Chapter 3

What To Measure

The performance measure of primary interest in most algorithmic research projects is that old devil, *time*: I want to predict how long I will have to wait for my output. Running a close second, in the context of algorithms and heuristics for NP-hard problems, is *solution quality*: I want to evaluate the quality of the output produced by my algorithm.

The choice of *performance indicator* – the quantity actually measured in the experiment – plays a big role the type of predictions and evaluations that can be made. Algorithms present special difficulties and opportunities for the experimenter. In the case of time performance, the problem is how to balance the competing goals of generality and precision when measuring the time required by a given algorithm. The main question when measuring solution quality is how to evaluate non-optimal solutions when the optimal solution isn't known.

This chapter surveys options available to the experimenter in both scenarios, and presents guidelines for choosing among them. Section 3.1 covers measurements of time performance, and Section 5 discusses ways to evaluate solution quality in the context of NP-hard problems.

We start with some general advice on reporting measurements from algorithmic experiments. Chapter **??** describes techniques for designing experiments to be "analysis friendly," in the sense that the data analysis phase of the research project is made more efficient and more informative. The choice of performance indicator, whether measuring time performance or solution quality, can also be evaluated from that point of view. Of course, different goals of an experimental project may be in conflict with the desire of the data analyist to have well behaved data to work with; in such cases the overall scientific goals should prevail. But giving a little early thought to the nature of the outcomes can produce better insights and greater efficiency of experimentation. The basic idea is to be sure that the test program reports

full information about the experiment in such a way as to allow maximum flexibility during data analysis.

Don't summarize prematurely. A *summary statistic*, such as mean or standard deviation, is a computed quantity that provides a concise description of a data set. As much as possible, test programs should report the "raw data" rather than summary statistics. Summarization is best performed at the analysis stage, by an analyst who can observe the distributional properties of the data in order to choose the right summarization technique. It should not be performed inside the test program, before the data sees the light of day. (Summary statistics may be perfectly justified in published research papers; what is important is to avoid creating a test program that obscures rather than reveals the true performance picture.)

This principle can be applied at many decision points in the planning process. First, if an experiment carries out t independent random trials, the test program should report the cost of each individual trial (whether time or solution quality), not just the average cost over those trials. Second, test programs should report primary cost components rather than secondary costs computed in a "lossy" fashion from those components. For example, suppose the goal is to study the the convergence from initial solution quality S_0 to final solution quality S_f for some iterative algorithm: the program only reports the difference $D = S_0 - S_f$ or the ratio $R = S_0/S_f$, information about scale and distribution in the original costs cannot be recovered; but if if the program reports S_0 and S_f both the difference and the ratio can be calculated during data analysis. Similarly, the case study in Section 3.1 describes an algorithm where total time is the sum of three component times: it is easier to have the test program report the three components and to derive their sum, than to have the program report just the sum and try to break it into components during analysis.

Of course a certain amount of summarization is necessary when the test program produces far too much data to be analyzed. For example it would be impractical for our hypothetical iterative algorithm to report solution quality S_i at every iteration, which is why S_0 and S_f are employed as summary statistics representing the maximum and minumum over all observations S_i. In cases with too much data it helps to consider carefully what kind of summarization to use: simple means and standard deviations, for example, will obscure bimodal or skewed data distributions which can lead to important insights about performance. It may be possible to *sample* from the distribution, say, by reporting S_i at regular or random intervals, if only for guidance in choosing the appropriate summary statistics to use in later experiments.

Note finally that if S_i is a numerical quantity representing the cost of a heuristic solution to an NP-Hard problem, then S_i itself is a type of summary

statistic that represents the full solution, likely some type of combinatorial structure such as a tour of a graph, or a packing of weights into bins. Such non-numerical experimental outcomes are usually too large and complicated to be easily managed during data analysis, but there is no requirement that the test program report only the single cost identified by theoretical analysis: more detailed measurements of solution quality may be useful. Often in experimental algorithm research the experiment evolves from *width* – great variety in inputs and input parameters, with simple performance indicators – to *depth* – less variation in inputs and input parameters, and greater detail in in performance indicators. Such detail produces a type of "algorithm visualization" experiment that can be key to the development of real insights about algorithm mechanisms.

The rest of this chapter presents more specialized considerations for choosing time performance indicators and solution quality indicators that best meet the goals of a given algorithmic project.

3.1 Measuring Time Performance

Algorithm analyzers face a fundamental dilemma when using experimental methods to develop good predictions about algorithm performance: which is more important, *accuracy* or *precision*?

The following story *Risks Digest*, a website that catalogs reported incidents of technology failures, illustrates the difference.[1]

> *Risks Digest, 21 November 1996*
> When the Russian Mars probe crashed earlier this week, it provided an interesting example of the difference between precision and accuracy ....
> The first estimates of where it would land were anywhere in an area about 2000 km across. The next reports said that it would be landing at about the New South Wales/Queensland border, and they seemed to think it would come down somewhere in an area about 500 km across. The next reports said that it would come down somewhere in an area in the north west of New South Wales, and the precision of this estimate seemed to be about 100 km. As it turned out, it came down about 2000 km west of Chile, in the Pacific Ocean, a third of the way around the world from

[1] Risks Digest `catless.ncl.ac.uk/Risks`, posted by Ben Morphett, 21 November 1996. Reused without explicit authorization under blanket permission granded for all Risks-Forum Digest materials. The authors(s), the RISKS moderator, and the ACM have no connection with this reuse.

> Australia. So as the precision of the reports was increasing, the accuracy of the reports was staying about the same – very wrong.

In experimental algorithmics the classic dillemma has been whether to measure the dominant algorithm cost identified by theory, or to measure CPU time.

Asymptotic bounds on dominant costs are extremely accurate: *The algorithm will perform no more than cn^2 basic operations on inputs of size n.* This claim will hold no matter what implementation, inputs, or platform are measured. But without more information about the unknown constant c, or about how to translate basic operations into time units, it is difficult to predict whether the algorithm take a few seconds or a few hours to run.

On the other hand, CPU time measurements are extremely precise – to the fraction of a second – but often very inaccurate when it comes to translating times from one platform to another. For example, participants in the DIMACS TSP Implementation Challenge [5] tested the same TSP code on the same suite of nine input instances, to compare runtimes on 23 different platforms. They observed runtimes ranging between 37 seconds and 380 seconds on one instance; on another, runtimes ranged between 638 and 6100; and on 11 of the 23 platforms the code sometimes crashed on large inputs. Millisecond precision is no use when cross-platform time predictions can be off by minutes or hours.

Fortunately, the experimenter has more options beyond these two. While the tradeoff between precision and accuracy cannot be avoided, it can be controlled somewhat by thoughtful selection of *time performance indicators* – the quantities actually measured in an experimental study. As a framework for understanding these options, we consider "algorithms" and "programs" to be two points on a *scale of instantiation* according to how much specificity is in their descriptions. Here are some more recognizable points on this scale.

- At the most abstract end of the scale are *metaheuristics* and *algorithm paradigms*, which describe generic algorithmic structures that are not tied to a particular problem domain. For example, Dijkstra's algorithm is a member of the Greedy paradigm.

- The *algorithm* is an abstract description of a process for solving a given abstract problem. At this level, we might consider Dijkstra's algorithm written in pseudocode, with a particular PQ implementation specified.

- The *source program* is a version of the algorithm implemented in a particular high-level language. Specificity is introduced by language and coding style, but source code is platform-independent. Here we

might see Dijkstra's algorithm implemented in C using standard library functions.

- The *object code* is the result of compiling a source program for a target architecture. This version of the algorithm is written in machine code.
- The *process* is a program actively running on a particular architecture. Performance at this level is affected by properties such as system load, the size and shape of the memory hierarchy, and multi-core design.

As we move down the scale we lose generality and gain precision in time measurements. Performance indicators at the abstract end of the spectrum are imprecise, but easier to extend beyond the range of experiments, while performance indicators at the instantiated end are precise but complex and difficult to model. Interesting experiments can take place at any point on this scale: the main idea is to match the performance indicator to an instantiation level appropriate to the project.

The following sections consider the merits of several types of performance indicators; a toy algorithm described in the next section provides a running example to illustrate various measurement strategies. The chapter concludes with a short list of guidelines for choosing performance indicators to match experimental goals.

