Data Structures and Algorithm

Moazzam Ali Sahi

Lecture # 4-5-6

Introduction to Algorithms

Last Lecture

- Basic definitions
- Memory Management of basic data types
- Overview of Data Structures
- Implementation of Data Structures
- Overview of Algorithm

This Lecture

- Big O Notation
- Time complexity

Introduction to Algorithm

- **Algorithm:** A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

STRUCTURED DESIGN

- Dividing a problem into smaller subproblems is called structured design.
- The structured design approach is also known as top-down design, stepwise refinement, and modular programming.
  - In structured design, the problem is divided into smaller problems.
  - Each subproblem is then analyzed, and a solution is obtained to solve the subproblem.
  - The solutions of all the subproblems are then combined to solve the overall problem.
- This process of implementing a structured design is called structured programming.

Algorithm [Brief intro…]

- A systematic (step-by-step) process/procedure to solve a particular problem is called an algorithm.

Rules:

1. **Unambiguous:** The algorithm should be clear and unambiguous. Each of its steps (or phases), inputs/outputs should be clear and must lead to only one meaning.
2. **Inputs and Outputs:** The algorithm should have well-defined inputs and outputs. There can be one or more inputs however; there will be only one output. The output should match with the desired output.
3. **Finiteness:** The algorithm must terminate after a finite number of steps.
4. **Feasibility:** The algorithm should be feasible with the available resources.

- Algorithms are written to reduce either of the following complexities.
  - Time complexity
  - Space complexity

Algorithm [Example

- Write an algorithm to calculate the sum of two numbers.

Algorithm:

```
a)  START
b)  Declare variable a, b, sum
c)  Get values of a and b
d)  sum = a + b
e)  Display sum
f)  END
```

Algorithm Analysis: The Big-O Notation

- Analyze algorithm after design
- Example
  - 50 packages delivered to 50 different houses
  - 50 houses one mile apart, in the same area

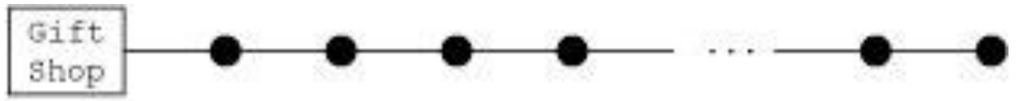


FIGURE 1-1 Gift shop and each dot representing a house

Algorithm Analysis: The Big-O Notation (cont'd.)

- Example (cont'd.)
  - Driver picks up all 50 packages
  - Drives one mile to first house, delivers first package
  - Drives another mile, delivers second package
  - Drives another mile, delivers third package, and so on
  - Distance driven to deliver packages
    - 1+1+1+... +1 = 50 miles
  - Total distance traveled: 50 + 50 = 100 miles

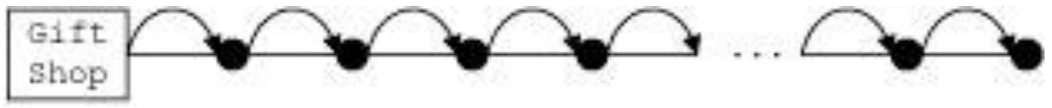


FIGURE 1-2 Package delivering scheme

Algorithm Analysis: The Big-O Notation (cont'd.)

- Example (cont'd.)
  - Similar route to deliver another set of 50 packages
    - Driver picks up first package, drives one mile to the first house, delivers package, returns to the shop
    - Driver picks up second package, drives two miles, delivers second package, returns to the shop
  - Total distance traveled
    - 2 * (1+2+3+…+50) = 2550 miles

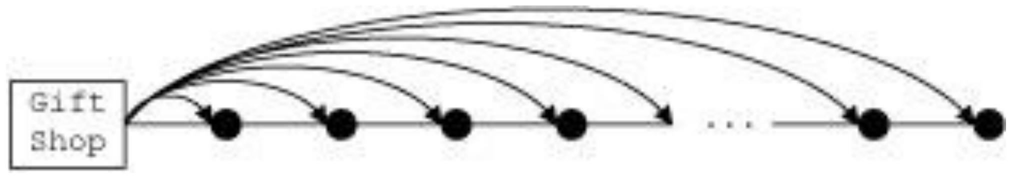


FIGURE 1-3 Another package delivery scheme

Algorithm Analysis: The Big-O Notation (cont’d.)

- Example (cont’d.)
  - *n* packages to deliver to *n* houses, each one mile apart
  - First scheme: total distance traveled
    - $1+1+1+\ldots+n = 2n$ miles
    - Function of n
  - Second scheme: total distance traveled
    - $2 * (1+2+3+\ldots+n) = 2*(n(n+1) / 2) = n^2+n$
    - Function of n^2

Algorithm Analysis: The Big-O Notation (cont'd.)

- Analyzing an algorithm

  Time
  - Count number of operations performed
    - Not affected by computer speed

  Space
  - Find out the space requirement

  - Network
    - For cloud/network, amount of data transfer matters
  - Power
    - Specially for handheld devices
  - CPU Registers
    - When writing the device drivers
  - And More

Algorithm

```
Algorithm swap (a,b)
{
        temp  ←    a
        a     ←    b
        b     ←    temp
}
```

Time:

Every simple statement takes 1 unit of time

Complexity of a statement doesn't matter

x= 5*a + 6*b

Algorithm

```
Algorithm swap (a,b)
{
        temp ←   a
        a    ←   b
        b    ←   temp
}
```

Space:

