Computer Science 1000: Part #10

Theoretical Computer Science

ASSESSING ALGORITHM EFFICIENCY

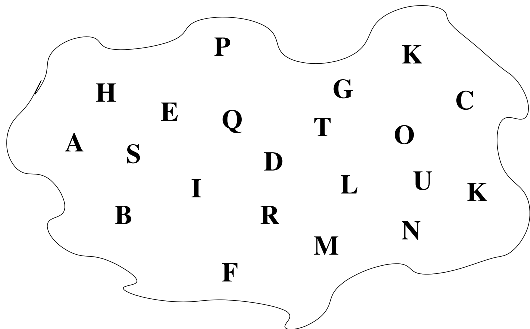
PROVING FAST UNSOLVABILITY

PROVING GENERAL UNSOLVABILITY

Assessing Algorithm Efficiency

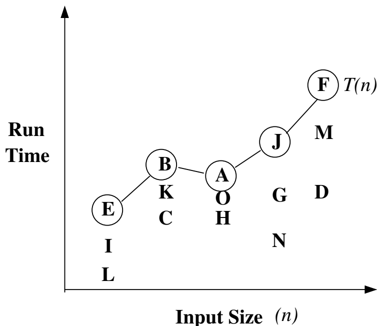
- In general, the best algorithm is the one with the lowest running time.
- Comparing algorithms by raw running time problematic:
  - Raw running times machine / language / OS dependent.
  - Raw running times input dependent.
  - Algorithm may not be implemented in a program.
- Need an abstract mathematical conception of algorithm efficiency, phrased in terms of a function of input size n, which is easily usable and comprehensible.
- The three abstractions involved in the conception sketched here can be viewed as necessary lies.

Necessary Lie #1: Focus on Important Instructions



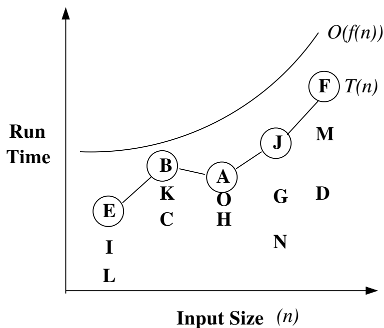
- Compute runtime on an input by counting the number of important instructions that are executed.
- Is machine-independent (raw abstract runtime).

Necessary Lie #2: Worst-Case Runtime Summary



- Group inputs by input size; summarize each size by largest runtime for that size.
- Is input-independent (worst-case abstract runtime).

Necessary Lie #3: Asymptotic Smoothing



- Reduce worst-case abstract runtime function to largest term.
- Is simple (asymptotic worst-case abstract runtime, i.e., worst-case time complexity).

Deriving Worst-Case Time Complexities

If already have worst-case abstract runtime function, select largest term, e.g.,

$$2\log n + 4 \quad \Rightarrow \quad O(\log n)$$

$$3n^2 + 1000n + 13 \quad \Rightarrow \quad O(n^2)$$

$$12n^4 + 5n^2 + 900 \quad \Rightarrow \quad O(n^4)$$

$$(3 \times 2^n) + 900n^{50} + 57 \quad \Rightarrow \quad O(2^n)$$

Deriving Worst-Case Time Complexities (Cont'd)

Otherwise, multiply out "deepest" loop-chain in algorithm, *e.g.*, $n \times n = O(n^2)$ time for Selection Sort.

```
Get values for list L and n
ENDUNSORTED = n
While (ENDUNSORTED > 1) do
    FOUNDPOS = 1
    for INDEX = 2 to ENDUNSORTED do
        If L_INDEX > L_FOUNDPOS then
            FOUNDPOS = INDEX
    TMP = L_ENDUNSORTED
    L_ENDUNSORTED = L_FOUNDPOS
    L_FOUNDPOS = TMP
    ENDUNSORTED = ENDUNSORTED - 1
```

. . . And This All Matters Because? . . .

