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.
- An algorithm must be correct. It should correctly solve the problem.
  - e.g. For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Algorithmic Performance

There are *two aspects* of algorithmic performance:

- Time
    - Instructions take time.
    - How fast does the algorithm perform?
    - What affects its runtime?
- Space
    - Data structures take space
    - What kind of data structures can be used?
    - How does choice of data structure affect the runtime?

➢ We will focus on time:

- How to estimate the time required for an algorithm
- How to reduce the time required

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?

  Naïve Approach: implement these algorithms in a programming language (C++), and run them to compare their time requirements. 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 that may cloud the issue of which algorithm is inherently more efficient.
  - *What computer should we use?*
    - We should compare the efficiency of the algorithms independently of a particular computer.
  - *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 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.

The Execution Time of Algorithms

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

  ➔ Each operation takes a certain 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;
```

Cost: c_1

Cost: c_2

➔ Total Cost = $c_1 + c_2$

The Execution Time of Algorithms (cont.)

Example: Simple If-Statement

	Cost	Times
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = 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

```
                                     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

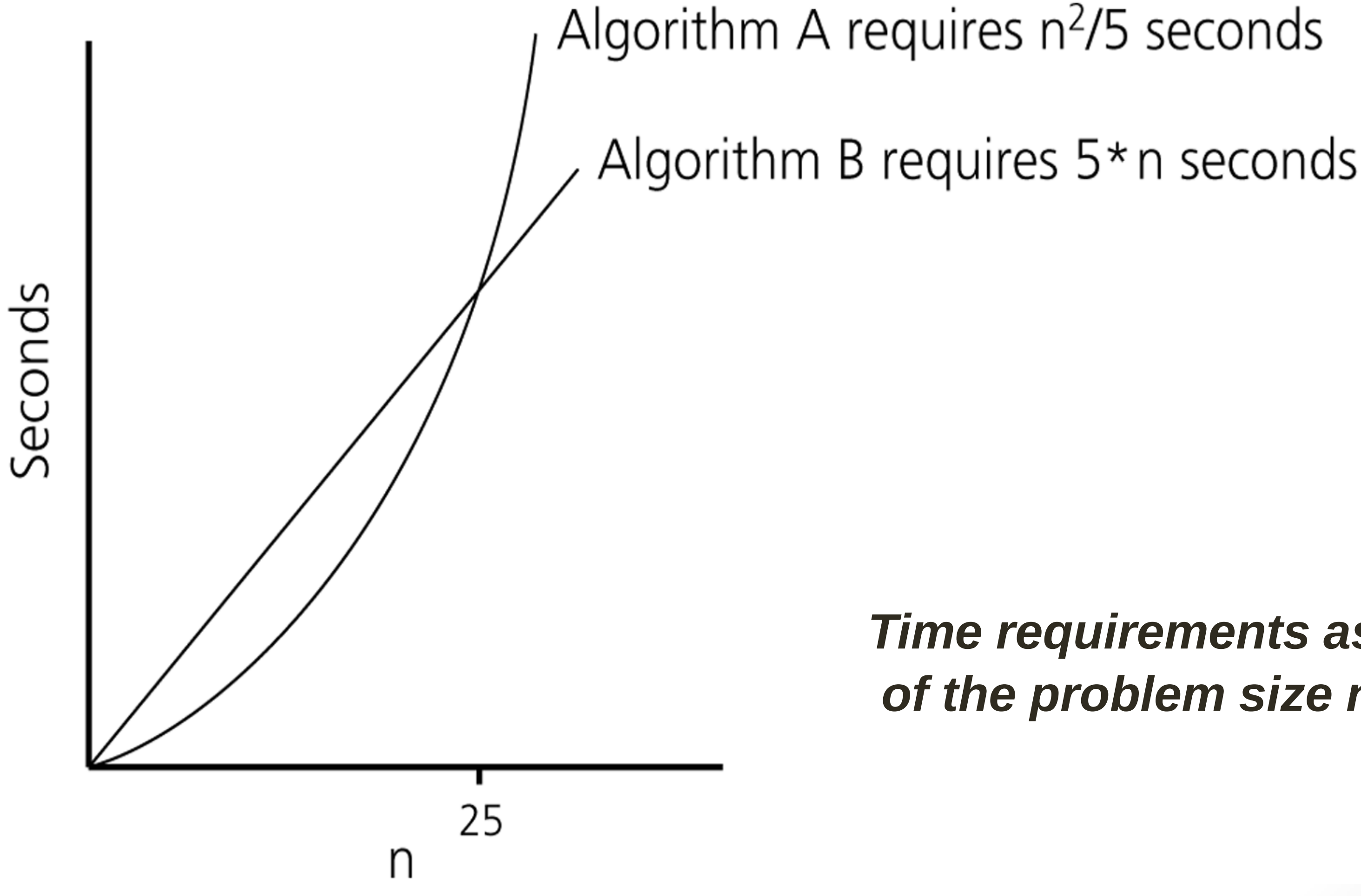
General Rules for Estimation

- **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 sized 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 **$5*n^2$** time units to solve a problem of size n.
  - Algorithm B requires **$7*n$** 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 **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

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
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 6.1

