schulz_lab03

April 13, 2021

1 Lab 3: Benchmarking Merge Sort

David Schulz

1.1 Introduction

We are going to implement the merge sort algorithm. We will then benchmark the run time of merge sort under the best, worst, and average case scenarios. We will plot the run times and interpret the plot in relation to the asymptotic run time. Furthermore, We will compare the run times of merge sort and insertion sort for three different scenarios.

1.1.1 Summary

In this lab, we prove that the time complexity for merge sort is always O(nlogn). We also benchmark insertion sort's runtimes for the best, worst, and average-case inputs and compare them to merge sort to prove which is more efficient for each case.

1.2 Implementation

[1]:
```
def insertion_sort(lst):
    """
    Takes a Python list and sorts it in place

    :param lst: The list that is being sorted
    """
    for i in range(1, len(lst)):
        key = lst[i]
        j = i-1
        while j >= 0 and key < lst[j]:
            lst[j+1] = lst[j]
            j -= 1
        lst[j+1] = key


def merge_sort(lst, p, r):
    """
    Takes a Python list and sorts it in place
```

```
    :param lst: The list that is being sorted
    :param p: The start index of the incoming list, i.e. 0
    :param r: The end index of the incoming list, i.e. len(lst)-1
    """
    if p < r:
        # Same as (l+r)//2, but avoids overflow for large l and h
        q = (p+(r-1))//2

        # Sort first and second halves
        merge_sort(lst, p, q)
        merge_sort(lst, q+1, r)
        merge(lst, p, q, r)


def merge(lst, p, q, r):
    """
    Merges two sorted partitions of a list

    :param lst: The list that is being sorted
    :param p: The starting index of first partition
    :param q: The ending index of first partition
    :param r: The ending index of second partition
    """
    n1 = q - p + 1
    n2 = r - q

    # Create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0 , n1):
        L[i] = lst[p + i]

    for j in range(0 , n2):
        R[j] = lst[q + 1 + j]

    # Merge the temp arrays back into lst[p..r]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = p     # Initial index of merged subarray

    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            lst[k] = L[i]
            i += 1
```

```
        else:
            lst[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there are any
    while i < n1:
        lst[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there are any
    while j < n2:
        lst[k] = R[j]
        j += 1
        k += 1
```

1.3 Benchmarking Merge Sort

[2]:
```
import numpy as np
import random
import time

list_sizes = [1000, 2500, 5000, 10000, 25000, 50000]
numbers = []

for i in range(0, 50000):
    numbers.append(random.random())

sort = numbers[:]
sort.sort()
reverse = numbers[:]
reverse.sort(reverse=True)
```

[3]:
```
def bench_insertion(lst, n_trials, n_nums):
    times = []
    for i in range(n_trials):
        start = time.perf_counter()
        lst2 = lst[:n_nums] # copy the input list so that we don't modify it
        insertion_sort(lst2)
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)

def bench_merge(lst, n_trials, n_nums):
    times = []
```

```
    for i in range(n_trials):
        start = time.perf_counter()
        lst2 = lst[:n_nums] # copy the input list so that we don't modify it
        merge_sort(lst2, 0, len(lst2)-1)
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)
```

[4]:
```
insert_shuffle = []
for size in list_sizes:
    insert_shuffle.append(bench_insertion(numbers, 3, size))

insert_sorted = []
for size in list_sizes:
    insert_sorted.append(bench_insertion(sort, 3, size))

insert_reverse = []
for size in list_sizes:
    insert_reverse.append(bench_insertion(reverse, 3, size))
```

[5]:
```
merge_shuffle = []
for size in list_sizes:
    merge_shuffle.append(bench_merge(numbers, 3, size))

merge_sorted = []
for size in list_sizes:
    merge_sorted.append(bench_merge(sort, 3, size))

merge_reverse = []
for size in list_sizes:
    merge_reverse.append(bench_merge(reverse, 3, size))
```

