The subscript "|I| = n" means that the maximum is taken over all (legal) inputs I whose
size is n.

**Average-case time:** What is the average running time over all inputs of size n? More gener-
ally, for each input I, let p(I) denote the probability of seeing this input. The average-case
running time is the weight sum of running times, with the probability being the weight.

\[ A(n) = \sum_{|I| = n} p(I) t(I). \]

We will usually work with worst-case running time because for many of the problems we
will work with, average-case running time is difficult to compute, and it is difficult to specify
probability distributions on inputs that are really meaningful for all applications. It turns
out that for most of the algorithms we will consider, there will be little difference between
worst-case and average-case times. However, if you believe that in your application there is
likely to be a significant difference between the two, then it is a good idea to perform some sort
of average case analysis. We will also usually refer to the worst-case running time as \( T(n) \).

For this brute-force closest pair algorithm, the worst-case running time is fairly easy to derive.
Since each distance computation involves essentially 4 coordinate accesses, and the inner loop
performs one distance calculation then it suffices to count the number of iterations of the inner
loop and multiply by 4. It is easy to see that the number of iterations of the inner loop starts
at \( n - 1 \) and decreases by one with each iteration of the outer loop, and thus is given by the
summation

\[ T(n) = 4((n-1) + (n-2) + \ldots + 2 + 1). \]

Reversing and writing this as a summation yields:

\[ 1 + 2 + \ldots + (n-1) = \sum_{i=1}^{n-1} i, \]

which by standard summation formulas is \( n(n-1)/2 \). (See Chapt 3 of CLR for other summations.)
Thus we have

\[ T(n) = 4 \frac{n(n-1)}{2} = 2n(n-1). \]

This running time grows quadratically (namely as \( n^2 \)) as a function of \( n \). Next time we will
discuss the growth rates of running times.

**Lecture 2: Divide-and-Conquer Closest Pair**

(Thursday, Sep 5, 1996)

Read: Section 35.4 for a discussion of the closest pair problem.

**Closest Pair:** Last time we introduced the closest pair problem: given a set of \( n \) points in the
plane, find the closest pair of points. We presented a simple brute-force algorithm with an
\( O(n^2) \) running time. Today we consider whether we can do better by presenting an \( O(n \log n) \)
algorithm. As often happens the \( O(n \log n) \) algorithm will be more complicated, and involve
larger constant factors. In the case these factor are about 20 times larger than the brute-force
algorithm. Is this really an improvement?

For example, if \( n = 100,000 = 10^5 \), then our simple brute-force algorithm will perform about
\( 2n^2 = 2 \cdot 10^{10} \) operations. If we perform operations at the rate of 1 per microsecond, then the
a total running time will be around 5 hours. If we compare this against a an algorithm taking
40n \lg n \text{ time (where } \lg n \text{ denotes logarithm base 2)} \text{ the number of operations will be about } 40 \cdot 10^5 \cdot 17 \approx 7 \cdot 10^7 \text{ which is about 7 seconds.}

On the other hand if } n \text{ is smaller, e.g. } n = 100, \text{ then } 2n^2 = 20,000 \text{ and } 40n \lg n \approx 27,000. \text{ So the asymptotically faster algorithm is not necessarily better. However notice in both of the latter cases the algorithm is running in just a fraction of a second, so there is really no need for a sophisticated algorithm.}

To get a intuitive feeling for what common asymptotic running times map into in terms of practical usage, here is a little list.

- \text{ } O(1): \text{ Constant time; can't beat it.}
- \text{ } O(\log n): \text{ This is typically the speed that most data structures operate in for a single access. Also the time to find an object in a sorted list by binary search.}
- \text{ } O(n): \text{ This is about the fastest that an algorithm can run, given that you need } O(n) \text{ time just to read in all the data.}
- \text{ } O(n \log n): \text{ Running time of typical sorting algorithms. If you cannot get } O(n), \text{ this is a reasonable substitute.}
- \text{ } O(n^2), O(n^3), \text{ etc.: Polynomial time. Acceptable either when the exponent is small or when the data size is not too large (e.g. } n \leq 1,000). \text{ }
- \text{ } O(2^n): \text{ Exponential time. Only good for fairly small inputs. (e.g. } n \leq 50). \text{ }
- \text{ } O(n!), O(n^n): \text{ Only good for really small inputs (e.g. } n \leq 20). \text{ }

Note that many algorithms may run in very bad time, e.g. \text{ } O(2^n) \text{ time, in the worst case, but may be quite efficient on average. But watch out for those bad cases.}

\textbf{Divide-and-Conquer Algorithm:} \text{ Today we consider an approach based on divide-and-conquer, a powerful algorithm design technique. Such an algorithm consists of 3 basic steps, divide, conquer, and combine.}

\textbf{Divide:} \text{ Recall that the points are stored in an array } P. \text{ If } P \text{ contains sufficiently few points, then we simply solve the problem by the brute-force method (compute distances between all pairs of points in } P). \text{ Otherwise, we find a vertical line } l \text{ that subdivides the list } P \text{ into two lists of roughly equal size, } P_L \text{ and } P_R.

