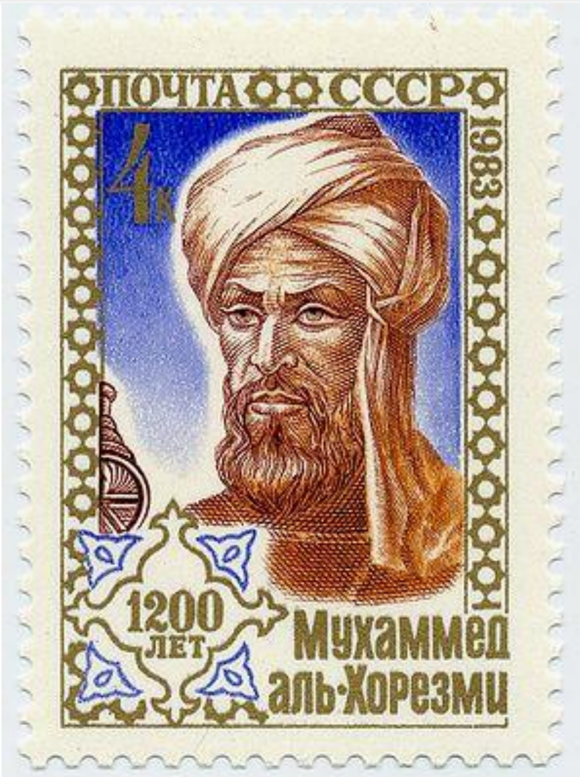
CS 311: Algorithm Design and Analysis

Lecture 2

Last Lecture we have

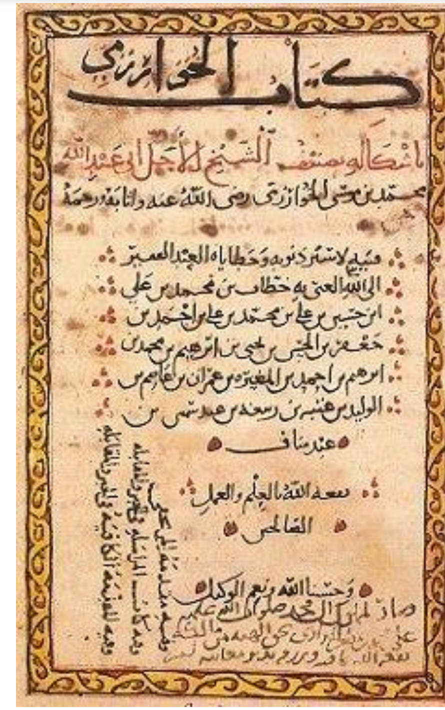
- Introduction
- Asymptotic notation

Khâwrázmî (780-850 A.D.)



A stamp issued
September 6, 1983
in the Soviet Union,
commemorating
Khâwrázmî's
1200th birthday.

Statue of Khâwrázmî
in front of the
Faculty of Mathematics,
Amirkabir University of
Technology,
Tehran, Iran.



A page from his
book.

Courtesy of Wikipedia

Algorithm

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Analysis of Algorithms

- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Example

Time complexity shows dependence of algorithm's running time on input size.

Let's assume: Computer speed = 10^6 IPS,
Input: a data base of size $n = 10^6$

Time Complexity	Execution time
n	1 sec
$n \log n$	20 sec
n^2	12 days
2^n	40 quadrillion (10^{15}) years

Machine Model

- **Algorithm Analysis:**
  - should reveal intrinsic properties of the algorithm itself.
  - should not depend on any computing platform, programming language, compiler, computer speed, etc.
- **Elementary steps:**
  - arithmetic: $+ \quad - \quad \times \quad \div$
  - logic: and or not
  - comparison: $= \quad < \quad > \quad \neq \quad \leq \quad \geq$
  - assigning a value to a scalar variable: $\leftarrow$
  - ….

