CS 3851 Algorithms - Lab 2: Comparing Insertion and Selection Sort

Stuart Harley

Introduction

In this lab we are implementing the sorting algorithms insertion sort and selection sort. 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.

Importing Libraries

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

Implementing Insertion Sort

In [2]:
```
def insertion_sort(lst) -> None:
    """
    Sorts a list in place by picking the next value from the unsorted part of
 the list
    and inserting it in the correct place in the sorted part.
    """
    for j in range(1, len(lst)):
        key = lst[j]
        i = j-1
        while i>=0 and lst[i] > key:
            lst[i+1] = lst[i]
            i = i-1
        lst[i+1] = key
```

Implementing Selection Sort

In [3]:
```
def selection_sort(lst) -> None:
    """
    Sorts a list in place by repeatedly finding the smallest element from the
 unsorted
    part and putting it at the beginning.
    """
    n = len(lst)
    for j in range(n-1):
        smallest = j
        for i in range(j+1, n):
            if lst[i] < lst[smallest]:
                smallest = i
        tmp = lst[j]
        lst[j] = lst[smallest]
        lst[smallest] = tmp
```

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 1000, 2500, 5000, 10000, 25000, and 50000 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 [4]:
```
l1000 = []
for x in range(1000):
    l1000.append(random.random())
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]:
```
l2500 = []
for x in range(2500):
    l2500.append(random.random())
random.shuffle(l2500) # shuffling the list to generate a random permutation
l2500_p = l2500[:]
l2500.sort() # sorting the list in order
l2500_o = l2500[:]
l2500.sort(reverse=True) # sorting the list in reverse order
l2500_r = l2500[:]
```

In [6]:
```
l5000 = []
for x in range(5000):
    l5000.append(random.random())
random.shuffle(l5000) # shuffling the list to generate a random permutation
l5000_p = l5000[:]
l5000.sort() # sorting the list in order
l5000_o = l5000[:]
l5000.sort(reverse=True) # sorting the list in reverse order
l5000_r = l5000[:]
```

In [7]:
```
l10000 = []
for x in range(10000):
    l10000.append(random.random())
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 [8]:
```
l25000 = []
for x in range(25000):
    l25000.append(random.random())
random.shuffle(l25000) # shuffling the list to generate a random permutation
l25000_p = l25000[:]
l25000.sort() # sorting the list in order
l25000_o = l25000[:]
l25000.sort(reverse=True) # sorting the list in reverse order
l25000_r = l25000[:]
```

In [9]:
```
l50000 = []
for x in range(50000):
    l50000.append(random.random())
random.shuffle(l50000) # shuffling the list to generate a random permutation
l50000_p = l50000[:]
l50000.sort() # sorting the list in order
l50000_o = l50000[:]
l50000.sort(reverse=True) # sorting the list in reverse order
l50000_r = l50000[:]
```

Benchmarking the Insertion Sort Algorithm

We are running each benchmark 3 times (trials).

In [10]:
```
n_trials = 3
run_times_best_i = []
run_times_average_i = []
run_times_worst_i = []
```

In [11]:
```
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    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_i.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.0321856546215713 sec
```

In [12]:
```
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('1000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_i.append(np.mean(times))
```

```
1000 elements: Sorted in Order
Average Time: 0.00015166564844548702 sec
```

In [13]:
```
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('1000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
1000 elements: Sorted in Reverse
Average Time: 0.05797686466636757 sec
```

In [14]:
```
# 2500 elements
times = []
for i in range(n_trials):
    lst2 = l2500_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('2500 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 = l2500_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('2500 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 = l2500_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('2500 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
2500 elements: randomly permuted
Average Time: 0.19090306929623088 sec
2500 elements: Sorted in Order
Average Time: 0.0003677193696300189 sec
2500 elements: Sorted in Reverse
Average Time: 0.3693540373351425 sec
```

In [15]:
```
# 5000 elements
times = []
for i in range(n_trials):
    lst2 = l5000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('5000 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 = l5000_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('5000 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 = l5000_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('5000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
5000 elements: randomly permuted
Average Time: 0.7608739707308511 sec
5000 elements: Sorted in Order
Average Time: 0.0007379932794719934 sec
5000 elements: Sorted in Reverse
Average Time: 1.484782649980237 sec
```

In [16]:
```
# 10,000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    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_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('10000 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('10000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 3.041042390279472 sec
10000 elements: Sorted in Order
Average Time: 0.0015111076645553112 sec
10000 elements: Sorted in Reverse
Average Time: 6.049909281001116 sec
```

