CHAPTER ONE

Introduction

Tim Roughgarden

Abstract: One of the primary goals of the mathematical analysis of algorithms is to provide guidance about which algorithm is the "best" for solving a given computational problem. Worst-case analysis summarizes the performance profile of an algorithm by its worst performance on any input of a given size, implicitly advocating for the algorithm with the best-possible worst-case performance. Strong worst-case guarantees are the holy grail of algorithm design, providing an application-agnostic certification of an algorithm's robustly good performance. However, for many fundamental problems and performance measures, such guarantees are impossible and a more nuanced analysis approach is called for. This chapter surveys several alternatives to worst-case analysis that are discussed in detail later in the book.

1.1 The Worst-Case Analysis of Algorithms

1.1.1 Comparing Incomparable Algorithms

Comparing different algorithms is hard. For almost any pair of algorithms and measure of algorithm performance, each algorithm will perform better than the other on some inputs. For example, the MergeSort algorithm takes $\Theta(n \log n)$ time to sort length-n arrays, whether the input is already sorted or not, while the running time of the InsertionSort algorithm is $\Theta(n)$ on already-sorted arrays but $\Theta(n^2)$ in general.[1]

The difficulty is not specific to running time analysis. In general, consider a computational problem Π and a performance measure PERF, with PERF(A, z) quantifying the "performance" of an algorithm A for Π on an input $z \in \Pi$. For example, Π could be the Traveling Salesman Problem (TSP), A could be a polynomial-time heuristic for the problem, and PERF(A, z) could be the approximation ratio of A – i.e., the ratio of the lengths of A's output tour and an optimal tour – on the TSP instance z.[2]

[1] A quick reminder about asymptotic notation in the analysis of algorithms: for nonnegative real-valued functions $T(n)$ and $f(n)$ defined on the natural numbers, we write $T(n) = O(f(n))$ if there are positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$; $T(n) = \Omega(f(n))$ if there exist positive c and n_0 with $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$; and $T(n) = \Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

[2] In the Traveling Salesman Problem, the input is a complete undirected graph (V, E) with a nonnegative cost $c(v, w)$ for each edge $(v, w) \in E$, and the goal is to compute an ordering $v_1, v_2, \ldots, v_n$ of the vertices V that minimizes the length $\sum_{i=1}^{n} c(v_i, v_{i+1})$ of the corresponding tour (with v_{n+1} interpreted as v_1).

Or Π could be the problem of testing primality, A a randomized polynomial-time primality-testing algorithm, and $\text{PERF}(A, z)$ the probability (over A's internal randomness) that the algorithm correctly decides if the positive integer z is prime. In general, when two algorithms have incomparable performance, how can we deem one of them "better than" the other?

Worst-case analysis is a specific modeling choice in the analysis of algorithms, in which the performance profile $\{\text{PERF}(A, z)\}_{z \in \Pi}$ of an algorithm is summarized by its worst performance on any input of a given size (i.e., $\min_{z\,:\,|z|=n} \text{PERF}(A, z)$ or $\max_{z\,:\,|z|=n} \text{PERF}(A, z)$, depending on the measure, where $|z|$ denotes the size of the input z). The "better" algorithm is then the one with superior worst-case performance. MergeSort, with its worst-case asymptotic running time of $\Theta(n \log n)$ for length-n arrays, is better in this sense than InsertionSort, which has a worst-case running time of $\Theta(n^2)$.

1.1.2 Benefits of Worst-Case Analysis

While crude, worst-case analysis can be tremendously useful and, for several reasons, it has been the dominant paradigm for algorithm analysis in theoretical computer science.

1. A good worst-case guarantee is the best-case scenario for an algorithm, certifying its general-purpose utility and absolving its users from understanding which inputs are most relevant to their applications. Thus worst-case analysis is particularly well suited for "general-purpose" algorithms that are expected to work well across a range of application domains (such as the default sorting routine of a programming language).
2. Worst-case analysis is often more analytically tractable to carry out than its alternatives, such as average-case analysis with respect to a probability distribution over inputs.
3. For a remarkable number of fundamental computational problems, there are algorithms with excellent worst-case performance guarantees. For example, the lion's share of an undergraduate algorithms course comprises algorithms that run in linear or near-linear time in the worst case.[3]

