Computer program

- A computer program is an instance, or concrete representation, for an algorithm in some programming language.

Estimation Techniques

- Determine the major parameters that affect the problem.
- Derive an equation that relates the parameters to the problem.
- Select values for the parameters, and apply the equation to yield an estimated solution.

Asymptotic Performance

- **In this course, we care most about *asymptotic performance***
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does $O(n)$ running time mean? $O(n^2)$? $O(n \lg n)$?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our first task is to define this notation more formally and completely

Input Size

- **Time and space complexity**
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time

- **Number of primitive steps that are executed**
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- **We can be more exact if need be**

Algorithm Efficiency

- **There are often many approaches (algorithms) to solve a problem. How do we choose between them?**
- **At the heart of computer program design are two (sometimes conflicting) goals:**
  - To design an algorithm that is easy to understand, code and debug.
  - To design an algorithm that makes efficient use of the computer's resources.

Algorithm Efficiency

- **Goal (1) is the concern of Software Engineering.**
- **Goal (2) is the concern of data structures and algorithm analysis.**
- **When goal (2) is important, how do we measure an algorithm's cost?**
  - An algorithm has both time and space requirements, called its complexity,that we can measure.

Algorithm's complexity

- we measure an algorithm's time complexity—the time it takes to execute—or its space complexity—the memory it needs to execute.
- Typically we analyze these requirements separately.
- So a "best" algorithm might be the fastest one or the one that uses the least memory.

How to Measure Efficiency?

- **For most algorithms, running time depends on "size" of the input.**
  - This problem size is the number of items that an algorithm processes.

How to Measure Efficiency?

- **Running time is expressed as T(n) for some function T on input size n.**
  - you find a function of the problem size that behaves like the algorithm's actual time requirement.
  - The value of the function is said to be directly proportional to the time requirement. Such a function is called a growth-rate function. Typical growth-rate functions are algebraically simple.
  - It measures how an algorithm's time requirement grows as the problem size grows.

How to Measure Efficiency?

- The process of measuring the complexity of algorithms is called the **analysis of algorithms**.
- Empirical comparison (run programs).
- Asymptotic Algorithm Analysis.
- Critical resources:
- Factors affecting running time:
  - An algorithm's basic operation is the most significant contributor to its total time requirement.

Examples of Growth Rate

- **Example 1:**

```
int largest(int* array, int n)     // Find largest value
{  int currlarge = array[0];       // Store largest seen
   for (int i=1; i<n; i++)         // For each element
       if (array[i] > currlarge)   // If largest
           currlarge = array[i];
                                   // Remember it
   return currlarge;               // Return largest
}
```

