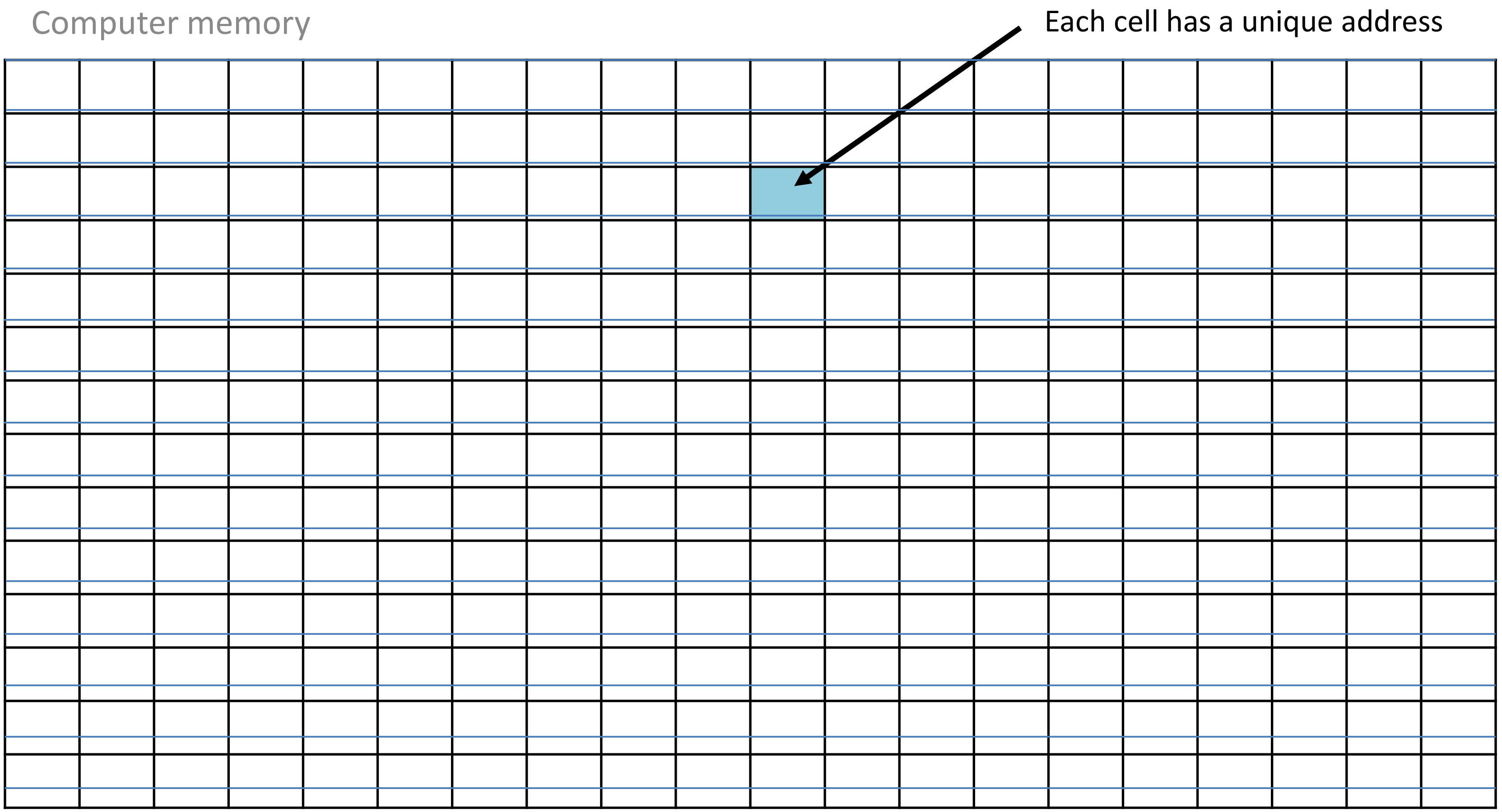
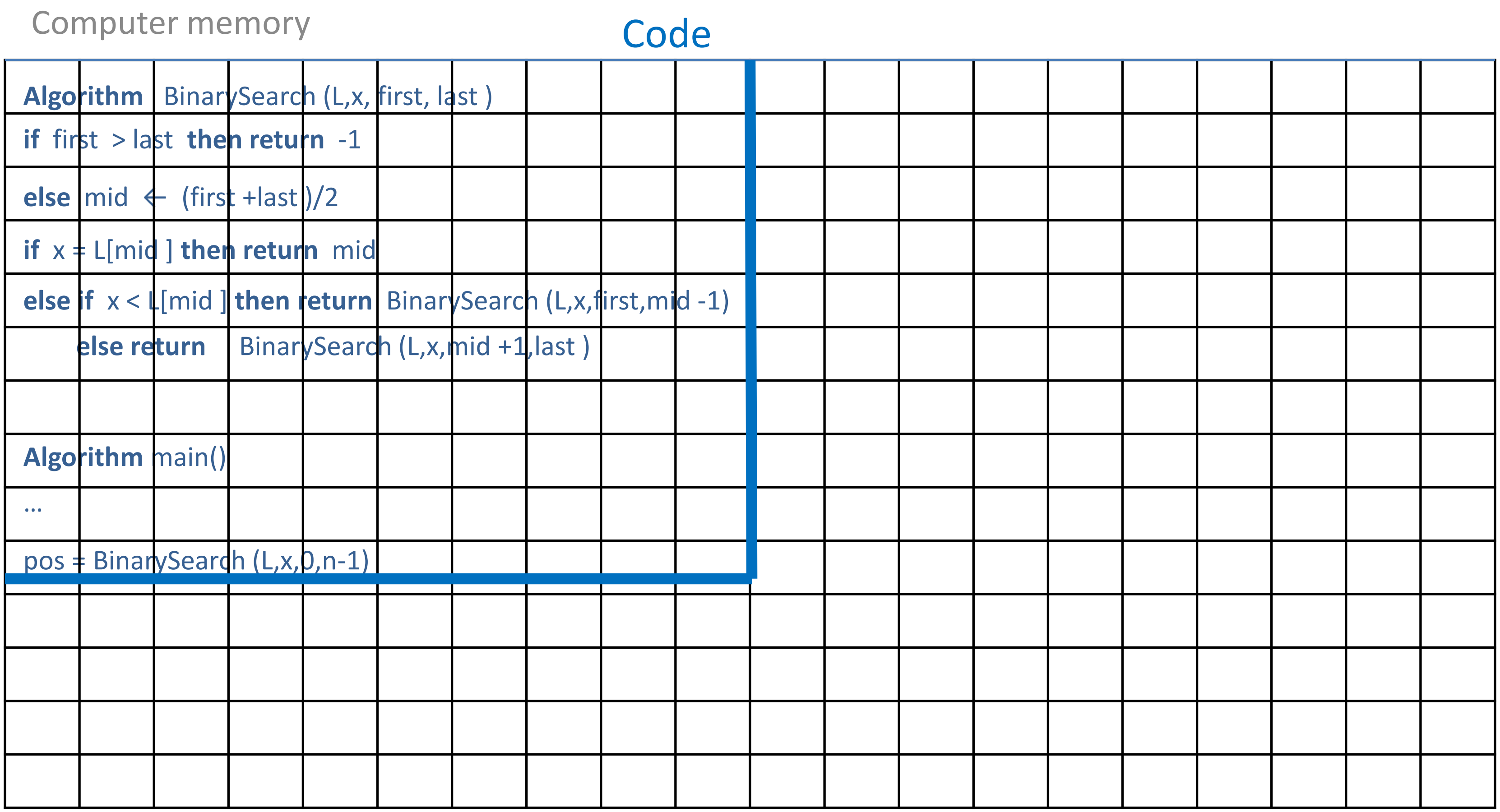
# Execution of a Recursive Algorithm

We will illustrate how a computer executes a recursive algorithm using the recursive version of binary search as example.