1.1.3 Goals of the Analysis of Algorithms

Before critiquing the worst-case analysis approach, it's worth taking a step back to clarify why we want rigorous methods to reason about algorithm performance. There are at least three possible goals:

1. *Performance prediction.* The first goal is to explain or predict the empirical performance of algorithms. In some cases, the analyst acts as a natural scientist, taking an observed phenomenon such as "the simplex method for linear programming is fast" as ground truth, and seeking a transparent mathematical model that explains it. In others, the analyst plays the role of an engineer, seeking a theory that

[3] Worst-case analysis is also the dominant paradigm in complexity theory, where it has led to the development of *NP*-completeness and many other fundamental concepts.

gives accurate advice about whether or not an algorithm will perform well in an application of interest.
2. *Identify optimal algorithms.* The second goal is to rank different algorithms according to their performance, and ideally to single out one algorithm as "optimal." At the very least, given two algorithms A and B for the same problem, a method for algorithmic analysis should offer an opinion about which one is "better."
3. *Develop new algorithms.* The third goal is to provide a well-defined framework in which to brainstorm new algorithms. Once a measure of algorithm performance has been declared, the Pavlovian response of most computer scientists is to seek out new algorithms that improve on the state-of-the-art with respect to this measure. The focusing effect catalyzed by such yardsticks should not be underestimated.

When proving or interpreting results in algorithm design and analysis, it's important to be clear in one's mind about which of these goals the work is trying to achieve.

What's the report card for worst-case analysis with respect to these three goals?

1. Worst-case analysis gives an accurate performance prediction only for algorithms that exhibit little variation in performance across inputs of a given size. This is the case for many of the greatest hits of algorithms covered in an undergraduate course, including the running times of near-linear-time algorithms and of many canonical dynamic programming algorithms. For many more complex problems, however, the predictions of worst-case analysis are overly pessimistic (see Section 1.2).
2. For the second goal, worst-case analysis earns a middling grade – it gives good advice about which algorithm to use for some important problems (such as many of those in an undergraduate course) and bad advice for others (see Section 1.2).
3. Worst-case analysis has served as a tremendously useful brainstorming organizer. For more than a half-century, researchers striving to optimize worst-case algorithm performance have been led to thousands of new algorithms, many of them practically useful.

1.2 Famous Failures and the Need for Alternatives

For many problems a bit beyond the scope of an undergraduate course, the downside of worst-case analysis rears its ugly head. This section reviews four famous examples in which worst-case analysis gives misleading or useless advice about how to solve a problem. These examples motivate the alternatives to worst-case analysis that are surveyed in Section 1.4 and described in detail in later chapters of the book.

1.2.1 The Simplex Method for Linear Programming

Perhaps the most famous failure of worst-case analysis concerns linear programming, the problem of optimizing a linear function subject to linear constraints (Figure 1.1). Dantzig proposed in the 1940s an algorithm for solving linear programs called the *simplex method*. The simplex method solves linear programs using greedy local

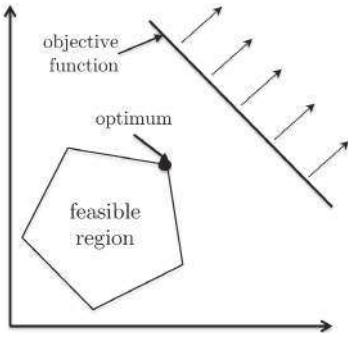

Figure 1.1 A two-dimensional linear programming problem.

search on the vertices of the solution set boundary, and variants of it remain in wide use to this day. The enduring appeal of the simplex method stems from its consistently superb performance in practice. Its running time typically scales modestly with the input size, and it routinely solves linear programs with millions of decision variables and constraints. This robust empirical performance suggested that the simplex method might well solve every linear program in a polynomial amount of time.

Klee and Minty (1972) showed by example that there are contrived linear programs that force the simplex method to run in time exponential in the number of decision variables (for all of the common "pivot rules" for choosing the next vertex). This illustrates the first potential pitfall of worst-case analysis: overly pessimistic performance predictions that cannot be taken at face value. The running time of the simplex method is polynomial for all practical purposes, despite the exponential prediction of worst-case analysis.

To add insult to injury, the first worst-case polynomial-time algorithm for linear programming, the ellipsoid method, is not competitive with the simplex method in practice.[4] Taken at face value, worst-case analysis recommends the ellipsoid method over the empirically superior simplex method. One framework for narrowing the gap between these theoretical predictions and empirical observations is *smoothed analysis*, the subject of Part Four of this book; see Section 1.4.4 for an overview.

