Chapter 3.

Algorithms

Part II: Complexity of Algorithms

Dept. of Software
Gachon University
Spring 2022

Contents

- Complexity of Algorithms
- The Growth of Functions
  - Big-O notation

Linear Search vs. Binary Search

- Which one is more effective (better) ?

Example : The steps taken by a binary search for 19 in the list:

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

target list : $a_1, a_2, \ldots, a_n$

procedure *linear search*
(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
  $i := 1$
  while ($i \le n \wedge x \ne a_i$)
    $i := i + 1$
  if $i \le n$ **then** *location* := i
  else *location* := 0
  return *location* {index or 0 if not found}

procedure *binary search*
(x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
  $i := 1$ {left endpoint of search interval}
  $j := n$ {right endpoint of search interval}
  while $i<j$ **begin** {while interval has >1 item}
    $m := \lfloor (i+j)/2 \rfloor$ {midpoint}
    if $x>a_m$ **then** $i := m+1$ **else** $j := m$
  end
  if $x = a_i$ **then** *location* := i **else** *location* := 0
  return *location*

Linear Search vs. Binary Search

- Obviously, on sorted sequences, binary search is more efficient than linear search.

- How can we analyze the efficiency of algorithms?

We can measure the

- **time** (number of elementary computations) and
- **space** (number of memory cells) that the algorithm requires.

These measures are called **computational complexity** and **space complexity**, respectively.

Complexity of Linear Search

- **What is the time complexity of the linear search algorithm?**

- We will determine the **worst-case** number of comparisons as a function of the number *n* (*n* elements) of terms in the sequence.

- The worst case : occurs when the element to be located is not included in the sequence.

- In that case, every item in the sequence is compared to the element to be located.

Complexity

- Comparison: time complexity of algorithms A and B
  - Let us assume these two solve the same class of problems.
  - for an input with n elements,
    the time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$.

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Quantifying the Growth of Functions

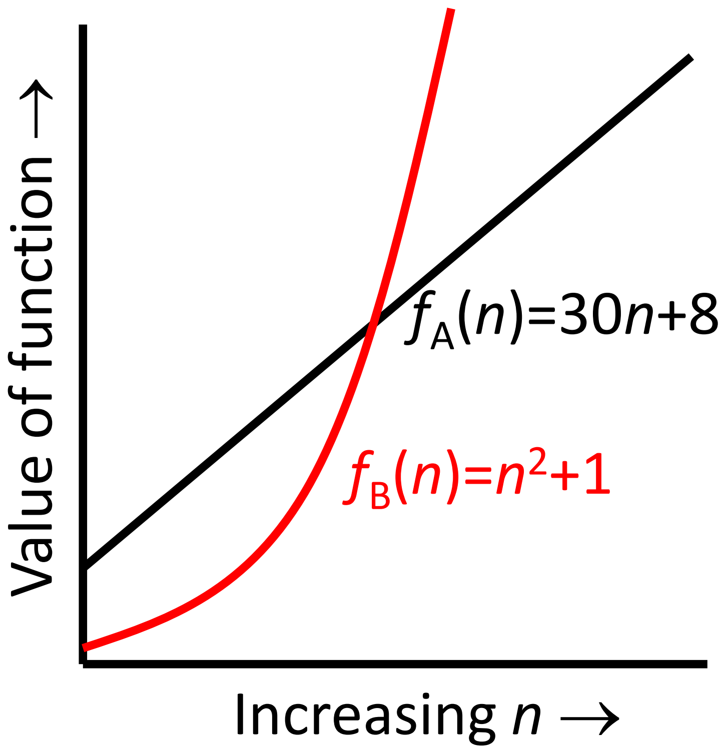
- This means that algorithm B cannot be used for large inputs, while running algorithm A is still feasible.

- So what is important is the **growth** of the complexity functions.

- The growth of time and space complexity with increasing input size n is a suitable measure for the **comparison** of algorithms.

Question

- Suppose you are designing a web site to process user data.
- Suppose database
  - program ***A*** takes $f_A(n)=30n+8$ msec to process any n records, while
  - program ***B*** takes $f_B(n)=n^2+1$ msec to process the n records.



- Which program do you choose, knowing you'll want to support millions of users?

The Growth of Functions

- For functions over numbers, we often need to know a rough measure of ***how fast a function grows***.
- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ *in the limit* (for large enough values of x).

The growth of functions is usually described using the **big-O notation**.

