Chapter 1

Algorithms:
Efficiency, Analysis, and Order

Objectives

- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Define Worst case, average case, and best case complexity analysis of algorithms
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega

Problem?

1.1 Algorithms

- Applying a technique to a problem results in a step-by-step procedure for solving the problem. This step-by-step procedure is called an **algorithm**.
- A computer program is composed of individual modules, that solve specific tasks.
- These tasks are called problems.

Example 1.1

- Sort a list *S of n* numbers in nondecreasing order.
- Answer? Numbers in sorted sequence

Example 1.2

- Input : Array *S*, size *n*
- Determine whether $x \in S$
- Answer? Yes or No
- Sequential Search: n operations
- Binary Search: *lg n* + 1 operations

- ***Parameters :*** A problem may contain variables that are not assigned specific values in the statement of the problem.
- Example 1.1 : two parameters: S and n .
- Example 1.2 : three parameters: S, n, and x.
- It is not necessary in these two examples to make n one of the parameters because its value is uniquely determined by *S.*

- Because a problem contains parameters, it represents a class of problems, one for each assignment of values to the parameters.

- **instance** of the problem : Each specific assignment of values to the parameters

Example 1.3

- An instance of the problem (sorting) in Example 1.1 is
- S = [10, 7, 11, 5, 13, 8] and n=6

Example 1.4

- An instance of the problem (searching) in Example 1.2 is

- S = [10, 7, 11, 5, 13, 8] and n=6 and x=5

The solution to this instance is, “yes, *x is in S.”*

Algorithm 1.1 - Sequential Search

- Problem: Is $x \in S$ of n keys?
- Inputs (parameters): n, S (from 1 to n), and x.
- Outputs: the location of x in S (**0** if x is not in S).

```
void seqsearch (int  n,  const keytype S[ ],
                        keytype   x, index&   location )
{
   location  =  1;
   while ( location <=  n  &&  S [ location  ]  !=  x )
         location++;
    if  (location >  n  )
         location = 0;
}
```

pseudocode

- We can declare

Example

```
void example (int n)
{
 keytype S[2..n];
  .
  .
  .
 }
```

- We can use mathematical expressions or English-like descriptions

  if ($low \leq x \leq high$)

  rather than

  if (*low* <= *x* && *x* <= *high*) { }

  Exchange *x* and *y* rather than

  ```
  t = x;
  x = y;
  y = t;
  ```

- Sometimes we use the following nonstandard control structure:

```
repeat (n times) { . . . }
```

- We use the following, which also are not predefined C++ data types:

Data Type	**Meaning**
index	A variable used as a index
number	A variable could be int or float
bool	A variable can take “true” or “false”

- ***function*** *:* the algorithm returns a value
- *Otherwise,* we write the algorithm as a **void** function and use reference parameters to return values.
- the defined data type **keytype** for the items means the items are from any ordered set.
- If the parameter is not an array, it is declared with an ampersand (**&**) . This means that the parameter contains a value returned by the algorithm.
- we use **const** to indicate that the array does not contain values returned by the algorithm.

Algorithm 1.2 Add Array Members

- Inputs: +ve integer *n,* array of numbers ***S*** indexed from 1 to *n.*
- Outputs: ***sum,*** the sum of the numbers in S.

```
number  sum (int  n,  const number S [ ] )
{
   index  i;
   number   result;
   result   =  0;
   for  ( i  =  1;  i <=  n;   i++ )
          result   =  result  +  S [ i ];
    return result ;
}
```

Exchange Sort

- Exchange Sort works by comparing the number in the *ith* slot with *the* numbers in the (*i + 1*)*st t*hrough *nth* slots.
- Whenever a number in a given slot is found to be smaller than the one in the *ith* slot, the two numbers are exchanged.
- In this way, the smallest number ends up in the first slot after the first pass through **for-*i*** loop, the second-smallest number ends up in the secondslot after the second pass, and so on.

Algorithm 1.3 Exchange Sort

- Problem: Sort *n* keys in nondecreasing order.
- Inputs: +ve integer *n,* array of keys *S* indexed from *1* to *n.*
- Outputs: the array *S* containing the keys in nondecreasing order.

```
void exchangesort (int n, keytype S[ ] )
{
   index i , j;
   for ( i = 1; i <= n -1; i ++ )
         for ( j = i + 1; j <= n; j++ )
          if ( S[ j ] < S[ i ] )
             exchange S [ i ] and S[ j ] ;
}
```

$$\begin{array}{cccc} 4 & 3 & 2 & 1 \\ \uparrow & \uparrow & & \end{array}$$

i=1, j=i+1=2

4 3 2 1 (j=2) → 3 4 2 1 (j=3) → 2 4 3 1 (j=4) → **1 4 3 2**

i=2 , j=i+1=3

1 4 3 2 (j=3) → 1 3 4 2 (j=4) → 1 2 4 3

i=3 , j=i+1=4

1 2 4 3 (j=4) → 1 2 3 4

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

their product $C = A \times B$ is given by

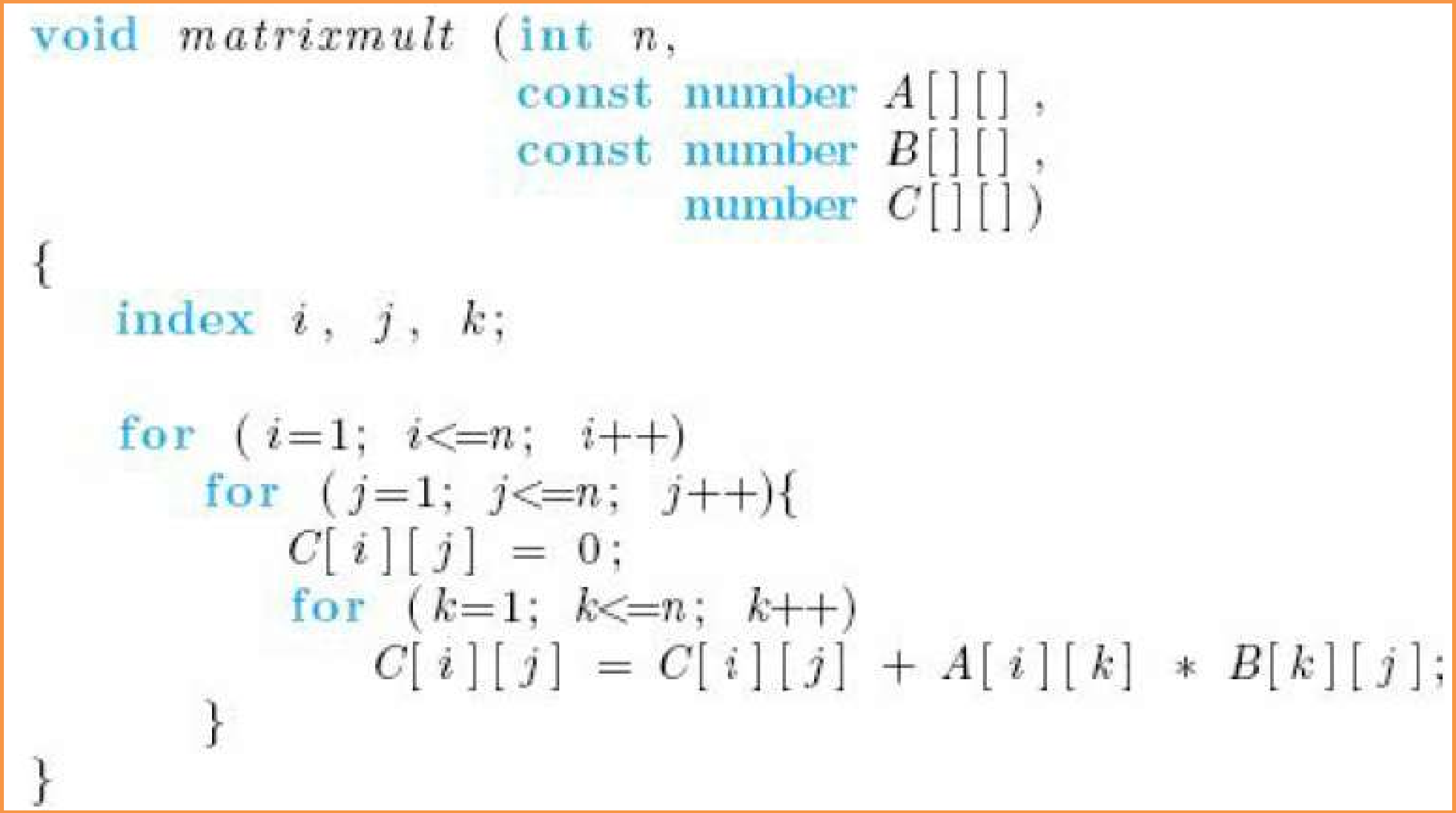
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}.$$

