Scientific Theory in Informatics A1N

UNIVERSITY OF SKÖVDE

Lecture 04

Complexity Theory

David Vernon
School of Informatics
University of Skövde

david.vernon@his.se

Lecture Overview

UNIVERSITY OF SKÖVDE

- Analysis of complexity of algorithms
  - Time complexity
  - Big-O Notation
  - Space complexity
- Introduction to complexity theory
  - P, NP, and NP-Complete classes of algorithm

Motivation

UNIVERSITY OF SKÖVDE

- Why study the Theory of Computation?
  - Determine what can and cannot be computed
  - How quickly & with how much memory
  - On what type of computational model
- Theory guides practice
  - New application-specific programming language … grammars
  - String searching and pattern matching … finite automata and regular expressions
  - Programs take too long to run … complexity analysis and algorithmic strategies
  - Security and cryptography … NP-completeness

Motivation

UNIVERSITY OF SKÖVDE

Theory of Computation normally divided in three parts

1. Automata Theory and Languages
2. Computability Theory
3. Complexity Theory

See: Introduction to the Theory of Computation,
Michael Sipser, 3rd edition, Cengage Learning, 2013
Chapters 1, 2 (Automata Theory)
Chapters 3, 4, 5 (Computability Theory)
Chapter 7, 9 (Complexity Theory)

Motivation

UNIVERSITY OF SKÖVDE

- Complexity Theory
  - Easy problems (sort a million items in a few seconds)
  - Hard problems (schedule a thousand classes in a hundred years)
  - What makes some problems hard and others easy (computationally) and how do we make hard problems easier?
  - Complexity Theory addresses these questions

Motivation

UNIVERSITY OF SKÖVDE

- Computability Theory
  - In the first half of the 20th century, mathematicians such as Kurt Gödel, Alan Turing, and Alonzo Church discovered that certain basic problems cannot be solved by computers
    - » e.g. determine whether a mathematical statement is true or false
  - Complexity Theory: classify problems as easy ones and hard ones
  - Computability Theory: classify problems are solvable and not solvable

Motivation

UNIVERSITY OF SKÖVDE

- Automata Theory
  - Deals with the definitions and properties of mathematical models of computation
  - Finite automaton (used in text processing, compilers, hardware design)
  - Context-free grammar (used in programming languages and artificial intelligence)

Complexity Analysis

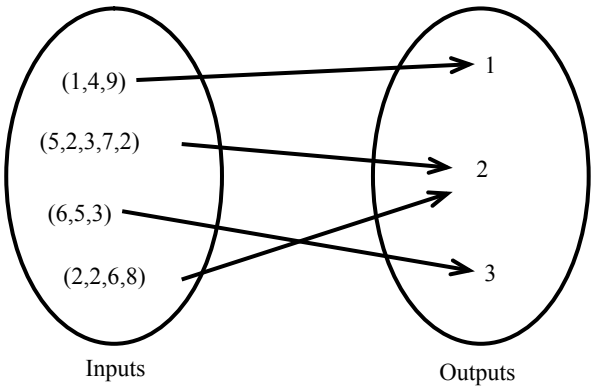
UNIVERSITY OF SKÖVDE

- Why do we write programs?
  - to perform some specific tasks
  - to solve some specific problems
  - We will focus on “solving problems”
  - What is a “problem”?
  - We can view a problem as a mapping of “inputs” to “outputs”

Complexity Analysis

UNIVERSITY OF SKÖVDE

For example, Find Minimum



Complexity Analysis

UNIVERSITY OF SKÖVDE

- How to describe a problem?
  - Input
    - Describe what an input looks like
  - Output
    - Describe what an output looks like and how it relates to the input

Complexity Analysis

UNIVERSITY OF SKÖVDE

- An instance is an assignment of values to the input variables

An instance of the Find Minimum function

$N = 10$
$(a_1, a_2, \ldots, a_N) = (5,1,7,4,3,2,3,3,0,8)$

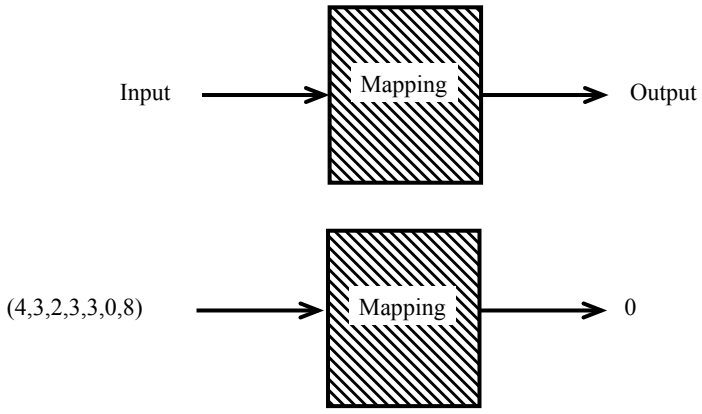
Another instance of the Find Minimum Problem

$N = 10$
$(a_1, a_2, \ldots, a_N) = (15,8,0,4,7,2,5,10,1,4)$

Complexity Analysis

UNIVERSITY OF SKÖVDE

A problem can be considered as a black box



Complexity Analysis

UNIVERSITY OF SKÖVDE

Example: Sorting

Input: A sequence of N numbers $a_1 \ldots a_n$

Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$

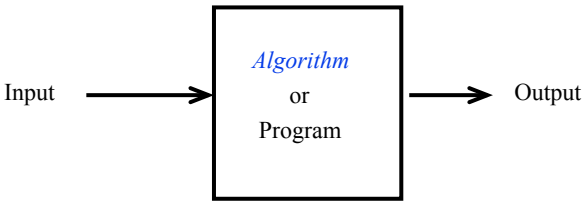
Complexity Analysis

UNIVERSITY OF SKÖVDE

How do we solve a problem?

Write an algorithm that implements the mapping

Takes an *input* in and produces a correct *output*



Complexity Analysis

UNIVERSITY OF SKÖVDE

- How do we judge whether an algorithm is good or bad?
- Analyse their efficiency
  - Determined by the amount of computer resources consumed by the algorithm
- What are the important resources?

• Amount of memory (space complexity) • Amount of computational time (time complexity)

Complexity Analysis

UNIVERSITY OF SKÖVDE

Consider the amount of resources

memory space and time

that an algorithm consumes

as a function of the size of the input to the algorithm.