Case Study: Markov Chain Text Generation. The Text Generation Problem is to generate a random text that looks like it was written by a human being. One famous solution is to employ monkeys at typewriters for many years, but we prefer a computational approach which is cheaper and faster: The *Markov Chain* (MC) algorithm generates random words according to word sequence probabilities calculated froman input text T.

This algorithm, primarily used for generating parodies and bypassing spam filters, has also been used by Kernighan and Pike [3], and by Bentley [2], to illustrate principles of algorithm design and code tuning. Bentley's implementation, described here, can be downloaded from *AlgLab*.[2]

The MC algorithm reads a text T containing n words, together with parameters k and m. It builds a dictionary W, containing all k-word phrases in the text as keys. The string `phrase` is initialized to contain the first k words of T, which are printed. The remaining $m - k$ words are generated and printed by this process:

1. Lookup: Find all k-word keys in W that match `phrase`. Return the suffixes ($k + 1$-st words) that follow each matching key.

[2]To learn about AlgLab see Appendix A or visit `www.cs.amherst.edu/ccm/alglab`.

2-Word Key	Suffix	2-Word Key	Suffix
this is	a	this is	a
is a	test	is a	test
a test	this	a test	of
test this	is	test of	the
this is	only	of the	emergency
is only	a	the emergency	broadcasting
only a	test	emergency broadcasting	system
a test	this	broadcasting system	this
test this	is	system this	is

Figure 3.1: Two-word keys and suffixes from the text T = *this is a test this is only a test this is a test of the emergency broadcasting system.* Notice the suffixes for the last two keys wrap around to the beginning.

2. Random Select: Choose a suffix word uniformly at random from this set. Print it, append it to `phrase`, and drop the first word from the phrase.

For example suppose T = *this is a test this is only a test this is a test of the emergency broadcasting system.* Figure 3.1 shows keys and suffixes assuming $k = 2$. If `phrase` contains `"this is"` the Lookup step returns three suffixes (`a`, `only`, and `a`). With probability 2/3 the word `''a''` is printed and `phrase` becomes `"is a"`.

Not surprisingly, larger values of k better-looking results. In the two examples below, T contains the texts of all nine comedies by William Shakespeare. The first sample was randomly generated with $k = 1$ (newlines added by hand):

> ALL'S WELL THAT ENDS WELL ACT II Troy.
> DON JOHN And he, If I heard of all the giddiness of the pairs of Beatrice between her song and till they've swallowed me are in sin in jest. I his characters; He'd Lay on't. Perdita I think you beat thee. Fie on him: 'Tis false. Soft, swain, Pompey surnamed the PRINCESS, KATHARINE¡ ROSALINE, MARIA, KATHARINE, BOYET, ROSALINE, MARIA, and let him keep not o'erthrown by mis-dread.

The random text below, generated with $k = 3$, shows much better grammer and punctuation.

> ALL'S WELL THAT ENDS WELL ACT III SCENE III A church.
> [Enter DON PEDRO, CLAUDIO, and LEONATO] BENEDICK

> Will your grace command me any service to the world's end will have bald followers.
> ANTIPHOLUS OF SYRACUSE Thy sister's sister.
> LUCIANA That's my sister.
> ANTIPHOLUS OF SYRACUSE There's none but asses will be bridled so.

Now, suppose we want to run the MC algorithm on a file of English text, with given parameters n, m, and k. How much time will it take? How precise and accurate is our prediction? The next few sections consider a variety of approaches to answering those questions.

Disclaimer: *The following experiments are far too limited in scope to support general conclusions about the MC algorithm or its implementations. The "conclusions" stated here are intended only to illustrate how different performance indicators reveal different kinds of information about performance, not to suggest new truths about the algorithm.*

3.1.1 Adding counters to source code.

We first consider measuring the algorithm by inserting counters into the the source code. It simple matter to insert a few integer variables that increment every time interesting operation is performed, and report their values once the task is finished. In a small-scale experiment nothing more is needed. However in a more ambitious experimental project, a little early attention to counter-code technique can produce big payoffs in experimental efficiency. The interested reader is invited to skip ahead to Section XX for some design guidelines such as using runtime switches to control counters, and creating self-documenting output files. The no-frills approach is adopted throughout this chapter.

Counting word operations. So far, our view of the MC algorithm too abstract to analyze: without more specificity, we can only say that the initialization step reads n words and the generation step iterates $m - k$ times. Therefore move down the instantiation scale and suppose the data structure W is implemented with a sorted array of keys.

Figure 3.2 shows two parts of the `markov.c` program (which can be downloaded from *Alglab*). This program uses an array `word[]` containing indices to words in the input text, which is then sorted to k-word keys. For example, the word indices in the text `"this is a test this is only a test"` are `0, 6, 9, 11, 16, 21, 24, 29, 31`. The table below shows indices in W after sorting, together with the 2-word keys and suffixes that they point to.

```
// Part 1: Sort by keys
qsort(word, nword, sizeof(word[0]), sortcmp);

// Part 2: Generate Output
for ( ; wordsleft > 0; wordsleft--) {
    // Lookup
    l = -1;
    u = nword;
    while (l+1 != u) {
        m = (l + u) / 2;
        if (wordncmp(word[m], phrase) < 0) l = m;
        else  u = m;
    }
    // Random Select
    for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++)
        if (rand() % (i+1) == 0) p = word[u+i];
  }
```

Figure 3.2: **A C implementation of the MC Algorithm**. This code fragment is from `markov.c`, downloadable from Alglab.

W	key	suffix
9	a test	this
29	a test	this
6	is a	test
21	is only	a
24	only a	test
11	test this	is
31	test this	is
0	this is	a
16	this is	only

The Lookup step uses binary search is used to find the beginning of the contiguous subarray of keys that match the current phrase. At the end of the Random Select loop each suffix in the subarray has equal probability of being assigned to `p`.

Our first performance indicator is the dominant operation suggested by theory: let the *word cost* of the algorithm equal the number of times the `wordcmp` function – which compares a pair of keys – is invoked. It is straightforward to insert a counter inside `wordcmp` to report this value.

Straightforward, but inappropriate. The `wordcmp` function is called from

three different places in the program: when `word` is sorted, during binary search, and during random suffix selection. Standard average-case analysis gives us the following preliminary formula describing word cost as a sum of three terms. (The parameter k may affect coefficients but not this general asymptotic form.)

$$C(n, m) = qn \log_2 n + bm \log_2 n + rmn.$$

Note this formula is preliminary because "average case" is not the same as "typical case": it has not yet been established that this form is appropriate to English text. For example, the n in the third term represents an untested conjecture that the number of duplicate keys traversed in the Select loop grows proportionally with n. Our first experiment is designed to evaluate how well the formula fits the data, and to find values for the coefficients.

Instead of measuring total word cost, a better strategyis to measure each program component separately. Therefore we insert three counters into the code as follows (refer to Figure 3.2).

- Put `qcount++` inside the `sortcomp` function, which is called by `qsort` and which calls `wordcmp`.
- Place `bcount++` inside the binary search loop, just above the `if` statement.
- Place `rcount++` in the random selection loop. Since the call to `wordcmp` is in the for loop header, this counter should appear twice, to cover tests that evaluate to both true and false:

```
for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++) {
    rcount++;                          // count the true tests
    if (rand() % (i+1) == 0)
    p = word[u+i];
}
rcount++;                              // count the false test
```

To support the functional analysis it is also necessary to insert code to report n, m, and k for each experimental trial; these correspond to the variables `nword`, `wordsleft`, and `k` in the program.

Figure 3.3 shows results of tests using three input files: file h ($n = 112,493$) is *Huckleberry Finn*, by Mark Twain; file b ($n = 207,423$) is *The Voyage of the Beagle*, by Charles Darwin; and file c ($n = 377,452$) is the collection of all nine comedies by William Shakespeare.[3] In these trials, k is

[3]All text files were downloaded from Project Gutenberg *www.pg.org*.

