

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

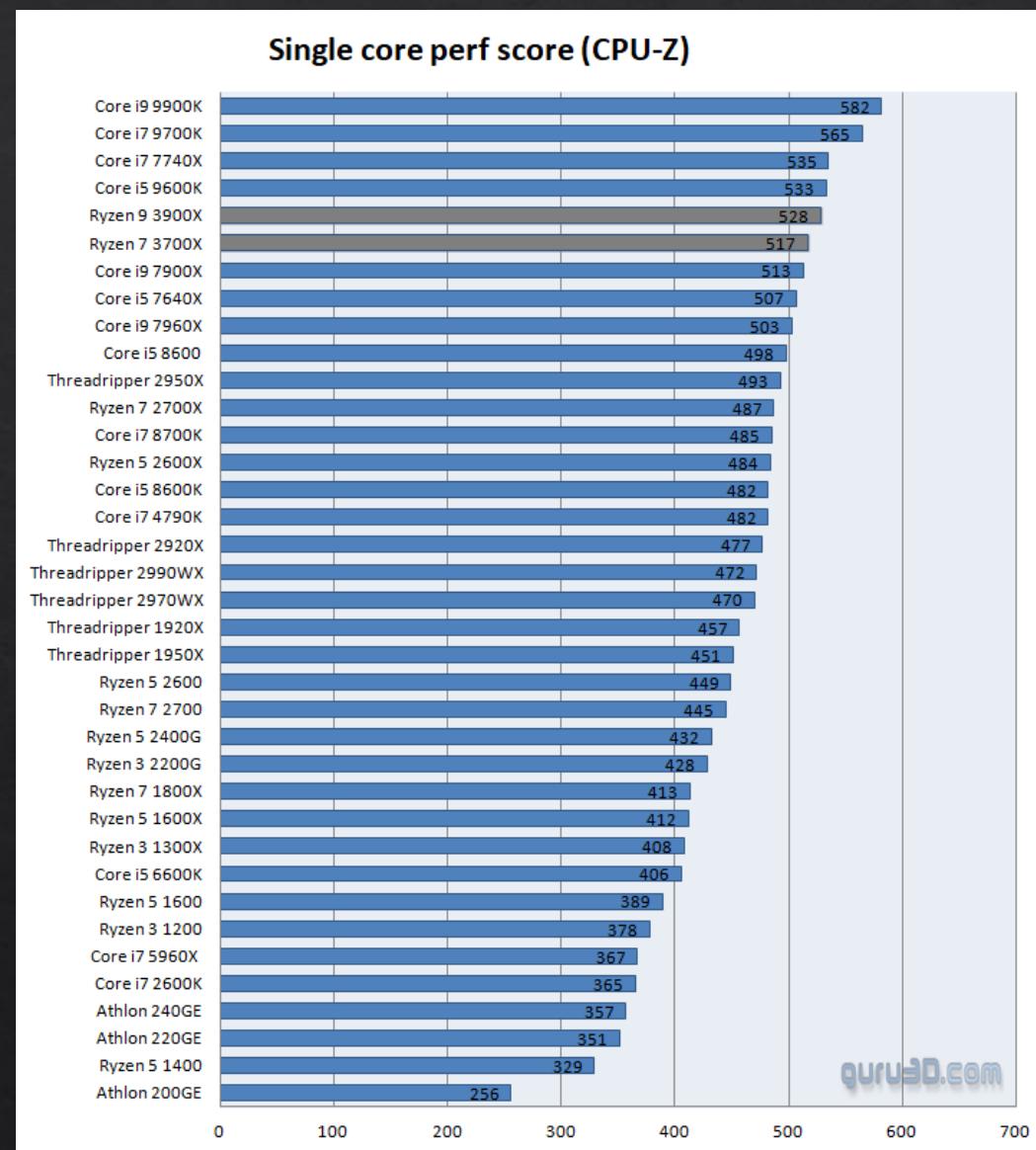
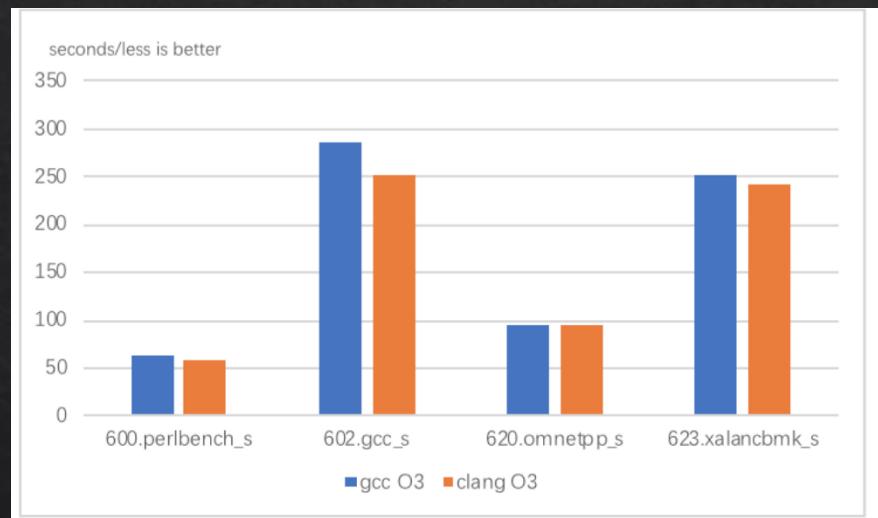
- ❖ Asymptotic Analysis: Big-Oh
- ❖ Relatives of Big-Oh
- ❖ Analyzing Time Complexity of Programs

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
 - ❖ Machine model
 - ❖ CPU
 - ❖ GPU
- ❖ 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 .

CPU benchmark

gcc vs llvm



Why Asymptotic?

- ❖ 3 Algorithms: A, B and C
- ❖ $T_A(n) = 2^n$
- ❖ $T_B(n) = 1024 * n$
- ❖ $T_C(n) = 4096 * n$

- ❖ When does $T_A > T_B$
 - ❖ $n > 13$
- ❖ When does $T_A > T_C$
 - ❖ $n > 15$
- ❖ $T_X(n) = c * n$
 - ❖ Will $T_A > T_X$ at some point?

- ❖ Asymptotic analysis → what happens with a very large input size?

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



The speed with regard to a parametrized size

```
// 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?

Complete the following statements:

- ❖ The best case occurs when **key** is the first element in the array.