In [17]:
```
# 25,000 elements
times = []
for i in range(n_trials):
    lst2 = l25000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('25000 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 = l25000_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('25000 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 = l25000_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('25000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
25000 elements: randomly permuted
Average Time: 19.068582944960024 sec
25000 elements: Sorted in Order
Average Time: 0.004138392396271229 sec
25000 elements: Sorted in Reverse
Average Time: 37.788394283115245 sec
```

In [18]:
```
# 50,000 elements
times = []
for i in range(n_trials):
    lst2 = l50000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    insertion_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('50000 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 = l50000_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('50000 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 = l50000_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('50000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_i.append(np.mean(times))
```

```
50000 elements: randomly permuted
Average Time: 78.1450227809449 sec
50000 elements: Sorted in Order
Average Time: 0.0093936532891045 sec
50000 elements: Sorted in Reverse
Average Time: 152.88598153266744 sec
```

Benchmarking the Selection Sort Algorithm

In [19]:
```
run_times_best_s = []
run_times_average_s = []
run_times_worst_s = []
```

In [20]:
```
# 1000 elements
times = []
for i in range(n_trials):
    lst2 = l1000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    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_s.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()
    selection_sort(lst2)
    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_s.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()
    selection_sort(lst2)
    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_s.append(np.mean(times))
```

```
1000 elements: randomly permuted
Average Time: 0.03073661293213566 sec
1000 elements: Sorted in Order
Average Time: 0.02791803873454531 sec
1000 elements: Sorted in Reverse
Average Time: 0.029116273236771423 sec
```

In [21]:
```
# 2500 elements
times = []
for i in range(n_trials):
    lst2 = l2500_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('2500 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l2500_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('2500 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l2500_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('2500 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_s.append(np.mean(times))
```

```
2500 elements: randomly permuted
Average Time: 0.1748408693820238 sec
2500 elements: Sorted in Order
Average Time: 0.17671284700433412 sec
2500 elements: Sorted in Reverse
Average Time: 0.18041938558841744 sec
```

In [22]:
```
# 5000 elements
times = []
for i in range(n_trials):
    lst2 = l5000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('5000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l5000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('5000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l5000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('5000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_s.append(np.mean(times))
```

```
5000 elements: randomly permuted
Average Time: 0.6901715364462385 sec
5000 elements: Sorted in Order
Average Time: 0.6885464199973891 sec
5000 elements: Sorted in Reverse
Average Time: 0.7164723892540982 sec
```

In [23]:
```
# 10,000 elements
times = []
for i in range(n_trials):
    lst2 = l10000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    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_s.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()
    selection_sort(lst2)
    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_s.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()
    selection_sort(lst2)
    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_s.append(np.mean(times))
```

```
10000 elements: randomly permuted
Average Time: 2.7626505060276636 sec
10000 elements: Sorted in Order
Average Time: 2.752296094006548 sec
10000 elements: Sorted in Reverse
Average Time: 2.8360447906112918 sec
```

In [24]:
```
# 25,000 elements
times = []
for i in range(n_trials):
    lst2 = l25000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('25000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l25000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('25000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l25000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('25000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_s.append(np.mean(times))
```

```
25000 elements: randomly permuted
Average Time: 17.22596439400998 sec
25000 elements: Sorted in Order
Average Time: 17.336456394366298 sec
25000 elements: Sorted in Reverse
Average Time: 17.957872592688847 sec
```

In [25]:
```
# 50,000 elements
times = []
for i in range(n_trials):
    lst2 = l50000_p[:] # copying the input list so we don't modify it
    random.shuffle(lst2); # Shuffling the list so the average is over 3 differ
ent permutations
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('50000 elements: randomly permuted')
print('Average Time:', np.mean(times), 'sec')
run_times_average_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l50000_o[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('50000 elements: Sorted in Order')
print('Average Time:', np.mean(times), 'sec')
run_times_best_s.append(np.mean(times))

times = []
for i in range(n_trials):
    lst2 = l50000_r[:] # copying the input list so we don't modify it
    start = time.perf_counter()
    selection_sort(lst2)
    end = time.perf_counter()
    elapsed = end - start
    times.append(elapsed)
print('50000 elements: Sorted in Reverse')
print('Average Time:', np.mean(times), 'sec')
run_times_worst_s.append(np.mean(times))
```

```
50000 elements: randomly permuted
Average Time: 78.33756884311636 sec
50000 elements: Sorted in Order
Average Time: 78.60186078663294 sec
50000 elements: Sorted in Reverse
Average Time: 81.05701429637459 sec
```

