CS711008Z Algorithm Design and Analysis

Lecture 1. Introduction and some representative problems [1]

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

[1] The slides are made based on Chapter 1 of Algorithms, Chapter 2 of Design and Analysis of Algorithms.

The origin of the word “Algorithm”

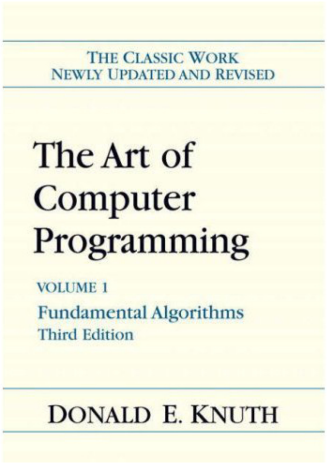
Figure: Muhammad ibn Musa al-Khwarizmi (C. 780—850), a Persian scholar, formerly Latinized as Algoritmi

- In the twelfth century, Latin translations of his work on the Indian-Arabic numerals introduced the decimal positional number system to the Western world.

Al-Khwarizmi‘s contributions

- Al-Khwarizmi‘s *The Compendious Book on Calculation by Completion and Balancing* presented the first systematic solution of **linear** and **quadratic equations** in Arabic.
- Two words:
  - **Algebra**: from Arabic "al-jabr" meaning "reunion of broken parts" — one of the two operations he used to solve equations
  - **Algorithm**: a step-by-step set of operations to get solution to a problem

Algorithm design: the art of computer programming


THE CLASSIC WORK
NEWLY UPDATED AND REVISED

The Art of Computer Programming

VOLUME 1

Fundamental Algorithms

Third Edition

DONALD E. KNUTH

V. Vazirani said:

Our philosophy on the design and exposition of algorithms is nicely illustrated by the following analogy with an aspect of Michelangelos's art:
A major part of his effort involved looking for interesting pieces of stone in the quarry and staring at them for long hours **to determine the form they naturally wanted to take***. The chisel work exposed, in a minimal manner, this form.*

V. Vazirani said: $cont'd$

By analogy, we would like to start with a clean, simply stated problem.
Most of the algorithm design effort actually goes into **understanding the algorithmically relevant combinatorial structure of the problem**.
The algorithm exploits this structure in a minimal manner..... with emphasis on stating the structure offered by the problems, and keeping the algorithms minimal.

(See extra slides.)

Basic algorithmic techniques

- **Divide-and-conquer**: Let's start from the "smallest" problem first, and investigate whether a large problem can **reduce to smaller subproblems**.
- **Improvement**: Let's start from **an initial complete solution**, and try to improve it step by step.
- **"Intelligent" enumeration**: we enumerate **all possible complete solutions**, but employ some techniques to prune the search tree.

The first example: calculating the greatest common divisor (gcd)

The first problem: calculating **gcd**

Definition (gcd)

The greatest common divisor of two integers a and b, when at least one of them is not zero, is the largest positive integer that divides the numbers without a remainder.

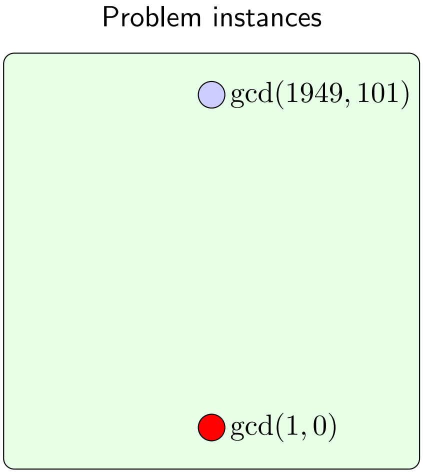
- Example:
  - The divisors of 54 are: $1, 2, 3, 6, 9, 18, 27, 54$
  - The divisors of 24 are: $1, 2, 3, 4, 6, 8, 12, 24$
  - Thus, $\gcd(54, 24) = 6$.

The first problem: calculating **gcd**

INPUT: two n bits numbers a, and b ($a \geq b$)
OUTPUT: $\gcd(a, b)$

- Observation: the problem size can be measured by using n;
- Let's start from the "smallest" instance: $\gcd(1, 0) = 1$;
- But how to efficiently solve a "larger" instance, say $\gcd(1949, 101)$?

