5.6.2 Binary Search

If the input array is sorted, we can use the *binary search*, which we perform from the middle of the array rather than the end.

If the input array is sorted, then we have an alternative to the sequential search, the *binary search*. We perform a binary search from the middle of the array rather than the end.

At any point in time we keep track of `low` and `high`, which delimit the portion of the array in which an item, if present, must reside. Initially, the range is from `0` to $N - 1$. If `low` is larger than `high`, we know that the item is not present, so we throw an exception. Otherwise, we let `mid` be the halfway point of the range (rounding down if the range has an even number of elements) and compare the item we are searching for with the item in position `mid`. If we find a match, we are done and can return. If the item we are searching for is less than the item in position `mid`, then it must reside in the range `low` to `mid-1`. If it is greater, then it must reside in the range `mid+1` to `high`. In Figure 5.11, lines 18 to 21 alter the possible range, essentially cutting it in half. By the repeated halving principle, we know that the number of iterations will be $O(\log N)$.

The *binary search* is logarithmic because the search range is halved in each iteration.

For an unsuccessful search, the number of iterations in the loop is $\lfloor \log N \rfloor + 1$. This is because we halve the range in each iteration (rounding down if the range has an odd number of elements); we add 1 because the final range encompasses zero elements. For a successful search, the worst case is $\lfloor \log N \rfloor$ iterations because in the worst case we get down to a range of only one element. The average case is only one iteration better. This is because half of the elements require the worst case for their search, a quarter of the elements save one iteration, and only one in 2^i elements will save i iterations from the worst case. The mathematics involves computing the weighted average by calculating the sum of a finite series. The bottom line, however, is that the running time for each search is $O(\log N)$. Exercise 5.19 asks you to complete the calculation.

For reasonably large values of N, the binary search outperforms the sequential search. For instance, if N is 1,000, then on average a successful sequential search requires about 500 comparisons. The average binary search, using the previous formula, requires eight iterations for a successful search. Since each iteration uses 1.5 comparisons on average (sometimes 1, other times 2), the total is 12 comparisons for a successful search. The binary search wins by even more in the worst case or when searches are unsuccessful.

Optimizing the binary search can cut the number of comparisons roughly in half.

If we want to make the binary search even faster, we need to make the inner loop tighter. A possible strategy is to remove the (implicit) test for a successful search from that inner loop and shrink the range down to one item in all cases. Then we can use a single test outside of the loop to determine if the item is in the array or is not found, as shown in Figure 5.12 (page 130). If the item we are searching for in Figure 5.12 is not larger than the item in the `mid` position, then it is in the range that includes the `mid` position. When we break the loop, the subrange is 1, and we can test to see if we have a match.

In the revised algorithm, the number of iterations is always $\lfloor \log N \rfloor$ because we always shrink the range in half, possibly by rounding down. The number of comparisons that are used thus is always $\lfloor \log N \rfloor + 1$.