separately. We have seen in Section 13.6 that dictionaries have a very efficient implementation using hashing, so abstracting out the dictionary operations allows us to treat the hashing as a "black box" and have the algorithm inherit an overall running time from whatever performance guarantee is satisfied by this hashing procedure. A concrete payoff of this is the following. It has been shown that with the right choice of hashing procedure (more powerful, and more complicated, than what we described in Section 13.6), one can make the underlying dictionary operations run in linear expected time as well, yielding an overall expected running time of $O(n)$. Thus the randomized approach we describe here leads to an improvement over the running time of the divide-and-conquer algorithm that we saw earlier. We will talk about the ideas that lead to this $O(n)$ bound at the end of the section.

It is worth remarking at the outset that randomization shows up for two independent reasons in this algorithm: the way in which the algorithm processes the input points will have a random component, regardless of how the dictionary data structure is implemented; and when the dictionary is implemented using hashing, this introduces an additional source of randomness as part of the hash-table operations. Expressing the running time via the number of dictionary operations allows us to cleanly separate the two uses of randomness.

The Problem

Let us start by recalling the problem's (very simple) statement. We are given n points in the plane, and we wish to find the pair that is closest together. As discussed in Chapter 5, this is one of the most basic geometric *proximity* problems, a topic with a wide range of applications.

We will use the same notation as in our earlier discussion of the closest-pair problem. We will denote the set of points by $P = \{p_1, \ldots, p_n\}$, where p_i has coordinates (x_i, y_i); and for two points $p_i, p_j \in P$, we use $d(p_i, p_j)$ to denote the standard Euclidean distance between them. Our goal is to find the pair of points p_i, p_j that minimizes $d(p_i, p_j)$.

To simplify the discussion, we will assume that the points are all in the unit square: $0 \leq x_i, y_i < 1$ for all $i = 1, \ldots, n$. This is no loss of generality: in linear time, we can rescale all the x- and y-coordinates of the points so that they lie in a unit square, and then we can translate them so that this unit square has its lower left corner at the origin.

Designing the Algorithm

The basic idea of the algorithm is very simple. We'll consider the points in random order, and maintain a current value δ for the closest pair as we process

the points in this order. When we get to a new point p, we look "in the vicinity" of p to see if any of the previously considered points are at a distance less than δ from p. If not, then the closest pair hasn't changed, and we move on to the next point in the random order. If there is a point within a distance less than δ from p, then the closest pair has changed, and we will need to update it.

The challenge in turning this into an efficient algorithm is to figure out how to implement the task of looking for points in the vicinity of p. It is here that the dictionary data structure will come into play.

We now begin making this more concrete. Let us assume for simplicity that the points in our random order are labeled $p_1, \ldots, p_n$. The algorithm proceeds in stages; during each stage, the closest pair remains constant. The first stage starts by setting $\delta = d(p_1, p_2)$, the distance of the first two points. The goal of a stage is to either verify that δ is indeed the distance of the closest pair of points, or to find a pair of points p_i, p_j with $d(p_i, p_j) < \delta$. During a stage, we'll gradually add points in the order $p_1, p_2, \ldots, p_n$. The stage terminates when we reach a point p_i so that for some $j < i$, we have $d(p_i, p_j) < \delta$. We then let δ for the next stage be the closest distance found so far: $\delta = \min_{j:j<i} d(p_i, p_j)$.

The number of stages used will depend on the random order. If we get lucky, and p_1, p_2 are the closest pair of points, then a single stage will do. It is also possible to have as many as $n - 2$ stages, if adding a new point always decreases the minimum distance. We'll show that the expected running time of the algorithm is within a constant factor of the time needed in the first, lucky case, when the original value of δ is the smallest distance.

Testing a Proposed Distance The main subroutine of the algorithm is a method to test whether the current pair of points with distance δ remains the closest pair when a new point is added and, if not, to find the new closest pair.

The idea of the verification is to subdivide the unit square (the area where the points lie) into subsquares whose sides have length $\delta/2$, as shown in Figure 13.2. Formally, there will be N^2 subsquares, where $N = \lceil 1/(2\delta) \rceil$: for $0 \leq s \leq N-1$ and $1 \leq t \leq N-1$, we define the subsquare S_{st} as

$$S_{st} = \{(x, y) : s\delta/2 \leq x < (s+1)\delta/2; t\delta/2 \leq y < (t+1)\delta/2\}.$$

We claim that this collection of subsquares has two nice properties for our purposes. First, any two points that lie in the same subsquare have distance less than δ. Second, and a partial converse to this, any two points that are less than δ away from each other must fall in either the same subsquare or in very close subsquares.

