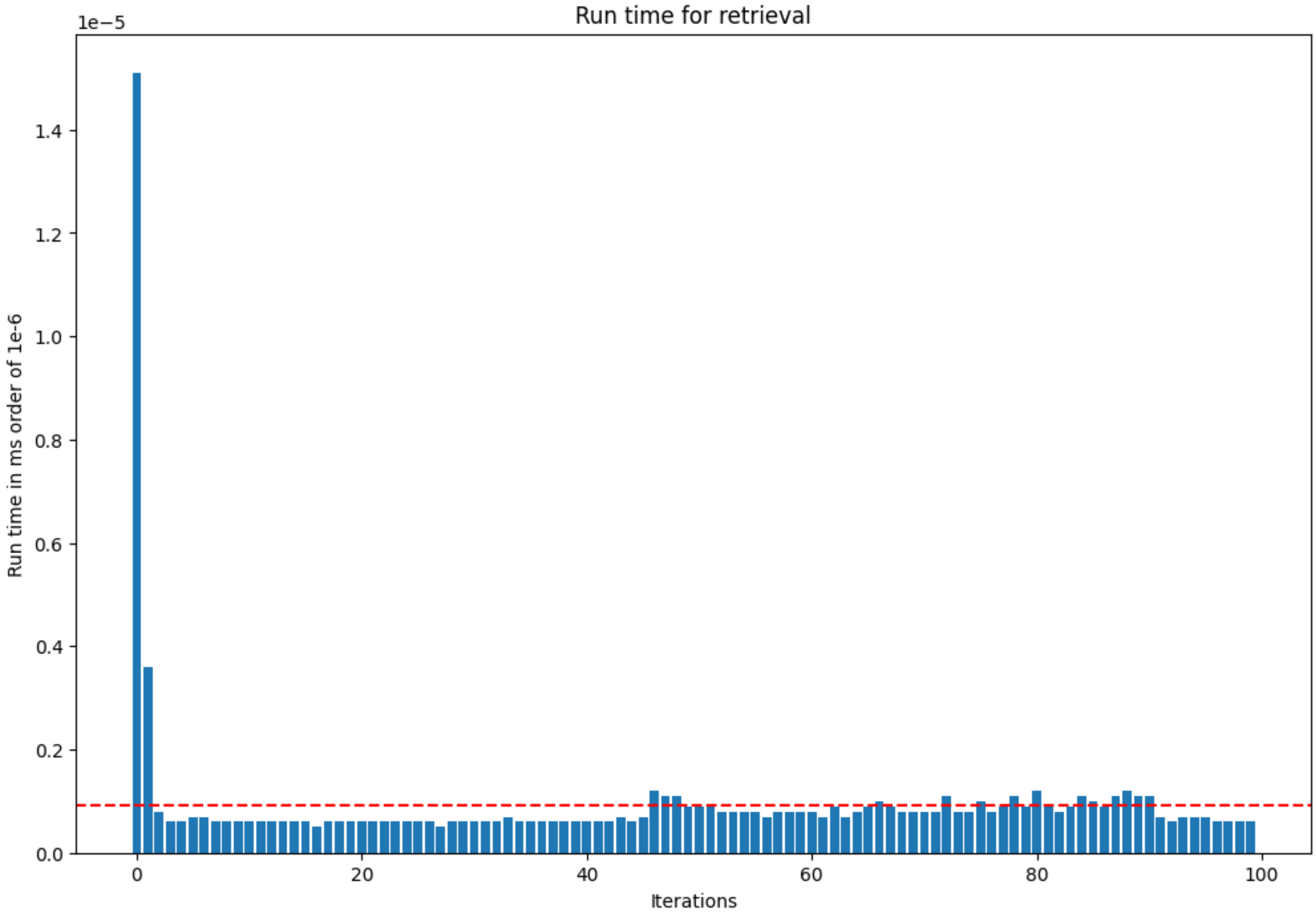
```python
print(list1)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(list1)
    #found = BubbleSort.bubble_sort(list1)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(list1)

draw_plot(run_times)
```

```
[41, 1470, 1082, 602, 1445, 1100, 92, 1059, 73, 1122]
[41, 73, 92, 602, 1059, 1082, 1100, 1122, 1445, 1470]
```



In [9]:
```python
### Insertion sort experiment code goes here
runs = 100
run_times = []
print(list2)
for _ in range(runs):
    start = timeit.default_timer()
    InsertionSort.insertion_sort(list2)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(list2)

draw_plot(run_times)
```

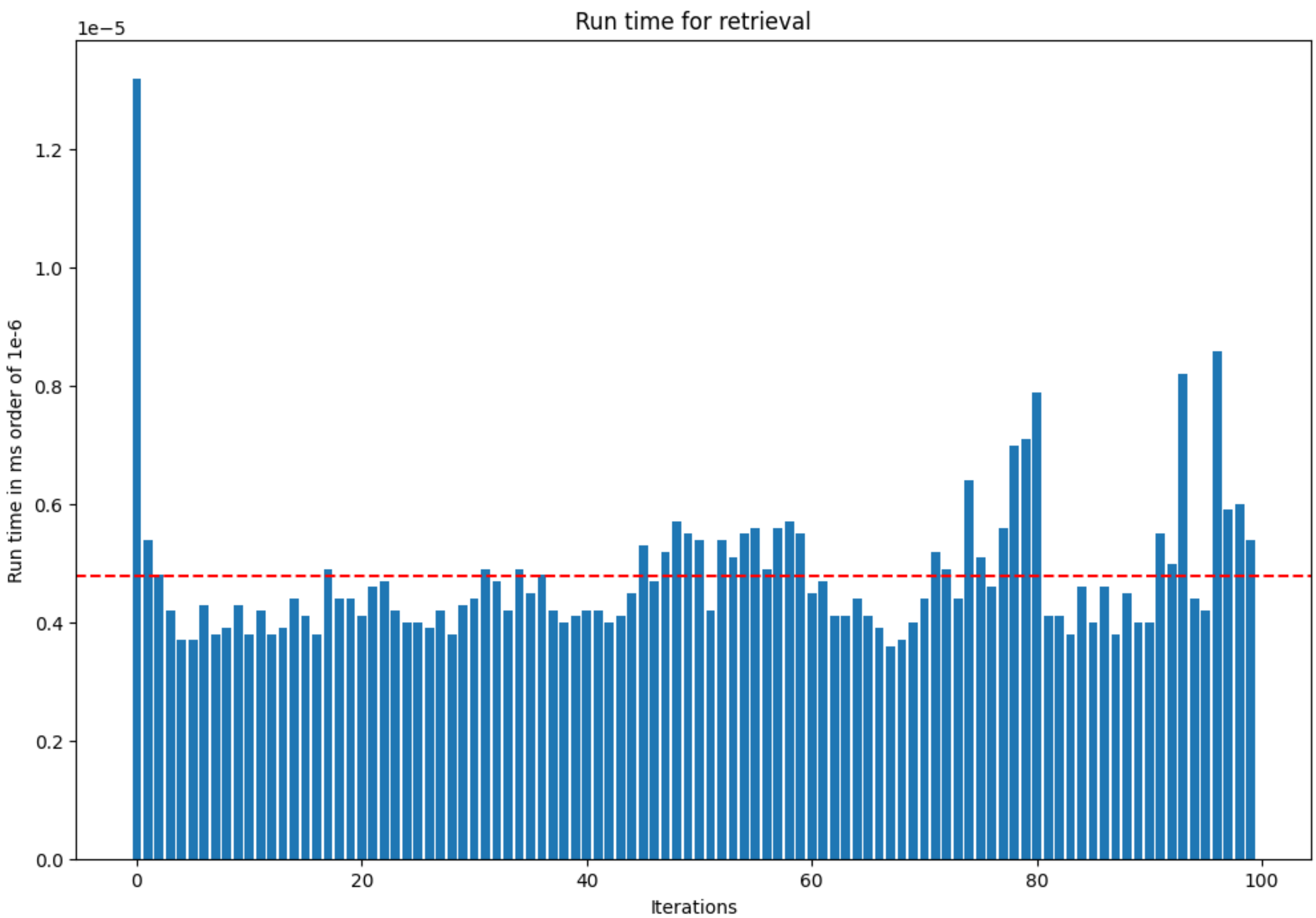
```
[41, 1470, 1082, 602, 1445, 1100, 92, 1059, 73, 1122]
[41, 73, 92, 602, 1059, 1082, 1100, 1122, 1445, 1470]
```



In [10]:

```python
### Selection sort experiment code goes here
runs = 100
run_times = []
print(list3)
for _ in range(runs):
    start = timeit.default_timer()
    SelectionSort.selection_sort(list3)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(list3)

draw_plot(run_times)
```

```
[41, 1470, 1082, 602, 1445, 1100, 92, 1059, 73, 1122]
[41, 73, 92, 602, 1059, 1082, 1100, 1122, 1445, 1470]
```



You would notice that certain sorting algorithms have better time complexity (or performance) than others. Write below a reflection of your observations. Can you confidently compare the performance across the 3 algorithms? Why does certain algorithm perform better than the other? What are the various factors impacting the best performing and the worst performing algorithm. Write a few sentences answering each of the above questions. Also describe any other observation you found important.

Reflection:

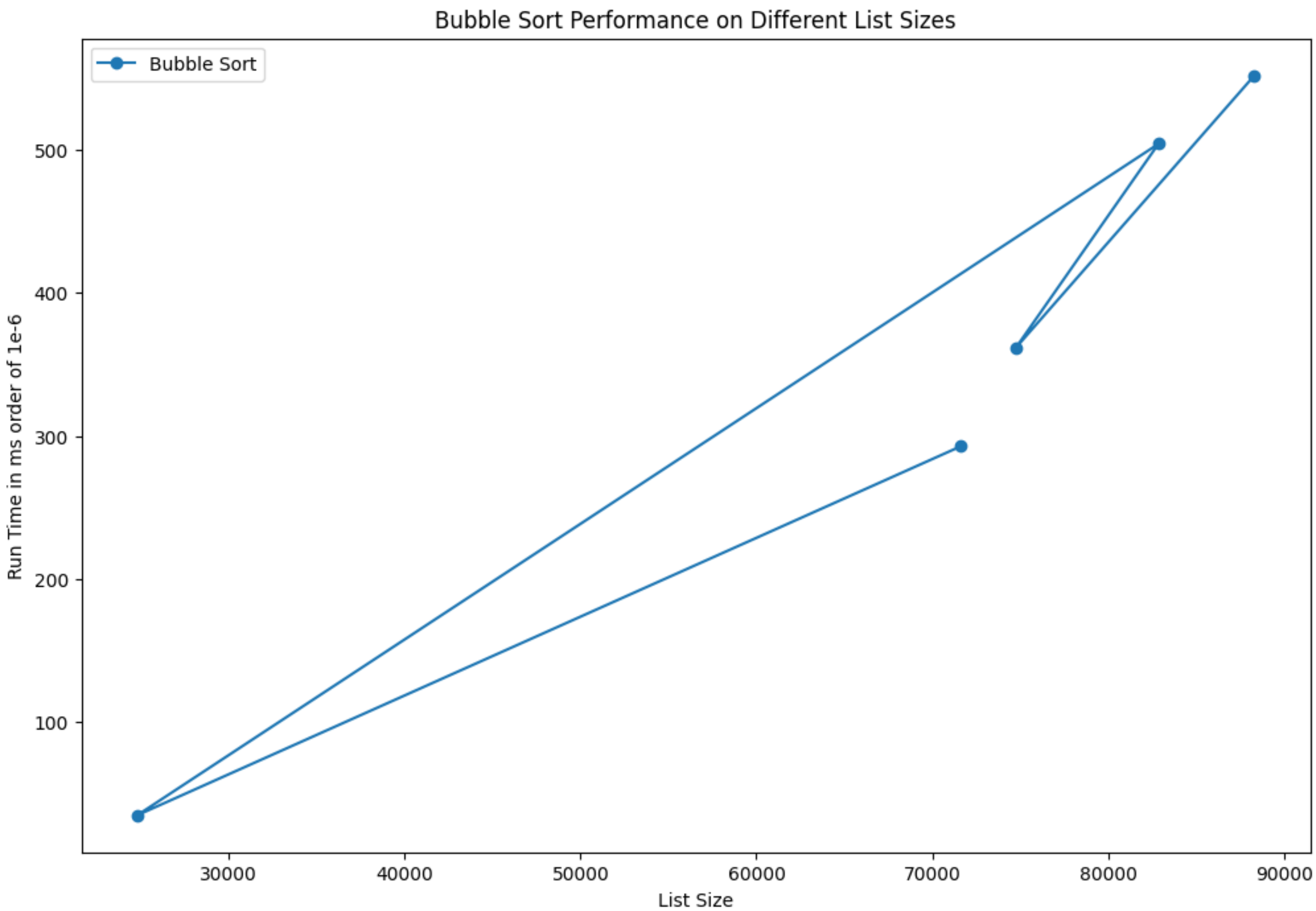
A3. Compute the performance of above 3 algorithms on a different list sizes.

The `create_custom_list()` helps you create lists of varying lengths and range of numbers. Plot a **line chart** that shows the performance of each algorithm on different list sizes ranging between 1 - 100,000 integers. If you think about this question, you are essentially plotting the time complexity on various list sizes.

In [11]:
```
### Bubble sort experiment code goes here
list_sizes = [random.randint(1, 100001) for _ in range(5)]
print(list_sizes)
bubble_sort_times = []
```

```python
for size in list_sizes:
    random_list = create_custom_list(size, 100)
    start = timeit.default_timer()
    BubbleSort.bubble_sort(random_list)
    stop = timeit.default_timer()
    bubble_sort_times.append(stop-start)

fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(list_sizes, bubble_sort_times, label='Bubble Sort', marker='o')

# Add labels and title
ax.set_xlabel('List Size')
ax.set_ylabel('Run Time in ms order of 1e-6')
ax.set_title('Bubble Sort Performance on Different List Sizes')

# Add legend
ax.legend()
# Show the plot
plt.show()
```

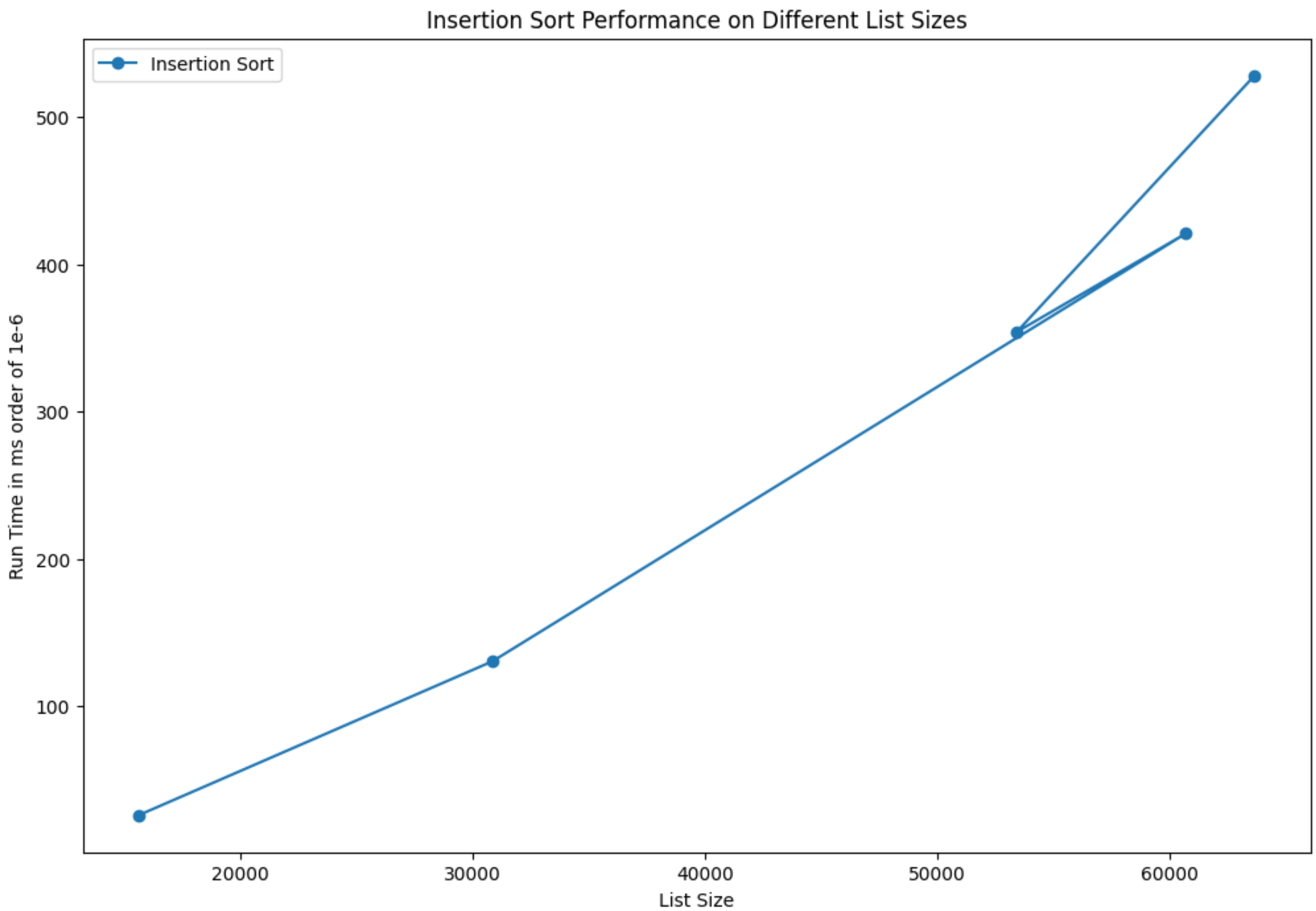
```
[88305, 74748, 82856, 24811, 71625]
```



In [92]:
```python
### Insertion sort experiment code goes here
list_sizes = [random.randint(1, 100001) for _ in range(5)]
print(list_sizes)
insertion_sort_times = []
for size in list_sizes:
```

```python
    random_list = create_custom_list(size, 100)
    start = timeit.default_timer()
    InsertionSort.insertion_sort(random_list)
    stop = timeit.default_timer()
    insertion_sort_times.append(stop-start)

fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(list_sizes, insertion_sort_times, label='Insertion Sort', marker='o')

# Add labels and title
ax.set_xlabel('List Size')
ax.set_ylabel('Run Time in ms order of 1e-6')
ax.set_title('Insertion Sort Performance on Different List Sizes')

# Add legend
ax.legend()

# Show the plot
plt.show()
```

```
[15646, 30869, 60683, 53414, 63648]
```

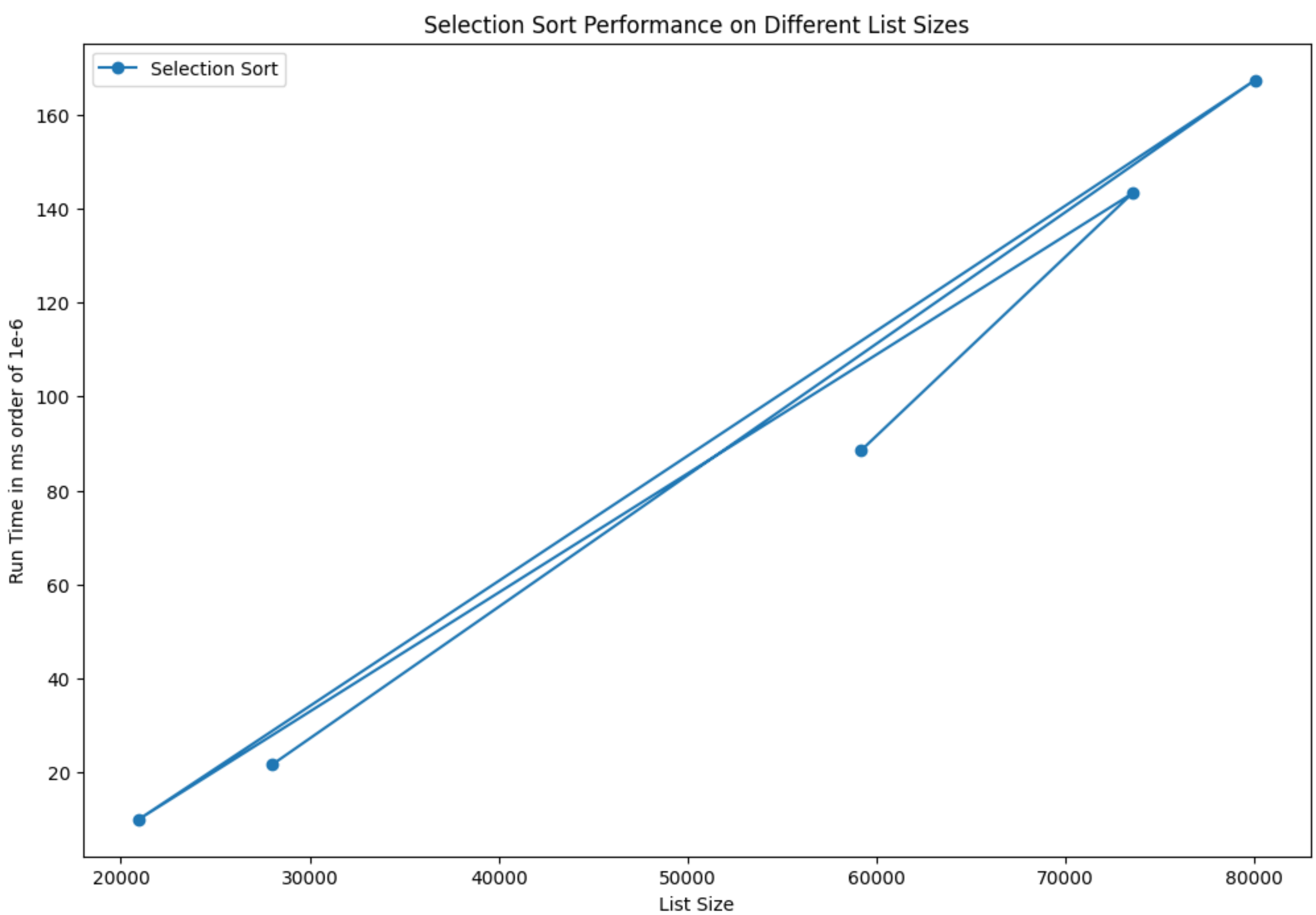


In [35]:

```python
### Selection sort experiment code goes here
list_sizes = [random.randint(1, 100001) for _ in range(5)]
print(list_sizes)
selection_sort_times = []
for size in list_sizes:
```

```
    random_list = create_custom_list(size, 100)
    start = timeit.default_timer()
    SelectionSort.selection_sort(random_list)
    stop = timeit.default_timer()
    selection_sort_times.append(stop-start)

fig, ax = plt.subplots(figsize=(12, 8))
ax.plot(list_sizes, selection_sort_times, label='Selection Sort', marker='o')

# Add labels and title
ax.set_xlabel('List Size')
ax.set_ylabel('Run Time in ms order of 1e-6')
ax.set_title('Selection Sort Performance on Different List Sizes')

# Add legend
ax.legend()

# Show the plot
plt.show()
```

```
[59193, 73533, 20952, 80019, 28038]
```



Describe you results here. What did you observe when comparing the charts? Which algorithm was more performant and why?

Reflection : We notice that the execution time increases eponentially as we increase the input size. These results align with the time complexity of the algorithms. However, insertion and selection sort were visibally faster than bubblesort.

A4. Compute the performance of above 3 algorithms on a different list "states".

Using the same above list generation function (or writing a function of your own), create two different lists states:

- A state where the list is **near** sorted.
- A state where the list is completely unsorted.

HINTS:

- You can implement a "controlled" Quicksort algorithm for such a function. While you can find many implementations of such a function online, significant number of those solutions originate from this psuedocode Generating Sorted Lists of Random Numbers.

- You can modify the list generation code given above to create the above list examples.

Compare the performance of all 3 sorting algorithms on these two lists. Plot their performance on bar chart and display them here.

