

Design & Analysis of Algorithms

Lecture 15

Summary of Brute-Force Approach

- Can be used to solve a problem as the trivial approach. [Exhaustive]
- Exhaustive search is **impractical** for all but very small instances of the problem.

TSP

$$\frac{1}{2}(n-1)!$$

Knapsack

$$2^n$$

Assignment

$$n!$$

**Decrease/Divide
&
Conquer**

Decrease & Conquer Approach

- This technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Once such a relationship is established, it can be exploited either **top-down** or **bottom-up**.

Decrease & Conquer Approach

- **Top-down** approach leads naturally to a **recursive** implementation.
- **bottom-up** approach is usually implemented **iteratively**, starting with a solution to the smallest instance of the problem; it is called sometimes the **incremental approach**.

Decrease & Conquer Approach

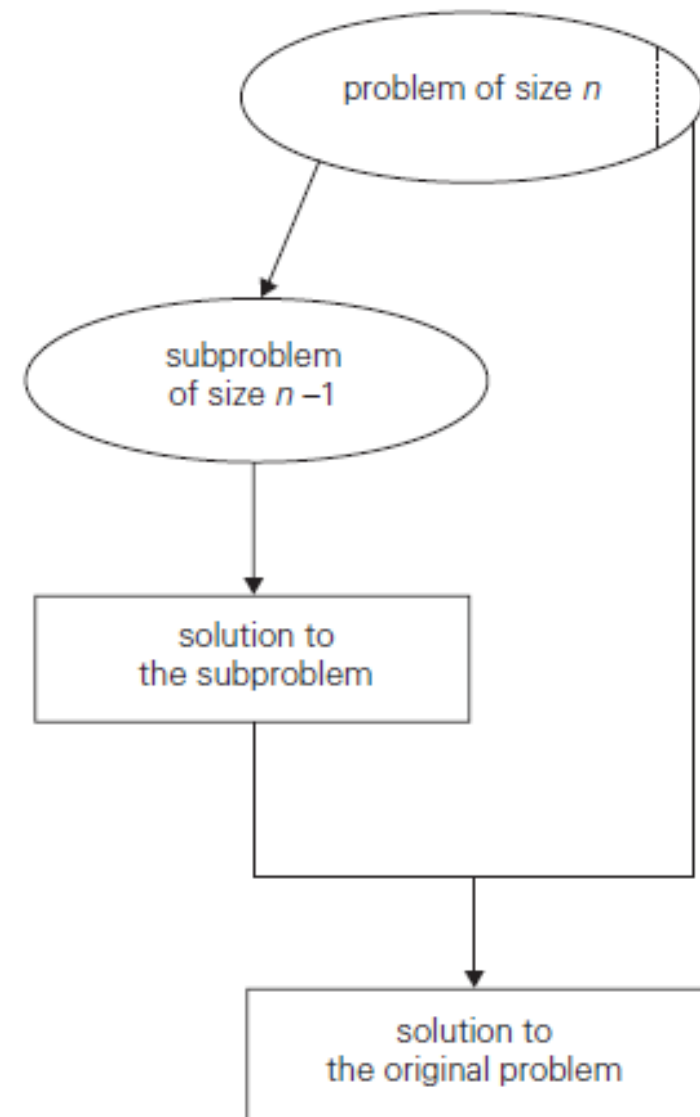
There are **three** major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

Decrease & Conquer Approach

■ Decrease by a constant

- the size of an instance is reduced by the same constant on each iteration of the algorithm.
- Typically, this constant is equal to one, although other constant size reductions do happen occasionally.
- **Example:** Finding a^n , $n!$, etc.

