DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 3

Algorithm Analysis

Outlines of Class (Lecture 3)

- Introduction
- Algorithm
- Asymptotic Analysis
- Algorithmic Performance
- Algorithm Growth Rates
- Best, Worst, and Average Cases
- Asymptotic Notations
  - ✓ O Notation
  - ✓ Ω Notation
  - ✓ θ Notation
- Growth-Rate Functions

Introduction

- Introduces the motivation, basic notation, and fundamental techniques of algorithm analysis
- Focus on a methodology known as asymptotic algorithm analysis, or simply asymptotic analysis.
- Asymptotic analysis
  - ✓ attempts to estimate the resource consumption of an algorithm,
  - ✓ allows to compare the relative costs of two or more algorithms for solving the same problem,
  - ✓ gives algorithm designers a tool for estimating whether a proposed solution is likely to meet the resource constraints for a problem before they implement an actual program.
- The running time of an algorithm depends on the "size" and "complexity" of the input.
- In the simplest terms, for a problem where the input size is n.

Algorithm

- An *algorithm* is a set of instructions to be followed to solve a problem.
  - ✓ There can be more than one solution (more than one algorithm) 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.
  - ✓ e.g. For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Asymptotic Analysis

- ***Given two algorithms for a task, how do we find out which one is better?***
  - ✓ One way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.
  - ✓ There are many problems with this approach for analysis of algorithms.
    - ✓ It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
    - ✓ It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis (cont.)

- In Asymptotic Analysis, evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time) and calculate, how does the time (or space) taken by an algorithm increases with the input size.
- When analyzing the running time or space usage of programs, try to estimate the time or space as function of the input size.

Algorithmic Performance

There are *two aspects* of algorithmic performance:

- Time
  - ✓ Instructions take time.
  - ✓ How fast does the algorithm perform?
  - ✓ What affects its runtime?
- Space
  - ✓ Data structures take space
  - ✓ What kind of data structures can be used?
  - ✓ How does choice of data structure affect the runtime?
- focus on **time**:
  - ✓ How to estimate the time required for an algorithm
  - ✓ How to reduce the time required

Algorithmic Performance (cont.)

- Many factors affect the running time of a program.
  - ✓ Some relate to the environment in which the program is compiled and run. Such factors include the speed of the computer's CPU, bus, and peripheral hardware.
  - ✓ Competition with other users for the computer's (or the network's) resources can make a program slow to a crawl.
  - ✓ The programming language and the quality of code generated by a particular compiler can have a significant effect.
  - ✓ The "coding efficiency" of the programmer who converts the algorithm to a program can have a tremendous impact as well.
- Of primary consideration when estimating an algorithm's performance is the number of basic operations required by the algorithm to process an input of a certain size.

Example 1

- Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the largest-value sequential search and is illustrated by the following function:

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
  int currlarge = 0; // Holds largest element position
  for (int i=1; i<n; i++)    // For each array element
   if (A[currlarge] < A[i]) // if A[i] is larger
      currlarge = i;        //   remember its position
  return currlarge;         // Return largest position
}
```

- The size of the problem is A.length, the number of integers stored in array A
- The time T to run the algorithm as a function of n (n= input size), written as T(n): non-negative value
- c is the amount of time required to compare two integers in function largest
- Running time expressed by the equation T(n) = cn . This equation describes the growth rate for the running time of the largest value sequential search algorithm.

Example 2

The running time of a statement that assigns the first value of an integer array to a variable is simply the time required to copy the value of the first array value.

- Takes a constant amount of time regardless of the value. Let us call c1, the amount of time necessary to copy an integer.
- No matter how large the array on a typical computer, the time to copy the value from the first position of the array is always c1
- The equation for this algorithm is simply T(n) = c1, indicating that the size of the input n has no effect on the running time. This is called a constant running time.

Example 3

Consider the following code:

```
sum = 0;
for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
sum++;
```

What is the running time for this code fragment?

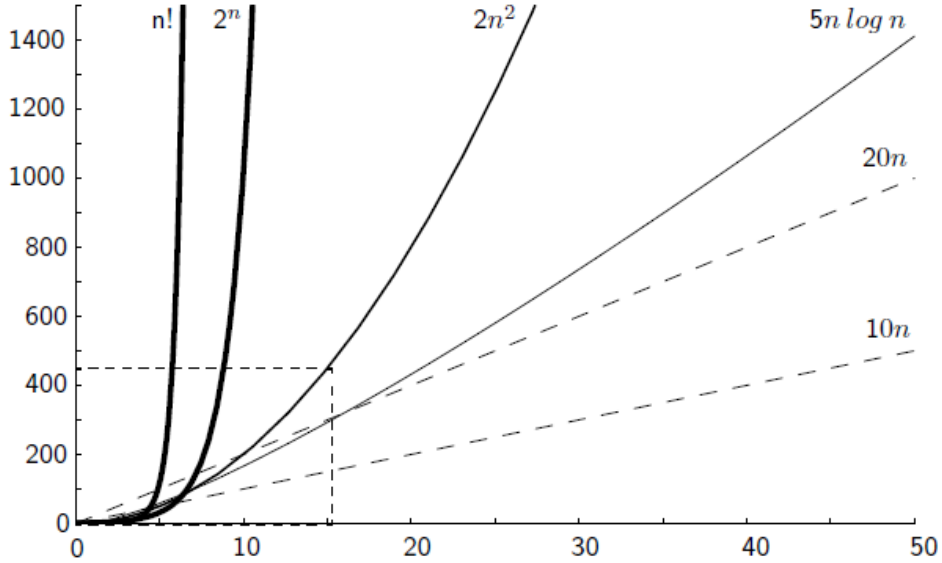
- It takes longer to run when n is larger.
- The basic operation in this example is the increment operation for variable sum. We can assume that incrementing takes constant time; call this time c_2.
- The total number of increment operations is n^2. Thus, the running time is $T(n) = c_2n^2$.

Algorithm Growth Rates

- Asymptotic Notations are languages that allow us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases. This is also known as an algorithm's growth rate.
- The asymptotic behavior of a function $f(n)$ (such as $f(n)=c*n$ or $f(n)=c*n^2$, etc.) refers to the growth of $f(n)$ as n gets large. We typically ignore small values of n, since we are usually interested in estimating how slow the program will be on large inputs.
- The slower the asymptotic growth rate, the better the algorithm .
- We measure an algorithm's time requirement as a function of the *problem size*.
  - ✓Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number of disks for towers of Hanoi.

Algorithm Growth Rates (cont.)

- So, for instance, we say that (if the problem size is n)
  - ✓ Algorithm A requires $5*n^2$ time units to solve a problem of size n.
  - ✓ Algorithm B requires $7*n$ time units to solve a problem of size n.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem 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*.
- We can compare the efficiency of two algorithms by comparing their growth rates.



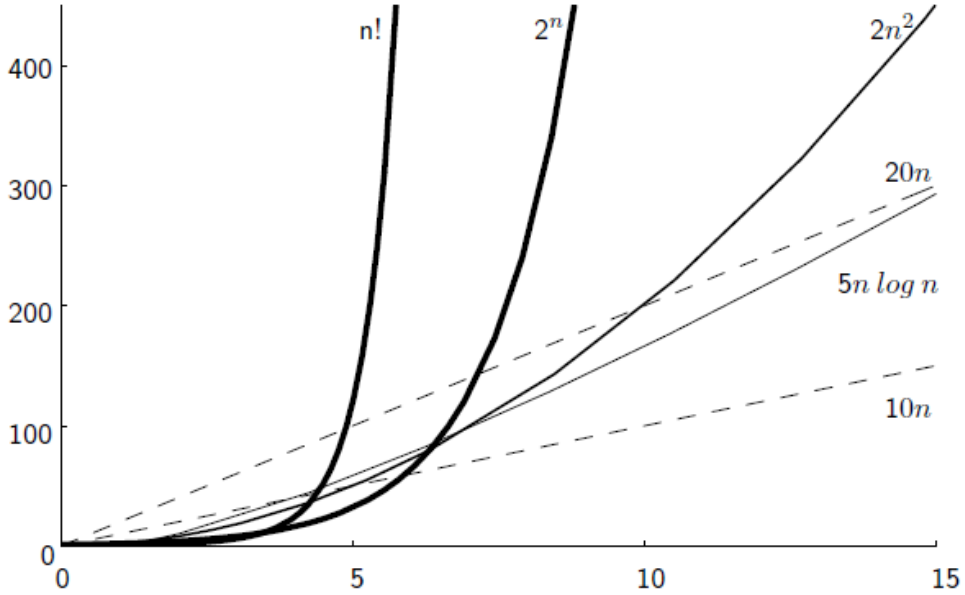


Figure 1 Growth Rate

Common Growth Rates

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

n	$\log\log n$	$\log n$	n	$n \log n$	n^2	n^3	2^n
16	2	4	2^4	$2 \cdot 2^4 = 2^5$	2^8	2^{12}	2^{16}
256	3	8	2^8	$8 \cdot 2^8 = 2^{11}$	2^{16}	2^{24}	2^{256}
1024	≈ 3.3	10	2^{10}	$10 \cdot 2^{10} \approx 2^{13}$	2^{20}	2^{30}	2^{1024}
64K	4	16	2^{16}	$16 \cdot 2^{16} = 2^{20}$	2^{32}	2^{48}	2^{64K}
1M	≈ 4.3	20	2^{20}	$20 \cdot 2^{20} \approx 2^{24}$	2^{40}	2^{60}	2^{1M}
1G	≈ 4.9	30	2^{30}	$30 \cdot 2^{30} \approx 2^{35}$	2^{60}	2^{90}	2^{1G}

Figure 2 Costs for growth rates representative of most computer algorithms

Best, Worst, and Average Cases

- **Best case**: fastest time to complete, For example, the best case for a sorting algorithm would be data that's already sorted.
  - ✓ **Best Case Analysis:** In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed.
- **Worst case**: slowest time to complete, For example, the worst case for a sorting algorithm might be data that's sorted in reverse order.
  - ✓ **Worst Case Analysis:** In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed.
- **Average case**: arithmetic mean. Run the algorithm many times, using many different inputs of size n that come from some distribution that generates these inputs, compute the total running time, and divide by the number of trials.