In [14]:
```
"""need to create 6 combinations
   try to see if we can quantify when a list is 60, 70 or 80% sorted

"""
length = 100
max_value = 2000
def partially_sorted_list(length, max_value):
    # Calculate the size of the sorted sublist (70% of the length)
    sorted_sublist_size = int(length * 0.7)

    # Generate a sorted sublist
    sorted_sublist = sorted(create_custom_list(sorted_sublist_size, max_value))

    # Fill the rest of the list with random numbers
    unsorted_list = sorted_sublist + create_custom_list(length - sorted_sublist

    return unsorted_list

unsorted = create_custom_list(length, max_value)
```

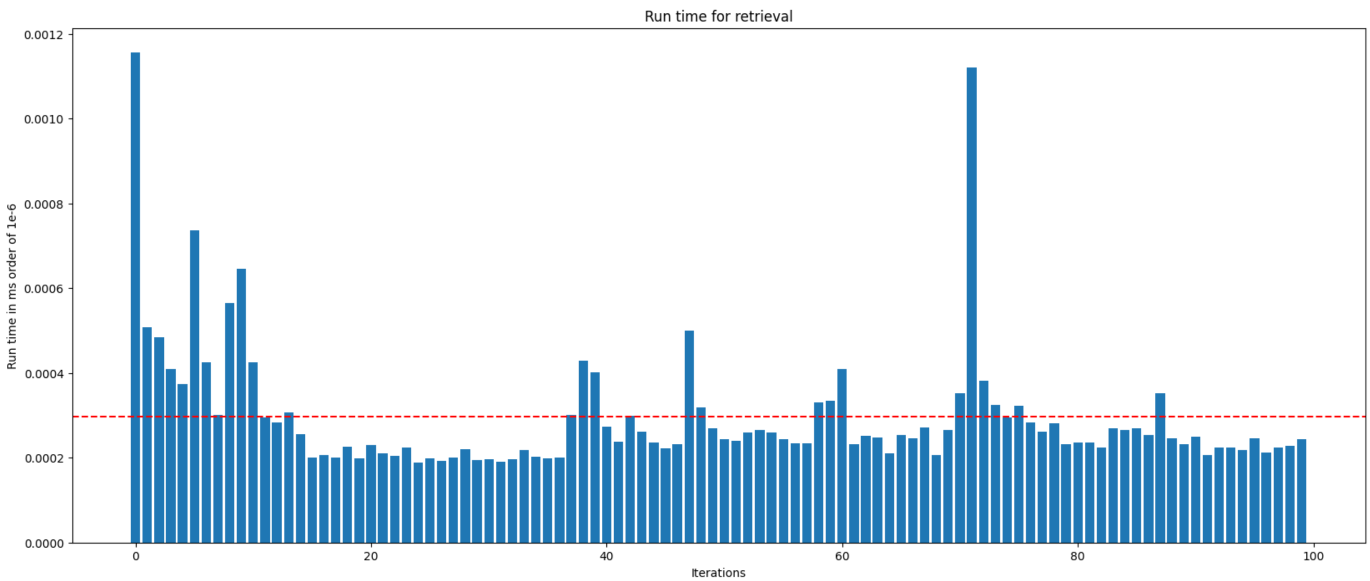
```
partially_sorted = partially_sorted_list(length, max_value)

print(unsorted)
print(partially_sorted)
```

```
[1249, 189, 1522, 1271, 483, 1275, 243, 1091, 30, 773, 937, 580, 1702, 645, 193
4, 408, 37, 1220, 98, 1826, 147, 301, 964, 747, 1243, 640, 124, 1245, 444, 1571,
1166, 211, 1116, 1872, 1441, 444, 1616, 1863, 987, 1748, 1443, 10, 765, 1438, 29
0, 382, 1747, 61, 1692, 1659, 525, 590, 1335, 1035, 1658, 403, 648, 653, 1006, 1
238, 1971, 1504, 557, 1773, 387, 97, 1625, 1684, 997, 1367, 686, 170, 240, 881,
616, 1315, 512, 669, 1493, 232, 123, 566, 513, 352, 1939, 1040, 158, 700, 344, 1
924, 1457, 998, 615, 1769, 930, 791, 166, 1167, 715, 204]
[19, 35, 66, 107, 153, 230, 246, 252, 374, 405, 408, 409, 507, 525, 532, 708, 71
9, 735, 764, 783, 821, 827, 827, 864, 873, 899, 903, 928, 944, 972, 990, 1057, 1
064, 1064, 1078, 1108, 1116, 1118, 1205, 1212, 1283, 1290, 1316, 1317, 1380, 144
4, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543, 1670, 1680, 1683, 1701, 1724,
1726, 1756, 1822, 1849, 1886, 1908, 1916, 1964, 1978, 1992, 1992, 946, 901, 116
5, 1771, 355, 249, 238, 714, 267, 773, 950, 1963, 606, 1859, 1791, 1682, 977, 19
94, 984, 1155, 957, 1077, 1737, 593, 1189, 1153, 1802, 1603, 854, 288]
```

In [15]:

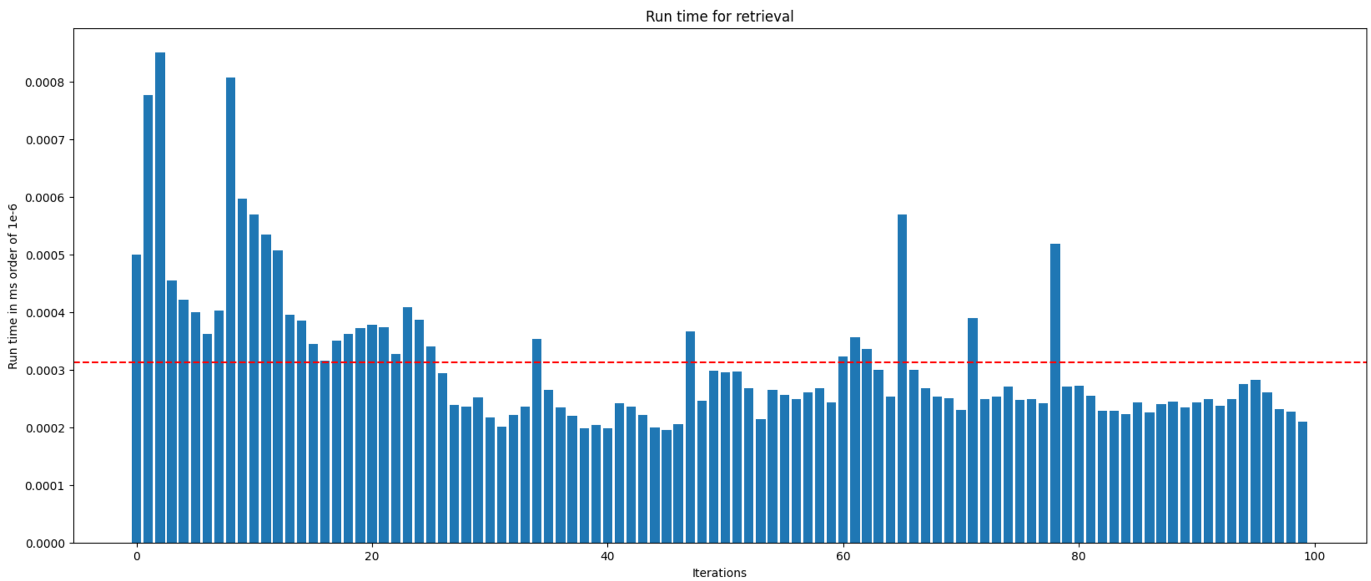
```
### Bubble sort experiment code goes here
#Unsorted
runs = 100
run_times = []
u_list1 = unsorted.copy()
print(u_list1)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(u_list1)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(u_list1)

draw_plot(run_times)
```

```
[1249, 189, 1522, 1271, 483, 1275, 243, 1091, 30, 773, 937, 580, 1702, 645, 193
4, 408, 37, 1220, 98, 1826, 147, 301, 964, 747, 1243, 640, 124, 1245, 444, 1571,
1166, 211, 1116, 1872, 1441, 444, 1616, 1863, 987, 1748, 1443, 10, 765, 1438, 29
0, 382, 1747, 61, 1692, 1659, 525, 590, 1335, 1035, 1658, 403, 648, 653, 1006, 1
238, 1971, 1504, 557, 1773, 387, 97, 1625, 1684, 997, 1367, 686, 170, 240, 881,
616, 1315, 512, 669, 1493, 232, 123, 566, 513, 352, 1939, 1040, 158, 700, 344, 1
924, 1457, 998, 615, 1769, 930, 791, 166, 1167, 715, 204]
[10, 30, 37, 61, 97, 98, 123, 124, 147, 158, 166, 170, 189, 204, 211, 232, 240,
243, 290, 301, 344, 352, 382, 387, 403, 408, 444, 444, 483, 512, 513, 525, 557,
566, 580, 590, 615, 616, 640, 645, 648, 653, 669, 686, 700, 715, 747, 765, 773,
791, 881, 930, 937, 964, 987, 997, 998, 1006, 1035, 1040, 1091, 1116, 1166, 116
7, 1220, 1238, 1243, 1245, 1249, 1271, 1275, 1315, 1335, 1367, 1438, 1441, 1443,
1457, 1493, 1504, 1522, 1571, 1616, 1625, 1658, 1659, 1684, 1692, 1702, 1747, 17
48, 1769, 1773, 1826, 1863, 1872, 1924, 1934, 1939, 1971]
```



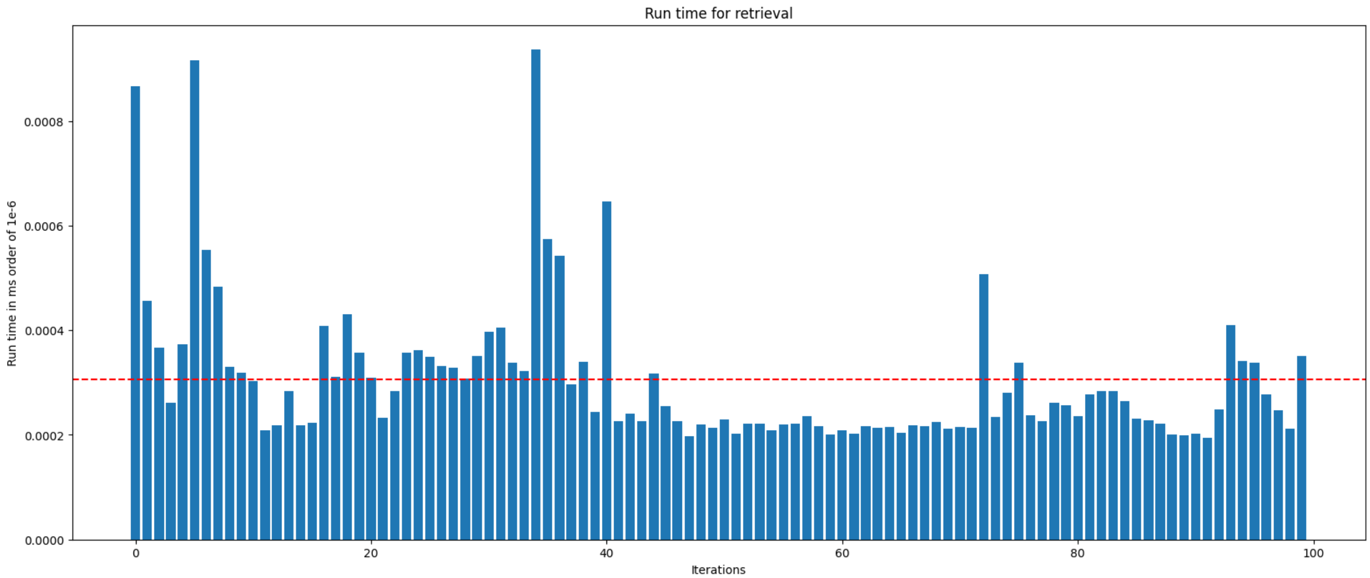
In [16]:
```
#Partially Sorted
runs = 100
run_times = []
p_list1 = partially_sorted.copy()
print(p_list1)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(p_list1)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(p_list1)

draw_plot(run_times)
```

```
[19, 35, 66, 107, 153, 230, 246, 252, 374, 405, 408, 409, 507, 525, 532, 708, 71
9, 735, 764, 783, 821, 827, 827, 864, 873, 899, 903, 928, 944, 972, 990, 1057, 1
064, 1064, 1078, 1108, 1116, 1118, 1205, 1212, 1283, 1290, 1316, 1317, 1380, 144
4, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543, 1670, 1680, 1683, 1701, 1724,
1726, 1756, 1822, 1849, 1886, 1908, 1916, 1964, 1978, 1992, 1992, 946, 901, 116
5, 1771, 355, 249, 238, 714, 267, 773, 950, 1963, 606, 1859, 1791, 1682, 977, 19
94, 984, 1155, 957, 1077, 1737, 593, 1189, 1153, 1802, 1603, 854, 288]
[19, 35, 66, 107, 153, 230, 238, 246, 249, 252, 267, 288, 355, 374, 405, 408, 40
9, 507, 525, 532, 593, 606, 708, 714, 719, 735, 764, 773, 783, 821, 827, 827, 85
4, 864, 873, 899, 901, 903, 928, 944, 946, 950, 957, 972, 977, 984, 990, 1057, 1
064, 1064, 1077, 1078, 1108, 1116, 1118, 1153, 1155, 1165, 1189, 1205, 1212, 128
3, 1290, 1316, 1317, 1380, 1444, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543,
1603, 1670, 1680, 1682, 1683, 1701, 1724, 1726, 1737, 1756, 1771, 1791, 1802, 18
22, 1849, 1859, 1886, 1908, 1916, 1963, 1964, 1978, 1992, 1992, 1994]
```



In [17]:
```
### Selection sort experiment code goes here
#Unsorted
runs = 100
run_times = []
u_list2 = unsorted.copy()
print(u_list2)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(u_list2)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(u_list2)

draw_plot(run_times)
```

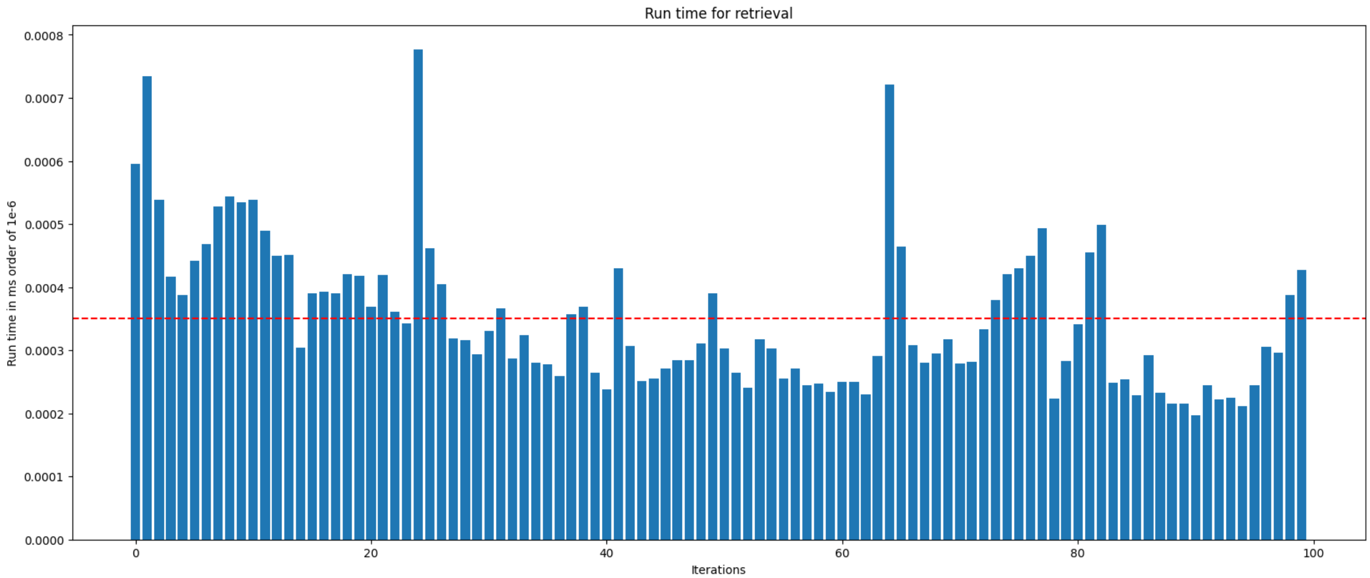
```
[1249, 189, 1522, 1271, 483, 1275, 243, 1091, 30, 773, 937, 580, 1702, 645, 193
4, 408, 37, 1220, 98, 1826, 147, 301, 964, 747, 1243, 640, 124, 1245, 444, 1571,
1166, 211, 1116, 1872, 1441, 444, 1616, 1863, 987, 1748, 1443, 10, 765, 1438, 29
0, 382, 1747, 61, 1692, 1659, 525, 590, 1335, 1035, 1658, 403, 648, 653, 1006, 1
238, 1971, 1504, 557, 1773, 387, 97, 1625, 1684, 997, 1367, 686, 170, 240, 881,
616, 1315, 512, 669, 1493, 232, 123, 566, 513, 352, 1939, 1040, 158, 700, 344, 1
924, 1457, 998, 615, 1769, 930, 791, 166, 1167, 715, 204]
[10, 30, 37, 61, 97, 98, 123, 124, 147, 158, 166, 170, 189, 204, 211, 232, 240,
243, 290, 301, 344, 352, 382, 387, 403, 408, 444, 444, 483, 512, 513, 525, 557,
566, 580, 590, 615, 616, 640, 645, 648, 653, 669, 686, 700, 715, 747, 765, 773,
791, 881, 930, 937, 964, 987, 997, 998, 1006, 1035, 1040, 1091, 1116, 1166, 116
7, 1220, 1238, 1243, 1245, 1249, 1271, 1275, 1315, 1335, 1367, 1438, 1441, 1443,
1457, 1493, 1504, 1522, 1571, 1616, 1625, 1658, 1659, 1684, 1692, 1702, 1747, 17
48, 1769, 1773, 1826, 1863, 1872, 1924, 1934, 1939, 1971]
```



In [18]:

```python
#Partially Sorted
runs = 100
run_times = []
p_list2 = partially_sorted.copy()
print(p_list2)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(p_list2)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(p_list2)

draw_plot(run_times)
```

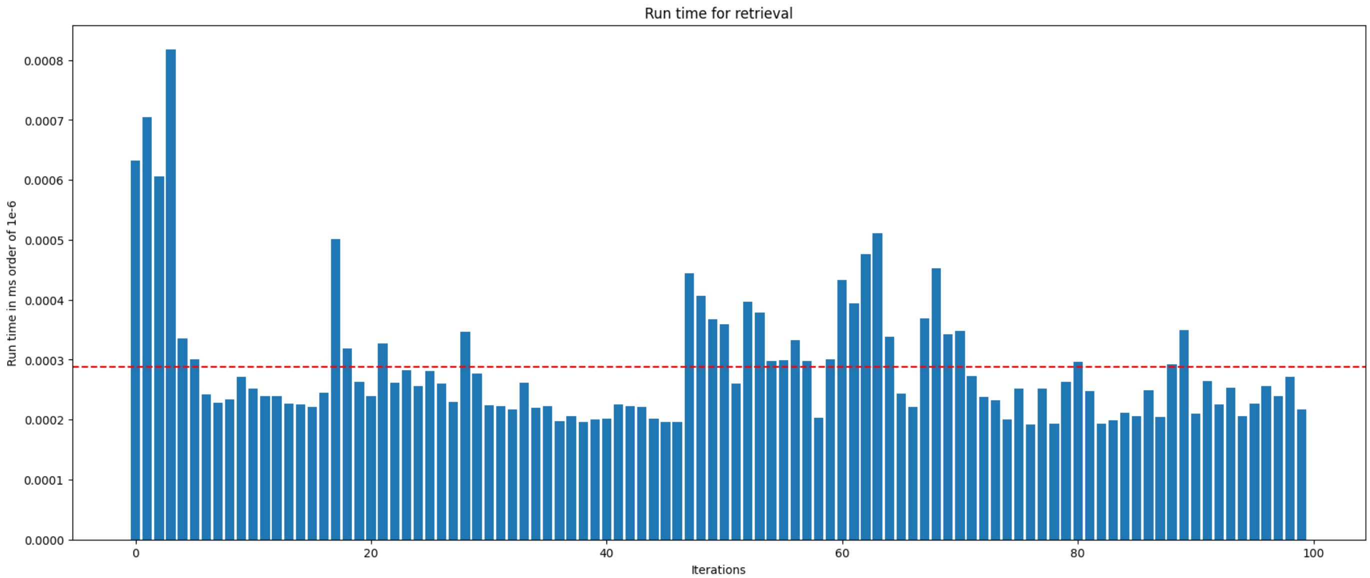
```
[19, 35, 66, 107, 153, 230, 246, 252, 374, 405, 408, 409, 507, 525, 532, 708, 719, 735, 764, 783, 821, 827, 827, 864, 873, 899, 903, 928, 944, 972, 990, 1057, 1064, 1064, 1078, 1108, 1116, 1118, 1205, 1212, 1283, 1290, 1316, 1317, 1380, 1444, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543, 1670, 1680, 1683, 1701, 1724, 1726, 1756, 1822, 1849, 1886, 1908, 1916, 1964, 1978, 1992, 1992, 946, 901, 1165, 1771, 355, 249, 238, 714, 267, 773, 950, 1963, 606, 1859, 1791, 1682, 977, 1994, 984, 1155, 957, 1077, 1737, 593, 1189, 1153, 1802, 1603, 854, 288]
[19, 35, 66, 107, 153, 230, 238, 246, 249, 252, 267, 288, 355, 374, 405, 408, 409, 507, 525, 532, 593, 606, 708, 714, 719, 735, 764, 773, 783, 821, 827, 827, 854, 864, 873, 899, 901, 903, 928, 944, 946, 950, 957, 972, 977, 984, 990, 1057, 1064, 1064, 1077, 1078, 1108, 1116, 1118, 1153, 1155, 1165, 1189, 1205, 1212, 1283, 1290, 1316, 1317, 1380, 1444, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543, 1603, 1670, 1680, 1682, 1683, 1701, 1724, 1726, 1737, 1756, 1771, 1791, 1802, 1822, 1849, 1859, 1886, 1908, 1916, 1963, 1964, 1978, 1992, 1992, 1994]
```



In [19]:

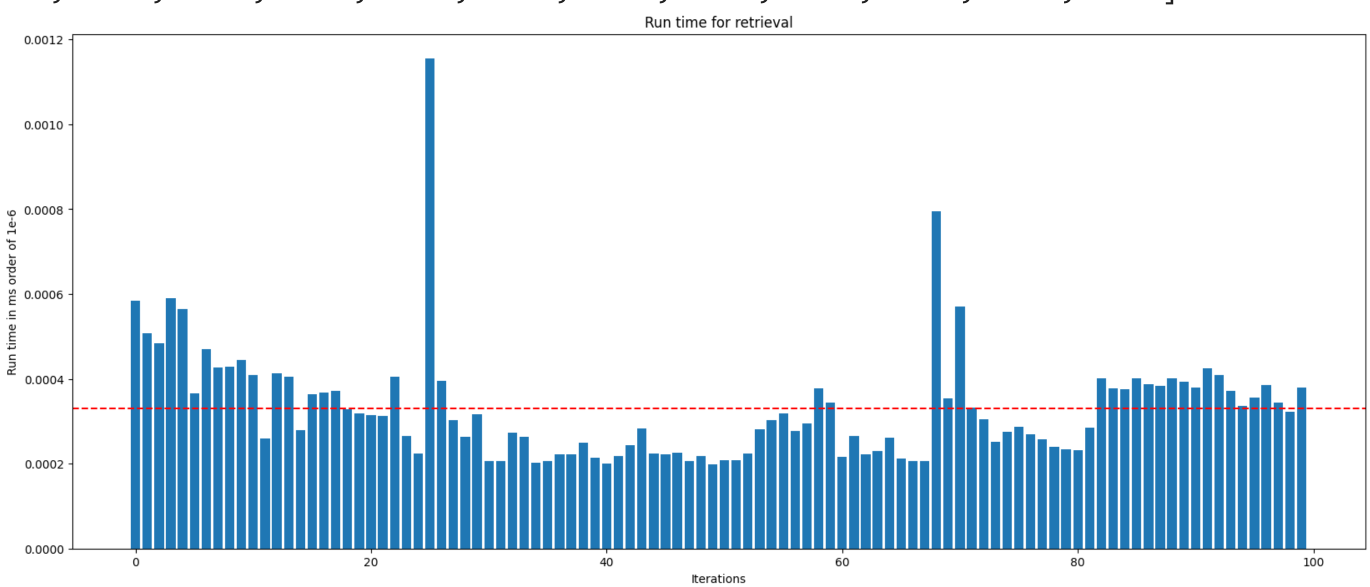
```python
### Insertion sort experiment code goes here
#Unsorted
runs = 100
run_times = []
u_list3 = unsorted.copy()
print(u_list3)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(u_list3)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(u_list3)

draw_plot(run_times)
```

```
[1249, 189, 1522, 1271, 483, 1275, 243, 1091, 30, 773, 937, 580, 1702, 645, 193
4, 408, 37, 1220, 98, 1826, 147, 301, 964, 747, 1243, 640, 124, 1245, 444, 1571,
1166, 211, 1116, 1872, 1441, 444, 1616, 1863, 987, 1748, 1443, 10, 765, 1438, 29
0, 382, 1747, 61, 1692, 1659, 525, 590, 1335, 1035, 1658, 403, 648, 653, 1006, 1
238, 1971, 1504, 557, 1773, 387, 97, 1625, 1684, 997, 1367, 686, 170, 240, 881,
616, 1315, 512, 669, 1493, 232, 123, 566, 513, 352, 1939, 1040, 158, 700, 344, 1
924, 1457, 998, 615, 1769, 930, 791, 166, 1167, 715, 204]
[10, 30, 37, 61, 97, 98, 123, 124, 147, 158, 166, 170, 189, 204, 211, 232, 240,
243, 290, 301, 344, 352, 382, 387, 403, 408, 444, 444, 483, 512, 513, 525, 557,
566, 580, 590, 615, 616, 640, 645, 648, 653, 669, 686, 700, 715, 747, 765, 773,
791, 881, 930, 937, 964, 987, 997, 998, 1006, 1035, 1040, 1091, 1116, 1166, 116
7, 1220, 1238, 1243, 1245, 1249, 1271, 1275, 1315, 1335, 1367, 1438, 1441, 1443,
1457, 1493, 1504, 1522, 1571, 1616, 1625, 1658, 1659, 1684, 1692, 1702, 1747, 17
48, 1769, 1773, 1826, 1863, 1872, 1924, 1934, 1939, 1971]
```



In [20]:
```
#Partially Sorted
runs = 100
run_times = []
p_list3 = partially_sorted.copy()
print(p_list3)
for _ in range(runs):
    start = timeit.default_timer()
    BubbleSort.bubble_sort(p_list3)
    stop = timeit.default_timer()
    run_times.append(stop-start)
print(p_list3)

draw_plot(run_times)
```

```
[19, 35, 66, 107, 153, 230, 246, 252, 374, 405, 408, 409, 507, 525, 532, 708, 71
9, 735, 764, 783, 821, 827, 827, 864, 873, 899, 903, 928, 944, 972, 990, 1057, 1
064, 1064, 1078, 1108, 1116, 1118, 1205, 1212, 1283, 1290, 1316, 1317, 1380, 144
4, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543, 1670, 1680, 1683, 1701, 1724,
1726, 1756, 1822, 1849, 1886, 1908, 1916, 1964, 1978, 1992, 1992, 946, 901, 116
5, 1771, 355, 249, 238, 714, 267, 773, 950, 1963, 606, 1859, 1791, 1682, 977, 19
94, 984, 1155, 957, 1077, 1737, 593, 1189, 1153, 1802, 1603, 854, 288]
[19, 35, 66, 107, 153, 230, 238, 246, 249, 252, 267, 288, 355, 374, 405, 408, 40
9, 507, 525, 532, 593, 606, 708, 714, 719, 735, 764, 773, 783, 821, 827, 827, 85
4, 864, 873, 899, 901, 903, 928, 944, 946, 950, 957, 972, 977, 984, 990, 1057, 1
064, 1064, 1077, 1078, 1108, 1116, 1118, 1153, 1155, 1165, 1189, 1205, 1212, 128
3, 1290, 1316, 1317, 1380, 1444, 1453, 1460, 1466, 1471, 1503, 1507, 1527, 1543,
1603, 1670, 1680, 1682, 1683, 1701, 1724, 1726, 1737, 1756, 1771, 1791, 1802, 18
22, 1849, 1859, 1886, 1908, 1916, 1963, 1964, 1978, 1992, 1992, 1994]
```



Describe you observations here. Which algorithm performs best / worst for sorted/near sorted lists and why? Does the performance vary significantly? Describe which runs times were higher and why do you think that is? You woul

Reflection : As seen from the graphs for unsorted arrays, selection and insertion sort had similar times however, bubble sort took way longer. As for the partially sorted arrays insertion sort was the most efficient. This could be because insertion sort takes advantage of the partially sorted elements in the array resulting in fewer comparisions and movements.

