

2. Algorithm Design and Analysis

Goals

- Understand algorithm design methodologies
- Learn how to analyze algorithms
- Learn asymptotic notations

What is Algorithm?

- Tool to solve a well-defined computational problem.
- Computation problem
 - Can be defined by input and output.
- Algorithm is a sequence of computational steps that transform input into output.



Example of Problem

- Problem
 - Sorting
- Input
 - A sequence of numbers (e.g., 25, 17, 52, 36, 11)
- Output
 - Permutation of input numbers in non-decreasing order (11, 17, 25, 36, 52)

Why Study Algorithms

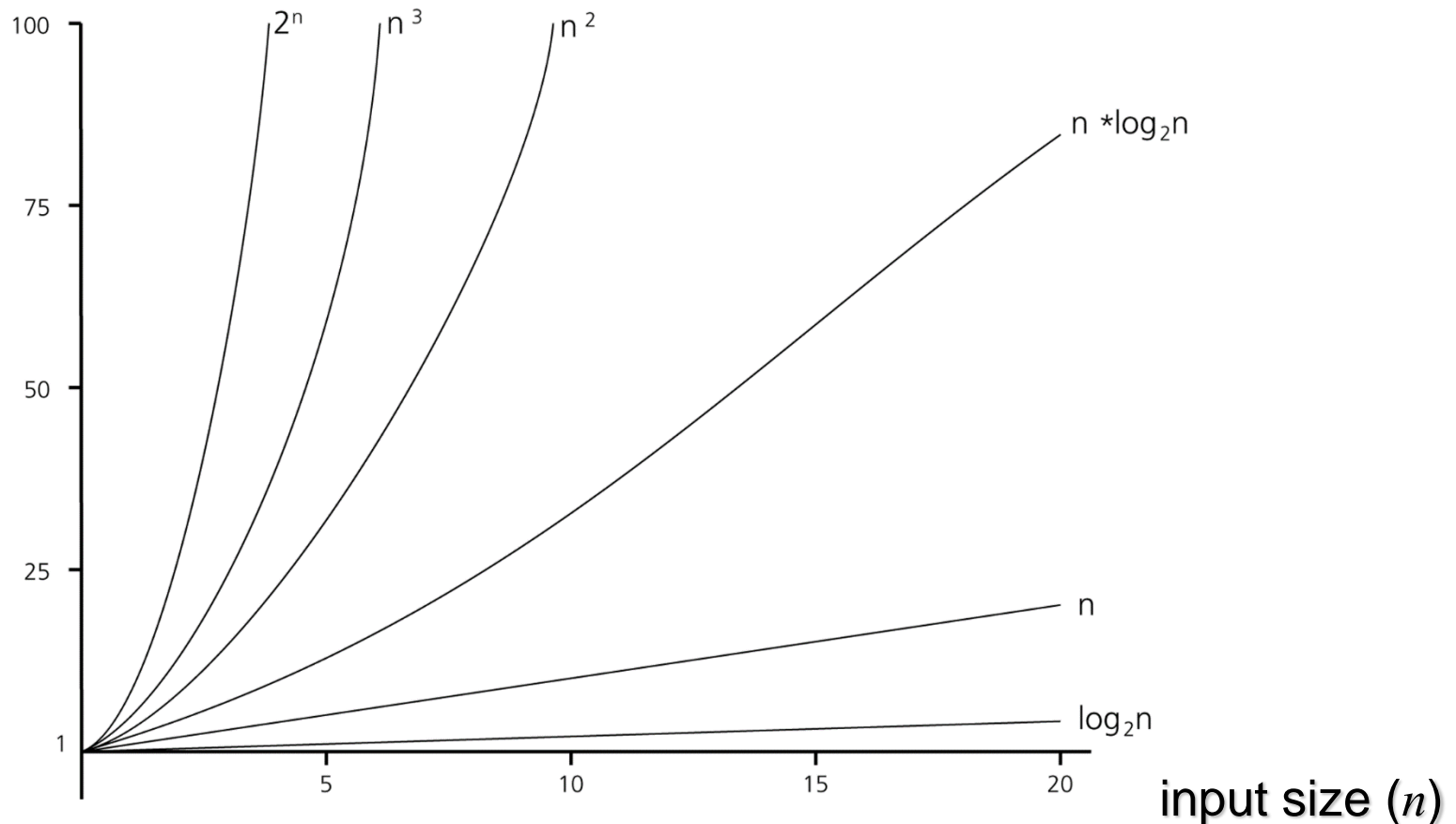
- Learning algorithms for specific problems
- Training algorithmic (procedural) thinking
- Leveling up abstraction
 - Intellectual abstraction
 - Important element to tackle complex and hard problems in research and development

Good Algorithm

- Be clear
 - Easy to understand, simple if possible
 - Too much mathematical notation may decrease clarity
 - Use words if they are clear enough.
- Be efficient
 - There can be big differences in running time of algorithms for the same problem.