Deals with variables used

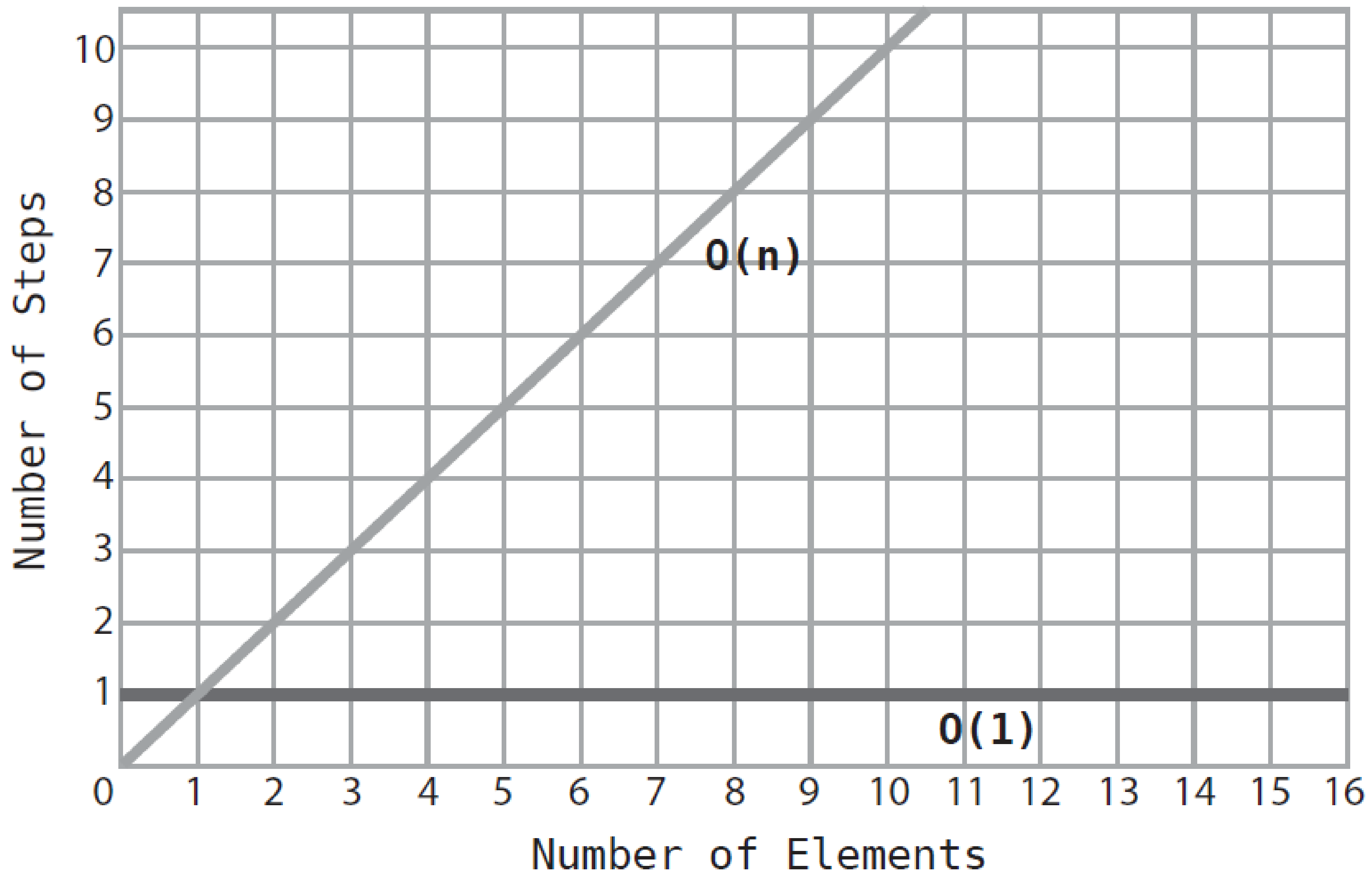
Big O: Count the Steps

- Instead of focusing on units of time, Big O achieves consistency by focusing only on the number of steps that an algorithm takes.
- In Why Data Structures Matter, we discovered that reading from an array takes just one step, no matter how large the array is. The way to express this in Big O Notation is O(1)
- O(1) simply means that the algorithm takes the same number of steps no matter how much data there is. In this case, reading from an array always takes just one step no matter how much data the array contains.
- On an old computer, that step may have taken twenty minutes, and on today's hardware it may take just a nanosecond. But in both cases, the algorithm takes just a single step.
- Other operations that fall under the category of O(1) are the **insertion** and **deletion** of a value at the end of an array. As we've seen, each of these operations takes just one step for arrays of any size, so we'd describe their efficiency as O(1).

Constant Time vs. Linear Time

- Another way of saying this is that Big O answers the following question: *how does the number of steps change as the data increases?*
- O(N) vs O(1)

