Ch1. Algorithms: Efficiency, Analysis, and Order

Contents

Algorithms:
Efficiency, Analysis, and Order

Algorithms

Preliminaries

- The word of 'problem'
  - Definition of problem
    - A computer program is composed of individual modules, understandable by a computer, that solve specific tasks (such as sorting).
    - **Our concern in this text is not the design of entire programs, but rather the design of the individual modules that accomplish the specific tasks. These specific tasks are called problems**.
  - Examples of the 'problem'

Example1.

Sort list *S* of *n* numbers in nondecreasing order. The answer is the numbers in sorted sequence.

By a list we mean a collection of items arranged in a particular sequence. For example,

$$S = [10, 7, 11, 5, 13, 8]$$

is a list of six numbers in which the first number is 10, the second is 7, and so on. In this example, we say the list is to be sorted in "nondecreasing order" instead of increasing order to allow for the possibility that the same number may appear more than once in the list.

Preliminaries

- The word of 'problem' (contd.)
  - Examples of the 'problem' (contd.)

Example2.

Determine whether the number *x* is in the list *S* of n numbers. The answer is *yes* if *x* is in *S* and *no* if it is not.

A problem may contain variables that are not assigned specific values in the statement of the problem. These variables are called parameters to the problem. In Example1, there are two parameters: *S* (the list) and *n* (the number of items in *S*). In Example 2, there are three parameters: *S*, *n*, and the number *x*. It is not necessary in these two examples to make *n* one of the parameters because its value is uniquely determined by *S*. However, making *n* a parameter facilitates our descriptions of problems.

Preliminaries

- The word of 'solution'
  - Definition of the 'solution'
    - Because a problem contains parameters, it represents a class of problems, one for each assignment of values to the parameters.
    - **Each specific assignment of values to the parameters is called an instance of the problem. A solution to an instance of a problem is the answer to the question asked by the problem in that instance.**

Example3.

An instance of the problem in Example1 is S = [10, 7, 11, 5, 13, 8] and n = 6

The solution of this instance is [5, 7, 8, 10, 11, 13]

Example4.

An instance of the problem in Example2 is S = [10, 7, 11, 5, 13, 8], n = 6 and x = 5

The solution of this instance is, "*yes*, x is in S"

Preliminaries

- The word of 'algorithm'
  - Definition of the 'algorithm'
    - We can find the solution to the instance in Example 3 by inspecting *S* and allowing the mind to produce the sorted sequence by cognitive steps that cannot be specifically described. This can be done because *S* is so small that at a conscious level, the mind seems to scan *S* rapidly and produce the solution almost immediately (and therefore one cannot describe the steps the mind follows to obtain the solution).
    - **To produce a computer program that can solve all instances of a problem, we must specify a general step-by-step procedure for producing the solution to each instance. This step-by-step procedure is called an algorithm.**
  - Examples of the 'algorithms'

Example5.

Starting with the first item in *S*, compare x with each item in *S* in sequence until *x* is found or until *S* is exhausted. If *x* is found, answer *yes*; if *x* is not found, answer *no*.

⬇ Problems of descriptive algorithms

First, it is difficult to write a complex algorithm this way, and even if we did, a person would have a difficult time understanding the algorithm.
Second, it is not clear how to create a computer language description of an algorithm from an English language description of it.

Preliminaries

- The word of 'algorithm' (contd.)
  - Examples by pseudocode
    - The following algorithm represents the list *S* by an array and, instead of merely returning *yes* or *no*, returns *x*'s location in the array if *x* is in *S* and returns 0 otherwise.

Sequential Search

Problem: Is the key *x* in the array *S* of *n* keys?

Inputs (parameters): positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, 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;
}
```

Preliminaries

- The word of 'algorithm' (contd.)
  - Differences between pseudocode and C/C++ code in array
    - We allow variable-length two-dimensional arrays as parameters to routines.

```
void seqsearch (int n,
                const keytype S[ ],
                keytype x,
                index& location)
```

    - We declare local variable-length arrays.

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

    - We allow mathematical expressions or English-like descriptions than we could using actual C++ instructions

```
if (low ≤ x ≤ high) {
      .
      .
      .
}
```

rather than

```
if (low <= x && x <= high){
      .
      .
      .
}
```

Preliminaries

- The word of 'algorithm' (contd.)
  - Differences between pseudocode and C/C++ code in array (contd.)
    - We allow mathematical expressions or English-like descriptions than we could using actual C++ instructions. (contd.)

```
exchange x and y;        rather than    temp = x;
                                        x = y;
                                        y = temp;
```

    - Besides the data type keytype, we often use the following, which also are not predefined C++ data types

Data Type	Meaning
index	An integer variable used as index
bool	A variable that can take the values "true" or "false"
number	A variable that could be defined as integral (**int**) or real (**float**)

    - Sometimes we use the following nonstandard control structure

```
repeat (n times){
  ⋮
}
```

Preliminaries

- The word of 'algorithm' (contd.)
  - Differences between pseudocode and C/C++ code in array (contd.)
    - We allow non-standard logical operators and certain relational operators unfamiliar.

Operator	C++ symbol
and	&&
or	||
not	!

Comparison	C++ code
$x = y$	$(x == y)$
$x \neq y$	$(x! = y)$
$(x \leq y)$	$(x <= y)$
$x \geq y$	$(x >= y)$

    - We allow no type in front of the function name

Add Array Members

Problem: Add all the numbers in the array *S* of *n* numbers.

Inputs: positive 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;
}
```

Preliminaries

- The word of 'algorithm' (contd.)
  - Example of matrix multiplication with two 2 x 2 matrices

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

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

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

For example,

$$\begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 2 \times 5 + 3 \times 6 & 2 \times 7 + 3 \times 8 \\ 4 \times 5 + 1 \times 6 & 4 \times 7 + 1 \times 8 \end{bmatrix} = \begin{bmatrix} 28 & 38 \\ 26 & 36 \end{bmatrix}.$$

In general, if we have two $n \times n$ matrices A and B, their product C is given by

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

Preliminaries

- The word of ‘algorithm’ (contd.)
  - Example of matrix multiplication with two 2 x 2 matrices (contd.)

