# Algorithm Analysis 02

Computer Science Department
Eastern Washington University
Yun Tian (Tony) Ph.D.

# Recall Last Lecture

- Two approaches to analyze the algorithm
- Better to use theoretical analysis
- Seven functions
- Growth Rate Function(GRF)
  - Measures how much(quickly) an algorithm becomes slower when we increase the input size.

# Today Class

- Big-Oh Notation and Why use it
- Rules about Big-Oh notation
- Analysis Case Study

# Counting Primitive Operations

1

N+1

N

```
public String toString() {
        String result = "";  (1) + (1)
        Node cur;(1)
        for( cur = this.head.next; cur!= this.head; cur = cur.next) {
                results += cur.data + "\n";  //(2N)
        }
        return result; //(1)
}//if the size of this list is N
```

Total  4N + 6. Here, f(n) = 4N + 6 is the growth rate function.

# Estimating Running Time

- Algorithm ***toString()*** executes $4n + 6$ primitive operations in the worst case. Define:
  - $a$ = Time taken by the fastest primitive operation
  - $b$ = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of ***toString***. Then
  $$a\,(4n + 6) \le T(n) \le b(4n + 6)$$
- Hence, the running time $T(n)$ is bounded by two linear functions

# Estimating Running Time

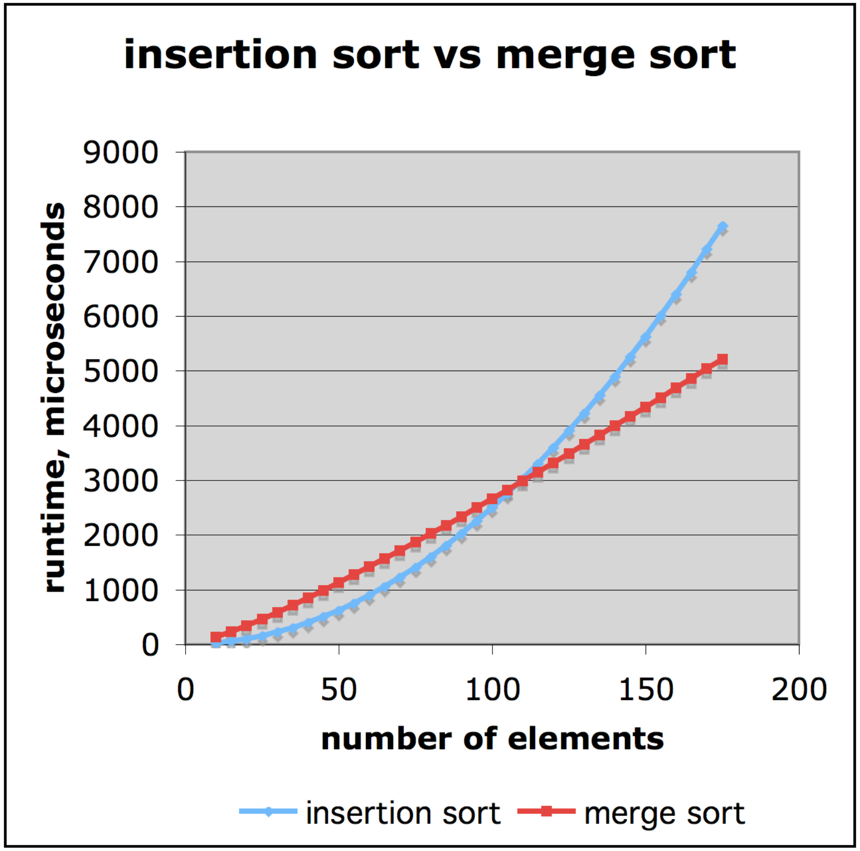
- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***toString.***
- ***In other words, even though you move your toString() from a slow computer to a fast computer, the growth rate function is still a straight line.***