- Considering all instructions rather than just important ones only increases raw abstract runtime and hence worst-case abstract runtimes by constant multiplicative factors. But this is not the problem that if we want to solve larger and larger inputs (**living in Asymptopia**).
- Could consider best- or average-case time complexity; however, worst-case best when trying to plan using algorithm-produced results in the real world, e.g., aircraft collision avoidance.
- Asymptotic smoothing gives much simpler functions than worst-case abstract runtimes, which eases algorithm comparison.

Time Complexity Orders of Magnitude

$O(\log n)$	Logarithmic Time	(Binary Search)
$O(n)$	Linear Time	(Sequential Search)
$O(n^2)$	Quadratic Time	(List Sort)
$O(2^n)$	Exponential Time	(Bin Packing #1)

Polynomial Time $= O(n^c)$ time for constant c

Table of Doom (1 Gigaflop/s Version)

Input Size (n)	Time Complexity				
	B-Search ($\log_2 n$)	S-Search (n)	Sort (n^2)	MST (n^3)	BP#1,IF (2^n)
10	< 1 second	< 1 second	< 1 second	< 1 second	< 1 second
50	< 1 second	< 1 second	< 1 second	< 1 second	13 days
100	< 1 second	< 1 second	< 1 second	< 1 second	4×10^{13} years
1000	< 1 second	< 1 second	< 1 second	1 second	4×10^{284} years
one million	< 1 second	< 1 second	2 minutes	30 years	–
300 million	< 1 second	< 1 second	10 days	9×10^5 years	–
five billion	< 1 second	5 seconds	8 centuries	4×10^{12} years	–

Proving Fast Unsolvability

- Some problems are solvable in polynomial time, e.g., binary search, list sorting, and can be solved in practice for large input sizes; some, e.,g., Bin Packing and Integer Factorization, cannot.
- With problems that are not known to be solvable in polynomial time, have we just not thought of a good algorithm yet, or are they genuinely intractable?

HOW CAN WE PROVE FAST UNSOLVABILITY?

The Key to Proving Fast Unsolvability: Arm Wrestling

Arnold **Betty**

Best in Two?
The Logic of Pairwise Comparison

If we know that

$$\textbf{Arnold} \quad \textit{beaten by} \quad \textbf{Betty} .$$

and Betty is easy to beat, what do we know about Arnold?

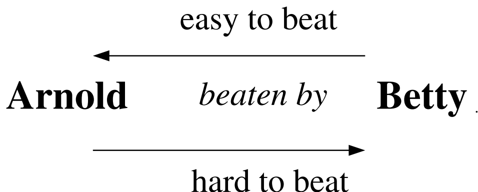
Best in Two?
The Logic of Pairwise Comparison (Cont'd)

If we know that

Arnold *beaten by* **Betty** .

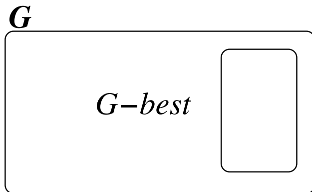
and Arnold is hard to beat, what do we know about Betty?

Best in Two?
The Logic of Pairwise Comparison (Cont'd)



- Establish better arm wrestler by a two-person match.
- If Arnold is beaten by Betty:
  1. Arnold is no better than Betty
     (if Betty is easy to beat then Arnold is easy to beat)
  2. Betty is at least as good as Arnold
     (if Arnold is hard to beat then Betty is hard to beat)

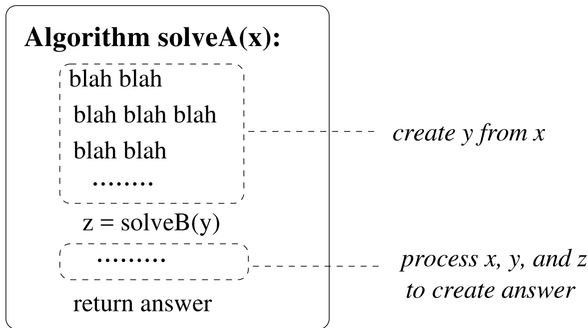
Best in Group?
Pairwise Comparison in Groups



- Establish best arm wrestler in group G by a tournament composed of two-person matches.
- The winner of a tournament is at least as good as everybody else in the group.