Matrix Multiplication

Problem: Determine the product of two $n \times n$ matrices.

Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from 1 to n.

Outputs: a two-dimensional array of numbers C, which has both 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];
      }
}
```

The Importance of Developing Efficient Algorithms

Sequential Search vs. Binary Search

- Binary Search Algorithm

Binary Search

Problem: Determine whether x is in the sorted array S of n keys.

Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

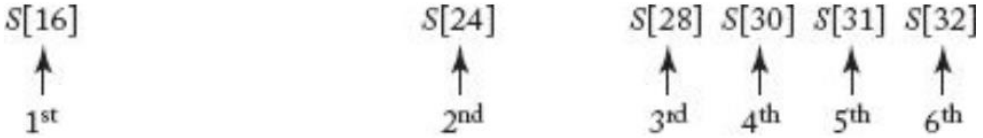
Outputs: *location*, the location of x in S (0 if x is not in S).

```
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;
   }
}
```

Sequential Search vs. Binary Search

- Binary Search Algorithm (contd.)
  - Two comparison or one comparison in the while loop?
    - There are two comparisons of x with S[mid] in each pass through the while loop (except when x is found). In an efficient assembler language implementation of the algorithm, x would be compared with S[mid] only once in each pass, the result of that comparison would set the condition code, and the appropriate branch would take place based on the value of the condition code. This means that there would be only one comparison of x with S[mid] in each pass through the while loop.
  - Number of comparison in 32 consecutive numbers of the array parameter on binary search algorithm



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

Array Size	# of Comparison by Sequential Search	# of Comparison by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,2968	33

The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence
  - In a recursive manner

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_n &= f_{n-1} + f_{n-2} \qquad \text{for } n \geq 2 .
\end{aligned}
$$

  - Computing the first few terms

$$
\begin{aligned}
f_2 &= f_1 + f_0 = 1 + 0 = 1 \\
f_3 &= f_2 + f_1 = 1 + 1 = 2 \\
f_4 &= f_3 + f_2 = 2 + 1 = 3 \\
f_5 &= f_4 + f_3 = 3 + 2 = 5, \text{ etc.}
\end{aligned}
$$

The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - Pseudo code

_n_th Fibonacci Term (Recursive)

Problem: Determine the *n*th term in the Fibonacci sequence.

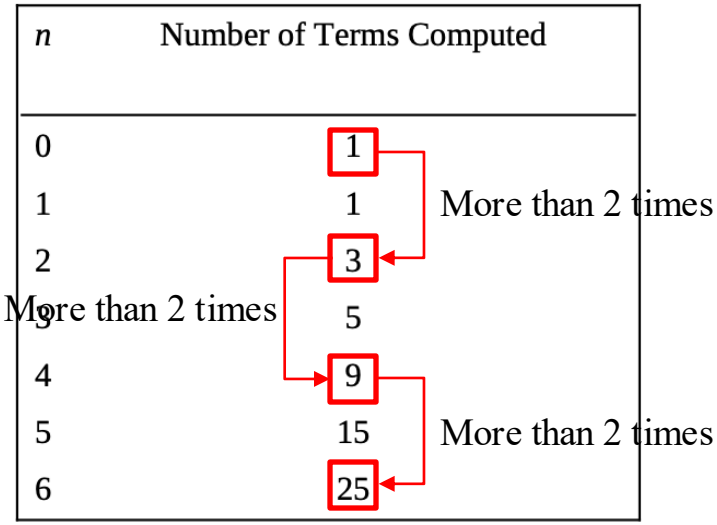
Inputs: a nonnegative integer *n*.

Outputs: *fib*, the *n*th term of the Fibonacci sequence.

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

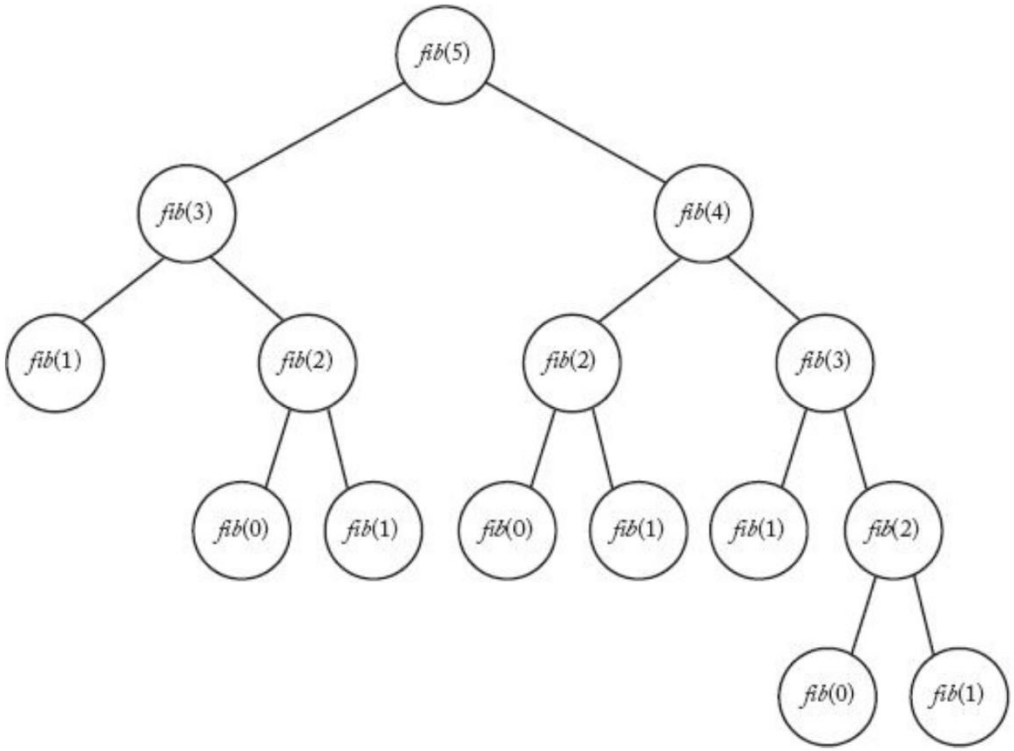
The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - To prove inefficiency of the algorithm in recursive manner by recursion tree
    - Notice that in the case of the first seven values, the number of terms in the tree more than doubles every time n increases by 2.



The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - To prove inefficiency of the algorithm in recursive manner by recursion tree (cont.)



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

The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - Theorem
    - If *T(n)* is the number of terms in the recursion tree corresponding to the Fibonacci algorithm, then, for $n \geq 2$,

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

  - Proof
    - Induction base: We need two base cases because the induction step assumes the results of two previous cases. For n = 2 and n = 3,

$$T(2) = 3 > 2 = 2^{2/2}$$

$$T(3) = 5 > 2.8323 \approx 2^{3/2}$$

    - Induction hypothesis: One way to make the induction hypothesis is to assume that the statement is true for all $m < n$. Then, in the induction step, show that this implies that the statement must be true for n. This technique is used in this proof. Suppose for all m such that $2 \leq m < n$

$$T(m) > 2^{m/2}.$$

    - Induction step: We must show that $T(n) > 2^{n/2}$. The value of $T(n)$ is the sum of $T(n-1)$ and $T(n-2)$ plus the one node at the root. Therefore,

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &> 2^{(n-1)/2} + 2^{(n-2)/2} + 1 \qquad \text{by induction hypothesis} \\ &> 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{(\frac{n}{2})-1} = 2^{n/2}. \end{aligned}$$

The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - Pseudo code in iterative manner

nth Fibonacci Term (Iterative)

Problem: Determine the *n*th term in the Fibonacci sequence.

Inputs: a nonnegative integer *n*.

Outputs: *fib*2, the *n*th 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];
}
```

