Algorithm Analysis

Algorithm

- An ***algorithm*** is a set of instructions to be followed to solve a problem.

  – There can be more than one solution (more than one algorithm) to solve a given problem.
  – An algorithm can be implemented using different programming languages on different platforms.

Algorithms (cont.)

- An algorithm must be correct. It should correctly solve the problem.

e.g. For sorting, this means even if

i. the input is already sorted, or

ii. it contains repeated elements.

- Once we have a correct algorithm for a problem, we have to determine the efficiency (running time performance and memory requirement) of that algorithm.

Performance Measures of Algorithms

There are *two* performance measures for algorithms:

1. Running Time
    - Every instruction takes time.
    - How fast does the algorithm perform?
    - What affects its runtime?
    - If an algorithm requires years to run on an average computer, then it is not useful.

Performance Measures of Algorithms (cont.)

2. Space (Memory Requirement)

- Data structures take space
- What kind of data structures can be used?
- How does choice of data structure affect the runtime?
- If an algorithm requires several gigabytes of main memory to run, then it is not useful.

Content that we will cover

- We will focus on running time:
  - How to estimate the time required for an algorithm
  - How to reduce the time required for an algorithm.
- We will also explain how to estimate memory requirement of an algorithm.

Analysis of Algorithms

- ***Analysis of Algorithms*** is the area of computer science that provides tools to analyze the efficiency of different methods of solutions.
- How do we compare the time efficiency of two algorithms that solve the same problem?

Analysis of Algorithms

Naïve Approach: implement these algorithms in a programming language (C), and run them to compare their time requirements.

Analysis of Algorithms

Comparing the programs (instead of algorithms) has difficulties.

- *How are the algorithms coded?*
  - Comparing running times means comparing the implementations.
  - We should not compare implementations, because they are sensitive to programming style.
- *What computer should we use?*
  - We should compare the efficiency of the algorithms independently of a particular computer (hardware).
- *What data should the program use?*
  - Any analysis must be independent of specific data.

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers, or data.*

- To analyze **running time of algorithms**:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we will express the efficiency of algorithms using growth functions.

Running Time

- **Running time of an algorithm** is a function of the *input size* such that it gives the *time requirement* of the algorithm to *solve the problem for a given input*.
- To compute running time of an algorithm, the number of computations that were made by the algorithm such as:
    - Number of assignment statements,
    - Number of arithmetical computations (+, -, *, /),
    - Number of iterations,
    - Number of disk accesses, etc.

  are computed.

Running Time

- As running time of an algorithm, time in terms of seconds or minutes is not measured, because

e.g.,

Algorithm A runs in 30 seconds in computer C1, the same algorithm A runs in 0.001 seconds in computer C2.

Instead, we do running time analysis.

The Running Time of Algorithms

- Each operation in an algorithm (or a program) has a cost.

  ➔ Each operation takes a certain amount of time.

`count = count + 1;` ➔ take a certain amount of time, but it is constant

A sequence of operations:

```
count = count + 1;
sum = sum + count;
```

`count = count + 1;`	**Cost: c_1**
`sum = sum + count;`	**Cost: c_2**

➔ **Total Cost = $c_1 + c_2$**

The Running Time of Algorithms (cont.)

Example: Simple If-Statement

Code	Cost	Times
`if (n < 0)`	c1	1
`    result = -1*n;`	c2	1
`else`		
`    result = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

The Execution Time of Algorithms (cont.)

Example: Simple Loop

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    i = i + 1;`	c4	n
`    sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

Example: Nested Loop

Code	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`    j = 1;`	c4	n
`    while (j <= n) {`	c5	n*(n+1)
`        sum = sum + i;`	c6	n*n
`        j = j + 1;`	c7	n*n
`    }`		
`    i = i +1;`	c8	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

