CS 3851 Algorithms - Lab 3: Benchmarking Merge Sort

Stuart Harley

Introduction

In this lab we are implementing the sorting algorithm merge sort. We also use our implementation of insertion sort from the last lab for comparison. We are then going to benchmark the algorithms under the best (already sorted), worst (reverse sorted), and average (random permutation) case scenarios. Then we will plot the benchmarking times in order to further analyze and compare the run times of the two algorithms. We show that merge sort O(nlogn) is slower than insertion sort O(n) for the best case scenario, but considerably slower than insertion sort O(n^2) for the average and worst case scenarios.

Importing Libraries

In [1]:
```
import time
import numpy as np
import random
import matplotlib.pyplot as plt

from sorting import *
from test_sorting import *
```

Testing Implementation of Merge Sort

In [2]:
```
!python test_sorting.py
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Creating Lists to use for Benchmarking

For benchmarking, we will evaluate 3 cases: list is already sorted, list is randomly permuted, and list is already sorted in reverse order.

Creating lists of 100, 1000, 10000, and 100000 numbers. Creating the lists from random numbers. Then shuffling them to generate a random permutation of the list or sorting them in regular/reverse order to create the desired cases.

_p, _o, _r refers to the list being randomly permutated, sorted in order, and sorted in reverse order respectively.

In [3]:
```
l100 = [random.random() for i in range(100)]
random.shuffle(l100) # shuffling the list to generate a random permutation
l100_p = l100[:]
l100.sort() # sorting the list in order
l100_o = l100[:]
l100.sort(reverse=True) # sorting the list in reverse order
l100_r = l100[:]
```

In [4]:
```
l1000 = [random.random() for i in range(1000)]
random.shuffle(l1000) # shuffling the list to generate a random permutation
l1000_p = l1000[:]
l1000.sort() # sorting the list in order
l1000_o = l1000[:]
l1000.sort(reverse=True) # sorting the list in reverse order
l1000_r = l1000[:]
```

In [5]:
```
l10000 = [random.random() for i in range(10000)]
random.shuffle(l10000) # shuffling the list to generate a random permutation
l10000_p = l10000[:]
l10000.sort() # sorting the list in order
l10000_o = l10000[:]
l10000.sort(reverse=True) # sorting the list in reverse order
l10000_r = l10000[:]
```

In [6]:
```
l100000 = [random.random() for i in range(100000)]
random.shuffle(l100000) # shuffling the list to generate a random permutation
l100000_p = l100000[:]
l100000.sort() # sorting the list in order
l100000_o = l100000[:]
l100000.sort(reverse=True) # sorting the list in reverse order
l100000_r = l100000[:]
```

Benchmarking the Merge Sort Algorithm

We are running each benchmark 10 times (trials).

In [7]:
```
n_trials = 10
run_times_best_m = []
run_times_average_m = []
run_times_worst_m = []
```

In [8]:
```
# 100 elements
times = []
for i in range(n_trials):
    lst2 = l100_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_m.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 0.0008743892889469862 sec
100 elements: Sorted in Order
Average Time: 0.0010021373629570008 sec
100 elements: Sorted in Reverse
Average Time: 0.0009507143869996071 sec
```

In [9]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('1000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_m.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.01145945661701262 sec
1000 elements: Sorted in Order
Average Time: 0.011629405664280057 sec
1000 elements: Sorted in Reverse
Average Time: 0.011316057154908776 sec
```

In [10]:
```
# 10000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('10000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_m.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 0.13955251453444362 sec
10000 elements: Sorted in Order
Average Time: 0.13793887230567634 sec
10000 elements: Sorted in Reverse
Average Time: 0.1434717732015997 sec
```

In [11]:
```
# 100000 elements
times = []
for i in range(n_trials):
    lst2 = l100000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_m.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    merge_sort(lst2, 0, len(lst2) - 1)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_m.append(np.mean(times))
```

```
100000 elements: randomly permuted
Average Time: 1.6516511477995663 sec
100000 elements: Sorted in Order
Average Time: 1.6215206089895218 sec
100000 elements: Sorted in Reverse
Average Time: 1.6171391546726226 sec
```