The Fibonacci Sequence

- Algorithm to Compute the *n*th Term of the Fibonacci Sequence (cont.)
  - Pseudo code in iterative manner (cont.)
    - The algorithm can be written without using the array *f* because only the two most recent terms are needed in each iteration of the loop.
    - To determine *fib*2(*n*), the previous algorithm computes every one of the first $n + 1$ terms just once. So it computes $n + 1$ terms to determine the *n*th Fibonacci term.
      - Assumption that one term can be computed in 1 ns.

n	$n+1$	$2^{n/2}$	Execution Time Using Algorithm 1.7	Lower Bound on Execution Time 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.

Analysis of Algorithms

Analysis of Algorithms

- First Step for Complexity Analysis (Determining the Input Size)
  - DO NOT analyze the efficiency of an algorithm in terms of time by dependent factors
    - We do not determine the actual number of CPU cycles because this depends on the particular computer on which the algorithm is run.
    - We do not even want to count every instruction executed, because the number of instructions depends on the programming language used to implement the algorithm and the way the programmer writes the program.
  - DO analyze the efficiency of an algorithm in terms of time by independent factors
    - In general, the running time of an algorithm increases with the size of the input, and the total running time is roughly proportional to how many times some basic operation (such as a comparison instruction) is done.
    - We therefore analyze the algorithm's efficiency by determining the number of times some basic operation is done as a function of the size of the input.
    - For many algorithms it is easy to find a reasonable measure of the size of the input, which we call the *input size*.
      - NOT the *size* of input
    - If we use binary representation, the input size will be the number of bits it takes to encode n, which is $lg\lfloor n+1 \rfloor$.

$$n = 13 = \underbrace{1101}_{4\text{ bits}}{}_2$$

      - Therefore, the size of the input $n = 13$.

Analysis of Algorithms

- Second Step for Complexity Analysis (Picking Basic Operation)
  - DO pick some instruction or group of instructions
    - such that the total work done by the algorithm is roughly proportional to the number of times this instruction or group of instructions, basic operation, is done.
    - For example, *x* is compared with an item *S* in each pass through the loops in the sequential search or binary search algorithms. Therefore, the compare instruction is a good candidate for the basic operation in each of these algorithms.
    - By determining how many times in the Algorithms do this basic operation for each value of *n*, we gained insight into the relative efficiencies of the two algorithms.