1.2.2 Clustering and *NP*-Hard Optimization Problems

Clustering is a form of unsupervised learning (finding patterns in unlabeled data), where the informal goal is to partition a set of points into "coherent groups" (Figure 1.2). One popular way to coax this goal into a well-defined computational problem is to posit a numerical objective function over clusterings of the point set, and then seek the clustering with the best objective function value. For example, the goal could be to choose k cluster centers to minimize the sum of the distances between points and their nearest centers (the k-median objective) or the sum of the squared

[4] Interior-point methods, developed five years later, led to algorithms that both run in worst-case polynomial time and are competitive with the simplex method in practice.

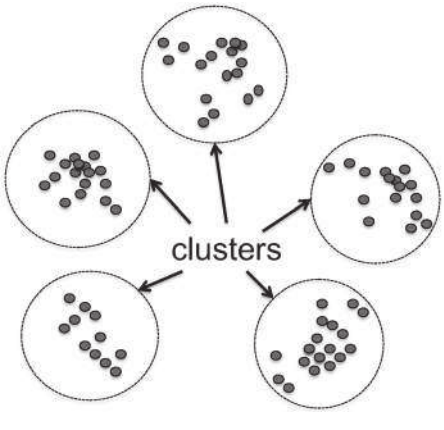

Figure 1.2 A sensible clustering of a set of points.

such distances (the k-means objective). Almost all natural optimization problems that are defined over clusterings are NP-hard.[5]

In practice, clustering is not viewed as a particularly difficult problem. Lightweight clustering algorithms, such as Lloyd's algorithm for k-means and its variants, regularly return the intuitively "correct" clusterings of real-world point sets. How can we reconcile the worst-case intractability of clustering problems with the empirical success of relatively simple algorithms?[6]

One possible explanation is that *clustering is hard only when it doesn't matter*. For example, if the difficult instances of an NP-hard clustering problem look like a bunch of random unstructured points, who cares? The common use case for a clustering algorithm is for points that represent images, or documents, or proteins, or some other objects where a "meaningful clustering" is likely to exist. Could instances with a meaningful clustering be easier than worst-case instances? Part Three of this book covers recent theoretical developments that support an affirmative answer; see Section 1.4.2 for an overview.

1.2.3 The Unreasonable Effectiveness of Machine Learning

The unreasonable effectiveness of modern machine learning algorithms has thrown the gauntlet down to researchers in algorithm analysis, and there is perhaps no other problem domain that calls out as loudly for a "beyond worst-case" approach.

To illustrate some of the challenges, consider a canonical supervised learning problem, where a learning algorithm is given a data set of object-label pairs and the goal is to produce a classifier that accurately predicts the label of as-yet-unseen objects

[5] Recall that a polynomial-time algorithm for an NP-hard problem would yield a polynomial-time algorithm for every problem in NP – for every problem with efficiently verifiable solutions. Assuming the widely believed $P \neq NP$ conjecture, every algorithm for an NP-hard problem either returns an incorrect answer for some inputs or runs in super-polynomial time for some inputs (or both).

[6] More generally, optimization problems are more likely to be NP-hard than polynomial-time solvable. In many cases, even computing an approximately optimal solution is an NP-hard problem. Whenever an efficient algorithm for such a problem performs better on real-world instances than (worst-case) complexity theory would suggest, there's an opportunity for a refined and more accurate theoretical analysis.

(e.g., whether or not an image contains a cat). Over the past decade, aided by massive data sets and computational power, neural networks have achieved impressive levels of performance across a range of prediction tasks. Their empirical success flies in the face of conventional wisdom in multiple ways. First, there is a computational mystery: Neural network training usually boils down to fitting parameters (weights and biases) to minimize a nonconvex loss function, for example, to minimize the number of classification errors the model makes on the training set. In the past such problems were written off as computationally intractable, but first-order methods (i.e., variants of gradient descent) often converge quickly to a local optimum or even to a global optimum. Why?

