

Divide-and-Conquer: Searching in an Array

Neil Rhodes

Department of Computer Science and Engineering
University of California, San Diego

Data Structures and Algorithms
Algorithmic Toolbox

Outline

- 1 Main Idea of Divide-and-Conquer
- 2 Linear Search
- 3 Binary Search







a problem to be solved

Divide: Break into non-overlapping subproblems of the same type



Divide: Break into non-overlapping subproblems of the same type



Divide: Break into non-overlapping subproblems of the same type



Divide: Break into non-overlapping subproblems of the same type

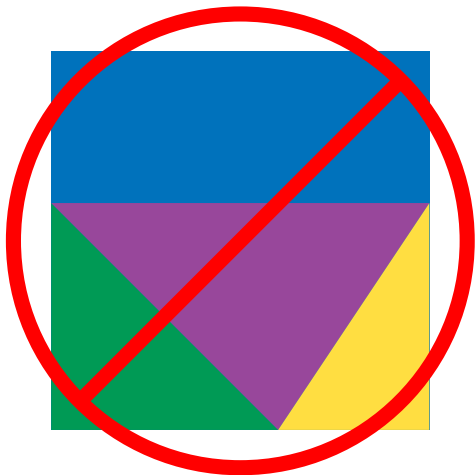












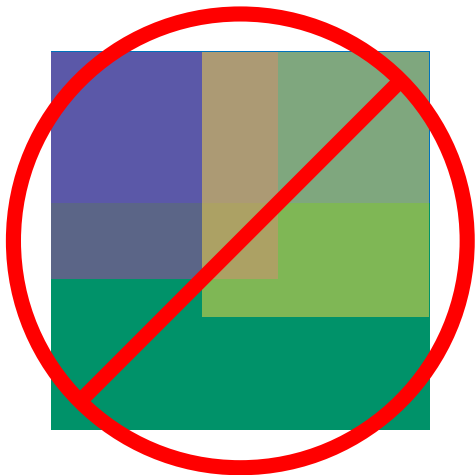
not the
same type









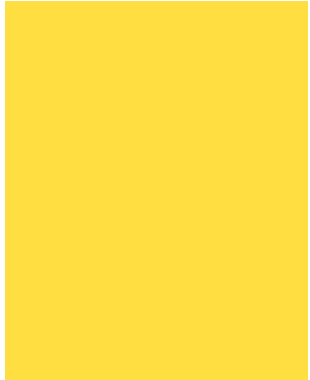


overlapping

Divide: break apart



Divide: break apart



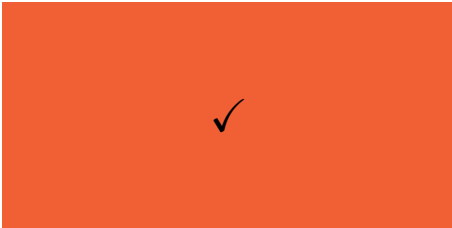
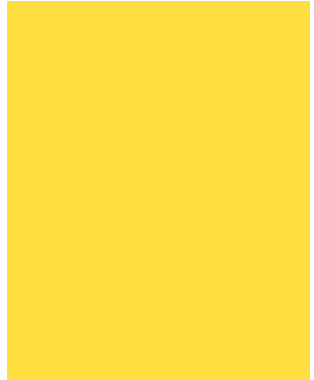
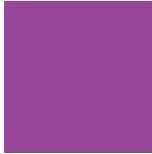
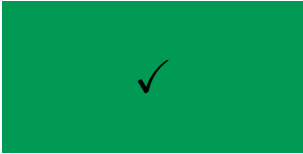
Conquer: solve subproblems



