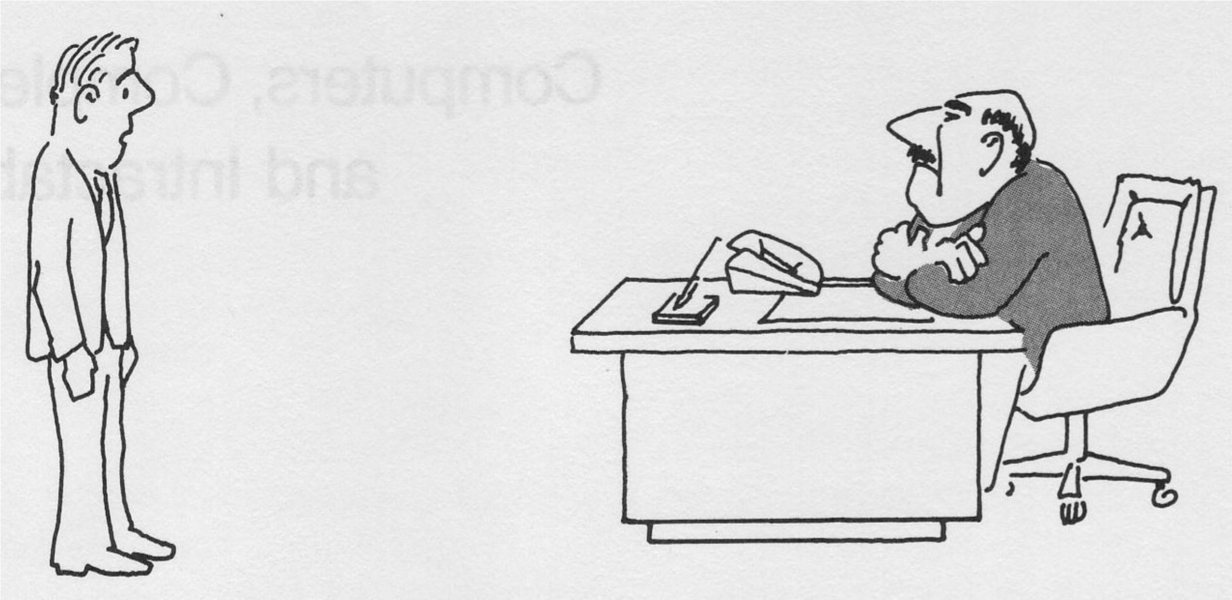
Limits to Computation

“B” Problem

- Your company entered the highly competitive “bandersnatch” market
- The bandersnatch design dominates the production process.
- Your boss asks you to write an efficient algorithm to solve the design process problem.
- After hours of hair-splitting, you can only come up with some “brute-force” approach (i.e. searching for all possible combinations).
- Since you took CSC2200, you are able to analyze the problem and conclude that it takes exponential time!

“B” Problem

You may find yourself in the following embarrassing situation:

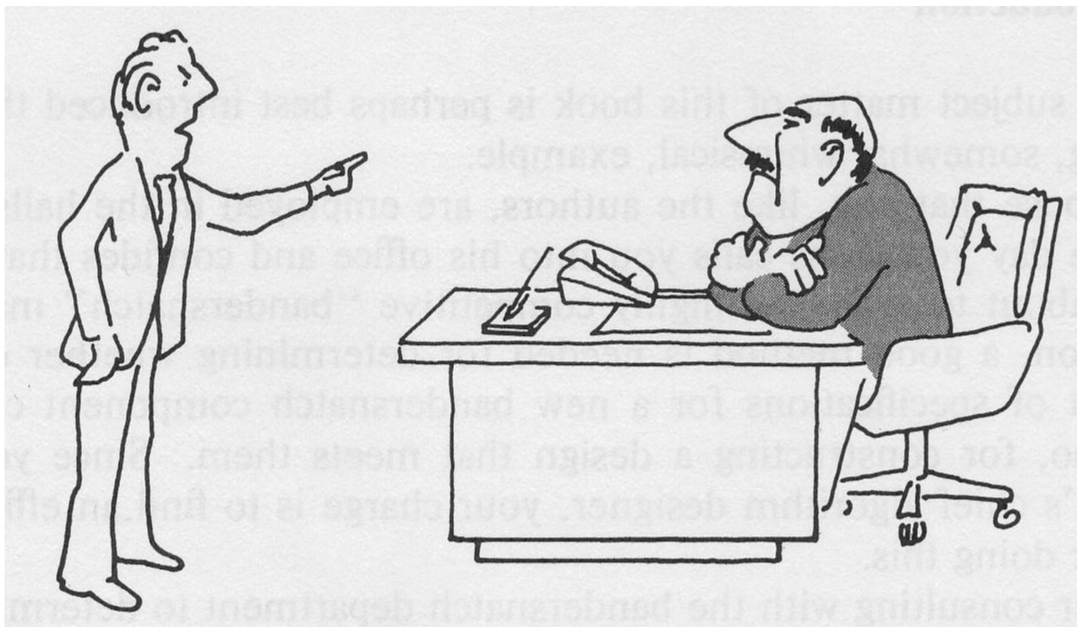


[Garey & Johnson, '79]

“I can't find an efficient algorithm, I guess I'm just too dumb.”

“B” Problem

You wish you could say to your boss:



[Garey & Johnson, '79]

“I can't find an efficient algorithm, because no such algorithm is possible.”

"B" Problem

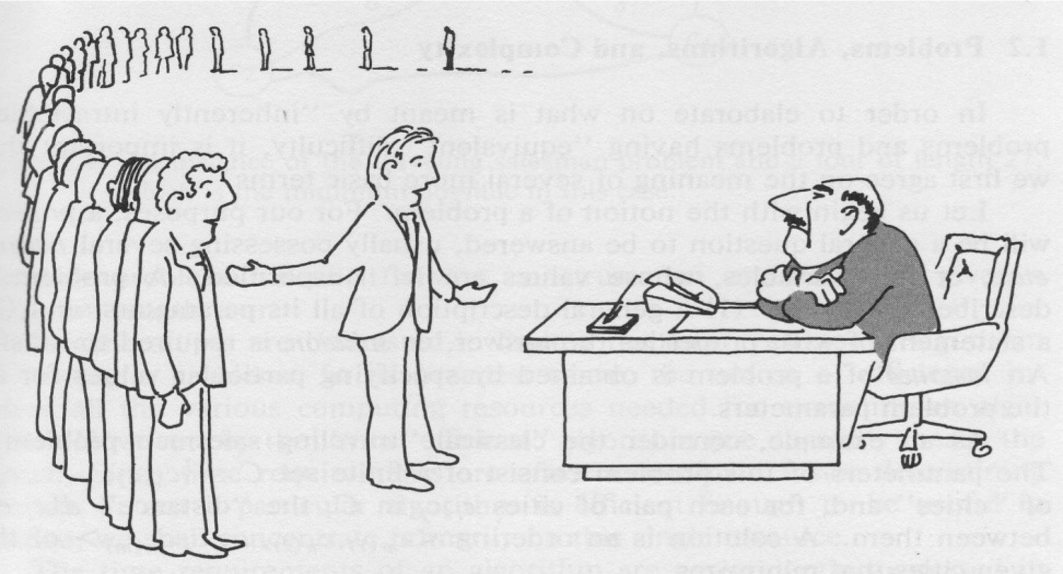
- For most problems, it is very hard to prove their intractability, because most practical problems belong to a class of well-studied problems called NP.
- The "hardest" problems in NP are the NP-complete problems:

  If you prove that one NP-complete problem can be solved by a polynomial-time algorithm, then it follows that all the problems in NP can be solved by a polynomial-time algorithm.
- Conversely, if you show that one particular problem in NP is intractable, then all NP-complete problems would be intractable.
- NP-complete problems seem intractable.
- Nobody has been able to prove that NP-complete problems are intractable.

“B” Problem

By mastering the basics of the theory of NP-completeness, you may be able to prove that the problem given by your boss is NP-complete.

In this case, you can say to your boss:



[Garey & Johnson, '79]

“I can't find an efficient algorithm, but neither can all these famous people."

