3. In an infinite array, the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. Present an algorithm that takes x as input and finds the position of x in the array in $\Theta(\log n)$ time. *You are not given the value of n.*

4. Devise a "binary" search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?

5. Devise a ternary search algorithm that first tests the element at position $n/3$ for equality with some value x, and then checks the element at $2n/3$ and either discovers x or reduces the set size to one-third the size of the original. Compare this with binary search.

6. (a) Prove that BinSearch1 works correctly.

   (b) Verify that the following algorithm segment functions correctly according to the specifications of binary search. Discuss its computing time.

```
$low := 1$; $high := n$;
**repeat** {
    $mid := \lfloor (low + high)/2 \rfloor$;
    **if** $(x \geq a[mid])$ **then** $low := mid$;
    **else** $high := mid$;
} **until** $((low + 1) = high)$
```

3.3 FINDING THE MAXIMUM AND MINIMUM

Let us consider another simple problem that can be solved by the divide-and-conquer technique. The problem is to find the maximum and minimum items in a set of n elements. Algorithm 3.5 is a straightforward algorithm to accomplish this.

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. The justification for this is that the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in $a[1 : n]$ are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operations. Hence the time is determined mainly by the total cost of the element comparisons.

StraightMaxMin requires $2(n-1)$ element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing

```
Algorithm StraightMaxMin($a, n, max, min$)
// Set $max$ to the maximum and $min$ to the minimum of $a[1:n]$.
{
    $max := min := a[1]$;
    for $i := 2$ to $n$ do
    {
        if ($a[i] > max$) then $max := a[i]$;
        if ($a[i] < min$) then $min := a[i]$;
    }
}
```

Algorithm 3.5 Straightforward maximum and minimum

that the comparison $a[i] < min$ is necessary only when $a[i] > max$ is false. Hence we can replace the contents of the **for** loop by

```
if ($a[i] > max$) then $max := a[i]$;
else if ($a[i] < min$) then $min := a[i]$;
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is $n - 1$. The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons is $2(n-1)$. The average number of element comparisons is less than $2(n-1)$. On the average, $a[i]$ is greater than max half the time, and so the average number of comparisons is $3n/2 - 1$.

A divide-and-conquer algorithm for this problem would proceed as follows: Let $P = (n, a[i], \ldots, a[j])$ denote an arbitrary instance of the problem. Here n is the number of elements in the list $a[i], \ldots, a[j]$ and we are interested in finding the maximum and minimum of this list. Let Small$(P)/$ be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances $P_1 = (\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$. After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. How can we combine the solutions for P_1 and P_2 to obtain a solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1) and MIN(P_2).

Algorithm 3.6 results from applying the strategy just described. MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements $\{a(i), a(i+1), \ldots, a(j)\}$. The situation of set sizes one $(i = j)$ and two $(i = j - 1)$ are handled separately. For sets containing more than two elements, the midpoint is determined (just as in binary search) and two new subproblems are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

```
✓ Algorithm MaxMin($i, j, max, min$)
// $a[1:n]$ is a global array. Parameters $i$ and $j$ are integers,
// $1 \le i \le j \le n$. The effect is to set $max$ and $min$ to the
// largest and smallest values in $a[i:j]$, respectively.
{
    if ($i = j$) then $max := min := a[i]$; // Small($P$)
    else if ($i = j - 1$) then // Another case of Small($P$)
        {
            if ($a[i] < a[j]$) then
            {
                $max := a[j]$; $min := a[i]$;
            }
            else
            {
                $max := a[i]$; $min := a[j]$;
            }
        }
        else
        {   // If $P$ is not small, divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (i + j)/2 \rfloor$;
            // Solve the subproblems.
                MaxMin($i, mid, max, min$);
                MaxMin($mid + 1, j, max1, min1$);
            // Combine the solutions.
                if ($max < max1$) then $max := max1$;
                if ($min > min1$) then $min := min1$;
        }
}
```

Algorithm 3.6 Recursively finding the maximum and minimum

The procedure is initially invoked by the statement

$$\mathsf{MaxMin}(1, n, x, y)$$

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm each node has four items of information: i, j, max, and min. On the array $a[\,]$ above, the tree of Figure 3.2 is produced.

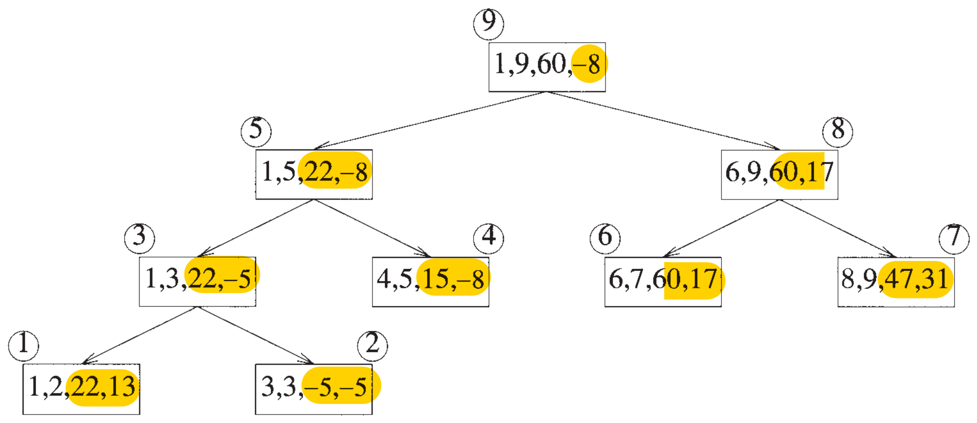

Figure 3.2 Trees of recursive calls of MaxMin

Examining Figure 3.2, we see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9, respectively, and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

Compared with the $2n - 2$ comparisons for the straightforward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons. So in this sense algorithm MaxMin is optimal (see Chapter 10 for more details). But does this imply that MaxMin is better in practice? Not necessarily. In terms of storage, MaxMin is worse than the straightforward algorithm because it requires stack space for i, j, $max, min, max1$, and $min1$. Given n elements, there will be $\lfloor \log_2 n \rfloor + 1$ levels of recursion and we need to save seven values for each recursive call (don't forget the return address is also needed).

Let us see what the count is when element comparisons have the same cost as comparisons between i and j. Let $C(n)$ be this number. First, we observe that lines 6 and 7 in Algorithm 3.6 can be replaced with

```
if ($i \ge j - 1$) { // Small($P$)
```

to achieve the same effect. Hence, a single comparison between i and $j - 1$ is adequate to implement the modified **if** statement. Assuming $n = 2^k$ for some positive integer k, we get

$$C(n) = \begin{cases} 2C(n/2) + 3 & n > 2 \\ 2 & n = 2 \end{cases}$$