Reductions between Problems

A **reduction** from problem $\mathcal{A}$ to problem $\mathcal{B}$ ($\mathcal{A}$ **reduces to** $\mathcal{B}$) is an algorithm for solving $\mathcal{A}$ that uses an algorithm for solving $\mathcal{B}$.



Focus here on polynomial-time (pt-) reductions, i.e., all dashed-box processing above done in polynomial time.

Hardest in Two?
The Logic of Reducibility

If we know that

$$\underset{\textbf{A}}{\textbf{Problem}} \; pt\!-\!reduces\ to \; \underset{\textbf{B}}{\textbf{Problem}}.$$

and Problem B is easy to solve, what do we know about Problem A?

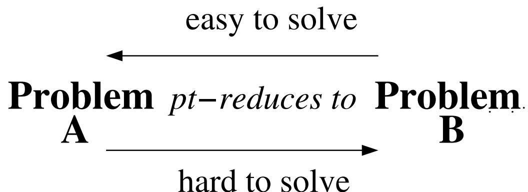
Hardest in Two? The Logic of Reducibility (Cont'd)

If we know that

$$\underset{\mathbf{A}}{\mathbf{Problem}}\ \textit{pt-reduces to}\ \underset{\mathbf{B}}{\mathbf{Problem}}.$$

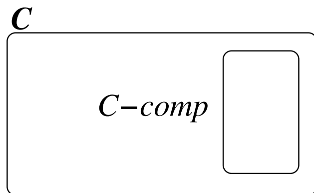
and Problem A is hard to solve, what do we know about Problem B?

Hardest in Two? The Logic of Reducibility (Cont'd)



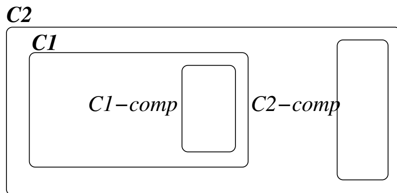
- Establish harder problem by poly-time reduction.
- If problem $\mathcal{A}$ reduces to problem $\mathcal{B}$:
  1. $\mathcal{A}$ is no harder than $\mathcal{B}$
     (if $\mathcal{B}$ is easy to solve then $\mathcal{A}$ is easy to solve)
  2. $\mathcal{B}$ is at least as hard as $\mathcal{A}$
     (if $\mathcal{A}$ is hard to solve then $\mathcal{B}$ is hard to solve)

Hardest in Class?
Reducibility in Classes



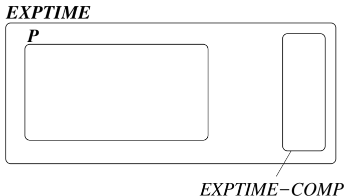
- Establish hardest problem in class C by reductions.
- The hardest problems in C (the C-Complete problems) are at least as hard as any problem in C.

Harder than Class?
The Logic of Class Inclusion (Part I)



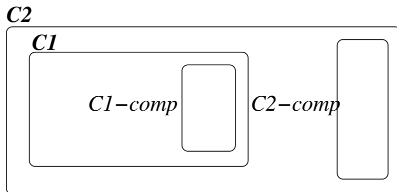
- Suppose we have two classes C_1 and C_2 such that C_1 is fully contained in C_2.
- If a problem is C_2-Complete, then that problem is properly harder than any problem in C_1 and hence not in C_1.

Harder than Poly-Time? The Logic of $EXPTIME$-Completeness



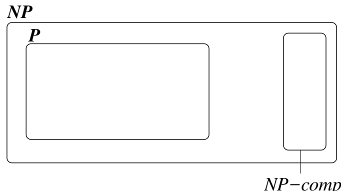
- Let P and $EXPTIME$ be the classes of poly-time and exponential-time solvable problems, respectively.
- As we know that P is fully contained in $EXPTIME$, i.e., $P \neq EXPTIME$, if a problem is $EXPTIME$-Complete, then that problem is not poly-time solvable.

