EECS 600 - High Performance Computing

Cal Al-Dhubaib, Assignment 2

Raw assignment files located at: @dhubaib on Github

Problem 1: Experiment with HPCC job submissions using sample file Timing2.java

```java
// Chris Fietkiewicz. For HW 2, Problem 1. Added argument passing.
public class Timing2 {
  public static void main(String[] args) {
    long n; // Number of iterations
    if (args.length >= 1) {
      n = Long.parseLong(args[0]);
    } else {
      n = 1000000L;
    }
    long startTime, stopTime; // For recording start/stop times
    long x = 0;
    for (int trial = 0; trial < 5; trial++) {
      startTime = System.currentTimeMillis();
      for (long i = 0; i < n; i++) {
        x = x + 1;
      }
      stopTime = System.currentTimeMillis();
      System.out.print(stopTime - startTime + "\t");
    }
  }
}
```

A) Modify the original file to print the average at the end of each run

```
// Chris Fietkiewicz. For HW 2, Problem 1. Added argument passing.
// Modified by Cal Al-Dhubaib
public class Timing2 {
  public static void main(String[] args) {
    long n; // Number of iterations
    if (args.length == 1) {
      n = Long.parseLong(args[0]);
    } else {
      n = 100000000L;
    }
    long startTime, stopTime; // For recording start/stop times
    long x = 0;
    long avgTime = 0;
    int trial;
    for (trial = 0; trial < 5; trial++) {
      startTime = System.currentTimeMillis();
      for (long i = 0; i < n; i++) {
        x = x + 1;
      }
      stopTime = System.currentTimeMillis();
      avgTime += (stopTime-startTime);
      System.out.print(stopTime - startTime + "\t");
    }
    System.out.print("Average: " + (float)avgTime/(float)trial + "\n");
  }
}
```

B) Run program on HPCC using at least three different loop sizes

The job was submitted on the HPCC using the slurm script timing_jv.slurm

```
#!/bin/bash
#SBATCH --output=timing_jv.txt

cp Timing2.java $PFSDIR/.
cd $PFSDIR

javac Timing2.java

java Timing2 100000000
java Timing2 1000000000
java Timing2 10000000000
```

This output was sent to timing_jv.txt

```
73      211     207     206     206     Average: 180.6
691     2064    2065    2058    2060    Average: 1787.6
6869    20597   20590   20588   20587   Average: 17846.2
```

C) Repeat problem with a different language

I decided to use Python for this, and the remaining problems as the repeated language.

```
import time
import sys

def timing(maxiter = 10000000, ntrial = 5):
    times = [] # To keep track of times
    for i in xrange(0,ntrial):
        x = 0
        start = time.time() # Start timer
        for j in xrange(0,maxiter):
            x = x + 1
        stop = time.time() # Stop timer
        times.append((stop-start)*1000)

    for i in range(0,len(times)):
        print (int)(times[i]), '\t',

    print 'Average: ', (int)(sum(times)/ntrial)

arg = [int(i) for i in sys.argv[1:]]
if(len(arg) > 0):
    timing(arg[0])
else:
    timing()
```

The submission script was modified slightly for Python

```
#!/bin/bash
#SBATCH --output=timing_py.txt
#SBATCH --mem=4g

cp Timing2.py $PFSDIR/.
cd $PFSDIR

module load intel
module load python

python Timing2.py 10000000
python Timing2.py 100000000
python Timing2.py 1000000000
```

This output was sent to timing_py.txt. Python seems to be more memory intensive than Java. It required more memory allocation and time to run a relatively smaller operation. (10e9 max iterations in python vs 10e10 max iterations in Java):

```
427     427     427     427     427     Average:  427
4273    4272    4273    4273    4273    Average:  4273
42716   42708   42714   42718   42721   Average:  42715
```

Problem 2: Measure and plot the average and standard deviation of quicksort runtime.