Big-O Notation O(g(x))

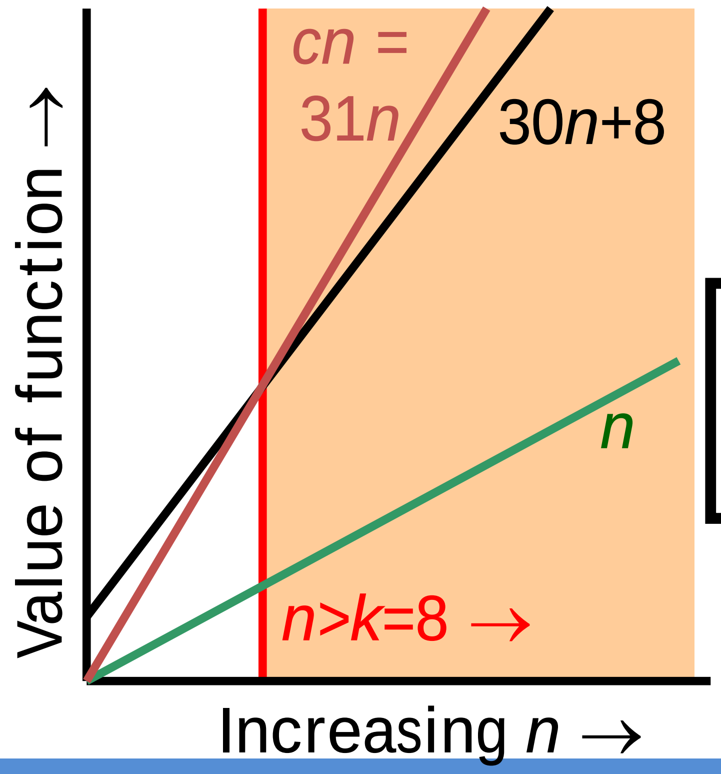
- **Definition O(g(x))**
  - f (x) is O(g(x)) if there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever x > k
  - $\{f : \mathbf{R} \rightarrow \mathbf{R} \mid \exists C,k: \forall x>k: f(x) \leq Cg(x)\}$.
  - "Beyond some point k, function f is at most a constant c times g (*i.e.*, proportional to g)."
- "*f* is *order g*", or "*f* is O(*g*)", or "*f*=O(*g*)" all just mean that $f \in O(g)$.
- Example
  - Show that $f(x)=x^2 + 2x + 1$ is $O(x^2)$
  - When $x > 1$, $f(x) \leq 4x^2$
    - $4x^2 - x^2 -2x -1 = 3x^2 -2x - 1 \geq 0$

"Big-O" Proof Example

- Show that $30n+8$ is $O(n)$.
  - Show $\exists C,k: \forall n>k: 30n+8 \le Cn$.
  - Let $C=31$, $k=8$. Assume $n>k(=8)$.
  - Then $Cn = 31n = 30n + n > 30n+8$, so $30n+8 < Cn$.

NOTE:

- Note that *f* is O(*g*) so long as *any* values of *c* and *k* exist that satisfy the definition.
- But: The particular *c*, *k*, values that make the statement true are *not* unique: **Any larger value of *c* and/or *k* will also work.**
- You are **not** required to find the smallest *c* and *k* values that work. (Indeed, in some cases, there may be no smallest values!)



$30n+8 \in O(n)$

Exercise

- Show that n^2+1 is $O(n^2)$.

Examples of Big-O (1/2)

- ***f*(*x*) = $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$**
  - $$\begin{aligned} f(x) &= a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \\ &\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \ldots + |a_1|x + |a_0| \\ &= x^n(|a_n| + |a_{n-1}|/x + \ldots + |a_1|/x^{n-1} + |a_0|/x^n) \\ &\leq x^n\underbrace{(|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|)}_{c} \\ &= cx^n = O(x^n) \end{aligned}$$

- $f(n) = 1 + 2 + 3 + \ldots + n$
  - $$\begin{aligned} f(n) &= 1 + 2 + 3 + \ldots + n \\ &\leq n + n + n + \ldots + n \\ &= n^2 = O(n^2) \end{aligned}$$

Examples of Big-O (2/2)

