Dustin Cook

CSC 372 Analysis of Algorithms, Lisa Rebenitsch

Final Project

12/4/2018

Fun-Sort, Sorting Unique SQL ID’s, C++

I implemented this algorithm in C++ however, I wrote a test script for it in python3. Usage for the program and correctness.py can be found near the end of this document.

You can binary search to find bad pairs in an array. The algorithm then performs insertions guided by the binary search on the unsorted array. It can be assumed that the array of integers given to Fun-Sort is an array of SQL ID’s.

# Fun-Sort runs in *n* phases. For phase *i*=1…*n*, binary search for the current content *ai* of cell *i* until the search is successful and “finds” the element in cell *i*. If the search fails, we find an index *h* such that *ah*−1<*ai*<*ah*. We then either swap the bad pair (*i*,*h*−1) if *i*<*h*−1, or the bad pair (*h*,*i*) if *h*<*i*. The phase continues with a search for the new content of cell *i*.

# Note: This algorithm can be modified for duplicate values; however, the authors did not define it for arrays containing duplicates.

# Where It Is Used

# Toy algorithm for computer scientists with no practical industry use.

# Theoretical Correctness

# The first i – 1 cells are ordered correctly at beginning of phase i. If a search fails, we are not done with phase yet, and it returns an index *h* ≠ *i.* If h > i, the subsequent swap of (i, h-1) does not affect the sordidness of the first i cells. If h < i then we swap (h, i). Since binary search guarantees *ah*−1<*ai*<*ah,* this swap also maintains the sortedness of the first i – 1 cells. If the search for ai is successful (and phase i ends) then we have ai-1 < ai, and the first i cells are sorted as claimed.

# Run time

# Each binary search takes time O(logn). There are n successful searches, one per phase, and there are at most F = (n2) unsuccessful searches. Thus, the worst case time is O ((n + F) logn) = O(n2logn).

# Built-in Code Correctness Tests

# To use the built-in tests, compile with make debug. (Or use your C++ compiler with the -DDEBUG tag).

# Example with make:

# 

# You can also compile with -DTIMED tag for an output of microseconds

# With the given array.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Input | Actual output |
| Case 1 | Small positive distinct values | 1 0 2 3 | 0 1 2 3 pass |
| Case 2 | Large distinct values | 100000 200000 400000 -500000 | -500000 100000 200000 400000 pass |
| Case 3 | Small repeating values | 2 2 3 1 | 1 2 3 2 FAIL |

# References

Fun-Sort. (2004, June 08). Retrieved December 9, 2018, from https://www.sciencedirect.com/science/article/pii/S0166218X04001131

# Build / Usage

You start by compiling fun\_sort.cpp. No executable name is required **“a.out” is fine and is what the python script is going to be looking for** and “a.out” is what the gnu g++ compiler defaults to when given no name. There is a Makefile provided in the project directory that has:

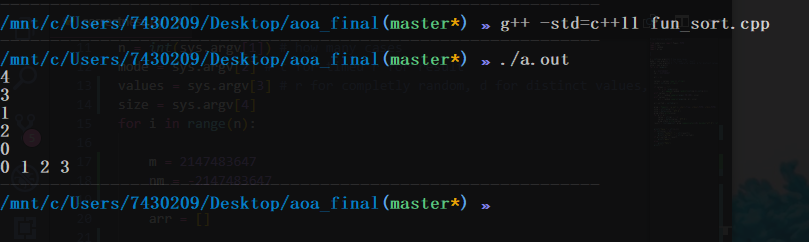
* make – Compiles without debug flag, the algorithm will return the array in sorted order.
* make debug – Compiles for built in tests, but its recommended to use correctness.py.
* make timed – Compiles for timing. Fun-Sort will return time in microseconds to sort the given array. If you want to use G++ instead of make, compile with -DTIMED tag.

**Input:**

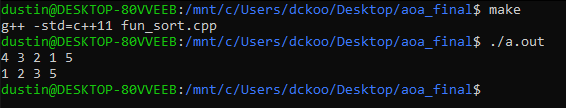
After you compile the program with a C++ compiler and run it, it will wait for input. It first expects an integer representing the problem size: N. It will then expect you to enter N integers separated by newlines or whitespace.

**Output:**

The numbers given in sorted order.

**Example with g++:**

**Example with make:**



Run requirements:

* Python3 to run correctness.py.
* A C++ compiler to compile fun\_sort.cpp.
* Make if you wish to use “make” to compile.
* C++ 11.

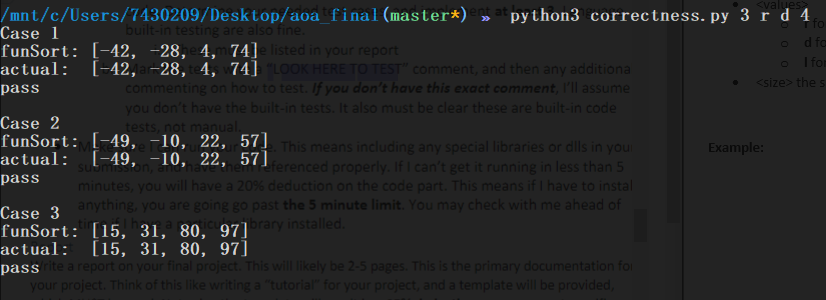
correctness.py

If you have make installed, correctness.py will compile fun\_sort.cpp for you**. If you don’t have make installed, you must compile fun\_sort.cpp yourself**. Correctness.py can run either value tests or timed tests. If you run it for value tests you must specify a size, but if you run it for timed tests an array size is not required because it will increase the array size per test case. If ran for values it tests if fun\_sort sorts the values correctly, if ran for timed, it prints to run\_time.csv **<array size, microseconds> in the current working directory**. I do apologize for the script not having a friendlier user interface, but I will try to make it simple.

