ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
VIETNAM - KOREA UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY
한-베정보통신기술대학교

Nhân bản – Phụng sự – Khai phóng

Algorithm Analysis

Data Structures & Algorithms

CONTENTS

- **Introduction to algorithm**
- **Algorithm analysis**
- **Estimating running time**
- **Algorithm growth rates** (Big O, Omega, Theta)
- **Worst-Case, Best-Case, Average-Case**

CONTENTS

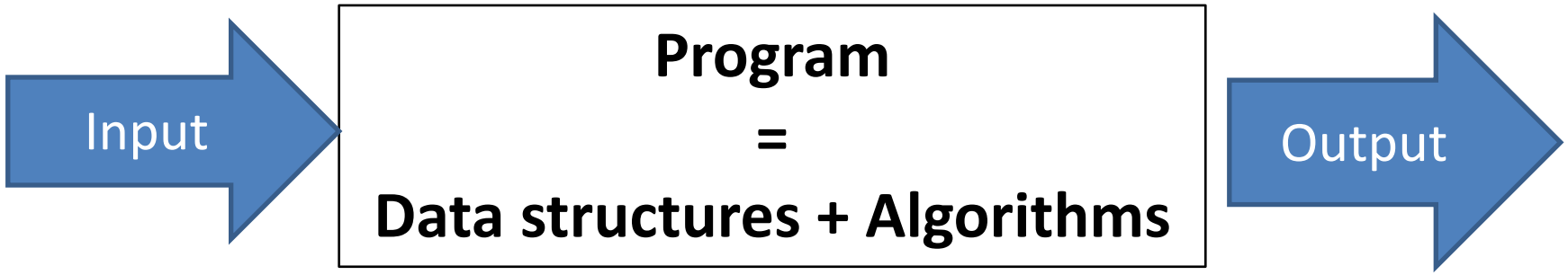
- **Introduction to algorithm**
- Algorithm analysis
- Estimating running time
- Algorithm growth rates (Big O, Omega, Theta)
- Worst-Case, Best-Case, Average-Case

- **An algorithm is a sequence of instructions to be followed to solve a problem**
  - There are often many solutions/algorithms to solve a given problem
  - An algorithm can be implemented using different programming languages on different platforms

- **An algorithm must be correct. It should correctly solve the problem**

- **Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm**

- **Program = Data structures + Algorithms**



- **Correctness**
  - An algorithm is said to be **correct** if for every input instance, it halts with the correct output.
- **Efficiency**
  - **Computing time** and **memory space** are two important resources.

- **Time**
  - Instructions take time
  - How fast does the algorithm perform?
  - What affects its running time?
- **Space**
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the running time?

⇨ Focusing on **running time**

  - How to estimate the time required for an algorithm?
  - How to reduce the required time?

CONTENTS

- Introduction to algorithm
- **Algorithm analysis**
- Estimating running time
- Algorithm growth rates (Big O, Omega, Theta)
- Worst-Case, Best-Case, Average-Case

- **Why do we need algorithm analysis?**
  - Showing the algorithm is correct
  - Writing a working program is not good enough
  - The program may be inefficient
  - If the program is run on a **large data set**, then the running time becomes an issue

- **Example: Selection Problem (1/3)**
  - Given a list of N numbers, determine the ***k***th largest, where **k** $\leq$ **N**.
  - Algorithm 1
    (1) Read **N** numbers into an array
    (2) Sort the array in decreasing order by some simple algorithm
    (3) Return the element in position ***k***

- **Example: Selection Problem (2/3)**
  - Algorithm 2
    (1) Read the first ***k*** elements into an array and sort them in decreasing order
    (2) Each remaining element is read one by one
      – If smaller than the kth element, then it is ignored
      – Otherwise, it is placed in its correct position in the array, getting one element out of the array
    (3) The element in the ***k***th position is returned as the answer

- **Example: Selection Problem (3/3)**
  - Which algorithm is better when

    1. **N** = 100 and ***k*** = 100?

    2. **N** = 100 and ***k*** = 1?

- **Factors affecting the running time**
  - computer
  - compiler
  - algorithm
  - input to the algorithm
    - The content of the input affects the running time
    - Typically, the **input size** (number of items in the input) is the main consideration
      - E.g. sorting problem $\Rightarrow$ the number of items to be sorted
      - E.g. multiply 2 matrices together $\Rightarrow$ the total number of elements in the 2 matrices

- **Analyzing algorithms**
  - Employing mathematical techniques that analyze algorithms independently of specific compilers, computers.