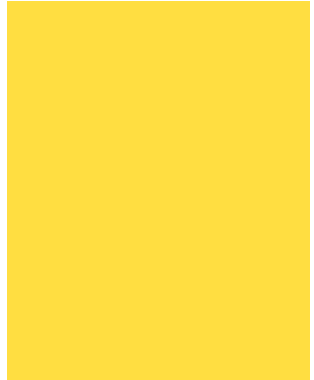
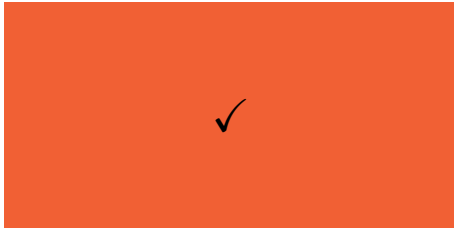
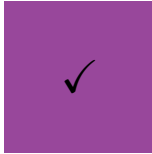
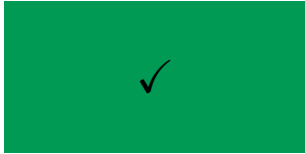
Conquer: solve subproblems



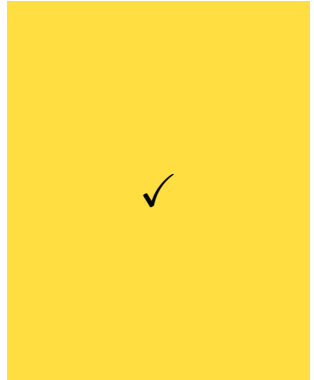
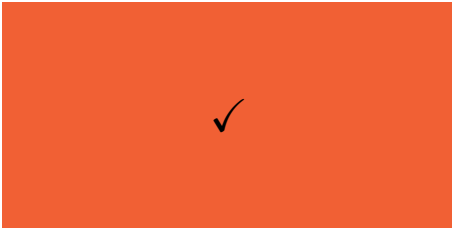
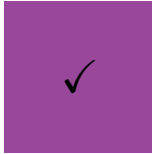
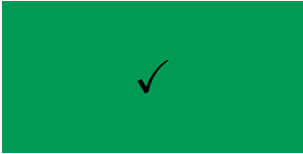
Conquer: solve subproblems



Conquer: solve subproblems

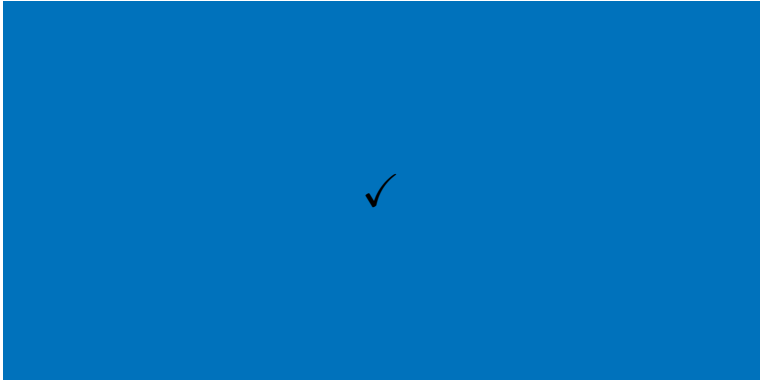


Conquer: solve subproblems



Conquer: combine





- 1 Break into non-overlapping subproblems of the same type
- 2 Solve subproblems
- 3 Combine results

Outline

- ① Main Idea of Divide-and-Conquer
- ② Linear Search
- ③ Binary Search

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Linear Search in Array

Ann	Pat	...	Joe	Bob
-----	-----	-----	-----	-----

Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
car	voiture	auto	Auto	auto
table	table	tavola	Tabelle	mesa

Searching in an array

Input: An array A with n elements.
A key k .

Output: An index, i , where $A[i] = k$.
If there is no such i , then
NOT_FOUND.