Second, there is a statistical mystery: Modern neural networks are typically overparameterized, meaning that the number of parameters to fit is considerably larger than the size of the training data set. Overparameterized models are vulnerable to large generalization error (i.e., overfitting), since they can effectively memorize the training data without learning anything that helps classify as-yet-unseen data points. Nevertheless, state-of-the-art neural networks generalize shockingly well – why? The answer likely hinges on special properties of both real-world data sets and the optimization algorithms used for neural network training (principally stochastic gradient descent). Part Five of this book covers the state-of-the-art explanations of these and other mysteries in the empirical performance of machine learning algorithms.

The beyond worst-case viewpoint can also contribute to machine learning by "stress-testing" the existing theory and providing a road map for more robust guarantees. While work in beyond worst-case analysis makes strong assumptions relative to the norm in theoretical computer science, these assumptions are usually weaker than the norm in statistical machine learning. Research in the latter field often resembles average-case analysis, for example, when data points are modeled as independent and identically distributed samples from some underlying structured distribution. The semirandom models described in Parts Three and Four of this book serve as role models for blending adversarial and average-case modeling to encourage the design of algorithms with robustly good performance.

1.2.4 Analysis of Online Algorithms

Online algorithms are algorithms that must process their input as it arrives over time. For example, consider the online paging problem, where there is a system with a small fast memory (the cache) and a big slow memory. Data are organized into blocks called *pages*, with up to k different pages fitting in the cache at once. A page request results in either a cache hit (if the page is already in the cache) or a cache miss (if not). On a cache miss, the requested page must be brought into the cache. If the cache is already full, then some page in it must be evicted. A cache replacement policy is an algorithm for making these eviction decisions. Any systems textbook will recommend aspiring to the Least Recently Used (LRU) policy, which evicts the page whose most recent reference is furthest in the past. The same textbook will explain why: Real-world page request sequences tend to exhibit locality of reference, meaning that recently requested pages are likely to be requested again soon. The LRU policy uses the recent past as a prediction for the near future. Empirically, it typically suffers fewer cache misses than competing policies like First-In First-Out (FIFO).

Worst-case analysis, straightforwardly applied, provides no useful insights about the performance of different cache replacement policies. For every deterministic policy and cache size k, there is a pathological page request sequence that triggers a page fault rate of 100%, even though the optimal clairvoyant replacement policy (known as Bélády's furthest-in-the-future algorithm) would have a page fault rate of at most $1/k$ (Exercise 1.1). This observation is troublesome both for its absurdly pessimistic performance prediction and for its failure to differentiate between competing replacement policies (such as LRU vs. FIFO). One solution, described in Section 1.3, is to choose an appropriately fine-grained parameterization of the input space and to assess and compare algorithms using parameterized guarantees.

1.2.5 The Cons of Worst-Case Analysis

We should celebrate the fact that worst-case analysis works so well for so many fundamental computational problems, while at the same time recognizing that the cherrypicked successes highlighted in undergraduate algorithms can paint a potentially misleading picture about the range of its practical relevance. The preceding four examples highlight the chief weaknesses of the worst-case analysis framework.

1. *Overly pessimistic performance predictions.* By design, worst-case analysis gives a pessimistic estimate of an algorithm's empirical performance. In the preceding four examples, the gap between the two is embarrassingly large.
2. *Can rank algorithms inaccurately.* Overly pessimistic performance summaries can derail worst-case analysis from identifying the right algorithm to use in practice. In the online paging problem, it cannot distinguish between the FIFO and LRU policies; for linear programming, it implicitly suggests that the ellipsoid method is superior to the simplex method.
3. *No data model.* If worst-case analysis has an implicit model of data, then it's the "Murphy's Law" data model, where the instance to be solved is an adversarially selected function of the chosen algorithm.[7] Outside of security applications, this algorithm-dependent model of data is a rather paranoid and incoherent way to think about a computational problem.

   In many applications, the algorithm of choice is superior precisely because of properties of data in the application domain, such as meaningful solutions in clustering problems or locality of reference in online paging. Pure worst-case analysis provides no language for articulating such domain-specific properties of data. In this sense, the strength of worst-case analysis is also its weakness.

These drawbacks show the importance of alternatives to worst-case analysis, in the form of models that articulate properties of "relevant" inputs and algorithms that possess rigorous and meaningful algorithmic guarantees for inputs with these properties. Research in "beyond worst-case analysis" is a conversation between models and algorithms, with each informing the development of the other. It has both a scientific dimension, where the goal is to formulate transparent mathematical

[7] Murphy's Law: If anything can go wrong, it will.

models that explain empirically observed phenomena about algorithm performance, and an engineering dimension, where the goals are to provide accurate guidance about which algorithm to use for a problem and to design new algorithms that perform particularly well on the relevant inputs.

