Chapter 4

Iterative Algorithm Design Issues

Objectives

After reading this chapter, you should understand:

- *The correct use of loops in programs*
- *Factors that affect the efficiency of algorithms*
- *How to estimate and specify execution times*
- *How to compare algorithms in terms of their efficiency*
- *The Order Notation: Big-Oh, Omega, Theta, small omega, small-Oh and their properties*
- *Justification for the use of Problem size as a measure*
- *Some Tradeoffs in algorithm design*

Chapter Outline

4.1 Introduction

4.2 Use of Loops

4.3 Efficiency of Algorithm

4.3.1 Removing Redundant Computations Outside Loops

4.3.2 Referencing of Array Elements

4.3.3 Inefficiency due to Late Termination

4.3.4 Early Detection of desired Output Conditions

4.4 Estimating and Specifying Execution Times

4.4.1 Justification for the use of Problem Size as a Measure

4.4.2 Computational cost as a function of Problem Size for a range of Computational Complexities

4.5 Order Notation

4.5.1 Big-Oh Notation

4.5.2 Theta Notation

4.5.3 Omega Notation

4.5.4 Small-Oh Notation

4.5.5 ω Notation

4.5.6 Measuring the Execution Times

4.5.7 Other Trade-offs

4.6 Algorithm Strategies

Summary | *Key Terms* | *Exercises* | *Web Resources*

4.1 INTRODUCTION

In this chapter we shall discuss the characteristics of most of the common and real life algorithms, that is, they have at least one iterative component or a *loop*. The iteration may be explicit through a looping construct or it can be due to a recursive algorithm. In any case, the idea is that a part of the algorithm statements (or code segments if it a program) will be executed repeatedly thousands or even millions of times. Even a small amount of excess time spent within the loop can lead to a major loss in

the efficiency of the algorithm. The situation can even be worse if there are loops within loops or *nested* iterations. There are algorithms in which the depth of nesting itself is controlled by an outer loop! The iterative components or loops are thus a major contributor to (in)efficiencies of algorithms. In this chapter, we shall discuss: contributors to loop inefficiency, initial conditions, loop invariant, proper loop termination, and so on. We shall also see how to remove redundant computations, late termination, and various trade-offs in setting up an iterative algorithm.

This chapter forms the foundation for an in-depth analysis of the efficiency of algorithms discussed, in Chapter 14. It can also be considered as a practical guide for writing efficient algorithms and programs.

4.2 USE OF LOOPS

As we break down algorithm into sub-algorithms, sooner or later we shall come across some iterative construct, that is, a loop, apart from conditional statements. Experience tells us that this is a stumbling block for many novice programmers. A loop executing one more or one less time than the intended number is so common that people have stopped raising eyebrows. So we should discuss it in detail.

To construct a loop we should be aware of three aspects:

1. The **initial condition** that needs to be true *before* the loop begins execution
2. The **invariant relation** that must hold before, during, and after each iteration of the loop
3. The condition under which the loop must **terminate**

The following design process is suggested to take care of these three aspects of algorithm development.

To establish initial condition Set the loop control variables, to values which are appropriate for solving the smallest instance of the problem in question. For example, suppose we want to add elements of the given ar-

question. For example, suppose we want to add elements of the given array using a loop. The loop variables are, i the loop control variable which is also used as the array index and s, the current value of the sum. The smallest problem in this case is the sum of 0 number of elements. In this case s must be 0. Thus the initial condition is:

$$i \leftarrow 0$$
$$s \leftarrow 0$$

To find the iterative construct Try to extend the smallest problem to the next smallest. In doing this, we will come to know what changes are to be done to the loop variables to achieve this change. For our example of summation of elements of an array, the next smallest is the case of n = 1, in which case s must be equal to $a[i]$. Thus the solution for n = 1 can be derived from the solution for n = 0 by first considering the solution for i = 1:

$$i \leftarrow 1$$
$$s \leftarrow a[1]$$

Then the same can be re-written as:

$i \leftarrow i + 1$	note:	i on RHS is 0
$s \leftarrow s + a[i]$		i is now 1

This is the general solution for any $n > 0$. Thus, in terms of the structured control constructs, the complete algorithm is:

$i \leftarrow 0$

$s \leftarrow 0$

while $i < n$ do

$i \leftarrow i + 1$

$s \leftarrow s$ + a[i]

endwhile

Loop termination The simplest termination condition occurs when it is known in advance the number of times the loop is to be iterated. In that case we can use a `FOR` structured programming construct. For example, to execute a loop 20 times, we may write:

for $i \leftarrow 1$ to 2 0 do

...

...

endfor

Note that it is not necessary that the initial value assigned and the final value are constants, the only requirement is that their values should be known at the start of the loop. Thus we may have:

for $i \leftarrow m$ to n

...

...

endfor

assuming that the values of m and n are known.

Another possibility is that the loop terminates when some condition becomes false.

For example:

For example:

while ($x > 0$) and ($x < 10$) do

...

...

endwhile

For loops of this type we cannot directly predict in advance the number of times the loop will iterate before it terminates. In fact, there is *no assurance that a loop of this type will terminate at all.* The responsibility to achieve it rests on the algorithm designer. This is one way an error can creep into an algorithm and is one topic in algorithm correctness.

Another way in which a loop can be made to terminate is by forcing the condition of termination. In some algorithms this can be very useful. An example will illustrate this:

Example 1

Suppose we want to check whether the elements in an array are strictly in increasing order, that is, $a[1] < a[2] < a[3] \ldots < a[n]$. The algorithm is:

$a\,[n + 1] \leftarrow a\,[n]$

$i \leftarrow 1$

while $a[i] < a\,[i + 1]$ do

$i \leftarrow i + 1$

endwhile

Consider what would happen with $n = 4$ and an array having elements 11 14 21 27. It would be interesting to express the same problem, using $i \leq$

n as the loop condition. Notice the role of the first statement $a[n + 1] \leftarrow a[n]$. It is a means of successful loop termination.