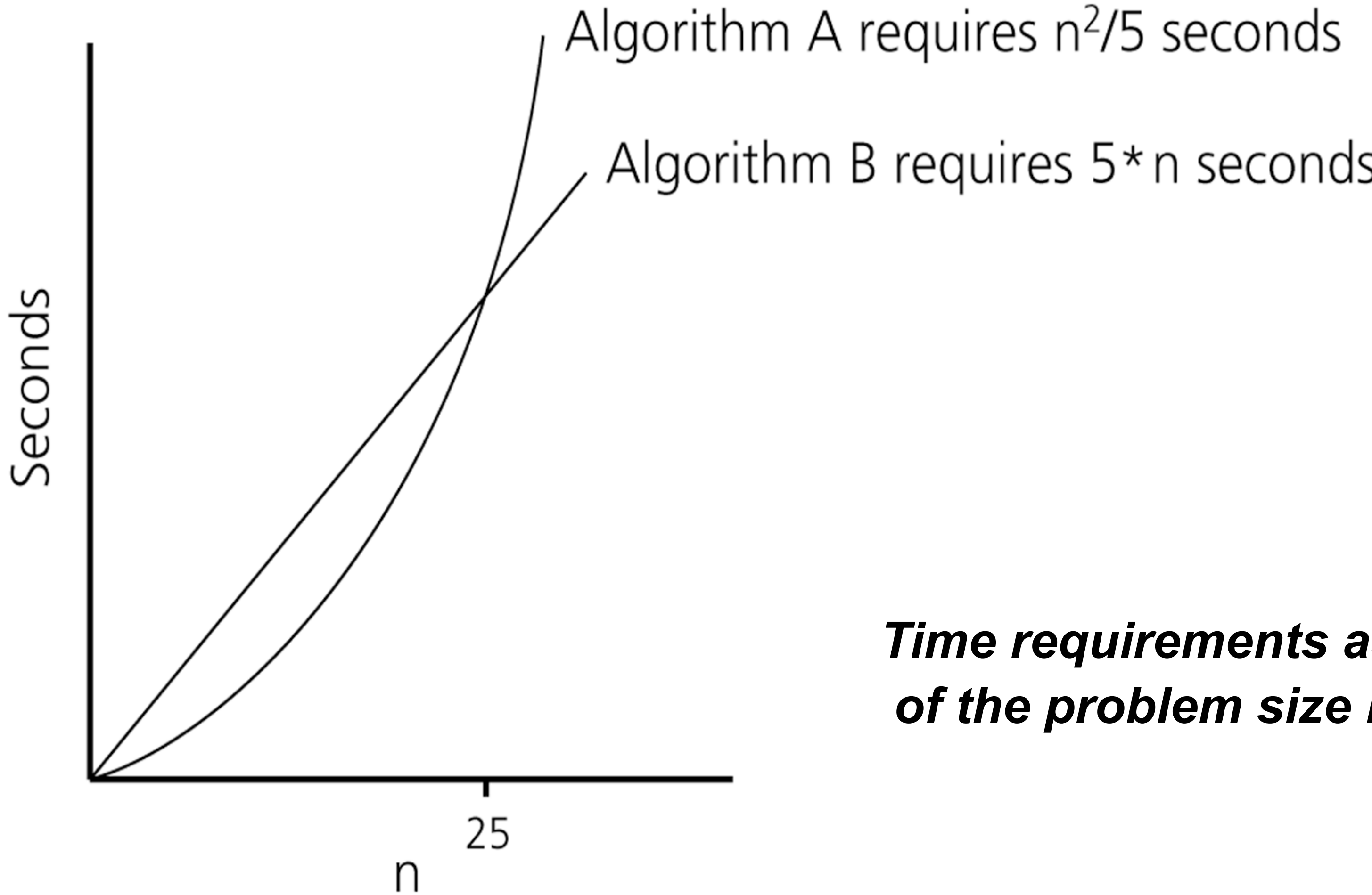
General Rules for Estimation

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sizes of all loops.
- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else**: Never more than the running time of the test plus the larger of running times of S1 and S2.

Algorithm Growth Rates

- We measure an algorithm's time requirement as a function of the *problem size*.
  - Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- So, for instance, we say that (if the problem size is n)
  - Algorithm A requires $\mathbf{5*n^2}$ time units to solve a problem of size n.
  - Algorithm B requires $\mathbf{7*n}$ time units to solve a problem of size n.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to $\mathbf{n^2}$.
  - Algorithm B requires time proportional to **n**.
- An algorithm's proportional time requirement is known as ***growth rate***.
- We can compare the efficiency of two algorithms by comparing their growth rates.

Algorithm Growth Rates (cont.)



