then uses $O(r)$ arithmetic operations on these smaller numbers to compute the hash-function value. So computing the hash value of a single point involves $O(\log N / \log n)$ multiplications, on numbers of size $\log n$. This is a total of $O(n \log N / \log n)$ arithmetic operations over the course of the algorithm, more than the $O(n)$ we were hoping for.

In fact, it is possible to decrease the number of arithmetic operations to $O(n)$ by using a more sophisticated class of hash functions. There are other classes of universal hash functions where computing the hash-function value can be done by only $O(1)$ arithmetic operations (though these operations will have to be done on larger numbers, integers of size roughly $\log N$). This class of improved hash functions also comes with one extra difficulty for this application: the hashing scheme needs a prime that is bigger than the size of the universe (rather than just the size of the set of points). Now the universe in this application grows inversely with the minimum distance $\delta$, and so, in particular, it increases every time we discover a new, smaller minimum distance. At such points, we will have to find a new prime and set up a new hash table. Although we will not go into the details of this here, it is possible to deal with these difficulties and make the algorithm achieve an expected running time of $O(n)$. 

### 13.8 Randomized Caching

We now discuss the use of randomization for the caching problem, which we first encountered in Chapter 4. We begin by developing a class of algorithms, the *marking algorithms*, that include both deterministic and randomized approaches. After deriving a general performance guarantee that applies to all marking algorithms, we show how a stronger guarantee can be obtained for a particular marking algorithm that exploits randomization.

#### The Problem

We begin by recalling the *Cache Maintenance Problem* from Chapter 4. In the most basic setup, we consider a processor whose full memory has $n$ addresses; it is also equipped with a cache containing $k$ slots of memory that can be accessed very quickly. We can keep copies of $k$ items from the full memory in the cache slots, and when a memory location is accessed, the processor will first check the cache to see if it can be quickly retrieved. We say the request is a *cache hit* if the cache contains the requested item; in this case, the access is very quick. We say the request is a *cache miss* if the requested item is not in the cache; in this case, the access takes much longer, and moreover, one of the items currently in the cache must be *evicted* to make room for the new item. (We will assume that the cache is kept full at all times.)
The goal of a Cache Maintenance Algorithm is to minimize the number of cache misses, which are the truly expensive part of the process. The sequence of memory references is not under the control of the algorithm—this is simply dictated by the application that is running—and so the job of the algorithms we consider is simply to decide on an *eviction policy*: Which item currently in the cache should be evicted on each cache miss?

In Chapter 4, we saw a greedy algorithm that is optimal for the problem: Always evict the item that will be needed the *farthest in the future*. While this algorithm is useful to have as an absolute benchmark on caching performance, it clearly cannot be implemented under real operating conditions, since we don't know ahead of time when each item will be needed next. Rather, we need to think about eviction policies that operate *online*, using only information about past requests without knowledge of the future.

The eviction policy that is typically used in practice is to evict the item that was used the least recently (i.e., whose most recent access was the longest ago in the past); this is referred to as the Least-Recently-Used, or LRU, policy. The empirical justification for LRU is that algorithms tend to have a certain locality in accessing data, generally using the same set of data frequently for a while. If a data item has not been accessed for a long time, this is a sign that it may not be accessed again for a long time.

Here we will evaluate the performance of different eviction policies without making any assumptions (such as locality) on the sequence of requests. To do this, we will compare the number of misses made by an eviction policy on a sequence $\sigma$ with the minimum number of misses it is possible to make on $\sigma$. We will use $f(\sigma)$ to denote this latter quantity; it is the number of misses achieved by the optimal Farthest-in-Future policy. Comparing eviction policies to the optimum is very much in the spirit of providing performance guarantees for approximation algorithms, as we did in Chapter 11. Note, however, the following interesting difference: the reason the optimum was not attainable in our approximation analyses from that chapter (assuming $P \neq NP$) is that the algorithms were constrained to run in polynomial time; here, on the other hand, the eviction policies are constrained in their pursuit of the optimum by the fact that they do not know the requests that are coming in the future.