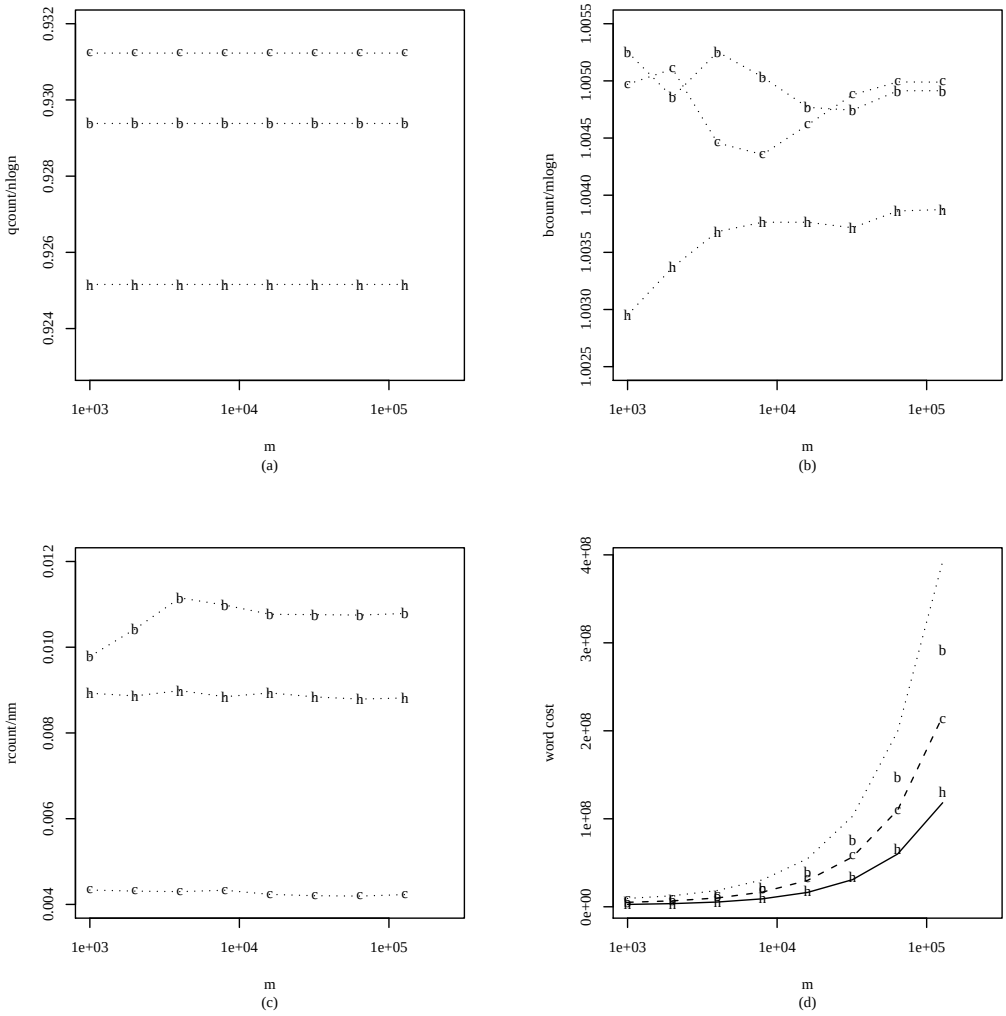

Figure 3.3: **MC comparisons.** The graphs show results of one trial each of MC on three text files h,b,c,of sizes $n = 112493, 207423, 377452,$ with $k = 1$, and several m values. The x scale is logarithmic.

fixed at 1 and m increases from 1,000 to 128,000, doubling each time: note the log-x scale on all graphs.

The first three panels show, respectively, the estimated coefficients $q = \texttt{qcount}/n \log_2 n$, $b = \texttt{bcount}/m \log_2 n$, and $r = \texttt{rcount}/nm$. In all three panels the lines are mostly horizontal, suggesting that each term describes asymptotic performance fairly well. Over the three files, the coefficients q, b and r vary from their means by at most ± 0.008 in absolute terms; this corresponds to about 0.3 percent variation for q and b, but almost 50 percent variation from the mean for r.

Averaging over the three text files, we obtain the following formula for word cost when $k = 1$.

$$C(n, m) = 0.9286n \log n + 1.0045m \log n + 0.0079nm.$$

Panel (d) shows total comparisons for the three files, together with three estimates (lines) from $C(n, m)$. The formula nicely estimates word costs for files h and c, but overestimates word cost for b by a factor of nearly 2 at large m. A little arithmetic shows that the random suffix selection term dominates word cost at these problems sizes and is the likely source of the discrepancy.

Althought the three counters are embedded in a particular program, can argue that these of performance indicators describe the abstract algorithm, since any reasonably efficient implementation of the sorted array data structure would have the same asymptotic form; different programmer styles would at most produce small variations in the coefficients.

One claimed advantage of asymptotic analysis is that since $C(n, m)$ is based on (validated) model of algorithmic mechanisms, the formula can be extended to predict word costs outside the range of experiments. To test this claim, the table below compares predictions from $C(n, m)$ with observations using new text files and problem sizes. The new text files are `twain.txt` containing the texts of *The Adventures of Tom Sawyer*, *A Connecticut Yankee in King Author's Court*, *Following the Equator*, by Mark Twain and *The Gilded Age* by Twain and Warner; `tragedies.txt`, containing the ten tragedies by William Shakespeare; and `austen.txt` contain ing *Persuasion* and *Pride and Prejudice* by JaneAusten. This experiment sets $k = 1$ (as before) and $m = 256000$, which is outside the original parameter range.

file	n	$word/C(n, m)$
austen	204,839	0.881
tragedies	251,181	0.491
twain	532,895	1.057

The largest prediction gap is on the `tragedies` file, where the estimate is about half the observed word cost. The Twain estimate is off by less than 6 percent, and the Austen estimate is within 20 percent of true cost.

If this level of accuracy is not sufficient, the next step in the analysis is to evaluate variation due to the random selection step, to incorporate a larger collection of texts, and to refine the formula by adding terms to account for differences among input files. The reader is invited to pursue these interesting topics; for now, we abandon this thread of investigation to consider a new performance indicator that incorporates variation in k.

Counting character operations. The parameter k has a different status from n and m in our analysis, since k remains a small integer rather than growing towards infinity. In this section we move from an asymptotic analysis to consider how k affects the cost of the MC algorithm.

First, coefficient r on the third term is likely to *decrease* with k, since the random selection loop depends on the number of duplicates of keys in W: there may be thousands of duplicates of one-word keys like `"we"`, but few duplicates of three-word keys like `"we catched fish"`. (It is less clear how or whether q or b depend on k.)

Second, the cost of the `wordcmp` function, shown below, is likely to *increase* with k, since this function contains a loop that compares keys character-by-character, and key length increases with k.

```
int wordcmp(char *p, char* q) {
    int n = k;
    for (; *p == *q; p++, q++)
        if (*p == 0 && --n == 0) return 0;
        return *p - *q;
}
```

Which effect prevails? To study this question we switch from the word cost model to measure *character cost*, equal to the number of iterations performed by the loop inside `wordcmp`. The new formula $C'_k(n, m)$ for character cost has the same general form as $C(n, m)$ except for new coefficients. Let r_k replace r to capture the above-mentioned dependence on key duplication. A quick experiment suggests that q and b do not vary with k, so we use the previous coefficients $q = 0.9286$ and $b = 1.0045$. Let r'_k denote the number of loop iterations per key comparison during suffix selection, with similar notation for the other terms. We have

$$C'_k(n, m) = q'_k(0.9286)n \log_2 n \; + \; b'_k(1.0045)m \log_2 n \; + \; r'_k r_k nm.$$

Our next experiment is designed to fill in values for the new coefficients. We use our original three text files, five values of $m = 800 \ldots 1200$ incrementing by 100 each time, and five values $k = 1 \ldots 5$.

We consider coefficient r_k first, or for convenience, $\bar{r}_k = r_k n$. The table below shows `rcount`$/m$ for each text file, equal to the number of select loop iterations per output word. Means over three files are shown in the bottom row.

	n	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
huckleberry	112,492	1002.4	19.9	2.8	2.1	2.0
beagle	207,423	2063.3	69.1	2.8	2.2	2.0
comedies	377,452	1649.0	19.2	3.5	2.1	2.1
$\bar{r}_k = r_k n$	1571.6	36.1	3.0	2.1	2.0	

As conjectured, the coefficient decreases with k, quite sharply from $k = 1$, but then tapering off. Note the minimum possible value for $r_k * n$ is 2 since the loop must iterate at least once per output word; this minimum occurs when the key in `phrase` has no duplicates.

