DISCRETE MATHEMATICS AND ITS APPLICATIONS

# 3.2 GROWTH OF FUNCTIONS

WENJING LI

**wjli@bupt.edu.cn**

SCHOOL OF COMPUTER SCIENCE

BEIJING UNIVERSITY OF POSTS & TELECOMMUNICATIONS

## Growth of Functions(函数增长)

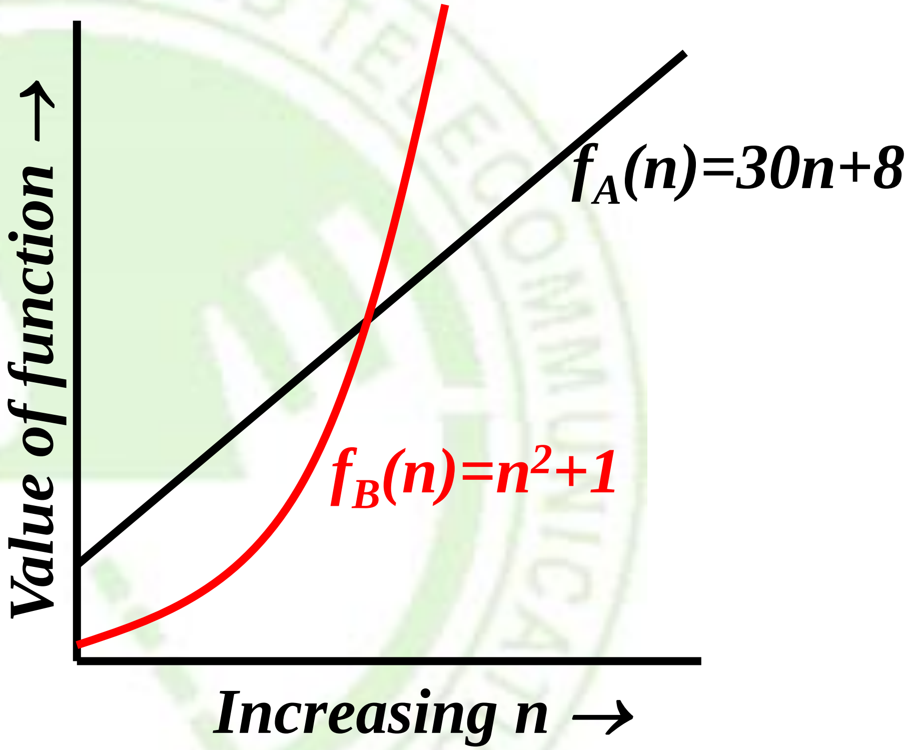
- We quantify the concept that ***g grows at least as fast as f***.
- What really matters in comparing the complexity of algorithms?
  - We only care about the behavior for ***large*** problems. Even bad algorithms can be used to solve small problems.
  - Ignore implementation details such as loop counter incrementation, etc. We can straight-line any loop.
- For functions over numbers, we often need to know a rough measure of *how fast a function grows*.

## Orders of Growth(增长的阶)

- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes *larger* than $g(x)$ *in the limit* (for large enough values of $x$).
- Useful in engineering for showing that one design scales better or worse than another.
- **Motivation:**
  - Suppose you are designing a web site to process user data (*e.g.*, financial records).
  - Suppose database program A takes $f_A(n)=30n+8$ microseconds to process any $n$ records, while program B takes $f_B(n)=n^2+1$ microseconds to process the $n$ records.
  - Which program do you choose, knowing you'll want to support millions of users?

# Orders of Growth

- **Visualizing:**
  - On a graph, as you go to the right, the faster-growing function always eventually becomes the larger one...



## ORDER OF GROWTH

- **Concept:**
  - We say $f_A(n)=30n+8$ is *(at most) order n*, or $O(n)$. It is, at most, roughly *proportional* to *n*.
  - We say $f_B(n)=n^2+1$ is *order* $n^2$, or $O(n^2)$. It is (at most) roughly *proportional* to $n^2$.
  - Any function whose *exact* (tightest) order is $O(n^2)$ is faster-growing than any $O(n)$ function.
    - Later we will introduce $\Theta$ for expressing *exact* order.
  - For large numbers of user records, the exactly order $n^2$ function will always take more time.

## The Big-O Notation

- **Definition:**
  - Let *f* and *g* be functions from N or R to R. Then *g* *asymptotically dominates (渐进地支配)* *f*, *denoted 'f is O(g)' or 'f is big-O of g'* iff

    $\exists k \exists c \, \forall n (n > k \rightarrow |f(n)| \le c|g(n)|)$
  - *'f is at most order g'*, or *'f is O(g)'* or *'f =O(g)'* all just mean that $f \in O(g)$.

*实际上相当于给出函数的渐进上界*

- **Note:**
  - Choose ***k***, choose ***c*** which may depend on the choice of *k*.
  - Once you choose ***k*** and ***c***, you must prove the truth of the ***implication*** (often by induction).

