CGI-ACT

Algorithms and Algorithmic Problems

Workshop Sessions

Session 1 - Introduction to Algorithms
by
Paul Minasian & Ahmet Taspinar

Agenda

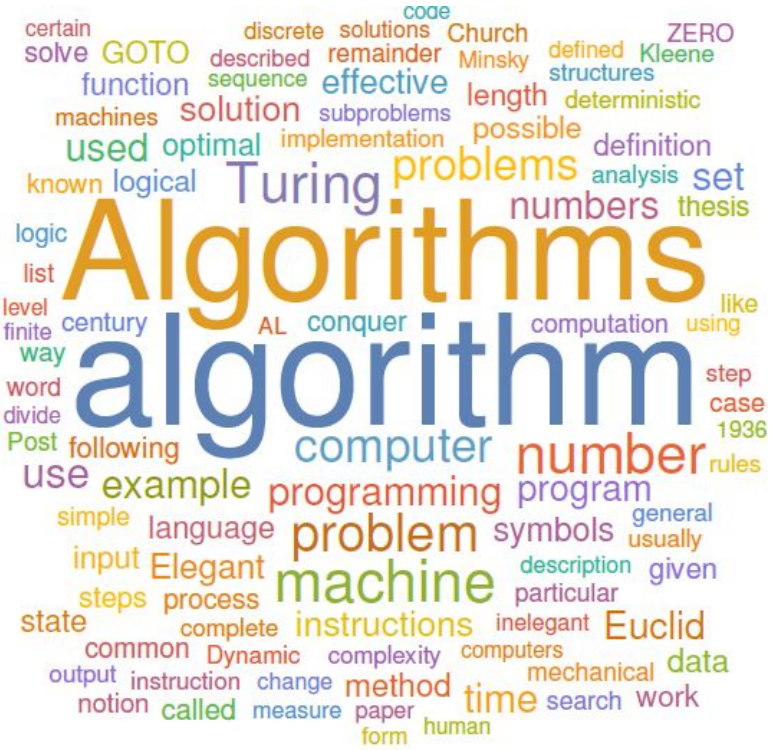
1. Overview of the Workshop Sessions
2. The Role of Algorithms in Computing
3. Gentle Introduction to Algorithm Analysis
4. Pseudocode
5. General Strategies for Algorithm Design
6. Exercises

Overview of the Workshop Sessions

- Session 1 - Introduction to Algorithms
- Sessions 2...n could be about:
  - machine learning algorithms
  - sorting-, searching-, graph-, numerical- algorithms
  - miscellaneous algorithmic problems / puzzles
  - computer vision algorithms
  - parallel, and distributed- algorithms
  - mathematics for algorithm analysis
  - genetic algorithms
  - randomized algorithms
  - approximation algorithms

The Role of Algorithms in Computing

WordCloud[DeleteStopwords[WikipediaData["algorithms"]]]



Computational Problem

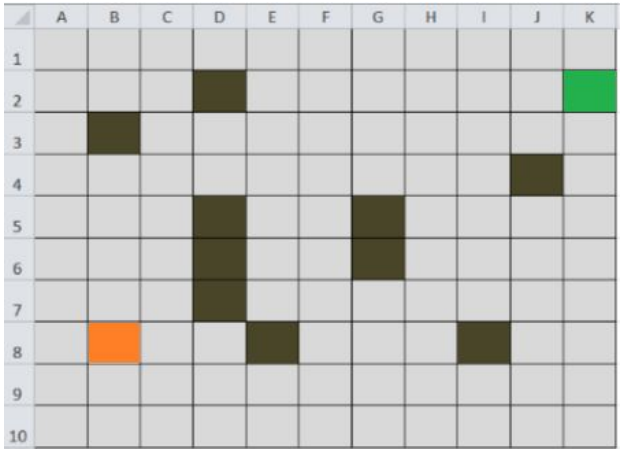
The field of algorithms studies methods to solve computational problems efficiently. A computational problem specifies an input-output relationship.

Example of a computational problem:

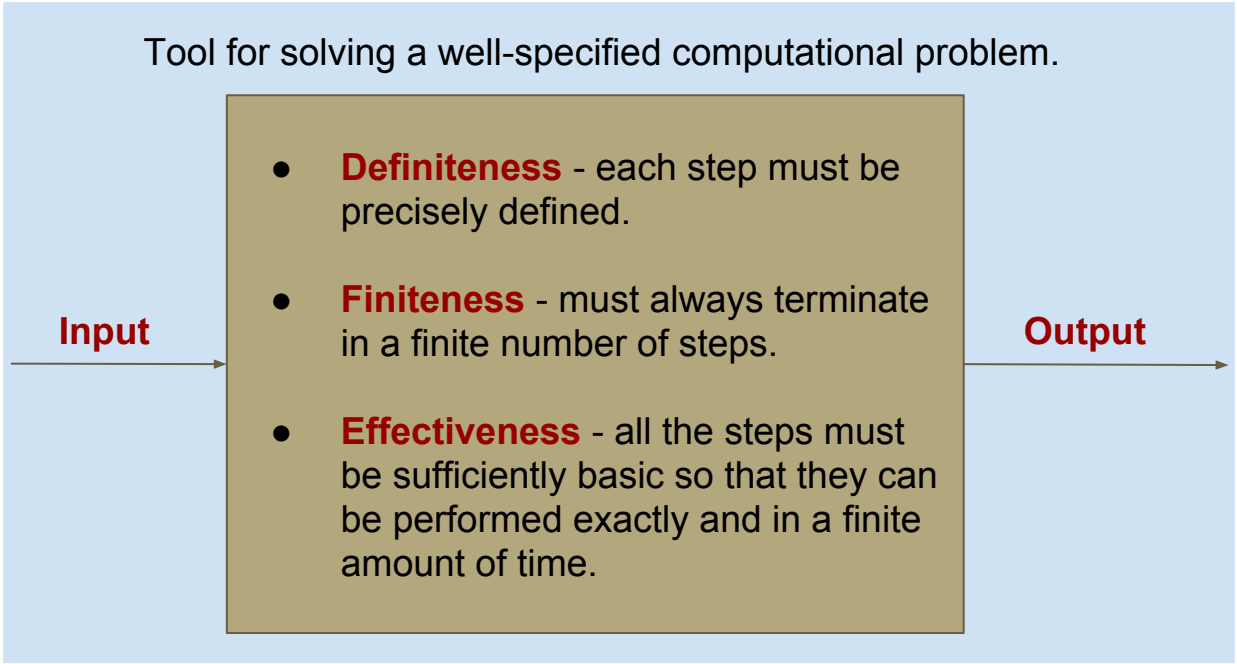
> **Compute all shortest paths from the start point to the end point by going from each cell either left, right, up or down and avoiding obstacles.**

Input: an array **m x n: A[1..m][1..n]**. Cell with 1 indicates the start point. Cell with 2 indicates the end point. Cell with 3 indicates an obstacle and a cell with 0 indicates empty space.

Output: list of lists which contain the points on the path from the start point to the end point.



What is an Algorithm?

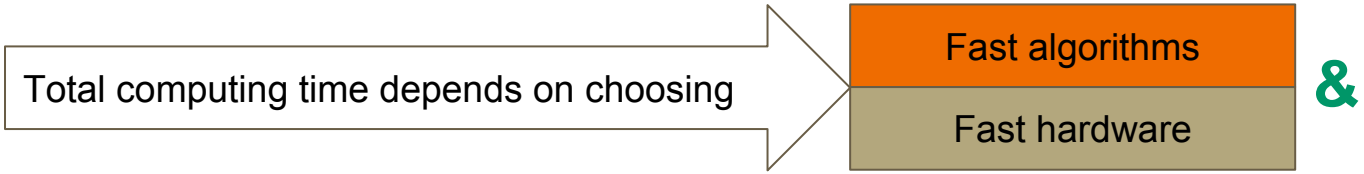


Suitable for implementation as a computer program

Algorithms as a technology

- Computers are not infinitely fast
- Computer memory is not free

Computing time is therefore a bounded resource, and so is space in memory.



