# 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, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1
**else** mid $\leftarrow \lfloor (first + last)/2 \rfloor$
**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, $0 \le i < n$, 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, $0 \le i < n$, 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 )

$c_2$

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

$$f(n) = c2 + \ldots \quad \text{for } n > 0$$

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 ← ⌊(first +last )/2⌋ $c_2$
**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 )

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, $0 \leq i < n$, such that L[i] = x if x in L, or -1 if x not in L

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

**else** mid ← $\lfloor$(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 )

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, $0 \le i < n$, such that L[i] = x if x in L, or -1 if x not in L

**if** first > last **then return** -1
**else** mid $\leftarrow \lfloor(\text{first} + \text{last})/2\rfloor$
**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 )

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 ≤ i < n, such that L[i] = x if x in L, or -1 if x not in L

$c_1$

**if** first > last **then return** -1
**else** mid ←⌊(first +last )/2⌋

$c_2$

**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 )

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 …

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

$\vdots$

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

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 …

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

.

.

.

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

j+2

We get

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

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 $f(n) = c_1 + c_2\log_2(n+1)$

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

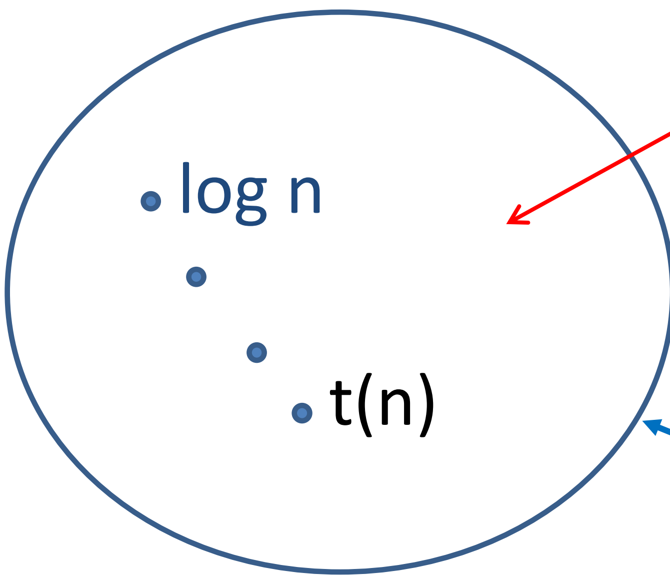
Linear search

$f(n)$ 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



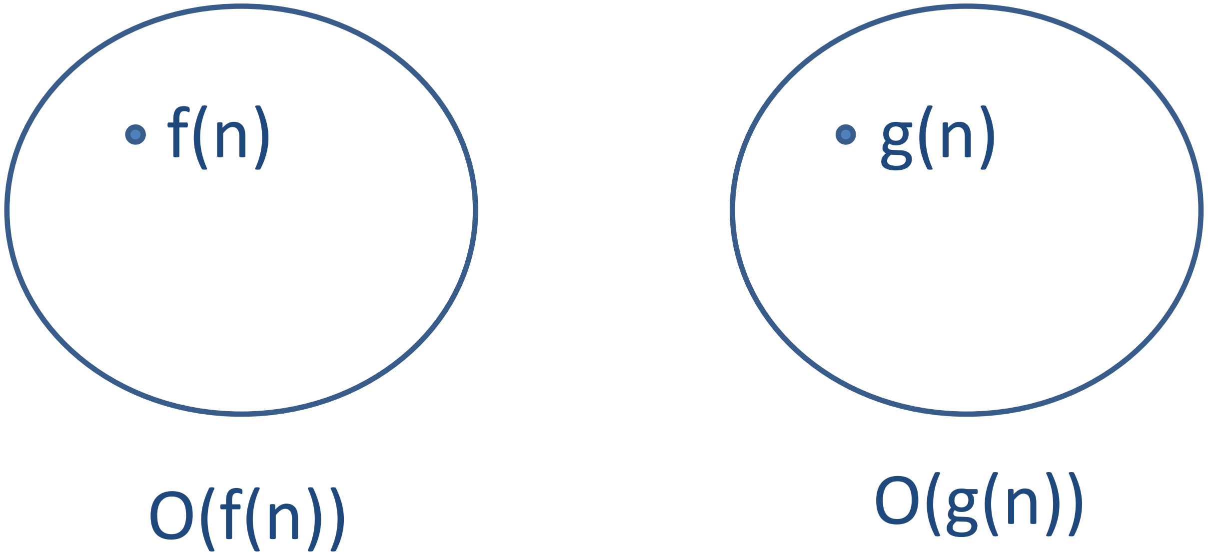

O (log n) = Running times of all possible implementations of binary search

# Comparing Orders

Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Which algorithm is faster?