# Why Growth Rate Matters

| if runtime is... | time for n + 1 | time for 2 n | time for 4 n |
|---|---|---|---|
| $c \lg n$ | $c \lg (n + 1)$ | $c (\lg n + 1)$ | $c(\lg n + 2)$ |
| $c\ n$ | $c\ (n + 1)$ | $2c\ n$ | $4c\ n$ |
| $c\ n \lg n$ | $\sim c\ n \lg n + c\ n$ | $2c\ n \lg n + 2cn$ | $4c\ n \lg n + 4cn$ |
| $c\ n^2$ | $\sim c\ n^2 + 2c\ n$ | **$4c\ n^2$** | $16c\ n^2$ |
| $c\ n^3$ | $\sim c\ n^3 + 3c\ n^2$ | $8c\ n^3$ | $64c\ n^3$ |
| $c\ 2^n$ | $c\ 2^{n+1}$ | $c\ 2^{2n}$ | $c\ 2^{4n}$ |

runtime quadruples when problem size doubles

# Why Growth Rate Matters




insertion sort is

$n^2 / 4$

merge sort is

$2 \, n \lg n$

sort a million items?

insertion sort takes roughly 70 hours

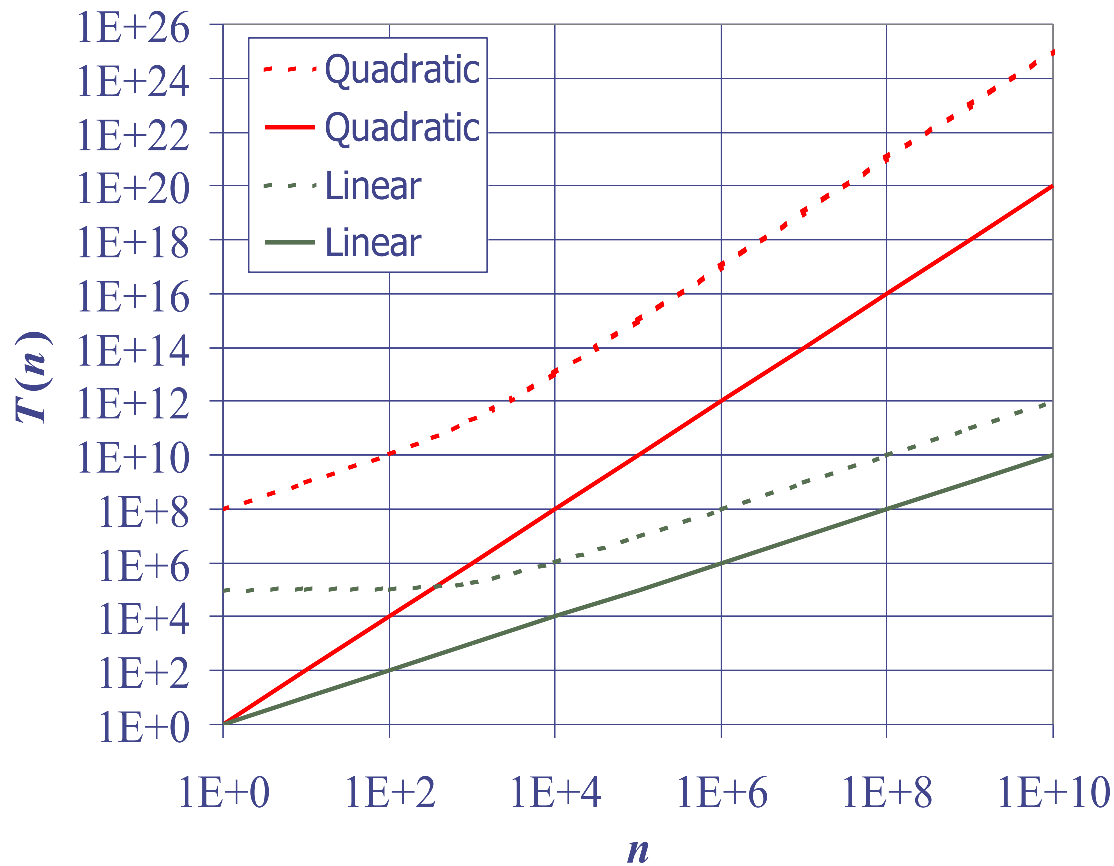
while

merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

# Growth Rate Function

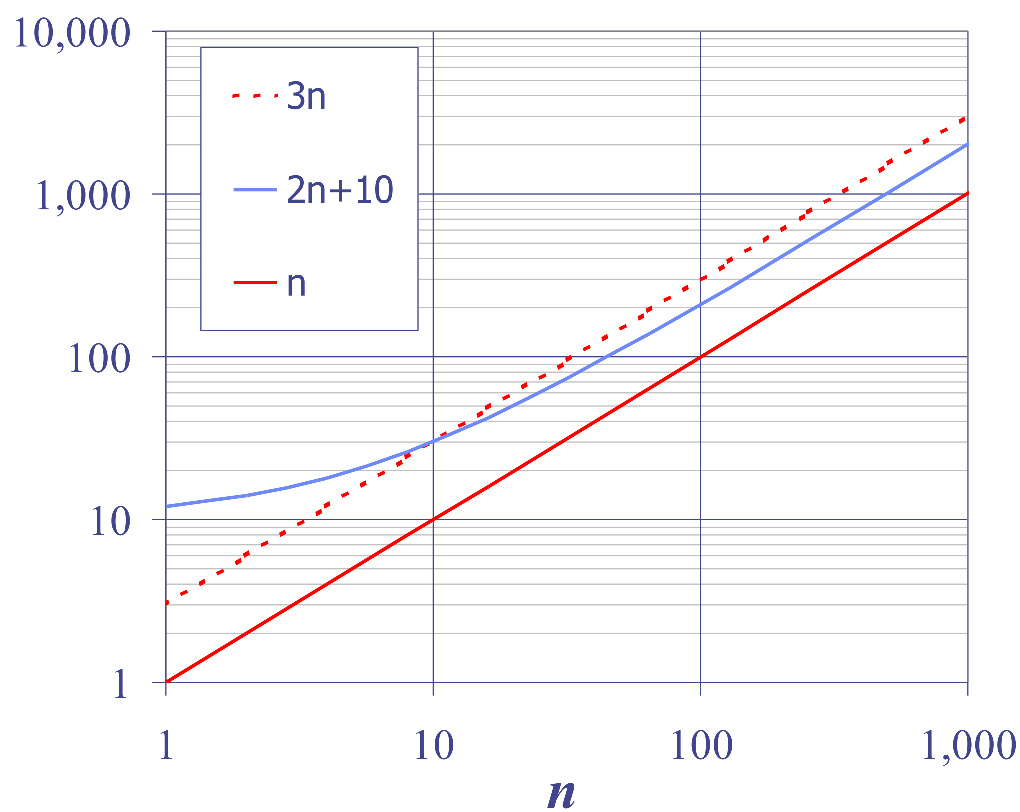
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