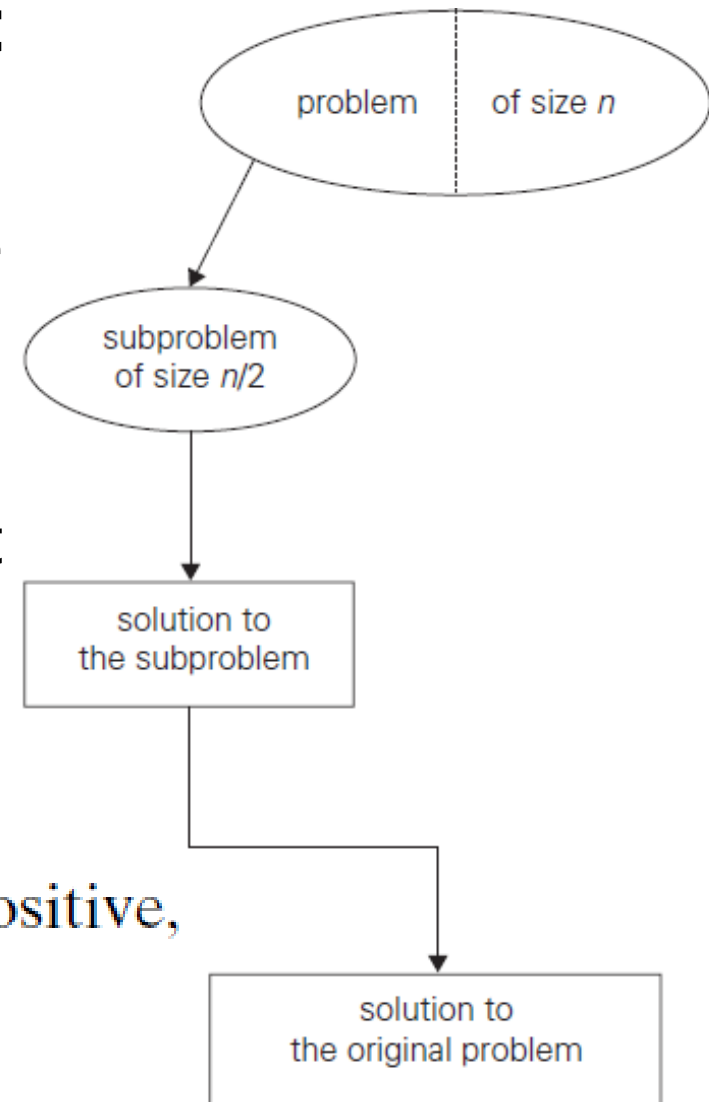


Decrease & Conquer Approach

■ Decrease by a constant factor

- reducing a problem instance by the same constant factor on each iteration of the algorithm.
- In most applications, this constant factor is equal to **two**.
- **Example:** Finding a^n , Binary Search

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$



Decrease & Conquer Approach

- Variable size Decrease
 - The size-reduction pattern varies from one iteration of an algorithm to another.
 - Example: Finding GCD(m, n) using

$$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$$

Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Decrease & Conquer

**Binary Search
Algorithm**

Searching in a Sorted List

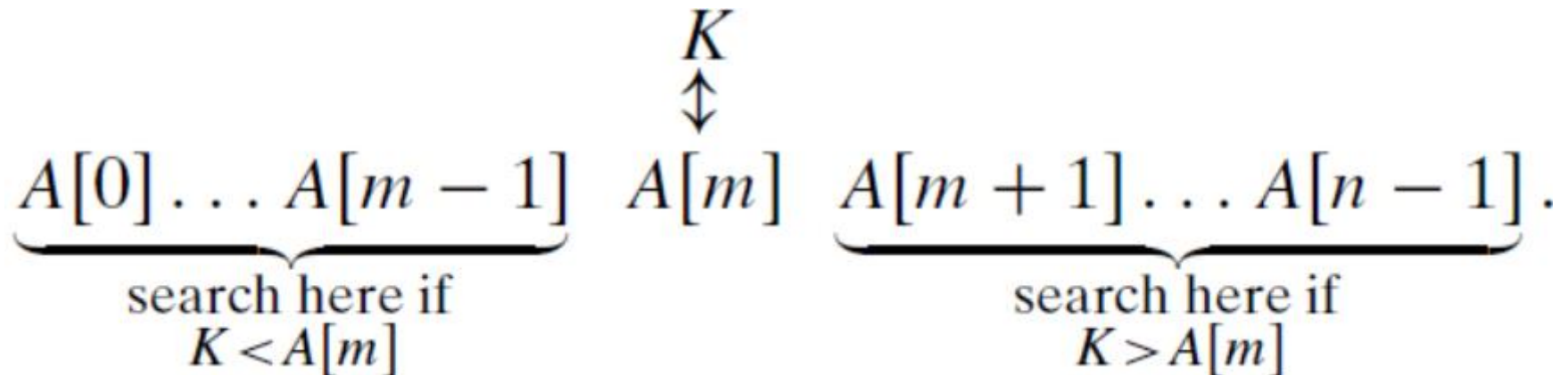
- We are given a list of records.
- Each record has an associated key.
- Give efficient algorithm for searching for a record containing a particular key.
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve an item.

