

VE281

Data Structures and Algorithms

Asymptotic Algorithm Analysis

Learning Objective:

- Understand best, worst, and average cases
- Understand Big-Oh, Big-Omega, Big-Theta notations
- Know how to analyze time complexity of a program

Outline

- Asymptotic Analysis: Big-Oh
- Relatives of Big-Oh
- Analyzing Time Complexity of Programs

How to Measure Efficiency?

- Empirical comparison: run programs
 - Use the wall-clock time to measure the runtime
 - Empirical comparison could be tricky. It depends on
 - Compiler
 - Machine (CPU speed, memory, etc.)
 - CPU load
- Asymptotic Algorithm Analysis
 - For most algorithms, running time depends on the “size” of the input.
 - Running time is expressed as $T(n)$ for some function T on input size n .
Space: $S(n)$

Input Dependency: Example

- Summing an array of n elements

```
// REQUIRES: a is an array of size n
// EFFECTS: return the sum
int sum(int a[], unsigned int n) {
    int result = 0;
    for(unsigned int i = 0; i < n; i++)
        result += a[i];
    return result;
}
```

- The runtime is roughly cn , where c is some constant.
- With n fixed, any array has roughly the **same** runtime.

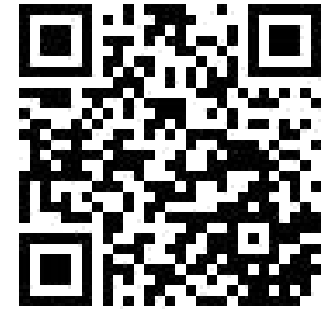
Best, Worst, Average Cases

- In the example of summing an array, all inputs of a given size take the same time to run.
- However, in some other cases, this is not true, i.e., not all inputs of a given size take the same time to run.
- Example: linear search

```
// REQUIRES: a is an array of size n
// EFFECTS: return the index of the element
// equals key. If no such element, return n.
int search(int a[], unsigned int n, int key) {
    for(unsigned int i = 0; i < n; i++)
        if(a[i] == key) return i;
    return n;
}
```



Which Statements Are True for Linear Search?



Select all the correct statements:

- **A.** The best case occurs when **key** is the first element in the array.
- **B.** In the worst case, we need to do **n** comparisons with **key**.
- **C.** The worst case in terms of the number of comparisons with **key** only occurs when **key** is not in the array.
- **D.** Suppose **key** is uniformly located in the array. Then, on average, the number of comparisons with **key** is **n/2**.

```
// REQUIRES: a is an array of size n
// EFFECTS: return the index of the element
// equals key. If no such element, return n.
int search(int a[], unsigned int n, int key) {
    for(unsigned int i = 0; i < n; i++)
        if(a[i] == key) return i;
    return n;
}
```

Best, Worst, Average Cases

- Best case: least number of steps required, corresponding to the ideal input
- Worst case: most number of steps required, corresponding to the most difficult input.
- Average case: average number of steps required, given any input.

need to consider the distribution

A Common Misunderstanding

“The best case for my algorithm is $n = 1$ because that is the fastest.”

- Wrong!
- Best case is a special input case of size n that is **cheapest** among all input cases of size n .

Which Case to Use?

- Average case or worst case is common.
- While average time appears to be the fairest measure, it may be difficult to determine.
 - Sometime, it requires domain knowledge, e.g., the distribution of inputs.
- Worst case is pessimistic, but it gives an upper bound.
 - Bonus: worst case usually easier to analyze.