- **To analyze algorithms**
  1. Starting to count the number of **significant operations** in a particular solution to assess its efficiency
  2. Expressing the efficiency of algorithms using **growth functions**

CONTENTS

- Introduction to algorithm
- Algorithm analysis
- **Estimating running time**
- Algorithm growth rates (Big O, Omega, Theta)
- Worst-Case, Best-Case, Average-Case

Estimating running time

- Each operation in algorithm/program has a cost

⇨ Each operation takes a certain of running time

Ex: count = count + 1;

⇨ takes a certain amount of time, but it is constant

A sequence of operations:

```
count = count + 1;          //cost: c1
sum = sum + count;          //cost: c2
```

⇨ **Total Cost = c1 + c2**

- **Ex: Simple If-Statement**

	Cost	Times
if (n < 0)	c1	1
absval = -n	c2	1
else		
absval = n;	c3	1

⇨ **Total Cost <= c1 + max(c2,c3)**

- **Ex: Simple Loop**

```
                                    Cost        Times
i = 1;                              c1          1
sum = 0;                            c2          1
while (i <= n) {                    c3          n+1
        i = i + 1;                  c4          n
        sum = sum + i;              c5          n
}
```

⇨ **Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5**

⇨ **The time required for this algorithm is proportional to n**

– When n tends to infinity

- **Ex: Nested Loop**

```
                                        Cost        Times
i=1;                                    c1          1
sum = 0;                                c2          1
while (i <= n) {                        c3          n+1
        j=1;                            c4          n
        while (j <= n) {                c5          n*(n+1)
                sum = sum + i;          c6          n*n
                j = j + 1;              c7          n*n
        }
        i = i + 1;                      c8          n
}
```

⇨ **Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8**

⇨ **The time required for this algorithm is proportional to n^2**

– When n tends to infinity

- **General rules for running time estimation**
  - **Consecutive Statements**: Just **add the running times** of those consecutive statements
  - **Conditional Statements (If/Else)**: Never more than the running time of the test plus **the larger of running times** of two branches
  - **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times **the number of iterations**
  - **Recursion**: Determine and solve the recurrence relation (we don't focus on this case in this course)

- **Function calls**
  - Non recursive calls
    - A function call is considered as a statement
      ⇨ The running time of a function call is considered as the running time of a statement (that function)
  - Recursive calls
    - Set up the recurrence relation
    - Solve the recurrence
      - May be very complicated

- **Example**

```
sum = 0
for (j=0; j<n; j++)
        sum++;
```

T(n) = n

- **Example**

```
sum = 0
for (j=0; j<n; j++)
    for (k=0; k<n; k++)
        sum++;
```

$T(n) = n^2$

- **Example**

$T(n) = n^3$

```
sum = 0
for (j=0; j<n; j++)
   for (k=0; k<n*n; k++)
      sum++;
```

- **Example**

$T(n) = n^4$

```
sum = 0;
for (j=0; j<n; j++)
   for (k=0; k<j*j; k++)
      if (k%j == 0)
            for (m=0; m<k; m++)
                  sum++;
```

- **Example**

```
int fact(int n){
        if (n==0)
                return 1;
        else
                return (n * fact(n-1));
  }
```

Recurrence relation

$$T(n) = T(n\text{-}1) + 1,\ T(0) = 0$$

$$T(n) = n$$

CONTENTS

- Introduction to algorithm
- Algorithm analysis
- Estimating running time
- **Algorithm growth rates (Big O, Omega, Theta)**
- Worst-Case, Best-Case, Average-Case