- $f(n) = n!$
  - $f(n) = n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

    $\leq n \cdot n \cdot n \cdot \ldots \cdot n \cdot n \cdot n$

    $= n^n = O(n^{\mathrm{n}})$

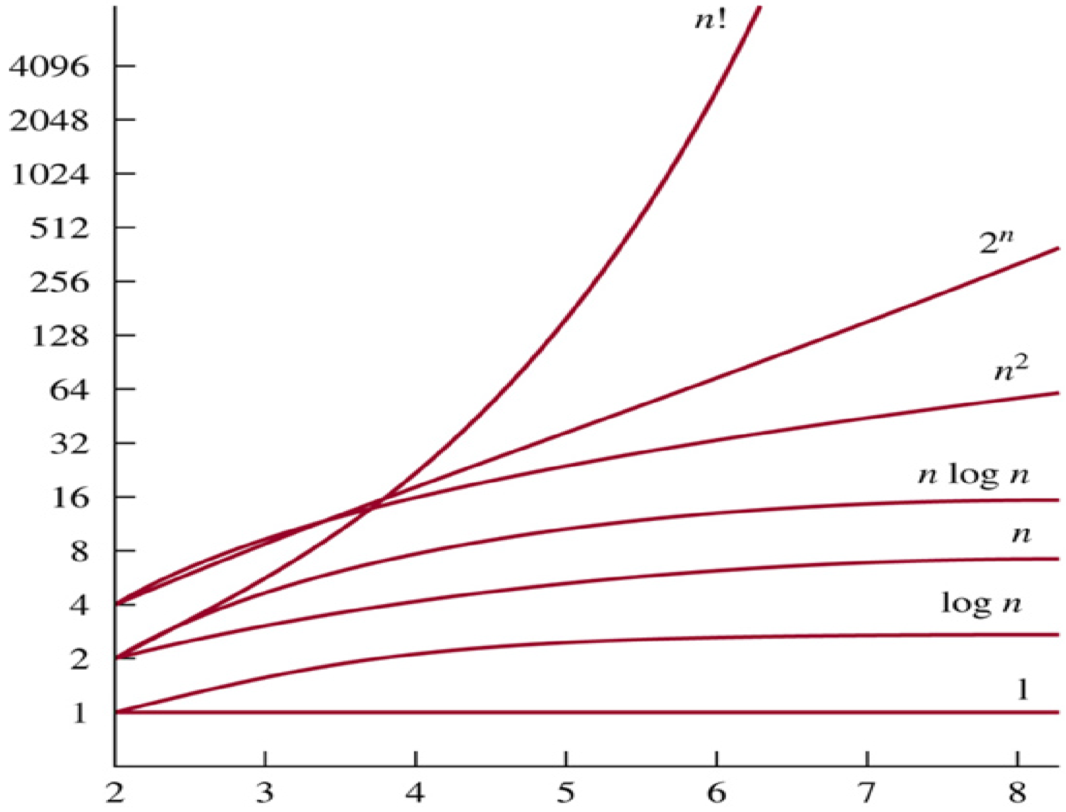
- $f(n) = \log(n!)$
  - $f(n) = \log(n!)$

    $\leq \log(n^n)$

    $= n\log n = O(n\log n)$

Some Important Big-O Functions

- Growth of Functions
  - 1, log*n*, *n*, ***n* log*n***, $\mathbf{n^2}$, 2^n, n!



Useful Facts about Big-O (1/2)

- Big O, as a relation, is transitive:

  $$f \in O(g) \wedge g \in O(h) \rightarrow f \in O(h)$$

- If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(\max(g_1, g_2))$
  - E.g., if $f_1(x) = x^2 = O(x^2)$, $f_2(x) = x^3 = O(x^3)$,
  - then $(f_1 + f_2)(x) = x^2 + x^3 = O(x^3) = O(\max(x^2, x^3))$

$$\begin{aligned} |f_1(x) + f_2(x)| &\le |f_1(x)| + |f_2(x)| \\ &\le C_1 |g_1(x)| + C_2 |g_2(x)| \\ &\le C_1 |g(x)| + C_2 |g(x)| \\ &= (C_1 + C_2)|g(x)| \end{aligned}$$

$g(x) = \max(g_1(x), g_2(x))$

Useful Facts about Big-O (2/2)

- f_1=O(g_1) and f_2 = O(g_2), then f_1f_2 = O(g_1g_2)
  - E.g., if $f_1(x) = x^2 = O(x^2)$, $f_2(x) = x^3 = O(x^3)$,

    then $(f1f2)(x) = x^2 \cdot x^3 = x^5 = O(x^5) = O(x^2 \cdot x^3))$

$$
\begin{aligned}
|(f_1 f_2)(x)| &= |f_1(x)||f_2(x)| \\
&\leq C_1|g_1(x)|C_2|g_2(x)| \\
&= C_1C_2|(g_1g_2)(x)|
\end{aligned}
$$