A naive algorithm on a supercomputer may take impractically long to finish, whereas a sophisticated algorithm on a cheaper hardware may suddenly become practical.

Impact of a fast algorithm vs fast hardware

Algorithm	* Predicted time for n = 10K	Predicted time for n = 10K for a 1mln times faster computer
Naive (n^5)	≈ 31710 centuries	≈ 3.17 years
Fast (n^2)	100 seconds	0.1 milliseconds

* Indication of size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f(n) microseconds.

Applications rely on algorithms

Even when the code that you've written at the application level is not based on heavy algorithmic content (solving complex computational problems), your application still relies heavily upon algorithms in other areas.

Does your application require:

- Fast hardware: hardware design used algorithms
- Database: sorting and searching of data rely on algorithms
- GUI: design of a GUI library relies on algorithms
- Networks: network routing makes use of algorithms
- Compiler, interpreter, assembler: make extensive use of algorithms

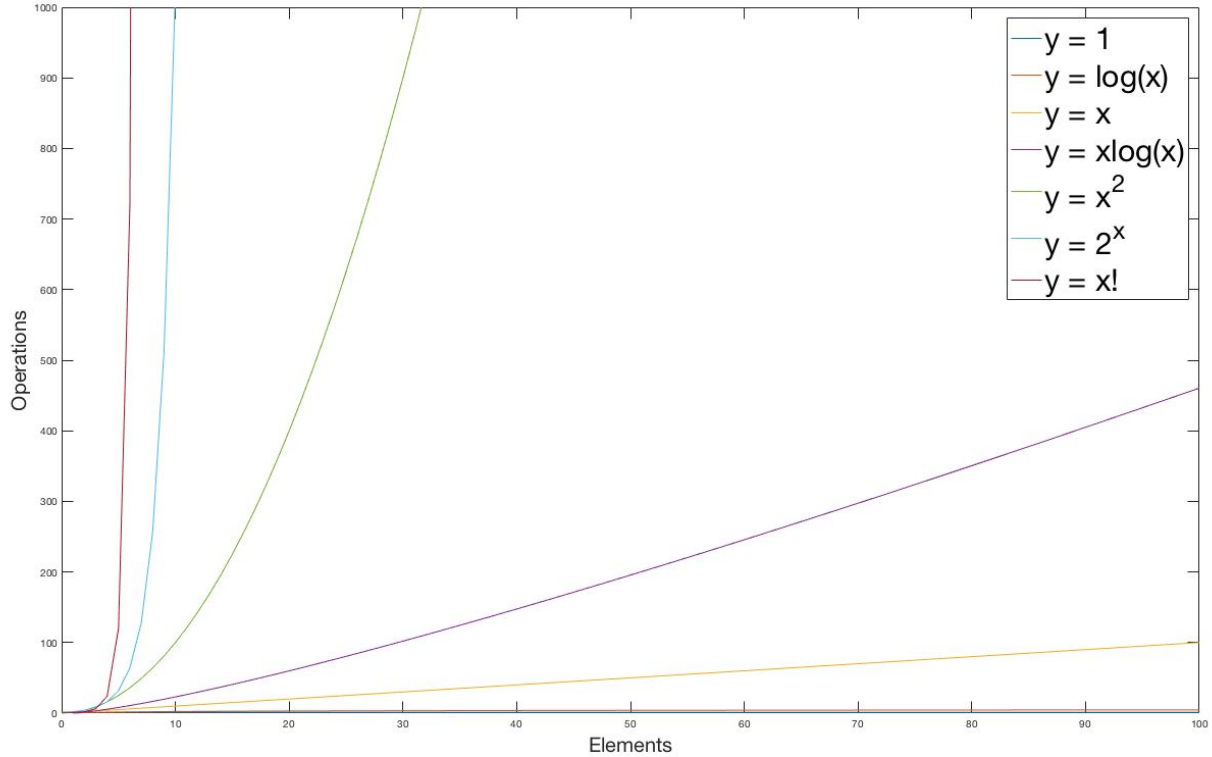
Some Application Areas

- Machine learning
- Robotics, computer vision
- Compilers, cryptography
- Computational geometry
- Operating systems, databases, networks, computer graphics
- Computational biology
- Logistics (planning and scheduling)
- CAD / CAM
- Geographical Information Systems
- ...

Why Study Algorithms?

- For solving **complex** computational problems.
- Understand the impact of algorithms on application **performance**.
- **Reuse** - be able to choose from a collection of existing algorithms to tackle a problem.
- Being able to identify the problem as being **hard** (e.g., **NP-complete**) (to prevent a fruitless search in finding an efficient algorithm - try finding a **good solution, but not the best possible / optimal solution**).
- For **challenge and fun**! (e.g., solving algorithmic puzzles as recreation).

Comparison of running times



Gentle Introduction to Algorithm Analysis

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i = j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Source: Introduction to Algorithms (3rd Edition): Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: 9780262033848.

Analysing Computational Complexity

Analysing the computational complexity of an algorithm consists of:

- **Time complexity analysis** - analysis of the time required to solve a problem of a particular size.

- **Space complexity analysis** - analysis of the computer memory required to solve a problem of a particular size.

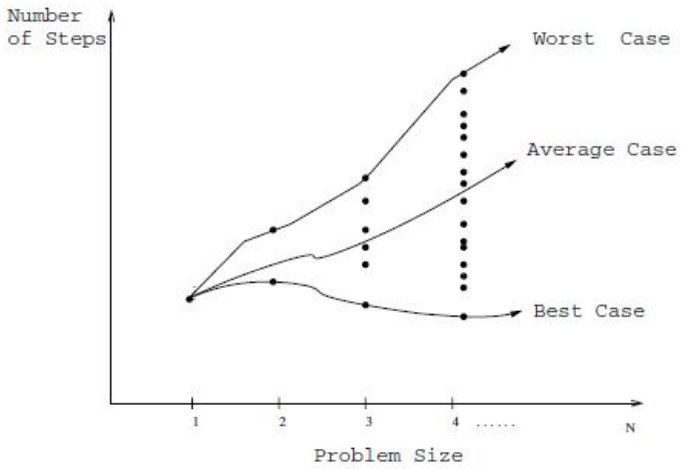
We will focus on time complexity analysis.

Input Size, Running Time

- The running time grows with the size of the input.
- **Input size** - depends on the problem being studied. (e.g., when sorting an array of n integers, the input size is the length of the array.)
- **Running time** - is the number of primitive operations or “steps” executed on a particular input.

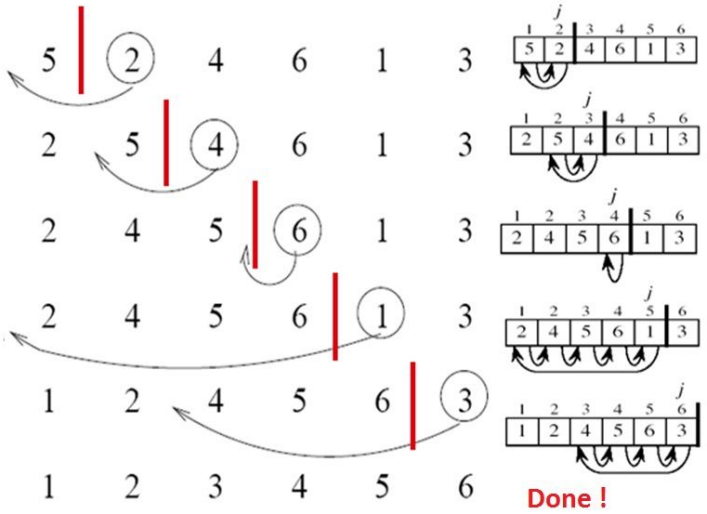
Running Time

- **Worst-case** - the maximum number of steps taken in any input instance of size n. Gives us an **upper bound** on the running time.
- **Best-case** - the minimum number of steps taken in any input instance of size n. Gives us **a lower bound** on the running time.
- **Average-case** - the average number of steps over all input instances of size n**.**



For the problem of sorting, the set of possible input instances consists of all possible arrangements of n items, over all possible values of n.

