Discrete Mathematics & Mathematical Reasoning
Algorithms

Colin Stirling

Informatics

Some slides based on ones by Myrto Arapinis

Algorithms

Definition

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

Algorithms

Definition

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

Euclidian algorithm

```
algorithm gcd(x,y)
    if y = 0
    then return(x)
    else return(gcd(y,x mod y))
```

Properties of an algorithm

Input it has input values from specified sets

Properties of an algorithm

Input it has input values from specified sets

Output from the input values, it produces the output values from specified sets which are the solution

Properties of an algorithm

Input it has input values from specified sets

Output from the input values, it produces the output values from specified sets which are the solution

Correctness it should produce the correct output values for each set of input values

Properties of an algorithm

Input it has input values from specified sets

Output from the input values, it produces the output values from specified sets which are the solution

Correctness it should produce the correct output values for each set of input values

Finiteness it should produce the output after a finite number of steps for any input

Properties of an algorithm

Input it has input values from specified sets

Output from the input values, it produces the output values from specified sets which are the solution

Correctness it should produce the correct output values for each set of input values

Finiteness it should produce the output after a finite number of steps for any input

Effectiveness it must be possible to perform each step correctly and in a finite amount of time

Properties of an algorithm

Input it has input values from specified sets

Output from the input values, it produces the output values from specified sets which are the solution

Correctness it should produce the correct output values for each set of input values

Finiteness it should produce the output after a finite number of steps for any input

Effectiveness it must be possible to perform each step correctly and in a finite amount of time

Generality it should work for all problems of the desired form

Euclidian algorithm (proof of correctness)

Lemma

If $a = bq + r$, where a, b, q, and r are positive integers, then $\gcd(a, b) = \gcd(b, r)$

Euclidian algorithm (proof of correctness)

Lemma

If $a = bq + r$, where a, b, q, and r are positive integers, then $\gcd(a, b) = \gcd(b, r)$

Proof.

($\Rightarrow$) Suppose that d divides both a and b. Then d also divides $a - bq = r$. Hence, any common divisor of a and b must also be a common divisor of b and r

($\Leftarrow$) Suppose that d divides both b and r. Then d also divides $bq + r = a$. Hence, any common divisor of b and r must also be a common divisor of a and b.

Therefore, $\gcd(a, b) = \gcd(b, r)$ □

Description of algorithms in pseudocode

- Intermediate step between English prose and formal coding in a programming language

Description of algorithms in pseudocode

- Intermediate step between English prose and formal coding in a programming language
- Focus on the fundamental operation of the program, instead of peculiarities of a given programming language

Description of algorithms in pseudocode

- Intermediate step between English prose and formal coding in a programming language
- Focus on the fundamental operation of the program, instead of peculiarities of a given programming language
- Analyze the time required to solve a problem using an algorithm, independent of the actual programming language

Maximum

Describe an algorithm for finding the maximum value in a finite sequence of integers

Maximum

Describe an algorithm for finding the maximum value in a finite sequence of integers

Input finite sequence of integers $a_1, \ldots, a_n$

Maximum

Describe an algorithm for finding the maximum value in a finite sequence of integers

Input finite sequence of integers $a_1, \ldots, a_n$

Output a_k, $k \in \{1, \ldots, n\}$, where for all $j \in \{1, \ldots, n\}$, $a_j \leq a_k$

Maximum

Describe an algorithm for finding the maximum value in a finite sequence of integers

Input finite sequence of integers $a_1, \ldots, a_n$

Output a_k, $k \in \{1, \ldots, n\}$, where for all $j \in \{1, \ldots, n\}$, $a_j \leq a_k$

```
procedure maximum(a_1,...,a_n)
max:=a_1
for i:=2 to n
    if max < a_i
    then max:=a_i
return max
```

Linear search

Describe an algorithm for locating an item in a sequence of integers

Input integer x and finite sequence of integers $a_1, \ldots, a_n$

Linear search

Describe an algorithm for locating an item in a sequence of integers

Input integer x and finite sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