Recursive Solution

LinearSearch(A , *low*, *high*, *key*)

Recursive Solution

LinearSearch(*A*, *low*, *high*, *key*)

```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low
```

Recursive Solution

LinearSearch(*A, low, high, key*)

```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low  
return LinearSearch(A, low + 1, high, key)
```

Recursive Solution

LinearSearch(*A, low, high, key*)

```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low  
return LinearSearch(A, low + 1, high, key)
```

Definition

A **recurrence relation** is an equation recursively defining a sequence of values.

Definition

A **recurrence relation** is an equation recursively defining a sequence of values.

Fibonacci recurrence relation

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Definition

A **recurrence relation** is an equation recursively defining a sequence of values.

Fibonacci recurrence relation

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, ...

LinearSearch(*A, low, high, key*)

```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low  
return LinearSearch(A, low + 1, high, key)
```


LinearSearch(*A, low, high, key*)

```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low  
return LinearSearch(A, low + 1, high, key)
```

Recurrence defining worst-case time:

$$T(n) = T(n - 1) + c$$

LinearSearch(*A, low, high, key*)

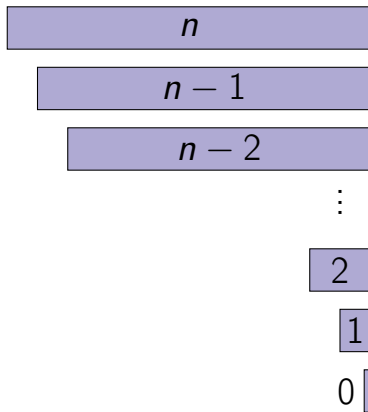
```
if high < low:  
    return NOT_FOUND  
if A[low] = key:  
    return low  
return LinearSearch(A, low + 1, high, key)
```

Recurrence defining worst-case time:

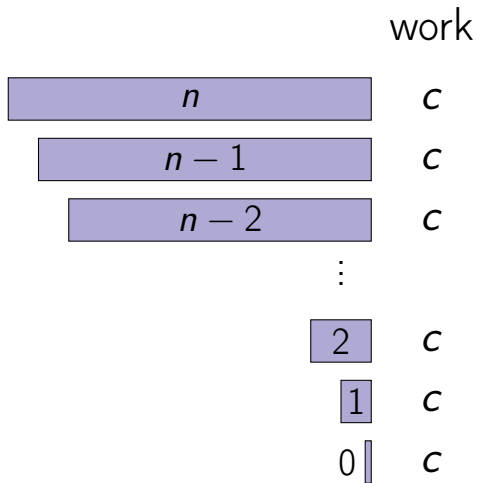
$$T(n) = T(n - 1) + c$$

$$T(0) = c$$

Runtime of Linear Search



Runtime of Linear Search



Runtime of Linear Search

	work
n	c
$n - 1$	c
$n - 2$	c
\vdots	
2	c
1	c
0	c

Total: $\sum_{i=0}^n c = \Theta(n)$

Iterative Version

`LinearSearchIt(A, low, high, key)`

for *i* from *low* to *high*:

 if $A[i] = \textit{key}$:

 return *i*

return NOT_FOUND

Summary

- Create a recursive solution

Summary

- Create a recursive solution
- Define a corresponding recurrence relation, T

Summary

- Create a recursive solution
- Define a corresponding recurrence relation, T
- Determine $T(n)$: worst-case runtime