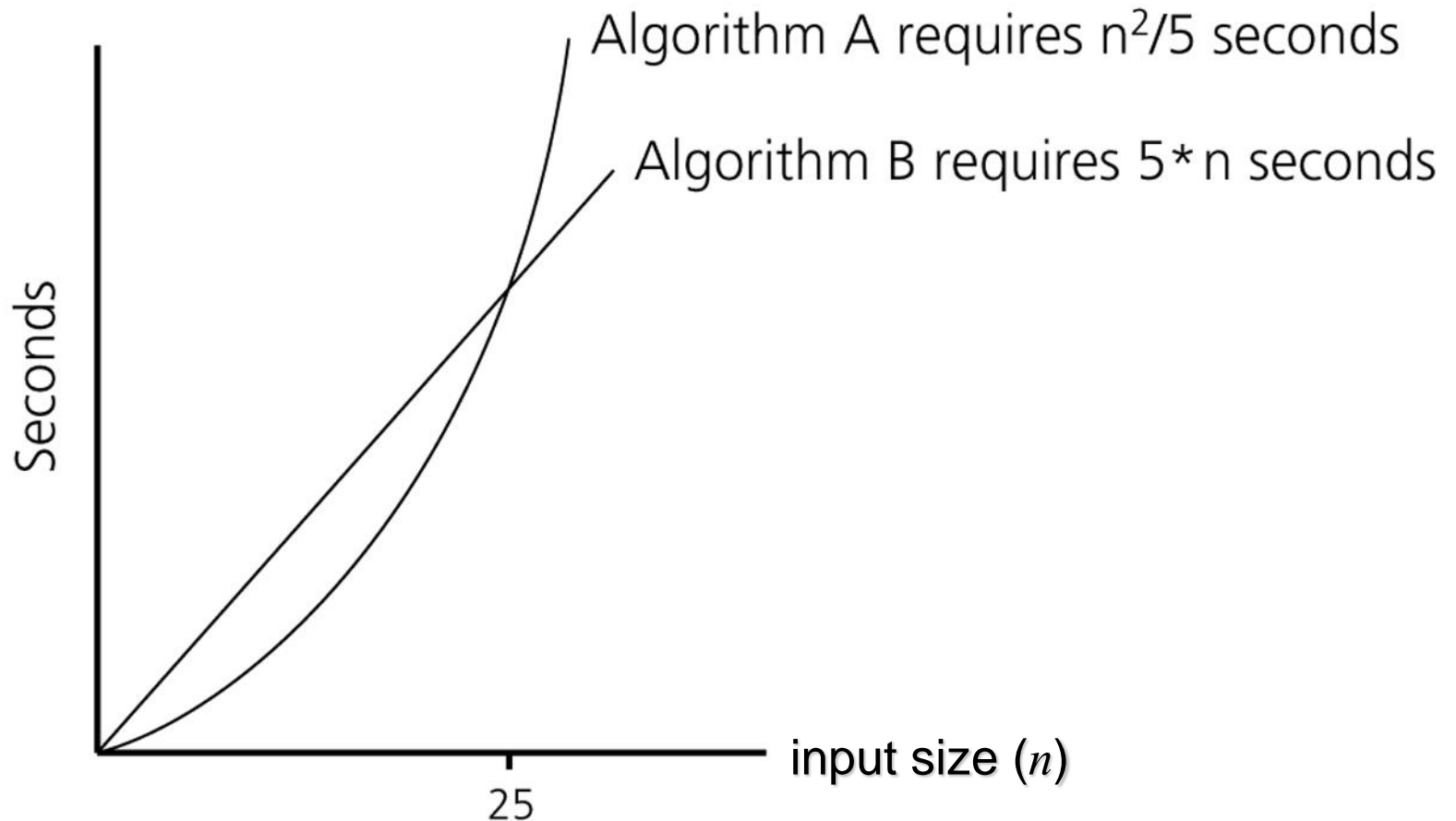
Running Time of Algorithms

running time



Running Time of Algorithms

running time



Running Time of Algorithms

(a)

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Logarithmic time: $(\log n)^c$

Polynomial time: n^c

Exponential time: 2^{n^c} , $n!$

Running Time of Algorithms

- How to analyze algorithms
- We start with simple examples.