(13.26) *If two points p and q belong to the same subsquare S_{st}, then $d(p, q) < \delta$.*

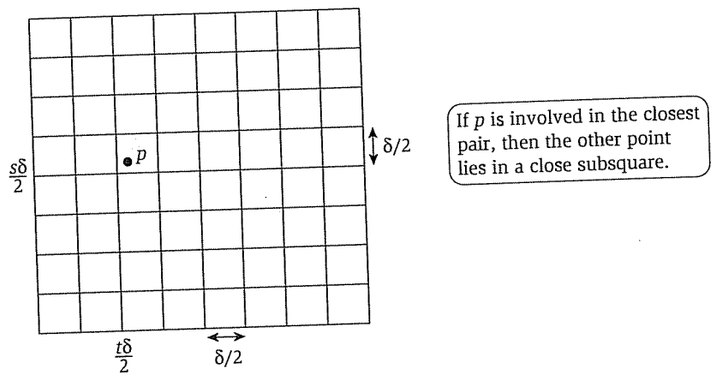

Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{st}.

Proof. If points p and q are in the same subsquare, then both coordinates of the two points differ by at most $\delta/2$, and hence $d(p,q) \leq \sqrt{(\delta/2)^2 + (\delta/2)^2)} = \delta/\sqrt{2} < \delta$, as required. ■

Next we say that subsquares S_{st} and $S_{s't'}$ are *close* if $|s - s'| \leq 2$ and $|t - t'| \leq 2$. (Note that a subsquare is close to itself.)

(13.27) *If for two points $p, q \in P$ we have $d(p,q) < \delta$, then the subsquares containing them are close.*

Proof. Consider two points $p, q \in P$ belonging to subsquares that are not close; assume $p \in S_{st}$ and $q \in S_{s't'}$, where one of s, s' or t, t' differs by more than 2. It follows that in one of their respective x- or y-coordinates, p and q differ by at least δ, and so we cannot have $d(p,q) < \delta$. ■

Note that for any subsquare S_{st}, the set of subsquares close to it form a 5×5 grid around it. Thus we conclude that there are at most 25 subsquares close to S_{st}, counting S_{st} itself. (There will be fewer than 25 if S_{st} is at the edge of the unit square containing the input points.)

Statements (13.26) and (13.27) suggest the basic outline of our algorithm. Suppose that, at some point in the algorithm, we have proceeded partway through the random order of the points and seen $P' \subseteq P$, and suppose that we know the minimum distance among points in P' to be δ. For each of the points in P', we keep track of the subsquare containing it.

Now, when the next point p is considered, we determine which of the subsquares S_{st} it belongs to. If p is going to cause the minimum distance to change, there must be some earlier point $p' \in P'$ at distance less than δ from it; and hence, by (13.27), the point p' must be in one of the 25 squares around the square S_{st} containing p. So we will simply check each of these 25 squares one by one to see if it contains a point in P'; for each point in P' that we find this way, we compute its distance to p. By (13.26), each of these subsquares contains at most one point of P', so this is at most a constant number of distance computations. (Note that we used a similar idea, via (5.10), at a crucial point in the divide-and-conquer algorithm for this problem in Chapter 5.)

A Data Structure for Maintaining the Subsquares The high-level description of the algorithm relies on being able to name a subsquare S_{st} and quickly determine which points of P, if any, are contained in it. A dictionary is a natural data structure for implementing such operations. The *universe* U of possible elements is the set of all subsquares, and the set S maintained by the data structure will be the subsquares that contain points from among the set P' that we've seen so far. Specifically, for each point $p' \in P'$ that we have seen so far, we keep the subsquare containing it in the dictionary, tagged with the index of p'. We note that $N^2 = \lceil 1/(2\delta) \rceil^2$ will, in general, be much larger than n, the number of points. Thus we are in the type of situation considered in Section 13.6 on hashing, where the universe of possible elements (the set of all subsquares) is much larger than the number of elements being indexed (the subsquares containing an input point seen thus far).

Now, when we consider the next point p in the random order, we determine the subsquare S_{st} containing it and perform a `Lookup` operation for each of the 25 subsquares close to S_{st}. For any points discovered by these `Lookup` operations, we compute the distance to p. If none of these distances are less than δ, then the closest distance hasn't changed; we insert S_{st} (tagged with p) into the dictionary and proceed to the next point.