Complexity

- Space complexity
- Time complexity
  - For iterative algorithms: sums
  - For recursive algorithms: recurrence relations

Time Complexity

- **Time complexity** shows dependence of algorithm's running time on input size.
  - Worst-case
  - Average or expected-case

- **What is it good for?**
  - Tells us how efficient our design is before its costly implementation.
  - Reveals inefficiency bottlenecks in the algorithm.
  - Can use it to compare efficiency of different algorithms that solve the same problem.
  - Is a tool to figure out the true complexity of the problem itself! How fast is the "fastest" algorithm for the problem?
  - Helps us classify problems by their time complexity.

$T(n) = Q(\ f(n)\ )$

$$T(n) = \cancel{23}\, n^3 \cancel{+ 5n^2 \log n + 7n \log^2 n + 4 \log n + 6.}$$

drop constant multiplicative factor

drop lower order terms

$$T(n) = \Theta(n^3)$$

Why do we want to do this?

1. Asymptotically (at very large values of n) the leading term largely determines function behavior.
2. With a new computer technology (say, 10 times faster) the leading coefficient will change (be divided by 10). So, that coefficient is technology dependent any way!
3. This simplification is still capable of distinguishing between important but distinct complexity classes, e.g., linear vs. quadratic, or polynomial vs exponential.

Asymptotic Notations: ϴ O Ω o ω

Rough, intuitive meaning worth remembering:

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c\, g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c\, g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c\, g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) \ll c\, g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) \gg c\, g(n)$

$$\lim_{n\to\infty} f(n)/g(n) = \begin{cases} 0 & \text{order of growth of } \mathbf{f(n)} < \text{order of growth of } \mathbf{g(n)} \\ & \mathbf{f(n) \in o(g(n)),\ f(n) \in O(g(n))} \\ \mathbf{c}>0 & \text{order of growth of } \mathbf{f(n)} = \text{order of growth of } \mathbf{g(n)} \\ & \mathbf{f(n) \in \Theta(g(n)),\ f(n) \in O(g(n)),\ f(n) \in \Omega(g(n))} \\ \infty & \text{order of growth of } \mathbf{f(n)} > \text{order of growth of } \mathbf{g(n)} \\ & \mathbf{f(n) \in \omega(g(n)),\ f(n) \in \Omega(g(n))} \end{cases}$$

Asymptotics by ratio limit

$L = \lim_{n \to \infty} f(n)/g(n)$. If L exists, then:

Theta	$f(n) = \Theta(g(n))$	$0 < L < \infty$
Big Oh	$f(n) = O(g(n))$	$0 \leq L < \infty$
Big Omega	$f(n) = \Omega(g(n))$	$0 < L$
Little Oh	$f(n) = o(g(n))$	$L = 0$
Little Omega	$f(n) = \omega(g(n))$	$L = \infty$

Examples:

- $\log_b n$ vs. $\log_c n$

$$\log_b n = \log_b c \log_c n$$

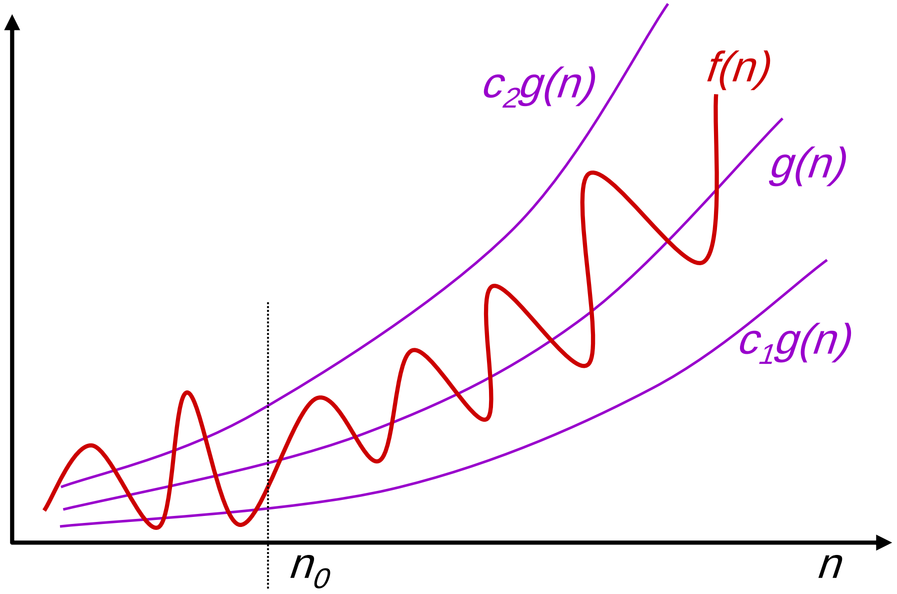
$$\lim_{n\to\infty}(\log_b n / \log_c n) = \lim_{n\to\infty} (\log_b c) = \log_b c$$

