Computational Optimization
ISE 407

Lecture 10

Dr. Ted Ralphs

Reading for This Lecture

- Assessing the Effectiveness of (Parallel) Branch-and-Bound Algorithms
- A Theoretician's Guide to the Experimental Analysis of Algorithms
- Statistical Analysis of Computational Tests of Algorithms and Heuristics

Empirical Analysis of Algorithms

- In practice, we will often need to resort to empirical rather than theoretical analysis to compare algorithms.
  - We may want to know something about effectiveness of the algorithm "on average" for real instances.
  - Our model of computation may not capture important effects of the hardware architecture that arise in practice.
  - There may be implementational details that affect constant factors and are not captured by asymptotic analysis.
- For this purpose, we need a methodology for comparing algorithms based on real-world performance.

Exact Versus Heuristic Algorithms

- In optimization, an "exact" algorithm is one that outputs a result (typically a solution) *and* a proof (i.e., a certificate).
- The proof is usually given in terms of primal and dual solutions/bounds.
- Because of numerical issues, it is usually not feasible to get "exact" solutions.
- Nevertheless, we can define termination criteria in terms of the "primal-dual gap" or some other criteria related to accuracy.
- The important thing is that the criteria is well-defined and *independent* of the algorithm.
- The methodology we describe is focused on exact algorithms having such well-defined termination criteria.
- This ensures comparability of results.
- Comparison of heuristic algorithms is much different and we won't discuss that.

Issues to Consider

- Empirical analysis introduces many more factors that need to be controlled for in some way.
  - Test platform (hardware, language, compiler)
  - Measures of effectiveness (what to compare)
  - Benchmark test set (what instances to test on)
  - Algorithmic parameters
  - Implementational details
  - Variability and non-deterministic behavior
  - Generalizability of results
  - Reproducibility
- It is not at all obvious how to perform a rigorous analysis in the presence of so many factors.
- Practical considerations prevent complete testing.

Assessing "Effectiveness"

- What do we mean by "effectiveness"?
  - For the time being, we focus on sequential algorithms.
  - We'll define effectiveness of sequential algorithms in terms of *efficiency* of *resource consumption*.
- What resources are we talking about?
  - "Time"
  - Memory/Space
  - Number of cores (in the parallel case)
  - Power
  - Operations
  - ??
- In the case of parallel algorithms, we consider *tradeoffs* between the resources (we'll discuss this in the next lecture).

Formal Definition

Definition 1. *A* resource *is an auxiliary input, some measurable quantity of which is required to produce the result of a computation.*

Definition 2. *A* measure of efficiency *for a given benchmark computation is the amount of one chosen resource that is required to perform that computation, with the level of all other resources fixed.*

- Note that we measure efficiency with respect to some particular benchmark computation.
- The output of this computation should be well-defined in order for comparison to be sensible.
- This kind of analysis is most appropriate for "exact" algorithms.

Empirical Resource Consumption Distribution Functions

- Empirical analysis can be viewed as a method of estimate the probability distribution of resource consumption of an algorithm.
  - Resource consumption is just an abstraction of the concept of "running time" that we discussed earlier.
  - Resource consumption function can be thought of as a random variable over the space of instances.
  - In contrast to the theoretical running time function, we may consider the resource consumption over a set of instances of a fixed size.
- The analysis is usually done over a "class" of instances.
- For this to be a well-defined concept, we need to be able to sample from the distribution of instances in the class.
- In practice, we may not know either the true distribution.
- We typically assume that the distribution on the instances is uniform.
- There are many unknowns and we need to customize our testing based on the situation.

Empirical CDF Example

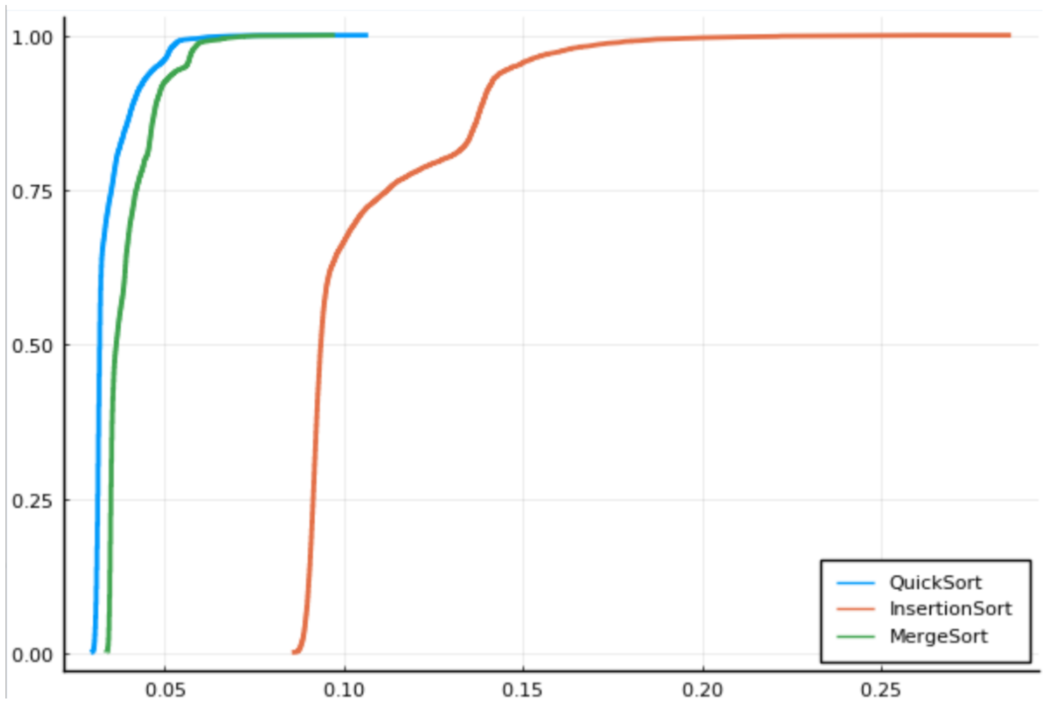

Figure 1: Empirical CDF for 10K samples of sorting algorithms

- Each sample here is a different randomly generated list.
- Note that *different* random samples were used in generating each eCDF.
- One could argue that we should use the same sample.

