Search Algorithms

Sequential or Linear Search

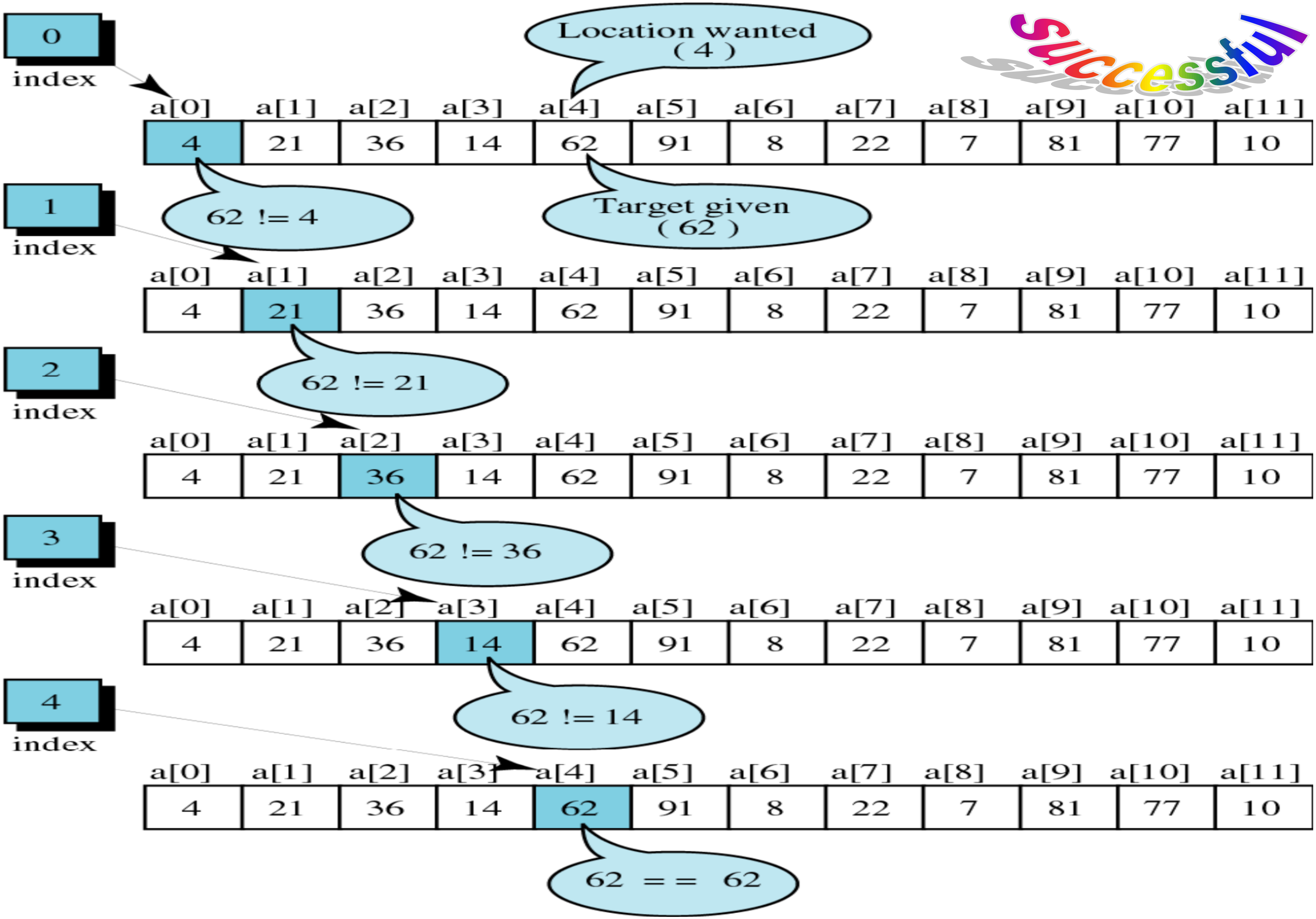
Indexed Sequential Search

Binary Search

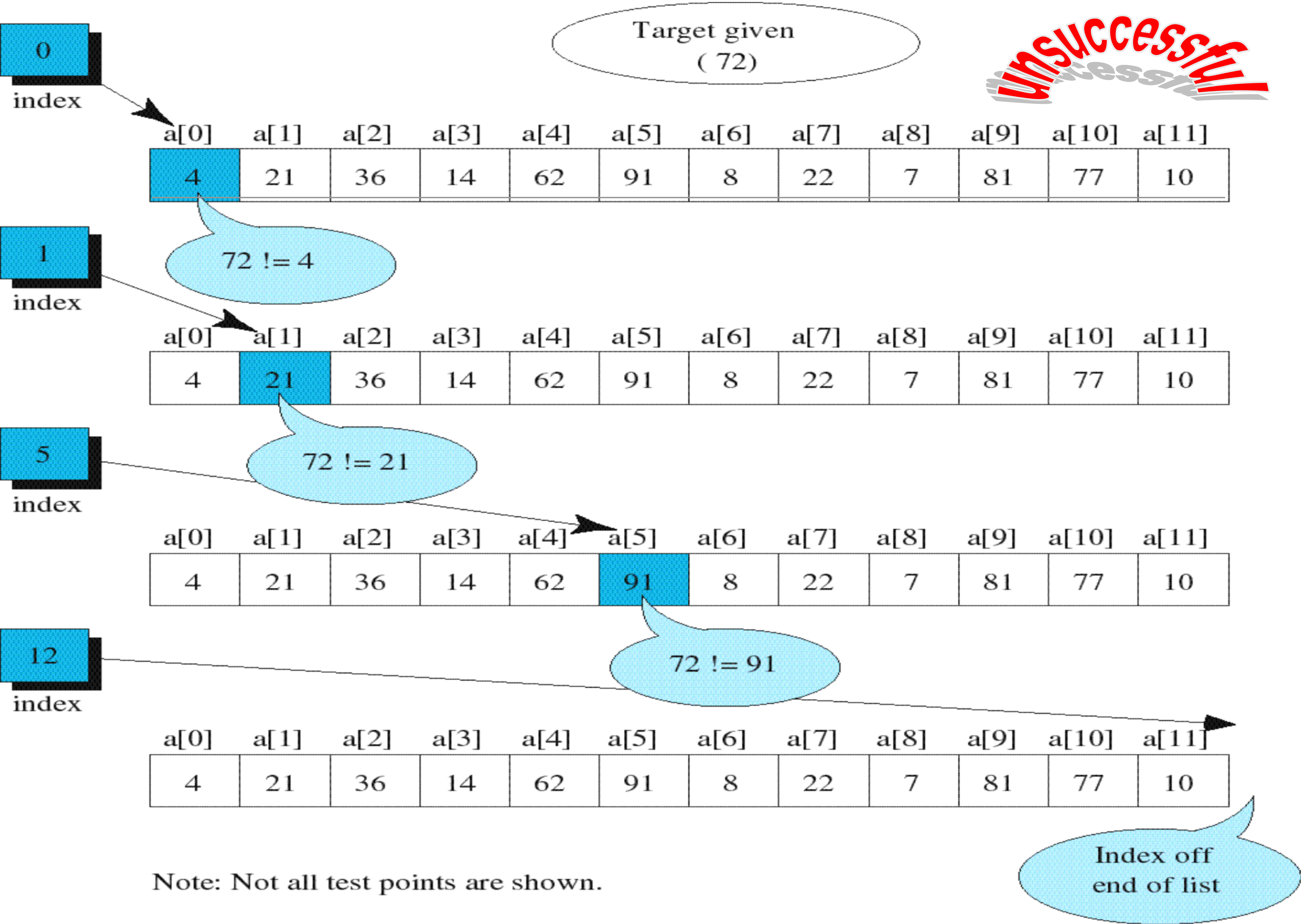
Searching

The process used to find the location of a target among a list of objects

Searching an array finds the index of first element in an array containing that value



Location wanted
(4)
successful
0
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 4
Target given
(62)
1
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 21
2
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 36
3
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 != 14
4
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
62 = = 62


Target given
(72)
unsuccessful
0
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
72 != 4
1
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
72 != 21
5
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
72 != 91
12
index
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 21 36 14 62 91 8 22 7 81 77 10
Index off
end of list
Note: Not all test points are shown.