Harder than Class? The Logic of Class Inclusion (Part II)



- Suppose we have two classes C_1 and C_2 such that C_1 is contained (but not necessarily fully contained) in C_2.
- If a problem is C_2-Complete, then that problem is harder than any problem in C_1 under the conjecture that C_1 is fully contained in C_2, i.e., $C_1 \neq C_2$.

Harder than Poly-Time? The Logic of NP-Completeness



- Let P be the class of poly-time solvable problems and P be contained (but not necessarily fully contained) in class NP.
- If a problem is NP-Complete, then that problem is not poly-time solvable under the conjecture that P is fully contained in NP, i.e., $P \neq NP$.

Dealing with Intractability

- First NP-Complete problem proven in 1971; tens of thousands proven since (including Bin Packing and many other industrially-important problems (**but not Integer Factorization**)).
- Unless $P = NP$, no NP-complete problem can be solved in poly-time ... but we still need to solve these problems!!!
- Consider less restrictive types of fast solvability, e.g., polynomial-time approximability (Bin Packing #2), and fall back on non-polynomial time algorithms if necessary ...

... assuming every problem has an algorithm ...

Proving General Unsolvability: An Overview

Alan Turing
(1912-1954)

Turing A. M.. “On computable numbers, with an application to the Entscheidungsproblem*.”* Proceedings of the London Mathematical Society*, 2 s. vol. 42 (1936–1937), pp. 230–265.*

Proving General Unsolvability: An Overview (Cont'd)

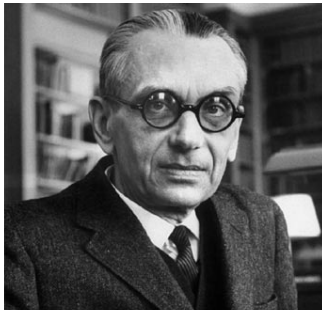
- To investigate the question of whether every problem has an algorithm, i.e., whether every problem is solvable, need basic model of computation (cf. computational time complexity as basic model for investigating algorithm runtime).
- Typical properties of a model compared to the real thing being modeled:
  1. Captures important properties of real thing.
  2. Probably differs in scale from real thing.
  3. Omits some details of real thing.
  4. Lacks full functionality of real thing.

Proving General Unsolvability: An Overview (Cont'd)

- Every model based on assumptions, and information derived with a model is only as good as those assumptions.
- Necessary properties of a model of a computing agent:
  1. Accepts input.
  2. Can store and retrieve information wrt memory.
  3. Acts on stored algorithm instructions based on the current state of and the data item currently being processed by the agent.
  4. Produces output.
- Many models of computation proposed in early 20th century in response to Hilbert's Program.

Proving General Unsolvability: An Overview (Cont'd)

David Hilbert
(1862–1943)



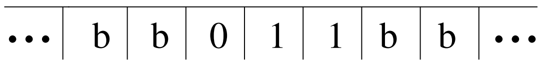
Kurt Gödel
(1906–1978)

- Hilbert (1920): Formalize mathematical proof to eliminate ambiguities and allow automation of proof.
- Gödel (1931): Every reasonable arithmetic system has true statements that are unprovable in that system.

Turing Machines

(S1,0) **configuration**

tape



read/write head

S1

state

1. ((S1,0),(1,S2,R))
2. ((S1,1),(1,S2,R))
3. ((S2,0),(1,S2,R))
4. ((S2,1),(0,S2,R))
5. ((S2,b),(b,S3,L))

instructions

Turing Machines (Cont'd)

- A **Turing Machine** consists of (1) a two-way infinite tape, (2) a tape-square alphabet, (3) a read/write head that can be positioned on any tape square, (4) a set of states $\{$ **S1, S2, . . . , Sn** $\}$, and (5) a set of instructions.
- The tape functions as input, memory, and output at TM start, execution, and termination.
- The alphabet can be any number of symbols plus a special blank (**b**) symbol; focus here on the alphabet $\{$ **b, 0, 1** $\}$.
- At any given time, a TM is in a particular state **Si** and the read/write head is reading symbol x; this pair (**Si,** x) is called the TM's **configuration**.

