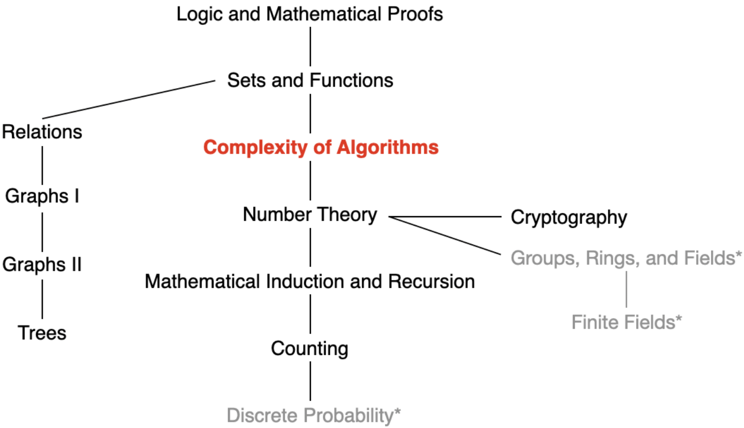
Discrete Mathematics for Computer Science

Lecture 6: Complexity of Algorithms

Dr. Ming Tang

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)
Email: tangm3@sustech.edu.cn

This Lecture



The growth of functions, complexity of algorithm, P and NP problem, ...

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 $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|,$$

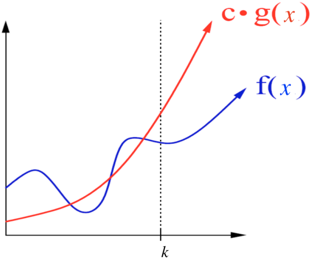
whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

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 $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|,$$

whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]



Big-O Estimates for Polynomials

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

Big-O Estimates for Polynomials

Let $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, where $a_0, a_1, ..., a_n$ 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 $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$, where $a_0, a_1, ..., a_n$ 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

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

$n! = O(n^n)$

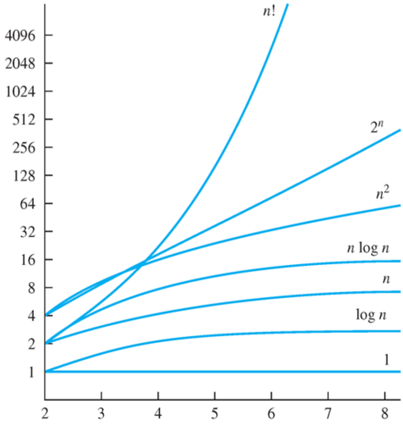
$\log n! = O(n \log n)$

$\log_a n = O(n)$ 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

Big-O Estimates for Some Functions



Big-O Estimates for Some Functions

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

Big-O Estimates for Some Functions

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

Proof: We always have $\log_a n \leq n$ for $n \geq 1$. This can be proven using mathematical induction. ...

- $n = 1$: $log_a 1 = 0 < 1$
- Suppose $\log_a n \leq n$ for $n > 1$:

$$\log_a(n+1) \leq \log_a(an) = \log_a n + 1 \leq n + 1$$

Big-O Estimates for Some Functions

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

Big-O Estimates for Some Functions

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

Proof: According to L'Hopital's rule,

$$\lim_{n \to \infty} \frac{n^a}{2^n} = 0$$

Thus, $n^a \leq 2^n$ for large enough n.

Big-O Estimates for Some Functions

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

Proof: According to L'Hopital's rule,

$$\lim_{n \to \infty} \frac{n^a}{2^n} = 0$$

Thus, $n^a \leq 2^n$ for large enough n.

Note: If f and g are functions such that

$$\lim_{n \to \infty} \frac{|f(x)|}{|g(x)|} = C < \infty,$$

then $f(x) \leq (C+1)g(x)$ for large enough x. So $f(n) = O(g(n))$. If that limit is ∞, then $f(n)$ is not $O(g(n))$.

Combinations of Functions

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

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 $C = C_1 + C_2$.

$k = \max\{k_1, k_2\}$.

Combinations of Functions

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

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 k $> \max(k_1, k_2)$,

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

where $C = C_1 C_2$.

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 $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

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 $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

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.

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 $f(x)$ is $\Omega(g(x))$ if there are positive constants C and k such that

$$|f(x)| \geq C|g(x)|$$

whenever $x > k$. [This is read as "$f(x)$ is big-Omega of $g(x)$."]