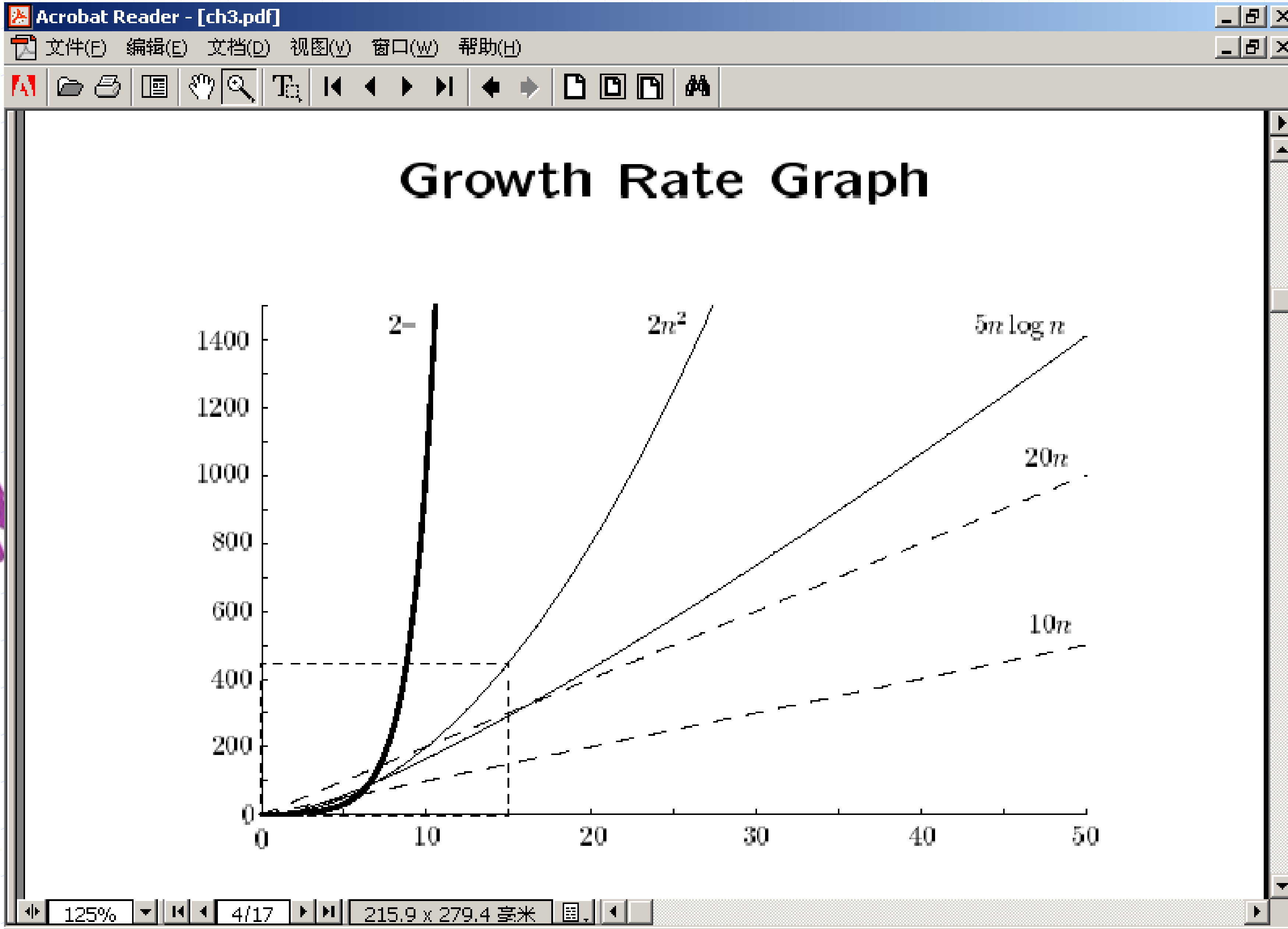
Examples of Growth Rate

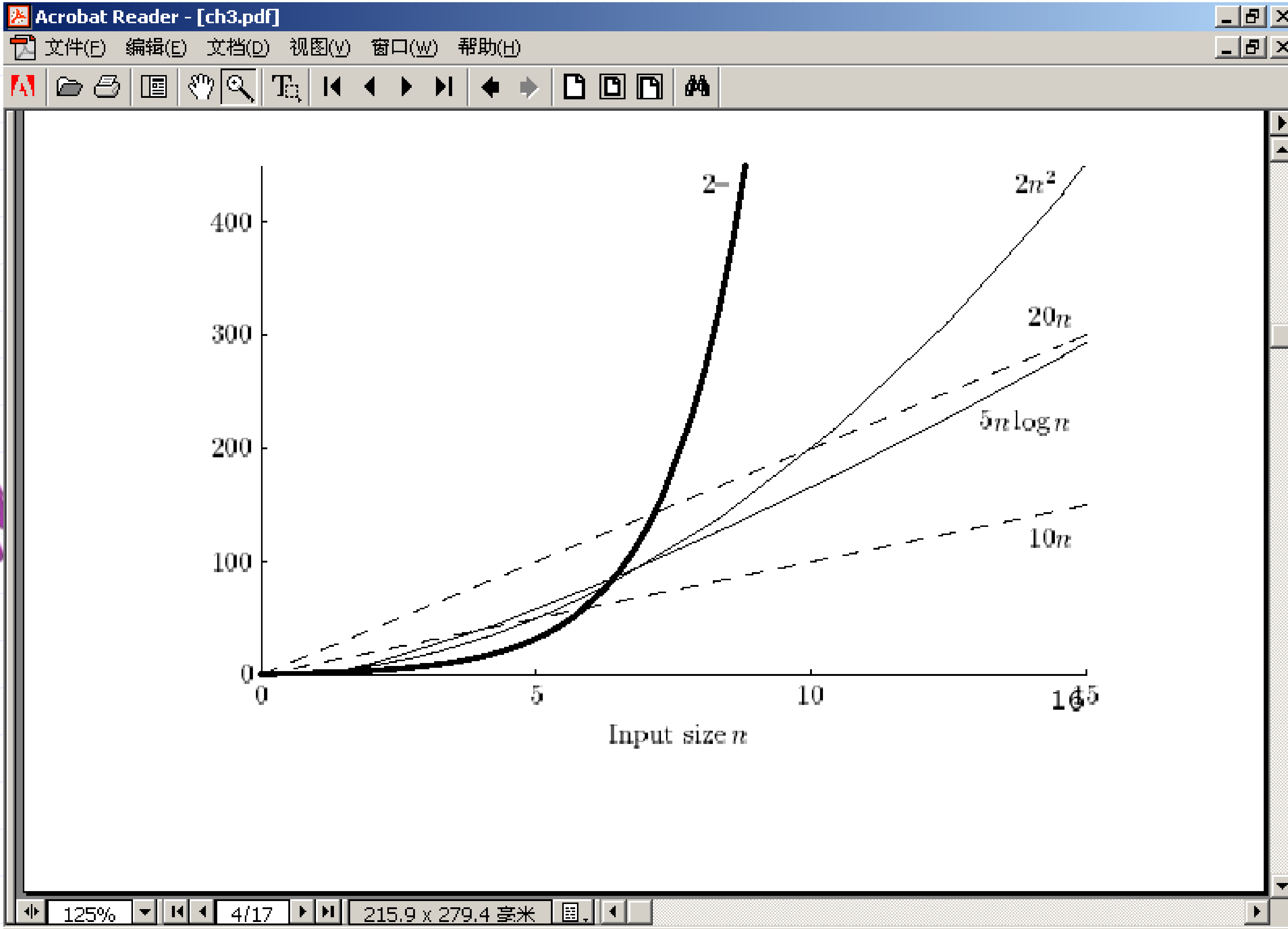
- Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum++;
```

Faster Computer or Algorithm?

- What happens when we buy a computer 10 times faster?


Acrobat Reader - [ch3.pdf]
文件(F) 编辑(E) 文档(D) 视图(V) 窗口(W) 帮助(H)
Growth Rate Graph
2^n
$2n^2$
$5n \log n$
$20n$
$10n$
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50
125%
4/17
215.9 x 279.4 毫米


Acrobat Reader - [ch3.pdf]
文件(F) 编辑(E) 文档(D) 视图(V) 窗口(W) 帮助(H)
2–
$2n^2$
$20n$
$5n \log n$
$10n$
400
300
200
100
0
0
5
10
15
Input size n
125%
4/17
215.9 x 279.4 毫米

Acrobat Reader - [ch3.pdf]

文件(F) 编辑(E) 文档(D) 视图(V) 窗口(W) 帮助(H)

$T(n)$	n	n'	Change	n'/n
$10n$	$1,000$	$10,000$	$n' = 10n$	10
$20n$	500	$5,000$	$n' = 10n$	10
$5n \log n$	250	$1,842$	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

150% 6/17 215.9 x 279.4 毫米

growth-rate functions

- **Typical growth-rate functions evaluated at increasing values of *n***

$$1 < \log(\log n) < \log n < \log^2 n < n < n \log n < n^2 < n^3 < 2^n < n!$$

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

- **The effect of doubling the problem size on an algorithm's time requirement**

Growth-Rate Function for Size *n Problems*	**Growth-Rate Function for Size 2*n Problems***	**Effect on Time Requirement**
1	1	None
$\log n$	$1+\log n$	Negligible
n	$2n$	Doubles
$n\log n$	$2n\log n+2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

Best, Worst and Average Cases

- **Not all inputs of a given size take the same time.**
- **Sequential search for K in an array of n integers:**
  - Begin at first element in array and look at each element in turn until K is found.

Analysis

- **Worst case**
  - Provides an upper bound on running time
  - An absolute guarantee
- **Average case**
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs
    - Real-life inputs