In general, if we have two $n \times n$ matrices A and B

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} \qquad \text{for } 1 \leq i, j \leq n.$$

Algorithm 1.4
Matrix Multiplication

- Problem: Find the product of two $n \times n$ matrices.
- Inputs: a +ve integer *n,* two-dimensional arrays of numbers *A and B,* rows and columns indexed from 1 to *n.*
- Outputs: a two-dimensional array of numbers *C,* - its rows and columns indexed from 1 to *n,* containing the product of *A* and *B.*



```
void matrixmult (int n,
                      const number A[][] ,
                      const number B[][] ,
                            number C[][])
{
   index i , j , k;

   for (i=1; i<=n; i++)
      for (j=1; j<=n; j++){
         C[i][j] = 0;
         for (k=1; k<=n; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
}
```

1.2 The Importance of Developing Efficient Algorithms

- Regardless of how fast computers become or how cheap memory gets, efficiency will always remain an important consideration.

1.2.1 Sequential Search Versus Binary Search

Algorithm 1.5 - Binary Search

- Problem: Determine whether *x* is in the **sorted** array *S of n* keys*.*
- Inputs: +ve int *n*, sorted array of keys *S* indexed from *1 to n*, a key *x.*
- Outputs: the location of *x* in *S* (*0* if *x* is not in *S*).

```
void  binsearch  ( int  n, const  keytype  S [ ],
                     keytype    x,   index&  location )
{
   index  low  , high, mid;
   low  = 1; , high =  n ;
   location  = 0 ;
   while (low <= high && location   ==0){
        mid=⌊( low + high )/2 ⌋;
        if ( x  == S [ mid ] )
           location  = mid;
        else if ( x  < S [ mid ] )
            high  = mid  - 1;
        else
            low = mid  + 1;
    }
}
```

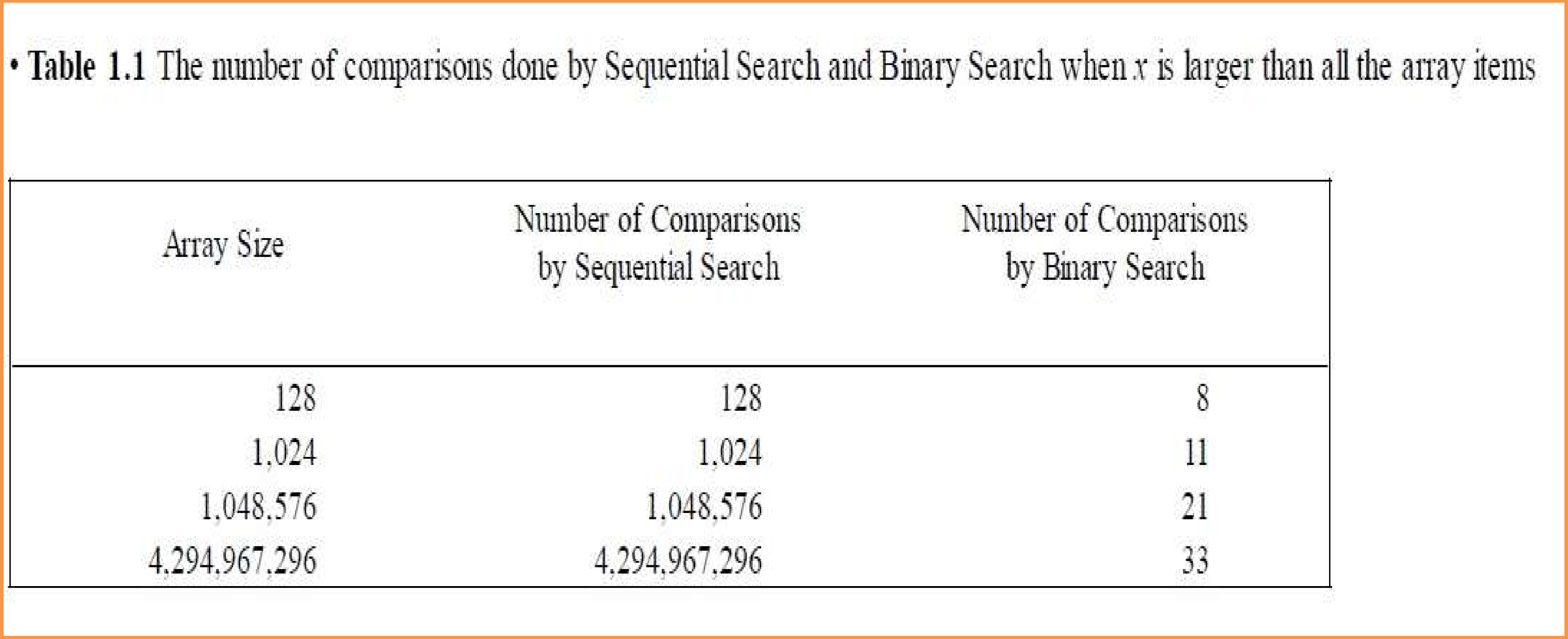
- We consider **the number of comparisons** done by each algorithm.
- Sequential Search *:* ***n*** **comparisons** if *x* is not in the array .
- There are two comparisons of *x* with *S* [*mid*] in each pass through the while loop.
- In an efficient assembler language implementation of the algorithm, *x* would be compared with *S[mid]* only .
- This means that there would be only one comparison of *x with S[mid]* in each pass through the while loop. We will assume the algorithm is implemented in this manner.

- n=32, *Assume that x* is larger than all the array
- The algorithm does six comparisons when *x is* larger than all the items in an array of size 32.
- low=1, high=32, mid=$\lfloor$ low+ high/2 $\rfloor$=16, high=32
- low=17, high=32, mid=$\lfloor$ (17+ 32)/2 $\rfloor$=24
- low=25, high=32, mid=$\lfloor$ (25+ 3)/ 2 $\rfloor$=28
- The algorithm does six comparisons
- …….

S[16]	S[24]	S[28]	S[30]	S[31]	S[32]
↑	↑	↑	↑	↑	↑
1st	2nd	3rd	4th	5th	6th

Notice that 6 =1+ *lg* 32

Suppose we double the size of the array so that it contains 64 items. Binary Search does only one comparison more Therefore, when x is larger than all the items in an array of size 64, Binary Search does seven comparisons. Notice that 7 = lg 64 + 1. In general, each time we double the size of the array we add only one comparison. Therefore, the number of comparisons done by Binary Search is lg n + 1.



