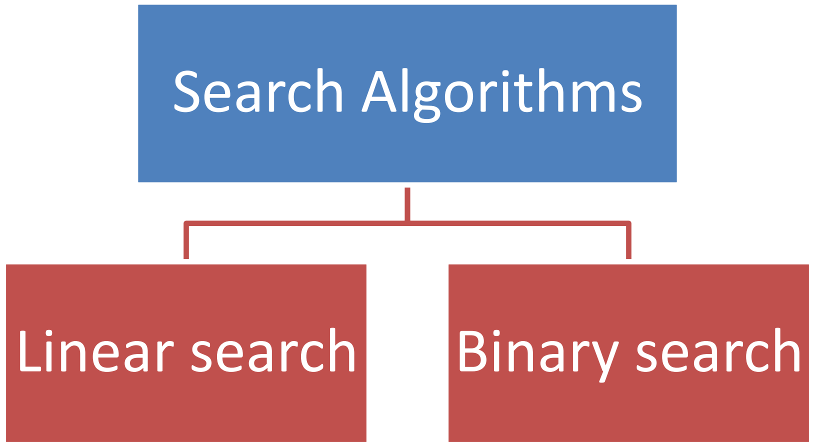
UNIVERSITY COLLEGE OF APPLIED SCIENCES

ALGORITHMS AND DATA STRUCTURES

SEARCH ALGORITHMS

SEARCH ALGORITHMS



Linear search

Linear searches operate in much the same way as the searches in the unordered array ,it steps along, looking for a match till reaching the end of the array

The difference between using ordered array or using unordered array is that in the ordered array, the search quits if an item with a larger (or small) value is found.

```
public static int linearSearch(int[] arr, int key){
    for(int i=0;i<arr.length;i++){
      if(arr[i] == key){
        return i; }    }
    return -1;  }
```

Time complexity for Linear Search is O(n).

Binary search

This kind of search is much faster than a linear search, especially for large arrays.

Binary search uses the same approach to guess a number in the well-known children's guessing game. In this game, a friend asks you to guess a number he's thinking of between 1 and 100. When you guess a number, he'll tell you one of three things:

- ✓ Your guess is larger than the number he's thinking of
- ✓ it's smaller,
- ✓ you guessed correctly

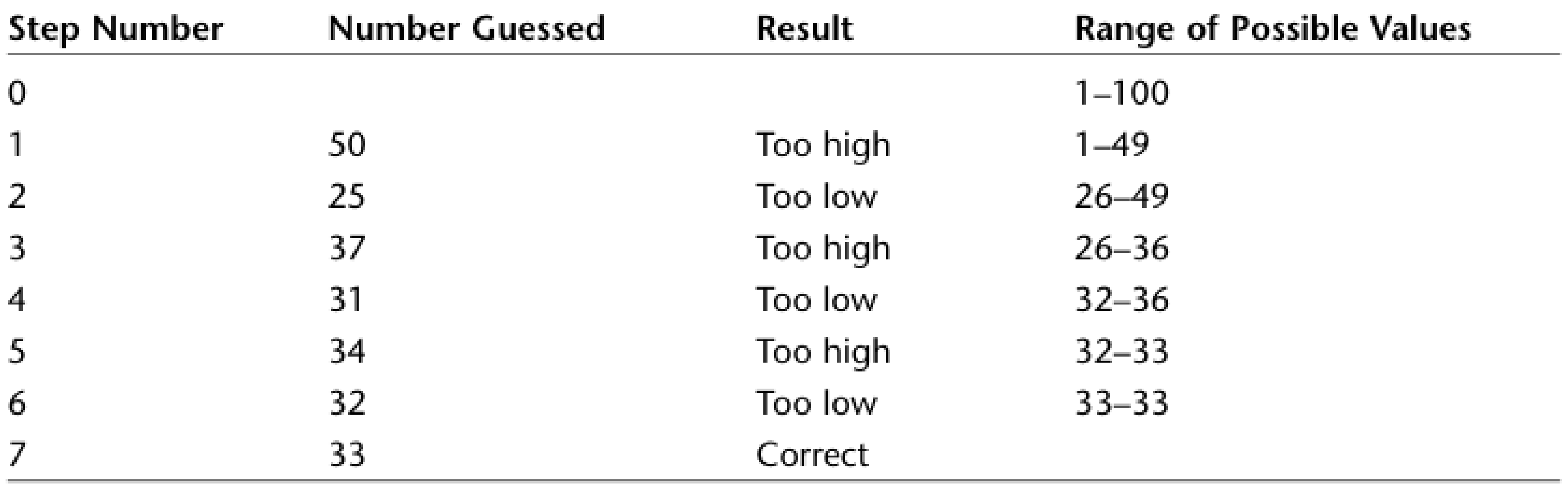
Binary search

To find the number in the fewest guesses, you should always start by guessing 50. If your friend says your guess is too low, you deduce the number is between 51 and 100, so your next guess should be 75 (halfway between 51 and 100). If he says it's too high, you deduce the number is between 1 and 49, so your next guess should be 25. Each guess allows you to divide the range of possible values in half. Finally, the range is only one number long, and that's the answer.

Notice how few guesses are required to find the number. If you used a linear search, guessing first 1, then 2, then 3, and so on, finding the number would take you, on the average, 50 guesses. In a binary search each guess divides the range of possible values in half, so the number of guesses required is far fewer

Binary search

The picture below shows the session when the number to be guessed is 33.



