Department of
Computer Science and Engineering
计算机科学与工程系

CS215 DISCRETE MATH

Dr. Qi Wang

Department of Computer Science and Engineering
Office: Room413, CoE South Tower
Email: wangqi@sustech.edu.cn

Cardinality of Sets

- Recall: the cardinality of a finite set is defined by the number of the elements in the set.

- The sets A and B have *the same cardinality* if there is a one-to-one correspondence between elements in A and B.

- A set that is **either finite** or **has the same cardinality as the set of positive integers** $\mathbb{Z}^+$ is called *countable*. A set that is **not countable** is called *uncountable*.

  Why are these called **countable**?

  ◇ The elements of the set can be enumerated and listed.

- **Theorem**

  The set $\mathcal{P}(\mathbb{N})$ is uncountable.

Proof by contradiction:

Assume that $\mathcal{P}(\mathbb{N})$ is countable. This implies that the elements of this set can be listed as $S_0, S_1, S_2, \ldots$, where $S_i \subseteq \mathbb{N}$, and each S_i can be represented uniquely by the bit string $b_{i0}b_{i1}b_{i2}\ldots$, where $b_{ij} = 1$ if $j \in S_i$ and $b_{ij} = 0$ if $j \notin S_i$

- $S_0 = b_{00}b_{01}b_{02}b_{03}\cdots$
- $S_1 = b_{10}b_{11}b_{12}b_{13}\cdots$
- $S_2 = b_{20}b_{21}b_{22}b_{23}\cdots$

$\vdots$

all $b_{ij} \in \{0, 1\}$.

Computable vs Uncomputable

- **Definition**

  We say that a function is *computable* if there is a computer program in some programming language that finds the values of this function. If a function is not computable, we say it is *uncomputable*.

Computable vs Uncomputable

- **Definition**

  We say that a function is *computable* if there is a computer program in some programming language that finds the values of this function. If a function is not computable, we say it is *uncomputable*.

Theorem*

There are functions that are not computable.

Definition

We say that a function is *computable* if there is a computer program in some programming language that finds the values of this function. If a function is not computable, we say it is *uncomputable*.

Theorem*

There are functions that are not computable.

Proof.

(1) prove that the set of computer programs is *countably infinite* (Example 5)

(2) prove that the number of functions is *uncountable*

Computable vs Uncomputable

Definition

We say that a function is *computable* if there is a computer program in some programming language that finds the values of this function. If a function is not computable, we say it is *uncomputable*.

Theorem*

There are functions that are not computable.

Proof.

(1) prove that the set of computer programs is *countably infinite* (Example 5)

(2) prove that the number of functions is *uncountable*

The set of functions from $\mathbb{Z}^+$ to the set $\{0, 1, 2, \ldots, 9\}$ is *uncountable*. Proof?

Cantor's Theorem

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Cantor's Theorem

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Proof.

(1) $|S| \leq |\mathcal{P}(S)|$?

Cantor's Theorem

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Proof.

(1) $|S| \leq |\mathcal{P}(S)|$?

(2) $|S| \neq |\mathcal{P}(S)|$

Cantor's Theorem

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Proof.

(1) $|S| \leq |\mathcal{P}(S)|$?
(2) $|S| \neq |\mathcal{P}(S)|$
We only need consider the case that $S \neq \emptyset$?

Cantor's Theorem

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Proof.

(1) $|S| \leq |\mathcal{P}(S)|$?
(2) $|S| \neq |\mathcal{P}(S)|$
We only need consider the case that $S \neq \emptyset$?
Proof by contradiction.
There is a bijective function f from S to $\mathcal{P}(S)$.

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

  Proof.

  (1) $|S| \leq |\mathcal{P}(S)|$?
  (2) $|S| \neq |\mathcal{P}(S)|$
  We only need consider the case that $S \neq \emptyset$?
  Proof by contradiction.
  There is a bijective function f from S to $\mathcal{P}(S)$.
  Consider the set $T = \{s \in S | s \notin f(s)\}$. Note that $T \neq \emptyset$.

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

Proof.

(1) $|S| \leq |\mathcal{P}(S)|$?

(2) $|S| \neq |\mathcal{P}(S)|$

We only need consider the case that $S \neq \emptyset$?

Proof by contradiction.

There is a bijective function f from S to $\mathcal{P}(S)$.

Consider the set $T = \{s \in S | s \notin f(s)\}$. Note that $T \neq \emptyset$.

Now f is bijective, and T is a subset of S, so there is an element $s_0 \in S$ s.t. $f(s_0) = T$.

- **Theorem***

  If S is a set, then $|S| < |\mathcal{P}(S)|$.

  Proof.

  (1) $|S| \leq |\mathcal{P}(S)|$?
  (2) $|S| \neq |\mathcal{P}(S)|$
  We only need consider the case that $S \neq \emptyset$?
  Proof by contradiction.
  There is a bijective function f from S to $\mathcal{P}(S)$.
  Consider the set $T = \{s \in S | s \notin f(s)\}$. Note that $T \neq \emptyset$.
  Now f is bijective, and T is a subset of S, so there is an element $s_0 \in S$ s.t. $f(s_0) = T$.
  $\mathcal{Q}$: Is $s_0 \in T$?

Algorithms

- An *algorithm* is a finite sequence of **precise instructions** for performing a computation or for solving a problem.

Abu Ja'far Mohammed ibn Musa al-Khowarizmi

Big-O Notation

- Which function is "bigger"?

  $\frac{1}{10}n^2$ or $100n + 10000$

Big-O Notation

- Which function is "bigger"?