4.3 EFFICIENCY OF ALGORITHMS

The design and implementation of algorithms have a profound influence on their efficiency. Every algorithm, when implemented must use some of the system resources to complete its task. The resources most relevant to the efficiency are the use of the central processor unit (CPU) time and internal memory (RAM). In the past, high cost of the computer resources was the driving force behind the desire to design algorithms that are economical in the use of CPU time and memory. As time passed, while on one hand, the cost of these resources has reduced and continues to do so, the requirement of more complex, and thereby time consuming algorithms has increased. Hence the need for designing efficient algorithms is present even today.

There is no standard method for designing efficient algorithms. Despite this, there are a few generalizations that can be made about the problem characteristics, while other characteristics would be specific to a particular problem. We shall discuss some means of improving the efficiency of an algorithm.

4.3.1 Removing Redundant Computations Outside Loops

Example 2

Most of the inefficiencies that creep into the implementation of algorithms are due to redundant computations or unnecessary storage. The effect of redundant computation is serious when it is embedded within a loop, which must be executed many a times. The most common mistake when using loops is to repeatedly recalculate the part of an expression that remains constant throughout the execution phase of the entire loop. Let us take an example:

$x = 0;$

```
for i = 1 to N do
begin
  x = x + 0.01;
  y = (a * a * a + c) * x * x + b * b * x;
  writeln(' x = ', x, ' y = ', y);
end
```

This loop does twice the number of multiplications (computations) necessary to arrive at the answer. *How can the unnecessary multiplications and computations be removed?* Declare two new constants a3c and b2 *before* execution of the loop.

```
a3c = a * a * a + c;
b2 = b * b;
x = 0;
for i = 1 to N do
begin
   x = x + 0.01;
   y = a3c * x * x + b2 * x;
   writeln('x =', x, 'y =', y);
end
```

In this example, the saving is not all that significant, but in actual commercial as well as scientific applications such improvements are known

to improve the performance by a reasonably good factor. Also, utmost care should be exercised in making sure that redundancies are removed in the innermost loops, as they are a major contributor to cost in terms of time.

4.3.2 Referencing of Array Elements

Generally, arrays are processed by iterative constructs. If care is not exercised while programming, redundant computations can creep into array processing. Consider, as an example, the following two versions of an algorithm in C, to find the maximum number and its position in an array of numbers:

Version I

```
p=0;
   for (i=1; i<n;i++){
    if (a [i] > a[p]){
      p = i;
    }
   }
   max = a[p]; /* p is the position of max */
```

Version II

```
p=0; i=0;
   max = a[i];
   for(i=1; i<n; i++){
```

```
    if (a[i] > max){
      max = a[i];
      p = i;
    }
  }
```

Which of the two versions of the algorithm is preferable? Why?

Note that indexing of any kind, whether it be of simple data elements or of structured elements requires more time. Thus it can be said that the second version of the implementation is preferable because the condition test (that is, `a[i] > max`) which is the dominant operation is much more efficient to perform than the corresponding test in the version I of the program. By using the variable `max`, only one array reference is being made each time, whereas when using the variable `p` as index, two array references are required. References to array elements requires *address arithmetic* to arrive at the correct element and this requires time. Thus more time is spent in locating the specific value for comparison. Secondly, the second version of the program is better *documented* and the purpose of the code is much more explicit to the reader. This is because, by adding the variable called `max` we are providing a clue to the reader of the program about its operation.

4.3.3 Inefficiency due to Late Termination

Another possibility of inefficiency creeping into the implementation of an algorithm is, when considerably more tests are carried out, than are required to solve the problem at hand. The following is a good example: Suppose we have to search an *alphabetically ordered* list of names for a particular name using *linear search*.

Example 3

An inefficient implementation for this case would be one where all names were examined even if a node in the list was reached where it could be definitely said that the name *cannot* occur later in the list. For example, suppose we are searching for Ketan, then, as soon as we reach a name that is alphabetically later than Ketan example, Lalit, we should *not* proceed further.

Inefficient algorithm

1. While Name sought ≠ Current Name and not(EOF) do get name from list. Where EOF denotes End of File.
2. ...

Efficient algorithm

1. While Name sought > Current Name and not (EOF) do get name from list
2. Test if current name is equal to the name sought.

Let us take as an example, the **Bubble Sort** algorithm (sorting in ascending order):

1. Establish the array[1 ... *N*] of *N* elements
2. While the array is still not sorted Do
    1. Set the order indicator Sorted to *True*
    2. For all *adjacent* pair of elements in the unsorted part of the array do
        1. if the current adjacent pair is not in non-descending order then
            1. Exchange the elements of the pair
            2. Set sorted to false
3. Return sorted array

Implementation

Algorithm 4.1 implements the efficient Bubble algorithm

Algorithm 4.1: Function bubble sort

```
1    I = N;
2    for i ← 1 to N − 1 do
3     for j ← 1 to I do
4      if Current Key > Next Key then
5       Exchange the items;
6      end
7     end
8     if no exchange was made during the above loop then
9      return;
10    else
11     I ← I − 1;
12    end
13   end
```

Analysis of Bubble sort algorithm The relevant parameters for analyzing the algorithm are the number of comparisons and number of exchanges made. The minimum number of comparisons made are $n - 1$, which is the case when the data are already sorted. The maximum number of comparisons occur when $(n - 1)$ passes are made. In this case $n*(n - 1)/2$ comparisons are made. If the array is already sorted (best case)

1)/2 comparisons are made. If the array is already sorted (best case), zero exchanges are made. In the worst case, there are as many exchanges as there are comparisons that is, $n*(n-1)/2$ exchanges are made or required. Thus on an average $n*(n-1)/4$ exchanges are made.