Binary Search Algorithm

- Can only be performed on a sorted list !!!
- Uses ***decrease by a constant factor and conquer*** technique to search the list.

Binary Search Algorithm

- It works by comparing a search key K with the array's middle element $A[m]$.
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:



Binary Search Algorithm

ALGORITHM *BinarySearch*($A[0..n - 1]$, K)

//Implements nonrecursive binary search

//Input: An array $A[0..n - 1]$ sorted in ascending order and

// a search key K

//Output: An index of the array's element that is equal to K

// or -1 if there is no such element

$l \leftarrow 0$; $r \leftarrow n - 1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$ $r \leftarrow m - 1$

else $l \leftarrow m + 1$

return -1

Binary Search Algorithm

Example

- Apply binary search to search for $K = 70$

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

- The iterations of the algorithm are given as:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l, m					r

Binary Search Algorithm

```
1  Algorithm BinSearch( $a, n, x$ )
2  // Given an array  $a[1 : n]$  of elements in nondecreasing
3  // order,  $n \geq 0$ , 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 ( $low \leq high$ ) do
8      {
9           $mid := \lfloor (low + high) / 2 \rfloor;$ 
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 }
```

Book: Horowitz & Sahni

Binary Search Algorithm

Example

−15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

<i>a:</i>	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

$x = 151$	<i>low</i>	<i>high</i>	<i>mid</i>
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

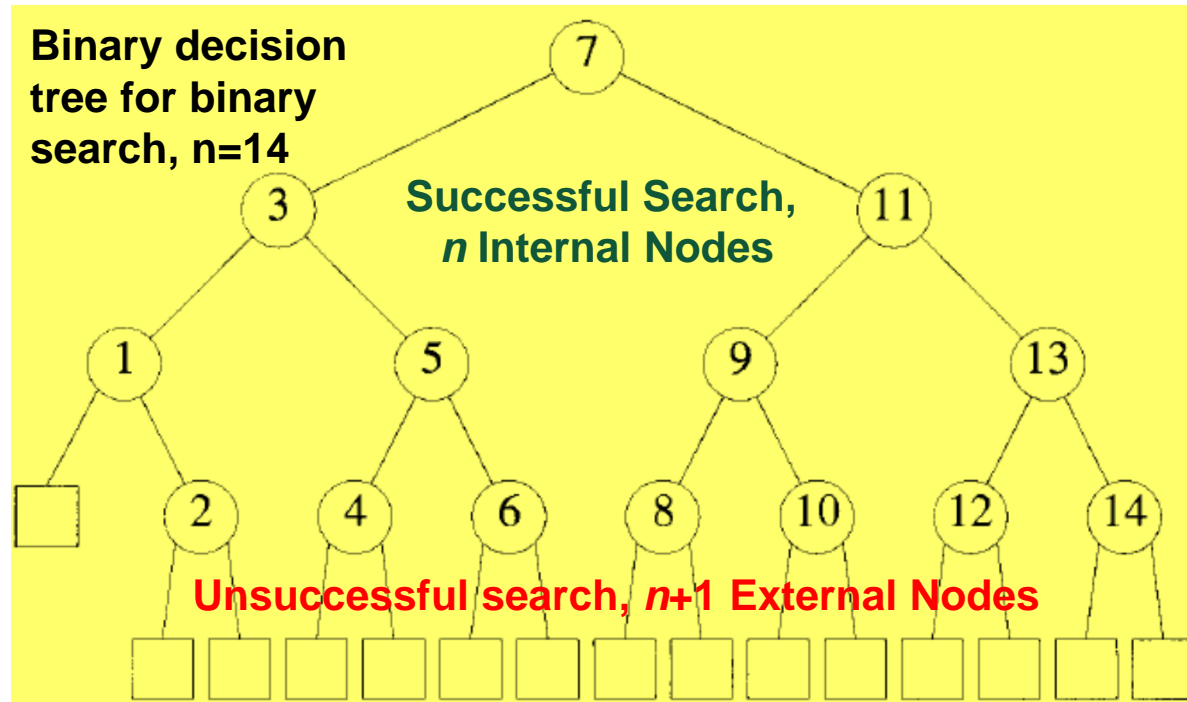
$x = -14$	<i>low</i>	<i>high</i>	<i>mid</i>
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	<i>low</i>	<i>high</i>	<i>mid</i>
	1	14	7
	1	6	3
	4	6	5
			found