Algorithm growth rates

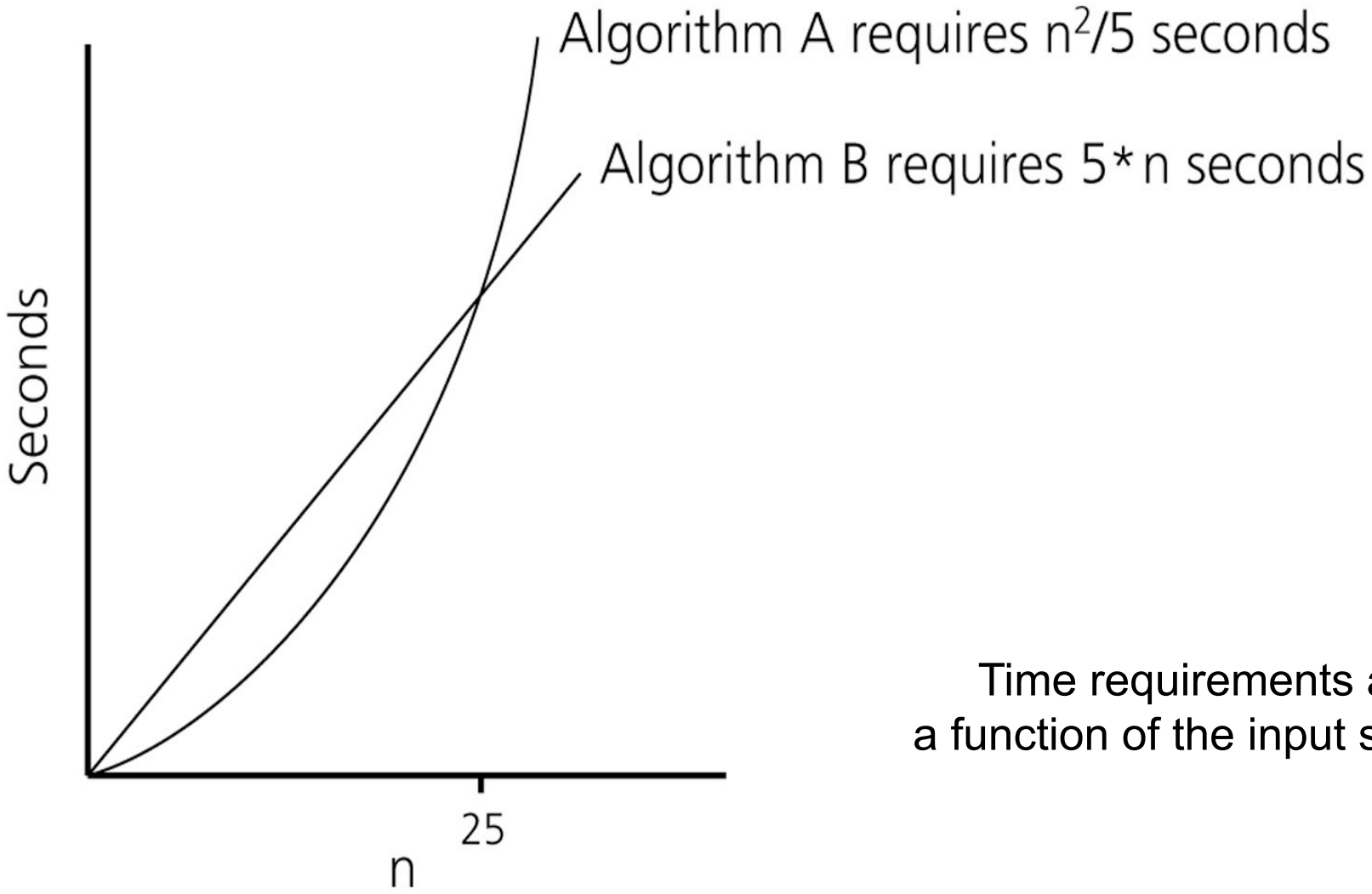
- **Measuring an algorithm's time requirement as a function of the input size**
- **Input size depends on the problem**

  Ex: number of elements in a list for a sorting algorithm

  If the input size is n:

  - Algorithm A requires **$5*n^2$** time units to solve a problem of input size **n**
  - Algorithm B requires **7*n** time units to solve a problem of input size **n**
- **The algorithm's time requirement grows as a function of the input size**
  - Algorithm A requires time **proportional** to **n^2**
  - Algorithm B requires time **proportional** to **n**
- **An algorithm's proportional time requirement is known as growth rate**
- **Comparing the efficiency of two algorithms by comparing their growth rates**