$$\log_b n \in \Theta(\log_c n)$$

Theta : Asymptotic Tight Bound

$f(n) = \Theta(g(n))$

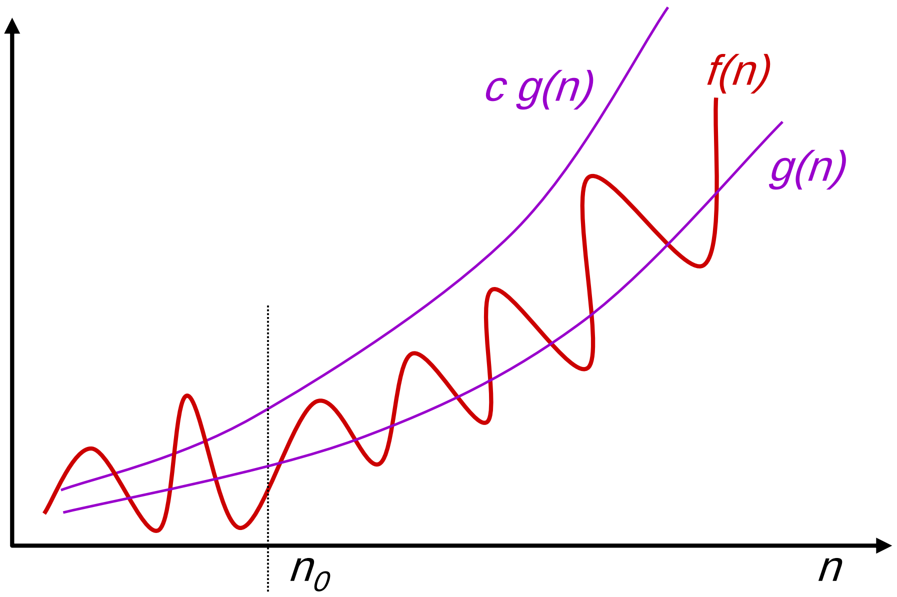
$\Updownarrow$



$\underbrace{\exists c_1, c_2, n_0}_{\in \Re^+} > 0 \;:\; \forall n \geq n_0, \quad c_1 g(n) \leq f(n) \leq c_2 g(n).$