***k, c : witnesses(凭证) to the relationship f(x) is O(g(x))***

## The Big-O Notation

- **About the definition:**
  - Note that *f* is O(*g*) so long as *any* values of *k* and *c* exist that satisfy the definition.
  - But: The particular *k*, *c*, values that make the statement true are *not* unique: **Any larger value of *c* and/or *k* will also work.**
  - You are **not** required to find the smallest *c* and *k* values that work. (Indeed, in some cases, there may be no smallest values!)

***However, you should prove that the values you choose do work.***

## The Big-O Notation

- **Example:**

The function $f(n) = \frac{1}{2}n^3 + \frac{1}{2}n^2$ is $O(n^3)$.

To see this, consider

$$\frac{1}{2}n^3 + \frac{1}{2}n^2 \le \frac{1}{2}n^3 + \frac{1}{2}n^3, \quad \text{if } n \ge 1$$

Thus,

$$\frac{1}{2}n^3 + \frac{1}{2}n^2 \le 1 \cdot n^3, \quad \text{if } n \ge 1$$

Choosing 1 for $c$ and 1 for $k$, we have shown that

$|f(n)| \le c \cdot |n^3|$ for all $n \ge 1$ and $f$ is $O(n^3)$

## The Little-o of g

- **Definition:**
  - In calculus
  - If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

  - Then

*f* is *o*(*g*) (called *little-o* of *g*)

## The Little-o of g

- **Theorem**
  - If $f$ is $o(g)$ then $f$ is $O(g)$.
- **Proof**:
  - By definition of limit as $n$ goes to infinity, $f(n)/g(n)$ gets arbitrarily small.
  - That is for any $\varepsilon > 0$, there must be an integer $N$ such that when $n > N$, $| f(n)/g(n) | < \varepsilon$, so $| f(n)| < \varepsilon|g(n)|$
  - Hence, choose $c = \varepsilon$ and $k = N$.

$$o(g) :\equiv \{f \mid \forall c>0 \exists k\ \forall x(x>k \rightarrow |f(x)| < c|g(x)|)\}$$

- Q. E. D.

## The Little-o of g

- **Example:**
  - Show $3n + 5$ is $O(n^2)$
- **Proof:**
  - It's easy to show

$$\lim_{n\to\infty} \frac{3n+5}{n^2} = 0$$

  - using the theory of limits.
  - Hence $3n + 5$ is $o(n^2)$ and so it is $O(n^2)$.

- Q. E. D.

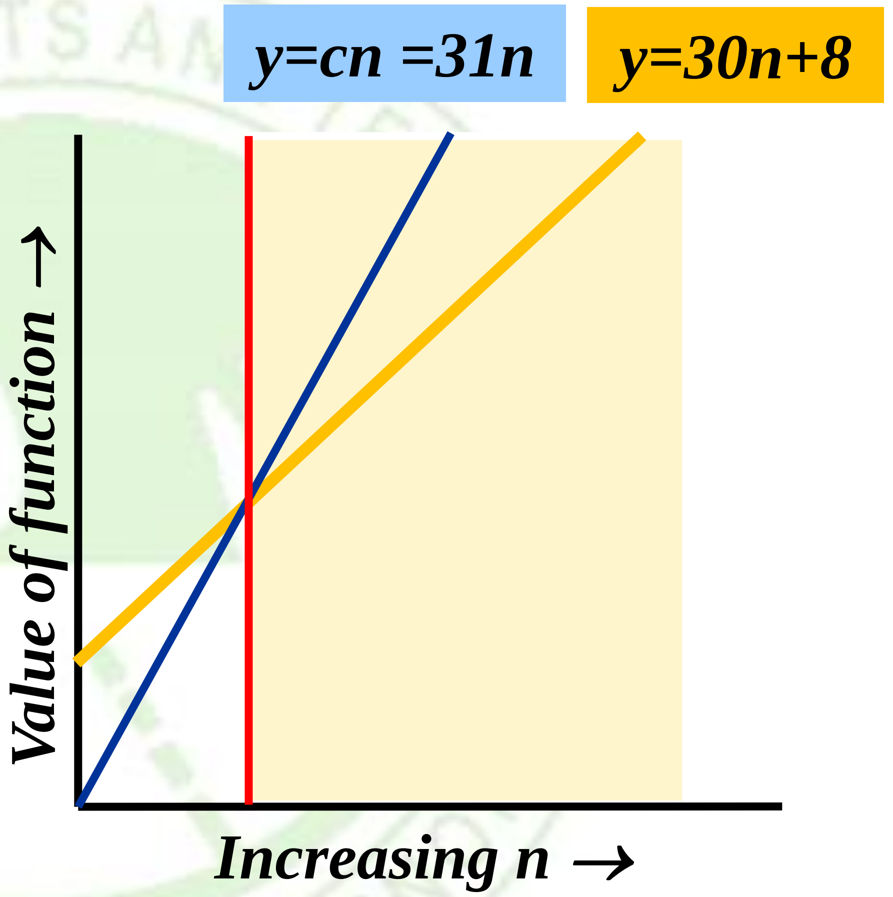
## Big-O Proof Examples

- Show that $n^2+1$ is $O(n^2)$.
  - Show $\exists c,k: \forall n>k: n^2+1 \le cn^2$.
    - $n^2+1< n^2+n^2 = 2n^2$ $(n>1)$
    - Let $c$=2, $k$=1. Assume $n$>k, then $n^2+1< cn^2$