“B” Problem

or alternatively:

“If I were able to design an efficient algorithm for this problem, I wouldn't be working for you! I would have claimed a prize of $1 million from the Clay Mathematics Institute."

- After the second argument, your boss will probably give up on the search for an efficient algorithm for the problem.
- But the need for a solution does not disappear like that...
- Possible Solution: use an efficient approximation algorithm.

See: `http://www.claymath.org/millenium-problems/p-vs-np-problem`

for details on how to get a $1 million prize (P?=NP)

Decidability

- *Problems:*
  - *Decidable:* if there exists an algorithm to solve the problem
  - *Undecidable:* there is no algorithm that can always give the correct answer

    They are so hard that they cannot be solved!
- Godel (1931):

  proved that there exist undecidable problems => incompleteness theorem

  (in certain systems completeness and consistency cannot be achieved simultaneously)
- Turing (1936):

  proved that as long as the problem remains unsolved there is absolutely no way of ascertaining whether it is undecidable or simply difficult.

Undecidable Problems: Example

- *The halting problem:*

  Given a description of any algorithm and a description of its initial arguments,

  determine whether the algorithm, when executed with these arguments, ever halts.

  (the alternative is that it runs forever without halting).

- Intuitive argument: such algorithm will have a hard time checking itself!

Undecidable Problems: Example

- Proof: given the function halt (s, i) we can construct:

function trouble(*string* s)

    if halt(s, s) = **false**

        return true

    else

        loop forever

Q: String t represents the function trouble. Does trouble(t) halt?

- Assume that trouble(t) halts => halt(t,t) returns **true**, but that in turn indicates that trouble(t) does not halt. Contradiction.
- Assume that trouble(t) does not halt. => halt(t,t) returns **false.** But that in turn would mean that trouble(t) does halt. Contradiction.

=> Algorithm halt does not exists.

Halting problem ⇔ Liar's paradox

(“Epimenides the Cretan says: All Cretans are liars”, ~600 BC)

Other Undecidable Problems

- *Determining the Kolmogorov complexity of a string*

  (determine the length of the shortest program that produces the string)

- *Hilbert 10^{th} problem*

  (deciding whether a Diophantine equation has integer solutions)