Running times for small inputs

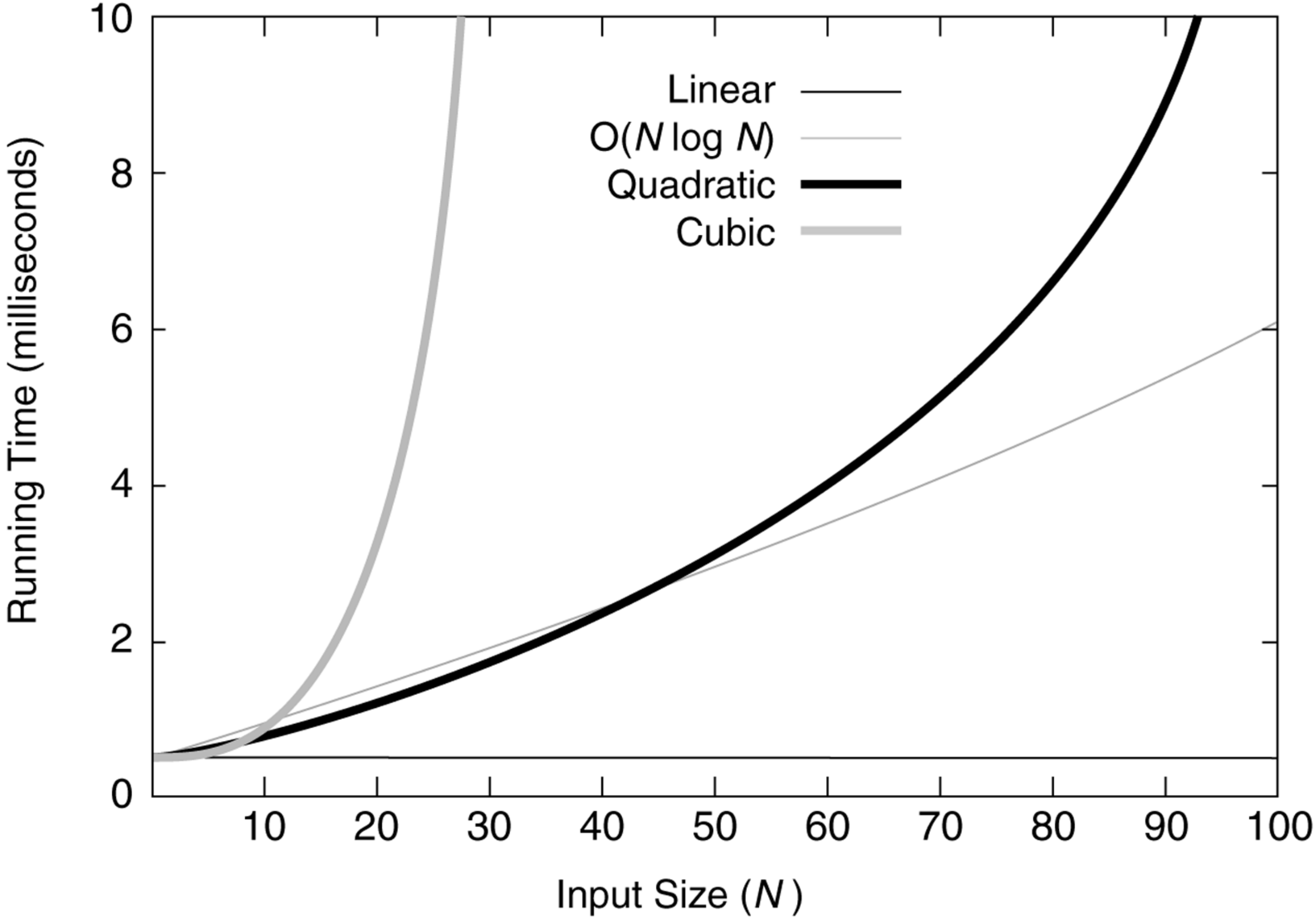