  - ✓ **Average Case Analysis:** In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs.

Asymptotic Notations

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - ✓ O Notation
  - ✓ Ω Notation
  - ✓ θ Notation

Upper Bounds (O Notation)

- Several terms are used to describe the running-time equation for an algorithm.
- One is the upper bound for the growth of the algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete. (big-O notation).
- If the upper bound for an algorithm's growth rate is f(n), then this algorithm is "in the set O(f(n))in the worst case"
- For example, if n^2 grows as fast as T(n) for the worst-case input, we would say the algorithm is "in $O(n^2)$ in the worst case."

The worst case provides an upper bound for an algorithm.

Example 3.4 Consider the sequential search algorithm for finding a specified value in an array of integers. If visiting and examining one value in the array requires c_s steps where c_s is a positive number, and if the value we search for has equal probability of appearing in any position in the array, then in the average case $\mathbf{T}(n) = c_s n/2$. For all values of $n > 1$, $c_s n/2 \leq c_s n$. Therefore, by the definition, $\mathbf{T}(n)$ is in $\mathrm{O}(n)$ for $n_0 = 1$ and $c = c_s$.

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$. So, $\mathbf{T}(n) \leq c n^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$ by the second definition.

Example 3.6 Assigning the value from the first position of an array to a variable takes constant time regardless of the size of the array. Thus, $\mathbf{T}(n) = c$ (for the best, worst, and average cases). We could say in this case that $\mathbf{T}(n)$ is in $\mathrm{O}(c)$. However, it is traditional to say that an algorithm whose running time has a constant upper bound is in $\mathrm{O}(1)$.

Upper Bounds (cont.)

- Complexity and running time are often expressed in "Big O notation," which is used to describe the approximate amount of time an algorithm requires to complete, based the size of its input.

- The most commonly used Big O descriptions are
  - ✓ **O(1)** always terminates in about the same amount of time, regardless of the input size.
  - ✓ **O(N)** takes twice as long to finish if the input size doubles.
  - ✓ **O(N^2)** takes four times as long if the input size doubles.
  - ✓ **O(2^N)** increases exponentially as the input size increases.

The Execution Time and Cost of Algorithms

- Each operation in an algorithm (or a program) has a cost.
  - ✓ Each operation takes a certain of time.
  - ✓ count = count + 1; ⇨ take a certain amount of time, but it is constant.
- A sequence of operations:

count = count + 1;	Cost: c_1
sum = sum + count;	Cost: c_2
.	.
.	.
.	.

Total Cost = $c_1 + c_2 + \ldots\ldots$

The Execution Time and Cost of Algorithms (cont.)

Example: Simple If-Statement

