Divide-and-conquer

Binary search

Dr. Bibhudatta Sahoo
Communication & Computing Group
Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Divide-and conquer

- Divide-and conquer is a general algorithm design paradigm:
  - **Divide**: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - **Recur**: solve the subproblems recursively
  - **Conquer**: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using **recurrence equations**

Binary search

- The binary search consists of examining a middle value of a list to see which half contains the desired value.
- The middle value of the appropriate half is then examined to see which half of the half contains the value in question.
- This halving process is continued until the value is located or it is determined that the value is not in the list. Precondition.
- Binary Search is O($\mathbf{log_2 n}$), because it takes approximately $\mathbf{log_2 n}$ passes to find the target element.

Assumption: pre-condition Binary search

Input array to be in sorted order.

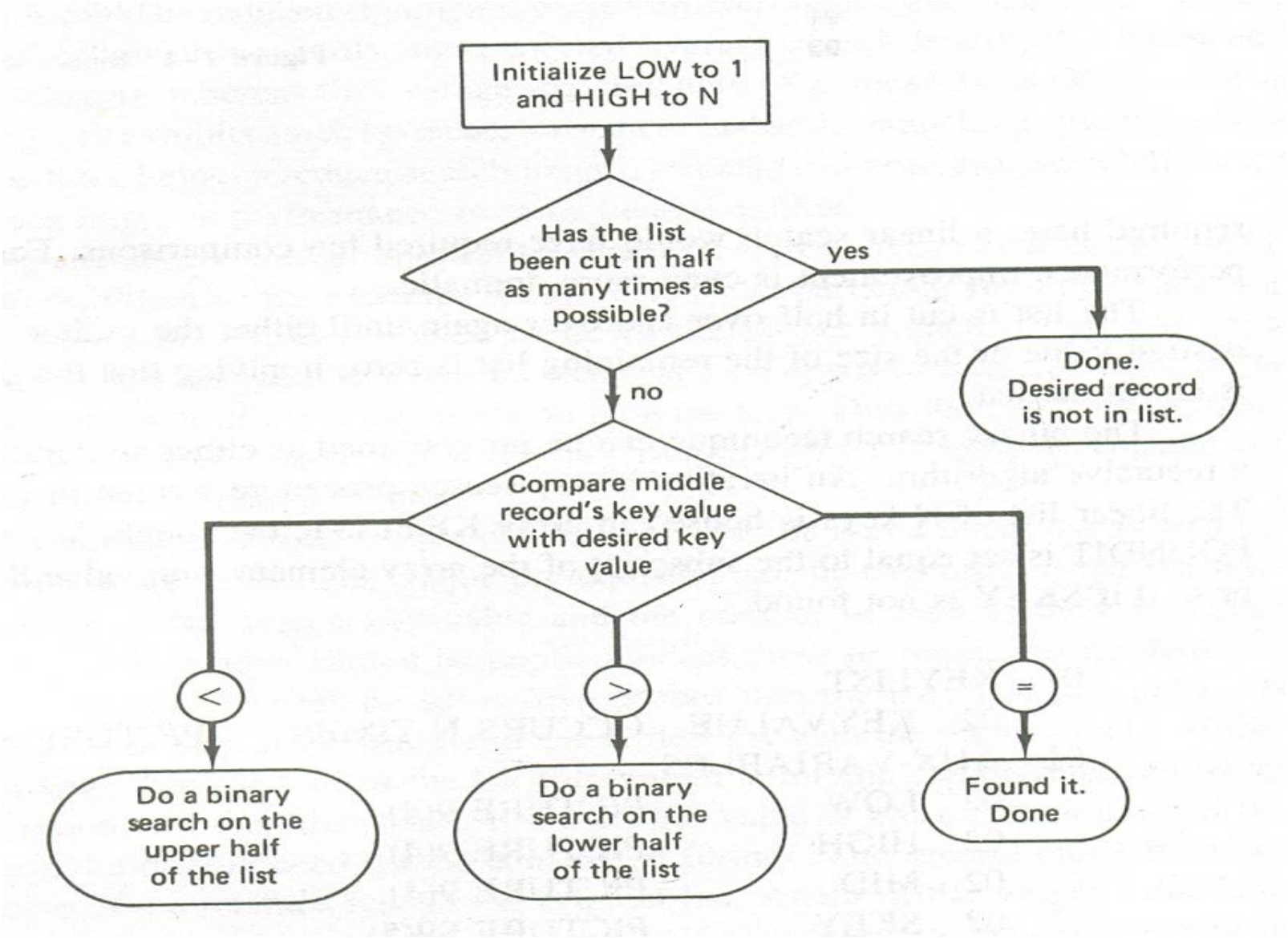
All the element in the array must be unique.

