

Initially $l = 0$ and $r = 8$, so step 2.1 sets $m = 4$. Since *target* is greater than $a[4]$, step 2.4 sets $l = m + 1$. Now $l = 5$ and $r = 8$, so step 2.1 sets $m = 6$ (say). Since *target* is less than $a[6]$, step 2.3 sets $r = m - 1$. Now $l = 5$ and $r = 5$, so step 2.1 sets $m = 5$. Since *target* is equal to $a[5]$, step 2.2 terminates the algorithm with the answer 5. In this case only three iterations are needed; in general, up to four iterations will be needed to search an array of that length.

Figure 3.20(b) illustrates an unsuccessful search. Since no array component equals the target value, the algorithm eventually reaches the state $l > r$, indicating that no part of the array remains to be searched (see Figure 3.19). In that state the algorithm reaches step 3, which gives the answer *none*.

Let us analyze the binary search algorithm's time efficiency. Let $n = \text{right} - \text{left} + 1$ be the length of $a[\text{left} \dots \text{right}]$. The number of iterations is at most the number of times that n can be halved until it reaches zero, i.e., $\text{floor}(\log_2 n) + 1$. (See Appendix A.2.) If we assume that steps 2.2 through 2.4 can be implemented with a single comparison, then we have:

$$\text{Max. no. of comparisons} = \text{floor}(\log_2 n) + 1 \tag{3.5}$$

Thus the binary search algorithm has time complexity $O(\log n)$.

Program 3.21 shows how the binary search algorithm would be implemented in Java.

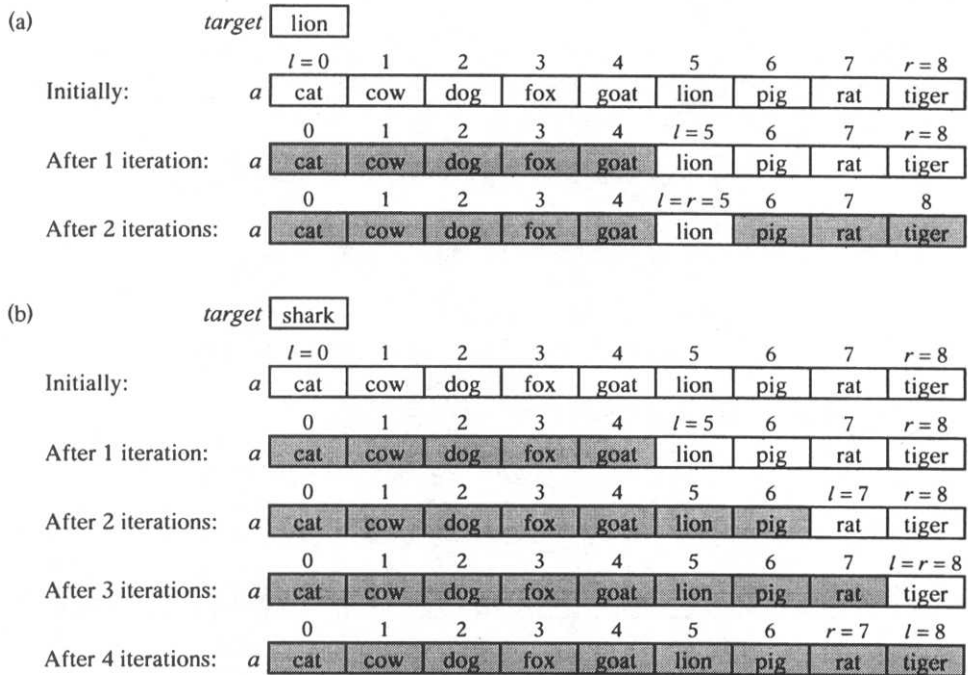


Figure 3.20 Illustration of the array binary search algorithm: (a) successful search; (b) unsuccessful search.