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c,k: \forall n>k: 30n+8 \le cn$.
    - $30n+8 < 30n + n = 31n = cn$. (n>8)
    - Let $c$=31, $k$=8. Assume $n>k$=8, then $30n+8 < cn$

# Big-O example, graphically

- Note $30n+8$ isn't less than $n$ *anywhere* ($n>0$).
- It isn't even less than $31n$ everywhere to the left of n=8.
- But it *is* less than $31n$ everywhere to the right of $n=8$.



**$\exists c,k$: $\forall n>k$: $30n+8 \le cn$, thus $30n+8 \in O(n)$**

## Some important Big-O results

- **Theorem 1**
  - Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ ,where $a_0$ , $a_1$ , … $a_{n-1}$, $a_n$ are real numbers. Then $f(x)$ is $O(x^n)$ .

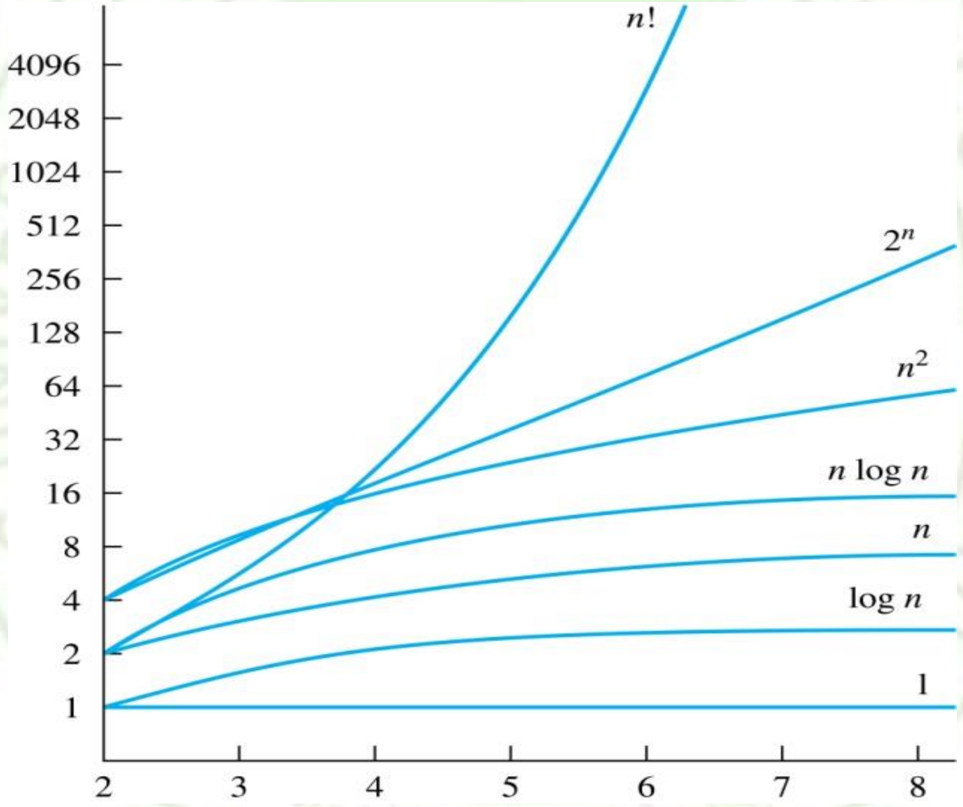
***Proof:*** *Using the triangle inequality to show* $|f(x)| \leq c|x^n|$.

***The leading term of a polynomial dominates its growth.***

- **Important results:**
  - $n!$ is $O(n^n)$ (Example 6)
  - $\log n!$ is $O(n\log n)$ (Example 6)
  - $\log n$ is $O(n)$ (Example 7)

## Some important Big-O results

- **Display of Growth of Functions**
  - ***1, logn, n, nlogn, $n^2$, $2^n$, n!***



***Note the difference in behavior of functions as n gets larger.***

## Useful Big-O Estimates

- If $d > c > 1$, then

  $n^c$ is $O(n^d)$, but $n^d$ is not $O(n^c)$.

- If $b > 1$ and $c$, $d$ are positive, then

  $(\log_b n)^c$ is $O(n^d)$, but $n^d$ is not $O((\log_b n)^c)$.

- If $b > 1$ and $d$ is positive, then

  $n^d$ is $O(b^n)$, but $b^n$ is not $O(n^d)$.

- If $c > b > 1$, then

  $b^n$ is $O(c^n)$, but $c^n$ is not $O(b^n)$.

- If $c > 1$, then

  $c^n$ is $O(n!)$, but $n!$ is not $O(c^n)$.