A) Modify submission from assignment 1 to collect 5 sample sizes for each array length

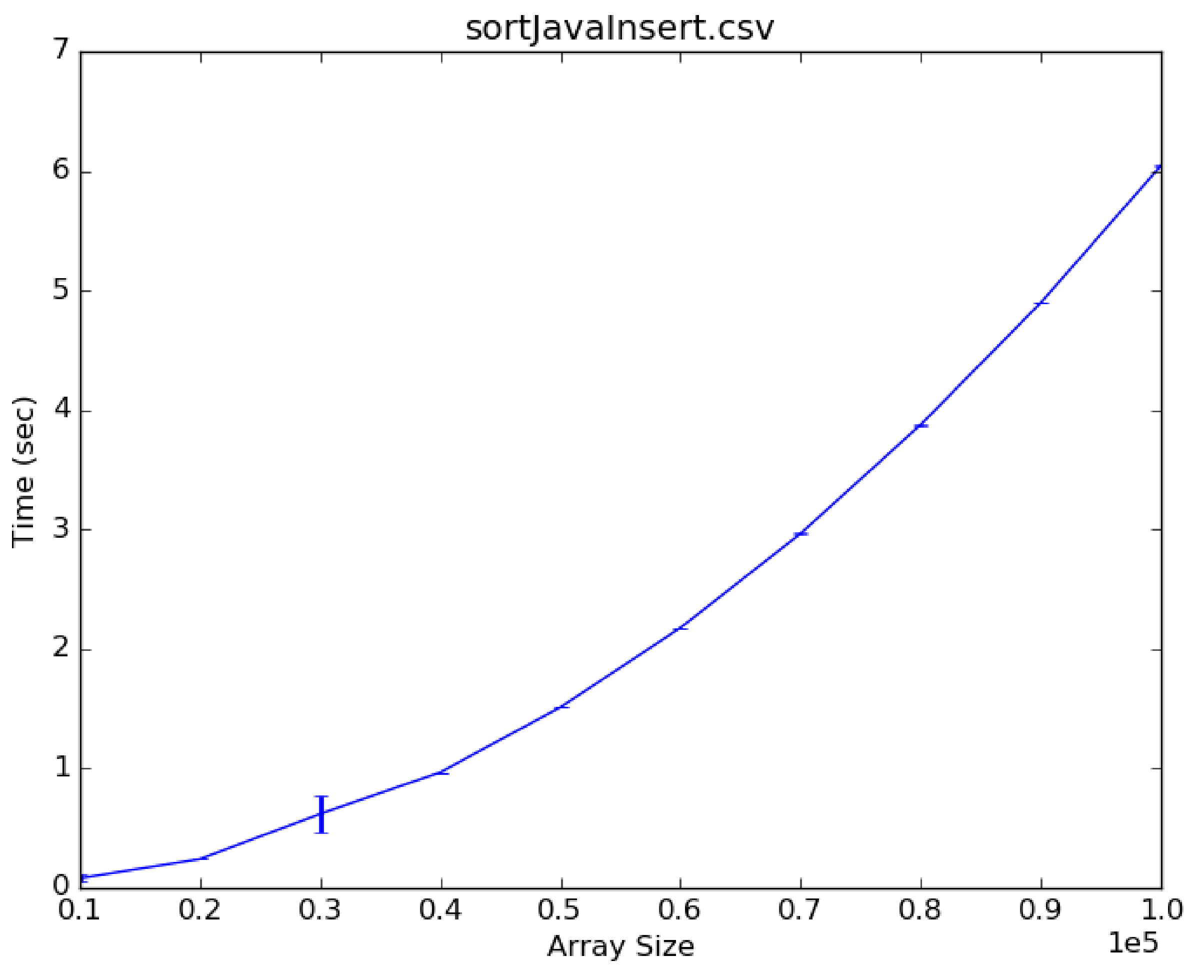
```
// Cal Al-Dhubaib, CWRU
// Assignment 2 - 2/1/16 (Modified from Assignment 1)

// Method to measuring performance of sorting algorithms by veco
```

```java
// Method to measuring performance of sorting algorithms by [illegible]

import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class sortTime {
  public static void main(String[] args) {

    long startTime, stopTime; // For recording start/stop times
    int baseSize;
    int nTrials = 5;

    // Set up sequence starting size
    if (args.length == 1) {
      baseSize = Integer.parseInt(args[0]);
    } else {
      baseSize = 1000000;
    }

    // Set up arrays to collect times
    int[] trialSizes = new int[10];
    float[][] sortTimes = new float[10][nTrials]; // Multiple times for each array size
    trialSizes[0] = (int)baseSize;

    // Run test case on various-sized arrays
    for (int trial = 0; trial < sortTimes.length; trial++){
      trialSizes[trial] = baseSize*(trial+1); // Fill array of sequence

      int maxNum = trialSizes[trial];
      int[] seq = new int[maxNum]; // Initialize the array

      // Build various reverse-ordered sequences
      for (int j = 0; j < seq.length; j++){
        seq[j] = maxNum--; // Fill with reverse order
      }

      // Run sort quick sort test on sequence here and collect 5 run-times
      for (int j = 0; j < sortTimes[0].length; j++){
        int[] seq1 = seq.clone();

        startTime = System.currentTimeMillis();
        if (args.length == 2){