Benchmarking the Insertion Sort Algorithm

In [12]:
```
run_times_best_i = []
run_times_average_i = []
run_times_worst_i = []
```

In [13]:
```
# 100 elements
times = []
for i in range(n_trials):
    lst2 = l100_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 0.0008634826168417931 sec
100 elements: Sorted in Order
Average Time: 4.2644329369068146e-05 sec
100 elements: Sorted in Reverse
Average Time: 0.0018964204005897044 sec
```

In [14]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l1000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 0.10018951925449074 sec
100 elements: Sorted in Order
Average Time: 0.0005901136435568332 sec
100 elements: Sorted in Reverse
Average Time: 0.1936948787420988 sec
```

In [15]:
```
# 10000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l10000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
100 elements: randomly permuted
Average Time: 10.331166822928935 sec
100 elements: Sorted in Order
Average Time: 0.00527221942320466 sec
100 elements: Sorted in Reverse
Average Time: 20.68036577217281 sec
```

In [16]:
```
# 100000 elements
times = []
for i in range(n_trials):
    lst2 = l100000_p[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l100000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('100 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

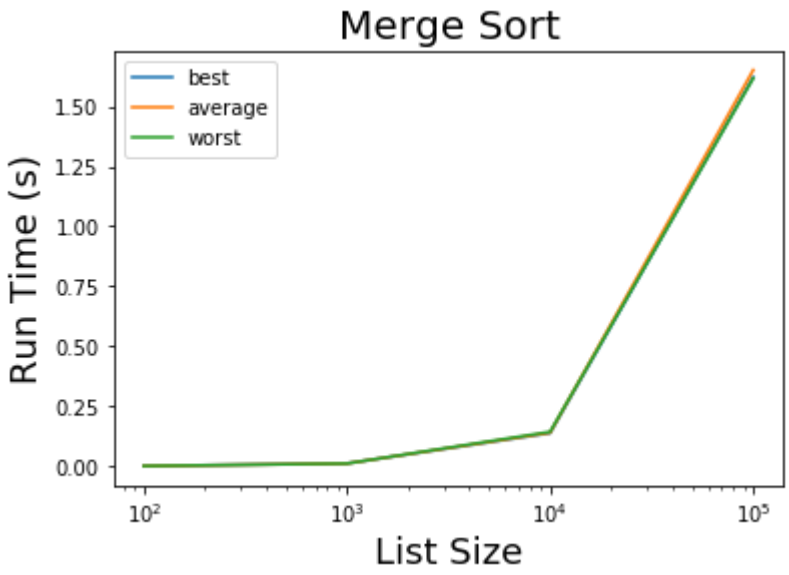
```
100 elements: randomly permuted
Average Time: 1109.823928119149 sec
100 elements: Sorted in Order
Average Time: 0.0568594029173255 sec
100 elements: Sorted in Reverse
Average Time: 2268.8596591409296 sec
```

Plotting the Benchmarked Run Times

In [17]:
```
list_sizes = [100, 1000, 10000, 100000]
```

Creating one plot for each of the two algorithms with all three cases on them.

In [18]:
```
plt.plot(list_sizes, run_times_best_m, label="best")
plt.plot(list_sizes, run_times_average_m, label="average")
plt.plot(list_sizes, run_times_worst_m, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Merge Sort", fontsize=20)
plt.legend()
plt.xscale('log')
```