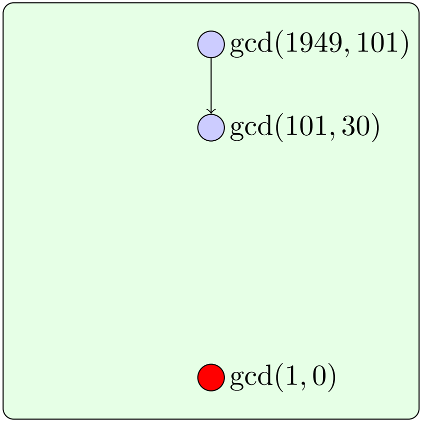
Euclid's rule



- Observation: a large problem can reduce to a smaller subproblems:
- $\gcd(1949, 101) = \gcd(101, 1949 \mod 101) = \gcd(101, 30)$

Strategy: reduce to “smaller” problems

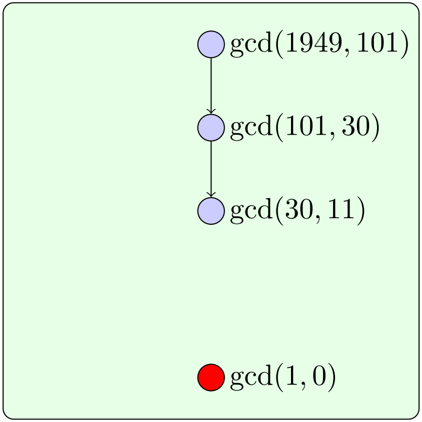
Problem instances



- $\gcd(101, 30) = \gcd(30, 101 \mod 30) = \gcd(30, 11)$

Strategy: reduce to "smaller" problems

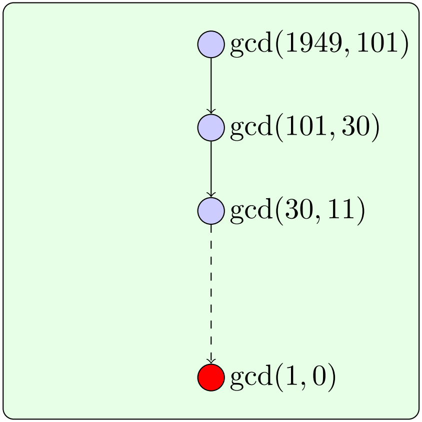
Problem instances



- $\gcd(30, 11) = \gcd(11, 30 \mod 11) = \gcd(11, 8)$

Strategy: reduce to "smaller" problems

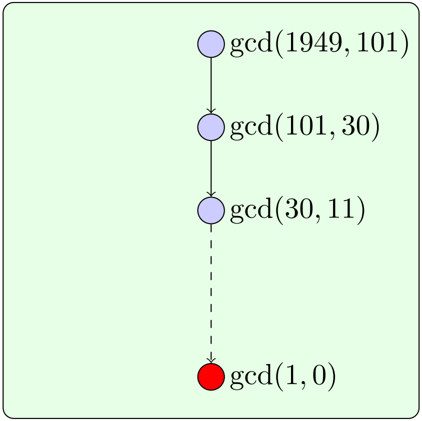
Problem instances



- $\gcd(30, 11) = \gcd(11, 8) = \gcd(8, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1$

Sub-instance relationship graph

Problem instances



- Node: subproblems
- Edge: reduction relationship

Euclid algorithm

```
function Euclid($a, b$)
if $b = 0$ then
    return $a$;
end if
return Euclid( $b$, $a \mod b$) ;
```

Time complexity analysis

Theorem

Suppose a is a n-digit integer. Euclid(a, b) ends in $O(n^3)$ time.

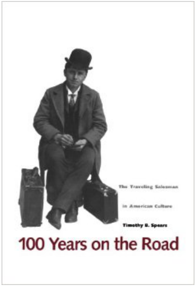
Proof.

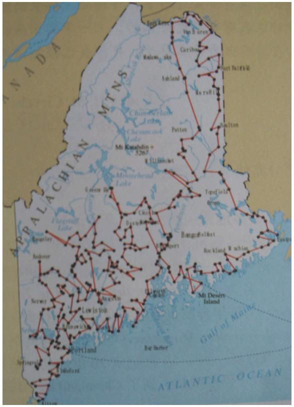
- There are at most $2n$ recursive calling.
    - Note that $a \mod b < \frac{a}{2}$.
    - After two rounds of recursive calling, both a and b shrink at least a half size.
- At each recursive calling, the $\mod$ operation costs $O(n^2)$ time.