          InsertionSort.insertionSort(seq1);
        }
        else{ // Use quicksort unless otherwise specified
          QuickSort.quickSort(seq1,0,seq1.length - 1);
        }
        stopTime = System.currentTimeMillis();

        sortTimes[trial][j] = (float)(stopTime - startTime);
        if(args.length == 2){
          System.out.print("Insertion Sort ");
        }else{
          System.out.print("Quick Sort ");
        }
        System.out.print("["+trial+", " + j + "]: " + (stopTime - startTime) +
"\n");
      }
    }

    // Output timing results to file
    File f = null;
    try{
      if(args.length == 2){
        f = new File("sortJavaInsert.csv");
      }else{
        f = new File("sortJavaQuick.csv");
      }
      f.createNewFile();

      FileOutputStream fis = new FileOutputStream(f);
      PrintStream out = new PrintStream(fis);
      System.out.print("\n");
      System.setOut(out);

      // First row in each file = sequence length
      for (int j = 0; j < trialSizes.length; j++){
        System.out.print(trialSizes[j]);
        if(j < trialSizes.length - 1){
          System.out.print(",");
        }
      }

      System.out.print("\n");

      // Second row in each file = sorting time
      for (int trial = 0; trial < nTrials; trial++){
        for (int j = 0; j < sortTimes.length; j++){
          System.out.print(sortTimes[j][trial]);

          if(j < sortTimes.length - 1){
            System.out.print(",");
          }
        }
        System.out.print("\n");
      }

      // Close print streams
      out.close();

    }catch(Exception e){
    }
  }
}
```