After the ith iteration all $a[n-i+1 \ldots n]$ are sorted and all $a[1, \ldots, n-i]$ are less than or equal to $a[n-i+1]$. In addition, the inner loop causes $a[1 \ldots n-i]$ elements to be sorted with $i < n$. For the inner loop, after the jth iteration the $a[j+1]$ element will be greater than or equal to all the elements in $a[1 \ldots j]$. The termination of the `while` loop is *guaranteed* since i increases by 1 with each iteration and a `for` loop must, by definition always terminate.

A weakness of this algorithm is that it relies more heavily on exchanges than most of the other sorting methods. Since exchanges are relatively time consuming, this particular characteristic makes the method costly for sorting a large random set. The importance of this method is that the algorithm is suitable for sorting a data set which has relatively a small percentage of its elements out of order, since it will require only a small number of comparisons and exchanges.

An examination of the intermediate configuration for the sample data set suggests that the algorithm lacks balance and symmetry. While larger data elements make rapid progress towards the **right end** of the array, elements of small values only make a slow progress to the left end o the array. Note the progress of 3 in Fig. 4.1.

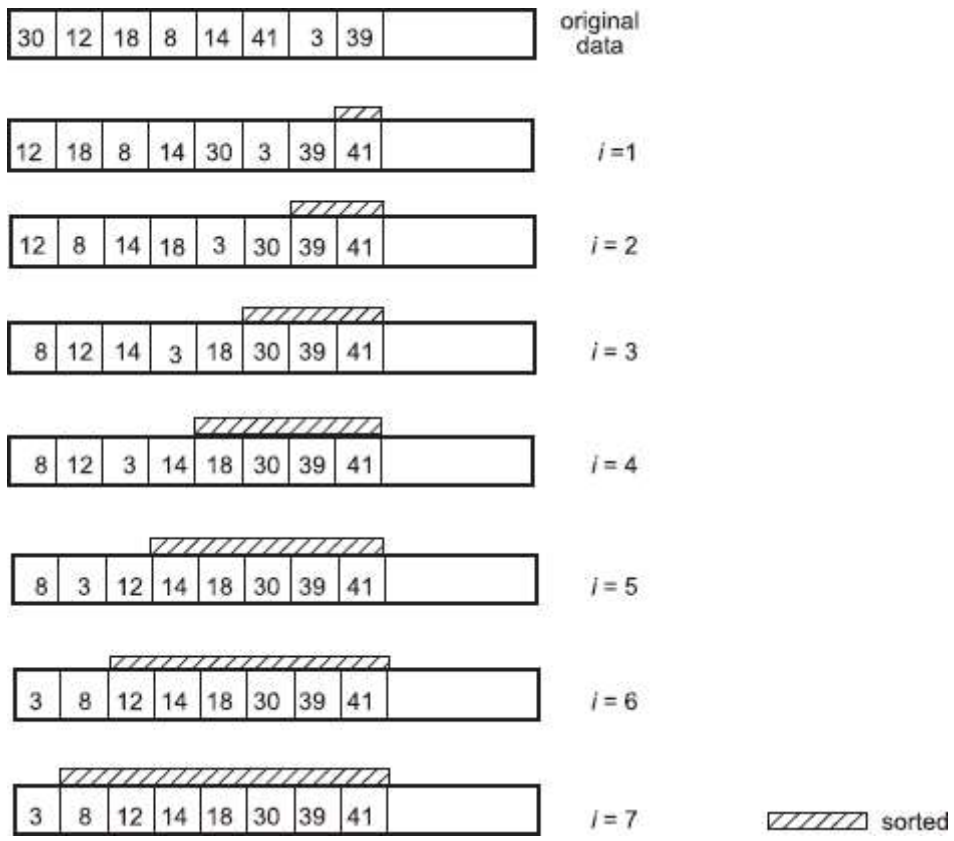

Fig. 4.1 Bubble sort

Example 4

The inefficiency discussed here can creep into the Bubble sort algorithm if care is not taken while implementing it. This can happen if the inner loop that drives the exchange mechanism always goes through the full length of the array. An example:

for $i = 1$ to $n - 1$

for $j = 1$ to $n - 1$

if (a[j] > a[$j + 1$]) then exchange a[j] with a[$j + 1$]

With this sorting mechanism, after the ith iteration the last i values in the array will be in sorted order. Thus for any given i, the inner loop should not proceed beyond $n - i$. Hence the loop structure should be:

for $i = 1$ to $n - 1$

for $j = 1$ to $n - i$

if (a[j] > a[$j + 1$]) exchange a[j] with a[$j + 1$]

4.3.4 Early Detection of desired Output Conditions

It may sometimes happen that, due to the very nature of the input data a particular algorithm, for example, the Bubble Sort algorithm, establishes the desired output condition *before* the general condition for termination is met. For example, a bubble sort algorithm might receive a set of data that is already in sorted condition. When this is the case, it is obvious that the algorithm will have the data in sorted condition long before the general loop termination conditions are satisfied. It is therefore desirable to terminate the sorting as soon as it is established that the input data is already sorted. How do we determine during the course of the execution of the algorithm, whether the data is already sorted? To do this, all we need to do is to check whether there have been *any* exchanges in the current pass of the inner loop. If there have been no exchanges in the current pass, the data must be already sorted and the loop can be terminated early. See line no. 8 in the Bubble Sort example, Algorithm 4.1.

In general we must include *additional tests* or steps to determine or detect the conditions for early termination. These tests should be included only if they are inexpensive with regards to computer time as was the case in the Bubble sort algorithm. In general, it may be necessary to trade-off *extra tests* and *storage* versus *CPU time* to achieve early termination of algorithms.