Unordered Linear Search

- Search an unordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
14	*2*	*10*	*5*	*1*	*3*	*17*	*2*

- Algorithm:

```
Start with the first array element (index 0)
while(more elements in array){
      if value found at current index, return index;
      Try next element (increment index);
}
Value not found, return -1;
```

Unordered Linear Search

```
// Searches an unordered array of integers
int search(int data[],  //input: array
           int size,    //input: array size
           int value){  //input: search value
    // output: if found, return index;
    //         otherwise, return -1.
    for(int index = 0; index < size; index++){
        if(data[index] == value)
            return index;
    }
    return -1;
}
```

Ordered Linear Search

- Search an ordered array of integers for a value and return its index if the value is found; Otherwise, return -1.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	*2*	*3*	*5*	*7*	*10*	*14*	*17*

- Linear search can stop immediately when it has passed the possible position of the search value.

Ordered Linear Search

- Algorithm:

```
Start with the first array element (index 0)
while(more elements in the array){
      if value at current index is greater than value,
             value not found, return -1;
      if value found at current index, return index;
      Try next element (increment index);
}
value not found, return -1;
```

Ordered Linear Search

```
// Searches an ordered array of integers
int lsearch(int data[],// input: array
            int size,  // input: array size
            int value  // input: value to find
            ) {       // output: index if found
    for(int index=0; index<size; index++){
        if(data[index] > value)
            return -1;
        else if(data[index] == value)
            return index;
    }
    return -1;
}
```

Efficiency of Linear Search

- **Best Case** Find at first place - **one comparison**
- **Worst Case** Find at *n*th place or not at all - ***n* comparisons**
- **Average Case** It is shown below that this case takes - **(*n*+1)/2** comparisons
  - In considering the average case there are n cases that can occur, i.e. find at the first place, the second place, the third place and so on up to the *n*th place. If found at the *i*th place then *i* comparisons are required. Hence the average number of comparisons over these n cases is:
  - average = (1+2+3.....+n)/n = (n+1)/2 where the result was used that 1+2+3 ...+*n* is equal to *n*(*n*+1)/2.
- Hence Linear Search is an **O(n)**

Indexed Sequential Search

- Another technique to improve searching efficiency in an ordered array.
- A sorted index is set aside in addition to the array
- Each element in the index points to a block of elements in the array
  - e.g., block of 10 or 20 elements
- The index is searched before the array and guides the search in the array
  - Sequential search is limited to smaller index table and a smaller part of the array itself
- Involves an increase in space complexity