Running Time of Algorithms

```
sample1(A[ ], n)
{
    k =  $\lfloor n/2 \rfloor$ ;
    return A[k];
}
```

- ✓ It takes constant time, irrespective of n .

Running Time of Algorithms

```
sample2(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        sum ← sum + A[i] ;
    return sum ;
}
```

✓ It takes time proportional to n .

Running Time of Algorithms

```
sample3(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        for j ← 1 to n
            sum ← sum + A[i]*A[j] ;
    return sum ;
}
```

✓ It takes time proportional to n^2 .

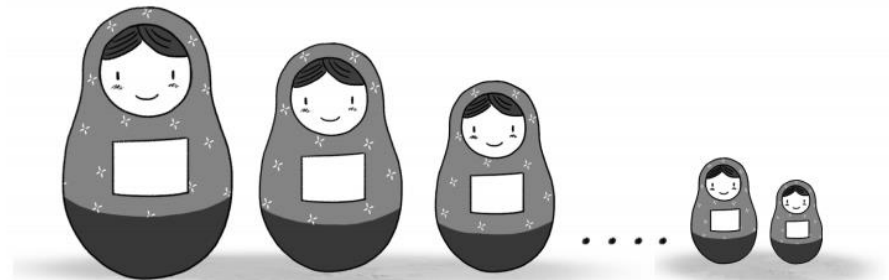
Running Time of Algorithms

```
factorial( $n$ )  
{  
    if ( $n=1$ ) return 1 ;  
    return  $n$ *factorial( $n-1$ ) ;  
}
```

✓ It takes time proportional to n .

Recursion and Inductive Thinking

- Recursive structure
 - Inside a problem, there exists a subproblem (i.e., identical problem with a smaller size)
 - Ex 1: factorial
 - $N! = N \times (N-1)!$
 - Ex 2: recurrence
 - $a_n = a_{n-1} + 2$



Merge Sort

(Divide-and-Conquer)

```
mergeSort(A[ ], p, r)    ▷ sort A[p ... r]
{
    if (p < r) then {
        q ← ⌊(p + r)/2⌋; ----- ①    ▷ mid point
        mergeSort(A, p, q); ----- ②    ▷ sort left half
        mergeSort(A, q+1, r); ----- ③    ▷ sort right half
        merge(A, p, q, r); ----- ④    ▷ merge
    }
}
```

```
merge(A[ ], p, q, r)
{
    Combine two sorted arrays A[p ... q] and A[q+1 ... r]
    into one sorted array A[p ... r].
}
```

```

mergeSort(A[ ], p, r)    ▷ sort A[p ... r]
{
    if (p < r) then {
        q ← ⌊(p + r)/2⌋; ----- ①    ▷ mid point
        mergeSort(A, p, q); ----- ②    ▷ sort left half
        mergeSort(A, q+1, r); ----- ③    ▷ sort right half
        merge(A, p, q, r); ----- ④    ▷ merge
    }
}

```

✓ ②, ③: recursive calls

✓ ①, ④: computations specific to mergesort

Applications of Algorithms

- Car navigation
 - Shortest path
- Scheduling
 - Traveling salesman problem, job scheduling, ...
- Human Genome Project
 - Sequence matching, functional analyses, ...
- Search
 - database, web search, ...
- VLSI design
 - Partitioning, placement, routing, ...
- ...

Why Analyze Algorithm

- Correctness
- Efficiency (complexity)
 - resources
 - **time**
 - Memory (space), communication cost, ...