For eviction policies operating under this online constraint, it initially seems hopeless to say something interesting about their performance: Why couldn't we just design a request sequence that completely confounds any online eviction policy? The surprising point here is that it is in fact possible to give absolute guarantees on the performance of various online policies relative to the optimum.
We first show that the number of misses incurred by LRU, on any request sequence, can be bounded by roughly $k$ times the optimum. We then use randomization to develop a variation on LRU that has an exponentially stronger bound on its performance: Its number of misses is never more than $O(\log k)$ times the optimum.

Designing the Class of Marking Algorithms

The bounds for both LRU and its randomized variant will follow from a general template for designing online eviction policies—a class of policies called marking algorithms. They are motivated by the following intuition. To do well against the benchmark of $f(\sigma)$, we need an eviction policy that is sensitive to the difference between the following two possibilities: (a) in the recent past, the request sequence has contained more than $k$ distinct items; or (b) in the recent past, the request sequence has come exclusively from a set of at most $k$ items. In the first case, we know that $f(\sigma)$ must be increasing, since no algorithm can handle more than $k$ distinct items without incurring a cache miss. But, in the second case, it's possible that $\sigma$ is passing through a long stretch in which an optimal algorithm need not incur any misses at all. It is here that our policy must make sure that it incurs very few misses.

Guided by these considerations, we now describe the basic outline of a marking algorithm, which prefers evicting items that don’t seem to have been used in a long time. Such an algorithm operates in phases; the description of one phase is as follows.

Each memory item can be either marked or unmarked
At the beginning of the phase, all items are unmarked

On a request to item $s$:
Mark $s$
If $s$ is in the cache, then evict nothing
Else $s$ is not in the cache:
If all items currently in the cache are marked then
Declare the phase over
Processing of $s$ is deferred to start of next phase
Else evict an unmarked item from the cache
Endif
Endif

Note that this describes a class of algorithms, rather than a single specific algorithm, because the key step—evict an unmarked item from the
cache—does not specify which unmarked item should be selected. We will see that eviction policies with different properties and performance guarantees arise depending on how we resolve this ambiguity.

We first observe that, since a phase starts with all items unmarked, and items become marked only when accessed, the unmarked items have all been accessed less recently than the marked items. This is the sense in which a marking algorithm is trying to evict items that have not been requested recently. Also, at any point in a phase, if there are any unmarked items in the cache, then the least recently used item must be unmarked. It follows that the LRU policy evicts an unmarked item whenever one is available, and so we have the following fact.

(13.34)  The LRU policy is a marking algorithm.

Analyzing Marking Algorithms

We now describe a method for analyzing marking algorithms, ending with a bound on performance that applies to all marking algorithms. After this, when we add randomization, we will need to strengthen this analysis.

Consider an arbitrary marking algorithm operating on a request sequence \( \sigma \). For the analysis, we picture an optimal caching algorithm operating on \( \sigma \) alongside this marking algorithm, incurring an overall cost of \( f(\sigma) \). Suppose that there are \( r \) phases in this sequence \( \sigma \), as defined by the marking algorithm.

To make the analysis easier to discuss, we are going to "pad" the sequence \( \sigma \) both at the beginning and the end with some extra requests; these will not add any extra misses to the optimal algorithm—that is, they will not cause \( f(\sigma) \) to increase—and so any bound we show on the performance of the marking algorithm relative to the optimum for this padded sequence will also apply to \( \sigma \). Specifically, we imagine a "phase 0" that takes place before the first phase, in which all the items initially in the cache are requested once. This does not affect the cost of either the marking algorithm or the optimal algorithm. We also imagine that the final phase \( r \) ends with an epilogue in which every item currently in the cache of the optimal algorithm is requested twice in round-robin fashion. This does not increase \( f(\sigma) \); and by the end of the second pass through these items, the marking algorithm will contain each of them in its cache, and each will be marked.