- **Binary search compares the target value to the item in the middle of the list.**
- **If the target is smaller than the middle item, the search examines the items in the left half of the list.**
- **If the target is bigger than the middle item, the search continues in the right half of the list.**
- **If the target matches the middle item, the search is finished.**
- **Each time the algorithm compares the target to the middle item, it divides the list in two halves.**

Binary Search

- Binary search
  - Check whether the middle element in the array is the desired item.
  - If not, determine which half of the array the desired item must be in, and discard the other half.
  - Repeatedly divide the array in half, until the desired item is the middle element of an array, or it is not found.
- Binary search requires the array to be sorted.
- The binary search algorithm searches successively smaller arrays, each of which is a constant fraction of the array previously searched: the size of a given array is approximately one-half the size of the array previously searched.

Binary Search

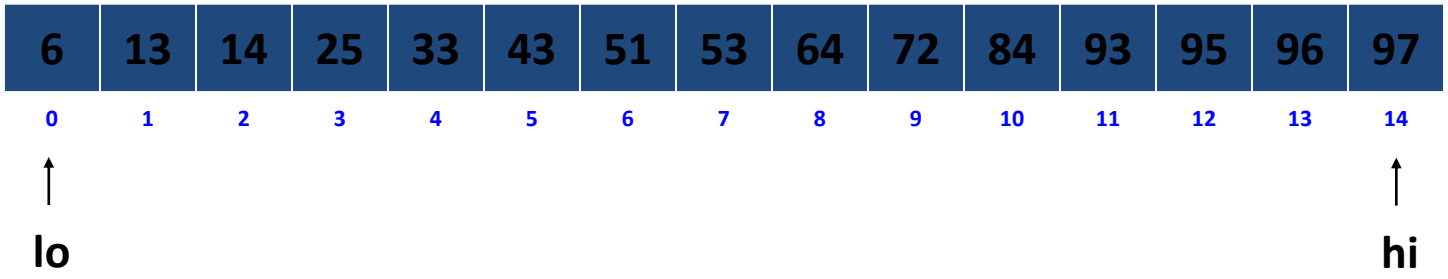


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains $a[lo] \leq value \leq a[hi]$.

Ex. **Binary search for 33**.

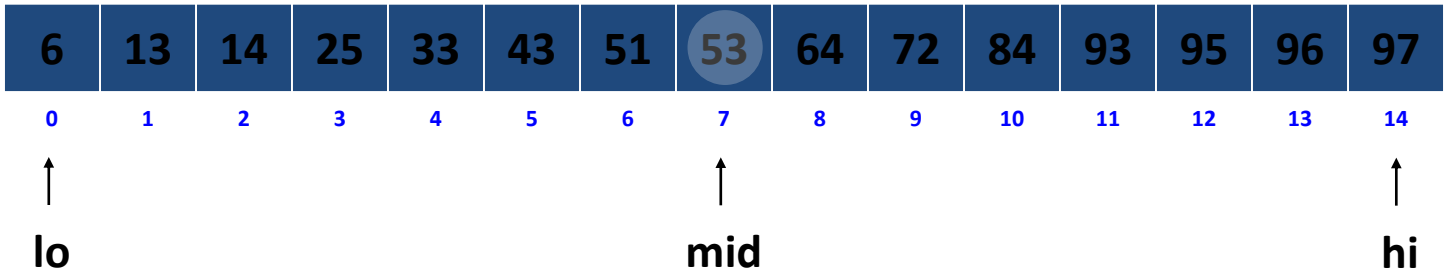


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] $\leq$ value $\leq$ a[hi].

Ex. **Binary search for 33**.