We also note that, contrary to our original assumption, the third term $r'_k r_k nm$ does not increase with n over this range of experiments. Therefore to provide a better *descriptive* fit to the data within this range, we replace the term with $r'_k \bar{r}_k m$. We shall summarize this table with $\bar{r}_1 = 1571.6$, $\bar{r}_2 = 36.1$, and $\bar{r}_{k\geq 3} = 2.0$. Note that each summary coefficient is within a factor of two of the data it summarizes.

Now consider coefficients q'_k, b'_k and r'_k which correspond to the number of loop iterations per key comparison. The table below shows means over the three text files and each m; in all cases the observed data was within 5 percent of the means shown here. The last column shows means for $k = 3..5$.

	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k \geq 3$
q'_k	3.24	4.02	4.17	4.20	4.20	4.19
b'_k	3.26	4.52	5.02	5.35	5.78	5.38
r'_k	4.56	9.43	13.75	16.23	19.35	16.44

As conjectured, these coefficients increase with k in all three cases. Coefficient r'_k is largest because the suffix selection loop iterates through identical keys, which maximizes this cost. Multiplying coefficients, we obtain the summarized formula below to describe character costs for the MC algorithm.

$$\begin{aligned} C'_1(n,m) &= 3.01 n \log_2 n + 3.27 m \log_2 n + 7166.5 m \\ C'_2(n,m) &= 3.44 n \log_2 n + 3.73 m \log_2 n + 340.4 m \\ C'_{k\geq 3}(n,m) &= 3.89 n \log_2 n + 5.41 m \log_2 n + 32.89 m \end{aligned}$$

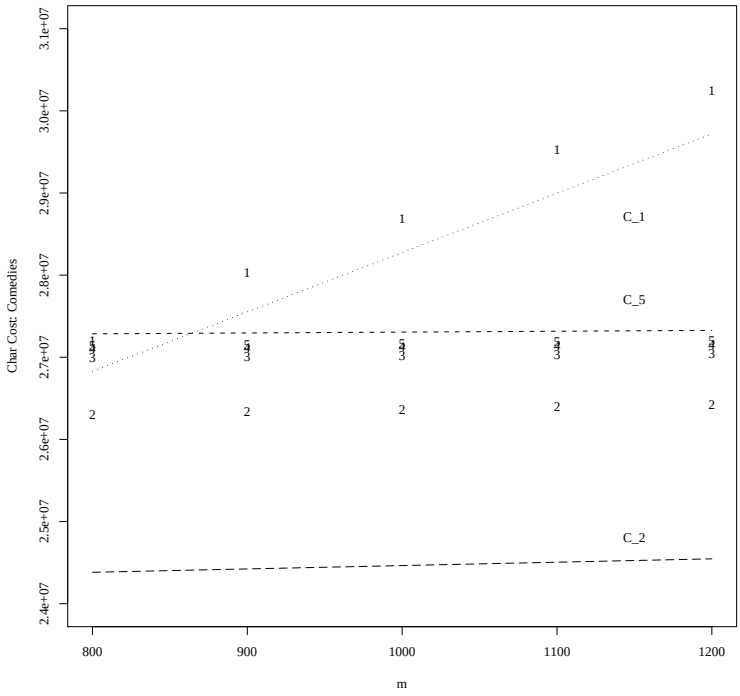

Figure 3.4: **Character Cost.** The graph shows total character cost for one trial of MC on the `comedies.txt` file, of size $n = 377452$, and for $k = 1 \ldots 5$. The three lines show estimates C_1, C_2 and C_5 of these costs obtained from formulas $C'_1(n, m)$, $C'_2(n, m)$ and $C'_{k>5}(n, m)$, respectively.

Where does this formula belong on the instantiation scale? As before, we can argue that any reasonable implementation would produce the same general form and nearly the same coefficients, so $C'_k(n,m)$ supports an algorithmic as well as source-code analysis. On the other hand, dropping n from the third term to better fit the data produces a function that is less likely to work if extended to asymptotic limits. Thus we consider $C'_k(n,m)$ to be an exact function that *describes* rather than *models* algorithm performance. This function should not be applied to predictions outside the parameter bounds of this experiment.

Given the variation of summary coefficients from the raw data, we might expect this formula to describe character cost, to within about 50 percent: Does it work? Figure 3.4 compares character costs for the `comedies` file with estimates (lines) obtained from $C'_k(n,m)$. While there is clearly some constant-factor error from averaging over three text files, the lines are quite close to the data: C_1 differs from the $k = 1$ points by at most 1.5 percent; C_2 differs from $k = 2$ points by less than 8 percent, and C_5 differs from the $k = 3 \ldots 5$ data by no more than 1 percent.

Inspection of $C'_k(n,m)$ reveals that the third term dominates when m is large compared to n, and the first term dominates when m is small (the second term may dominate for some middle values), and the coefficients determine the range of domination. This observation is illustrated in Figure 3.4: the third term which is linear in m dominates C_1, while the first term which is constant in m dominates the other two functions.

As before, experiments with more text files, a wider range of parameter settings, and averages over several random trials would produce better coefficients and increased confidence in the generality of these formulas. As before, we abandon this interesting topic to pursue other experimental goals.

3.1.2 Clocks and Timers

We now leap from the abstract to the instantiated end of our spectrum to consider problems of measuring process time. A process – an actively running program – is highly instantiated because performance depends on a great number of factors, including programmer style, compiler and compiler switches, and properties of the platform and operating system.

We start with a short tutorial on how time is measured in the Unix family of operating systems. For more detailed information about timers and timing see the discussion by Bryant and O'Hallaron [11] (Chapter 9). A short discussion of timing on Windows systems appears at the end of the section.

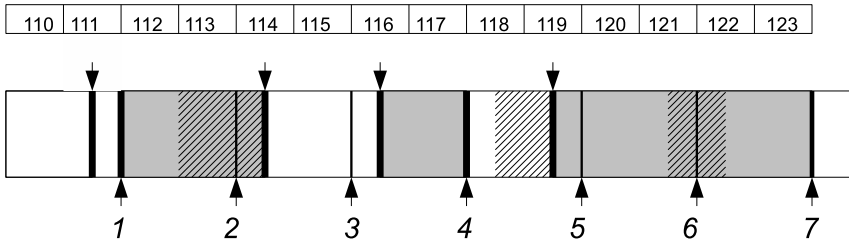

Figure 3.5: **Process Time.** The numbers in boxes show elapsed time according to the system clock. The white and grey processes alternate time in the CPU, together with the process scheduler shown in black. The time spent in system mode is depicted with hashmarks. Seven timer interrupts are shown at the bottom of the CPU band and four external interrupts are above it.

CPU time and elapsed time. Modern operating systems run several processes by interleaving them in time: at any moment one process is using the CPU and the others are waiting their turn. When a CPU interrupt occurs, the active process pauses and the process scheduler takes over; after handling the interrupt the scheduler may decide to let the paused process resume, or to swap out that process and start up a new one. Interrupts may occur for many reasons; in particular a *timer interrupt* is emitted at regular intervals by a timer circuit.

For example, Figure 3.5 shows how a white process and a grey process might alternate use of the CPU, sharing time with the scheduler shown in black. The active process may be runnining in *user mode* – which means it is executing instructions from the program – or in *system mode* – executing operating system instructions such as file input/output on behalf of the program. Time spent in system mode is shown with hashmarks in the figure. Seven timer interrupts are marked by arrows at the bottom of the time band, and four miscellaneous interrupts appear above it. Very roughly, instructions execute at rates near 1 per nanosecond and timer interrupts occur every 10 milliseconds, which means that about 100,000 instructions can be executed between timer interrupts. The scheduler may takes a few microseconds to handle an interrupt and several more to perform a process swap.

Separately from all this, the *system clock*, also called the *time-of-day clock*, increments a clock register at regular intervals. The numbers at the

top of Figure 3.5 show the low order digits of the system clock register over this time period. The time interval used by the system clock is called the *timestamp resolution* of the system; timestamp resolutions vary from nanoseconds to milliseconds depending on the platform.