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)

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 $f(x)$ is $\Theta(g(x))$ if

- $f(x)$ is $O(g(x))$ and
- $f(x)$ is $\Omega(g(x))$.

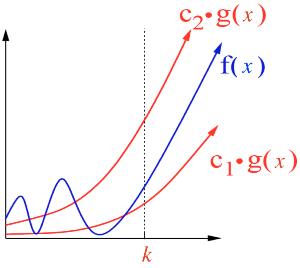
When $f(x)$ is $\Theta(g(x))$, we say that $f(x)$ is big-Theta of $g(x)$, that $f(x)$ is of order $g(x)$, and that $f(x)$ and $g(x)$ are of the same order.

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

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 $f(x)$ is $\Theta(g(x))$ if

- $f(x)$ is $O(g(x))$ and
- $f(x)$ is $\Omega(g(x))$.

When $f(x)$ is $\Theta(g(x))$, we say that $f(x)$ is big-Theta of $g(x)$, that $f(x)$ is of order $g(x)$, and that $f(x)$ and $g(x)$ are of the same order.



Big-Theta Notation

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

Big-Theta Notation

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

- $f(x) = O(x^n)$
- $f(x) = \Omega(x^n)$

Big-Theta Notation

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

- $f(x) = O(x^n)$
- $f(x) = \Omega(x^n)$

Note: If f and g are functions such that

$$\lim_{n\to\infty} \frac{|f(x)|}{|g(x)|} = C < \infty,$$

and

$$C \neq 0,$$

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

Big-Theta Notation: Examples

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

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

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

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

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

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

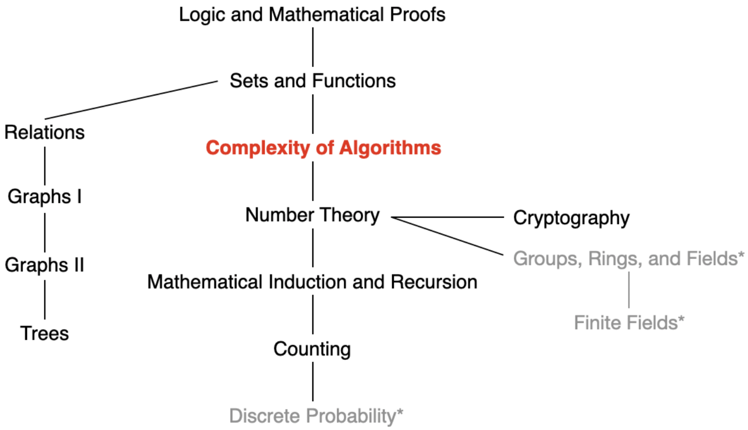
Big-Theta Notation: Examples

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

The Growth of Functions

- Big-O notation, e.g., $O(n^2)$
  - Upper bound
- Big-Omega notation, e.g., $\Omega(n^2)$
  - Lower bound
- Big-Theta notation, e.g., $\Theta(n^2)$
  - Of the same order

This Lecture



The growth of functions, complexity of algorithm, P and NP, ...

Algorithms

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

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.

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):

- Computational Problem: Input n numbers $a_1, a_2, ..., a_n$; Output the sum of the n numbers.

Southern University of Science and Technology

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):

- Computational Problem: Input n numbers $a_1, a_2, ..., a_n$; Output the sum of the n numbers.
- Algorithm: the following procedures

  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 a realization of all the inputs needed to compute a solution to the problem.

Instance

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

Example: $8, 3, 6, 7, 1, 2, 9$

Instance

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

Example: $8, 3, 6, 7, 1, 2, 9$

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

- Time complexity: The number of machine operations (addition, multiplication, comparison, replacement, etc) needed in an algorithm.

Time and Space Complexity

- Time complexity: The number of machine operations (addition, multiplication, comparison, replacement, etc) needed in an algorithm.
- Space complexity: the amount of memory needed.

Time and Space Complexity

- Time complexity: The number of machine operations (addition, multiplication, comparison, replacement, etc) needed in an algorithm.
- Space complexity: the amount of memory needed.

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

- Time complexity: The number of machine operations (addition, multiplication, comparison, replacement, etc) needed in an algorithm.
- Space complexity: the amount of memory needed.