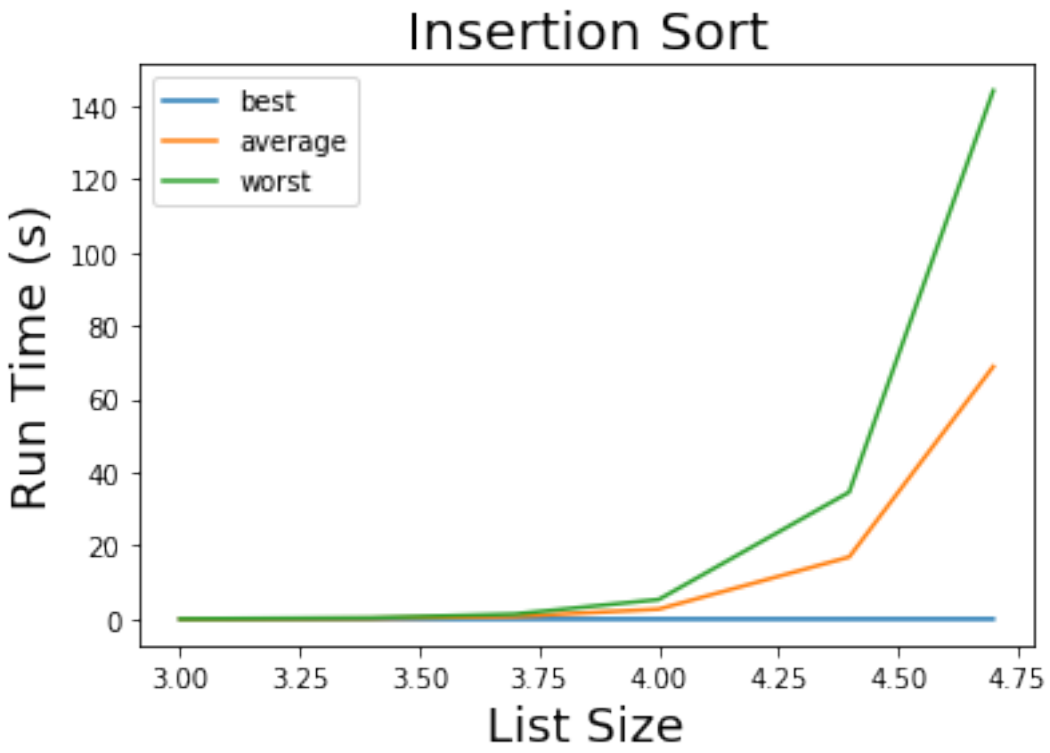
1.4 Plotting Benchmark Run Times

1.4.1 Comparing Three Cases Within Each Algorithm

[11]:
```
import matplotlib.pyplot as plt
```

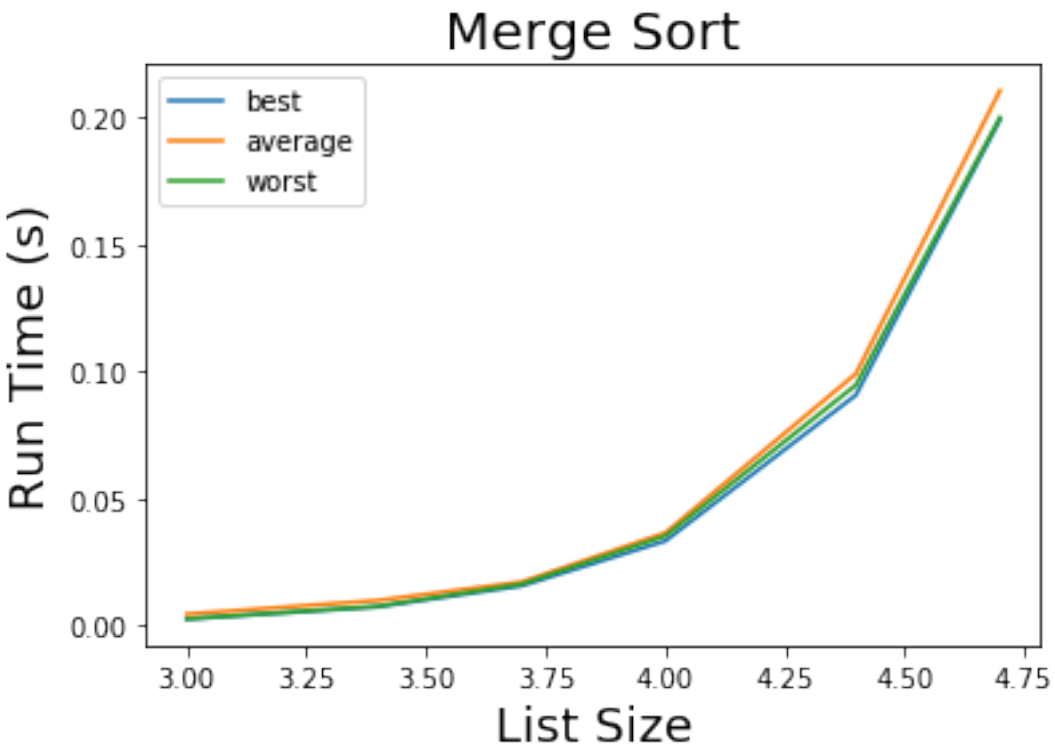
[12]:
```
plt.plot(np.log10(list_sizes), insert_sorted, label="best")
plt.plot(np.log10(list_sizes), insert_shuffle, label="average")
plt.plot(np.log10(list_sizes), insert_reverse, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Insertion Sort", fontsize=20)