Linear search

Describe an algorithm for locating an item in a sequence of integers

Input integer x and finite sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

```
procedure linear_search(x,a₁,...,aₙ)
i:=1
while i≤n and x≠aᵢ
    i:=i+1
if i≤n
then location:=i
else location:=0
return location
```

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element
- If the middle element is strictly smaller than x, then the search proceeds with the upper half of the list

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element
- If the middle element is strictly smaller than x, then the search proceeds with the upper half of the list
- Otherwise the search proceeds with the lower half of the list (including the middle element)

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element
- If the middle element is strictly smaller than x, then the search proceeds with the upper half of the list
- Otherwise the search proceeds with the lower half of the list (including the middle element)
- Repeat this process until we have a list of size 1

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element
- If the middle element is strictly smaller than x, then the search proceeds with the upper half of the list
- Otherwise the search proceeds with the lower half of the list (including the middle element)
- Repeat this process until we have a list of size 1
- If x is equal to the single element in the list, then its position is returned

Binary search

Describe an algorithm for locating an item in an ordered sequence of integers

Input integer x and finite ordered sequence of integers $a_1, \ldots, a_n$

Output integer $i \in \{0, \ldots, n\}$ where $a_i = x$ or $i = 0$ if $x \neq a_j$ for all a_j

- The algorithm begins by comparing x with the middle element
- If the middle element is strictly smaller than x, then the search proceeds with the upper half of the list
- Otherwise the search proceeds with the lower half of the list (including the middle element)
- Repeat this process until we have a list of size 1
- If x is equal to the single element in the list, then its position is returned
- Otherwise 0 is returned to indicate that the element was not found

Binary search

```
procedure binary_search(x, a_1, ..., a_n)
i:=1
j:=n
while i<j
    m:=⌊(i + j)/2⌋
    if x > a_m
    then i:=m+1
    else j:=m
if x = a_i
then location:=i
else location:=0
return location
```

Big-O notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $O(g)$ if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \leq c|g(x)|)$$

Big-O notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $O(g)$ if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \leq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g

Big-O notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $O(g)$ if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \leq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g
- $O(g)$ is the set of all functions f that satisfy the condition above: it would be formally correct to write $f \in O(g)$

Big-O notation for function growth

Definition

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$ or $f, g : \mathbb{R} \rightarrow \mathbb{R}$. Then f is $O(g)$ if there is a constant k and a positive constant c such that

$$\forall x > k \; (|f(x)| \leq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g
- $O(g)$ is the set of all functions f that satisfy the condition above: it would be formally correct to write $f \in O(g)$
- Often the condition is: $\forall x > k \; (f(x) \leq cg(x))$

Examples

- $f(x) = x^2 + 2x + 1$

Examples

- $f(x) = x^2 + 2x + 1$
- Show $f(x)$ is $O(g)$ where $g(x) = x^2$

Examples

- $f(x) = x^2 + 2x + 1$
- Show $f(x)$ is $O(g)$ where $g(x) = x^2$
- Show $f(x)$ is also $O(g)$ where $g(x) = x^3$

Examples

- $f(x) = x^2 + 2x + 1$
- Show $f(x)$ is $O(g)$ where $g(x) = x^2$
- Show $f(x)$ is also $O(g)$ where $g(x) = x^3$
- Show $f(x)$ is not $O(h)$ where $h(x) = x$

Examples

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$

Examples

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$
- $f(x) = 1 + 2 + \ldots + x$ is $O(x^2)$

Examples

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$
- $f(x) = 1 + 2 + \ldots + x$ is $O(x^2)$
- $\log(n)$ is $O(n)$

Examples

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$
- $f(x) = 1 + 2 + \ldots + x$ is $O(x^2)$
- $\log(n)$ is $O(n)$
- $n! = 1 \times 2 \times \cdots \times n$ is $O(n^n)$

Examples

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $O(x^n)$
- $f(x) = 1 + 2 + \ldots + x$ is $O(x^2)$
- $\log(n)$ is $O(n)$
- $n! = 1 \times 2 \times \cdots \times n$ is $O(n^n)$
- $\log(n!)$ is $O(n \log(n))$