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 Complexity:

- Steps 1 and 3 take one operation.
- Step 2 takes $2n$ operations.

Therefore, altogether this algorithm takes $1 + 2n + 1$ operations. The time complexity is $O(n)$.

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.

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?

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.

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.

Horner's algorithm for computing
$f(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1} + a_nx^n = a_0 + x(a_1 + ... + x(a_{n-1} + a_nx))$
at a particular x:

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

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.

Horner's algorithm for computing
$f(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1} + a_nx^n = a_0 + x(a_1 + \ldots + x(a_{n-1} + a_nx))$
at a particular x:

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 number of operations needed in this algorithm is $1 + 3n + 1 = 3n + 2$. So the time complexity of this algorithm is $O(n)$.

Note: Operations: addition, multiplication, comparison, replacement, etc.

Time Complexity: Example

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: Example

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

- Computing $a := 2 \times n + i \times j$ takes 4 operations (two multiplications, one addition, and one replacement).

Time Complexity: Example

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

- Computing $a := 2 \times n + i \times j$ takes 4 operations (two multiplications, one addition, and one replacement).
- For each i, it takes $4n$ operations to complete the second loop.

Time Complexity: Example

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

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

Time Complexity: Example

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: Example

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 $S := S + i \times j$ takes 3 operations.

Time Complexity: Example

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 $S := S + i \times j$ takes 3 operations.
- For each i , completing the second loop takes $3i$ operations.

Time Complexity: Example

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

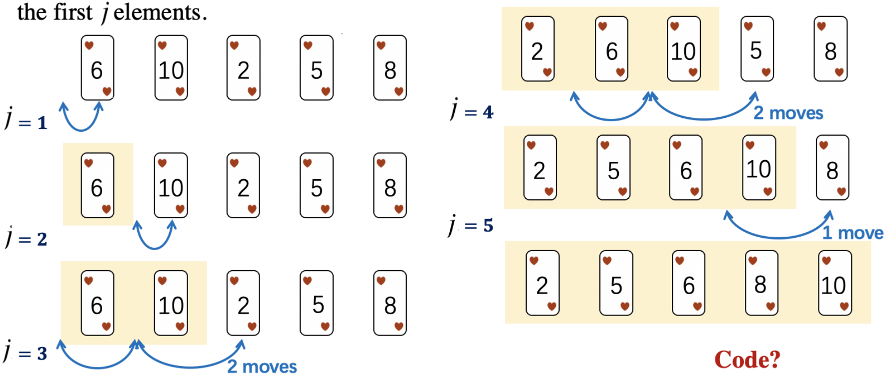
- Computing $S := S + i \times j$ takes 3 operations.
- For each i , completing the second loop takes $3i$ operations.
- Thus, this algorithm takes

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

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

Time Complexity: Example - Insertion Sort

In iteration j, we move the j-th element left until its correct place is found among the first j elements.



Code?

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

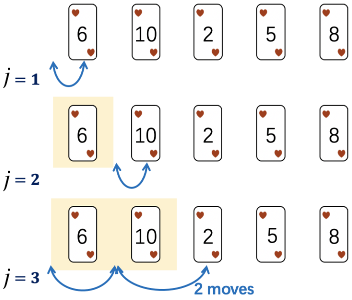
    $A[i+1] = A[i]$;

    $i - -$;

  end while

  $A[i+1] = key$;

end for



Time Complexity: Example - Insertion Sort

Input: $A[1\ldots n]$ 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

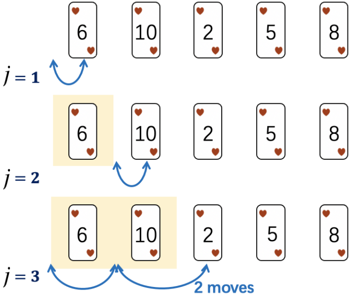
    $A[i+1] = A[i]$;

    $i - -$;

  end while

  $A[i+1] = key$;

end for



The time complexity depends on the input array $A[1, ..., n]$.