Big Oh: Asymptotic Upper Bound

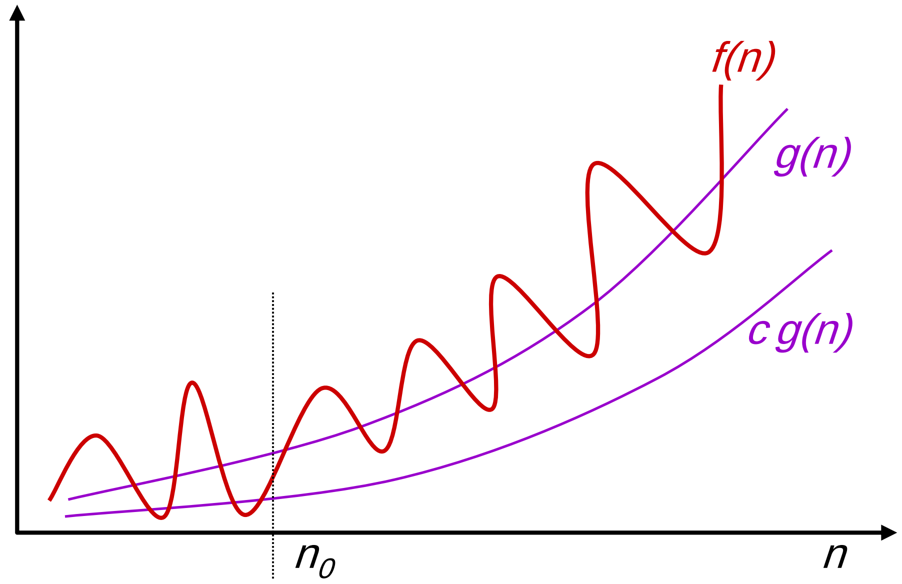
$$f(n) = O(g(n))$$



$$\exists \underbrace{c,\ n_0}_{\in \Re^+} > 0 \ : \ \forall n \geq n_0, \quad f(n) \leq c\, g(n).$$

Big Omega : *Asymptotic Lower Bound*

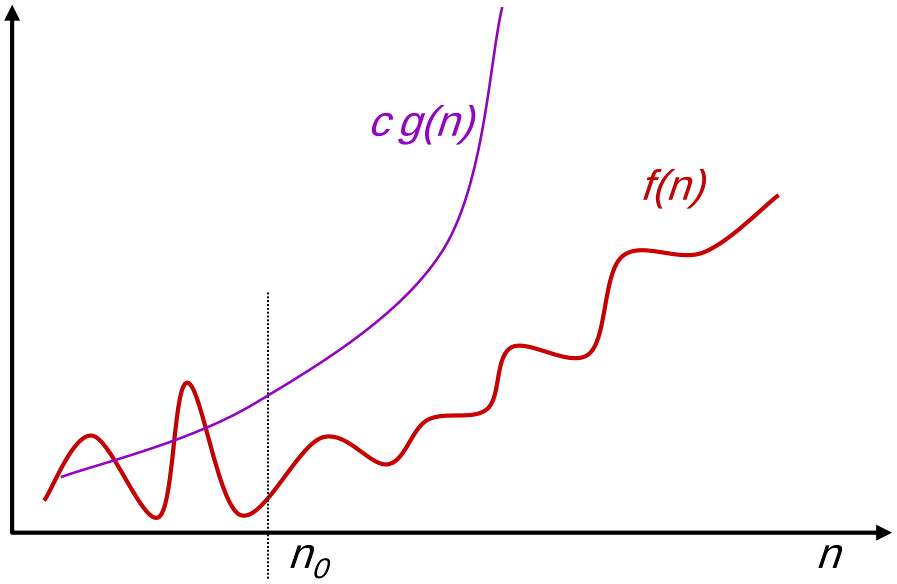
$$f(n) = \Omega(g(n))$$



$$\underbrace{\exists c,\ n_0}_{\in \mathfrak{R}^+} > 0\ :\ \forall n \geq n_0, \quad cg(n) \leq f(n).$$