Complexity Analysis

UNIVERSITY OF SKÖVDE

- Suppose there is an assignment statement in your program

  x := x +1

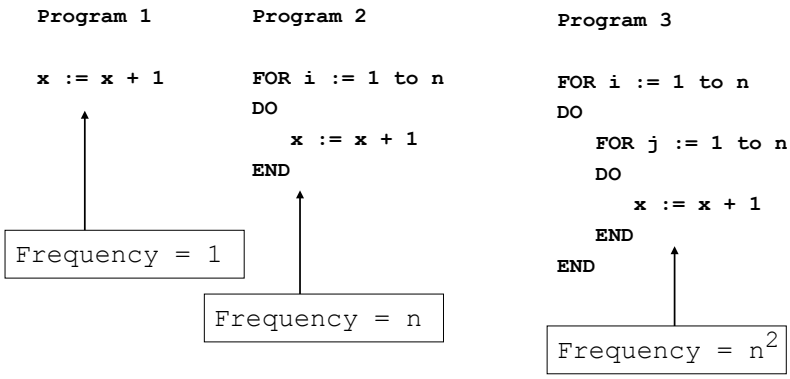
- We'd like to determine:
  - The time a single execution would take
  - The number of times it is executed: Frequency Count

Time Complexity

UNIVERSITY OF SKÖVDE

- Product of execution time and frequency is *approximately* the total time taken
- But, since the execution time will be very machine dependent (and compiler dependent), we neglect it and concentrate on the frequency count
- Frequency count will vary from data set to data set (*input to the algorithm*)

Time Complexity



Time Complexity

- Program 1
  - statement is not contained in a loop (implicitly or explicitly)
  - Frequency count is 1
- Program 2
  - statement is executed *n* times
- Program 3
  - statement is executed n^2 times

Big-O Notation

UNIVERSITY OF SKÖVDE

- 1, n, and n^2 are said to be different and increasing orders of magnitude

  (e.g. let $n = 10 \Rightarrow 1, 10, 100$)

- We are interested in determining the order of magnitude of the time complexity of an algorithm

Big-O Notation

UNIVERSITY OF SKÖVDE

- Let's look at an algorithm to print the n^{th} term of the Fibonnaci sequence

  0 1 1 2 3 5 8 13 21 34 …
  $t_n = t_{n-1} + t_{n-2}$
  $t_0 = 0$
  $t_1 = 1$

Big-O Notation


```
1       procedure fibonacci {print nth term}
2          read(n)
3             if n<0
4                then print(error)
5                else if n=0
6                   then print(0)
7                   else if n=1
8                      then print(1)
9                    else
10                        fnm2 := 0;
11                       fnm1 := 1;
12                       FOR i := 2 to n DO
13                          fn := fnm1 + fnm2;
14                          fnm2 := fnm1;
15                          fnm1 := fn
16                       end
17                    print(fn);
```

step	n<0
1	1
2	1
3	1
4	1
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Big-O Notation


```
1       procedure fibonacci {print nth term}
2          read(n)
3             if n<0
4                then print(error)
5                else if n=0
6                   then print(0)
7                   else if n=1
8                      then print(1)
9                    else
10                        fnm2 := 0;
11                       fnm1 := 1;
12                       FOR i := 2 to n DO
13                          fn := fnm1 + fnm2;
14                          fnm2 := fnm1;
15                          fnm1 := fn
16                       end
17                    print(fn);
```

step	n=0
1	1
2	1
3	1
4	0
5	1
6	1
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Big-O Notation


```
                                                    step   n=1
1       procedure fibonacci {print nth term}          1      1
2          read(n)                                    2      1
3             if n<0                                  3      1
4                then print(error)                    4      0
5                else if n=0                          5      1
6                   then print(0)                     6      0
7                   else if n=1                       7      1
8                      then print(1)                  8      1
9                    else                             9      0
10                        fnm2 := 0;                  10     0
11                       fnm1 := 1;                   11     0
12                       FOR i := 2 to n DO           12     0
13                          fn := fnm1 + fnm2;        13     0
14                          fnm2 := fnm1;             14     0
15                          fnm1 := fn                15     0
16                       end                          16     0
17                     print(fn);                     17     0
```


Big-O Notation


```
                                                    step   n>1
1       procedure fibonacci {print nth term}          1      1
2          read(n)                                    2      1
3             if n<0                                  3      1
4                then print(error)                    4      0
5                else if n=0                          5      1
6                   then print(0)                     6      0
7                   else if n=1                       7      1
8                      then print(1)                  8      0
9                    else                             9      1
10                        fnm2 := 0;                  10     1
11                       fnm1 := 1;                   11     1
12                       FOR i := 2 to n DO           12     n-1
13                          fn := fnm1 + fnm2;        13     n-2
14                          fnm2 := fnm1;             14     n-2
15                          fnm1 := fn                15     n-2
16                       end                          16     0
17                     print(fn);                     17     1
```

Big-O Notation

step	n<0	n=0	n=1	n>1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	1
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	n-1
13	0	0	0	n-2
14	0	0	0	n-2
15	0	0	0	n-2
16	0	0	0	0
17	0	0	0	1

Big-O Notation

- The cases where $n<0$, $n=0$, $n=1$ are not particularly instructive or interesting
- In the case where $n>1$, we have the total statement frequency of

  $9 + (n-1) + 3(n-2) = 4n + 2$

Big-O Notation

UNIVERSITY OF SKÖVDE

- $4n + 2$
- We write this as $O(n)$, ignoring the constants
- *This is called Big-O notation*
- More formally, $f(n) = O(g(n))$
  where $g(n)$ is an **asymptotic upper bound** for $f(n)$

Big-O Notation

UNIVERSITY OF SKÖVDE

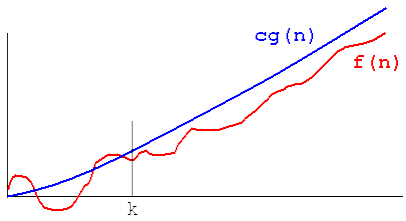
- The notation $f(n) = O(g(n))$ has a precise mathematical definition
- Read $f(n) = O(g(n))$ as
  f of *n* is big-O of *g* of *n*
- Definition:
  Let $f, g: Z^+ \rightarrow R^+$

  $f(n) = O(g(n))$ if there exist two constants c and k such that $f(n) \leq c\, g(n)$ for all $n \geq k$

Big-O Notation

Suppose $f(n)= 2n^2+4n+10$

and $f(n) = O(g(n))$ where $g(n)=n^2$



Proof:

$f(n)= 2n^2+4n+10$

$f(n) \leq 2n^2+ 4n^2+10n^2$ for $n \geq 1$

$f(n) \leq 16n^2$

$f(n) \leq 16g(n)$ Where c= 16 and $k = 1$

Time & Space Complexity

- $f(n)$ will normally represent the computing time of some algorithm

  Time complexity $T(n)$

- $f(n)$ can also represent the amount of memory an algorithm will need to run

  Space complexity $S(n)$

Time Complexity

- If an algorithm has a time complexity of $O(g(n))$ it means that its execution will take no longer than a constant times $g(n)$
- More formally, g(n) is an **asymptotic upper bound** for *f(n)*