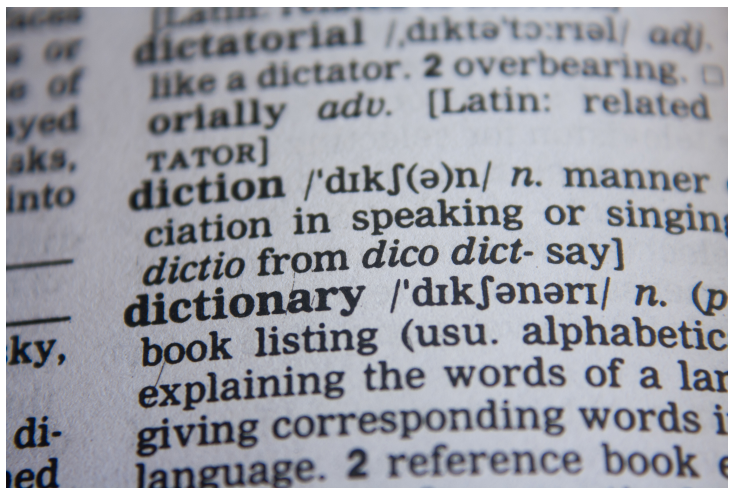
Summary

- Create a recursive solution
- Define a corresponding recurrence relation, T
- Determine $T(n)$: worst-case runtime
- Optionally, create iterative solution

Outline

- ① Main Idea of Divide-and-Conquer
- ② Linear Search
- ③ Binary Search

Searching Sorted Data



Searching in a sorted array

Input: A sorted array $A[low \dots high]$
($\forall low \leq i < high: A[i] \leq A[i + 1]$).
A key k .

Output: An index, i , ($low \leq i \leq high$) where
 $A[i] = k$.
Otherwise, the greatest index i ,
where $A[i] < k$.
Otherwise ($k < A[low]$), the result is
 $low - 1$.

Searching in a Sorted Array


Example

3	5	8	20	20	50	60
1	2	3	4	5	6	7

Searching in a Sorted Array

Example

search(2) \rightarrow 0




3	5	8	20	20	50	60
1	2	3	4	5	6	7

Searching in a Sorted Array

Example

search(2) \rightarrow 0

search(3) \rightarrow 1



3	5	8	20	20	50	60
1	2	3	4	5	6	7


Searching in a Sorted Array

Example

search(2) \rightarrow 0

search(3) \rightarrow 1

search(4) \rightarrow 1



3	5	8	20	20	50	60
1	2	3	4	5	6	7


Searching in a Sorted Array

Example

search(2) \rightarrow 0 *search*(20) \rightarrow 4

search(3) \rightarrow 1

search(4) \rightarrow 1




3	5	8	20	20	50	60
1	2	3	4	5	6	7

Searching in a Sorted Array

Example

$search(2) \rightarrow 0$ $search(20) \rightarrow 4$
 $search(3) \rightarrow 1$ $search(20) \rightarrow 5$
 $search(4) \rightarrow 1$



3	5	8	20	20	50	60
1	2	3	4	5	6	7


Searching in a Sorted Array

Example

search(2) \rightarrow 0 *search*(20) \rightarrow 4

search(3) \rightarrow 1 *search*(20) \rightarrow 5

search(4) \rightarrow 1 *search*(60) \rightarrow 7



3	5	8	20	20	50	60
1	2	3	4	5	6	7

Searching in a Sorted Array


Example

search(2) \rightarrow 0 *search*(20) \rightarrow 4

search(3) \rightarrow 1 *search*(20) \rightarrow 5

search(4) \rightarrow 1 *search*(60) \rightarrow 7

search(90) \rightarrow 7



3	5	8	20	20	50	60
1	2	3	4	5	6	7

BinarySearch(A , low , $high$, key)

BinarySearch(*A*, *low*, *high*, *key*)

```
if high < low:  
    return low - 1
```

BinarySearch(*A*, *low*, *high*, *key*)

if *high* < *low*:

 return *low* - 1

mid $\leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$

BinarySearch(*A*, *low*, *high*, *key*)

```
if high < low:  
    return low - 1  
mid  $\leftarrow \left\lfloor \text{low} + \frac{\text{high} - \text{low}}{2} \right\rfloor$   
if key = A[mid]:  
    return mid
```

BinarySearch($A, low, high, key$)

```
if  $high < low$ :  
    return  $low - 1$   
 $mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$   
if  $key = A[mid]$ :  
    return  $mid$   
else if  $key < A[mid]$ :  
    return BinarySearch( $A, low, mid - 1, key$ )
```