$\square$

The second example: travelling salesman problem (TSP)

TSP: a concrete example

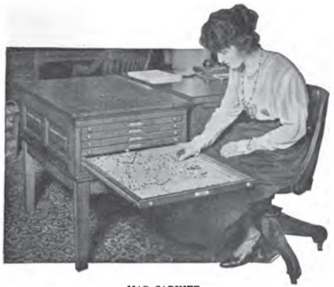




- In 1925, H. M. Cleveland, a salesman of the Page seed company, traveled 350 cities to gather order form.
- Of course, the shorter the total distance, the better.

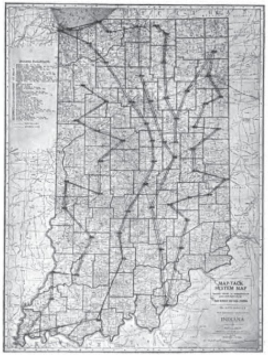
How did they do? Pin and wire!

OFFICE APPLIANCES 201



MAP CABINET *Courtesy of Rand-McNally*

202 SECRETARIAL STUDIES

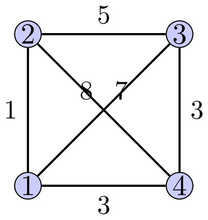


MAP SHOWING ROUTING OF SALESMEN BY PINS AND CORDS
Courtesy of Rand-McNally

- Two pictures excerpted from *Secretarial Studies*, 1922.

Travelling Salesman Problem

INPUT: a list of n cities, and the distances between each pair of cities d_{ij} $(1 \leq i, j \leq n)$;
OUTPUT: the shortest tour that visits each city exactly once and returns to the origin city

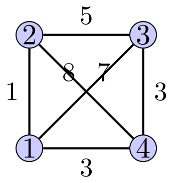


#Tours: 6

- Tour 1: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ (distance: 12)
- Tour 2: $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ (distance: 21)
- Tour 3: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 1$ (distance: 23)
- ....

Trial 1: divide and conquer

Consider a tightly related problem



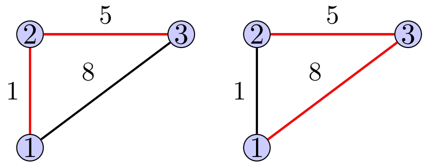
Definition

$D(S, e)$ = the minimum distance, starting from city 1, visiting all cities in S, and finishing at city e.

- There are 3 cases of the city from which we return to 1.
- Thus, the shortest tour can be calculated as:
  $\min\{d_{2,1} + D(\{2,3,4\}, 2),$
  $d_{3,1} + D(\{2,3,4\}, 3),$
  $d_{4,1} + D(\{2,3,4\}, 4)\}$

Consider the smallest problem

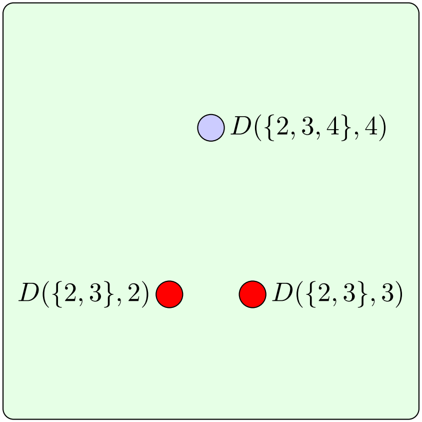
- It is trivial to calculate $D(S, e)$ when S consists of only 2 cities.



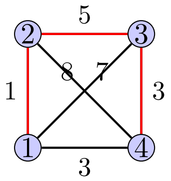
- $D(\{2,3\}, 2) = d_{13} + d_{32}$;
- $D(\{2,3\}, 3) = d_{12} + d_{23}$;
- But how to solve a large problem, say $D(\{2,3,4\}, 4)$?