Time Complexity: Example - Insertion Sort

Input: $A[1 \ldots n]$ 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

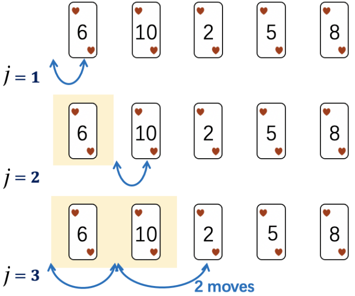
        $A[i+1] = A[i]$;

        $i - -$;

    end while

    $A[i+1] = key$;

end for



The time complexity depends on the input array $A[1, \ldots, n]$.

Consider only the number of comparisons

Three Cases of Analysis: Best-Case

Best-Case Complexity: The smallest number of operations needed to solve the given problem using this algorithm on input of specified size.

Three Cases of Analysis: Best-Case

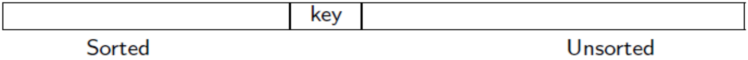
Best-Case Complexity: The smallest number of operations needed to solve the given problem using this algorithm on input of specified size.

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: Worst-Case

Worst-Case Complexity: The largest number of operations needed to solve the given problem using this algorithm on input of specified size.

Three Cases of Analysis: Worst-Case

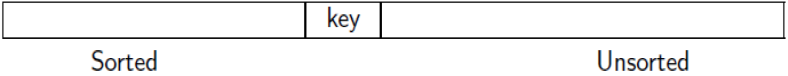
Worst-Case Complexity: The largest number of operations needed to solve the given problem using this algorithm on input of specified size.

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: Average-Case

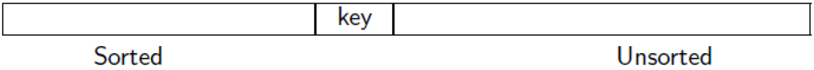
Average-Case Complexity: The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis.

Three Cases of Analysis: Average-Case

Average-Case Complexity: The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis.

Example: (Insertion Sort)

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



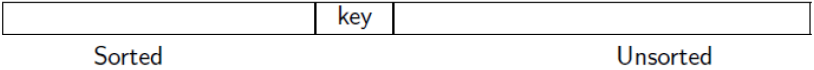
On average, “key” is compared to half of the elements before it.

Three Cases of Analysis: Average-Case

Average-Case Complexity: The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis.

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.

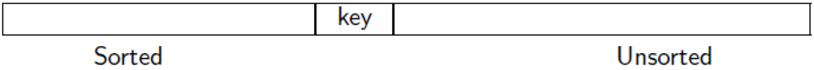
- For a particular instance, compute the number of comparisons
- Since we assume equal probability, take the average

Three Cases of Analysis: Average-Case

Average-Case Complexity: The average number of operations used to solve the problem over all possible inputs of a given size is found in this type of analysis.

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.

- For a particular instance, compute the number of comparisons
- Since we assume equal probability, take the average

Average-case complexity is usually difficult to compute.

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 solve problems using brute force techniques and end up with slow complicated code!

- The most straightforward manner based on the statement of the problem and the definitions of terms

Some Thoughts on Algorithm Design

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

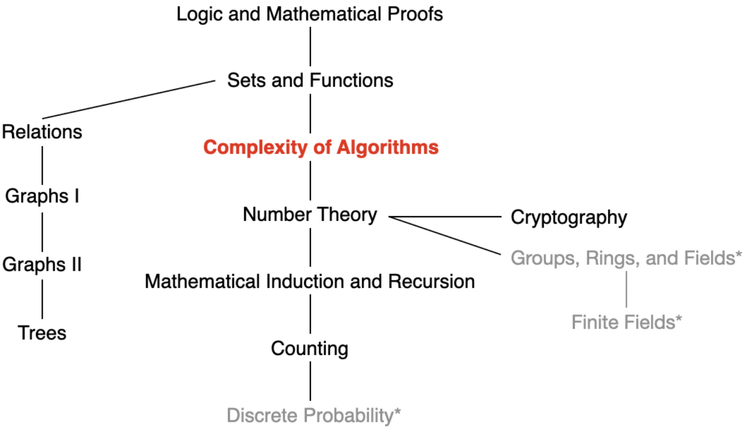
Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code!

- The most straightforward manner based on the statement of the problem and the definitions of terms

A few hours of abstract thought devoted to algorithm design could speed up the solution substantially and simplified it!

This Lecture



The growth of functions, complexity of algorithm, P and NP problem, ...

Dealing with Hard Problems

What happens if you cannot find an efficient algorithm for a given problem?