Measuring Time

- In the remainder of the lecture, we focus primarily on time as the resource of interest.
- There are three relevant measures of time we can measure.
  - *User time* measures the amount of time (number of cycles taken by a process in "user mode."
  - *System time* is the time taken by the kernel executing on behalf of the process.
  - *Wallclock time* is the total "real" time taken to execute the process.
- Generally speaking, user time is the most relevant, though it ignores some important operations (I/O, etc.).
- Wallclock time should be used cautiously/sparingly, but may be necessary for assessment of parallel codes,

Dealing with Stochasticity

- Measurement of empirical running times is noisy in general for multiple reasons.
- For the noise that occurs in performing deterministic experiments, replications help smooth out the results.
- Here is the same CDF as before, but with 10 replications of each sample.

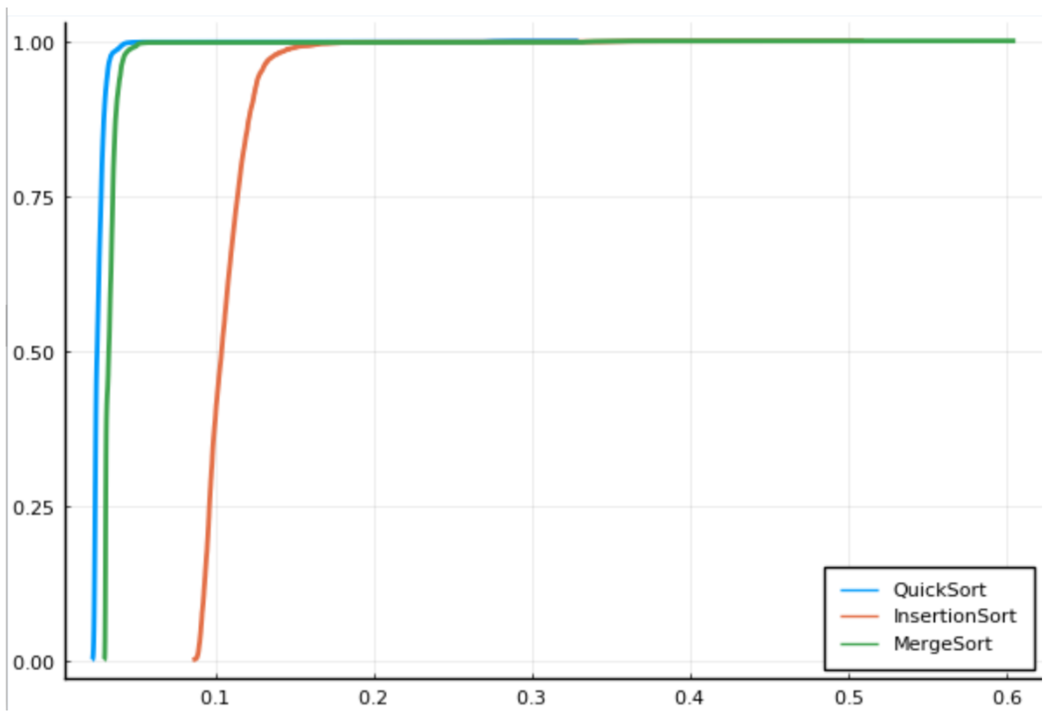

Figure 2: Empirical CDF for 10K samples with 10 replications per sample

Test Set

- The test set you use largely determines the validity of your results.
- The instances must be chosen carefully in order to allow proper conclusions to be drawn.
- Generally speaking, the test set should be a "representative sample" of the overall class of instances.
- This is difficult to achieve and it is even difficult to know whether we have achieved it or not.
- We may need to pay close attention to their size, inherent difficulty, and other important structural properties.
- This is especially important if we are trying to distinguish among multiple algorithms.
- Example: Sorting

Example: Insertion Sort

```
def insertion_sort(l):
    for i in range(1, len(l)):
        save = l[i]
        j = i
        while j > 0 and l[j - 1] > save:
            l[j] = l[j - 1]
            j -= 1
        l[j] = save
```

- As an example of the importance of test sets, consider insertion sort.
- What is the maximum number of steps the insertion sort algorithm can take?
- On what kinds of inputs is the worst-case behavior observed?
- What is the "best" case?
- On what kinds of inputs is this best case observed?
- Do you think that empirical analysis based on random instance generation will tell us what we really want to know about this algorithm?

