Computer Algorithms

Segment 1

References:

1. Introduction to Algorithms- Cormen
2. Computer Algorithms-Shanny
3. Algorithms-S. Dasgupta

Definition

- Algorithm is considered as one of the fundamental concepts in computer science
- Informally

1) A*n algorithm is a set of steps that define how a task is performed*

-Examples include algorithms for constructing model airplanes, for operating washing machines and for playing music.

2) A*n algorithm is any well-defined* ***computational procedure*** *that takes some value, or set of values, as* ***input*** *and produces some value or set of values as* ***output.*** An algorithm is thus a sequence of computational steps that transform the input into the output.

- Example of this type is a sorting problem (i.e. sort a sequence of numbers into non-decreasing order)

What kinds of problems are solved by algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. Practical applications of algorithms are ubiquitous and include the following examples:

- The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms.

What kinds of problems are solved by algorithms?

- The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data will travel and using a search engine to quickly find pages on which particular information resides.
- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures, which are based on numerical algorithms and number theory.

What kinds of problems are solved by algorithms?

- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way.
- An oil company may wish to know where to place its wells in order to maximize its expected profit.
- A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.
- An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met.
- An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming,

What kinds of problems are solved by algorithms? (Some Specific Problems)

- We are given a road map on which the distance between each pair of adjacent intersections is marked, and we wish to determine the shortest route from one intersection to another. The number of possible routes can be huge, even if we disallow routes that cross over themselves. How do we choose which of all possible routes is the shortest?
- We are given two ordered sequences of symbols, X =<x1, x2,…, xm> and Y=<y1, y2, …,yn>, and we wish to find a longest common subsequence of X and Y . A subsequence of X is just X with some (or possibly all or none) of its elements removed.
- We are given n points in the plane, and we wish to find the convex hull of these points.

Definition

- Formally

  - ***An algorithm is an ordered set of unambiguous, executable steps, defining a terminating process***

- Question
- In what sense do the steps described by the following list of instructions fail to constitute an algorithm?

  Step 1. Take a coin out of your pocket and put it on the table

  Step 2. Return to Step 1

- A machine-compatible representation of an algorithm is called a **program**

Criteria/Properties

•All algorithms must satisfy the following criteria:

1. Input: Zero or more quantities are externally supplied.

2. Output: At least one quantity is produced.

3. Definiteness: Each instruction is clear and umabiguous.

4. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It must be feasible.

•Algorithms that are definite and effective are also called ***computational procedures***. One important example of computational procedures is the operating system of a digital computer.

Areas of Study

- The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify

1. How to devise algorithms: an understanding of the algorithmic structures and their representation in the form of Pseudocode or flowcharts.

   - algorithm discovery - problem solving approaches/techniques

2. How to validate algorithms: determining the correctness
3. How to analyze algorithms: determining the time and storage of an algorithm requires.
4. How to test a program: debugging

- Algorithm specification

  - Pseudocode: algorithms are represented with precisely defined textural structure
  - Flowcharts

History

- The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammad ibn Musa al Khowarizmi, who wrote a text book on mathematics.
- Began as a subject in mathematics.
- Greek mathematician Euclid invented an algorithm for finding the greatest common divisor (gcd) of two positive integer (between 400 and 300 B.C.) - meaningful algorithm ever discovered.
- Weaving loom invented in 1801 by a Frenchman, Joseph Jacquard
- Charles Babbage, English mathematician - *the difference engine* and ***the analytical engine***, capable of executing algorithms

Algorithms as a technology

- Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms?
- Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.
- Analysis of algorithm or performance analysis refers to the task of determining how mach computing time and storage an algorithm requires. Generally, by analyzing several candidate algorithm for a problem, a most efficient one can be easily identified.

Algorithms as a technology

Efficiency

• Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

we will see two algorithms for sorting. The first, ***insertion sort, takes time roughly equal to*** $c_1 n^2$ ***to sort n items, where c1*** is a constant that does not depend on n. The second, ***merge sort, takes time roughly equal to*** $c_2 n \log_2 n$***, where*** c2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that c1 < c2. Let us fit a faster computer (computerA) running insertion sort against a slower computer (ComputerB) running merge sort. Each computer sort an array of 10 million numbers.