- General Manner to Complexity Analysis
  - Time complexity analysis of an algorithm
    - The determination of how many times the basic operation is done for each value of the input size.
    - Although we do not want to consider the details of how an algorithm is implemented, we will ordinarily assume that the basic operation is implemented as efficiently as possible.
  - DO NOT include the instructions that increment and compare the index in order to control the passes through the while loop.
    - Sometimes it suffices simply to consider one pass through a loop as one execution of the basic operation.

Analysis of Algorithms

- General Manner to Complexity Analysis (cont.)
  - Time complexity analysis of an algorithm
    - The determination of how many times the basic operation is done for each value of the input size.
    - Although we do not want to consider the details of how an algorithm is implemented, we will ordinarily assume that the basic operation is implemented as efficiently as possible.
  - DO NOT include the instructions that increment and compare the index in order to control the passes through the while loop.
    - Sometimes it suffices simply to consider one pass through a loop as one execution of the basic operation.
  - May want to consider two different basic operations
    - For example, in an algorithm that sorts by comparing keys, we often want to consider the comparison instruction and the assignment instruction each individually as the basic operation.
    - we have two distinct basic operations, one being the comparison instruction and the other being the assignment instruction.
    - Therefore, we can gain more insight into the efficiency of the algorithm by determining how many times each is done.

Analysis of Algorithms

- Every-Case Time Complexity (Add Array Members)
  - Example of 'Add Array Members'
    - Basic operation: the addition of an item in the array to sum.
    - Input size: n, the number of items in the array.
    - Regardless of the values of the numbers in the array, there are *n* passes through the for loop. Therefore, the basic operation is always done *n* times and

$$T(n) = n$$

Analysis of Algorithms

- Every-Case Time Complexity
  - Example of 'Exchange Sort'
    - As mentioned previously, in the case of an algorithm that sorts by comparing keys, we can consider the comparison instruction or the assignment instruction as the basic operation. We will analyze the number of comparisons here.
    - Basic operation: the comparison of $S[j]$ with $S[i]$.
    - Input size: n, the number of items to be sorted.
    - We must determine how many passes there are through the **for**-*j* loop. For a given *n* there are always $n - 1$ passes through the **for**-*i* loop. In the first pass through the **for**-*i* loop, there are $n - 1$ passes through the **for**-*j* loop, in the second pass there are $n - 2$ passes through the **for**-*j* loop, in the third pass there are $n - 3$ passes through the **for**-*j* loop, … , and in the last pass there is one pass through the **for**-*j* loop. Therefore, the total number of passes through the **for**-*j* loop is given by

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)n}{2}$$

Analysis of Algorithms

- Every-Case Time Complexity (cont.)
  - Example of 'Matrix Multiplication'
    - The only instruction in the innermost **for** loop is the one that does a multiplication and an addition. The algorithm can be implemented in such a way that fewer additions are done than multiplications. Therefore, we will consider only the multiplication instruction to be the basic operation.
    - Basic operation: multiplication instruction in the innermost **for** 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 through the **for**-*j* loop there are always *n* passes through the **for**-*k* loop. Because the basic operation is inside the **for**-*k* loop,

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

Analysis of Algorithms

- Worst-Case Time Complexity (cont.)
  - Example of 'Sequential Search'
    - Basic operation: the comparison of an item in the array with x.
    - Input size: n, the number of items in the array.
    - The basic operation is done at most n times, which is the case if x is the last item in the array or if x is not in the array. Therefore,

$$W(n) = n$$

Analysis of Algorithms

Average-Case Time Complexity

Overview

- ***A*(*n*)** is defined as the average (expected value) of the number 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, and the determination of ***A*(*n*)** is called an average-case time complexity analysis.
- As is the case for ***W*(*n*)**, 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*. **It is important to assign probabilities based on all available information.**

Example of 'Sequential Search'

- Basic operation: the comparison of an item in the array with *x*.
- Input size: *n*, the number of items in the array.
- We first analyze the case in which it is known that *x* is in *S*, where the items in *S* are all distinct, and where we have no reason to believe that *x* is more likely to be in one array slot than it is to be in another. Based on this information, for $1 \leq k \leq n$, the probability that *x* is in the *k*th array slot is $1/n$. If *x* is in the *k*th array slot, the number of times the basic operation is done to locate *x* (and, therefore, to exit the loop) is *k*. This means that the average time complexity is given by

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

Analysis of Algorithms

- Average-Case Time Complexity (cont.)
  - Example of 'Sequential Search' (cont.)
    - Next we analyze the case in which x may not be in the array. To analyze this case we must assign some probability p to the event that x is in the array. If x is in the array, we will again assume that it is equally likely to be in any of the slots from 1 to n. The probability that x is in the kth slot is then p/n, and the probability that it is not in the array is $1 - p$. 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. The average time complexity is therefore given by

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

$$= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

    - 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.