Big-Omega $\Omega(g)$ and Big Theta $\Theta(g)$

- Definition $\Omega(g)$
  - $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that $|f(x)| \geq C|g(x)|$ whenever $x > k$
  - $f(x)$ is big-Omega of $g(x)$
- Definition $\Theta(g)$
  - If $f(x)$ is $O(g(x))$ and $\Omega(g(x))$, $f(x)$ is $\Theta(g(x))$
  - $f(x)$ is big-Theta of $g(x)$ or of order $g(x)$

Big-Theta $\Theta(g)$, exactly order g

- $\Theta(g) \equiv \{f{:}\mathbf{R}\rightarrow\mathbf{R} \mid \exists c_1 c_2 k\ \forall x>k{:}\ |c_1 g(x)| \leq |f(x)| \leq |c_2 g(x)|\ \}$

- If $f\in O(g)$ and $g\in O(f)$ then we say *"g and f are of the same order"* or *"f is (exactly) order g"* and write $f\in\Theta(g)$.

Examples of Θ (1/2)

- $f(n) = 1 + 2 + 3 + \ldots + n = \Theta(n^2)$?
  - $f(n) = 1 + 2 + 3 + \ldots + n$

    $\leq n + n + n + \ldots + n$

    $= n^2$
  - $f(n) = 1 + 2 + 3 + \ldots + n$

    $= (n \cdot (n + 1))/2$

    $= n^2/2 + n/2$

    $\geq n^2/2$
  - $n^2/2 \leq f(n) \leq n^2$, i.e., $c_1 = ½$, $c_2 = 1$

  $\therefore \quad f(n) = \Theta(n^2)$

Examples of Θ (2/2)

- $f(y) = 3y^2 + 8y\log y = \Theta(y^2)$?
  - $f(n) \quad = 3y^2 + 8y\log y$
    $\quad\quad \le 11y^2 \quad (\text{if } y > 1) \quad (\text{since } 8y\log y \le 8y^2)$
  - $f(n) \quad = 3y^2 + 8y\log y$
    $\quad\quad \ge y^2 \quad\quad (\text{if } y > 1)$
  - $y^2 \le f(y) \le 11y^2$, i.e., $c_1 = 1$, $c_2 = 11$

  $\therefore \quad f(y) = \Theta(y^2)$

Computational Complexity

- Time complexity
  - Analysis of the time required to solve a problem of a particular size
  - # of operations or steps required

Our focus

- Space complexity
  - Analysis of the computer memory required to solve a problem of a particular size
  - # of memory bits required

Complexity **Depends on Input**

- Most algorithms have different complexities for inputs of different sizes. (*E.g.* searching a long list takes more time than searching a short one.)
- Therefore, complexity is usually expressed as a *function* of input length.
- This function usually gives the complexity for the *worst-case* input of any given length.

Example: *Max* Algorithm

procedure *max*($a_1, a_2, \ldots, a_n$: integers)

$v := a_1$ {largest element so far}

for $i := 2$ **to** n {go thru rest of elems}

if $a_i > v$ **then** $v := a_i$ {found bigger?}

{at this point *v*'s value is the same as the largest integer in the list}

return v

- **Problem**: Find the *exact* order of growth (Θ) of the *worst-case* time complexity of the *max* algorithm.
- Assume that each line of code takes some constant time every time it is executed.

Complexity Analysis of Max Algorithm (1/2)

procedure *max*($a_1, a_2, \ldots, a_n$: integers)	
$v := a_1$	t_1
for $i := 2$ **to** n	t_2
if $a_i > v$ **then** $v := a_i$	t_3
return v	t_4

n : input size

Times for *each* execution of each line.

- What's an expression for the *exact* total worst-case time? (Not its order of growth.)

Complexity Analysis of Max Algorithm (1/2)

procedure *max*($a_1, a_2, \ldots, a_n$: integers)	
$v := a_1$	t_1
for $i := 2$ **to** n	t_2
if $a_i > v$ **then** $v := a_i$	t_3
return v	t_4

n : input size

Times for *each* execution of each line.

- What's an expression for the *exact* total worst-case time? (Not its order of growth.)

Complexity Analysis of Max Algorithm (2/2)

procedure *max*($a_1, a_2, \ldots, a_n$: integers)	
$v := a_1$	t_1
for $i := 2$ **to** n	t_2
if $a_i > v$ **then** $v := a_i$	t_3
return v	t_4

n : input size

Times for *each* execution of each line.