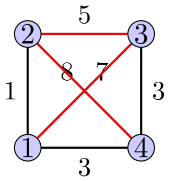
Sub-instance relationship graph

Problem instances



Divide a larger problem into smaller problems

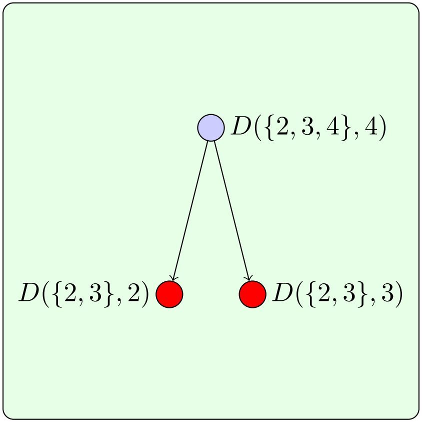




- $D(\{2,3,4\},4) = \min\{d_{34} + D(\{2,3\},3), d_{24} + D(\{2,3\},2)\};$

Reduce to smaller problems

Problem instances



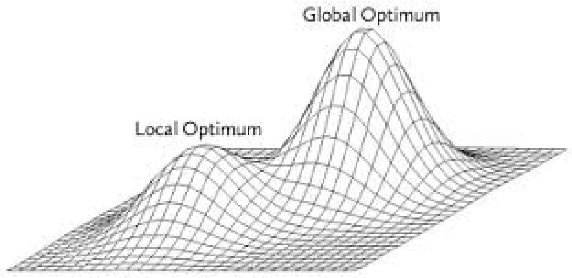
Held-Karp algorithm [1962]

```
1:  function $D(S, e)$
2:  if $S = \{v, e\}$ then
3:      $D(S, e) = d_{1v} + d_{ve}$
4:      return $D(S, e)$;
5:  end if
6:  $D(S, e) = \infty$
7:  for all city $i \in S$, and $i \neq e$ do
8:      if $D(S - \{e\}, i) + d_{ie} < D(S, e)$ then
9:          $D(S, e) = D(S - \{e\}, i) + d_{ie}$;
10:     end if
11: end for
12: return $D(S, e)$;
```

- Time complexity: $O(2^n n^2)$.

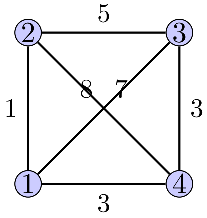
Trial 2: Improvement strategy

Solution space



- Node: a complete solution. Each node is associated with an objective function value.
- Edge: if two nodes are neighboors, an edge is added to connect them.
- Improvement strategy: start from an initial solution, and try to improve it step by step.

Improvement strategy



- Note that a **complete solution** can be expressed as a permutations of the n cities;
- Let's start from **an initial complete solution**, and try to improve it;

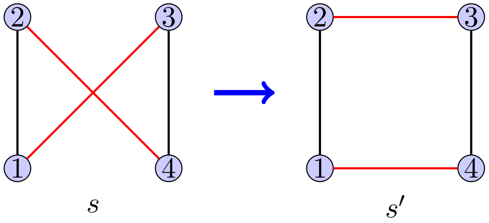
IMPROVEMENT strategy

```
1: Let $s$ be an initial tour;
2: **while** TRUE **do**
3:     select a new tour $s'$ from the **neighbourhood** of $s$;
4:     **if** $s'$ is shorter than $s$ **then**
5:         $s = s'$;
6:     **end if**
7:     **if** stopping($s$) **then**
8:         **return** $s$;
9:     **end if**
10: **end while**
```

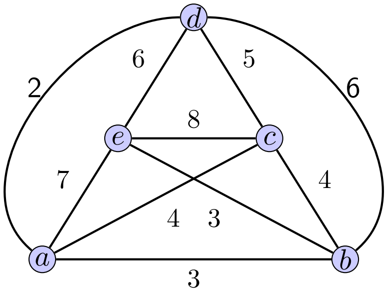
Here, **neighbourhood** is introduced to describe how to change an existing tour into a new one;

But how to define neighbourhood of a tour?

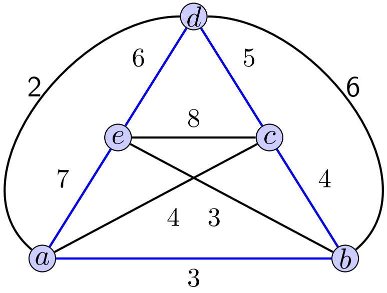
- 2-opt strategy: if s' and s differ at only two edges (Note: 1-opt is impossible)