Remember

- $f(n) \leq c\ g(n)$

  n is typically the size of the data set

Time Complexity

$O(1)$	Constant (computing time)
$O(n)$	Linear (computing time)
$O(n^2)$	Quadratic (computing time)
$O(n^3)$	Cubic (computing time)
$O(2^n)$	Exponential (computing time)
$O(\log n)$	is faster than $O(n)$ for sufficiently large n
$O(n \log n)$	is faster than $O(n^2)$ for sufficiently large n

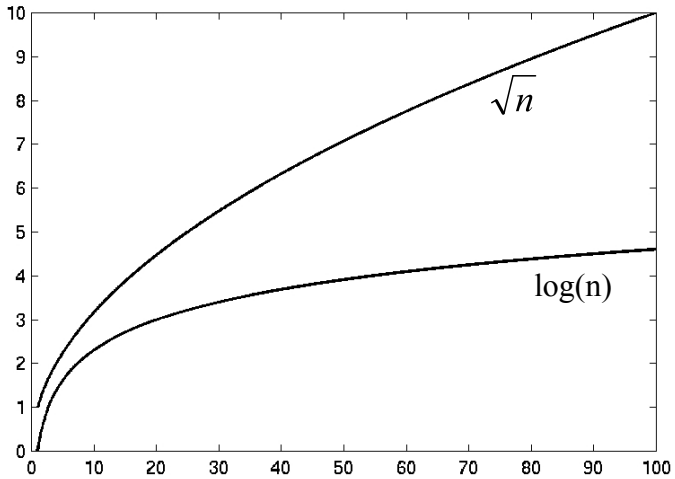
Time Complexity

UNIVERSITY OF SKÖVDE

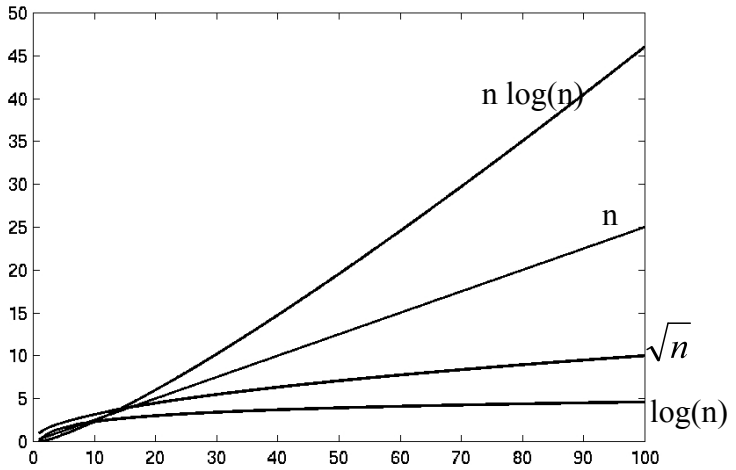
n	O(1)	O(log2(n))	O(n)	O(nlog2(n))	O(n^2)	O(n^3)	O(n^4)	O(2^n)	O(n^n)
1	7	0.0	1	0.0	1	1	1	2	1
2	7	1.0	2	2.0	4	8	16	4	4
3	7	1.6	3	4.8	9	27	81	8	27
4	7	2.0	4	8.0	16	64	256	16	256
5	7	2.3	5	11.6	25	125	625	32	3125
6	7	2.6	6	15.5	36	216	1296	64	46656
7	7	2.8	7	19.7	49	343	2401	128	823543
8	7	3.0	8	24.0	64	512	4096	256	16777216
9	7	3.2	9	28.5	81	729	6561	512	3.87E+08
10	7	3.3	10	33.2	100	1000	10000	1024	1E+10
11	7	3.5	11	38.1	121	1331	14641	2048	2.85E+11
12	7	3.6	12	43.0	144	1728	20736	4096	8.92E+12
13	7	3.7	13	48.1	169	2197	28561	8192	3.03E+14
14	7	3.8	14	53.3	196	2744	38416	16384	1.11E+16
15	7	3.9	15	58.6	225	3375	50625	32768	4.38E+17
16	7	4.0	16	64.0	256	4096	65536	65536	1.84E+19
17	7	4.1	17	69.5	289	4913	83521	131072	8.27E+20
18	7	4.2	18	75.1	324	5832	104976	262144	3.93E+22
19	7	4.2	19	80.7	361	6859	130321	524288	1.98E+24
20	7	4.3	20	86.4	400	8000	160000	1048576	1.05E+26
21	7	4.4	21	92.2	441	9261	194481	2097152	5.84E+27
22	7	4.5	22	98.1	484	10648	234256	4194304	3.41E+29
23	7	4.5	23	104.0	529	12167	279841	8388608	2.09E+31
24	7	4.6	24	110.0	576	13824	331776	16777216	1.33E+33
25	7	4.6	25	116.1	625	15625	390625	33554432	8.88E+34
26	7	4.7	26	122.2	676	17576	456976	67108864	6.16E+36
27	7	4.8	27	128.4	729	19683	531441	1.34E+08	4.43E+38
28	7	4.8	28	134.6	784	21952	614656	2.68E+08	3.31E+40
29	7	4.9	29	140.9	841	24389	707281	5.37E+08	2.57E+42
30	7	4.9	30	147.2	900	27000	810000	1.07E+09	2.06E+44