BinarySearch(*A*, *low*, *high*, *key*)

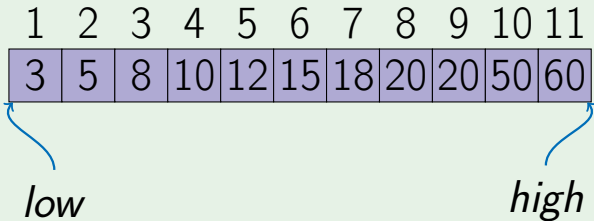
```
if high < low:  
    return low - 1  
mid  $\leftarrow \left\lfloor \text{low} + \frac{\text{high} - \text{low}}{2} \right\rfloor$   
if key = A[mid]:  
    return mid  
else if key < A[mid]:  
    return BinarySearch(A, low, mid - 1, key)  
else:  
    return BinarySearch(A, mid + 1, high, key)
```

Example: Searching for the key 50

1	2	3	4	5	6	7	8	9	10	11
3	5	8	10	12	15	18	20	20	50	60

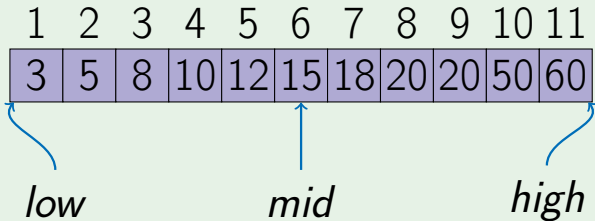
Example: Searching for the key 50

BinarySearch(A , 1, 11, 50)



Example: Searching for the key 50

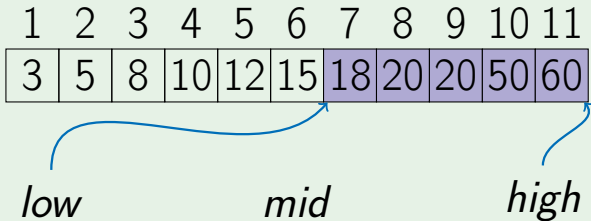
BinarySearch(A , 1, 11, 50)



Example: Searching for the key 50

BinarySearch(A, 1, 11, 50)

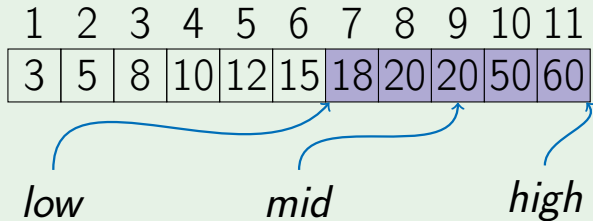
BinarySearch(A, 7, 11, 50)



Example: Searching for the key 50

BinarySearch(A, 1, 11, 50)

BinarySearch(A, 7, 11, 50)

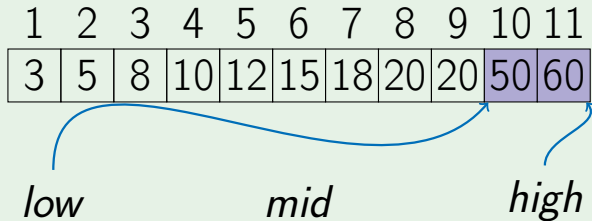


Example: Searching for the key 50

BinarySearch(*A*, 1, 11, 50)

BinarySearch(*A*, 7, 11, 50)

BinarySearch(*A*, 10, 11, 50)

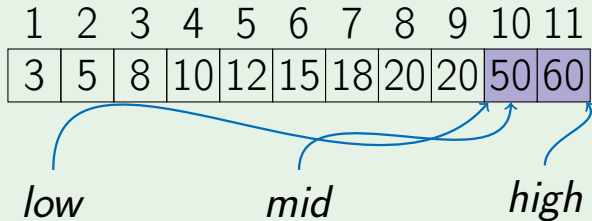


Example: Searching for the key 50

BinarySearch(A, 1, 11, 50)

