CSCE 221: Programming Assignment 2 (100 points)

Due February 20 by 11:59 pm

Project Cover Page

This project is a group project. For each group member, please print first and last name and e-mail address.

1. Asa Hayes, email: asahayes@tamu.edu, UIN: 525003952

2. Dakota Skinkis, email: dakotaskinkis@tamu.edu, UIN: 725004450

3. Jeslyn Wegner, email: jeslynwegner@tamu.edu, UIN: 925007462

Please write how each member of the group participated in the project.

1. Asa Hayes: Created and set up GitHub repository, wrote report

2. Dakota Skinkis: Implemented file reading code and selection sort

3. Jeslyn Wegner: Implemented bubble and insertion sorts, carried out testing for operations per sort

Please list all sources: web pages, people, books or any printed material, which you used to prepare a report and implementation of algorithms for the project.

Type of sources:	
People	
Web Material (give URL)	
Printed Material	Book: Data Structures & Algorithms in C++, 2nd Edition, Wiley
Other Sources	

I certify that I have listed all the sources that I used to develop solutions to the submitted project and code.

Your signature: Asa Hayes Date: 20 February, 2018

I certify that I have listed all the sources that I used to develop a solution to the submitted project and code.

Your signature: Dakota Skinkis Date: 20 February, 2018

I certify that I have listed all the sources that I used to develop solution to the submitted project and code.

Your signature: Jeslyn Wegner Date: 20 February, 2018

Aggie Code of Honor: "An Aggie does not lie, cheat, or steal, or tolerate those who do."

1 General Details; Purpose, Description, I/O formats, Run Instructions

1.1 Purpose/Description

This project is intended to give us a better understanding of sorting processes. Consider a situation that you have booked a flight from Houston to London. But due to unfortunate reasons, you reach the airport late. Now in this situation you look up to the board with flight details to check the status of your flight. If the flight board is not ordered in some way, this reduces readability. However once you sort the list by any field, it is much easier to check for your flight. This gives a reasonable scenarios where we can easily see the need for sorting data.

1.2 Input/Output Format, Run/Compile Instructions

The folder of flight tables for input do not need to be moved from where they are in the main directory, nor do any new files need to be created. The program is compiled and run with "make all && ./PA2" as shown in the assignment document. After a table is run thru whichever sort, a new text file is created for the sorted table to be written to. The amount of comparisons will be shown in the terminal for each of the applied sorts.

2 Program Design

2.1 Class Definitions

While the program has three different files for functions, only one of those constitutes a class, that being the Flight class. Objects from the Flight class have four main members, strings for each category of flight data, as well as two comparison functions (not overloading any comparison operators) to help more accurately compare data members between two flight objects due to string comparison in C++ not being as automatic as numerical comparison. As for the other non-main file, it contains the sorting functions which will be discussed later. Aside from the regular practice of splitting the program into functions for reusability, there are no major C++ OOP features to demonstrate as they were not necessary in this application, nor were any generic programming features, as this program only deals with a single specific type of input.

2.2 Sorting Algorithms

2.2.1 Selection Sort

The selection sort is a simple, if inefficient sort. Given a list of elements, it searches through the entire list, moves the least element in the list to the start of a new list (the sorted list), and removes that element from the first list. While this does result in the unsorted list becoming smaller with each iteration, a full search through every element of the unsorted list has to happen each iteration, making the amount of time needed to complete a selection sort grow very fast with inputs of larger sizes.

2.2.2 Insertion Sort

The insertion sort is a bit of a smarter use of the same principle behind selection sort, in that it moves elements from one unsorted list to a new sorted one based on value. However, selection sort is more efficient in most cases as it does not search through the entire unsorted list for each

element, instead taking the next element from the unsorted list and inserting it into the sorted list based on value. This is more efficient due to the smarter searches that can be used on sorted data sets, such as binary search or others, that can dramatically reduce the needed amount of calculations.