Little oh : Non-tight Asymptotic Upper Bound

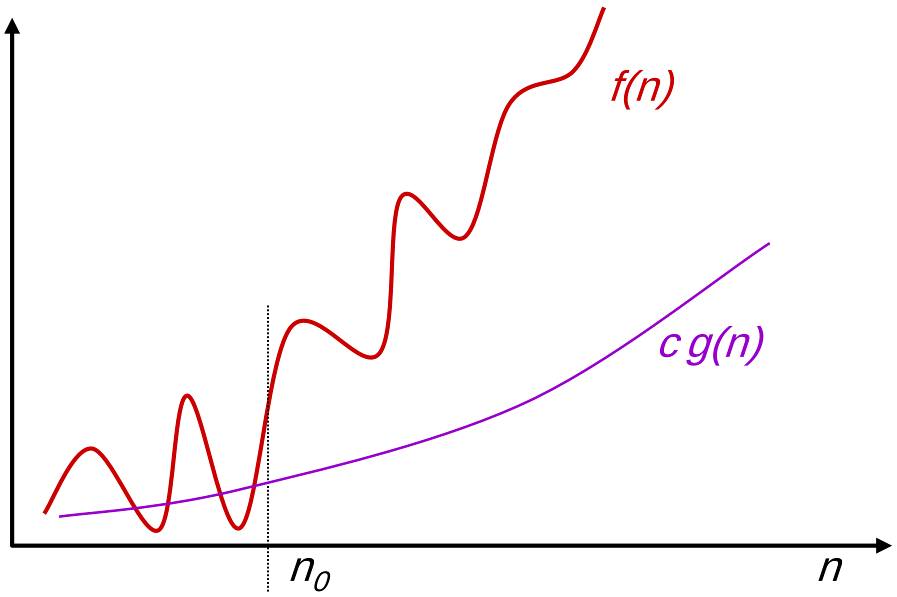
$$f(n) = o(g(n))$$



$$\forall c > 0, \exists n_0 > 0 : \forall n \ge n_0, \quad f(n) < c\, g(n).$$

No matter how small $\in \Re^+$

Little omega : Non-tight Asymptotic Lower Bound



$$f(n) = \omega(g(n))$$

$$\Updownarrow$$

$$\forall c > 0,\ \exists n_0 > 0 :\ \forall n \geq n_0,\quad f(n) > c\, g(n).$$

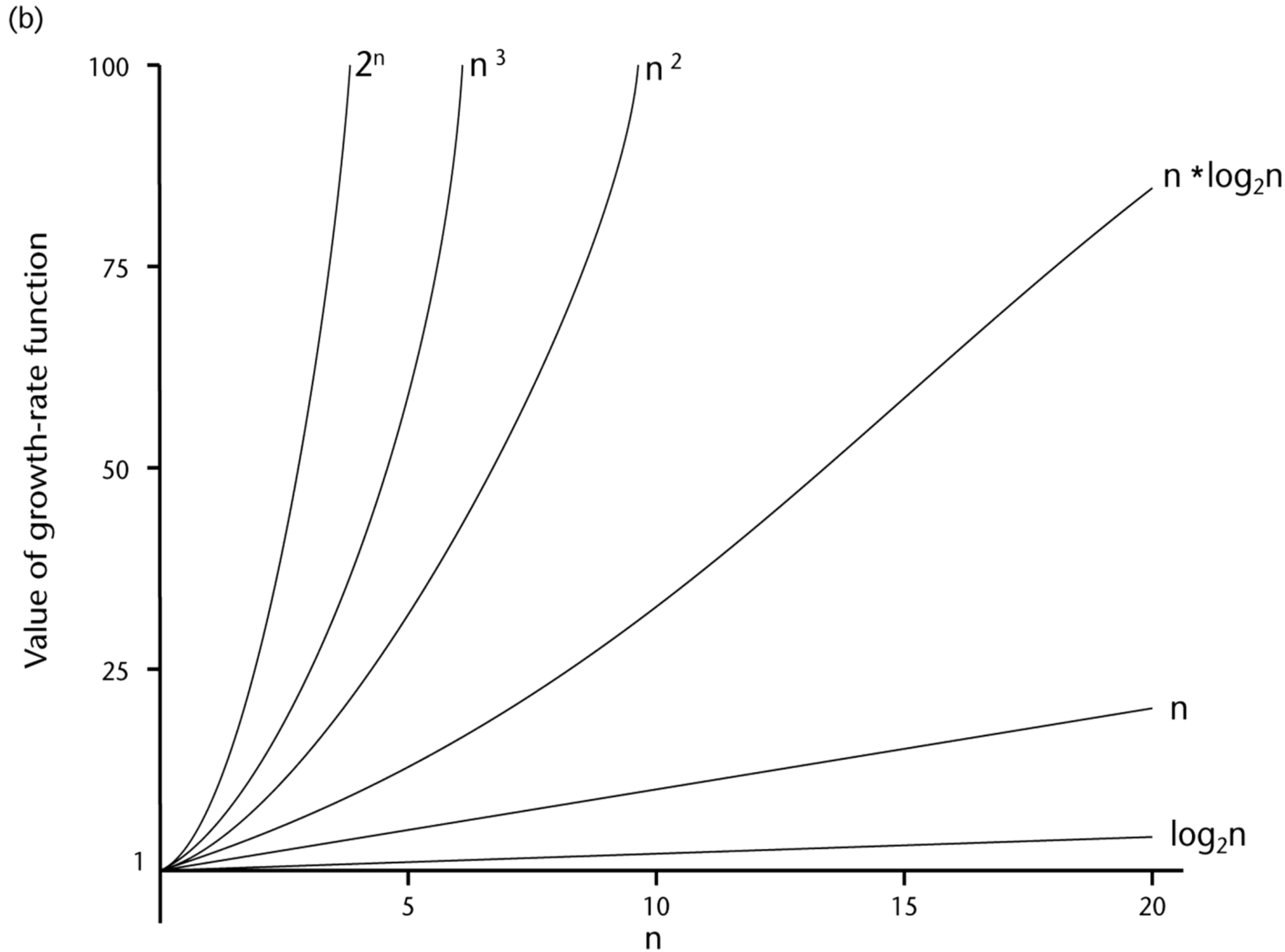
No matter how large $\in \Re^+$

Definitions of Asymptotic Notations

$f(n) = \Theta(g(n))$	$\exists c_1, c_2 > 0, \exists n_0 > 0: \forall n \geq n_0, \; c_1 g(n) \leq f(n) \leq c_2 g(n)$
$f(n) = O(g(n))$	$\exists c > 0, \quad \exists n_0 > 0: \forall n \geq n_0, \quad f(n) \leq c\, g(n)$
$f(n) = \Omega(g(n))$	$\exists c > 0, \quad \exists n_0 > 0: \forall n \geq n_0, \; c\, g(n) \leq f(n)$
$f(n) = o(g(n))$	$\forall c > 0, \quad \exists n_0 > 0: \forall n \geq n_0, \quad f(n) < c\, g(n)$
$f(n) = \omega(g(n))$	$\forall c > 0, \quad \exists n_0 > 0: \forall n \geq n_0, \quad c\, g(n) < f(n)$

(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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$



(b)
Value of growth-rate function
100
75
50
25
1
2^n
n^3
n^2
$n * \log_2 n$
n
$\log_2 n$
5
10
15
20
n

Ordering Functions

Functions

Constant << Logarithmic << Poly Logarithmic << Polynomial << Exponential << Factorial

$5 \ll 5 \log n \ll (\log n)^5 \ll n^5 \ll 2^{5n} \ll n!$

Classifying Functions

Polynomial