Dealing with Hard Problems

What happens if you cannot 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 cannot 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:

- Design such an algorithm.

Dealing with Hard Problems

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

- Design such 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:

- Design such 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:

- Design such 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 provides us with a way to approach this question.

NP-Complete

P: Problems that are solvable using an algorithm with polynomial worst-case complexity

NP-Complete

P: Problems that are solvable using an algorithm with polynomial worst-case complexity

NP: Problems for which a solution can be checked in polynomial time.

NP-Complete

P: Problems that are solvable using an algorithm with polynomial worst-case complexity

NP: Problems for which a solution can be checked in polynomial time.

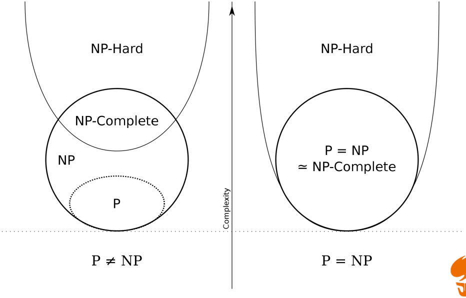
NP-Complete: If any of these problems can be solved by a polynomial worst-case time algorithm, then all problems in the class NP can be solved by polynomial worst-case time algorithms.

NP-Complete

P: Problems that are solvable using an algorithm with polynomial worst-case complexity

NP: Problems for which a solution can be checked in polynomial time.

NP-Complete: If any of these 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.



NP-Complete

Researchers have spent many years trying to find efficient solutions to these problems but failed.

NP-Complete

Researchers have spent many years trying to find efficient solutions to these problems but failed.

NP-Complete and NP-Hard problems are very likely to be hard.

NP-Complete

Researchers have spent many years trying to find efficient solutions to these problems but failed.

NP-Complete and NP-Hard problems are very likely to be hard.

Thus, to proving that no efficient algorithm exists for a particular problem?

NP-Complete

Researchers have spent many years trying to find efficient solutions to these problems but failed.

NP-Complete and NP-Hard problems are very likely to be hard.

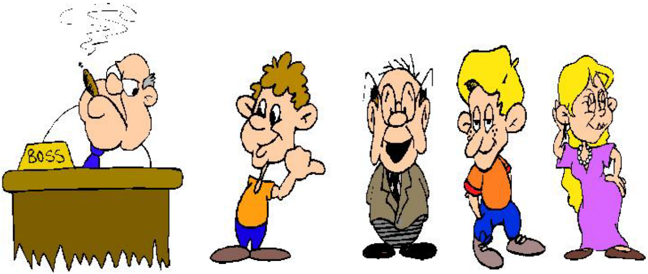
Thus, to proving that no efficient algorithm exists for a particular problem?

Prove that your problem is NP-Complete or even NP-Hard:

- Show that your problem can be reduced to a typical (well-known) NP-Complete or NP-Hard problem.

NP-Complete

What do you actually do:



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

Dealing with Hard Problems

What is a polynomial-time algorithms?

- Preliminary: Input size of a problem
- Polynomial-time algorithms

What types of problem that P and NP account for?

- Decision problems and optimization problem

Details for P and NP

Encoding the Inputs of Problems

Complexity of a problem is measure with respect to the size of input:

- E.g., for insertion sort, $\Theta(n^2)$ is the average-case complexity, where n is the length of the array.

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.

Now, we consider the following definition:
Definition: The input size of a problem is the minimum number of bits (i.e., $\{0, 1\}$) 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.

Now, we consider the following definition:
Definition: The input size of a problem is the minimum number of bits (i.e., $\{0, 1\}$) 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.

Now, we consider the following definition:
Definition: The input size of a problem is the minimum number of bits (i.e., $\{0, 1\}$) 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.

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

- E.g., 5 can be encoded as 101.

Input Size Example: Composite

Example: Input a positive integer n; output if there are integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Input Size Example: Composite

Example: Input a positive integer n; output if there are 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: Input a positive integer n; output if there are 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 $a_0 a_1 ... a_k$ of length $\lceil \log_2(n+1) \rceil$.

Input Size Example: Composite

Example: Input a positive integer n; output if there are 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 $a_0 a_1 ... a_k$ 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 $a_1, \ldots, a_n$.