• **Table 1.1** The number of comparisons done by Sequential Search and Binary Search when x is larger than all the array items

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

1.2.2 Fibonacci Sequence

- $Fib_0 = 0$
- $Fib_1 = 1$
- $Fib_n = Fib_{n-1} + Fib_{n-2}$
- Calculate the nth Fibonacci Term:
  - Recursive calculates $2^{n/2}$ terms
  - Iterative calculates n+1 terms

Algorithm 1.6

***nth Fibonacci Term* (*Recursive*)**

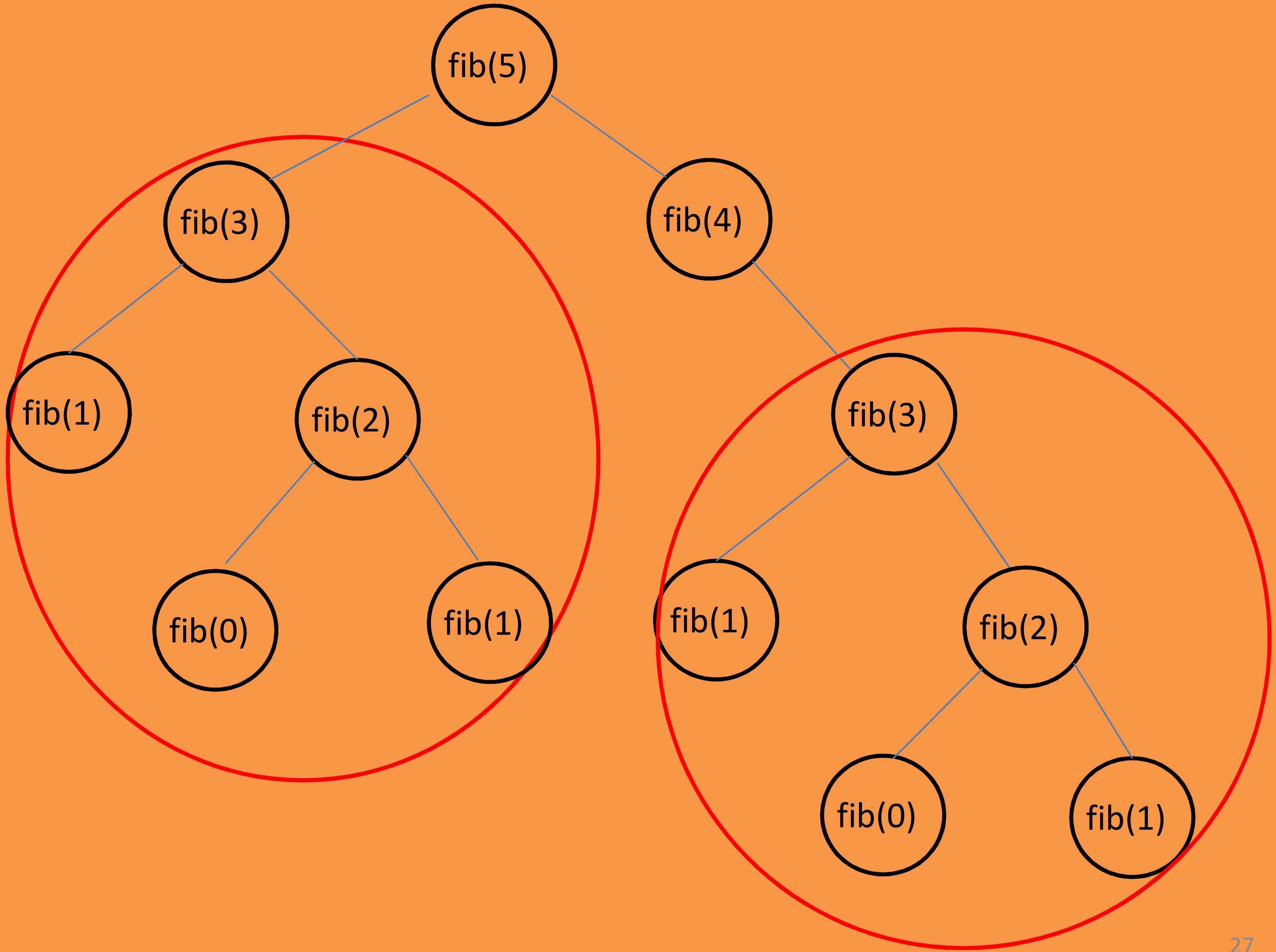
Problem: Determine the *nth* term in the Fib sequence.

Inputs: a nonnegative integer *n.*

Outputs: *fib,* the *nth* term of the Fibonacci sequence.

```
int  fib (int n)
{
          if (n <=1) return n;
          else  return  fib (n -1) + fin (n -2) ;
}
```

Recursion Tree for the 5th Fibonacci Term



- *T* (*n*) *:* the number of terms in the recursion tree for *n.*
- If the number of terms more than doubled every time n increased by 2, we would have the following for n even:

$$
\begin{aligned}
T(n) &> 2 \times T(n-2) \\
&> 2 \times 2 \times T(n-4) \\
&> 2 \times 2 \times 2 \times T(n-6) \\
&\vdots \\
&> \underbrace{2 \times 2 \times \ldots \times 2}_{n/2 \text{ times}} \times T(0)
\end{aligned}
$$

$T(n) > 2^{n/2}$ $(T(0)=1)$

Theorem 1.1

If *T* (*n*) is the number of terms in the recursion tree corresponding to Algorithm 1.6, then, for $n \geq 2$,

$T(n) > 2^{n/2}$

Proof : See page 32(by induction on n)

let’s develop an efficient algorithm for computing the nth Fibonacci term. Recall that the problem with the recursive algorithm is that the same value is computed over and over. As Figure 1.2 shows, fib(2) is computed three times in determining fib(5). If when computing a value, we save it in an array, then whenever we need it later we do not need to recompute it.

Algorithm 1.7

nth Fibonacci Term (Iterative)

Problem: ....

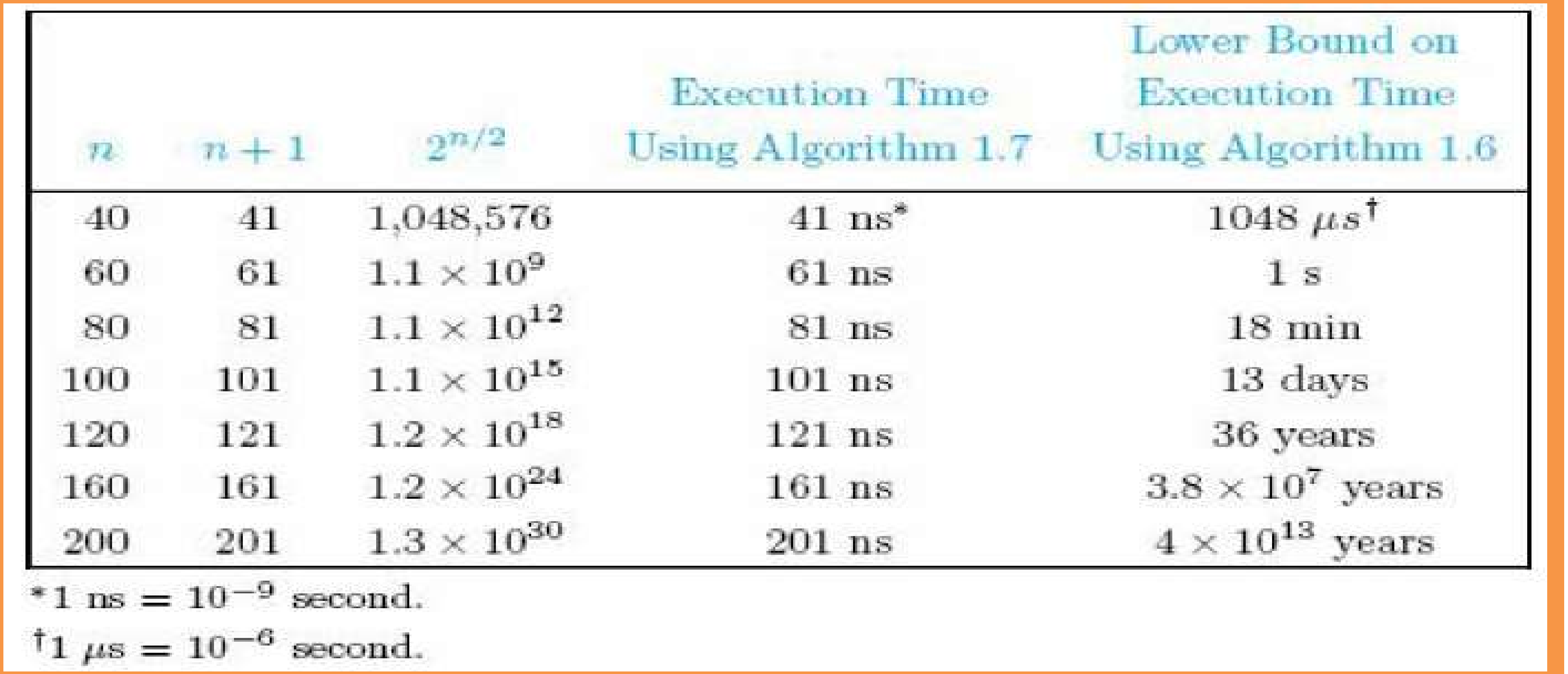
Inputs: n.