2.2.3 Bubble Sort

The bubble sort differs from the two other sorts as it works within the same list. Instead of searching, bubble sort moves thru a list two elements at a time and swaps them if the further element of the two is greater than the closer element, assuming the list is being sorted in ascending order. While it does not perform any searches, it still requires a large amount of operations as each element may require many swaps to get to its proper location.

3 Theoretical Analysis

For the sorting times of each sort in big O notation (Table 3-2), we can reasonably assume correspondence to the best/worst/average number of operations (Table 3-1). Lists in increasing order would be the best case, as the list already has every element in its proper place. The worst case would be the decreasing list, as the distance between each element and its proper location are maximized (overall, they could be larger on an element by element basis). The average case would be random, as the randomly sorted list most resembles a list that would be seen in non-theoretical applications.

# of comps.	best	average	worst
Selection Sort	n^2	n^2	n^2
Insertion Sort	n	n^2	n^2
Bubble Sort	n	n^2	n^2

big-O notation	inc	ran	dec
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

4 Experiments and Statistics

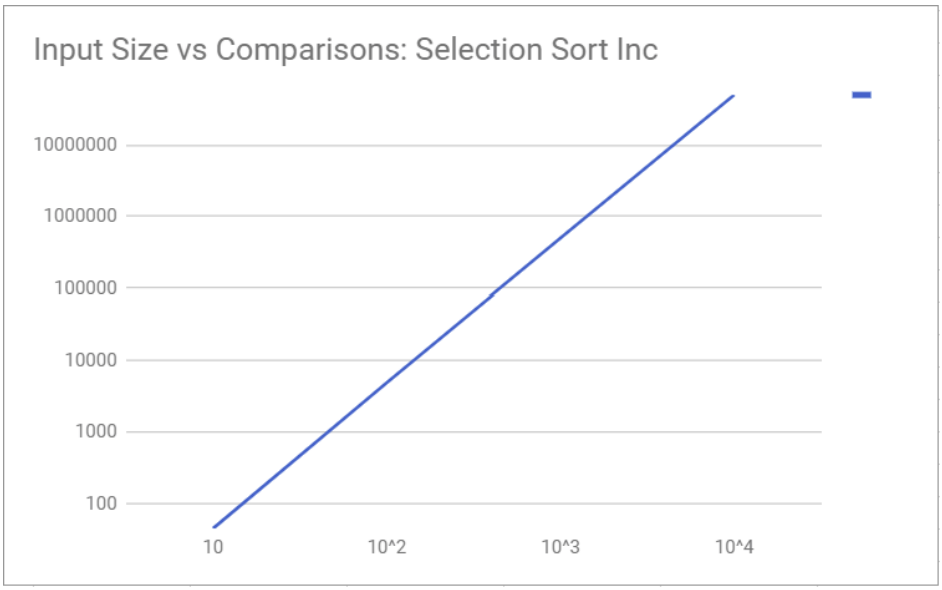
For the experiment, we simply allowed the program to run as normal. The amount of comparisons was kept track of by the provided "num_cmps" and displayed for each sort after it was completed. As our implementation has all of the files sorted in one run, no separate tests were necessary.

