CSC 421: Algorithm Design & Analysis

Spring 2019

Complexity & Computability

- classifying problem complexity
- tractability, decidability
- P vs. NP, Turing machines
- NP-complete, reductions
- approximation algorithms, genetic algorithms

1

Classifying problem complexity

throughout this class, we have considered problems, designed algorithms, and classified their efficiency

- e.g., sorting a list – could use $O(N^2)$ selection sort or $O(N \log N)$ quick sort
- big-Oh provides for direct comparison between two algorithms

when is a problem too difficult?

EXTREMELY LOW BAR: we say that a problem is *intractable* if there does not exist a polynomial time O(p(n)) algorithm that solves it

$\theta(2^N)$ is definitely intractable

note: N = 20 → millions of steps
2^{60} > # of seconds since Big Bang
2^{273} > # of atoms in the universe

but $\theta(N^{100})$ is tractable?!?

in reality, anything worse than N^3 is not practical

2

Beyond intractable

Turing showed there are UNSOLVABLE problems (regardless of efficiency)

THE HALTING PROBLEM: Given a computer program and an input to it, determine whether the program will halt on the input.

Proof by contradiction:

Assume that there is a program that solves the Halting Problem.

```
def haltsOn(sourceFile, dataFile):
    """Returns true if the program in sourceFile halts when
       executed on the data in dataFile; else false."""
    return ???
```

We need to show that the existence of this program leads to a logical contradiction.

- If the steps leading to a contradiction are valid, must conclude that the original assumption was wrong (and that there cannot be such a program).

3

Halting problem proof (cont.)

```
def haltsOn(sourceFile, dataFile):
    """Returns true if the program in sourceFile halts when
       executed on the data in dataFile; else false."""
    return ???
```

Note: a program is stored in a text file, so can be input to another program (stored in file hp.py):

```
def haltsOnlyIfSelfieDoesnt(sourceFile):
    """Returns true if the program in sourceFile does not halt
       when run with itself as input; else loops forever."""
    if haltsOn(sourceFile, sourceFile):
        while true:
            print("infinite loop")
    else:
        return true
```

Question: does the following call halt?

```
haltsOnlyIfSelfieDoesnt("hp.py");
```

if `haltsOnlyIfSelfieDoesnt("hp.py")` halts

then, according to the code, this means that

`haltsOn("hp.py", "hp.py")` returns false

which, by assumption, means that

the program in hp.py does not halt on input file hp.py

but the program in hp.py is haltsOnlyIfSelfieDoesnt, so same as

`haltsOnlyIfSelfieDoesnt("hp.py")` does not halt

CONTRADICTION!

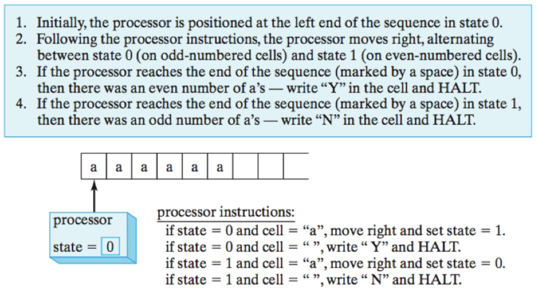
4

Impact of Turing's proof

Turing's proof of the Halting Problem in 1936 had profound impact

- it proved that there are problems that are unsolvable
- not only that, but easily understood (and potentially useful) unsolvable problems

- for his proof, Turing introduced a simple, theoretical model of computation
- a Turing Machine (TM) consists of:
  1. a potentially infinite tape on which characters can be written
  2. a processing unit that can read and write on the tape, move in either direction, and distinguish between a finite number of states