Binary Search Algorithm

Example

−15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151



- For a **successful** search, the BinSearch algorithm will end at one of the **circular** nodes (**Internal Nodes**)
- For an **unsuccessful** search, the BinSearch algorithm will end at one of the **square** nodes (**External Nodes**)

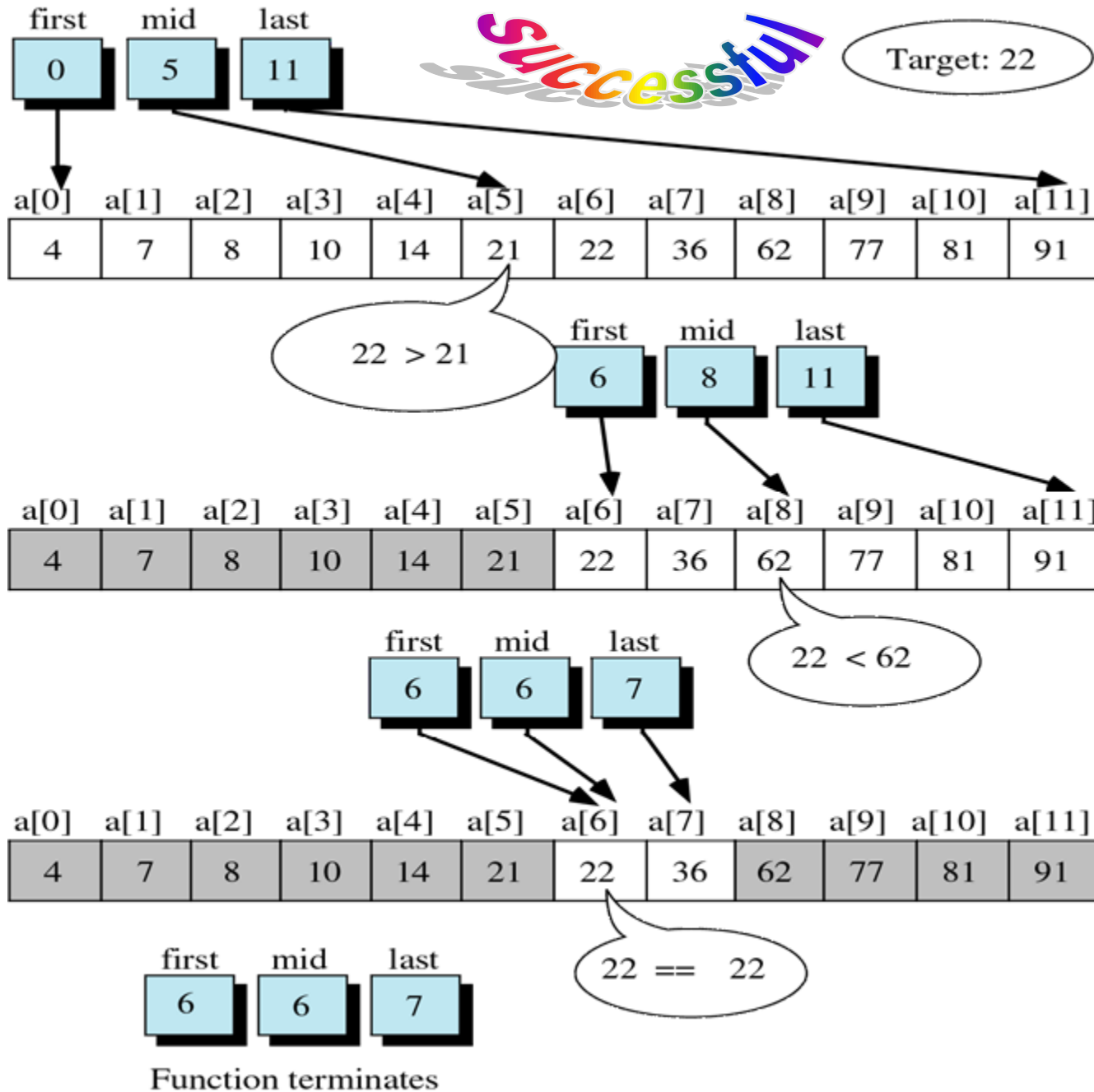
Binary Search Algorithm

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

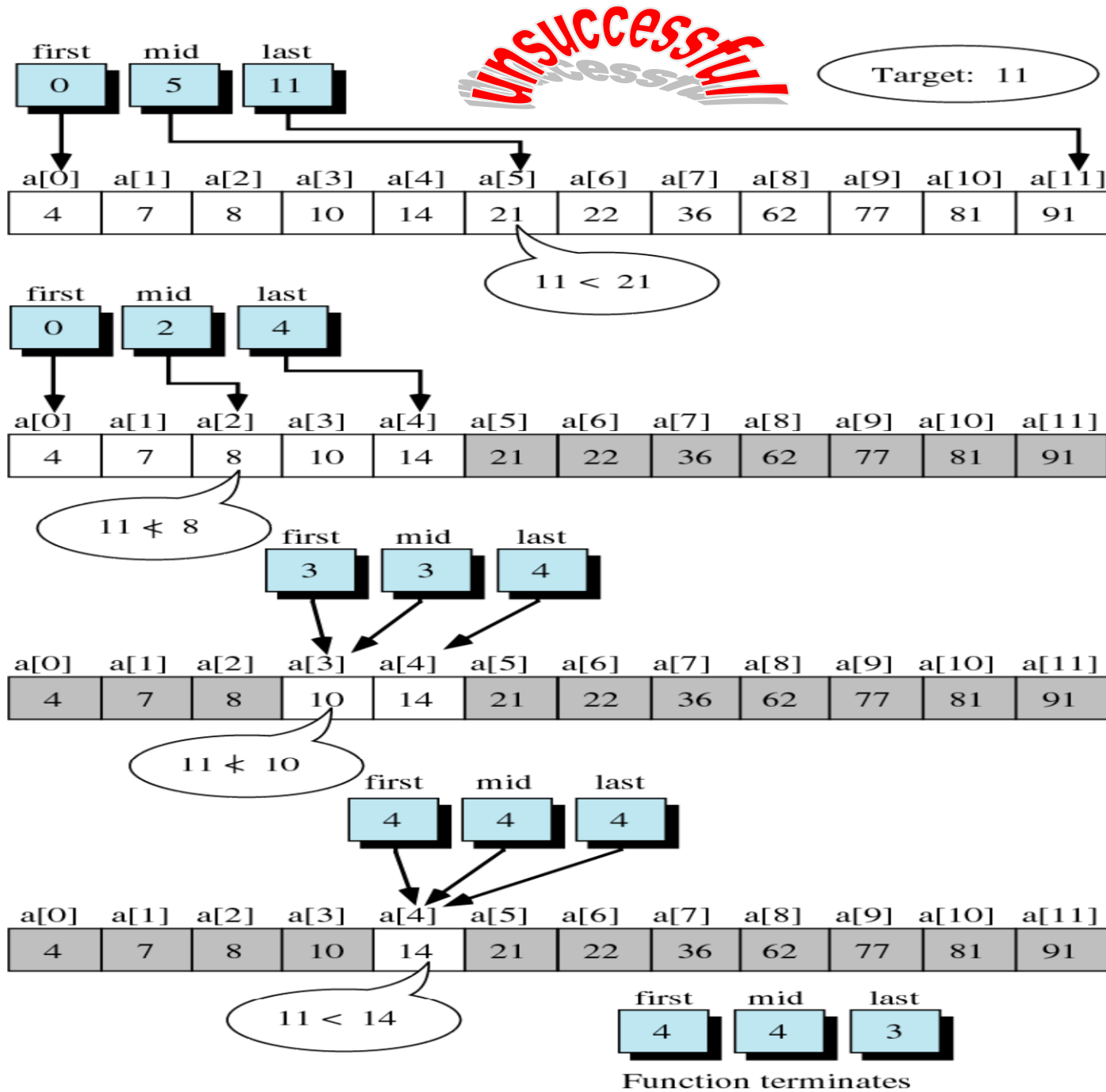
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
1	2	3	5	7	10	14	17

- Binary search skips over parts of the array if the search value cannot possibly be there.
- Try if K=1,5,7, and 17. (How many comparisons?)

Example 1



Example 2



Example: binary search



■ 14 ?

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[4]	A[5]	A[6]	A[7]
7	10	14	17
first	mid	last	

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

A[6]		A[7]
14	17	
first	mid	last

Example: binary search

Unsuccessful

■ 8 ?