There are two basic approaches to measuring process time. The first is to calculate *wall clock time*, also called *elapsed time*, by finding the difference between two timestamps of the clock register at the start and end of the process. For example the grey process uses $12 = 123 - 111$ time units in Figure 3.5. The accuracy of an elapsed time measurement depends on timestamp resolution as well as on *timer latency*, which is the time taken during the system call to obtain a timestamp value. Timer latency can be significant: Bryant and O'Halloran [11] report for example that the Java `System.currentTimeMillis()` method may have up to half a millisecond latency, depending on platform.

Cycle counting is a new variation on elapsed time measurement. All Intel x86 platforms since Pentium make available a Time Stamp Counter Register that increments once per instruction cycle. Cycle counting therefore represents a million-fold improvement in resolution and latency over millisecond-scale time-of-day clocks. But it can be difficult to translate cycle counts to time units, because instruction cycle times can vary dynamically depending on factors like battery power and network activity. Also, some CPUs use datapath parallelism to achieve instruction throughputs that are higher than the advertised instruction cycle rate. As a result it can be hard to predict whether executing one billion instructions on a 1Ghz processer would take significantly more or less time than one second.

The obvious problem with elapsed-time measurements of either type is that they may include a substantial amount of "noise" from the scheduler, operating system, and competing processes that has nothing to do with the process being measured.

The second approach is to sample CPU time using *interval timing*: at each timer interrupt, the active process is assigned one credit for the entire interrupt interval. Thus in Figure 3.5 the white process uses 2 units of user time (intervals 1 and 3), and the grey process is recorded as using 5 units of CPU time, 3 in user mode and 2 in system mode. (Note that scheduler time is always assigned to a "nearby" process.) This approach is intended to ignore system noise by measuring only the time a process is active in the CPU, but obviously some degree of statistical sampling error must occur.

Modern processers add new complications to this basic scenario. First, a process may be split into some number of *threads*, or independent instruction streams, that can run concurrently. Some Intel processors use *hyperthreading*, by which two threads (perhaps from different processes) can be inter-

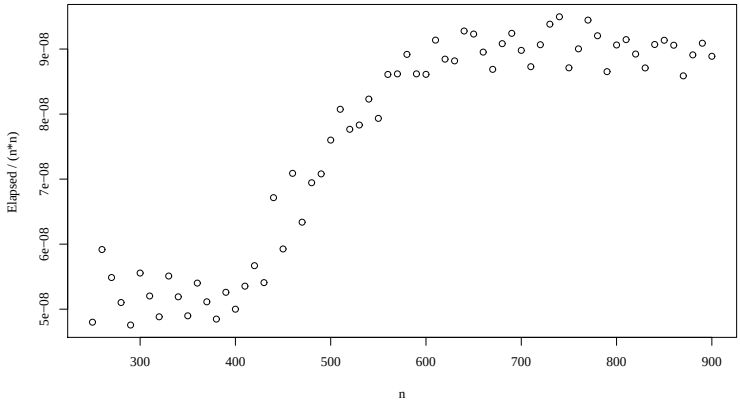

Figure 3.6: **Space and time.** Time-per-instruction depends on the size of the memory accessed by the program. When n is small the data fits in the cache; as data size grows past cache capacity the time per instruction increases.

leaved on one CPU. Both processes are considered to be active in the CPU: an interval timer may allocate 50 percent of the interval to each active process. This can give rise to significant timing inaccuracies, since hyperthreading can appear to double CPU time and halve wall-clock time. Remedies include killing all competing processes, averaging times over several runs, or turning off hyperthreading to obtain more stable measurements.[4]

Also, processors typically contain multiple CPUs – usually 2,4,or 8 *cores* – running in tandem. The elapsed time taken by a process can depend significantly on whether the process is scheduled to run on a busy or an empty core. Timing inaccuracies can arise when a process starts on one core and finishes on another, because clocks and instruction count registers are not perfectly synchronized.

Finally, memory hierarchies affect time measurements in two important ways. First, although a single instruction takes a nanosecond to execute, the time the instruction needs to *get ready* to execute can take several nanoseconds to milliseconds, depending on whether the data it needs is located in registers (fastest), the L1 or L2 cache, main memory, or secondary memory (slowest). Therefore process time depends on the program's memory access pattern.

[4]For more information, visit the online Intel Software Network `software.intel.com/en-us/`, keyword search on hyperthreading process time.

Figure 3.6 demonstrates a frequently-observed phenomenon. The simple C program measured here fills two $n \times n$ matrices with random numbers and then adds them: therefore both the instruction count and the memory size in this program are proportional to n^2. The graph shows that time-per-instruction ($time/n^2$) is not constant, but rather steps up when the three matrices in the program grow too big to fit into cache memory. This type of step can appear whenever program space exceeds the capacity of one of the memories in the hierarchy. (These elapsed time measurements were taken on a lightly loaded Intel Xeon processor see page 103 for details.)

Second, if the same code is run several times, late runs will be faster than early ones because referenced data remains available in fast memories. A *cold start* time measurement is obtained by emptying all memories (or loading with irrelevant data), before starting timing; the alternative, a *warm start* measurement runs the test code at least once before measurement begins.

Note that CPU times are less sensitive to memory effects than are elapsed time measurements, but, as the next section shows, by no means oblivious. [5]

Tips on timing. With this background we perform a small experiment on `markov.c` to compare timing tools. Figure 3.7, shows total CPU times (user mode plus system mode), and wall clock times for eight process-level instantiations of `markov.c` with T =`comedies.txt`, $m = 1,000,000$ and $k = 1$. These times were reported by the Unix `time` command, invoked as follows:

```
time markov 1 1000000 <comedies.txt
```

(The diamonds mark times calculated from cycle counts, described later.) The random number generator seed was fixed so that the instruction sequence is identical throughout all trials. The eight trials correspond to variations in runtime environment, as follows.

- Tests 1 to 4 (labelled `h`) were run on an HP ProLiant DL160 G5, with two Intel Xeon E5472 processors containing 4 CPUs, (totaling 8 CPUS) with 3Ghz instruction cycles, a 12MB L2 cache, and 256 Kb of swap space, running Ubuntu Linux 2.6.24. Tests 5 to 8 (labelled `m`) were run on an Apple MacBookPro2.2 with an Intel Core 2 Dual processor (2 CPUs) running at 2.16GHz, with 4MB L2 Cache, running Mac OSX 10.5.8. The interval timer on both machines has 10ms frequency.

[5] For advanced readers: the L1 cache typically affects CPU time because the pipeline stalls on L1 cache misses. Misses and faults at other levels have less effect on CPU time because they typically generate interrupts which cause the process to be swapped out until the data is ready.

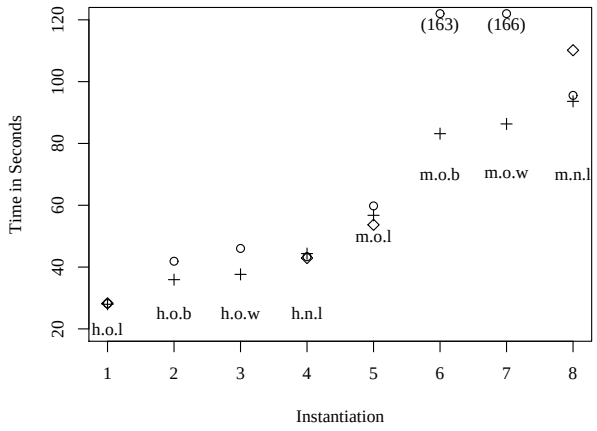

Figure 3.7: **Process Time.** CPU time (crosses) and elapsed time (circles) measured by the Unix `time` command. The diamonds mark times estimated from cycle counts. Note that two outliers (trials 6 and 7) are not plotted to scale.

- Tests 1 to 3 and 5 to 7 (labelled `o`) were compiled with `gcc -O3`, signifying the highest level of code optimization. Tests 5 and 8 (labelled `n`) were compiled with `gcc`, signifying no optimization.