4.1 Table for Number of Comparisons Per Sort and Input Size/Arrangement

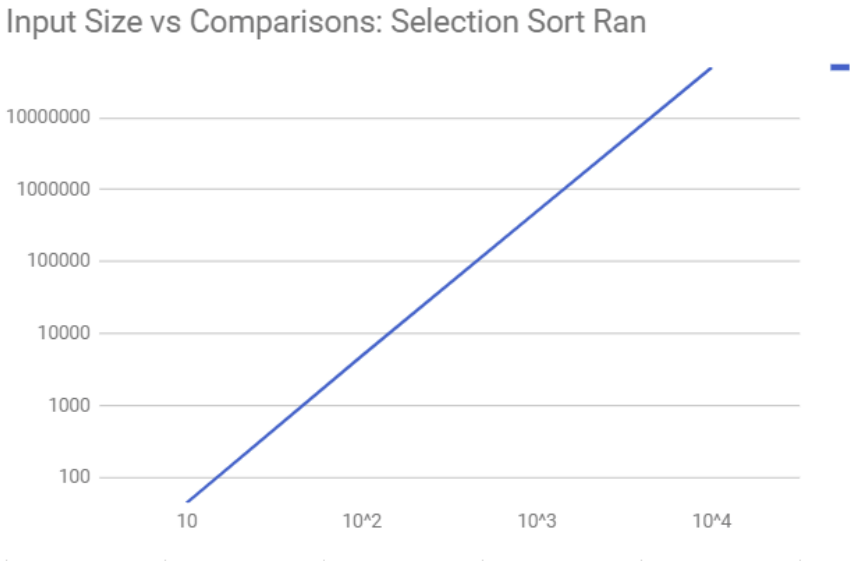
#COMP	Selection Sort			Bubble Sort			Insertion Sort		
n	inc	ran	dec	inc	ran	dec	inc	ran	dec
10	45	45	45	9	42	45	10	24	46
10^2	4950	4950	4950	99	4947	4950	100	2570	4951
10^3	499500	499500	499500	999	498554	499500	1000	248377	499501
10^4	49995000	49995000	49995000	9999	49969349	49994990	10000	24753524	49965226

inc: increasing order; dec: decreasing order; ran: random order

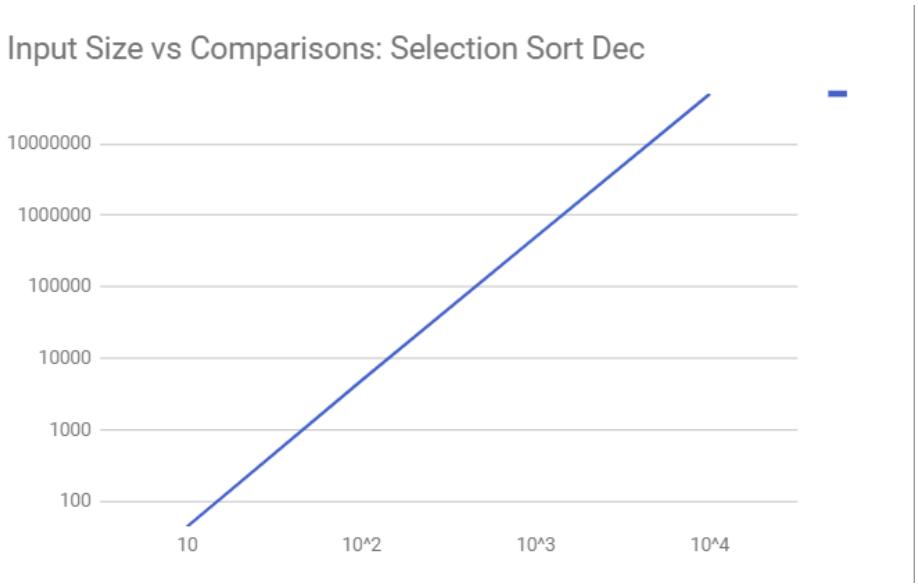
4.2 Graphs of Each Search's Number of Comparisons vs. Input Size of $n = 10^4$