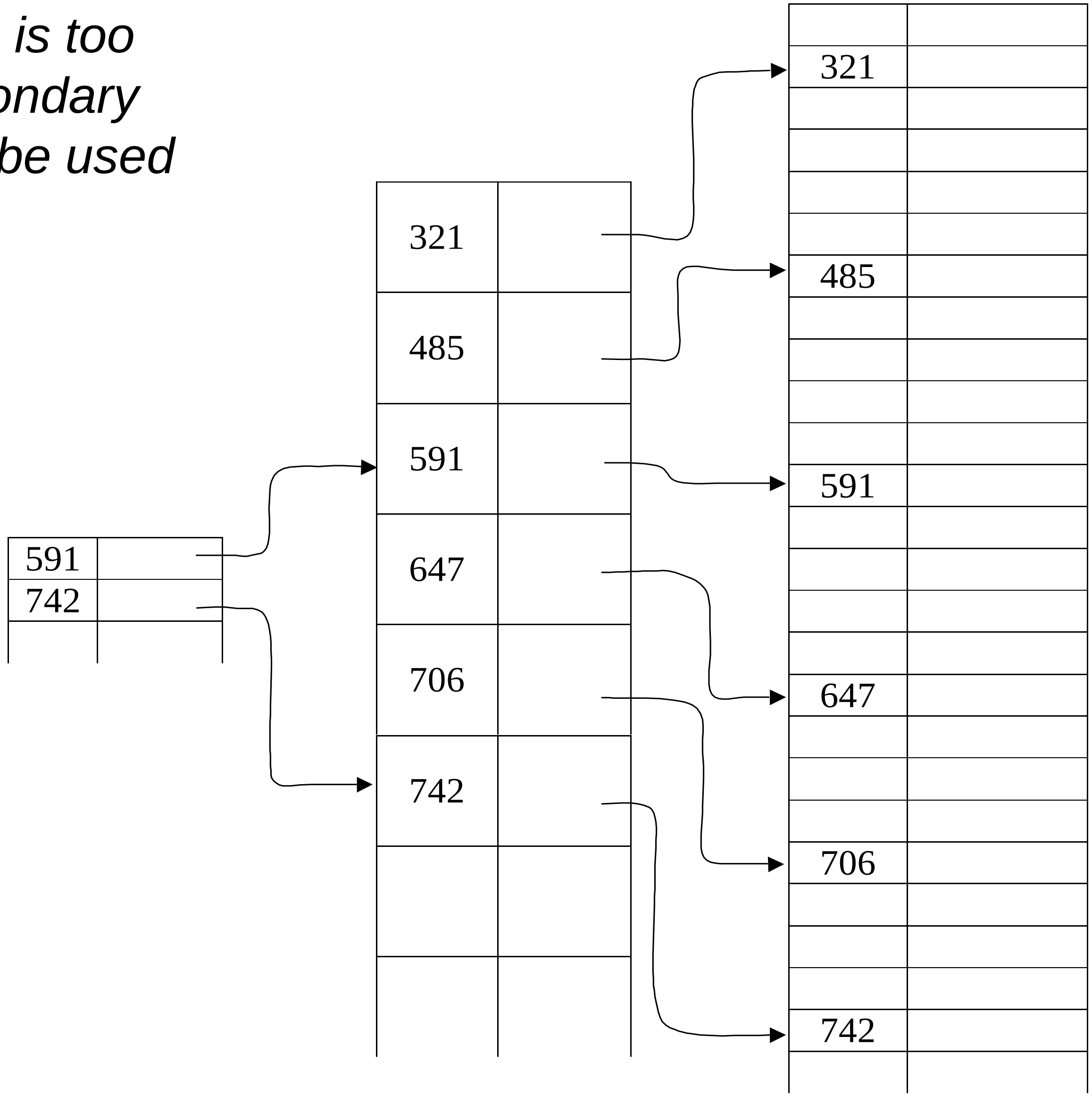
321	
592	
876	

8	
14	
26	
38	
72	
115	
306	
321	
329	
387	
409	
512	
540	
567	
583	
592	
602	
611	
618	
741	
798	
811	
814	
876	

Indexed Sequential Search

```
ISSearch(int a[], int n, int index[], int m, int
x)
{ int i,l,h;
for (i=0; i<m && index[i].key <= x; i++);
l = (i==0) ? 0 : index[i-1].ptr;
h = (i==n) ? n-1 : index[i].ptr;
for(i = l; i<=h && a[i] != x; i++);
return ((i>h) ? -1 : i);
{
```