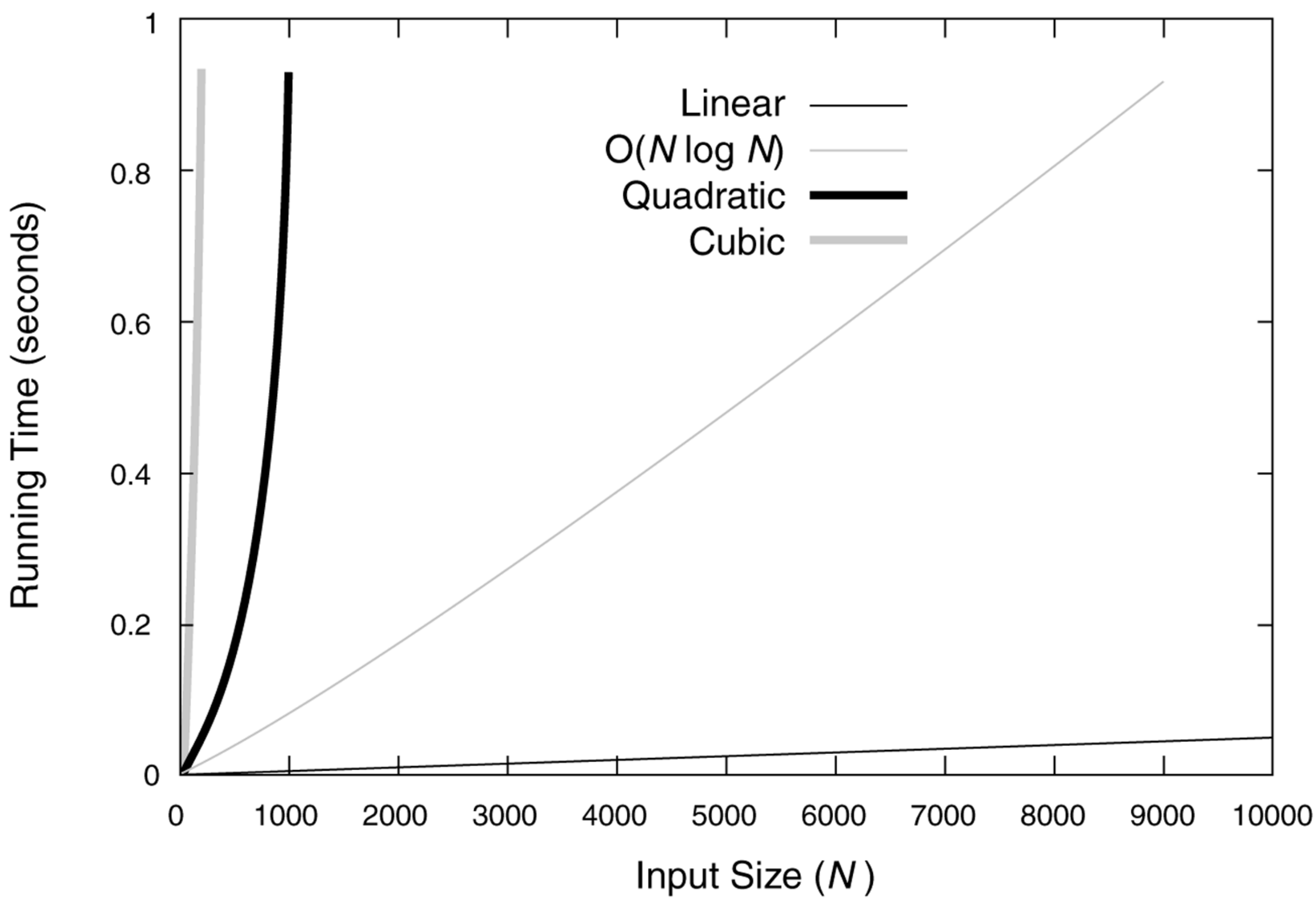