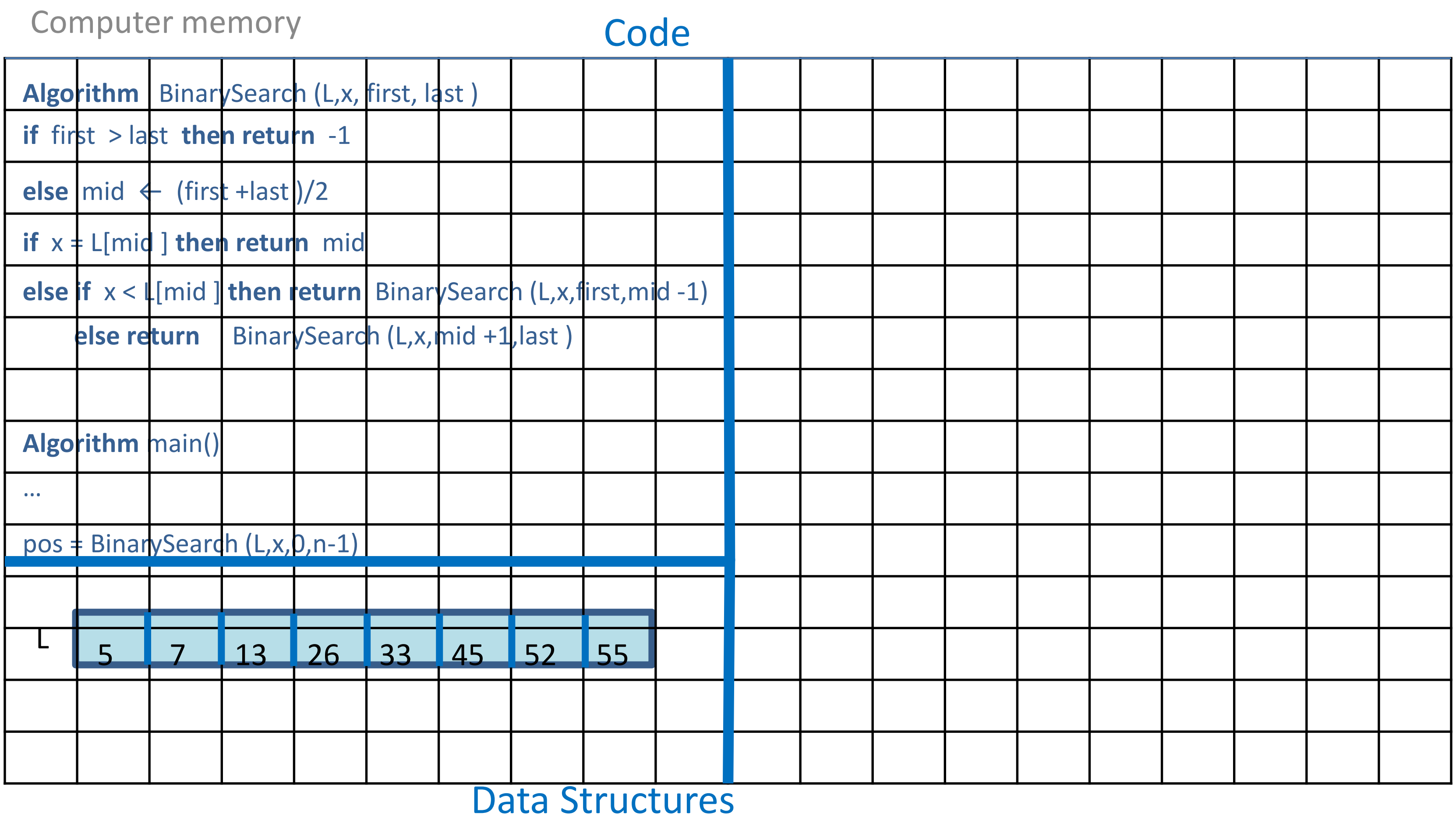
# Partitions of the Memory



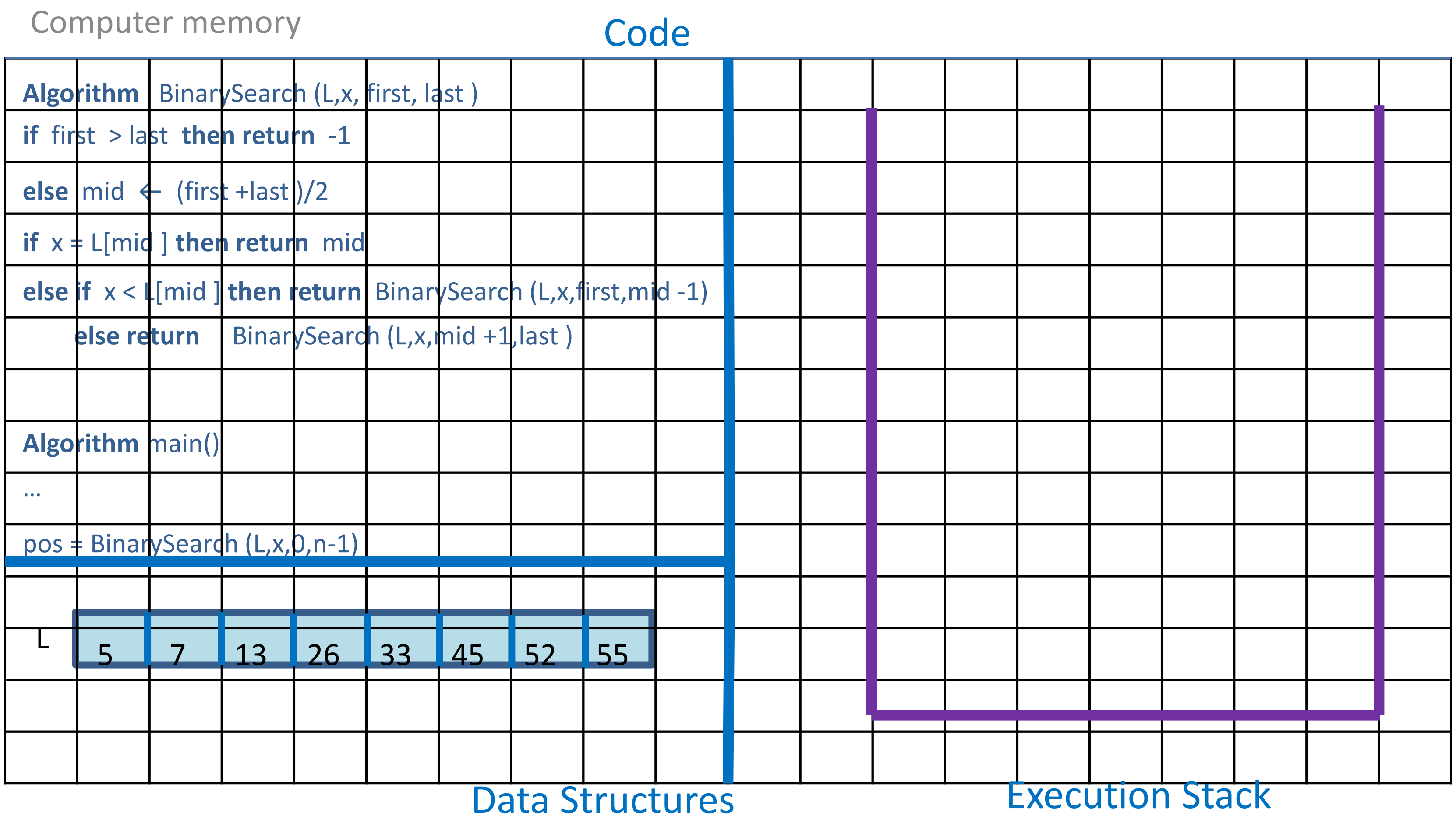
# Partitions of the Memory



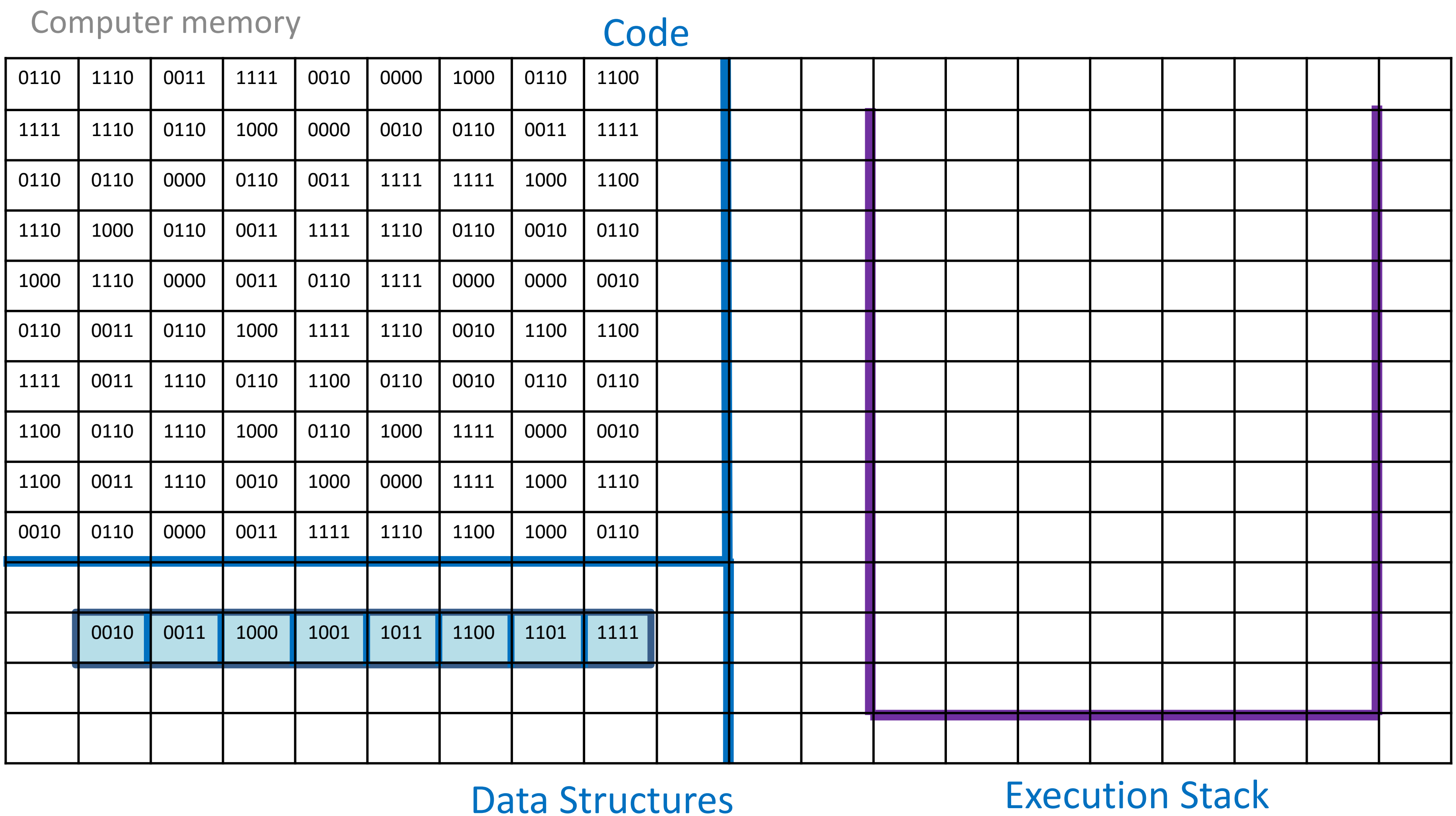
# Partitions of the Memory



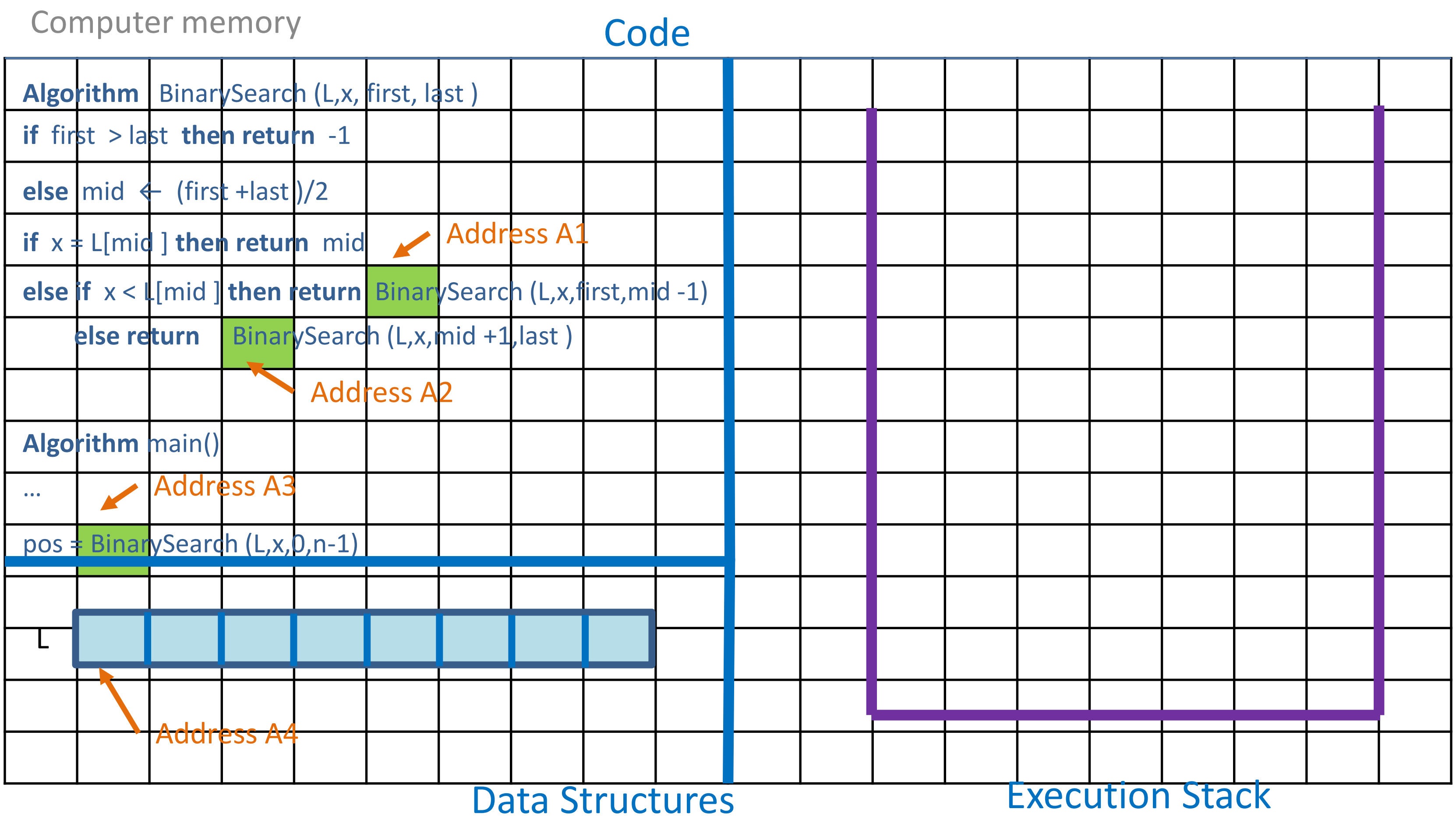
# Partitions of the Memory



# Information is Stored in Binary



# Execution of Binary Search



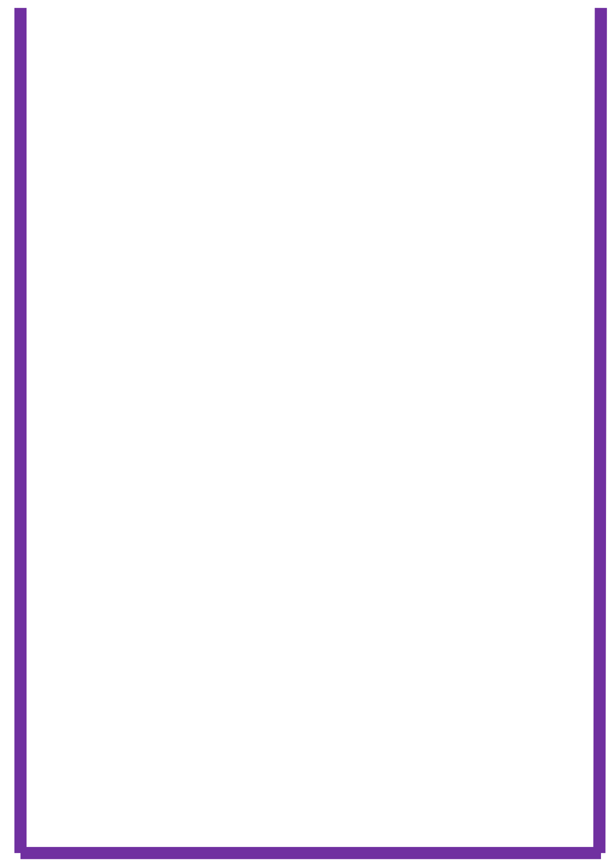
# Execution of Binary Search

```
Algorithm   BinarySearch (L,x, first, last )
if  first  > last  then return  -1
else  mid  ←  (first +last )/2
if  x = L[mid ] then return  mid
else if  x < L[mid ] then return  BinarySearch (L,x,first,mid -1)  A1
      else return     BinarySearch (L,x,mid +1,last )  A2
```

```
Algorithm main()
...
pos = BinarySearch (L,x,0,n-1)   A0
...
```

| L | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |

x = 23