  $\frac{1}{10}n^2$ or $100n + 10000$

  It depends on the value of n.

Big-O Notation

- Which function is "bigger"?

  $\frac{1}{10}n^2$ or $100n + 10000$

  It depends on the value of n.

  In Computer Science, we are usually interested in what happens when our problem input size gets large.

Big-O Notation

- Which function is "bigger"?

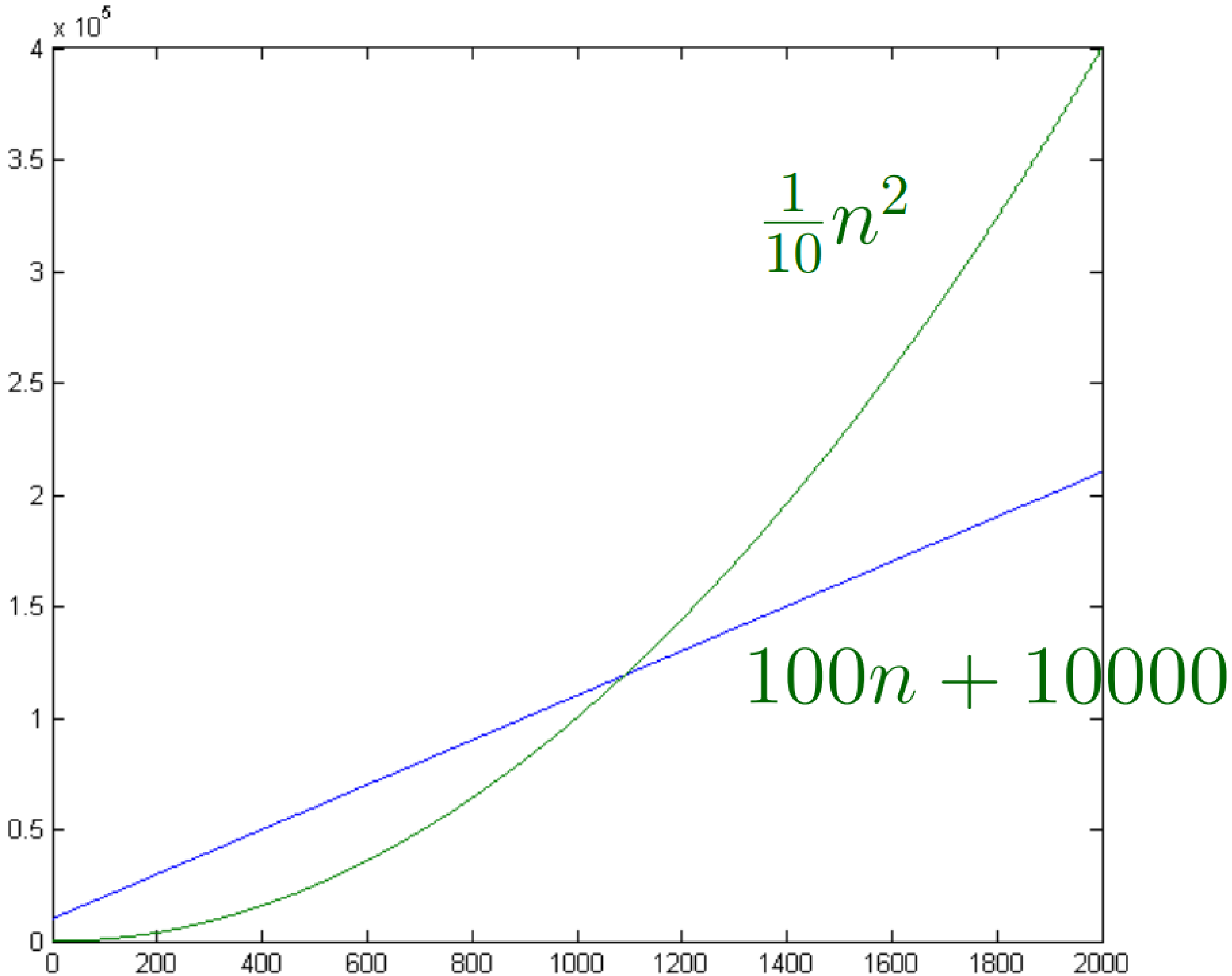
  $\frac{1}{10}n^2$ or $100n + 10000$

  It depends on the value of n.

  In Computer Science, we are usually interested in what happens when our problem input size gets large.