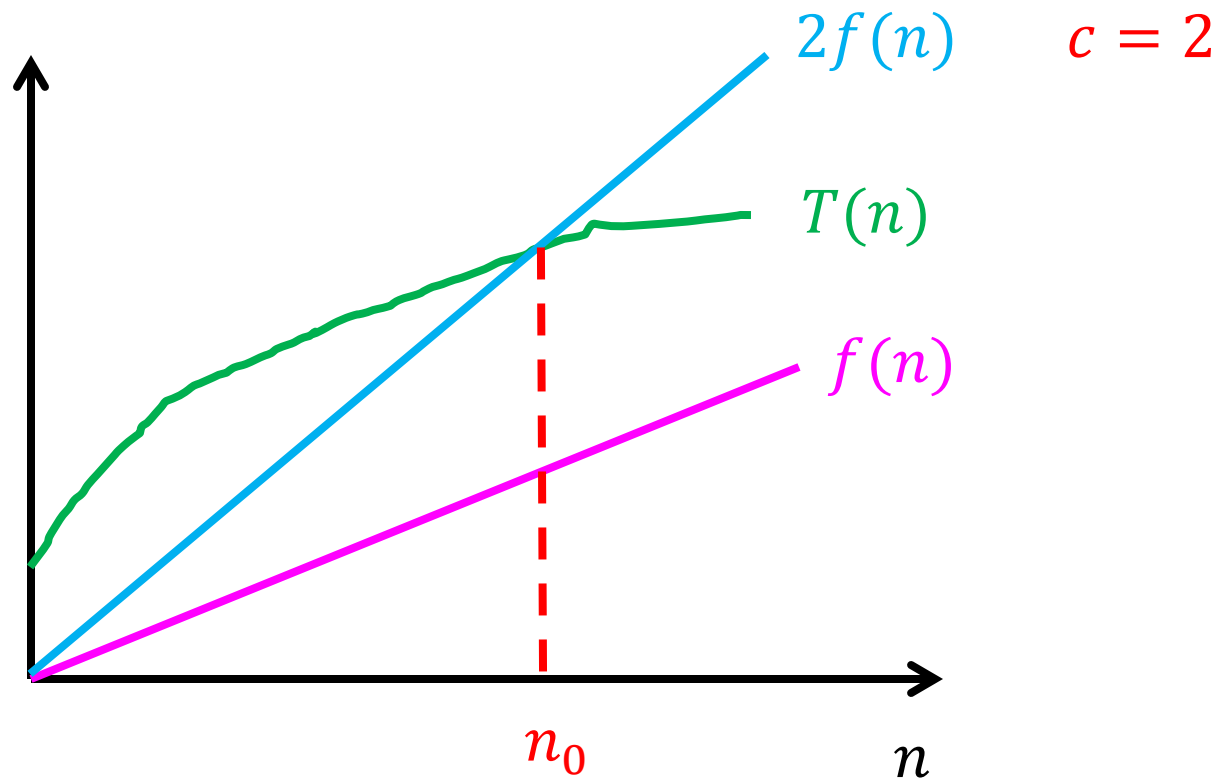
How to Analyze Complexity of Algorithm?

- Guiding Principle #1: Ignore constant factors.
 - Justification:
 1. Way easier.
 2. Constants depend on architecture, compiler, etc.
 3. Lose very little predictive power (as we will see).
- Guiding Principle #2: Focus on running time for large input size n .
 - Justification: only big problem are interesting!
 - Thus, we will compare the runtime of two algorithms when n is very large.
 - E.g., $1000 \log_2 n$ is “better” than $0.001n$.

Asymptotic Analysis: Big-Oh

- Definition: 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 $T(n) \leq cf(n)$ for all $n > n_0$.
- 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/worst case.

Graphic View of Big-Oh



Big-Oh Notation

- Strictly speaking, we say that $T(n)$ is **in** $O(f(n))$, i.e.,
 $T(n) \in O(f(n))$
- However, for convenience, people also write
 $T(n) = O(f(n))$

$$\begin{aligned} |IN| &= \Omega(S) \\ |IR| &= 2^{|IN|} = \Omega(S) \end{aligned}$$

Big-Oh Example

- Claim: If $T(n) = a_k n^k + \dots + a_1 n + a_0$, then
$$T(n) = O(n^k)$$
- Proof:
 - Need to pick constants c and n_0 so that for any $n > n_0$,
$$T(n) \leq c \cdot n^k.$$
 - Choose $n_0 = 1$ and $c = |a_k| + \dots + |a_1| + |a_0|$
 - Only need to show that for any $n > n_0$, $T(n) \leq cn^k$.

Big-Oh Example

- Claim: $2^{n+10} = O(2^n)$
- Proof:
 - Need to pick constants c and n_0 so that for any $n > n_0$,
$$2^{n+10} \leq c \cdot 2^n \quad (*)$$
 - We note $2^{n+10} = 1024 \cdot 2^n$.
 - So if we choose $c = 1024$ and $n_0 = 1$, then $(*)$ holds.

Big-Oh Notation

- Big-oh notation indicates an **upper bound**.
- Example: If $T(n) = 3n^2$ then $T(n)$ is in $O(n^2)$.
- Look for the **tightest** upper bound:
 - While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.