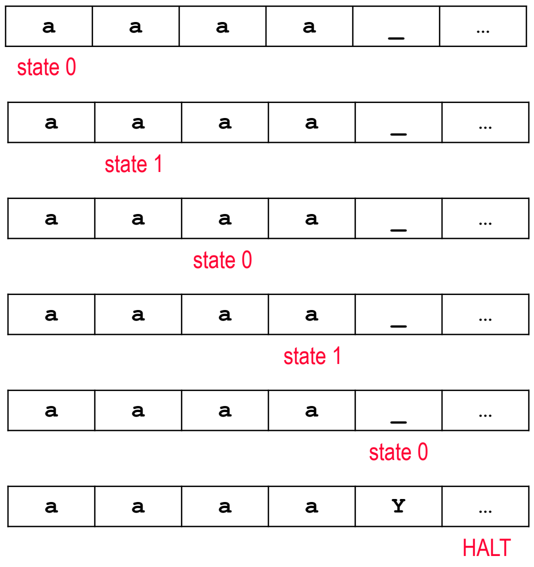


1. Initially, the processor is positioned at the left end of the sequence in state 0.
2. Following the processor instructions, the processor moves right, alternating between state 0 (on odd-numbered cells) and state 1 (on even-numbered cells).
3. If the processor reaches the end of the sequence (marked by a space) in state 0, then there was an even number of a's — write "Y" in the cell and HALT.
4. If the processor reaches the end of the sequence (marked by a space) in state 1, then there was an odd number of a's — write "N" in the cell and HALT.

processor instructions:
if state = 0 and cell = "a", move right and set state = 1.
if state = 0 and cell = " ", write " Y" and HALT.
if state = 1 and cell = "a", move right and set state = 0.
if state = 1 and cell = " ", write " N" and HALT.

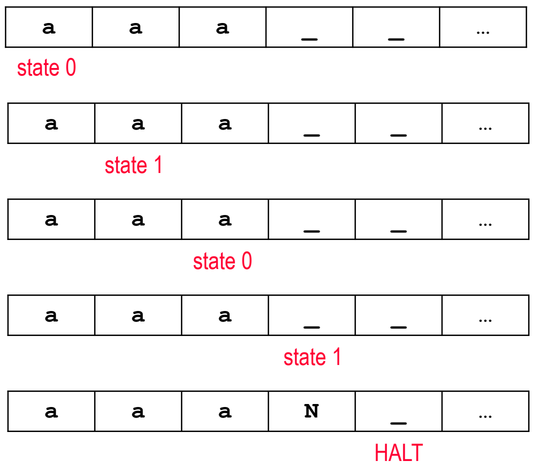
Example execution (even)

```
if in state 0 & read "a", then
  write "a", move R, goto state 1

if in state 0 and read " ", then
  write "Y", move right, goto HALT


if in state 1 & read "a", then
  write "a", move R, goto state 0

if in state 1 and read " ", then
  write "N", move right, goto HALT
```



Example execution (odd)

```
if in state 0 & read "a", then
  write "a", move R, goto state 1

if in state 0 and read " ", then
  write "Y", move right, goto HALT


if in state 1 & read "a", then
  write "a", move R, goto state 0

if in state 1 and read " ", then
  write "N", move right, goto HALT
```



7

Another example

consider the task of recognizing a power of 2

- want to be able to represent a number on a tape
- accept that number if it is of the form 2^n for some n > 0; otherwise reject

- similar to odd/even example, can represent the number in unary
  e.g., the number 4 would be represented as a sequence of 4 0's on the tape

0	0	0	0	_	…

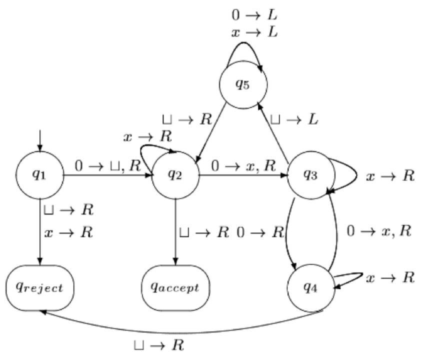
algorithm (in general terms):

1. traverse left to right, crossing off every other 0
2. if the result contains a single 0, then accept
3. if the result contains an odd number of 0's (> 1), then reject
4. otherwise, return to the left edge of the tape & repeat