BinarySearch(A, 7, 11, 50)

BinarySearch(A, 10, 11, 50)



Example: Searching for the key 50

BinarySearch(*A*, 1, 11, 50)

BinarySearch(*A*, 7, 11, 50)

BinarySearch(*A*, 10, 11, 50) → 10

1	2	3	4	5	6	7	8	9	10	11
3	5	8	10	12	15	18	20	20	50	60

Summary

Summary

- Break problem into non-overlapping subproblems of the same type.

Summary

- Break problem into non-overlapping subproblems of the same type.
- Recursively solve those subproblems.

Summary

- Break problem into non-overlapping subproblems of the same type.
- Recursively solve those subproblems.
- Combine results of subproblems.

BinarySearch(*A*, *low*, *high*, *key*)

```
if high < low:  
    return low - 1  
mid  $\leftarrow \left\lfloor \text{low} + \frac{\text{high} - \text{low}}{2} \right\rfloor$   
if key = A[mid]:  
    return mid  
else if key < A[mid]:  
    return BinarySearch(A, low, mid - 1, key)  
else:  
    return BinarySearch(A, mid + 1, high, key)
```


Binary Search Recurrence Relation

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

Binary Search Recurrence Relation

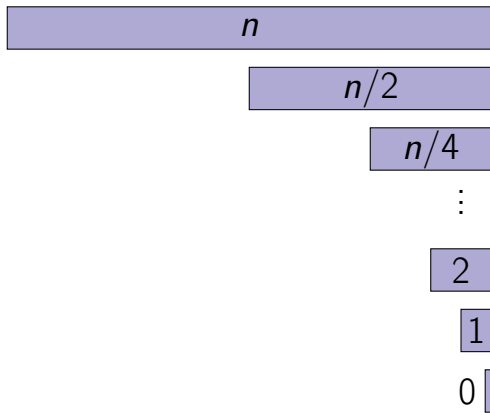
$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

Binary Search Recurrence Relation

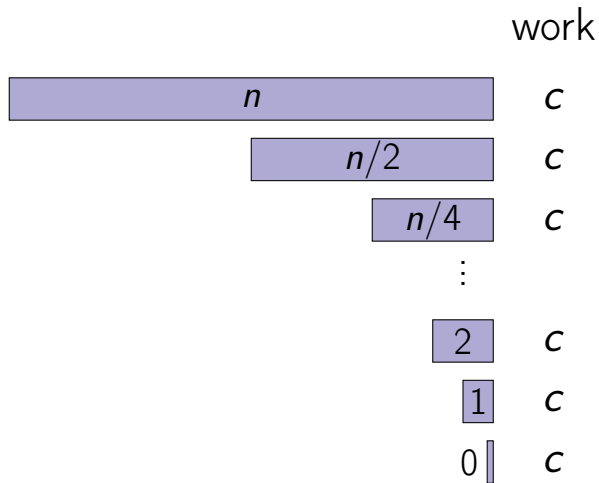
$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c$$

$$T(0) = c$$

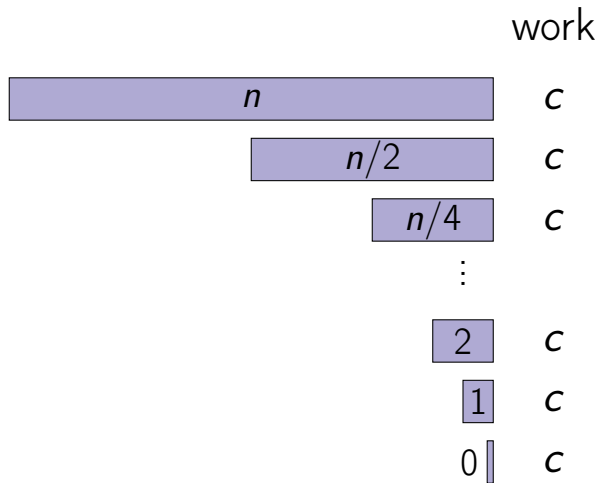
Runtime of Binary Search



Runtime of Binary Search



Runtime of Binary Search



Total: $\sum_{i=0}^{\log_2 n} c = \Theta(\log_2 n)$

Iterative Version

BinarySearchIt(*A*, *low*, *high*, *key*)