Analysis of Algorithms

- Best-Case Time Complexity
  - Overview
    - $B(n)$ is defined as the minimum number of times the algorithm will ever do its basic operation for an input size of n.
    - So $B(n)$ is called the best-case time complexity of the algorithm, and the determination of $B(n)$ is called a best-case time complexity analysis.
    - ~~As is the case for $W(n)$ and $A(n)$, if $T(n)$ exists, then $B(n) = T(n)$.~~
  - Example of 'Sequential Search'
    - Basic operation: the comparison of an item in the array with x.
    - Input size: n, the number of items in the array.
    - Because $n \geq 1$, there must be at least one pass through the loop, If $x = S[1]$, there will be one pass through the loop regardless of the size of n. Therefore,

$$B(n) = 1$$

Analysis of Algorithms

- Complexity Function
  - In general,
    - a ***complexity function*** can be any function that maps the positive integers to the nonnegative reals. When not referring to the time complexity or memory complexity for some particular algorithm, we will usually use standard function notation, such as $f(n)$ and $g(n)$, to represent complexity functions.

$$f(n) = n$$
$$f(n) = n^2$$
$$f(n) = \lg n$$
$$f(n) = 3n^2 + 4n$$

Analysis of Algorithms

- Other Instructions
  - Overhead instructions
    - such as initialization instructions before a loop.
    - The number of times these instructions execute does not increase with input size.
  - Control instruction
    - such as incrementing an index to control a loop.
    - The number of times these instructions execute increases with input size.

Suppose we have two algorithms for the same problem with the following every-case time complexities: n for the first algorithm and n^2 for the second algorithm. The first algorithm appears more efficient. Suppose, however, a given computer takes 1,000 times as long to process the basic operation once in the first algorithm as it takes to process the basic operation once in the second algorithm. By "process" we mean that we are including the time it takes to execute the control instructions. Therefore, if t is the time required to process the basic operation once in the second algorithm, $1{,}000t$ is the time required to process the basic operation once in the first algorithm. For simplicity, let's assume that the time it takes to execute the overhead instructions is negligible in both algorithms. This means the times it takes the computer to process an instance of size n are $n \times 1{,}000t$ for the first algorithm and $n^2 \times t$ for the second algorithm. We must solve the following inequality to determine when the first algorithm is more efficient:

$$n^2 \times t > n \times 1{,}000t.$$

Analysis of Algorithms

- Analysis of Correctness
  - Requirement
    - we can analyze the correctness of an algorithm by developing a proof that the algorithm actually does what it is supposed to do.

Order

Order

- Overview
  - Linear-time algorithm
    - Such as n and $100n$
    - Time complexities are linear in the input size n
  - Quadratic-time algorithm
    - Such as n^2 and $0.01n^2$
    - Time complexities are quadratic in the input size n.
  - Linear-time algorithm better than quadratic-time algorithm
    - Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm.
    - In the theoretical analysis of an algorithm, we are interested in eventual behavior.
    - Next we will show how algorithms can be grouped according to their eventual behavior.
    - In this way we can readily determine whether one algorithm's eventual behavior is better than another's.

Order

- An Intuitive Introduction to Order
  - Type of functions
    - Pure quadratic functions
      - Such as $5n^2$ and $5n^2 + 100$
      - Because they contain no linear term
    - Quadratic-time algorithm
      - Such as $0.1n^2 + n + 100$ (complete quadratic)
      - Because it contains a linear term
    - The values of the other than the quadratic terms eventually become insignificant compared with the value of the quadratic term.

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

    - Therefore, although the function is not a pure quadratic function, we can classify it with the pure quadratic functions.
      - Intuitively, it seems that we should always be able to throw away low-order terms when classifying complexity functions.

Order

- An Intuitive Introduction to Order (cont.)
  - Complexity categories
    - Example
      - Classify $0.1n^3 + 10n^2 + 5n + 25$ with pure cubic functions
      - The set of all complexity functions that can be classified with pure quadratic functions is called $\Theta(n^2)$, where Θ is the Greek capital letter "theta."
      - If a function is a member of the set $\Theta(n^2)$, we say that the function is order of n^2.

$$g(n) = 5n^2 + 100n + 20 \in \Theta(n^2)$$

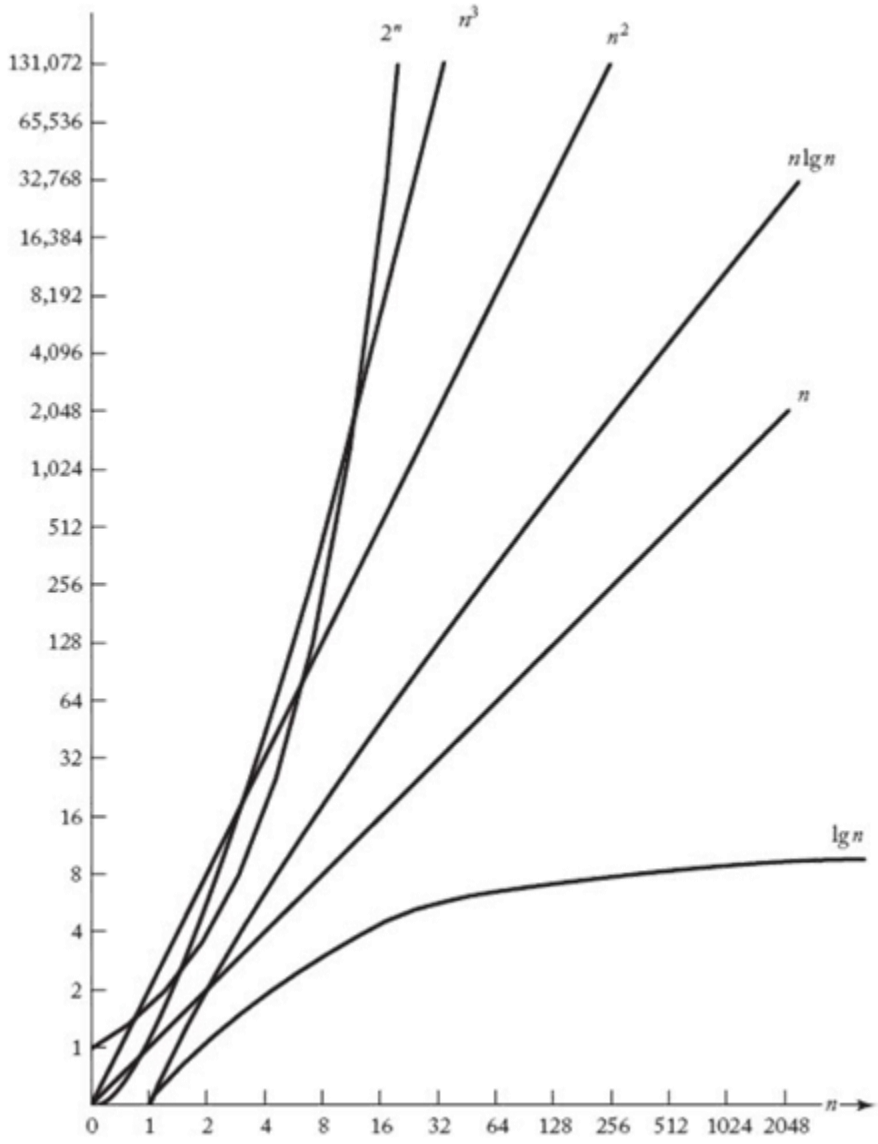
      - In the case of exchange sort algorithm, $T(n) = n(n - 1) / 2$