Time requirements as a function of the problem size n

Common Growth Rates

Function	Growth Rate Name
c	Constant
$log\ N$	Logarithmic
log^2N	Log-squared
N	Linear
$N\ log\ N$	Log with linear multiplier
N^2	Quadratic
N^3	Cubic
2^N	Exponential
$N\ !$	Factorial

Figure 6.1

Running times for small inputs

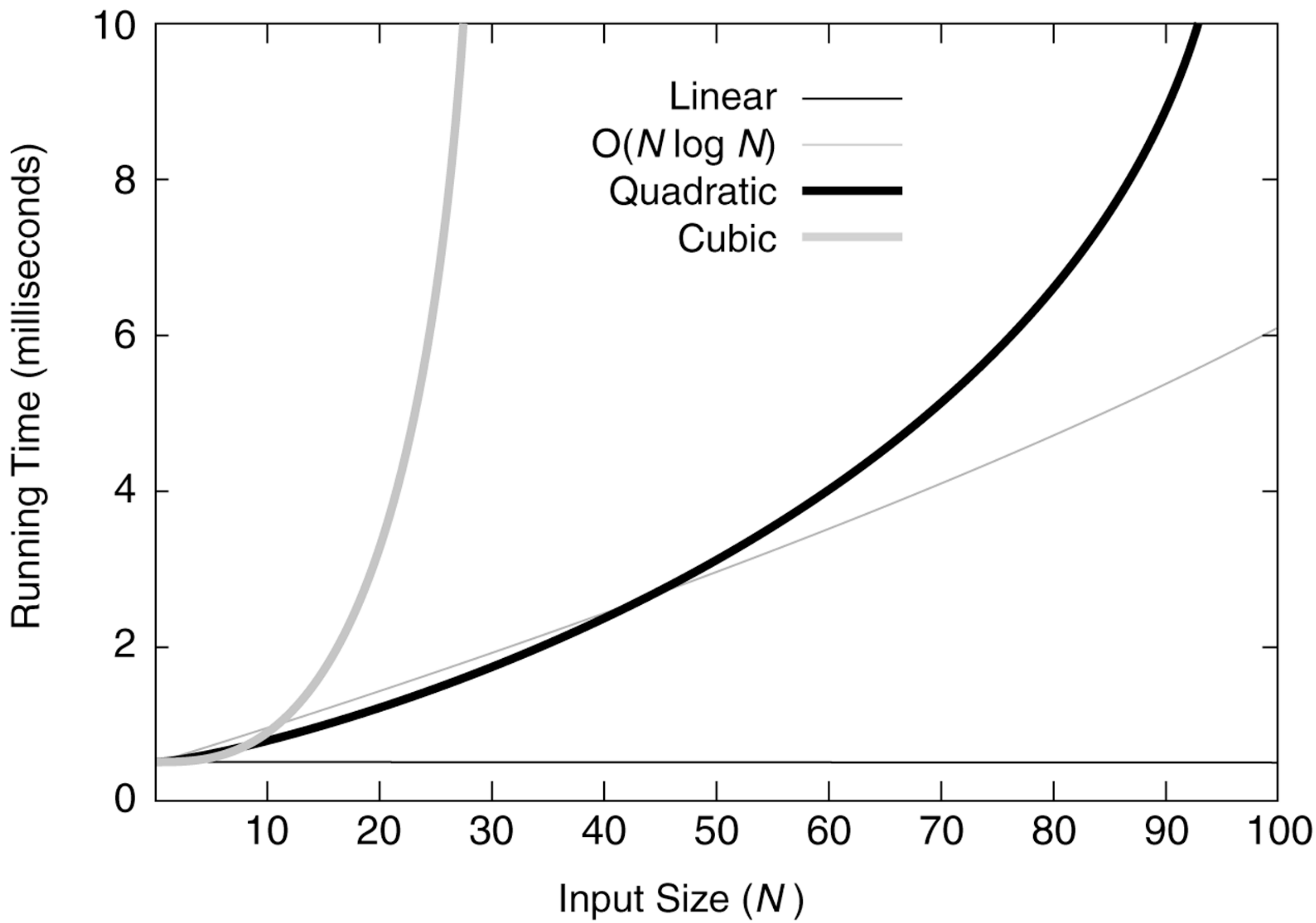
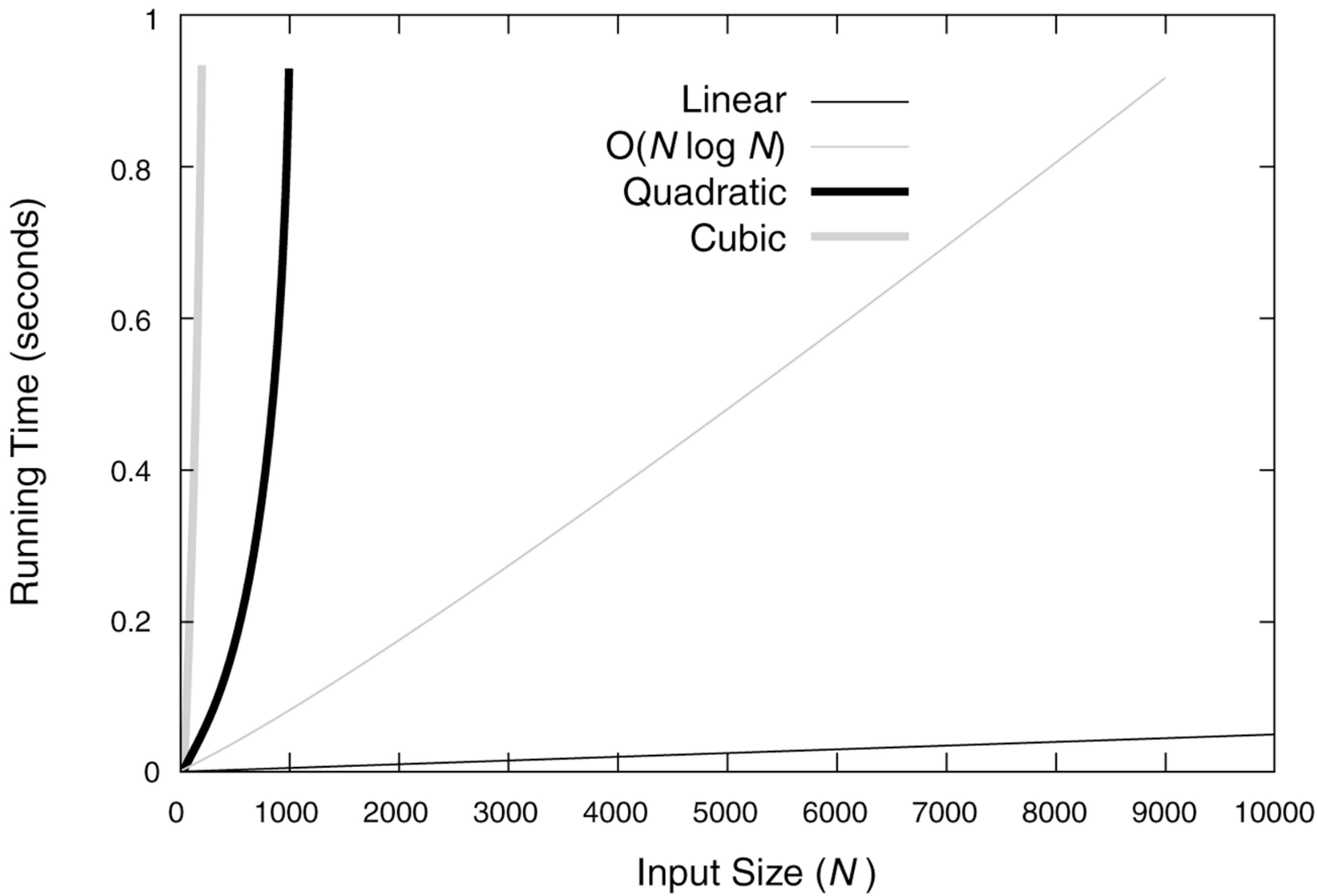