Execution Stack

# Activation Records

Every time that an algorithm (or method) is invoked an activation record is created at the top of the execution stack.

# Activation Records

Every time that an algorithm (or method) is invoked an activation record is created at the top of the execution stack.

An activation record stores all the information that an algorithm needs to be executed:

- parameters
- local variables
- return address
- return value (if any)

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )

**if** first > last **then return** -1

**else** mid ← (first +last )/2

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1

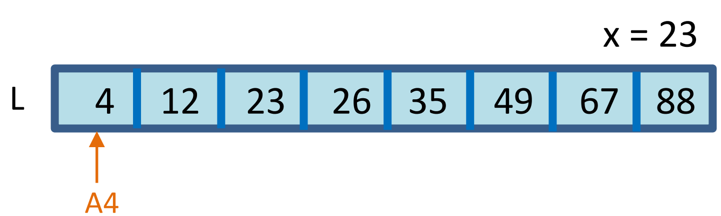
**else return** BinarySearch (L,x,mid +1,last ) ← A2

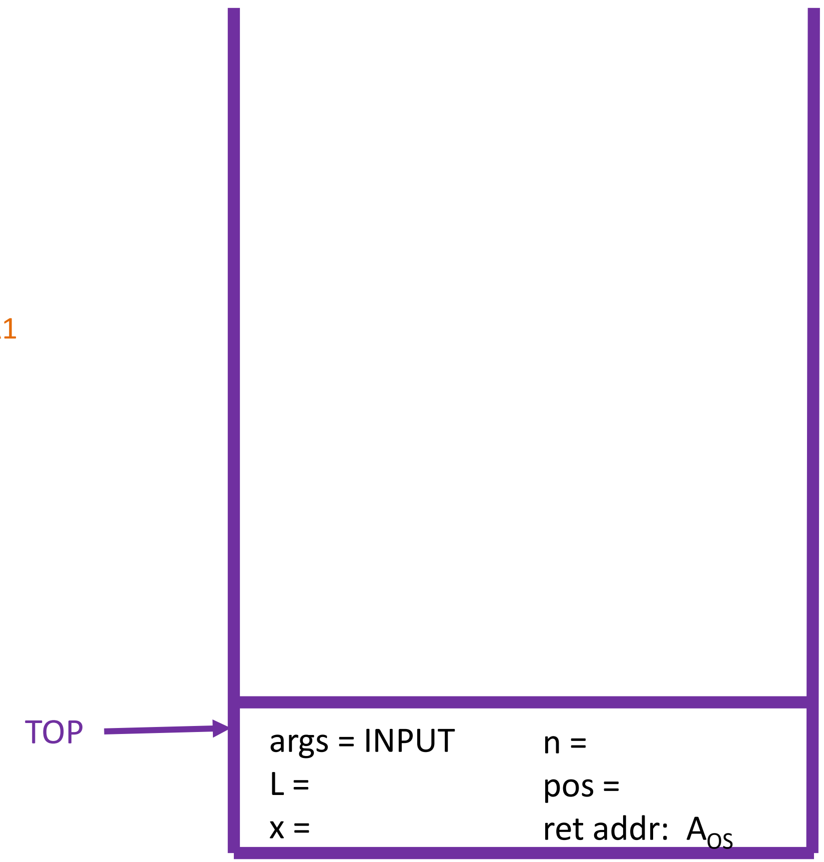
**Algorithm main(args)**

...

pos = BinarySearch (L,x,0,n-1) ← A3

...