- ❖ In the worst case, we need to do n comparisons with **key**.
- ❖ When does worst case happen? When **key** is not in the array.
- ❖ 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
 - ◊ What is “average”?
 - ◊ Often defined as “over purely random inputs”

Is the Following Statement Wrong?

“The best case for my algorithm is $n = 1$ (only a single input) because that is the fastest.”

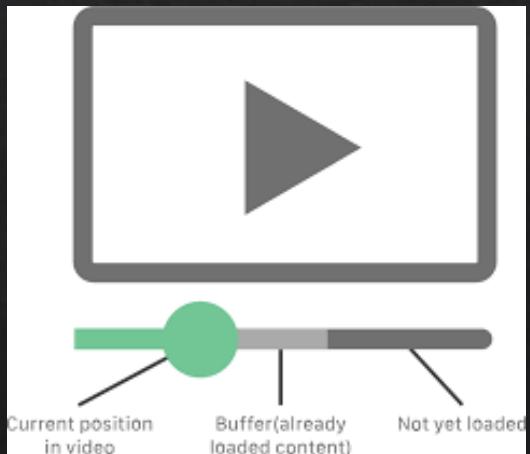
- ❖ Wrong!
- ❖ Best case is a special input case of a [defined size n] that is **cheapest** among all input cases of size n
 - ❖ The input size is fixed during the analysis!

Which Case to Evaluate an Algo?

- ❖ The average case or the worst case are the most 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

Average or Worst? Reality Check

- ❖ Whichever is the most advantageous
 - ❖ Quicksort is usually quite fast
- ❖ Fibonacci Heap is quite cumbersome but it **always** scales well
 - ❖ Very important in Quality of Service (QoS) analysis



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: **scaling** is very important!