However, if we find a point p' such that $\delta' = d(p,p') < \delta$, then we need to update our closest pair. This updating is a rather dramatic activity: Since the value of the closest pair has dropped from δ to δ', our entire collection of subsquares, and the dictionary supporting it, has become useless—it was, after all, designed only to be useful if the minimum distance was δ. We therefore invoke `MakeDictionary` to create a new, empty dictionary that will hold subsquares whose side lengths are $\delta'/2$. For each point seen thus far, we determine the subsquare containing it (in this new collection of subsquares), and we insert this subsquare into the dictionary. Having done all this, we are again ready to handle the next point in the random order.

Summary of the Algorithm We have now actually described the algorithm in full. To recap:

```
Order the points in a random sequence p_1, p_2, ..., p_n
Let δ denote the minimum distance found so far
Initialize δ = d(p_1, p_2)
Invoke MakeDictionary for storing subsquares of side length δ/2
For i = 1, 2, ..., n:
  Determine the subsquare S_st containing p_i
  Look up the 25 subsquares close to p_i
  Compute the distance from p_i to any points found in these subsquares
  If there is a point p_j (j < i) such that δ' = d(p_j, p_i) < δ then
    Delete the current dictionary
    Invoke MakeDictionary for storing subsquares of side length δ'/2
    For each of the points p_1, p_2, ..., p_i:
      Determine the subsquare of side length δ'/2 that contains it
      Insert this subsquare into the new dictionary
    Endfor
  Else
    Insert p_i into the current dictionary
  Endif
Endfor
```

Analyzing the Algorithm

There are already some things we can say about the overall running time of the algorithm. To consider a new point p_i, we need to perform only a constant number of `Lookup` operations and a constant number of distance computations. Moreover, even if we had to update the closest pair in every iteration, we'd only do n `MakeDictionary` operations.

The missing ingredient is the total expected cost, over the course of the algorithm's execution, due to reinsertions into new dictionaries when the closest pair is updated. We will consider this next. For now, we can at least summarize the current state of our knowledge as follows.

(13.28) *The algorithm correctly maintains the closest pair at all times, and it performs at most $O(n)$ distance computations, $O(n)$ `Lookup` operations, and $O(n)$ `MakeDictionary` operations.*

We now conclude the analysis by bounding the expected number of `Insert` operations. Trying to find a good bound on the total expected number of `Insert` operations seems a bit problematic at first: An update to the closest pair in iteration i will result in i insertions, and so each update comes at a high cost once i gets large. Despite this, we will show the surprising fact that the expected number of insertions is only $O(n)$. The intuition here is that, even as the cost of updates becomes steeper as the iterations proceed, these updates become correspondingly less likely.

Let X be a random variable specifying the number of `Insert` operations performed; the value of this random variable is determined by the random order chosen at the outset. We are interested in bounding $E[X]$, and as usual in this type of situation, it is helpful to break X down into a sum of simpler random variables. Thus let X_i be a random variable equal to 1 if the i^{th} point in the random order causes the minimum distance to change, and equal to 0 otherwise.

Using these random variables X_i, we can write a simple formula for the total number of `Insert` operations. Each point is inserted once when it is first encountered; and i points need to be reinserted if the minimum distance changes in iteration i. Thus we have the following claim.

(13.29) *The total number of `Insert` operations performed by the algorithm is $n + \sum_i iX_i$.*

Now we bound the probability $\Pr[X_i = 1]$ that considering the i^{th} point causes the minimum distance to change.

(13.30) $\Pr[X_i = 1] \leq 2/i$.

Proof. Consider the first i points $p_1, p_2, \ldots, p_i$ in the random order. Assume that the minimum distance among these points is achieved by p and q. Now the point p_i can only cause the minimum distance to decrease if $p_i = p$ or $p_i = q$. Since the first i points are in a random order, any of them is equally likely to be last, so the probability that p or q is last is $2/i$. ∎

Note that $2/i$ is only an upper bound in (13.30) because there could be multiple pairs among the first i points that define the same smallest distance.

By (13.29) and (13.30), we can bound the total number of `Insert` operations as

$$E[X] = n + \sum_i i \cdot E[X_i] \leq n + 2n = 3n.$$

Combining this with (13.28), we obtain the following bound on the running time of the algorithm.

(13.31) *In expectation, the randomized closest-pair algorithm requires $O(n)$ time plus $O(n)$ dictionary operations.*

Achieving Linear Expected Running Time

Up to this point, we have treated the dictionary data structure as a black box, and in (13.31) we bounded the running time of the algorithm in terms of computational time plus dictionary operations. We now want to give a bound on the actual expected running time, and so we need to analyze the work involved in performing these dictionary operations.

To implement the dictionary, we'll use a universal hashing scheme, like the one discussed in Section 13.6. Once the algorithm employs a hashing scheme, it is making use of randomness in two distinct ways: First, we randomly order the points to be added; and second, for each new minimum distance δ, we apply randomization to set up a new hash table using a universal hashing scheme.