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7fc6242f3f98>
```



[7]:
```
plt.plot(np.log10(list_sizes), merge_sorted, label="best")
plt.plot(np.log10(list_sizes), merge_shuffle, label="average")
plt.plot(np.log10(list_sizes), merge_reverse, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Merge Sort", fontsize=20)
plt.legend()
```

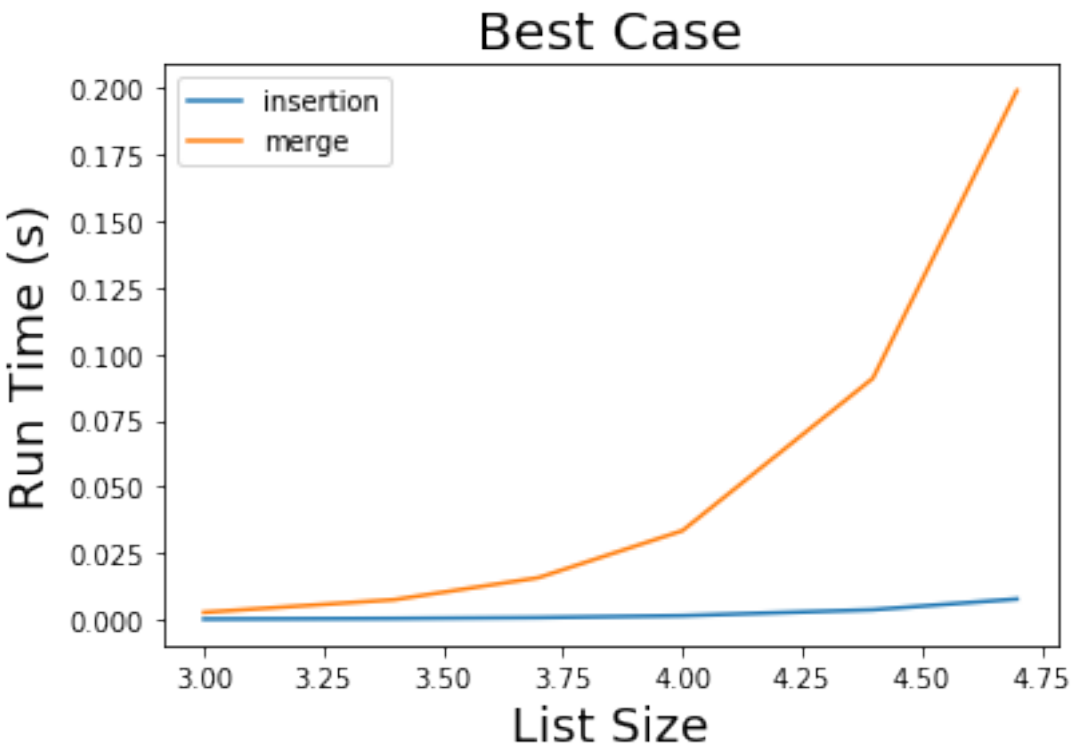
[7]:
```
<matplotlib.legend.Legend at 0x7fc6244e1470>
```