A Sufficient Condition of Big-Oh

If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$, then $f(n)$ is $O(g(n))$.

- With this theorem, we can easily prove that

$$T(n) = c_1 n^2 + c_2 n \text{ is } O(n^2)$$

- Proof: $\lim_{n \rightarrow \infty} \frac{c_1 n^2 + c_2 n}{n^2} = c_1 < \infty$

actually big
\theta of g(n)

Rules of Big-Oh

- **Rule 1:** If $f(n) = O(g(n))$, then $cf(n) = O(g(n))$.
 - Example: $3n^2 = O(n^2)$
- **Rule 2:** If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$
 - Example: $n^3 + 2n^2 = O(\max\{n^3, n^2\}) = O(n^3)$

Rules of Big-Oh

- **Rule 3:** If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
- **Rule 4:** If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

Common Functions and Their Growth Rates

constant: 1

logarithmic: $\log n$

refers to $\log_2 n$

square root: \sqrt{n}

linear: n

loglinear: $n \log n$

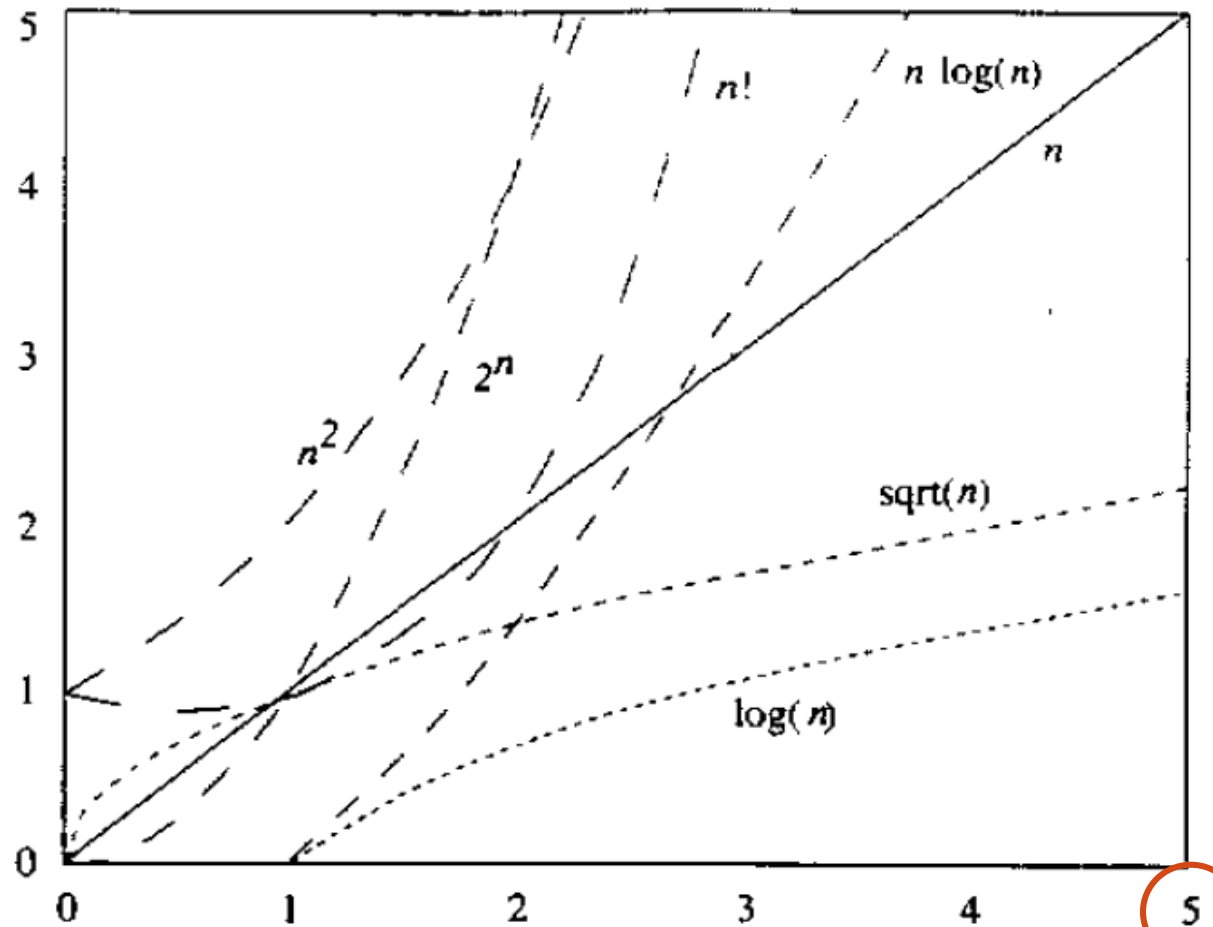
quadratic: n^2

cubic: n^3

general polynomial: n^k
 $k \geq 1$

exponential: $a^n, a > 1$

factorial: $n!$



Common Functions and Their Growth Rates

constant: 1

logarithmic: $\log n$

refers to $\log_2 n$

square root: \sqrt{n}

linear: n

loglinear: $n \log n$

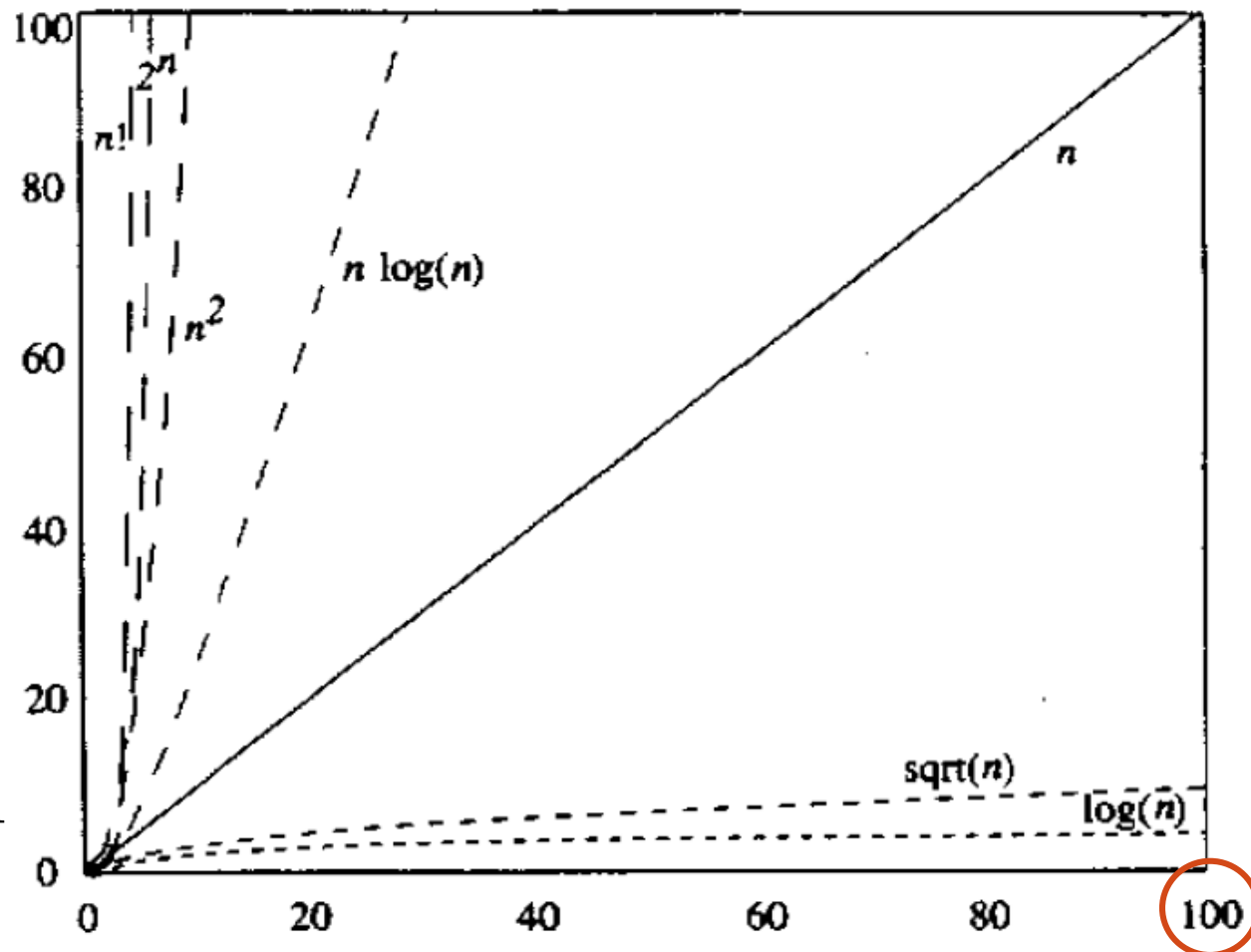
quadratic: n^2

cubic: n^3

general polynomial: n^k
 $k \geq 1$

exponential: $a^n, a > 1$