Algorithms as a technology

Suppose that computer A executes 10 billion instructions per second and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B. Suppose insertion sort solved in $2n^2$ time and merge sort solved in $50n\log_2 n$ time.

Computer A takes 20,000 seconds (more than 5.5 hours) while computer B takes ≈1163 seconds (less than 20 minutes). Computer B runs more than 17 times faster than computer A!

The example above shows that we should consider algorithms, like computer hardware, as a ***technology. Total system performance depends on choosing efficient*** algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

Time needed by algorithms of diff time functions for diff problem sizes

Size of n	n	n^2	n^3	2^n	n!	n^n
1	0.000001	0.000001	0.000001	0.000002	0.000001	0.000001
5	0.000005	0.000025	0.000125	0.000032	0.000120	0.003125
10	0.00001	0.0001	0.001	0.001024	3.6288	2.778 hrs
20	0.00002	0.0004	0.008	1.04858	78218 yrs	3.37×10^{12} yrs
50	0.00005	0.0025	0.125	36.1979 yrs	9.77×10^{60} yrs	2.87×10^{70} yrs
100	0.0001	0.01	1.00	4×10^{17} yrs	3×10^{143} yrs	3.2×10^{183} yrs

Assuming 1m operations/sec.

Insertion Sort

We start with *insertion sort, which is an efficient algorithm for sorting a small* number of elements.

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

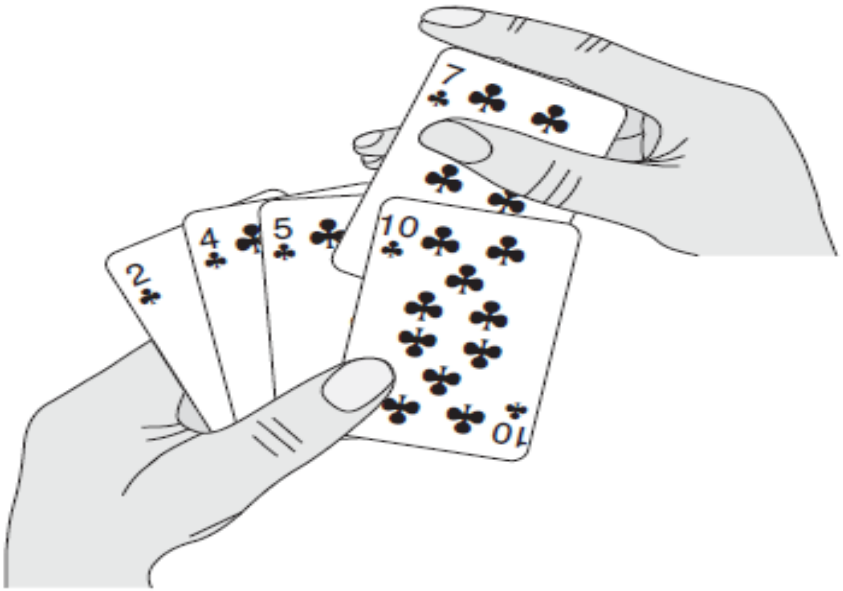


Figure 2.1 Sorting a hand of cards using insertion sort.

Insertion Sort (Cont.)