**原则：**增长慢的可以向增长快的估计，但增长快的不能向增长慢的估计

**应用：**可以对不同函数的增长速度进行排序

## Ordering by Order of Growth

- Put the functions below in order so that each function is big-O of the next function on the list.
- $f_1(n) = (1.5)^n$
- $f_2(n) = 8n^3+17n^2 +111$
- $f_3(n) = (\log n )^2$
- $f_4(n) = 2^n$
- $f_5(n) = \log (\log n)$
- $f_6(n) = n^2 (\log n)^3$
- $f_7(n) = 2^n (n^2 +1)$
- $f_8(n) = n^3+ n(\log n)^2$
- $f_9(n) = 10000$
- $f_{10}(n) = n!$

***Successively find the function that grows slowest among all those left on the list.***

- *$f_9(n) = 10000$ (constant, does not increase with n)*
- *$f_5(n) = \log (\log n)$ (grows slowest of all the others)*
- *$f_3(n) = (\log n)^2$ (grows next slowest)*
- *$f_6(n) = n^2 (\log n)^3$ ($(\log n)^3$ factor smaller than any power of n)*
- *$f_8(n) = n^3+ n(\log n)^2$ (tied with the one below)*
- *$f_2(n) = 8n^3+17n^2 +111$ (tied with the one above)*
- *$f_1(n) = (1.5)^n$ (an exponential function)*
- *$f_4(n) = 2^n$ (grows faster than one above since 2 > 1.5)*
- *$f_7(n) = 2^n (n^2 +1)$ (grows faster than above because of the $n^2 +1$ factor)*
- *$f_{10}(n) = n!$ ( n! grows faster than $c^n$ for every c)*

## Growth of combinations of functions

- **Theorem 2**
  - Suppose that $f_1$ is $O(g_1)$ and $f_2$ is $O(g_2)$ . Then $f_1 + f_2$ is $O(max\{ g_1, g_2\})$
  - **Proof:**

$$|f_1(x)|+|f_2(x)| \leq c_1|g_1(x)| + c_2|g_2(x)|$$

$$\leq c_1|g(x)|+c_2|g(x)| = (c_1+c_2)|g(x)| = c|g(x)|$$

$$where\ g(x)=max(g_1,g_2),\ c=c_1+c_2,\ k=max\{k_1, k_2\}$$

- **Corollary 1**
  - If $f_1$ , $f_2$ are both $O(g)$ then $f_1 + f_2$ is $O(g)$.

## Growth of combinations of functions

- **Theorem 3**
  - Suppose that $f_1$ is $O(g_1)$ and $f_2$ is $O(g_2)$ .Then $f_1f_2$ is $O(g_1g_2)$
- **Proof:**
  - There is a $k_1$ and $c_1$ such that
    - 1. $f_1(n) \leq c_1g_1(n)$, when $n > k_1$.
  - There is a $k_2$ and $c_2$ such that
    - 2. $f_2(n) \leq c_2g_2(n)$, when $n > k_2$.
  - We must find a $k_3$ and $c_3$ such that
    - 3. $f_1(n)f_2(n) \leq c_3g_1(n)g_2(n)$, when $n > k_3$.

## Growth of combinations of functions

- **Proof (cont) :** ***$f_1 f_2$* is *O*(g$_1$g$_2$)**
  - We use the inequality
    - if $0 < a < b$ and $0 < c < d$ then $ac < bd$
  - to conclude that
    - $f_1(n)f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$
  - as long as $k > max\{k_1, k_2\}$ so that *both* inequalities 1 and 2 hold at the same time.
  - Therefore, choose
    - $c_3 = c_1 c_2$
    - $k_3 = max\{k_1, k_2\}$.

- Q. E. D.

## Growth of combinations of functions

- **Example:**
  - Find the complexity class of the function (Give a big-O estimate for the function)
    - $(nn! + 3^{n+2} + 3n^{100})(n^n + n2^n)$
- **Solution:**
  - This means to simplify the expression.
  - Throw out stuff which you know doesn't grow as fast. And at last:
  - $O(nn! \cdot n^n)$

## Big-omega notation: Ω

- **Definition:**
  - Let *f* and *g* be functions from N or R to R. We say *f* is Ω(*g*) or '*f* is big- Ω of *g*,' iff

    $\exists k \exists c\ \forall n[n > k \rightarrow |f(n)| \geq c|g(n)|]$

- **Note:**

  *实际上相当于给出函数的渐进下界*

  - Choose *k*
  - Choose *c*; it may depend on your choice of *k*
  - Once you choose *k* and *c*, you must prove the truth of the implication (often by induction)

***In particular, f(x) is Ω(g(x)) if and only if g(x) is O(f(x)).***

## Big-Theta notation: Θ

- **Definition:**
  - If $f \in O(g)$ and $g \in O(f)$, then we say "*g and f are of the same order*" or "*f is (exactly) of order g*" and write $f \in \Theta(g)$.
  - $f \in \Theta(g)$ iff $f$ and $g$ have same order.

