Algorithm-testing exercise

The goal of this exercise is to recognize predicted run-time behaviour for some of the sorting algorithms we have studied: *Insertion-sort, Heapsort, Quicksort.* It may help increase appreciation for good algorithms.

Given are 4 types of data: (pseudo)random(**r**), sorted(**s**), reverse sorted(**i**), and almost sorted(**a**), and 4 executables (one for each algorithm, and makedata). You need to produce 4 datafiles of sizes $n, 2n, 4n, 8n$ for each datatype using **makedata** and run them through all 3 sorting programs, while measuring execution time. Each code will print the number of comparisons performed at the end of execution. You will need to put the results in a table and recognize then behaviour predicted by the analysis.

Steps:

1. *Build the codes*
   - copy the file **people.cs.uchicago.edu/ brady/CSPP570/homework/sorts.tar** to your working directory (it may be worth making a directory specifically to be used with this application)
   - run the command: **tar -xf sorts.tar**
   - run the command: **make** which will build the 4 executables: **heapsort, insertsort, makedata, quicksort**
2. *Make data*
   - Choose an n ($10000 \leq n \leq 50000$ depending on how fast your machine is) and type: **makedata k datatype filename** for each of $k = n, 2n, 4n, 8n$, and datatypes **r, s, i, a**. Give meaningful names to the files.
   - Usage example: **makedata 100 a a_100_as.dat** will produce an almost sorted array of 100 and store it in the file **a_100_as.dat**
3. Run all codes on each datafile and record data in 4 tables, one for each data type (see a model below).

- For anything that takes longer than 2 minutes don't bother; just mark this in the table.
- Usage and example: **time quicksort a_100_as.dat** will sort the array and output the number of comparisons performed. The **time** command will output the execution times in seconds. Record the first one of the numbers (user time).
- To see the actual results of the sorting add a **w** as an option at the end of the command line. Do not do this when measuring time and with large datafiles, i/o to the screen takes a long time. example: **quicksort a_100_as.dat w**.

4. *Interpret data*

   - Verify in detail that predicted behavior takes place, i.e., if predicted behavior is : #of comparisons $\approx Cn^2$, verify on the data that such a C exists (find its value).
   - Example: Quicksort on sorted data is expected to require Cn^2 work. What we do is take $C = T(n)/n^2$ and put the corresponding constant below. Here's how my table looks:

n	100	200	400	800	1600
$T(n)$	5346	20696	81396	322796	1.2856e+06
C	0.5346	0.5174	0.508725	0.504368	0.5021875

   We notice a certain convergence in the constants which shows that the data fits with the formula $T(n) \approx Cn^2$, and we can certainly say $T(n) \leq 0.5022n^2$ for $n \geq 1600$, but we believe $T(n) \approx 0.5022n^2 \approx n^2/2$ is a good guess for quicksort on this data type.

5. Given your results from step 3, predict the behavior of each code on two data files:

   (a) (our) "random" data with $n = 1,000,000 (10^6)$

   (b) (our) almost sorted data with $n = 10^6$

   Use your predictions to rank the algorithms on each type of data by running time, then run the fastest for each case and compare your prediction to the actual result, both in terms of running time and number of comparisons.

6. How relevant is the # of comparisons to the actual runtime? (Check for proportionality)

Turn in: the tables and the answers to 4, 5, 6