Outputs: *fib2*, the nth term in the Fibonacci sequence.

```
int  fib2 (int  n)
{
          index  i;
          int  f [0 . . n];
           f [0] = 0;
          if (n  >  0)  {
             f [1] = 1;
             for  ( i  =  2;  i <=  n;   i ++ )
                         f [ i ] =  f [i  - 1] +  f [i  - 2] ;
          };
          return  f [ n ] ;
}
```

- To determine *fib2* (*n*)*,* the previous algorithm computes every one of the first *n* terms just once.
- So it computes *n + 1* terms to determine the nth Fibonacci term.

Comparison of Recursive and Iterative Solutions



				Lower Bound on
			Execution Time	Execution Time
n	$n+1$	$2^{n/2}$	Using Algorithm 1.7	Using Algorithm 1.6
40	41	1,048,576	41 ns*	1048 μs†
60	61	1.1×10^9	61 ns	1 s
80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 years
160	161	1.2×10^{24}	161 ns	3.8×10^7 years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns $= 10^{-9}$ second.

†1 μs $= 10^{-6}$ second.

ns: nanosecond

µs: microsecond

Assume that one term can be computed in 10^{-9} *second.*

1.3 Analysis of Algorithms

- To determine how efficiently an algorithm solves a problem.

1.3.1 Complexity Analysis

- When analyzing the **efficiency** of an algorithm:
  - we do not count every instruction executed (depends on the programming language and the programmer).
- We want a measure that is independent of the computer.

- Define **Basic Operation** (instruction or group of instructions)
- **Time complexity analysis of an algorithm**: how many times the basic operation is done as a **function of the size of the input**.
- Consider Algorithms Sequential Search, Add Array Members, Exchange Sort, and Binary Search: *n,* the number of items in the array, is a simple measure of the size of the input. Therefore, we can call *n the input size.*
- Algorithm Matrix Multiplication, *n,* is a simple measure of the size of the input. Therefore, we can again call *n the input size.*

- In Algorithms 1.6 (*nth Fibonacci* Term, Recursive) ***n* is the input**; it is not the size of the input.
- *For* this algorithm, a reasonable measure of the size of the input is the number of symbols used to encode *n.*
- There is no hard-and-fast rule for choosing the basic operation. It is largely a matter of judgment and experience.
- We ordinarily do not include the instructions that make up the control structure (we do not include the instructions that increment and compare the index in order to control the passes through the while loop)

- Sometimes complexity depends on instance.
- Consider Sequential Search. If *x* is the first item in the array, the basic operation is done once, whereas if *x* is not in the array, it is done *n* times.
- ***T* (*n*)** (***every-case*** time complexity of the algorithm) : exists when the basic operation is always done the same number of times for every instance of size *n.*

Analysis of Algorithm 1.2 (Add Array Members)

Every-Case Time Complexity

- The *basic operation*: addition of an item to *sum.*
- Input size: *n,* the number of items in the array.
- Therefore, the basic operation is always done *n times and*

  T*(*n*) =*n

Analysis of Algorithm 1.3

Every-Case Time Complexity (Exchange Sort)

- Basic operation: the comparison of *S* [*j*] with *S* [*i*].
- Input size: *n,* the number of items to be sorted.
- there are always *n − 1* passes through the for-*i* loop.
  - first pass, there are *n − 1* passes through the for-*j* loop,
  - second pass there are *n − 2* passes through the *for-j* loop,
  - third pass there are *n−3* passes through the *for-j* loop, … ,
  - last pass, there is one pass through the for-*j* loop.
  - Therefore, total number of passes is given by

$$T(n)=(n-1)+(n-2)+\dots+1=(n-1)*n/2$$

Analysis of Algorithm 1.4

Every-Case Time Complexity (Matrix Multiplication)

- Basic operation: multiplication instruction in the innermost **loop.**
- Input size: *n,* the number of rows and columns*.*
- There are always *n* passes through the *for-i loop*, in each pass there are always n passes through the for-*j loop,* and in each pass there are always n passes through the *for-k loop.*
- Because the basic operation is inside the **for-*k loop,***

$T(n)=n\times n\times n= n^3$

If the algorithm does not have an every-case time complexity. This does not mean that we cannot analyze such algorithms,

- Worst-case time complexity W(n)
- Average-case time complexity A(n)
- Best-case time complexity B(n)

- The worst-case time complexity: ***W* (*n*)** is the maximum number of times the algorithm will ever do its basic operation for an input size of *n.*
- ***T* (*n*)** (***every-case*** time complexity of the algorithm) : exists when the basic operation is always done the same number of times for every instance of size *n.*
- ***If T* (*n*)** exists, then clearly ***W* (*n*) = *T* (*n*).**
- The following is an analysis of *W* (*n*) in a case in which *T* (*n*) does not exist.
- In sequential search algorithm if x is the first element in the array we need only one comparison operation.

- The basic operation is done *n* times, if *x* is the last item in the array or if *x* is not in the array.
- Therefore, W(n) = n

Average

- ***A* (*n*)** is the **average** (expected value) of the no of times the algorithm does the basic operation for an input size of *n*
- *A* (*n*) is called the ***average-case*** time complexity of the algorithm.
- If *T* (*n*) exists, then

  $A\ (n\ ) = T\ (n\ ).$

- To compute **A(n)**, we need to assign probabilities to all possible inputs of size n.
- Consider Algorithm 1.1.
- We will assume that if x is in the array, it is equally likely to be in any of the array slots, and the items in S are all distinct
- It is reasonable to assign equal probabilities to all array slots.

1.3 Analysis of Algorithms

Analysis of Algorithm 1.1 (Sequential Search)

- **Average-Case Time Complexity**

- Basic operation: comparison with *x.*
- Input size: *n*
- First, assume that *x* is i*n S,* where the items in S are all distinct
- For $1 \leq k \leq n$*,* the probability that *x* is in the kth array slot is ***1/n****.*
- If x is in the kth array slot, the number of times the basic operation is done to locate *x*) is *k.*

So the average time complexity :

$$A(n) = \sum_{k=1}^{n} \left(k \times \frac{1}{n} \right) = \frac{1}{n} \times \sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- Next we analyze the case in which *x* may **not be** in the array.
- *P* (*x* is in the *array*) = *p*
- **P**(*x* is in the kth slot i)=***p/n**, and*
- P(x is not in the array)= **1 – *p***.
- Recall that there are k passes through the loop if x is found in the kth slot, and n passes through the loop if x is not in the array.

$$
\begin{aligned}
A(n) &= \sum_{k=1}^{n}\left(k \times \frac{p}{n}\right) + n(1-p) \\
&= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1-\frac{p}{2}\right) + \frac{p}{2}.
\end{aligned}
$$

If $p = 1$, $A(n) = (n + 1)/2$, as before, whereas if $p = 1/2$, $A(n) = 3n/4 + 1/4$. This means that about 3/4 of the array is searched on the average.