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[4]	A[5]	A[6]	A[7]
7	10	14	17
first	mid	last	

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

A[4]
7
f m l

Example: binary search

unsuccessful

■ 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;*

Binary Search Example 2

- Determine whether **75** is in the list

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 1: Array list with twelve (12) elements

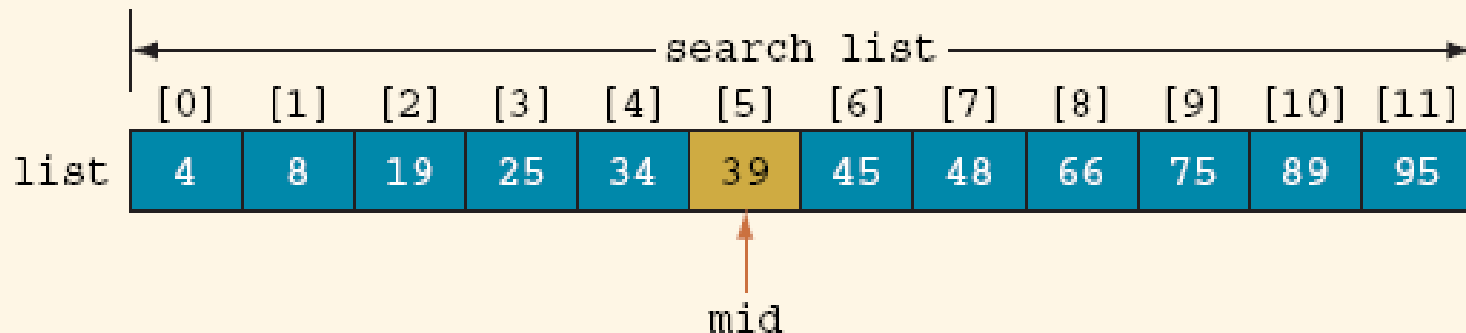
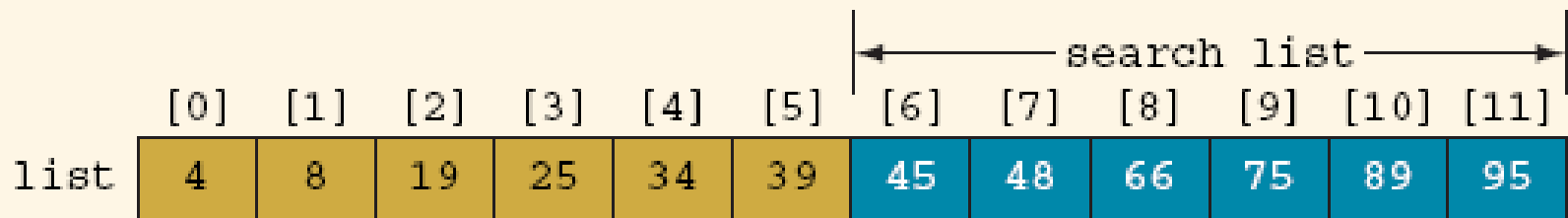


Figure 2: Search list, list[0] ... list[11]

Binary Search Example 2



The diagram shows a horizontal array of 12 elements, indexed from [0] to [11]. The first six elements (indices [0] to [5]) are in yellow boxes with values 4, 8, 19, 25, 34, and 39. The last six elements (indices [6] to [11]) are in blue boxes with values 45, 48, 66, 75, 89, and 95. Above the array, a double-headed arrow labeled 'search list' spans from index [6] to index [11]. The label 'list' is positioned to the left of the array.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 3: Search list, list[6] ... list[11]

Binary Search Example 2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 4: Sorted list for binary search

key = 89

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is true)

key = 34

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	4	4	4	34	1 (found is true)

Binary Search Example 2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 5: Sorted list for binary search

key = 22

Iteration	first	last	mid	list[mid]	Number of key comparisons
1	0	11	5	39	2
2	0	4	2	19	2
3	3	4	3	25	2
4	3	2	the loop stops (because first > last) unsuccessful search		