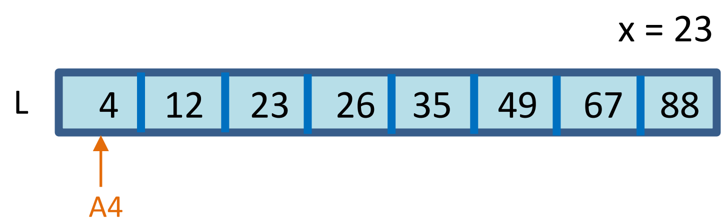



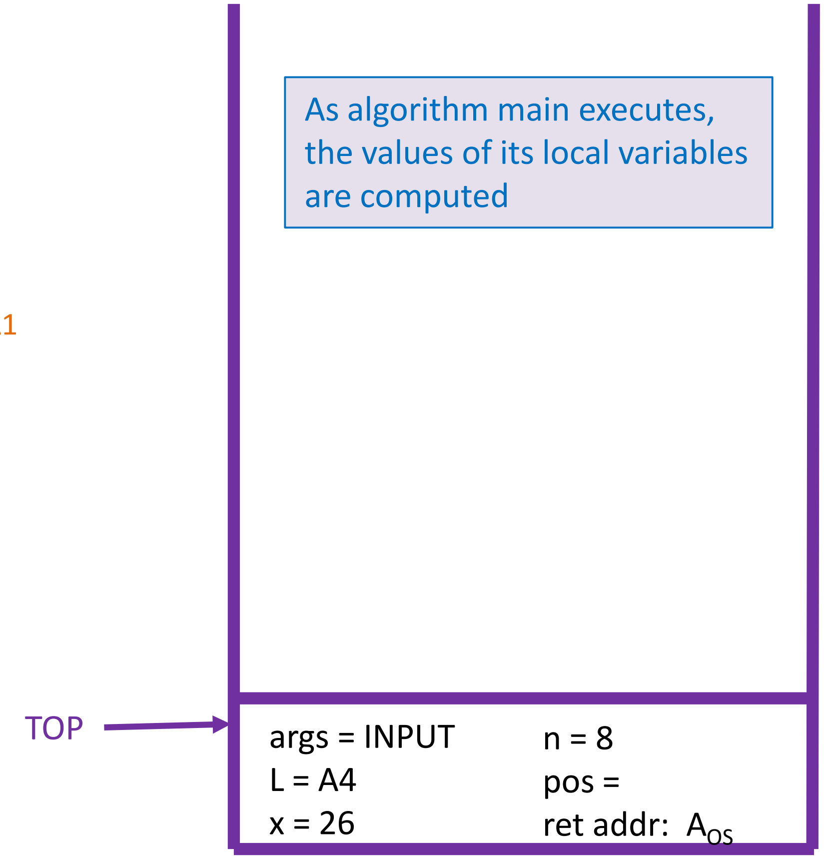

Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
    **else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...




Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )

**if** first > last **then return** -1

**else** mid ← (first +last )/2

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1

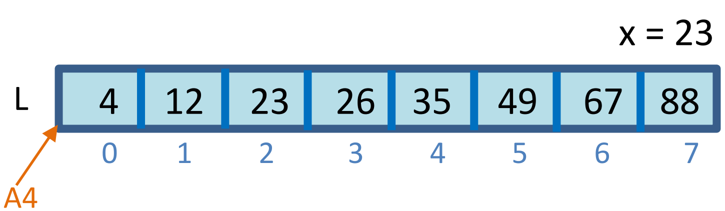
**else return** BinarySearch (L,x,mid +1,last ) ← A2

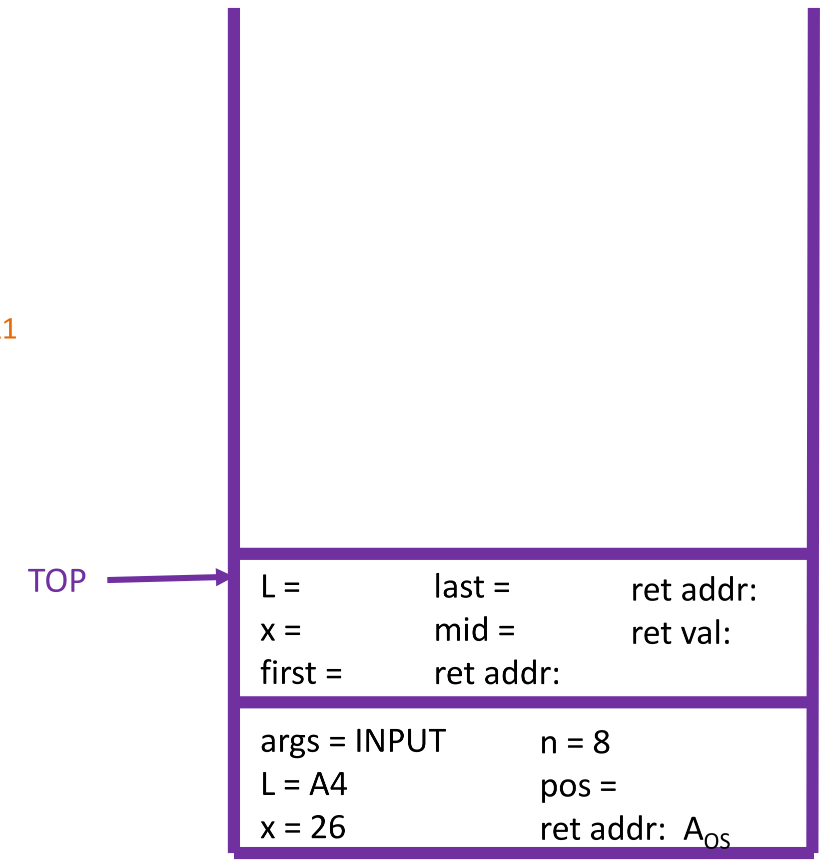
**Algorithm** main(args)

…

pos = BinarySearch (L,x,0,n-1) ← A3

…




Execution Stack

# Execution of Binary Search

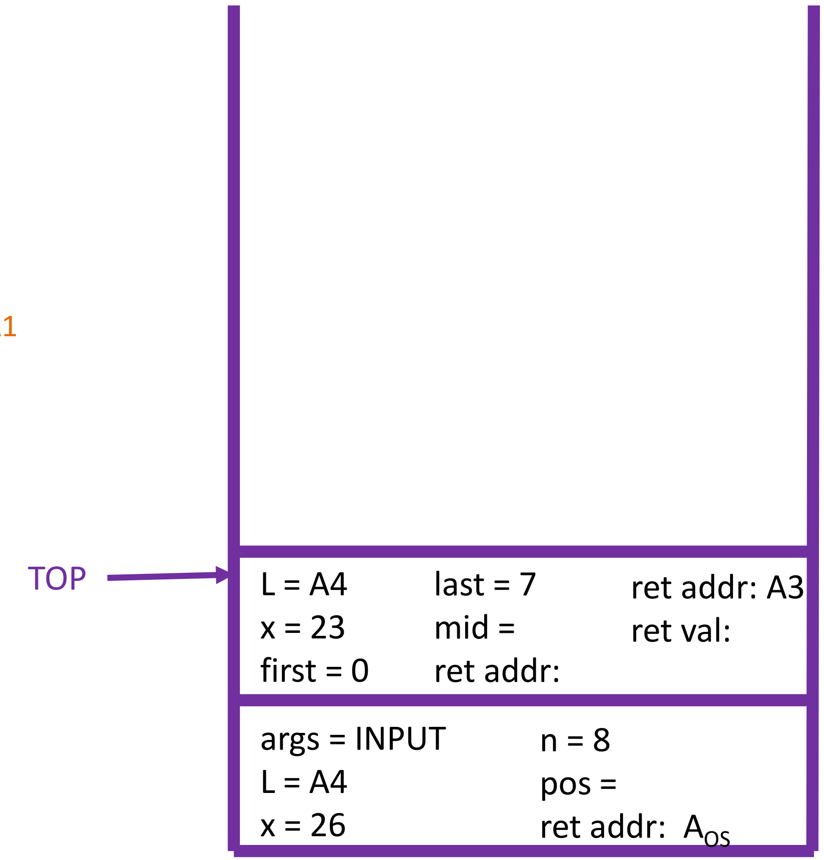
```
Algorithm  BinarySearch (L,x, first, last )
if first > last then return -1
else mid ← (first +last )/2
if x = L[mid ] then return mid
else if x < L[mid ] then return BinarySearch (L,x,first,mid -1)  ← A1
     else return  BinarySearch (L,x,mid +1,last )  ← A2
```

```
Algorithm main(args)
...
pos = BinarySearch (L,x,0,n-1)  ← A3
...
```

x = 23

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| L | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |

A4

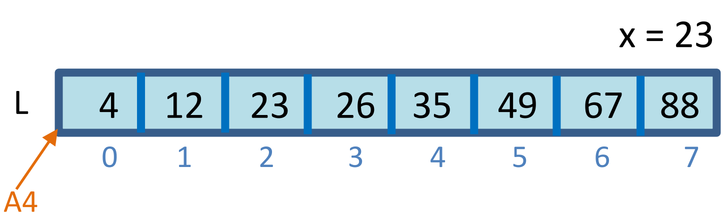


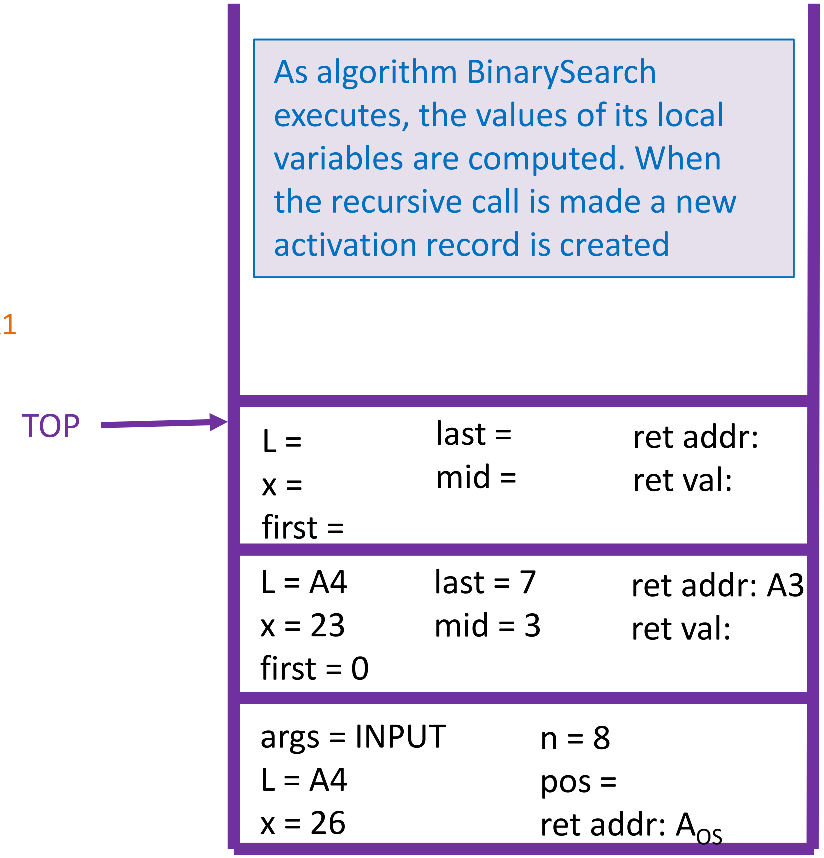

Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
**else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...




Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
    **else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...

x = 23

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| L | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |

A4

Since x = L[mid] the algorithm returns the value mid. The value of mid is stored in the activation record ...

TOP →

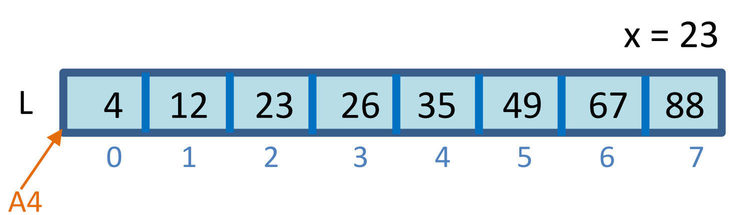
| | | |
|---|---|---|
| L = A4<br>x = 23<br>first = 2 | last = 2<br>mid = 1 | ret addr: A2<br>ret val: 2 |
| L = A4<br>x = 23<br>first = 0 | last = 2<br>mid = 1 | ret addr: A1<br>ret val: |
| L = A4<br>x = 23<br>first = 0 | last = 7<br>mid = 3 | ret addr: A3<br>ret val: |
| args = INPUT<br>L = A4<br>x = 26 | n = 8<br>pos =<br>ret addr: $A_{OS}$ | |

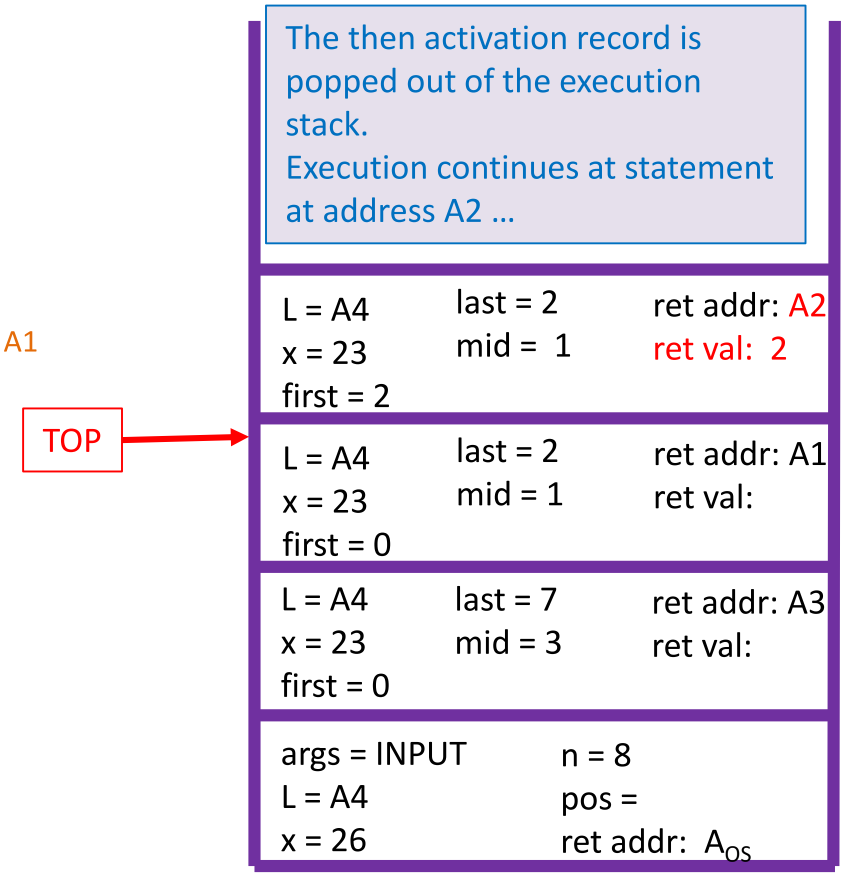
Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
    **else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
…
pos = BinarySearch (L,x,0,n-1) ← A3
…




Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )

**if** first > last **then return** -1

**else** mid ← (first +last )/2

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1

**else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)

...

pos = BinarySearch (L,x,0,n-1) ← A3

...

x = 23

| L | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

A4

The statement at address A2 returns the value of mid. This value is stored in the activation record ...

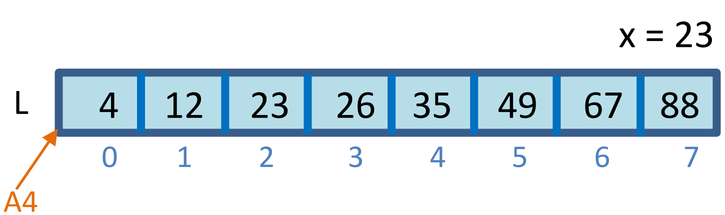
| | | |
|---|---|---|
| L = A4<br>x = 23<br>first = 2 | last = 2<br>mid = 1 | ret addr: A2<br>ret val: 2 |
| TOP → L = A4<br>x = 23<br>first = 0 | last = 2<br>mid = 1 | ret addr: A1<br>ret val: 2 |
| L = A4<br>x = 23<br>first = 0 | last = 7<br>mid = 3 | ret addr: A3<br>ret val: |
| args = INPUT<br>L = A4<br>x = 26 | n = 8<br>pos =<br>ret addr: $A_{OS}$ | |

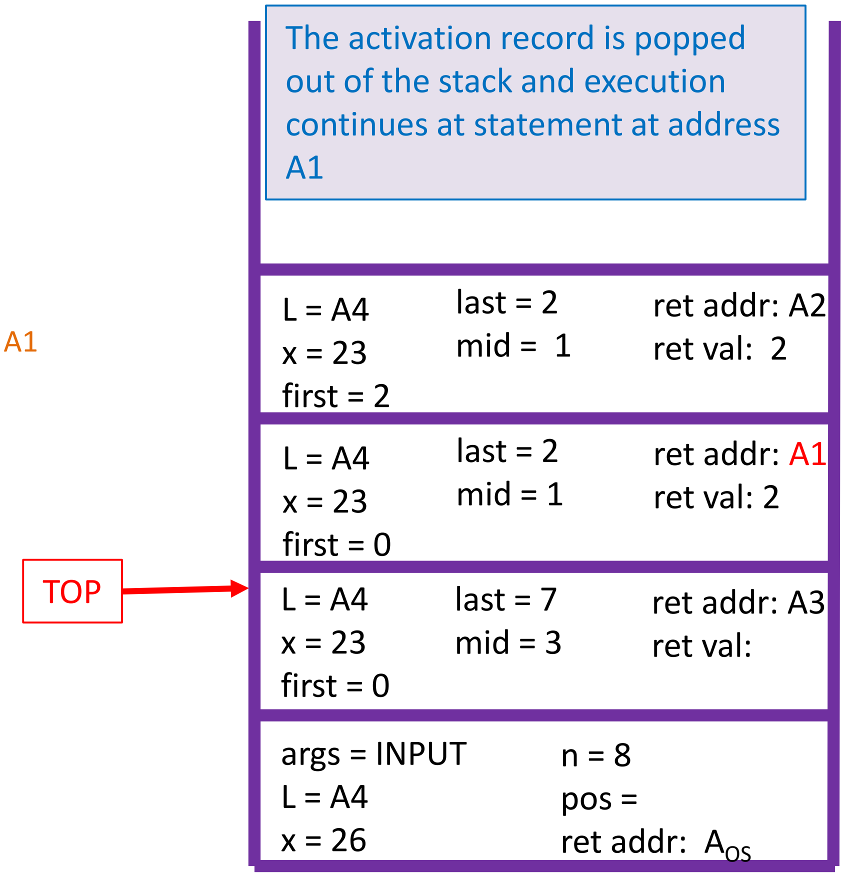
Execution Stack

# Execution of Binary Search

```
Algorithm  BinarySearch (L,x, first, last )
if  first  > last  then return  -1
else  mid  ←  (first +last )/2
if  x = L[mid ] then return  mid
else if  x < L[mid ] then return  BinarySearch (L,x,first,mid -1)  ← A1
      else return    BinarySearch (L,x,mid +1,last )  ← A2
```

```
Algorithm main(args)
...
pos = BinarySearch (L,x,0,n-1)  ← A3
...
```