B) Submit an HPCC job and present a plot as output

insert_jv.slurm

```
#!/bin/bash
#SBATCH --output=insert_jv.txt
#SBATCH --mem=8g

cp plot_png.py $PFSDIR/.
cp sortTime.java $PFSDIR/.
cp QuickSort.java $PFSDIR/.
cp InsertionSort.java $PFSDIR/.

cd $PFSDIR

module load intel
module load python

javac InsertionSort.java
javac QuickSort.java
javac sortTime.java

java sortTime 10000 insert

python plot_png.py sortJavaInsert.csv

cp -u *.csv $SLURM_SUBMIT_DIR/.
cp -u *.png $SLURM_SUBMIT_DIR/.
```



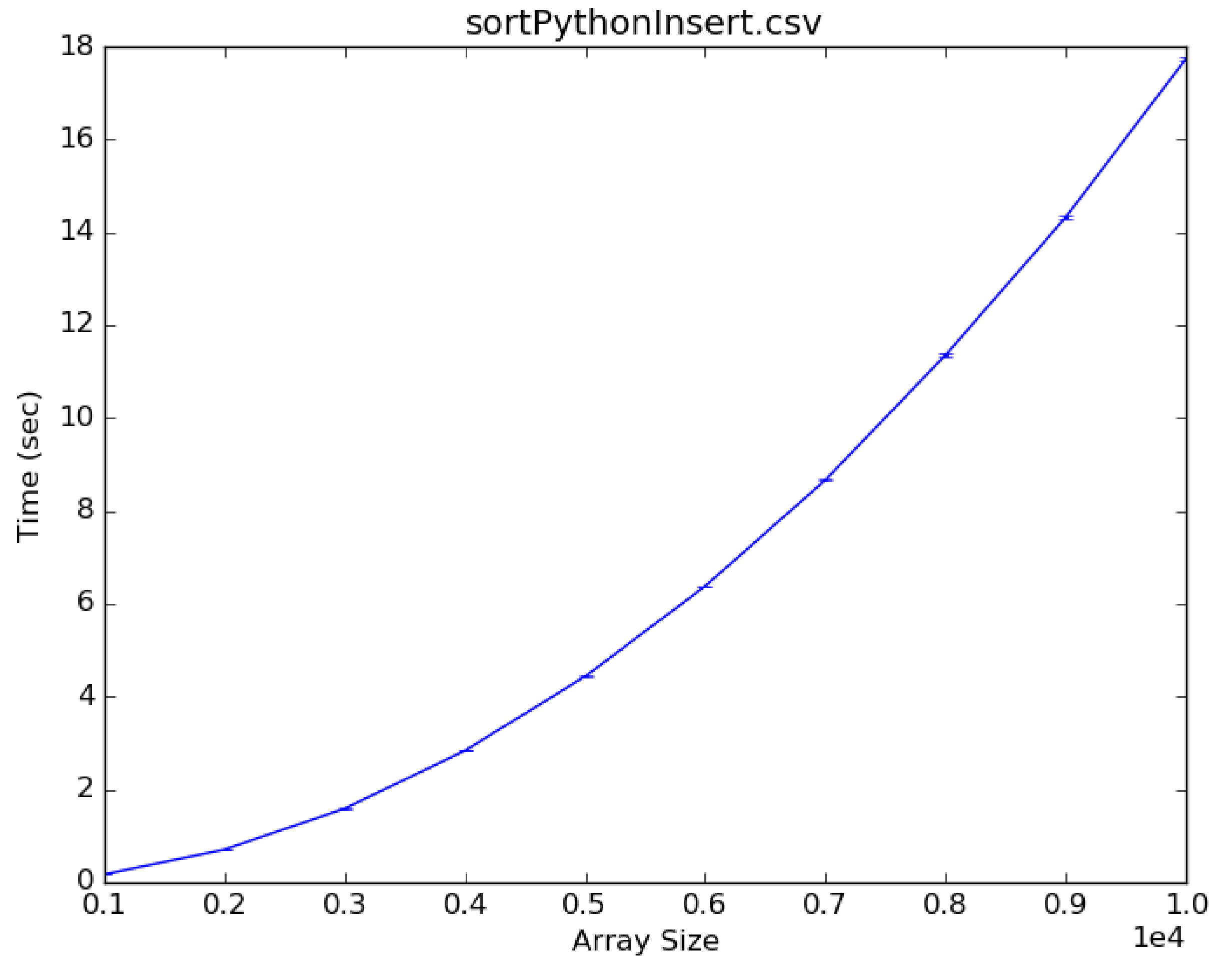
C) Repeat problem in a different language (Python)

```
import time
import numpy
import csv
import sys

## Test for sort functions
def test(sortFunc, ntest = 10, baseSize = int(1e3)):
    arrSize = range(baseSize, baseSize*10 + 1, baseSize*(10/ntest))
    times = [[] for x in range(5)]
    i = 0

    # Create test arrays & sort
    for j in range(0,5):
```

```
    for j in range(0,3):

        for size in arrSize:
            testData = range(size,0,-1) # Worst case = reverse ordered

            start = time.time() # Start timer
            sortFunc(testData) # Run test function
            stop = time.time() # Stop timer

            times[i].append(round((stop-start)*1000,0))

            print "Trial [", i+1, ", ",j, "]: ", times[i],"ms \n"

        i+=1;


    print " "
    return arrSize, times

# Take args here
if(len(sys.argv) > 1):
    baseSize = int(sys.argv[1])
else:
    baseSize = int(1e3)

if(len(sys.argv) > 2):
    sizes, times = test(insertionSort, baseSize = baseSize)
else:
    sizes, times = test(quickSort, baseSize = baseSize)

if(len(sys.argv) > 2):
    f = open('sortPythonInsert.csv','wb')
else:
    f = open('sortPythonQuick.csv','wb')

writer = csv.writer(f)
writer = csv.writer(f)
writer.writerow(sizes)
writer.writerows(times)

f.close()
```

insert_py.slurm

```
#!/bin/bash
#SBATCH --output=insert_py.txt
#SBATCH --mem=8g

cp sort.py $PFSDIR/.
cp plot_png.py $PFSDIR/.

cd $PFSDIR

module load intel
module load python

python sort.py 1000 insert

python plot_png.py sortPythonInsert.csv

cp -u *.csv $SLURM_SUBMIT_DIR/.
cp -u *.png $SLURM_SUBMIT_DIR/.
```