Figure 6.2

Running times for moderate inputs



Order-of-Magnitude Analysis and Big O Notation

- If *Algorithm A requires time proportional to f(n),* Algorithm A is said to be **order f(n),** and it is denoted as **O(f(n)).**
- The **function f(n)** is called the algorithm's **growth-rate function**.
- Since the capital O is used in the notation, this notation is called the **Big O notation**.
- If Algorithm A requires time proportional to $\mathbf{n^2}$, it is $\mathbf{O(n^2)}$.
- If Algorithm A requires time proportional to **n**, it is **O(n).**

Definition of the Order of an Algorithm

Definition:

Algorithm A is order f(n) – denoted as O(f(n)) – if constants c and n_0 exist such that A requires no more than c*f(n) time units to solve a problem of size $n \geq n_0$.

- The requirement of $\mathbf{n \geq n_0}$ in the definition of O(f(n)) formalizes the notion of sufficiently large problems.
  - In general, many values of c and n can satisfy this definition.

Order of an Algorithm

- If an algorithm requires $n^2 – 3\ n+10$ seconds to solve a problem of size n. If constants c and n_0 exist such that

  $$c\ n^2 > n^2 – 3\ n+10 \qquad \text{for all } n \geq n_0 .$$

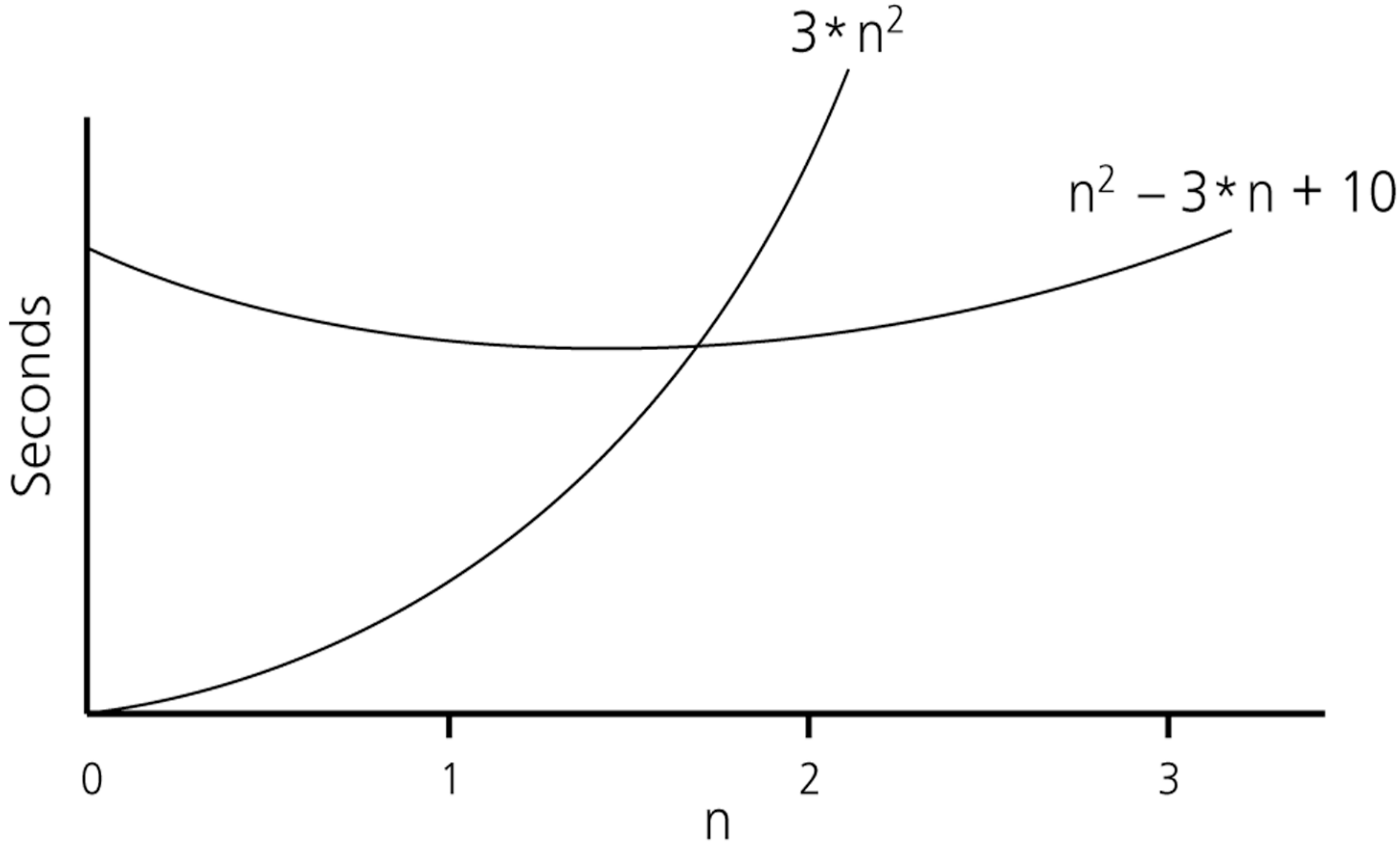
  the algorithm is order n^2 (In fact, c is 3 and n_0 is 2)

  $$3\ n^2 > n^2 – 3\ n+10 \qquad \text{for all } n \geq 2 .$$

  Thus, the algorithm requires no more than $c*n^2$ time units for $n \geq n_0$,

  So it is **O(n^2)**

Order of an Algorithm (cont.)