\textbf{Conquer:} \text{ We recursively call the closest pair procedure on each list. Let } \delta_L \text{ and } \delta_R \text{ be the closest pair distances in each case. Let } \delta = \min(\delta_L, \delta_R).

\textbf{Combine:} \text{ Is } \delta \text{ the final result? Not quite. It might be that the splitting line } l \text{ passes between the closest pair. We need to check whether there is a closer pair with one point to the left of } l \text{ and the other two to the right. First observe that we do not need to check points that are very far away from } l. \text{ In particular, if a point is further from } l \text{ than } \delta, \text{ then we know that there cannot be a neighbor on the other side of } l \text{ that is closer than } \delta.

We create a list } P' \text{ of points of } P \text{ that lie within distance } \delta \text{ of either side of } l \text{ (we might be more clever by trying to separate points from the left and right, but we will see that this is not really necessary). We determine the closest pair in } P' \text{ (say at distance } \delta') \text{ and return the minimum of } \delta \text{ and } \delta' \text{ as the final result. (Actually we return the pair that achieves this distance.)}

\textbf{Whoa!!} \text{ How did you find the closest pair in } P'? \text{ (I guess this was all a little too good to be true.) This requires a little bit of cleverness. To get the running time we want, we will need to do this in at-most } O(n) \text{ time. I will show that we can do this assuming the points of } P' \text{ are}
Figure 2: Closest Pairs by Divide and Conquer.

given in sorted order according to y-coordinate. (We will have to come back and deal with the question of how to sort these points later.)

Suppose that the points of \( P' \) have been sorted according to their y-coordinates. Consider a point \( P'[i] \) in this sorted list. Which point in \( P' \) is the nearest neighbor of \( P'[i] \)? As in the brute force algorithm we may restrict the search to points with higher indices (since this will be done for every point in the list). Must it necessarily be \( P'[i+1] \)? That would be great if it were true, but it is not. Just because a point has a closer y-coordinate does not mean that it will be closer. So how far away from point \( P'[i] \) might we have to look? \( P[i+2] \)? \( P[i+8] \)? \( P[i+200] \)? Maybe there is no constant bound.

It turns out that we can put a limit on how far we might have to look for a point's nearest neighbor. In particular, we claim that we need never look more than 7 points away.

**Lemma:** If \( P'[i] \) and \( P'[j] \) (\( i < j \)) are the closest pair in \( P' \) and the distance between them is less than \( \delta \), then \( j - i \leq 7 \).

**Proof:** Suppose that \( P'[i] \) and \( P'[j] \) are the closest pair in \( P' \) and their distance is less than \( \delta \). Since they are in \( P' \) they are within distance \( \delta \) of \( l \). Since they are closer than \( \delta \), their y-coordinates can differ by at most \( \delta \). So they must both reside in a rectangle of width \( 2\delta \) and height \( \delta \) centered about \( l \). Split this rectangle into 8 identical squares of side length \( \delta/2 \). Observe that the diagonal of each of these squares is of length

\[
\frac{\delta\sqrt{2}}{2} = \frac{\delta}{\sqrt{2}} < \delta.
\]

Since each square lies entirely on one side of \( l \) or the other, no square can contain two points of \( P \) (for otherwise, these two points would contradict the fact that \( \delta \) is the closest pair distance on either side of \( l \)). Thus there can be at most 8 points of \( P' \) in this rectangle, and hence \( j - i \leq 7 \).

**Question:** Is this the best possible? What is the smallest range you need to search for the nearest neighbor in the worst case?

For each \( P'[i] \) we need only search a constant number of points, and from this it follows that we can perform the search in \( O(n) \) time.
Presorting: The only issue that was left unresolved is how we sort the points of $P'$, since we want this phase of the algorithm to run in $O(n)$ time, we cannot afford to call a general sorting routine, which would take $O(n \log n)$ time in general.

There is a clever solution to this problem. We presort the points. In particular, we store the points (redundantly) in two lists, $X$ and $Y$, sorted along the $x$- and $y$-coordinates of the points, respectively. This is done by calling a sorting algorithm on $P$ twice before beginning the divide-and-conquer algorithm.

We no longer need the list $P$. When we split $P$ into $P_L$ and $P_R$, what we actually do is split $X$ and $Y$ into $X_L$, $X_R$ and $Y_L$, $Y_R$, respectively. This is easy to do in $O(n)$ time. Since the original lists are sorted, we can do this so that the resulting lists are sorted as well. To form the list $P'$, we simply extract the elements of $Y$ that are within distance $\delta$ of $l$, and copy them to a new sorted list, $Y'$, which we search.

Summary: Here is a summary of the algorithm. It returns the minimum distance, but it is easy to modify so it returns the actual pair of points as well.