Big-Omega notation for function growth

Definition

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$ or $f, g : \mathbb{R} \rightarrow \mathbb{R}$. Then f is $\Omega(g)$ if there if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \geq c|g(x)|)$$

Big-Omega notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Omega(g)$ if there if there is a constant k and a positive constant c such that

$$\forall x > k \; (|f(x)| \geq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g

Big-Omega notation for function growth

Definition

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$ or $f, g : \mathbb{R} \rightarrow \mathbb{R}$. Then f is $\Omega(g)$ if there if there is a constant k and a positive constant c such that

$$\forall x > k \; (|f(x)| \geq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g
- Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound

Big-Omega notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Omega(g)$ if there if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \geq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g
- Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound
- Often the condition is: $\forall x > k \ (f(x) \geq cg(x))$

Big-Omega notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Omega(g)$ if there if there is a constant k and a positive constant c such that

$$\forall x > k \ (|f(x)| \geq c|g(x)|)$$

- c and k are witnesses to the relationship between f and g
- Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound
- Often the condition is: $\forall x > k \ (f(x) \geq cg(x))$
- f is $\Omega(g)$ if and only if g is $O(f)$

Big-Theta notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Theta(g)$ iff f is $O(g)$ and $\Omega(g)$

Big-Theta notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Theta(g)$ iff f is $O(g)$ and $\Omega(g)$

- f and g are of the same order

Big-Theta notation for function growth

Definition

Let $f, g : \mathbb{N} \rightarrow \mathbb{R}$ or $f, g : \mathbb{R} \rightarrow \mathbb{R}$. Then f is $\Theta(g)$ iff f is $O(g)$ and $\Omega(g)$

- f and g are of the same order
- f is $\Theta(g)$ iff there exists constants c_1, c_2 and k such that

$$\text{for all } x > k(c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|)$$

Big-Theta notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Theta(g)$ iff f is $O(g)$ and $\Omega(g)$

- f and g are of the same order
- f is $\Theta(g)$ iff there exists constants c_1, c_2 and k such that

$$\text{for all } x > k(c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|)$$

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $\Theta(x^n)$ if $a_n \neq 0$

Big-Theta notation for function growth

Definition

Let $f, g : \mathbb{N} \to \mathbb{R}$ or $f, g : \mathbb{R} \to \mathbb{R}$. Then f is $\Theta(g)$ iff f is $O(g)$ and $\Omega(g)$

- f and g are of the same order
- f is $\Theta(g)$ iff there exists constants c_1, c_2 and k such that

$$\text{for all } x > k(c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|)$$

- $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ is $\Theta(x^n)$ if $a_n \neq 0$
- $f(x) = 1 + 2 + \ldots + x$ is $\Theta(x^2)$

Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?

Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?
- How much time does it take or how much computer memory does it need

Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?
- How much time does it take or how much computer memory does it need
- We measure time complexity in terms of the number of basic operations executed and use big-O and big-Theta notation to estimate it

Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?
- How much time does it take or how much computer memory does it need
- We measure time complexity in terms of the number of basic operations executed and use big-O and big-Theta notation to estimate it
- Focus on worst-case time complexity. Derive an upper bound on the number of operations it uses to solve a problem with input of particular size (as opposed to the average-case complexity)