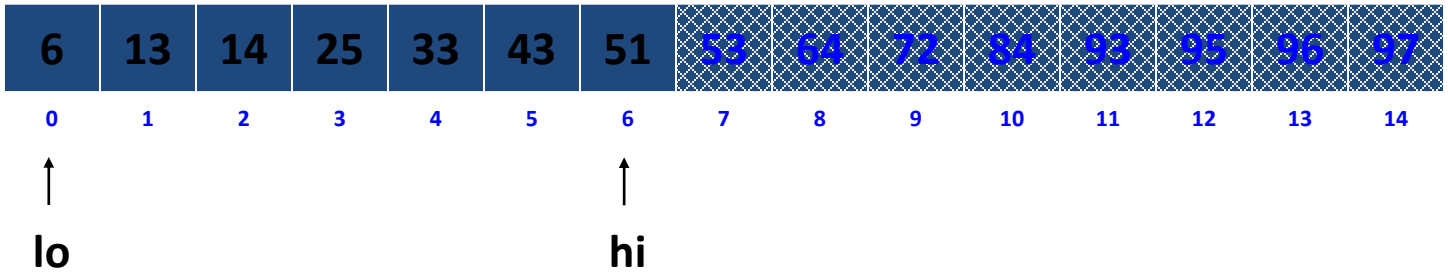


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. Binary search for 33.

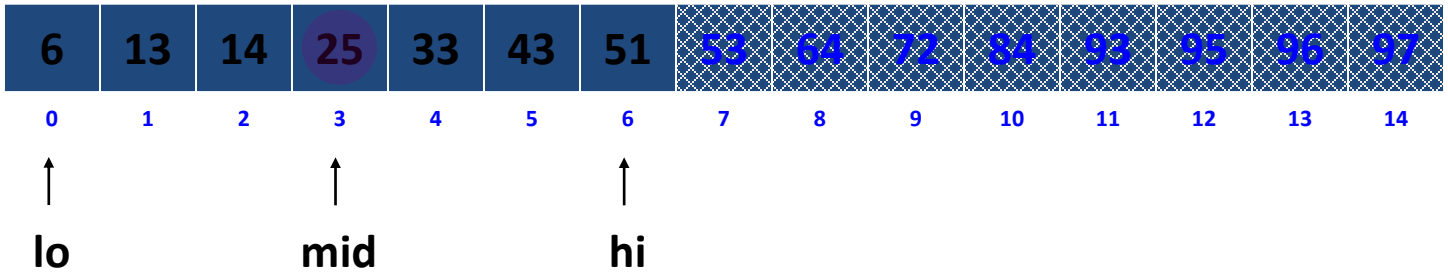


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. Binary search for 33.

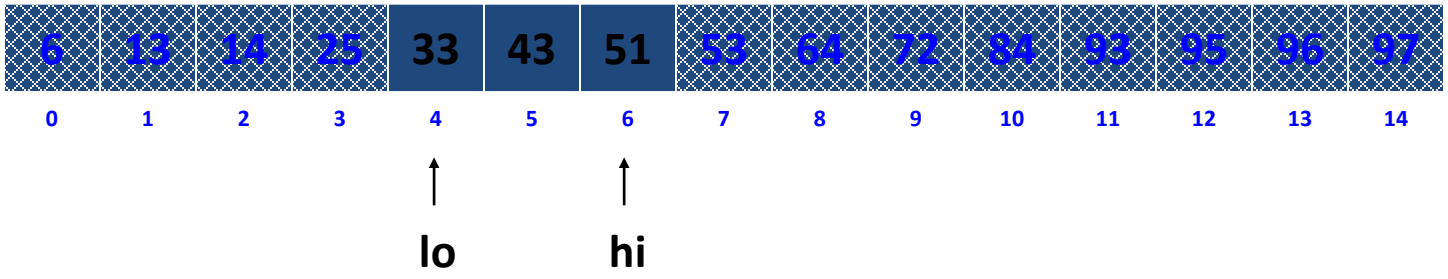


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] $\leq$ value $\leq$ a[hi].

Ex. **Binary search for 33**.

