Array Searching Algorithms

Array Searching Algorithms

Two methods for searching an array for a given item:

1. The Sequential Search method can be used with any array.
2. The Binary Search method can only be used with arrays that are known to be sorted, but is much faster than Sequential Search.

Sequential Search

- To search an array A[0..N-1] for a value X, start an index at one end of the array, say 0.
- Step index through the array, examining each A[index] to see if it is equal to X.
- Stop if you find X and return index. Otherwise you get to the end of the array and return -1.

Sequential Search
Search an array A[0..N-1] for X

```
int search(int A[ ], int X)
 {
     // Default assumption is X won't be found
     int position  = -1;
     boolean found = false;
     int index = 0;

     while (!found && index < N)
      {
          // check A[index]
          if (A[index] == X)
          {
              found = true;
              position = index;
          }
          index ++;
     }
     return position;
 }
```

Efficiency of Sequential Search

- In the worst case, you search the entire array, peforming *N* comparisons
- If you are lucky, you find *X* the first place you look, requiring only one comparison
- On average, you perform *N*/2 comparisons

Binary Search

Binary Search

- Works on a sorted portion *A*[lower..upper]:

- Compare X to A[middle], where middle is the midpoint between lower and upper:

  middle = (lower + upper)/2

- If X == A[middle], return middle (we found it!)

- If X < A[middle], then continue search in A[lower..middle-1]

- If X > A[middle], then continue search in A[middle+1..upper]

- Search terminates if **X** is found, or when we try to search an empty segment.

Binary Search of A[lower..upper]

- To continue search in A[lower..middle-1], keep lower the same and replace upper with middle-1:

    upper = middle -1

- To continue search in A[middle+1..upper], replace lower with middle+1 and keep upper the same:

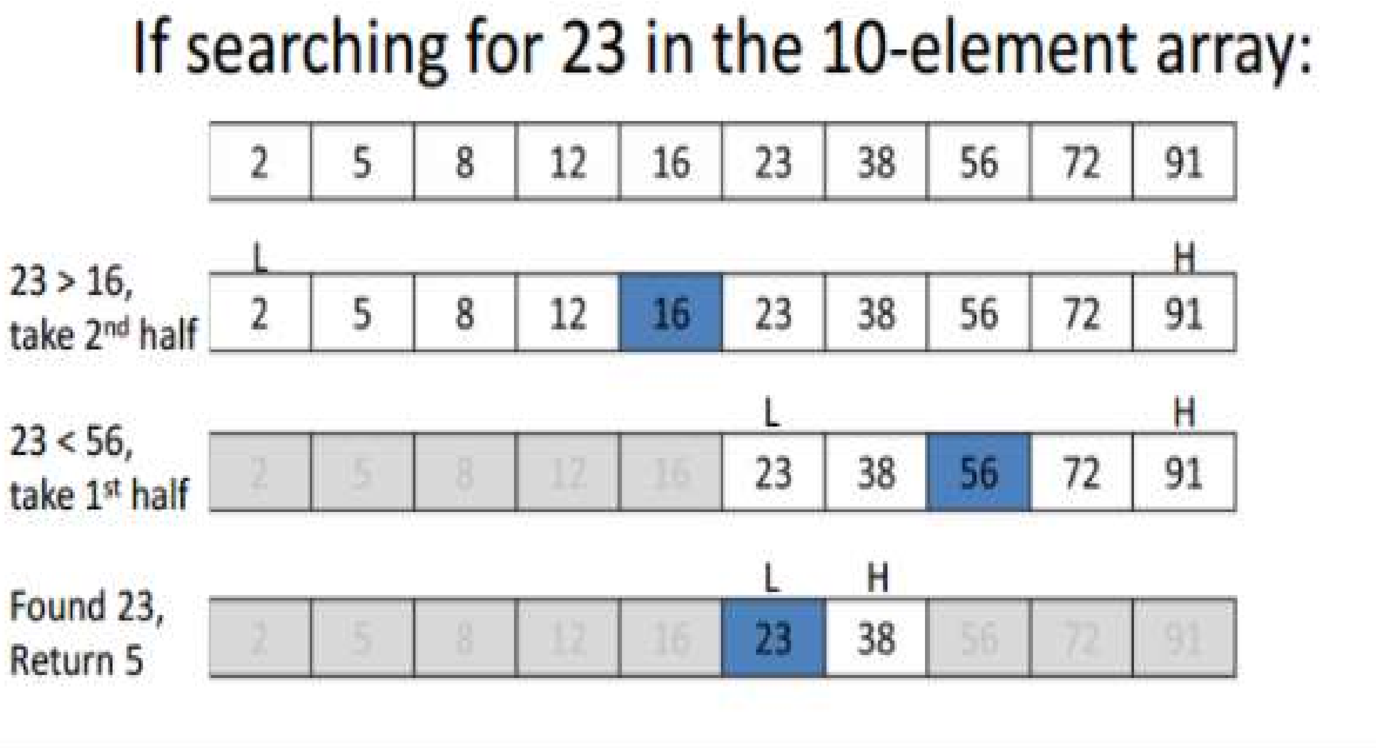
    lower = middle+1

Binary Search of A[0..N-1]

```
// returns index of X if found, -1 otherwise
int binSearch(int A[ ], int X)
 {
     int lower = 0, upper = N-1;
     int position = -1;             // index of X to be returned
     boolean found = false;   // assumption is X will not be found

    // if X is there, it must be in A[lower..upper]
     while (!found && lower <= upper)
     {
           int middle = (lower + upper)/2;
           if (A[middle] == X)
           {
               found = true;   position = middle;
           }
          else if (A[middle] > X)
          {  // if X is there, it is in A[lower..middle-1]
                     upper = middle -1;
          }
         else
          {  //  if X is there, it is in A[middle+1, upper]
                   lower = middle +1;
          }
     }
     return position;
 }
```

Example of Binary Search



Credits: Binary Search - Geeks for Geeks (https://www.geeksforgeeks.org/binary-search/)

Recursive Binary Search

- The logic of binary search has a natural recursive implementation:

- If lower > upper, then return -1 (base case).

- Compare X to A[middle], where middle is the midpoint between lower and upper:

  middle = (lower + upper)/2

- If X == A[middle], return middle (we found it!)

- If X < A[middle], then continue search in A[lower..middle-1]

- If X > A[middle], then continue search in A[middle+1..upper]

Recursive Binary Search of A[lower..upper]

```
int  binSearch(int A[ ], int lower, int upper, int X)
{
     // check base case for missing X
     if (lower > upper)
        return -1;

     // check if X is at the middle
     int middle = (lower + upper)/2;

      if (A[middle] == X)
        return middle;

     if (A[middle] < X)
        return binSearch(A, middle+1, upper, X);
     else
        return binSearch(A, lower, middle-1, X);
}
```

Efficiency of Binary Search

Binary Search is very efficient: large increases in the size of the array require very small increases in the number of basic steps, approximately:

size of array	# steps needed
500	8
1 thousand	10
1 million	20