Part B

In the class, we discussed three implementations of Binary Search.

In [20]:
```python
def binary_search_1(item_list, to_find):
    lower=0
    upper=len(item_list)-1
    count = 0
    while lower < upper:
        count += 1
        mid = (lower+upper)//2
        if item_list[mid] == to_find:
            # print(f"Binary search 1 took {count} steps")
            return True,count
        if item_list[mid] < to_find:
            lower = mid+1
        else:
            upper=mid
    # print(f"Binary search 1 took {count} steps")
    return item_list[lower]==to_find, count
```

In [21]:
```python
def binary_search_2(item_list, to_find):
    lower=0
    upper=len(item_list)-1
    count = 0
    while lower <= upper:
        count += 1
        mid = (lower+upper)//2
        if item_list[mid] == to_find:
            # print(f"Binary search 2 took {count} steps")
            return True,count
        if item_list[mid] < to_find:
            lower = mid+1
        else:
            upper=mid-1
    # print(f"Binary search 2 took {count} steps")
    return item_list[lower]==to_find, count
```

In [22]:
```python
def binary_search_3(item_list, to_find):
    left=0
    right=len(item_list)-1
    count = 0
    while left != right:
        count += 1
        mid = (left+right)//2
        if item_list[mid] < to_find:
            left = mid+1
        elif item_list[mid] > to_find:
            right = mid
        else:
            # print(f"Binary search 3 took {count} steps\n\n")
            return True,count
```

```
    # print(f"Binary search 3 took {count} steps\n\n")
    return item_list[left]==to_find, count
```

Compare the performance of each implementation (or variation) with two lists:

1. List is odd numbered (minimum 1000 integers)
2. List is even numbered (minimum 1000 integers)

Run the above experiments when the item to be found is:

1. At the begining of the list.
2. Towards the end of the list.
3. Right at the middle of the list.

The above three combinations would yield 3X2 experiments. Provide detailed outline of the experiments, plots, and a brief description of the observations in the reflections section.

In [23]:

```
def generate_test_data(size):
    # Determine the lengths of the lists
    # Length of odd-numbered list: random odd number between size and 1.5 * siz
    random_length = random.choice(range(size,1000*size))
    parity = random_length%2
    odd_length = random_length + parity + 1
    even_length = random_length + parity


    # Generate random numbers
    numbers = range(odd_length + even_length)

    # Separate the numbers into two lists and ensure the lengths are odd and ev
    odd_numbered_list = numbers[:odd_length]
    even_numbered_list = numbers[odd_length:odd_length + even_length]

    return odd_numbered_list, even_numbered_list
```

In [24]:

```
def conduct_multiple_trials(num_trials):
    size = 1000
    all_results = []

    for _ in range(num_trials):
        odd_numbered_list, even_numbered_list = generate_test_data(size)
        odd_length = len(odd_numbered_list)
        even_length = len(even_numbered_list)
        positions = {'odd': {'beginning': 0, 'middle': (odd_length // 2), 'end'
        binary_search_functions = [binary_search_1, binary_search_2, binary_sea
        results = {'odd': {key: {'time': [], 'found': [], 'correct': [], 'steps
```

```python
                'even': {key: {'time': [], 'found': [], 'correct': [], 'step

        for list_type in ['odd', 'even']:
            for position_name, position_index in positions[list_type].items():
                item_list = odd_numbered_list if list_type == 'odd' else even_n
                target = item_list[position_index]
                for search_function in binary_search_functions:
                    start_time = time.time()
                    found, count = search_function(item_list, target)
                    end_time = time.time()
                    duration = end_time - start_time
                    #is_correct will be false if item was found but doesn't mat
                    #is_correct will be true if
                    is_correct = (found == True and item_list[position_index] =
                    results[list_type][position_name]['time'].append(duration)
                    results[list_type][position_name]['found'].append(found)
                    results[list_type][position_name]['correct'].append(is_corr
                    results[list_type][position_name]['steps'].append(count)

        all_results.append(results)
    return all_results
```

In [25]:

```python
def plot_average_times(all_results):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('Average Execution Times')

    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]
            # Initialize an array to store average times for the three function
            avg_times = []

            # Calculate average time for each function
            for func_index in range(3):  # There are 3 binary search functions
                # Extract times for this function across all trials
                times = [trial[list_type][position_name]['time'][func_index] fo
                # Calculate average time and append to avg_times
                avg_times.append(sum(times) / len(times) if times else 0)

            # Plotting the average times for the three functions
            ax.bar(['binary_search_1', 'binary_search_2', 'binary_search_3'], a
            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Average Time (seconds)')
            ax.set_xlabel('Function')
            ax.grid(True)

    plt.tight_layout()
    plt.show()
```

In [26]:
```
def plot_average_steps(all_results):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('Average Execution Steps')

    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]
            # Initialize an array to store average times for the three function
            avg_times = []

            # Calculate average time for each function
            for func_index in range(3):  # There are 3 binary search functions
                # Extract times for this function across all trials
                times = [trial[list_type][position_name]['steps'][func_index] f
                # Calculate average time and append to avg_times
                avg_times.append(sum(times) / len(times) if times else 0)

            # Plotting the average times for the three functions
            ax.bar(['binary_search_1', 'binary_search_2', 'binary_search_3'], a
            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Average Time (seconds)')
            ax.set_xlabel('Function')
            ax.grid(True)

    plt.tight_layout()
    plt.show()
```

In [27]:
```
def plot_average_steps_marginals(all_results):
    # Initialize a dictionary to store the sum of times and count for calculati
    marginal_sums = {'even': [0, 0, 0], 'odd': [0, 0, 0], 'beginning': [0, 0, 0
    counts = {'even': 0, 'odd': 0, 'beginning': 0, 'middle': 0, 'end': 0, 'tota

    # Plotting each individual case
    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            plt.figure(figsize=(6, 4))
            avg_times = []
            for func_index in range(3):  # There are 3 binary search functions
                times = [trial[list_type][position_name]['steps'][func_index] f
                if times:
                    avg_time = sum(times) / len(times)
                    avg_times.append(avg_time)
                    # Add to marginal sums and counts
                    marginal_sums[list_type][func_index] += avg_time
                    marginal_sums[position_name][func_index] += avg_time
                    marginal_sums['total'][func_index] += avg_time
                else:
                    avg_times.append(0)
            counts[list_type] += 1
```

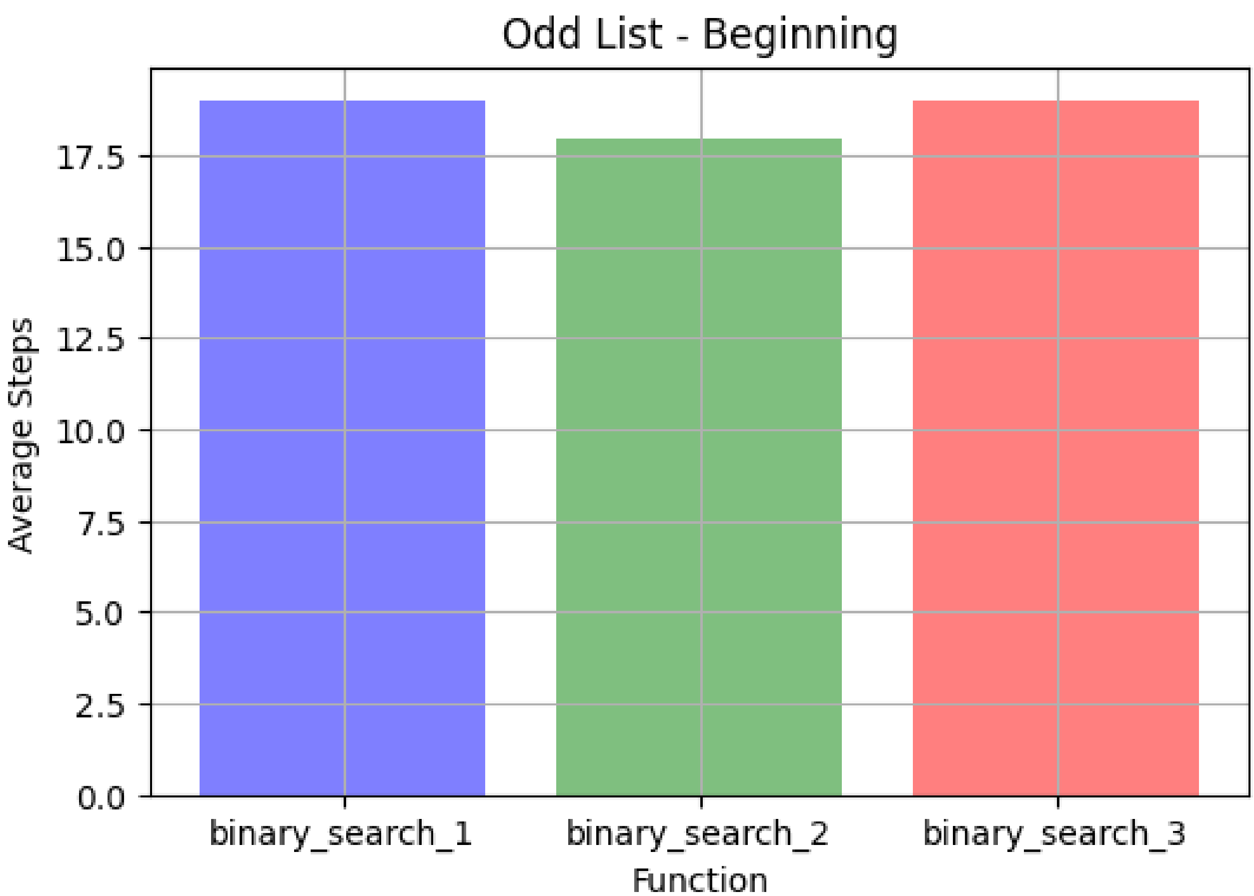
```
            counts[position_name] += 1
            counts['total'] += 1

            plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3'],
            plt.title(f'{list_type.capitalize()} List - {position_name.capitali
            plt.ylabel('Average Steps')
            plt.xlabel('Function')
            plt.grid(True)
            plt.show()

    # Plotting marginal cases
    for marginal in ['even', 'odd', 'beginning', 'middle', 'end']:
        plt.figure(figsize=(6, 4))
        avg_times = [sum_time / counts[marginal] if counts[marginal] else 0 for
        plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3'], avg_
        plt.title(f'Marginal Average for {marginal.capitalize()}')
        plt.ylabel('Average Steps')
        plt.xlabel('Function')
        plt.grid(True)
        plt.show()

    # Plotting overall average
    plt.figure(figsize=(6, 4))
    avg_times = [sum_time / counts['total'] if counts['total'] else 0 for sum_t
    plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3'], avg_time
    plt.title('Overall Average')
    plt.ylabel('Average Steps')
    plt.xlabel('Function')
    plt.grid(True)
    plt.show()

# Call the function with your data
# plot_average_steps_marginals(all_results)
```

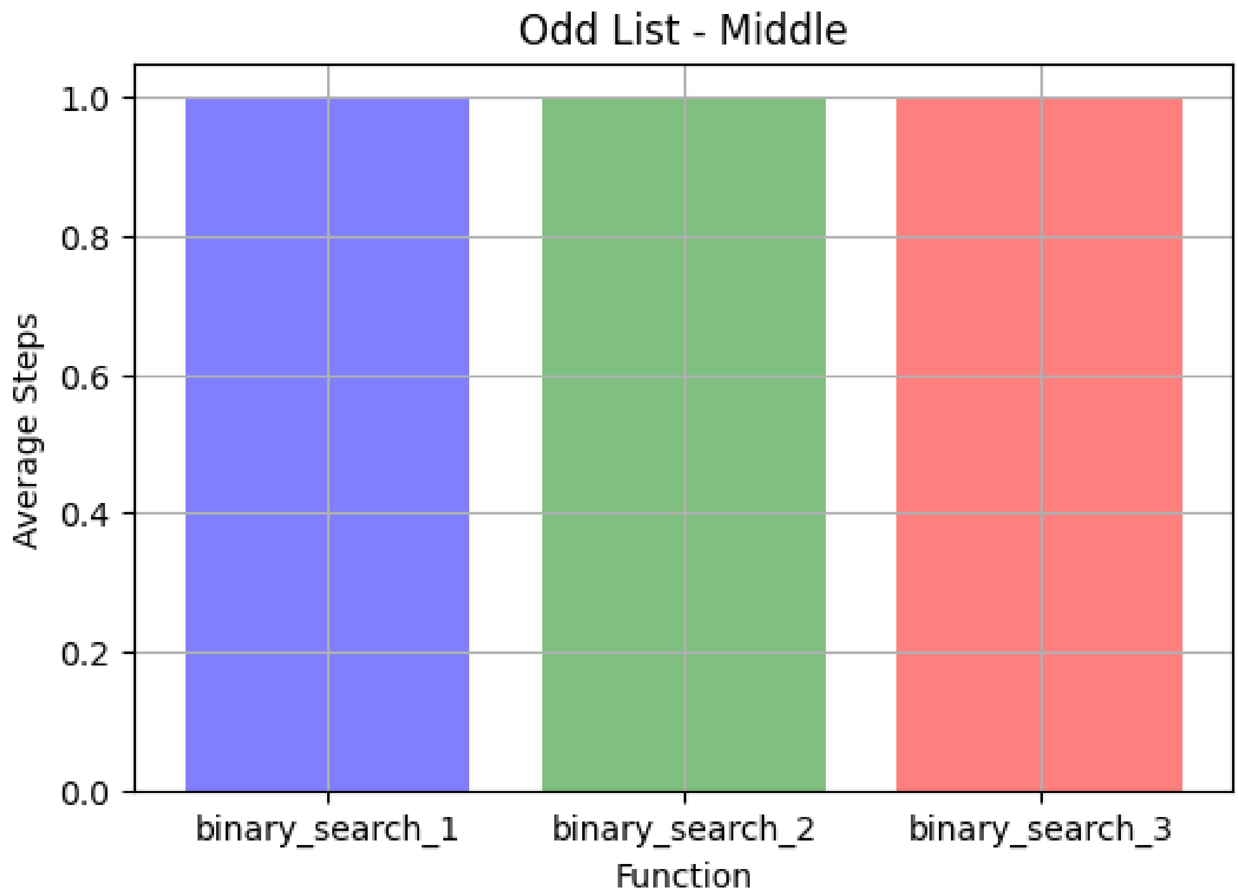
In [28]:

```
def plot_histograms(all_results, num_bins=10):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12), sharex='col', sharey='row'
    fig.suptitle('Histograms of Execution Times')

    # Define colors for the histograms of each binary search function
    colors = ['blue', 'green', 'red']

    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]

            # Extract times for each binary search function
            for func_index in range(3):
                times = [trial[list_type][position_name]['steps'][func_index] f

                # Plot histogram for each binary search function
```

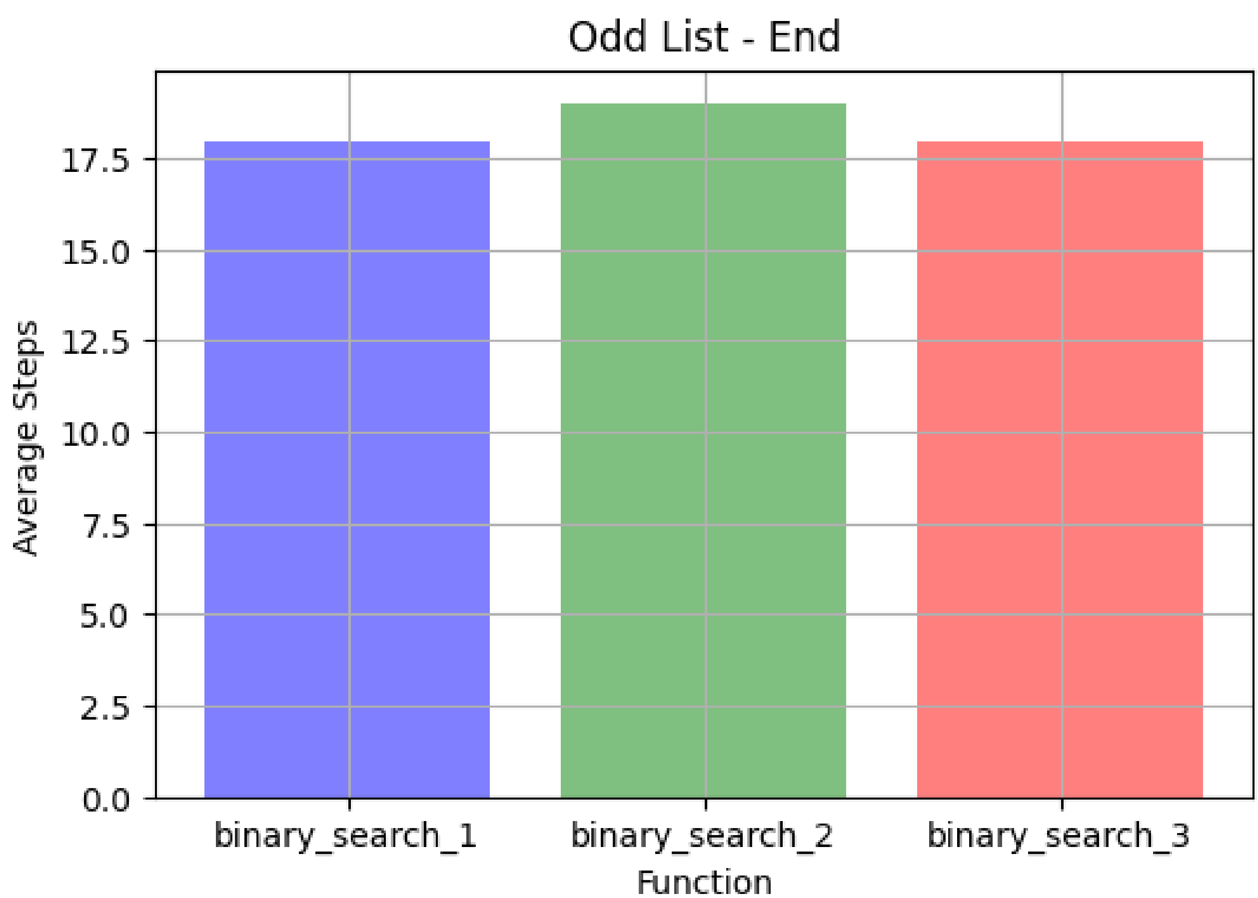
```
                ax.hist(times, bins=num_bins, color=colors[func_index], alpha=0

            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Frequency')
            ax.set_xlabel('Time (seconds)')
            ax.legend()
            ax.grid(axis='y')

    plt.tight_layout()
    plt.show()
```

In [29]:
```
num_trials = 10000  # or any other number of trials you want
all_results = conduct_multiple_trials(num_trials)
# plot_average_times(all_results)
plot_average_steps_marginals(all_results)
# plot_histograms(all_results)
```