*互为渐进上界*

- **Another equivalent definition:**

***f∈Ω(g)  and f∈O(g)***

$$\Theta(g) \equiv \{f: \mathbf{R} \to \mathbf{R} \mid \exists c_1 c_2 k > 0\ \forall x > k: |c_1 g(x)| \le |f(x)| \le |c_2 g(x)|\ \}$$

  - Everywhere beyond some point $k$, $f(x)$ lies in between two multiples of $g(x)$.

***通过上界和下界确定了函数的阶***
***f(x)和 g(x)具有相同的阶***

***f∈Θ(g) and g∈Θ(f)***

## Big-Theta notation: Θ

- **Example:**
  - Determine whether: $\left(\sum_{i=1}^{n} i\right) \overset{?}{\in} \Theta(n^2)$
- **Quick solution:**

$$\left(\sum_{i=1}^{n} i\right) = n(n-1)/2$$

$$= n \cdot \Theta(n)/2$$

$$= n \cdot \Theta(n)$$

$$= \Theta(n^2)$$

## Big-Theta notation: Θ

- **Theorem 4**
  - Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ ,where $a_0, a_1, \ldots a_{n-1}, a_n$ are real numbers with $a_n \neq 0$ . Then $f(x)$ is of order $x^n$ .

***Method:***
*Show f(x) is O($x^n$) and f(x) is Ω($x^n$).*

***Remember:***
*Leading term of a polynomial determines its **order**.*

## Review: Orders of Growth

- Definitions of order-of-growth sets, $\forall g:\mathbf{R}\to\mathbf{R}$
  - $O(g) :\equiv \{f \mid \underline{\exists\, \mathbf{c}}>0\ \exists k\ \forall x>k\ |f(x)| \le |cg(x)|\}$
  - $o(g) :\equiv \{f \mid \underline{\forall \mathbf{c}}>0\ \exists k\ \forall x>k\ |f(x)| \le |cg(x)|\}$
  - $\Omega(g) :\equiv \{f \mid \exists\, c>0\ \exists k\ \forall x>k\ |f(x)| \ge |cg(x)|\}$
  - $\Omega(g) :\equiv \{f \mid g \in O(f)\}$
  - $\Theta(g) \equiv \{f \mid \exists c_1 c_2 k>0\ \forall x>k\ |c_1 g(x)| \le |f(x)| \le |c_2 g(x)|\}$
  - $\Theta(g) :\equiv O(g) \cap \Omega(g)$

## 3.3 Complexity of Algorithmic

Wenjing Li

wjli@bupt.edu.cn

School of Computer Science

Beijing University of Posts & Telecommunications

## Algorithmic Complexity(复杂度)

- **Definition:**
  - The *algorithmic complexity* of a computation is, most generally, a *measure* of how *difficult* it is to perform the computation.
  - That is, it measures some aspect of the *cost* of computation (in a general sense of “cost”).
  - Amount of resources required to do a computation.
  - Some of the most common complexity measures:
    - “Time” complexity: # of operations or steps required
    - “Space” complexity: # of memory bits required

## Complexity Depends on Input

- Most algorithms have different complexities for inputs of different sizes.
  - *E.g.* searching a long list typically takes more time than searching a short one.
- Therefore, complexity is usually expressed as a *function of the input length* (*表示为输入长度的函数*).
  - This function usually gives the complexity for the *worst-case* input of any given length.

## Example 1: Max algorithm

- **Problem:**
  - Find the *simplest form* of the *exact* order of growth (Θ) of the *worst-case time complexity* (w.c.t.c.) of the *max* algorithm, assuming that each line of code takes some constant time every time it is executed (with possibly different times for different lines of code).

## Example 1: Max algorithm

- **Complexity analysis of *Max***

```
procedure max($a_1, a_2, \ldots, a_n$: integers)
    $v := a_1$                          $t_1$
    for $i := 2$ to $n$                  $t_2$
        if $a_i > v$ then $v := a_i$     $t_3$
    return $v$                          $t_4$
```

$\left.\begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{matrix}\right\}$ ***Times for each execution of each line.***

- First, what's an expression for the exact total worst-case time? (Not its order of growth.)

$$t(n) = t_1 + \left( \sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

## Example 1: Max algorithm

- **Complexity analysis of *Max* (Cont)**
  - Now, what is the simplest form of the exact (Θ) order of growth of *t(n)*?

$$t(n) = t_1 + \left( \sum_{i=2}^{n} (t_2 + t_3) \right) + t_4$$

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

$$= \Theta(1) + \Theta(n)\Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$$

## Example 2: Linear Search

- **Complexity analysis of *Linear Search***

```
procedure linear search ($x$: integer, $a_1, a_2, \ldots, a_n$: integers)
    $i := 1$                                  $t_1$
    while ($i \le n \wedge x \ne a_i$)          $t_2$
        $i := i + 1$                          $t_3$
    if $i \le n$ then $location := i$          $t_4$
    else $location := 0$                      $t_5$
    return $location$                         $t_6$
```

## Example 2: Linear Search

- **Complexity analysis of *Linear Search (cont)***
  - Worst case time complexity order:

$$t(n) = t_1 + \left( \sum_{i=1}^{n} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