Time Complexity

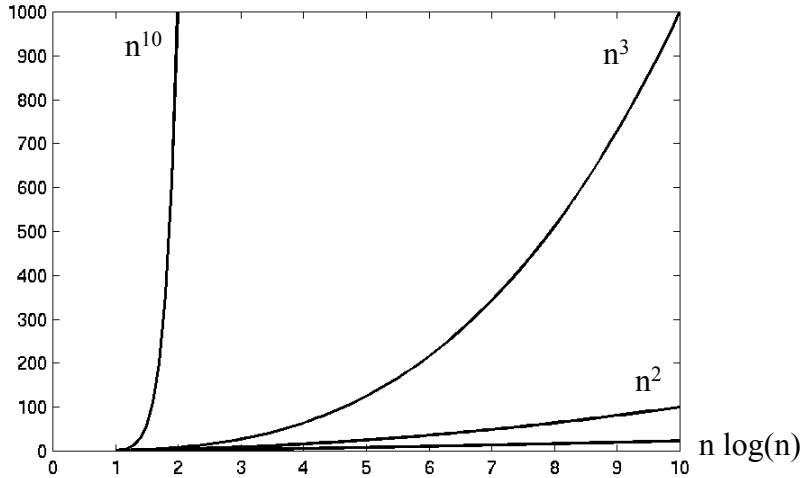
UNIVERSITY OF SKÖVDE



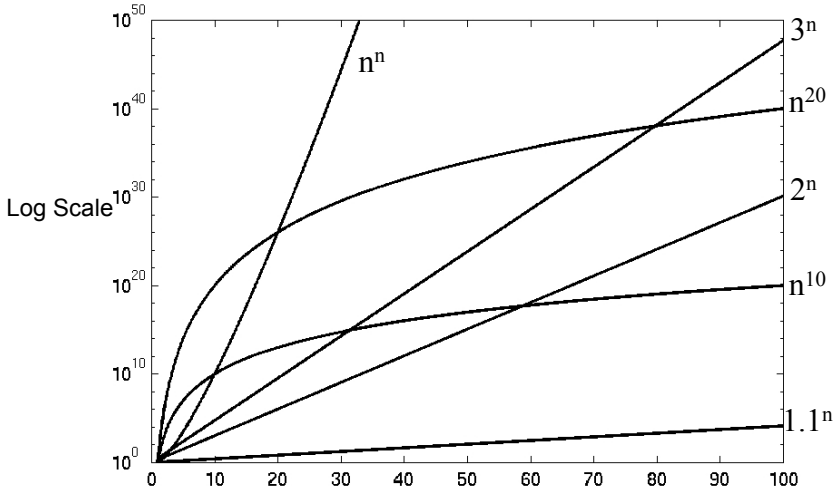
Time Complexity



Time Complexity



Time Complexity



Time Complexity

$f1(n) = 10\ n + 25\ n^2$	$O(n^2)$
$f2(n) = 20\ n \log n + 5\ n$	$O(n \log n)$
$f3(n) = 12\ n \log n + 0.05\ n^2$	$O(n^2)$
$f4(n) = n^{1/2} + 3\ n \log n$	$O(n \log n)$

Time Complexity

UNIVERSITY OF SKÖVDE

- Arithmetic of Big-O notation

  if

  $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

  then

  $T_1(n) + T_2(n) = O(max(f(n), g(n))$

Time Complexity

UNIVERSITY OF SKÖVDE

- Arithmetic of Big-O notation

  if

  $f(n) \leq g(n)$

  then

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

Time Complexity

UNIVERSITY OF SKÖVDE

- Arithmetic of Big-O notation

  if

  $$T_1(n) = O(f(n)) \text{ and } T_2(n) = O(g(n))$$

  then

  $$T_1(n)\ T_2(n) = O(f(n)\ g(n))$$

Time Complexity

UNIVERSITY OF SKÖVDE

- Rules for computing the time complexity
  - the complexity of each read, write, and assignment statement can be taken as $O(1)$
  - the complexity of a sequence of statements is determined by the summation rule
  - the complexity of an if statement is the complexity of the executed statements, plus the time for evaluating the condition

Time Complexity

UNIVERSITY OF SKÖVDE

- Rules for computing the time complexity
  - the complexity of an if-then-else statement is the time for evaluating the condition plus the larger of the complexities of the then and else clauses
  - the complexity of a loop is the sum, over all the times around the loop, of the complexity of the body and the complexity of the termination condition

Time Complexity

UNIVERSITY OF SKÖVDE

- Given an algorithm, we analyse the frequency count of each statement and total the sum
- This may give a polynomial $P(n)$:

$$P(n) = c_k n^k + c_{k-1} n^{k-1} + ...+ c_1 n + c_0$$

where the c_i are constants, c_k are non-zero, and n is a parameter

Time Complexity

- If the big-O notation of a portion of an algorithm is given by:

  $P(n) = O(n^k)$

  and on the other hand, if any other step is executed 2^n times or more, we have:

  $c\ 2^n + P(n) = O(2^n)$

Time Complexity

- What about computing the complexity of a recursive algorithm?
- In general, this is more difficult
- The basic technique
  - Identify a recurrence relation implicit in the recursion

    $T(n) = f(T(k)),\ k \in \{1, 2, \ldots, n\text{-}1\}$

  - Solve the recurrence relation by finding an expression for $T(n)$ in term which do not involve $T(k)$

Time Complexity

UNIVERSITY OF SKÖVDE

```
int factorial(int n) {
   int factorial_value;

   factorial_value = 0;

   /* compute factorial value recursively */

   if (n <= 1) {
      factorial_value = 1;
   }
   else {
      factorial_value = n * factorial(n-1);
   }
   return (factorial_value);
}
```


Time Complexity

UNIVERSITY OF SKÖVDE

Let the time complexity of the function be $T(n)$

… which is what we want to compute!

Now, let's try to analyse the algorithm

Time Complexity

```
                                                          n>1
int factorial(int n)
{
   int factorial_value;                                    1

   factorial_value = 0;                                    1

   if (n <= 1) {                                           1
      factorial_value = 1;                                 0
   }
   else {                                                  1
      factorial_value = n * factorial(n-1);                T(n-1)
   }
   return (factorial_value);                               1
}
```


Time Complexity

$T(n) = 5 + T(n-1)$
$T(n) = c + T(n-1)$
$T(n-1) = c + T(n-2)$
$T(n) = c + c + T(n-2)$
$= 2c + T(n-2)$
$T(n-2) = c + T(n-3)$
$T(n) = 2c + c + T(n-3)$
$= 3c + T(n-3)$
$T(n) = ic + T(n-i)$

Time Complexity

$T(n) = ic + T(n\text{-}i)$

Finally, when $i = n\text{-}1$

$$T(n) = (n\text{-}1)c + T(n\text{-}(n\text{-}1)) = (n\text{-}1)c + T(1) = (n\text{-}1)c + d$$

Hence, $T(n) = O(n)$

Space Complexity

Compute the space complexity of an algorithm by analysing the storage requirements (as a function on the input size) in the same way

Space Complexity

UNIVERSITY OF SKÖVDE

For example

- if you read a stream of *n* characters
- and only ever store a constant number of them,
- then it has space complexity *O*(1)

Space Complexity

UNIVERSITY OF SKÖVDE

For example

- if you read a stream of n records
- and store all of them,
- then it has space complexity O(n)

Space Complexity

For example

- if you read a stream of n records
- and store all of them,
- and each record causes the creation of (a constant number) of other records,
- then it still has space complexity O(n)

Space Complexity

For example

- if you read a stream of n records
- and store all of them,
- and each record causes the creation of a number of other records (and the number is proportional to the size of the data set n)
- then it has space complexity $O(n^2)$

Time vs Space Complexity

In general, we can often decrease the time complexity but this will involve an increase in the space complexity

and *vice versa* (decrease space, increase time)

This is the *time-space tradeoff*

Time vs Space Complexity

For example

- the average time complexity of an iterative sort (e.g. bubble sort) is $O(n^2)$
- but we can do better:
- the average time complexity of the Quicksort is $O(n \log n)$
- But the Quicksort is recursive and the recursion causes an increase in memory requirements (*i.e.* an increase in space complexity)

Time vs Space Complexity

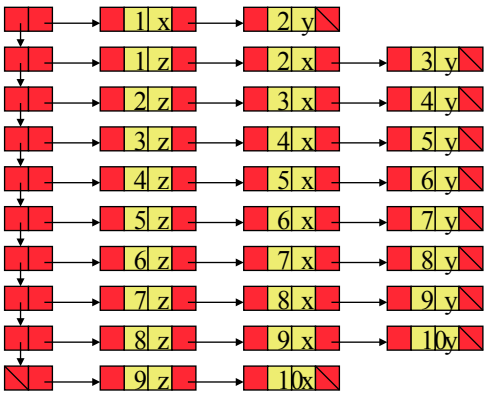
For example

- The space complexity of 2-D matrix is O(n^2)
- If the matrix is sparse we can do better: we can represent the matrix as a 2-D linked list and often reduce the space complexity to O(n)
- But the time taken to access each element will rise (*i.e.* the time complexity will rise)