- Worst case execution time:

$$t(n) = t_1 + \left(\sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

$$= \Theta(1) + \left(\sum_{i=2}^{n} \Theta(1) \right) + \Theta(1) = \Theta(1) + (n-1)\Theta(1)$$

$$= \Theta(1) + \Theta(n)\Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$$

Example: Linear Search

procedure *linear search* x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$i := 1$	t_1
while $(i \le n \wedge x \ne a_i)$	t_2
$\quad i := i + 1$	t_3
if $i \le n$ **then** *location* $:= i$	t_4
else *location* $:= 0$	t_5
return *location*	t_6

- Worst case: $t(n) = t_1 + \left(\sum_{i=1}^{n} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$
- Best case: $t(n) = t_1 + t_2 + t_4 + t_6 = \Theta(1)$
- Average case (if item is present):

$$t(n) = t_1 + \left(\sum_{i=1}^{n/2} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

Example: Binary Search

procedure *binary search* (x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)

$\left.\begin{array}{l} i := 1 \\ j := n \end{array}\right\} \Theta(1)$

while $i<j$ **begin**

$\left.\begin{array}{l} m := \lfloor (i+j)/2 \rfloor \\ \textbf{if } x>a_m \textbf{ then } i := m+1 \textbf{ else } j := m \end{array}\right\} \Theta(1)$

end

$\left.\begin{array}{l} \textbf{if } x = a_i \textbf{ then } location := i \textbf{ else } location := 0 \\ \textbf{return } location \end{array}\right\} \Theta(1)$

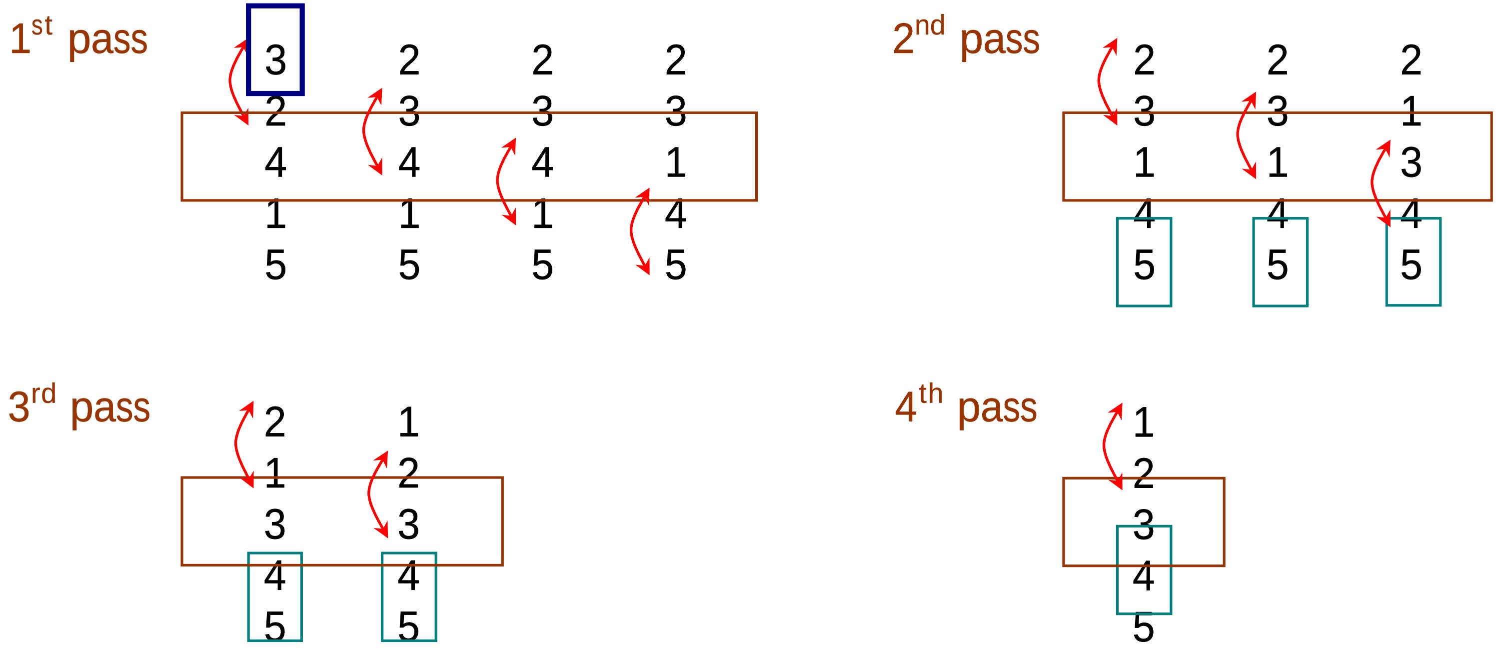
- Key Question: How Many Loop Iterations?