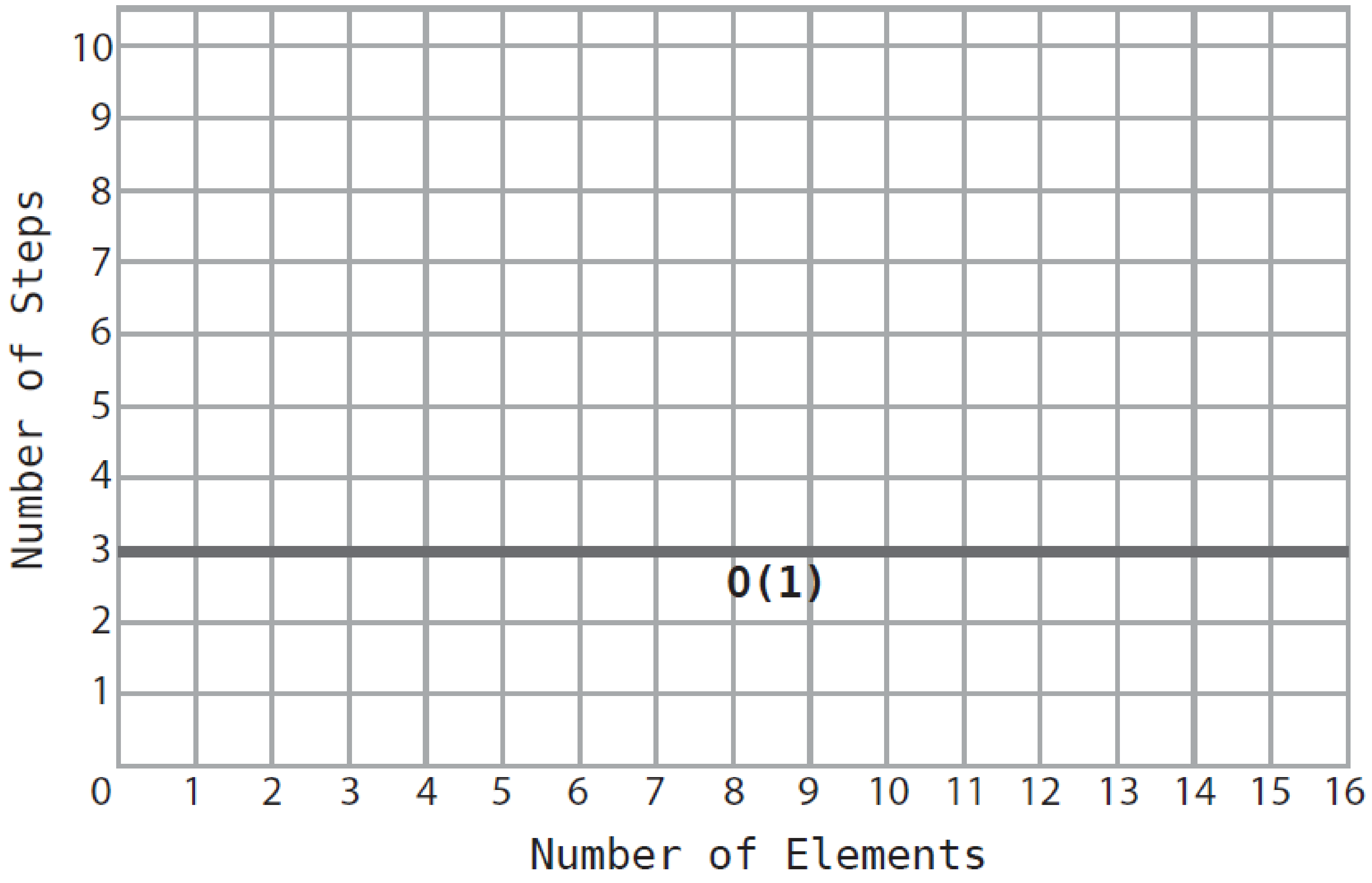
Constant Time vs. Linear Time



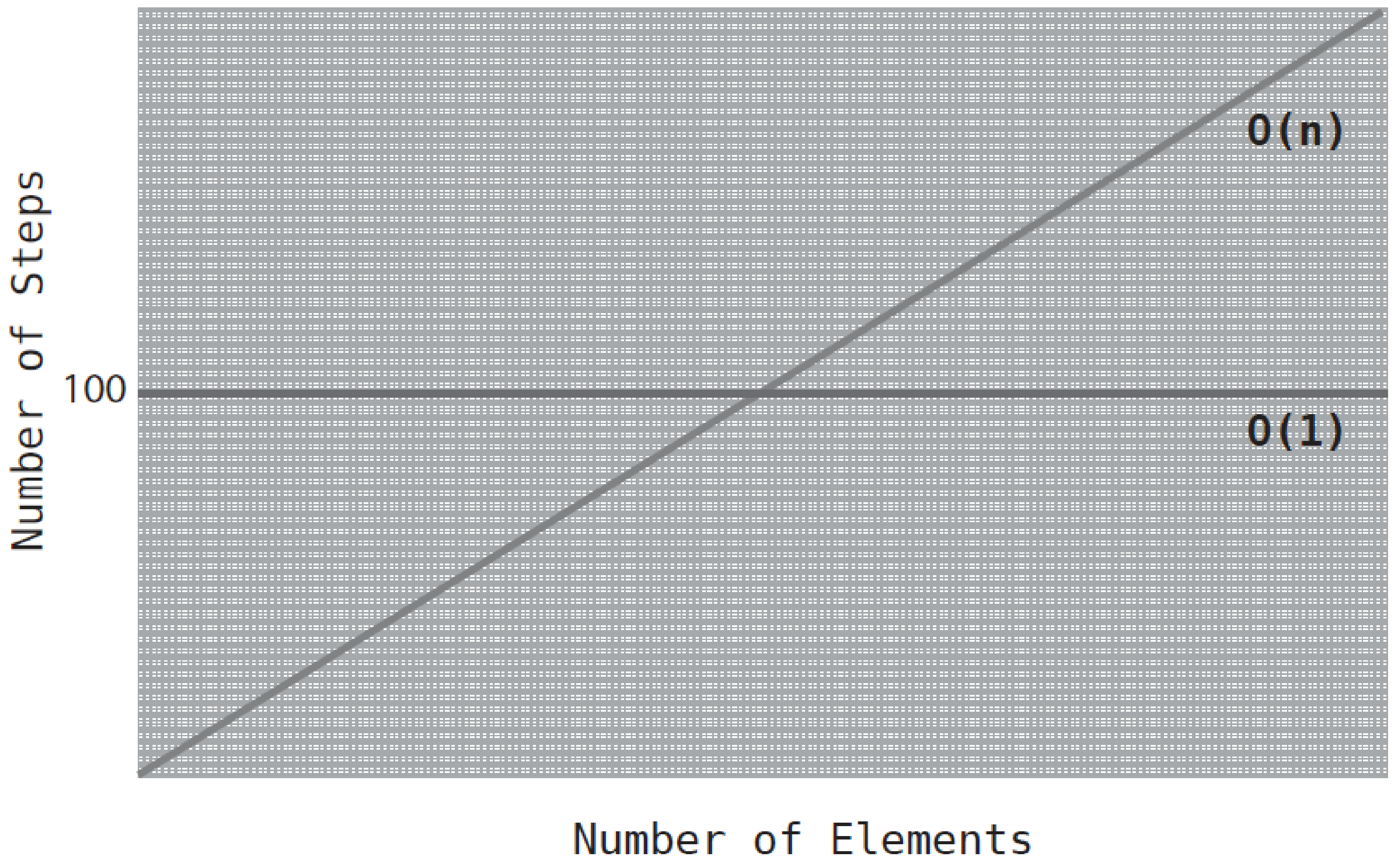
Constant Time vs. Linear Time

- As Big O is primarily concerned about how an algorithm performs across varying amounts of data, an important point emerges: an algorithm can be described as O(1) even if it takes more than one step.
- Let’s say that a particular algorithm always takes three steps, rather than one—but it always takes these three steps no matter how much data there is. On a graph, such an algorithm would look like this:

Constant Time vs. Linear Time



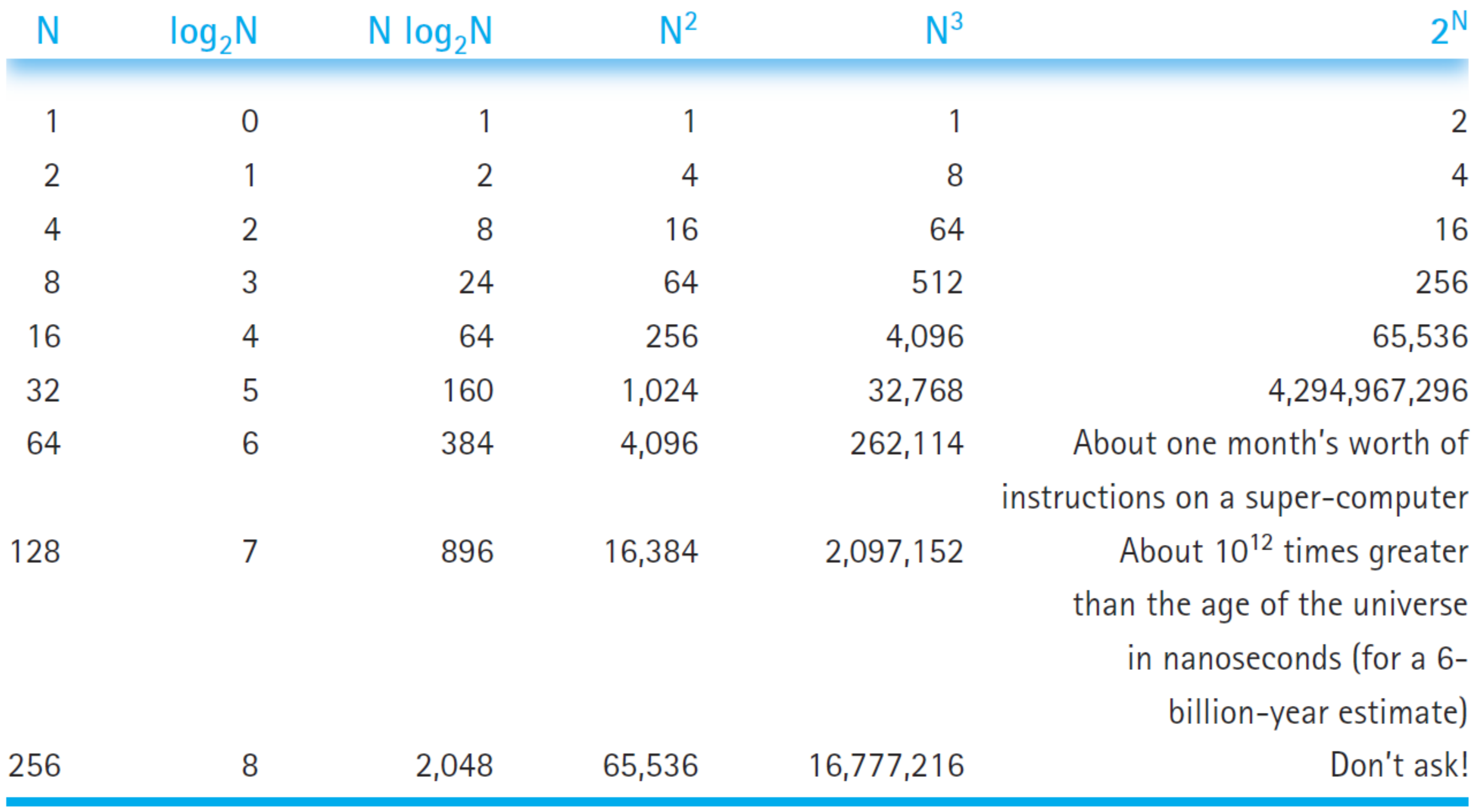
Constant Time vs. Linear Time



An Algorithm of the Third Kind

- Is there anything other than O(1) and O(N) ?
- Binary search → O(log N)
- Describing an algorithm that *increases one step each time the data is doubled*.
- a

Comparison of rate of growth



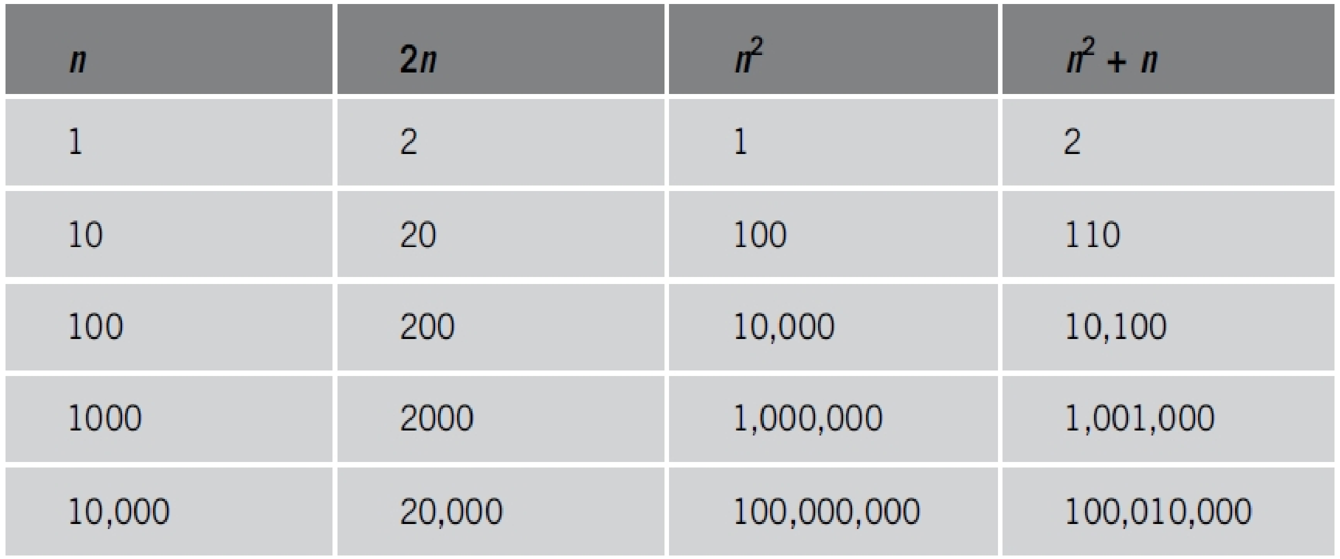
N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,114	About one month's worth of instructions on a super-computer
128	7	896	16,384	2,097,152	About 10^{12} times greater than the age of the universe in nanoseconds (for a 6-billion-year estimate)
256	8	2,048	65,536	16,777,216	Don't ask!

- Neil Dale C++ Data Structures pdf page 178

Algorithm Analysis: The Big-O Notation (cont'd.)

- Analyzing an algorithm
  - Count number of operations performed
    - Not affected by computer speed

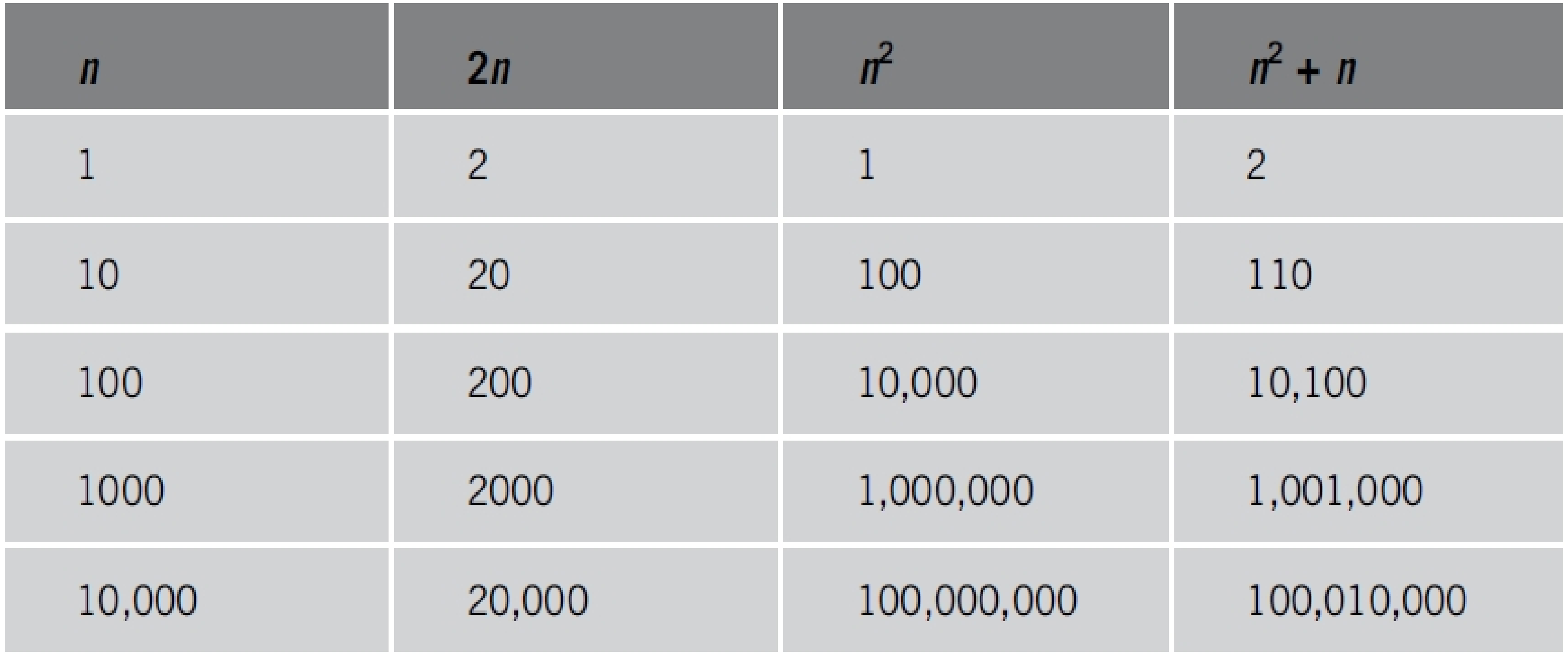
TABLE 1-1 Various values of n, $2n$, n^2, and $n^2 + n$



n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

Algorithm Analysis: The Big-O Notation (cont’d.)