Linear	Quadratic	Cubic	?
$5n$	$5n^2$	$5n^3$	$5n^4$

Big O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\le \quad n^k \sum a_i \frac{n^i}{n^k} \quad \le \quad n^k \sum a_i \quad \le \quad cn^k$$

Example Problem: Sorting

Some sorting algorithms and their worst-case time complexities:

Quick-Sort:	$\Theta(n^2)$
Insertion-Sort:	$\Theta(n^2)$
Selection-Sort:	$\Theta(n^2)$
Merge-Sort:	$\Theta(n \log n)$
Heap-Sort:	$\Theta(n \log n)$

there are infinitely many sorting algorithms!

So, Merge-Sort and Heap-Sort are worst-case optimal, and

SORTING complexity is Q(n log n).

Input size and basic operation examples

Problem	***Input size measure***	***Basic operation***
Search for key in list of n items	Number of items in list n	Key comparison
Multiply two matrices of floating point numbers	Dimensions of matrices	Floating point multiplication
Compute a^n	n	Floating point multiplication
Graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the <u>basic operation</u> as a function of <u>input size</u>

Best-case, average-case, worst-case

- *Worst case: W(n) – maximum over inputs of size n*
- *Best case: B(n) – minimum over inputs of size n*
- *Average case: A(n) – "average" over inputs of size n*
  - *NOT the average of worst and best case*
  - *Under some assumption about the probability distribution of all possible inputs of size n, calculate the weighted sum of expected C(n) (numbers of basic operation repetitions) over all possible inputs of size n.*