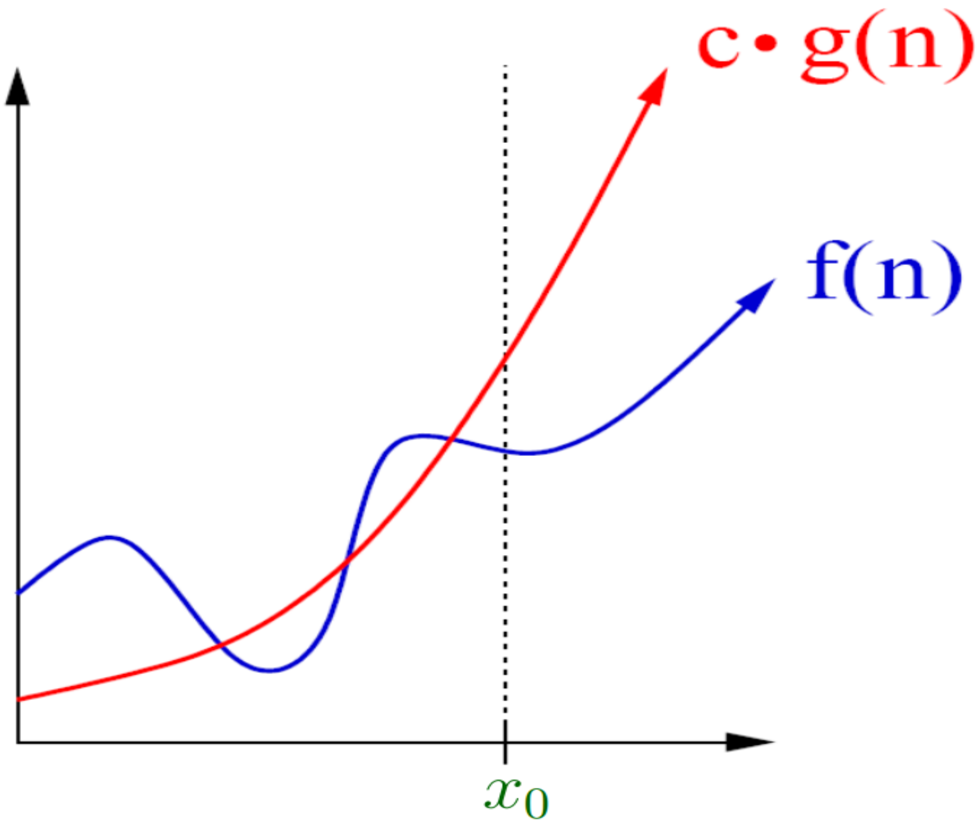
  Notice that when n is "large enough", $\frac{1}{10}n^2$ gets much bigger than $100n + 10000$ and stays larger.

Big-O Notation



Big-O Notation

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(n) = O(g(n))$ (reads: $f(n)$ is O of $g(n)$), if there exist some positive constants C and x_0 such that $|f(n)| \leq C|g(n)|$, whenever $n > x_0$.



Big-O Notation

- $\frac{1}{10}n^2$ and $100n + 10000$

Big-O Notation

- $\frac{1}{10}n^2$ and $100n + 10000$

  Let $k = 1091$

  Can verify that $\forall n \geq k$, $100n + 10000 \leq \frac{1}{10}n^2$

  Thus, $100n + 10000 = O(\frac{1}{10}n^2)$

Big-O Notation

- $\frac{1}{10}n^2$ and $100n + 10000$

  Let $k = 1091$

  Can verify that $\forall n \geq k$, $100n + 10000 \leq \frac{1}{10}n^2$

  Thus, $100n + 10000 = O(\frac{1}{10}n^2)$

Note that the opposite is **not** true! (Why?)

Big-O Notation

- $\frac{1}{10}n^2$ and $100n + 10000$

  Let $k = 1091$

  Can verify that $\forall n \geq k$, $100n + 10000 \leq \frac{1}{10}n^2$

  Thus, $100n + 10000 = O(\frac{1}{10}n^2)$

Note that the opposite is **not** true! (Why?)

(Proof by contradiction)

Big-O Notation

- $\frac{1}{10}n^2$ and $100n + 10000$

  Let $k = 1091$

  Can verify that $\forall n \geq k$, $100n + 10000 \leq \frac{1}{10}n^2$

  Thus, $100n + 10000 = O(\frac{1}{10}n^2)$

  Note that the opposite is **not** true! (Why?)

  (Proof by contradiction)

  Examples

  $4n^2$
  $8n^2 + 2n - 3$
  $n^2/5 + \sqrt{n} - 10 \log n$
  $n(n-3)$

  are all $O(n^2)$

Big-O Estimates for Polynomials

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

Big-O Estimates for Polynomials

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

  Proof:

  Assuming $x > 1$, we have

$$\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0| \\
&\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0| \\
&= x^n(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n) \\
&\leq x^n(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|).
\end{aligned}$$

Big-O Estimates for Polynomials

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

Proof:

Assuming $x > 1$, we have

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0| \\
&\leq |a_n|x^n + |a_{n-1}|x^{n-1} + \cdots + |a_1|x + |a_0| \\
&= x^n(|a_n| + |a_{n-1}|/x + \cdots + |a_1|/x^{n-1} + |a_0|/x^n) \\
&\leq x^n(|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|).
\end{aligned}
$$

The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-O Estimates for Some Functions

