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 $d(p, q) < \delta$.*

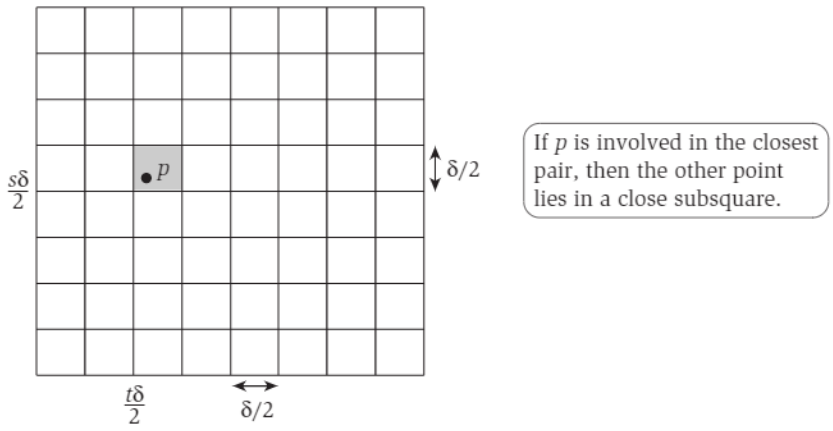

Figure 13.2 Dividing the square into size $\delta/2$ subsquares. The point p lies in the subsquare S_{st}.

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