Time efficiency of nonrecursive algorithms

- *Steps in mathematical analysis of nonrecursive algorithms:*
  - *Decide on parameter n indicating input's size*
  - *Identify algorithm's basic operation*
  - *Determine worst, average, & best case for inputs of size n*
  - *Set up summation for C(n) reflecting algorithm's loop structure*
  - *Simplify summation using standard formulas*

Series

$$S = \sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

- ***Proof by Gauss when 9 years old (!):***

$$S = 1 + 2 + 3 + \ldots + (N-2) + (N-1) + N$$

$$S = N + (N-1) + (N-2) + \ldots + 3 + 2 + 1$$

$$- -$$

$$2S = N(N+1)$$

General rules for sums

$$\sum_{i=m}^{n} c = c\sum_{i=m}^{n} 1 = c(n-m+1)$$

$$\sum_{i} (a_i + b_i) = \sum_{i} a_i + \sum_{i} b_i$$

$$\sum_{i} ca_i = c\sum_{i} a_i$$

$$\sum_{i=m}^{n} a_{i+k} = \sum_{i=m+k}^{n+k} a_i$$

$$\sum_{i} a_i x^{i+k} = x^k \sum_{i} a_i x^i$$

Some Mathematical Facts

- Some mathematical equalities are:

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

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

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

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

Code	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

Growth-Rate Functions – Example1

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
`}`		

$$
\begin{aligned}
T(n) &= c1 + c2 + (n+1)*c3 + n*c4 + n*c5 \\
&= (c3+c4+c5)*n + (c1+c2+c3) \\
&= a*n + b
\end{aligned}
$$

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

Growth-Rate Functions – Example2

Code	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
`}`		

$$T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8$$
$$= (c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$$
$$= a*n^2 + b*n + c$$

➔ So, the growth-rate function for this algorithm is **$O(n^2)$**

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)**

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:

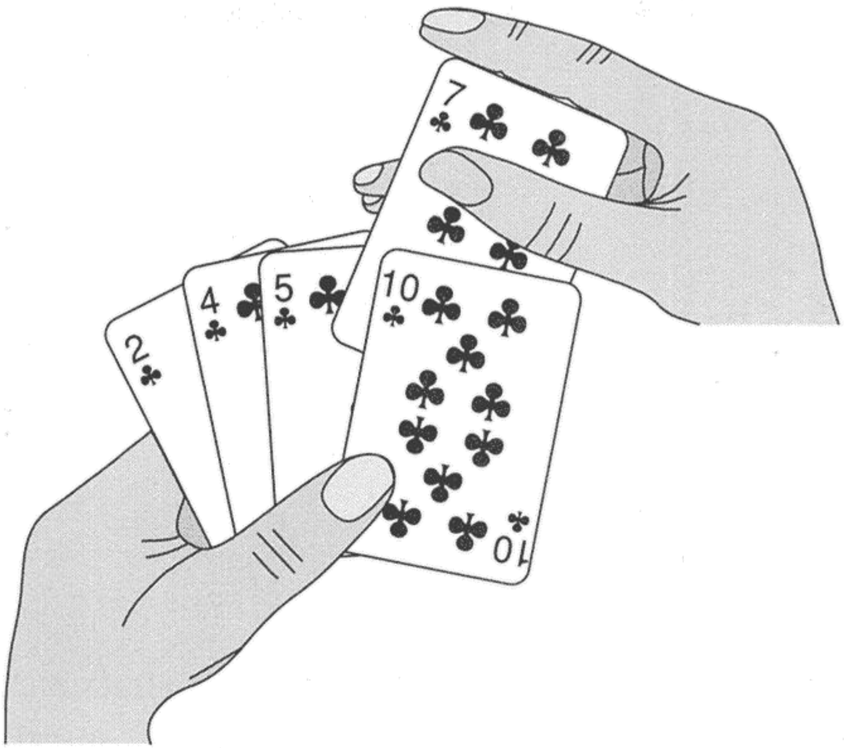
Best-Case: *item* is in the first location of the array ➔O(1)
Worst-Case: *item* is in the last location of the array ➔O(n)
Average-Case: The number of key comparisons 1, 2, ..., n