4.4 ESTIMATING AND SPECIFYING EXECUTION TIMES

To arrive at a quantitative measure of an algorithm's performance it is necessary to set up a computational model that reflects its behaviour under specified input conditions. This model must deal with the essence (the

heart) of computation and at the same time be independent of any specific programming language.

Thus we usually characterize an algorithm's performance in terms of *the size of the problem being solved*. It is obvious that more computing resources are required to solve larger problems in the same class of problems.

An important question is "*How does the "COST" of solving a problem vary as N increases*". An intuitive response is "*The cost increases linearly with an increase in N*". (example, a problem with n = 200 takes twice as much time as a problem with n = 100). While this linear dependence on N is justifiable for some simple problems or algorithms, in general the relationship between complexity and the problem size follows a completely different pattern. At the lower end of the scale we have algorithms with logarithmic (that is, better) dependence on N, while on the other end of the scale we have algorithms with an exponential dependence on N. With an increase in N the relative difference in the cost of computation is enormous for these two extremes. Table 4.1 illustrates the comparative computational cost for a range of N values.

4.4.1 Justification for the use of Problem Size as a Measure

We shall discuss an algorithm to raise x to some power n, called `Power1()`, given in section 4.5.8. The algorithm is provided here for ready reference:

Algorithm 4.2: Power1(real x, positive integer n)

```
1  [Brute-Force method]
2  local integer i;
3  result ← x; [t1 1]
```

```
4    for i ← 1 to n − 1 do
5      [t2 n − 1]
6      result ← result * x; [t3 n − 1]
7    end
8    return result; [t4 1]
```

Now suppose we are interested in finding the total running time for this algorithm. The time taken by each statement and the number of times it will be executed is shown by the side of each statement. Thus the total running time is:

$$T(\text{Power1}, n) = t1 + (n - 1) * t2 + (n - 1) * t3 + t4$$

$$= (t1 + t4) + (n - 1) * (t2 + t3)$$

Note that the first term ($t1 + t4$) contributes only a constant amount of time, no matter whatever be the value of n. The behaviour of the algorithm as the problem size grows, that is, as n increases, is governed by the second term ($t2 + t3$).

Also remember that we want to have a figure of merit that is independent of the software implementation or the specific hardware used. We thus disregard the multiplying factor ($t2 + t3$) and simply say that T(Power1, n) increases as n, that is, in direct proportion to n. So we refer to the *rate of growth of the running time* and not the actual running time of an algorithm.

Later, we shall define a notation to explicitly specify such growth in quantities, known as *asymptotic* running time. We shall talk about the *time complexity* of algorithms, strictly speaking *asymptotic running time complexity.* Since, we are interested only in this measure and not the ac-

tual running time, we can concentrate on the behaviour of the loops within the algorithm, as they, generally, are the major contributors to the execution time of a program and/or function. If it is a recursive algorithm, our emphasis will be on finding the number of times the algorithm will call itself.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	$2n$	$n!$
1	2	2	4	8	4	2
3.322	10	33.22	100	1000	$> 10^3$	3628800
6.644	10^2	664.4	10^4	10^6	$>> 10^{25}$	9.3×10^{157}
9.966	10^3	9966.0	10^6	10^9	$>> 10^{250}$	10^{2567}
13.287	10^4	132877	10^8	10^{12}	$>> 10^{2500}$	10^{35659}

Table 4.1 Cost of computation as a function of problem size

4.4.2 Computational cost as a function of Problem Size for a range of Computational Complexities

From Table 4.1, we observe that for problems that exhibit exponential behaviour, only limited input data size is allowable. Thus, we can solve *only very small* problems with an algorithm that exhibits exponential behaviour. Assume that a computer can perform one million operations per second. Then, an algorithm having exponential behaviour with a sufficiently large problem size N, will take extremely long time—even longer than the age of the Earth. On the other extreme, for an algorithm with logarithmic dependence on N, a problem with $N = 10^4$ would require only 13 steps which amounts to about 13 microseconds of computer time. These examples emphasize how important it is to have an understanding of the behaviour of the algorithms as a function of the problem size. Also by analyzing the algorithms, a theoretical model or the value of the inherent computational complexity can be gauged.

In analyzing an algorithm we must also look at the operations in the problem, to determine which is the *dominant* one. Sometimes, in a particular problem a specific arithmetic expression is evaluated a number of

times or a number of comparisons and/or exchanges are made and they increase non-linearly as N grows.

For example, computations, exchanges, and comparisons characterize most sorting algorithms. In such algorithms, the number of comparisons dominate and therefore we use comparisons as the dominant operation in our computational model for sorting algorithms.

4.5 ORDER NOTATION

The Order Notation is a standard notation developed to denote the computing time (or space) of algorithms and bounds on the upper limit or the lower limit on the computing time (or space) of algorithms. It refers collectively to the **Big-Oh**, the **Theta**, the **Omega** and the **Small-Oh** notations. An algorithm in which the dominant mechanism (one which is most executed) does not exceed $c * n^2$ times, where c is a constant and n is the problem size, is said to have an order n^2 complexity which is expressed as $O(N^2)$.

4.5.1 Big-Oh Notation

Formally, a function $g(n)$ is $O(f(n))$ provided there exists a constant c for which the relationship $g(n) \leq c.f(n)$ holds for all values of n that are finite and positive. Thus we have a means of characterizing the *asymptotic* complexity of algorithms and hence of determining the size of problems that it can solve, using a conventional sequential computer.

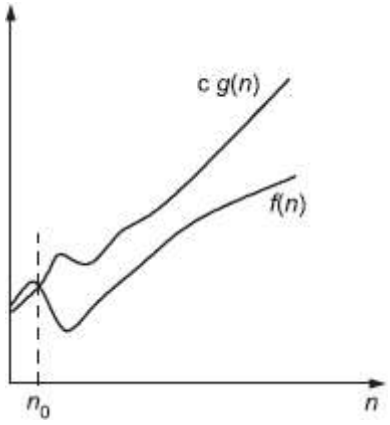

Fig. 4.2 Definition of O(g(n))

The order notation Big-Oh is defined as follows:

$$f(n) = O(g(n)) = \lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where $c \neq 0$.