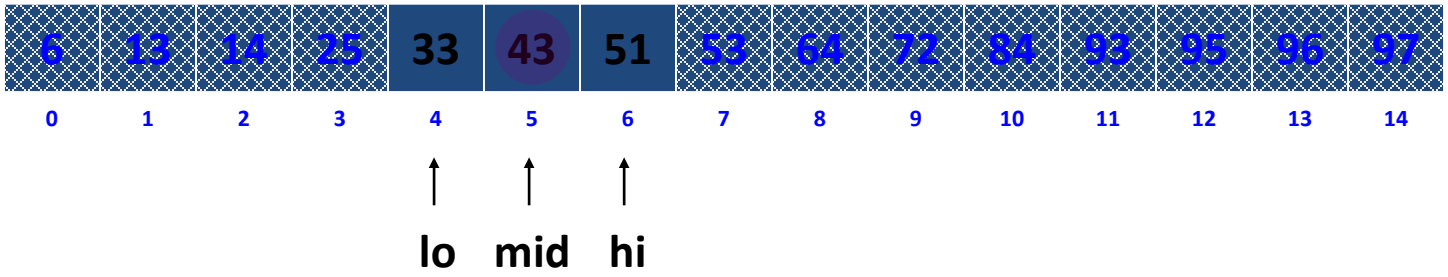


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. **Binary search for 33.**

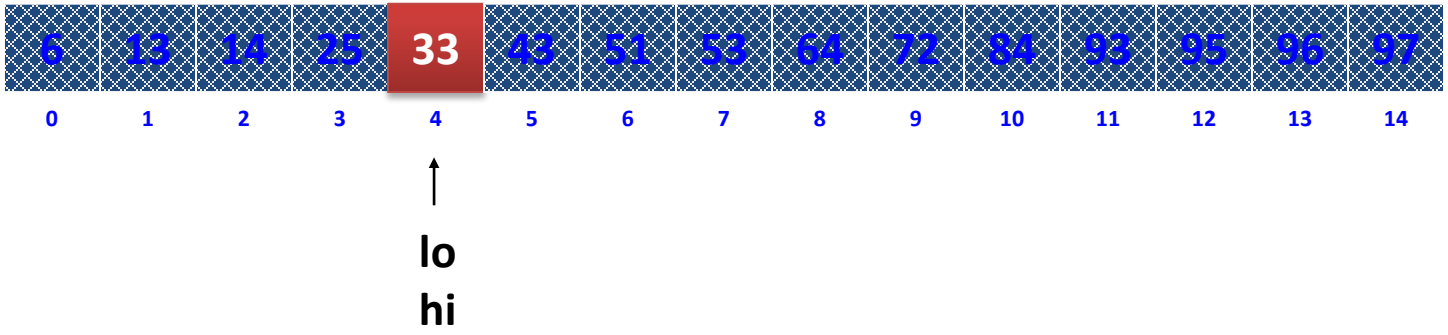


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. **Binary search for 33**.

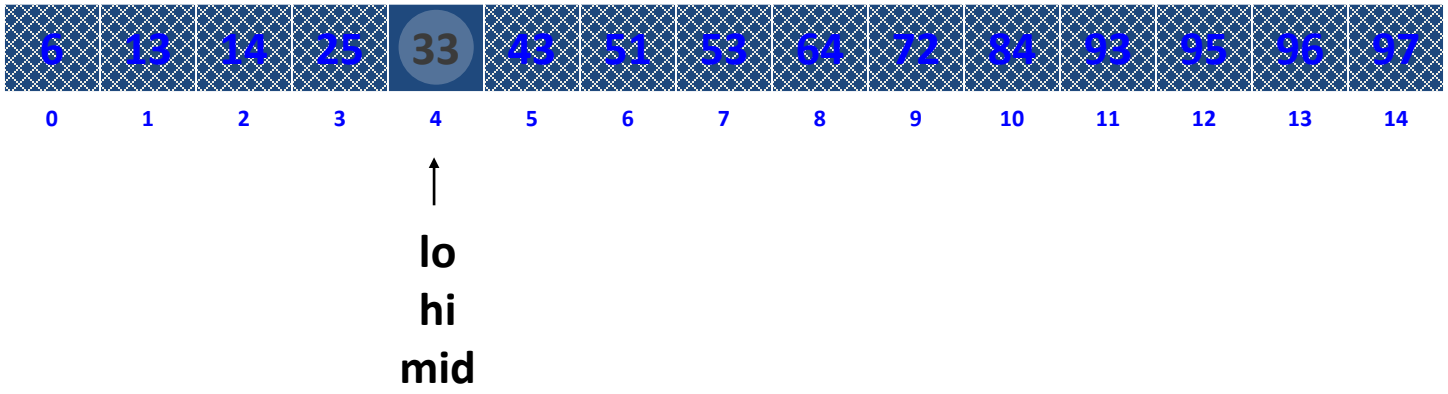


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. Binary search for 33.