- ❖ Thus, we will compare the runtime of two algorithms when n is very large

- ❖ E.g., $1000 \log_2 n$ is “better” than $0.001n$

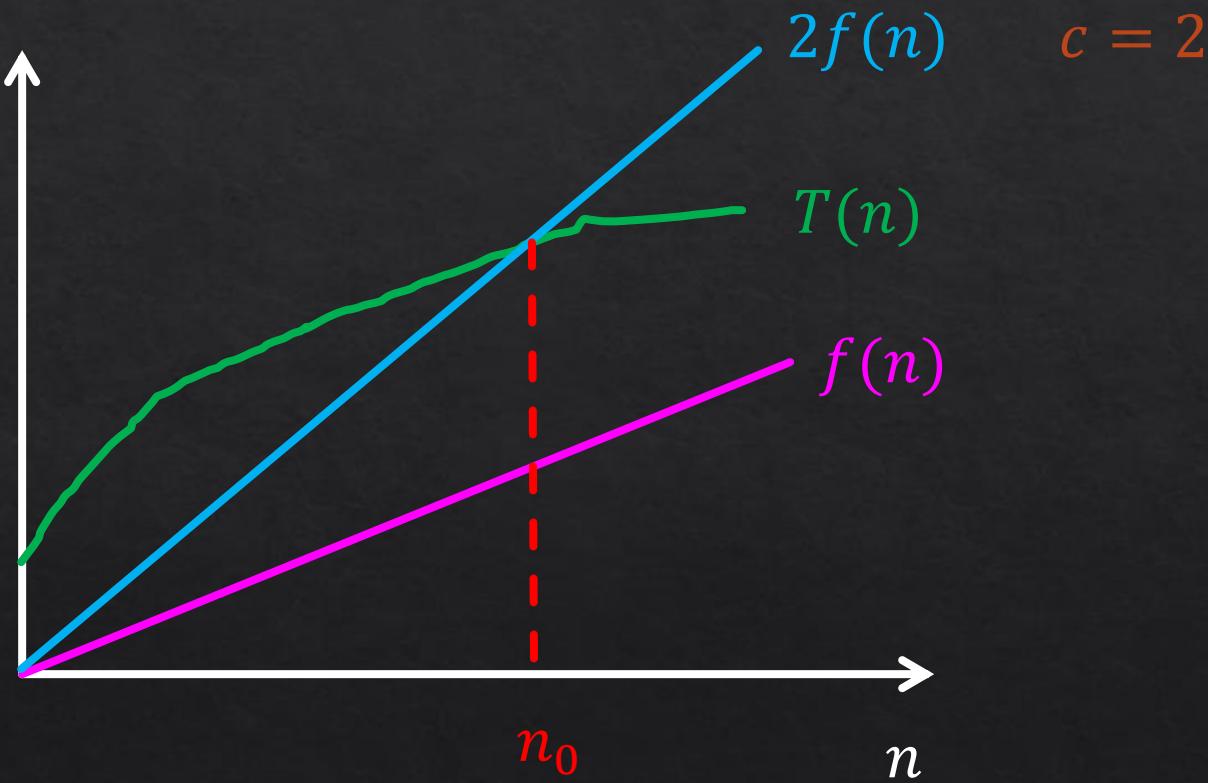
Short Break – 5 min

- ❖ Q & A
- ❖ Keep your discussion in low volume

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
 - ❖ E.g., quicksort has an average-case complexity of $O(n \log n)$
 - ❖ E.g., quicksort has a worst-case complexity of $O(n^2)$
- ❖ 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

Graph Visualization 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:
 $T(n) = O(f(n))$
- ❖ Notice that the “=” here might not communicative
 $2n = O(n^2)$
 $2^n \neq O(2n)$

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 $n > n_0$, $T(n) \leq cn^k$.
- ◊ Anyone?

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 $O(n^2)$.
- ❖ However, we can also say $T(n)$ is in $O(n^3)$ or $O(n^4)$ (why?).
- ❖ We seek the **tightest** upper bound:
 - ❖ While $T(n) = 3n^2$ is in $O(n^3)$, we prefer $O(n^2)$.
 - ❖ In CS research, tightening the upper bound is a common focus:
 - ❖ E.g., prove that the avg. complexity is $O(n \log n)$ rather than $O(n^2)$



True or False?

- ❖ Consider the following statements, are they true:
- ❖ $3n = O(2n)$?
- ❖ $3^n = O(2^n)$?
- ❖ $n^3 = O(n^2)$?