An example

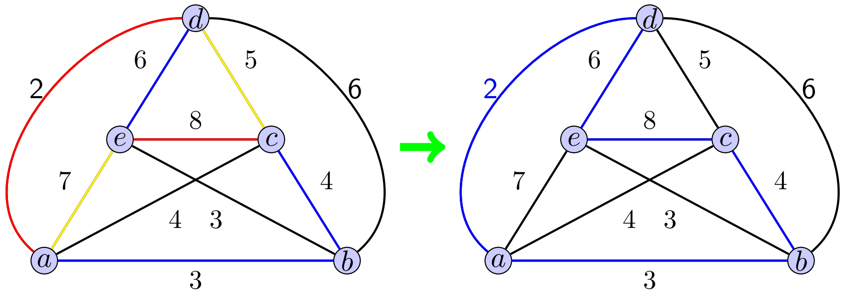


Initial tour s



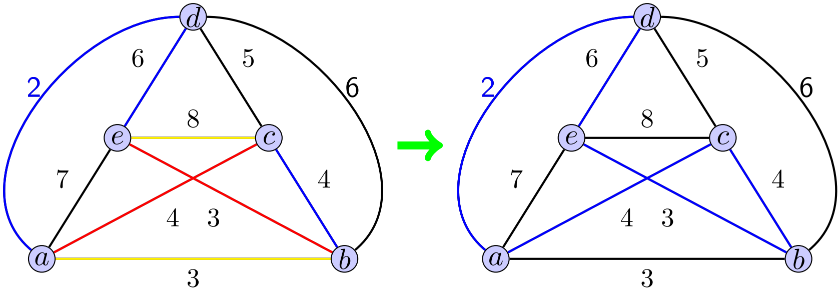
Initial complete solution s: $a \to b \to c \to d \to e \to a$ (distance: 25)

A 2-opt operation: $s \Rightarrow s'$



Initial solution s: $a \to b \to c \to d \to e \to a$ (distance: 25)
Improve from s to s': $a \to b \to c \to e \to d \to a$ (distance: 23)

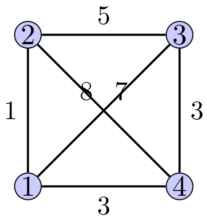
One more 2-opt operation: $s' \Rightarrow s''$



A complete solution s': $a \rightarrow b \rightarrow c \rightarrow e \rightarrow d \rightarrow a$ (distance: 23)
Improve from s' to s'': $a \rightarrow c \rightarrow b \rightarrow e \rightarrow d \rightarrow a$ (distance: 19)
Done! No 2-OPT can be found to improve further.

Trial 3: backtracking: an intelligent enumeration strategy

BACKTRACKING strategy

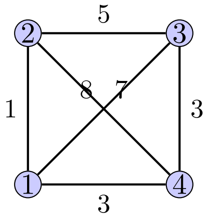


- Note that a complete solution can be expressed as a sequence of $n-1$ edges;
- Formally, we can represent a complete solution as: $X = [x_1, x_2, ..., x_m]$, $x_i = 0/1$.

Backtrack algorithm

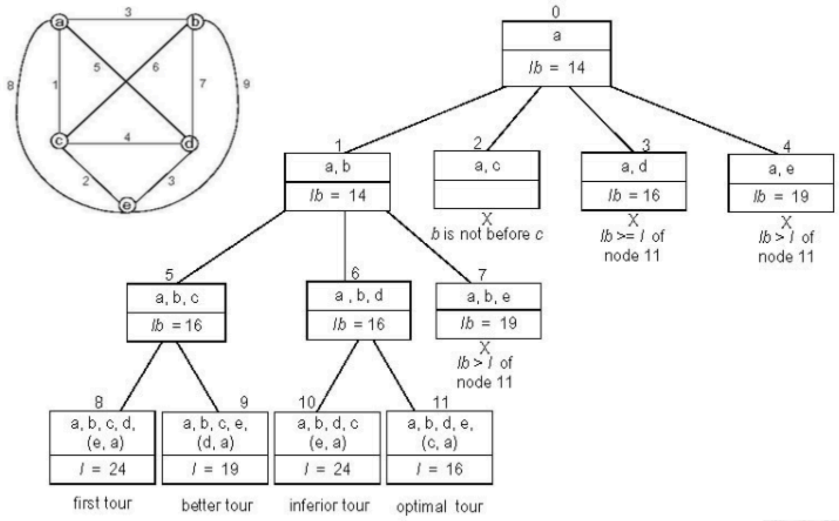
```
1: function TSP( $G, d$)
2: Assign edges with an arbitrary order;
3: $edge1[1..m] = 0$;
4: $Backtrack(G, edge1, 0, edge2, Length)$;
```

Backtracking strategy: another trial



- Note that a complete solution can be expressed as a sequence of n nodes;
- Formally, we can represent a complete solution as: $X = [v_1, v_2, ..., v_n]$.