Analysis of Insertion Sort



INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = \left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2 + \left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n - \left(c_2+c_4+c_5+c_8\right)$$

Worst-case: $an^2 + bn + c$

Source: Introduction to Algorithms (3rd Edition): Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: 9780262033848.

Simplifying Abstractions during analysis of Insertion Sort

- c_i is used to represent the costs and thus ignoring the actual cost of each statement.
- Worst-case running time was then expressed as **$an^2 + bn + c$** for some constants a, b, and c that depend on the statement costs . c_i
- Even the expression **$an^2 + bn + c$** can be simplified. For that we need to define the **rate of growth** or **order of growth**.

Order of Growth

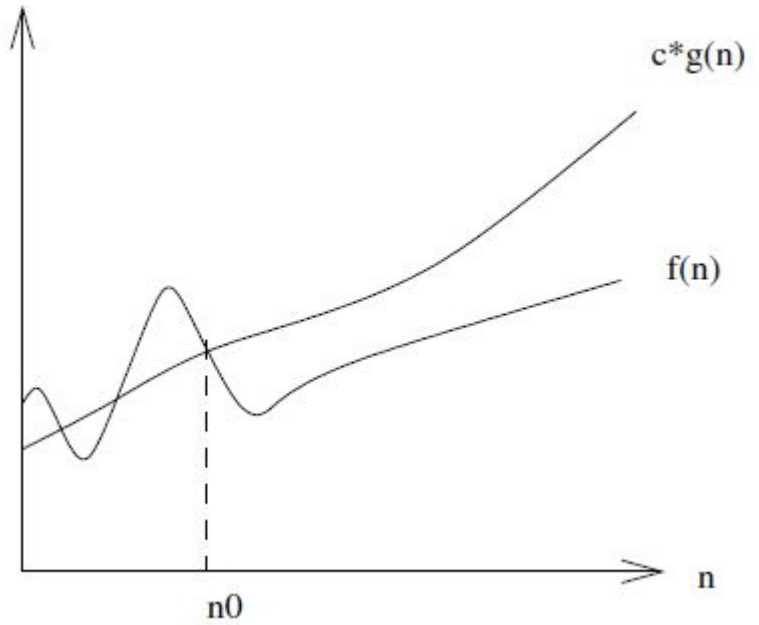
- Gives a **simple** characterization of the algorithm's efficiency. Thus, it does not show the exact running time of an algorithm.
- Allows to **compare relative performance** of alternative algorithms.
- Take for example the expression: $\mathbf{an^2 + bn + c}$. For large enough inputs, the multiplicative constants (a, b, c), and the lower-order terms (n, n^0) are dominated by the effects of the input size itself.
- **Asymptotic efficiency of algorithms** - applies when considering the performance of algorithms for large input sizes which make only the order of growth of the running time relevant.

Asymptotic Notation

- Applies to functions (e.g., $an^2 + bn + c$).
- Is used to characterize the running time of algorithms.
- Can also be used to characterize some other aspect of algorithms, the amount of space for instance.
- When asymptotic notation is applied to a running time, we need to understand to which running time (worst-case, best-case, average-case) it applies.

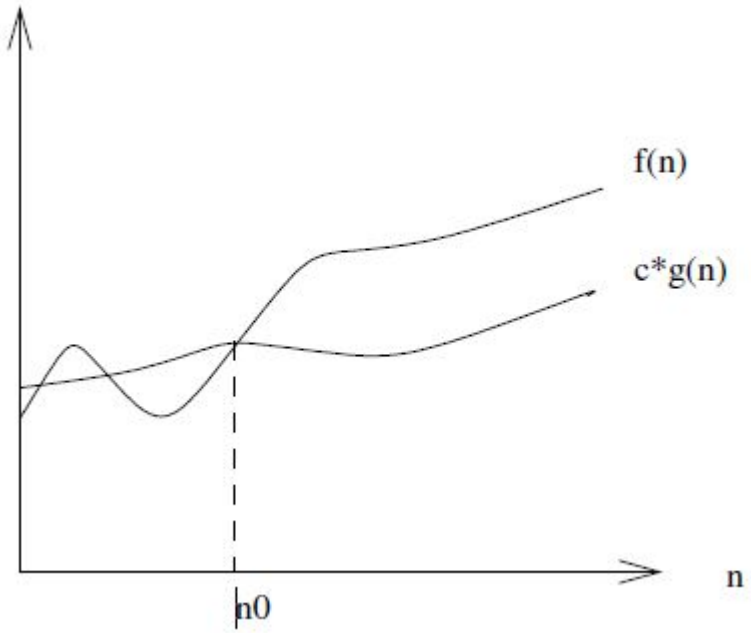
Asymptotic Notation - O (Big-O)

- Gives an asymptotic **upper** bound.
- **$O(g(n)) = \{f(n) \mid \exists_c \exists_{n0} (c > 0 \wedge n_0 > 0 \wedge (0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0))\}$.**
- Examples:
  - **$3n^2 - 100n + 6 = O(n^2)$**, because for c = 3 we have $3n^2 > 3n^2 - 100n + 6$ when $n > 0$
  - **$3n^2 - 100n + 6 = O(n^3)$**, because for c = 1 we have $n^3 > 3n^2 - 100n + 6$ when $n > 0$
  - **$3n^2 - 100n + 6 \neq O(n)$**, because for any c we have $cn < 3n^2$ when $n > c$



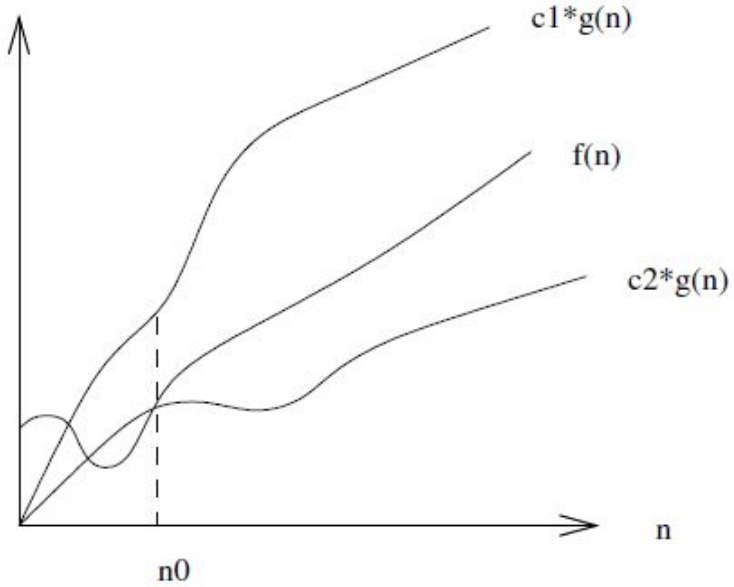
Asymptotic Notation - Ω (Big-Omega)

- Gives an asymptotic **lower** bound.
- **$\Omega(g(n)) = \{f(n) \mid \exists_c \exists_{n0}(c > 0 \wedge n_0 > 0 \wedge (0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0))\}$.**
- Examples:
  - **$3n^2 - 100n + 6 = \Omega(n^2)$**, because for c = 2 we have $2n^2 < 3n^2 - 100n + 6$ when $n > 100$
  - **$3n^2 - 100n + 6 = \Omega(n)$**, because for any c we have $cn < 3n^2 - 100n + 6$ when $n > 100c$
  - **$3n^2 - 100n + 6 \neq \Omega(n^3)$**, because for c = 1 we have $n^3 > 3n^2 - 100n + 6$ when $n > 0$



Asymptotic Notation - Θ (Big-Theta)

- Gives an asymptotic **tight** bound.
- **$\Theta(g(n)) = \{f(n) \mid \exists_{c1} \exists_{c2} \exists_{n0} (c_1>0 \wedge c_2>0 \wedge n_0>0 \wedge (0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0))\}$.**
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$
- Examples:
  - **$3n^2 - 100n + 6 = \Theta(n^2)$**, because both O and Ω apply;
  - **$3n^2 - 100n + 6 \neq \Theta(n^3)$**, because only O applies;
  - **$3n^2 - 100n + 6 \neq \Theta(n)$**, because only Ω applies.