8

Turing Machine program

we can define the TM using a finite state machine, or as a program

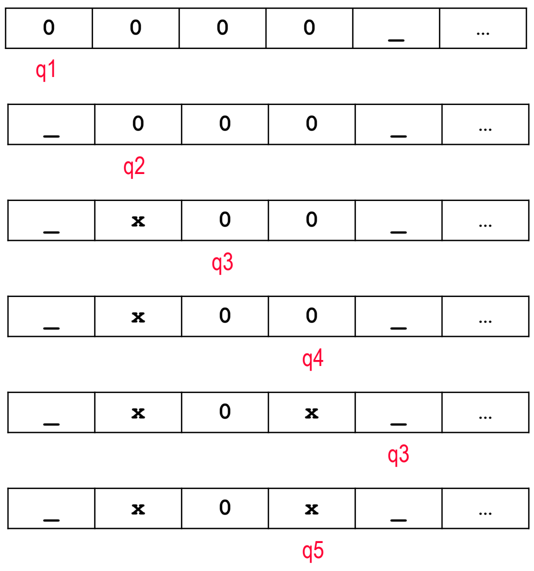


```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```

9

Example execution (accept)

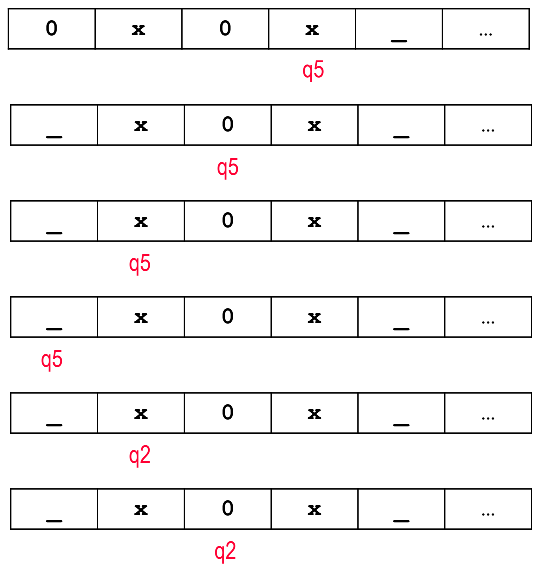
```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



10

Example execution (cont.)

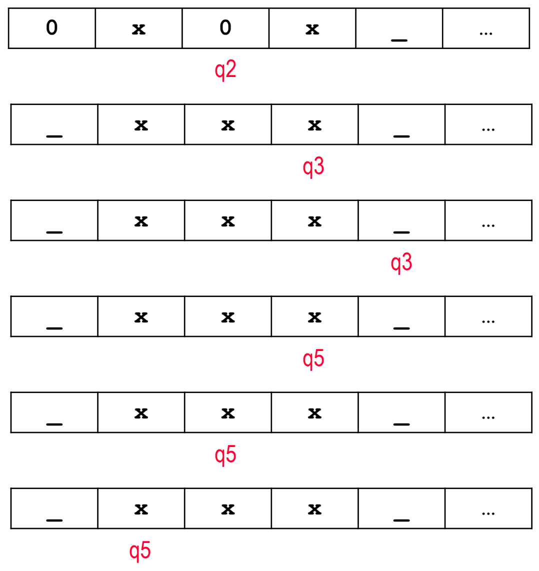
```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



11

Example execution (cont.)

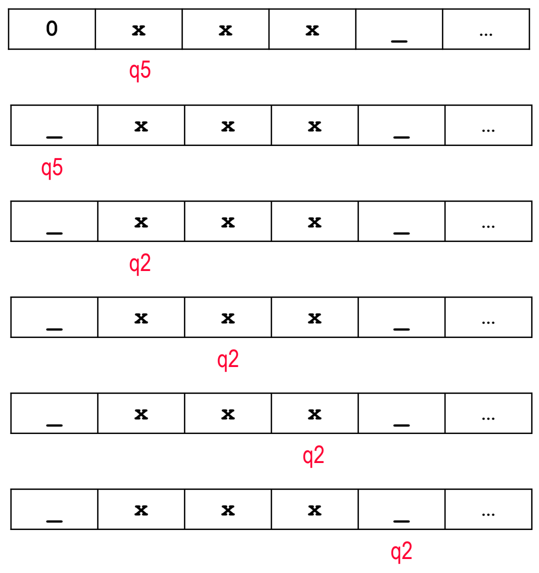
```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



12

Example execution (cont.)