```
                                    Cost        Times
if (marks < 50)                     c1            1
        result= pass;               c2            1
else
        result= fail;               c3            1
```

Total Cost = c1 + max(c2,c3)

The Execution Time and Cost of Algorithms (cont.)

Example: Simple Loop

```
                                    Cost        Times
i = 1;                              c1            1
sum = 0;                            c2            1
while (i <= n) {                    c3          n+1
        i = i + 5;                  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**

The Execution Time and Cost of Algorithms (cont.)

Example: Nested Loop

	Cost	Times
a=1;	c1	1
sum = 0;	c2	1
while (a <= n) {	c3	n+1
b=1;	c4	n
while (b <= n) {	c5	n*(n+1)
sum = sum + a;	c6	n*n
b = b + 2;	c7	n*n
}		
a = a +2;	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 $\mathbf{n^2}$

Lower Bounds (Ω Notation)

- Like big-Oh notation, it works for any resource, but often measure the least amount of time required. Like big-Oh notation, we are measuring the resource required for some particular class of inputs: input of size n.
- The lower bound for an algorithm (or a problem) is denoted by the symbol Ω, pronounced "big-Omega" or just "Omega."
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

The best case provides an lower bound for an algorithm.

Lower Bounds (cont.)

Example 3.7 Assume $\mathbf{T}(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

- This running time is in (n^2).