TABLE 1-1 Various values of n, $2n$, n^2, and $n^2 + n$

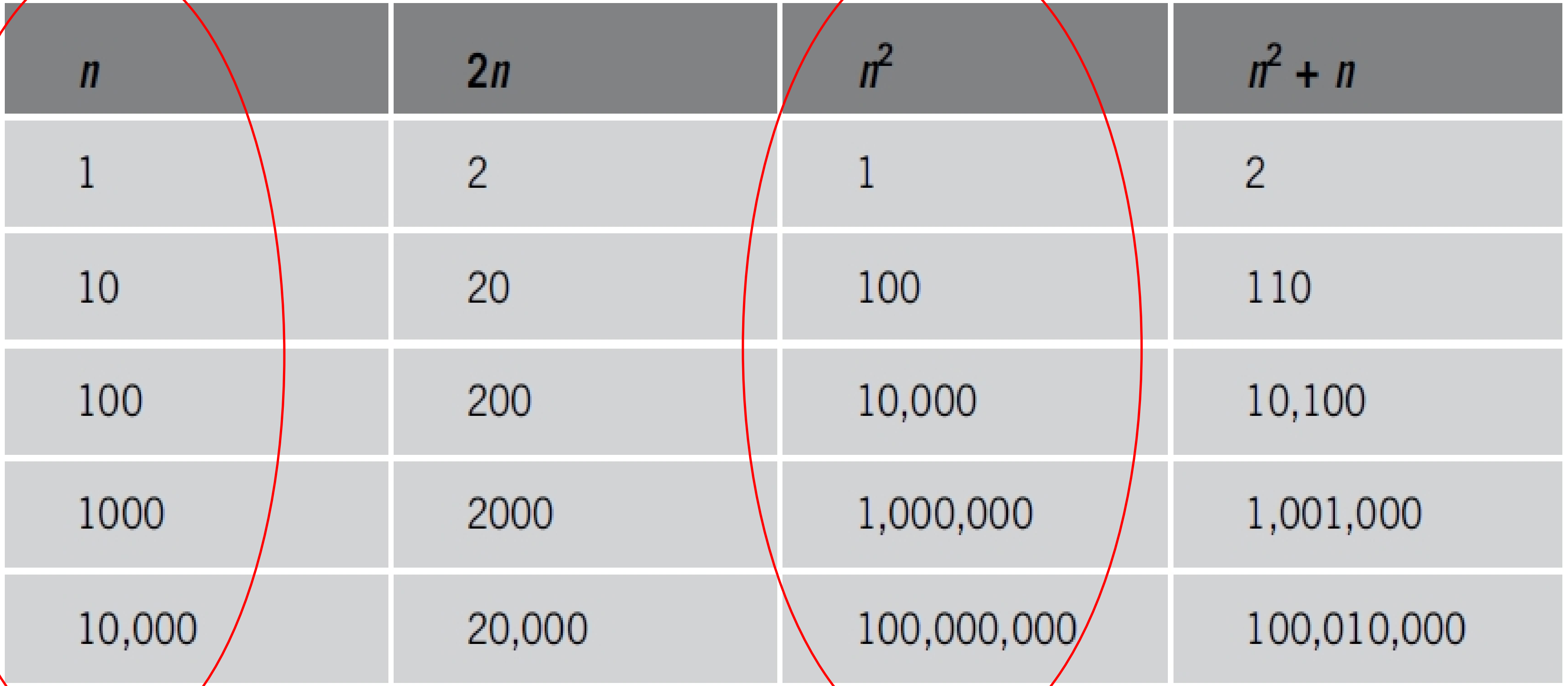


n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

(n) and ($2n$) are close, so we magnify (n)
(n^2) and ($n^2 + n$) are close, so we magnify (n^2)

Algorithm Analysis: The Big-O Notation (cont'd.)

TABLE 1-1 Various values of n, $2n$, n^2, and $n^2 + n$



n	$2n$	n^2	$n^2 + n$
1	2	1	2
10	20	100	110
100	200	10,000	10,100
1000	2000	1,000,000	1,001,000
10,000	20,000	100,000,000	100,010,000

When n becomes too large, n and n^2 becomes very different

Algorithm Analysis: The Big-O Notation (cont'd.)

- Example 1-1
  - Illustrates fixed number of executed operations

```
cout << "Enter two numbers";                           //Line 1

cin >> num1 >> num2;                                   //Line 2

if (num1 >= num2)                                      //Line 3
    max = num1;                                        //Line 4
else                                                   //Line 5
    max = num2;                                        //Line 6

cout << "The maximum number is: " << max << endl;      //Line 7
```

Only one of them will be executed

//Line 1	1 operation
//Line 2	2 operations
//Line 3	1 operation
//Line 4	1 operation
//Line 5	
//Line 6	1 operation
//Line 7	3 operations

Total of 8 operations

Algorithm Analysis: The Big-O Notation

- Example 1-2 Illustrates dominant operations

```
cout << "Enter positive integers ending with -1" << endl;   //Line 1    2 operations

count = 0;                                                  //Line 2    1 operation
sum = 0;                                                    //Line 3    1 operation

cin >> num;                                                 //Line 4    1 operation

while (num != -1)                                           //Line 5    N+1 operations
{
    sum = sum + num;                                        //Line 6    2N operations
    count++;                                                //Line 7    N operations
    cin >> num;                                             //Line 8    N operations
}

cout << "The sum of the numbers is: " << sum << endl;       //Line 9    3 operations

if (count != 0)                                             //Line 10   1 operation
    average = sum / count;                                  //Line 11   2 operations
else                                                        //Line 12
    average = 0;                                            //Line 13   1 operation

cout << "The average is: " << average << endl;              //Line 14   3 operation
```

