schulz_lab02

March 24, 2021

1 Lab 2: Comparing Insertion and Selection Sort

David Schulz

1.1 Introduction

We are going to implement the insertion and selecting sort algorithms. We will then benchmark the run time of the two algorithms under the best, worst, and average case scenarios. We will plot the run times and interpret the plots in relation to the asymptotic run time.

1.2 Implement Insertion and Selection Sort

[1]:
```
def insertion_sort(A):
    for i in range(1, len(A)):
        key = A[i]
        j = i-1
        while j >= 0 and key < A[j]:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = key


def selection_sort(A):
    n = len(A)
    for j in range(1, n-1):
        smallest = j
        for i in range(j+1, n):
            if A[i] < A[smallest]:
                smallest = i
        tmp = A[j]
        A[j] = A[smallest]
        A[smallest] = tmp
```

1.3 Benchmark the Sorting Algorithms

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

numbers = []

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

random.shuffle(numbers)
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_selection(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
        selection_sort(lst2)
        end = time.perf_counter()
        elapsed = end - start
        times.append(elapsed)
    return np.mean(times)
```

[4]:
```
insert_shuffle = []
insert_shuffle.append(bench_insertion(numbers, 3, 1000))
insert_shuffle.append(bench_insertion(numbers, 3, 2500))
insert_shuffle.append(bench_insertion(numbers, 3, 5000))
insert_shuffle.append(bench_insertion(numbers, 3, 10000))
insert_shuffle.append(bench_insertion(numbers, 3, 25000))
insert_shuffle.append(bench_insertion(numbers, 3, 50000))

insert_sorted = []
```

```
insert_sorted.append(bench_insertion(sort, 3, 1000))
insert_sorted.append(bench_insertion(sort, 3, 2500))
insert_sorted.append(bench_insertion(sort, 3, 5000))
insert_sorted.append(bench_insertion(sort, 3, 10000))
insert_sorted.append(bench_insertion(sort, 3, 25000))
insert_sorted.append(bench_insertion(sort, 3, 50000))

insert_reverse = []
insert_reverse.append(bench_insertion(reverse, 3, 1000))
insert_reverse.append(bench_insertion(reverse, 3, 2500))
insert_reverse.append(bench_insertion(reverse, 3, 5000))
insert_reverse.append(bench_insertion(reverse, 3, 10000))
insert_reverse.append(bench_insertion(reverse, 3, 25000))
insert_reverse.append(bench_insertion(reverse, 3, 50000))
```

[5]:
```
select_shuffle = []
select_shuffle.append(bench_selection(numbers, 3, 1000))
select_shuffle.append(bench_selection(numbers, 3, 2500))
select_shuffle.append(bench_selection(numbers, 3, 5000))
select_shuffle.append(bench_selection(numbers, 3, 10000))
select_shuffle.append(bench_selection(numbers, 3, 25000))
select_shuffle.append(bench_selection(numbers, 3, 50000))

select_sorted = []
select_sorted.append(bench_selection(sort, 3, 1000))
select_sorted.append(bench_selection(sort, 3, 2500))
select_sorted.append(bench_selection(sort, 3, 5000))
select_sorted.append(bench_selection(sort, 3, 10000))
select_sorted.append(bench_selection(sort, 3, 25000))
select_sorted.append(bench_selection(sort, 3, 50000))