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

      - When an algorithm's time complexity is in $\Theta(n^2)$, the algorithm is called a quadratic-time algorithm or a $\Theta(n^2)$ algorithm.

Order

- An Intuitive Introduction to Order (cont.)
  - Complexity categories (cont.)
    - The most common complexity categories
      - $\Theta(\lg n)$, $\Theta(n)$, $\Theta(nlg\ n)$, $\Theta(n^2)$, $\Theta(n^3)$, $\Theta(2^n)$
      - In this ordering, if $f(n)$ is in a category to the left of the category containing $g(n)$, then $f(n)$ eventually lies beneath $g(n)$ on a graph.
    - Hypothetical algorithms
      - $100n$ and $0.01n^2$ or $\Theta(n)$ and $\Theta(n^2)$



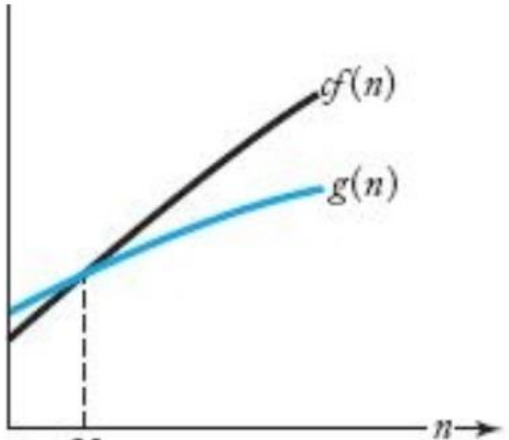
Order

- Big *O* Notation
  - Definition

For a given complexity function $f(n)$, $O(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

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

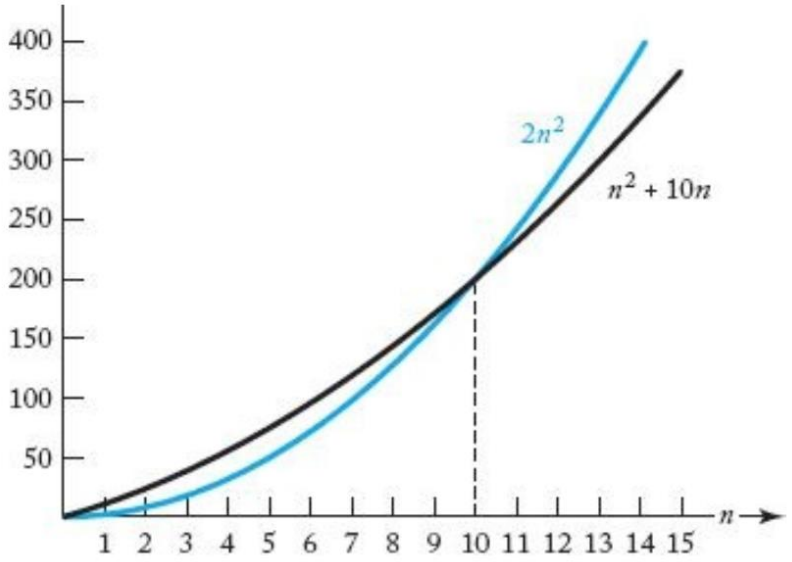
- If $g(n) \in O(f(n))$, we say that $g(n)$ is big O of $f(n)$
- Although $g(n)$ starts out above $cf(n)$ in that figure, eventually it falls beneath $cf(n)$ and stays there.



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

Order

- Big *O* Notation (cont.)
  - Examples
    - $g(n) = n^2 + 10n$ and $f(n) = 2n^2$, for $n \geq 10$
    - If, for example, $g(n)$ is in $O(n^2)$, then eventually $g(n)$ lies beneath some pure quadratic function cn^2 on a graph. This means that if $g(n)$ is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast as quadratic. For the purposes of analysis, we can say that eventually g(n) is at least as good as a pure quadratic function.
    - "Big *O*" (and other concepts that will be introduced soon) are said to describe the asymptotic behavior of a function because they are concerned only with eventual behavior. We say that "big *O*" puts an asymptotic upper bound on a function.