- Tests 1, 4, 5, and 8 (labelled `l`) were run on a lightly loaded system where competing user processes were shut down. The remaining tests were run on a moderately loaded system with 9 copies of the process on the 8-core HP platform, and 3 copies on the 2-core Mac platform. Trials 2 and 3 (`b,w`) show the best and worst times on the loaded HP, and trials 6 and 7 (`b,w`) show the best and worst times on the loaded Mac.

Note that wall clock times in trials 6 and 7 are not plotted to scale. There appeared to be no difference between warm-start and cold-start times in these trials, so this factor was not further evaluated.

Overall, variations in the testing environment contribute about a factor of 8 slowdown in elapsed times (trials 7 and 1), and a factor of 4 slowdown in CPU times (trials 8 and 1). Recall that participants in the DIMACS Challenge observed platform-based slowdowns in CPU times by factors of 10 and more (see page 88).

CPU and elapsed times are quite close when measured on lightly-loaded systems. We observe that moving from an HP to a Mac produces a slowdown ratio near 2.2 for both wall clock and CPU times. (Given the similar architectures but different clock speeds (3Ghz vs 2.16Ghz), we might expect this ratio to be closer to 1.4.) Changing from optimized to unoptimized code produces a slowdown ratio near 1.6 on both systems and both timers.

However these two performance indicators diverge when the system load changes, and elapsed times are more sensitive than are CPU times. Increasing the load by a factor of X decreases elapsed time by more than a factor of X: for example, 9 copies on the 8-core HP should produce wall clock times near $35 = 28(9/8)$ seconds, but observed times are at least 40 seconds in trials 2 and 3. Running 3 processes on the 2-core Mac should raise wall clock times to 90 seconds, but times of 163 seconds and more are observed (trials 6 and 7). CPU times are not completely unaffected by system load: the program took up to 37.6 seconds on the loaded HP (trial 3), and 86.3 seconds on the loaded Mac (trial 7).

Figure 3.7 also shows the results of four cycle time measurements (diamonds), for the lightly-loaded trials 1,4 and 5,8. In this experiment the assembly language `RDTSC` (Read Time Stamp Counter) instruction was invoked from inside `markov.c` at the beginning and end of the program. Bryant and O'Hallaron [11] (Section 9.4) point out that the smallest cycle counts recorded over several runs of a process must be the most accurate, and propose a K-Best measurement strategy that involves recording the smallest count observed in repeated trials. The times shown here represent the minimum cycle counts observed over 3 trials. (See [11] (Section 9.3) for C code to access this counter register.) Cycle times were calculated by multiplying elapsed instruction counts by advertised instruction frequencies, 3Ghz for the HP and 2.16Ghz for the Mac. A small validation experiment showed that measured instruction frequencies were indeed within two percent of the advertised rates.

On the HP platform, cycle times are consistent with wall clock times (trials 1 and 4). However cycle times on the Mac platform are difficult to reconcile: in trial 5, cycle time is a full second *less* than both wall clock and CPU time, but in trial 8 cycle time is about 15 seconds *more* than wall clock time. Repeated measurements on this platform give (similarly) inconsistent results, showing variations by as much as 30 percent in some cases.

Here are some guidelines for measuring process time in Unix-style operating systems. Windows systems have no direct analog to interval counting, although Windows NT does provide the `GetThreadTime` and `GetProcessTime` system functions which apparently report a similar statistic. Many Windows systems also provide a CPU Usage function which reports the percentage of

CPU times used by each active process within a timing interval. Several tools for elapsed time measurement are available: See Wilson [6] for good survey.

Use CPU times to measure processes that take at least 0.1 second. As a general rule, algorithmic researchers prefer interval timing to elapsed time measurements because CPU times focus on properties intrinsic to the algorithm or program, and are easier to translate from one test environment to another.

Also, most interesting algorithms take at a few seconds to run. As process time increases, CPU times become more accurate due to averaging of interval sampling errors, while elapsed times become less reliable due to interrupts and system activity. A good rule of thumb is to choose experimental parameters so that the measured process takes at least 0.1 second, or 10 times longer than the standard 10ms timer interval: 1 second or more is better. If the measured process takes less than a few timer intervals to run, CPU times are vulnerable to sampling error; time-of-day measurements can also be unreliable if the process time is close to timestamp resolution and/or timer latency. Cycle counting is best for process times below this threshold.

It is possible to measure time spent by interesting process *components* (such as functions or data structure updates) by invoking timers from inside a program; the above rule applies to measuring process components as well.

The Unix `sysconf()` and `clock_getres(()` can be invoke to find the interrupt timer interval and system timestamp resolution on a given platform. See also the `time(7)` manual page.

Always measure time on lightly loaded systems. As we have seen, both elapsed times and CPU times respond to system load. For best results, kill competing applications and background processes, and avoid generating keystroke interrupts and screen update events on personal computers and laptops. Although this ideal competition-free scenario cannot always be reached, good results can be obtained on multi-core systems if at least one idle core is available to run the measured process.

Finally, as we have observed, timing results are highly context-specific. The reader – or future user of your code – may have little choice about which platform to use, but likely can take steps to tune the runtime environment for best performance. Follow these guidelines to ***maximize the generality of published timing results*** and help the reader translate your results to a new situation.

1. **Full disclosure.** Report all properties of the testing environment that might affect time measurements. These include the instruction cycle rate, the compiler and compiler optimization level, cache and main

memory sizes, the system load during measurement, and what timing tool was used.

2. **Platform variety.** Time measurements on several plaforms are more informative than measurements on one platform. A good basic strategy is to report times on the fastest and the slowest platforms available to you.

3. **Best-case scenario.** Run tests on a given platform that is tuned for fastest process times: set compilier optimization switches to maximum levels, test on a lightly loaded system, and use warm start measurements when possible. This produces less variation in your timing statistics and gives a lower bound on times for the given platform, which may be easier for the reader to interpret.

3.1.3 Code Profilers

Time for a thought experiment: Given the experimental results so far, how closely can we estimate the time needed by `markov.c` to generate $m = 100,000$ words with $k = 1$, according to probabilities based on the works of Jane Austen ($n = 204,893$), using optimized code on the Mac platform?

Formulas $C(n, m)$ and $C_k(n, m)$ tell us the algorithm would perform about 166.9 million word comparisons and 733.3 million character comparisons for these parameter values. This translates to at least 0.339 seconds of CPU time on a 2.16 Ghz Mac platform. The timing statistics show that optimized code takes about 60 CPU seconds on this platform when $m = 1,000,000$, $n = 377,543$, and $k = 1$ (using $T =$`comedies.txt`); the Austen trial would presumably take less time since n and m are smaller. So far we have narrowed the time estimate to between 0.339 and 60 CPU seconds.

This section shows how *code profiling* can be used to further narrow the gap. A code profiler is a software tool that reports execution counts for program components such as functions, lines of source code, and/or machine instructions. Several commercial and open source profilers are available – *Valgrind*, for example, is a suite of Open Source profiling tools that measure many aspects of program performance. Here we illustrate the *Cachegrind* tool, which reports machine instruction execution counts. Visit `valgrind.org` for more information about Valgrind and Cachegrind.

Measuring code blocks. A *code block* is a collection of instructions that are executed about the same number of times. In this section we will measure blocks of machine code associated with the `wordcmp` function, shown below.

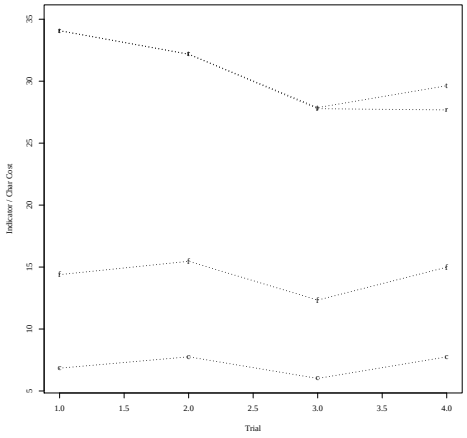

Figure 3.8: The dotted lines show total with IO, total without IO, function and character instruction counts, divided by character cost.

```
1 int (wordncmp(char *p , char* q) {
2    int n=k;
3    for ( ; *p == *q;  p++,q++)
4         if (*p == 0 && --n == 0)
5           return 0;
6    return *p - *q;
 }
```

Recall that *character cost*, denoted C here, is the number of character operations performed in the `for` loop. Let *mchar* denote the number of machine code instructions executed in that loop header. Inspection of the compiled code reveals that the loop header code performs 6 instructions the first time (when increment is not performed) and 8 instructions on subsequent iterations. So we expect $6c \leq mchar \leq 8C$. Let *mfunction* denote the number of machine code instructions executed in the `wordcmp` function on lines 1 through 6, and let *mtotal* denote the total number of instructions executed in the program.