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 $m = \lceil \log_2 \max(|a_i| + 1) \rceil$.

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 of nm.

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 of nm.

Note: Back to our earlier discussions for complexity, when we use fixed length encoding regardless of a_i for $i = 1, 2, \ldots, n$, the value of m becomes a constant. Thus, we can omit the constant m.

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.

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, the input size of this problem is $\log_2 n$ instead of n. The number of comparisons performed is actually $\Theta(n)$, which can be represented as $\Theta(2^{(\log_2 n)})$. It is exponential with respect to the input size.

Dealing with Hard Problems

What is a polynomial-time algorithms?

- Preliminary: Input size of a problem
- Polynomial-time algorithms

What types of problem that P and NP account for?

- Decision problems and optimization problem

Details for P and NP

Polynomial-Time Algorithms

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Polynomial-Time Algorithms

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial-time.

Polynomial-Time Algorithms

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Whether we use n or n^a (for a fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial-time.

Example:
The standard multiplication algorithm has time $O(m_1 m_2)$, where m_1 and m_2 denote the number of digits in the two integers, respectively.

Nonpolynomial-Time Algorithms

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Nonpolynomial-Time Algorithms

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

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.

Nonpolynomial-Time Algorithms

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

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.

- Let $m = \log_2 n$ be the input size of this problem

Nonpolynomial-Time Algorithms

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

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.

- Let $m = \log_2 n$ be the input size of this problem
- Thus, the complexity if $\Theta(n) = \Theta(2^{(\log_2 n)})$, which is $\Theta(2^m)$

Nonpolynomial-Time Algorithms

Definition: An algorithm is nonpolynomial-time if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

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.

- Let $m = \log_2 n$ be the input size of this problem
- Thus, the complexity if $\Theta(n) = \Theta(2^{(\log_2 n)})$, which is $\Theta(2^m)$
- The algorithm is nonpolynomial!

Polynomial- vs. Nonpolynomial-Time

Nonpolynomial-time algorithms are impractical.

- 2^n for $n = 100$: it takes billions of years!!!

Polynomial- vs. Nonpolynomial-Time

Nonpolynomial-time algorithms are impractical.

- 2^n for $n = 100$: it takes billions of years!!!

In reality, an $O(n^{20})$ algorithm is not really practical.

Dealing with Hard Problems

What is a polynomial-time algorithms?

- Preliminary: Input size of a problem
- Polynomial-time algorithms

What types of problem that P and NP account for?

- Decision problems and optimization problem

Details for P and NP

Decision Problems and Optimization Problem

Definition: A decision problem is a question that has two possible answers: yes and no.

Decision Problems and Optimization Problem

Definition: A decision problem is a question that has two possible answers: yes and no.

Definition: An optimization problem requires an answer that is an optimal configuration.

- Decision variables
- Maximize or minimize certain objective subject to some constraints

Decision Problems and Optimization Problem

Definition: A decision problem is a question that has two possible answers: yes and no.

Definition: An optimization problem requires an answer that is an optimal configuration.

- Decision variables
- Maximize or minimize certain objective subject to some constraints

An optimization problem usually has a corresponding decision problem.

Decision Problems and Optimization Problem

Definition: A decision problem is a question that has two possible answers: yes and no.

Definition: An optimization problem requires an answer that is an optimal configuration.

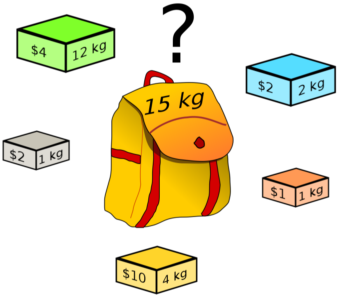
- Decision variables
- Maximize or minimize certain objective subject to some constraints

An optimization problem usually has a corresponding decision problem.

Examples:
Knapsack vs. Decision Knapsack (DKnapsack)

Knapsack V.S. DKnapsack

We have a knapsack of capacity W (a positive integer) and N objects with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, where v_n and w_n are positive integers.



Knapsack V.S. DKnapsack

We have a knapsack of capacity W (a positive integer) and N objects with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, where v_n and w_n are positive integers.

Optimization problem (Knapsack):

- Decision variable $x_n \in \{0, 1\}$: $x_n = 1$, object x is placed in the knapsack; $x_n = 0$, otherwise
- Maximize $\sum_{n=\{1,\ldots,N\}} x_n v_n$, subject to constraint $\sum_{n=\{1,\ldots,N\}} x_n w_n \leq W$.