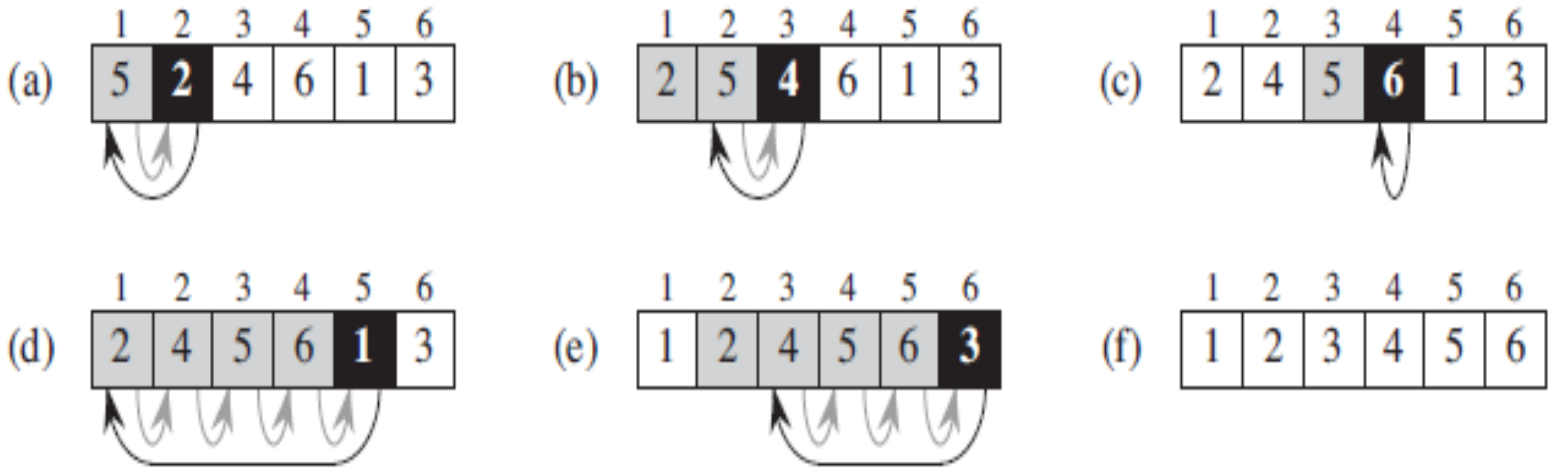


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. (f) The final sorted array.

Insertion Sort (Cont.)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariants and the correctness of insertion sort

Figure 2.2 shows how this algorithm works for A =<5; 2; 4; 6; 1; 3>. The index j indicates the "current card" being inserted into the hand. At the beginning of each iteration of the **for loop, which is indexed by j , the subarray consisting** of elements A[1.. j-1] constitutes the currently sorted hand, and the remaining subarray A[j+1, n] corresponds to the pile of cards still on the table. In fact, elements A[1…j-1] are the elements *originally in positions 1 through j-1, but* now in sorted order. We state these properties of A[1… j-1] formally as a ***loop invariant.***

Loop invariants and the correctness of insertion sort

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.
Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Let us see how these properties hold for insertion sort.

Correctness of insertion sort

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when j=2. The subarray A[1.. j-1], therefore, consists of just the single element A[1], which is in fact the original element in A[1]. Moreover, this subarray is sorted, which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the for loop works by moving A[j-1], A[j -2], A[j-3], and so on by one position to the right until it finds the proper position for A[j] (lines 4–7), at which point it inserts the value of A[j] (line 8). The subarray A[1… j-1] then consists of the elements originally in A[1… j], but in sorted order. Incrementing j for the next iteration of the for loop then preserves **the loop invariant.**

Correctness of insertion sort

Termination: Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that j >A.*length=n. Because* each loop iteration increases j by 1, we must have j=n+1 at that time. Substituting n+1 for j in the wording of loop invariant, we have that the subarray A[1… n] consists of the elements originally in A[1…n], but in sorted order. Observing that the subarray A[1…n] is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

Pseudocode conventions

Home work: See the page 19 & 20

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

Analyzing algorithms

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure.

Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one.

Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

Level of analysis

- Time taken by an algorithm?
  - Number of seconds a program takes to run? No…
  - Too machine dependent: not measuring the algorithm so much as the program, machine, implementation
- Here, not interested in low level machine details
  - Want technology-independent answers
- Interested rather in measuring
  - Which data take most space
  - Which operations take most time in a *device-independent* way

Level of analysis (Cont.)

- Notion of elementary operation
  - 1 assignment, 1 addition, 1 comparison (test), …
- Determine how many elementary operations algorithm performs for inputs of different sizes (problem sizes), or same size diff. order
- Difference thus between performance and complexity
  - Complexity affects performance, but not v.v.
  - Interested in how resource requirements *scale* as problem gets larger

Best, worse, average cases

- Hardly ever true that algorithm takes same time on different instances of same size
- Choice: best, worse or average case analysis
- Best case analysis
  - Time taken on best input of size *n*
- Worst case analysis
  - Time taken on worst input of size *n*
- Average case analysis
  - Time taken as *average* of time taken on inputs of size *n*

Which to choose?