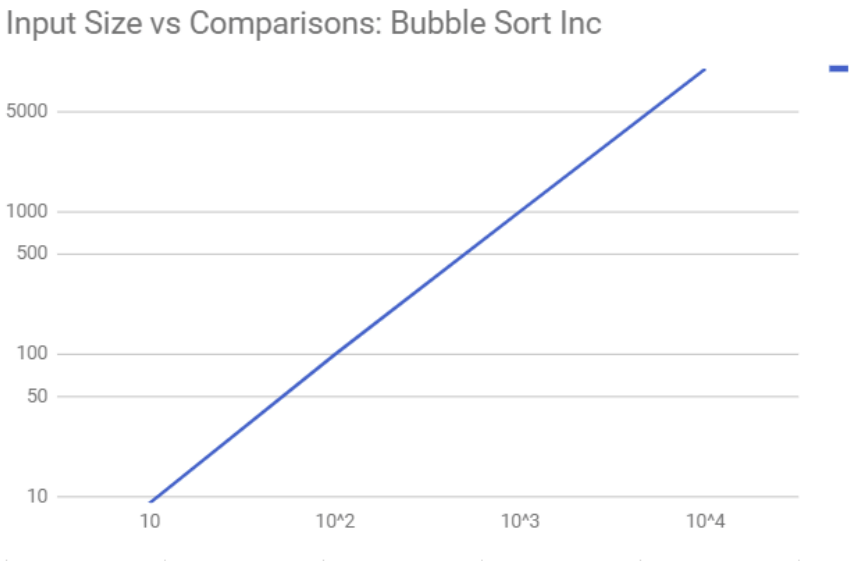
-



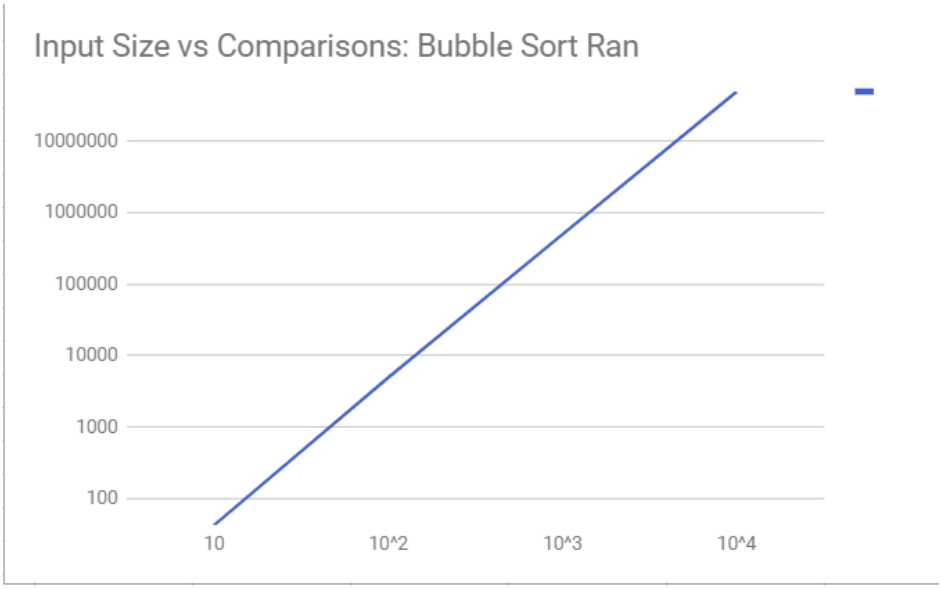
-


Input Size vs Comparisons: Selection Sort Dec
10000000
1000000
100000
10000
1000
100
10
10^2
10^3
10^4