Figure 6.2
Running times for moderate inputs



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 k and n_0 exist such that A requires no more than k*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 k and n can satisfy this definition.

Order of an Algorithm

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

  $k*n^2 > n^2 – 3*n+10$ for all $n \geq n_0$.

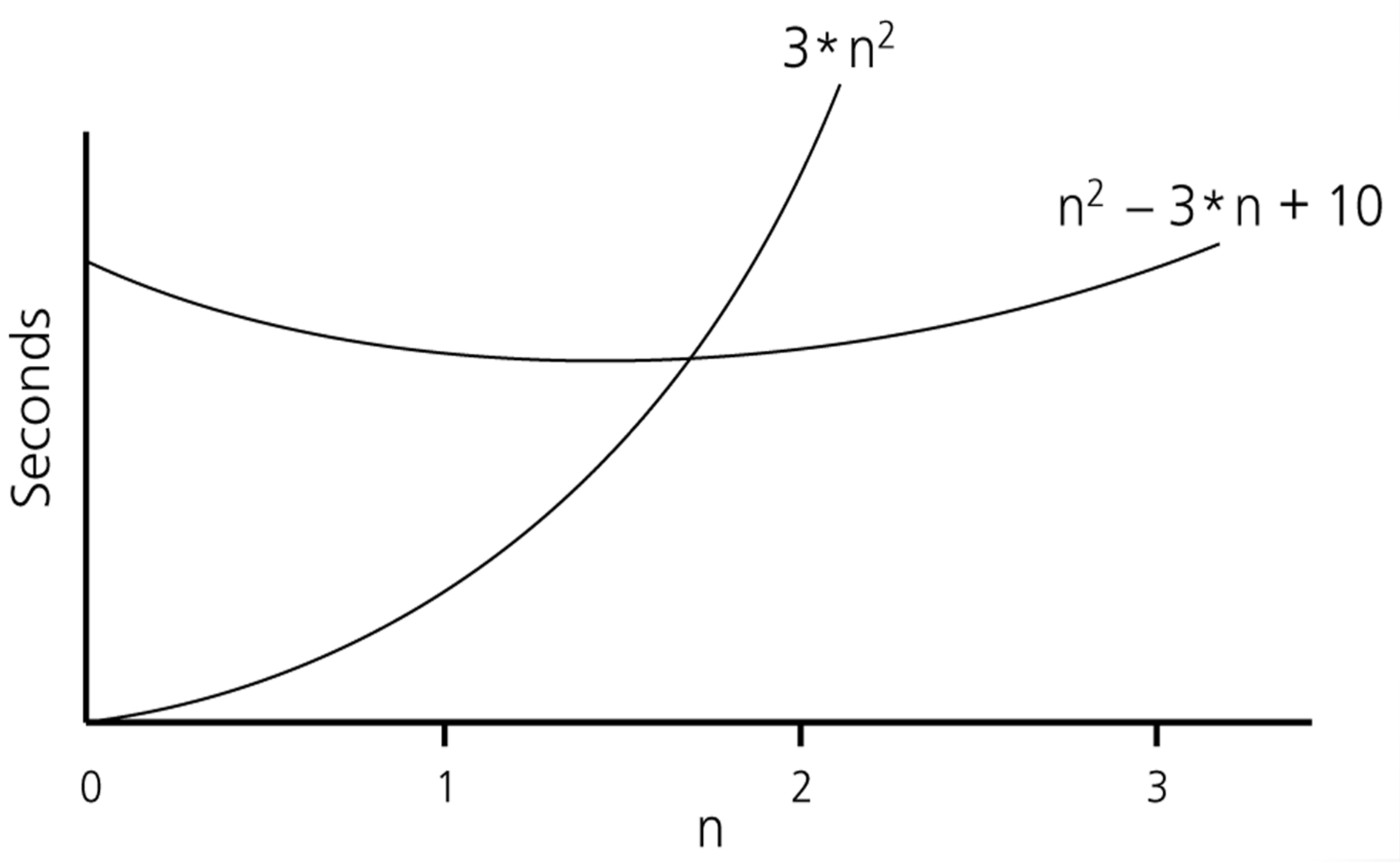
  the algorithm is order n^2 (In fact, k is 3 and n_0 is 2)