- ❖ $\log_2 n = \log_3 n$?

A Sufficient Condition of Big-Oh

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

❖ Why?

There exists a n_0 , for $n > n_0$, $f(n) < c \cdot g(n)$

❖ With this theorem, we can easily prove that

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

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

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 $f_1(n) + f_2(n) = \max\{O(g_1(n)), O(g_2(n))\}$
 - ❖ Example: $n^3 + 2n^2 = \max\{O(n^3), O(n^2)\} = O(n^3)$

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

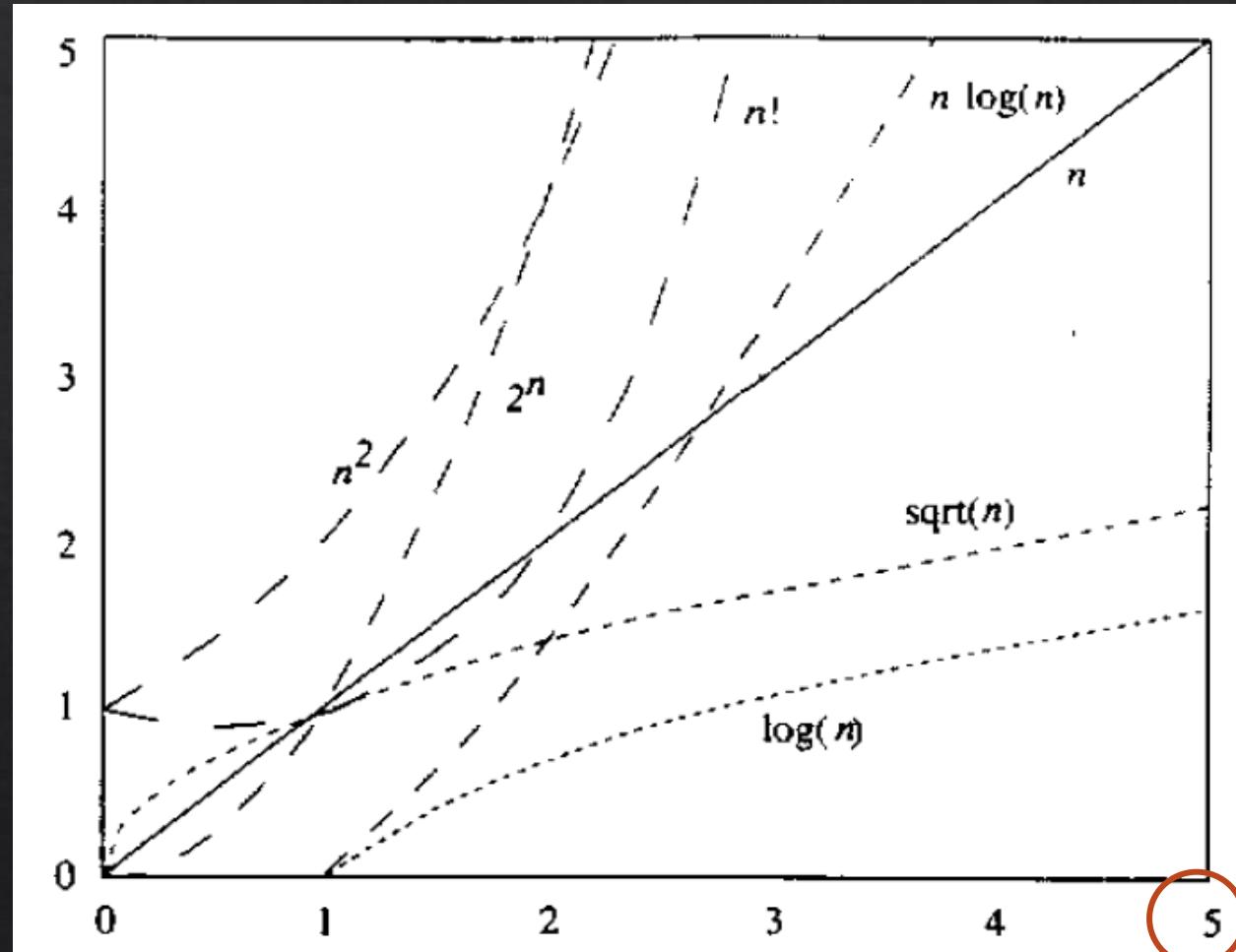
cubic: n^3

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

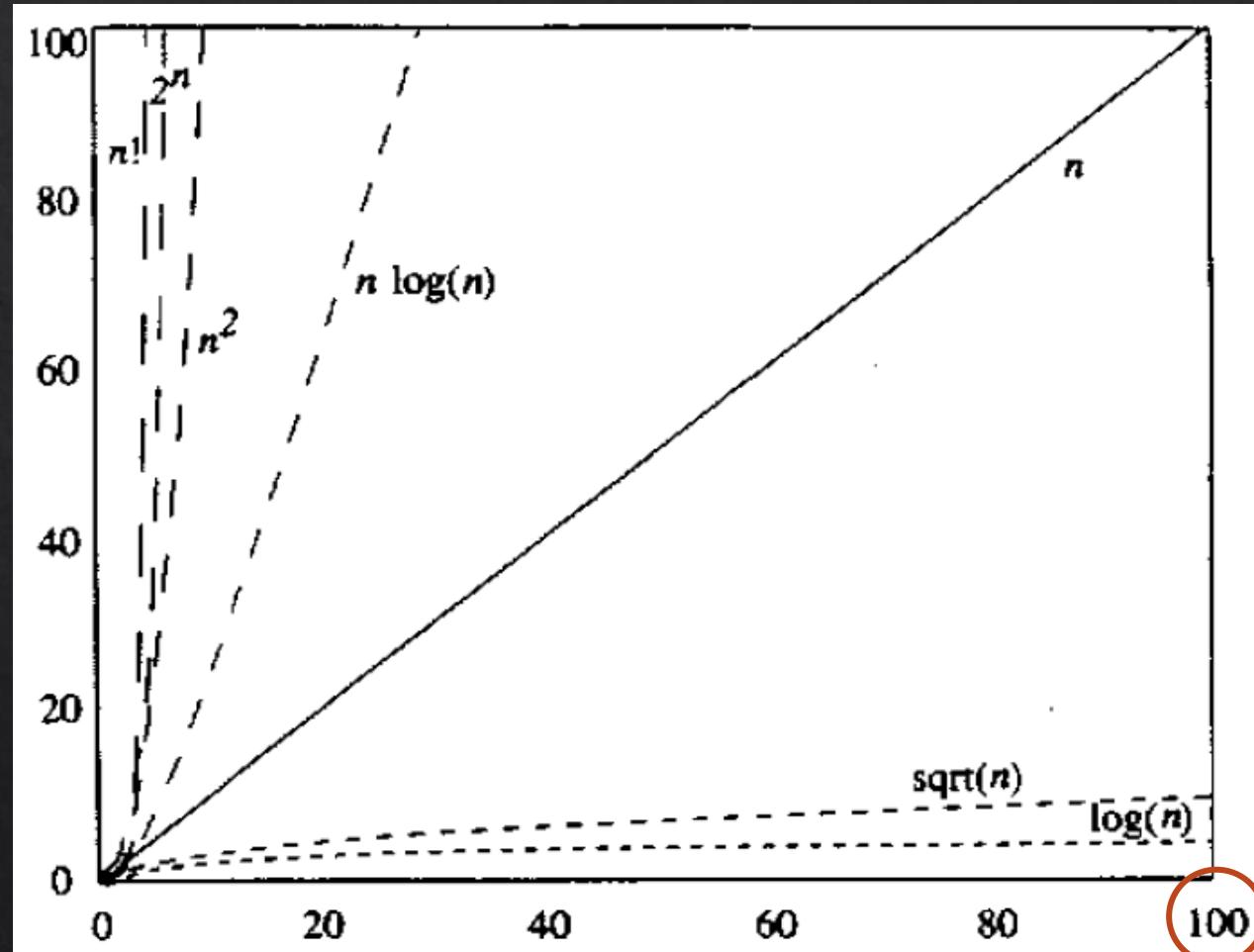
cubic: n^3

general polynomial: n^k

$k \geq 1$

exponential: $a^n, a > 1$

factorial: $n!$



Short Break

- ❖ Time to ask questions
- ❖ Discuss in low volume
- ❖ Come back in 5 minutes!!