Each of these performance indicators is correlated with C and also with CPU time (denoted T). We expect smaller blocks of code to correlate more closely with C, and larger blocks to correlate more closely with T. Thus we can move between abstract and instantiated performance indicators by adjusting the size of code blocks.

The next experiment demonstrates this correlation. Figure 3.8 shows the four block costs `mtotal` (labelled `t`), *mfunction* (`f`), and *mchar* (`c`),

all divided by character cost C, in trials that use *comedies.txt* and four combinations of parameter settings. Trials 1,2 correspond to $m = 1000$ and trials 3,4 to $m = 100,000$. Trials 1,3 corresond to $k = 1$ and trials 2,4 to $k = 5$.

The machine instruction counts were produced by Cachegrind using command sequences like that shown below. The `gcc -g` compiler option activates the insertion of debugging trace code by the compiler. Valgrind simulates a run of the program and produces a data file tagged with a process number (1234 in this example). The data file is read by `cg_annotate` which produces a human-readible text; the `--auto=yes` option specifies the output format to be a labelled source code listing.

```
gcc -g  markov.c -o mc
valgrind --tool=cachegrind mc 1 1000 <comedies.txt
cg_annotate cachegrind.out.1234 --auto=yes
```

The data in Figure 3.8 represents unoptimized code. As a general rule, code profilers should not be combined with compiler optimization because the optimizer moves code around in ways that confuse the trace code, producing errors in counts associated with source code lines. Character costs were measured in separate runs with counter code inserted; the random number generator seed was fixed so that the instruction sequence was identical in corresponding tests.

The figure shows that, as expected, `mchar` is near $7.1C$, slightly more when $k = 1$ and slightly less when $k = 5$. We can estimate `mfunction` with the mean $14.3C$.

As a first step in our estimation project we combine these coefficients with $C_1'(n, m)$, our formula for character cost (see page **??**), using parameters $n = 204,839$ and $m = 100,000$ for the `austen` file. The resulting estimates are compared below to instruction counts produced by Cachegrind.

estimate	perf. indicator	estimate	observed	error
$C' = C_1'(n, m)$	char cost	733m	733m	¡ 1 %
$7.1C'$	mchar	5,206m	4,423m	¡ 2 %
$14.3C'$	mfunction	10,486m	9,074m	¡ 14 %
$T' = 30.5C'$	mtotal	22,366m	20,501m	¡ 10 %

Note that the observed variation of `mtotal` in Figure 3.8 suggests that T' could range between $T'_{hi} = 27,487$ million and $T'_{lo} = 15,225$ million instructions. Our next step is to convert instruction counts to CPU times. We multiply T' by instruction frequency 3Ghz to obtain an estimate of 7.46 seconds for unoptomized code on the the HP platform. Comparing timing results from Figure 4.7, we might expect optimized code on the Mac platform

to run about 1.27 times slower than unoptimized code on the HP platform, which produces a time estimate of 9.47 seconds.

In fact the optimized code runs in 3.84 CPU seconds, about 2.5 times faster than the estimate. Of course several layers of inaccuracy contribute to increase this error from the 10 percent or so observed in predicting instruction counts, including: a different input text file; different instruction counts on different machines; different instruction counts for optimized code; and CPU time measurements that do not scale. However the estimate is within the correct order of magnitude and much closer than our original guess between 0.339 seconds and 60 seconds, which spans two orders of magnitude.

3.1.4 Other Time Performance Indicators

Several other performance indicators are available to the experimenter besides those discussed in previous sections. Here is a short survey of tools and techniques.

- The Unix `prof` and `gprof` tools can be used to report CPU times and invocation counts for every function executed in a given program. The `profil()` function can be invoked to measure CPU times for blocks of code that are marked by the programmer in the source code. The `gettimeofday()` function can be used to obtain timestamps for sections of code within a program.

- Some code profilers report line and/or block counts for source code instructions rather than machine code. The GNU `gcov` tool can generate execution counts for either type. Source code line counting is sometimes called coverage analysis, used during software testing to discover which parts of a program are exercised by a given suite of test instances. Note that instruction counters tied to source code generally cannot be used together with compiler optimizers.

- Java code profiling is complicated by the presence of the garbage collector and by the fact that object code runs on a virtual machine. In JDK, the `-verboseusage` flag can be used for separating out time spent during garbage collection. The profiler invoked by `-Xrunhprof` can be modified to produce statistics about methods using a form of interval timing known as *call stack sampling*.

- It is certainly possible to custom-profile sections of code by inserting counters for interesting code blocks. Ahuja, Magnanti, Orlin [**?**] (Chapter 18) describe a process for identifying *representative operation counts* tied to a small set of code blocks that dominate the computation

time. Block counts are used to identify *bottleneck operations* that account for the largest proportion of computaton time. The authors also show how to attach timing statistics to bottleneck operation counts to obtain predictions of CPU times that are portable among platforms.

- Experimental studies focusing on the memory hierarchy require performance indicators that capture those interactions rather than instruction counts or CPU times. Many tools for profiling and simulating memory access are available. Cachegrind, records memory reference counts as well as cache hits and misses tied to each line of source code. Other tools generally fall into two categories: a *trace driven simulator* runs the program and makes a record of all memory references, producing a file for subsequent analysis; and a *memory monitor* periodically checks the memory state by a method similar to interval sampling. One property of the trace driven approach is that programs are measured in isolation rather than in competition with other processes: this may be a drawback or an advantage depending on the purpose of the experiment. One problem with memory monitors is that their own use of memory interferes with process measurement.

- Knuth [4] (pp. 464-467) proposes a method for introducing counters, called *mems*, by hand to tabulate memory access costs. He argues that memory access counts are more platform-portable and provide more accurate predictions of real computation time than are instruction counts. His approach involves inserting counters for every instruction that causes a main memory (rather than register) access. Perfect mem-counting requires deep understanding of compilers and architectures, but Knuth argues that approximate counts are adequate for most situations. An example of this approach may be found by visiting the Stanford GraphBase [6] and reading the text of the MILES_SPAN program.

Finally we note that evaluation of algorithms for parallel and distributed systems introduce new levels of complexity in time measurement. As a general rule the important question is not so much "How much time does it take?" but rather "How much does adding processors improve running time?" Thus measurements of *speedup* due to concurrency are preferred. Speedups typically can't be measured from inside the code, and processer synchronization and scheduling policies can wreak havoc with monitor-based timing strategies. Simulators and monitors can give good information about performance on particular platforms; however a wide variety of architectures

[6] www-cs-faculty.stanford.edu/ knuth/sgb.html

and communication topologies exist, and such measurements are very difficult to scale or translate to other systems. See Bader, Moret, and Sanders [1] for a thorough survey of difficulties that arise in measuring time performance on concurrent systems, see [1].

Despite these hazards, time measurements are sometimes needed. The best general strategy may be to measure total elapsed time, especially when distributed components and communication patterns are irregular. To perform such a measurement: synchronize all processors and obtain a timestamp, then run the (distributed) process, synchronize processors again, and obtain a second timestamp. Here is a list of commonly used performance metrics and their meanings.

- The *absolute speedup* A_p is the total time T of the fastest known sequential algorithm, to the total time T'_p of a given parallel (or distributed) algorithm running on p processors: that $A_p = T/T'_p$. The sequential and parallel algorithms need not be similar.

- If a concurrent algorithm is parameterized to run on p processors, the *relative speedup* R_P, is the ratio of time on a p-processor implementation to time on a 1-processor version. That is, relative speedup is $R_p = T'_1/T'_p$ on one system.

- The *efficiency* of a parallel algorithm is the ratio of the relative speedup to the number of processors it uses. That is, $Eff(p) = R_p/p$. While it seems reasonable to assume that efficiency can never be greater than one – doubling the processors should not make the program run more than twice as fast – such non-intuitive behavior is not uncommon. Superlinear speedups may occur because of changes in memory capacity or because of dependencies between processor counts and input sizes.