...Algorithm growth rates

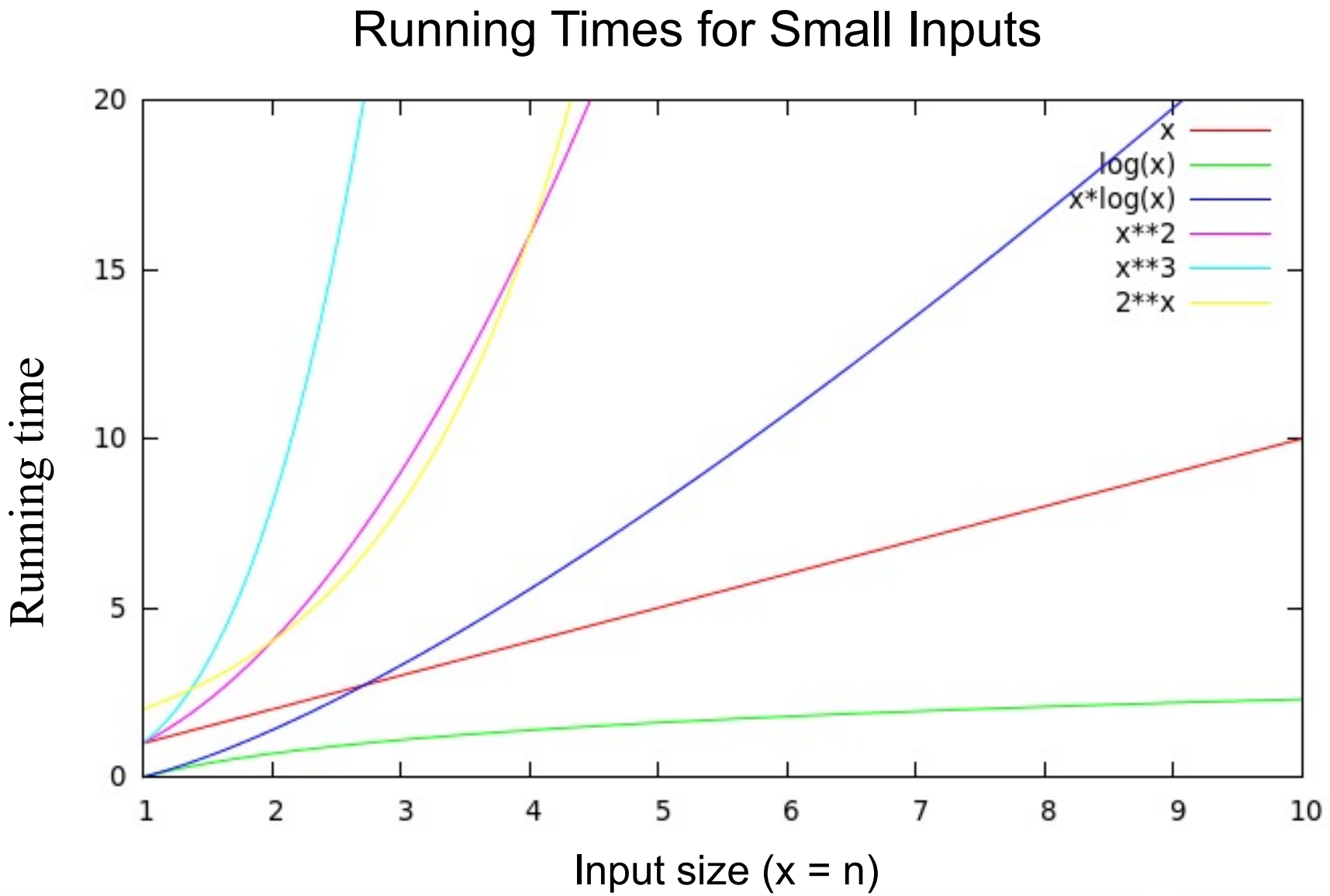


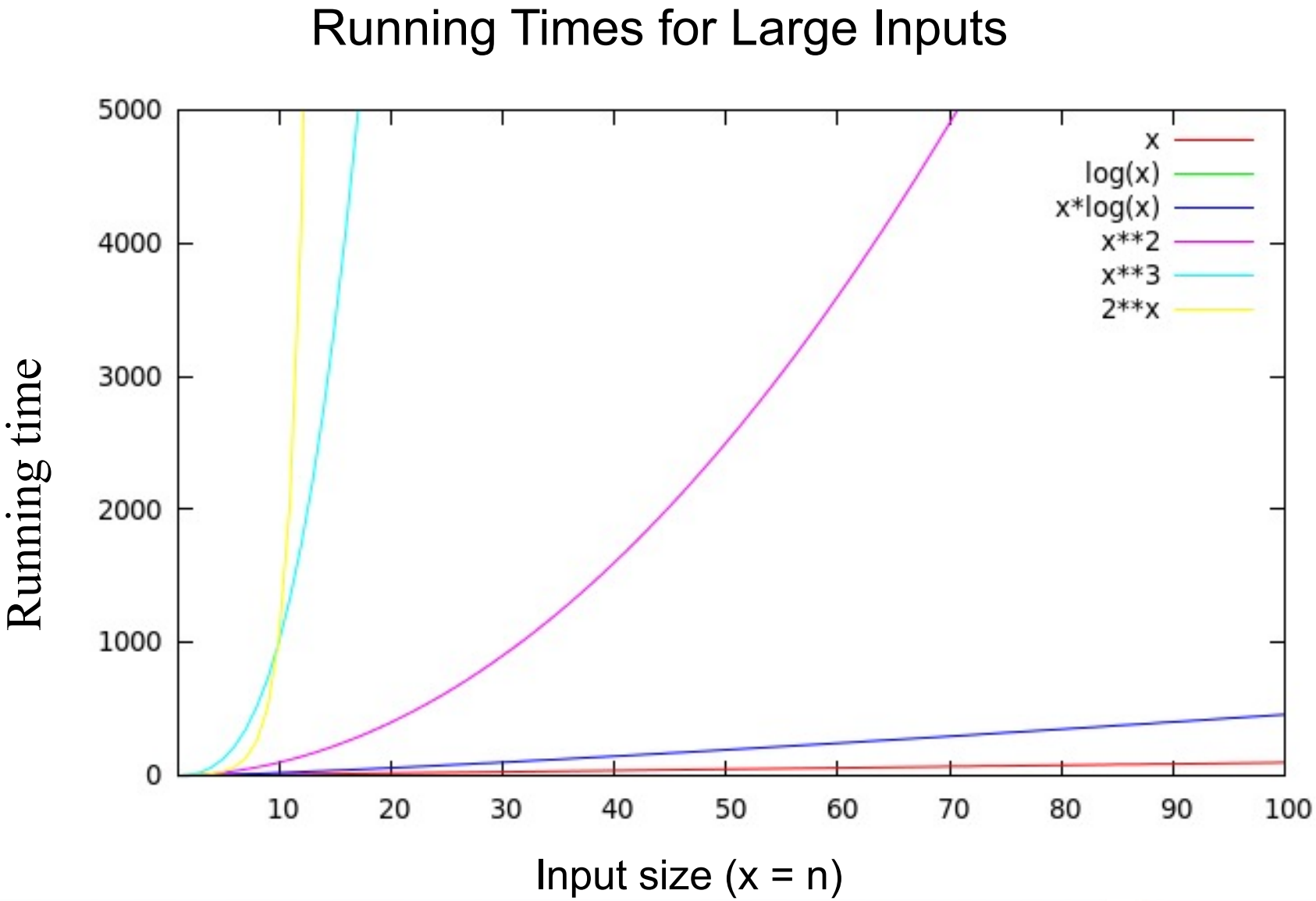
Time requirements as
a function of the input size **n**

- **Common Growth Rates**

Function	Growth Rate Name
c	Constant
log N	Logarithmic
$\log^2 N$	Log-squared
N	Linear
N log N	Log-linear
N^2	Quadratic
N^3	Cubic
2^N	Exponential

...Algorithm growth rates