- ***B*** (***n***) is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of *n.*
- *B* (*n*) is called the best-case time complexity of the algorithm,
- if T(n) exists, then ***B* (*n*) = *T* (*n*).**
- Because n ≥ 1, there must be at least one pass through Therefore, B(n)=1

- A **complexity function** can be any function that maps the positive integers to the nonnegative reals
- **Complexity function**: a function

$$f: N^+ \rightarrow R^{\geq 0}$$

Examples

$f(n) = n^3$

$g(n) = lg\ n$

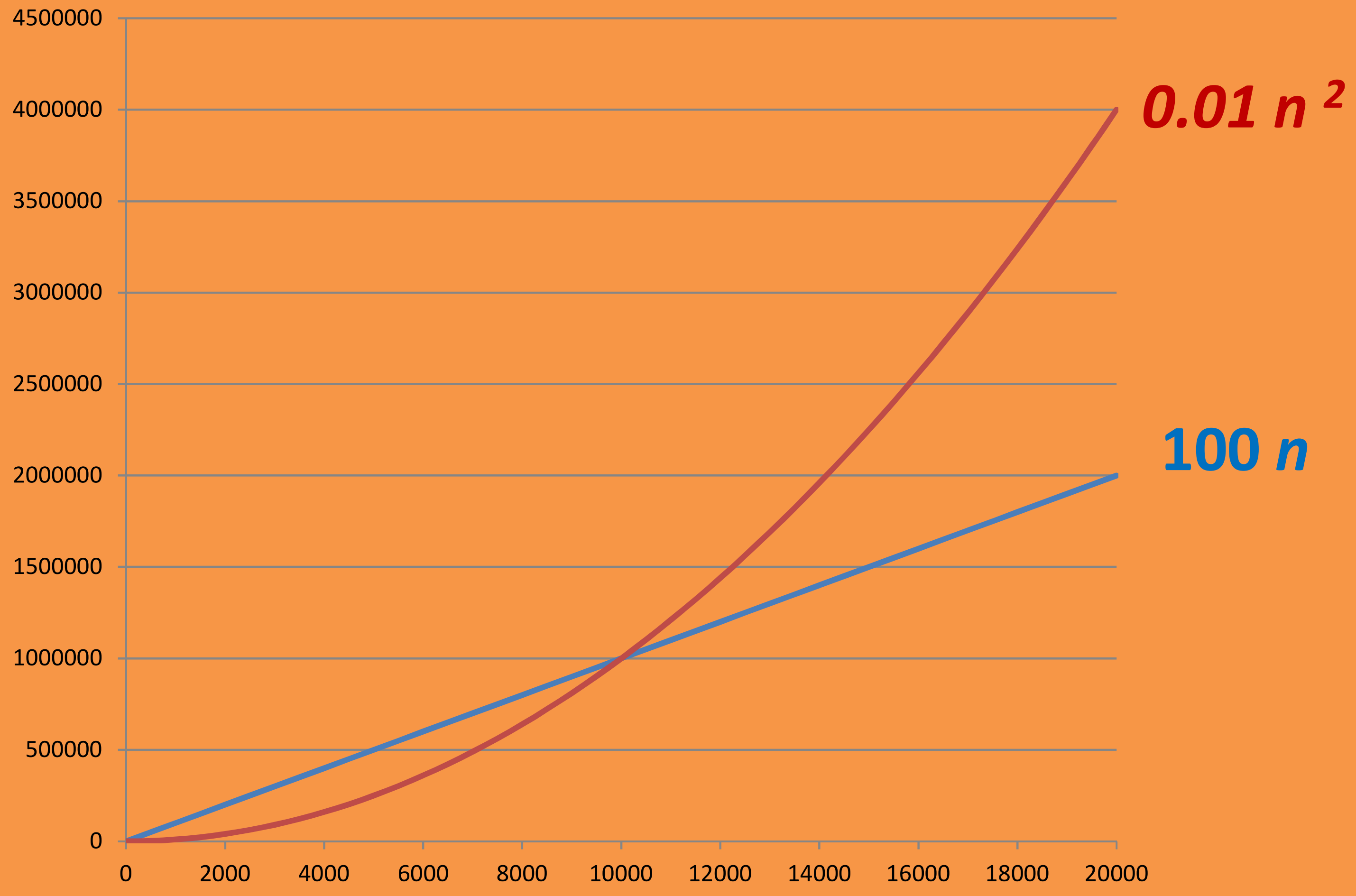
Analysis of Correctness

- we can analyze the correctness of an algorithm by developing a proof that the algorithm actually does what it is supposed to do. Although we will often informally show that our algorithms are correct and will sometimes prove that they are, you should.
- see Dijkstra (1976), Gries (1981), or Kingston (1990) for a comprehensive treatment of correctness.

1.4 Order

- An algorithm with a time complexity of ***n*** is more efficient than one with a time complexity of ***n*** ^{2} *for* sufficiently large values of n.
- Consider two algorithms A, and B for the same problem such that
  - **A**: every-case time complexities **100 *n***
  - ***B:*** every-case time complexities ***0.01 n*** ^{2}
- The first algorithm will be more efficient if

  $0.01n^2 > 100\,n \rightarrow n > 10000$

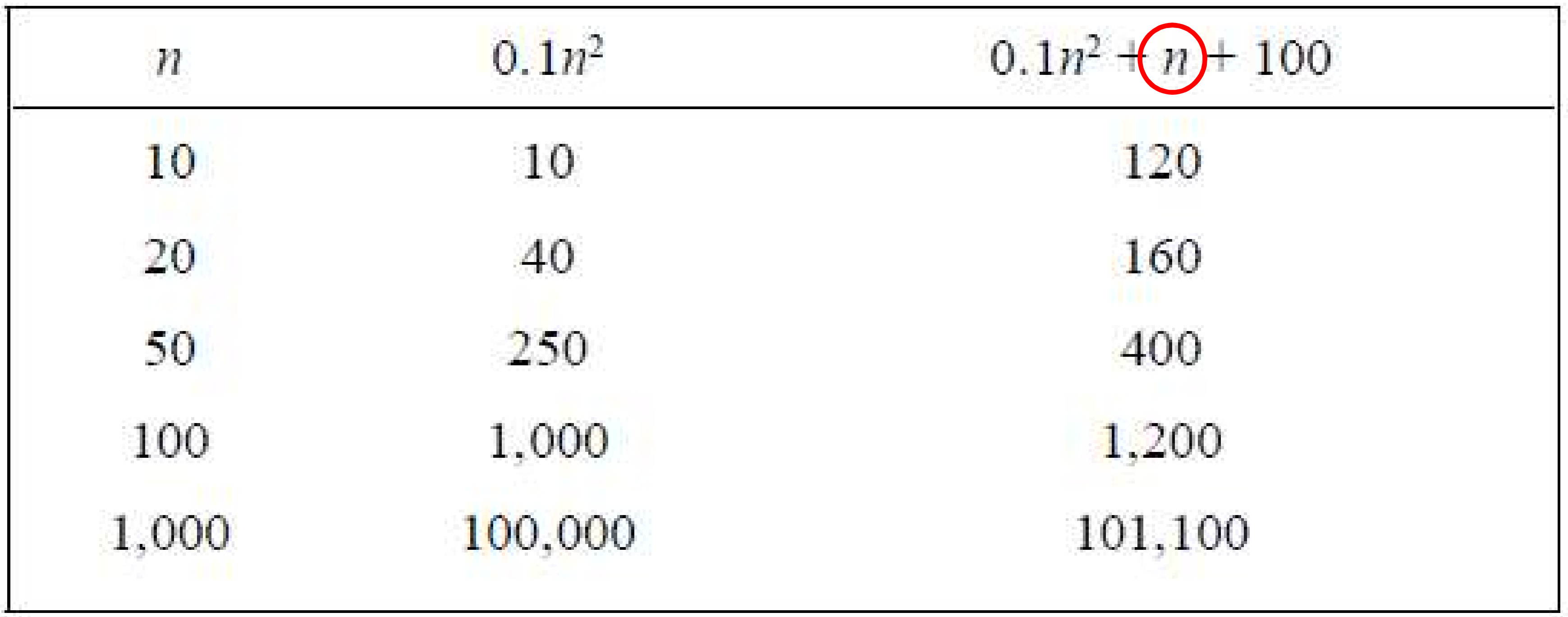

4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
$0.01\,n^2$
$100\,n$

- **linear-time algorithms *:*** time complexities are linear in the input size *n(n, 100n, 0.7n).*
- n^2 and $0.01n^2$: ***quadratic-time algorithms*** .
- Any linear-time algorithm is more efficient than any quadratic-time algorithm.