Knapsack V.S. DKnapsack

We have a knapsack of capacity W (a positive integer) and N objects with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, where v_n and w_n are positive integers.

Optimization problem (Knapsack):

- Decision variable $x_n \in \{0, 1\}$: $x_n = 1$, object x is placed in the knapsack; $x_n = 0$, otherwise
- Maximize $\sum_{n=\{1,\ldots,N\}} x_n v_n$, subject to constraint $\sum_{n=\{1,\ldots,N\}} x_n w_n \leq W$.

Decision problem (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

Knapsack V.S. DKnapsack

We have a knapsack of capacity W (a positive integer) and N objects with weights $w_1, \ldots, w_N$ and values $v_1, \ldots, v_N$, where v_n and w_n are positive integers.

Optimization problem (Knapsack):

- Decision variable $x_n \in \{0, 1\}$: $x_n = 1$, object x is placed in the knapsack; $x_n = 0$, otherwise
- Maximize $\sum_{n=\{1,\ldots,N\}} x_n v_n$, subject to constraint $\sum_{n=\{1,\ldots,N\}} x_n w_n \leq W$.

Decision problem (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

The optimization problem is at least as hard as the decision problem.

Decision Problems and Optimization Problem

Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually trivial.

- First, solve the optimization problem
- Then, check the decision problem.

Thus, if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Complexity Classes

Theory of Complexity deals with

1. the classification of certain "decision problems" into several classes:
   - the class of "easy" problems
   - the class of "hard" problems
   - the class of "hardest" problems
2. relations among the three classes
3. properties of problems in the three classes

Complexity Classes

Theory of Complexity deals with

1. the classification of certain "decision problems" into several classes:
   - the class of "easy" problems
   - the class of "hard" problems
   - the class of "hardest" problems
2. relations among the three classes
3. properties of problems in the three classes

Question: How to classify decision problems?

Complexity Classes

Theory of Complexity deals with

1. the classification of certain "decision problems" into several classes:
   - the class of "easy" problems
   - the class of "hard" problems
   - the class of "hardest" problems
2. relations among the three classes
3. properties of problems in the three classes

Question: How to classify decision problems?

Answer: Use polynomial-time algorithms.

SUSTech Southern University of Science and Technology

Complexity Classes

Theory of Complexity deals with

1. the classification of certain "decision problems" into several classes:
   - the class of "easy" problems
   - the class of "hard" problems
   - the class of "hardest" problems
2. relations among the three classes
3. properties of problems in the three classes

Question: How to classify decision problems?

Answer: Use polynomial-time algorithms.

P problem, NP problem, ...

Dealing with Hard Problems

What is a polynomial-time algorithms?

- Preliminary: Input size of a problem
- Polynomial-time algorithms

What types of problem that P and NP account for?

- Decision problems and optimization problem

Details for P and NP

The Class P

Definition: A problem is solvable in polynomial time (or more simply, the problem is in polynomial time) if there exists an algorithm which solves the problem in polynomial time

- This problem is called tractable.

Definition (The Class P): The class P consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

The Class P

Question: How to prove that a decision problem is in P?

The Class P

Question: How to prove that a decision problem is in P?

Answer: Find a polynomial-time algorithm.

The Class P

Question: How to prove that a decision problem is in P?

Answer: Find a polynomial-time algorithm.

Question: How to prove that a decision problem is not in P?

The Class P

Question: How to prove that a decision problem is in P?

Answer: Find a polynomial-time algorithm.

Question: How to prove that a decision problem is not in P?

Answer: You need to prove that there is no polynomial-time algorithm for this problem. (much much harder)

The Class P

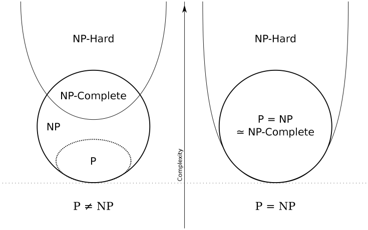
Question: How to prove that a decision problem is in P?

Answer: Find a polynomial-time algorithm.

Question: How to prove that a decision problem is not in P?

Answer: You need to prove that there is no polynomial-time algorithm for this problem. (much much harder)

- Some other definitions for potentially harder problems ....