Best, Worst and Average Cases

- Best Case:
- Worst Case:
- Average Case:
- While average time seems to be the fairest measure, it may be difficult to determine.
- When is worst case time important?

Asymptotic Analysis: Big-oh

- Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the set $O(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \leqq cf(n)$ for all $n > n_0$.

Asymptotic Analysis: Big-oh

- Usage: The algorithm is in $O(n^2)$ in [best, average, worst] case.
- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in less than $cf(n)$ steps [in best, average or worst case].

Asymptotic Analysis: Big-oh

- Upper Bound.
- Example: if $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.
- Wish tightest upper bound:
- While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

Simplifying Rules:

- **If f(n) is in O(g(n)) and g(n) is in O(h(n)),then f(n) is in O(h(n)).**
  - In simple terms, *f(n) is O(g(n)) means that* $c \times g(n)$ *provides an upper bound on f(n)'s growth* rate when *n is large enough. For all data sets of a sufficient size, the algorithm will always require* fewer than $c \times g(n)$ *basic operations.*
- **If f(n) is in O(kg(n)) for any constant k >0, then f(n) is in O(g(n)).**

Simplifying Rules:

- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

Identities

- **The following identities hold for Big Oh notation:**
  - $O(k\, g(n)) = O(g(n))$ for a constant k
  - $O(g1(n)) + O(g2(n)) = O(g1(n) + g2(n))$
  - $O(g1(n)) \times O(g2(n)) = O(g1(n) \times g2(n))$
  - $O(g1(n) + g2(n) + \ldots + gm(n)) = O(\max(g1(n), g2(n), \ldots, gm(n)))$
  - $O(\max(g1(n), g2(n), \ldots, gm(n))) = \max(O(g1(n)), O(g2(n)), \ldots, O(gm(n)))$

Running Time of a Program

- Example 1:

  a = b;

  This assignment takes constant time, so it is(1).

- Example 2:

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

Running Time of a Program

- Example 3:

```
sum = 0;
for (j=1; j<=n; j++) // First for loop
    for (i=1; i<=j; i++) // is a double loop
        sum++;
for (k=0; k<n; k++) // Second for loop
    A[k] = k;
```

More Examples

- Example 4.

```
sum1 = 0;
for (i=1; i<=n; i++) // First double loop
    for (j=1; j<=n; j++) // do n times
        sum1++;
sum2 = 0;
for (i=1; i<=n; i++) // Second double loop
    for (j=1; j<=i; j++) // do i times
        sum2++;
```

Other Control Statements

- while loop: analyze like a for loop.
- if statement: Take greater complexity of then/else clauses.
- switch statement: Take complexity of most expensive case.
- Subroutine call: Complexity of the subroutine.

The complexities of program constructs

Construct	Time Complexity
Consecutive program segments *S1, S2, . . . , Sk whose* growth-rate functions are *g1, . . . , gk, respectively*	max(O(*g1), O(g2), . . . , O(gk)*)
An if statement that chooses between program segments *S1 and S2 whose growth-rate functions are g1 and g2,* respectively	O(*condition) + max(O(g1), O(g2))*
A loop that iterates *m times and has a body whose growthrate* function is *g*	$m \times O(g(n))$

Analyzing Problems

- Upper bound: Upper bound of best known algorithm.
- Lower bound: Lower bound for every possible algorithm.

Other notations

- **Big Oh.** *f(n) is of order at most g(n)—that is, f(n) is O(g(n))—if positive constants c* **and** *N exist such that f(n) ≤ c × g(n) for all n ≥ N. That is, c × g(n) is an upper bound* **on the time requirement** *f(n). In other words, f(n) is no larger than c × g(n).*
- *Thus, an* **analysis that uses Big Oh produces a maximum time requirement for an algorithm.**

Other notations

- Big Omega. *f(n) is of order at least g(n)—that is, f(n) is Ω(g(n))—if g(n) is O(f(n)). In* other words, *f(n) is Ω(g(n)) if positive constants c and N exist such that f(n) ≥ c × g(n)* for all *n ≥ N. The time requirement f(n) is not smaller than c × g(n), its lower bound.*
- Thus, a Big Omega analysis produces a minimum time requirement for an algorithm.

Other notations

- **Big Theta.** *f(n) is of order g(n)—that is, f(n) is Θ(g(n))—if f(n) is O(g(n)) and g(n) is* O(*f(n)). Alternatively, we could say that f(n) is O(g(n)) and f(n) is Ω(g(n)). The time* requirement *f(n) is the same as g(n). That is, c* × *g(n) is both a lower bound and an* upper bound on *f(n).*
- *A Big Theta analysis assures us that the time estimate is as* good as possible.

Space Bounds

- Space bounds can also be analyzed with asymptotic complexity analysis.
- Time: Algorithm
- Space: Data Structure

Space/Time Tradeoff Principle