- Usually interested in worst case scenario:
  - The most operations that might be made for some problem size
- Worst case is only *safe* analysis – guaranteed upper bound (best case too optimistic)
- Average case analysis harder
  - Usually have to assume some probability distribution of the data, more sophisticated model
  - E.g. if looking for a specific letter in a random list of letters, might expect letter to appear 1/26 of time (for English)

```
int Max(int A[n])
\\ returns Max value in array A
{
 int i;
 Max = A[0];
 for (i = 1; i < n; i++)
    if (Max < A[i])
      Max = A[i];
 return(Max);
}
```

- Worst case: THEN always executed

Scale, order vs exactness

- When looking at complexity, usually ignore *exact* number of operations and cost of small, infrequent ones – concentrate on size of input, i.e.
- Interested more in how number of operations relates to *problem size* – of *big* problems

Analysis example

```
// sum first n integers in array a
int sum(int a[], int n)
{
 int sum = 0;
 for (int j = 0; j < n; j++)
    sum = sum + a[j];
 return(sum);
}
```

General methodology

1. Characterize the size of the input
   - Input is an array containing at least *n* integers
   - Thus, size of input is *n*
2. Count how many operations (steps) are taken in the algorithm for an input of size *n*
   - 1 step is an elementary operation
   - +, <, a[j] (indexing into an array), =, …

Detail of analysis

```
int sum(int a[], int n) {
 int sum = 0;  ←————————————  ①
 for  (int    j = 0;    j < n;      j++    )
                ②         ③          ④
    sum     =      sum       +        a[j];
              ⑦              ⑥        ⑤
 return(sum);  ←————————  ⑧
}
```

- 1,2,8: only happen once (so 3 such operations)
- 3,4,5,6,7: happen once *for each iteration* of loop (5 * *n*)
- Total operations: 5*n* + 3
- Complexity function: f(n) = 5*n* + 3

How does size affect running time?

- $5n + 3$ is an estimate of running time for different values of *n*

n	Operations
10	53
100	503
1000	5003
1000000	5000003

- As *n* grows, number of operations grows in *linear* proportion to *n*, for this sum function

Summary of methodology

- Count operations/steps taken by algorithm
- Use count to derive formula, based on size of problem n
  - Another formula might be, e.g.: n^2
  - Or $2n$ or $n^2/2$ or $(n+1)^2$ or $7n^2 + 12n + 4$
- Use formula to help understand overall efficiency

Usefulness?

- What if we kept doubling size of *n*?

n	$\log_2 n$	5n	$n\log_2 n$	n^2	2^n
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$
10000	13	50000	10^5	10^8	$\sim 10^{3010}$

Analyzing Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Analyzing Insertion Sort

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes ci steps to execute and executes n times will contribute ci.n to the total running time. To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the *cost and times columns, obtaining*

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Analyzing Insertion Sort

- Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given.
- The best case occurs if the array is already sorted. For each i=2, 3,4, ….n, we find that A[j]<= *key* in while loop when j has its initial value of i-1. Thus $t_i = 1$ for i=2,3,4,….n, and the best case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

- T(n) can be expressed as $an+b$ for constant a and b. It is a linear function of n.

Analyzing Insertion Sort

- If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element A[j] with each element in the entire sorted subarray A[1… j-1], and so tj = j for j=2,3…,n. Noting that.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

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

Analyzing Insertion Sort

We find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

- The worst case can be expressed as $an^2 + bn + c$ for some constant a,b,c . It is a quadratic function of n

Analyzing Insertion Sort

Worst-case and average-case analysis

In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the ***worst-case running time, that is, the longest running time for any*** input of size n. We give three reasons for this orientation.

1. The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.

Worst-case and average-case analysis

2. For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.

3. The "average case" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray A[1.. j-1] to insert element A[j]? On average, half the elements in A[1…j-1] are less than A[j], and half the elements are greater. On average, therefore, we check half of the subarray A[1…j-1], and so tj is about j/2. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

•The scope of average-case analysis is limited, because it may not be apparent what constitutes an "average" input for a particular problem.