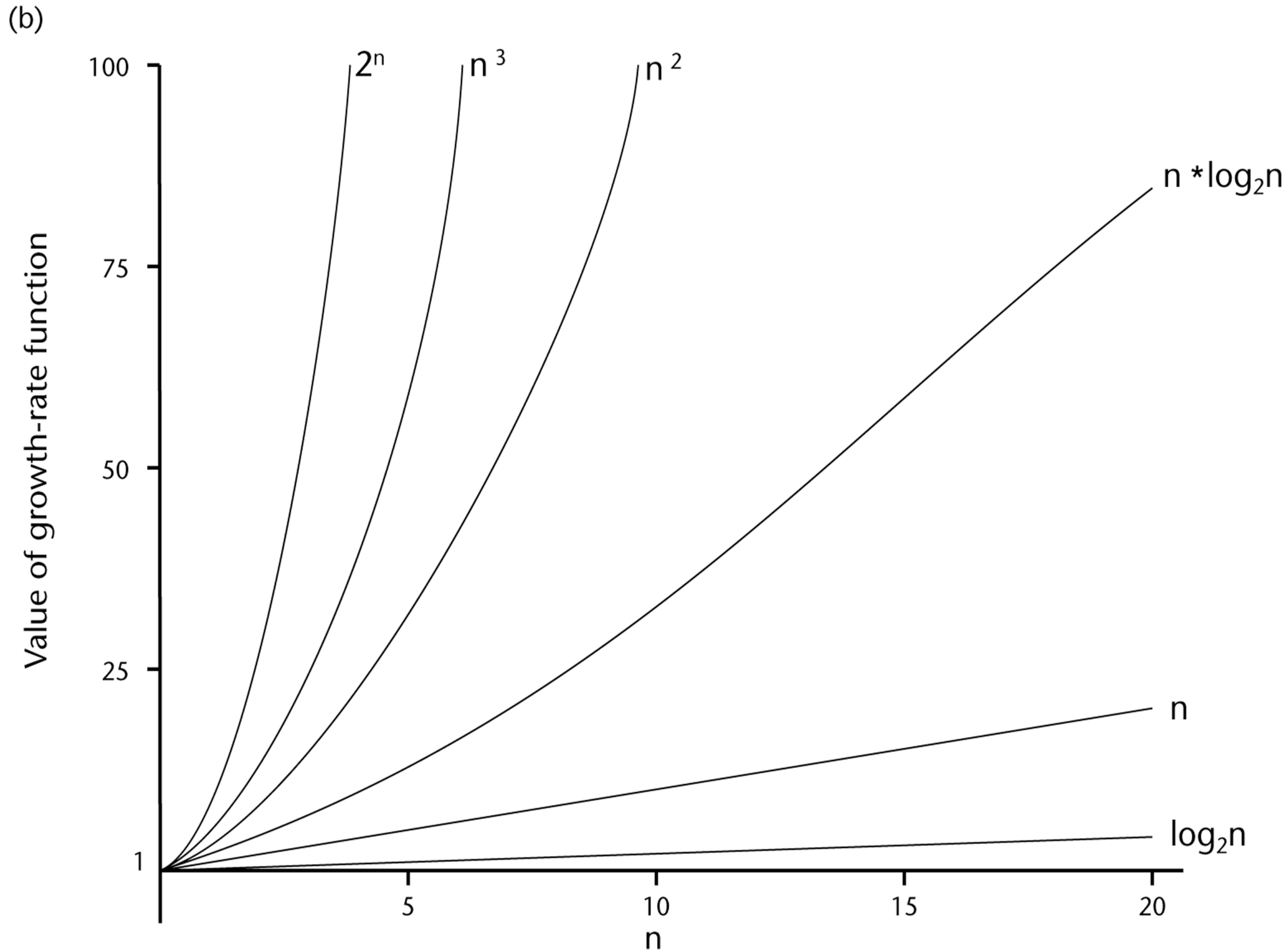


A Comparison of Growth-Rate Functions

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A Comparison of Growth-Rate Functions (cont.)



Growth-Rate Functions

O(1) Time requirement is **constant**, and it is independent of the problem's size.

$O(\log_2 n)$ Time requirement for a **logarithmic** algorithm increases slowly as the problem size increases.

O(n) Time requirement for a **linear** algorithm increases directly with the size of the problem.