select_reverse = []
select_reverse.append(bench_selection(reverse, 3, 1000))
select_reverse.append(bench_selection(reverse, 3, 2500))
select_reverse.append(bench_selection(reverse, 3, 5000))
select_reverse.append(bench_selection(reverse, 3, 10000))
select_reverse.append(bench_selection(reverse, 3, 25000))
select_reverse.append(bench_selection(reverse, 3, 50000))
```

1.4 Plot Benchmark Run Times

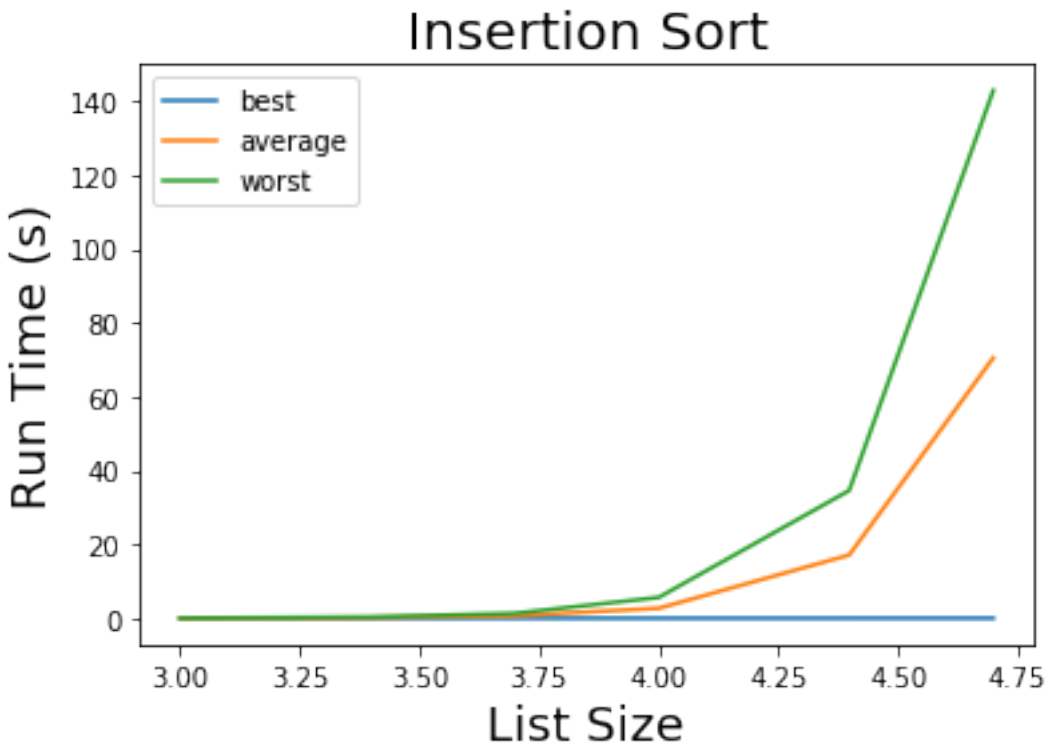
1.4.1 Comparing Three Cases Within Each Algorithm

[9]:
```
import matplotlib.pyplot as plt