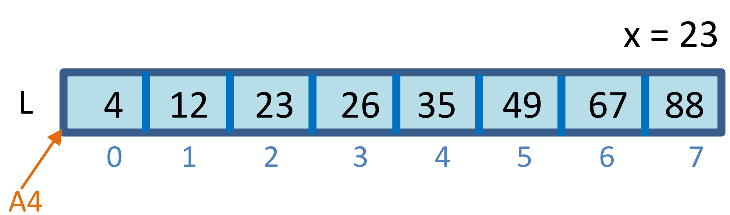



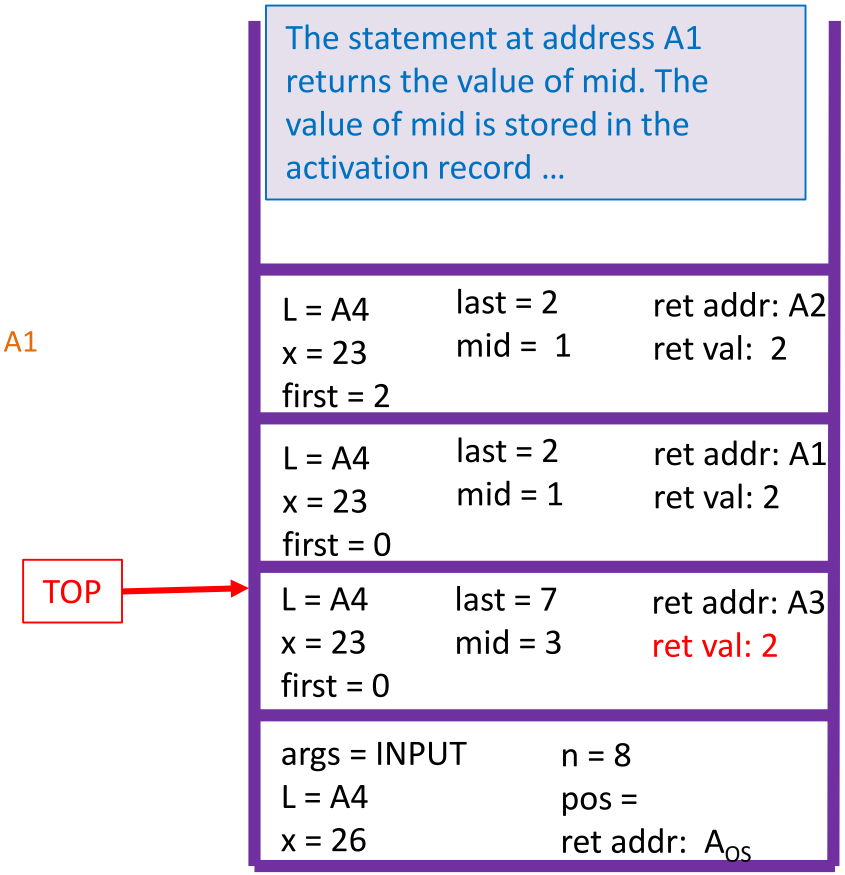

Execution Stack

# Execution of Binary Search

```
Algorithm  BinarySearch (L,x, first, last )
if first > last then return -1
else mid ← (first +last )/2
if x = L[mid ] then return mid
else if x < L[mid ] then return BinarySearch (L,x,first,mid -1)  ← A1
     else return  BinarySearch (L,x,mid +1,last )  ← A2
```

```
Algorithm main(args)
…
pos = BinarySearch (L,x,0,n-1)  ← A3
…
```




Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1

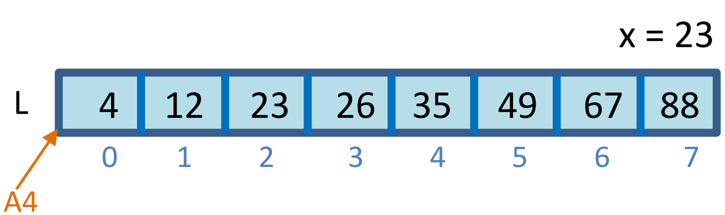
**else** mid ← (first +last )/2

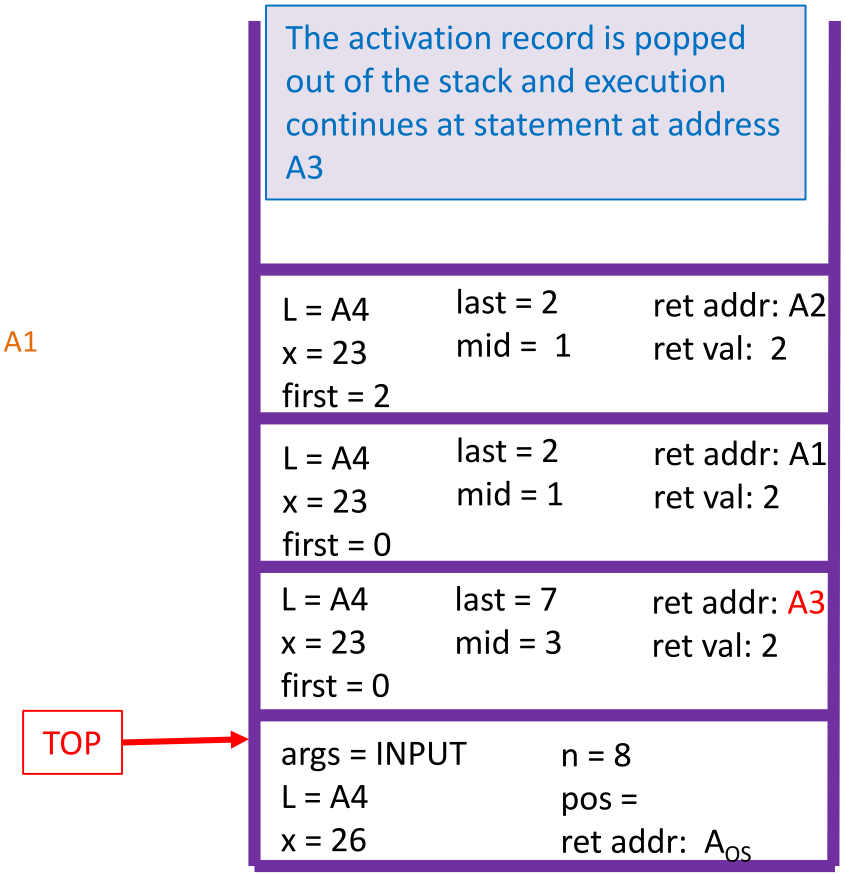
**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1

**else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...




Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
  **else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...

x = 23

| L | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

A4

Statement at address A3 stores the value returned by BinarySearch into variable pos

L = A4 last = 2 ret addr: A2
x = 23 mid = 1 ret val: 2
first = 2

L = A4 last = 2 ret addr: A1
x = 23 mid = 1 ret val: 2
first = 0

L = A4 last = 7 ret addr: A3
x = 23 mid = 3 ret val: 2
first = 0

TOP →

args = INPUT n = 8
L = A4 pos = 2
x = 26 ret addr: $A_{OS}$

Execution Stack

# Execution of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**if** first > last **then return** -1
**else** mid ← (first +last )/2
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1) ← A1
  **else return** BinarySearch (L,x,mid +1,last ) ← A2

**Algorithm** main(args)
...
pos = BinarySearch (L,x,0,n-1) ← A3
...

x = 23

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| L | 4 | 12 | 23 | 26 | 35 | 49 | 67 | 88 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

A4

The rest of algorithm main is executed. When the algorithm ends the activation record is popped out of the stack and control goes back to the operating system

| | | |
|---|---|---|
| L = A4<br>x = 23<br>first = 2 | last = 2<br>mid = 1 | ret addr: A2<br>ret val: 2 |
| L = A4<br>x = 23<br>first = 0 | last = 2<br>mid = 1 | ret addr: A1<br>ret val: 2 |
| L = A4<br>x = 23<br>first = 0 | last = 7<br>mid = 3 | ret addr: A3<br>ret val: 2 |
| args = INPUT<br>L = A4<br>x = 26 | n = 8<br>pos = 2<br>ret addr: $A_{OS}$ | |

TOP →

Execution Stack

# Time Complexity of Binary Search

We will now compute the time complexity of binary search by first writing a recurrence equation for the time complexity function and then solving this equation using repeated substitution.

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \le i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1

**else** mid $\leftarrow \lfloor (first + last)/2 \rfloor$

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)

**else return** BinarySearch (L,x,mid +1,last )

The worst case for binary search is when x is not in L. Let

$f(n)$ = number of primitive operations performed by binary search in the worst case when the size of the input is n

We will compute $f(n)$ for the base case and the recursive case.

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

$$\left.\begin{array}{l} \textbf{if}\ \ \text{first} > \text{last}\ \ \textbf{then return}\ \ -1 \\ \textbf{else}\ \ \text{mid} \leftarrow \lfloor(\text{first} + \text{last})/2\rfloor \end{array}\right\} c_1$$

**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)
    **else return** BinarySearch (L,x,mid +1,last )

In the base case the algorithm performs a constant number $c_1$ of primitive operations. Note that in the base case fist > last, so the number of elements n is 0:

$$f(0) = c_1$$

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

$$\left.\begin{array}{l}
\textbf{if} \text{ first } > \text{ last } \textbf{then return } -1 \\
\textbf{else} \text{ mid } \leftarrow \lfloor(\text{first} + \text{last})/2\rfloor \\
\textbf{if } x = L[\text{mid}] \textbf{ then return } \text{mid} \\
\textbf{else if } x < L[\text{mid}] \textbf{ then return}
\end{array}\right\} c_2$$

BinarySearch (L,x,first,mid -1)

**else return** BinarySearch (L,x,mid +1,last )

Ignoring the recursive calls, when $n > 0$ the algorithm performs a constant number $c_2$ of primitive operations ...