Turing Machines (Cont'd)

- A TM instruction specifies what the TM does next when it is in a specified configuration, e.g.,

```
if state is S1 and symbol is 0 then
    write 1 in current tape square
    set state to S2
    move r/w head one square right
```

$$\Updownarrow$$

((1,0),(1,2,R))

Turing Machines (Cont'd)

- Starting from an initial configuration, a TM executes instructions until it halts.
- TM operation conventions:
  1. The input is placed on the TM tape.
  2. The initial position of the TM read/write head is the leftmost non-blank tape square, i.e., the leftmost square of the input.
  3. The initial TM state is S1.
  4. At each point, there is at most one instruction that matches the current TM configuration, i.e., the TM is **deterministic**.
  5. The TM halts when there is no instruction that matches the current TM configuration.
  6. On halting, the output is the contents of the TM tape.
- Note that TM instructions execute in TM-configuration order, not instruction-order, cf. Python programs.

Turing Machines (Cont'd)

initial

...	b	b	0	1	1	b	b	...

S1

1. ((S1,0),(1,S2,R))

...	b	b	1	1	1	b	b	...

S2

4. ((S2,1),(0,S2,R))

...	b	b	1	0	1	b	b	...

S2

4. ((S2,1),(0,S2,R))

...	b	b	1	0	0	b	b	...

S2

5. ((S2,b),(b,S3,L))

...	b	b	1	0	0	b	b	...

S3 **halt**

Turing Machines (Cont'd)

initial

…	b	b	1	1	0	b	b	…

S1

2. ((S1,1),(1,S2,R))

…	b	b	1	1	0	b	b	…

S2

4. ((S2,1),(0,S2,R))

…	b	b	1	0	0	b	b	…

S2

3. ((S2,0),(1,S2,R))

…	b	b	1	0	1	b	b	…

S2

5. ((S2,b),(b,S3,L))

…	b	b	1	0	1	b	b	…

S3 **halt**

Turing Machines (Cont'd)

- A TM is an adequate model of computing agent:
  1. **Accepts input**: TM encodes input on and reads symbols from tape.
  2. **Can store and retrieve information wrt memory**: During execution, TM writes symbols on and later can read these symbols from tape.
  3. **Acts on stored algorithm instructions based on the current state of and the data item currently being processed by the agent**: TM configuration dictates executed instruction.
  4. **Produces output**: If TM halts, tape is output.
- TM are more capable from real computers because TM tape (memory) is unlimited; hence, a task that is TM-solvable might not be real-computer-solvable.

Turing Machines (Cont'd)

- A TM instruction-set is an algorithm:
  1. **Is well-ordered**: As our TM are deterministic, at most one instruction executable for any TM configuration.
  2. **Consists of unambiguous and effectively computable operations**: TM instructions are unambiguous to TMs.
  3. **Halts in finite time**: Relative to TM-appropriate inputs, a TM always halts (appropriate inputs also key to algorithms halting).
  4. **Produces output**: Output is tape contents after execution and halting on TM-appropriate input.
- When we write a TM for a task, we write a set of TM instructions to do that task.

Proving General Unsolvability: The Church-Turing Thesis

- We know that every TM is an algorithm — does every algorithm have a corresponding TM?

The Church-Turing Thesis: For every symbol-manipulation algorithm there is a TM.

- Not provable, but two lines of evidence:
  1. Every proposed s-m algorithm has a TM.
  2. TM can simulate and is thus equivalent to other proposed models of computation.
- C-T Thesis $\Rightarrow$ TM defines limits of solvability!