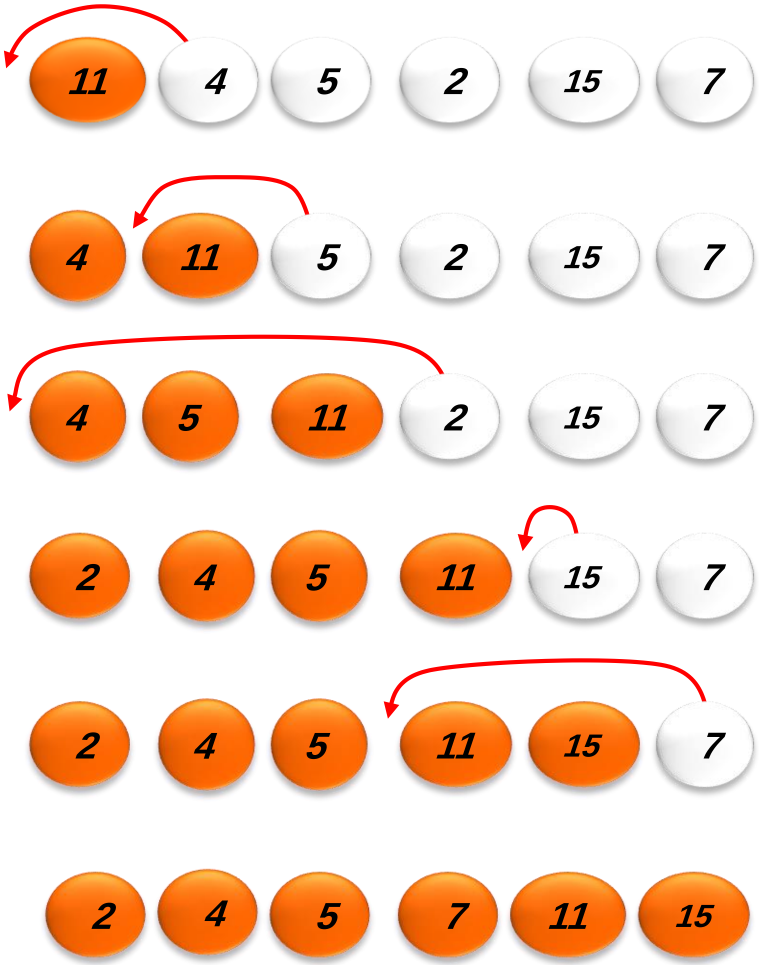
$$\frac{\sum_{i=1}^{n} i}{n} = \frac{(n^2+n)/2}{n}$$

➔ O(n)

Insertion Sort

an incremental algorithm





Insertion Sort: Time Complexity

```
Algorithm InsertionSort(A[1..n])
for i ← 2 .. n do
        LI: A[1..i –1] is sorted, A[i..n] is untouched.
    § insert A[i] into sorted prefix A[1..i–1] by right-cyclic-shift:
        key ← A[i]
        j ← i –1
        while j > 0 and A[j] > key do
            A[j+1] ← A[j]
            j ← j –1
        end-while
        A[j+1] ← key
    end-for
end
```

$$T(n)$$
$$= \Theta\left(\sum_{i=2}^{n} \left(1 + t_i + 1\right) \right)$$
$$= \Theta\left(n + \sum_{i=2}^{n} t_i \right)$$
$$= \Theta\left(n + \sum_{i=2}^{n} i \right)$$
$$= \Theta\left(n + n^2\right)$$
$$= \Theta\left(n^2\right).$$

Worst-case: $t_i = i$ iterations (reverse sorted).

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \Theta\left(n^2\right).$$

Master theorem

- If $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O\left(n^d\right)$ for some constants a > 0, b> 1, and d $\geq$ 0, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

1. $a < b^d$ $\quad T(n) \in \Theta(n^d)$
2. $a = b^d$ $\quad T(n) \in \Theta(n^d \lg n)$
3. $a > b^d$ $\quad T(n) \in \Theta(n^{\log_b a})$

The End