Concretely, what might a result that goes "beyond worst-case analysis" look like? The next section covers in detail an exemplary result by Albers et al. (2005) for the online paging problem introduced in Section 1.2.4. The rest of the book offers dozens of further examples.

1.3 Example: Parameterized Bounds in Online Paging

1.3.1 Parameterizing by Locality of Reference

Returning to the online paging example in Section 1.2.4, perhaps we shouldn't be surprised that worst-case analysis fails to advocate LRU over FIFO. The empirical superiority of LRU is due to the special structure in real-world page request sequences (locality of reference), which is outside the language of pure worst-case analysis.

The key idea for obtaining meaningful performance guarantees for and comparisons between online paging algorithms is to parameterize page request sequences according to how much locality of reference they exhibit, and then prove parameterized worst-case guarantees. Refining worst-case analysis in this way leads to dramatically more informative results. Part One of the book describes many other applications of such fine-grained input parameterizations; see Section 1.4.1 for an overview.

How should we measure locality in a page request sequence? One tried and true method is the *working set* model, which is parameterized by a function f from the positive integers $\mathbb{N}$ to $\mathbb{N}$ that describes how many different page requests are possible in a window of a given length. Formally, we say that a page sequence σ *conforms to* f if for every positive integer n and every set of n consecutive page requests in σ, there are requests for at most $f(n)$ distinct pages. For example, the identity function $f(n) = n$ imposes no restrictions on the page request sequence. A sequence can only conform to a sublinear function like $f(n) = \lceil \sqrt{n} \rceil$ or $f(n) = \lceil 1 + \log_2 n \rceil$ if it exhibits locality of reference.[8] We can assume without loss of generality that $f(1) = 1, f(2) = 2$, and $f(n+1) \in \{f(n), f(n)+1\}$ for all n (Exercise 1.2).

We adopt as our performance measure $\textsc{Perf}(A, z)$ the fault rate of an online algorithm A on the page request sequence z – the fraction of requests in z on which A suffers a page fault. We next state a performance guarantee for the fault rate of the LRU policy with a cache size of k that is parameterized by a number $\alpha_f(k) \in [0, 1]$. The parameter $\alpha_f(k)$ is defined below in (1.1); intuitively, it will be close to 0 for slow-growing functions f (i.e., functions that impose strong locality of reference) and close to 1 for functions f that grow quickly (e.g., near-linearly). This performance guarantee requires that the function f is *approximately concave* in the sense that the number m_y of inputs with value y under f (that is, $|f^{-1}(y)|$) is nondecreasing in y (Figure 1.3).

[8] The notation $\lceil x \rceil$ means the number x, rounded up to the nearest integer.

$f(n)$	1	2	3	3	4	4	4	5	⋯
n	1	2	3	4	5	6	7	8	⋯

Figure 1.3 An approximately concave function, with $m_1 = 1$, $m_2 = 1$, $m_3 = 2$, $m_4 = 3, \ldots$

Theorem 1.1 (Albers et al., 2005) *With $\alpha_f(k)$ defined as in* (1.1) *below:*

(a) For every approximately concave function f, cache size $k \geq 2$, and deterministic cache replacement policy, there are arbitrarily long page request sequences conforming to f for which the policy's page fault rate is at least $\alpha_f(k)$.
(b) For every approximately concave function f, cache size $k \geq 2$, and page request sequence that conforms to f, the page fault rate of the LRU policy is at most $\alpha_f(k)$ plus an additive term that goes to 0 with the sequence length.
(c) There exists a choice of an approximately concave function f, a cache size $k \geq 2$, and an arbitrarily long page request sequence that conforms to f, such that the page fault rate of the FIFO policy is bounded away from $\alpha_f(k)$.

Parts (a) and (b) prove the worst-case optimality of the LRU policy in a strong and fine-grained sense, f-by-f and k-by-k. Part (c) differentiates LRU from FIFO, as the latter is suboptimal for some (in fact, many) choices of f and k.