while *low* ≤ *high*:

$$mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$$

Iterative Version

BinarySearchIt(*A*, *low*, *high*, *key*)

while *low* ≤ *high*:

$mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$

if *key* = *A*[*mid*]:

return *mid*

Iterative Version

BinarySearchIt(*A*, *low*, *high*, *key*)

while $low \leq high$:

$mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$

if $key = A[mid]$:

 return mid

else if $key < A[mid]$:

$high = mid - 1$

Iterative Version

BinarySearchIt(*A*, *low*, *high*, *key*)

while $low \leq high$:

$mid \leftarrow \left\lfloor low + \frac{high - low}{2} \right\rfloor$

if $key = A[mid]$:

 return *mid*

else if $key < A[mid]$:

$high = mid - 1$

else:

$low = mid + 1$

Iterative Version

BinarySearchIt(*A*, *low*, *high*, *key*)

```
while low ≤ high:  
    mid ←  $\left\lfloor \textit{low} + \frac{\textit{high} - \textit{low}}{2} \right\rfloor$   
    if key = A[mid]:  
        return mid  
    else if key < A[mid]:  
        high = mid - 1  
    else:  
        low = mid + 1  
return low - 1
```

Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
chair	chaise	sedia	Sessel	silla
pimple	bouton	foruncolo	Pickel	espenilla

Real-life Example

english **french** **italian** **german** **spanish**
(sorted) (sorted) (sorted) (sorted) (sorted)

chair	chaise	casa	Haus	casa
house	bouton	foruncolo	Pickel	espenilla
pimple	maison	sedia	Sessel	silla

Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
chair	chaise	sedia	Sessel	silla
pimple	bouton	foruncolo	Pickel	espenilla

english

sorted

2
1
3

spanish

sorted

1
3
2

Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
chair	chaise	sedia	Sessel	silla
pimple	bouton	foruncolo	Pickel	espenilla

english

sorted



2
1
3

spanish

sorted


1
3
2

Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
chair	chaise	sedia	Sessel	silla
pimple	bouton	foruncolo	Pickel	espenilla

english
sorted
2
1
3

spanish
sorted
1
3
2



Real-life Example

english	french	italian	german	spanish
house	maison	casa	Haus	casa
chair	chaise	sedia	Sessel	silla
pimple	bouton	foruncolo	Pickel	espenilla

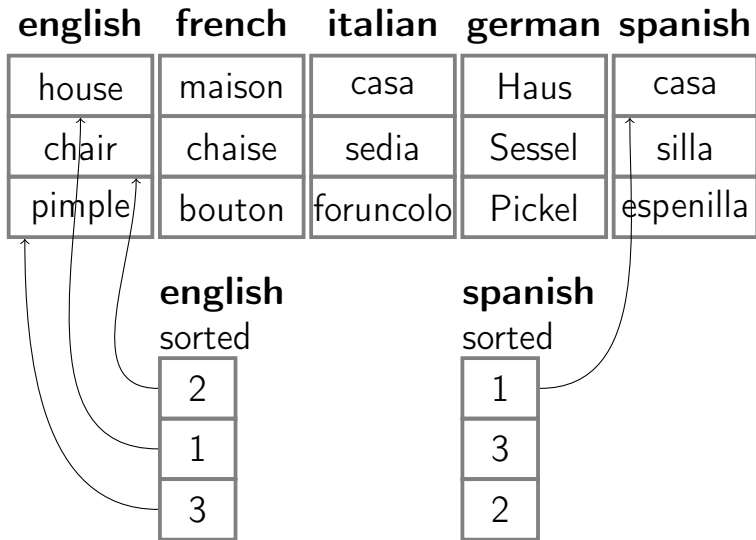
english	spanish
sorted	sorted
2	1
1	3
3	2