- $1 + 2 + \cdots + n = O(n^2)$

  $n! = O(n^n)$

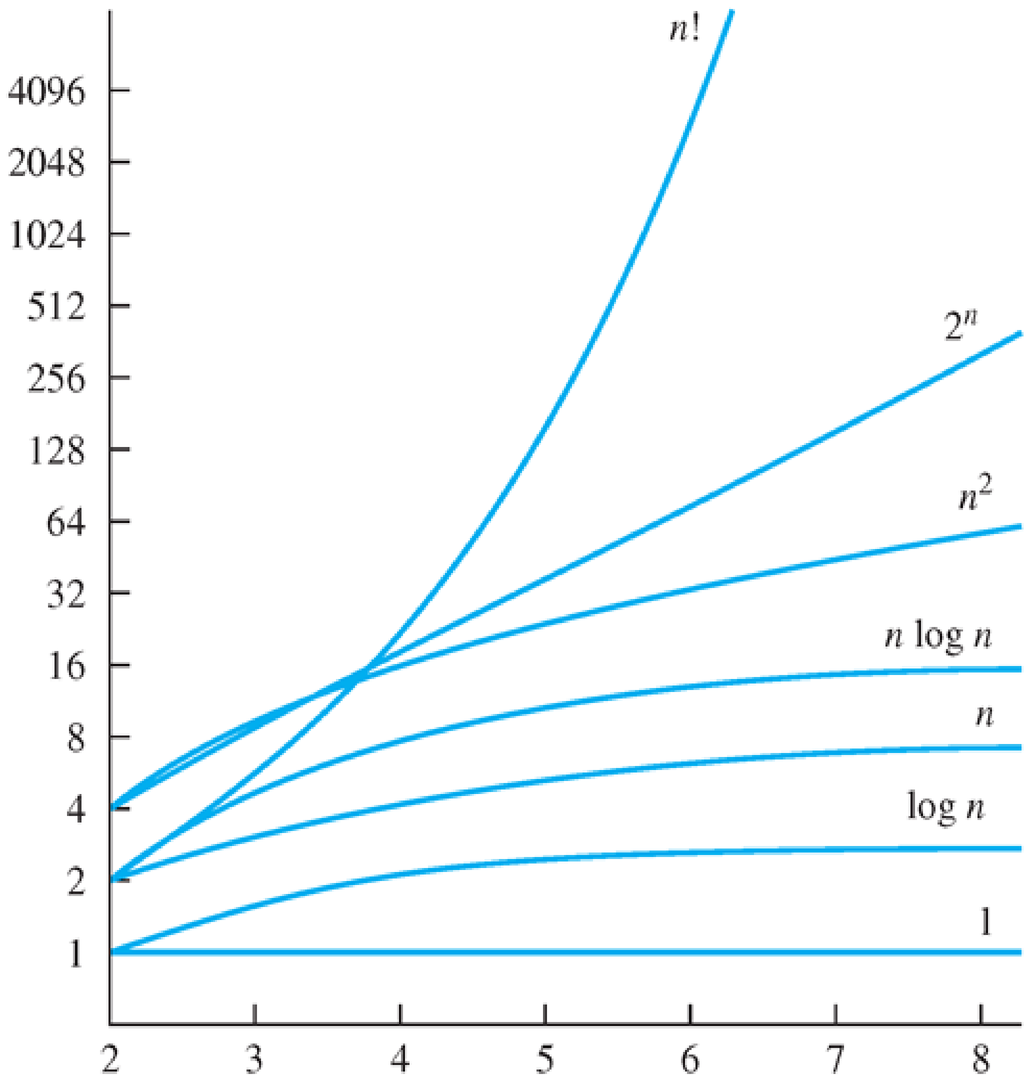
  $\log n! = O(n \log n)$

  $\log_a n = O(n)$ for an integer $a \geq 2$

  $n^a = O(n^b)$ for integers $a \leq b$

  $n^a = O(2^n)$ for an integer a

Display of Growth of Functions



Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then
  $(f_1 + f_2)(x) = O(\max(|g_1(x)|, |g_2(x)|))$

Proof:

By definition, there exist constants C_1, C_2, k_1, k_2 such that
$|f_1(x)| \leq C_1|g_1(x)|$ when $x > k_1$ and
$|f_2(x)| \leq C_2|g_2(x)|$ when $x > k_2$. Then

$$\begin{aligned}
|(f_1 + f_2)(x)| &= |f_1(x) + f_2(x)| \\
&\leq |f_1(x)| + |f_2(x)| \\
&\leq C_1|g_1(x)| + C_2|g_2(x)| \\
&\leq C_1|g(x)| + C_2|g(x)| \\
&= (C_1 + C_2)|g(x)| \\
&= C|g(x)|,
\end{aligned}$$

where $g(x) = \max(|g_1(x)|, |g_2(x)|)$ and $C = C_1 + C_2$.

Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then
  $(f_1 f_2)(x) = O(g_1(x) g_2(x))$

Proof:

When $x > \max(k_1, k_2)$,

$$\begin{aligned} |(f_1 f_2)(x)| &= |f_1(x)||f_2(x)| \\ &\leq C_1|g_1(x)|C_2|g_2(x)| \\ &\leq C_1 C_2|(g_1 g_2)(x)| \\ &\leq C|(g_1 g_2)(x)|, \end{aligned}$$

where $C = C_1 C_2$.

Ordering Functions by Order of Growth

- $f_1(n) = (1.5)^n$
  $f_2(n) = 8n^3 + 17n^2 + 111$
  $f_3(n) = (\log n)^2$
  $f_4(n) = 2^n$
  $f_5(n) = \log(\log n)$
  $f_6(n) = n^2(\log n)^3$
  $f_7(n) = 2^n(n^2 + 1)$
  $f_8(n) = n^3 + n(\log n)^2$
  $f_9(n) = 100000$
  $f_{10}(n) = n!$

Big-Omega Notation

- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(n) = \Omega(g(n))$ (reads: $f(n)$ is Ω of $g(n)$), if there exist some positive constants C and x_0 such that $|f(n)| \geq C|g(n)|$, whenever $n > x_0$.

