Theory versus Experiment:
Analysis and Measurements of Allocation Costs in Sorting
(With Hints for Applying Similar Techniques to Garbage Collection)

CS 152 Staff

February, 2007

Introduction

This handout illustrates techniques you can use to compare experimental data with theoretical predictions.

- We have chosen units so that theoretical and experimental results can be compared directly, with no "curve fitting."
- We present theoretical and experimental results on the same graph.
- We use a single graph to present the cumulative results of many experiments.
- We plot a single point for each experiment, using different "point markers" for different kinds of experiments.

As a simple illustrative problem, we study answers to the question "how many cons cells are required to sort a list in μScheme?"

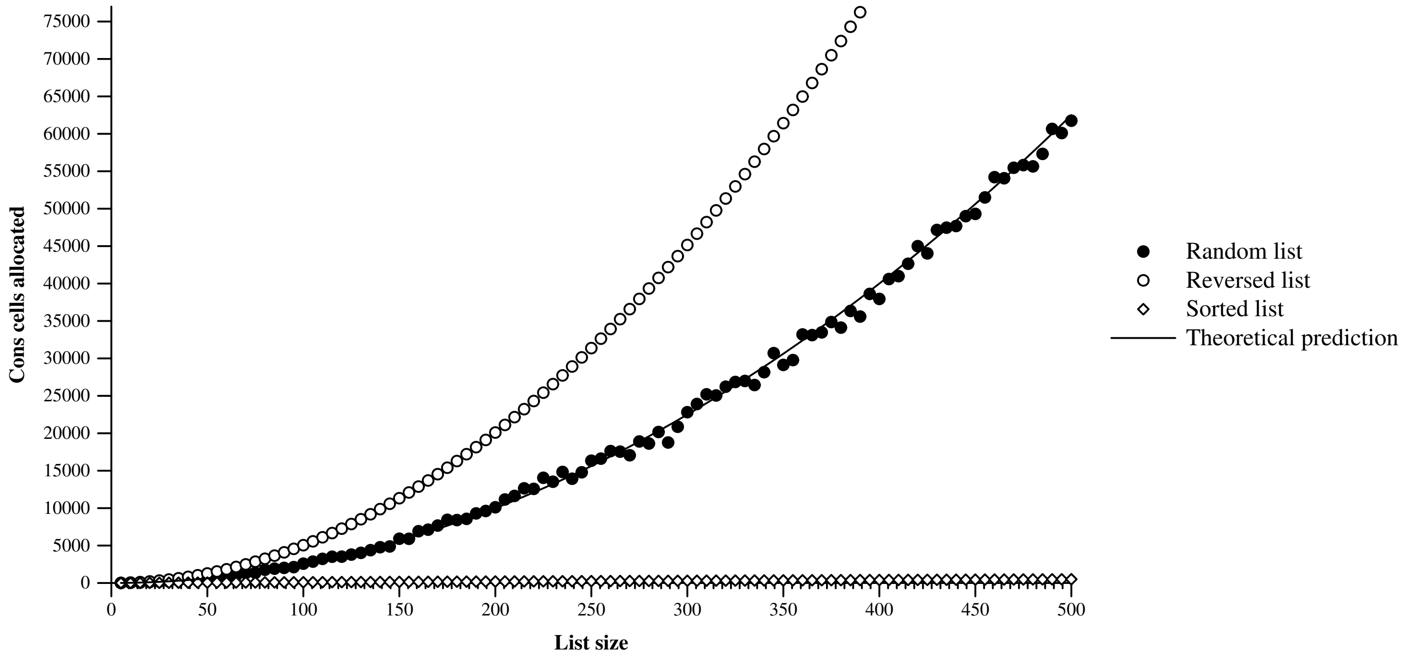
Insertion sort

To predict the cost of insertion sort, consider I_n, the number of cons cells allocated by insertion sort on a list of length n. We first sort the tail at cost I_{n-1}, then insert the new element. If the new element is chosen at random, then on average it goes into the middle of the list, at an expected cost of $\frac{1}{2}n$ cons cells. So $I_n = I_{n-1} + \frac{1}{2}n$, and this equation has the nearly exact solution $I_n = \frac{1}{4}n^2$.