Asymptotic Notation

- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N={0,1,2,…}. Such notations are convenient for describing the worst-case running-time function T(n), which is usually defined only on integer input sizes.
- Three basic asymptotic notations

1. *O*-notation (Big "oh"): When we have an asymptotic upper bound, we use *O*-notation. For a given function g(n), we denote by *O(g(n))* (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

O(g(n)) ={*f(n)*: there exist positive constants c and N such that 0 <= *f(n)* <= *c g(n)* for all *n*>=N}

Asymptotic Notation (cont.)

2. Ω-notation (Big "omega"): Ω-notation provides an **asymptotic lower bound**. For a given function g(n), we denote by *Ω(g(n))* (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

Ω(g(n)) ={*f(n)*: there exist positive constants c and N such that

0 <= *c g(n)* <= *f(n)* for all *n*>=N}.

3. Θ-notation (Big "theta"): Θ-notation asymptotically bounds a function from above and below. For a given function g(n), we denote by *Θ(g(n))* (pronounced "big-theta of g of n" or sometimes just "theta of g of n") the set of functions

Θ(g(n)) ={*f(n)*: there exist positive constants c1, c2 and N such that 0 <= *c1 g(n)* <= *f(n)* <= *c2 g(n)* for all *n*>=N}.

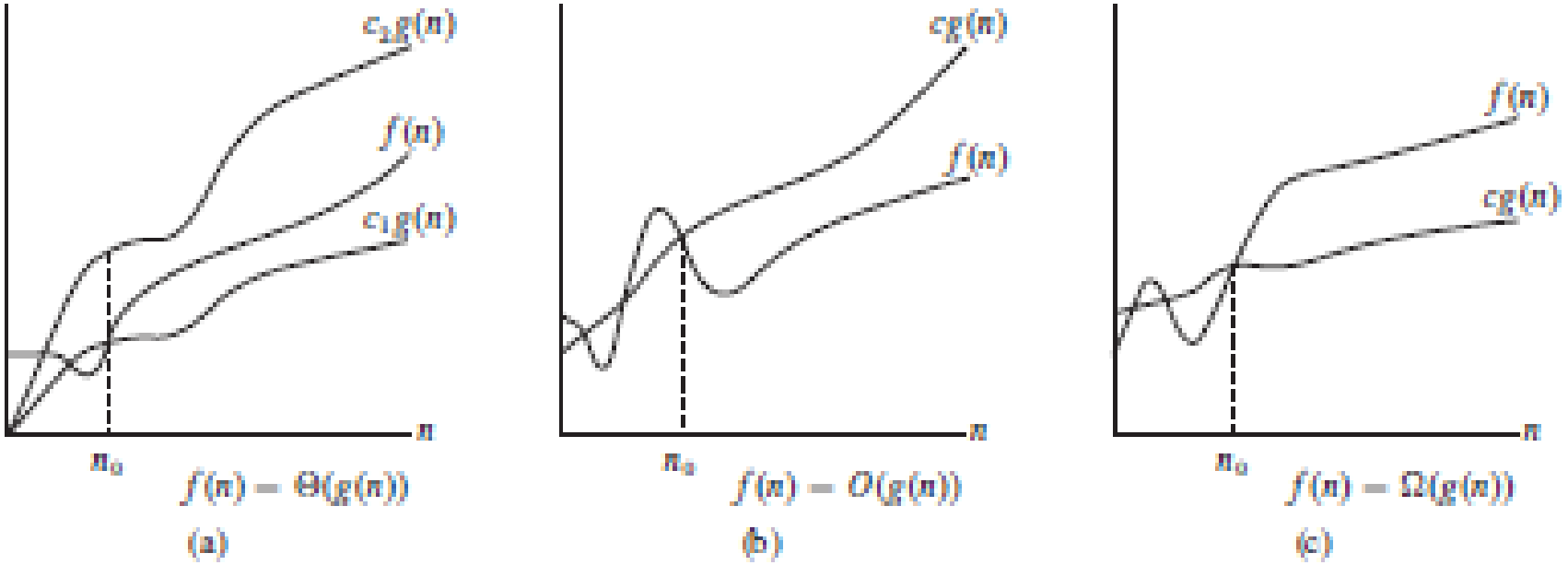


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. (a) Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive. (b) O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. (c) Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

Asymptotic Notation (cont.)