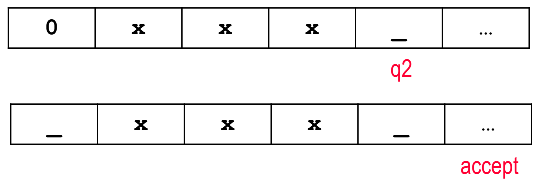
```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



13

Example execution (cont.)

```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



note that the algorithm defined by this Turing Machine is slow and cumbersome

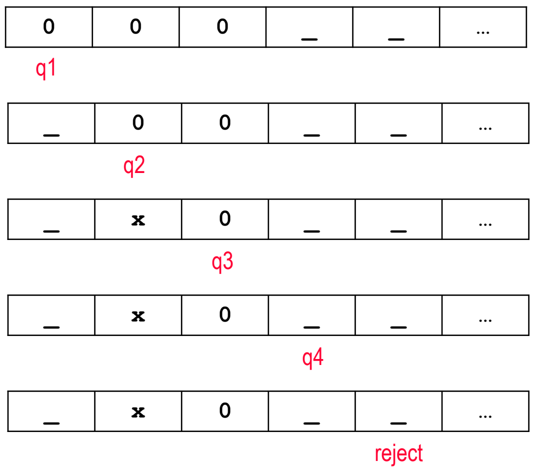
in general, the simper the model, the more steps it requires to perform a task

e.g., a machine-language program will require more instructions to complete the same task as a high-level language program

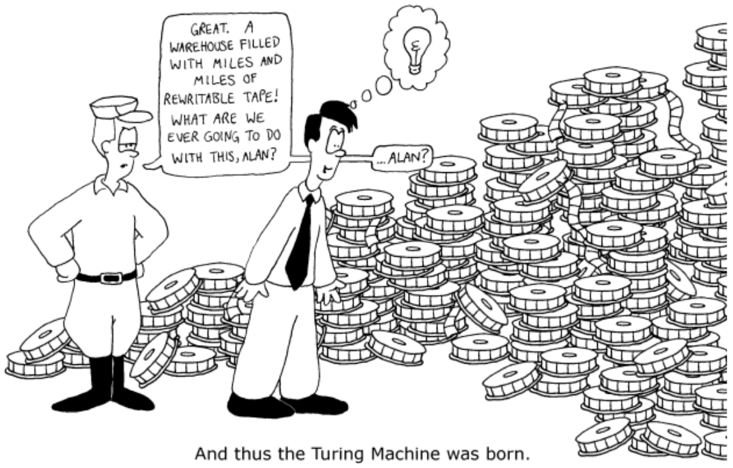
14

Example execution (reject)

```
if in q1 & read 0, then
  write _, move R, goto q2
if in q1 & read x, then
  write x, move R, goto reject
if in q1 & read _, then
  write _, move R, goto reject

if in q2 & read 0, then
  write x, move R, goto q3
if in q2 & read x, then
  write x, move R, goto q2
if in q2 & read _, then
  write _, move R, goto accept

if in q3 & read 0, then
  write 0, move R, goto q4
if in q3 & read x, then
  write x, move R, goto q3
if in q3 & read _, then
  write _, move L, goto q5

if in q4 & read 0, then
  write x, move R, goto q3
if in q4 & read x, then
  write x, move R, goto q4
if in q4 & read _, then
  write _, move R, goto reject

if in q5 & read 0, then
  write 0, move L, goto q5
if in q5 & read x, then
  write x, move L, goto q5
if in q5 & read _, then
  write _, move R, goto q2
```



15



GREAT. A WAREHOUSE FILLED WITH MILES AND MILES OF REWRITABLE TAPE! WHAT ARE WE EVER GOING TO DO WITH THIS, ALAN?
...ALAN?

And thus the Turing Machine was born.