Time vs Space Complexity

x	y	0	0	0	0	0	0	0	0
z	x	y	0	0	0	0	0	0	0
0	z	x	y	0	0	0	0	0	0
0	0	z	x	y	0	0	0	0	0
0	0	0	z	x	y	0	0	0	0
0	0	0	0	z	x	y	0	0	0
0	0	0	0	0	z	x	y	0	0
0	0	0	0	0	0	z	x	y	0
0	0	0	0	0	0	0	z	x	y
0	0	0	0	0	0	0	0	z	x

n x n matrix:
O(n^2) space complexity



2x(2 + 4 + 4) + (n-2)x(2 + 4 + 4 + 4)
= 20 + 14n - 28 = 14n - 8:
O(n) space complexity

Time vs Space Complexity

UNIVERSITY OF SKÖVDE

Order of space complexity for the matrix representation of the banded matrix is $O(n^2)$ >> order of space complexity for the linked list representation $O(n)$

However, the matrix implementation will sometimes be more effective:

Time vs Space Complexity

UNIVERSITY OF SKÖVDE

$n^2 <= 14n - 8$

$n^2 - 14n + 8 <= 0$

$n = \pm 13$ is the cutoff at which the list representation is more efficient in terms of storage space

Typically, in real engineering problems, n can be much greater than 100 and the saving is very significant

Worst-case and average-case complexity

So far we have looked only at worst-case complexity (i.e. we have developed an upper-bound on complexity)

However, there are times when we are more interested in the average-case complexity (especially it differs significantly)

Worst-case and average-case complexity

For example

the Quicksort algorithm has

$T(n) = O(n^2)$, worst case (for inversely sorted data)

$T(n) = O(n \log_2 n)$, average case (for randomly ordered data)

Complexity Theory: P, NP, NP-complete

The following slides are adapted from notes by
Simonas Šaltenis, Aalborg University

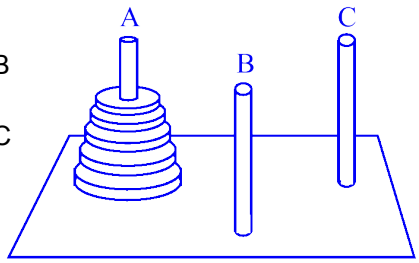
Complexity and Intractability

- Tractable and intractable problems
  - What is a ”reasonable” running time?
  - NP problems, examples
  - NP-complete problems and polynomial reducibility

Towers of Hanoi

UNIVERSITY OF SKÖVDE

- **Goal:** transfer all *n* disks from peg A to peg C
- Rules:
  - move one disk at a time
  - never place larger disk above smaller one
- Recursive solution:
  - transfer $n - 1$ disks from A to B
  - move largest disk from A to C
  - transfer $n - 1$ disks from B to C
- Total number of moves:
  - $T(n) = 2T(n - 1) + 1$



Towers of Hanoi

UNIVERSITY OF SKÖVDE

- Recurrence relation:
  $T(n) = 2\ T(n - 1) + 1$
  $T(1) = 1$
- Solution by unfolding:
  $T(n) = 2\ (2\ T(n - 2) + 1) + 1 =$
  $= 4\ T(n - 2) + 2 + 1 =$
  $= 4\ (2\ T(n - 3) + 1) + 2 + 1 =$
  $= 8\ T(n - 3) + 4 + 2 + 1 = ...$
  $= 2^i\ T(n - i) + 2^{i-1} + 2^{i-2} + ... + 2^1 + 2^0$
- the expansion stops when $i = n - 1$
  $T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0$

Towers of Hanoi

- This is a **geometric sum**, so that we have
  $T(n) = 2^n - 1 = O(2^n)$
- The running time of this algorithm is **exponential** (k^n) rather than **polynomial** (n^k)
- Good or bad news?
  - the Tibetan monks were confronted with a tower of 64 rings...
  - assuming one could move **1 million rings per second**, it would take **half a million years** to complete the process...

Aside: Recursive Programming

- Divide
  - Break the problem into several problems that are similar to the original problem but smaller in size
- Conquer
  - Solve the sub-problems recursively, or,
  - If they are small enough, solve them directly
- Combine the solutions to the sub-problems into a solution of the original problem

Aside: Recursive Programming

UNIVERSITY OF SKÖVDE

Factorial

n! = n x (n-1) x (n-2) x … x 1

Also given by the recurrence formula

$f_n = n \times f_{n-1}$ n>0
$f_0 = 1$

In other words

n! = n x (n-1)! n>0
0! = 1

Aside: Recursive Programming

UNIVERSITY OF SKÖVDE

```
int factorial(int n) { // assume n >= 0
   if (n == 0)
      return(1);
   else
      return(n x factorial(n-1));
   }
}
```

Aside: Recursive Programming

Fibonnaci Sequence

Given by the recurrence formula

$f_0 = 1$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ n>= 3

Aside: Recursive Programming


```
int fibonnaci_number(int n) { // assume n >= 0
   if (n == 0 || n == 1)
      return(1);
   else
      return(fibonnaci_number(n-1) + fibonnaci_number(n-2));
   }
}
```

Aside: Recursive Programming

UNIVERSITY OF SKÖVDE

Tower of Hanoi

The objective of the puzzle is to move the entire stack to another peg, obeying the following rules:

- Only one disk may be moved at a time
- Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg, on top of the other disks that may already be present on that peg
- No disk may be placed on top of a smaller disk.

Aside: Recursive Programming

UNIVERSITY OF SKÖVDE

```
void hanoi(int n, char a, char b, char c) {
  if (n > 0) {
      hanoi(n-1, a, c, b);
      printf("Move disk of diameter %d from %c to %c\n", n, a, b);
      hanoi(n-1, c, b, a);
   }
}

…

Hanoi(5, 'A', 'B', 'C');
```

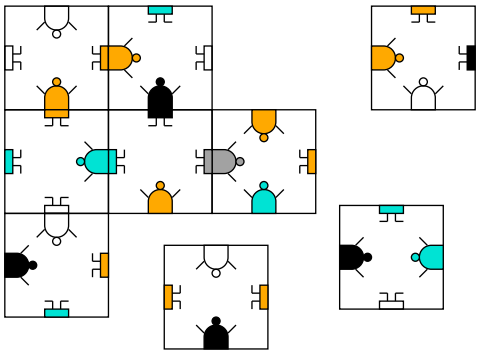
Monkey Puzzle

UNIVERSITY OF SKÖVDE

Are such long running times linked to the size of the solution of an algorithm?