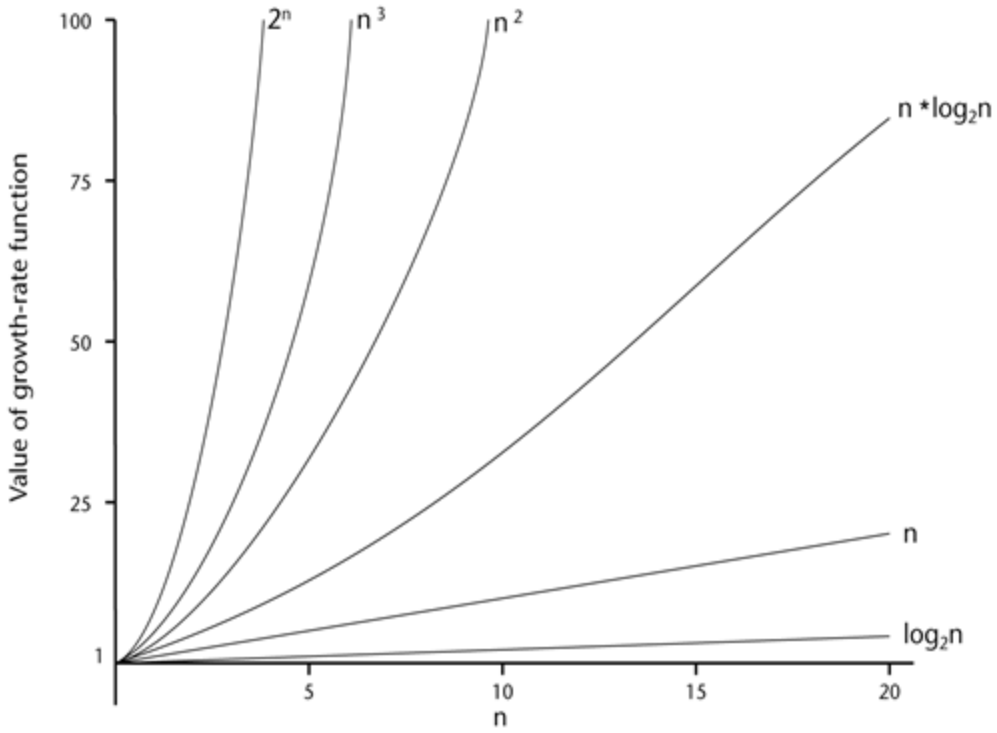
Θ Notation

- The definitions for big-O and Ω give us ways to describe the upper bound for an algorithm (if we can find an equation for the maximum cost of a particular class of inputs of size n) and the lower bound for an algorithm (if we can find an equation for the minimum cost for a particular class of inputs of size n).
- When the upper and lower bounds are the same within a constant factor, we indicate this by using Θ(big-Theta) notation.
- An algorithm is said to be Θ(h(n)) if it is in O(h(n)) and it is in Ω(h(n)).
- In other words, if f(n) is Θ(g(n)), then g(n) is Θ(f(n)).
- Because the sequential search algorithm is both in O(n) and in Ω(n) in the average case, we say it is Θ(n) in the average case.

A Comparison of Growth-Rate Functions

- Some common orders of growth seen often in complexity analysis are

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"n log n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential



Growth-Rate Functions

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

Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 10, what is the time requirement (approximately) for that algorithm with the problem size 30?

- If its order is:

O(1)	➔	T(n) = 1 second
O($\log_2 n$)	➔	T(n) =
O(n)	➔	T(n) =
O(n*$\log_2 n$)	➔	T(n) =
O(n^2)	➔	T(n) =
O(n^3)	➔	T(n) =
O(2^n)	➔	T(n) =

Contd.

Properties of Growth-Rate Functions

- Ignore low-order terms
  - ✓ If an algorithm is **$O(n^3+2n^2+7n)$**, it is also **$O(n^3)$**.
  - ✓ Use the higher-order term as algorithm's growth-rate function.

- Ignore a multiplicative constant in the higher-order term
  - ✓ If an algorithm is **$O(7n^2)$**, it is also **$O(n^2)$**.

- $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
  - ✓ Can combine growth-rate functions.
  - ✓ If an algorithm is $O(2n^3) + O(5n^2)+O(3n)$, it is also $O(2n^3 +5n^{2+}+3n)$ ➔ it is $O(n^3)$.

Simplifying Rules

1. If f(n) is in O(g(n)) and g(n) is in O(h(n)), then f(n) is in O(h(n)).
2. If f(n) is in O(kg(n)) for any constant k > 0, then f(n) is in O(g(n)).
3. If f1(n) is in O(g1(n)) and f2(n) is in O(g2(n)), then f1(n) + f2(n) is in O(max(g1(n); g2(n))).
4. If f1(n) is in O(g1(n)) and f2(n) is in O(g2(n)), then f1(n)f2(n) is in O(g1(n)g2(n)).