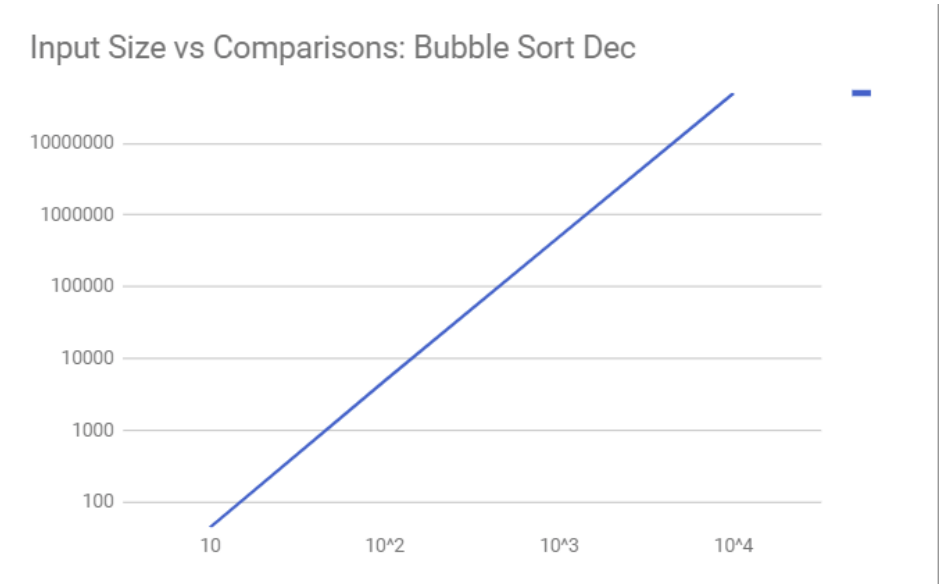
-


Input Size vs Comparisons: Bubble Sort Inc
5000
1000
500
100
50
10
10
10^2
10^3
10^4

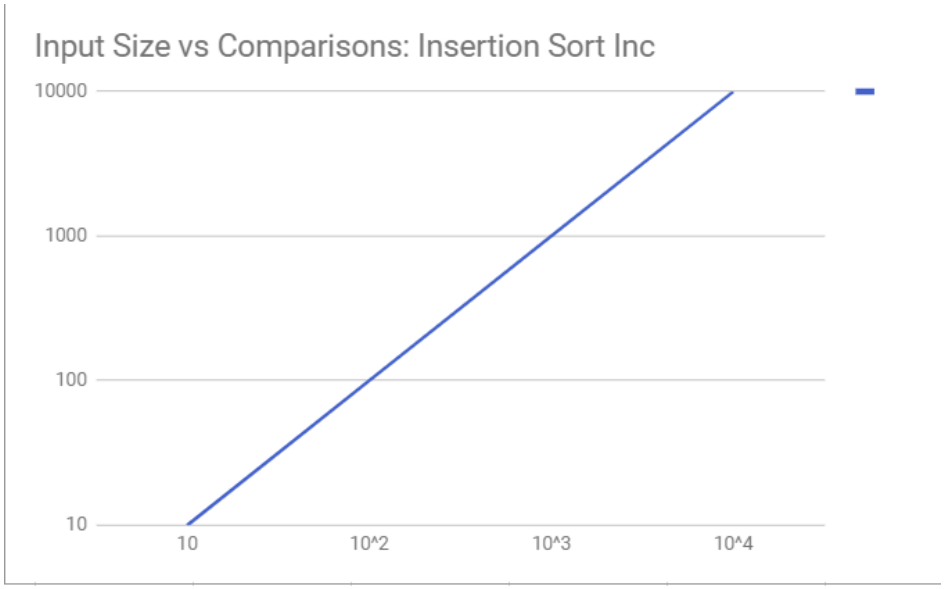
-


Input Size vs Comparisons: Bubble Sort Ran
10000000
1000000
100000
10000
1000
100
10
10^2
10^3
10^4