For the performance bound, we need two things: an upper bound on the number of misses incurred by the marking algorithm, and a lower bound saying that the optimum must incur at least a certain number of misses.

The division of the request sequence \( \sigma \) into phases turns out to be the key to doing this. First of all, here is how we can picture the history of a
phase, from the marking algorithm's point of view. At the beginning of the phase, all items are unmarked. Any item that is accessed during the phase is marked, and it then remains in the cache for the remainder of the phase. Over the course of the phase, the number of marked items grows from 0 to $k$, and the next phase begins with a request to a $(k+1)$st item, different from all of these marked items. We summarize some conclusions from this picture in the following claim.

(13.35) In each phase, $\sigma$ contains accesses to exactly $k$ distinct items. The subsequent phase begins with an access to a different $(k+1)$st item.

Since an item, once marked, remains in the cache until the end of the phase, the marking algorithm cannot incur a miss for an item more than once in a phase. Combined with (13.35), this gives us an upper bound on the number of misses incurred by the marking algorithm.

(13.36) The marking algorithm incurs at most $k$ misses per phase, for a total of at most $kr$ misses over all $r$ phases.

As a lower bound on the optimum, we have the following fact.

(13.37) The optimum incurs at least $r - 1$ misses. In other words, $f(\sigma) \geq r - 1$.

Proof. Consider any phase but the last one, and look at the situation just after the first access (to an item $s$) in this phase. Currently $s$ is in the cache maintained by the optimal algorithm, and (13.35) tells us that the remainder of the phase will involve accesses to $k-1$ other distinct items, and the first access of the next phase will involve a $k^{th}$ other item as well. Let $S$ be this set of $k$ items other than $s$. We note that at least one of the members of $S$ is not currently in the cache maintained by the optimal algorithm (since, with $s$ there, it only has room for $k-1$ other items), and the optimal algorithm will incur a miss the first time this item is accessed.

What we've shown, therefore, is that for every phase $j < r$, the sequence from the second access in phase $j$ through the first access in phase $j + 1$ involves at least one miss by the optimum. This makes for a total of at least $r - 1$ misses.

Combining (13.36) and (13.37), we have the following performance guarantee.
Proof. The number of misses incurred by the marking algorithm is at most

\[ kr = k(r - 1) + k \leq k \cdot f(\sigma) + k, \]

where the final inequality is just (13.37).

Note that the "\(+k\)" in the bound of (13.38) is just an additive constant, independent of the length of the request sequence \(\sigma\), and so the key aspect of the bound is the factor of \(k\) relative to the optimum. To see that this factor of \(k\) is the best bound possible for some marking algorithms, and for LRU in particular, consider the behavior of LRU on a request sequence in which \(k + 1\) items are repeatedly requested in a round-robin fashion. LRU will each time evict the item that will be needed just in the next step, and hence it will incur a cache miss on each access. (It's possible to get this kind of terrible caching performance in practice for precisely such a reason: the program is executing a loop that is just slightly too big for the cache.) On the other hand, the optimal policy, evicting the page that will be requested farthest in the future, incurs a miss only every \(k\) steps, so LRU incurs a factor of \(k\) more misses than the optimal policy.

Designing a Randomized Marking Algorithm

The bad example for LRU that we just saw implies that, if we want to obtain a better bound for an online caching algorithm, we will not be able to reason about fully general marking algorithms. Rather, we will define a simple Randomized Marking Algorithm and show that it never incurs more than \(O(\log k)\) times the number of misses of the optimal algorithm—an exponentially better bound.

Randomization is a natural choice in trying to avoid the unfortunate sequence of "wrong" choices in the bad example for LRU. To get this bad sequence, we needed to define a sequence that always evicted precisely the wrong item. By randomizing, a policy can make sure that, "on average," it is throwing out an unmarked item that will at least not be needed right away.