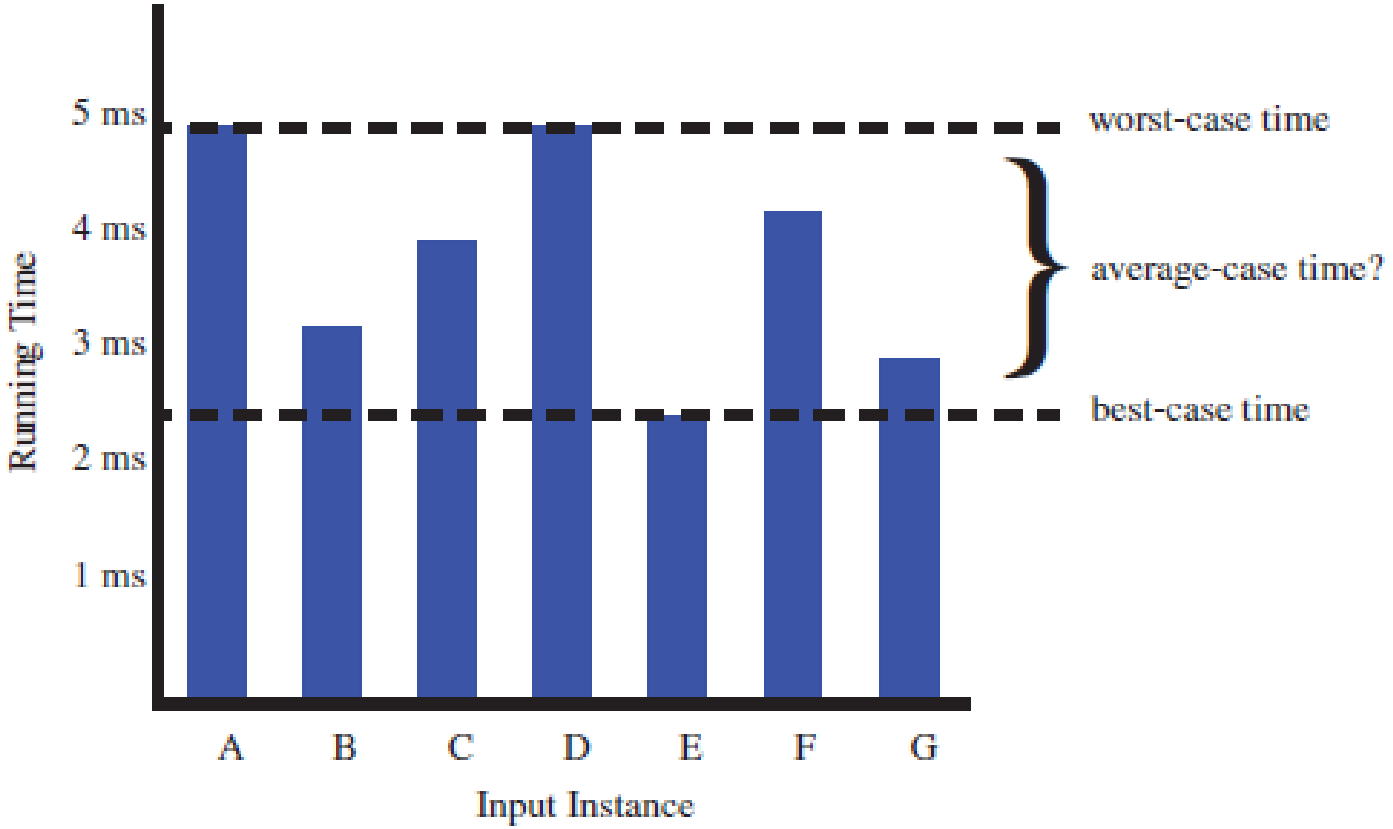


Figure 3 The difference between best-case and worst-case time

Example of Case Analysis (Sequential Search)

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

Successful Search:

Best-Case: search element is in the first location of the array ➔ O(1)

Worst-Case: search element is in the last location of the array ➔ O(n)

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

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

Unsuccessful Search: ➔ **O(n)**

Algorithms with Same Complexity

- If the functions representing the number of operations have **same rate of growth**, it means that two algorithms have **same complexity**,

- Among all functions with same growth rate, we **choose the simplest** one to represent the complexity.

Assignments

1. Graph the following expressions. For each expression, state the range of values of n for which that expression is the most efficient.

$4n^2$ $\log_3 n$ 3^n $20n$ 2 $\log_2 n$ $n^{2/3}$

2. Arrange the following expressions by growth rate from slowest to fastest.

$4n^2$ $\log_3 n$ $n!$ 3^n $20n$ 2 $\log_2 n$ $n^{2/3}$

3. Find the growth rate function for the following code segment.

```
i=1;
sum = 0;
while (i <= n) {
        j=1;
        while (j <= n) {
        sum = sum + i;
        j = j + 1;
}
i = i +1;
}
```

Next Week Lecture (Week 4)

Lecture 4: Lists, Stacks and Queues

Thank you!