In this case we say that $g(n)$ is an asymptotic upper bound for $f(n)$. See Fig. 4.2.

1. Θ is stronger than O
   $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$, or $\Theta(g(n)) \subseteq O(g(n))$.
2. We write $f(n) = O(g(n))$ for $f(n) \in O(g(n))$.

Examples

Assuming that a, b and c are constants:

1. $an^2 + bn + c = O(n^2)$ provided $a > 0$.
2. $an + b = O(n^2)$, provided $a > 0$.
3. $n \log n + n = O(n^2)$.
4. $log^k(n) = O(n)$ for all $k \in \mathbb{N}$.

O can be used as an upper bound for the running time for the worst case input and hence for *any* input. Sometimes O is used to informally de-

scribe tight bounds, but it is preferable, as we have done, to use Θ for tight bounds and O for upper bounds.

4.5.2 Theta Notation

The Theta notation is defined as:

Let $g(n)$ be an asymptotically non-negative function on the set of natural numbers.

$$\Theta(g(n)) = \{f(n) \mid \exists\ c_1 > 0, c_2 > 0, n_0 \in \mathbb{N} \wedge \forall n \geq n_0 \wedge 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

In other words, function $f(n)$ belongs to $\Theta(g(n))$ if it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for some constants c_1, c_2 for all n greater than some n_0. See Fig. 4.3.

Examples

1. $\frac{n^2}{2} - 3n = \Theta(n^2)$

   We have to determine $c_1 > 0$, $c_2 > 0$, $n_0 \in \mathbb{N}$ such that $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2$

   for any $n \geq n_0$.
   Dividing by n^2 yields:
   $c_1 \leq 1/2 - 3/n \leq c_2$ This is satisfied for $c_1 = 1/14$, $c_2 = 1/2$, $n_0 = 7$.

2. $6n^3 \neq \Theta(n^2)$
   If the above relation were true we would have to determine $c_1 > 0$, $c_2 > 0$, $n_0 \in /$ such that:
   $c_1 n^2 \leq 6n^3 \leq c_2 n^2$ for any $n \geq n_0$ which cannot exist.

Properties of Theta

Assume $f(n)$ and $g(n)$ are asymptotically positive. Then,

1. $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$ (transitivity)
2. $f(n) = \Theta(f(n))$ (reflexivity)
3. $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$ (symmetry)
4. $\max(f(n), g(n)) = \Theta(f(n) + g(n))$ (maximum)

If $f(n)$ and $g(n)$ are the running times of the two branches of an `if` statement, then the above result can be used to get a tight bound on the *worst-case* running time of the entire `if` statement, assuming that nothing is known about the condition test of `if`. Such a situation may arise, for example, if the condition test depends on an unknown input.

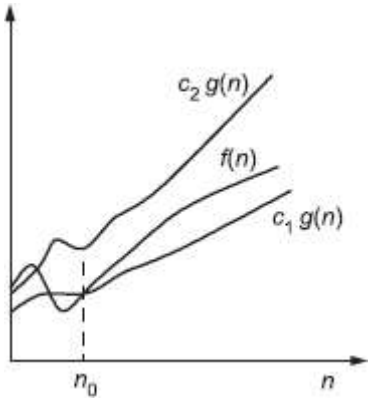

Fig. 4.3 Definition of Θ (g(n))

4.5.3 Omega Notation

The Omega notation is defined as:

Let $g(n)$ be an asymptotically non-negative function on the set of natural numbers.

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 \in \mathbb{N} \wedge \forall n \geq n_0 \wedge 0 \leq cg(n) \leq f(n)\}$$

In this case we say that $g(n)$ is an asymptotic **lower bound** for $f(n)$. See Fig. 4.4

1. Θ is stronger than Ω

$f(n) = \Omega(g(n)) \Rightarrow f(n) = O(g(n))$, or $\Omega(g(n)) \subset O(g(n))$

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$, or $\Theta(g(n)) \subseteq \Omega(g(n))$.

2. we write $f(n) = \Omega(g(n))$ for $f(n) \in \Omega(g(n))$.

Examples

Assuming that *a, b* and *c* are constants,

1. $an^2 + bn + c = \Omega(n^2)$ provided $a > 0$, since it is also $\Theta(n^2)$.
2. $an + b = O(n^2)$, provided $a > 0$.
3. $n \log n + n = O(n^2)$.
4. $\log^k(n) = O(n)$ for all $k \in \mathbb{N}$.

Ω can be used as a lower bound for the running time for the *best case* input and hence for *any* input. For example, the best case running time of insertion sort is $\Omega(n)$.

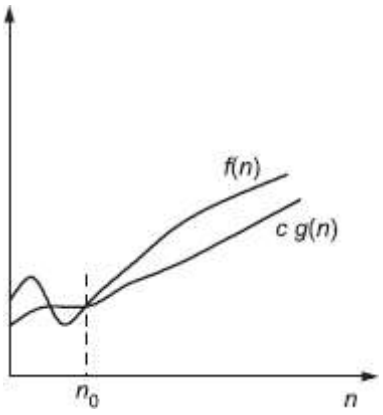

Fig. 4.4 Definition of Ω (g(n))

Properties of Theta, Big-Oh and Omega

The following list gives some of the combined properties of theta, big-oh and omega notations:

1. $f(n)$ is a tight bound if it is an upper bound and a lower bound:
   $f(n) = \Theta(n) \Leftrightarrow f(n) = O(n) \wedge f(n) = \Omega(n)$
2. $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Leftrightarrow f(n) = O(h(n))$
   $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Leftrightarrow f(n) = \Omega(h(n))$ (transitivity)
3. $f(n) = O(f(n))$

$f(n) = \Omega(f(n))$ (reflexivity)

4. $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ (transpose symmetry)

Further Examples

In order to help better understand these notations, we give below a few more illustrative examples.