Order

- Big *O* Notation (cont.)
  - Examples (cont.)

We show that $5n^2 \in O(n^2)$. Because, for $n \geq 0$

$$5n^2 \leq 5n^2$$

We can take $c = 5$ and $N = 0$ to obtain our desired result

Given exchange sort algorithm.

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

Because, for $n \geq 0$

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

We can take $c = 1/2$ and $N = 0$ to conclude that $T(n) \in O(n^2)$

Order

- Big O Notation (cont.)
  - Examples (cont.)

We show that $n^2 + 10n \in O(n^2)$. Because, for $n \geq 1$

$$n^2 + 10n \leq n^2 + 10n^2 = 11n^2$$

We can take c = 11 and N = 1 to obtain our desired result

- The purpose of this last example is to show that the function inside "big O" does not have to be one of the simple functions.

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

$$n^2 \leq 1 \times (n^2 + 10n)$$

We can take c = 1 and N = 0 to obtain our desired result

- A complexity function need not have a quadratic term to be in $O(n^2)$.
  - It need only eventually lie beneath some pure quadratic function on a graph.
  - Therefore, any logarithmic or linear complexity function is in $O(n^2)$. Similarly, any logarithmic, linear, or quadratic complexity function is in $O(n^3)$, and so on.

We show that $n \in O(n^2)$. Because, for $n \geq 1$

$$n \leq 1 \times n^2$$

We can take c = 1 and N = 1 to obtain our desired result

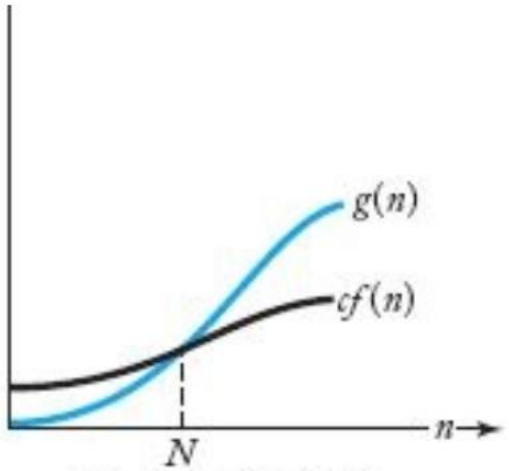
Order

- Omega Ω Notation
  - Definition

For a given complexity function $f(n)$, $\Omega(f(n))$ is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

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

- Although $g(n)$ starts out below $cf(n)$ in that figure, eventually it falls beneath $cf(n)$ and stays there.



(b) $g(n) \in \Omega(f(n))$

Order

- Omega Ω Notation (cont.)
  - Examples

We show that $5n^2 \in \Omega(n^2)$. Because, for $n \geq 0$

$$5n^2 \geq 1 \times n^2$$

We can take $c = 1$ and $N = 0$ to obtain our desired result

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

$$n^2 + 10n \geq n^2$$

We can take $c = 1$ and $N = 0$ to obtain our desired result

Order

- Omega Ω Notation (cont.)
  - Examples

Given exchange sort algorithm.

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

Because, for $n \geq 2$

$$n - 1 \geq \frac{n}{2}$$

Therefore, for $n \geq 2$

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

We can take $c = 1/4$ and $N = 2$ to conclude that $T(n) \in \Omega(n^2)$

- 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

Order

- Omega Ω Notation (cont.)
  - Examples

Given exchange sort algorithm.

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

Because, for $n \geq 2$

$$n - 1 \geq \frac{n}{2}$$

Therefore, for $n \geq 2$

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

We can take c = 1/4 and N = 2 to conclude that $T(n) \in \Omega(n^2)$

- 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

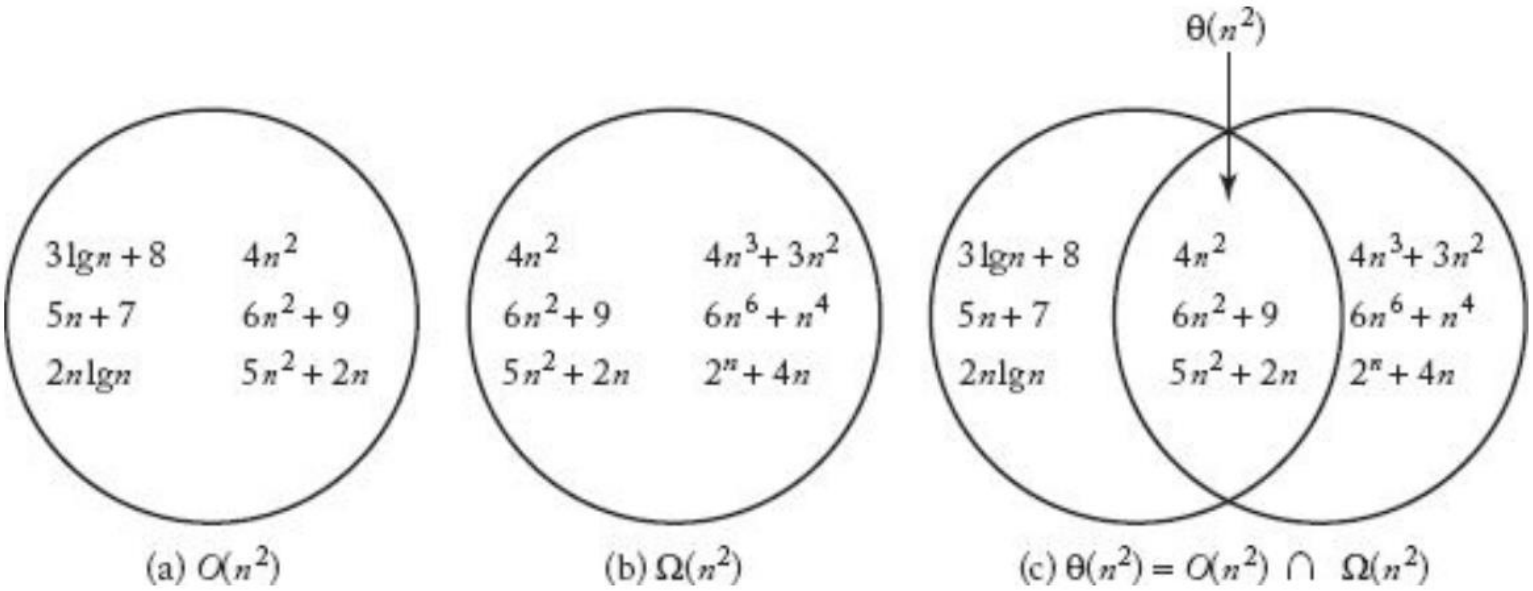
We show that $n^3 \in \Omega(n^2)$. Because, for $n \geq 1$