Step Number	Number Guessed	Result	Range of Possible Values
0			1–100
1	50	Too high	1–49
2	25	Too low	26–49
3	37	Too high	26–36
4	31	Too low	32–36
5	34	Too high	32–33
6	32	Too low	33–33
7	33	Correct	

The correct number is identified in only seven guesses. This is the maximum. You might get lucky and guess the number before that

Java Code for Binary Search with an Ordered Array

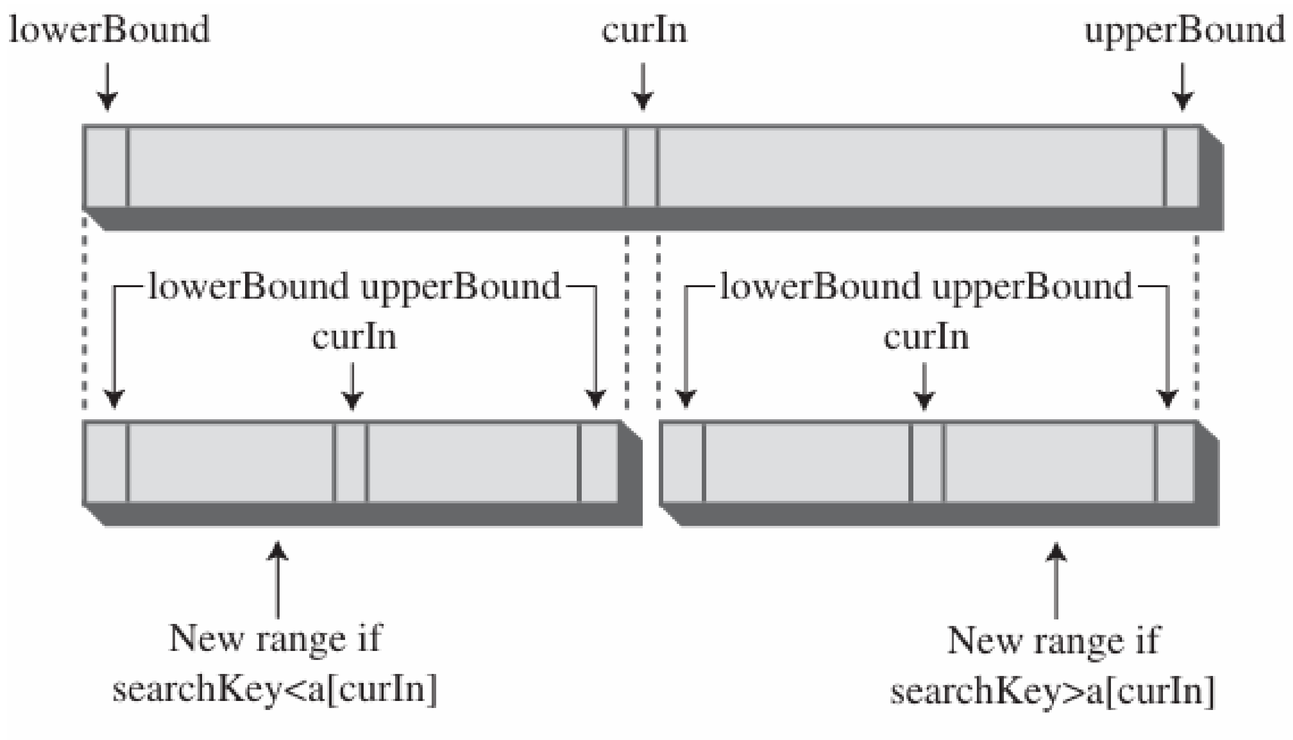
```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // found it
      else if(lowerBound > upperBound)
         return nElems;             // can't find it
      else                          // divide range
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1; // it's in upper half
         else
            upperBound = curIn - 1; // it's in lower half
         }  // end else divide range
      }  // end while
   }  // end find()
```

Please find the complete code page **59** in text book

Time complexity for binary Search is O(log n)

Binary search



Dividing the range in a binary search.

Advantages of Ordered Arrays

What have we gained by using an ordered array? The major advantage is that search times are much faster than in an unordered array. The disadvantage is that insertion takes longer because all the data items with a higher key value must be moved up to make room. Deletions are slow in both ordered and unordered arrays because items must be moved down to fill the hole left by the deleted item. Ordered arrays are therefore useful in situations in which searches are frequent, but insertions and deletions are not.

Advantages of Ordered Arrays

An ordered array might be appropriate for a data base of company employees, for example. Hiring new employees and laying off existing ones would probably be infrequent occurrences compared with accessing an existing employee’s record for information, or updating it to reflect changes in salary, address, and so on. A retail store inventory, on the other hand, would not be a good candidate for an ordered array because the frequent insertions and deletions, as items arrived in the store and were sold, would run slowly.

Why Not Use Arrays for Everything?

In an unordered array you can insert items quickly, in O(1) time, but searching takes slow O(N) time.
In an ordered array you can search quickly, in O(logN) time, but insertion takes O(N) time.
For both kinds of arrays, deletion takes O(N) time because half the items (on the average) must be moved to fill in the hole.
arrays have fixed size when they are first created. Usually, when the program first starts, you don't know exactly how many items will be placed in the array later, so you guess how big it should be. If your guess is too large, you'll waste memory by having cells in the array that are never filled. If your guess is too small, you'll overflow the array, causing at best a message to the program's user, and at worst a program crash.

It would be nice if there were data structures that could do everything–insertion, deletion, and searching–quickly, ideally in O(1) time, but if not that, then in O(logN) time.

Other data structures (linked list for example) are more flexible and can expand to hold the number of items inserted in them.