When inserting a new point p_i, the algorithm uses the hash-table Lookup operation to find all nodes in the 25 subsquares close to p_i. However, if the hash table has collisions, then these 25 Lookup operations can involve inspecting many more than 25 nodes. Statement (13.23) from Section 13.6 shows that each such Lookup operation involves considering $O(1)$ previously inserted points, in expectation. It seems intuitively clear that performing $O(n)$ hash-table operations in expectation, each of which involves considering $O(1)$ elements in expectation, will result in an expected running time of $O(n)$ overall. To make this intuition precise, we need to be careful with how these two sources of randomness interact.

(13.32) *Assume we implement the randomized closest-pair algorithm using a universal hashing scheme. In expectation, the total number of points considered during the* Lookup *operations is bounded by* $O(n)$.

Proof. From (13.31) we know that the expected number of Lookup operations is $O(n)$, and from (13.23) we know that each of these Lookup operations involves considering only $O(1)$ points in expectation. In order to conclude that this implies the expected number of points considered is $O(n)$, we now consider the relationship between these two sources of randomness.

Let X be a random variable denoting the number of Lookup operations performed by the algorithm. Now the random order σ that the algorithm chooses for the points completely determines the sequence of minimum-distance values the algorithm will consider and the sequence of dictionary operations it will perform. As a result, the choice of σ determines the value of X; we let $X(\sigma)$ denote this value, and we let $\mathcal{E}_\sigma$ denote the event the algorithm chooses the random order σ. Note that the conditional expectation $E\left[X \mid \mathcal{E}_\sigma\right]$ is equal to $X(\sigma)$. Also, by (13.31), we know that $E[X] \leq c_0 n$, for some constant c_0.

Now consider this sequence of Lookup operations for a fixed order σ. For $i = 1, \ldots, X(\sigma)$, let Y_i be the number of points that need to be inspected during the i^{th} Lookup operations—namely, the number of previously inserted points that collide with the dictionary entry involved in this Lookup operation. We would like to bound the expected value of $\sum_{i=1}^{X(\sigma)} Y_i$, where expectation is over both the random choice of σ and the random choice of hash function.

By (13.23), we know that $E\left[Y_i \mid \mathcal{E}_\sigma\right] = O(1)$ for all σ and all values of i. It is useful to be able to refer to the constant in the expression $O(1)$ here, so we will say that $E\left[Y_i \mid \mathcal{E}_\sigma\right] \leq c_1$ for all σ and all values of i. Summing over all i, and using linearity of expectation, we get $E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \leq c_1 X(\sigma)$. Now we have

$$\begin{aligned} E\left[\sum_{i=1}^{X(\sigma)} Y_i\right] &= \sum_\sigma \Pr\left[\mathcal{E}_\sigma\right] E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \\ &\leq \sum_\sigma \Pr\left[\mathcal{E}_\sigma\right] \cdot c_1 X(\sigma) \\ &= c_1 \sum_\sigma E\left[X \mid \mathcal{E}_\sigma\right] \cdot \Pr\left[\mathcal{E}_\sigma\right] = c_1 E[X]. \end{aligned}$$

Since we know that $E[X]$ is at most $c_0 n$, the total expected number of points considered is at most $c_0 c_1 n = O(n)$, which proves the claim. ∎

Armed with this claim, we can use the universal hash functions from Section 13.6 in our closest-pair algorithm. In expectation, the algorithm will consider $O(n)$ points during the Lookup operations. We have to set up multiple hash tables—a new one each time the minimum distance changes—and we have to compute $O(n)$ hash-function values. All hash tables are set up for the same size, a prime $p \geq n$. We can select one prime and use the same table throughout the algorithm. Using this, we get the following bound on the running time.

(13.33) *In expectation, the algorithm uses* $O(n)$ *hash-function computations and* $O(n)$ *additional time for finding the closest pair of points.*

Note the distinction between this statement and (13.31). There we counted each dictionary operation as a single, atomic step; here, on the other hand, we've conceptually opened up the dictionary operations so as to account for the time incurred due to hash-table collisions and hash-function computations.

Finally, consider the time needed for the $O(n)$ hash-function computations. How fast is it to compute the value of a universal hash function h? The class of universal hash functions developed in Section 13.6 breaks numbers in our universe U into $r \approx \log N / \log n$ smaller numbers of size $O(\log n)$ each, and

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 δ, 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 σ with the minimum number of misses it is possible to make on σ. 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 $\mathcal{P} \neq \mathcal{NP}$) 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.