Results with Pre-sorted Input

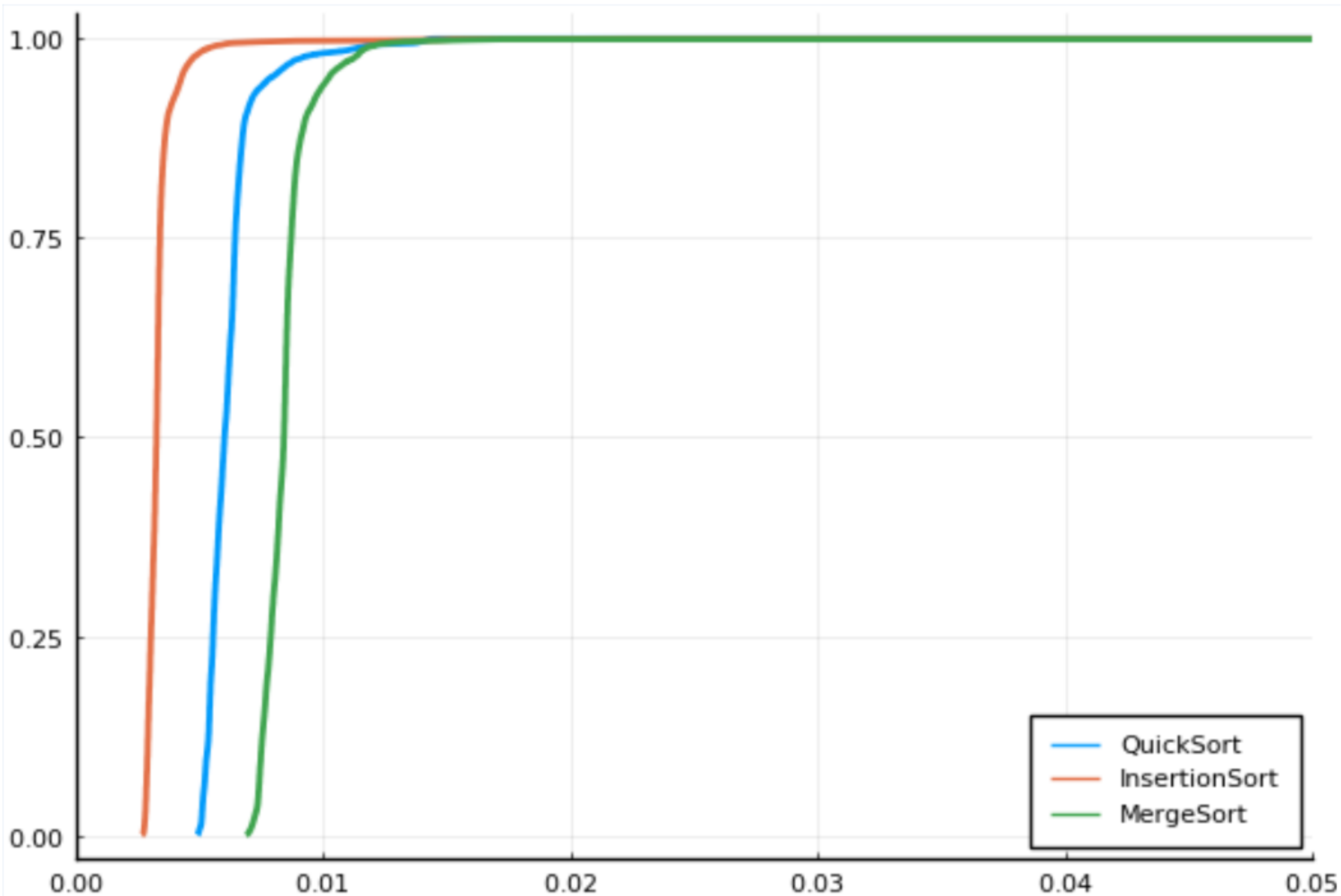

Figure 3: Empirical CDF for already sorted input

Results with Reverse Sorted Input

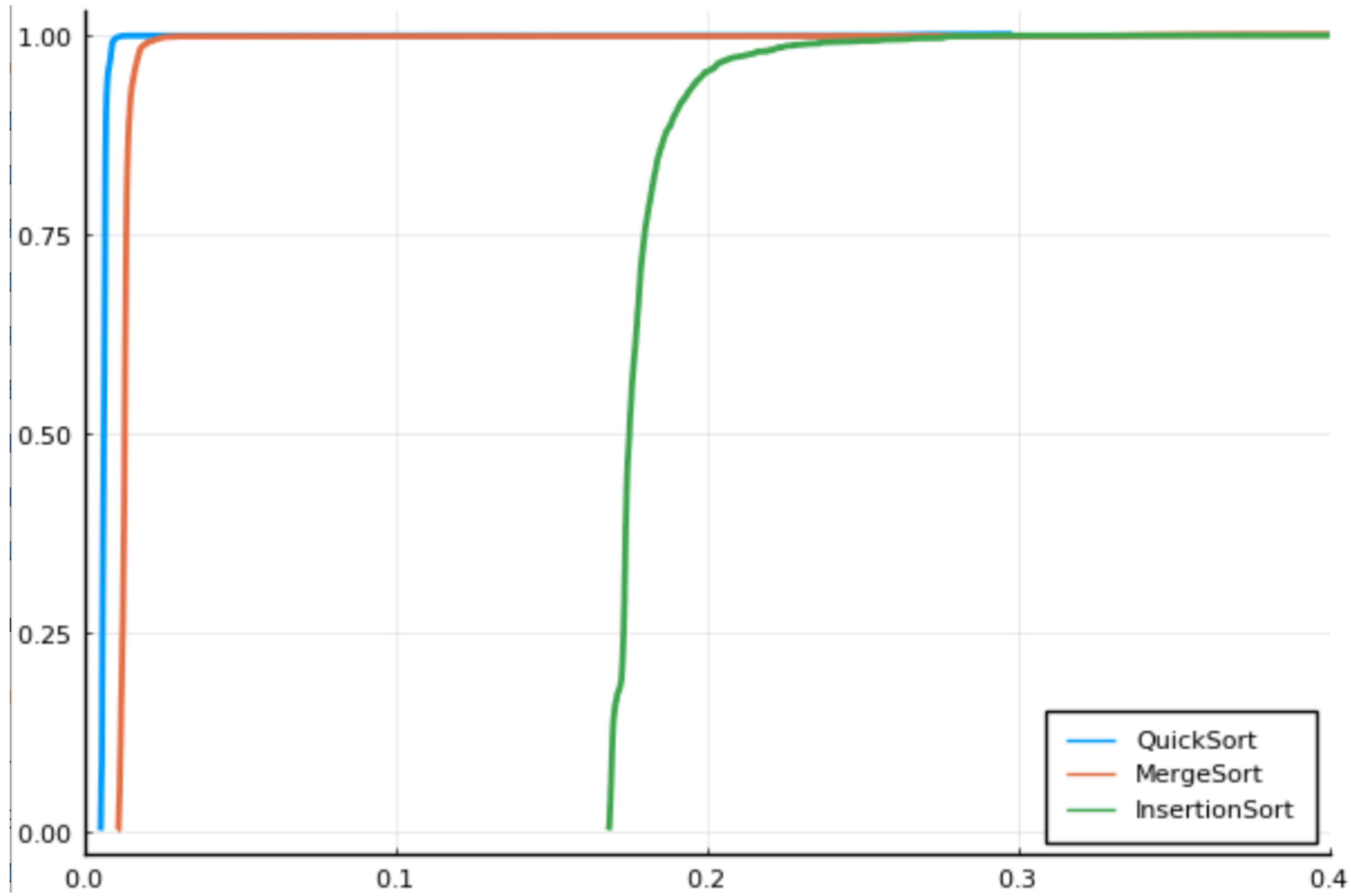

Figure 4: Empirical CDF for already sorted input

Example: Navigating a Maze

- In this example, we show the empirical distribution function of number of steps needed to navigate a random maze.
- Note the strong dependence on density.