Specifically, where the general description of a marking contained the line

\[
\text{Else evict an unmarked item from the cache}
\]

without specifying how this unmarked item is to be chosen, our Randomized Marking Algorithm uses the following rule:

\[
\text{Else evict an unmarked item chosen uniformly at random from the cache}
\]
This is arguably the simplest way to incorporate randomization into the marking framework.\footnote{It is not, however, the simplest way to incorporate randomization into a caching algorithm. We could have considered the \textit{Purely Random Algorithm} that dispenses with the whole notion of marking, and on each cache miss selects one of its \textit{k} current items for eviction uniformly at random. (Note the difference: The Randomized Marking Algorithm randomizes only over the unmarked items.) Although we won't prove this here, the Purely Random Algorithm can incur at least \textit{c} times more misses than the optimum, for any constant \textit{c} < \textit{k}, and so it does not lead to an improvement over LRU.}

\section*{Analyzing the Randomized Marking Algorithm}

Now we'd like to get a bound for the Randomized Marking Algorithm that is stronger than (13.38); but in order to do this, we need to extend the analysis in (13.36) and (13.37) to something more subtle. This is because there are sequences \( \sigma \), with \( r \) phases, where the Randomized Marking Algorithm can really be made to incur \( kr \) misses—just consider a sequence that never repeats an item. But the point is that, on such sequences, the optimum will incur many more than \( r - 1 \) misses. We need a way to bring the upper and lower bounds closer together, based on the structure of the sequence.

This picture of a "runaway sequence" that never repeats an item is an extreme instance of the distinction we'd like to draw: It is useful to classify the unmarked items in the middle of a phase into two further categories. We call an unmarked item \textit{fresh} if it was not marked in the previous phase either, and we call it \textit{stale} if it was marked in the previous phase.

Recall the picture of a single phase that led to (13.35): The phase begins with all items unmarked, and it contains accesses to \( k \) distinct items, each of which goes from unmarked to marked the first time it is accessed. Among these \( k \) accesses to unmarked items in phase \( j \), let \( c_j \) denote the number of these that are to fresh items.

To strengthen the result from (13.37), which essentially said that the optimum incur at least one miss per phase, we provide a bound in terms of the number of fresh items in a phase.

\begin{equation}
\label{eq:13.39}
f(\sigma) \geq \frac{1}{2} \sum_{j=1}^{r} c_j.
\end{equation}

\textbf{Proof.} Let \( f_j(\sigma) \) denote the number of misses incurred by the optimal algorithm in phase \( j \), so that \( f(\sigma) = \sum_{j=1}^{r} f_j(\sigma) \). From (13.35), we know that in any phase \( j \), there are requests to \( k \) distinct items. Moreover, by our definition of \textit{fresh}, there are requests to \( c_{j+1} \) further items in phase \( j + 1 \); so between phases \( j \) and \( j + 1 \), there are at least \( k + c_{j+1} \) distinct items requested. It follows that the optimal algorithm must incur at least \( c_{j+1} \) misses over the course of phases \( j \).
and \( j + 1 \), so \( f_j(\sigma) + f_{j+1}(\sigma) \geq c_{j+1} \). This holds even for \( j = 0 \), since the optimal algorithm incurs \( c_1 \) misses in phase 1. Thus we have

\[
\sum_{j=0}^{r-1} (f_j(\sigma) + f_{j+1}(\sigma)) \geq \sum_{j=0}^{r-1} c_{j+1}.
\]

But the left-hand side is at most \( 2 \sum_{j=1}^{r} f_j(\sigma) = 2f(\sigma) \), and the right-hand side is \( \sum_{j=1}^{r} c_j \).

We now give an upper bound on the expected number of misses incurred by the Randomized Marking Algorithm, also quantified in terms of the number of fresh items in each phase. Combining these upper and lower bounds will yield the performance guarantee we’re seeking. In the following statement, let \( M_\sigma \) denote the random variable equal to the number of cache misses incurred by the Randomized Marking Algorithm on the request sequence \( \sigma \).

\[ (13.40) \quad \text{For every request sequence } \sigma, \text{ we have } E[M_\sigma] \leq H(k) \sum_{j=1}^{r} c_j. \]

**Proof.** Recall that we used \( c_j \) to denote the number of requests in phase \( j \) to fresh items. There are \( k \) requests to unmarked items in a phase, and each unmarked item is either fresh or stale, so there must be \( k - c_j \) requests in phase \( j \) to unmarked stale items.