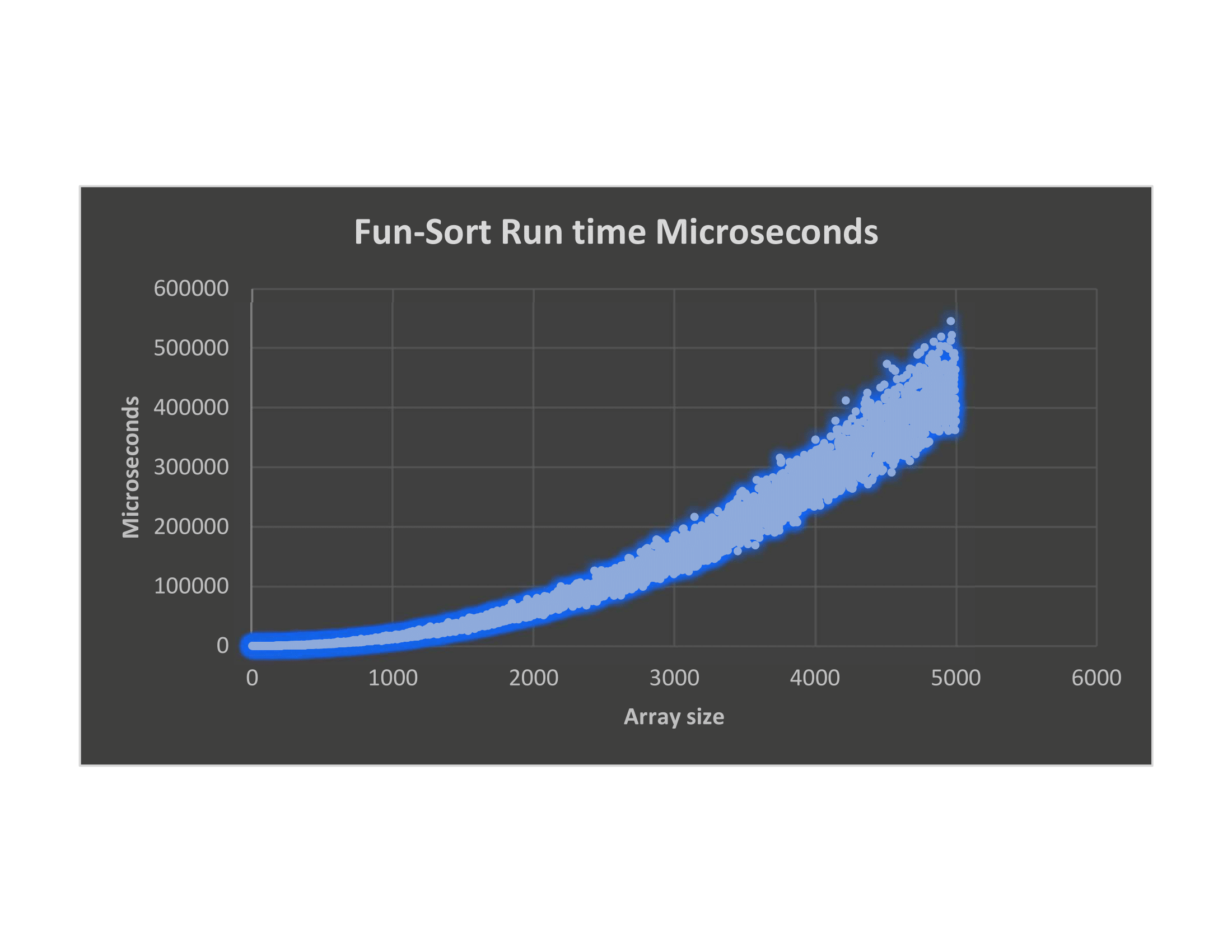
**correctness.py usage**

python3 correctness.py <n> <mode> <values> <size>

* <n> Number of cases
* <mode> **r** for value test, **t** for time tests
* <values>
  + **r** for random (possible duplicates) [-50, 50]
  + **d** for small distinct [-size, size]
  + **l** for large distinct [-2147483647, 2147483647]
* <size> the size of the array per test case, if size > 20, it will not print the two arrays to be viewed. **If you run with t for <mode>, you do not have to define this.**

**Example: Note: The script is case sensitive, and the order of arguments matters.**

**The following plot was generated with ./correctness.py 5000 t d**

**Some observations: the algorithm is chaotic.**

**It appears to be giving us logarithmic behavior even though its theoretically n2logn.**