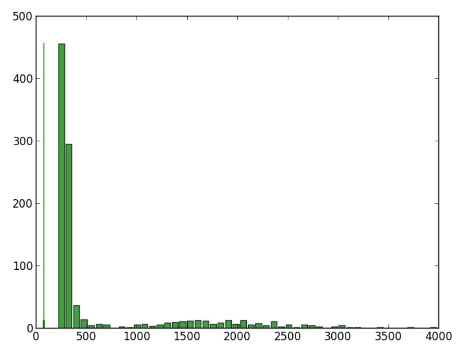

Figure 5: Size 100, Density 20%

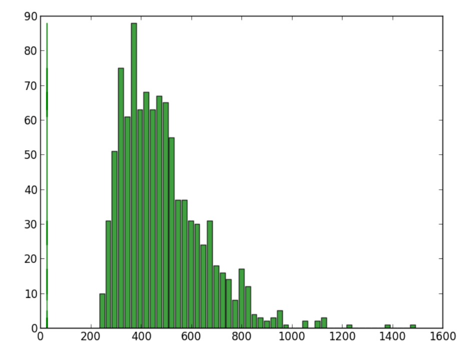

Figure 6: Size 100, Density 50%

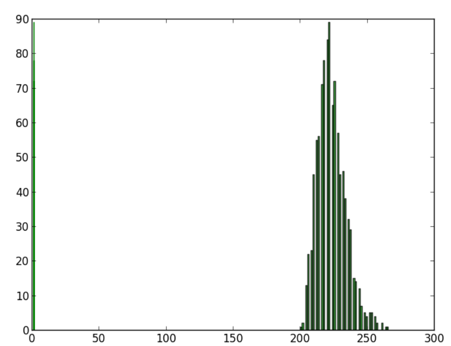

Figure 7: Size 100, Density 80%

Randomized Instance Generation

- In general, instances used for testing should be representative of what will be encountered when the algorithm is deployed.
- A test set drawn randomly from a distribution representing the true distribution of instances in the "real world" is ideal.
- However, the "real-world" distribution of instances is rarely known with any certainty.
- In some cases, it is possible to devise random generators for instances that produce good test cases.
- In most cases, randomized instances are not appropriate because they don't represent the true nature of instances arising in practice.

Performing Experiments

- In addition to choosing the test set and the measure of efficiency, we must also determine other experimental parameters.
  - Resource limits (time, memory, etc.)
  - Parameter settings
  - Replications
- All efforts should be made to eliminate confounding sources of variability by running experiments in a "sandbox" if possible (e.g., using `cset`).
- Roughly speaking, there are three steps in the process.
  - Construct a test set.
  - Measure resource consumption for each single instance with each algorithm individually (with appropriate replications).
  - Construct an empirical probability distribution from the data.
  - Compare the distribution and draw conclusions.

Illustrating Concepts: BenchmarkTools in Julia

- Julia has a package specifically designed for doing rigorous benchmarking.

```
julia> t = @benchmark sum(rand(1000))
BenchmarkTools.Trial:
  memory estimate:  7.94 KiB
  allocs estimate:  1
  --------------
  minimum time:     1.210 μs (0.00% GC)
  median time:      1.500 μs (0.00% GC)
  mean time:        3.319 μs (8.28% GC)
  maximum time:     248.330 μs (93.73% GC)
  --------------
  samples:          10000
  evals/sample:     10
```

- Here, we are apparently measuring the time to sum 100 random numbers.
- Notice, however, that we are also including the time to do the memory allocation and generate the list.
- The garbage collector is also running in some iterations.

Benchmarking Parameters

- Parameters
  - `samples`: Number of experiments, number of instances to run.
  - `evals`: Number of times to replicate each experiment.
  - `seconds`: Total time budget for benchmarking.
  - `overhead`: Estimate of looping overhead to be deducted from time.
  - `gctrial`: Whether to do garbage collection before each trial.
  - `gcsample`: Whether to do garbage collection before each sample.
  - `time_tolerance`: Tolerance for delcaring a regression.
  - `memory_tolerance`: Tolerance for delcaring a regression.
- Overall process
  - Define the benchmark (`@benchmarkable`): Generate code from macro.
  - Tune parameters (`tune!()`): Mainly to determine `evals` by measuring time for one sample—shorter time means more evals..
  - Run experiments (`run`): Do warm-up and then sample.
- In most case, you should set all parameters yourself.
- Beware that 5 seconds is the default time budget!

Garbage Collection and Interpolation

```
julia>  = @benchmark sum(rand(1000)) gcsample=true
BenchmarkTools.Trial:
  memory estimate:  7.94 KiB
  allocs estimate:  1
  --------------
  minimum time:     2.700 μs (0.00% GC)
  median time:      3.056 μs (0.00% GC)
  mean time:        3.596 μs (0.00% GC)
  maximum time:     7.600 μs (0.00% GC)
  --------------
  samples:          20
  evals/sample:     9
```

Setting `gcsample=true` seems to increase the running time for some reason.

```
julia> @benchmark sum($(rand(1000)))
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     70.270 ns (0.00% GC)
  median time:      70.686 ns (0.00% GC)
  mean time:        77.540 ns (0.00% GC)
  maximum time:     205.821 ns (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     962
```

The reason running times are so fast is because with interpolation, the sum is just a constant and the compiler optimizes away the whole computation.

Setup and Teardown

```
julia> @benchmark sort(x) setup=(x = rand(1000)) evals=10 samples=10000
BenchmarkTools.Trial:
  memory estimate:  7.94 KiB
  allocs estimate:  1
  --------------
  minimum time:     22.810 μs (0.00% GC)
  median time:      25.910 μs (0.00% GC)
  mean time:        27.594 μs (0.60% GC)
  maximum time:     161.820 μs (66.25% GC)
  --------------
  samples:          10000
  evals/sample:     10

julia> q = @benchmark sort(x, alg=QuickSort) evals=10 samples=10000 setup=(x = rand(1000));
julia> i = @benchmark sort(x, alg=InsertionSort) evals=10 samples=10000 setup=(x = rand(1000));
julia> m = @benchmark sort(x, alg=MergeSort) evals=10 samples=10000 setup=(x = rand(1000));
julia> pc(n) = (1:length(n))./length(n);
julia> plot(i.times*1e-6, pc(i.times), l=2, label="InsertionSort")
julia> plot!(q.times*1e-6, pc(q.times), l=2, label="QuickSort")
julia> plot!(m.times*1e-6, pc(m.times), l=2, label="MergeSort")
```

- Note that setup and teardown are only done once per sample, not once per evaluation!
- This means that we can't do an in-place sort if `evals > 1` because the sorted vector would then be incorrectly used in later replications.
- To avoid this, we would need to make copies of the data in each replication, which would also take time.