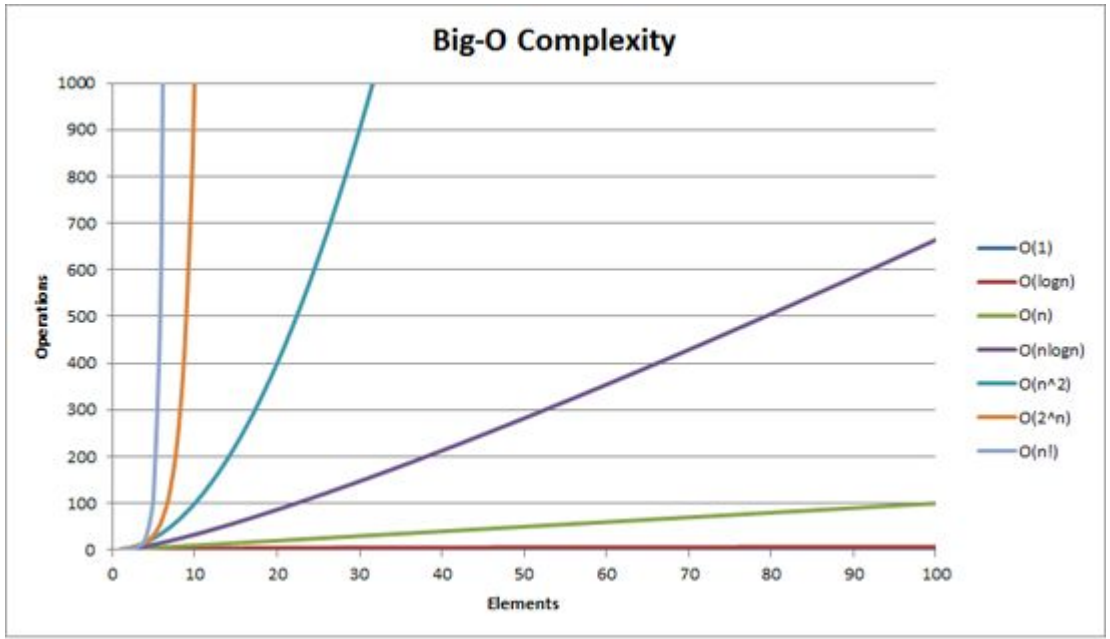


Convenient Notations

- $f(n) \in O(g(n))$ is usually written as: $f(n) = O(g(n)$.
- $f(n) \in \Omega(g(n))$ is usually written as: $f(n) = \Omega(g(n)$.
- $f(n) \in \Theta(g(n))$ is usually written as $f(n) = \Theta(g(n))$.
- $O(n^0)$ is usually written as $O(1)$.

Growth of Functions

Growth of functions commonly used in Big-O estimates.



We say g dominates f when f(n) = O(g(n)) and f(n) ≠ Θ(g(n))), also written as g ≫ f.

$n! \gg 2^n \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$

Order-of-growth Examples

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$		*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`  if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$		*divide and conquer*	*mergesort*

Source: Algorithms (4th Edition): Robert Sedgewick, Kevin Wayne: 9780321573513.

Order-of-growth Examples (continued)

description	order of growth	typical code framework	description	example

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  if (a[i] + a[j] == 0)`
		`  cnt++;`
			double loop	*check all pairs*

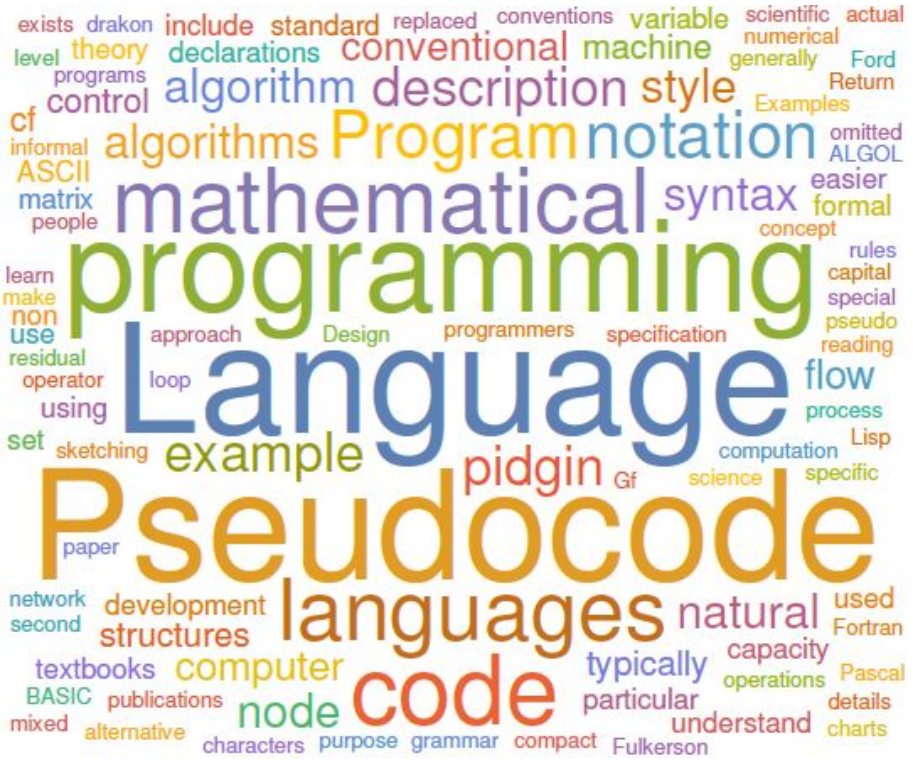
cubic	N^3	`for (int i = 0; i < N; i++)`
		`  for (int j = i+1; j < N; j++)`
		`  for (int k = j+1; k < N; k++)`
		`  if (a[i] + a[j] + a[k] == 0)`
		`  cnt++;`
			triple loop	*check all triples*

exponential	2^N		*exhasutive search*	*check all subsets*

Source: Algorithms (4th Edition): Robert Sedgewick, Kevin Wayne: 9780321573513.

Pseudocode

```
WordCloud[DeleteStopwords[WikipediaData["pseudocode"]]]
```



Pseudocode Description

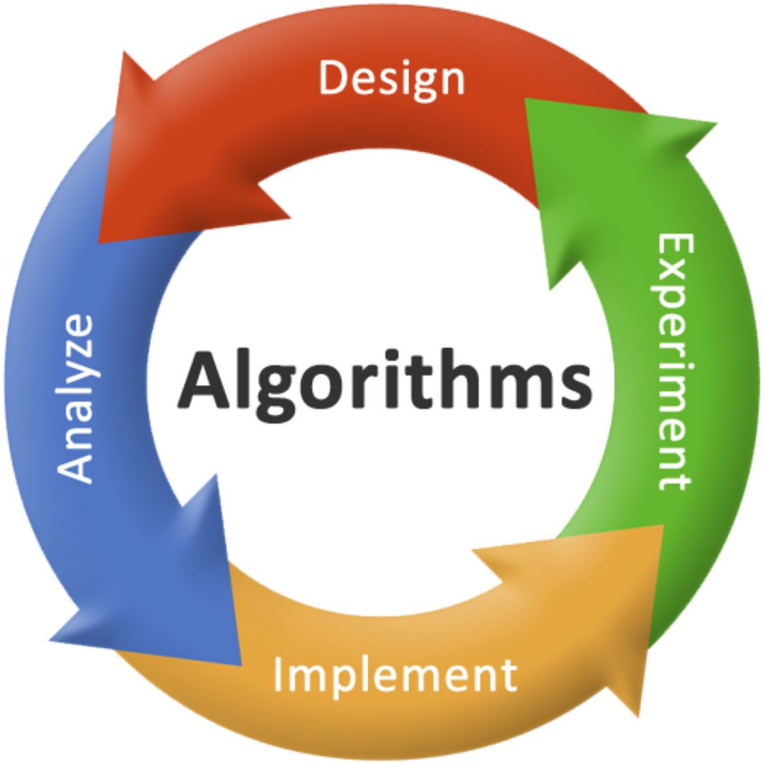
- An informal high-level description of the operating principle of a computer program or an algorithm.
- Intended to be for human reading, contains natural language descriptions, is independent from machine and environment.
- Generally does not obey the syntax rules of any particular language; there is no systematic standard form.
- Commonly used in textbooks and scientific publications that are documenting various algorithms.
- The algorithms used in this workshop exercises are written in LaTeX using the algorithmicx package with pseudocode layout. Algorithmicx Manual