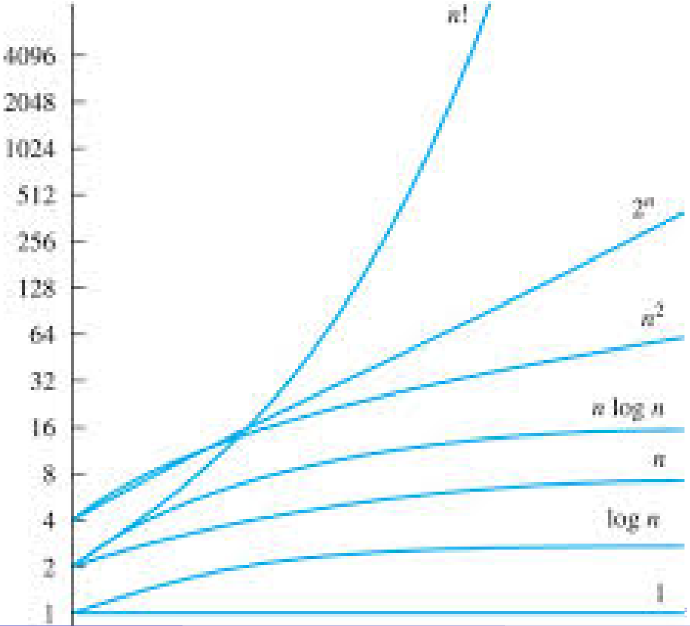
Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?
- How much time does it take or how much computer memory does it need
- We measure time complexity in terms of the number of basic operations executed and use big-O and big-Theta notation to estimate it
- Focus on worst-case time complexity. Derive an upper bound on the number of operations it uses to solve a problem with input of particular size (as opposed to the average-case complexity)
- Compute an $f(n)$ as worst case for input size n

Complexity of algorithms

- Given an algorithm, how efficient is it for solving the problem relative to input size?
- How much time does it take or how much computer memory does it need
- We measure time complexity in terms of the number of basic operations executed and use big-O and big-Theta notation to estimate it
- Focus on worst-case time complexity. Derive an upper bound on the number of operations it uses to solve a problem with input of particular size (as opposed to the average-case complexity)
- Compute an $f(n)$ as worst case for input size n
- Compare efficiency of different algorithms for the same problem

Growth



Linear search

```
procedure linear_search(x,a₁,...,aₙ)
i:=1
while i≤n and x≠aᵢ
   i:=i+1
if i≤n
then location:=i
else location:=0
return location
```

Worst-Case complexity of linear search

- Count the number of comparisons

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$
- to end the loop, one comparison $i \leq n$ is made

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$
- to end the loop, one comparison $i \leq n$ is made
- after the loop, one more $i \leq n$ comparison is made

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$
- to end the loop, one comparison $i \leq n$ is made
- after the loop, one more $i \leq n$ comparison is made
- If $x = a_i$, $2i + 1$ comparisons are used

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$
- to end the loop, one comparison $i \leq n$ is made
- after the loop, one more $i \leq n$ comparison is made
- If $x = a_i$, $2i + 1$ comparisons are used
- If x is not in the list, $2n + 2$ comparisons are made which is the worst case

Worst-Case complexity of linear search

- Count the number of comparisons
- at each step two comparisons are made $i \leq n$ and $x \neq a_i$
- to end the loop, one comparison $i \leq n$ is made
- after the loop, one more $i \leq n$ comparison is made
- If $x = a_i$, $2i + 1$ comparisons are used
- If x is not in the list, $2n + 2$ comparisons are made which is the worst case
- This means that the complexity is $\Theta(n)$

Binary search

```
procedure binary_search(x,a_1,...,a_n)
i:=1
j:=n
while i<j
    m:=⌊(i + j)/2⌋
    if x > a_m
    then i:=m+1
    else j:=m
if x = a_i
then location:=i
else location:=0
return location
```

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$
- Two comparisons are made at each stage $i < j$ and $x > a_m$

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$
- Two comparisons are made at each stage $i < j$ and $x > a_m$
- At the first iteration the size of the list is 2^k; after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$
- Two comparisons are made at each stage $i < j$ and $x > a_m$
- At the first iteration the size of the list is 2^k; after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$
- At the last step, a comparison tells us that the size of the list is $2^0 = 1$ and the element is compared with the single remaining element

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$
- Two comparisons are made at each stage $i < j$ and $x > a_m$
- At the first iteration the size of the list is 2^k; after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$
- At the last step, a comparison tells us that the size of the list is $2^0 = 1$ and the element is compared with the single remaining element
- Hence, at most $2k + 2 = 2log_2 n + 2$ comparisons are made