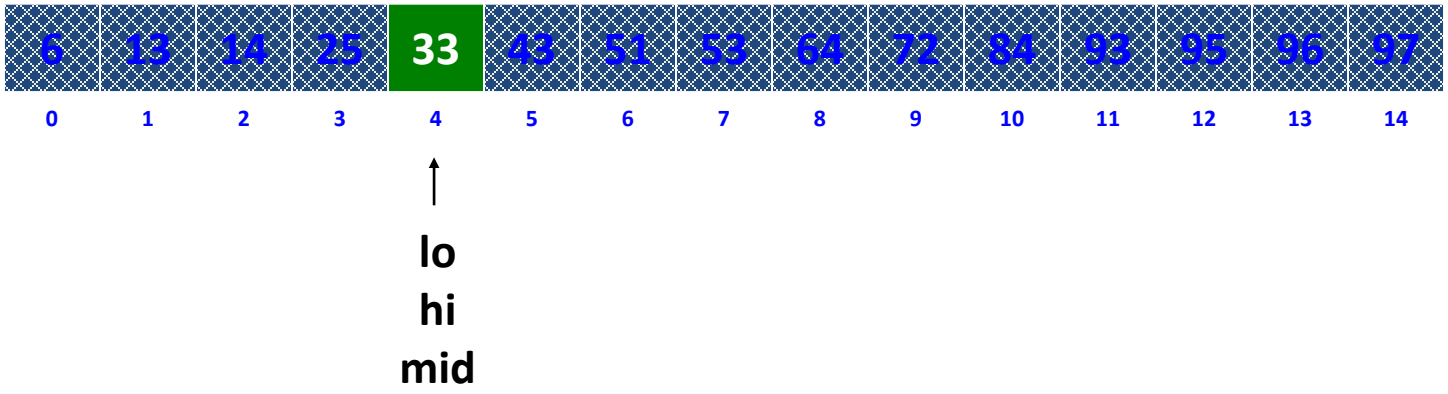


Binary Search

Binary search. Given value and sorted array a[], find index i such that a[i] = value, or report that no such index exists.

Invariant. Algorithm maintains a[lo] ≤ value ≤ a[hi].

Ex. **Binary search for 33**.



Binary search

- Binary search is an o (log n) algorithm for searching in sorted arrays.

Iterative binary search

```
1.  Algorithm BinSearch(a, n, x)
2.  / / Given an array a[1 : n] of elements in non-decreasing
3.  / / order, n ≥ 0,  determine whether x is present, and
4.  / / if so, return j such that x = a[j]; else return 0.
5.  {
6.          low := 1; high := n;
7.          while (low ≤ high) do
8.  {
9.           mid:= ⌊(low + high)/2⌋;
10. if (x < a[mid]) then high := mid - 1;
11. else if (x > a[mid]) then low := mid + 1;
12. else return mid;
13. }
14. return 0;
15. }
```

Recursive binary search

```
1.   Algorithm BinSrch(a, i, l, x)
2.   / / Given an array a[i : l] of elements in nondecreasing
3.   / / order, 1 ≤ i ≤ l, determine whether x is present, and
4.   / / if so, return j such that x = a[j]; else return 0.
5.   {
6.   if (l = i) then / / If Small(P)
7.   { ..
8.            if (x = a[i]) then return i;
9.   else return 0;
10.  }
11.  else
12.  { / / Reduce P into a smaller subproblem.
13.  mid := ⌊(i + l)/2⌋;
14.  if (x = a[mid]) .then return mid;
15.  else if (x < a[mid]) then
16.            return BinSrch(a, i, mid - 1, x);
17.     else   return BinSrch(a, mid + 1, l, x);
18.  }
19.  }
```

Analysis of Binary search

$$T(1) = c \qquad (\text{assume} \quad N = 2^k)$$

$$T(N) = T(N/2) + f$$

$$T(N) = T(N/4) + 2f$$

$$T(N) = T(N/2^k) + kf = c + kf$$

$$T(N) = c + k \log N = O(\log N)$$

Analysis of Binary search

- Let us assume for the moment that the size of the array is a power of 2, say $n = 2^k$.
- Each time in the while loop, when we examine the middle element, we cut the size of the sub-array into half. So before the 1^{st} iteration size of the array is 2^k.
- After the 1^{st} iteration size of the sub-array of our interest is: 2^{k-1}
- After the 2^{nd} iteration size of the sub-array of our interest is: 2^{k-2}