N times the condition is TRUE
+ 1 time the condition is FALSE

Executed while the cond. is TRUE

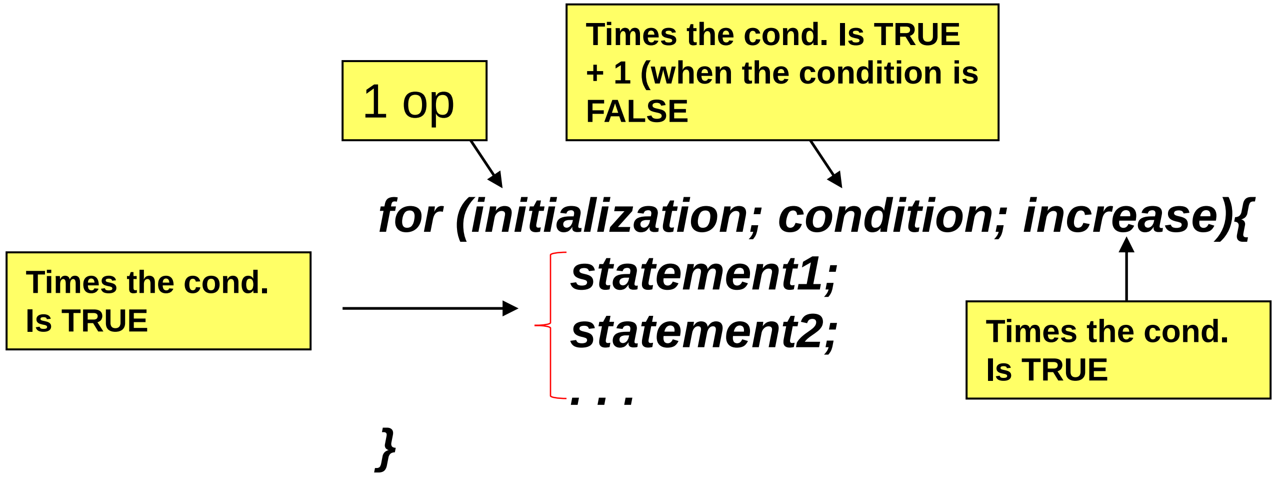
Only one of them will be executed, take the max: 2

If the while loop executes N times then:

2+1+1+1+5*N + 1 + 3 + 1 + (2) + 3 = 5N+(15)

Algorithm Analysis: The Big-O Notation

How to count *for* loops



```
for (initialization; condition; increase){
        statement1;
        statement2;
        . . .
}
```

Example

1 op   6 op   5 op

```
for(i=1; i<=5; i++)
```

5 op   30 op   25 op

```
    for(j=1;j<=5; j++)
    {
        cout <<"*";
        sum =sum+j;
    }
```

25 op

25 op   25 op

Finally: 147

```
for(i=1; i<=n; i++)
    for(j=1; j<=n; j++){
        cout <<"*";
        Sum=sum + j;
        }
```

$2n+2+$
$2n^2+2n+$
$3n^2$

$= 5n^2+ 4n+2$

Algorithm Analysis: The Big-O Notation (cont'd.)

- Sequential search algorithm

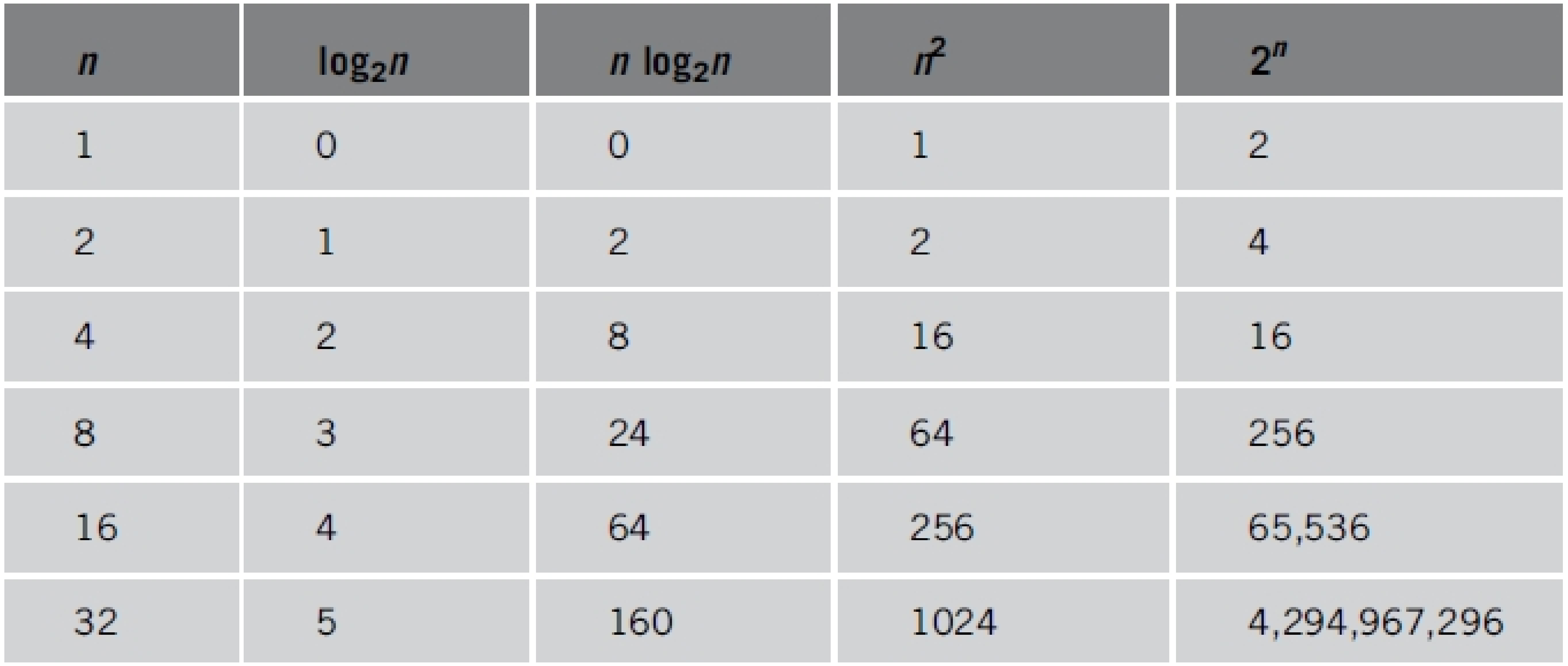
```
for(i=0; i<n; i++)                 // at most  1 + (n+1) + n
    if(a[i] = =  searchKey)        // at most n
        return true;               // 0 or 1 time
```

Total ops = 3n + 2

- *n:* represents list size
- *f*(*n*)*:* number of basic operations (3n + 2)
- *c*: units of computer time to execute one operation
  - Depends on computer speed (varies)
- *cf*(*n*): computer time to execute *f*(*n*) operations

Algorithm Analysis: The Big-O Notation (cont'd.)