- ❖ EZ Question: what is the complexity of the code below?

```
// 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;
}
```

A Few Results about Common Functions

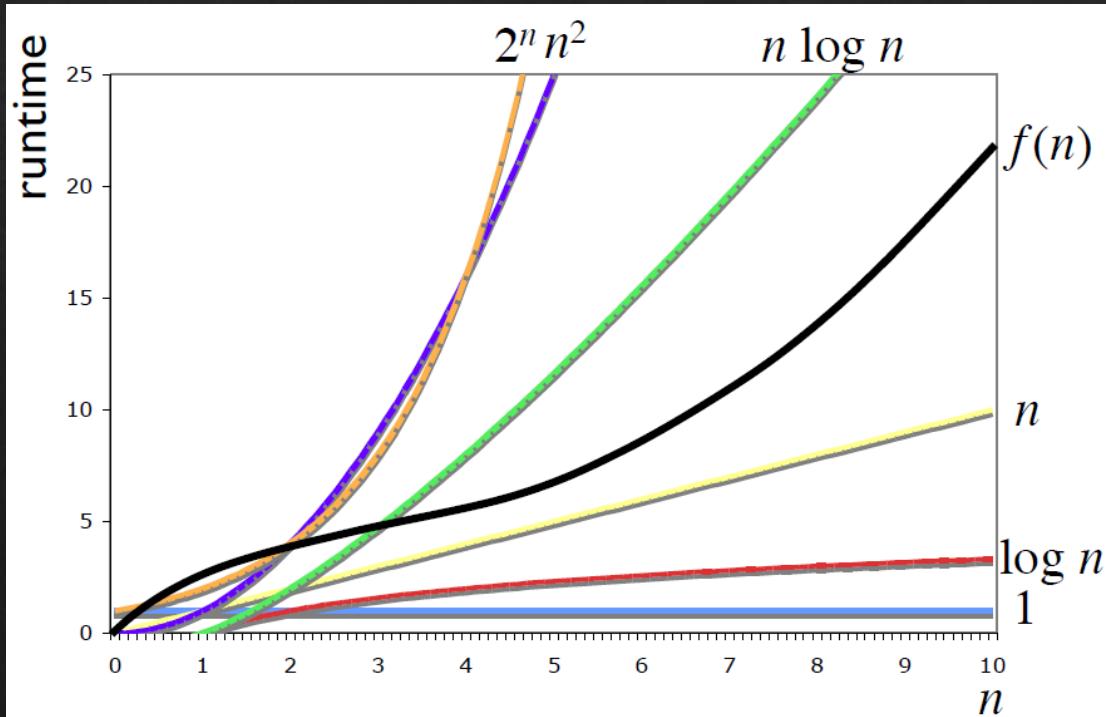
- ❖ For a polynomial in n of the form $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$ where $a_m > 0$, we have $f(n) = O(n^m)$.
- ❖ For every integer $k \geq 1$, $\log_k n = O(n)$.
 - ❖ Tightest bound: $\log_k n = O(\log n)$
- ❖ For every integer $k \geq 1$, $n^k = O(2^n)$.
 - ❖ Tightest bound: $n^k = O(n^k)$.



How Fast Is Your Code?

- Let $f(n) = 0.5n + n \log_2 n$ be the complexity of your code, how fast would you advertise it as?

- A. $O(\log n)$
- B. $O(n \log n)$
- C. $O(n)$
- D. $O(n^2)$



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

What Is a “Fast” Algorithm?

fast algorithm

worst-case/average-case running time
grows slowly with input size
It scales well!

- ❖ Usually as close to linear ($O(n)$) as possible.
 - ❖ Going sublinear (e.g., $O(\log n)$) is usually very hard! But still possible!
 - ❖ Which algorithm has a $O(\log n)$ complexity?