$f(n) = c2 + \ldots$ for $n > 0$

We need to add to this the number of operations performed by the recursive calls.

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \le i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1
**else** mid $\leftarrow \lfloor$(first +last )/2$\rfloor$
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)
    **else return** BinarySearch (L,x,mid +1,last )

$c_2$

Let L[first..last] denote the part of array L that starts at index first and ends at index last.

first    last

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *L* | 3 | 9 | 11 | 17 | 18 | 26 | 29 | 43 | 48 | 55 |

L[first..last]

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \le i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1

**else** mid ← ⌊(first +last )/2⌋

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)

**else return** BinarySearch (L,x,mid +1,last )

If the number of elements in L is n and the first recursive call is made, the number of elements in the first half of the array is (n-1)/2, so the number of primitive operations performed by the first recursive call is

f((n-1)/2)

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1

**else** mid ← ⌊(first +last )/2⌋

**if** x = L[mid ] **then return** mid

**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)

**else return** BinarySearch (L,x,mid +1,last )

Similarly, if the second recursive call is made, the number of elements in the second half of the array is (n-1)/2, so the number of primitive operations performed by the second recursive call is also

$f((n-1)/2)$

# Time Complexity of Binary Search

**Algorithm** BinarySearch (L,x, first, last )
**Input**: Array L of size n and value x
**Output**: Index i, $0 \le i < n$, such that L[i] = x if x in L, or -1 if x not in L

$c_1$ $c_2$

**if** first > last **then return** -1
**else** mid $\leftarrow \lfloor$(first +last )/2$\rfloor$
**if** x = L[mid ] **then return** mid
**else if** x < L[mid ] **then return** BinarySearch (L,x,first,mid -1)
    **else return** BinarySearch (L,x,mid +1,last )

f((n-1)/2)

So, the number of primitive operations performed by the algorithm is

$$f(0) = c_1$$
$$f(n) = c_2 + f((n-1)/2) \text{ for } n > 0$$

This equation is called a recurrence equation.

# Solving the Recurrence Equation using Repeated Substitution

$f(0) = c_1$ (1)

$f(n) = f\left(\frac{n-1}{2}\right) + c_2$ (2)

Start with equation (2):

$f(n) = f\left(\frac{n-1}{2}\right) + c_2$ Use (2) to compute $f\left(\frac{n-1}{2}\right)$

$f\left(\frac{n-1}{2}\right) = f\left(\frac{\frac{n-1}{2}-1}{2}\right) + c_2 = f\left(\frac{n-1-2}{2^2}\right) + c_2 = f\left(\frac{n-2^0-2^1}{2^2}\right) + c_2$ Use (2) to compute $f\left(\frac{n-2^0-2^1}{2^2}\right)$

$f\left(\frac{n-2^0-2^1}{2^2}\right) = f\left(\frac{\frac{n-2^0-2^1}{2^2}-1}{2}\right) + c_2 = f\left(\frac{n-2^0-2^1-2^2}{2^3}\right) + c_2$ And so on …

$f\left(\frac{n-2^0-2^1-2^2}{2^3}\right) = f\left(\frac{n-2^0-2^1-2^2-2^3}{2^4}\right) + c_2$

$\vdots$

$$f\left(\frac{n-2^0-2^1-2^2-\ldots-2^j}{2^{j+1}}\right) = f\underbrace{\left(\frac{n-2^0-2^1-2^2-\ldots-2^j}{2^{j+2}}\right)}_{=0} + c_2 = \underbrace{c_1}_{f(0) = c_1} + c_2$$

Now we substitute each equation into the equation above it

# Solving the Recurrence Equation using Repeated Substitution

$f(0) = c_1$ (1)

$f(n) = f\left(\frac{n-1}{2}\right) + c_2$ (2)

Start with equation (2):

$$f(n) = f\left(\frac{n-1}{2}\right) + c_2$$

$$f\left(\frac{n-1}{2}\right) = f\left(\frac{\frac{n-1}{2}-1}{2}\right) + c_2 = f\left(\frac{n-1-2}{2^2}\right) + c_2 = f\left(\frac{n-2^0-2^1}{2^2}\right) + c_2$$ Use (2) to compute $f\left(\frac{n-2^0-2^1}{2^2}\right)$

$$f\left(\frac{n-2^0-2^1}{2^2}\right) = f\left(\frac{\frac{n-2^0-2^1}{2^2}-1}{2}\right) + c_2 = f\left(\frac{n-2^0-2^1-2^2}{2^3}\right) + c_2$$ And so on …

$$f\left(\frac{n-2^0-2^1-2^2}{2^3}\right) = f\left(\frac{n-2^0-2^1-2^2-2^3}{2^4}\right) + c_2$$

.

.

.

$$f\left(\frac{n-2^0-2^1-2^2-\ldots-2^j}{2^{j+1}}\right) = f\left(\frac{n-2^0-2^1-2^2-\ldots-2^{j+1}}{2^{j+2}}\right) + c_2 = c_1 + c_2$$

j+2

We get

$$f(n) = c_2 + c_2 + c_2 + \ldots + c_2 + c_1 = (j+2)\, c_1$$

Since $n-2^0-2^1-2^2-\ldots-2^{j+1} = 0$ then $n = 2^0+2^1+2^2+\ldots 2^{j+1} = 2^{j+2}-1$. Taking logarithms on both sides we get $\log_2(n+1) = j+2$, therefore $f(n) = c_1 \log_2(n+1)$

Using the rules we learned for computing the order of functions we finally get that f(n) is O(log n)

# Comparing Time Complexities

Linear search

$f(n)$ is $O(n) = \{t(n) \mid t(n) \leq c\, n$ for all $n \geq n_0$, $n_0$, $c$ constants$\}$

Binary search

$f(n)$ is $O(\log n) = \{t(n) \mid t(n) \leq c \log n$ for all $n \geq n_0$, $n_0$, $c$ const$\}$

running time of EVERY implementation of binary search

# Comparing Time Complexities

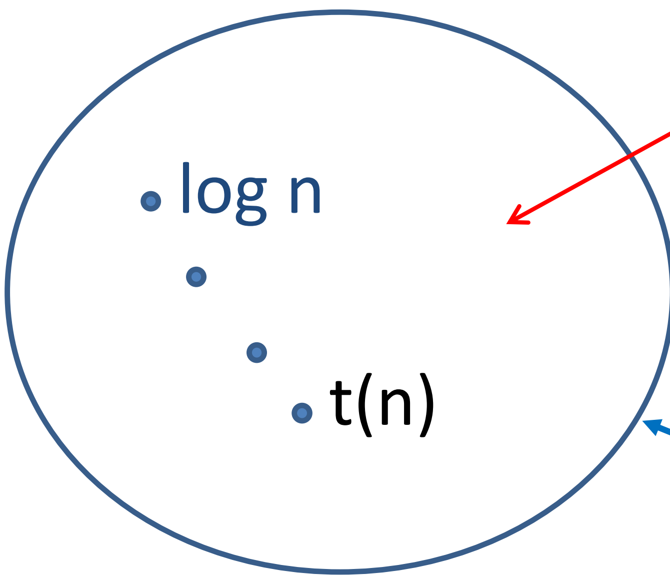
Linear search

f(n) is $O(n) = \{t(n) \mid t(n) \le c\, n$ for all $n \ge n_0$, $n_0$, $c$ constants$\}$

Binary search

f(n) is $O(\log n) = \{t(n) \mid t(n) \le c \log n$ for all $n \ge n_0$, $n_0$, $c$ const$\}$

running time of EVERY implementation of binary search



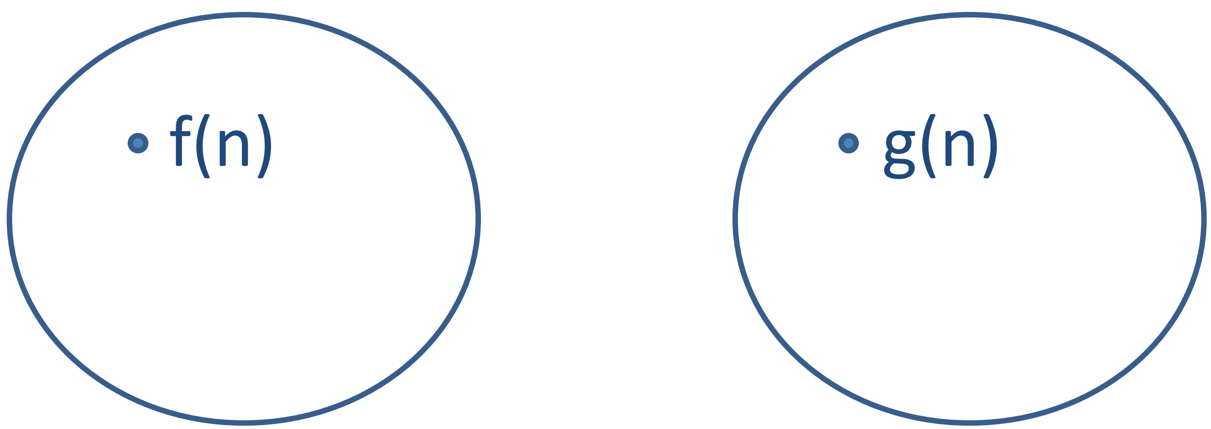

O (log n) = Running times of all possible implementations of binary search

# Comparing Orders

Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Which algorithm is faster?