Binary Search Example 2

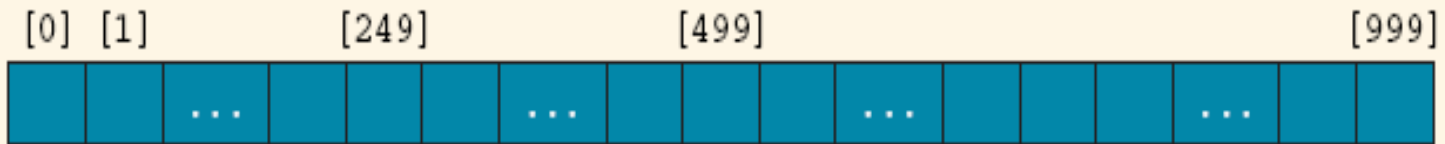
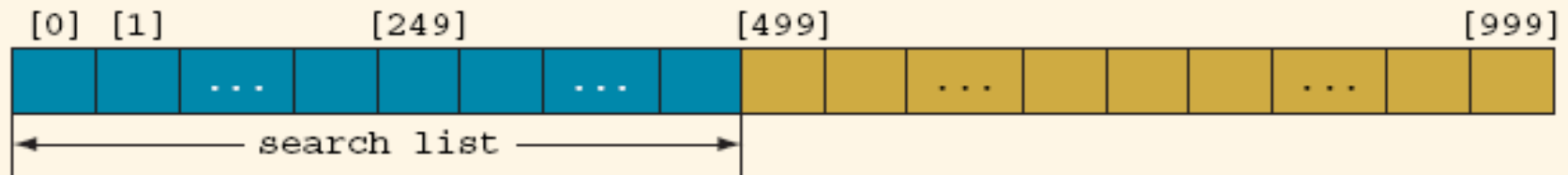


Figure 6: A Sorted list for binary search

key \neq List[499]

key < List[499]



first	0
last	498
mid	249

Figure 7: Search list after first iteration

Binary Search Example 2

key > List[249]

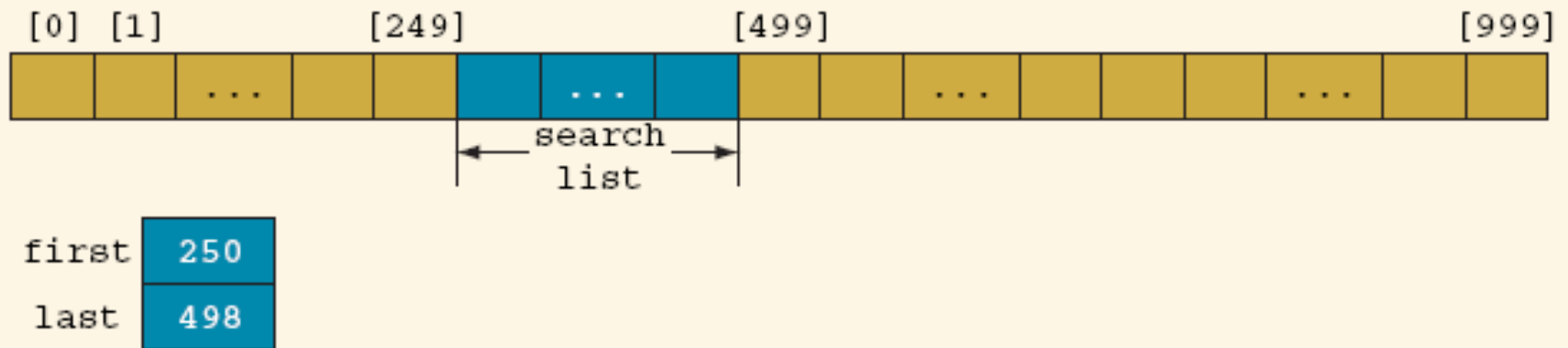


Figure 8: Search list after second iteration

Binary Search Algorithm

- How many such **comparisons** does the algorithm make on an array of **n** elements?
- The answer obviously depends not only on **n** but also on the specifics of a particular instance of the problem.
- Let us find the number of key comparisons in the **worst** case **$C_{worst}(n)$** .
- The **worst-case** inputs include all arrays that do not contain a given search key, as well as some successful searches.

Binary Search Algorithm

- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1.$$

- It can be solved with a different initial condition and n is of the form 2^k . For the initial condition $C_{worst}(1) = 1$, we obtain

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n + 1) \rceil$$

Binary Search Algorithm

- What can we say about the average-case efficiency of binary search?
- The average number of key comparisons made by binary search is only **slightly smaller** than that in the **worst case**: $C_{avg}(n) \approx \log_2 n$.
- More accurate formulas for the average number of comparisons in a **successful** and an **unsuccessful** search are respectively:

$$C_{avg}^{yes}(n) \approx \log_2 n - 1 \quad \text{and} \quad C_{avg}^{no}(n) \approx \log_2(n + 1)$$

Binary Search Algorithm

Worst-case Time Complexity Analysis Proof

Theorem 3.2 If n is in the range $[2^{k-1}, 2^k)$, then BinSearch makes at most k element comparisons for a successful search and either $k-1$ or k comparisons for an unsuccessful search. (In other words the time for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$).