$O(n*\log_2 n)$ Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

$O(n^2)$ Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

$O(n^3)$ Time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

$O(2^n)$ As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

Growth-Rate Functions

- If an algorithm takes 1 second to run with the problem size 8, what is the time requirement (approximately) for that algorithm with the problem size 16?
- If its order is:

$O(1)$ ➔ $T(n) = 1$ second

$O(\log_2 n)$ ➔ $T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds

$O(n)$ ➔ $T(n) = (1*16) / 8 = 2$ seconds

$O(n*\log_2 n)$ ➔ $T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds

$O(n^2)$ ➔ $T(n) = (1*16^2) / 8^2 = 4$ seconds

$O(n^3)$ ➔ $T(n) = (1*16^3) / 8^3 = 8$ seconds

$O(2^n)$ ➔ $T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds

Properties of Growth-Rate Functions

1. *We can ignore low-order terms in an algorithm's growth-rate function.*
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

2. *We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.*
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

3. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

Some Mathematical Facts

- Some mathematical equalities are:

$$\sum_{i=1}^{n} i = 1 + 2 + ... + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + ... + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + ... + 2^{n-1} = 2^n - 1$$

Growth-Rate Functions – Example1

Code	# of operations	Times
`i = 1;`	1	1
`sum = 0;`	1	1
`while (i <= n) {`	1	n+1
`    i = i + 1;`	2	n
`    sum = sum + i;`	2	n
`}`		

$$T(n) = 1 + 1 + (n+1)*1 + n*2 + n*2$$
$$= (1+2+2)*n + (1+1+1)$$
$$= 5*n + 3$$

➔ So, the growth-rate function for this algorithm is **O(n)**

Growth-Rate Functions – Example2

Code	# of operations	Times
`i=1;`	1	1
`sum = 0;`	1	1
`while (i <= n) {`	1	n+1
`    j=1;`	1	n
`    while (j <= n) {`	1	n*(n+1)
`        sum = sum + i;`	2	n*n
`        j = j + 1;`	2	n*n
`    }`		
`    i = i +1;`	2	n
`}`		

$$
\begin{aligned}
T(n) &= 1 + 1 + (n+1)*1 + n*1 + n*(n+1)*1 + n*n*1 + n*n*2 + n*2 \\
&= (1+1+2)*n^2 + (1+1+1+2)*n + (1+1+1) \\
&= 4*n^2 + 5*n + 3
\end{aligned}
$$

➔ So, the growth-rate function for this algorithm is **$O(n^2)$**

Growth-Rate Functions – Example3

	# of operations	**Times**
`for (i=1; i<=n; i++)`	1,1,1	1,n+1,n
`  for (j=1; j<=i; j++)`	1,1,1	n, $\sum_{i=1}^{n}(i+1)$, $\sum_{i=1}^{n} i$
`    for (k=1; k<=j; k++)`	1,1,1	$\sum_{i=1}^{n} i$, $\sum_{i=1}^{n}\sum_{j=1}^{i}(j+1)$, $\sum_{i=1}^{n}\sum_{j=1}^{i} j$
`      x=x+1;`	2	$\sum_{i=1}^{n}\sum_{j=1}^{i} j$

$$T(n) = 1 + 1*(n+1) + 1*n + 1*n + 1*\left(\sum_{i=1}^{n}(i+1)\right) + 1*\left(\sum_{i=1}^{n} i\right) + 1*\left(\sum_{i=1}^{n} i\right) +$$

$$1*\left(\sum_{i=1}^{n}\sum_{j=1}^{i}(j+1)\right) + 1*\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right) + 2*\left(\sum_{i=1}^{n}\sum_{j=1}^{i} j\right)$$

$$= a*n^3 + b*n^2 + c*n + d$$

➔ So, the growth-rate function for this algorithm is **$O(n^3)$**

Growth-Rate Functions – Example 4

Running time analysis of a function that computes the average of the elements of an array.

```
float computeAvg(float A[ ], int n){
  float avg, total=0;
  int k;
  for (k = 0; k < n; k++)
    total = total + A[k];
  avg = total / n;
  return(avg);
}
```

Growth-Rate Functions – Example 5

Running time analysis of findMin( ) function which finds and returns the minimum element of an array having n element.

```
float findMin(float A[ ], int n){
  float min;
  int k;
  min = A[0];
  for (k = 1; k < n; k++){
    if (A[k] < min){
      min = A[k];
    }
  }
  return min;
}
```

Growth-Rate Functions – Example 6

Running time analysis of MatrixSum( ) function which adds 2 n*m matrices A and B into matrix C.

```
void matrixSum(int A[ ][100], int B[ ][100], int C[ ][100], int n, int
    m){
  int i, j;
  for (i = 0; i < n; i++){
    for (j = 0; j < m; j++){
      C[i][j] = A[i][j] + B[i][j];
    }
  }
}
```

Growth-Rate Functions – Recursive Algorithms

```
void hanoi(int n, char source, char dest, char spare) {   #of op.
  if (n > 0) {                                                 1
    hanoi(n-1, source, spare, dest);                           c1
    printf("Move top disk from pole %c                         1
            to pole %c\n ", source, dest);
    hanoi(n-1, spare, dest, source);                           c2
} }
```

- The time-complexity function T(n) of a recursive algorithm is defined in terms of itself, and this is known as **recurrence equation** for T(n).
- To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.