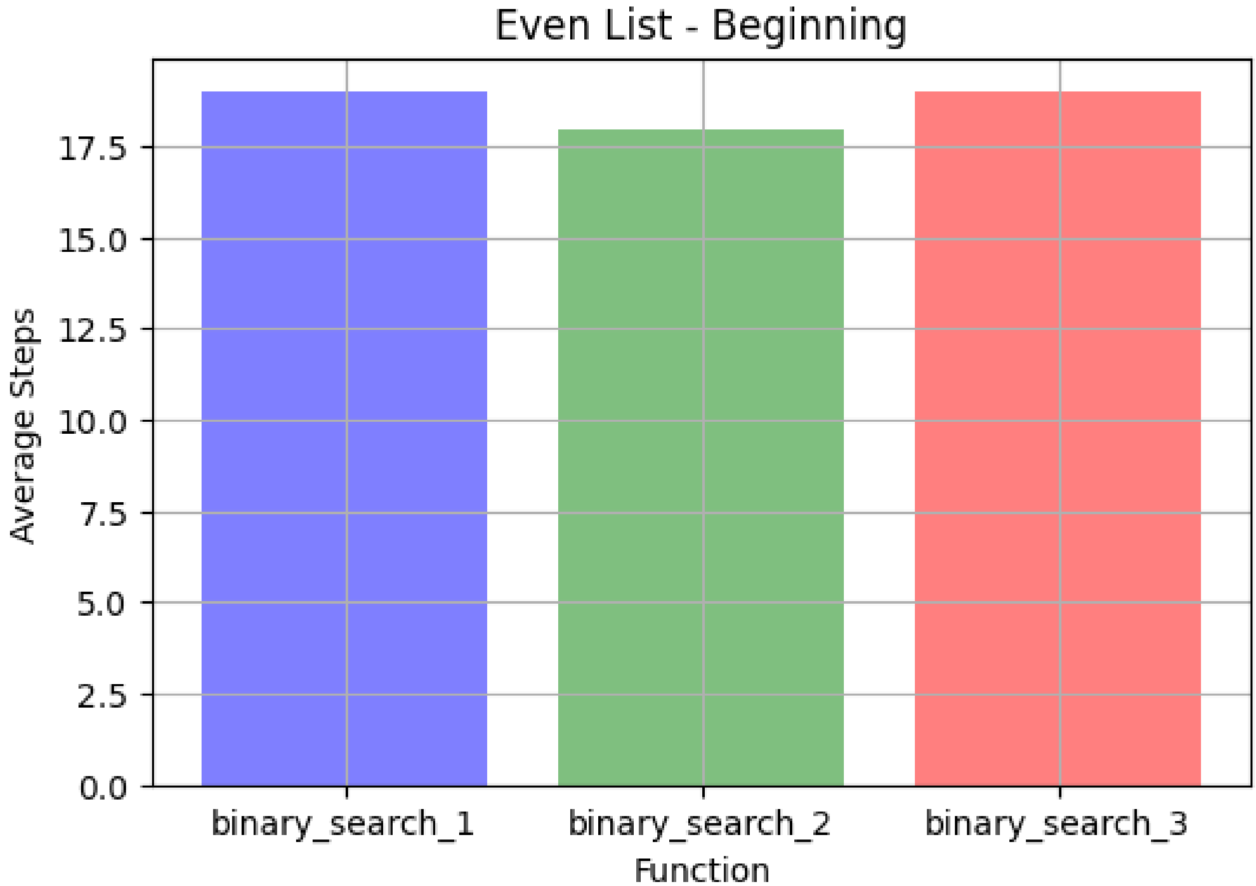


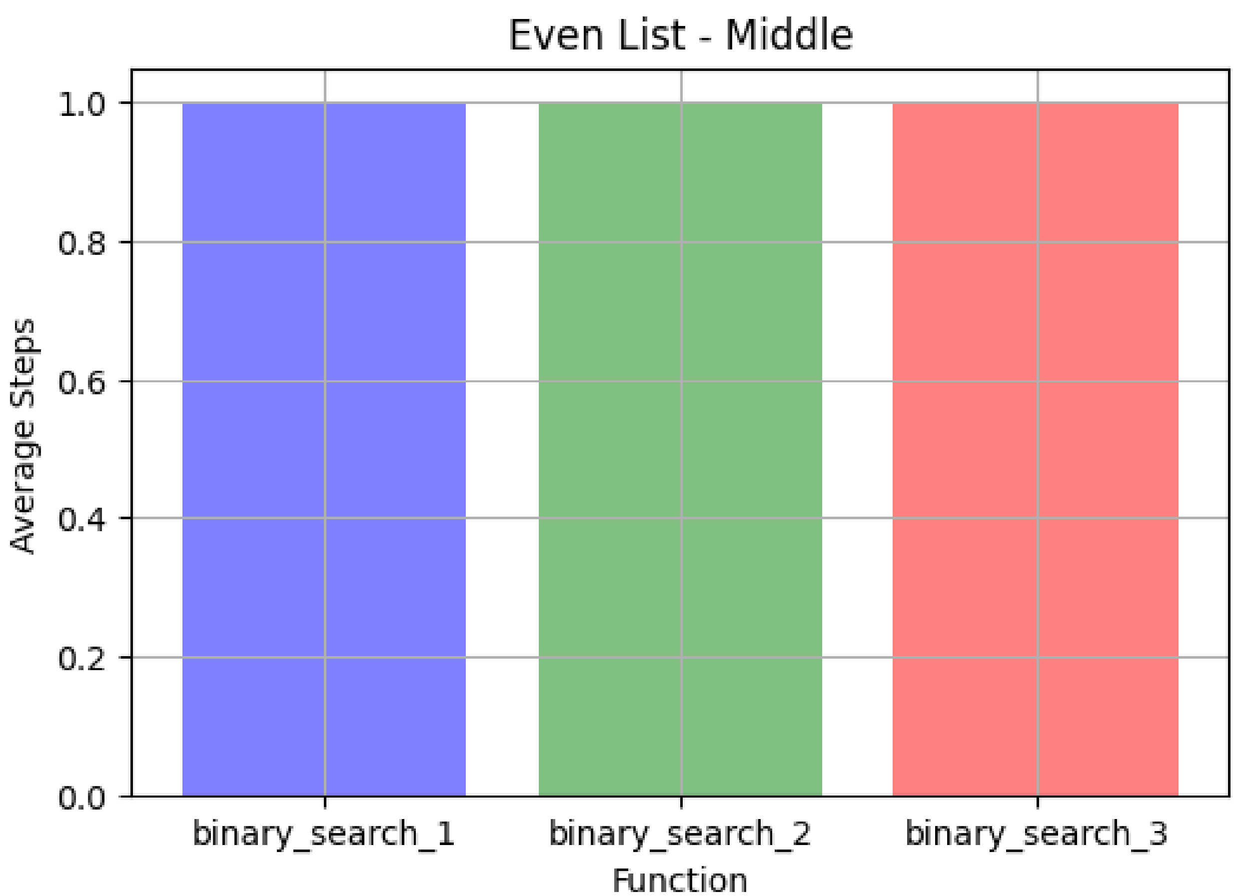


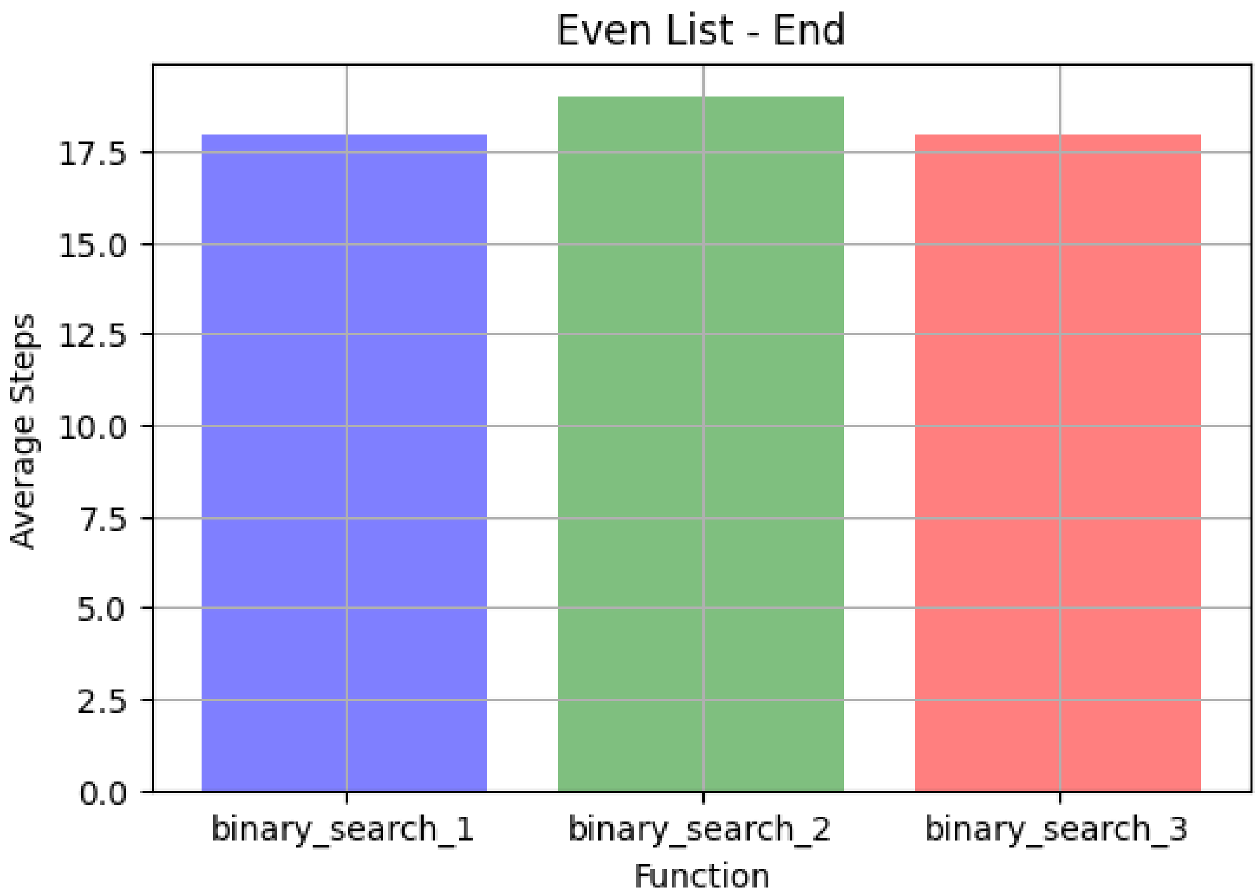
Odd List - Middle
Average Steps
1.0
0.8
0.6
0.4
0.2
0.0
binary_search_1
binary_search_2
binary_search_3
Function



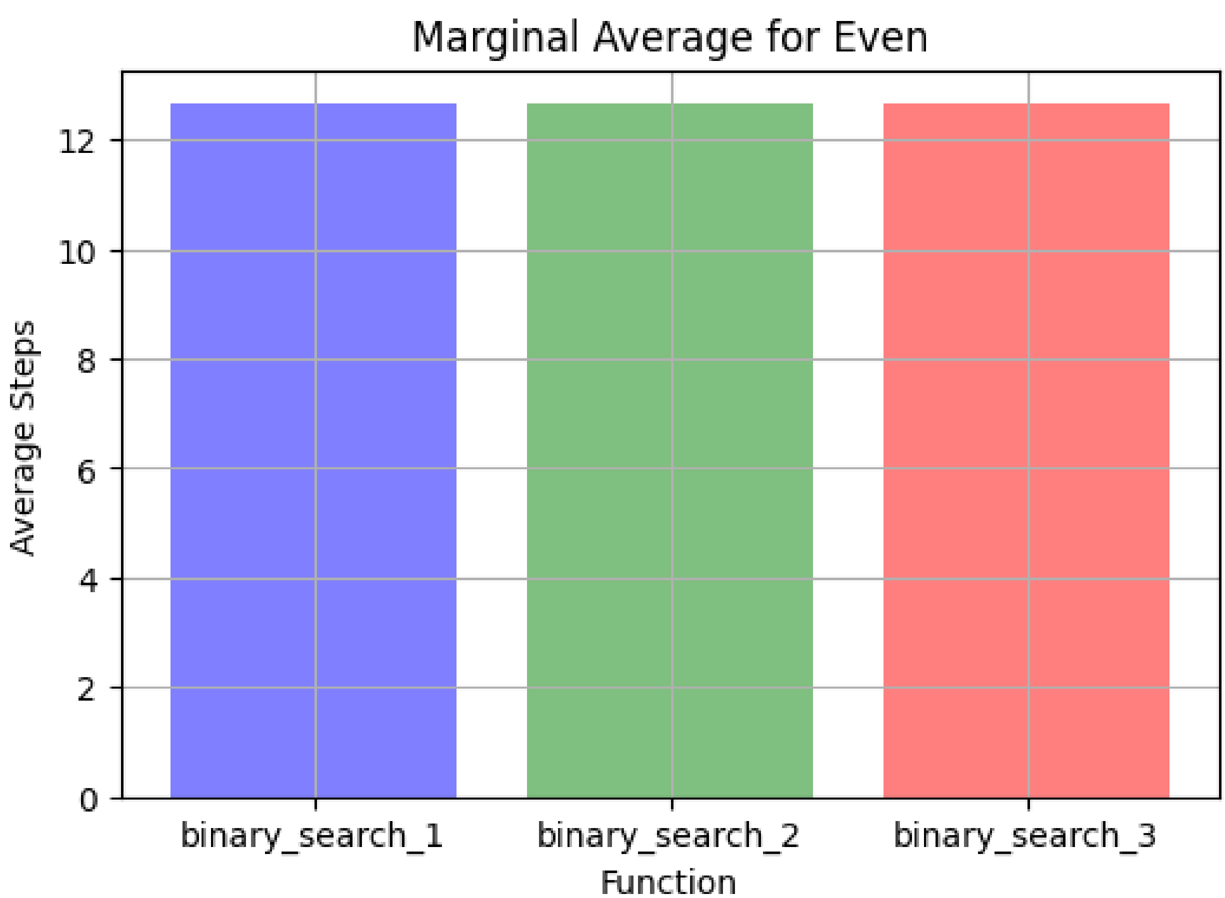
Odd List - End
Average Steps
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
binary_search_1
binary_search_2
binary_search_3
Function


Even List - Beginning
Average Steps
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
binary_search_1
binary_search_2
binary_search_3
Function

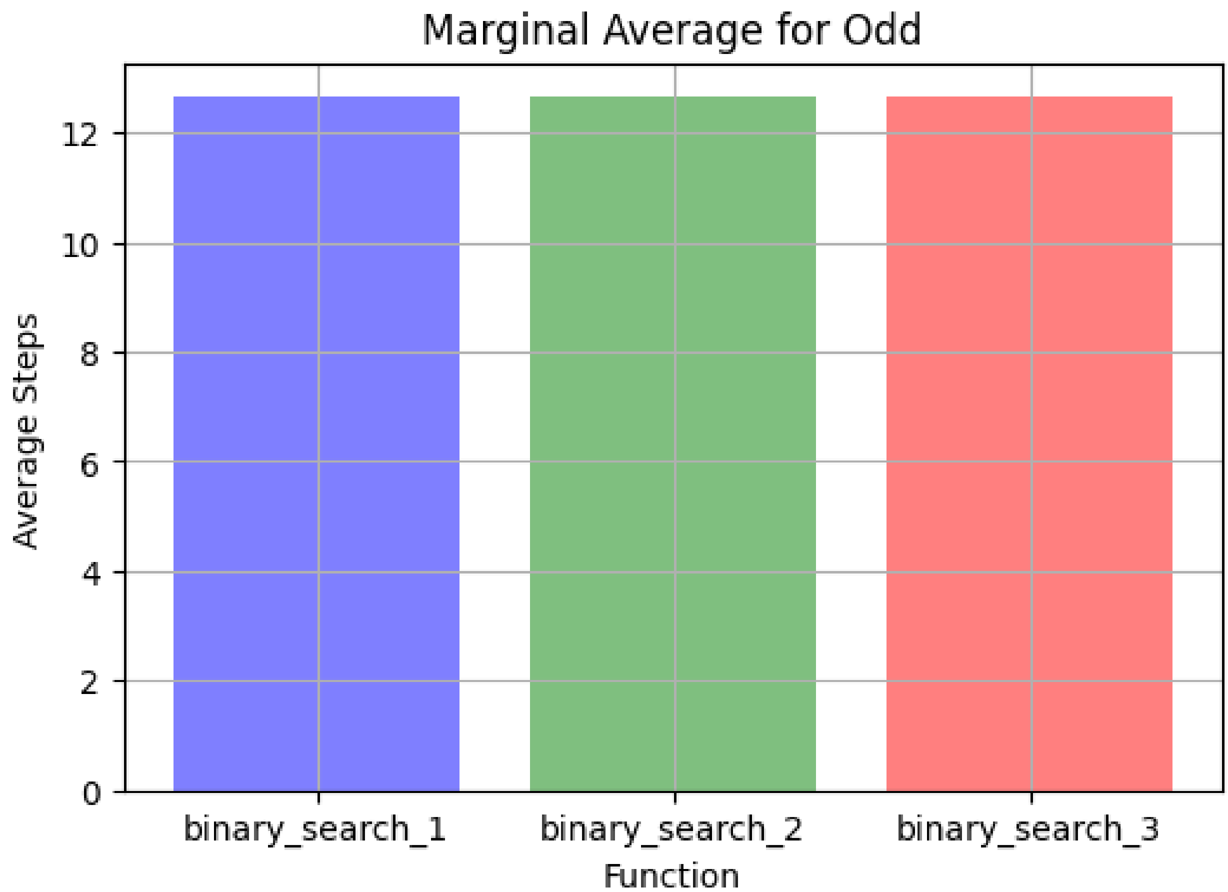

Even List - Middle
Average Steps
1.0
0.8
0.6
0.4
0.2
0.0
binary_search_1
binary_search_2
binary_search_3
Function



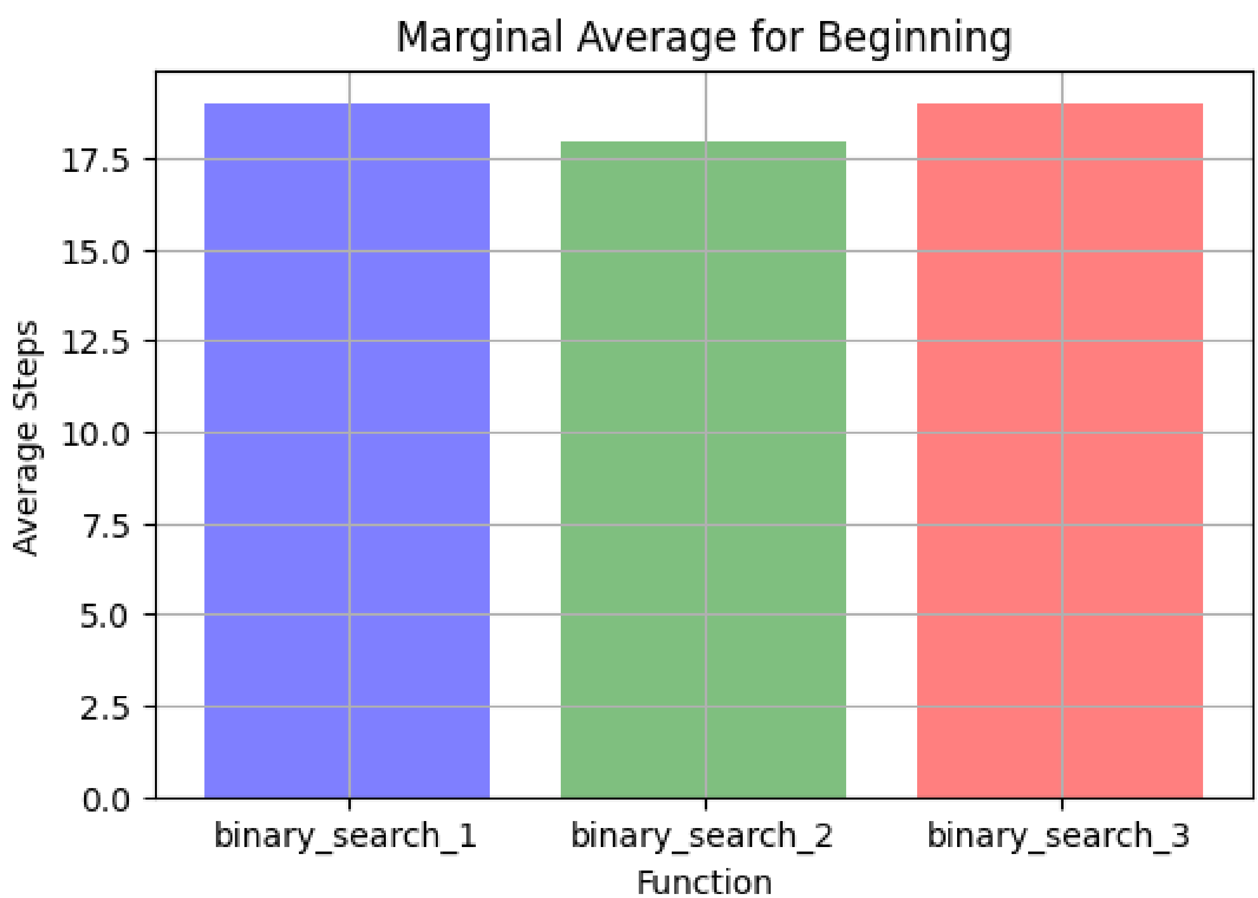
Even List - End
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Average Steps
binary_search_1
binary_search_2
binary_search_3
Function



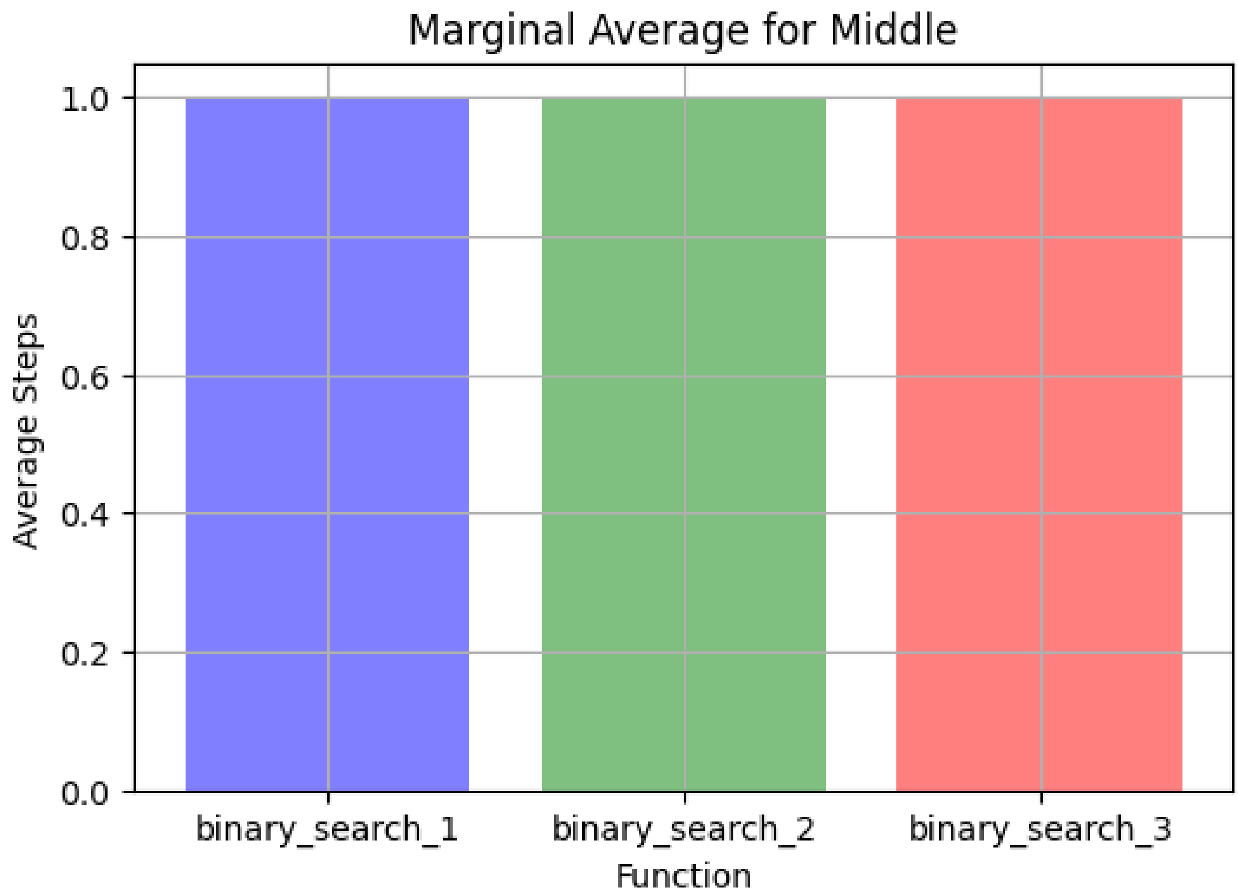
Marginal Average for Even
12
10
8
6
4
2
0
Average Steps
binary_search_1
binary_search_2
binary_search_3
Function



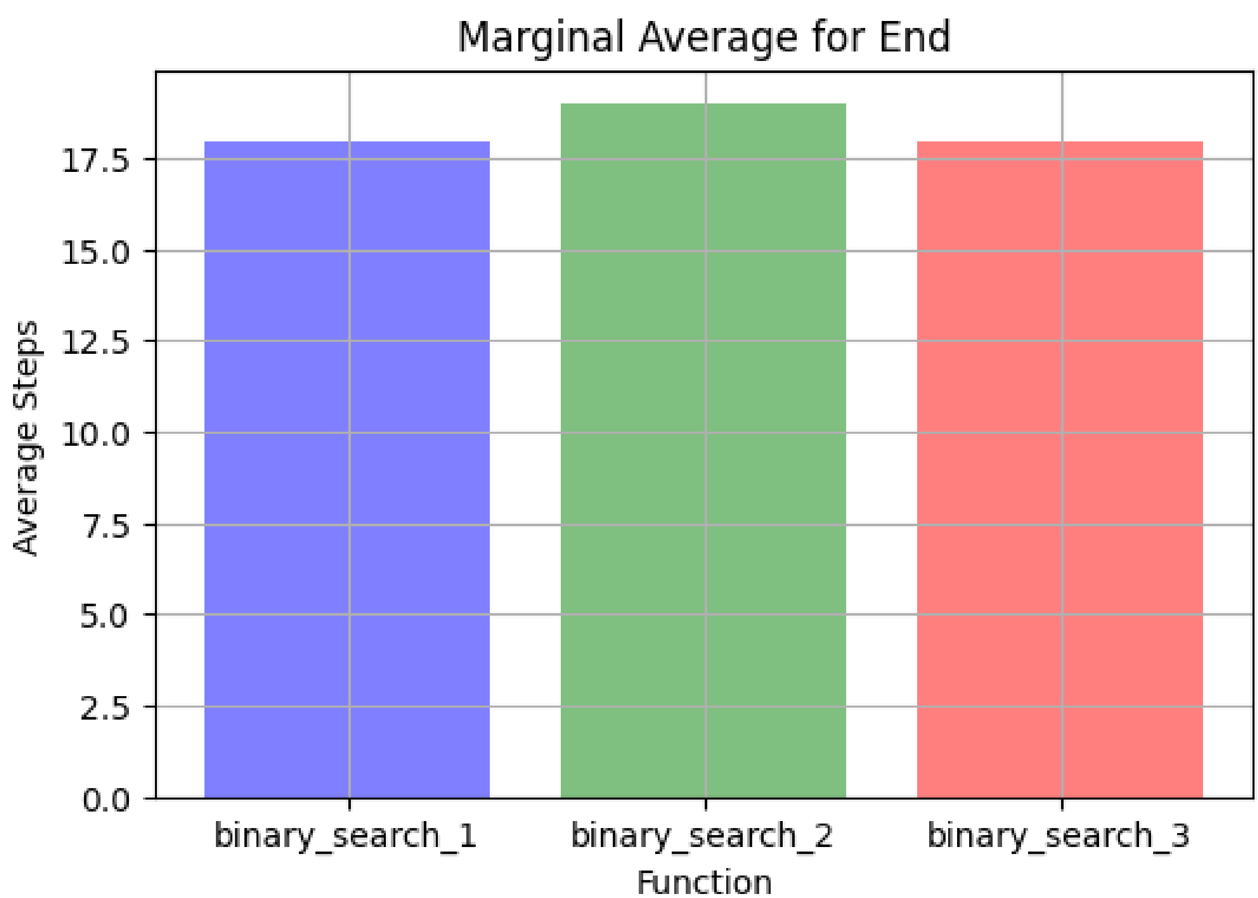
Marginal Average for Odd
Average Steps
0
2
4
6
8
10
12
binary_search_1
binary_search_2
binary_search_3
Function