Proving General Unsolvability: The Church-Turing Thesis (Cont'd)

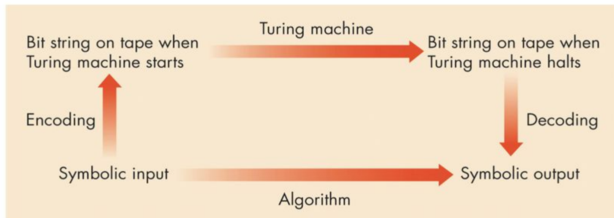


Figure 11.9

Emulating an Algorithm by a Turing Machine

Proving General Unsolvability: The Halting Problem

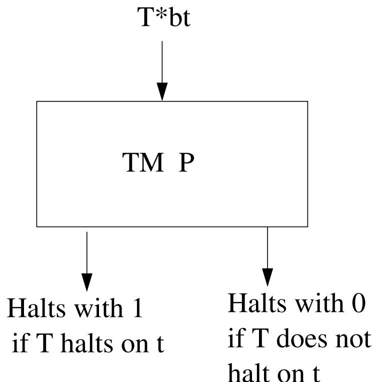
- Easy to prove if a given TM halts on a given configuration; what about if a given TM halts on a given input (**Halting Problem**), e.g., does the TM with instruction-set

  ((S1,b),(b,S1,R))
  ((S1,0),(0,S1,R))
  ((S1,1),(1,S1,R))

  halt on input tape ...b000b ...?
- Prove that HP is unsolvable by contradiction — that is, start by assuming that HP is solvable and then derive something that is impossible, which is a contradiction and would hence imply that HP is not solvable.

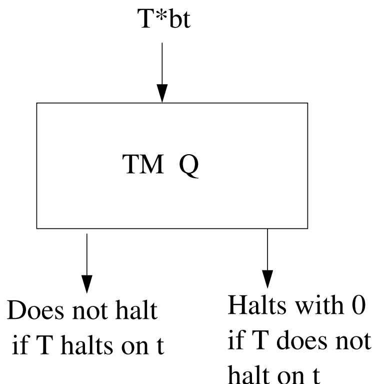
Proving General Unsolvability: The Halting Problem (Cont'd)

Suppose you have a TM P that solves HP:



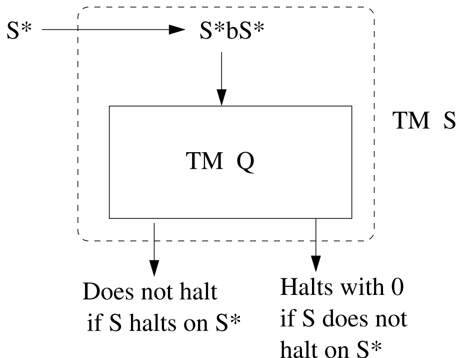
Proving General Unsolvability: The Halting Problem (Cont'd)

Modify TM P to create TM Q:



Proving General Unsolvability: The Halting Problem (Cont'd)

Modify TM Q to create TM S:



...which is impossible — hence, HP is not solvable!

Proving General Unsolvability: The Halting Problem (Cont'd)

- The unsolvability of HP has practical consequences:
  - No program can decide if a given program halts on all possible inputs.
  - No program can decide if two given programs produce the same output for all possible inputs.
  - No program can decide if a given program run on a given input will produce a given output.
- **The Fine Print**: All of this unsolvability holds in general, i.e., relative to **all** possible programs and inputs — there may yet be programs that work relative to specific classes of given programs, e.g., programs that halt in $\leq 10^9$ steps.

. . . And If You Liked This . . .

- MUN Computer Science courses on this area:
  - COMP 2002: Data Structures and Algorithms
  - COMP 4740: Design and Analysis of Algorithms
  - COMP 4741: Formal Languages and Computability
  - COMP 4742: Computational Complexity
- MUN Computer Science professors teaching courses / doing research in in this area:
  - Miklos Bartha
  - Antonina Kolokolova
  - Manrique Mata-Montero
  - Todd Wareham