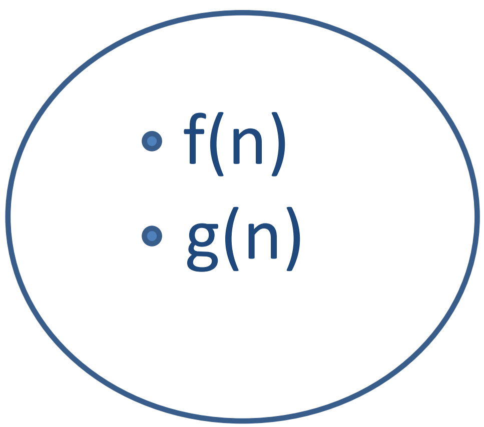
# Comparing Orders

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

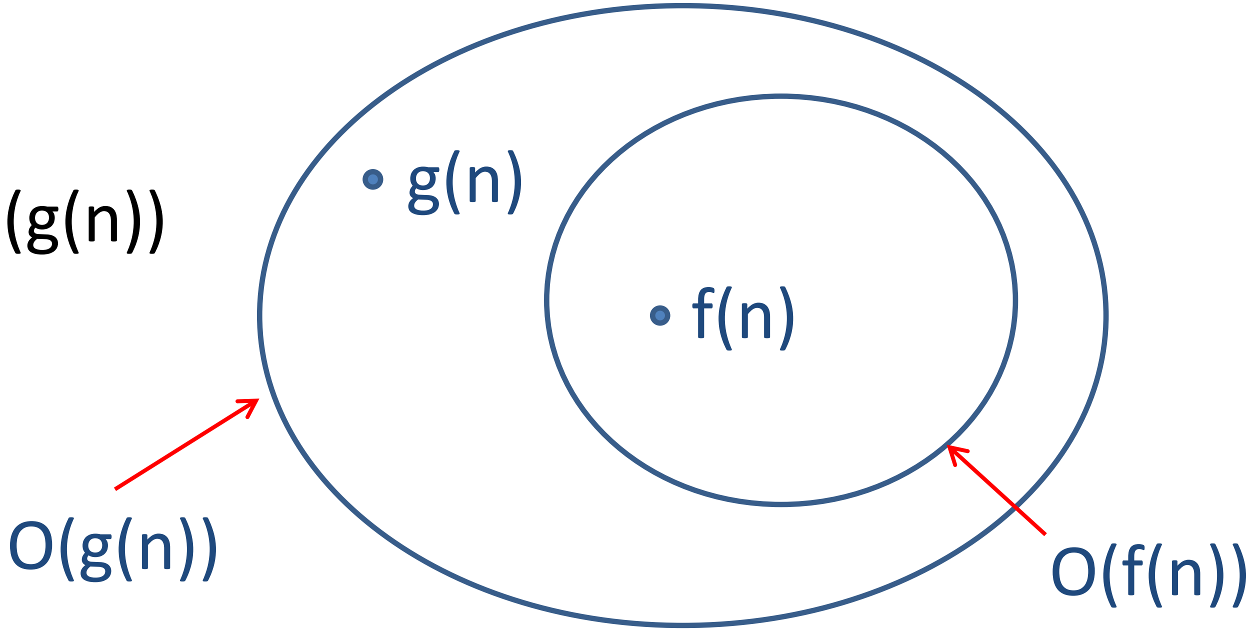
Algorithm A has complexity O(f(n))

Algorithm B has complexity O(g(n))

Two cases:

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

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



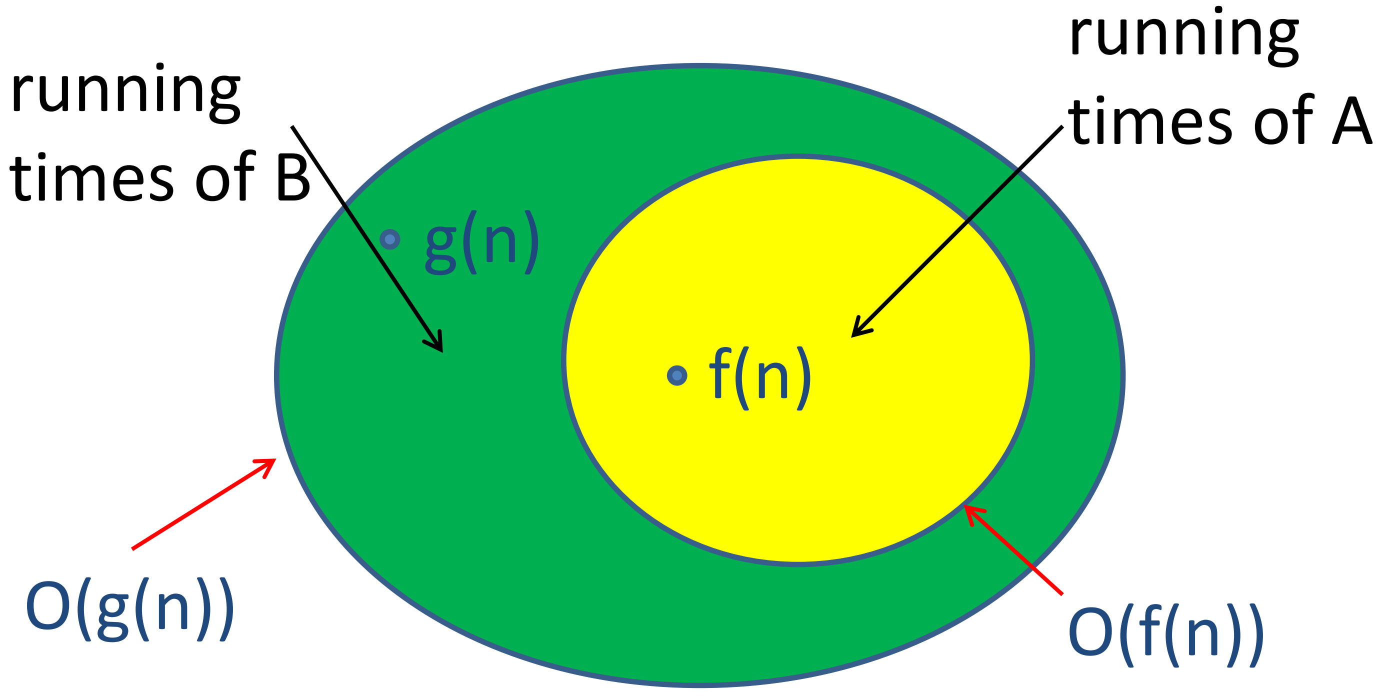
# Comparing Orders

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

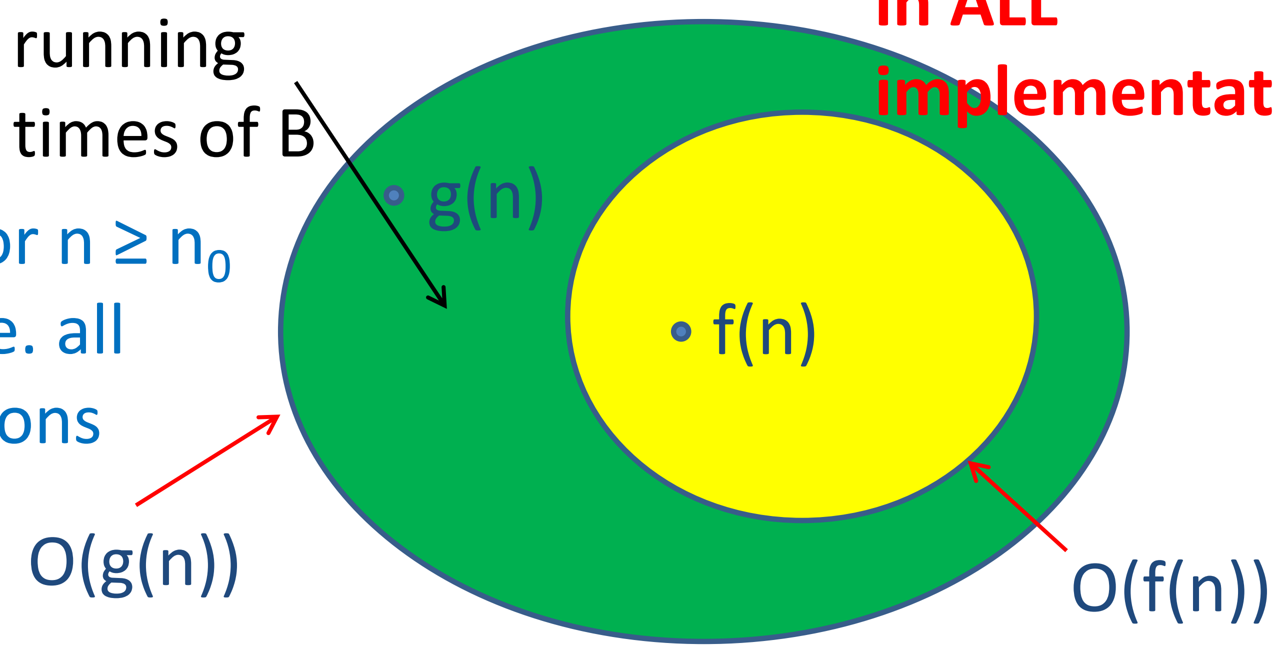
Algorithm A has complexity O(f(n))

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**

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



# Complexity Classes

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

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

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

Efficient algorithms