  - Best case:

$$t(n) = t_1 + t_2 + t_4 + t_6 = \Theta(1)$$

  - Average case, if item is present:

$$t(n) = t_1 + \left( \sum_{i=1}^{n/2} (t_2 + t_3) \right) + t_4 + t_5 + t_6 = \Theta(n)$$

## Example 3: Binary Search

- **Complexity analysis of *Binary Search***

```
procedure binary search (x:integer, $a_1, a_2, \ldots, a_n$: distinct
    integers, sorted smallest to largest)
    $i := 1$
    $j := n$                                          } $\Theta(1)$
    while $i<j$ begin
      $m := \lfloor (i+j)/2 \rfloor$
      if $x>a_m$ then $i := m+1$ else $j := m$        } $\Theta(1)$
    end
    if $x = a_i$ then location := i else location := 0   } $\Theta(1)$
    return location
```

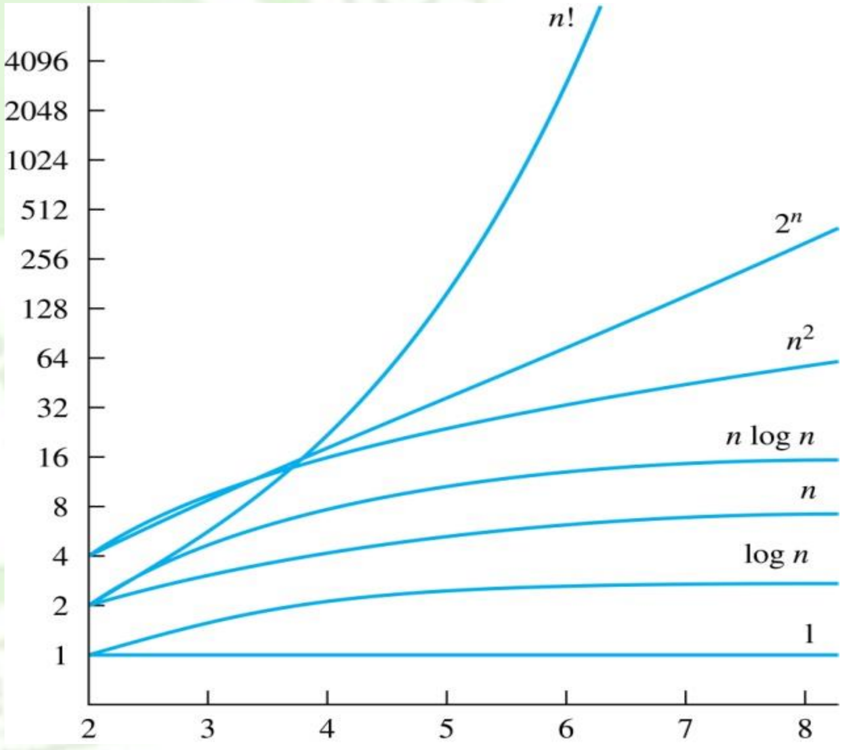
***Key question:***
***How many loop iterations?***

## Example 3: Binary Search

- **Complexity analysis of *Binary Search (cont)***
  - Suppose that $n$ is a power of 2, *i.e.*, $\exists k: n=2^k$.
  - Original range from $i=1$ to $j=n$ contains $n$ items.
  - Each iteration: Size $j-i+1$ of range is cut in half.
  - Loop terminates when size of range is $1=2^0$ ($i=j$).
  - Therefore, the number of iterations is:
    $$k = \log_2 n = \Theta(\log_2 n) = \Theta(\log n)$$
  - Even for $n \neq 2^k$ (not an integral power of 2), time complexity is still $\Theta(\log_2 n) = \Theta(\log n)$.

## Some orders of growth

| | |
|---|---|
| $\Theta(1)$ | Constant |
| $\Theta(\log_c n)$ | Logarithmic (same order $\forall c$) |
| $\Theta(\log n^c)$ | Polylogarithmic(With $c$ a constant.) |
| $\Theta(n)$ | Linear |
| $\Theta(n\log n)$ | Linear logarithmic |
| $\Theta(n^c)$ | Polynomial (for any $c$) |
| $\Theta(c^n)$ | Exponential (for $c$>1) |
| $\Theta(n!)$ | Factorial |



# Computer Time Examples

| *Ops(n)* | n=10 | n=$10^6$ |
|---|---|---|
| $log_2n$ | 3.3ns | 19.9ns |
| $n$ | 10ns | 1ms |
| $nlog_2n$ | 33ns | 19.9ms |
| $n^2$ | 100ns | 16m40s |
| $2^n$ | 1.024us | $10^{301,004,5}$ Year |
| $n!$ | 3.63ms | O.M.G |

- *Ops* is a function of *n*.
- Assume time = 1ns ($10^{-9}$ second) per op
- problem size = *n* bits

## Problem Complexity

- **Definition:**
  - The complexity of a computational *problem* or *task* is (the order of growth of) the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
  - *E.g.* the problem of searching an ordered list has *at most logarithmic* time complexity. (Complexity is O(log *n*).)