Big-Omega Notation

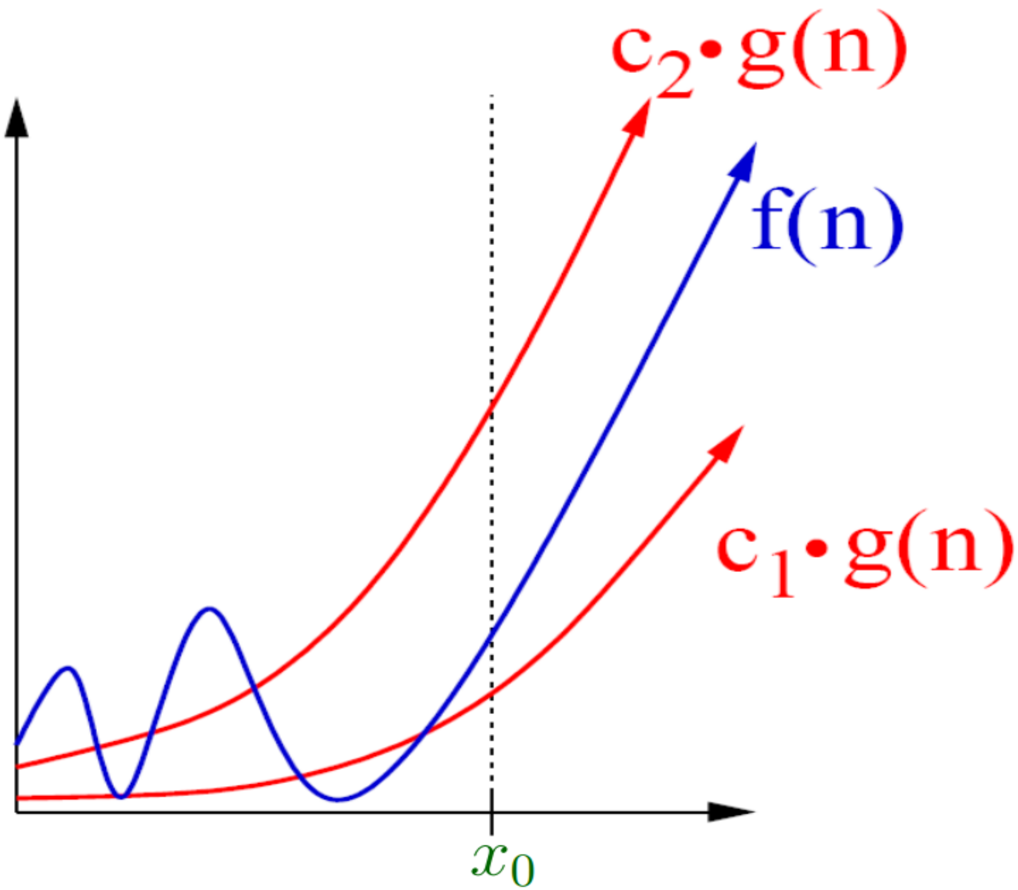
- Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(n) = \Omega(g(n))$ (reads: $f(n)$ is Ω of $g(n)$), if there exist some positive constants C and x_0 such that $|f(n)| \geq C|g(n)|$, whenever $n > x_0$.

Big-O gives an upper bound on the growth of a function, while Big-Ω gives a lower bound. Big-Ω tells us that a function grows at least as fast as another.

Note: $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$.

Big-Theta Notation (Big-O & Big-Omega)

- Two functions $f(n)$, $g(n)$ have the same order growth if $f(n) = O(g(n))$ and $g(n) = O(f(n))$. In this case, we say that $f(n) = \Theta(g(n))$, which is the same as $g(n) = \Theta(f(n))$.



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

- $3n^2 + 4n = \Theta(n)$?
  $3n^2 + 4n = \Theta(n^2)$?
  $3n^2 + 4n = \Theta(n^3)$?
  $n/5 + 10n \log n = \Theta(n^2)$?
  $n^2/5 + 10n \log n = \Theta(n \log n)$?
  $n^2/5 + 10n \log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$?   No

  $3n^2 + 4n = \Theta(n^2)$?

  $3n^2 + 4n = \Theta(n^3)$?

  $n/5 + 10n\log n = \Theta(n^2)$?

  $n^2/5 + 10n\log n = \Theta(n\log n)$?

  $n^2/5 + 10n\log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$? No

  $3n^2 + 4n = \Theta(n^2)$? Yes

  $3n^2 + 4n = \Theta(n^3)$?

  $n/5 + 10n \log n = \Theta(n^2)$?

  $n^2/5 + 10n \log n = \Theta(n \log n)$?

  $n^2/5 + 10n \log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$? No

  $3n^2 + 4n = \Theta(n^2)$? Yes

  $3n^2 + 4n = \Theta(n^3)$? No, but $O(n^3)$

  $n/5 + 10n \log n = \Theta(n^2)$?

  $n^2/5 + 10n \log n = \Theta(n \log n)$?

  $n^2/5 + 10n \log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$? No
  $3n^2 + 4n = \Theta(n^2)$? Yes
  $3n^2 + 4n = \Theta(n^3)$? No, but $O(n^3)$
  $n/5 + 10n \log n = \Theta(n^2)$? No, but $O(n^2)$
  $n^2/5 + 10n \log n = \Theta(n \log n)$?
  $n^2/5 + 10n \log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$? No

  $3n^2 + 4n = \Theta(n^2)$? Yes

  $3n^2 + 4n = \Theta(n^3)$? No, but $O(n^3)$

  $n/5 + 10n \log n = \Theta(n^2)$? No, but $O(n^2)$

  $n^2/5 + 10n \log n = \Theta(n \log n)$? No

  $n^2/5 + 10n \log n = \Theta(n^2)$?

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

- $3n^2 + 4n = \Theta(n)$? — No

  $3n^2 + 4n = \Theta(n^2)$? — Yes

  $3n^2 + 4n = \Theta(n^3)$? — No, but $O(n^3)$

  $n/5 + 10n \log n = \Theta(n^2)$? — No, but $O(n^2)$

  $n^2/5 + 10n \log n = \Theta(n \log n)$? — No

  $n^2/5 + 10n \log n = \Theta(n^2)$? — Yes

Algorithms

- An *algorithm* is a finite sequence of **precise instructions** for performing a computation or for solving a problem.