  ⋮
- After the k^{th}. iteration size of the sub-array of our interest is : 2^{k-k}
- So we stop after the next iteration. Thus we have at most (k+1) = (log n +1) iterations.
- Since with each iteration, we perform a constant amount of work: Computing a mid point and few comparisons. So overall, for an array of size n, we perform C (log n + 1) = O(log n) computation,
- Thus T(n) = O(log)

A function prototype for a binary Search named BinarySearch

```
int BinarySearch (int a[], int key, int low, int high, int size);

Here's the function that goes with it.

int BinarySearch (int a[], int key, int size)
{
   int middle, low = 0, high = size - 1;
    while (low <= high) {
      middle = (low + high) / 2;

      if (key == a[middle])
         return middle;
      else if (key < a [middle])
         high = middle -1;        // search low end of array
      else
         low = middle + 1;        // search high end of array
    }
       return -1;
 }
```

Binary Search

Recursive solution

```
public static int binarySearch(
    int[] A, int first, int last,
    int value)
{
  if (first > last)
    return -1;
  else {
    int mid = (first + last) / 2;
    if (value == A[mid])
      return mid;
    else if (value < A[mid])
      return binarySearch(
        A, first, mid-1, value);
    else
      return binarySearch(
        A, mid+1, last, value);
  }
}
```

Iterative solution

```
public static int binarySearch(
    int[] A, int value)
{
  int low = 0;
  int high = A.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;
    if (value == A[mid])
      return mid;
    else if (value < A[mid])
      high = mid - 1;
    else
      low = mid + 1;
  }

  return -1;
}
```

Binary search using one comparison per cycle

```
1.  Algorithm BinSearch1(a, n, x)
2.  // Same specifications as BinSearch except n > 0
3.  {
4.  low := 1; high := n + 1;
5.  // high is one more than possible.
6.  while (low < (high - 1)) do
7.  {
8.  mid := ⌊(low + high) /2⌋;
9.  if (x < a[mid]) then high := mid;
10. / / Only one comparison in the loop.
11. else low := mid; / / x ≥ a[mid]
12. }
13. if (x = a[low]) then return low; / / x is present.
14. else return 0; / / x is not present.
15. }
```

Binary Search: Analysis

- The number of comparisons in the binary search algorithm is equal to one plus the number of times that it divides the array in half.
- Best-case analysis
  - The best case is when the desired item is the middle item of the original undivided array.
  - Only one comparison is performed.
  - Complexity: O(1)
- Worst-case analysis (n is a power of 2)
  - Suppose that $n = 2^k$ for some k, so $k = \log n$.
  - To simplify analysis, assume that the middle item is also included in the first half of the array when division needs to be done.
  - After the i^{th} division, the size of the array is $n/2^i$.
  - The worst case happens when the binary search algorithm repeatedly divides the array in half until only one item remains.
  - k divisions and thus k+1 comparisons are performed ($n/2^k = 1$).
  - Complexity: O(k+1) = O(k) = O(log n)

Binary Search: Analysis

- **Worst-case analysis (n is not a power of 2)**
  - If n is not a power of 2, then there exists an integer k such that $2^{k-1} < n < 2^k$.
  - It still requires at most k divisions to obtain a sub-array with one item, and so at most k+1 comparisons are performed in the worst case.
  - It follows that $k-1 < \log n < k$ and therefore $k < \log n + 1$.
  - **Complexity: $O(k+1) = O((\log n + 1) + 1) = O(\log n)$**

Improved insertion: Application of Binary search

- Binary search technique can be used to improve the insertion into an ordered array list

- First, using the binary search to find the location where the new element should be inserted

- Second, insert the new element into the array list

BinaryInsertionSort

```
1.  Algorithm BinaryInsertionSort(A[0…n-1])
2.  //Sorts a given array by binary insertion sort
3.  //Input: An array A[0…n-1] of n orderable elements
4.  //Output: Array A[0…n-1] sorted in nondecreasing order
5.  for i ← 1 to n -1 do
6.      v ← A[i];
7.       left ← 0;
8.       right ← i;
9.      while left < right do
10.                 middle ← (left + right) / 2;
11.                 if v ≥ A[middle]
12.                         left ← middle + 1;
13.                 else
14.                         right ← middle;
15.     for j ← i to left + 1 do
16.                 A[j] ← A[j-1];
17.     A[left] ← v
```

Analysis of the Binary Search

- The best-case running time of a binary search of n elements is O(1), corresponding to when we find the element for which we are looking on our first comparison.
- The worst-case running time is O(log n).