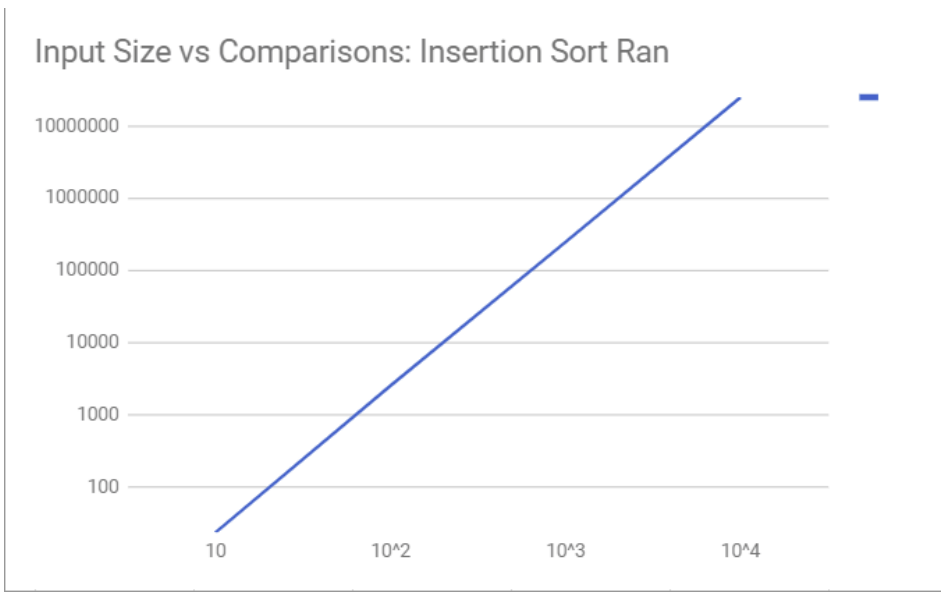
-


Input Size vs Comparisons: Bubble Sort Dec
10000000
1000000
100000
10000
1000
100
10
10^2
10^3
10^4

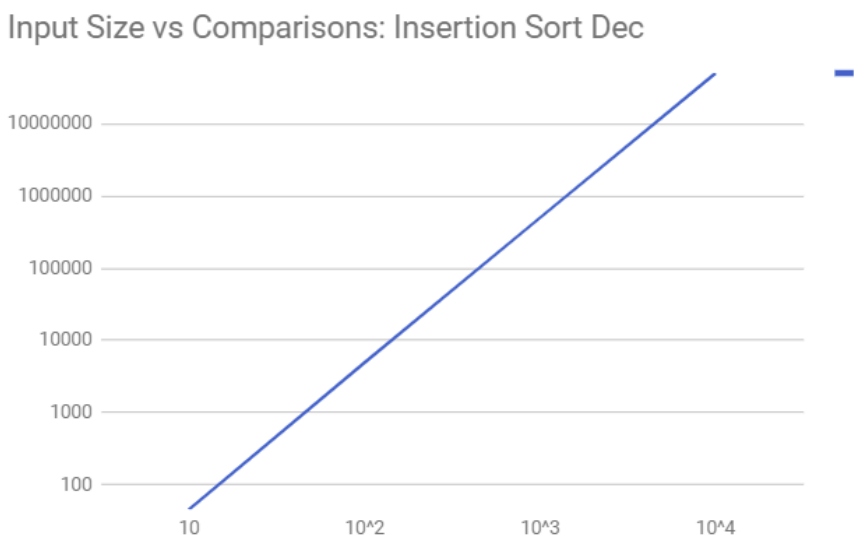
-


Input Size vs Comparisons: Insertion Sort Inc
10000
1000
100
10
10
10^2
10^3
10^4

-


Input Size vs Comparisons: Insertion Sort Ran
10000000
1000000
100000
10000
1000
100
10
10^2
10^3
10^4

-



4.3 Testing Cases for $n = 10$

```
// prints ascending order flights by departu
// change the bound of I to 10000 to print a

  //loop for sorting by departure time
  cout << "This is '_____10.cpp' sorted by d
for(int i = 10; i <= 10; i*=10){
   cout << "inc" << endl;
   printFlights(insertion_sort(readFlights(P
   cout << endl << "dec" << endl;
   printFlights(insertion_sort(readFlights(P
   cout << endl << "ran" << endl;
   printFlights(insertion_sort(readFlights(P
}
  cout << endl;
  //cout << "This is '_____10.cpp' sorted by
/* for(int i = 10; i <= 10; i*=10){
   printFlights(bubble_sort(readFlights(PATH
```

```
[jeslynwegner]@compute ~/FlightSorts> (20
:: make all && ./PA2
make: Nothing to be done for `all'.
This is '_____10.cpp' sorted by departure
inc
# of comparisons: 10

dec
# of comparisons: 46

ran
# of comparisons: 24


[jeslynwegner]@compute ~/FlightSorts> (20
::
```