  A *computational problem* is a specification of the desired input-output relationship.

  Example (Computational Problem and Algorithm)

  The following procedure is an algorithm for calculating the sum of n given numbers $a_1, a_2, \ldots, a_n$.

  Step 1: set $S = 0$

  Step 2: for $i = 1$ to n, replace S by $S + a_i$

  Step 3: output S

Instance

- An *instance* of a problem is all the inputs needed to compute a solution to the problem.

Example (Instance of Problem)

$$< 8, 3, 6, 7, 1, 2, 9 >$$

- A *correct algorithm* halts with the correct output for **every input instance**. We can then say that the algorithm solves the problem.

Time and Space Complexity

- The number of **machine operations**(addition, multiplication, comparison, replacement, etc) needed in an algorithm is the *time complexity* of the algorithm, and **amount of memory** needed is the *space complexity* of the algorithm.

Time and Space Complexity

- The number of **machine operations**(addition, multiplication, comparison, replacement, etc) needed in an algorithm is the *time complexity* of the algorithm, and **amount of memory** needed is the *space complexity* of the algorithm.

  Example (Algorithm)

  Step 1: set $S = 0$

  Step 2: for $i = 1$ to n, replace S by $S + a_i$

  Step 3: output S

Time and Space Complexity

- The number of **machine operations**(addition, multiplication, comparison, replacement, etc) needed in an algorithm is the *time complexity* of the algorithm, and **amount of memory** needed is the *space complexity* of the algorithm.

Example (Algorithm)

Step 1: set $S = 0$

Step 2: for $i = 1$ to n, replace S by $S + a_i$

Step 3: output S

Step 1 and Step 3 take one operation. Step 2 takes $2n$ operations. Therefore, altogether this algorithm takes $2n + 2$ operations. The time complexity is $O(n)$.

- **Example**

Consider the evaluation of $f(x) = 1 + 2x + 3x^2 + 4x^3$.
Direct computation takes 3 additions and 6 multiplications.
Can we do better?

Horner's Algorithm and Its Complexity

- **Example**

  Consider the evaluation of $f(x) = 1 + 2x + 3x^2 + 4x^3$.
  Direct computation takes 3 additions and 6 multiplications.
  Can we do better?

  Another way is $f(x) = 1 + x(2 + x(3 + 4x))$, which takes 3 additions and 3 multiplications.

- **Example**

Consider the evaluation of $f(x) = 1 + 2x + 3x^2 + 4x^3$. Direct computation takes 3 additions and 6 multiplications. Can we do better?

Another way is $f(x) = 1 + x(2 + x(3 + 4x))$, which takes 3 additions and 3 multiplications.

Step 1: set $S = a_n$

Step 2: for $i = 1$ to n, replace S by $a_{n-i} + Sx$

Step 3: output S

Horner's Algorithm and Its Complexity

- Step 1: set $S = a_n$
  Step 2: for $i = 1$ to n, replace S by $a_{n-i} + Sx$
  Step 3: output S

Horner's Algorithm and Its Complexity

- Step 1: set $S = a_n$
  Step 2: for $i = 1$ to n, replace S by $a_{n-i} + Sx$
  Step 3: output S

The final value of S output at Step 3 is the desired value of $a_0 + a_1x + \cdots + a_nx^n$. The number of operations needed in this algorithm is $1 + 3n + 1 = 3n + 2$. So the time complexity of this algorithm is $O(n)$.

Time Complexity

- Determine the time complexity of the following algorithm:

  for $i := 1$ to n

      for $j := 1$ to n

          $a := 2 * n + i * j;$

      end for

  end for

Time Complexity

- Determine the time complexity of the following algorithm:

  for $i := 1$ to n

  for $j := 1$ to n

  $a := 2 * n + i * j;$

  end for

  end for

  In the second loop, computing a takes 4 operations (two multiplications, one addition, and one replacement). For each i, it takes $4n$ operations to complete the second loop. So it takes $n \times 4n = 4n^2$ operations to complete the two loops. The time complexity of this algorithm is $O(n^2)$.

Time Complexity

- Determine the time complexity of the following algorithm:

  $S := 0$

  for $i := 1$ to n

      for $j := 1$ to i

          $S := S + i * j;$

      end for

  end for

Time Complexity

- Determine the time complexity of the following algorithm:

  $S := 0$

  for $i := 1$ to n

  for $j := 1$ to i

  $S := S + i * j;$

  end for

  end for

  Computing S takes 3 operations. For each i, completing the second loop takes $3i$ operations. So altogether it takes

  $$1 + \sum_{i=1}^{n} 3i = 1 + 3\frac{n(n+1)}{2}$$

  operations. So the complexity of this algorithm is $O(n^2)$.

More on Time Complexity

- **Example:** (**Insertion Sort**)

  Input: $A[1 \dots n]$ is an array of numbers

  for $j := 2$ to n

  $key = A[j]$;

  $i = j - 1$;

  while $i \geq 1$ and $A[i] > key$ do

  $A[i + 1] = A[i]$;

  $i - -$;

  end while

  $A[i + 1] = key$;

  end for

More on Time Complexity

- **Example:** (**Insertion Sort**)

  Input: $A[1 \dots n]$ is an array of numbers

  for $j := 2$ to n

  $\quad key = A[j];$

  $\quad i = j - 1;$

  $\quad$ while $i \geq 1$ and $A[i] > key$ do

  $\qquad A[i+1] = A[i];$

  $\qquad i--;$

  $\quad$ end while

  $\quad A[i+1] = key;$

  end for

Where in the sorted part to put "key"?