问题复杂度指解决该问题或执行该任务的复杂度增长最低阶的算法的复杂度

## Tractable vs. Intractable

- **Definition:**
  - A problem or algorithm with at most polynomial time complexity is considered *tractable* (or *feasible*).
  - **P** (*Polynomial*) problem is the set of all tractable problems.
  - A problem or algorithm that has complexity greater than polynomial is considered *intractable* (or *infeasible*).
- **Note:**
  - $n^{1,000,000}$ is *technically* tractable, but really very hard.
  - $n^{\log\log\log n}$ is *technically* intractable, but easy.
  - Such cases are rare though.

# P vs. NP

- **Definition:**
  - **NP** (*Nondeterministic **P**olynomial time*) is the set of problems for which there exists a tractable algorithm for *checking a proposed solution* to tell if it is correct.
    - E.g. Eight Queen problem, Sudoku Problem, TSP …..
  - We know that **P⊆NP.**
  - **But** the most famous unproven conjecture in computer science is that this inclusion is *proper*.
  - *i.e.*, that **P⊂NP** rather than **P=NP**.
- Whoever first proves or disprove this will be famous!

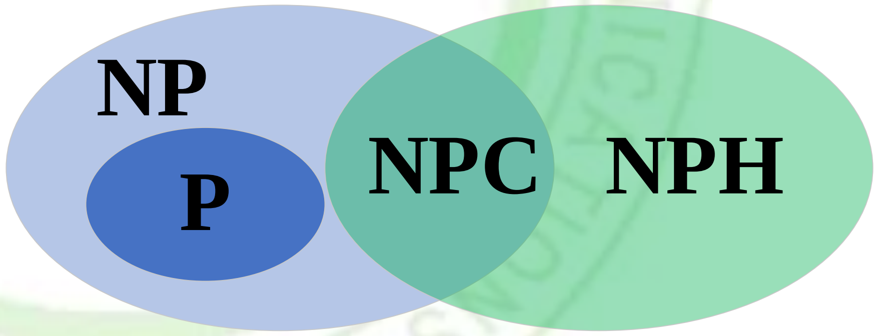
世界七大数学难题之首: ***NP=P?***

## NP-Complete Problem

- **Definition:**
  - NP-Complete problems (NP-C): If **some of NP** problems can be solved by a polynomial worst-case time algorithm, then all problems in the class NP can be solved by polynomial worst-case time algorithms. (所有NP问题都可以在多项式时间内**规约**为某些NP问题，这些问题的复杂性与所有NP问题的复杂性相关联，若这些问题存在多项式时间求解算法，则所有NP问题都是多项式时间可解的，这些问题被称为NP-完全问题)
  - **NPC满足两个条件：**
    - 是一个NP问题。
    - 其他NP问题能够在多项式时间内规约（转化）为该问题（该问题的复杂度大于原NP问题）。
  - **NPC示例：**可满足性问题、旅行商问题TSP、汉密尔顿回路问题等。

## NP-Hard Problem

- **Definition:**
  - **NP-hard problem (NP-H):** NP问题能够在多项式时间内规约（转化）为某个问题X，X的复杂度一般大于原NP问题，若X是一个NP问题，则称X是NPC的，否则X是NP-Hard的。
  - NPH问题满足NPC问题的第2个条件但不满足第1个条件，也就是说NPH不一定是NP问题，NPH问题比NPC问题范围广，更难以解决。



假设P≠NP

## Unsolvable problems

- Turing discovered in the 1930's that there are problems unsolvable by *any* algorithm.
  - Or equivalently, there are undecidable yes/no questions, and uncomputable functions.
- Classic example: the *halting problem*.
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an "*infinite loop*?"

## The Halting Problem (Turing'36)

- The ***halting problem*** was the first mathematical function proven to have *no* algorithm that computes it!
  - We say, it is ***uncomputable***.
- The desired function is *Halts*(*P*, *I*) :≡ the truth value of this statement:
  - *"Program P, given input I, eventually terminates."*
- **Theorem: *Halts* is uncomputable!**
  - *I.e.*, there does *not* exist *any* algorithm *A* that computes *Halts* correctly for *all* possible inputs.
  - Its proof is thus a *non*-existence proof.