The guarantees in Theorem 1.1 are so good that they are meaningful even when taken at face value – for strongly sublinear f's, $\alpha_f(k)$ goes to 0 reasonably quickly with k. The precise definition of $\alpha_f(k)$ for $k \geq 2$ is

$$\alpha_f(k) = \frac{k-1}{f^{-1}(k+1)-2}, \tag{1.1}$$

where we abuse notation and interpret $f^{-1}(y)$ as the smallest value of x such that $f(x) = y$. That is, $f^{-1}(y)$ denotes the smallest window length in which page requests for y distinct pages might appear. As expected, for the function $f(n) = n$ we have $\alpha_f(k) = 1$ for all k. (With no restriction on the input sequence, an adversary can force a 100% fault rate.) If $f(n) = \lceil \sqrt{n} \rceil$, however, then $\alpha_f(k)$ scales with $1/\sqrt{k}$. Thus with a cache size of 10,000, the page fault rate is always at most 1%. If $f(n) = \lceil 1 + \log_2 n \rceil$, then $\alpha_f(k)$ goes to 0 even faster with k, roughly as $k/2^k$.

1.3.2 Proof of Theorem 1.1

This section proves the first two parts of Theorem 1.1; part (c) is left as Exercise 1.4.

Part (a). To prove the lower bound in part (a), fix an approximately concave function f and a cache size $k \geq 2$. Fix a deterministic cache replacement policy A.

We construct a page sequence σ that uses only $k+1$ distinct pages, so at any given time step there is exactly one page missing from the algorithm's cache. (Assume that the algorithm begins with the first k pages in its cache.) The sequence comprises $k-1$ blocks, where the jth block consists of m_{j+1} consecutive requests for the same page p_j, where p_j is the unique page missing from the algorithm A's cache at the start of the

Figure 1.4 Blocks of $k-1$ faults, for $k=3$.

block. (Recall that m_y is the number of values of x such that $f(x)=y$.) This sequence conforms to f (Exercise 1.3).

By the choice of the p_j's, A incurs a page fault on the first request of a block, and not on any of the other (duplicate) requests of that block. Thus, algorithm A suffers exactly $k-1$ page faults.

The length of the page request sequence is $m_2+m_3+\cdots+m_k$. Because $m_1=1$, this sum equals $(\sum_{j=1}^{k} m_j)-1$ which, using the definition of the m_j's, equals $(f^{-1}(k+1)-1)-1=f^{-1}(k+1)-2$. The algorithm's page fault rate on this sequence matches the definition (1.1) of $\alpha_f(k)$, as required. More generally, repeating the construction over and over again produces arbitrarily long page request sequences for which the algorithm has page fault rate $\alpha_f(k)$.

Part (b). To prove a matching upper bound for the LRU policy, fix an approximately concave function f, a cache size $k \geq 2$, and a sequence σ that conforms to f. Our fault rate target $\alpha_f(k)$ is a major clue to the proof (recall (1.1)): we should be looking to partition the sequence σ into blocks of length at least $f^{-1}(k+1)-2$ such that each block has at most $k-1$ faults. So consider groups of $k-1$ consecutive faults of the LRU policy on σ. Each such group defines a *block*, beginning with the first fault of the group, and ending with the page request that immediately precedes the beginning of the next group of faults (see Figure 1.4).

Claim Consider a block other than the first or last. Consider the page requests in this block, together with the requests immediately before and after this block. These requests are for at least $k+1$ distinct pages.

The claim immediately implies that every block contains at least $f^{-1}(k+1)-2$ requests. Because there are $k-1$ faults per block, this shows that the page fault rate is at most $\alpha_f(k)$ (ignoring the vanishing additive error due to the first and last blocks), proving Theorem 1.1(b).

We proceed to the proof of the claim. Note that, in light of Theorem 1.1(c), it is essential that the proof uses properties of the LRU policy not shared by FIFO. Fix a block other than the first or last, and let p be the page requested immediately prior to this block. This request could have been a page fault, or not (cf., Figure 1.4). In any case, p is in the cache when this block begins. Consider the $k-1$ faults contained in the block, together with the kth fault that occurs immediately after the block. We consider three cases.

First, if the k faults occurred on distinct pages that are all different from p, we have identified our $k+1$ distinct requests (p and the k faults). For the second case, suppose that two of the k faults were for the same page $q \neq p$. How could this have happened? The page q was brought into the cache after the first fault on q and was not evicted until there were k requests for distinct pages other than q after this page fault. This gives $k+1$ distinct page requests (q and the k other distinct requests between the two