Three Cases of Analysis: I

- Best Case: constraints on the input, other than size, resulting in the fastest possible running time for the given size.

Three Cases of Analysis: I

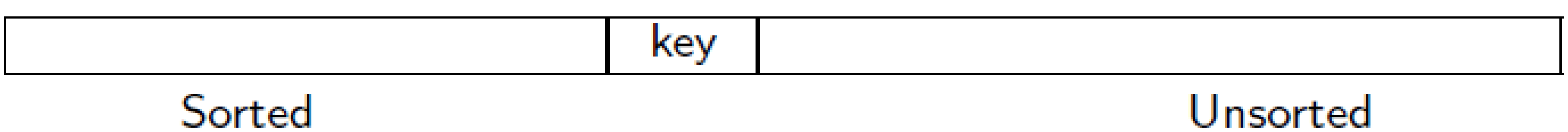
- Best Case: constraints on the input, other than size, resulting in the fastest possible running time for the given size.

  Example: (Insertion Sort)

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[n]$$

The number of comparisons needed is

$$\underbrace{1 + 1 + 1 + \cdots + 1}_{n-1} = n - 1 = \Theta(n)$$



“key” is compared to only the element right before it.

Three Cases of Analysis: II

- Worst Case: constraints on the input, other than size, resulting in the slowest possible running time for the given size.

Three Cases of Analysis: II

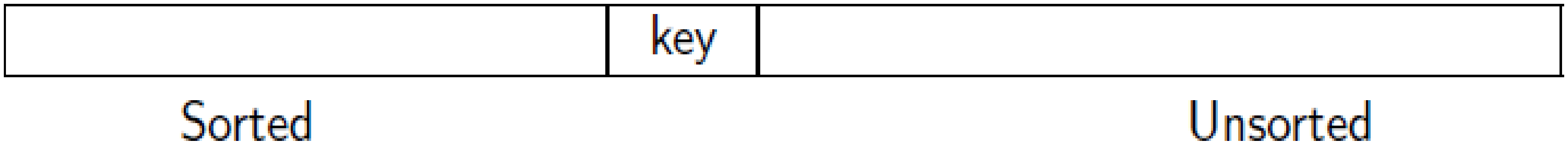
- Worst Case: constraints on the input, other than size, resulting in the slowest possible running time for the given size.

  Example: (Insertion Sort)

  $$A[1] \geq A[2] \geq A[3] \geq \cdots \geq A[n]$$

  The number of comparisons needed is

  $$1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$



"key" is compared to everything element before it.

Three Cases of Analysis: III

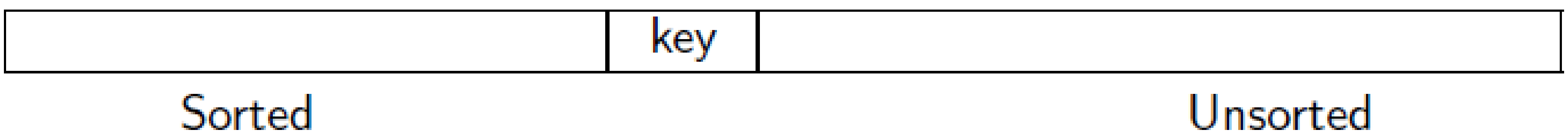
- Average Case: average running time over every possible type of input for the given size (usually involve probabilities of different types of input)

Three Cases of Analysis: III

- Average Case: average running time over every possible type of input for the given size (usually involve probabilities of different types of input)

Example: (Insertion Sort)

$\Theta(n^2)$ assuming that each of the $n!$ instances are equally likely



On average, "key" is compared to half of the elements before it.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.

- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.
- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.
- Too often, programmers try to slove problems using brute force techniques and end up with slow complicated code!

Some Thoughts on Algorithm Design

- **Algorithm Design**, is mainly about designing algorithms that have small Big-O running time.
- Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.
- Too often, programmers try to slove problems using brute force techniques and end up with slow complicated code!
- A few hours of abstract thought devoted to algorithm design could have speeded up the solution substantially and simplified it!

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Blame yourself.

I couldn't find a polynomial-time algorithm.
I guess I am too dumb.

Dealing with Hard Problems

- What happens if you can't find an efficient algorithm for a given problem?

Show that no-efficient algorithm exists.

I couldn't find a polynomial-time algorithm, because no such algorithm exists.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

  How can we prove the non-existence of something?

Dealing with Hard Problems

- Showing that a problem has an efficient algorithm is, relatively easy:

  "All" that is needed is to demonstrate an algorithm.

- Proving that no efficient algorithm exists for a particular problem is difficult:

  How can we prove the non-existence of something?

  We will now learn about NP-Complete problems, which provide us with a way to approach this question.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.
- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.
- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.
- So, NP-Complete problems are very likely to be hard.

Introduction

- A very large class of thousands of practical problems for which it is not known if the problems have "efficient" solutions.