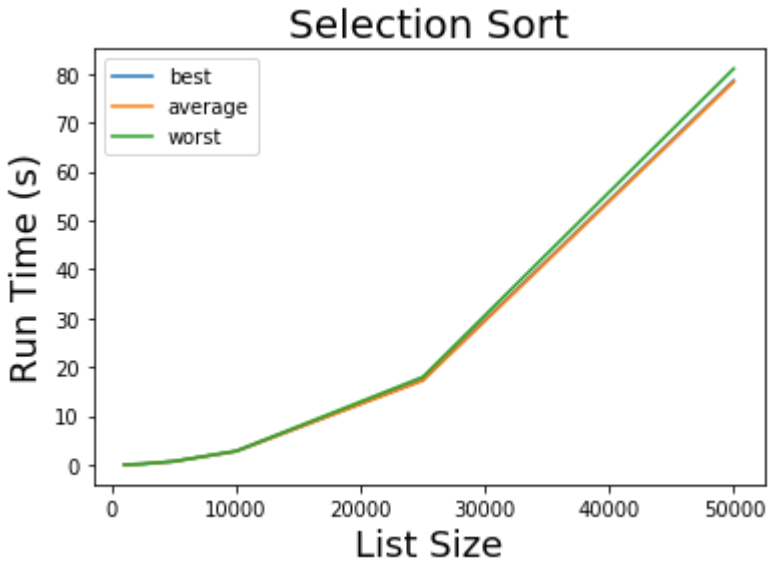
Plotting the Benchmarked Run Times

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

In [26]:
```
list_sizes = [1000, 2500, 5000, 10000, 25000, 50000]
```

In [27]:
```
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();
```

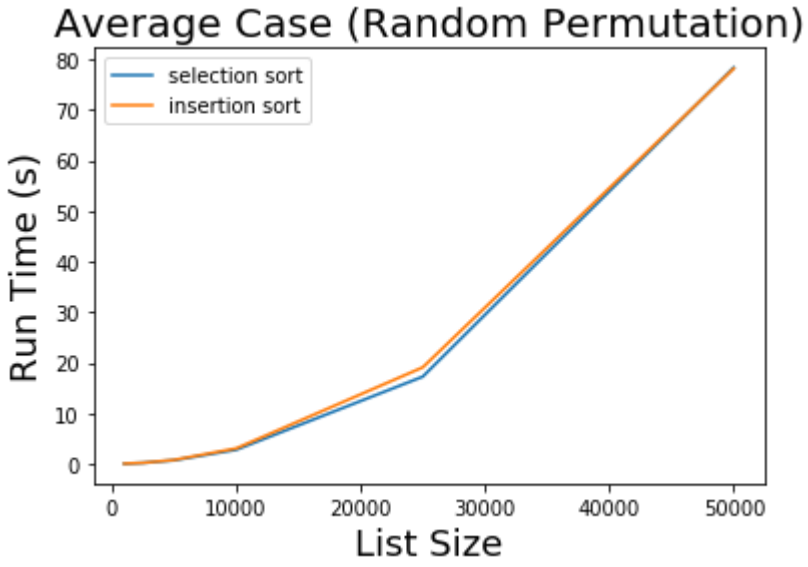
In [28]:
```
plt.plot(list_sizes, run_times_best_s, label="best")
plt.plot(list_sizes, run_times_average_s, label="average")
plt.plot(list_sizes, run_times_worst_s, label="worst")
plt.xlabel("List Size", fontsize=18)
plt.ylabel("Run Time (s)", fontsize=18)
plt.title("Selection Sort", fontsize=20)
plt.legend();
```



Creating one plot for each case.

In [29]:
```
plt.plot(list_sizes, run_times_best_s, label="selection 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();
```

In [30]:
```
plt.plot(list_sizes, run_times_average_s, label="selection 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();
```



In [31]:
```
plt.plot(list_sizes, run_times_worst_s, label="selection 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();
```

Reflection Questions

1) There were substantial differences between the run times of the three cases for insertion sort. The best case (already sorted) was the fastest by far.

2) There were not substantial differences between the run times of the three cases for selection sort.

3)

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

4) One case is significantly faster than the others for insertion sort because there is a while loop within insertion sort. In the best case scenario, the while loop is never true so no values have to be shifted. In selection sort, the inner loop is a for loop which checks every value past the current value for the smallest value. It doesn't matter if the list is already in order, it still checks every next value for every loop.

5) Based on my results, I would use selection sort in practice. It runs in a consistent amount of time for all cases so it can be easily predicted how long it will take. Also, if I am needing to sort data in practice, it is likely that that data is not ordered currently, so insertion sort would be unlikely to perform significantly better.

Conclusion

The benchmarking above provided valuable insights into the time complexities of insertion and selection sort. Insertion sort performs better on data that is closer to already being sorted and performs worse on data that is closer to reverse sorted order. Selection sort sorts data in basically the same amount of time no matter the order.

However, both sorting algorithms are O(n^2). If you were looking to actually sort data better you could just call the sort() method in python, which uses Timsort under the hood, which is O(nlogn) and so is much quicker.