Empirical CDF Example

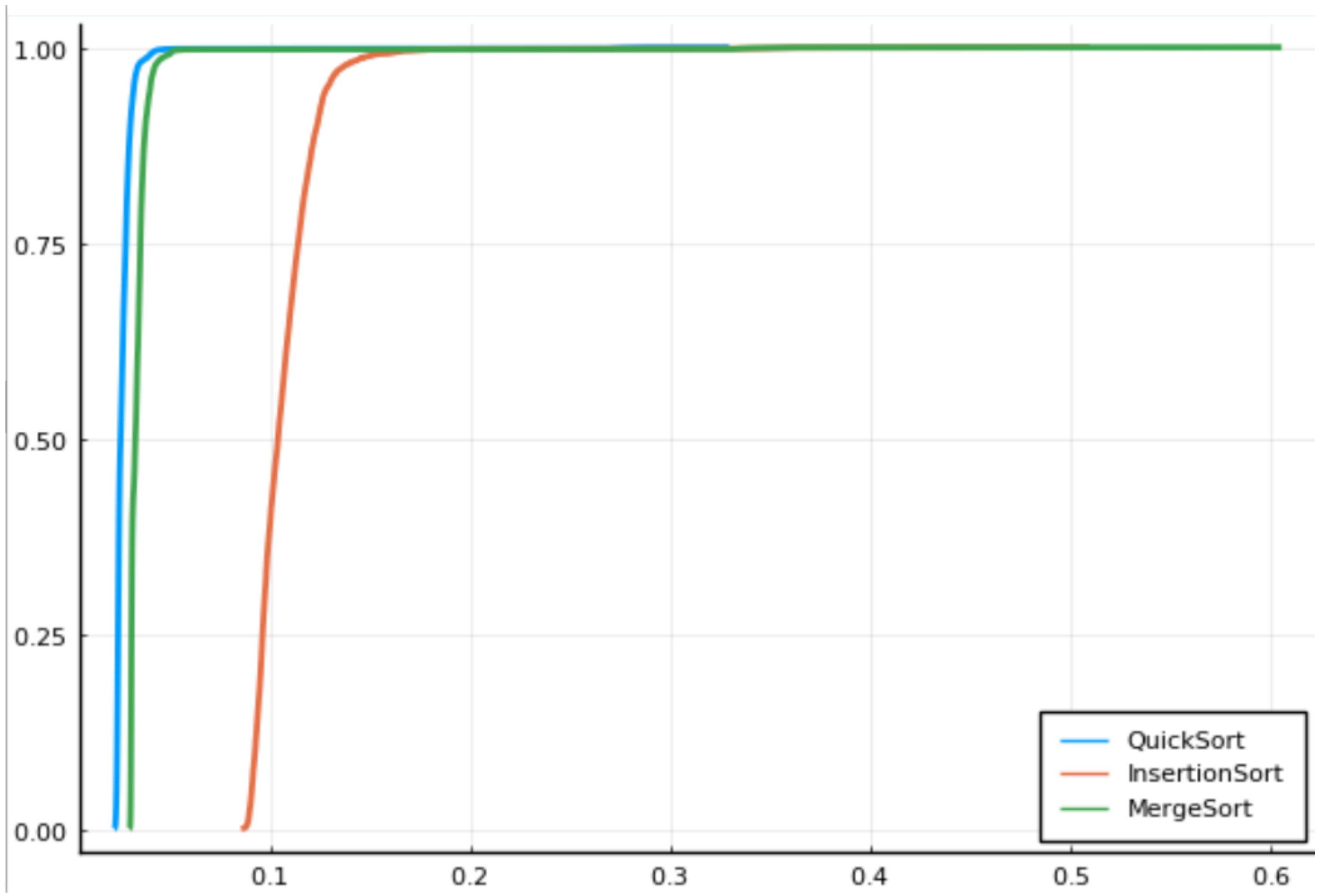

Figure 8: Empirical CDF for 10K replications of sorting algorithms

Ensuring Replicability

- In the results on the previous slide, we used independent random smaples to estimates the CDFs for each sorting algorithm.
- One could argue that this is incorrect because we are using a different test set for each algorithm.
- We are also not seeding the random number generator so the test set would be different if we repeat the experiment.
- For large samples like these, these effects probably don't matter, but in general, they might.
- For some of the visualizations we'll see later, we must use the same test set for all algorithms.

```
julia> using Random
julia> rng = MersenneTwister(12345);
julia> q = @benchmark sort(x, alg=QuickSort) evals=10 samples=10000 setup=(x=rand(rng, 1000));
```

Comparing Distributions

- Given (empirical) probability distribution functions for each algorithm, how do we decide which algorithm is "better"?
- There are methods of comparing statistical distributions, but we will not cover those methods here.
- Which algorithm is "best" depends on the practical usage and it is usually best to present the data and let the reader draw their own conclusions.
- One common approach to presenting the data is simply to present big tables of numbers and let the reader interpret them $\Leftarrow$ don't do this!
- With the ability to interactively manipulate the data in order to draw conclusions (could be coming!), presenting raw data could be a viable alternative at some point in the future.
- Generally speaking, however, we should help the user with the task of assimilating the data.
- We'll use the two most common methods of doing this: summarization and visualization.

Empirical Resource Consumption Functions

- Empirical resource consumption functions plot instance size versus empirical resource (e.g., running time or operations count) consumption).
- Data points represent a summary measure across a set of instances of the same size.
- It may be necessary to break out the instances into groups with different properties, such as density in the case of matrices or graphs.
- If the variation within instances of the same size is important, then we must either
  - Make a 3D empirical distribution in which in put size is a parameter.
  - Produce different plots for different input sizes.

Summarization

- To compare results across multiple dimensions, as described in the previous slide, we must use a summary statistic.
- For example, we may want to plot a traditional empirical running time function with results for each input size summarized.
- We may also simply want to be able to make a comparison based on a single statistic.
  - Arithmetic mean $\Leftarrow$ can be biased by (large) outliers.
  - Geometric mean $\Leftarrow$ can be biased by (small) outliers.
  - Variance $\Leftarrow$ can be used to understand how variability in the results.
- The shifted geometric mean attempts to summarize without introducing (too much) bias due to very large or very small inputs.