1.4.2 Comparing Three Cases Accross Each Algorithm

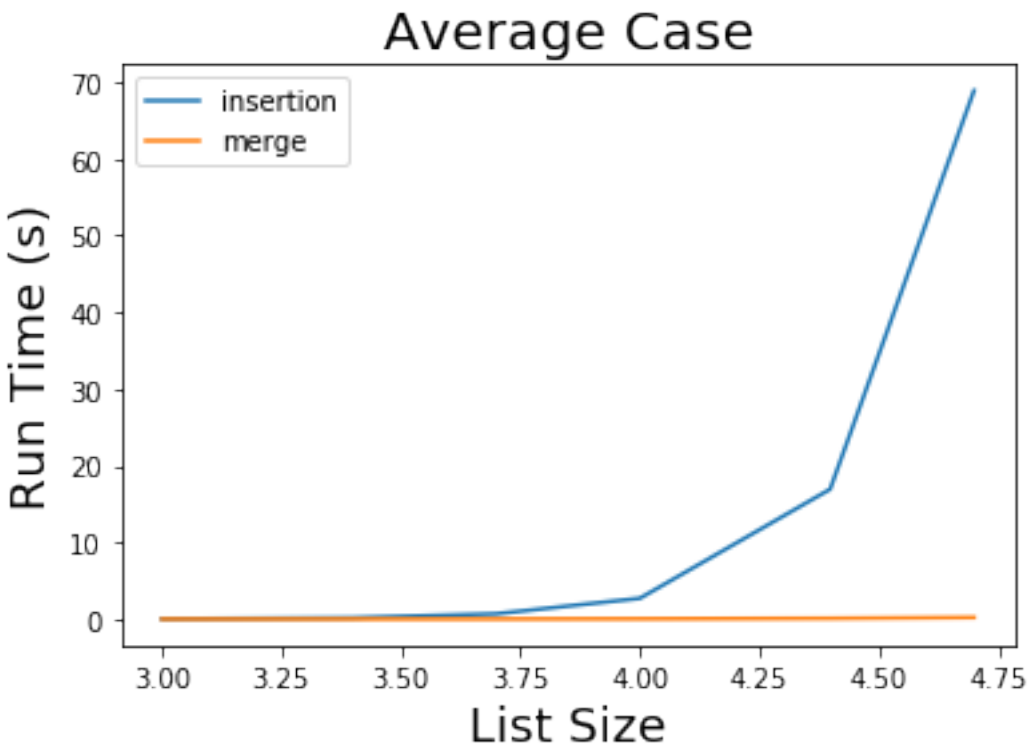
[8]:
```
plt.plot(np.log10(list_sizes), insert_sorted, label="insertion")
plt.plot(np.log10(list_sizes), merge_sorted, label="merge")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Best Case", fontsize=20)
plt.legend()
```