Enumeration tree and backtracking



Backtracking

```
1:  Start with the original problem $P_0$;
2:  Let $A = \{P_0\}$. Here $A$ denotes the active subproblems.
3:  $bestsofar = \infty$;
4:  while $A \neq NULL$ do
5:      choose a subproblem $P \in A$, and remove it from $A$;
6:      expand $P$ into smaller subproblems $P_1, P_2, ..., P_k$;
7:      for $i = 1$ to $k$ do
8:          if $(P_i)$ corresponds to a complete solution then
9:              update $bestsofar$;
10:         end if
11:         if lowerbound$(P_i) \leq bestsofar$ then
12:             insert $P_i$ into $A$;
13:         end if
14:     end for
15: end while
```

Time complexity and space complexity

Time complexity and space complexity

- Time (space) complexity of an algorithm quantifies the time (space) taken by the algorithm.
- Since the time costed by an algorithm grows with the size of the input, it is traditional to describe running time as a function of the input size.
    - **input size**: The best notation of input size depends on the problem being studied.
        - For the TSP problem, the `number of cities in the input`.
        - For the MULTIPLICATION problem, the `total number of bits` needed to represent the input number is the best measure.

Running time: we are interested in its growth rate

- A straightforward way is to use the exact seconds that a program used. However, this measure highly depends on CPU, OS, compiler, etc.
- Several simplifications to ease analysis of Held-Karp algorithm:
    1. We simply use the number of primitive operations (rather than the exact seconds used) under the assumption that a primitive operation costs constant time. Thus the running time is $T(n) = an^2 + bn + c$ for some constants a, b, c.
    2. We consider only the leading term, i.e. an^2, since the lower order terms are relatively insignificant for large n.
    3. We also ignore the leading term's coefficient a since the it is less significant than the growth rate.
- Thus, we have $T(n) = an^2 + bn + c = O(n^2)$. Here, the letter O denotes order.

Big O notation

- Recall that big O notation is used to describe the error term in Taylor series, say:

$$e^x = 1 + x + \frac{x^2}{2} + O(x^3) \text{ as } x \to 0$$

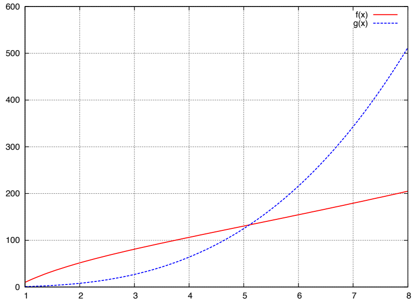


Figure: Example: $f(x) = O(g(x))$ as there exists $c > 0$ (e.g. $c = 1$) and $x_0 = 5$ such that $f(x) < cg(x)$ whenever $x > x_0$

Big Ω and Big Θ notations

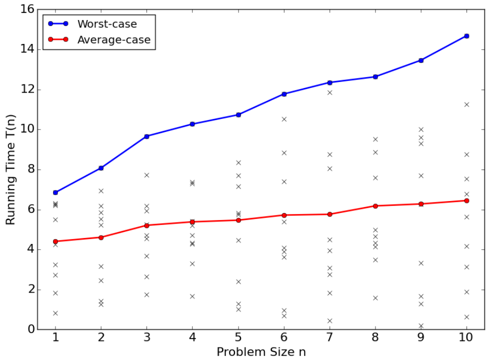
- In 1976 D.E. Knuth published a paper to justify his use of the Ω-symbol to describe a stronger property. Knuth wrote: "For all the applications I have seen so far in computer science, a stronger requirement [...] is much more appropriate".
- He defined

$$f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))$$

  with the comment: "Although I have changed Hardy and Littlewood's definition of Ω, I feel justified in doing so because their definition is by no means in wide use, and because there are other ways to say what they want to say in the comparatively rare cases when their definition applies".
- Big Θ notation is used to describe "$f(n)$ grows asymptotically as fast as $g(n)$".

$$f(x) = \Theta(g(x)) \Leftrightarrow g(x) = O(f(x)) \text{ and } f(x) = O(g(x)).$$

Worst case and average case



- Worst-case: the case that takes the longest time;
- Average-case: we need know the distribution of the instances;