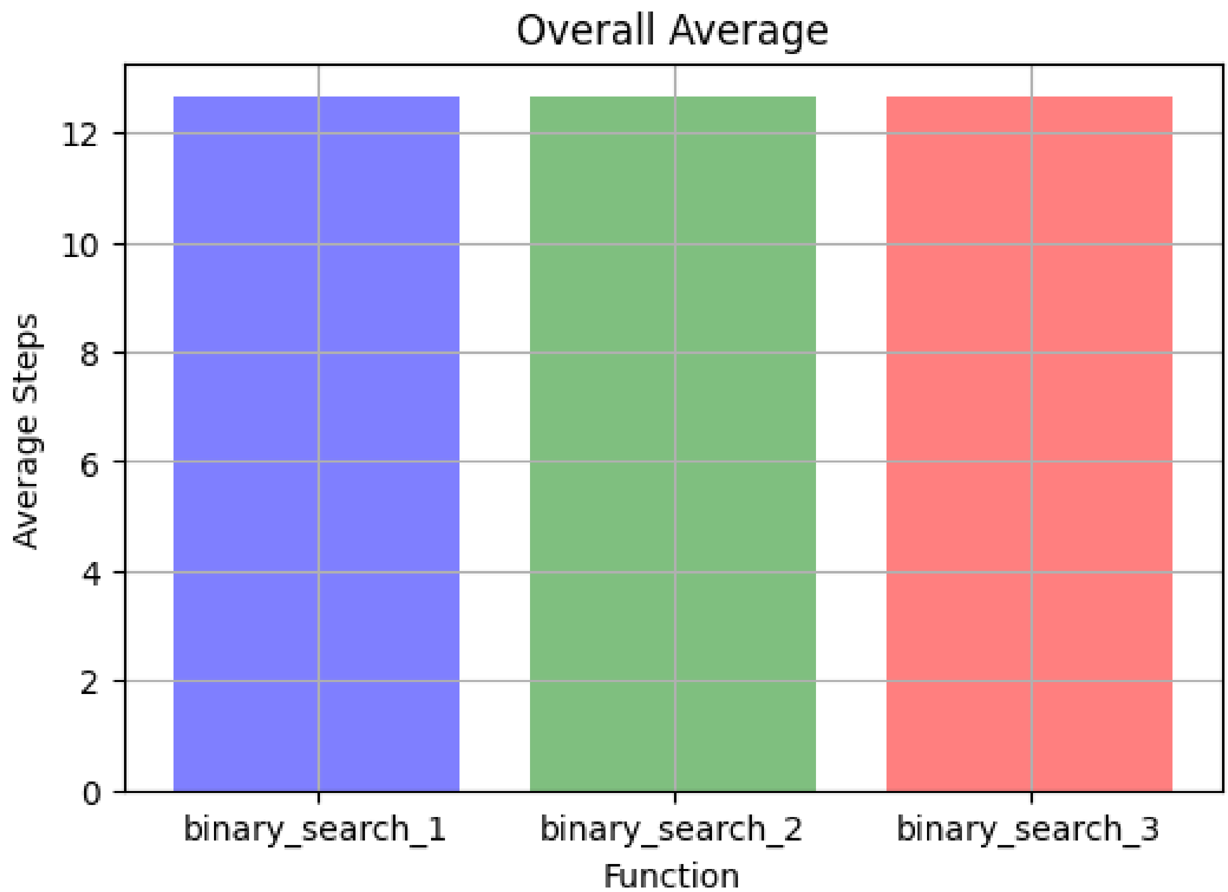
Marginal Average for Beginning
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
Function



Marginal Average for Middle
Average Steps
1.0
0.8
0.6
0.4
0.2
0.0
binary_search_1
binary_search_2
binary_search_3
Function



Marginal Average for End
Average Steps
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
binary_search_1
binary_search_2
binary_search_3
Function



Reflection:

Experiment overview

1. Goal: The goal was to evaluate and compare the three binary search algorithms provided under various conditions.
2. Test data generation:
   - We generated random odd and even numbered lists
   - odd lists have the lengths between 'size' and '1.5 * size'
   - made sure that even lists match lengths of corresponding odd lists
3. Trials and measurements:
   - ran multiple trials for variablity
   - recorded execution time, if the item was found, correctness and step count
   - tested at different positions such as beginning, middle and end of the list.
4. Plots:
   - created plots which made comparision between the searches much easier and visually appealing.
   - Marginal average for even graph corresponds to the case where we are searching for a random item in an even list
   - Marginal average for odd graph corresponds to the case where we are searching for a random item in an odd list

- Marginal average for beginning, middle and end graphs respectively correspond to the case where we are searchin for the beginning, middle and end items in a randomly generated list.

5. Modifications: I made one modification to the code provided, by including a variable count, which keeps track of the number of iterations taking place when searching for an item through the list. This is very helpful to understand the reason for the behaviour of the graphs.

Results and their Interpretations

1. From the graphs, we observe that the middle item is found at the same rate of time in all searches. This is because in all the algorithms, the first condition is if the middle item is the target.
2. binary_search_1 and binary_search_3 have very similar behaviour because of the way their upper and lower bounds are being updated. This, in case of binary search for beginning item in the list, always increases their time as the number of iterations are also higher.
3. similarly, binary_search_2 acts differently than 1 and 3 because of the way its upper and lower bounds are being updated. Because of this, we observe that whenver binary_search 1 and 3 take longer time, binary_search_2 takes lesser time. Again, this is because of the number of iterations are lesser in that case.

Part C

Recall that I discussed in the class, the possibility of "reducing the comparisons" in Binary Search implementation. One solution came up is to remove the comparison with "mid". If you design an experiment to test this, you will soon realize that while this speeds up the execution time by reducing the number of comparisons needed, it fails when the element to be searched is right in the middle. So are there any ways to improve the speed of Binary Search that is not dependent on data? The answer is recursion! In this section, implement a Binary Search recursively.

In [30]:
```
def binary_search_4(item_list, to_find):
    def recursive_binary_search(low, high, count = 0):
        # If the range is small, perform a direct search.
        # This is necessary to ensure that the target is not missed.
        if high - low <= 1:
            return (item_list[low] == to_find or item_list[high] == to_find, co
```

```
        mid = low + (high - low) // 2  # Calculate mid

        # If the target is not at the middle, choose a side to continue
        if to_find < item_list[mid]:
            return recursive_binary_search(low, mid - 1, count + 1)  # Search l
        elif to_find > item_list[mid]:
            return recursive_binary_search(mid + 1, high, count + 1)  # Search

        # If none of the conditions met, the target is at the middle
        return (True, count)

    return recursive_binary_search(0, len(item_list) - 1)
```

Run all the experiments in Part B comapring all 4 implementations under all 6 cases. Plot the timings, and describe the results in the below section. Write a short description of your observation; why is recursion better in this case?

In [31]:
```
def conduct_multiple_trials(num_trials):
    size = 1000
    all_results = []

    for _ in range(num_trials):
        odd_numbered_list, even_numbered_list = generate_test_data(size)
        odd_length = len(odd_numbered_list)
        even_length = len(even_numbered_list)
        positions = {'odd': {'beginning': 0, 'middle': odd_length // 2, 'end':

        binary_search_functions = [binary_search_1, binary_search_2, binary_sea
        results = {'odd': {key: {'time': [], 'found': [], 'correct': [], 'steps
                   'even': {key: {'time': [], 'found': [], 'correct': [], 'step

        for list_type in ['odd', 'even']:
            for position_name, position_index in positions[list_type].items():
                item_list = odd_numbered_list if list_type == 'odd' else even_n
                target = item_list[position_index]
                for search_function in binary_search_functions:
                    start_time = time.time()
                    found, count = search_function(item_list, target)
                    end_time = time.time()
                    duration = end_time - start_time
                    is_correct = (found == True and item_list[position_index] =
                    results[list_type][position_name]['time'].append(duration)
                    results[list_type][position_name]['found'].append(found)
                    results[list_type][position_name]['correct'].append(is_corr
                    results[list_type][position_name]['steps'].append(count)

        all_results.append(results)
    return all_results
```

In [32]:
```
def plot_average_times(all_results):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('Average Execution Times')

    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]
            # Initialize an array to store average times for the three function
            avg_times = []

            # Calculate average time for each function
            for func_index in range(4):  # There are 4 binary search functions
                # Extract times for this function across all trials
                times = [trial[list_type][position_name]['time'][func_index] fo
                # Calculate average time and append to avg_times
                avg_times.append(sum(times) / len(times) if times else 0)
```

```
            # Plotting the average times for the three functions
            ax.bar(['binary_search_1', 'binary_search_2', 'binary_search_3', 'b
            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Average Time (seconds)')
            ax.set_xlabel('Function')
            ax.grid(True)

    plt.tight_layout()
    plt.show()
```

In [33]:
```
def plot_average_steps(all_results):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12))
    fig.suptitle('Average Execution Steps')

    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]
            # Initialize an array to store average times for the 4 functions
            avg_times = []

            # Calculate average time for each function
            for func_index in range(4):  # There are 4 binary search functions
                # Extract times for this function across all trials
                times = [trial[list_type][position_name]['steps'][func_index] f
                # Calculate average time and append to avg_times
                avg_times.append(sum(times) / len(times) if times else 0)

            # Plotting the average times for the three functions
            ax.bar(['binary_search_1', 'binary_search_2', 'binary_search_3', 'b
            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Average Time (seconds)')
            ax.set_xlabel('Function')
            ax.grid(True)

    plt.tight_layout()
    plt.show()
```

In [34]:
```
def plot_average_steps_marginals(all_results):
    # Initialize a dictionary to store the sum of times and count for calculati
    marginal_sums = {'even': [0, 0, 0, 0], 'odd': [0, 0, 0, 0], 'beginning': [0
    counts = {'even': 0, 'odd': 0, 'beginning': 0, 'middle': 0, 'end': 0, 'tota

    # Plotting each individual case
    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            plt.figure(figsize=(6, 4))
            avg_times = []
            for func_index in range(4):  # There are 4 binary search functions
                times = [trial[list_type][position_name]['steps'][func_index] f
                if times:
```

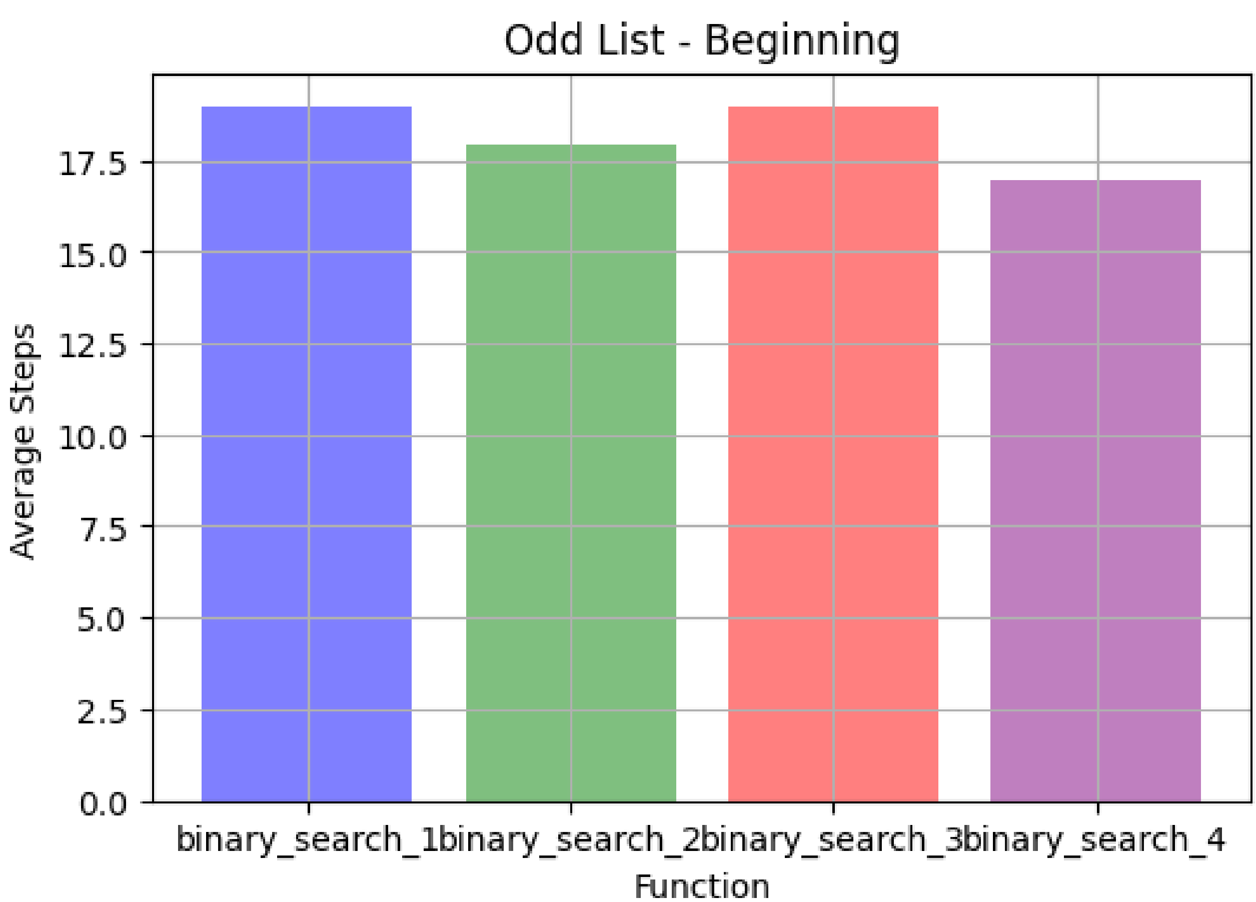
```
                avg_time = sum(times) / len(times)
                avg_times.append(avg_time)
                # Add to marginal sums and counts
                marginal_sums[list_type][func_index] += avg_time
                marginal_sums[position_name][func_index] += avg_time
                marginal_sums['total'][func_index] += avg_time
            else:
                avg_times.append(0)
        counts[list_type] += 1
        counts[position_name] += 1
        counts['total'] += 1

        plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3', '
        plt.title(f'{list_type.capitalize()} List - {position_name.capitali
        plt.ylabel('Average Steps')
        plt.xlabel('Function')
        plt.grid(True)
        plt.show()

    # Plotting marginal cases
    for marginal in ['even', 'odd', 'beginning', 'middle', 'end']:
        plt.figure(figsize=(6, 4))
        avg_times = [sum_time / counts[marginal] if counts[marginal] else 0 for
        plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3', 'bina
        plt.title(f'Marginal Average for {marginal.capitalize()}')
        plt.ylabel('Average Steps')
        plt.xlabel('Function')
        plt.grid(True)
        plt.show()

    # Plotting overall average
    plt.figure(figsize=(6, 4))
    avg_times = [sum_time / counts['total'] if counts['total'] else 0 for sum_t
    plt.bar(['binary_search_1', 'binary_search_2', 'binary_search_3', 'binary_s
    plt.title('Overall Average')
    plt.ylabel('Average Steps')
    plt.xlabel('Function')
    plt.grid(True)
    plt.show()

# Call the function with your data
# plot_average_steps_marginals(all_results)
```

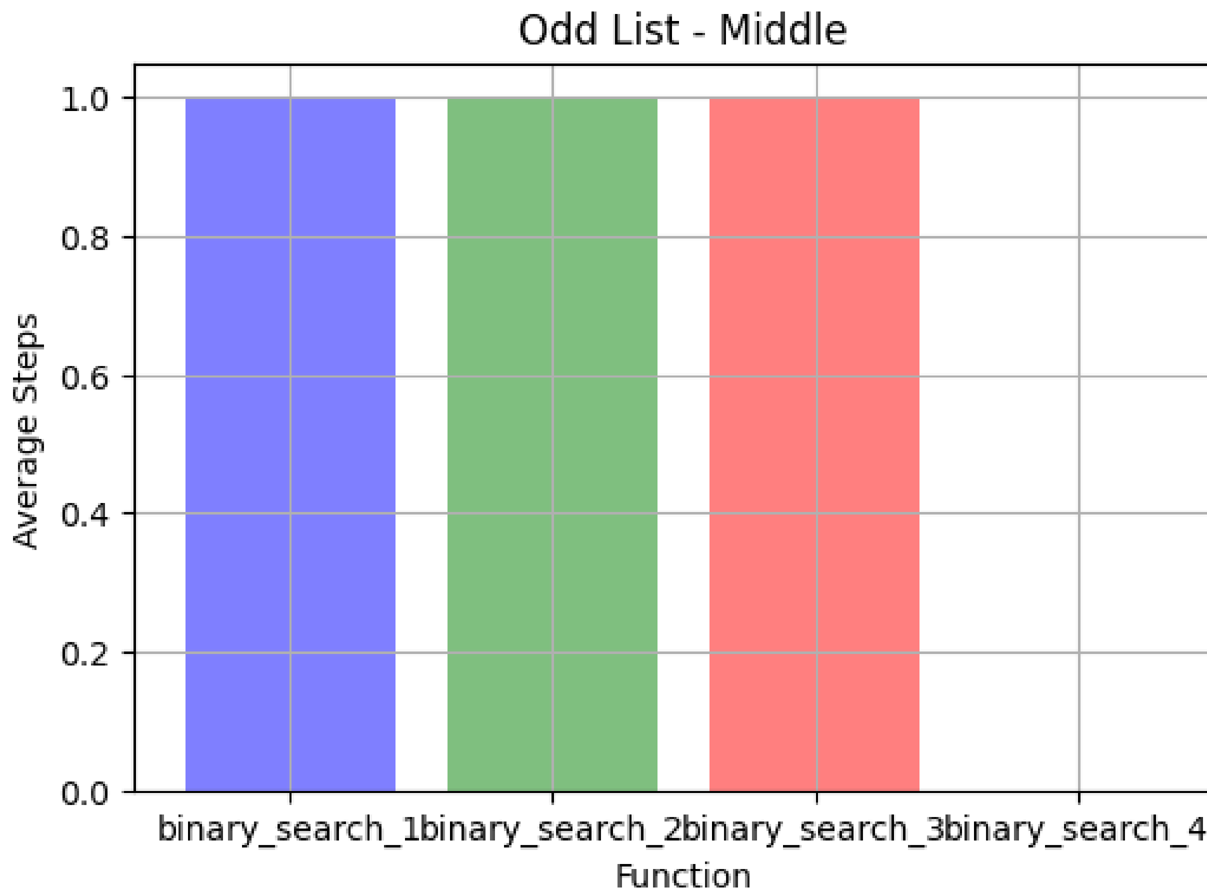
In [35]:

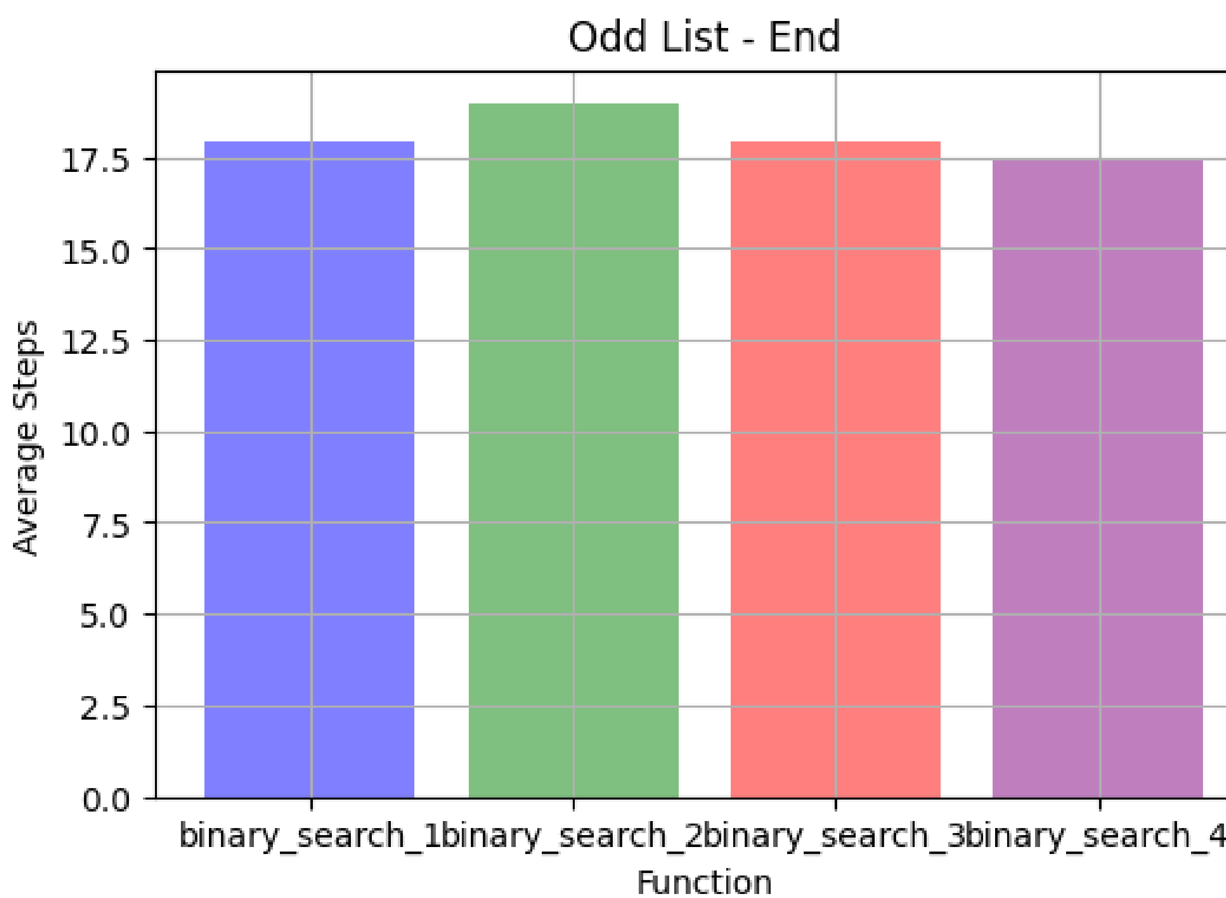
```
def plot_histograms(all_results, num_bins=10):
    fig, axes = plt.subplots(2, 3, figsize=(18, 12), sharex='col', sharey='row'
    fig.suptitle('Histograms of Execution Times')

    # Define colors for the histograms of each binary search function
    colors = ['blue', 'green', 'red', 'purple']
```