  $3*n^2 > n^2 – 3*n+10$ for all $n \geq 2$.

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

  So it is **O(n²)**

Order of an Algorithm (cont.)

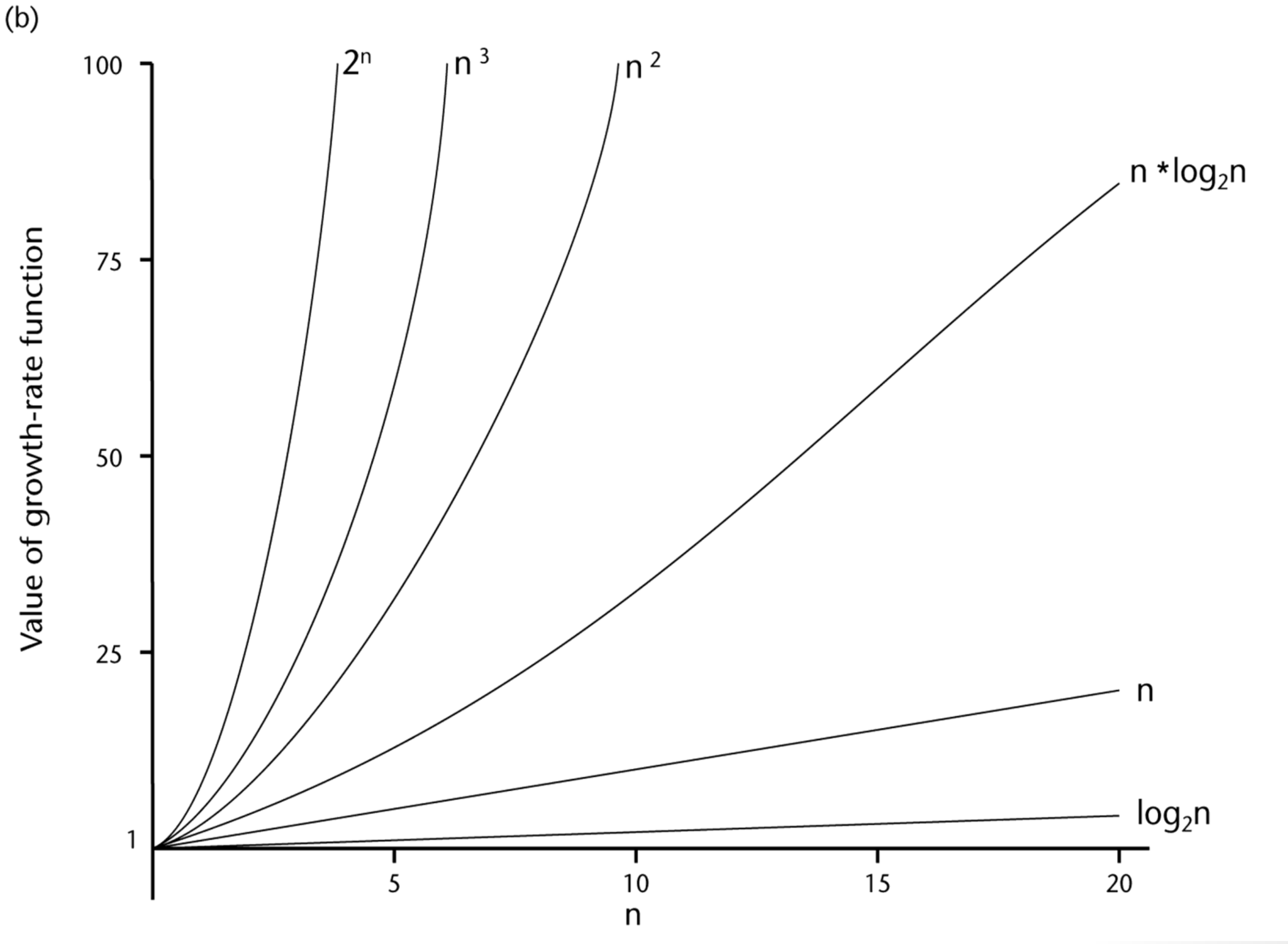


A Comparison of Growth-Rate Functions

(a)

Function	n					
	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 F



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 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 c**ubic** 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:

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

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

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

Growth-Rate Functions – Example1

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
}
```

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

= (c3+c4+c5)*n + (c1+c2+c3)

Growth-Rate Functions – Example2

	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
`}`		

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

= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)

= a*n^2 + b*n + c

Growth-Rate Functions – Example3

	Cost	**Times**
`for (i=1; i<=n; i++)`	c1	n+1
`  for (j=1; j<=i; j++)`	c2	$\sum_{j=1}^{n}(j+1)$
`    for (k=1; k<=j; k++)`	c3	$\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)$
`      x=x+1;`	c4	$\sum_{j=1}^{n}\sum_{k=1}^{j}k$

T(n) $= c1*(n+1) + c2*\left(\sum_{j=1}^{n}(j+1)\right) + c3*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)\right) + c4*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}k\right)$

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

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

Growth-Rate Functions – Recursive Algorithms