To test the validity of this prediction, we have run three classes of experiments:

- Insertion sort on a random list
- Insertion sort on a sorted list
- Insertion sort on a reverse-sorted list

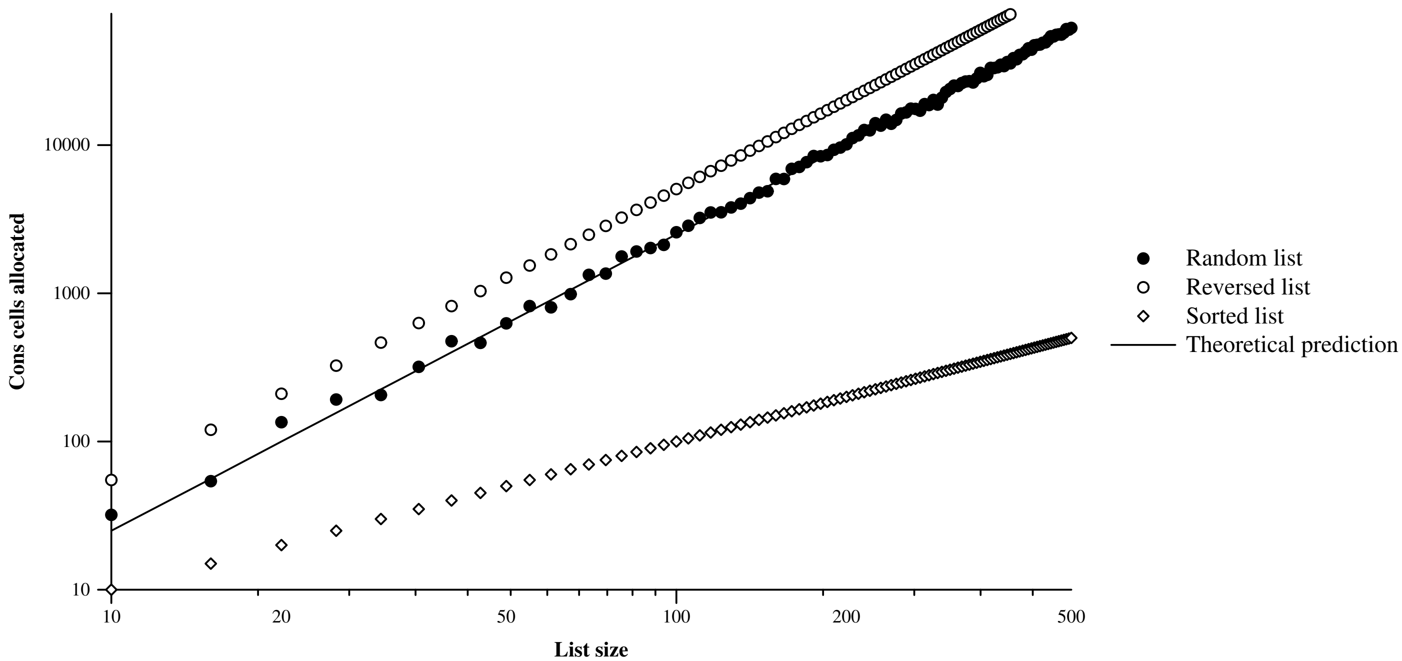
For each experiment we have run a single sort on lists of sizes ranging from 5 to 500 elements, by fives. The results are as shown.



Cons Allocation for Insertion Sort

The experimental results show that our simple analysis is very good at predicting the costs of sorting random lists, but terrible at predicting the costs of sorting sorted or reverse-sorted lists. Obviously further theoretical work is required to explain the dramatic differences in costs.

The costs for the sorted lists, although they appear to be zero, are actually nonzero, but the growth is so slow that it is indistinguishable on this graph. Another graph is probably called for. One idea is to use a log-log scale, on which any power-law curve appears as a straight line.[1]