If the table is too large, secondary index can be used

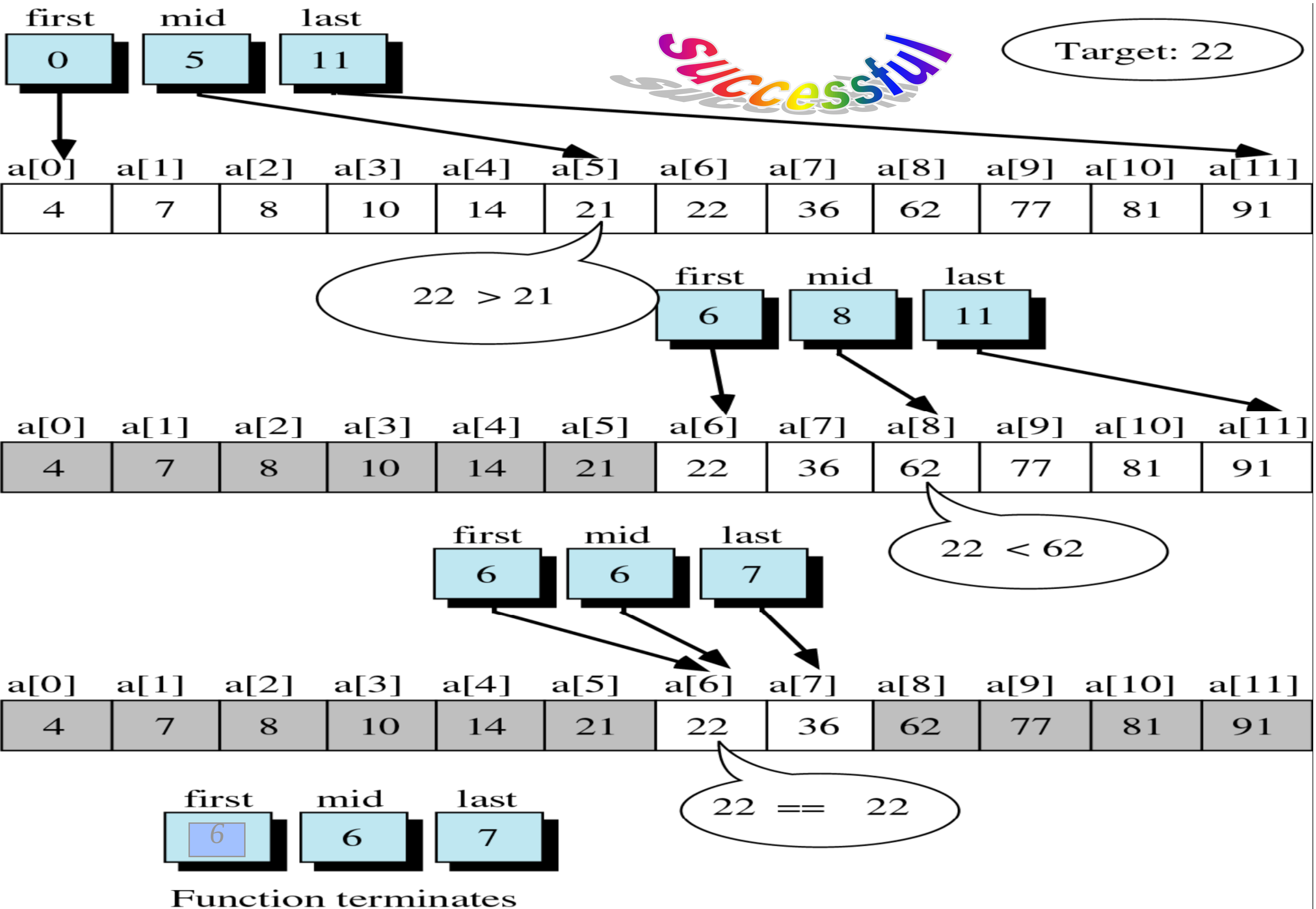


Binary Search

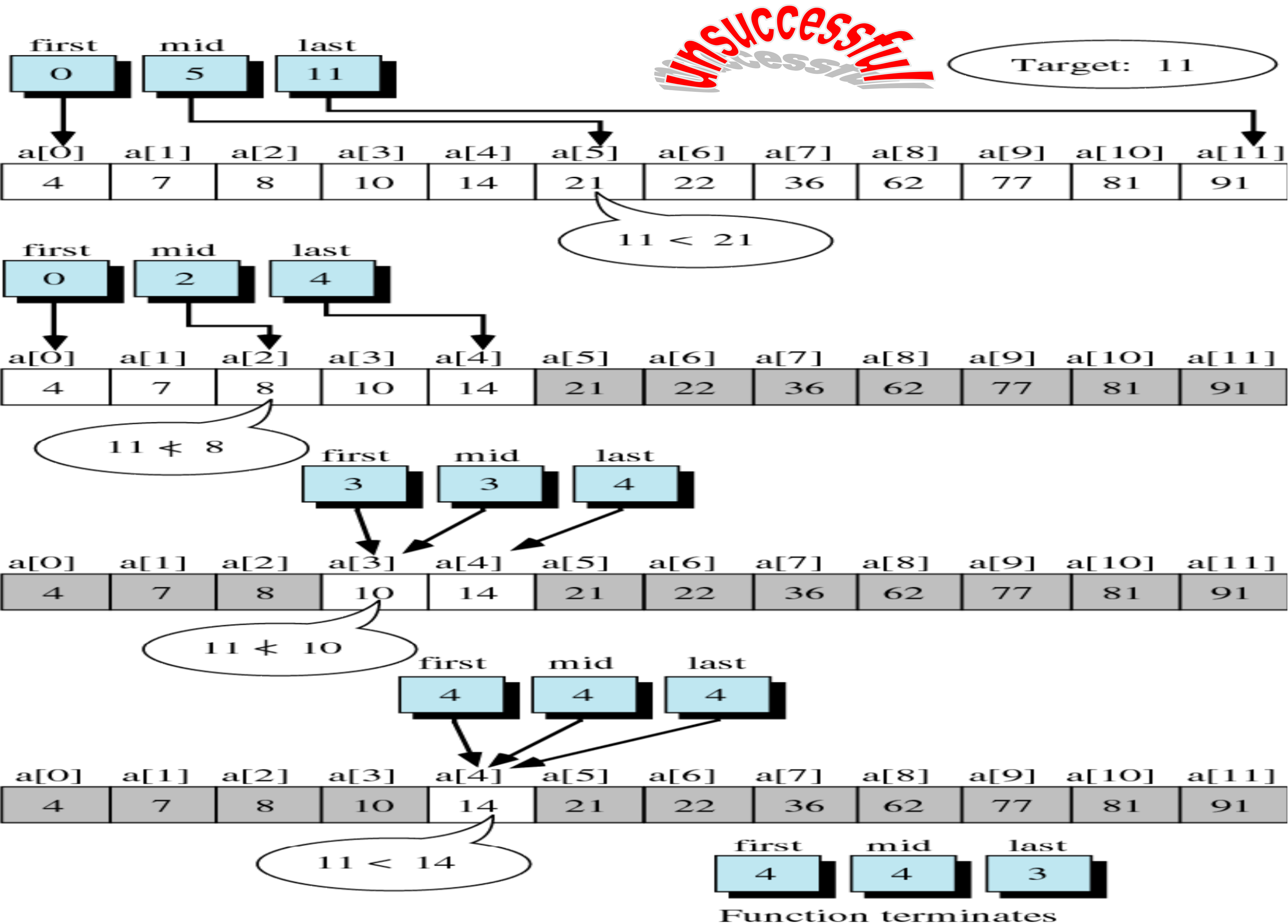
- Search an ordered array of integers for a value and return its index if the value is found. Otherwise, return -1.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	*2*	*3*	*5*	*7*	*10*	*14*	*17*

- Binary search skips over parts of the array if the search value cannot possibly be there.



first
0
mid
5
last
11
successful
Target: 22
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 > 21
first
6
mid
8
last
11
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 < 62
first
6
mid
6
last
7
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
22 == 22
first
6
mid
6
last
7
Function terminates


unsuccessful
Target: 11
first 0 mid 5 last 11
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
11 < 21
first 0 mid 2 last 4
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
11 ≮ 8
first 3 mid 3 last 4
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
11 ≮ 10
first 4 mid 4 last 4
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
4 7 8 10 14 21 22 36 62 77 81 91
11 < 14
first 4 mid 4 last 3
Function terminates

Binary Search

- Binary search is based on the “divide-and-conquer” strategy which works as follows:
  - Start by looking at the middle element of the array
    - 1. If the value it holds is lower than the search element, eliminate the first half of the array from further consideration.
    - 2. If the value it holds is higher than the search element, eliminate the second half of the array from further consideration.
  - Repeat this process until the element is found, or until the entire array has been eliminated.