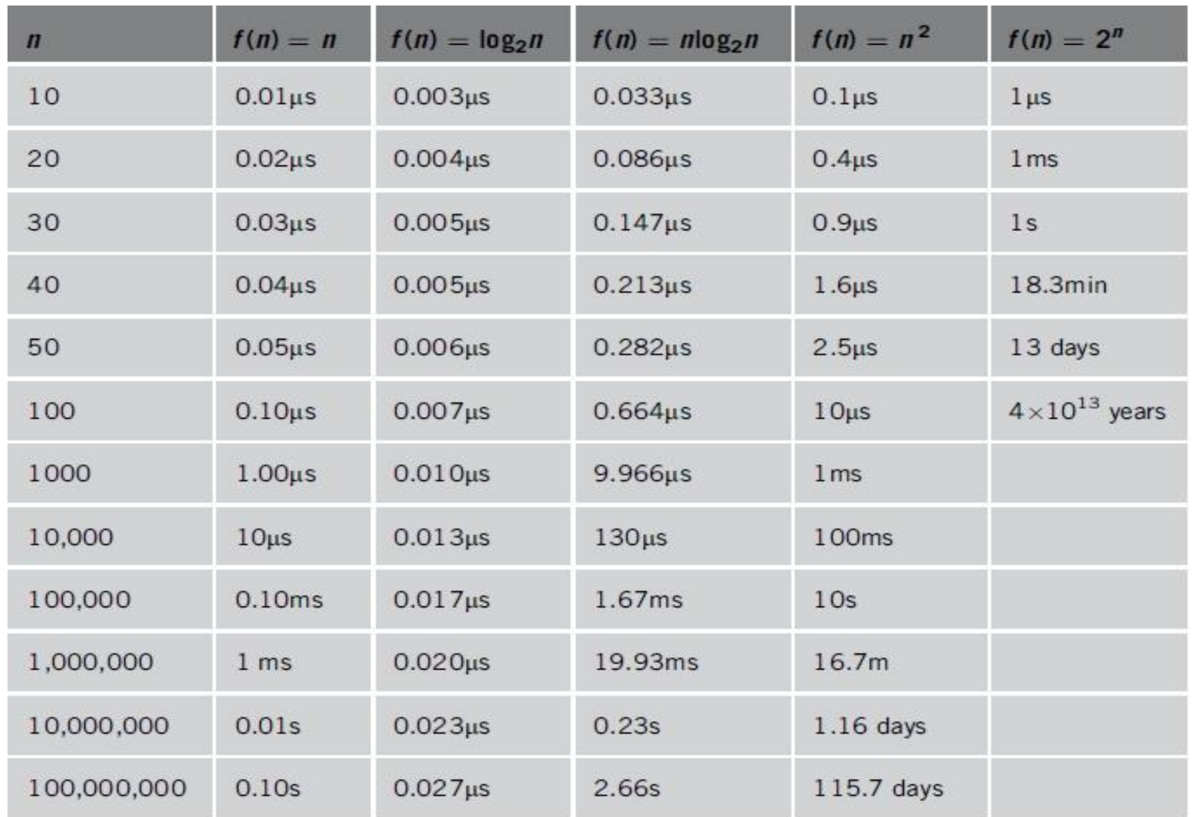
TABLE 1-2 Growth rates of various functions



n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Algorithm Analysis: The Big-O Notation (cont'd.)

TABLE 1-3 Time for *f*(*n*) instructions on a computer that executes 1 billion instructions per second (1 GHz)



n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n\log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01μs	0.003μs	0.033μs	0.1μs	1μs
20	0.02μs	0.004μs	0.086μs	0.4μs	1ms
30	0.03μs	0.005μs	0.147μs	0.9μs	1s
40	0.04μs	0.005μs	0.213μs	1.6μs	18.3min
50	0.05μs	0.006μs	0.282μs	2.5μs	13 days
100	0.10μs	0.007μs	0.664μs	10μs	4×10^{13} years
1000	1.00μs	0.010μs	9.966μs	1ms	
10,000	10μs	0.013μs	130μs	100ms	
100,000	0.10ms	0.017μs	1.67ms	10s	
1,000,000	1 ms	0.020μs	19.93ms	16.7m	
10,000,000	0.01s	0.023μs	0.23s	1.16 days	
100,000,000	0.10s	0.027μs	2.66s	115.7 days	