```
void hanoi(int n, char source, char dest, char spare) {
  Cost
   if (n > 0) {                                                    c1
    hanoi(n-1, source, spare, dest);                               c2
      cout << "Move top disk from pole " << source                 c3
          << " to pole " << dest << endl;
    hanoi(n-1, spare, dest, source);                               c4
} }
```

- 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) = c1

when n>0

T(n) = c1 + c2 + T(n-1) + c3 + c4 + T(n-1)

= 2*T(n-1) + (c1+c2+c3+c4)

= **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

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

when i=n

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

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

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

Properties of Big-O Notation

PROPERTIES OF BIG-O

- Big-O notation has some helpful properties that can be used when estimating the efficiency algorithms.

PROPERTIES OF BIG-O

- 1. Transitivity
  - If f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)).
  - This can be rephrased as O(O(g(n))) is O(g(n)).

PROPERTIES OF BIG-O

- 2. If $f(n)$ is $O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.

PROPERTIES OF BIG-O

- 3. The function an^k is $O(n^k)$.
- 4. The function n^k is $O(n^{k+j})$ for any positive j.
- 5. If $f(n) = cg(n)$, then $f(n)$ is $O(g(n))$.

PROPERTIES OF BIG-O

- 6. The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and b != 1
  - This correspondence holds between logarithmic functions.
  - It states that regardless of their bases, logarithmic functions are big-O of each other; that is, all these functions have the same rate of growth.

PROPERTIES OF BIG-O

- 7. $\log_a n$ is $O(\lg n)$ for any positive a != 1, where $\lg n = \log_2 n$.

Ω and Θ Notations

Ω NOTATIONS

- Big-O notation refers to the upper bounds of functions.

- There is a symmetrical definition for a lower bound in the definition of Big-Ω.

Ω NOTATIONS

- Definition:
  - The function f(n) is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) >= cg(n)$ for all $n >= N$.

Ω NOTATIONS

- This definition reads:
  - f is Ω (big-omega) of g if there is a positive number c such that f is at least equal to cg for almost all ns.
  - In other words, cg(n) is a lower bound on the size of f(n), or, in the long run, f grows at least at the rate of g.

Ω NOTATIONS

- The only difference between this definition and the definition of big-O notation is the direction of the inequality;
- One definition can be turned into the other by replacing “>=“ with “<=“.
- There is an interconnection between these two notations expressed by the equivalence
  - f(n) is Ω(g(n)) iff g(n) is O(f(n))

Θ NOTATIONS

- This suggests a way of restricting the sets of possible lower and upper bounds.

- This restriction can be accomplished by the following definition of Θ notation.

Θ NOTATIONS

- Definition:
  - f(n) is Θ(g(n)) if there exist positive numbers c_1, c_2, and N such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for all n >= N.

Θ NOTATIONS

- This definition reads:
  - f has an order of magnitude g, f is on the order of g, or both functions grow at the same rate in the long run.
  - We see that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Ω AND Θ NOTATIONS

- When applying any of these notations (big-O, Ω and Θ), do not forget that they are approximations that hide some detail that in many cases may be considered important.

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.***

What is Important?

- An array-based list `retrieve` operation is O(1), a linked-list-based list `retrieve` operation is O(n).
- But insert and delete operations are much easier on a linked-list-based list implementation.

- If the problem size is always small, we can probably ignore the algorithm’s efficiency.
  - In this case, we should choose the simplest algorithm.

What is Important? (cont.)

- 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.

Sequential Search

```
int sequentialSearch(const int a[], int item, int n){
  for (int i = 0; i < n && a[i]!= item; i++);
  if (i == n)
      return -1;
  return i;
}
```

Unsuccessful Search: → O(n)

Successful Search:

Best-Case: *item* is in the first location of the array → O(1)

Worst-Case: *item* is in the last location of the array → O(n)

Average-Case: The number of key comparisons 1, 2, ..., n

$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$

→ O(n)

Binary Search

```
int binarySearch(int a[], int size, int x) {
   int low =0;
   int high = size -1;
   int mid;           // mid will be the index of
                      // target when it's found.
   while (low <= high) {
     mid = (low + high)/2;
     if (a[mid] < x)
          low = mid + 1;
     else if (a[mid] > x)
          high  = mid - 1;
      else
          return mid;
   }
   return -1;
}
```

Binary Search – Analysis

- For an unsuccessful search:
  - The number of iterations in the loop is $\lfloor \log_2 n \rfloor + 1$

    ➔ $O(\log_2 n)$
- For a successful search:
  - ***Best-Case:*** The number of iterations is 1. ➔ $O(1)$
  - ***Worst-Case:*** The number of iterations is $\lfloor \log_2 n \rfloor + 1$ ➔ $O(\log_2 n)$
  - ***Average-Case:*** The avg. # of iterations $< \log_2 n$ ➔ $O(\log_2 n)$

0	1	2	3	4	5	6	7	← an array with size 8
3	2	3	1	3	2	3	4	← # of iterations

The average # of iterations = $21/8 < \log_2 8$

How much better is $O(\log_2 n)$?

n	$O(\log_2 n)$
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30