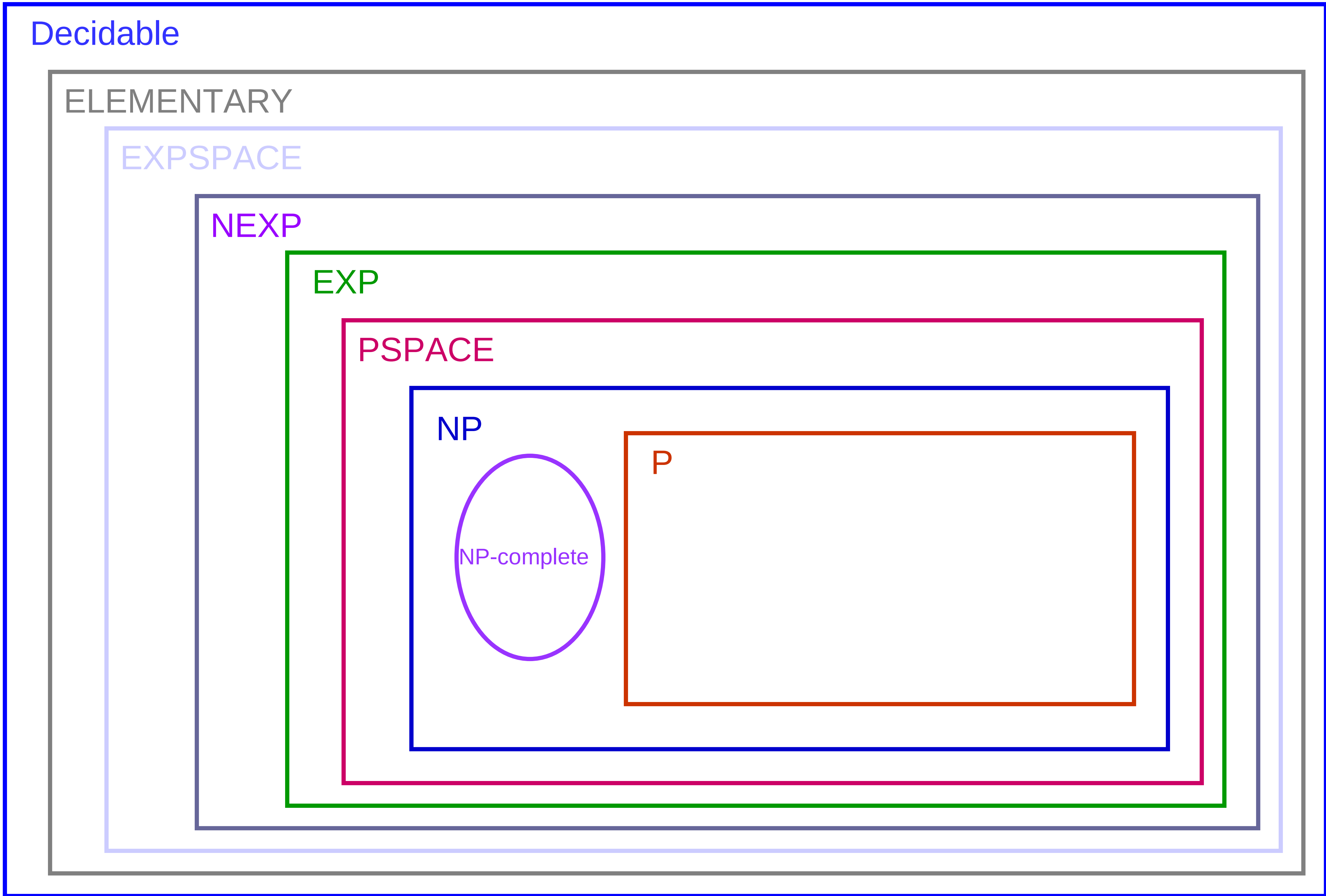
- *Matrix mortality problem*

  (given a finite set of square integer matrices, decide whether some product of these matrices results in the zero matrix)

Complexity Classes

- **ELEMENTARY:** the set of all decision problems solvable by a deterministic Turing machine in time $O\left(2^{2^{\cdot^{\cdot^{\cdot^{2^{p(n)}}}}}}\right)$ ($p(n)$) is a polynomial function)
- **EXPSPACE:** the set of all decision problems solvable by a deterministic machine in O($2^{p(n)}$) space
- **NEXP:** the set of all decision problems solvable by a non-deterministic Turing machine in O($2^{p(n)}$) time
- **EXP:** the set of all decision problems solvable by a deterministic machine in O($2^{p(n)}$) time
- **PSPACE:** the set of all decision problems solvable by a deterministic machine using a polynomial amount of memory
- **NP:** the set of decision problems solvable in polynomial time on a non-deterministic machine.
- **NP-complete:** A problem is NP-complete if it is in NP and if every other problem in NP can be reduced to it in polynomial time on a deterministic machine.
- **P:** the set of decision problems solvable in polynomial time on a deterministic machine.

Complexity Classes



NP-Complete Problems

- *Traveling salesman problem*
- *Scheduling in multiprocessor systems*
- *Knapsack*
- *Longest path in a graph*
- *Bin packing*

PSPACE-Complete Problems

- *Solitaire Mahjong (nxn board)*
- *Quantified Boolean formulas (QBF)*
- *Regular Expression (determining whether R generates every string over its alphabet)*

EXP-Complete Problems

- *Generalized Chess (nxn board)*

  (e.g., one king per player, other gamepiece counts increase proportionally with n; starts with random placement of gamepieces on board)
- *Generalized Go (nxn board)*
- *Generalized Checkers (nxn board)*

Goal: determine whether a specified player has a winning strategy

NEXP-Complete Problems

- *Succinct Circuit SAT*
- *Succinct Hamilton Path*

(succinct representation of a graph:

A circuit computing the adjacency matrix: given two integers (i,j) as input compute a(i,j))