Running Times for Large Inputs
Running time
Input size (x = n)
5000
4000
3000
2000
1000
0
10
20
30
40
50
60
70
80
90
100
x
log(x)
x*log(x)
x**2
x**3
2**x

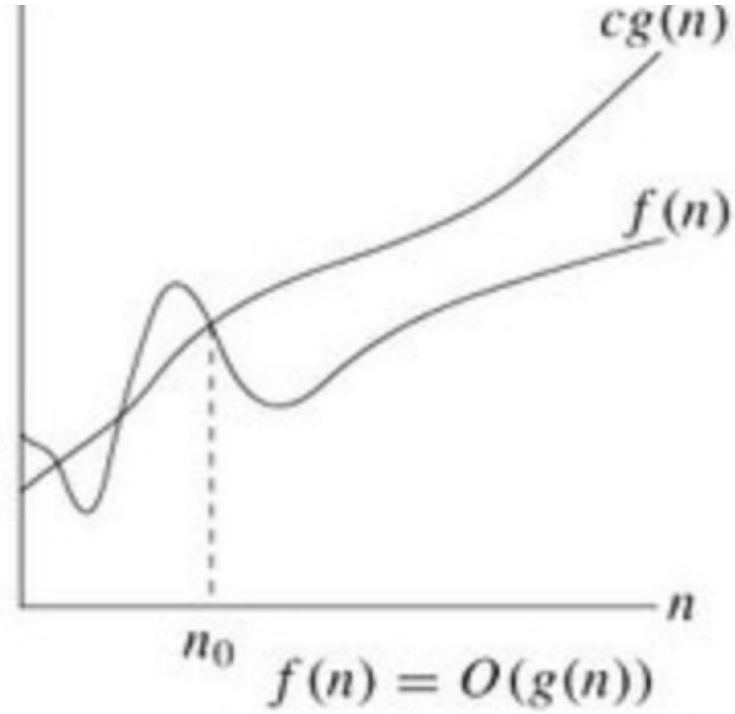
…Algorithm growth rates

- **Asymptotic notations / growth-rate functions**
  - Upper bound O(g(n))
  - Lower bound Ω(g(n))
  - Tight bound Θ(g(n))