1. $3n^2 - 100n + 6 = O(n^2)$ because $3n^2 > 3n^2 - 100n + 6$.
2. $3n^2 - 100n + 6 = O(n^3)$ because $0.00001n^3 > 3n^2 - 100n + 6$.
3. $3n^2 - 100n + 6 \neq O(n)$ because $cn < 3n^2$ when $n > c$.
4. $3n^2 - 100n + 6 = \Omega(n^2)$ because $2.99n^2 < 3n^2 - 100n + 6$ when $n > c$.
5. $3n^2 - 100n + 6 \neq \Omega(n^3)$ because $3n^2 - 100n + 6 < n^3$ when $n > c$.
6. $3n^2 - 100n + 6 = \Omega(n)$ because $10^{10}n < 3n^2 - 100n + 6$.
7. $3n^2 - 100n + 6 = \Theta(n^2)$ because both O and Ω.
8. $3n^2 - 100n + 6 \neq \Theta(n^3)$ because not Ω.
9. $3n^2 - 100n + 6 \neq \Theta(n)$ because not O.

Asymptotic Notations in Equations

The following results may be helpful in applying the asymptotic notations.

1. $f(n) = \Theta(n)$ simply means $f(n) \in \Theta(n)$.
2. More generally, $\Theta(n)$ can be replaced by an anonymous function which is an element of $\Theta(n)$, for example,
   $3n^2 + 3n+1 = 2n^2 + \Theta(n)$ means
   $3n^2 + 3n+1 = 2n^2 + f(n)$ and $f(n) \in \Theta(n)$ for some $f(n)$.
3. In recurrences,
   $T(n) = 2T(n/2) + \Theta(n)$
4. In calculations,
   $2n^2 + 3n+1 = 2n^2 + \Theta(n) = \Theta(n^2)$

4.5.4 Small-Oh Notation

The upper bound provided by O may or may not be tight. We use Small-oh, o for an upper bound which is *not* tight.

Let $g(n)$ be an asymptotically non-negative function on the set of natural numbers.

$$o(g(n)) = \{f(n) \mid \forall c > 0 \land \exists n_0 \in \mathbb{N} \land \forall n \leq n_0 \land 0 \leq f(n) \geq cg(n)\}$$

The implication of this definition is that $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. For example,

1. $2n = o(n^2)$
2. $2n^2 \neq o(n^2)$
3. $2n^3 \neq o(n^2)$

4.5.5 ω Notation

The lower bound provided by Ω may or may not be tight. We use ω for an lower bound which is *not* tight.

Let $g(n)$ be an asymptotically non-negative function on the set of natural numbers.

$$\omega(g(n)) = \{f(n) \mid \forall c > 0 \land \exists n_0 \in \mathbb{N} \land \forall n \geq n_0 \land 0 \leq cg(n) \leq f(n)\}$$

The implication of this definition is that $g(n)$ becomes insignificant relative to $f(n)$ as n approaches infinity. For example,

1. $n^2/2 = \omega(n)$
2. $n^2/2 \neq \omega(n^2)$
3. $n^2/2 \neq \omega(n^3)$

Example

We consider as an example of o and ω notations, the factorial function, $n!$, which is defined as: $0! = 1$, $n! = n^* (n - 1)!$ From Sterling's approximation,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

One can derive the following.

1. $n! = o(n^n)$
2. $n! = \omega(2^n)$

4.5.6 Measuring the Execution Times

While working with the big-oh notation, we discarded the multiplicative constants. The functions $f(n) = 0.001n^2$ and $g(n) = 1000n^2$ are treated identically, even though $g(n)$ is a million times larger than $f(n)$ for all values of n. The rationale behind this attitude is illustrated by Table 4.1, which tabulates the growth rate of several functions arising in algorithm analysis, on problems of reasonable size, denoted by n. Specifically, assuming that each operation in Table 4.1 takes 1 nanosecond, we can draw the following conclusions.

1. The algorithms of all complexity classes take roughly the same amount of time for $n \leq 10$.
2. The algorithm whose running time is $n!$ becomes useless well before $n = 20$.
3. The algorithm whose running time is 2^n has a greater operating range than the one with running time of $n!$, but it becomes impractical for $n > 40$.
4. The algorithm whose running time is n^2 is perfectly reasonable up to about $n = 100$, but it quickly deteriorates with larger n. For $n > 1000000$ it is likely to be unusable.
5. The algorithms whose complexity is n and $n \log n$ remain practically usable with the size of input up to 10^9.
6. All algorithms which exhibit $\log n$ behaviour are always usable.

This discussion shows that as far as selection of an algorithm is concerned, we can generally go by its complexity in terms of order notations, but to select between actual implementations, we should compare the actual execution times, or derive the multiplying constants, as suggested later in Chapter 14.

For example, even though an algorithm with a complexity of n^3 will be considered to be slower compared to the one with a complexity of n^2, for $n < 1\ 000\ 000$ the former will be faster, if the multiplying factor for the second algorithm is 1 000 000, that is, its complexity is $1\ 000\ 000n^2$.

Empirical execution times of the algorithms or parts thereof can be done by using various measuring devices—both hardware and software—that are available. Usually software solutions are used, though hardware solutions give better accuracy. These tools are generally called monitors or profiling software.

4.5.7 Other Trade-offs

Consider Algorithms 4.2 and 4.3, for calculating x^n:

Algorithm 4.2 uses the definition of an integer power $x^n = x * x * \ldots * x$ n times. Let us define $f1(n)$ to be the number of multiplications required by Power1(n) to compute x^n as a function of n. Thus $f1(n) = n - 1$ and these multiplications are always executed.