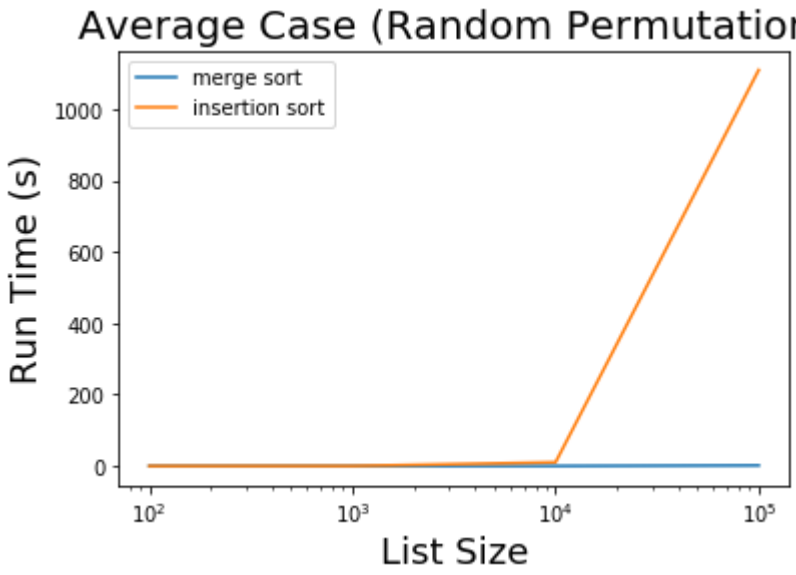


In [19]:
```
plt.plot(list_sizes, run_times_best_i, label="best")
plt.plot(list_sizes, run_times_average_i, label="average")
plt.plot(list_sizes, run_times_worst_i, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Insertion Sort", fontsize=20)
plt.legend()
plt.xscale('log')
```

Creating one plot for each case comparing merge sort and insertion sort.

In [20]:
```
plt.plot(list_sizes, run_times_best_m, label="merge sort")
plt.plot(list_sizes, run_times_best_i, label="insertion sort")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Best Case (Sorted Order)", fontsize=20)
plt.legend()
plt.xscale('log')
```

In [21]:
```
plt.plot(list_sizes, run_times_average_m, label="merge sort")
plt.plot(list_sizes, run_times_average_i, label="insertion sort")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Average Case (Random Permutation)", fontsize=20)
plt.legend()
plt.xscale('log')
```



In [22]:
```
plt.plot(list_sizes, run_times_worst_m, label="merge sort")
plt.plot(list_sizes, run_times_worst_i, label="insertion sort")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Worst Case (Reverse Order)", fontsize=20)
plt.legend()
plt.xscale('log')
```

Reflection Questions

1) The differences between the run times of the three cases for merge sort were negligible. All the cases ran quickly with very minimal differences. It seemed that the reverse order was the best, the sorted order was next best, and the random permutation was the worst. But again, the differences were negligible.

2) The sorting of the sorted and reverse sorted lists would involve a lot of comparisons with infinity with the way we implemented the merge function. Whereas the random permuation would have less because the numbers would be mixed up more. It is possible that the computer can compare a number and infinity quicker than two numbers that are closer together which could explain why the random permutation takes ever so slightly longer.

3) Merge sort is O(nlogn) for all cases. This is consistent with our benchmarking which showed no significant differences between the different cases.

4) There were differences between every case of insertion sort vs merge sort. Insertion sort ran faster for the best case (already sorted), but performed much worse in the worst case (reverse sorted) and the average case (random permutation).

5) Insertion sort is O(n) for the best case scenario which is faster than merge sort's O(nlogn). In the best case insertion sort only has to make 1 comparison for each number in the list (n total comparisons), but merge sort has to make nlogn comparisons, which is slighlty slower. However, insertion sort is O(n^2) for the average case and worst case scenarios which is much slower than merge sort's O(nlogn), which the differences growing exponentially as the size of the list grows. These time complexities were consistent with my benchmarking.

Conclusion

Merge sort is O(nlogn) for all cases. It only slighlty underperforms insertion sort O(n) in the best case scenario. However, the best case scenario is that the list is already sorted. If you knew that your list was already sorted than you probably wouldn't go about sorting it. For the average and worst case scenarios merge sort greatly outperforms insertion sort O(n^2). As the size of the list goes up, the difference increases exponentially. For this reason, insertion sort is only a viable algorithm to choose over merge sort in very small lists, or lists that are already very close to being ordered. Otherwise, you should always use merge sort over insertion sort.