Binary Search Analysis

- Suppose $n=2^k$.
- Original range from $i=1$ to $j=n$ contains n elements.
- Each iteration: Size $j-i+1$ of range is cut in half.
  - $(2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow \ldots \rightarrow 2^1 \rightarrow 2^0$, number of iteration = $\boldsymbol{k})$
  - Loop terminates when size of range is $1=2^0$ $(i=j)$.
- Therefore, number of iterations is $k = \log_2 n$

  $= \Theta(\log_2 n) = \Theta(\log n)$
- Even for $n \neq 2^k$ (not an integral power of 2), time complexity is still $\Theta(\log_2 n) = \Theta(\log n)$.

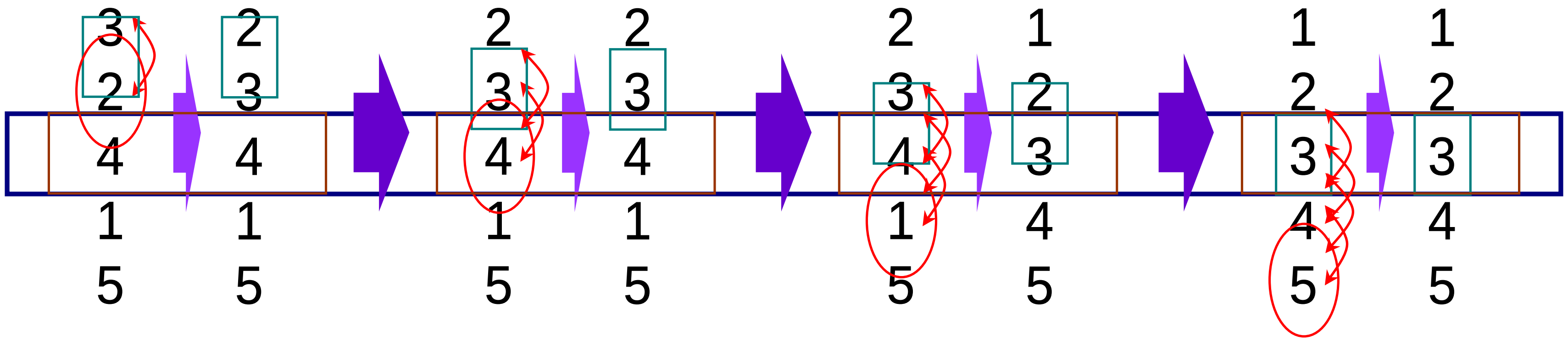
Example: Bubble Sort



- Consider # of compare operations only!

$(n\text{-}1) + (n\text{-}2) + \ldots + 2 + 1 = ((n\text{-}1)n)/2 = \Theta(n^2)$

Example: Insertion Sort



- Also, consider # of compare operations only!

  $1 + 2 + \ldots + (n\text{-}2) + (n\text{-}1) = ((n\text{-}1)n)/2 = \Theta(n^2)$

- Then, are all sorting algorithm's complexities $\Theta(n^2)$?

  NO! …, merge sort, heap sort, quick sort, …

Understanding the Complexity of Algorithms

- Names for Some Orders of Growth

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Understanding the Complexity of Algorithms

TABLE 2 **The Computer Time Used by Algorithms.**

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

Times of more than 10^{100} years are indicated with an *.

Polynomial Time Complexity

- *Tractable Problem*: A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*). **P** is the set of all tractable problems.
- *Intractable Problem*: A problem or algorithm that has more than polynomial complexity is considered *intractable* (or *infeasible*).
- *Unsolvable Problem* : No algorithm exists to solve this problem, e.g., halting problem.

Cont.

- *Class NP*: Solution can be checked in polynomial time. But no polynomial time algorithm has been found for finding a solution to problems in this class.

- *NP-Complete Class*: If you find a polynomial time algorithm for one member of the class, it can be used to solve all the problems in the class.

Section Summary

- Time Complexity
- Worst-Case Complexity
- Algorithmic Paradigms
- Understanding the Complexity of Algorithms