[8]:
```
<matplotlib.legend.Legend at 0x7fc624463ba8>
```



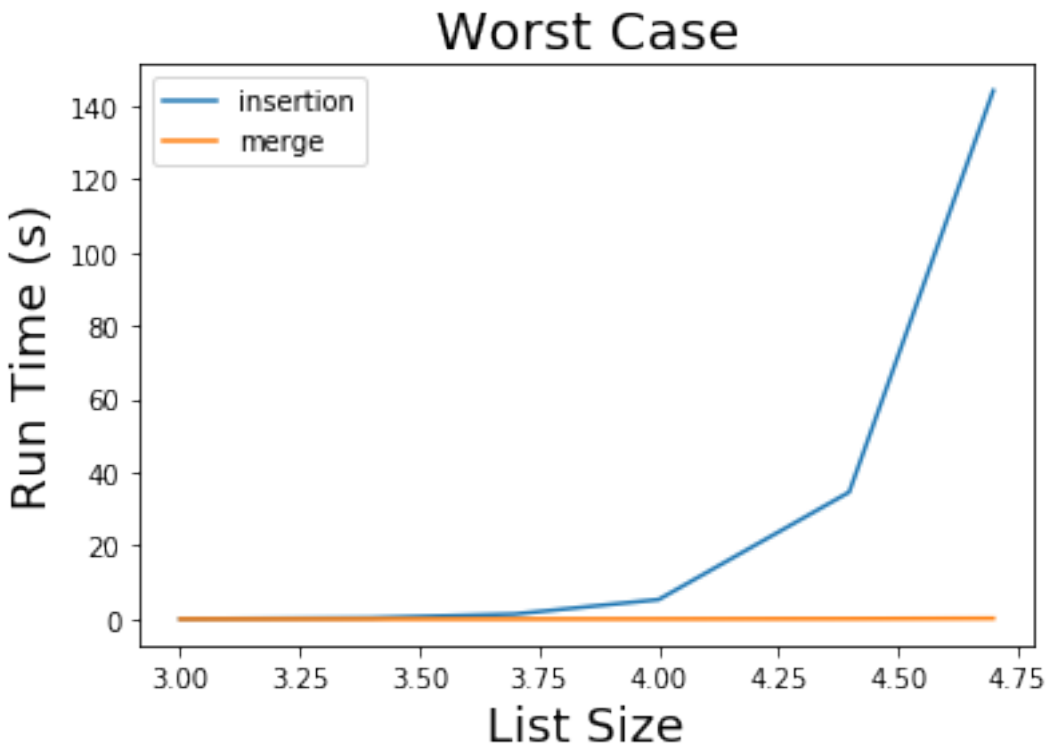
[9]:
```
plt.plot(np.log10(list_sizes), insert_shuffle, label="insertion")
plt.plot(np.log10(list_sizes), merge_shuffle, label="merge")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Average Case", fontsize=20)
plt.legend()
```

[9]:
```
<matplotlib.legend.Legend at 0x7fc6243fae80>
```



[10]:
```
plt.plot(np.log10(list_sizes), insert_reverse, label="insertion")
plt.plot(np.log10(list_sizes), merge_reverse, label="merge")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Worst Case", fontsize=20)
plt.legend()
```

[10]:
```
<matplotlib.legend.Legend at 0x7fc624376e48>
```



1.5 Reflection Questions

- Were there any differences between the run times of the three cases for merge sort? What inputs gave the best-, worst-, and average-case runs time?
  - No, there was almost no difference. Best-case input was a list that was already sorted, worst-case input was a list that was sorted in reverse order, and average-case input was a list with random values in no particular order.
- Why do you think the different inputs caused different run times for merge sort?
  - They didn't, since the tight bound is always nlogn.
- Look up the best-, worst-, and average-case run time of merge sort in big-oh notation and provide those. Are the formally determined run times consistent with your benchmarks?
  - All three cases are O(nlogn). This is consistent with my benchmarks.
- Were there any differences between the run times of the three cases for insertion sort vs merge sort?
  - Merge sort was extremely faster than insertion sort for the worst and average cases, but insertion sort turned out to be slightly faster than merge sort for the best case.
- Why do you think the different inputs caused different run times for insertion sort vs merge sort? Why do you think the best-case run time of insertion sort is faster than merge sort?

  - The average and worst case runtimes for insertion sort are both n^2, but they are both only nlogn for merge sort, which is why merge sort is so much faster. However, the best case time complexity for insertion sort is only n, which is slightly faster than nlogn, which is why insertion sort is faster than merge sort for the best case input.

- Are the formally determined run times (in terms of big-oh) consistent with your benchmarks?
  - Yes, the big-oh time complexities are consistent with my benchmarks.