O(f(n))    O(g(n))

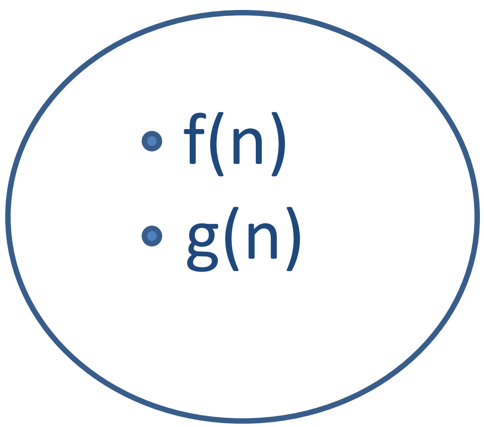
# Comparing Orders

Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Two cases:

- f(n) is O(g(n)) and g(n) is O(f(n))




Both algorithms have the same set of possible running times

O(f(n)) = O(g(n))

# Comparing Orders

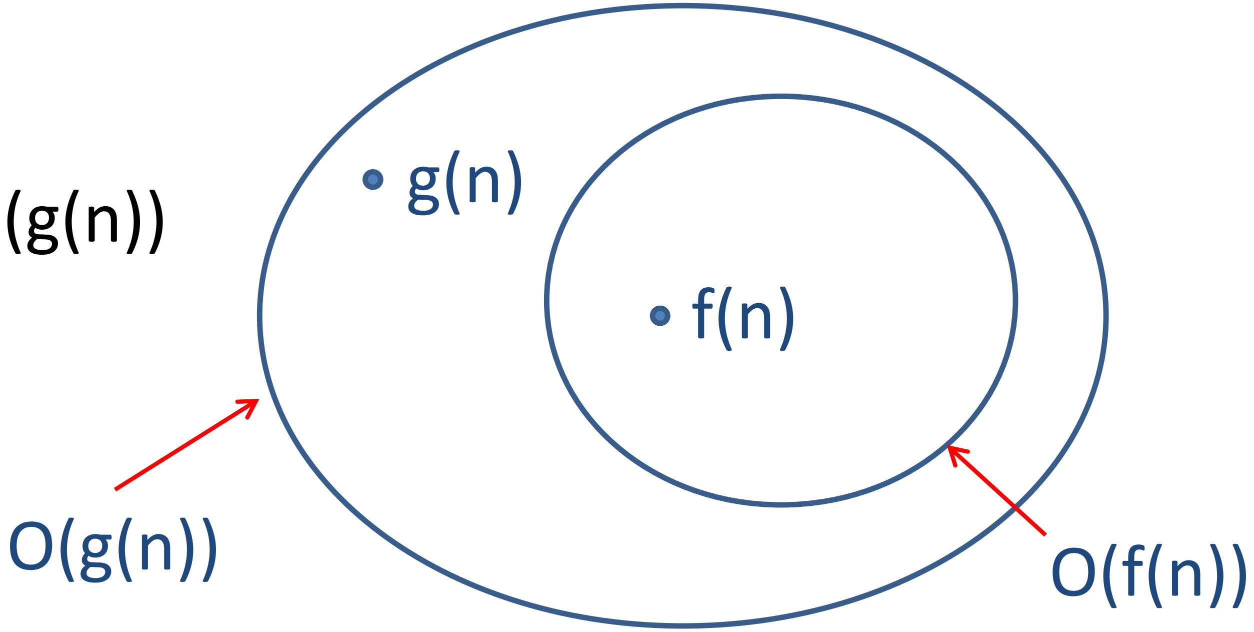
Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Two cases:

- f(n) is O(g(n)) and g(n) is **not** O(f(n))

$O(f(n)) \subset O(g(n))$



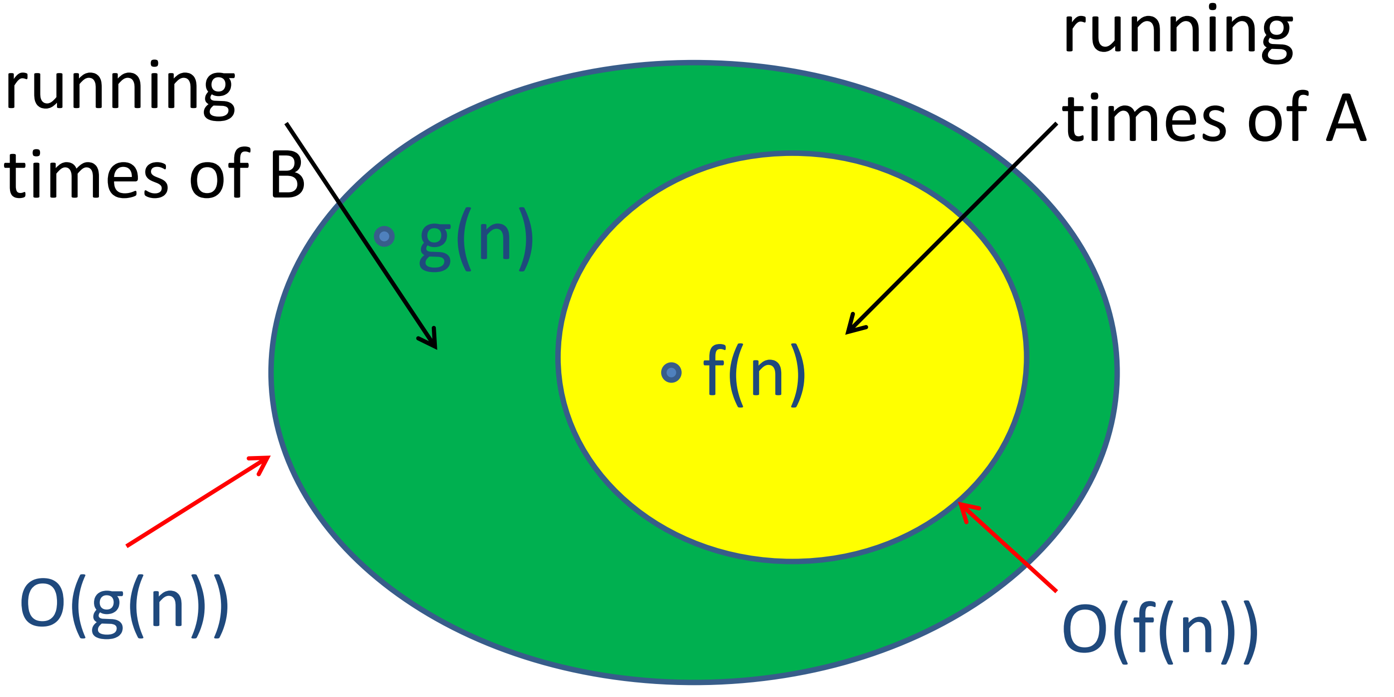
# Comparing Orders

Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Two cases:

- f(n) is O(g(n)) and g(n) is **not** O(f(n))



# Comparing Orders

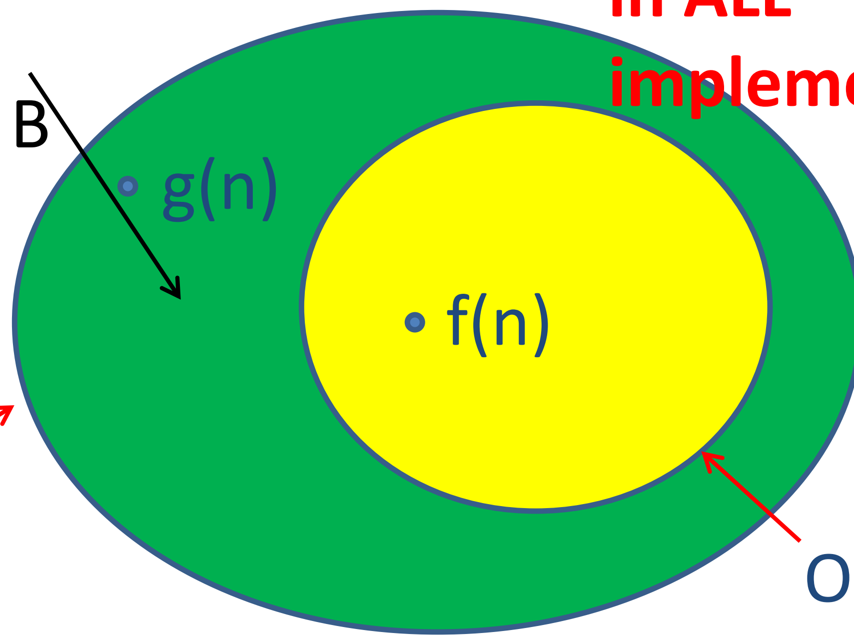
Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Two cases:

- f(n) is O(g(n)) and g(n) is **not** O(f(n)): **B is slower than A in ALL implementations**

running times of B




$g(n) > c\, f(n)$ for $n \geq n_0$ for all $c$, $n_0$, i.e. all implementations

O(g(n))

O(f(n))

# Complexity Classes

$$\underbrace{O(1)}_{\text{constant}} \subset \underbrace{O(\log n)}_{\text{logarithmic}} \subset \underbrace{O(n)}_{\text{linear}} \subset O(n \log n)$$

$$\subset \underbrace{O(n^2)}_{\text{quadratic}} \subset \underbrace{O(n^a)}_{\substack{\text{polynomial}\\ \text{(constant } a > 2)}} \subset \underbrace{O(b^n)}_{\substack{\text{exponential}\\ \text{(} b \text{ constant)}}}$$

$$\subset \underbrace{O(n!)}_{\text{factorial}} \subset O(n^n) \quad . \; . \; .$$

Efficient algorithms