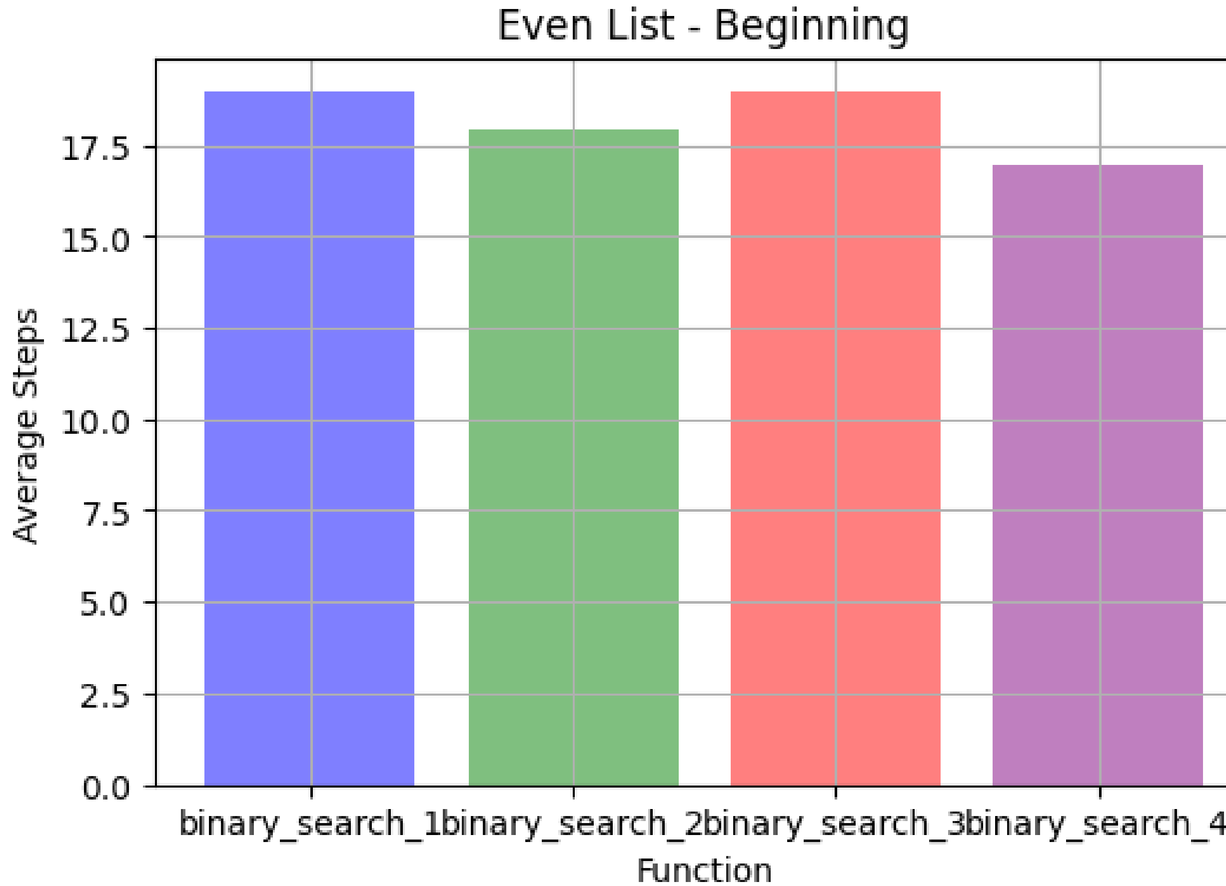
```python
    for i, list_type in enumerate(['odd', 'even']):
        for j, position_name in enumerate(['beginning', 'middle', 'end']):
            ax = axes[i, j]

            # Extract times for each binary search function
            for func_index in range(4):
                times = [trial[list_type][position_name]['steps'][func_index] f

                # Plot histogram for each binary search function
                ax.hist(times, bins=num_bins, color=colors[func_index], alpha=0

            ax.set_title(f'{list_type.capitalize()} List - {position_name.capit
            ax.set_ylabel('Frequency')
            ax.set_xlabel('Time (seconds)')
            ax.legend()
            ax.grid(axis='y')

    plt.tight_layout()
    plt.show()
```

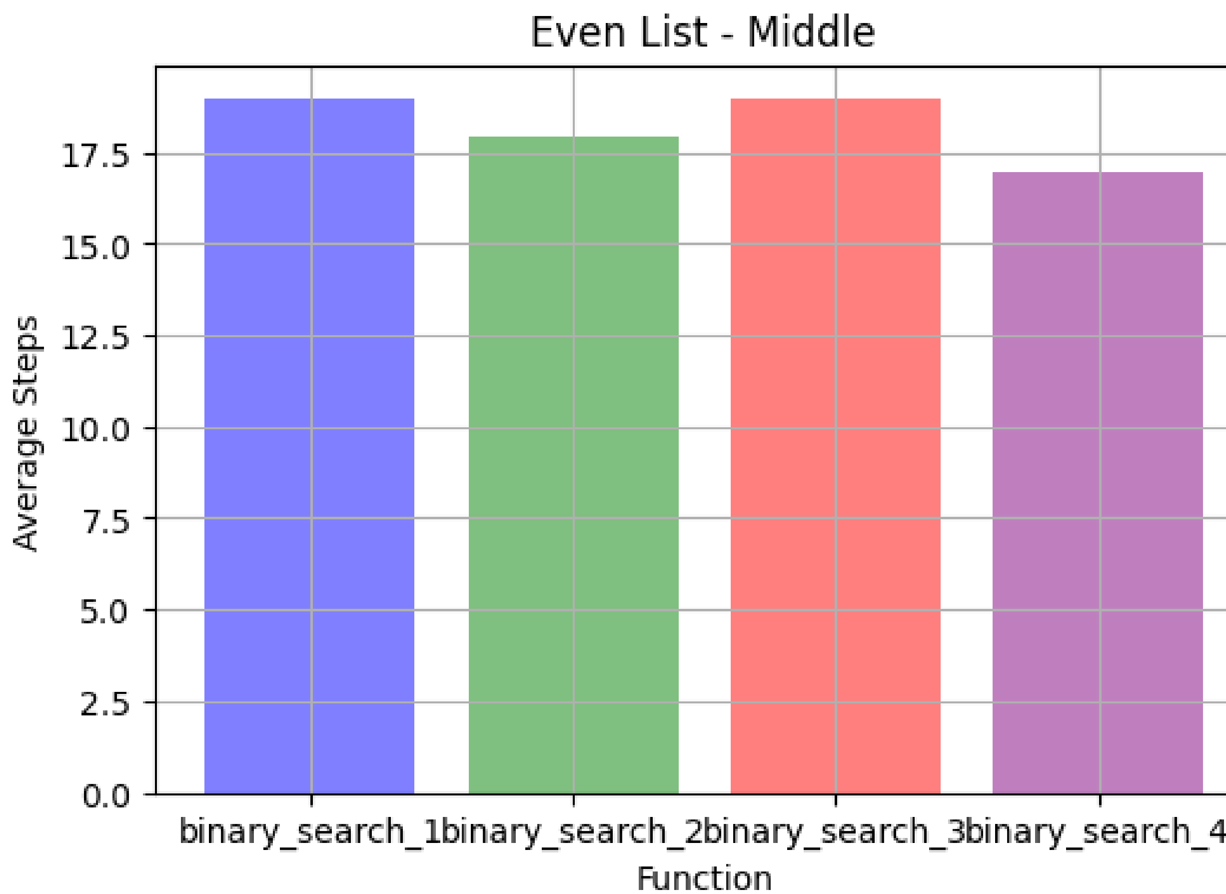
In [36]:
```python
num_trials = 10000  # or any other number of trials you want
all_results = conduct_multiple_trials(num_trials)
# plot_average_times(all_results)
plot_average_steps_marginals(all_results)
# plot_histograms(all_results)
```

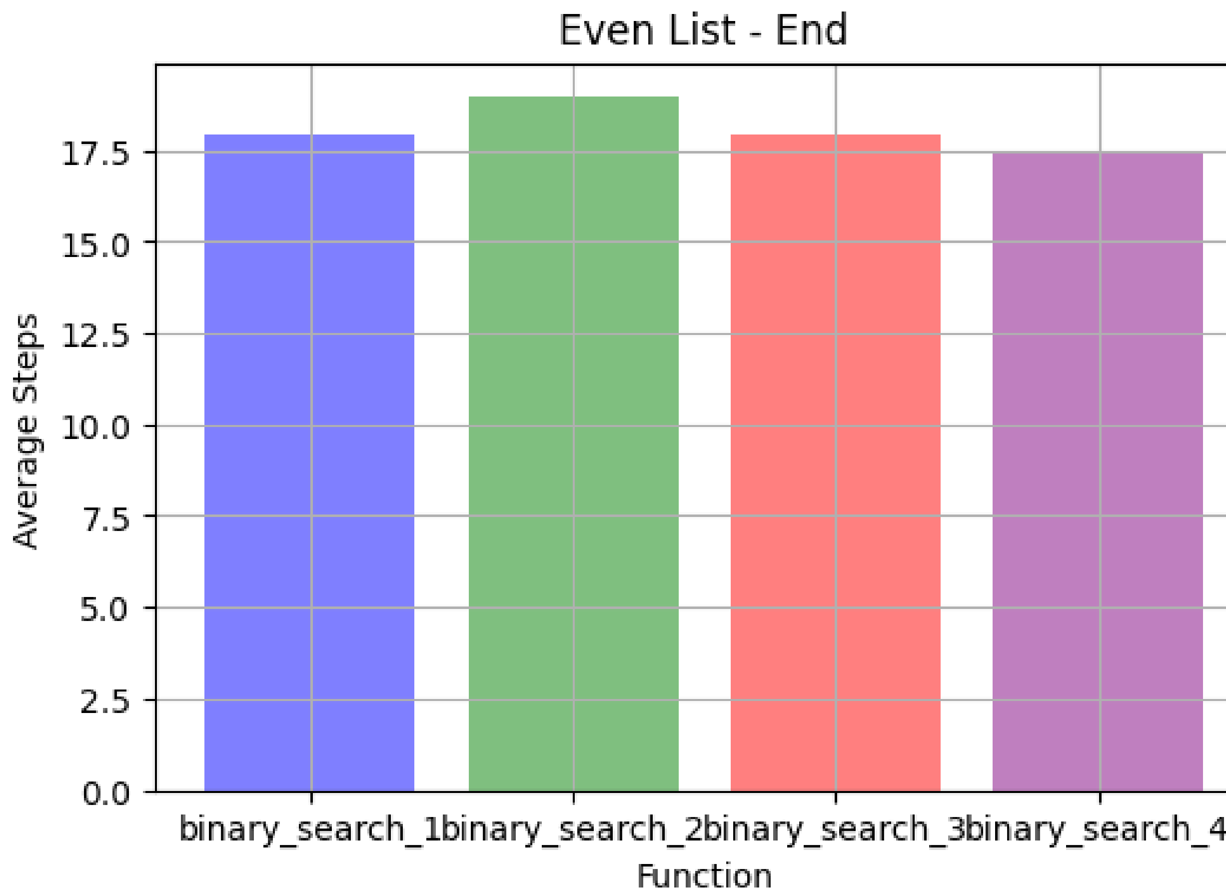


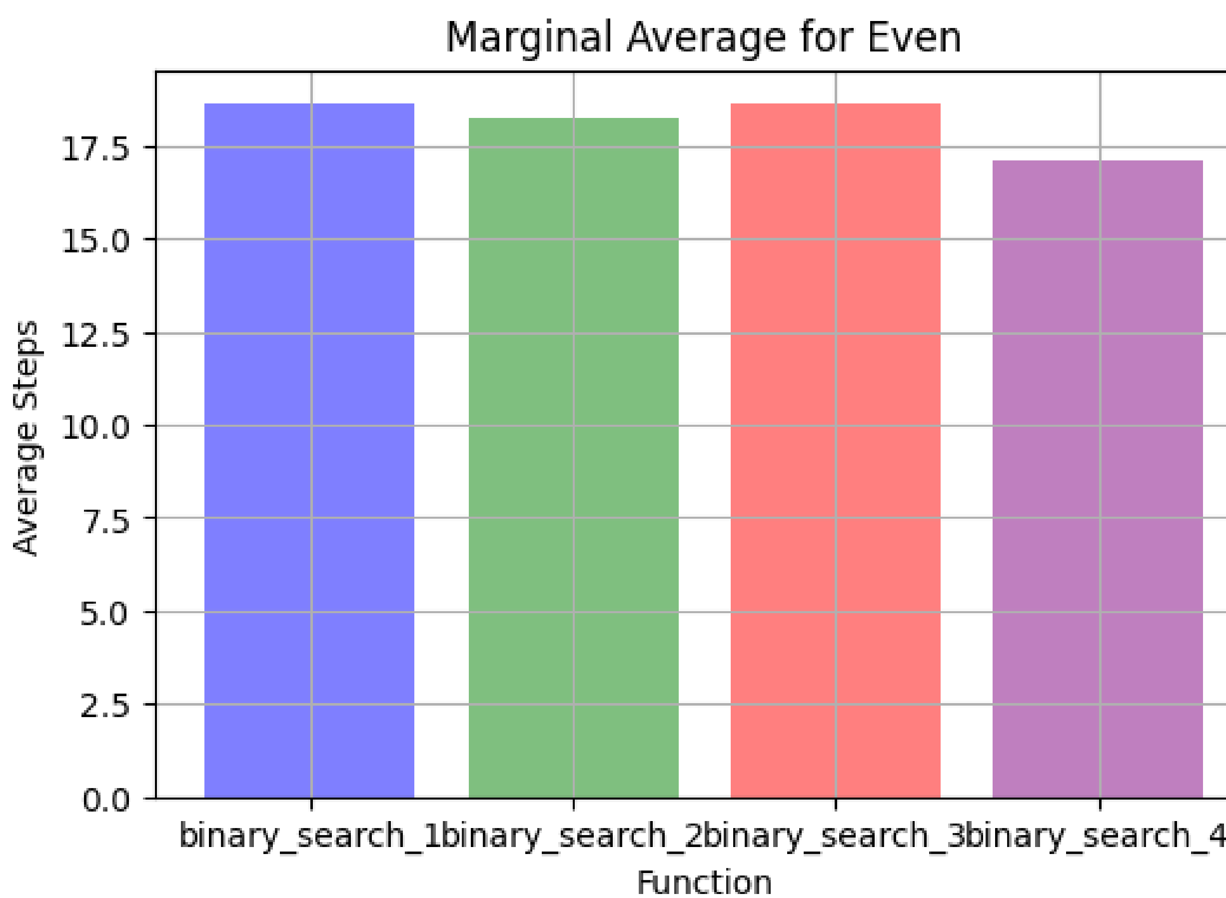

Odd List - Middle
Average Steps
1.0
0.8
0.6
0.4
0.2
0.0
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function

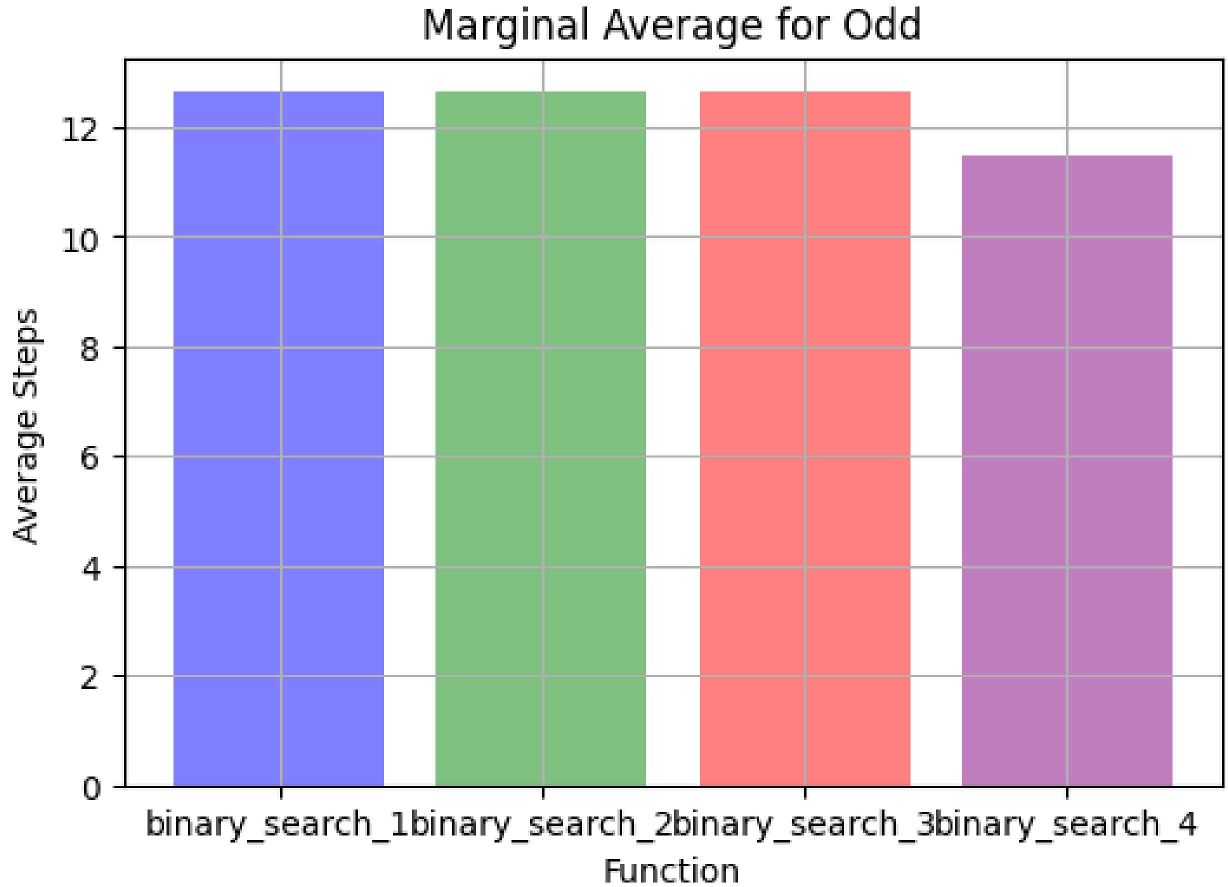

Odd List - End
Average Steps
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function

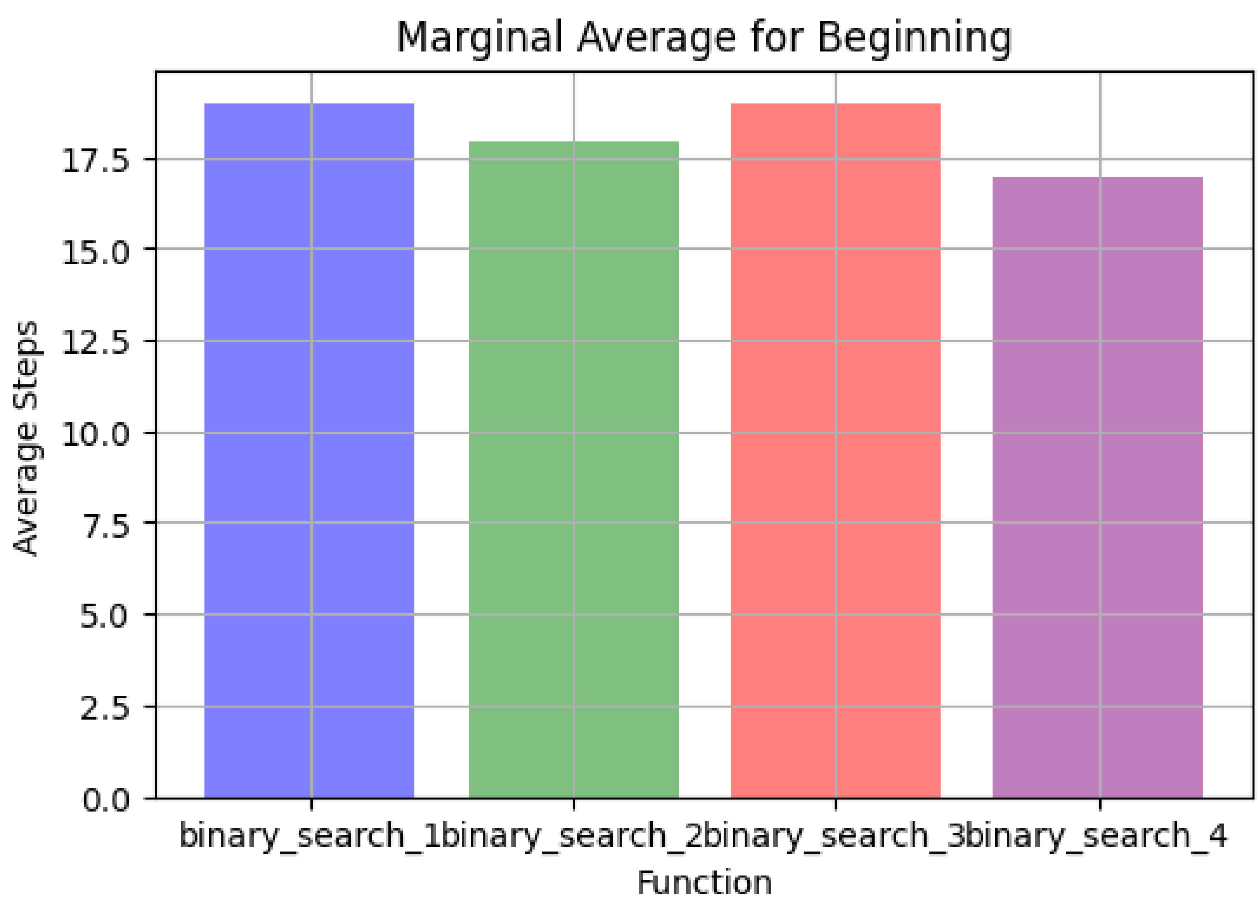

Even List - Beginning
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function

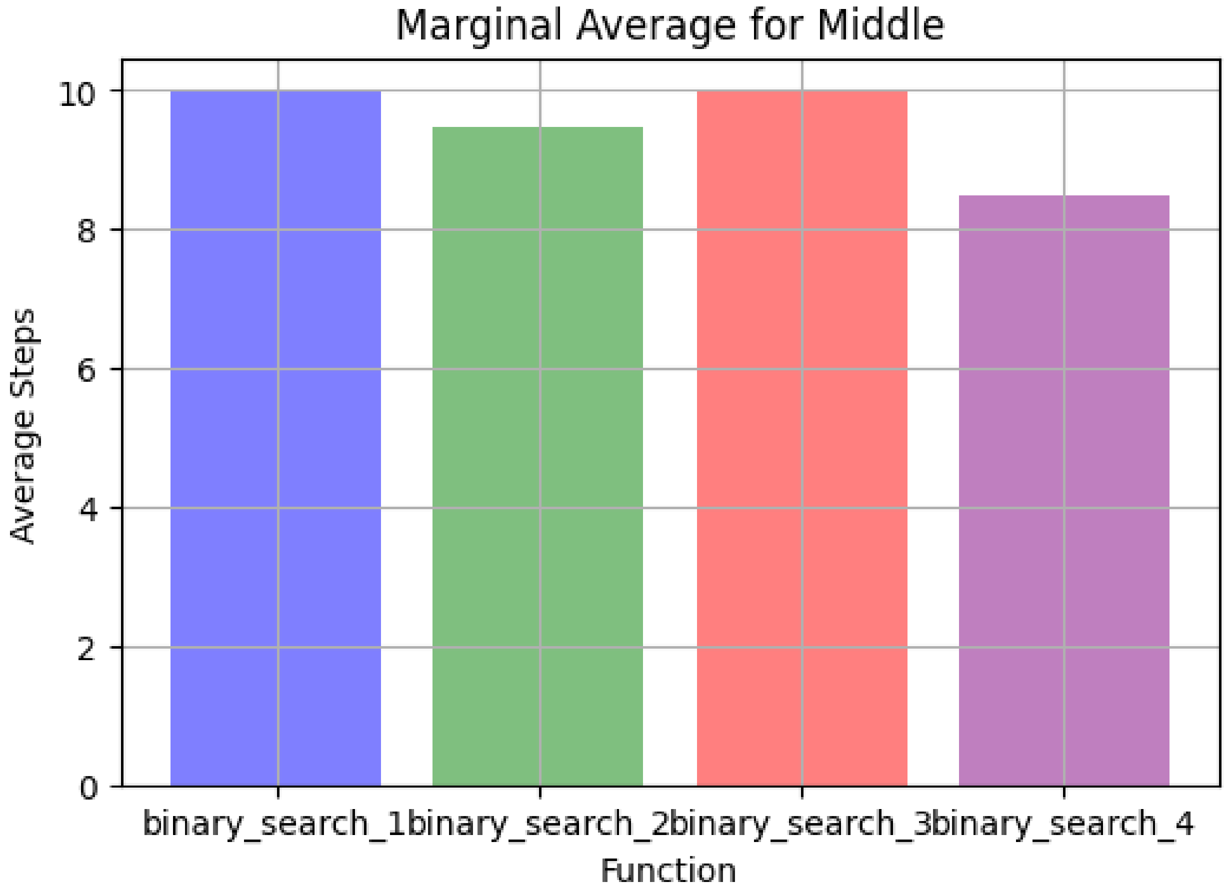

Even List - Middle
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function


Even List - End
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function

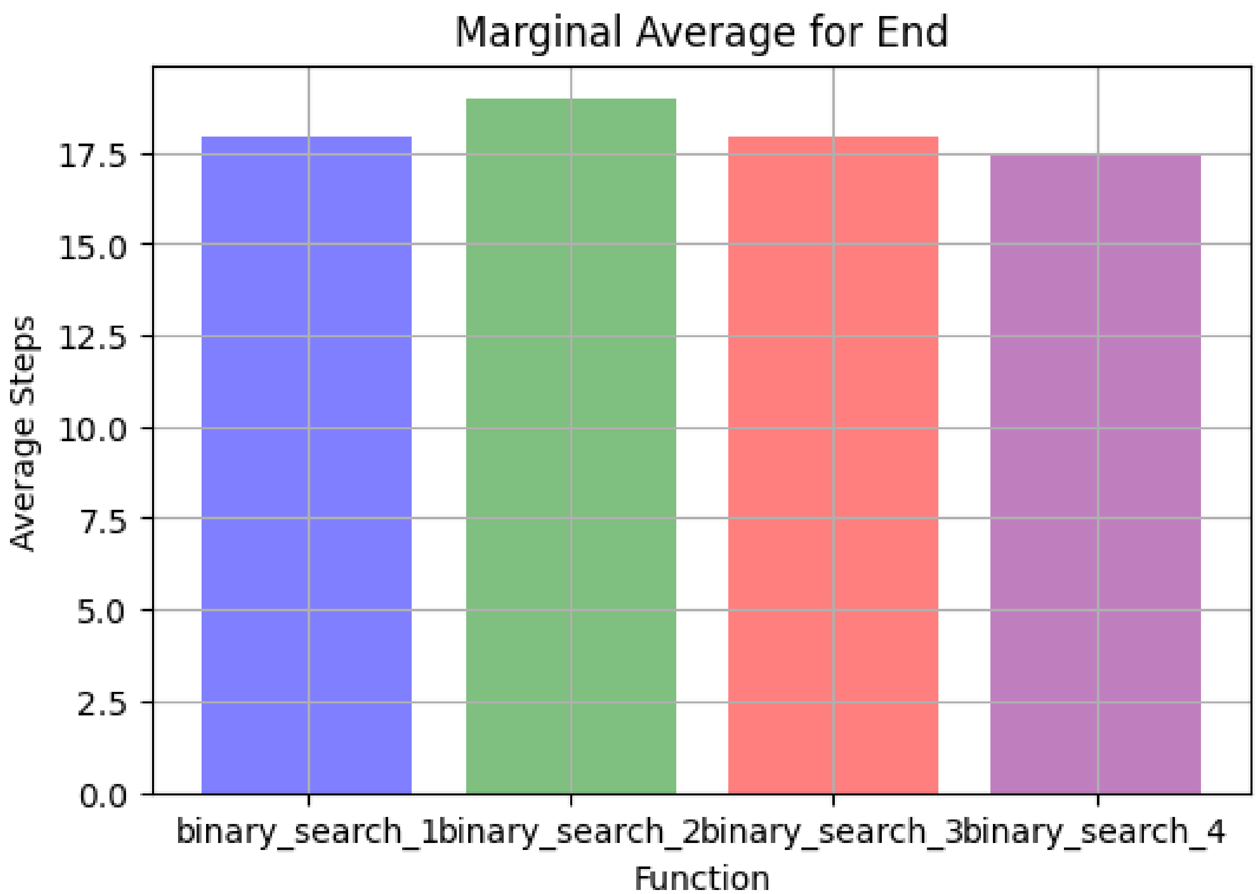

Marginal Average for Even
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function