Algorithm 4.2: Power1(real x, positive integer n)

```
1  [ Brute-Force method ]
2  local integer i;
3  result ← x; [ t1 1]
4  for i ← 1 to n − 1 do
```

5 [$t2\ n - 1$]

6 result ← result * x; [$t3\ n - 1$]

7 **end**

8 **return** result; [$t4\ 1$]

Algorithm 4.3 is a recursive algorithm and depends upon the fact that x^n can be calculated by squaring $x^{[n/2]}$, and then multiplying the result by x if n was odd. If power1 was used to compute $x^{[n/2]}$, it will require only half the number of multiplications, then we require one more multiplication to square it. In fact this argument can be used recursively, to further divide [n/2] by two, and so on. See Algorithm 4.3. The following trace of the working of power2(2, 5) is obtained:

Algorithm 4.3: Power2(real x, positive integer n)

---

1 [recursive method]

2 **local real** y;

3 **if** $n = 1$ **then**

4 **return** x;

5 **end**

6 $y \leftarrow$ Power2 (x, [n/2]);

7 **if** *odd(n)* **then**

8 **return** $y * y * x$;

9 **else**

10 **return** $y * y$

```
10    return y * y;
11  end
```

invocation no.executed statement

1	$y \leftarrow$ Power2(2, 2)
2	$y \leftarrow$ Power2(2, 1)
3	return 2
2	return 2*2
1	return 4*4*2

How many multiplications are done? only 3, compared to the 4 that would have been required with power1( ). Thus we define $f2(n)$ to be the number of multiplications needed by power2( ). If n is even on every recursive invocation of power2( ), then the minimum of multiplications will be needed. On the other hand, if n is odd every time, then the maximum number of multiplications will be required. The first case occurs when $n = 2^k$, for some positive k.

Then

$$f2(n) = 0 \quad \text{if } k = 0$$

$$= f2(2^{k-1}) + 1 \quad \text{if } k > 0 \qquad (4.1)$$

If $n = 2^k - 1$ then

$$f(n) = 0 \quad \text{if } k = 1$$

$$= f2(2^{k-1} - 1) + 2 \quad \text{for } k > 1 \qquad (4.2)$$

Now from (1), we get:

$k = 0, \quad f2(1) = 0$

$k = 1, \quad f2(2) = 1$

$k = 2, \quad f2(4) = 2$

$k = 3, \quad f2(8) = 3, \qquad$ and so on.

may be $f2(2^{\boldsymbol{m}}) = m$. This can be proven by an inductive proof.

Now from (2) we get:

$k = 1, \quad f2(1) = 0$

$k = 2, \quad f2(3) = 2$

$k = 3, \quad f2(7) = 4$

$k = 4, \quad f2(15) = 6, \qquad$ and so on.

may be $f2(2^{\boldsymbol{m}} - 1) = 2(m - 1)$. This also can be proven by an inductive proof.

This discussion shows that the number of multiplications required $\approx \log_2 n$.

We went through the detailed analysis of these two algorithms because what follows is important. The first is a simple straight forward algorithm, seemingly inefficient, the second is a complicated, but more efficient algorithm. Which one should be used by a user?

If the user needs to compute the exponentials of small powers, that is, n is small, say 5 or less, then the number of multiplications saved by power2( ) compared to power1( ) will be quite small. Further, the programmer may require a significant amount of time to understand, implement, debug and test the more complicated power2( ). As it is the programmer's (human) time that is valuable and not the computer's, use power1( ) in this case.

On the other hand, if exponentials to large integer powers are to be calculated, and there are many instances of these in a program, example, they are within a large loop, then power2( ) should be used. Suppose in a program x^{256} is to be calculated 10 000 times, then power1( ) will require 2 560 000 multiplications, while power2( ) will require 80 000 multiplications. That is where power2( ) will be needed to outperform power1( ).

4.6 ALGORITHM STRATEGIES

The earlier example might have convinced you that the strategy adopted by an algorithm is important in deciding its efficiency. There are a number of classical strategies which are well studied and used extensively. Some of the well known ones are:

1. Brute-force
2. Divide-and-conquer
3. Dynamic programming
4. Greedy algorithms
5. Exotic methods like simulated annealing, genetic algorithms, artificial neural networks, and so on, which are used in special and complex cases.

We shall discuss these later in the forthcoming chapters.

Summary

In this chapter, we have considered characteristics of most of the common and real life algorithms, namely, they have at least one iterative

mon and real-life algorithms, namely, they have at least one iterative component or a loop. The iteration may be explicit through a looping construct or it can be due to a recursive algorithm. In any case, the idea is that a portion of the algorithm statements (or code segments if it is a program) will be executed repeatedly thousands or millions of times. The iterative components or loops are thus major contributors to (in)efficiencies of algorithms.

We have discussed what contributes to the time taken by loops, initial condition, loop invariant, loop termination, and so on. We have also seen how to remove redundant computations, the effect of late termination, and various trade-offs, in setting up an iterative algorithm.

As an expression of a quantitative estimate of the running time of an algorithm, we have introduced the order notation, which helps in specifying various bounds. In Chapter 14, big-oh notation is discussed in detail.

Key Terms

1. **Initial condition** the condition obtained or imposed just before start of some action, usually an iterative process.
2. **Iterative algorithms** see Key Terms in Chapter 2.
3. **Late termination** is termination of an algorithm later than expected, i.e., after going through more number of iterations than required.
4. **Loop invariant** is an invariant used to prove properties of loops. An *invariant* is a condition that does not change, or should not, if the system is working correctly.
5. **Loop termination** in an iterative algorithm, is the event of stopping of the iterations and conditions under which it occurs.
6. **Loops** iterations.
7. **Order notation** is used to denote the execution time (or space) of algorithms and their upper and lower bounds, asymptotically. Big-O, small-o, Big-omega, small-omega and theta are the notations used to denote various kinds of behaviours and their specifications.