  - Worst case is when item X is not found.
  - How many iterations are executed before Low > High?
  - After first iteration: N/2 items remaining
  - 2^{nd} iteration: (N/2)/2 = N/4 remaining
  - Kth iteration: ?

Worst case running time for Binary search

- How many iterations are executed before Low > High
- After first iteration: N/2 items remaining
- 2^{nd} iteration: (N/2)/2 = N/4 remaining
- Kth iteration: $N/2^K$ remaining
- Worst case: Last iteration occurs when $N/2^K \geq 1$ and $N/2^{K+1} < 1$ item remaining

  ⇨ $2^K \leq N$ and $2^{K+1} > N$ [take log of both sides]
- Number of iterations is $K \leq \log N$ and $K > \log N - 1$
- Worst case running time = $\Theta(\log N)$

Average case analysis of Binary search

Average case running time of binary search

- Let us consider the case where each key in the array is equally likely to be searched for, and what the average time is to find such a key.

To make the analysis simpler, we have the following assumptions

(1) The value we are searching for is in the array.

(2) Each value is equally likely to be in the array.

(3) let us assume that $n = 2^k - 1$, for some integer $k \geq 1$. (Why does it make the analysis simpler?)

- The first assumption isn't necessary, but makes life easier so we don't have to assign a probability to how often a search fails.
- The second assumption is necessary since we don't actually know how often each value would be searched for.
- The third assumption will make our math easier since the sum we will have to calculate will more easily follow a pattern. (Our general result we obtain will still hold w/o this assumption.)

- First, we note that using 1 comparison, we can find 1 element. If we use two comparisons exactly, there are 2 possible elements we can find. In general, after using k comparisons, we can find 2^{k-1} elements. (To see this, consider doing a binary search on the array 2, 5, 6, 8, 12, 17, 19. 8 would be found in 1 comparison, 5 and 17 in two, and 1, 6, 12 and 19 would be found in 3 comparisons.)
- The expected number of comparisons we make when running the algorithm would be a sum over the number of comparisons necessary to find each individual element multiplied by the probability we are searching for that element. Let p(j) represent the number of comparisons it would take to find element j, then the sum we have is:

$$\sum_{j=1}^{n} \frac{1}{n} p(j) = \frac{1}{n} \sum_{j=1}^{n} p(j)$$

$$\sum_{j=1}^{n} \frac{1}{n} p(j) = \frac{1}{n} \sum_{j=1}^{n} p(j)$$

- we have already out lined that p(j) will be 1 for one value of j, 2 for 2 values of j, 3 for 4 values of j, etc. Since $n=2^k-1$, we can formulate the sum as follows:

$$\sum_{j=1}^{n} \frac{1}{n} p(j) = \frac{1}{n} \sum_{j=1}^{n} p(j) = \frac{1}{n} \sum_{j=1}^{k} j 2^{j-1}$$

- This is because the value j appears exactly 2^{j-1} times in the original sum.

$$\sum_{j=1}^{k} j2^{j-1} = 1(2^0) + 2(2^1) + \ldots\ldots\ldots\ldots + k(2^{k-1})$$

$$-2\sum_{j=1}^{k} j2^{j-1} = 1(2^1) + 2(2^2) + \ldots + (k-1)(2^{k-1}) + k(2^k)$$

Subtracting the bottom equation from the top, we get the following:

$$-\sum_{j=1}^{k} j2^{j-1} = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} - k2^k$$

$$-\sum_{j=1}^{k} j2^{j-1} = 2^k - 1 - k2^k$$

$$\sum_{j=1}^{k} j2^{j-1} = -2^k + 1 + k2^k$$

$$\sum_{j=1}^{k} j2^{j-1} = (k-1)2^k + 1$$

$$\sum_{j=1}^{k} j2^{j-1} = (k-1)2^k + 1$$

Thus, the average run-time of the binary search is

$$\frac{(k-1)2^k + 1}{n} = \frac{(k-1)2^k + 1}{2^k - 1} \approx k - 1 = O(\log n)$$

So, for this particular algorithm, the average case run-time is much closer to the worst case run-time than the best-case run time. (Notice that the worst case number of comparisons is k with the average number of comparisons is k-1, where k = log n.)

