Binary search algorithm

Definition

Search a *sorted array* by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search *key* is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Generally, to find a value in unsorted array, we should look through elements of an array one by one, until searched value is found. In case of searched value is absent from array, we go through all elements. In average, complexity of such an algorithm is proportional to the length of the array.

Divide in half

A fast way to search a sorted array is to use a binary search. The idea is to look at the element in the middle. If the key is equal to that, the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater, do a binary search of the second half.

Performance

The advantage of a binary search over a linear search is astounding for large numbers. For an array of a million elements, binary search, O(log N), will find the target element with a worst case of only 20 comparisons.

Linear search, O(N), on average will take 500,000 comparisons to find the element.

Algorithm

Algorithm is quite simple. It can be done either recursively or iteratively:

1. get the middle element;
2. if the middle element equals to the searched value, the algorithm stops;
3. otherwise, two cases are possible:
    - searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    - searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now, we should define when iterations should stop?

First case is when searched element is found.

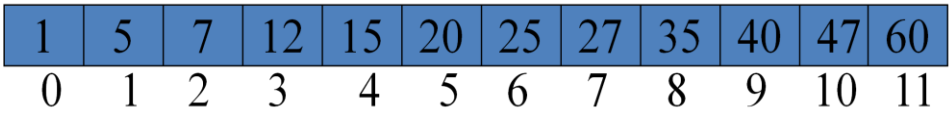
Second one is when subarray has no elements.

In this case,

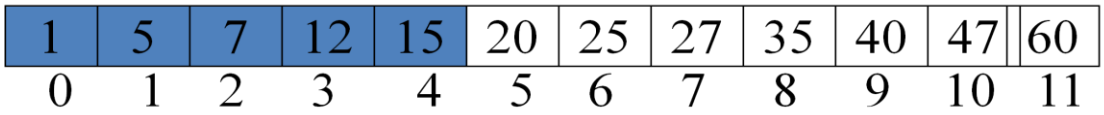
We can conclude, that searched value doesn't present in the array.

Illustration of Binary search

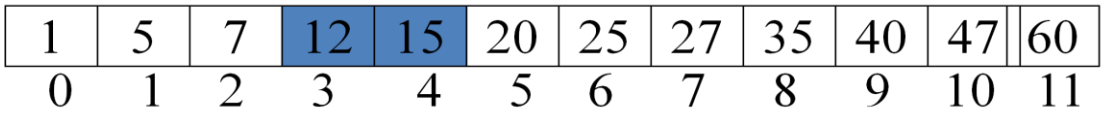
- X[12]:



- Search for b=12
- *mid* = (0+11)/2 = 5. Compare b with X[5]: 12<20.
- So search in left half X[0..4]



- *mid* = (0+4)/2 = 2. Compare b with X[2]: 12 > 7.
- So search right half X[3..4]



- *mid* = (3+4)/2 = 3.Compare b with X[3]: b=X[3]=12.
- Return 3.

The Recursive Code of Binary Search

```
int binarySearch(double b, double X[], int left, int right)
{
        if (left == right)
                if (b==X[left]) return left;
                else return -1;
        int mid = (left+right)/2;
        if (b==X[mid]) return mid;
        if (b < X[mid]) return binarySearch (b, X, left, mid-1);
        if (b > X[mid]) return binarySearch(b, X, mid+1, right);
}
```

Example 1. Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): -1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 5 < 6): -1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6): -1 5 6 18 19 25 46 78 102 114

Example 2. Find 103 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 < 103): 1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 78 < 103): 1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 102 < 103): 1 5 6 18 19 25 46 78 102 114

Step 4 (middle element is 114 > 103): 1 5 6 18 19 25 46 78 102 114

Step 5 (searched value is absent): 1 5 6 18 19 25 46 78 102 114

```
/*
* searches for a value in sorted array
*   arr is an array to search in
*   value is searched value
*   left is an index of left boundary
*   right is an index of right boundary
* returns position of searched value, if it presents in the array
* or -1, if it is absent
*/
int binarySearch(int arr[], int value, int left, int right) {
      while (left <= right) {
            int middle = (left + right) / 2;  //compute mid point.
            if (arr[middle] == value)
                  return middle;          // found it. return position
            else if (arr[middle] > value)
                  right = middle - 1;  // repeat search in bottom half.
            else
                  left = middle + 1; // repeat search in top half.
      }
      return -1;
}
```

Example

```
int binarySearch(int sortedArray[], int first, int last, int key) {
   // function:
   //   Searches sortedArray[first]..sortedArray[last] for key.
   // returns: index of the matching element if it finds key,
   //         otherwise  -(index where it could be inserted)-1.
   // parameters:
   //   sortedArray in  array of sorted (ascending) values.
   //   first, last in  lower and upper subscript bounds
   //   key         in  value to search for.
   // returns:
   //   index of key, or -insertion_position -1 if key is not
   //                 in the array. This value can easily be
   //                 transformed into the position to insert it.

   while (first <= last) {
       int mid = (first + last) / 2;  // compute mid point.
       if (key > sortedArray[mid])
           first = mid + 1;  // repeat search in top half.
       else if (key < sortedArray[mid])
           last = mid - 1; // repeat search in bottom half.
       else
           return mid;     // found it. return position /////
   }
   return -(first + 1);    // failed to find key
}
```

Complexity analysis

Huge advantage of this algorithm is that its complexity depends on the array size logarithmically **in worst case.** In practice it means, that algorithm will do at most log2(n) iterations, which is a very small number even for big arrays. It can be proved very easily. Indeed, on every step the size of the searched part is reduced by half. Algorithm stops, when there are no elements to search in. The binary search algorithm time complexity is O(log2(n)).