Let \( X_j \) denote the number of misses incurred by the Randomized Marking Algorithm in phase \( j \). Each request to a fresh item results in a guaranteed miss for the Randomized Marking Algorithm; since the fresh item was not marked in the previous phase, it cannot possibly be in the cache when it is requested in phase \( j \). Thus the Randomized Marking Algorithm incurs at least \( c_j \) misses in phase \( j \) because of requests to fresh items.

Stale items, by contrast, are a more subtle matter. The phase starts with \( k \) stale items in the cache; these are the items that were unmarked *en masse* at the beginning of the phase. On a request to a stale item \( s \), the concern is whether the Randomized Marking Algorithm evicted it earlier in the phase and now incurs a miss as it has to bring it back in. What is the probability that the \( i^{th} \) request to a stale item, say \( s \), results in a miss? Suppose that there have been \( c \leq c_j \) requests to fresh items thus far in the phase. Then the cache contains the \( c \) formerly fresh items that are now marked, \( k - 1 \) formerly stale items that are now marked, and \( k - c - i + 1 \) items that are stale and not yet marked in this phase. But there are \( k - i + 1 \) items overall that are still stale; and since exactly \( k - c - i + 1 \) of them are in the cache, the remaining \( c \) of them are not. Each of the \( k - i + 1 \) stale items is equally likely to be no longer in the cache, and so \( s \) is not in the cache at this moment with probability \( \frac{c}{k-i+1} \leq \frac{c_j}{k-i+1} \).
Chapter 13  Randomized Algorithms

This is the probability of a miss on the request to $s$. Summing over all requests to unmarked items, we have

$$E[X_j] \leq c_j + \sum_{i=1}^{k-\ell} \frac{c_j}{k-i+1} \leq c_j \left[ 1 + \sum_{\ell=\ell+1}^{k} \frac{1}{\ell} \right] = c_j(1 + H(k) - H(\ell)) \leq c_j H(k).$$

Thus the total expected number of misses incurred by the Randomized Marking Algorithm is

$$E[M_\ast] = \sum_{j=1}^{r} E[X_j] \leq H(k) \sum_{j=1}^{r} c_j. \quad \blacksquare$$

Combining (13.39) and (13.40), we immediately get the following performance guarantee.

13.9 Chernoff Bounds

In Section 13.3, we defined the expectation of a random variable formally and have worked with this definition and its consequences ever since. Intuitively, we have a sense that the value of a random variable ought to be "near" its expectation with reasonably high probability, but we have not yet explored the extent to which this is true. We now turn to some results that allow us to reach conclusions like this, and see a sampling of the applications that follow.

We say that two random variables $X$ and $Y$ are independent if, for any values $i$ and $j$, the events $\Pr[X = i]$ and $\Pr[Y = j]$ are independent. This definition extends naturally to larger sets of random variables. Now consider a random variable $X$ that is a sum of several independent 0-1-valued random variables: $X = X_1 + X_2 + \cdots + X_n$, where $X_i$ takes the value 1 with probability $p_i$, and the value 0 otherwise. By linearity of expectation, we have $E[X] = \sum_{i=1}^{n} p_i$. Intuitively, the independence of the random variables $X_1, X_2, \ldots, X_n$ suggests that their fluctuations are likely to "cancel out," and so their sum $X$ will have a value close to its expectation with high probability. This is in fact true, and we state two concrete versions of this result: one bounding the probability that $X$ deviates above $E[X]$, the other bounding the probability that $X$ deviates below $E[X]$. We call these results Chernoff bounds, after one of the probabilists who first established bounds of this form.