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

Rules of Big-Omega

- ❖ **Rule 1:** If $f(n) = \Omega(g(n))$, then $cf(n) = ?$
- ❖ **Rule 2:** If $f_1(n) = \Omega(g_1(n))$ and $f_2(n) = \Omega(g_2(n))$
 - ❖ Then $f_1(n) + f_2(n) = ?$

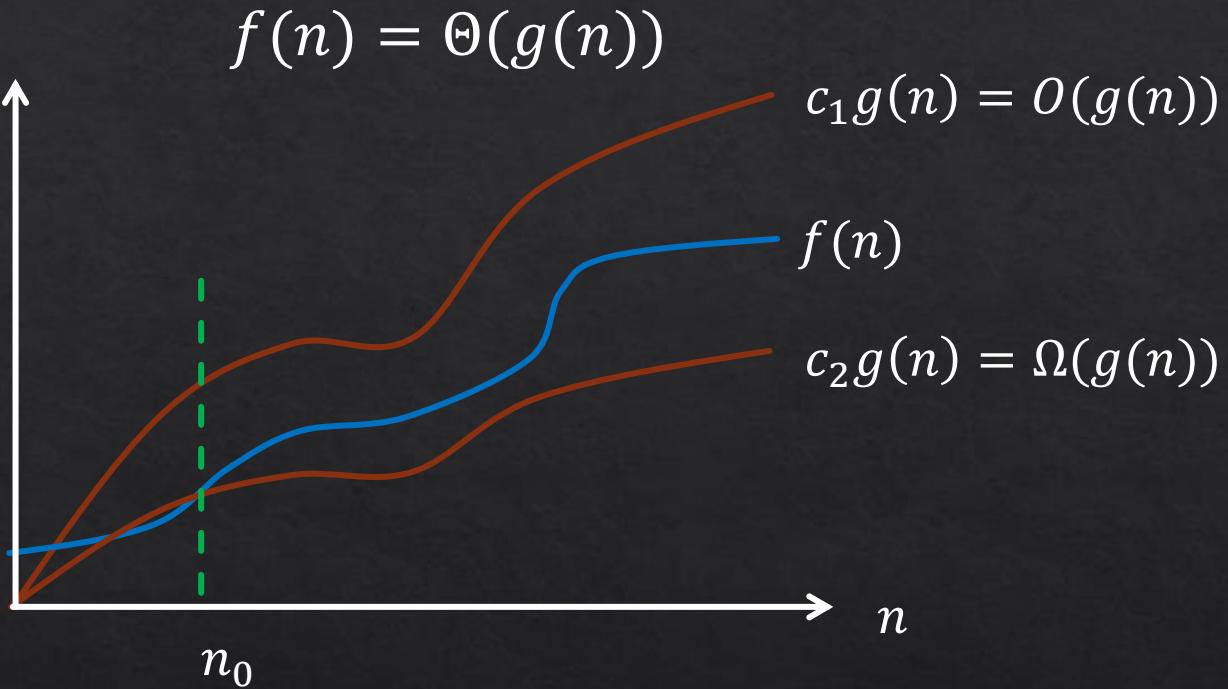
Rules of Big-Omega

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

Theta Notation

- ❖ When big-oh and big-omega are the same, 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 $c_1g(n) \leq T(n) \leq c_2g(n)$ **for all** $n > n_0$.
- ❖ What is the Θ of $T(n) = c_1n^2 + c_2n$?