- **Big O**
  - f(n) = O(g(n))
  - There are positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \quad \text{when} \quad n \geq n_0$$

  - growth rate of f(n) $\leq$ growth rate of g(n)
  - g(n) is an *upper bound* on f(n)



- **Big O**
  - If **Algorithm A requires time proportional to g(n)**, Algorithm A is said to be **order g(n),** and it is denoted as **O(g(n))**.
  - The function **g(n)** is called the algorithm's **growth-rate function**.
  - The capital O is used in the notation
    ⇨ called the **Big O notation.**
  - If Algorithm A requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$.
  - If Algorithm A requires time proportional to **n**, it is **O(n)**.

- **Big O – Example**
  - Let $f(n) = 2n^2$. Then
    - $f(n) = O(n^4)$
    - $f(n) = O(n^3)$
    - $f(n) = O(n^2)$ (best answer, asymptotically tight)
  - $O(n^2)$: reads "**order n-squared**" or "**Big-O n-squared**"

- **Big O – Some rules**
  - Ignore the lower order terms
  - Ignore the coefficients of the highest-order term
  - If T(n) is an asymptotically positive polynomial of degree **k**, then $T(n) = O(n^k)$

Ex: $\mathbf{7n^2 + 10n + 3 = O(n^2)}$

- **Big O – Some rules**
  - No need to specify the base of logarithm
    - Changing the base from one constant to another changes the value of the logarithm by only a constant factor
  - For logarithmic functions,

    $T(\log_m n) = O(\log n)$, (use: $T(\log_m n) = T((\log_2 n) / (\log_2 m))$)

  - If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$,
    - $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$
    - $T_1(n) * T_2(n) = O(f(n) * g(n))$

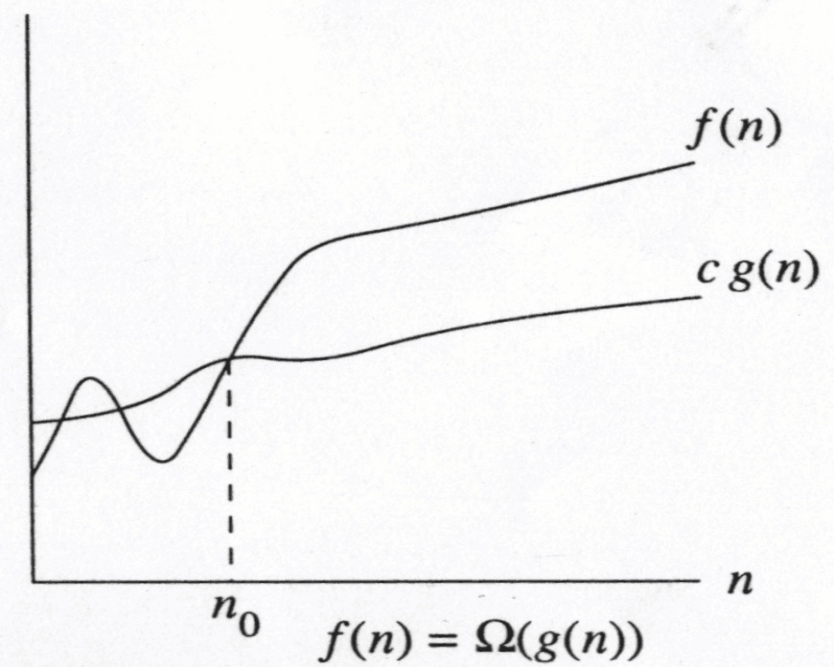
- **Big O – more example**
  - $n^2 / 2 – 3n = O(n^2)$
  - $1 + 4n = O(n)$
  - $7n^2 + 10n + 3 = O(n^2) = O(n^3)$
  - $\log_{10} n = \log_2 n / \log_2 10 = O(\log_2 n) = O(\log n)$
  - $10 = O(1), \quad 10^{10} = O(1)$
  - $\log n + n = O(n)$

$$\sum_{i=1}^{N} i \leq N \cdot N = O(N^2) \qquad \sum_{i=1}^{N} i^2 \leq N \cdot N^2 = O(N^3)$$