Binary Search

- Algorithm:

```
Set first and last boundary of array to be searched
Repeat the following:
   Find middle element between first and last boundaries;
   if (middle element contains the search value)
          return middle_element position;
   else if (first >= last )
          return -1;
   else if (value < the value of middle_element)
          set last to middle_element position - 1;
   else
          set first to middle_element position + 1;
```

Iterative Binary Search

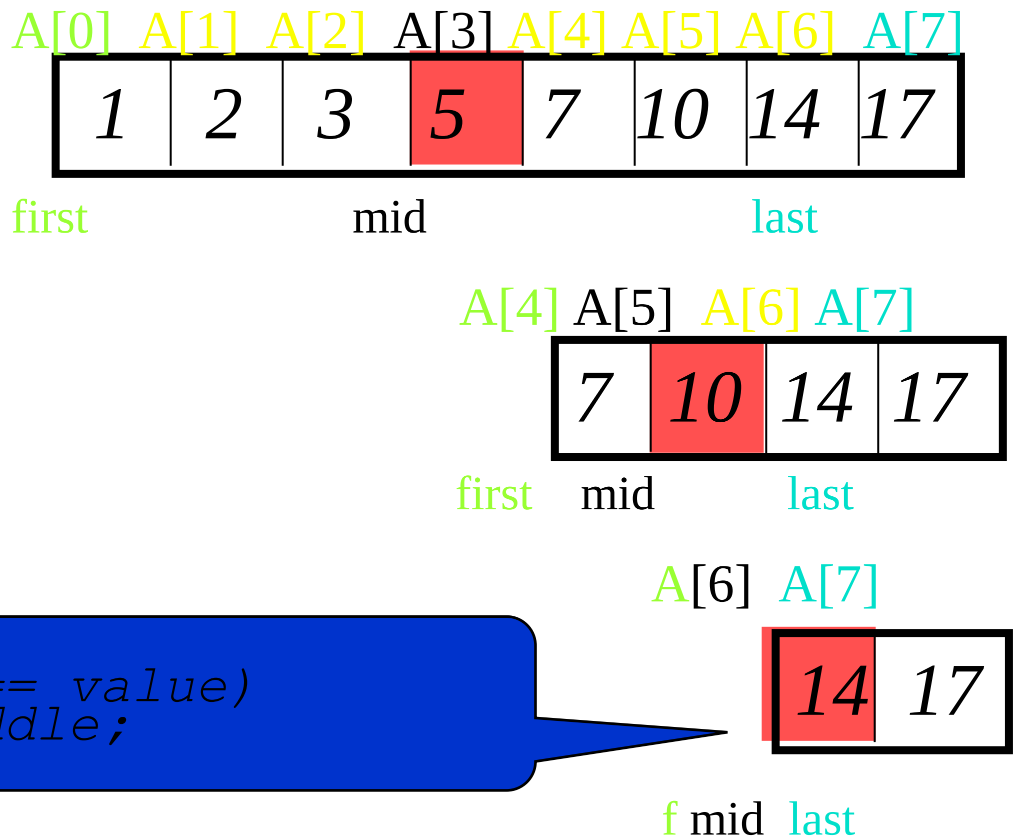
```
int binarySearch(int arr[], int n, int x)
{ int l, r, m;
  l=0; r=n-1;
while (l <= r)   {
    int m = l + (r-l)/2;
      // Check if x is present at mid
    if (arr[m] == x)
        return m;
      // If x greater, ignore left half
    if (arr[m] < x)
        l = m + 1;
      // If x is smaller, ignore right half
    else
         r = m - 1;
  }
    // if we reach here, then element was not present
  return -1;
}
```

Binary Search

```
// Searches an ordered array of integers
int bsearch(int data[], // input: array
           int size,    // input: array size
           int value    // input: value to find
           )            // output: if found,return index
{                       //         otherwise, return -1
      int first, middle, last;
       first = 0;
       last = size - 1;
      while (true) {
         middle = (first + last) / 2;
         if (data[middle] == value)
              return middle;
         else if (first >= last)
              return -1;
         else if (value < data[middle])
            last = middle - 1;
         else
            first = middle + 1;
     }
}
```

Example: binary search

- 14 ?