No. To show that, we in the following consider only TRUE/FALSE or yes/no problems – decision problems

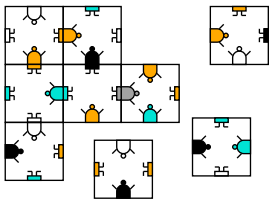
- Nine square cards with imprinted “monkey halfs”
- The goal is to arrange the cards in 3x3 square with matching halfs...



Monkey Puzzle

UNIVERSITY OF SKÖVDE

- Assumption: orientation is fixed
- Does any MxM arrangement exist that fulfills the matching criterion?
- Brute-force algorithm would take ***n*!** **times** to verify whether a solution exists (why?)
  - assuming n = 25, it would take 490 billion years on a one-million-per- second arrangements computer to verify whether a solution exists



Monkey Puzzle

UNIVERSITY OF SKÖVDE

- Assume *n*, the number of cards, is 25
- The size of the final square is 5x5

Monkey Puzzle

UNIVERSITY OF SKÖVDE

- Brute force solution:
  - Go through all possible arrangements of the cards
  - pick a card and place it - there are 25 possibilities for the first placement
  - pick the next card and place it - there are 24 possibilities
  - Pick the next card, there are 23 possibilities ...

Monkey Puzzle

- There are 25x24x23x22x.....x2x1 possible arrangements
- That is, there are factorial 25 possible arrangements (25!)
- 25! contains 26 digits
- If we make 1000000 arrangements per second, the algorithm will take 490 000 000 000 years to complete

Monkey Puzzle

- Improving the algorithm
  - discarding partial arrangements (backtracking)
  - etc.
- A smart algorithm would still take a couple of thousand years in the worst case
- Is there an easier way to find solutions?
  Perhaps, but nobody has found them, yet …

Complexity and Intractability

- We classify functions as ‘good’ and ‘bad’
- Polynomial functions are good
- Super-polynomial (or exponential) functions are bad

Complexity and Intractability

- The order of complexity of this algorithm is $O(n!)$
- $n!$ grows at a rate which is orders of magnitude larger than the growth rate of the other functions we mentioned before

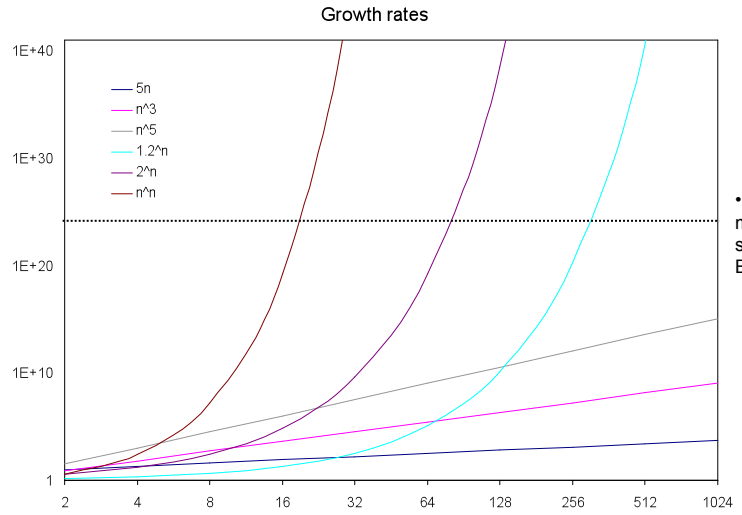
Complexity and Intractability

UNIVERSITY OF SKÖVDE

- Other functions exist that grow even faster, e.g. n^n (super-exponential)
- Even functions like 2^n exhibit unacceptable sizes even for modest values of n

Reasonable vs. Unreasonable

UNIVERSITY OF SKÖVDE



•Number of microseconds since "Big-Bang"

Reasonable vs. Unreasonable

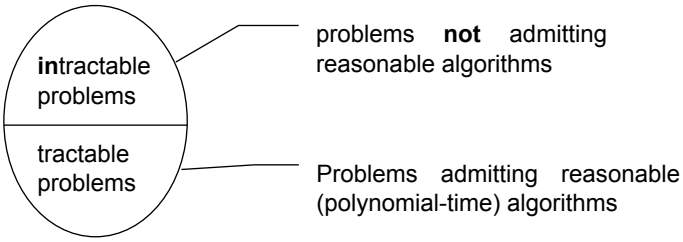
UNIVERSITY OF SKÖVDE

	function/ n	10	20	50	100	300
Polynomial	n^2	1/10,000 second	1/2,500 second	1/400 second	1/100 second	9/100 second
Polynomial	n^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
Exponential	2^n	1/1000 second	1 second	35.7 years	400 trillion centuries	a 75 digit-number of centuries
Exponential	n^n	2.8 hours	3.3 trillion years	a 70 digit-number of centuries	a 185 digit-number of centuries	a 728 digit-number of centuries

Reasonable vs. Unreasonable

UNIVERSITY OF SKÖVDE

- ”Good”, reasonable algorithms
  - Algorithms bound by a polynomial function n^k
  - **Tractable problems**
- ”Bad”, unreasonable algorithms
  - Algorithms whose running time is above n^k
  - **Intractable problems**



So What!

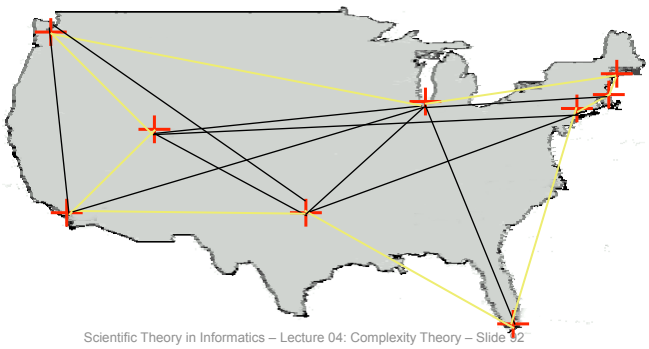
UNIVERSITY OF SKÖVDE

- Computers become faster every day
  - Doesn't matter: insignificant (a constant) compared to exp. running time
- Maybe the Monkey puzzle is just one specific one we could simply ignore
  - the monkey puzzle falls into a category of problems called NPC (NP complete) problems (~1000 problems)
  - all admit **unreasonable** solutions
  - **not known** to admit **reasonable** ones…

Travelling Salesman Problem (TSP)

UNIVERSITY OF SKÖVDE

- TSP is the problem of a salesman who wants to find, starting from his home town, a shortest possible trip through a given set of customer cities and to return to its home town; visiting exactly once each city



Travelling Salesman Problem (TSP)

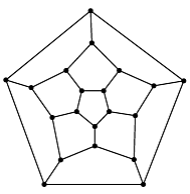
UNIVERSITY OF SKÖVDE

- Naive solutions take *n*! time in worst-case, where *n* is the number of edges of the graph
- No polynomial-time algorithms are known
  - TSP is an NP-complete problem
- Longest Path problem between A and B in a weighted graph is also NP-complete