- **Big Omega**
  - f(n) = Ω(g(n))
  - There are positive constants c and n_0 such that

    $f(n) \geq c\, g(n)$ when $n \geq n_0$

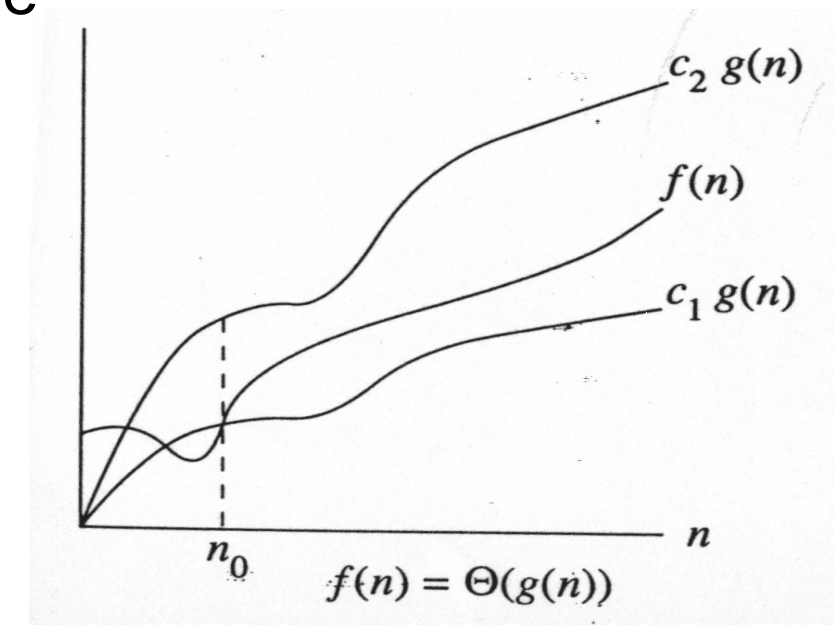
  - growth rate of f(n) ≥ growth rate of g(n)
  - g(n) is a **lower bound** on f(n)
  - f(n) grows no slower than g(n) for "large" n



- **Big Omega – Examples**

  - Let $f(n) = 2n^2$
    - $f(n) = \Omega(n)$
    - $f(n) = \Omega(n^2)$ (best answer)

- **Big Theta**
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- growth rate of f(n) = growth rate of g(n)
- Big-Theta means the bound is the **tightest** possible
- Example: Let $f(n)=2n^2$, $g(n)=n^2$
  - since $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$,
    thus $f(n) = \Theta(g(n))$



- **Big Theta – Some rules**

  - If T(n) is a asymptotically positive polynomial of degree k, then $T(n) = \Theta(n^k)$

  - For logarithmic functions,

    $T(\log_m n) = \Theta(\log n)$, (use: $T(\log_m n) = T((\log_2 n) / (\log_2 m))$)

A Comparison of Growth-Rate Functions

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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

- **Asymptotic notations**
  - When n goes to infinity
    - Upper bound **O(g(n)**
      - **—the most popular**
    - Lower bound **Ω(g(n))**
    - Tight bound **Θ(g(n))**

Growth-Rate Functions

O(1)	Time requirement is **constant**, and it is independent of the input size.
O($\log_2 n$)	Time requirement for a **logarithmic** algorithm increases increases slowly as the input size increases.
O(n)	Time requirement for a **linear** algorithm increases directly with the input size.
O(n*$\log_2 n$)	Time requirement for a **n*$\log_2 n$** algorithm increases more rapidly than a linear algorithm.
O(n^2)	Time requirement for a **quadratic** algorithm increases rapidly with the input size.
O(n^3)	Time requirement for a **cubic** algorithm increases more rapidly with the input size than the time requirement for a quadratic algorithm.
O(2^n)	As the input size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

- **Reminder of Properties of Growth-Rate Functions**
  - We can ignore low-order terms in an algorithm's growth-rate function.
    - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
    - We only use the higher-order term as algorithm's growth-rate function.
  - We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.
    - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.
  - $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
    - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n)$ ⇨ it is $O(n^3)$
  - $O(f(n)) * O(g(n)) = O(f(n) * g(n))$