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

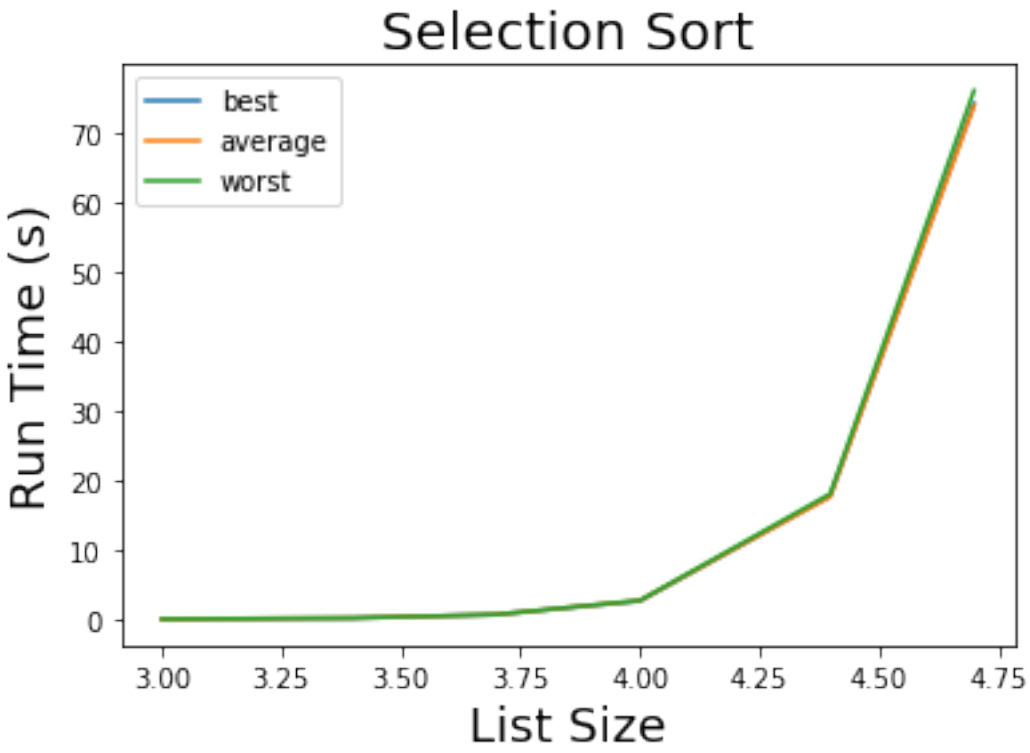
```
plt.plot(np.log10(list_sizes), insert_sorted, label="best")
plt.plot(np.log10(list_sizes), insert_shuffle, label="average")
plt.plot(np.log10(list_sizes), insert_reverse, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Insertion Sort", fontsize=20)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

[9]: `<matplotlib.legend.Legend at 0x7f96a6c659e8>`



[10]:
```
plt.plot(np.log10(list_sizes), select_sorted, label="best")
plt.plot(np.log10(list_sizes), select_shuffle, label="average")
plt.plot(np.log10(list_sizes), select_reverse, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Selection Sort", fontsize=20)
plt.legend()
```

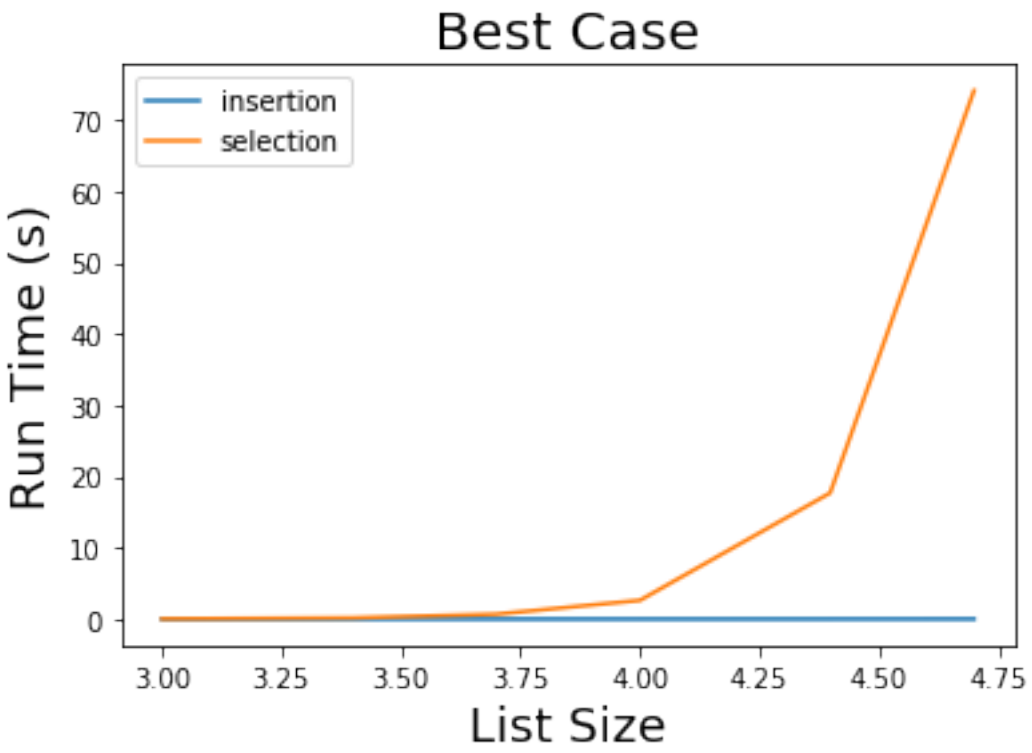
[10]: `<matplotlib.legend.Legend at 0x7f96a6beb7f0>`



1.4.2 Comparing Three Cases Accross Each Algorithm

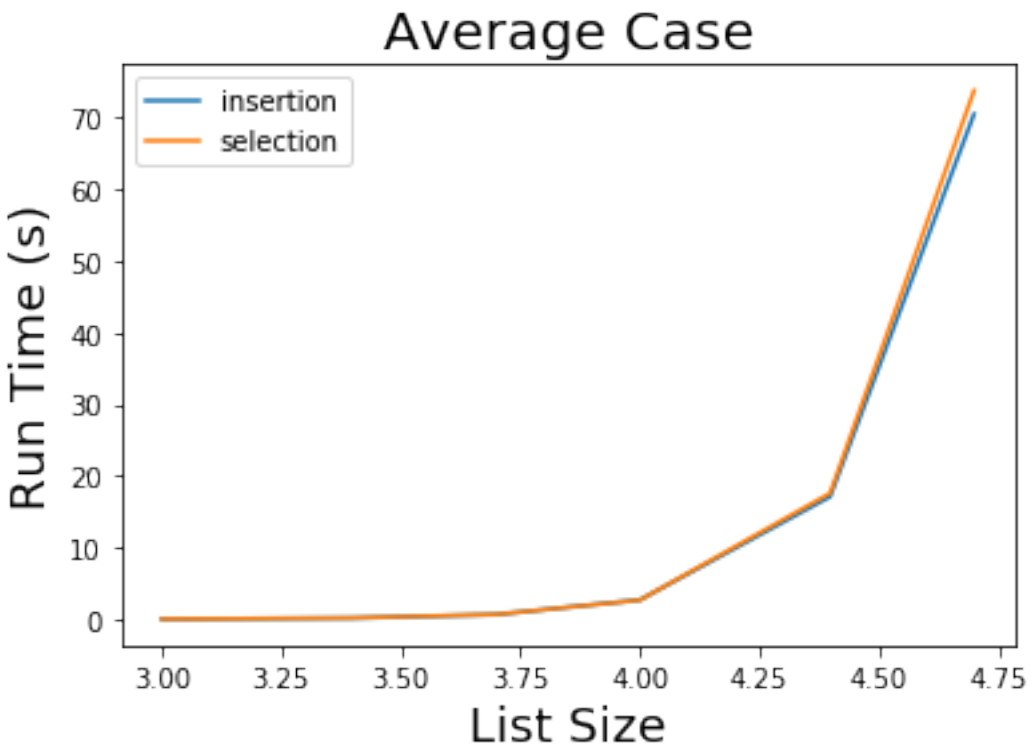
[11]:
```
plt.plot(np.log10(list_sizes), insert_sorted, label="insertion")
plt.plot(np.log10(list_sizes), select_sorted, label="selection")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Best Case", fontsize=20)
plt.legend()