# Big-Oh Notation

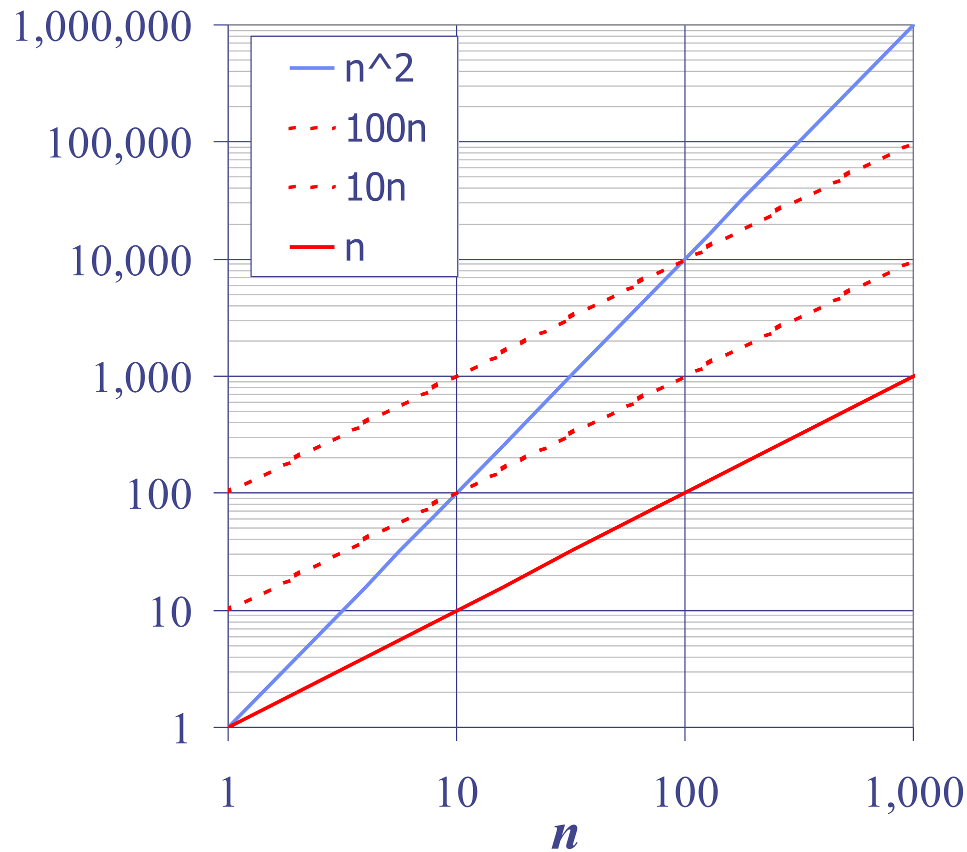
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that

  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - Pick $c = 3$ and $n_0 = 10$



# Big-Oh Notation

- Example: the function $n^2$ is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since $c$ must be a constant.



# More Big-Oh Examples

- $7n - 2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

# More Big-Oh Examples

- 3 log n + 5

  3 log n + 5 is O(log n)

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

- The big-Oh notation allows us to say that a function f(n) is "less than or equal to" another function g(n) up to a constant factor and in the asymptotic sense as ***n*** grows towards infinity.

# Big-Oh and Growth Rate

- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.

# Big-Oh Rules

- If is $f(n)$ a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of $O(2n)$ OR $O(n^2)$
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

# Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size.
  - We express this function with big-Oh notation.

# Counting Primitive Operations

```
public String toString() {
        String result = "";  (1) + (1)
        Node cur;(1)
                                              1        N+1                 N
        for( cur = this.head.next; cur!= this.head; cur = cur.next) {
                results += cur.data + "\n";  //(2N)
        }
        return result; //(1)
}//if the size of this list is N
```

Total    4N + 6. Here, f(n) = 4N + 6 is the growth rate function.

# Asymptotic Algorithm Analysis

- Example
  - We determine that algorithm ***toString()*** executes at most $4n + 6$ primitive operations
  - *f(n) = 4n + 6 and g(n) = n*, because f(n) < c*g(n) when c = 5 and n > 6.
  - We say that algorithm ***toString()*** "runs in $O(n)$ time".
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations sometimes.

# Math you need to Review

- Summations, $1 + 2 + 3 + \ldots + n = n(1+n) / 2$;
- Geometric Sums:

  $1 + a + a^2 + a^3 + \ldots + a^n = (a^{n+1} - 1) / (a-1)$
- Logarithms and Exponents

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b x^a = a\log_b x$

  $\log_b a = \log_x a / \log_x b$
- **properties of exponentials**:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

- Proof techniques
- Basic probability

# Take Home Summary

- Big-Oh notation
- Rules of Big-Oh notation
- Growth Rate Function(GRF) with Big-Oh

# Next Class

- Asymptotic Algorithm Analysis using Big-Oh notation
  - Analysis Case Studies