Correctness

- Insertion Sort (Incremental Approach)

```
for j = 2 to n
    key = A[j]
    // insert A[j] into sorted A[1..j-1]
    i = j-1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

Loop Invariant

- At the start of each iteration of the for loop, $A[1..j-1]$ consists of elements originally in $A[1..j-1]$, but in sorted order.
- Initially: it holds when $j=2$.
- Maintain: Assume the invariant holds at the start of an iteration. Show that the invariant holds at the start of the next iteration.
- Terminate: when loop terminates (i.e., $j=n+1$), obtain correctness from the invariant.

Complexity Analysis:

Random-Access Machine Model

- CPU
 - Instructions: add, subtract, multiply, divide, load, store, conditional branch, unconditional branch, subroutine call, return, etc.
 - Unit of time: instruction
- Memory
 - A collection of words
 - Each word can store character, integer, floating number
 - Unit of space: word
 - (memory hierarchy not modeled)

Complexity Analysis

INSERTION-SORT(A)	<i>cost</i>	<i>times</i>
1 for $j = 2$ to $A.length$	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i + 1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] = key$	c_8	$n - 1$

t_j : number of times line 5 is executed

Complexity Analysis

- $T(n) = c_1n + c_2(n - 1) + c_4(n - 1)$
 $+ c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1)$
 $+ c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1)$
- Worst case: $t_j = j$
- Average case: $t_j = j/2$

Complexity Analysis

- $$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) \\ &\quad + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n-1)}{2} \right) \\ &\quad + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \\ &= an^2 + bn + c, \text{ where } a > 0 \end{aligned}$$

Algorithm Analysis

- When the problem size is small
 - Efficiency of algorithm is not so important
 - Inefficient and simple algorithm works fine
- When the problem size is big
 - Efficiency of algorithm matters
 - Using inefficient algorithm is disastrous
- **Asymptotic analysis** deals with efficiency of algorithm, as the input size increases.

Asymptotic Analysis

- Efficiency of algorithm, as the input size increases
- Asymptotic notation we know

$$\lim_{n \rightarrow \infty} f(n)$$

- O , Ω , Θ , ω , o notations

Asymptotic Notations

$O(g(n))$

- Asymptotically less than or equal to
- e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, ...

- Formal definition

- $O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ such that } \forall n \geq n_0, cg(n) \geq f(n) \}$
- $f(n) \in O(g(n))$: we write $f(n) = O(g(n))$ conventionally

- Meaning

- $f(n) = O(g(n)) \Rightarrow f$ asymptotically less than or equal to g
- Take the largest term; ignore the constant in that term

Asymptotic Notations

- For example, $O(n^2)$
 - $3n^2 + 2n$
 - $7n^2 - 100n$
 - $n \log n + 5n$
 - $an^2 + bn + c$ ($a > 0$)
- as tight as possible
 - $n \log n + 5n = O(n \log n)$ rather than $O(n^2)$

Asymptotic Notations

$\Omega(g(n))$

- Asymptotically greater than or equal to
- Symmetric to $O(g(n))$

- Formal definition

- $\Omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ such that } \forall n \geq n_0, cg(n) \leq f(n) \}$

- Meaning

- $f(n) = \Omega(g(n)) \Rightarrow f$ asymptotically greater than or equal to g

Asymptotic Notations

$\Theta(g(n))$

– Asymptotically equal to

- Formal definition

– $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

- Meaning

– $f(n) = \Theta(g(n)) \Rightarrow f$ is asymptotically equal to g

Asymptotic Notations

$o(g(n))$

– Asymptotically less than

- Formal definition

- $o(g(n)) = \{f(n) \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0\}$

- Meaning

- $f(n) = o(g(n)) \Rightarrow f$ is asymptotically less than g

Asymptotic Notations

$\omega(g(n))$

– Asymptotically greater than

- Formal definition

– $\omega(g(n)) = \{ f(n) \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \}$

- Meaning

– $f(n) = \omega(g(n)) \Rightarrow f$ is asymptotically greater than g

Asymptotic Notations

- $O(g(n))$
 - **Tight** or **loose** upper bound
- $\Omega(g(n))$
 - **Tight** or **loose** lower bound
- $\Theta(g(n))$
 - **Tight** bound
- $o(g(n))$
 - **Loose** upper bound
- $\omega(g(n))$
 - **Loose** lower bound


Analyses of Time Complexity

- **Worst-case**
 - Analysis for the worst-case input(s)
 - Count the parts which are executed most times for worst-case input (insertion sort)
- **Average-case**
 - Analysis for all inputs
 - More difficult to analyze
- **Best-case**
 - Analysis for the best-case input(s)
 - Not useful

Asymptotic Analysis

- Time complexity of sorting algorithms
 - Bubble sort
 - $\Theta(n^2)$
 - Heap sort (Algorithm Design using Data Structure)
 - $O(n \log n)$
 - Quicksort
 - Worst-case $O(n^2)$
 - Average-case $\Theta(n \log n)$

Time Complexity of Search

- 
- Array
 - $O(n)$
 - Binary search trees
 - Worst-case $\Theta(n)$
 - Average-case $\Theta(\log n)$
 - Balanced binary search trees
 - Worst-case $\Theta(\log_2 n)$
 - B-trees
 - Worst-case $\Theta(\log_B n)$
 - Hash table
 - Average-case $\Theta(1)$

Search in Array

- Sequential search
 - When elements are stored in arbitrary order
 - Worst case: $\Theta(n)$
 - Average case: $\Theta(n)$
- Binary search
 - When elements are stored in sorted order
 - Worst case: $\Theta(\log n)$
 - Average case: $\Theta(\log n)$

Algorithm Design Methodologies

- Divide-and-Conquer
- Incremental Approach
- Using Data Structure
- Dynamic Programming
- Greedy Approach
- ...



Thank you