Big-O: $3n+2=O(n)$ as $3n+2<=4n$ for all $n>=2$.

$100n+6=O(n)$ as $100n+6<=101n$ for all $n>=6$.

$10n^2+4n+2=O(n^2)$ as $10n^2+4n+2<=11n^2$ for all $n>=5$.

$1000n^2+100n-6=O(n^2)$ as $1000n^2+100n-6<=1001n^2$ for all $n>=100$.

$3n+2!=O(1)$ as $3n+2$ is not less than or equal to c for any constant c and all $n>=N$. $10n^2+4n+2!=O(n)$.

Big-Ω: $3n+2=\Omega(n)$ as $3n+2>=3n$ for all $n>=1$.

$100n+6=\Omega(n)$ as $100n+6>=100n$ for all $n>=1$.

$10n^2+4n+2=\Omega(n^2)$ as $10n^2+4n+2>=n^2$ for all $n>=1$. Also observe that $3n+3=\Omega(1)$. $10n^2+4n+2=\Omega(n)$. $10n^2+4n+2=\Omega(1)$.

Big-Θ: $3n+2=\Theta(n)$ as $3n+2>=3n$ for all $n>=2$ and $3n+2<=4n$ for all $n>=2$ so $c1=3, c2=4$ and $N=2$. $10n^2+4n+2=\Theta(n^2)$. $10logn+4=\Theta(logn)$. $3n+2 != \Theta(1)$. $3n+3!=\Theta(n^2)$. $10n^2+4n+2!=\Theta(n)$.

$10n^2+4n+2!=\Theta(1)$.

Asymptotic Notation (cont.)

- For more example see the pages 29, 30, 31, 32, 33 of book by Sahni)
- Home work : (i) Is $2^{n+1} = O(2^n)$? (ii) Is $2^{2n} = O(2^n)$?