Worst-Case complexity of binary search

- Assume (for simplicity) $n = 2^k$; so $k = log_2 n$
- Two comparisons are made at each stage $i < j$ and $x > a_m$
- At the first iteration the size of the list is 2^k; after the first iteration it is 2^{k-1}. Then 2^{k-2} and so on until the size of the list is $2^1 = 2$
- At the last step, a comparison tells us that the size of the list is $2^0 = 1$ and the element is compared with the single remaining element
- Hence, at most $2k + 2 = 2 log_2 n + 2$ comparisons are made
- This means that complexity is $\Theta(log\, n)$

Computer time

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-9} sec	10^{-8} sec	3×10^{-8} sec	10^{-7} sec	10^{-6} sec	3×10^{-3} sec
10^2	7×10^{-9} sec	10^{-7} sec	7×10^{-7} sec	10^{-5} sec	4×10^{13} yr	*
10^3	1.0×10^{-8} sec	10^{-6} sec	1×10^{-5} sec	10^{-3} sec	*	*
10^4	1.3×10^{-8} sec	10^{-5} sec	1×10^{-4} sec	10^{-1} sec	*	*
10^5	1.7×10^{-8} sec	10^{-4} sec	2×10^{-3} sec	10 sec	*	*
10^6	2×10^{-8} sec	10^{-3} sec	2×10^{-2} sec	17 min	*	*

However, the time required for an algorithm to solve a problem of a specified size can be determined if all operations can be reduced to the bit operations used by the computer. Table 2 displays the time needed to solve problems of various sizes with an algorithm using the indicated number of bit operations. Times of more than 10^{100} years are indicated with an asterisk. (In Section 2.4 the number of bit operations

15/12/2009

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with P $\subseteq$ NP and which has complete problems such as satisfiability of boolean formulas

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with P $\subseteq$ NP and which has complete problems such as satisfiability of boolean formulas
- Open problem: NP $\subseteq$ P ?

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with $\mathrm{P} \subseteq \mathrm{NP}$ and which has complete problems such as satisfiability of boolean formulas
- Open problem: $\mathrm{NP} \subseteq \mathrm{P}$?
- If there is a polynomial time algorithm for any NP complete problem then $\mathrm{P} = \mathrm{NP}$

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with $\mathrm{P} \subseteq \mathrm{NP}$ and which has complete problems such as satisfiability of boolean formulas
- Open problem: $\mathrm{NP} \subseteq \mathrm{P}$?
- If there is a polynomial time algorithm for any NP complete problem then $\mathrm{P} = \mathrm{NP}$
- There are quick algorithms for testing whether a large integer is prime $O((log\, n)^6)$

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with P $\subseteq$ NP and which has complete problems such as satisfiability of boolean formulas
- Open problem: NP $\subseteq$ P ?
- If there is a polynomial time algorithm for any NP complete problem then P $=$ NP
- There are quick algorithms for testing whether a large integer is prime $O((\log n)^6)$
- How hard is it to factorise integers?

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with $P \subseteq NP$ and which has complete problems such as satisfiability of boolean formulas
- Open problem: $NP \subseteq P$?
- If there is a polynomial time algorithm for any NP complete problem then $P = NP$
- There are quick algorithms for testing whether a large integer is prime $O((log\, n)^6)$
- How hard is it to factorise integers?
- We don't know if it belongs to P (it is in NP)

Further topics

- An algorithm is polynomial time if for some k it is $\Theta(n^k)$
- Tractable problem: there is a polynomial time algorithm that solves it. (Class P is tractable problems)
- Intractable problem: there is no polynomial time algorithm that solves it
- Class NP with $P \subseteq NP$ and which has complete problems such as satisfiability of boolean formulas
- Open problem: $NP \subseteq P$?
- If there is a polynomial time algorithm for any NP complete problem then $P = NP$
- There are quick algorithms for testing whether a large integer is prime $O((log\, n)^6)$
- How hard is it to factorise integers?
- We don't know if it belongs to P (it is in NP)
- It is very unlikely to be NP complete