1.4.1 An Introduction to Order

- $5n^2$ *and* $5n^2 + 100$: ***pure quadratic functions*** **(**no linear term),
- $0.1\,n^2 + n + 100$: a ***complete quadratic function***
- **Complete quadratic function** can be classified with the **pure quadratic functions.**
- We should always be able to throw away low-order terms when classifying complexity functions.
- For example, it seems that we should be able to classify $0.1\,n^3 + 10n^2 + 5n + 25$ *with pure* ***cubic*** *functions.*

Table 1.3 shows that the quadratic term dominates this function.



n	$0.1n^2$	$0.1n^2 + n + 100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

- **$\Theta(n^2)$** *:* The set of all complexity functions that can be classified with pure quadratic functions.
- If $f \in \Theta(n^2)$, we say that the function is order of n^2.

Example

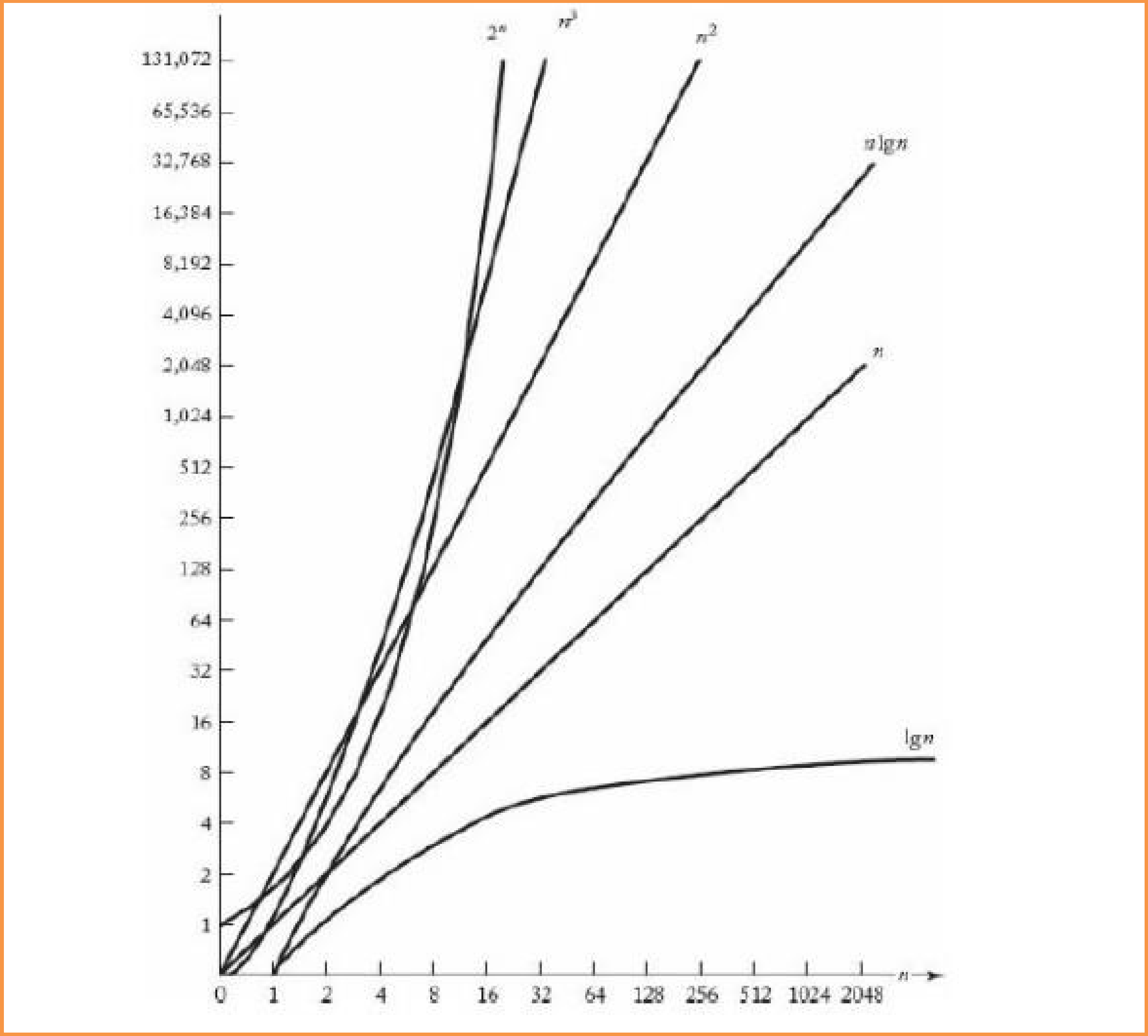
- The time complexity for Algorithm Exchange Sort is given by

  $T(n) = n(n - 1)/2.$

- *So* $T(n) \in \Theta(\boldsymbol{n^2})$.

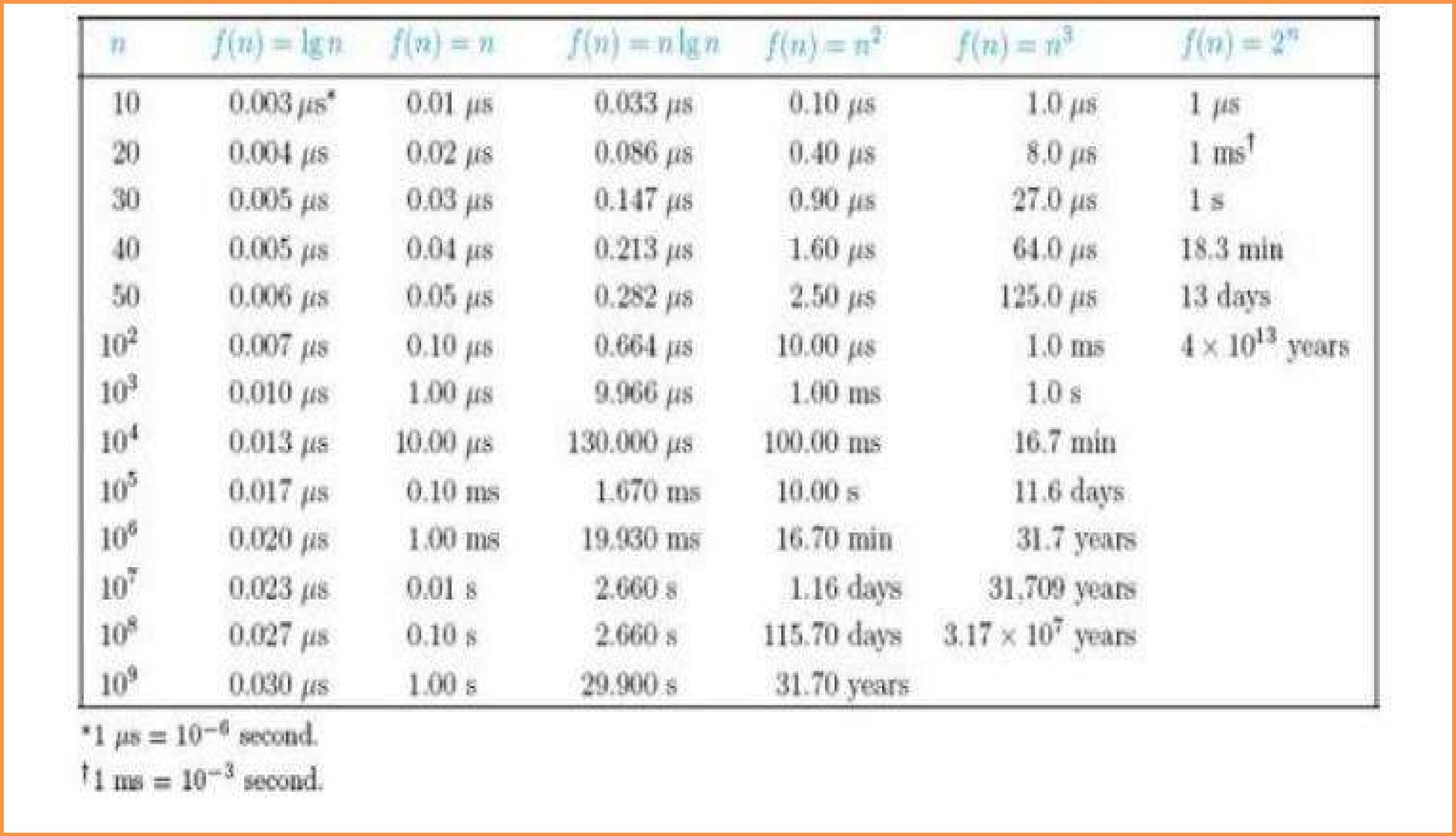
- quadratic-time algorithm : the algorithm's time complexity is in $\Theta(n^2)$. We also say that the algorithm is $\Theta(n^2)$.
- Exchange Sort is a quadratic-time algorithm.
- Similarly, the set of complexity functions that can be classified with pure cubic functions is called $\Theta(n^3)$, and functions in that set are said to be order of n^3.
- We will call these **sets complexity categories**.
- Examples of the most common complexity categories:

$\Theta(lg\ n)$, $\Theta(n)$, $\Theta(\mathrm{n\ lg}n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$,



2^n
n^3
n^2
$n \lg n$
n
$\lg n$
131,072
65,536
32,768
16,384
8,192
4,096
2,048
1,024
512
256
128
64
32
16
8
4
2
1
0
1
2
4
8
16
32
64
128
256
512
1024
2048
n

Execution times for algorithms with the given time complexities



n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s	
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days	
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years	
10^7	0.023 μs	0.01 s	2.660 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.660 s	115.70 days	3.17×10^7 years	
10^9	0.030 μs	1.00 s	29.900 s	31.70 years		

*1 μs $= 10^{-6}$ second.

†1 ms $= 10^{-3}$ second.

1.4.2 A Rigorous Introduction to Order

Definition

- $g(n) \in \boldsymbol{O}(\boldsymbol{f}(\boldsymbol{n}))$ *if* :

  there exists some positive real constant ***c*** and some nonnegative integer ***N*** such that

  $$\boxed{\text{for all } n \geq N,\ g(n) \leq c \times f(n)}$$

- If $g(n) \in O(f(n))$, we say that $g(n)$ is **big O** of $\boldsymbol{f}(\boldsymbol{n})$**.**
- If $g(n) \in O(n^2)$, then eventually g(n) lies beneath some pure quadratic function $c\,n^2$ on a graph.

Example 1.8

- Recall that the time complexity of Algorithm 1.3 (Exchange Sort)

$$T(n) = \frac{n(n-1)}{2}$$

$$\textit{because} \quad \textit{for} \quad n \geq 0,$$

$$\frac{n(n-1)}{2} \leq \frac{n(n)}{2} = \frac{1}{2}n^2$$

we can take *c = 1/2 and N = 0 to conclude that T(n) ∈ O (*n^2*).*

Example 1.9

- We show that $n^2 + 10n \in O(n^2)$.
- *Because, for $n \geq 0$,*

  $n^2 + 10n \leq n^2 + 10n^2 \leq 11n^2$
- We can also take c = 2 and N = 10
- we can say that eventually *g*(*n*) is at least as *good* as a pure quadratic function.
- A difficulty students often have with "big *O*" is that they erroneously think there is some unique *c* and unique *N* that must be found to show that one function is "big *O*" of another. This is not the case at all.

- *Example:* $n^2 \in O(n^2 + 10n)$. Because, for $n \geq 0$,

$$n2 \leq 1 \times (n2 + 10n)$$

we can take $c = 1$ and $N = 0$ to obtain our result.

- Note: the function inside "big *O*" does not have to be one of the simple functions

- Example: $n \in O(n^2)$.
- A complexity function need not have a quadratic term to be in $O(n2)$.
- Therefore, any logarithmic or linear complexity function is in O(n2).

any
logarithmic, linear, or quadratic complexity function is in O(n3), and so on.

Definition

- $f(n)$, $g(n)$: complexity functions.
- $g(n) \in \boldsymbol{\Omega}(\boldsymbol{f(n)})$ if there exists some +ve real constant $\boldsymbol{c}$ and some nonnegative integer $\boldsymbol{N}$ such that,

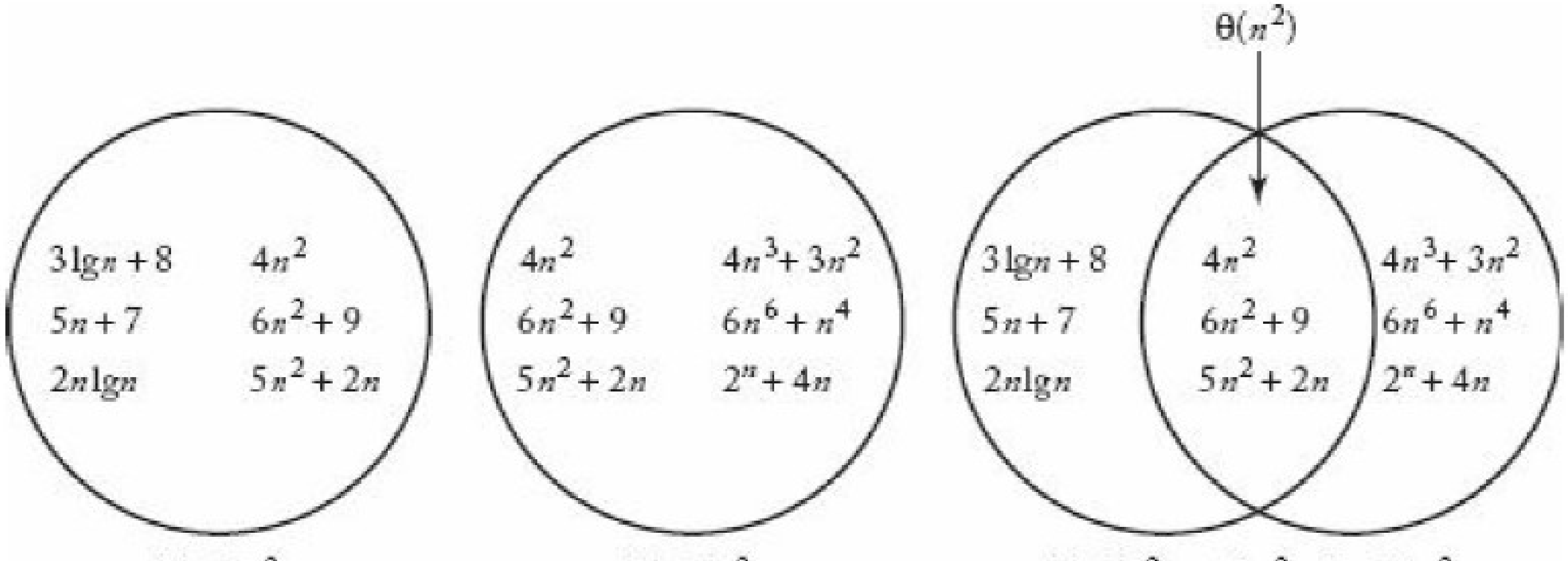
  for all $n \geq N$, $g(n) \geq c \times f(n)$

- If $g(n) \in \Omega(f(n))$, we say that $g(n)$ *is* ***omega of*** $f(n)$

Example:

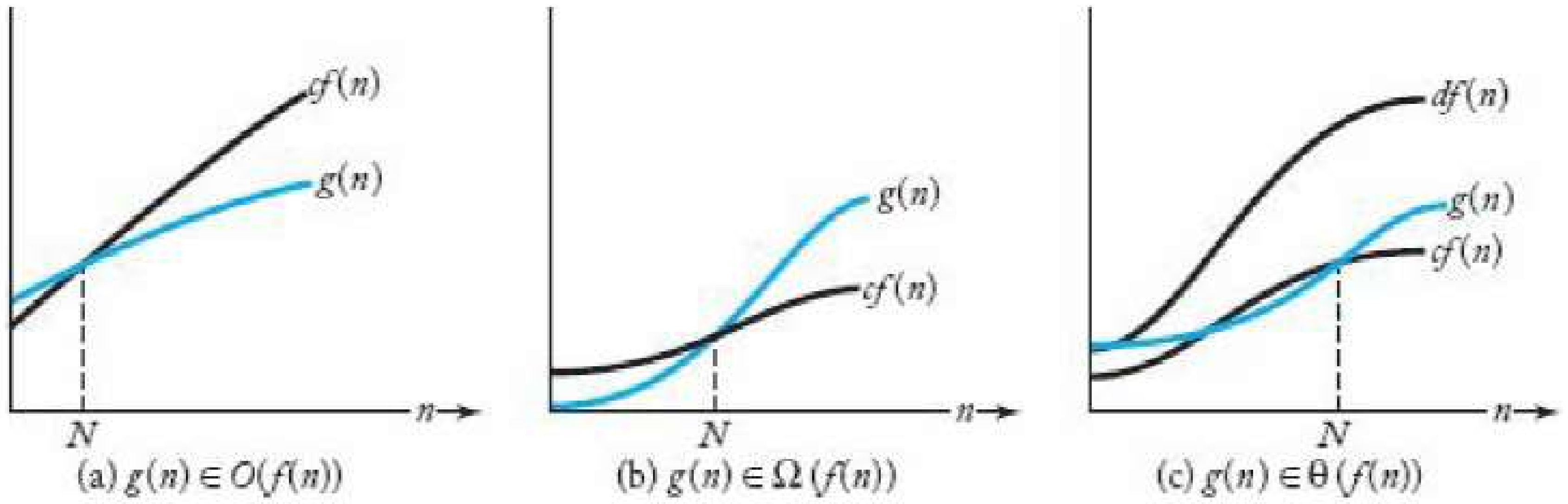
- $5\ n^{2} \in \Omega(n^{2})$.
- $n^{2} + 10n \in \Omega(n^{2})$.

N=0 , c=1



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\Theta(n^2) = O(n^2) \cap \Omega(n^2)$

Figure 1.4 Illustrating "big O," Ω, and Θ.



$$n^2 + 10n \le 2n^2.$$

Example 1.14

- Consider again Exchange Sort. We show that

$$T(n) = \frac{n(n-1)}{2} \in \Omega(n^2)$$

$$For \quad n \geq 2, \qquad n - 1 \geq \frac{n}{2}$$

$$Therefore \qquad For \quad n \geq 2,$$

$$\frac{n(n-1)}{2} \geq \frac{n}{2} \times \frac{n}{2} = \frac{1}{4} n^2$$

which means we can take *c = 1/4 and N = 2 to obtain our result.*

- If a function is in $\Omega(n^2)$, then eventually the function lies above some pure quadratic function on a graph. For the purposes of analysis, this means that eventually it is at least as *bad* as a pure quadratic function.

- $n^3 \in \Omega(n^2)$, $c = 1$ and $N = 1$.

Definition

- For a given complexity function $f(n)$,

  $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- If $g(n) \in \Theta(f(n))$, we say that $g(n)$ *is* ***order of f(n).***

- This means that $\boldsymbol{\Theta(f(n))}$ is the set of complexity functions g(n) for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

Example 1.16

Consider again Exchange Sort

$T(n) \in \boldsymbol{\Theta}(n^2)$

Example 1.17

Show that *n is not in* $\Omega(n^2)$ *by using* ***proof by contradiction.***

- Assuming that $n \in \Omega(n^2)$
- *Then there exists some positive constant c and* some nonnegative integer *N such that, for* $n \geq N$, $n >= c\, n^2$

  $1/c >= n$, *for* $n \geq N$
- *this inequality cannot hold*

Definition

Given $f(n)$, $\boldsymbol{o(f(n))}$ is the set of all complexity functions $g(n)$ satisfying the following: For every positive real constant c there exists a nonnegative integer N such that,

$$\textit{for}\ \text{all}\ n \geq N,\ \ g(n) \leq c \times f(n)$$

- This definition says that the bound must hold for every real positive constant *c,* which implies that the bound holds for arbitrarily small *c.*
- For example, if g(n) ∈ o(f(n)), there is an N such that,

  for $n > N$, $g(n) \leq 0.0001\, f(n)$

Example 1.18

$n \in o(n^2)$

Example 1.19

- *n* is not in o(5*n*). Proof by contradiction to show this, take c=1/6

Theorem 1.2

If $g(n) \in o(f(n))$, *then*

$$g(n) \in O(f(n)) - \Omega(f(n)).$$

- Proof: Because $g(n) \in o(f(n))$, *for every positive real constant c there exists an* N *such that, for **all*** n ≥ N,

$$g(n) \le c \times f(n)$$

which means that the bound certainly holds for some *c.*

$$g(n) \in O(f(n))$$

To show that $g(n)$ is not in $\Omega(f(n))$ we use proof by contradiction. *If* $g(n) \in \Omega(f(n))$, then there exists some real constant c > 0 and some N1 such that, for all ***n ≥ N1**,*

$$g(n) \ge c \times f(n)$$

But, because $g(n) \in o(f(n))$, *there exists some* **N2** *such that, for all n ≥ N2,* $g(n) \le c/2 \times f(n)$

Both inequalities would have to hold for all *n* greater than both *N*1 and *N*2. This contradiction proves that $g(n)$ cannot be in $\Omega(f(n))$.

- You may think that $o(f(n))$ and $O(f(n))-\Omega(f(n))$ must be the same set. This is not true. There are unusual functions that are in $O(f(n)) - \Omega(f(n))$ but that are not in $o(f(n))$. The following example illustrates this.

Example 1.20

Consider the function $g(n) = \begin{cases} n \text{ if } n \text{ is even} \\ 1 \text{ if } n \text{ is odd} \end{cases}$

- It is left as an exercise to show that g(n) $\in O(n) - \Omega(n)$.
- But g(n) is not in o(n)

Properties of Order:

1. $g(n) \in O(f(n))$ if and only if $f(n) \in \Omega(g(n))$.

2. $g(n) \in \Theta(f(n))$ if and only if $f(n) \in \Theta(g(n))$.

3. If $b > 1$ *and* $a > 1$, *then* $\log_a n \in \Theta(\log_b n)$.

- This implies that all logarithmic complexity functions are in the same complexity category.
- We will represent this category by $\Theta(\lg n)$.

4. If $b > a > 0$, then $a^n \in o((b^n))$

5. For all $a > 0$, $a^n \in o((n!))$

6. Consider the following ordering of complexity categories:

$\Theta(\lg n)$, $\Theta(n)$, $\Theta(n \lg n)$, $\Theta(n^2)$, $\Theta(n^j)$, $\Theta(n^k)$, $\Theta(a^n)$, $\Theta(b^n)$ $\Theta(\boldsymbol{n!})$, where $k > j > 2$ and $b > a > 1$.

If a complexity function $g(n)$ is in a category that is to the left of the category containing $f(n)$, then $g(n) \in o(f(n))$

7. If $c \geq 0$, $d > 0$, $g(n) \in O(f(n))$, and $h(n) \in \Theta(f(n))$, then $c\,g(n) + d\,h(n) \in \Theta(f(n))$

Example 1.21

- Property 3 : all logarithmic complexity functions are in the same complexity category.
- For example,

  $\Theta\ (log_4\ n\ )=\Theta\ (\lg_2 n)$.

Example 1.23

- Properties 6 and 7 can be used repeatedly. For example, we can show that $5n + 3\ lg\ n + 10n\ lg\ n + 7n^2 \in \Theta(n^2)$,

Theorem 1.3

We have

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases} c \;\; implies \;\; g(n)\in\Theta(f(n)) \; if \;\; c>0 \\ 0 \;\; implies \;\; g(n)\in o(f(n)) \\ \infty \;\; implies \;\; f(n)\in o(g(n)) \end{cases}$$

Example 1.24

Theorem 1.3 implies that

$$\frac{n^2}{2} \in o(n^3)$$

because

$$\lim_{n\to\infty} \frac{n^2/2}{n^3} = \lim_{n\to\infty} \frac{1}{2n} = 0$$

Example 1.25

- Theorem 1.3 implies that, for $b > a > 0$, $a^n \in o(b^n)$

  Because

  $\text{Lim}_{n \to \infty} \; a^n / b^n = 0$ (The limit is 0 because $0 < a/b < 1$)

  From th. 1.3 , $a^n \in o(b^n)$