$$n^3 \geq 1 \times n^2$$

We can take c = 1 and N = 1 to obtain our desired result

Order

- Theta Θ Notation
  - $O(n^2), \Omega(n^2), \Theta(n^2)$



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

Order

- Theta Θ Notation (cont.)
  - Definition

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

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

This means that Θ ($f(n)$) is the set of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$,

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

- If $g(n) \in \Theta(f(n))$, we say that $g(n)$ is order of $f(n)$.

  - Examples

Given exchange sort algorithm.

$$T(n) = \frac{n(n-1)}{2} \quad \text{is in both} \quad O(n^2) \quad \text{and} \quad \Omega(n^2)$$

This means that T(n) $\in O(n^2) \cap \Omega(n^2) = \Theta(n^2)$

Order

- Theta Θ Notation (cont.)
  - Examples (cont.)

To prove n is not in $\Omega(n^2)$.
Assuming that n $\in \Omega(n^2)$ means we are assuming that there exists some positive constant c a nd some nonnegative integer N such that, for $n \geq N$,

$$n \geq cn^2$$

If we divide both sides of this inequality by cn, we have, for $n \geq N$,

$$\frac{1}{c} \geq n$$

However, for any $n > 1/c$, this inequality cannot hold, which means that it cannot hold for all $n \geq N$. This contradiction proves that n is not in $\Omega(n^2)$.

Order

- Small *o* Notation
  - Definition

For a given complexity function $f(n)$, $o(f(n))$ 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, for all $n \geq N$

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

  - Compare to the Big O
    - Recall that "big *O*" means there must be ***some*** real positive constant c for which the bound holds. This definition says that the bound must hold for ***every*** real positive constant c.
      - Because the bound holds for every positive c, it holds for arbitrarily small c. For example, if $g(n) \in o(f(n))$, there is an N such that, for $n > N$,

$$g(n) \leq 0.00001 \times f(n)$$

      - We see that $g(n)$ becomes insignificant relative to $f(n)$ as n becomes large. For the purposes of analysis, if $g(n)$ is in $o(f(n))$, then $g(n)$ is eventually much better than functions such as $f(n)$.

Order

- Small *o* Notation (cont.)
  - Examples

We show that

$$n \in o(n^2).$$

Let $c > 0$ be given. We need to find an N such that, for $n \geq N$,

$$n \leq cn^2$$

If we divide both sides of this inequality by cn, we get

$$\frac{1}{c} \leq n.$$

Therefore, it suffices to choose any $N \geq 1/c$.

Notice that the value of N depends on the constant c. For example, if $c = 0.00001$, we must take N equal to at least 100,000. That is, for

$$n \leq 0.00001n^2$$

Order

- Small *o* Notation (cont.)
  - Examples (cont.)

We show that n is not in $o(5n)$. We will use proof by contradiction to show this. Let c=1/6
If $n \in o(5n)$, then there must exist some N such that, for $n \geq N$,

$$n \leq \frac{1}{6}5n = \frac{5}{6}n$$

This contradiction proves that n is not in $o(5n)$.

Order

- Small *o* Notation (cont.)
  - Theorem 1.2

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

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

That is, $g(n)$ is in $O(f(n))$ but is not in $\Omega(f(n))$.

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

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

which means that the bound certainly holds for some c. Therefore,

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

We will show that $g(n)$ is not in $\Omega(f(n))$ using 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 $n \geq N_1$,

$$g(n) \geq c \times f(n).$$

But, because $g(n) \in o(f(n))$, there exists some $N2$ such that, for all $n \geq N_2$,

$$g(n) \leq \frac{c}{2} \times f(n).$$

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 $\Omega(f(n))$.

Order

- Small *o* Notation (cont.)
  - Examples

Consider the function

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

$g(n) \in O(n) - \Omega(n)$ but that $g(n)$ is not in $o(n)$

- When complexity functions represent the time complexities of actual algorithms, ordinarily the functions in $O(f(n)) - \Omega(f(n))$ are the same ones that are in $o(f(n))$