Marginal Average for Odd
Average Steps
0
2
4
6
8
10
12
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function

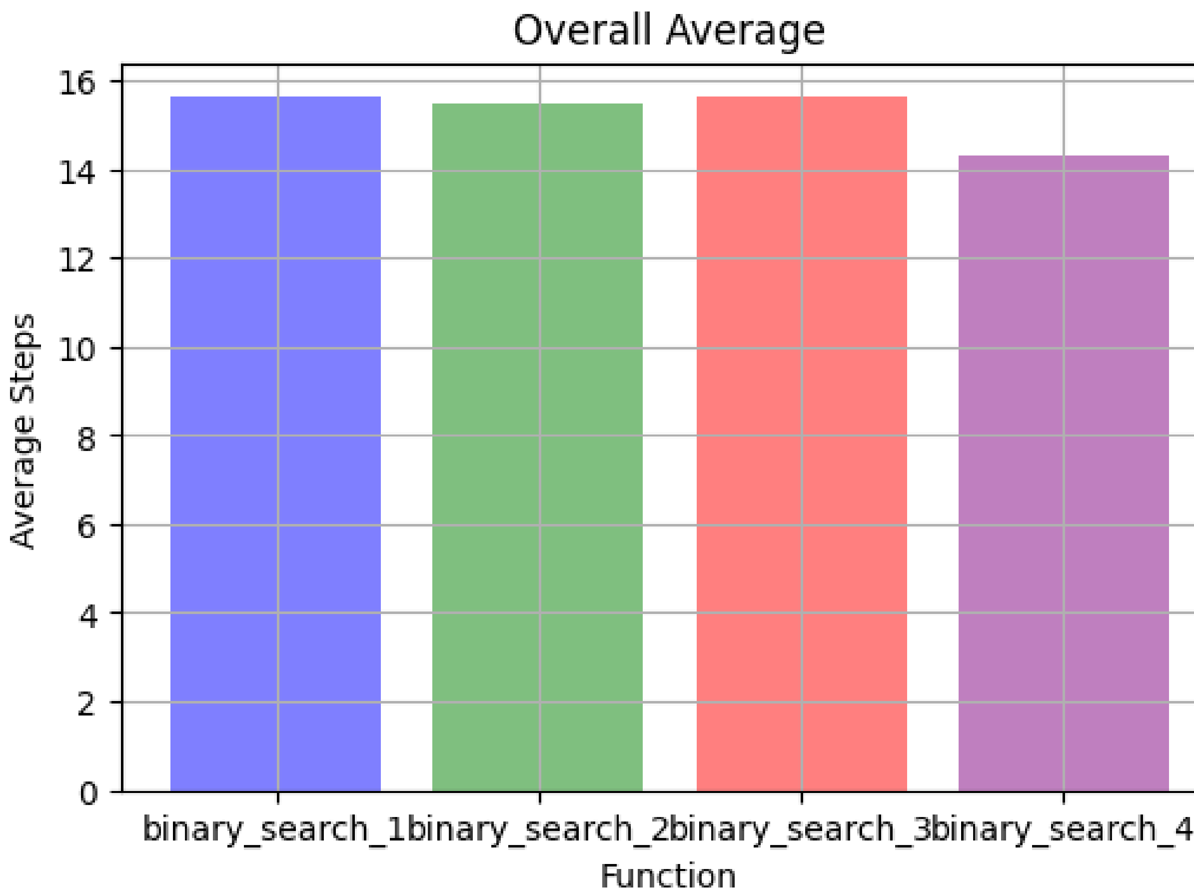

Marginal Average for Beginning
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function



Marginal Average for Middle
Average Steps
0
2
4
6
8
10
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function



Marginal Average for End
Average Steps
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
binary_search_1
binary_search_2
binary_search_3
binary_search_4
Function



Result Discussion:

Experiment overview

1. Goal: The goal was Compare the recursive binary search implementation (binary_search_4) with the non-recursive implementations under various conditions.
2. Test data generation:
    - Use the same data generation process as in previous experiments.
    - Generate random odd and even-numbered lists with varying lengths.
3. Trials and measurements:
    - ran multiple trials for variablity
    - recorded execution time, if the item was found, correctness and step count
    - tested at different positions such as beginning, middle and end of the list.
4. Plots:
    - created plots which made comparision between the searches much easier and visually appealing.
    - Marginal average for even graph corresponds to the case where we are searching for a random item in an even list
    - Marginal average for odd graph corresponds to the case where we are searching for a random item in an odd list

- Marginal average for beginning, middle and end graphs respectively correspond to the case where we are searching for the beginning, middle and end items in a randomly generated list.
- Overall average graph corresponds to the case where we are searching for a random item from a random list.

5. Modifications:
   - The recursive function directly searches the relevant portion of the list, reducing the number of comparisons.

Results and their Interpretations

1. From the graphs, we observe that the middle item is found at the same rate of time in all searches. This is because in all the algorithms, the first condition is if the middle item is the target.
2. binary_search_4 takes the least time to search for both beginning and end items of the list due the lesser number of iterations that take place.
3. overall, we see that binary_search_4 takes lesser time to search for a random item in a random list compared to all the other given algorithms
4. the recursive implementation has a much cleaner and readable code and prevents repition of code.

PART D

Now that you are comfortable in designing experiments, in this section, use the implementations of **Heap**, **Merge**, and **Quick** sort discussed in class and run suitable experiments to compare the runtimes of these three algorithms.

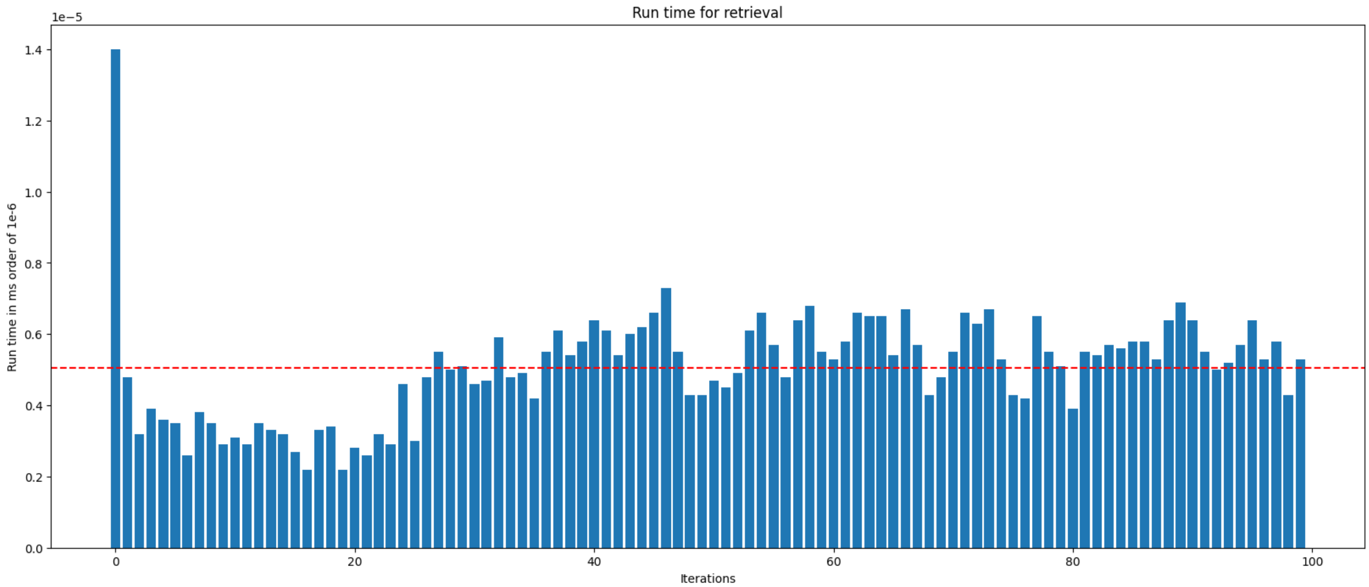
Hint: it should become clear where Quick sort gets its name.

In [10]:
```
import math
class Heap:

    def __init__(self, data):
        self.items = data
        self.length = len(data)
        self.build_heap()

    def find_left_index(self,index):
        return 2 * (index + 1) - 1
```

```python
    def find_right_index(self,index):
        return 2 * (index + 1)

    def heapify(self,index):
        largest_known_index = index
        left_index = self.find_left_index(index)
        right_index = self.find_right_index(index)

        # condition:  item at left index is greater than item at current index,
        # and left index is less than length
        if left_index < self.length and self.items[left_index] > self.items[ind
            largest_known_index = left_index
        #condition: item at right index is greater than item at largest_known i
        # and righ index is less than length
        if right_index < self.length and self.items[right_index] > self.items[l
            largest_known_index = right_index

        if largest_known_index!=index:
            self.items[index],self.items[largest_known_index]=self.items[larges
            self.heapify(largest_known_index)


    # running heapify - top down
    def build_heap(self,):
        for i in range(self.length // 2 - 1, -1, -1):
            self.heapify(i)

    # to print if pretty print does not work
    def __str__(self):
        height = math.ceil(math.log(self.length + 1, 2))
        whitespace = 2 ** height
        to_print = ""
        for i in range(height):
            for j in range(2 ** i - 1, min(2 ** (i + 1) - 1, self.length)):
                to_print += " " * whitespace
                to_print += str(self.items[j]) + " "
            to_print += "\n"
            whitespace = whitespace // 2
        print(to_print)
```

In [11]:

```python
runs = 100
run_times = []
for _ in range(runs):
    list = create_custom_list(10, 2000)
    start = timeit.default_timer()
    Heap(list)
    stop = timeit.default_timer()
    run_times.append(stop-start)

draw_plot(run_times)
```

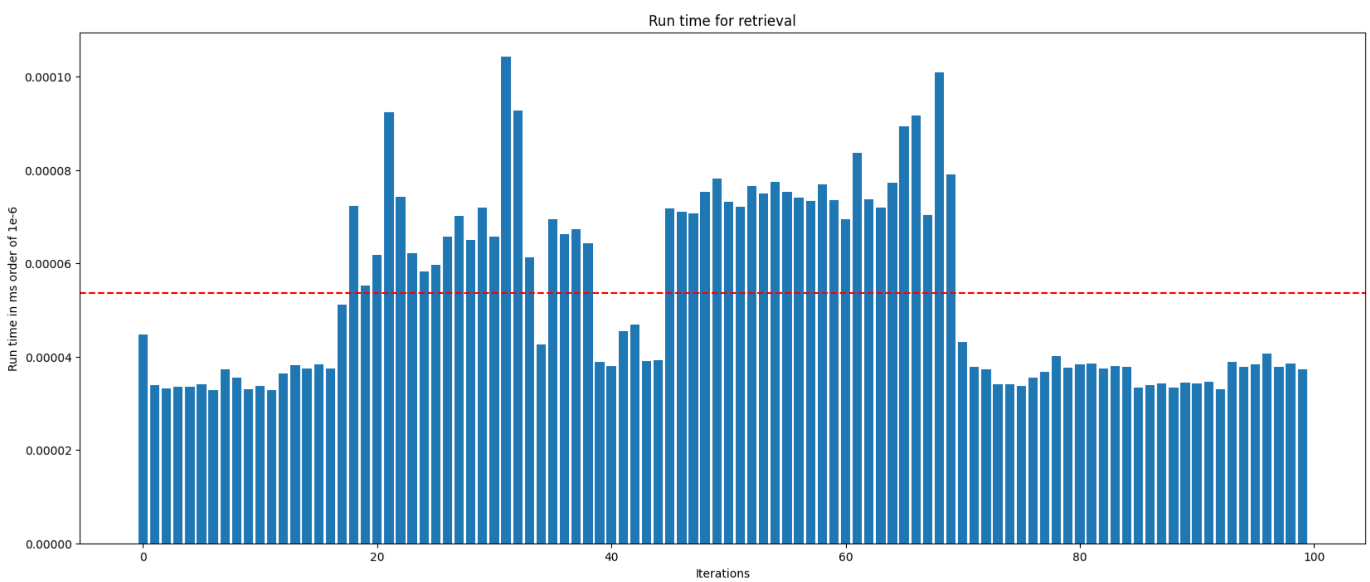


In [14]:
```
class Merge:
    def __init__(self, data):
        self.items = data
        self.length = len(data)
        self.merge_sort()

    def merge_sort(self):
        if self.length > 1:
            mid = self.length // 2
            L = self.items[:mid]
            R = self.items[mid:]

            # Recursive calls to sort left and right sublists
            merge_instance_left = Merge(L)
            merge_instance_right = Merge(R)

            merge_instance_left.merge_sort()
            merge_instance_right.merge_sort()


            i = j = k = 0
            #sorted_list = []

            # Merge the sorted sublists
            while i < len(L) and j < len(R):
                if L[i] <= R[j]:
                    self.items[k] = L[i]
                    i += 1
                else:
                    self.items[k] = R[j]
                    j += 1
                k += 1

            # Checking if any element was left in the sublists
```

```
            while i < len(L):
                self.items[k] = L[i]
                i += 1
                k += 1

            while j < len(R):
                self.items[k] = R[j]
                j += 1
                k += 1
```

In [16]:

```
runs = 100
run_times = []
for _ in range(runs):
    list = create_custom_list(10, 2000)
    start = timeit.default_timer()
    Merge(list)
    stop = timeit.default_timer()
    run_times.append(stop-start)

draw_plot(run_times)
```



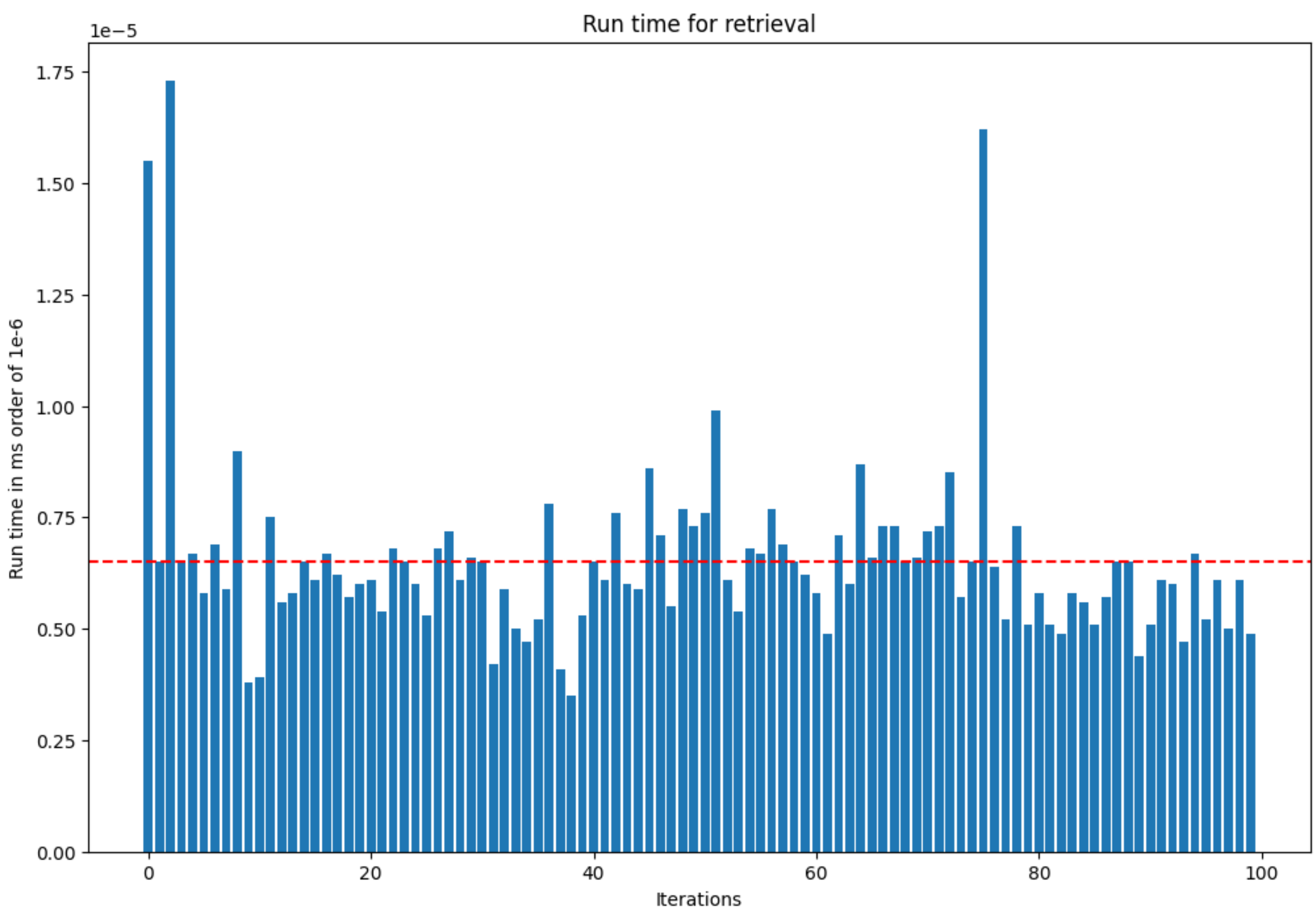
The first two times the average time was 5 ms. Now it is approximately 0.00005.

In [2]:

```
class QuickSort:
    def partition(self, 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 quick_sort(self, array, low, high):
        if low < high:
            pi = self.partition(array, low, high)
            self.quick_sort(array, low, pi - 1)
            self.quick_sort(array, pi + 1, high)
```

In [10]:
```
runs = 100
run_times = []
quick_sort_instance = QuickSort()
for _ in range(runs):
    list = create_custom_list(10, 2000)
    start = timeit.default_timer()
    quick_sort_instance.quick_sort(list, 0, len(list) - 1)
    stop = timeit.default_timer()
    run_times.append(stop-start)

draw_plot(run_times)
```



In this section, provide a detailed outline of:

- The experiments you ran, length values of the list you chose, number of runs, etc.
- The plots showing the run times corresponding to each algorithm.

- A brief discussion and conclusion regarding the results. A few sentences are fine here.

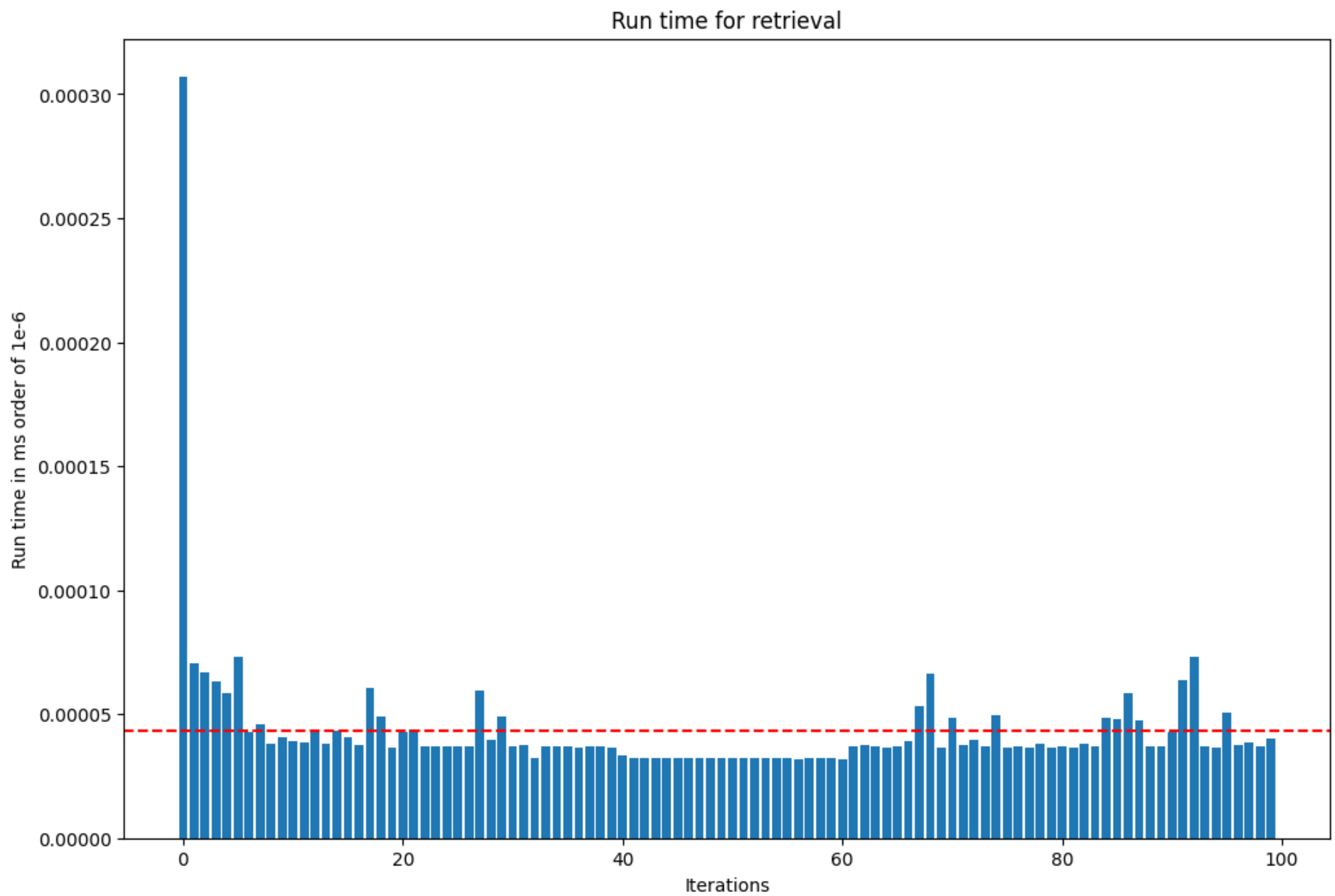
Reflection: I ran the experiements 100 times. The length of my arrays is 10 with the maximum element being 2000. Looking at the graphs its obvious that quicksort is the fastest algorithm, on average quicksort is two times faster than heap sort. Merge sort is the slowest algorithm.