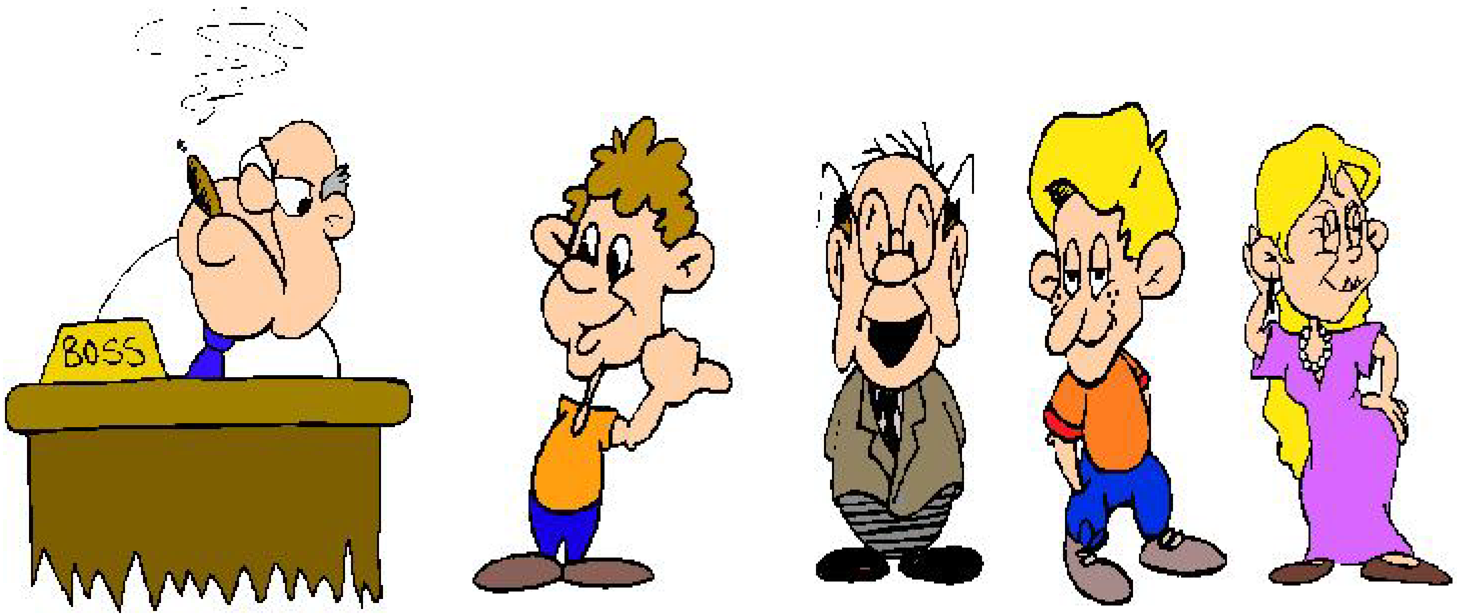
- It is known that if any one of the NP-Complete problems has an efficient solution then all of the NP-Complete problems have efficient solutions.

- Researchers have spent innumberable man-years trying to find efficient solutions to these problems but failed.

- So, NP-Complete problems are very likely to be hard.

- What do you do: prove that your problem is NP-Complete.

What do you actually do:



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people!

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.

Encoding the Inputs of Problems

- Complexity of a problem is measure w.r.t the size of input.

- In order to formally discuss how hard a problem is, we need to be much more formal than before about the input size of a problem.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

- The exact input size s, determined by an optimal encoding method, is hard to compute in most cases.

The Input Size of Problems

- The input size of a problem might be defined in a number of ways.

  Definition The *input size* of a problem is the minimum number of bits ($\{0,1\}$) needed to encode the input of the problem.

- The exact input size s, determined by an optimal encoding method, is hard to compute in most cases.

  However, we do not need to determine s exactly.

  For most problems, it is sufficient to choose some natural, and (usually) simple, encoding and use the size s of this encoding.

Input Size Example: Composite

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Input Size Example: Composite

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

  Question:

  What is the input size of this problem?

Input Size Example: Composite

- **Example:**

  Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

  Question:

  What is the input size of this problem?

  Any integer $n > 0$ can be represented in the binary number system as a string $a_0 a_1 \cdots a_k$ of length $\lceil \log_2(n+1) \rceil$.

  Thus, a natural measure of input size is $\lceil \log_2(n+1) \rceil$ (or just $\log_2 n$)

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

  Question:

  What is the input size of this problem?

Input Size Example: Sorting

- **Example:**

  Sort n integers $a_1, \ldots, a_n$

  Question:

  What is the input size of this problem?

  Using fixed length encoding, we write a_i as a binary string of length $m = \lceil \log_2 \max(|a_i| + 1) \rceil$.

  This coding gives an input size nm.

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

Complexity in terms of Input Size

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

  This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

Complexity in terms of Input Size

- **Example:** (Composite)

  The naive algorithm for determining whether n is composite compares n with the first $n-1$ numbers to see if any of them divides n

  This makes $\Theta(n)$ comparisons, so it might seem linear and very efficient.

  But, note that the input size of this problem is $size(n) = \log_2 n$, so the number of comparisons performed is actually $\Theta(n) = \Theta(2^{size(n)})$, which is exponential.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

Input Size of Problems

- **Definition** Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

Example:

All polynomials are of the same type, but *polynomials* and *exponentials* are of different types.

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

  Question:

  What is the input size of this problem?

Input Size Example: Integer Multiplication

- **Example:** (Integer Multiplication problem)

  Compute $a \times b$.

  Question:

  What is the input size of this problem?

  The minimum inpute size is

  $$s = \lceil \log_2(a+1) \rceil + \lceil \log_2(b+1) \rceil.$$

  A natural choice is to use $t = \log_2 \max(a, b)$ since $\frac{s}{2} \le t \le s$.

Next Lecture

- P vs NP, number theory ...