```

[11]:
```
<matplotlib.legend.Legend at 0x7f96a6b7aeb8>
```



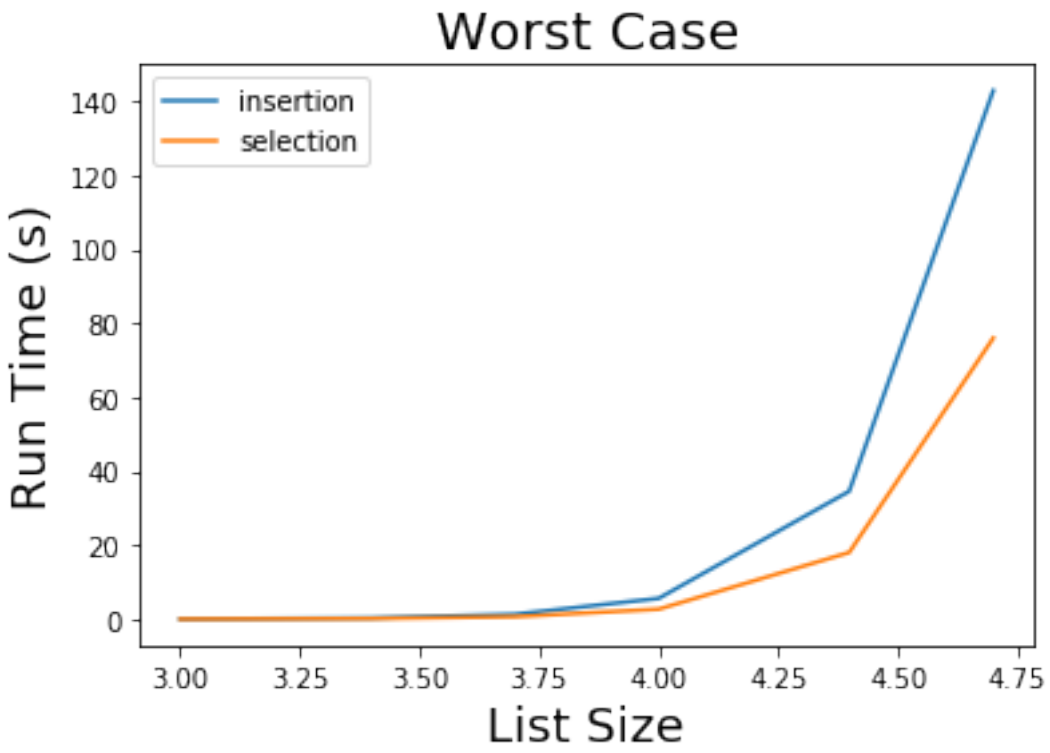
[12]:
```
plt.plot(np.log10(list_sizes), insert_shuffle, label="insertion")
plt.plot(np.log10(list_sizes), select_shuffle, label="selection")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Average Case", fontsize=20)
plt.legend()
```

[12]:
```
<matplotlib.legend.Legend at 0x7f96a6ada0f0>
```



[13]:
```
plt.plot(np.log10(list_sizes), insert_reverse, label="insertion")
plt.plot(np.log10(list_sizes), select_reverse, label="selection")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Worst Case", fontsize=20)
plt.legend()
```

[13]:
```
<matplotlib.legend.Legend at 0x7f96a6a6ef60>
```



1.5 Reflection Questions

1. Were there substantial differences between the run times of the three cases for insertion sort? If so, which case was fastest?
   - Yes. The best case was the fastest.
2. Were there substantial differences between the run times of the three cases for selection sort?
   - No.
3. Look up the best-, worst-, and average-case run time of insertion and selection sort in big-O notation and provide those. Create a table. Were the run times consistent with the big-o asymptotic analysis?
   - Yes, the run times were consistent with their Big-O analysis.

	Best	Worst	Average
Insertion	O(n)	O(n^2)	O(n^2)
Selection	O(n^2)	O(n^2)	O(n^2)

4. Why do you think one case was substantially faster for insertion sort but not selection sort? (Hint: focus on the inner loop of insertion sort.)

- While selection sort's inner loop checks through every single remaining unchecked value for the minimum no matter what, insertion sort's inner loop stops checking through the already sorted values when the key is greater than or equal to the value. Because of this, when an already perfectly sorted list is used as input, it doesn't matter how sorted a list is already for selection sort because it always checks everything anyways, but insertion sort will only have to compare the one value before the key as it iterates and not have to move anything.

5. Based on your results, which of the two sorting algorithms would you use in practice?

    - Since there is much more reward for the better cases and not much of a difference for worse cases, I would probably always use insertion sort.