factorial: $n!$



A Few Results about Common Functions

- For a polynomial in n of the form

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$$

where $a_m > 0$, we have $f(n) = O(n^m)$.

- For every integer $k \geq 1$, $\log^k n = O(n)$.

not a tight bound

- For every integer $k \geq 1$, $n^k = O(2^n)$.



How Fast Is Your Code?

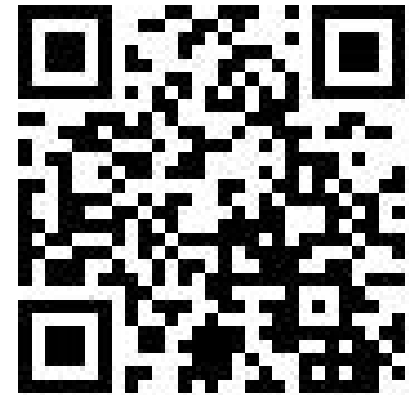
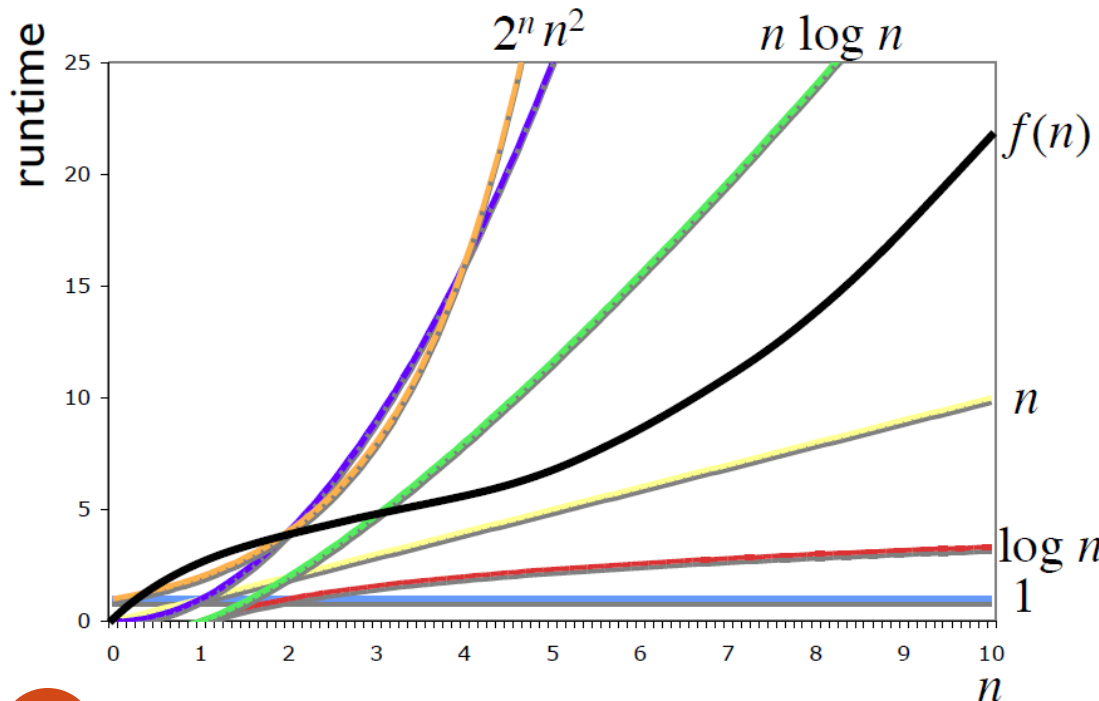
- Let $f(n)$ be the complexity of your code, how fast would you advertise it as? Choose one proper answer.

A. $\log n$

B. $n \log n$

C. n

D. n^2



$f(n) = O(g(n))$; You want to pick a $g(n)$ that is as close to $f(n)$ as possible.