Cons Allocation for Insertion Sort

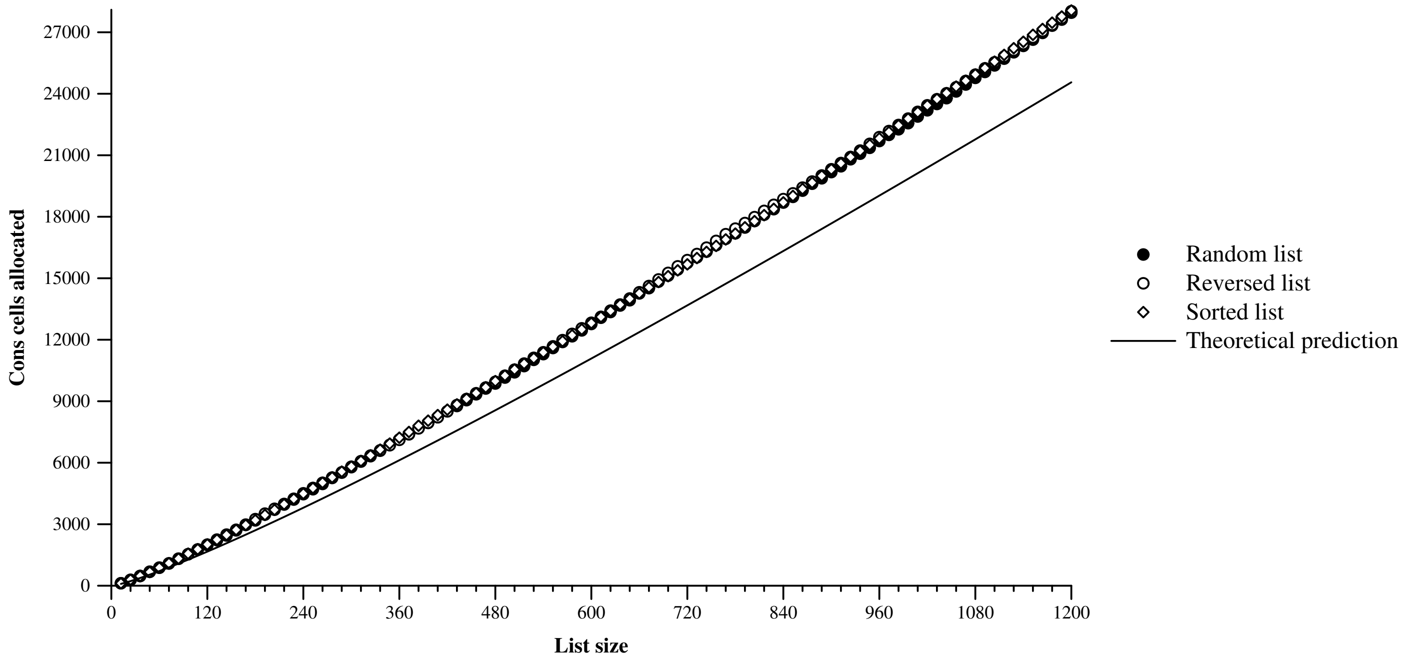
The new graph makes the cost differences clearer: the random list and reversed list have the same slope as the theoretical curve, making both $O(n^2)$, but with different constants. The sorted list also has a straight line, but with a different slope; this curve shows $O(n)$ behavior. These insights should be enough to guide us to a more exact theoretical analysis.

Merge sort

The analysis of merge sort is similar. To sort a list of length n, we split it into two lists of length $\frac{1}{2}n$, which requires allocating n cons cells to hold the two new lists. After recursively sorting these lists, we merge the results, costing another n cons cells. The recurrence equation for the total cost of merge sort is therefore $M_n = 2n + 2M_{\frac{n}{2}}$. This equation has an approximate solution $M_n \approx 2n \log n$. Interestingly, unlike the analysis of insertion sort, this analysis of merge sort doesn't make any assumptions about the ordering of list elements.

[1]A power law is one in which $y = kx^n$. This means that $\log y = \log k + n \log x$, so on a log-log plot a power-law appears as a straight line with slope n.

We perform the same experiments as above, with these results:



Cons Allocation for Merge Sort

Here we have a more interesting result: the cost of merge sorting does indeed depend only on the size of the list, not on the order of the elements. But the cost is not exactly what we predicted. There are two possibilities: either our approximate solution to the recurrence on M_n is not good enough, or our implementation allocates cells that aren't accounted for in our analysis. Careful study of the implementation should reveal which is the case.

How much data is enough?

To make graphs, we had to choose

- The range of list sizes to be explored
- The intervals between list sizes in different parts of the range
- The total number of points in the graph

How did we make these choices?

To begin, we were guided by our theoretical analyses. Such analyses lead to hypotheses about how a program is going to behave under various circumstances. A measurement has to gather enough data to validate or invalidate the hypotheses.

When picking a range, we have to make sure that test cases are big enough to hide small overheads that might not be accounted for in our analyses, but small enough to complete in a reasonable amount of time. In the case of sorting, it's easy to gather lots of data as long as the lists aren't too big. So we chose a range of list sizes such that even the slowest test—in our case the longest list—would complete in a reasonable amount of time. We don't have to worry much about overheads because we're measuring cons cells, and there aren't a lot of hidden cons's floating around. Had we been measuring CPU time, we would have been better off with larger test cases, because there is a lot of hidden CPU overhead floating around the calls and returns.

What about density, i.e., the intervals between test cases? The sort curves don't jump around violently: they are nice smooth curves that are $O(n^2)$ or $O(n \log n)$. Because each individual measurement is cheap, it's easy enough to make the intervals uniform. But if each individual measurement is very expensive, we could cut corners by putting most of the measurements where the curves are steepest, which means large n. (But it's not an exact science. If we want to use the log-log scale, the slopes are flat, and we should have n coordinates evenly distant in log space, which means logarithmically far apart, which means putting most of the measurements at small n.)