PART E

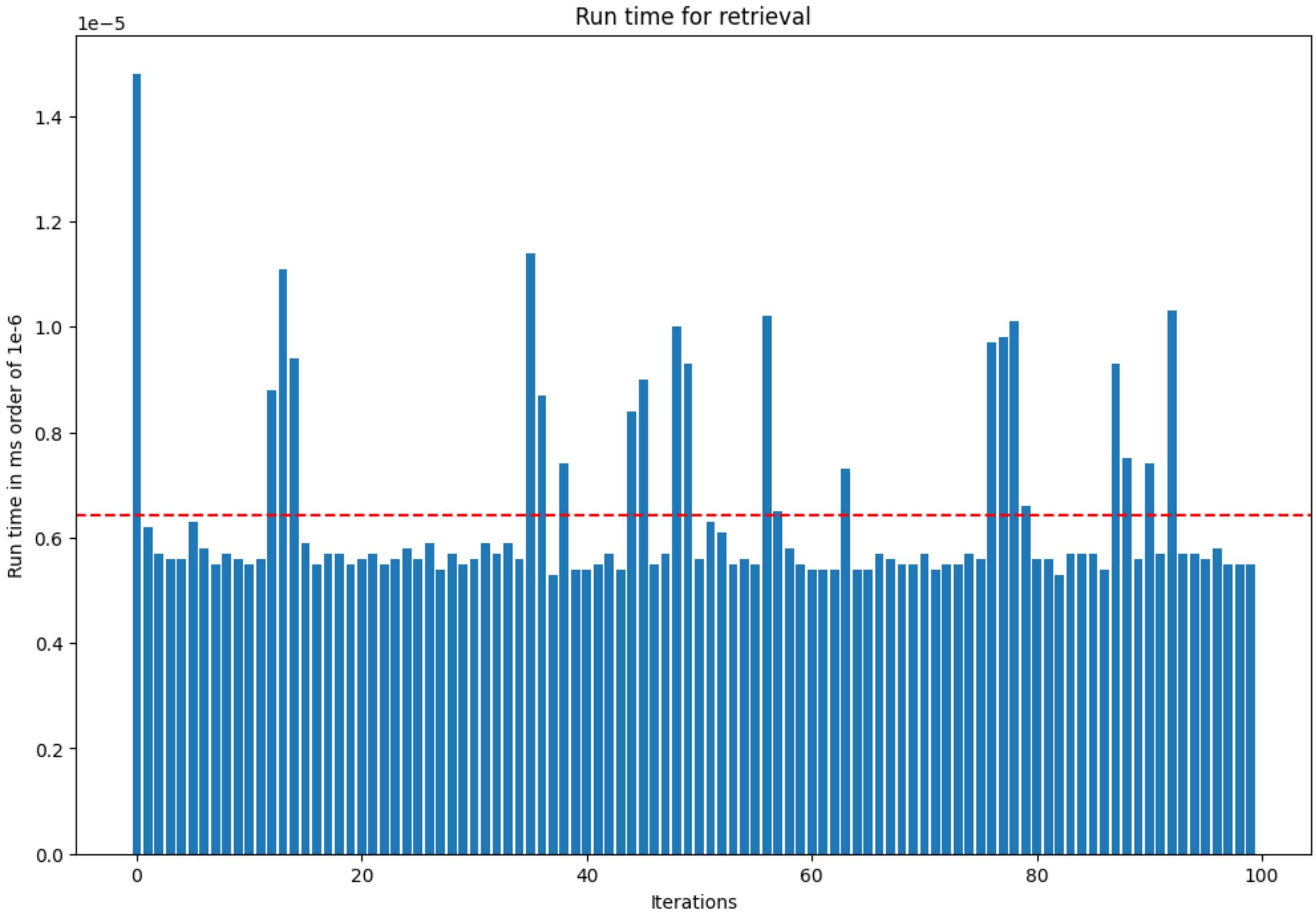
E1. In previous experiments you also saw that not all algorithms are suitable for all scenarios. For instance, Merge Sort is better than Quick sort for certain situations. In this section, design a experiment to compare the scenarios where Merge Sort is better/worse than Quick Sort. You can use the traditional version of Merge Sort or use improved version (maybe via recursion) to compare this performance.

In [14]:

```
# your implementation and experiments go here.
runs = 100
run_times = []
for run in range(runs):
    list = sorted(create_custom_list(10,2000))
    start = timeit.default_timer()
    Merge(list)
    stop = timeit.default_timer()
    run_times.append(stop-start)

draw_plot(run_times)
```



In [15]:
```
runs = 100
run_times = []
quick_sort_instance = QuickSort()
for run in range(runs):
    list = sorted(create_custom_list(10,2000))
    start = timeit.default_timer()
    quick_sort_instance.quick_sort(list, 0, len(list) - 1)
    stop = timeit.default_timer()
    run_times.append(stop-start)

draw_plot(run_times)
```



In this section, provide a detailed outline of:

- The experiments you ran, length values of the list you chose, number of runs, etc.
- The plots showing the run times corresponding to each algorithm.
- A brief discussion and conclusion regarding the results. A few sentences are fine here.

Reflection: Since we were asked to design an experiment where merge sort is more efficient than quicksort, we decided to use sorted lists as the input. As seen from the graphs we acheived the worst case complexity (O(n^2)) of quicksort which results in it performing slower than merge sort.

E2. Recall that on the first day of class I asked which two algorithms have similar complexity - Merge Sort and Quick Sort under (O(nlogn)) are likely to perform similar under average cases. However, under worst case, the complexity of quick sort is much worse (O(n^2). Design an experiment to show this behavior. Plot this behavior on a bar/line chart.

Next, count the number of "swaps" after which Quick sort starts behaving comparable to Merge sort.

HINT: This will be a threshold at which the quick sort algorithm picks up again.

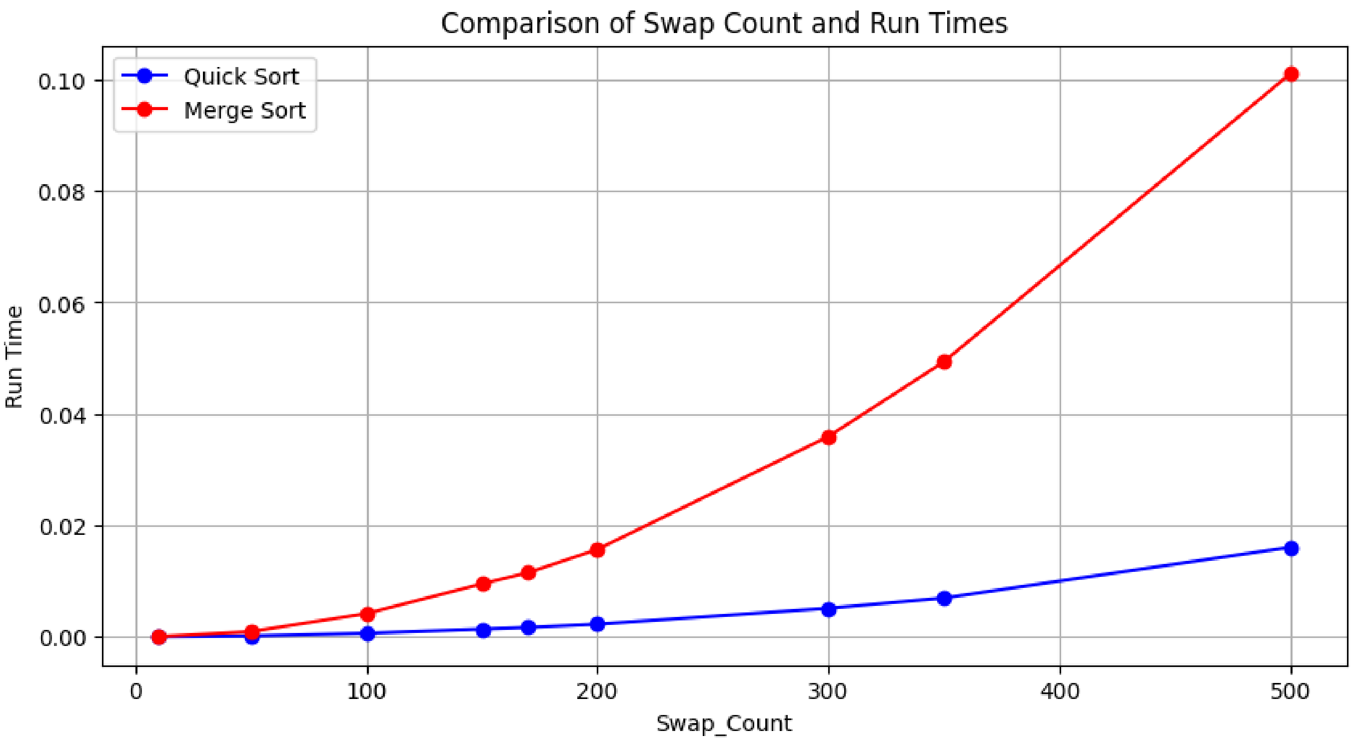
In [15]:
```
## your implementation and code goes here
class QuickSort1:
    def __init__(self):
        self.swap_count = 0

    def partition(self, array, low, high):
        pivot = max(array)
        i = low - 1

        for j in range(low, high):
            if array[j] <= pivot:
                i = i + 1
                array[i], array[j] = array[j], array[i]
                self.swap_count += 1  # Increment swap count

        array[i + 1], array[high] = array[high], array[i + 1]
        self.swap_count += 1  # Increment swap count for the final swap
        return i + 1

    def quick_sort(self, array, low, high):
        if low < high:
            pi = self.partition(array, low, high)
            self.quick_sort(array, low, pi - 1)
            self.quick_sort(array, pi + 1, high)
```

In [37]:
```
def avg(data):
    return sum(data)/len(data)
```

In [90]:
```
#Worst Case time complexity for Quick Sort is when the elements are already sor
runs = 100
qs = QuickSort1()
list_size = [10, 50, 100, 150, 170, 200, 300, 350, 500]

quick_dic = {}
merge_dic = {}

for size in list_size:
    swap_count=[]
    run_times_q = []
    run_times_m = []
    for run in range(runs):
        list_main = sorted(create_custom_list(size, 2000))
        list_q = list_main.copy()
        list_m = list_main.copy()
```

```
        start = timeit.default_timer()
        qs.quick_sort(list_q, 0, size - 1)
        stop = timeit.default_timer()
        duration=stop-start
        run_times_q.append(duration)
        swap_count.append(qs.swap_count)


        start = timeit.default_timer()
        Merge(list_m)
        stop = timeit.default_timer()
        duration=stop-start
        run_times_m.append(stop-start)

    avg_swap = round(avg(swap_count))
    avg_rtq = avg(run_times_q)
    avg_rtm = avg(run_times_m)

    quick_dic[size]=(avg_rtq, avg_swap)
    merge_dic[size]=avg_rtm

# Print header
print("Size\tQuickSort Time\t\tMergeSort Time\t\tSwap Count")

# Iterate over sizes in quick_data (assuming sizes are same in both dictionarie
for size in quick_dic:
    quick_time, quick_swaps = quick_dic[size]
    merge_time = merge_dic[size]
    print(f"{size}\t{quick_time}\t{merge_time}\t{quick_swaps}")


#Extracting values from dictionaries for Quick Sort
quick_durations = [value[0] for value in quick_dic.values()]

# Extracting values from dictionaries for Merge Sort
merge_durations = list(merge_dic.values())

# Plotting
plt.figure(figsize=(10, 5))

plt.plot(list_size, quick_durations, marker='o', color='b', label='Quick Sort')
plt.plot(list_size, merge_durations, marker='o', color='r', label='Merge Sort')

plt.title('Comparison of Swap Count and Run Times')
plt.xlabel('Swap_Count')
plt.ylabel('Run Time')
plt.legend()
plt.grid(True)
plt.show()
```

Size	QuickSort Time	MergeSort Time	Swap Count
10	9.256000339519233e-06	3.6875999794574457e-05	2727
50	0.00015218299929983914	0.0009216770001512487	69737
100	0.0006279859990172554	0.004156673999386839	387774
150	0.0013665239994588773	0.009494911000365392	1209562
170	0.00166761100011675	0.011500511999911396	2504067
200	0.0022517869991133923	0.015673143999738386	4238500
300	0.00510697499944214	0.03593580099972314	7513424
350	0.006951580000895774	0.0493455929990l593	12850212
500	0.016063574999716366	0.10106564699977753	22215774



In this section, provide a detailed outline of:

- The experiments you ran and the rationale behind your worst case scenario.
- The plots showing the run times.

Further explain how you computed the swaps and verify that you calculation is correct, by applying it on a diifferent list under same experimental conditions.

Reflection:

The worst-case time complexity for Quick Sort arises when the pivot selection leads to significantly unbalanced partitions, resulting in a time complexity of O(n^2). This scenario occurs when the algorithm consistently selects either the smallest or largest element as the pivot, causing one partition to contain n−1 elements while the other partition remains empty. In our code, we deliberately chose the worst-case scenario to be an already sorted array. This choice mimics the behavior where Quick Sort selects the maximum element as the pivot, exacerbating the unbalanced partition issue.

Consequently, the algorithm's performance degrades to O(n^2), demonstrating the worst-case scenario's impact on Quick Sort's efficiency.

For testing, we chose the list sizes to be [10, 50, 100, 150, 170, 200, 300, 350, 500]. For each size, we ran the quick sort and merge sort 100 times each with a different lists. We calculated the average run time and swap count and added it to a dictionary to help us with comparison and plotting. From the plotted graph we see that at the swap count of 2727, merge sort and quick sort acts in a similar way with similar runtimes. We can see that as the size of the list increases the time it takes to run quick sort and merge sort varies significantly with quick sort being faster.

PART F

Traditionally, Insertion Sort is worst than Heap Sort and Merge Sort. Now that you are a master at critical evaluation of sorting and searching algorithms, design an experiment to show that this may not be universally true. That is, there maybe scenarios where insertion sort is better than merge and heap sort.

HINT: Think about the Best Case of insertion sort.

Again, provide:

- An explicit outline of the experiments you ran. That is, list length values, how many “runs”, etc.
- A graph of list length vs time displaying the appropriate three curves showing. List lengths should be small here.
- A brief discussion and conclusion regarding the results. A few sentences are fine here.
- Reflect on why these are experiments are important.

HINT: Can you create some sort of "hybrid" sort that would be better?

In [53]:
```
## your implementation and code goes here
## Merge sort
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
```

```python
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort each half
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Append the remaining elements, if any
    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

In [54]:

```python
## Heap sort
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    # Check if left child exists and is greater than the root
    if left_child < n and arr[left_child] > arr[largest]:
        largest = left_child

    # Check if right child exists and is greater than the largest so far
    if right_child < n and arr[right_child] > arr[largest]:
        largest = right_child

    # Swap the root if needed and recursively heapify the affected sub-tree
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
```

```
    # Build a max-heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Swap the root (max element) with the
        heapify(arr, i, 0)  # Heapify the reduced heap
```

In [55]:
```
## Hybrid sort
def hybrid_sort(data):

    def is_mostly_sorted(lst, threshold_factor=0.1):
        inversions = 0
        length = len(lst)

        # Define a threshold for inversions based on the length of the list
        threshold = length * threshold_factor

        for i in range(length - 1):
            if lst[i] > lst[i + 1]:
                inversions += 1
        threshold = min(100, length - 1)
        # Early exit if inversions exceed the threshold
        if inversions > threshold:
            return False
        return True


    # if list is nearly sorted then insertion sort is faster
    if is_mostly_sorted(data):
        return InsertionSort.insertion_sort(data)
    else:
        return merge_sort(data)

# Function to generate nearly sorted lists
def generate_nearly_sorted_list(length, max_displacements):
    sorted_list = list(range(length))

    # Limit the number of displacements based on the list length
    num_displacements = min(max_displacements, length - 1)

    for _ in range(num_displacements):
        i, j = random.sample(range(length), 2)
        sorted_list[i], sorted_list[j] = sorted_list[j], sorted_list[i]
    return sorted_list
```

In [56]:
```
# Assuming the sorting functions and generate_nearly_sorted_list function are d

def run_experiments(list_lengths):
```

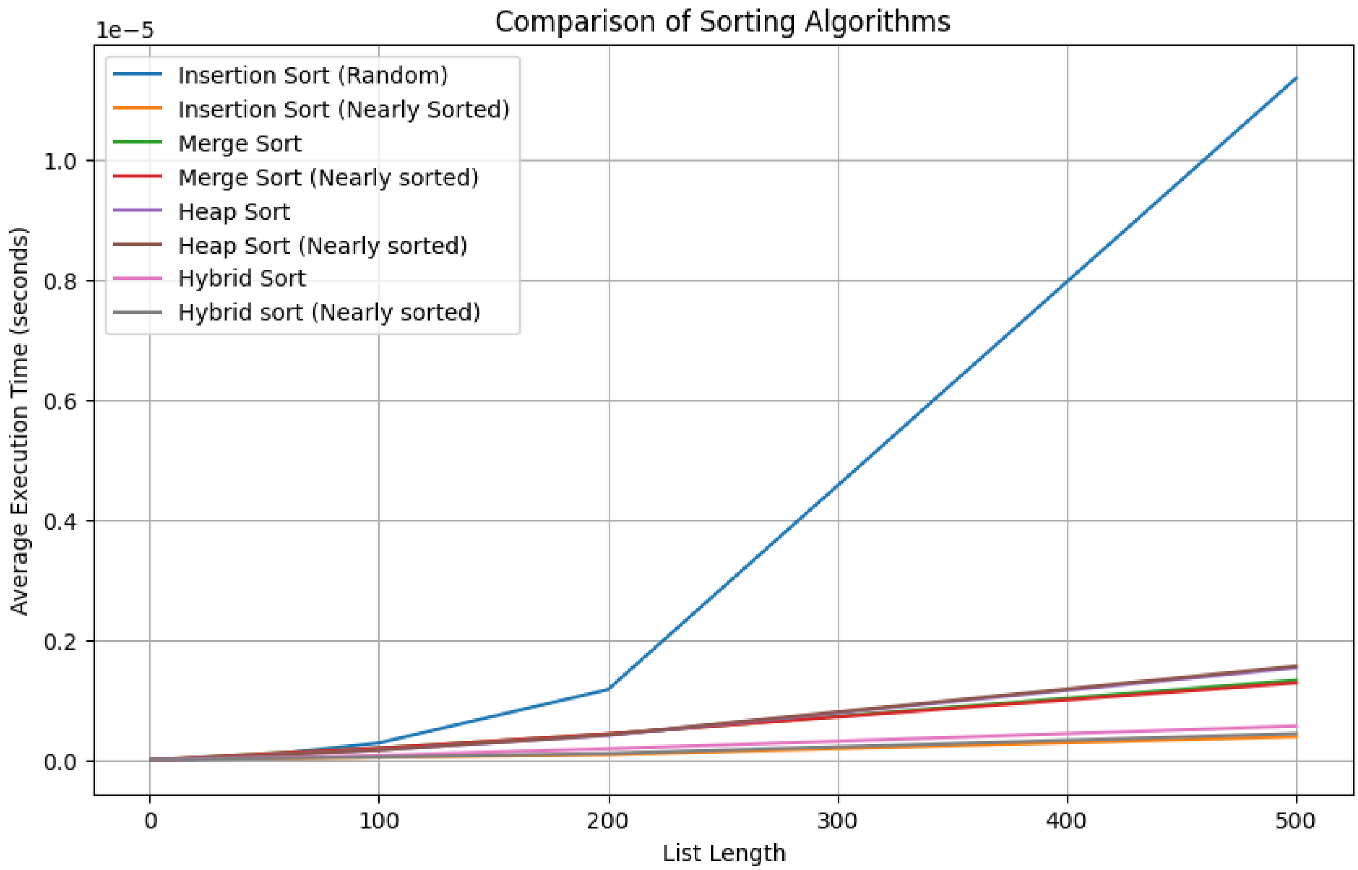
```python
    num_runs = 1000  # Number of runs for each experiment

    results = {
        'Insertion Sort (Random)': [],
        'Insertion Sort (Nearly Sorted)': [],
        'Merge Sort': [],
        'Merge Sort (Nearly sorted)': [],
        'Heap Sort': [],
        'Heap Sort (Nearly sorted)': [],
        'Hybrid Sort': [],
        'Hybrid sort (Nearly sorted)': []
    }

    for length in list_lengths:
        print(length)
        # Initialize variables to accumulate total time for each sorting algori
        insertion_random_total_time = 0
        insertion_nearly_sorted_total_time = 0
        merge_total_time = 0
        merge_nearly_sorted_total_time = 0
        heap_total_time = 0
        heap_nearly_sorted_total_time = 0
        hybrid_total_time = 0
        hybrid_nearly_sorted_total_time = 0

        for _ in range(num_runs):
            p = 0.9
            # Generate new random and nearly sorted lists for each run
            random_list = [random.randint(1, 1000) for _ in range(length)]
            nearly_sorted_list = generate_nearly_sorted_list(length, 5)  # Limi
            final_list = random_list
            if random.random() < p:
                final_list = nearly_sorted_list

            # Experiment with Insertion Sort on random list
            insertion_random_total_time += timeit.timeit(lambda: InsertionSort.

            # Experiment with Insertion Sort on nearly sorted list
            insertion_nearly_sorted_total_time += timeit.timeit(lambda: Inserti

            # Experiment with Merge Sort
            merge_total_time += timeit.timeit(lambda: merge_sort(final_list.cop

            # Experiment with Merge Sort nearly sorted
            merge_nearly_sorted_total_time += timeit.timeit(lambda: merge_sort(

            # Experiment with Heap Sort
            heap_total_time += timeit.timeit(lambda: heap_sort(final_list.copy(

            # Experiment with Heap Sort nearly sorted
```

```
            heap_nearly_sorted_total_time += timeit.timeit(lambda: heap_sort(ne

            # Experiment with Hybrid Sort
            hybrid_total_time += timeit.timeit(lambda: hybrid_sort(final_list.c

            # Experiment with Hybrid Sort nearly sorted
            hybrid_nearly_sorted_total_time += timeit.timeit(lambda: hybrid_sor

        # Calculate average times and add to results
        results['Insertion Sort (Random)'].append(insertion_random_total_time /
        results['Insertion Sort (Nearly Sorted)'].append(insertion_nearly_sorte
        results['Merge Sort'].append(merge_total_time / num_runs)
        results['Merge Sort (Nearly sorted)'].append(merge_nearly_sorted_total_
        results['Heap Sort'].append(heap_total_time / num_runs)
        results['Heap Sort (Nearly sorted)'].append(heap_nearly_sorted_total_ti
        results['Hybrid Sort'].append(hybrid_total_time / num_runs)
        results['Hybrid sort (Nearly sorted)'].append(hybrid_nearly_sorted_tota

    return results
```

In [58]:

```
# Plotting function
def plot_results(list_lengths, results):
    num_runs = 1000
    plt.figure(figsize=(10, 6))

    for algorithm, times in results.items():
        plt.plot(list_lengths, [time / num_runs for time in times], label=algor

    plt.xlabel('List Length')
    plt.ylabel('Average Execution Time (seconds)')
    plt.title('Comparison of Sorting Algorithms')
    plt.legend()
    plt.grid(True)
    plt.show()

# Run experiments and plot results
list_lengths = [1, 5, 10, 20, 50, 100, 200, 500]
results = run_experiments(list_lengths)
plot_results(list_lengths, results)
```

```
1
5
10
20
50
100
200
500
```



Reflection:

Experiment overview

1. Goal: The goal was to evaluate the performance of Insertion Sort, Merge Sort, Heap Sort, and Hybrid Sort (combination of Insertion Sort and Merge Sort) under different scenarios.
2. list lengths:
   - The experiment is conducted for varying list lengths: [1, 5, 10, 20, 50, 100, 200, 500].
3. Runs:
   - Each scenario is tested across 1000 runs to obtain reliable average execution times.
4. Scenarios:
   - created plots which made comparision between the searches much easier and visually appealing.
   - Random list for Insertion Sort.
   - Nearly sorted list for Insertion Sort.
   - Random list for Merge Sort.
   - Nearly sorted list for Merge Sort.
   - Random list for Heap Sort.

- Nearly sorted list for Heap Sort.
- Random list for Hybrid Sort.
- Nearly sorted list for Hybrid Sort.

5. Modifications:
    - The Hybrid Sort function is modified to switch to Insertion Sort for nearly sorted lists.

Results and their Interpretations

1. The plot displays the average execution time for each sorting algorithm under different scenarios and list lengths.
2. Insertion sort is slower on random lists but is significantly faster on the nearly sorted ones. Merge Sort, while generally fast, loses efficiency on nearly sorted lists, where Insertion Sort outperforms it.
3. The Hybrid Sort optimizes this by dynamically choosing between Insertion Sort for nearly sorted input and Merge Sort for random lists, ensuring better overall performance.
4. The experiment highlights situations where typically deemed slower algorithms, like Insertion Sort, can actually outshine others.
5. The Hybrid Sort's ability to switch between strategies emphasizes the significance of choosing the right sorting approach based on the unique characteristics of the input data.

Team Contributions: In below section describe in detail how you distributed the workload and contributions of each member in the task.

Sana Ashraf: code for Parts A, D, and E and reflection for part E.2 Prakhar Saxena: Reflections for parts A, D and E.1 Sriya Dhanvi Mokhasunavisu: code and reflections for parts B, C and F

Note: Prakhar joined the group later in the project and, unfortunately, didn't have the opportunity to contribute as much as he would have liked. He is prepared to do more work in the next project to compensate for this, and the professor has kindly agreed to this arrangement.