- **Example 1**

```
                                    Cost        Times
i = 1;                              c1          1
sum = 0;                            c2          1
while (i <= n) {                    c3          n+1
      i = i + 1;                    c4          n
      sum = sum + i;                c5          n
}
```

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5
= (c3+c4+c5)*n + (c1+c2+c3)
= a*n + b

⇨ the growth-rate function for this algorithm is **O(n)**

- **Example 2**

```
for (i=1; i<=n; i++)
        for (j=1; j<=i; j++)
                for (k=1; k<=j; k++)
                        x=x+1;
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1$$

⇨ the growth-rate function for this algorithm is **$O(n^3)$**

- **Example 3**

```
                                        Cost        Times
i=1;                                    c1          1
sum = 0;                                c2          1
while (i <= n) {                        c3          n+1
      j=1;                              c4          n
      while (j <= n) {  c5              n*(n+1)
          sum = sum + i;                c6          n*n
          j = j + 1;                    c7          n*n
   }
   i = i +1;                            c8          n
}
```

$T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8$

$= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$

$= a*n^2 + b*n + c$

⇨ the growth-rate function for this algorithm is **$O(n^2)$**

CONTENTS

- Introduction to algorithm
- Algorithm analysis
- Estimating running time
- Algorithm growth rates (Big O, Omega, Theta)
- **Worst-Case, Best-Case, Average-Case**

Worst-Case, Best-Case, Average-Case

- **Running time of algorithm depends on not only the input size but also the input content**
- **Worst-Case**
  - The maximum amount of time that an algorithm require to solve a problem of size n.
  - This gives an upper bound for the time complexity of an algorithm.
  - Normally, we try to find worst-case behavior of an algorithm.
- **Best-Case**
  - The minimum amount of time that an algorithm require to solve a problem of size n.
  - The best case behavior of an algorithm is NOT so useful.
- **Average-Case**
  - The average amount of time that an algorithm require to solve a problem of size n.
  - Sometimes, it is difficult to find the average-case behavior of an algorithm.
  - We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.
  - Worst-case analysis is more common than average-case analysis.

- **Sequential Search – Analysis**

```
int sequentialSearch(const int a[],  int item,  int n){
    for (int i = 0; i < n && a[i]!= item; i++);
    if (i == n)       return   –1;
    return  i;
}
```

- Unsuccessful Search: O(n)
- Successful Search:

Best-Case: item is in the first location of the array ⇨ O(1)

Worst-Case: item is in the last location of the array ⇨ O(n)

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n} \quad \Rightarrow O(n)$$

- **Binary Search – Analysis**

```
int binarySearch(int a[],  int size,  int x) {
   int low =0;
   int high = size –1;
   int mid;                    // mid will be the index of target when it's found.
   while (low <= high) {
         mid = (low + high)/2;
         if (a[mid] < x)              low = mid + 1;
         else if (a[mid] > x)         high  = mid – 1;
                 else                 return mid;
   }
   return –1;
}
```

We can do binary search if the array is sorted

- **Binary Search – Analysis**
  - Unsuccessful Search
    - The size of the list for each iteration: $n/2, n/2^2, n/2^3, \ldots, n/2^k$
    - Loop stops when $n/2^k = 1$, where k is the number of iterations
    - Then, the number of iterations k in the loop is $\log_2 n \Rightarrow O(\log_2 n)$
  - Successful Search

    Best-Case: The number of iterations is 1 ⇨ $O(1)$

    Worst-Case: The number of iterations is $\log_2 n$ ⇨ $O(\log_2 n)$

    Average-Case: The avg. number of iterations < $\log_2 n$ ⇨ $O(\log_2 n)$

- How much better is $O(\log_2 n)$?

n	$O(\log_2 n)$
16	4
64	6
256	8
1024	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576	20
1,073,741,824	30

Conclusions

- **Running time depends on**
  - Input size
    - A function of n (input size)
    - Running time is significant, when n goes to infinity
  - Input content
    - Best-case, Average-case, Worst-case
- **Steps for estimating running time** (compelxity)
  1. Counting the number of significant operations/statements
     - We often consider **the worst-case**
  2. Applying the growth rate functions (O, Ω, Θ)
     - When **n goes to infinity**
     - **O is the most popularly used**
  3. Classifying the algorithm running time/complexity
     - The algorithm is **impractical**, if its running time is more than $O(n^3)$

SUMMARY

- Introduction to algorithm
- Algorithm analysis
- Estimating running time
- Algorithm growth rates
- Worst-Case, Best-Case, Average-Case

Enjoy the Course…!