Consistent with problem 1, Python runs slower by a factor ~ 100

Problem 3: Repeat problem 2 with Quick Sort

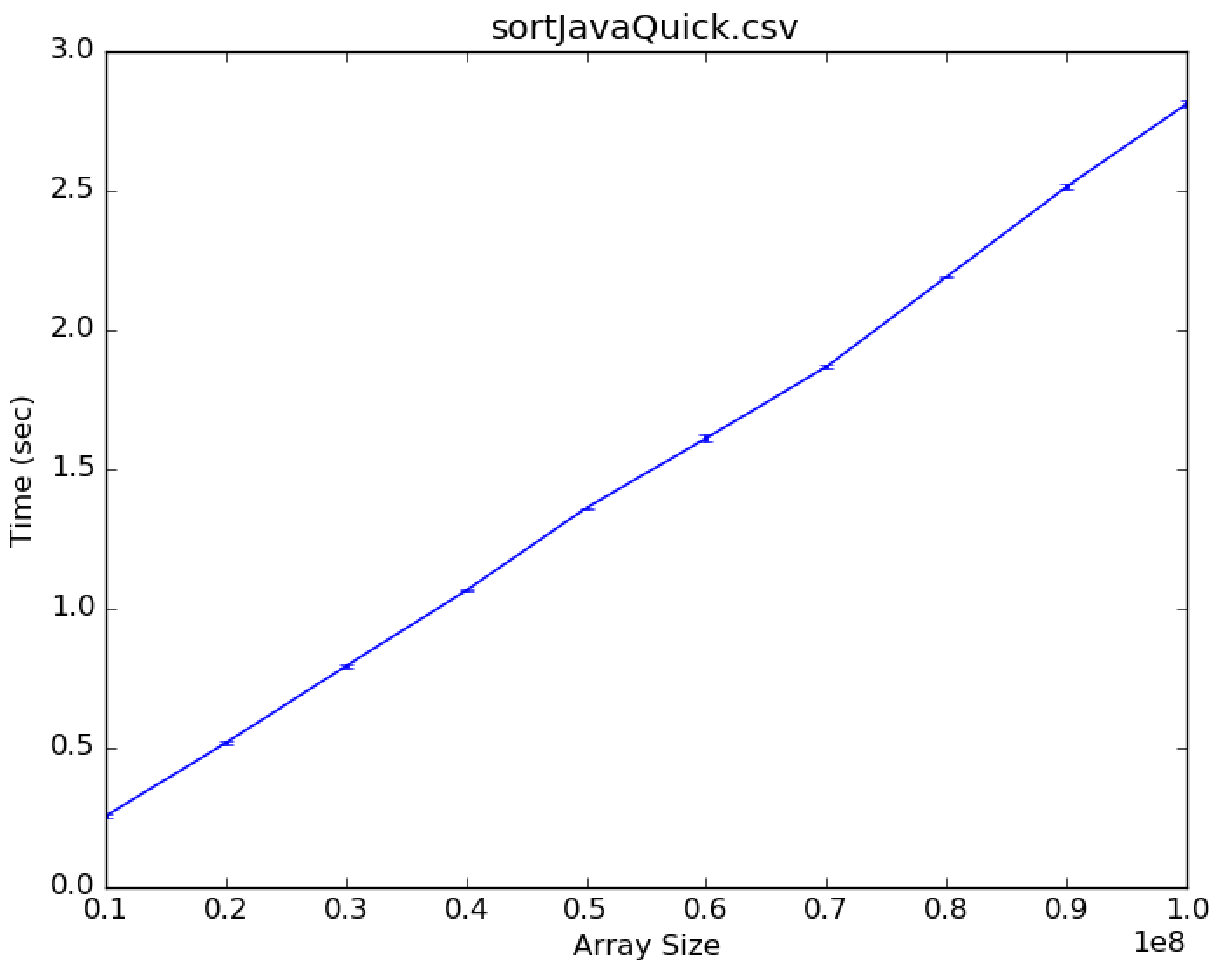
A) Modify submission from assignment 1 to collect 5 sample sizes for each array length

See the code in 2.A above. Programmed to work for either insertion or quick sort.

B) Submit an HPCC job and present a plot as output

quick_jv.slurm

```bash
#!/bin/bash
#SBATCH --output=quick_jv.txt
#SBATCH --mem=8g

cp sortTime.java $PFSDIR/.
cp QuickSort.java $PFSDIR/.
cp InsertionSort.java $PFSDIR/.
cp plot_png.py $PFSDIR/.

cd $PFSDIR

module load intel
module load python

javac QuickSort.java
javac sortTime.java

java sortTime 10000000

python plot_png.py sortJavaQuick.csv

cp -u *.png $SLURM_SUBMIT_DIR/.
cp -u *.csv $SLURM_SUBMIT_DIR/.
```