What Is a “Fast” Algorithm?

fast algorithm \approx worst-case/average-case running
time grows slowly with input size

- Usually as close to linear ($O(n)$) as possible.

Outline

- Asymptotic Analysis: Big-Oh
- Relatives of Big-Oh
- Analyzing Time Complexity of Programs

Relative of Big-Oh: Big-Omega

- Definition: For $T(n)$ a non-negatively valued function, $T(n)$ is in the **set** $\Omega(g(n))$ if there **exist** two positive constants c and n_0 such that $T(n) \geq cg(n)$ for all $n > n_0$.
- Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always requires **more than** $cg(n)$ steps.
- Big-omega gives a **lower bound**.
- We usually want the greatest lower bound.

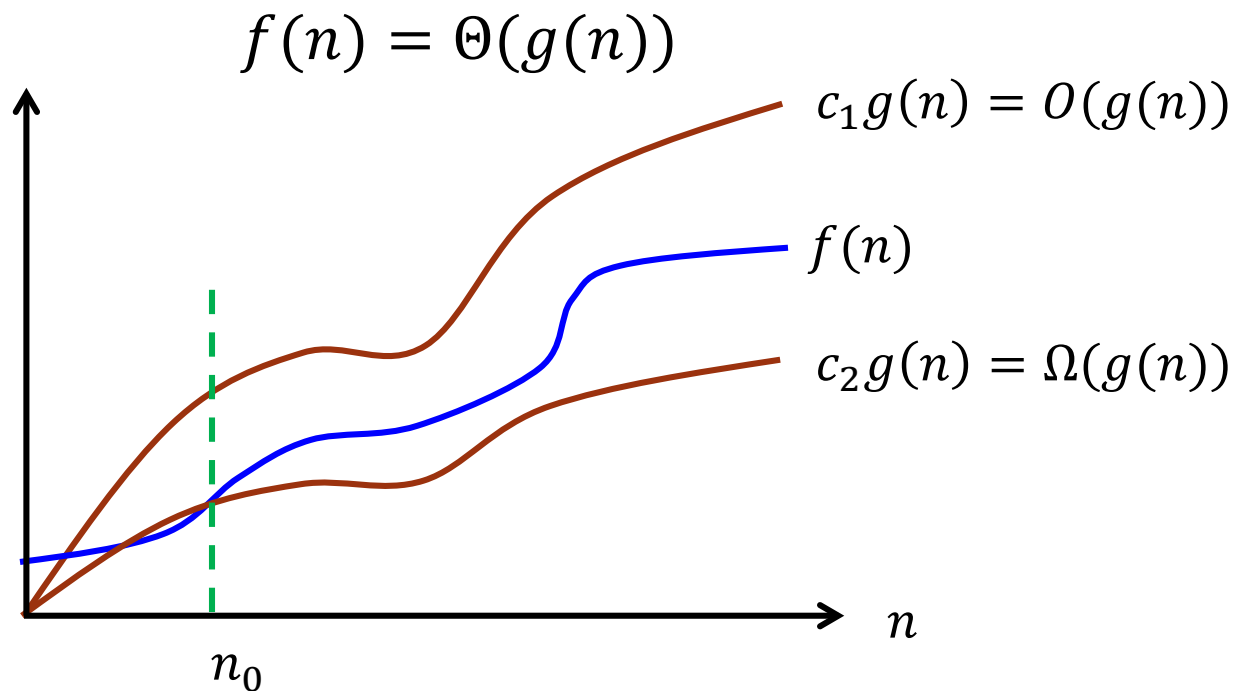
Big-Omega Example

- Consider $T(n) = c_1n^2 + c_2n$, where c_1 and c_2 are positive.
- What is the big-omega notation for $T(n)$?
- Solution:
 - $c_1n^2 + c_2n \geq c_1n^2$ for all $n > 1$.
 - $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.
 - Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

Theta Notation

- When big-oh and big-omega coincide, we indicate this by using big-theta (Θ) notation.
- Definition: $T(n)$ is said to be in the set $\Theta(g(n))$ if it is in $O(g(n))$ and it is in $\Omega(g(n))$.
 - In other words, there **exist** three positive constants c_1 , c_2 , and n_0 such that $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for all $n > n_0$.

Theta Notation



- Question: Does $f(n) = \Theta(g(n))$ indicate $g(n) = \Theta(f(n))$?