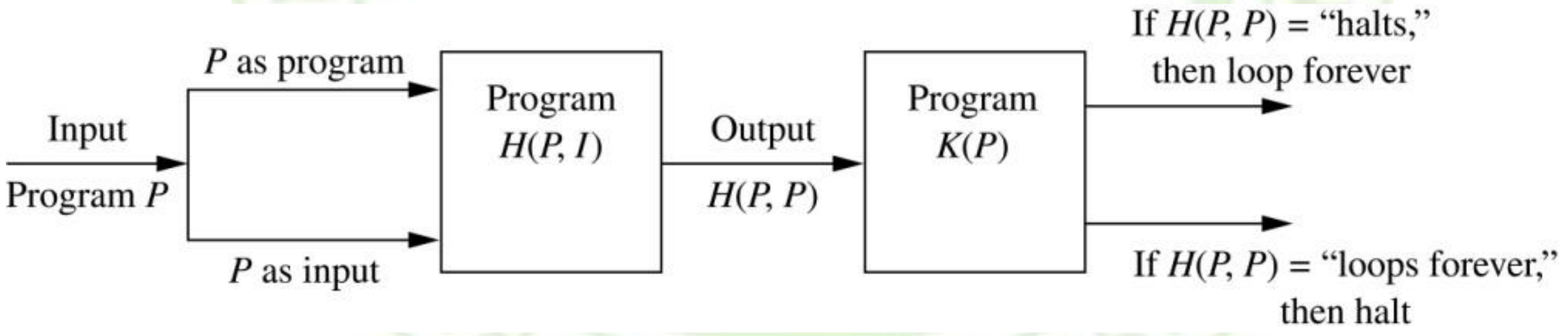
***Alan Turing***
***1912-1954***

## The Halting Problem

- **Problem description:**
  - Develop a procedure that takes as input a computer program along with its input and determines whether the program will eventually halt with that input.
  - Assume that there is such a procedure and call it *H*(*P*, *I*).
  - The procedure *H*(*P*, *I*) takes as input a program *P* and the input *I* to *P*.
  - H outputs “halt” if it is the case that *P* will stop when run with input *I*. Otherwise, *H* outputs “loops forever.”

## The Halting Problem

- **Solution**: Proof by contradiction
  - Since a program is a string of characters, we can call *H*(*P*, *P*).
  - Construct a procedure *K*(*P*), which works as follows.
  - If *H*(*P*,*P*) outputs "loops forever" then *K*(*P*) halts.
  - If *H*(*P*,*P*) outputs "halt" then *K*(*P*) goes into an infinite loop, printing "ha" on each iteration.


Input
Program P
P as program
P as input
Program H(P, I)
Output
H(P, P)
Program K(P)
If H(P, P) = "halts," then loop forever
If H(P, P) = "loops forever," then halt

## The Halting Problem

- **Solution**: Proof by contradiction (cont)
  - Now we call *K* with *K* as input, i.e. *K*(*K*).
    - If the output of *H*(*K*,*K*) is “loops forever” (means *K(K)* loops forever), then *K*(*K*) halts. **Contradiction.**
    - If the output of *H*(*K*,*K*) is “halts” (means *K(K)* halts), then *K*(*K*) loops forever. **Contradiction.**
  - Therefore, there can not be a procedure that can decide whether or not an arbitrary program halts. The halting problem is unsolvable.
- **Corollary:**
  - General impossibility of predictive analysis of arbitrary computer programs (对任意计算机程序进行预测分析一般是不可能的).

## Important Complexity Classes

- The most common orders in computer science applications are
  - $\Theta(1)$, $\Theta(logn)$, $\Theta(n)$ , $\Theta(nlogn)$, $\Theta(n^2)$ , $\Theta(n^3)$
  - $\Theta(2^n)$, $\Theta(n!)$, $\Theta(n^n)$
- Some Terms

| | |
|---|---|
| constant time | $\Theta(1)$ |
| logarithmic time | $\Theta(log\ n)$ |
| linear time | $\Theta(n)$ |
| quadratic time | $\Theta(n^2)$ |
| polynomial time | $\Theta(n^c)$ |
| exponential time | $\Theta(c^n)$ |
| Factorial time | $\Theta(n!)$ |

## Rules for determining the $\Theta$-Class

1. $\Theta(1)$ functions are constant and have zero growth.
2. $\Theta(log(n))$ is lower than $\Theta(n^k)$ if $k>0$.
3. $\Theta(n^a)$ is lower than $\Theta(n^b)$ if and only if $0<a<b$.
4. $\Theta(a^n)$ is lower than $\Theta(b^n)$ if and only if $0<a<b$.
5. $\Theta(n^k)$ is lower than $\Theta(a^n)$ for any power of $n^k$ and any $a>1$.
6. $\Theta(a^n)$ is lower than $\Theta(n!)$ for any $a>1$.
7. $\Theta(n!)$ is lower than $\Theta(n^n)$
8. If $r$ is not zero, then $\Theta(rf) = \Theta(f)$ for any function $f$.
9. If $h$ is a non zero function and $\Theta(f)$ is lower than (or same order as) $\Theta(g)$, then $\Theta(fh)$ is lower than (or same order as) $\Theta(gh)$.
10. If $\Theta(f)$ is lower than $\Theta(g)$, then $\Theta(f+g)= \Theta(g)$.

## Review of § 3.3

- Algorithmic complexity = *cost* of computation.
- Focus on *time* complexity for our course.
  - Although space & energy are also important.
- Characterize complexity as a function of input size:
  - Worst-case, best-case, or average-case.
- How to analyze the order of growth for simple algorithms？
- Use orders-of-growth notation to concisely summarize the growth properties of complexity functions.
- Complexity of problem: *P*, *NP*, *NPC*, *NPH*

## Homework

- **§ 3.2**
  - 34, 74

- **§ 3.3**
  - 36