Pseudocode Example

Algorithm 1 Euclid's algorithm

```
1: procedure EUCLID(a, b)                    ▷ The g.c.d. of a and b
2:     r ← a mod b
3:     while r ≠ 0 do                        ▷ We have the answer if r is 0
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     return b                              ▷ The gcd is b
```

General Strategies for Algorithm Design



Algorithm Design Strategies

- Exhaustive Search
- Backtracking
- Decrease-and-Conquer
- Divide-and-Conquer
- Transform-and-Conquer
- Greedy Approach
- Iterative Improvement
- Dynamic Programming

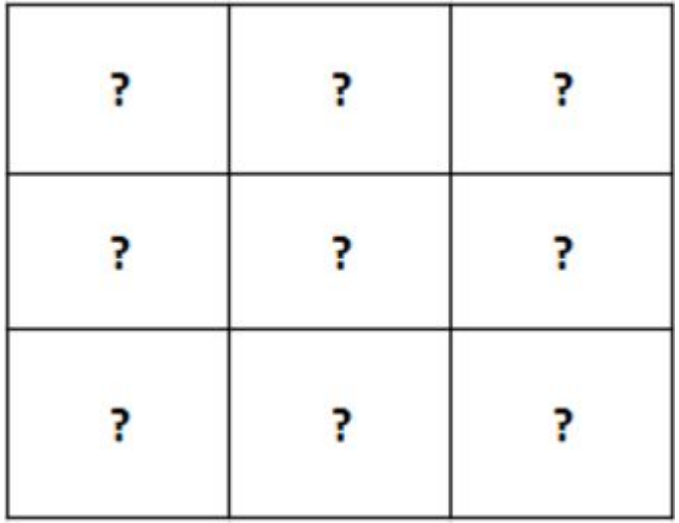
Exhaustive Search

Generates and checks all possible candidate solutions until a solution to the problem is found. Is practical only when problem size is small.

Magic Square Puzzle: fill the 3x3 table with nine distinct integers from 1 to 9 so that the sum of the numbers in each row, column, and corner-to-corner diagonal is the same.

$(3^2)! = 9 \cdot 8 \cdot 7 \cdot \ldots \cdot 1 = 362{,}880$

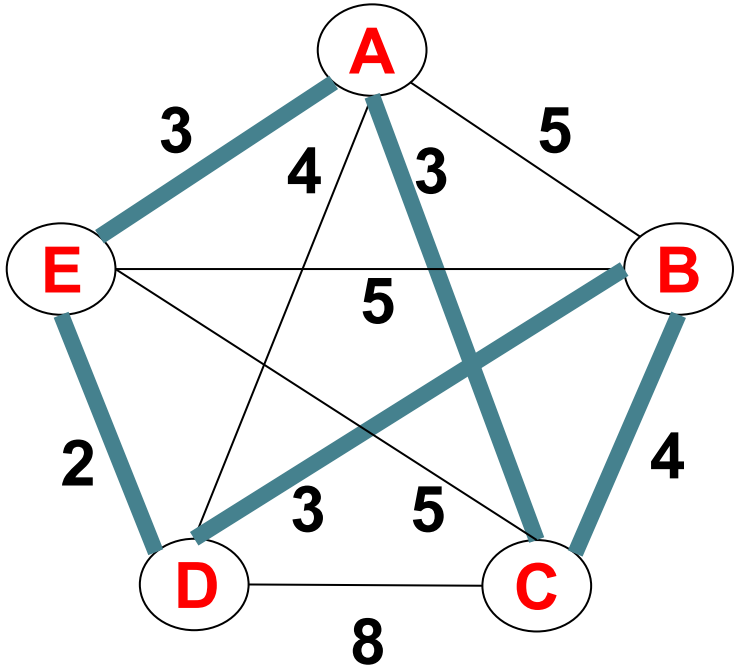
$(5^2)! \approx 1.5 \cdot 10^{25}$ - would take a computer with 10 trillion operations per second about 49,000 years to finish the job.



Exhaustive Search - Traveling Salesman Problem (TSP)

Find the circuit with the **minimum total weight** in a weighted, complete, undirected graph that visits each vertex exactly once and returns to its starting point.

ACBDEA : 15
ACBEDA : 18
ADBCEA : 19
ACEBDA : 20
ACDBEA : 22
ABEDCA : 23
ADCBEA : 24
ABECDA : 27



TSP Analysis

- Number of circuits to examine if there are n vertices in the graph: **(n - 1)! / 2**
- No need to examine a circuit in reverse order: **ACBDEA = AEDBCA = 15**
- Graph with 5 vertices: **12 circuits to examine**
- Graph with 25 vertices: **≈ 3.1 x 10^{23} circuits to examine**
  - if examing 1 circuit takes 1 nanosecond (10^{-9} second), then it would take approx. 10 mln years to find the minimum total weight

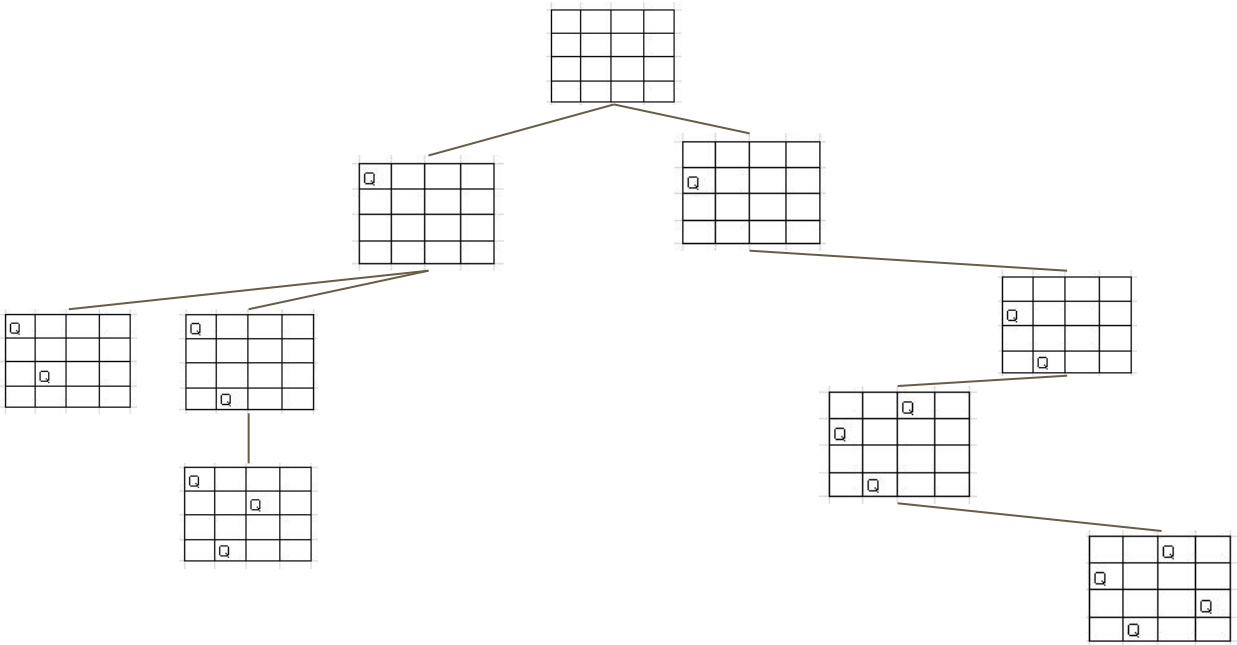
Backtracking

Is an improvement over the brute-force approach of exhaustive search. Generates and checks candidate solutions while making it possible to avoid generating unnecessary candidates.

