Word lists

People typically use a mixture of the binary search and interpolative search algorithms when searching a telephone book, after the initial guess we exploit the fact that the entries are sorted and can rapidly find the required entry. For example, when searching for Smith, if Rogers and Thomas have been found, one can flip to a page about halfway between the previous guesses. If this shows Samson, it can be concluded that Smith is somewhere between the Samson and Thomas pages so these can be divided.

Applications to complexity theory

Even if we do not know a fixed range the number k falls in, we can still determine its value by asking $2\lceil \log_2 k \rceil$ simple yes/no questions of the form "Is k greater than x?" for some number x. As a simple consequence of this, if you can answer the question "Is this integer property k greater than a given value?" in some amount of time then you can find the value of that property in the same amount of time with an added factor of $\log_2 k$. This is called a *reduction*, and it is because of this kind of reduction that most complexity theorists concentrate on decision problems, algorithms that produce a simple yes/no answer.

For example, suppose we could answer "Does this n x n matrix have permanent larger than k?" in $O(n^2)$ time. Then, by using binary search, we could find the (ceiling of the) permanent itself in $O(n^2 \log p)$ time, where p is the value of the permanent. Notice that p is not the size of the input, but the *value* of the output; given a matrix whose maximum item (in absolute value) is m, p is bounded by $m^n n!$. Hence $\log p = O(n \log n + n \log m)$. A binary search could find the permanent in $O(n^3 \log n + n^3 \log m)$.

Algorithm

Recursive

A straightforward implementation of binary search is recursive. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray. Each of the calls is tail recursive, so a compiler need not make a new stack frame for each call. The variables `imin` and `imax` are the lowest and highest inclusive indices that are searched.

```
int binary_search(int A[], int key, int imin, int imax)
{
  // test if array is empty
  if (imax < imin)
    // set is empty, so return value showing not found
    return KEY_NOT_FOUND;
  else
    {
      // calculate midpoint to cut set in half
      int imid = midpoint(imin, imax);

      // three-way comparison
      if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid - 1);
      else if (A[imid] < key)
        // key is in upper subset
        return binary_search(A, key, imid + 1, imax);
      else
        // key has been found
        return imid;
    }
```

```
}
```

It is invoked with initial `imin` and `imax` values of `0` and `N-1` for a zero based array of length N.

The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as `"(imin + imax) / 2"`. In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result `"(imin + imax)"` might overflow. With limited numbers, the midpoint can be calculated correctly as `"imin + ((imax - imin) / 2)"`.

Iterative

The binary search algorithm can also be expressed iteratively with two index limits that progressively narrow the search range.[3]

```
int binary_search(int A[], int key, int imin, int imax)
{
  // continue searching while [imin,imax] is not empty
  while (imin <= imax)
    {
      // calculate the midpoint for roughly equal partition
      int imid = midpoint(imin, imax);
      if (A[imid] == key)
        // key found at index imid
        return imid;
      // determine which subarray to search
      else if (A[imid] < key)
        // change min index to search upper subarray
        imin = imid + 1;
      else
        // change max index to search lower subarray
        imax = imid - 1;
    }
  // key was not found
  return KEY_NOT_FOUND;
}
```

Deferred detection of equality

The above iterative and recursive versions take three paths based on the key comparison: one path for less than, one path for greater than, and one path for equality. (There are two conditional branches.) The path for equality is taken only when the record is finally matched, so it is rarely taken. That branch path can be moved outside the search loop in the deferred test for equality version of the algorithm. The following algorithm uses only one conditional branch per iteration.[4]

```
// inclusive indices
//   0 <= imin when using truncate toward zero divide
//     imid = (imin+imax)/2;
//   imin unrestricted when using truncate toward minus infinity divide
//     imid = (imin+imax)>>1; or
//     imid = (int)floor((imin+imax)/2.0);
int binary_search(int A[], int key, int imin, int imax)
{
  // continually narrow search until just one element remains
  while (imin < imax)
    {
      int imid = midpoint(imin, imax);

      // code must guarantee the interval is reduced at each iteration
      assert(imid < imax);
```

```
    // note: 0 <= imin < imax implies imid will always be less than imax

    // reduce the search
    if (A[imid] < key)
      imin = imid + 1;
    else
      imax = imid;
  }
 // At exit of while:
 //   if A[] is empty, then imax < imin
 //   otherwise imax == imin

 // deferred test for equality
 if  (A[imin] == key)
   return imin;
 else
   return KEY_NOT_FOUND;
}
```

The deferred detection approach foregoes the possibility of early termination on discovery of a match, so the search will take about $\log_2(N)$ iterations. On average, a *successful* early termination search will not save many iterations. For large arrays that are a power of 2, the savings is about two iterations. Half the time, a match is found with one iteration left to go; one quarter the time with two iterations left, one eighth with three iterations, and so forth. The infinite series sum is 2.

The deferred detection algorithm has the advantage that if the keys are not unique, it returns the smallest index (the starting index) of the region where elements have the search key. The early termination version would return the first match it found, and that match might be anywhere in region of equal keys.

Performance

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, N, is odd then both sub-intervals will contain $(N-1)/2$ elements, while if N is even then the two sub-intervals contain $N/2-1$ and $N/2$ elements.

If the original number of items is N then after the first iteration there will be at most $N/2$ items remaining, then at most $N/4$ items, at most $N/8$ items, and so on. In the worst case, when the value is not in the list, the algorithm must continue iterating until the span has been made empty; this will have taken at most $\lfloor \log_2(N)+1 \rfloor$ iterations, where the $\lfloor\ \rfloor$ notation denotes the floor function that rounds its argument down to an integer. This worst case analysis is tight: for any N there exists a query that takes exactly $\lfloor \log_2(N)+1 \rfloor$ iterations. When compared to linear search, whose worst-case behaviour is N iterations, we see that binary search is substantially faster as N grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

Average performance

$\log_2(N)-1$ is the expected number of probes in an average successful search, and the worst case is $\log_2(N)$, just one more probe. If the list is empty, no probes at all are made. Thus binary search is a logarithmic algorithm and executes in $O(\log N)$ time. In most cases it is considerably faster than a linear search. It can be implemented using iteration, or recursion. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.