Definition 3. *Given a set of values* $N := \{x_1, x_2, \ldots, x_n\}$ *and a shift value* s, *the* shifted geometric mean *is given by*

$$SG(N) = \left(\prod_{k=1}^{n}(x_k + s)\right)^{\frac{1}{n}} - s.$$

Example: Empirical Running Time Functions

- In the below empirical running time function, the result for each input size is the mean of 10K samples.
- The curve is obtained from samples at 10 different list sizes.

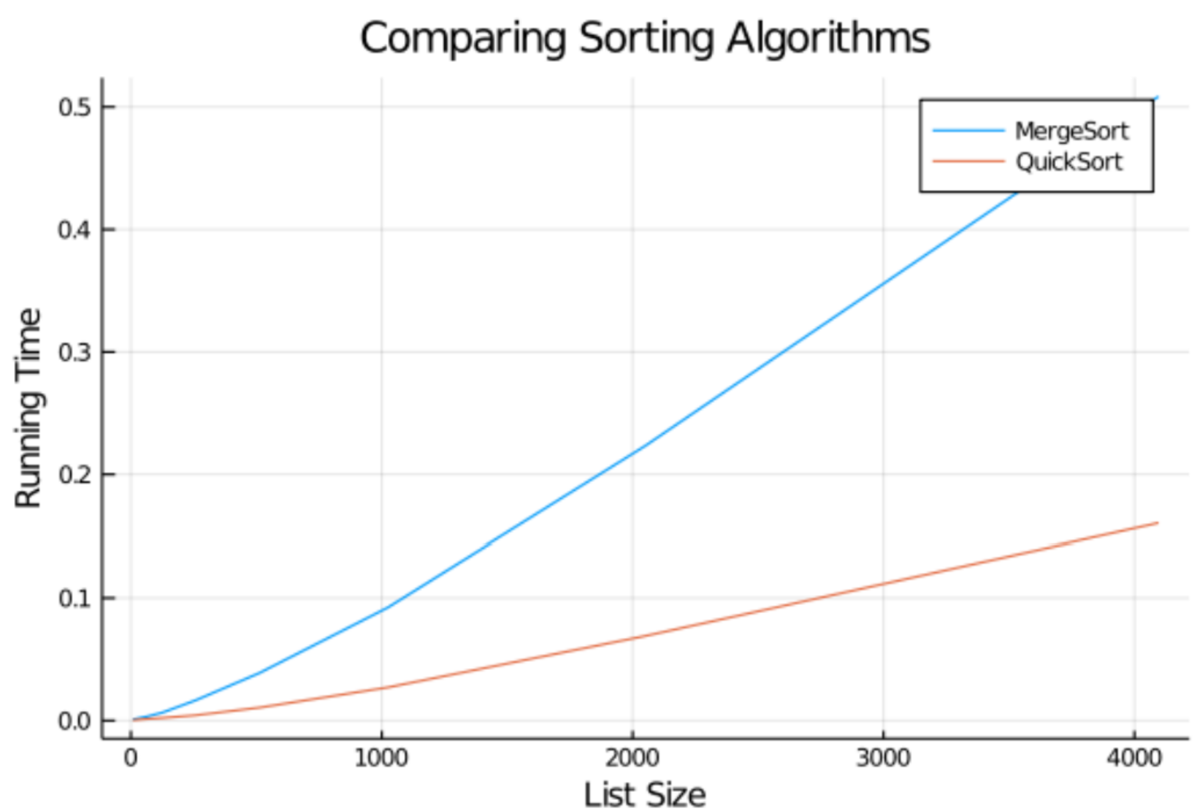

Figure 9: Empirical CDF for 10K replications of sorting algorithms

Proxies

- In practice, we may not always be able to directly measure the consumption of the resource we care about, so we use various proxies.
- We must be careful to justify that these proxies make sense.
- Typical measures in practice
  - Representative operation counts
  - Measures specific to a problem class (iteration counts, etc.)

Representative Operation Counts

- In some cases, we may want to count operations, rather than time.
- This eliminates some of the irrelevant factors that influence algorithmic performance.
- Using operation counts smooth some of the rough edges introduced by empirical analysis and provide a clean way of doing such analysis.
- What operations should we count?
  - Profilers can count function calls and executions of individual lines of code to identify bottlenecks.
  - We may know a priori what operations we want to measure (example: comparisons and swaps in sorting).

Atomic Operations

- In the case of particular algorithm classes, we sometimes consider higher-level operations to be atomic.
- For example, in branch and bound, we may consider
  - Number of total iterations in solving bounding problems.
  - Number of bounding problems solved.
  - Number of branch-and-bound nodes.
- In all cases, we must justify that the operations being counted really are a good proxy for resource usage (i.e., is in the "spirit" of a measure of efficiency).
- The goal is to obtain sensible results and to make a "fair" comparison.

Example: Empirical Analysis of Insertion Sort

Generating random inputs of different sizes, we get the following empirical running time function.