The n-Queens Problem: place n queens on an n x n chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

It is convenient to interpret a backtracking algorithm as a process of constructing a tree that mirrors decisions being made. For a backtracking algorithm, such a tree is called a **state-space tree**. The root corresponds to the start of a solution construction process.

Backtracking - State-Space Tree Example



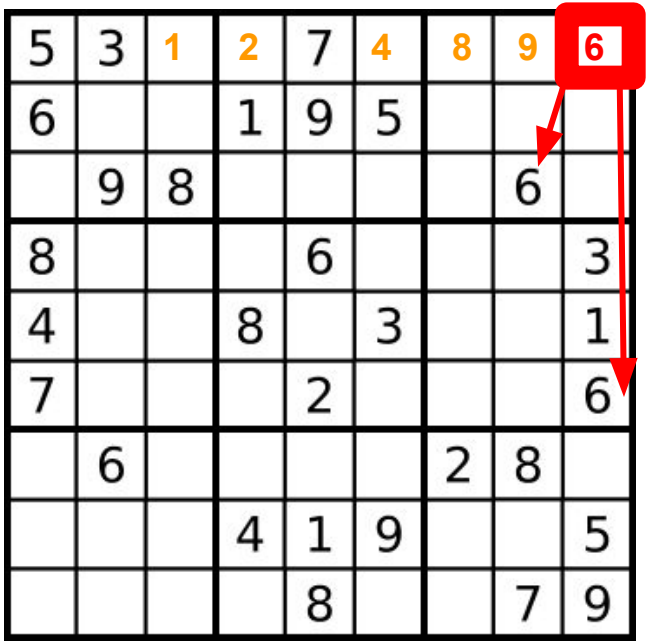
Backtracking - Sudoku

Place the numbers 1 to 9 in each open box such that each row, column and 3x3 box contains the numbers 1 to 9.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku

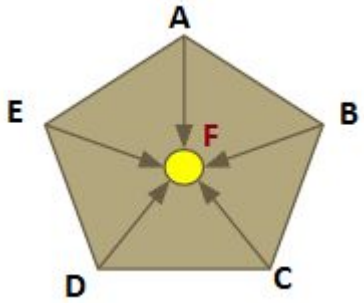
- Move from top row to the bottom row, from left to right.
- In each blank box place one of the possible numbers without violating the constraints.
- If it is not possible to place one of the remaining numbers of a row in the blank box then backtrack to the previous blank box and choose another possible number.
- Continue until all blank boxes are filled.



Decrease-and-Conquer

It is based on finding a relationship between a solution to a given problem and a solution to its smaller instance. Such a relationship leads to a ***recursive algorithm***, which reduces the problem to a sequence of its diminishing instances until it becomes small enough to be solved directly.

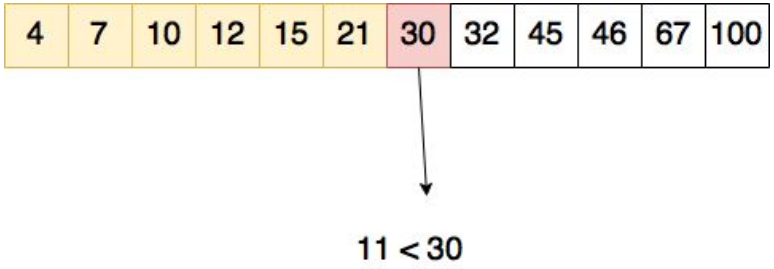
Celebrity Problem: A celebrity among a group of n people is a person who knows nobody but is known by everybody else. The task is to identify a celebrity by only asking questions to people of the following form: "Do you know this person?"



N = {A, B, C, **F**, D, E} - select two people from the group of n people, say, A and B, and ask A whether he or she knows B. If A knows B, remove A from the remaining people who can be a celebrity; if A does not know B, remove B from this group. Then solve the problem recursively for the remaining group of n-1 people. It is using **decrease-by-one** variation of decrease-and-conquer strategy.

Decrease-and-Conquer - Binary Search

- Search for a number n in a sorted array A.
- Pick the middle number.
- If n is less than the middle number, search in the left section of the array.
- If n is greater than the middle number, search in the right section of the array.
- If n equals the middle number, then return index of the middle number.



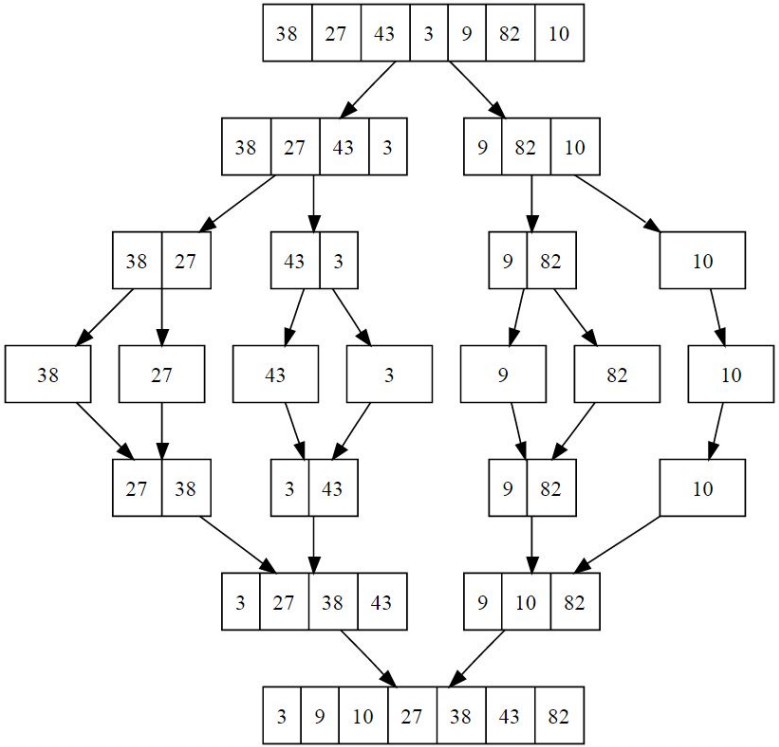
Divide-and-Conquer

Recursively breakdown / partition a problem into subproblems, until you can solve each subproblem. Then, if necessary, combining their solutions to get a final solution to the original problem.

Advantages:

- Conceptually difficult problems are broken down into more trivial problems.
- Distinct subproblems can be run parallel in a multiprocessor system.
- Large problems are solved more efficiently.

Divide-and-Conquer - Merge Sort



root

level 1 (1st recursive call)

level 2

level 3

How many levels are there in general, as a function of the length n of the input array?

Source: https://en.wikipedia.org/wiki/Merge_sort

Transform-and-Conquer

A problem is solved in two stages:

1. In the transformation stage, it is modified or transformed into another problem that, for one reason or another, is more amenable to solution.
2. In the conquering stage the problem is solved.

Within algorithmic problem solving one can identify three varieties of this strategy:

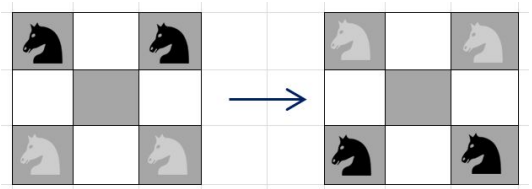
1. **Instance simplification** - transformation of a given instance into another instance of the same problem with some special property that makes the problem easier to solve.
    a. e.g., presorting before computing the median

2. **Representation change** - transformation of the problem's input to a different representation that is more conducive to an efficient algorithmic solution.