To choose the total number of points in a graph is the most difficult decision. You have to have enough points so you can make an informed judgment about whether your measurement actually follows your theoretical prediction. The sorting graphs in this paper have about 50 measurements apiece, which gives a lot of information—probably more than is needed. But if we had only 10 measurements, that would definitely be too few. It really depends on the shape of the curve.

To see these principles in action, let's consider a more wacky example: suppose your analysis produces a curve that looks like

$$y = \frac{(x+10)^2}{(x-5)^4}.$$

This thing is going to tail off to near nothing at zero and for large values of x, so you'll need just a few confirmation points at small and large x. But the function has a nasty spike at $x = 5$, so you'll need to gather a substantial amount of data near $x = 5$ to confirm, for example, that the denominator is raised to the fourth power and not the sixth.

For a garbage collector, we suggest reasonable coverage for a sensible range of γ—enough to confirm or invalidate your model. We also suggest you gather some points outside the sensible range, to be sure you understand what is happening there.

Tools

Gathering data

The code for the course includes a function `gammadesired` that will help you set γ in your collector. By using this function, you can change γ from within μScheme itself, simply by setting the μScheme variable `&gamma-desired`. It would be a big mistake to recompile your collector each time you want to change γ.

It can be quite useful to write shell scripts that create test cases. The course staff recommend learning Bourne shell (`sh`) and Awk, but many students get good results using Perl. You might have a look at our `inserttest` and `mergetest` scripts to see how they work. To learn `sh`, *An Introduction to the Unix Shell* by S. R. Bourne is more helpful than a man page. To learn Awk, try *Awk—A Pattern Scanning and Processing Language (Second Edition)* by A. Aho, B. W. Kernighan, and P. J. Weinberger. Both of these documents are part of the BSD Unix "User's Supplementary Documents" and should be available for free on the Internet. There is also a book by Aho, Kernighan, and Weinberger.

Drawing graphs

Many students report that it is very tedious and time-consuming to try to produce decent graphs using Microsoft Excel. The graphs in this paper were produced with `jgraph`, and the `jgraph` sources for these graphs are available online. We have had very good luck with `jgraph`, which can be freely downloaded from the Internet and which is installed on the FAS "nice" servers in `/usr/local/bin`.[2] We have also heard good things about `gnuplot`, although we can't confirm these from our own experience.

Avoid common mistakes

Comparing theoretical predictions with experimental results requires thought. The following advice will help you avoid mistakes that students have made in the past.

- If your theory predicts experiment to within a factor of 2, don't just say "the theory fits pretty well." Factors of 2 demand explanation.
- Don't overlook smaller deviations. For example, something is clearly wrong with our prediction of the cost of merge sort above. Further investigation is needed to figure out what the problem is.
- When you're trying to understand your results, don't overlook the load that your test cases place on the memory-management system. You might find it useful to construct different kinds of test cases. One interesting test case is one in which every allocated cell lives forever. Another is one in which every allocated cell dies quickly. You can imagine others.
- If your theory does not indicate that $y = ax^b$, do not fit your experimental data to a curve of the form $y = ax^b$.
- Don't forget to say what you are measuring. A label on a y axis is usually sufficient.
- Don't compare quantities that come in different units. For example, if your theory predicts comparisons, and your experiment measures allocations, it is meaningless to compare the two.[3]

[2] If you compile `jgraph` from sources, *be sure to set* `-DLCC` *as described in the Makefile*. If you do not, you will get mysterious failures when you try to view Jgraph's output.

[3] Unless you have additional theory that relates comparisons to allocations, of course.

- Don't add quantities that come in different units. For example, computing "total sorting work" by adding comparisons and allocations is meaningless.

- Don't graph your theoretical prediction as if it were just another experiment. It is quite useful to use a single graph to compare multiple experiments with theory, but only if the theoretical prediction is clearly distinguishable.

- Don't use your theoretical prediction as a substitute for measurement. For example, if you are trying to predict cons cells, and if your theory says the number of cons cells is $\frac{1}{4}n^2$, it is not useful to run your code and measure $\frac{1}{4}n^2$—you must measure the number of calls to `cons`.

- Don't hide all the interesting behavior by using the wrong scale on your graphs. If your theory says $y = \frac{1}{x}$, and you graph y on a linear scale of 0 to 10, it will be impossible to compare theory and experiment for values of x much greater than 1. For some graphs, a log-linear or log-log plot may help with this problem. You could also consider showing both y and $10y$ on the same graph.

- Don't throw out interesting data in order to fit the rest on a reasonable scale. If your theory says $y = \frac{1}{x}$, you definitely want to see what happens near the singularity at $x = 0$. Points near $x = 0$ should not be dismissed as "outliers."