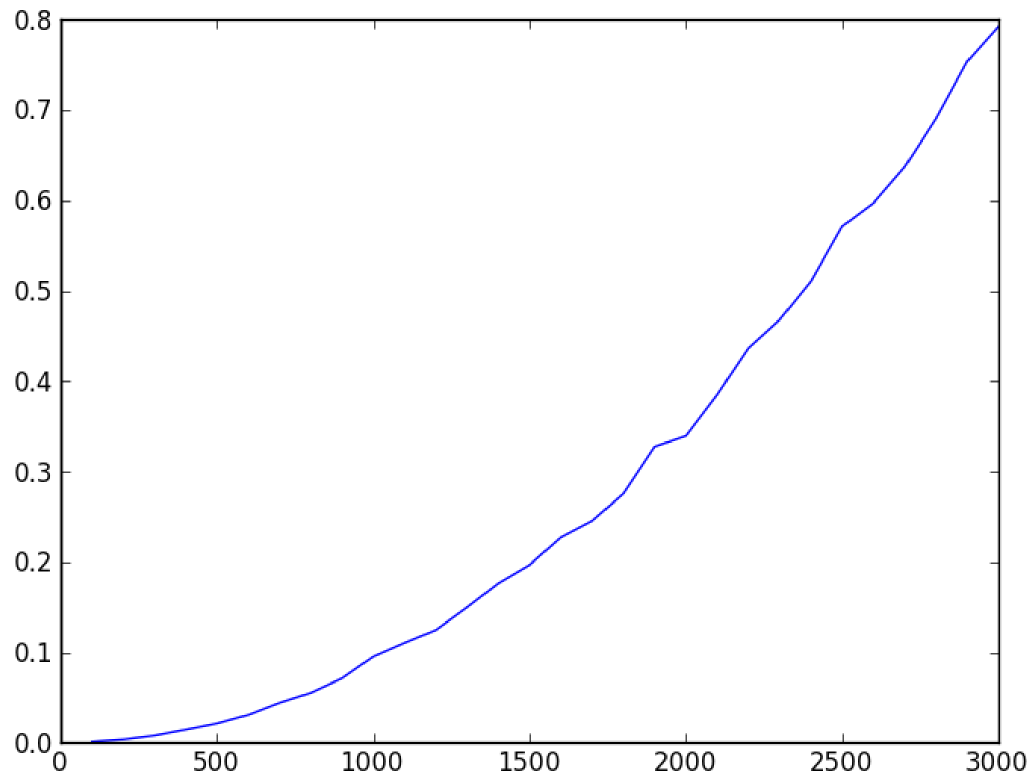

Figure 10: Running time of insertion sort on randomly generated lists

What is your guess as to what function this is?

Operation Counts

- What are the basic operations in a sorting algorithm?
  - Compare
  - Swap
- Most sorting algorithms consist of repetitions of these two basic operations.
- The number of these operations performed is a proxy for the empirical running time that is independent of hardware.

Plotting Operation Counts

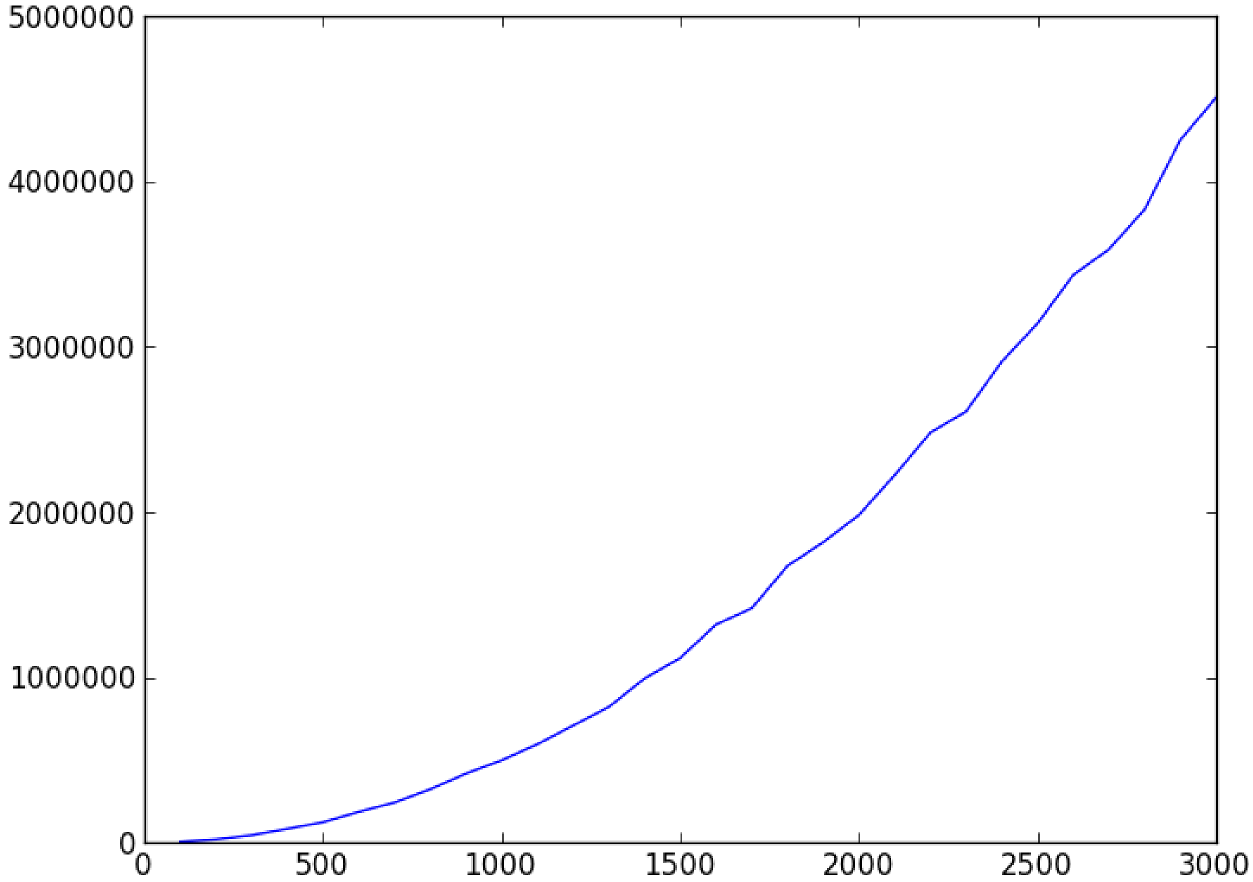

Figure 11: Operation counts for insertion sort on randomly generated lists

Obtaining Operation Counts

- One way to obtain operation counts is using a profiler.
- A profiler counts function calls and all reports the amount of time spent in each function in your program.

```
>>> cProfile.run('insertion_sort_count(aList)', 'cprof.out')
>>> p = pstats.Stats('cprof.out')
>>> p.sort_stats('cumulative').print_stats(10)

   ncalls  tottime  percall  cumtime  percall function
        1    1.011    1.011    3.815    3.815 insertion_sort
   251040    0.507    0.000    0.507    0.000 shift_right
   252027    0.393    0.000    0.393    0.000 compare
      999    0.002    0.000    0.002    0.000 assign
```