Thanks for Your Attention!

Exercises

Exercises: Binary search: 1

1. Devise a "binary" search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other. How does this algorithm compare with binary search?
2. Devise a ternary search algorithm that first tests the element at position *n/3* for equality with some value *x,* and then checks the element at *2n/3* and either discovers *x* or reduces the set size to one-third the size of the original. Compare this with binary search.
3. Prove that you can never *search* unknown data in faster than $\Omega(\log n)$ time using a decision-based search algorithm.

Exercise: Binary search :2

4. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains *n* numerical values, so there *2n* values total-and you may assume that no two values are the same. You'd like to determine the median of this set of *2n* values, which we will define here to be the n^{th} smallest value. However, the only way you can access these values is through quaeries to the databases. In a single query, you can specify a value *k* to one of the two databases, and the chosen database will return the k th smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most O(log n) queries.

Exercise: Binary search : 3

5. Given a sorted array of distinct integers A[1…n], you want to find out whether there is an index i for which A[i] = i. Given a dive-and-conquer algorithm that runs in time O(logn).
6. Consider the modified binary search algorithm so that it splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write down the recurrence for this ternary search algorithm and find the asymptotic complexity of this algorithm.
7. Consider another variation of the binary search algorithm so that it splits the input not only into two sets of almost equal sizes, but into two sets of sizes approximately one third and two-thirds. Write down the recurrence for This search algorithm and find the asymptotic complexity of this algorithm?

Exercise: Binary search : 4

8. Binary search takes $\theta(\log n)$ time, where n is the size of the array. Write an algorithm that takes $\theta(\log n)$ time to search for a value x in a sorted array of n positive integers, where you do not know the value of n in advance. You may assume that the array has 0's stored after the last positive number. Prove that your algorithm has the correct time complexity.

Applications of Binary search

Twenty Questions

- In Twenty questions one player selects a word, and the other repeatedly asks true/false questions in an attempt to identify the word. If the word remains unidentified after 20 questions, the first party wins; otherwise, the second player wins. In fact, the second player always has a winning strategy, based on binary search. Given a printed dictionary, the player opens it in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain 50,000 to 200,000 words, we can be certain that the process will always terminate within twenty questions.

Finding a Transition

- Suppose we have an array A consisting of a run of 0's, followed by an unbounded run of 1's, and would like to identify the exact point of transition between them:

  0000000000000000000000011111111111

  Binary search on the array would provide the transition point in ⌈lg n⌉ tests. Clearly there is no way to solve this problem any faster.

One-Sided Binary Search

- Suppose that we want to search in a sorted array, but we do not know how large the array is. All we know is the starting point. {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, .. .} How can we use binary search without both boundaries? In the absence of such a bound, we can test repeatedly at larger intervals (A[1], A[2], A[4], A[8], A[16], . . .) until we find a first nonzero value. Now we have a window containing the target and can proceed with binary search. This one-sided binary search finds the transition point p using at most 2⌊lg p⌋ comparisons, regardless of how large the array actually is. One-sided binary search is most useful whenever we are looking for a key that probably lies close to our current position.

Square and Other Roots

- The square root of n is the number r such that $r^2 = n$. Square root computations are performed inside every pocket calculator – but how? Observe that the square root of $n \geq 1$ must be at least 1 and at most n. Let $l = 1$ and $r = n$. Consider the midpoint of this interval, $m = (l + r)/2$. How does m^2 compare to n? If $n \geq m^2$, then the square root must be greater than m, so the algorithm repeats with $l = m$. If $n < m^2$, then the square root must be less than m, so the algorithm repeats with $r = m$. Either way, we have halved the interval with only one comparison. Therefore, after only $\lg n$ rounds we will have identified the square root to within ±1. This bisection method, as it is called in numerical analysis, can also be applied to the more general problem of finding the roots of an equation. We say that x is a root of the function f if $f(x) = 0$.

Home work for binary search

1. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

b. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.

c. Find the average number of key comparisons made by binary search in a successful search in this array. (Assume that each key is searched for with the same probability.)

d. Find the average number of key comparisons made by binary search in an unsuccessful search in this array. (Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.)