 - ❖ $\Theta(T(n)) = n^2$

Theta Notation



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

Last Break: 5 min

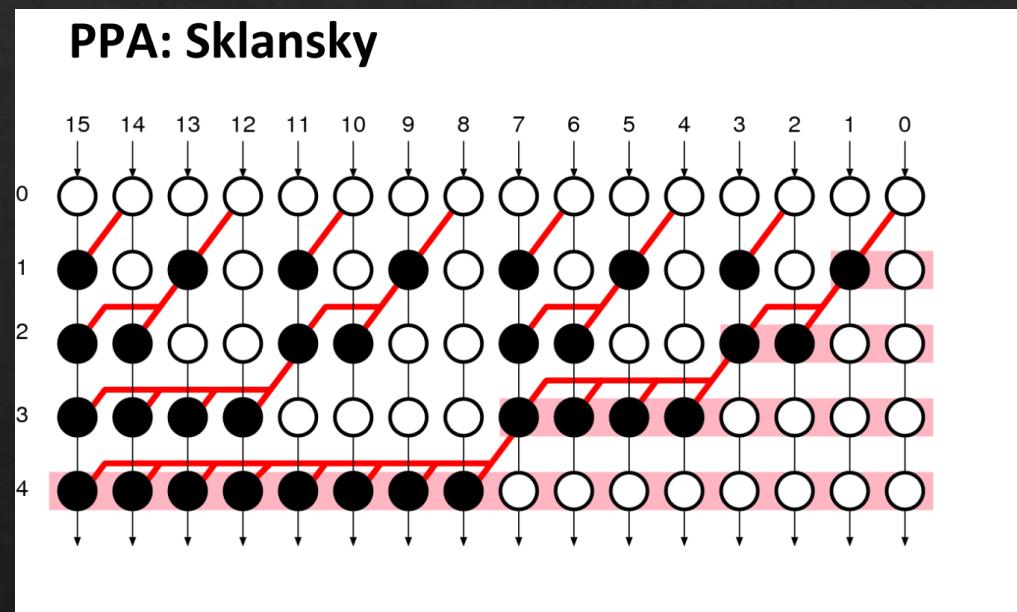
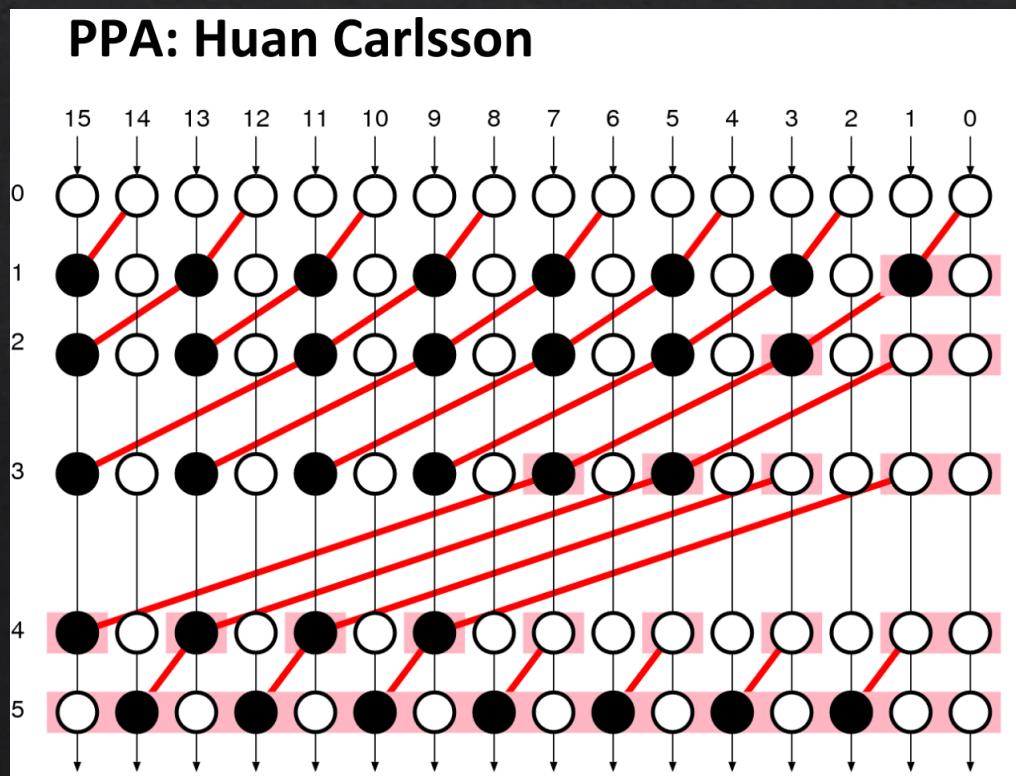
- ❖ Time to ask questions!

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 or addition, its complexity is $\Theta(1)$
 - ❖ Addition is atomic?
 - ❖ Conceptually yes, but in reality... it's complicated!



Analyzing Time Complexity of Programs

- ❖ 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.
 - ❖ What is an addition of two complexity statements?

```
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}
```

- ❖ Why?

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.

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

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

$\Theta(C \log C)$ `sort(count); // Sort pixel counts`

- ◊ The time complexity is $\Theta(P + C \log C)$
- ◊ One general application is to analyze graph algorithm (#nodes and #edges)

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)

That is All for today!

- ❖ Questions?