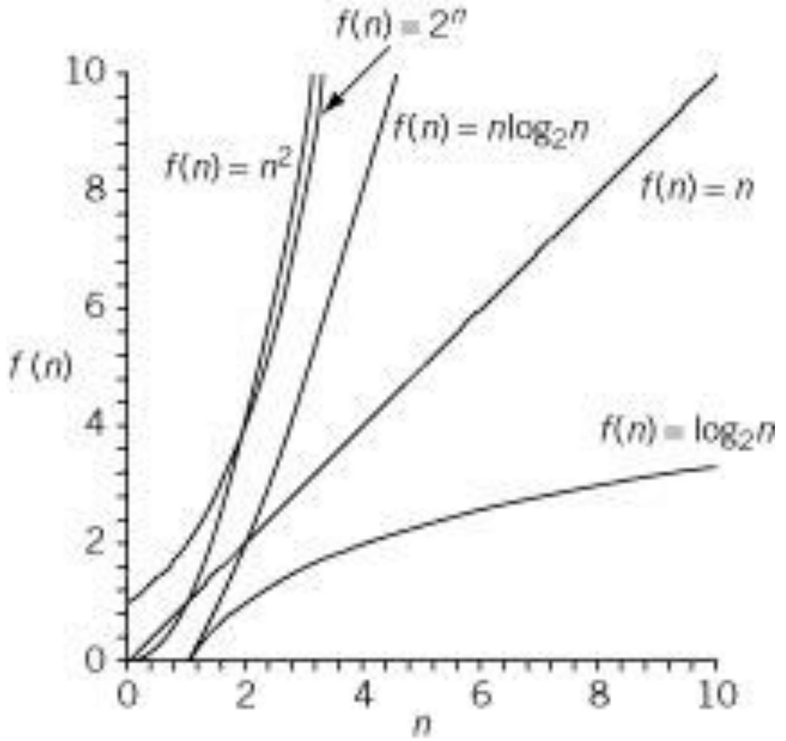


Figure 1-4 Growth rate of functions in Table 1-3

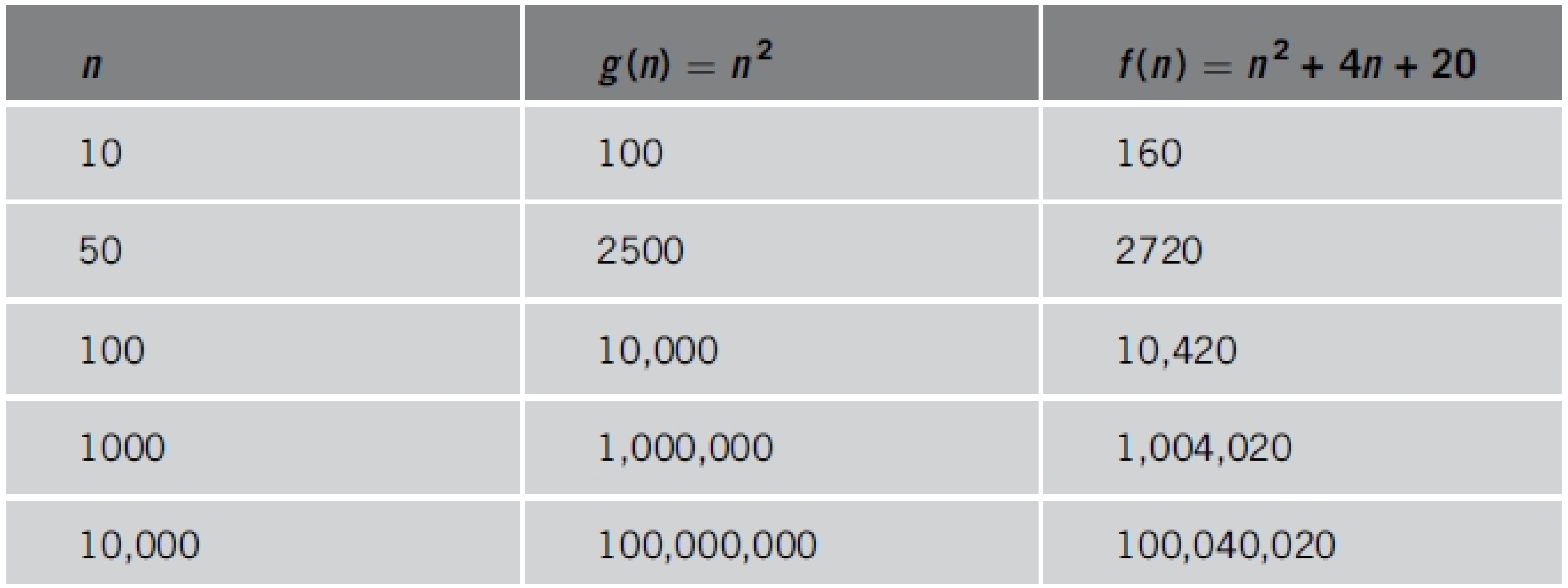
Algorithm Analysis: The Big-O Notation (cont'd.)

- Notation useful in describing algorithm behavior
  - Shows how a function $f(n)$ grows as n increases without bound
- Asymptotic
  - Study of the function f as n becomes larger and larger without bound
  - Examples of functions
    - $g(n)=n^2$ (no linear term)
    - $f(n)=n^2 + 4n + 20$

Algorithm Analysis: The Big-O Notation (cont'd.)

- As *n* becomes larger and larger
  - Term $4n + 20$ in $f(n)$ becomes insignificant
  - Term n^2 becomes dominant term

TABLE 1-4 Growth rate of n^2 and $n^2 + 4n + 20n$



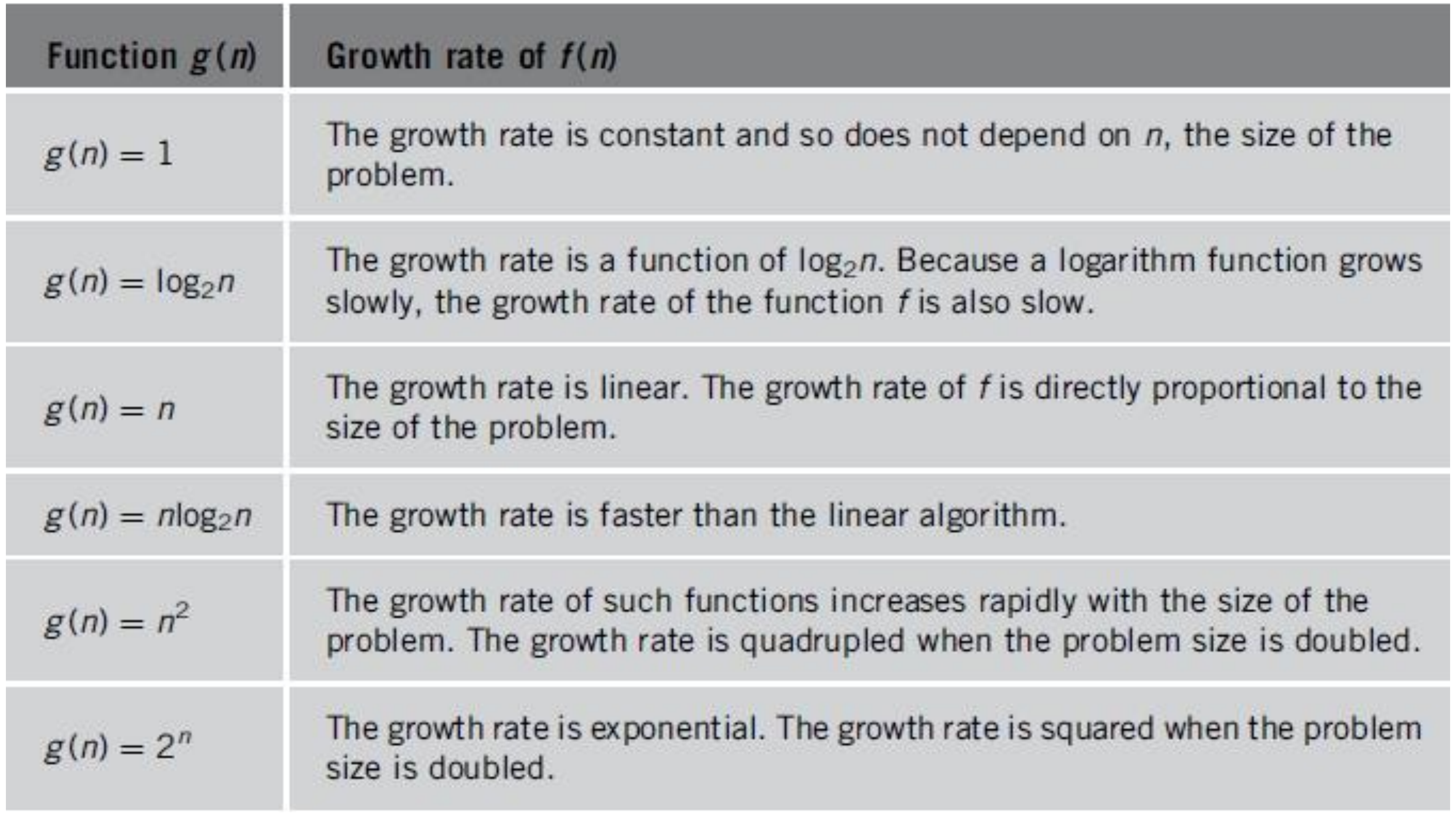
n	$g(n) = n^2$	$f(n) = n^2 + 4n + 20$
10	100	160
50	2500	2720
100	10,000	10,420
1000	1,000,000	1,004,020
10,000	100,000,000	100,040,020

Algorithm Analysis: The Big-O Notation (cont'd.)

- Algorithm analysis
  - If function complexity can be described by complexity of a quadratic function without the linear term
    - We say the function is of $O(n^2)$ or Big-O *of* n^2
- Let f and g be real-valued functions
  - Assume f and g nonnegative
    - For all real numbers n, $f(n) >= 0$ and $g(n) >= 0$
- $f(n)$ is Big-O of $g(n)$: written $f(n) = O(g(n))$
  - If there exists positive constants c and n_0 such that $f(n) <= cg(n)$ for all $n >= n_0$

Algorithm Analysis: The Big-O Notation

TABLE 1-5 Some Big-O functions that appear in algorithm analysis



Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on n, the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n\log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.