Certificates and Verifying Certificates

Before introduce NP Problem, some new definitions ...

Certificates and Verifying Certificates

Before introduce NP Problem, some new definitions ...

A decision problem is usually formulated as:

Is there an object satisfying some conditions?

Certificates and Verifying Certificates

Before introduce NP Problem, some new definitions ...

A decision problem is usually formulated as:

Is there an object satisfying some conditions?

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Certificates and Verifying Certificates

Before introduce NP Problem, some new definitions ...

A decision problem is usually formulated as:

Is there an object satisfying some conditions?

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Example (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

To show V is a yes-input, a certificate is a subset of the objects that

- fit in the knapsack (i.e., the sum weight does not exceed the capacity)
- have a total value at least V

Certificates and Verifying Certificates

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

Certificates and Verifying Certificates

A certificate (or witness) is a specific object corresponding to a yes-input, such that it can be used to show that the input is indeed a yes-input.

Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

Proposition: The problem **LongPath(G,k)** is in **NP**.
Proof: (PARTIAL!)

1. Note that **LongPath(G,k)** is a decision problem, as the definition of NP requires!
2. Here's my notion of certificate: A certificate is a list of vertices comprising a path of length at least k
3. Here's my algorithm for verifying a certificate:

Verify(G,k,C)

```
1. Read G, k, store graph G in an adjacency matrix
2. Read certificate C into an array
3. if m < k, where m is the length of C, return FALSE
4. for i = 1 to m - 1 do
     if G has no edge from vertex C[i-1] to C[i] return FALSE
5. for i = 0 to m - 1 do
     for j = i + 1 to m - 1 do
       if C[i] == C[j] return FALSE
6. return TRUE
```

The Class NP

Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

The Class NP

Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

NP – "nondeterministic polynomial-time"

The Class NP

Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

NP – "nondeterministic polynomial-time"

Example (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

To show V is a yes-input, a certificate is a subset of the objects that

- fit in the knapsack (i.e., the sum weight does not exceed the capacity)
- have a total value at least V

The Class NP

Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate which allows one to verify in polynomial time that the input is indeed a yes-input.

NP – "nondeterministic polynomial-time"

Example (DKnapsack): Given V, is there a subset of the objects that fits in the knapsack and has total value at least V?

To show V is a yes-input, a certificate is a subset of the objects that

- fit in the knapsack (i.e., the sum weight does not exceed the capacity)
- have a total value at least V

DKnapsack is an NP problem.

P = NP?

One of the most important problems in CS is

Whether P = NP or P $\neq$ NP?

P = NP?

One of the most important problems in CS is

$$\text{Whether P} = \text{NP or P} \neq \text{NP?}$$

- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is doubtful.

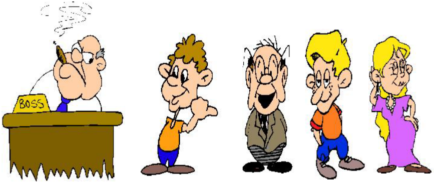
P = NP?

One of the most important problems in CS is

Whether P = NP or P ≠ NP?

- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is doubtful.

What do you actually do:



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

- NP-Hard: informally "at least as hard as the hardest problems in NP"
- NP-Complete: If the problem is NP and all other NP problems are polynomial-time reducible to it.

SUSTech Southern University of Science and Technology

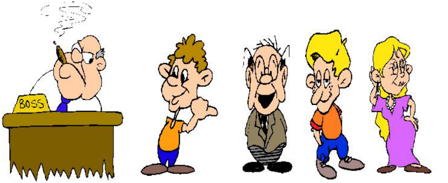
$P = NP$?

One of the most important problems in CS is

$$\text{Whether } P = NP \text{ or } P \neq NP?$$

- Observe that $P \subseteq NP$.
- Intuitively, $NP \subseteq P$ is doubtful.

What do you actually do:



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

- NP-Hard: informally "at least as hard as the hardest problems in NP"
- NP-Complete: If the problem is NP and all other NP problems are polynomial-time reducible to it.

However, we are still no closer to solving it.

What We Covered

- Decision problem and optimization
- Polynomial-time algorithms
- P problem and NP problem

We will not cover the concept of P and NP problems and the related proofs in homework or exam. If you decide to do research, these concepts and proofs are important.