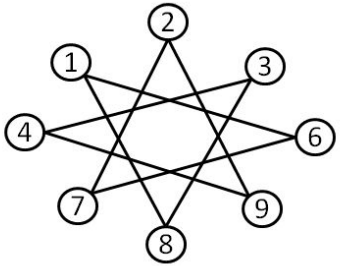
3. **Problem reduction** - transformation of a given instance into an instance of a different problem.
    a. e.g., least common multiple of two integers is the smallest integer divisible by m and n.
    lcm(m, n) = m•n / gcd(m, n)

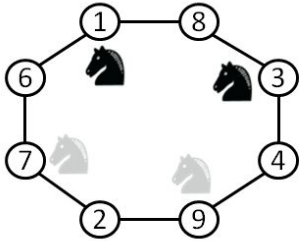
Transform-and-Conquer - Representation change

Guarini's Puzzle: there are four knights on the 3x3 chessboard: the two white knights are at the two bottom corners, and the two black knights are at the two upper corners of the board. The goal is to switch the knights in the minimum number of moves so that the white knights are at the upper corners and the black knights are at the bottom corners.



1	2	3
4	5	6
7	8	9

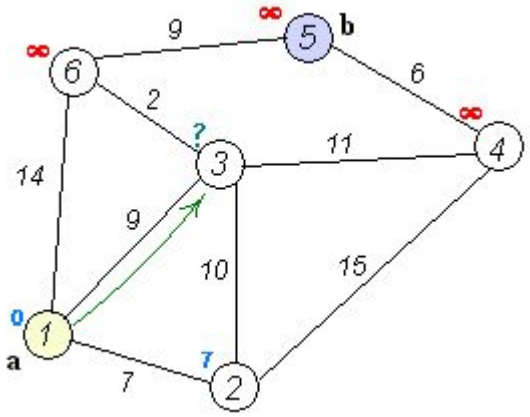




Greedy Approach

- A greedy algorithm makes a locally optimal choice at each stage, in the hope that this will lead to a globally optimal solution.
- Does not always lead to optimal solutions.
- Generally faster than other optimization strategies like Dynamic Programming.

Greedy Approach - Dijkstra's shortest path algorithm



- Algorithm for finding the shortest path in a graph between two nodes.
- Traverse the graph by looking at the distance between the current node, and all of its neighbours.
- From its neighbouring nodes it selects the one with the smallest distance.

Iterative Improvement

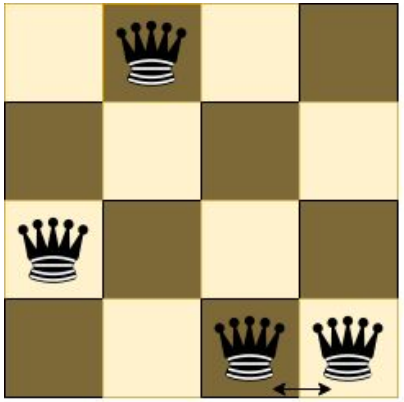
- Typically applies to optimization problems in which we make a set of choices in order to arrive at an optimal solution.
- Starts with some easily obtainable approximation to a solution and improves upon it by repeated applications of some simple step.
- To validate such an algorithm, one needs to make sure that the algorithm in question does stop after a finite number of steps, and that the final approximation obtained indeed solves the problem.

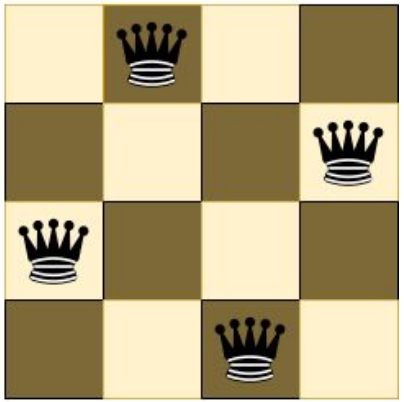
Iterative Improvement - N-Queens Problem

- Place **n** queens on an **n** x **n** chessboard.
- No two queens on the same row, column, diagonal.

Iterative improvement:

- Start with one queen in each column.
- Move a queen to reduce number of conflicts.



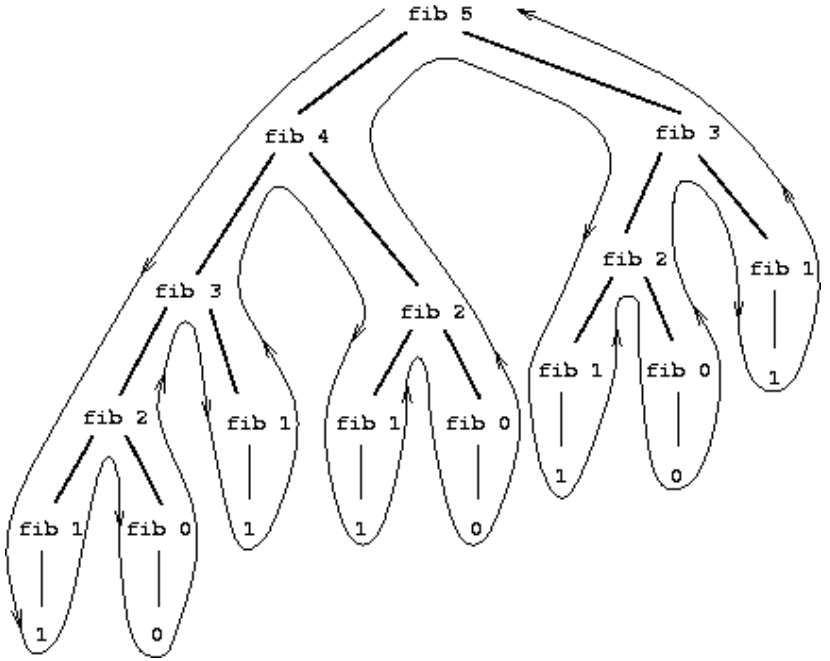


Dynamic Programming

Is a technique for solving problems with overlapping subproblems. Rather than solving overlapping subproblems again and again, it suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Dynamic Programming - Fibonacci

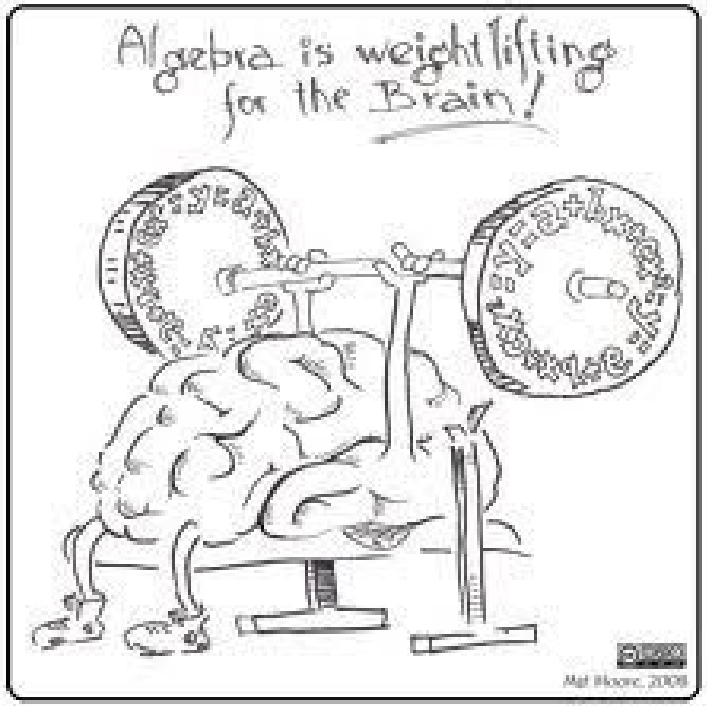
$$Fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ Fib(n-1) + Fib(n-2) & otherwise \end{cases}$$