Growth-Rate Functions – Hanoi Towers

- What is the cost of `hanoi(n,'A','B','C')`?

when n=0

$T(0) = 1$

when n>0

$$T(n) = 1 + c1 + T(n-1) + 1 + c2 + T(n-1)$$
$$= 2*T(n-1) + (1+c1+1+c2)$$
$$= \mathbf{2*T(n-1) + c}$$

🡐 recurrence equation for the growth-rate function of hanoi-towers algorithm

- Now, we have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

Growth-Rate Functions – Hanoi Towers (cont.)

- There are many methods to solve recurrence equations, but we will use a simple method known as *repeated substitutions*.

$T(n) = 2*T(n-1) + c$

$= 2 * (2*T(n-2)+c) + c$

$= 2 * (2* (2*T(n-3)+c) + c) + c$

$= 2^3 * T(n-3) + (2^2+2^1+2^0)*c$ (assuming n>2)

when substitution repeated i-1th times

$= 2^i * T(n-i) + (2^{i-1}+ \ldots +2^1+2^0)*c$

when i=n

$= 2^n * T(0) + (2^{n-1}+ \ldots +2^1+2^0)*c$

$= 2^n * c1 + (\sum_{i=0}^{n-1} 2^i)*c$

$= 2^n * c1 + (2^n-1)*c = 2^n*(c1+c) - c$ ➔ So, the growth rate function is $\mathbf{O(2^n)}$

What to Analyze

- An algorithm can require different times to solve different problems of the same size.
  - Eg. Searching an item in a list of n elements using sequential search. ➔ Cost: 1,2,...,n
- ***Worst-Case Analysis*** –The maximum amount of time that an algorithm require to solve a problem of size n.
  - This gives an upper bound for the time complexity of an algorithm.
  - Normally, we try to find worst-case behavior of an algorithm.
- ***Best-Case Analysis*** –The minimum amount of time that an algorithm require to solve a problem of size n.
  - The best case behavior of an algorithm is NOT so useful.
- ***Average-Case Analysis*** –The average amount of time that an algorithm require to solve a problem of size n.
  - Sometimes, it is difficult to find the average-case behavior of an algorithm.
  - We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.
  - ***Worst-case analysis is more common than average-case analysis.***

Memory Requirement of a Computer Program

- It is the amount of memory that is needed to run the program.
- It involves three components:

1. **Memory requirement of the source code**: equal to the size of the .exe file of the source code. It can be computed after the source code is compiled.
2. **Memory requirement of the data:** is the amount of memory to store the variables, data structures, and the arrays of the source code. *We can compute this value.*
3. **Memory requirement of the program stack:** If the source code does not include a recursive function, we can ignore this memory requirement.

→ So, do not use recursion!

Example

- Compute the memory requirement of data used in the below source code:

main(){	**# of bytes needed**
int A[20] = {7, 62, 3, 21, …};	20 * 4
int k, sum;	4 + 4
float avg;	4

```
main(){
  int A[20] = {7, 62, 3, 21, …};
  int k, sum;
  float avg;
  for (k = 0; k < 20; k++)
    sum += A[k];
  avg = sum / 20.0;
  printf("Average of the 20 elements=%f\n", avg);
}
```

→ Total bytes for the data = 80 + 8 + 4 = 92 bytes.

What is Important?

- If the problem size is always small, we can probably ignore the algorithm's efficiency.
  - In this case, we should choose the simplest algorithm.
- We have to weigh the trade-offs between an algorithm's time requirement and its memory requirements.
- We have to compare algorithms for both style and efficiency.
  - The analysis should focus on gross differences in efficiency and not reward coding tricks that save small amount of time.
  - That is, there is no need for coding tricks if the gain is not too much.
  - Easily understandable program is also important.
- Order-of-magnitude analysis focuses on large problems.