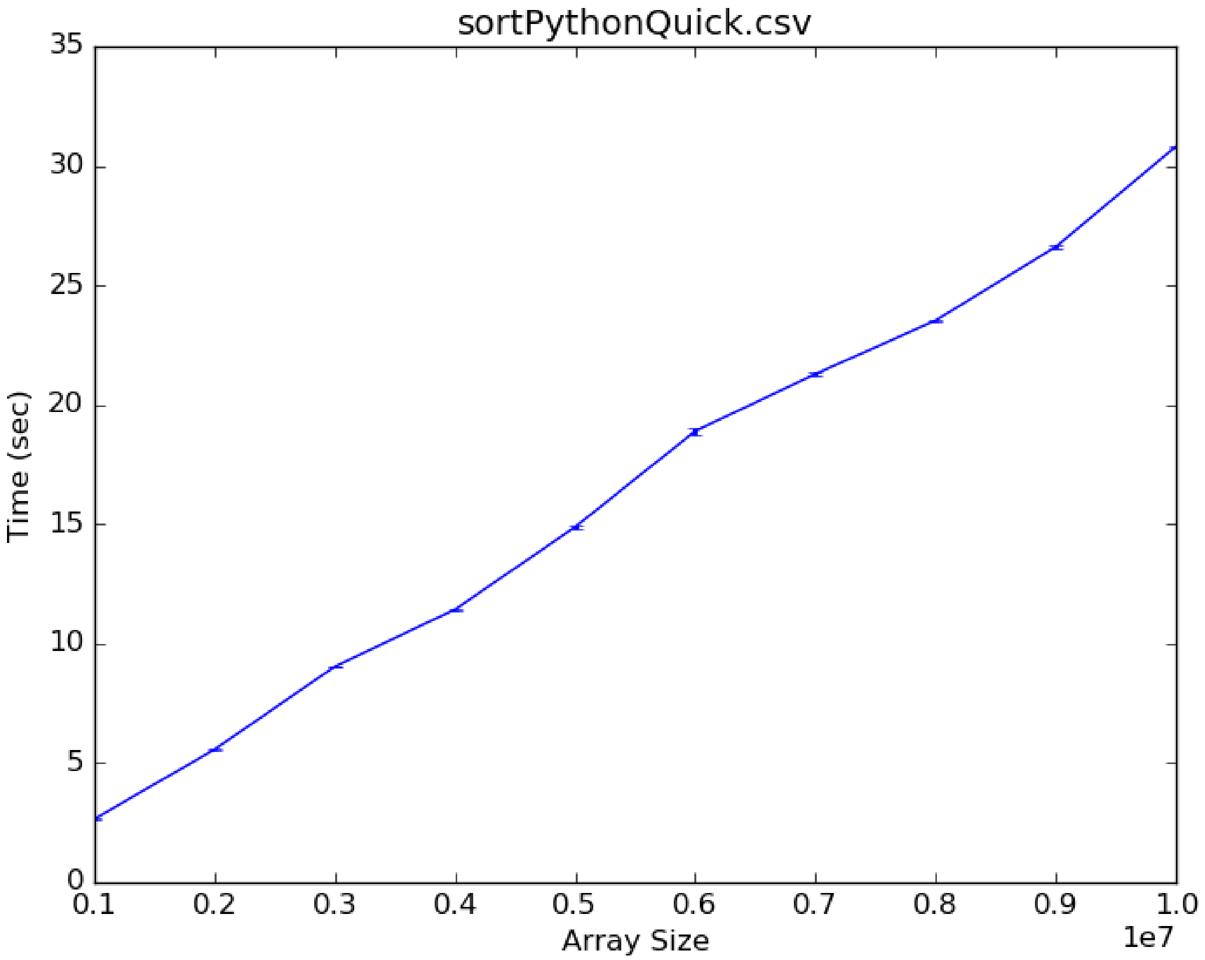


C) Repeat problem in a different language (Python)

See the code in 2.C above. Programmed to work for either insertion or quick sort.

quick_py.slurm

```bash
#!/bin/bash
#SBATCH --output=quick_py.txt
#SBATCH --mem=8g

cp sort.py $PFSDIR/.
cp plot_png.py $PFSDIR/.

cd $PFSDIR

module load intel
module load python

python sort.py 1000000

python plot_png.py sortPythonQuick.csv

cp -u *.csv $SLURM_SUBMIT_DIR/.
cp -u *.png $SLURM_SUBMIT_DIR/.
```



Consistent with problem 1, Python runs slower by a factor ~ 100