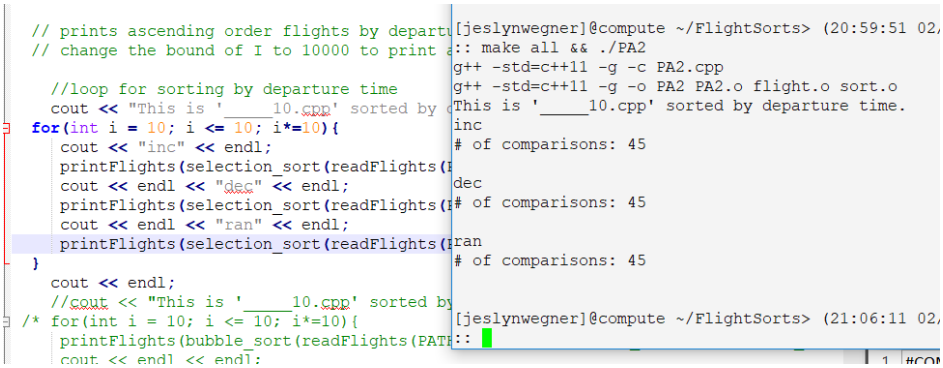
-

```
// prints ascending order flights by departu
// change the bound of I to 10000 to print a

  //loop for sorting by departure time
  cout << "This is '_____10.cpp' sorted by d
for(int i = 10; i <= 10; i*=10){
   cout << "inc" << endl;
   printFlights(bubble_sort(readFlights(PATH
   cout << endl << "dec" << endl;
   printFlights(bubble_sort(readFlights(PATH
   cout << endl << "ran" << endl;
   printFlights(bubble_sort(readFlights(PATH
}
  cout << endl;
  //cout << "This is '_____10.cpp' sorted by
/* for(int i = 10; i <= 10; i*=10){
   printFlights(bubble_sort(readFlights(PATH
   cout << endl << endl;
   printFlights(bubble_sort(readFlights(PATH + DESCEN + std::to_string(i) + FILE_FOR
   cout << endl << endl;
   printFlights(bubble_sort(readFlights(PATH + RAND + std::to_string(i) + FILE_FORMA
```

```
[jeslynwegner]@compute ~/FlightSorts> (20:59
:: make all && ./PA2
g++ -std=c++11 -g -c PA2.cpp
g++ -std=c++11 -g -o PA2 PA2.o flight.o sort
This is '_____10.cpp' sorted by departure ti
inc
# of comparisons: 9

dec
# of comparisons: 45

ran
# of comparisons: 42


[jeslynwegner]@compute ~/FlightSorts> (20:59
::
```

cases

-



5 Discussion & Conclusion

5.1 Discussion

Looking at each result in Table 4.1-1, each correspond to the complexities in Tables 3-1 and -2. That is to say, each one is within the expected amount of comparisons. For example, the bubble sort's best case (inc) for all n is less than or equal to n, while its average (rand) and worst (dec) cases fall under or equal to n^2 and above n. This is true for all three of the sorts, and thus matches the theoretical analyses we have learned previously.

5.2 Conclusion

Give your observations and conclusion. For instance, which sorting algorithm is more suitable for a specific input data? What factors can affect your experimental results?

Selection sort looks to be inferior to the other two sorts in all cases, so there would be few applications to use it in over the other two sorts. The only advantage it has is a lower amount of memory uses for its read and write, so it might be useful on systems with very limited resources. Past this, bubble sort is slightly less efficient than insertion sort, and certainly worse on memory usage, so insertion sort seems to be the best for this program. However, bubble sort is the best for simplicity if there aren't any memory constraints, so it might help slightly for maintainability. From the graphs, it can also be seen that all three of the algorithms look roughly the same in their worst case (or in selection sort's case, all of its cases), which matches the fact of all three algorithms sharing a worst-case runtime in big-O. Experimental results would most likely depend on the distance of each element from its proper location, but the upper and lower limits (at least for this implementation of each sort) is shown on the table. The increase of the size of the input affects the results on a larger scale, but that is just a regular scaling change.

In conclusion, this activity gave a good framework on comparing sorts and gives a much more visible examination on how big-O notation works. Though everyone in the group has encountered these sorting methods before, seeing the actual amount of comparison for each helps put into perspective what big-O means other than just a vague measure of runtime.