Outline

- Asymptotic Analysis: Big-Oh
- Relatives of Big-Oh
- Analyzing Time Complexity of Programs

Analyzing Time Complexity of Programs

- For atomic statement, such as assignment, its complexity is $\Theta(1)$.
- For branch statement, such as if-else statement and switch statement, its complexity is that of the most expensive Boolean expression plus that of the most expensive branch.

```
if(Boolean_Expression_1) {Statement_1}  
else if (Boolean_Expression_2) {Statement_2}  
...  
else if (Boolean_Expression_n) {Statement _n}  
else {Statement_For_All_Other_Possibilities}
```

Analyzing Time Complexity of Programs

- For subroutine call, its complexity is that of the subroutine.
- For loops, such as while and for loop, its complexity is related the number of operations required in the loop.

Time Complexity Example One

- What is the time complexity of the following code?

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

- The entire time complexity is $\Theta(n)$.

Time Complexity Example Two

- What is the time complexity of the following code?

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

- Note that the statements

```
j <= i;  
j++;  
sum++;
```

all occur (roughly) $1 + 2 + \dots + n = n(n + 1)/2$ times.

- The time complexity is $\Theta(n^2)$.

Time Complexity Example Three

- What is the time complexity of the following code?

```
sum = 0;  
for(i = 1; i <= n; i *= 2)  
    for(j = 1; j <= n; j++)  
        sum++;
```

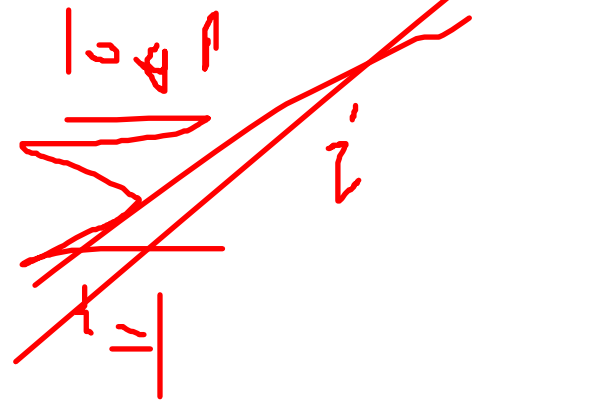
- The outer loop occurs $\log n$ times.
- The statements **sum++** / **j<=n** / **j++** occur $n \log n$ times.
- The time complexity is $\Theta(n \log n)$.



What Is the Time Complexity of the Following Code?

- Choose the correct answer.

```
sum = 0;  
for(i = 1; i <= n; i *= 2)  
    for(j = 1; j <= i; j++)  
        sum++;
```

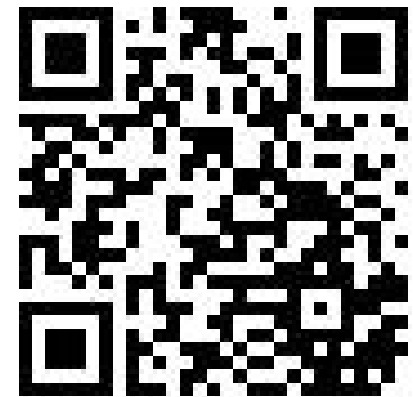


A. $\Theta(\log n)$

B. $\Theta(n \log n)$

C. $\Theta(n)$

D. $\Theta(n^2)$



Multiple Parameters

- Example: Compute the rank ordering for all C (i.e., 256) pixel values in a picture of P (i.e., 64×64) pixels.

```
for(i=0; i<C; i++)    // Initialize count
    count[i] = 0;
```

$\Theta(C)$

```
for(i=0; i<P; i++)    // Look at all pixels
    count[value[i]]++; // Increment count
```

$\Theta(P)$

```
sort(count);          // Sort pixel counts
```

$\Theta(C \log C)$

- The time complexity is $\Theta(P + C \log C)$.
- One general application is to analyze graph algorithm

Space/Time Trade-off Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.
- Example: factorial
 - Iterative method: Get “n!” using a for-loop.
 - This requires $\Theta(1)$ memory space and $\Theta(n)$ runtime.
 - Table lookup method: Pre-compute the factorials for $1, 2, \dots, N$ and store all the results in an array.
 - This requires $\Theta(n)$ memory space and $\Theta(1)$ runtime (fetching from an array).