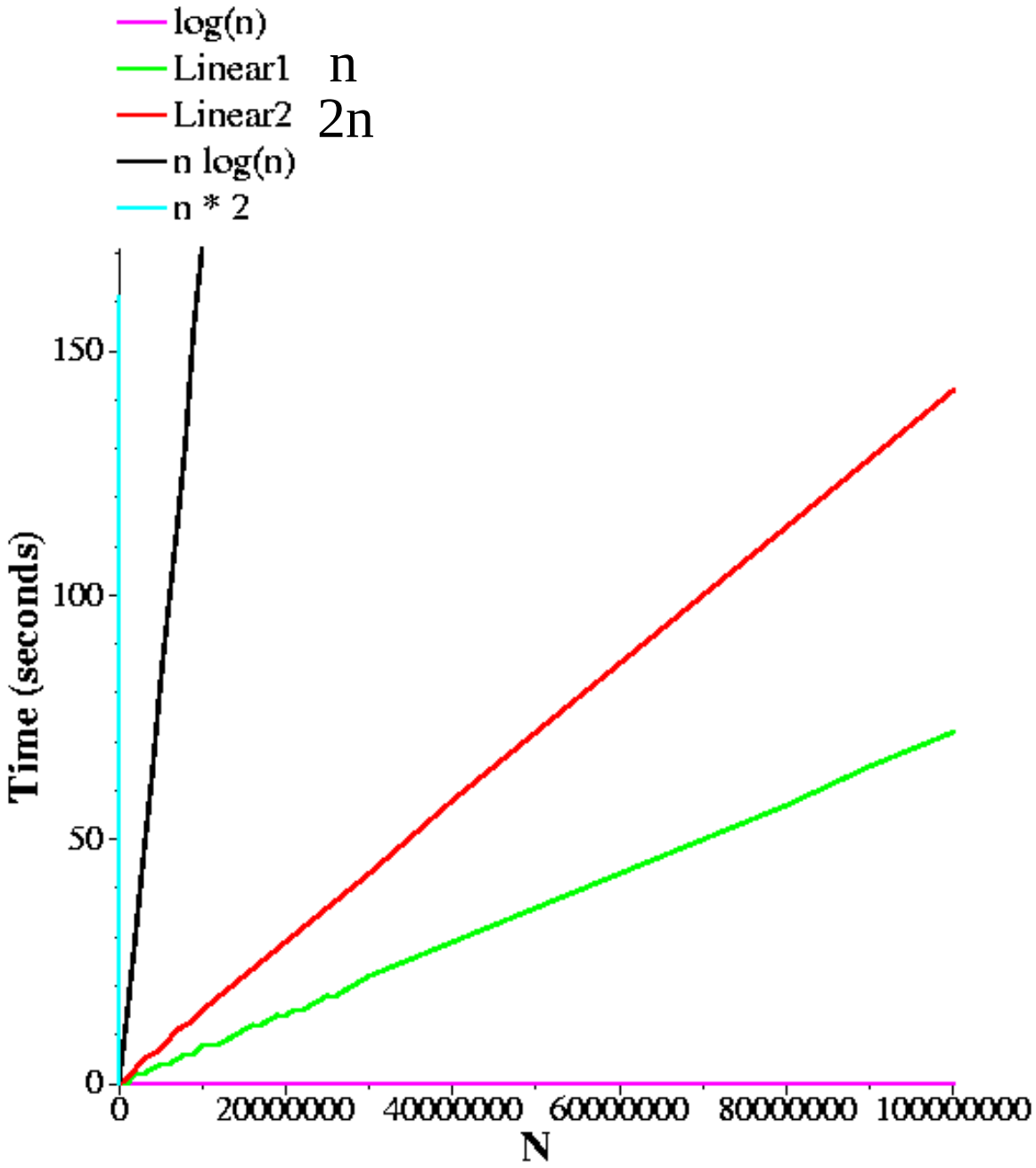
Some commonly found orders of growth

increasing complexity

O(1)	Constant (bounded)
O(log *n*)	Logarithmic
O(n)	Linear
O(*n*log*n*)	Log linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential
O(*n*!)	Exponential (Factorial)
O(n^n)	Exponential

polynomial

polynomial good, exponential bad


log(n)
Linear1
Linear2
n log(n)
n * 2
n
2n
150
100
50
0
Time (seconds)
0
20000000
40000000
60000000
80000000
100000000
N

Determining complexities

- Simple statements: O(1)
  - Assignments of simple data types: int x = y;
  - Arithmetic operations: x = 5 * y;
  - Indexing into an array: array[i];
  - (de)referencing pointers: listptr =list -> next;
  - Declarations of simple data types: int x, y;
  - Simple conditional tests: if (i < 12)
    - But not e.g. if (SomeFunction() < 25)

If…then…else statements

- Worst case is slowest possibility

```
if (cond) {
sequence of statements 1        say this is O(n)
}
else {
sequence of statements 2        say this is O(1)
}
```

- Overall, *this* if…then…else is thus O(*n*)

Loops

- Two parts to loop analysis
  - How many iterations?
  - How many steps for each iteration?

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

- Loop executes *n* times
- 4 = O(1) steps per iteration
- Total time is here *n* * O(1) = O(*n* * 1) = O(*n*)

Loops: simple

```
int sum=0;
for (int j=0; j < 100; j++)
   sum = sum + j;
```

- Loop executes 100 times
- 4 = O(1) steps per iteration
- Total time here is

  100 * O(1) = O(100* 1) = O(1)
- Thus, faster than previous loop, for values up to 100

Loops: while loops

```
done=FALSE; n=22;
while (!done) {
  result = result * n;
  n--;
  if (n == 1) done = TRUE;
}
```

- Loop terminates when done==TRUE
- This happens after *n* iterations
- O(1) time per iteration
- O(*n*) total time here

Loops: nested

```
for (i=0; i < n; i++) {
   for (j=0; j<m; j++) {
     sequence of statements
  }
}
```

- Outer loop executes *n* times
- For every outer, inner executes *m* times
- Thus, inner loop total = *n* * *m* times
- Complexity is O(*n***m*)
- Where inner condition is **j<n** (a common case), total complexity is O(n^2)

Sequences of statements

- For a sequence, determine individual times and add up

```
for (j=0; j<n; j++)
  for (k=0; k<j; k++)
    sum = sum + j*k;
```

$O(n^2)$

```
for (l=0; l<n; l++)
  sum = sum –1;
```

$O(n)$

```
printf("Sum is now %d",sum);
```

$O(1)$

- Total is: $O(n^2) + O(n) + O(1) = O(n^2)$

When do constants matter?