3.1.5 The Right Tool For the Job

We have surveyed several approaches to time performance measurement; now we develop some guidelines for choosing a performance indicator best suited to experimental goals. These guidelines are organized around three principles: generality, analyzability, and relevance.

The instantiation rule: Balance abstraction with precision. *Match the performance indicator to the most abstract instantiation level possible.* The more abstract the performance indicator, the more general and portable are your research conclusions.

Are you trying to understand whether your algorithm is asymptotically cubic or sublinear? This is an abstract property independent of programs

and platforms: choose a similarly abstract performance indicator such as word count or character count in our example experiment.

Of course there is no such thing as perfect abstraction. No matter what level of instantiation is intended, measurements always take place on a specific programs and processes. However we can accept an operation count as a reflecting an abstract cost if it can be argued that any reasonable implementation would have the same asymptotic cost: i.e. that differences in counts among implementations would only be reflected in coefficients and low-order terms.

Suppose your goal is to engineer a fast text generation program that will be placed in a public repository and run on a variety of (unknown) platforms and inputs. The source program is common to these instantiations: focus on performance indicators tied to code strategies, analagous to character counts, or instruction block counts in our example. Measurements of code properties are less generalizable than measurements of algorithm properties, but more closely tied to time performance. Thus they represent a midpoint between the abstract and instantiated ends of our spectrum.

If the objective is to win a contest with the fastest code possible on a given platform, then machine instruction counts, CPU times, and measurements of memory access costs are appropriate.

If the most abstract performance indicator is not precise enough to give useful results, then dial up the precision.

- *Increase generality by adding variety.* Implement the algorithm several ways, or insert counters in a single implementation that measure alternative costs.

- *Prefer to measure best case scenarios.* Choose the best implementation of the algorithm, or introduce counters to record that cost, whether or not the program actually is implemented that way. A lower bound on operation counts can be analyzed more

The Kiss rule: Abstract performance indicators are easier to model than instantiated performance indicators. Some quantities are easier to analyze than others; select performance indicators of suitable complexity to match your experimental goals. Data sets showing high variance and complicated interactions produce less reliable results than data sets displaying low variance and simple functional relationships.

If your plan involves extrapolating well beyond the boundaries of your experiment – for example to estimate asymptotic performance, or performance on untested systems – then a simple, smoothly-changing performance indicator based on model with a sound theoretical foundatation will be more suc-

cessful than noisy performance indicators measuring complex interactions. Sometimes this means breaking a total cost into simple components, as we saw in the word cost model. The appropriate strategy was not to measure total comparisons, but rather three components which could be described by simple terms.

Character cost is more sensitive to some properties of input files number of duplicate keys, length of matches in duplicate keys. The "descriptive" function $C'_k(n, m)$ can't be extended with confidence beyond the boundaries of the experiment. Nevertheless the function serves as a predictive tool within those bounds.

Block-based instruction counts can move from more simple to less simple as their scope grows.

CPU times are approximately based on instruction counts, and wall clock times are least simple to model and understand.

Don't draw conclusions that can't be supported within the limits of precision of the performance indicator.

Maximize relevance and comparbility. *Choose a performance indicator that focuses on relevant costs and ignores irrelevant costs.* The quantity you choose to measure should focus exactly on the property you are interested in measuring, nothing more, nothing less. Some additional principles – some overlapping, and some conflicting with one another – are listed below.

The first step is to decide which properties are *intrinsic,* and which are *extrinsic,* to your research question.

This is about instantiation and also focus. Do you want to evaulate the sorted list data structure for MC? Or the MC algorithm with a sorted data structure? Or the efficiency of your operating system when it is running the MC algorithm?

Choose performance indicators that allow direct comparison to the literature. Past and future. It should be possible to compare your results directly to those in existing literature on the problem; just as importantly, future researchers should be able to build upon the groundwork you have laid. The unhappy alternative is early obscelescence in published algorithmic research: readers have no way of converting your results to incorporate new information, and all your hard work must be deprecated.

To engineer a program to support your Internet-based random text generation server. If the code is intended for a network application you may want to focus on costs related to network communication costs instead of instruction counts and CPU times.

A count of comparisons will last until computer architectures change in deeply fundamental ways; but a reported CPU time of 3.54 user seconds

may last no more than a couple years. If the community has choosen poorly, report both.

Choose performance indicators that allow direct comparisons of alternatives. To learn if one type of balanced binary tree is better than another, choose performance indicators that isolate the costs related to binary search trees: number of node accesses, number of link updates, et cetera. If your two options are incomparable at an abstract level, then you may may have to move down the instantiation scale to find a common denominator. For example, "node accesses" would not necessarily be appropriate for comparing a tree to a hash table: instead it may be necessary to some operation common to both such as main memory accesses or instruction executations.

Here are some guidelines for selecting a performance indicator to *compare* two algorithms, data structures, or implementation strategies, which we can apply to the hash table vs sorted array experiment.

First, be clear about the goal of the experiment: do you want to know (1) if the hash table is faster than the sorted array, or (2) if the program runs faster with a hash table than with a sorted array. The first question is appropriate to an experiment to evaluate the merits of two data structure alternatives: in this case the performance indicator should *isolate* the costs that distinguish the alternatives, and ignore costs that are common to both programs. This rules out *total line cost* as an ideal performance indicator, because it incorporates information that is irrelevant to a direct comparison of hash tables and arrays. The second question is appropriate to an engineering project to find the fastest code for the application, where it is important both to know which alternative is better, and also how much the change affects overall computation time. In this scenario total line cost might be preferred.

Second, the performance indicator, when possible, should measure an operation that has equal cost in both alternatives. This criterion rules out *word cost*. At the word-cost level, we must compare `wordcmp` invocations in `markov.c` to `hash` + `worcmp` invocations in `markovhash.c`. Such a comparison is quite problematic without more information about the relative costs of hashing and word comparisons. Measuring character operations instead of word operations is likely to produce more satisfying results, because character operations are likely to have the same cost in both implementations.

Source code lines, of course, can produce varying numbers of machine code instructions, and the variation has more to do with programmer style than with properties of the two data structures. Of course, not all lines of source code have the same cost: a `for` loop header, for example, is likely to be compiled into at least three machine instructions, while another line might require only one machine instruction. Therefore source instruction counts

are off by a (hopefully small) constant factor when it comes to predicting machine instruction counts. Furthermore, machine instruction counts are not as precise as might be expected in predicting program running times, which can be affected by pipeline stalls, and irregularities of access to the memory hierarchy. However source code line counts do have the advantage of being platform-independent, more precise than dominant cost counts, and precise enough for many experimental tasks.

To compare several alternative data structures. Choose a performance indicator that isolates the costs associated with data structure performance, to allow direct and focused comparison.

The hash table: find something that you can compare:

- insert in an array, then sort. Vs insert in a hash table
- lookup binary search vs hashing
- random selection: contiguous subarray vs open addressed. (Even with chaining there will be collisions.)

Word operations vs hash operations. Character comparisons vs character operations in hashing. Hashing is necessarily more instantiated because the hash function is tuned to properties of the text (if it is a decent hash function. Which is the "real hash function." Can try boundaries: maximum hashing cost and minimum collision. Or varieties.

If you are interested in comparing different data structures then measure properties. properties tied directly to data structure performance – number of update operations and costs per operation – are intrinsic. Properties that do not change when one data structure is substituted for another, such as input/output costs, and other houskeeping costs, are extrinsic. If, on the other hand, your goal is to tune an implementation for best performance on three different platforms, then profilier statistics, and measurements of bottleneck operations, and some types of memory reference patterns are most likely to give you the answers you need.

3.2 Coping with NP-Hard Problems

Perhaps the second most frequently measured property of algorithm performance – after time – is *solution quality*. This metric is of critical interest in research on approximation algorithms and heuristics for NP-Hard problems. (Readers unfamiliar with this term should jump to the brief introduction to NP-Completeness Theory in Appendix **??**.)

While the the choice of performance indicator seems pretty straightforward – just measure the cost defined by the problem – some performance