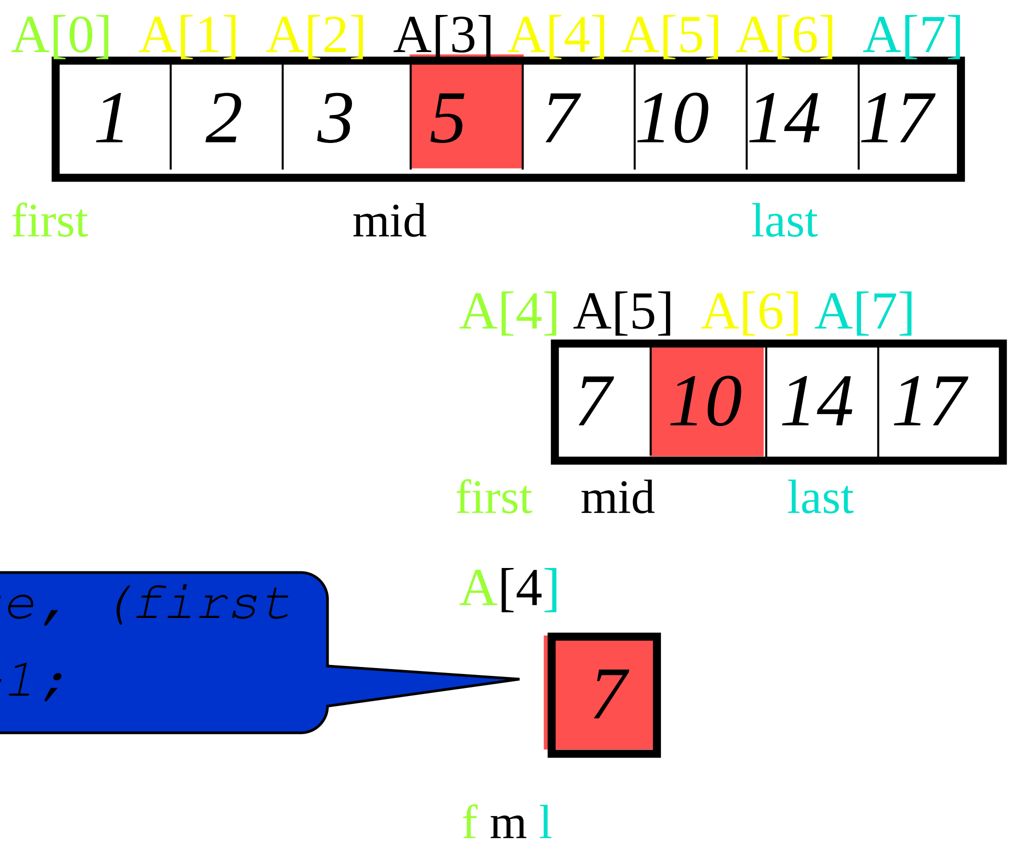



```
In this case,
(data[middle] == value)
      return middle;
```

Example: binary search

unsuccessful

- 8 ?




```
In this case, (first == last)
  return -1;
```

unsuccessful

Example: binary search

- 4 ?

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	*2*	*3*	*5*	*7*	*10*	*14*	*17*
first			mid			last	

A[0]	A[1]	A[2]
1	*2*	*3*
first	mid	last

A[2]
3
f m l

In this case, (first == last)
return -1;

Efficiency of Binary Search

- It can be shown that the number of comparisons required to find an entry is at worst (and on average) **$O(\log_2(n))$**, where *n* is the size of the array.
- Let us say the iteration in Binary Search terminates after k iterations (k is the number of comparisons in worst case)
- At each iteration, the array is divided by half.
- After k^{th} iteration length of array = $n/2^k$
- Also, after k iterations length of the array becomes 1
- Therefore, $n/2^k = 1 \Rightarrow k = \lg n$