Proof: Consider the binary decision tree describing the action of BinSearch on n elements. All successful searches end at a circular node whereas all unsuccessful searches end at a square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels $1, 2, \dots, k$ whereas all square nodes are at levels k and $k+1$ (note that the root is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i whereas the number of element comparisons needed to terminate at a square node at level i is only $i-1$. The theorem follows. \square

Book: Horowitz & Sahni

Binary Search Algorithm

Average-case Time Complexity Analysis

Theorem 3.2 states the worst-case time for binary search. To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparisons in the algorithm. The *distance* of a node from the root is one less than its level. The *internal path length* I is the sum of the distances of all internal nodes from the root. Analogously, the *external path length* E is the sum of the distances of all external nodes from the root. It is easy to show by induction that for any binary tree with n internal nodes, E and I are related by the formula

$$E = I + 2n$$

It turns out that there is a simple relationship between E , I , and the average number of comparisons in binary search. Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ the average number of comparisons in an unsuccessful search. The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root. Hence,

$$A_s(n) = 1 + I/n$$

Book: Horowitz & Sahni

Binary Search Algorithm

Average-case Time Complexity Analysis

The number of comparisons on any path from the root to an external node is equal to the distance between the root and the external node. Since every binary tree with n internal nodes has $n + 1$ external nodes, it follows that

$$A_u(n) = E / (n + 1)$$

Using these three formulas for E , $A_s(n)$, and $A_u(n)$, we find that

$$A_s(n) = (1 + 1/n)A_u(n) - 1$$

From this formula we see that $A_s(n)$ and $A_u(n)$ are directly related. The minimum value of $A_s(n)$ (and hence $A_u(n)$) is achieved by an algorithm whose binary decision tree has minimum external and internal path length. This minimum is achieved by the binary tree all of whose external nodes are on adjacent levels, and this is precisely the tree that is produced by binary search. From Theorem 3.2 it follows that E is proportional to $n \log n$. Using this in the preceding formulas, we conclude that $A_s(n)$ and $A_u(n)$ are both proportional to $\log n$. Thus we conclude that the average- and worst-case numbers of comparisons for binary search are the same to within a constant factor.

Binary Search Algorithm

Best-case Time Complexity Analysis

The best-case analysis is easy. For a successful search only one element comparison is needed. For an unsuccessful search, Theorem 3.2 states that $\lfloor \log n \rfloor$ element comparisons are needed in the best case.

In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:

successful searches

$\Theta(1)$, $\Theta(\log n)$, $\Theta(\log n)$
best, average, worst

unsuccessful searches

$\Theta(\log n)$
best, average, worst

Recursive Binary Search Algorithm

ALGORITHM *BinarySearchRec*(A, l, r, K)

//Implements **recursive** binary search

//Input: An array $A[0..n - 1]$ sorted in ascending order, a

// search key K , start of index l , and end of index r

//Output: An index of the array's element that is equal to K

// or -1 if there is no such element (unsuccessful)

$m = (l + r) / 2$

if ($l > r$) **return** -1

if ($A[m] == K$) **return** m

if ($A[m] < K$)

return *BinarySearchRec*($A, m+1, r, K$)

else

return *BinarySearchRec*($A, l, m-1, K$)

Recursive Binary Search Algorithm

```
1  Algorithm BinSrch( $a, i, l, x$ )
2  // Given an array  $a[i : l]$  of elements in nondecreasing
3  // order,  $1 \leq i \leq l$ , 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          $mid := \lfloor (i + l) / 2 \rfloor$ ;
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 }
```

Book: Horowitz & Sahni

Recursive Binary Search: Analysis

- Worst case complexity?
- What is the maximum **depth of recursive calls** in binary search as function of **n** ?
 - Each level in the recursion, we split the array in half (divide by two).
 - Therefore maximum recursion depth is $\text{floor}(\log_2 n)$ and worst case = $O(\log_2 n)$.
 - Average case is also = $O(\log_2 n)$.

Can we do better than $O(\log_2 n)$?

- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$
- Can we do better than this?

YES.

How?

Use a hash table!

References

- **Chapter 4:** Anany Levitin, “Introduction to the Design and Analysis of Algorithms”, Pearson Education, Third Edition, 2017.
- **Chapter 3:** E Horowitz, S Sahni, S Rajasekaran, “Computer Algorithms”, Computer Science Press, Third Edition, 2008.

Homework

- **Fake Coin Problem:** Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin.
- **Ternary Search** (By making three partitions)