References

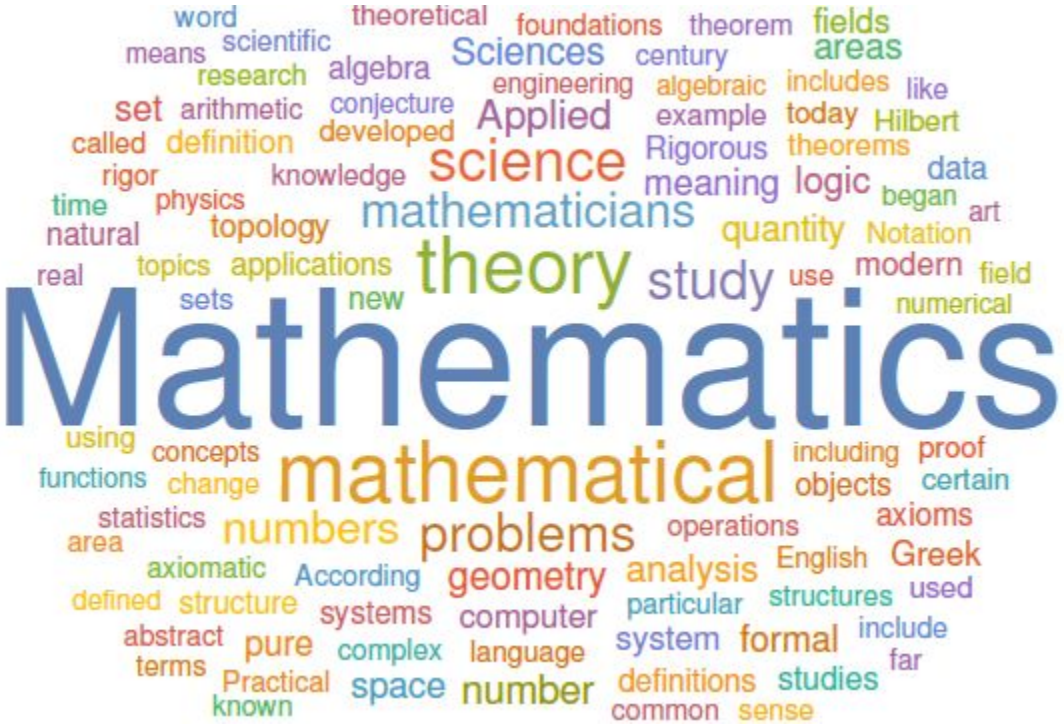
1. Introduction to Algorithms (3rd Edition): Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: 9780262033848. 3-29, 43-60.
2. The Algorithm Design Manual (2nd Edition): Steven S Skiena: 9781848000698. 3-57.
3. Algorithms (4th Edition): Robert Sedgewick, Kevin Wayne: 9780321573513. 172-205.
4. Algorithmic Puzzles (1st Edition): Anany Levitin, Maria Levitin: 9780199740444. 3-22.
5. Design and analysis of algorithms course by Stanford: https://www.coursera.org/specializations/algorithms

Exercises



Appendix A - Mathematical Background

```
WordCloud[DeleteStopwords[WikipediaData["mathematics"]]]
```



Some definitions

Description	Notation	Definition
floor	$\lfloor x \rfloor$	Largest integer not greater than x.
ceiling	$\lceil x \rceil$	Smallest integer not smaller than x.
natural logarithm	$\ln N$	$\log_e$ N (x such that e^x = N)
binary logarithm	$\lg N$	$\log_2$ N (x such that 2^x = N)
factorial	N!	1 x 2 x 3 x 4 x … x *N*
divisibility	**b|a**, b\a	For integers a and b≠0, the expression means a is divisible by b, i.e., there is an integer q such that a = bq. The statement b|a is read as b divides a.

Some definitions (continued)

Description	Notation	Definition
Summation - finite	$\sum_{k=1}^{n} a_k$	Given a sequence $a_1, a_2, \ldots, a_n$ of numbers, where n is a nonnegative integer, the expression means: $a_1 + a_2 + \cdots + a_n$. If n = 0 the value of the summation is defined to be 0. The value of a finite series is always well defined, and we can add its terms in any order.
Summation - infinite	$\sum_{k=1}^{\infty} a_k$	Given an infinite sequence $a_1, a_2, \ldots$ of numbers, the expression means: $a_1 + a_2 + \cdots$. Interpreted to mean $\lim_{n\to\infty} \sum_{k=1}^{n} a_k$

Some definitions (continued)

Symbol	Meaning
$p \wedge q$	Conjunction of p and q. The statement is true when both p and q are true, and is false otherwise.
$p \Leftrightarrow q$	Biconditional of p and q. The statement is the proposition "p if and only if (iff) q.". It is true when p and q have the same truth values, and is false otherwise.
$\exists_x P(x)$	Existential quantification of P(x) is the proposition "There exists an element x in the domain such that P(x)." $\exists$ is called the *existential quantifier*.
$x \in S$	x is a member of S. The objects in a set are called the elements, or members, of the set.
$\{x \mid P(x)\}$	Set builder notation which describes a set. We characterize all those elements in the set by stating the property or properties they must have to be members.

Modular Arithmetic

For any integer a and any positive integer n, the value a mod n is the **remainder** (or **residue**) of the quotient a / n:

$a \mod n = a - n\lfloor \frac{a}{n} \rfloor$

It follows that: $0 \leq a \mod n < n$.

If $(a \mod n) = (b \mod n)$, we write $a \equiv b \pmod{n}$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod{n} \Leftrightarrow (b - a) \backslash n$

If $(a \mod n) \neq (b \mod n)$, we write $a \not\equiv b \pmod{n}$.

Polynomials

Given a nonnegative integer d, a ***polynomial in n of degree d*** is a function p(n) of the form

$$p(n) = \sum_{i=0}^{d} a_i n^i = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_2 n^2 + a_1 n + a_0,$$

where the constants $a_0, a_1, ..., a_d$ are the ***coefficients*** of the polynomial and $a_d \neq 0$. A polynomial is asympotically positive if and only if $a_d > 0$. For an asymptotically positive polynomial p(n) of degree d, we have $p(n) = \theta(n^d)$. We say that a function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponentials

For all real a > 0, m, and n, we have the following identities:

$a^0 = 1$
$a^1 = a$
$a^{-1} = 1/a$
$(a^m)^n = a^{m\,n}$
$(a^m)^n = a^{n\,m}$
$a^m a^n = a^{m+n}$

We can relate the rates of growth of polynomials and exponentials by the following fact. For all real constants a and b such that $a > 1$,

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

Thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

Logarithms

We shall use the following notations:

$$\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)},\\
\ln n &= \log_e n && \text{(natural logarithm)},\\
\lg^k n &= (\lg n)^k && \text{(exponentiation)},\\
\lg \lg n &= \lg (\lg n) && \text{(composition)}.
\end{aligned}$$

An important notational convention we shall adopt is that logarithm functions will apply only to the next term in the formula, so that $\lg n + k$ will mean $(\lg n) + k$ and not $\lg (n + k)$. If we hold $b > 1$ constant, then for $n > 0$, the function $\log_b n$ is strictly increasing.

Logarithms - Rules

For all real $a > 0, b > 0, c > 0$, and n,

$$
\begin{aligned}
a &= b^{\log_b a} \\
\log_c ab &= \log_c a + \log_c b \\
\log_b a^n &= n \log_b a \\
\log_b a &= \frac{\log_c a}{\log_c b} \\
\log_b 1/a &= -\log_b a \\
\log_b a &= \frac{1}{\log_a b} \\
a^{\log_b c} &= c^{\log_b a}
\end{aligned}
$$

where, in each equation above, logarithm bases are not 1.

Factorials

The notation $n!$ (read "n factorial") is defined for integers $n \geq 0$ as

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

Thus, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

A weak upper bound on the factorial function $n! \leq n^n$, since each of the n terms in the factorial product is at most n.

Appendix B - Gentle Introduction to Algorithm Analysis

Adding Functions

- $O(f(n)) + O(g(n)) \rightarrow O(\max(f(n), g(n)))$
- $\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(\max(f(n), g(n)))$
- $\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(\max(f(n), g(n)))$

Multiplying Functions

- $O(c \cdot f(n)) \rightarrow O(f(n))$
- $\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$
- $\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$
- $O(f(n)) \cdot O(g(n)) \rightarrow O(f(n) \cdot g(n))$
- $\Omega(f(n)) \cdot \Omega(g(n)) \rightarrow \Omega(f(n) \cdot g(n))$
- $\Theta(f(n)) \cdot \Theta(g(n)) \rightarrow \Theta(f(n) \cdot g(n))$