Presort: Given the list $P$, make two copies in $X$ and $Y$. Sort $X$ according to $x$-coordinates, and $Y$ according to $y$-coordinates.

Recursive Part: ClosestPair($X, Y$):

Basis: If the number of elements remaining is at most 3, then solve the problem by brute force, and return the closest pair distance $\delta$.

Divide: Otherwise, let $l$ be the median $x$-coordinate of $X$. Split both lists about this line, maintaining their ordering, creating $X_L$, $X_R$ and $Y_L$, $Y_R$.

Conquer: $\delta_L = \text{ClosestPair}(X_L, Y_L)$, and $\delta_R = \text{ClosestPair}(X_R, Y_R)$.

Combine: Let $\delta = \min(\delta_L, \delta_R)$. Create list $Y'$ by copying all points of $Y$ that are within distance $\delta$ of $l$. For $i$ running from 1 to the length of $Y'$, and for $j$ running from $i + 1$ to $i + 7$, compute the distances between $Y'[i]$ and $Y'[j]$. Let $\delta'$ be the minimum of these distances. Return $\min(\delta, \delta')$.

Note that this description is far from being a program, but a competent programmer should have no difficulty in converting this into a C or C++ program.

Analysis: How long does this program take to execute in the worst case? (Note that an average case analysis might be quite tricky because it would depend on how many points are in $P'$ on the average, and this depends on what the expected closest pair distance is in each sublist.) In the worst case, we may assume that $P'$ contains all $n$ points. The presort takes $O(n \log n)$ time, since this is the time needed for any competent sorting algorithm (e.g. heapsort or merge sort).

To analyze the recursive part, it is natural to set up a recurrence, that is, a recursively defined function to measure the running time. For simplicity, we will not count the exact number of coordinate hits, and treat all constant multiplicative factors as 1. As a basis, when $n$ is small (at most 3) the running time there are at most a constant number of distance computations, for a running time of $O(1)$. Otherwise we need to split the lists $X$ and $Y$ about the median ($O(n)$ time) make two recursive calls on lists of half the size ($2T(n/2)$), form the list $Y'$ ($O(n)$ time), and then compute the distance between each point in $Y'$ and its next 7 successors ($O(n)$ time). The total is given by the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3, \\ 2T(n/2) + n & \text{otherwise}. \end{cases}$$

7
This is perhaps the best known recurrence, since it is the same recurrence that appears for all \( O(n \log n) \) sorting algorithms.

**Solving Recurrences:** There are a number of methods for solving the sort of recurrences that show up in divide-and-conquer algorithms. The easiest method is to apply the Master Theorem that is given in CLR. Here is a slightly more restrictive version, but adequate for a lot of instances.

**Theorem:** (Simplified Master Theorem) Let \( a \geq 1, b > 1 \) be constants and let \( T(n) \) be the recurrence

\[
T(n) = aT(n/b) + cn^b,
\]

defined for \( n \geq 0 \).

**Case 1:** \( a > b^k \) then \( T(n) \) is \( \Theta(n^{\log_b a}) \).

**Case 2:** \( a = b^k \) then \( T(n) \) is \( \Theta(n^{k \log n}) \).

**Case 3:** \( a < b^k \) then \( T(n) \) is \( \Theta(n^k) \).

Using this version of the Master Theorem we can see that in our recurrence \( a = 2, b = 2 \), and \( k = 1 \), so \( a = b^k \) and case 2 applies. Thus \( T(n) \) is \( \Theta(n \log n) \).

However there are many recurrences that cannot be put into this form. For example, if we had to sort the elements of \( P \), we might have seen a recurrence of the form \( T(n) = 2T(n/2) + n \log n \). This solves to \( T(n) = \Theta(n \log^2 n) \), but the Master Theorem (either this form or the one in CLR will not tell you this.)

A more basic method for solving recurrences is that of repeated expansion. This is a painstaking process of repeatedly applying the definition of the recurrence until (hopefully) a simple pattern emerges that usually results in a summation that is easy to solve.

**Lecture 3: More on Closest Pair**

(Tuesday, Sep 10, 1996)

Read: Today's material is not covered in our text.

**Solving Recurrences:** Last time we presented an algorithm for solving the closest pair problem whose running time could be defined by the recurrence

\[
\begin{align*}
T(n) &= 1 & \text{if } n \leq 3, \\
T(n) &= 2T(n/2) + n & \text{otherwise.}
\end{align*}
\]

We stated that \( T(n) \) is \( O(n \log n) \) without proof. One of the standard methods for solving recurrences is by a method called expansion. The idea is to repeatedly expand the recurrence until a pattern emerges. The pattern usually takes the form of a summation. Then solve this summation.

Consider the following example (which is interesting, because it cannot be solved by the Master Theorem).

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= 2T(n/2) + n \log n & \text{for } n > 1.
\end{align*}
\]