Example: Naive Sorting Algorithms

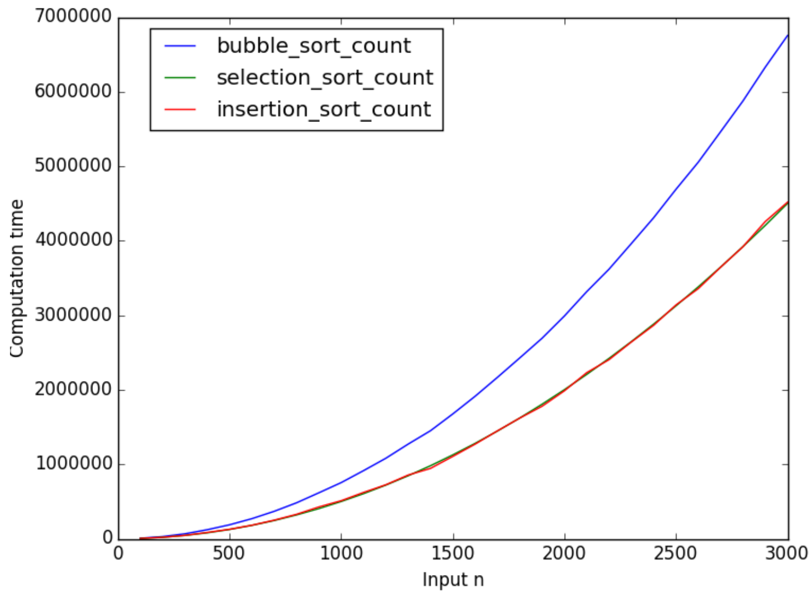

Figure 12: Empirical operation counts

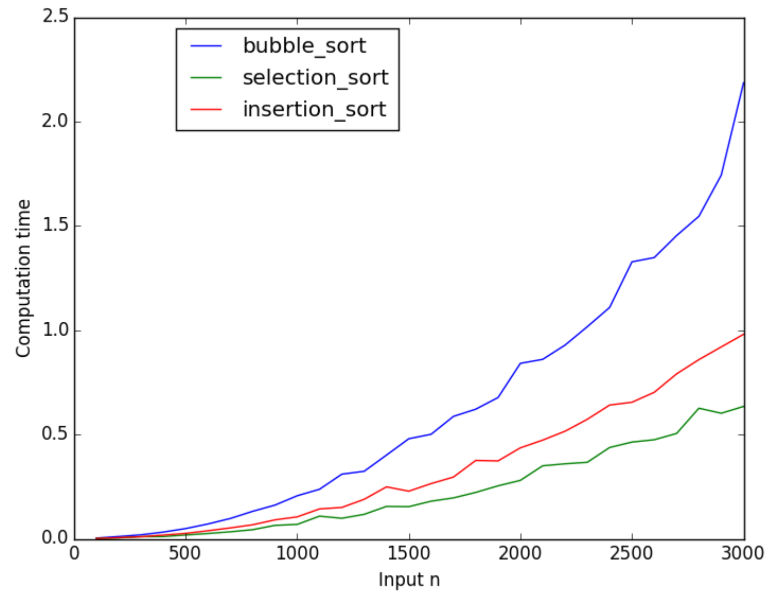

Figure 13: Empirical running times

Example: Optimal Sorting Algorithms

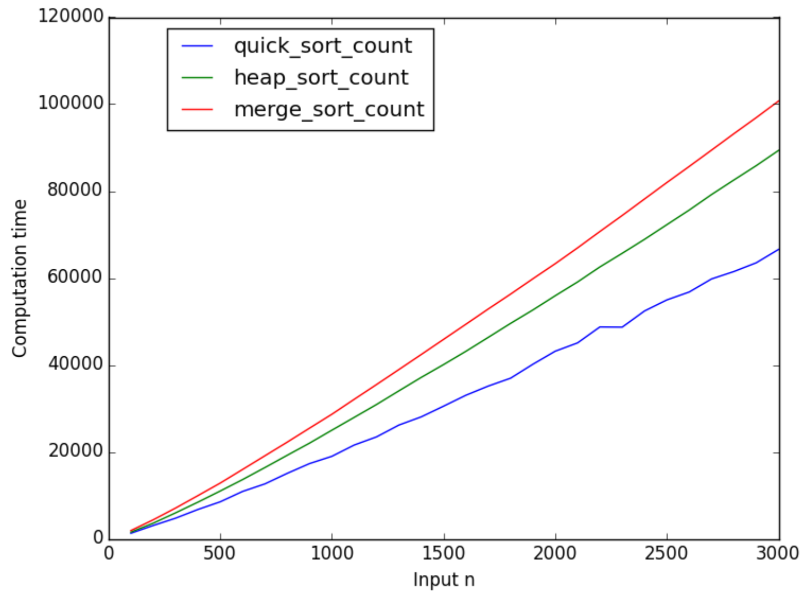

Figure 14: Empirical operation counts

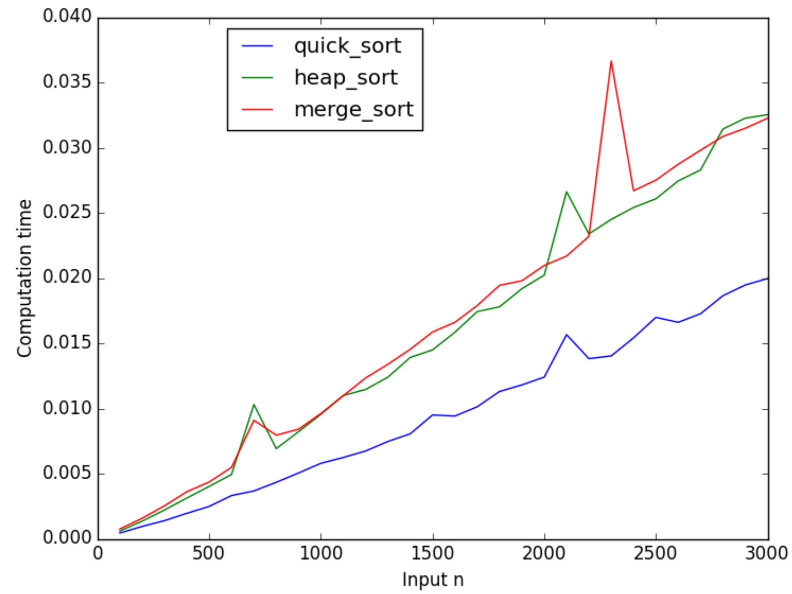

Figure 15: Empirical running times

Some Takeaways

- Depending on the language there may be confounding factors that are difficult to account for.
- In Julia, for example, running times can vary hugely due to garbage collection, loading of modules initial compilation, etc.
- It is also easy to include computations in your analysis that are not actually relevant (generation of random data, etc.)
- It is important to control for all of this to the extent possible.
- This is what Julia's BenchmarkTools attempts to help you to do in an automated way, but it is also important to do this in other settings.