- Normally we drop constants and lower-order terms when calculating Big-O.
- This means that 2 algorithms could have the same Big-O, although one may always be faster than the other
- In such cases, the constants do matter if we want to tell which algorithm is actually faster
- However, they do not matter when we want to know how algorithms scale under growth
- $O(n^2)$ always faster than $10n^2$ but for both, if problem size doubles, time quadruples

Example 1

- Prove $7n^2 + 12n$ is $O(n^2)$
- Must show that 2 constants c and N exist such that

  $7n^2 + 12n \leq c\, n^2 \qquad \forall n \geq N$
- Let N=1 (existence proof – can pick any N we want to, only need to show one exists)
- Now, does there exist some c such that

  $7n^2 + 12n \leq c\, n^2 \qquad \forall n > 1$

- As N > 1, this means

  $7n^2 + 12n < 7n^2 + 12n^2$

  add: $7n^2 + 12n^2 = 19n^2$

  and observe: $19n^2 \leq 19n^2$
- Thus, let c be 19 (to be sure, let us say 20)
- Checking: check for N > 1, so take N=2
- Is $7 * 2^2 + 12 * 2 < 20 * 2^2$?
  - Yes: 52 < 80
- Thus $7 * n^2 + 12 * n \leq 20 * n^2 \quad \forall n \geq 2$
- Thus $7n^2 + 12n \leq 20n^2 \quad \forall n \geq 2$
- Proved: $7n^2 + 12n$ is $O(n^2)$

Example 2

- Prove $(n + 1)^2$ is $O(n^2)$
- $(n + 1)^2$ expands to $n^2 + 2n + 1$
- $n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$
- $n^2 + 2n^2 + n^2 = 4n^2$
- Thus, c is 4
- We can find N=1 as before
- Arrive at: $(n + 1)^2 \leq 4n^2 \quad \forall n \geq 1$
- Thus, $(n + 1)^2$ is $O(n^2)$

More on Example 2

- Could also pick N=3 and c=2
- As $(n + 1)^2 \leq 2n^2 \quad \forall n \geq 3$
- Because $2n + 1 \leq n^2 \quad \forall n \geq 3$

- However, cannot pick N=0
- Because $(0+1)^2 > c(0)^2$ for any $c > 0$

Example 3

- Prove $3n^2 + 5$ is $O(n^2)$
- We choose constant c = 4 and N = 2
- Why? Because $\forall n \geq \mathbf{2}$:

$$\mathbf{3}n^2 + 5 \leq \mathbf{4}n^2$$

Thus, by our definition, $3n^2 + 5 \in O(n^2)$

- Now, prove $3n^2 + 5$ is **not** $O(n)$ – easy:
- Whatever constant c and value N one chooses, we can always find a value of *n* greater than N such that $3n^2 + 5$ **is greater** than c *n*

Example 4

- Prove $2n+4$ has order n i.e. $2n+4 \in O(n)$
- Clearly, $2n = 2n \quad \forall n$
- $4 \leq n$ if $n \geq 4$
- $2n + 4 \leq 3n \quad \forall n \geq 4$
- Thus, our constants are c = 3 and N = 4
- Re-express: $2n + 4 \leq c\, n \quad \forall n \geq N$
- Thus, by our definition
- $2n+4 \in O(n)$

Example 5

- Prove $17n^2 + 45n + 46$ is $O(n^2)$
- $17n^2 \leq 17n^2 \quad \forall n$
- $45n \leq n^2 \quad \forall n \geq 45$
- $46 \leq n^2 \quad \forall n \geq 7$
- $17n^2 + 45n + 46 \leq 17n^2 + n^2 + n^2$
- So, $17n^2 + 45n + 46 \leq 19n^2$
- Thus c = 19 and N = 45
- Thus $17n^2 + 45n + 46 \leq 19n^2 \quad \forall n \geq 45$
- Proved: $17n^2 + 45n + 46$ is $O(n^2)$