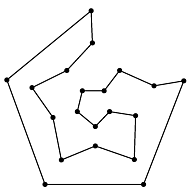
TSP & Hamiltonian

UNIVERSITY OF SKÖVDE

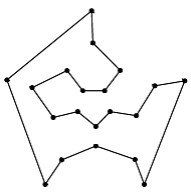
- An Hamiltonian circuit for a given graph ***G=(V, E)*** consists on finding an ordering of the vertices of the graph ***G*** such that each vertex is visited exactly once



Typical Input for HCP



Hamiltonian cycle for the graph



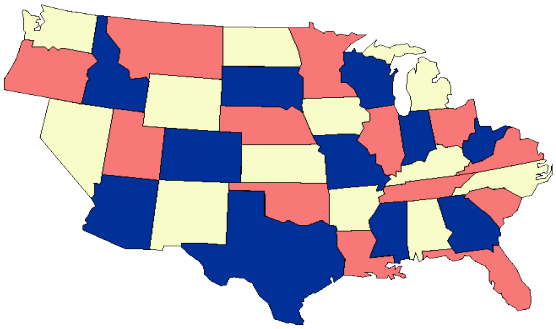
Another Hamiltonian cycle for the same graph in

Coloring Problem

UNIVERSITY OF SKÖVDE

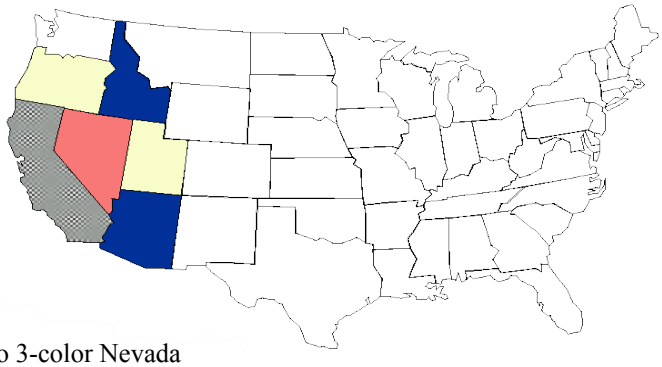
- 3-colour
  - given a planar map, can it be colored using 3 colors so that no adjacent regions have the same color

YES instance



Coloring Problem

UNIVERSITY OF SKÖVDE



NO instance
Impossible to 3-color Nevada and bordering states

Coloring Problem

- Any map can be **4-colored**
- Maps that contain no points that are the junctions of an odd number of states can be **2-colored**
- No polynomial algorithms are known to determine whether a map can be **3-colored** – it's an NP-complete problem

Satisfiability (SAT)

- Determine the truth or falsity of formulae in Boolean algebra (or, equivalently, in propositional calculus)
- Using Boolean variables and operators

  $\wedge$ (and)
  $\vee$ (or)
  ~ (not)

  we compose formula such as the following

  $\phi = (\sim x \wedge y) \vee (x \wedge \sim z)$

Satisifiability (SAT)

UNIVERSITY OF SKÖVDE

- The algorithmic problem calls for determining the **satisfiability** of such formulae

  Is there some assignment of value to x, y, and z for which ϕ evaluates to 1 (TRUE)

  x = 0, y = 1, z = 0 makes $\phi = (\sim x \wedge y) \vee (x \wedge \sim z)$ evaluate to 1

- Exponential time algorithm on n = the number of distinct elementary assertions ($O(2^n)$)

- Best known solution, problem is in NP-complete class

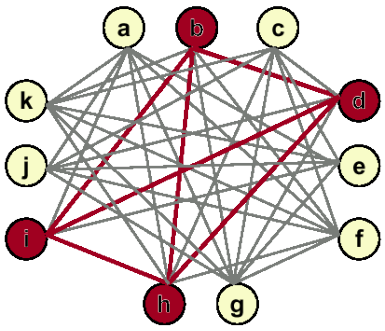
CLIQUE

UNIVERSITY OF SKÖVDE

- Given n people and their pairwise relationships, is there a group of s people such that every pair in the group knows each other

  - people: a, b, c, …, k
  - friendships: (a,e), (a,f),…
  - clique size: s = 4?
  - YES, {b, d, i, h} is a ***certificate***

Friendship Graph



P

UNIVERSITY OF SKÖVDE

- Definition of P:
  - Set of all decision problems solvable in polynomial time on a deterministic Turing machine
- Examples
  - MULTIPLE: Is the integer y a multiple of x?
    - » YES: (x, y) = (17, 51)
  - RELPRIME: Are the integers x and y relatively prime?
    - » YES: (x, y) = (34, 39)
  - MEDIAN: Given integers x_1 , …, x_n , is the median value < M?
    - » YES: (M, x_1 , x_2 , x_3 , x_4 , x_5) = (17, 2, 5, 17, 22, 104)

P

UNIVERSITY OF SKÖVDE

- P is the set of all decision problems solvable in polynomial time on **REAL** computers.

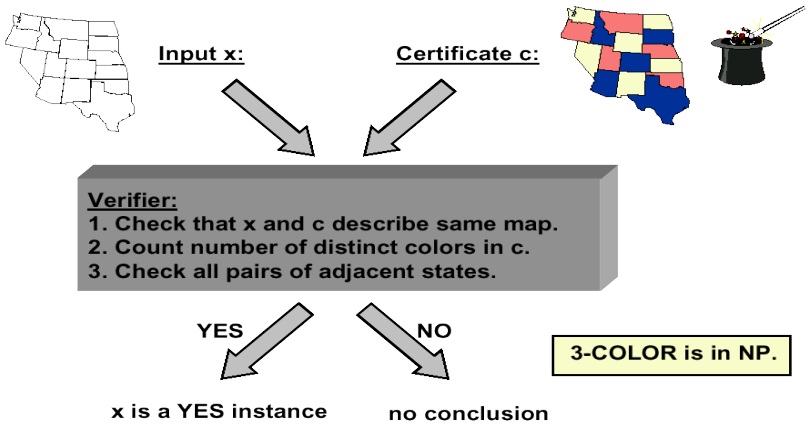
Certificates

UNIVERSITY OF SKÖVDE

- To find a solution for an NPC problem, we seem to be required to try out exponential amounts of partial solutions
- Failing in extending a partial solution requires **backtracking**
- However, once we found a solution, convincing someone of it is easy, if we keep a proof, i.e., a **certificate**
- The problem is finding an answer (exponential), but not verifying a potential solution (polynomial)

Certificates

UNIVERSITY OF SKÖVDE



Non-deterministic ….