The diagram illustrates word sorting in two languages. The main table lists words in English, French, Italian, German, and Spanish. Below it, two columns show the sorted order of the English and Spanish words. Arrows indicate the mapping from the sorted indices back to the original word positions in the English column.

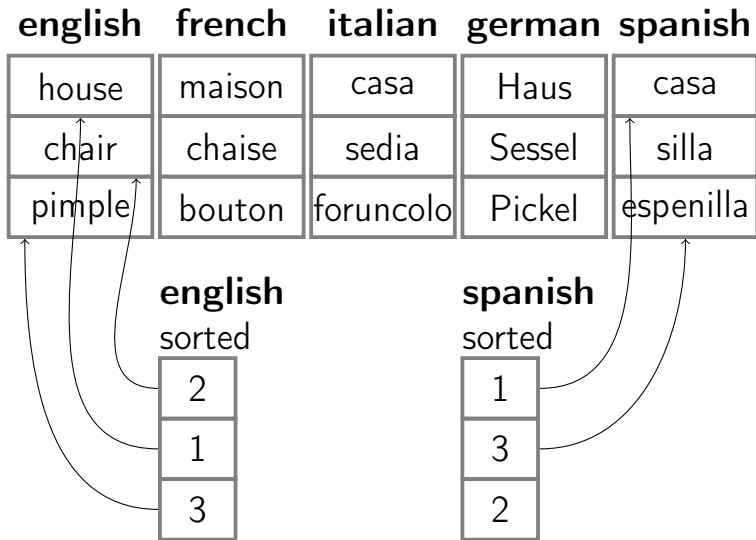
English sorted: 2, 1, 3. Arrows point from these indices to the words 'chair' (index 2), 'pimple' (index 1), and 'house' (index 3) in the main table.

Spanish sorted: 1, 3, 2. Arrows point from these indices to the words 'casa' (index 1), 'espenilla' (index 3), and 'silla' (index 2) in the main table.

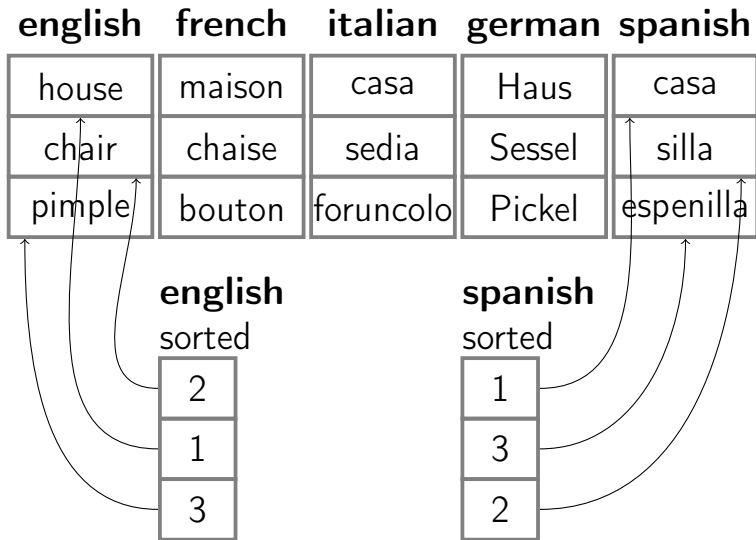
Real-life Example



Real-life Example



Real-life Example



Summary

Summary

The runtime of binary search is $\Theta(\log n)$.