8. **Redundant computations** are executed but have no effect on the output of a program.
9. **Trade-offs** Losing one quality or aspect of something in return for gaining another quality or aspect. A space-time or time-memory trade-off is a situation where the programmer can reduce memory use at the cost of slower program execution, or can reduce computation time at the cost of increased memory use.

Exercises

4.1 What is the maximum number of activation records that will reside on the run-time stack of a computer executing `Power2(2, 16)` and `Power1(2, 16)`?

4.2 For the following algorithm segments 4.4 to 4.10, estimate the execution times by assuming that each procedure call takes t_0 unit times and each statement takes time t_1, t_2, and so on. For example, the following algorithm will take $t_0 + n*(t_1 + t_2)$ unit times.

Algorithm 4.4: func0(n, x, y)

```
1  [ t0 ]
2  local integer i;
3  for i ← 1 to n do
4    [ t1 ]
5    x ← x + y; [ t2 ]
6  end
```

In these following algorithms, $[n]$ denotes ceiling of n, that is, the least integer greater than or equal to n, and $[n]$ denotes the floor of n, that is, the greatest integer that is less than or equal to n.

Make suitable assumptions and specify where necessary.

Algorithm 4.5: func1(positive integer n)

```
1   local integer i, j;
2   for i ← 1 to n − 1 do
3    for j ← i + 1 to n do
4     a statement requiring 1 unit time;
5    end
6   end
```

Algorithm 4.6: func2(positive integer n)

```
1   if n = 1 then
2    return 1;
3   else
4    return func2(n − 1) + func2(n − 1);
5   end
```

Algorithm 4.7: func3(positive integer n, positive integer r)

```
1  local integer result, x;
2  result ← 0;
3  for x ← 1 to n do
4    result ← result + Power2(x, r);
5  end
6  return result;
```

Algorithm 4.8: func4(positive integer n)

```
1  local integer result, i;
2  result ← 0;
3  for i ← 2 to n do
4    result ← result + i * func4a (i);
5  end
6  return result;
```

Here, func4a(i) takes log2(i) time units.

Algorithm 4.9: func5(positive integer n)

```
1   local integer temp, i;
```

```
2     if i = 1 then
3      return 1;
4     else
5      temp ← func5([n/2]) + func5([n/2]);
6      for i ← 1 to n do
7       temp ← temp + i;
8      end
9      return temp;
10    end
```

Algorthm 4.10: func6(positive integer n

```
1     local integer temp, i, j;
2     if i = 1 then
3      return 1;
4     else
5      temp ← func6([n/2]) + func6([n/2]);
6      for i ← 1 to n do
7       for j ← 1 to n do
```

```
8          temp ← temp + i * j;
9        end
10     end
11     return temp;
12   end
```

4.3 In this chapter some suggestions are given to reduce the execution time of an algorithm. Suggest a few more ways in which inefficiencies creep into an algorithm or program and suggest remedies.

4.4 In the algorithms `func1` to `func6` given in Exercise 4.2, check whether you can reduce the execution times by any suitable means. Of course, the computed function should not change.

4.5 Are the functions “floor” and “ceiling” available in C, C++ and Java? If not, how will they be implemented?

4.6 If $n = 7$, what are the values of $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$?

4.7 Many of the efficiency problems are handled by a good compiler, by suitably adjusting the generated code, so that some of the inefficiencies which can get introduced are removed. Such compilers are known as optimizing compilers and one good example, which should be available in

the laboratory is the GNU gcc. Obtain details for this compiler and find out various kinds of optimizations it can provide.

4.8 Implement the algorithms given in examples 5, 6 and 7 in Chapter 3, in either Java or C or both. Test in laboratory.

4.9 Implement the following algorithm in Java or C.

Algorithm 4.11: Algorithm max(A, n, j)

```
1   [ find maximum of n items in an array, put it in amax and its
    index in j ]
2   amax ← A[1];
3   j ← 1;
4   for i ← 2 to n do
5     if A[i] > amax then
6       amax ← A[i];
7       j ← i;
8     end
9   end
```

4.10 Develop the required algorithms to read-in the values of the height of the students in your class, then calculate the mean, standard deviation, and variance of these values and print them as outputs. Implement the program in JAVA or C. The program should adhere to the suggestions regarding efficiency, given in the text. Prepare the documentation for the program.

4.11 Modify the algorithm in Exercise 4.10 so that the maximum and minimum values are rejected and not used in finding the mean. Implement this in a manner similar to that explained in Exercise 4.10.

4.12 Knight's Tour: In the game of Chess, a knight can move to any of the eight different positions from a given position, if it is not near an edge of the board. These movements can be described as two steps in horizontal (H) plus one step in vertical (V) directions or one H step and two V steps. Now, the question is "Is it possible for a knight to start on some square and make a sequence of 63 moves in such a way that it visits every square on the board exactly once?" Such a traversal is called Knight's tour.

Design an algorithm for the knight's tour and implement it in JAVA or C. Note that a successful tour will depend upon the strategy you select for choosing the next move from a given position. You should structure your program in such a way that the policy by which the next move is chosen can be modified easily.

Web Resources

Slides for Algorithms Course:

URL: *http://www.cs.yorku.ca/~jeff/otes/3101/slides.html*

Iterative Technique: 01-Iterative.ppt

Iterative Technique. 01-Iterative.ppt.

Iterative Algorithms:

URL: *www.ece.jhu.edu/ABET/520.419.pdf*

An introduction to the study of the structure, behavior and design of iterative algorithms.

Intro to Iterative Algorithms:

URL: *http://www.math.dartmouth.edu/~doyle/docs/icos/icos/node3.html*

Slightly involved.