- Assume we use a magic coin in the backtracking algorithm
  - whenever it is possible to extend a partial solutions in “two” ways, we perform a coin toss (two monkey cards, next truth assignment, etc.)
  - the outcome of this “act” determines further actions –
  - we use magical insight (guess right every time)
- Such algorithms are termed “**non-deterministic**”
  - they **guess** which option is better, **rather** than employing some **deterministic procedure** to go through the alternatives

NP

- Definition of NP:
  - Set of all decision problems solvable in polynomial time on a nondeterministic Turing machine
  - Important definition because it links many fundamental problems
- Useful alternative definition
  - Set of all decision problems with efficient verification algorithms
    - » efficient = polynomial number of steps on deterministic TM
  - Verifier: algorithm for decision problem with extra input

NP

UNIVERSITY OF SKÖVDE

- NP = set of decision problems with efficient verification algorithms
- Why doesn't this imply that all problems in NP can be solved efficiently?
  - BIG PROBLEM: need to know certificate ahead of time
    - » real computers can simulate by guessing all possible certificates and verifying
    - » naïve simulation takes exponential time unless you get "lucky"

NP-Completeness

UNIVERSITY OF SKÖVDE

- Informal definition of NP-hard:
  - A problem with the property that if it can be solved efficiently, then it can be used as a subroutine to solve any other problem in NP efficiently
- NP-complete problems are NP problems that are NP-hard
  - "Hardest computational problems" in NP

NP-Completeness

- A problem B is NP-complete if it satisfies two conditions
  - B is in NP
  - Every problem A in NP is polynomial time reducible to B

NP-Completeness

- Each NPC problem's fate is tightly coupled to all the others (complete set of problems)
- Finding a **polynomial time algorithm for one NPC problem** would **automatically** yield an a polynomial time algorithm **for all NP problems**
- Proving that one NP-complete problem has an **exponential lower bound** would **automatically** prove that **all other NP-complete** problems have exponential lower bounds

CLIQUE is NP-complete

UNIVERSITY OF SKÖVDE

- CLIQUE is NP-complete
  - CLIQUE is in NP
  - SAT is in NP-complete
  - SAT reduces to CLIQUE
  - CLIQUE is NP-complete
- Hundreds of problems can be shown to be NP-complete that way…

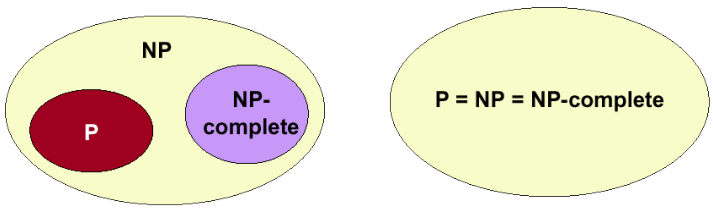
The Big Question

UNIVERSITY OF SKÖVDE

- Does P = NP?

  Is the original DECISION problem as easy as VERIFICATION?

- Most important open problem in theoretical computer science. Clay Institute of Mathematics offers $1m prize



If $P \neq NP$ If P = NP

The Big Question

UNIVERSITY OF SKÖVDE

- If P=NP, then
  - There are efficient algorithms for TSP and factoring
  - Cryptography is impossible on conventional machines
  - Modern banking system will collapse
- If not, then
  - Can't hope to write efficient algorithm for TSP
  - But maybe efficient algorithm still exists for testing the primality of a number – i.e., there are some problems that are NP, but not NP-complete

The Answer?

UNIVERSITY OF SKÖVDE

- Probably no, since
  - Thousands of researchers have spent four decades in search of polynomial algorithms for many fundamental NP-complete problems without success
  - Consensus opinion: $P \neq NP$
- But maybe yes, since
  - No success in proving $P \neq NP$ either

Dealing with NP-Completeness

- Hope that a worst case doesn't occur
  - Complexity theory deals with worst case behavior. The instance(s) you want to solve may be "easy"
    - » TSP where all points are on a line or circle
    - » 13,509 US city TSP problem solved (Cook et. al., 1998)
- Change the problem
  - Develop a heuristic, and hope it produces a good solution.
  - Design an approximation algorithm: algorithm that is guaranteed to find a high- quality solution in polynomial time
    - » active area of research, but not always possible
- Keep trying to prove P = NP

Conclusion

- It is not known whether NP problems are tractable or intractable
- But, there exist provably intractable problems
  - Even worse – there exist problems with running times unimaginably worse than exponential
- More bad news: there are **provably noncomputable (undecidable)** problems
  - There are no (and there will never be) algorithms to solve these problems

Summary

- NP - class of problems which admit non-deterministic polynomial-time algorithms
- P - class of problems which admit (deterministic) polynomial-time algorithms
- NP-Complete - the hardest of the NP problems (every NP problem can be transformed to an NP-Complete problem in polynomial time)
- So, is NP = P or not?

Summary

- We don't know!
- The NP=P? problem has been open since it was posed in 1971 and is one of the most difficult unresolved problems in computer science

Summary

UNIVERSITY OF SKÖVDE

- A polynomial function is one that is bounded from above by some function n^k for some fixed value of k (*i.e.* $k \neq f(n)$)
- An exponential function is one that is bounded from above by some function k^n for some fixed value of k (*i.e.* $k \neq f(n)$)
- Strictly speaking, n^n is not exponential but super-exponential

Summary

UNIVERSITY OF SKÖVDE

- Polynomial-time algorithm
  - Order-of-magnitude time performance bounded from above by a polynomial function of *n*
  - Reasonable algorithm
- Super-polynomial / exponential and super-exponential time algorithms
  - Order-of-magnitude time performance bounded from above by a super-polynomial, exponential, or super-exponential function of *n*
  - Unreasonable algorithm

Summary

- There are many (approx. 1000) important and diverse problems which exhibit the same properties as the monkey puzzle problem (*e.g.* TSP)
- All admit unreasonable, exponential-time, solutions
- None are known to admit reasonable ones

Summary

- But no-one has been able to prove that any of them REQUIRE super-polynomial time
- Best known lower-bounds are O(n)

Summary

- Examples of NP-Complete Problems
  - 2-D arrangments (cf. pattern matching / recognition)
  - Path-finding (e.g. travelling salesman TSP; Hamiltonian)
  - Scheduling and matching (e.g. time-tabling)
  - Determining logical truth in the propositional calculus
  - Colouring maps and graphs

Summary

- All NP-Complete problems seem to require
  - construction of partial solutions
  - and then backtracking when we find they are wrong

  in the development of the final solution
- However
  - if we could ‘guess’ at each point in the construction which partial solutions were to lead to the ‘right’ answer
  - then we could avoid the construction of these partial solutions and construct only the correct solution

Summary

UNIVERSITY OF SKÖVDE

- Important property of NP-Compete problems
  - Either all NP-Complete problems are tractable or none of them are
  - If there exists a polynomial-time algorithm for any single NP-Complete problem, then there would be necessarily a polynomial-time algorithm for all NP-Complete problems
  - If there is an exponential lower bound for any NP-Complete problem, they all are intractable

Summary

UNIVERSITY OF SKÖVDE