Data Structures & Algorithms (COMP2113)
Lecture # 10
Algorithm Analysis | Part01

Course Instructor

Dr. Aftab Akram

PhD CS

Assistant Professor

Department of Information Sciences, Division of Science & Technology

University of Education, Lahore

Comparing Algorithms

- How do you compare two algorithms?
  - Run these algorithms are computer programs
  - Compare the resource used by them
- This approach may not work due to following reasons:
  - Have to write two programs but want to keep just one
  - “Better Written” program
  - The choice of empirical test cases might unfairly favor one algorithm
  - What if both algorithm did not fall into resource budget?

Asymptotic Analysis

- Asymptotic analysis measures the efficiency of an algorithm
- Actually an estimating technique
- Asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.
- The critical resource for a program is most often its running time.
- Typically you will analyze the time required for an algorithm (or the instantiation of an algorithm in the form of a program), and the space required for a data structure.

Factor affecting Running Time

- The environment in which program is compiled and run
  - Speed of CPU
  - Bus Speed
  - Peripheral Hardware, etc.
- Competition for network resources
- The programming languages and quality of code generated by compiler
- The Coding Efficiency of Computer Programmer

Estimation Algorithm's Performance

- Estimate Algorithm's Performance
  - number of basic operations required by the algorithm to process an input of a certain size.
- For example, when comparing sorting algorithms, the size of the problem is typically measured by the number of records to be sorted.
- A basic operation must have the property that its time to complete does not depend on the particular values of its operands.
- Adding or comparing two integer variables are examples of basic operations in most programming languages
- Summing the contents of an array containing n integers is not, because the cost depends on the value of n

Example 3.1 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
}
```

Determining Running Time

- The most important factor affecting running time is normally size of the input
- For a given input size n we often express the time T to run the algorithm as a function of n, written as $T(n)$
- We will always assume $T(n)$ is a non-negative value.
- Let us call c the amount of time required to compare two integers in function **`largest`**.
- The total time to run **`largest`** is therefore approximately cn, because we must make n comparisons, with each comparison costing c time.
- $T(n) = cn$
- This equation describes the growth rate for the running time of the largest value sequential search algorithm.

Example 3.2 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. We can assume this assignment takes a constant amount of time regardless of the value. Let us call c_1 the amount of time necessary to copy an integer. No matter how large the array on a typical computer (given reasonable conditions for memory and array size), the time to copy the value from the first position of the array is always c_1. Thus, the equation for this algorithm is simply

$$\mathbf{T}(n) = c_1,$$

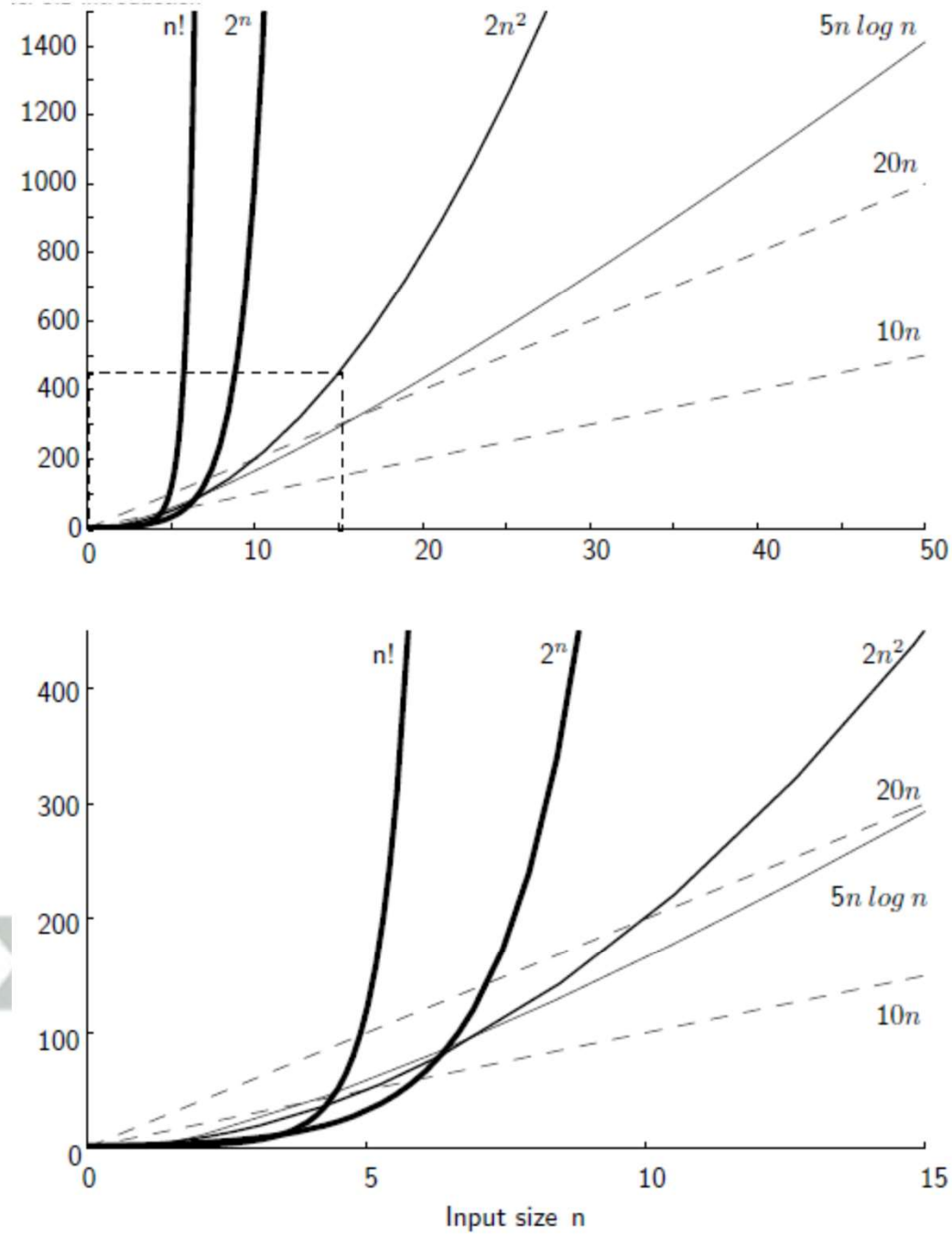
indicating that the size of the input n has no effect on the running time. This is called a **constant** running time.

Example 3.3 Consider the following code:

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

Algorithm Growth Rate

- The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
- **Linear Growth Rate:** as the value of n grows, the running time of the algorithm grows in the same proportion
- **Quadratic Growth Rate:** algorithm's running-time equation has a highest-order term containing a factor of n^2
- **Logarithmic Growth Rate:** algorithm having a $\log n$ factor in running-time equation
- **Exponential Growth Rate:** an algorithm having 2^n running time


n!
2^n
$2n^2$
$5n\ log\ n$
$20n$
$10n$
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50
n!
2^n
$2n^2$
$20n$
$5n\ log\ n$
$10n$
400
300
200
100
0
0
5
10
15
Input size n

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 3.2 Costs for growth rates representative of most computer algorithms.

Next Lecture

- In next lecture, we will discuss best, worst and average cases.