16

Church/Turing Thesis

Turing Machines provide a simple model of computation that is easier to reason with than code

- Turing's proof of the undecidability of the Halting Problem used TMs
- recall that the proof relied on self reference: a program that can take another program (including itself) as input
- similarly, a TM can take the program of any other TM as input and emulate that machine

Church/Turing Thesis: a TM is comparable to any other model of computation

→ proofs about the limitations of TMs apply to computation in general

17

Problem types: decision & optimization

the Halting Problem is an example of a *decision problem*

- solving the problem requires answering a yes/no question

another common type of problems is an *optimization problem*

- solving the problem requires finding the best/largest/shortest answer
- e.g., shortest path, minimal spanning tree

many problems have decision and optimization versions

- find the shortest path between vertices v_1 and v_2 in a graph
- is there a path between v_1 and v_2 whose length is $\leq d$

decision problems are more convenient for formal investigation of their complexity

18

Class P

P: the class of decision problems that are solvable in polynomial time O(*p*(*n*))

- i.e., the class of tractable decision problems

O(log N)	binary search
	add/remove/search a BST/AVL/red-black tree
	add/remove/min a heap
O(N)	sequential search
	preorder/inorder/postorder of a binary tree
	quick select (find kth largest value)
	Boyer-Moore algorithm (search for pattern in string of N chars)
O(N log N)	merge sort, quick sort, heap sort
	Huffman coding (text compression)
	closest pair on a plane
O(N^2)	insertion sort, selection sort
	Prim's algorithm (minimal spanning tree)
	Dijkstra's algorithm (shortest path)
O(N^3)	Floyd's algorithm (all-pairs shortest path)

P or not?

interestingly, there are many important problems for which no polynomial-time algorithm has been devised

- *Hamiltonian Circuit Problem:* determine whether a graph has a path that starts and ends at the same vertex and passes through every other vertex once
- *Traveling Salesman Problem:* find the shortest Hamiltonian circuit in a complete graph)
- *Graph Coloring Problem:* Determine the smallest number of colors needed so that adjacent vertices are different colors
- *Partition Problem:* Given N positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- *Knapsack Problem:* Given a set of N items with integer weights and values, determine the most valuable subset that fits in a knapsack with limited capacity.
- *Bin-packing Problem:* Given N items with varying sizes, determine the smallest number of uniform-capacity bins required to contain them.

Class NP

however, many of these problems fit into a (potentially) broader class

a *nondeterministic polynomial algorithm* is a two-stage procedure that:

1. generates a random string purported to solve the problem (guessing stage)
2. checks whether this solution is correct in polynomial time (verification stage)

NP: class of decision problems that can be solved by a nondeterministic polynomial algorithm

i.e., whose proposed solutions can be verified in polynomial time

example: Hamiltonian Circuit Problem is in NP

given a path of length N, can verify that it is a Hamiltonian circuit in O(N)

example: Partition Problem is in NP

given two partitions of size N, can verify that their sums are equal in O(N)

21

P vs. NP

decision versions of Traveling Salesman, Knapsack, Graph Coloring, and many other optimization problems are also in NP

note that problems in *P* can also be solved using the 2-stage procedure

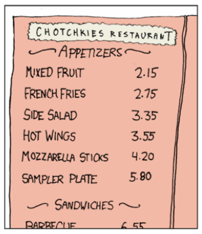
- the guessing stage is unnecessary
- the verification stage generates and verifies in polynomial time

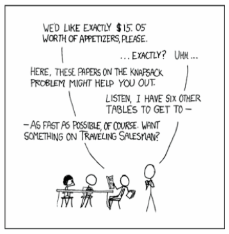
so, $P \subseteq NP$

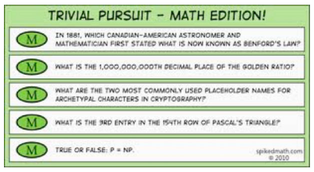
big question: does $P = NP$?

- considerable effort has gone into trying to find polynomial-time solutions to NP problems (without success)
- most researchers believe they are not equal (i.e., P is a proper subset), but we don't know for sure

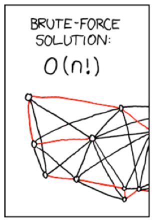
22

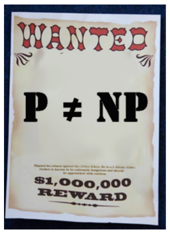



WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

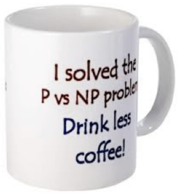

TRIVIAL PURSUIT – MATH EDITION!
IN 1881, WHICH CANADIAN-AMERICAN ASTRONOMER AND MATHEMATICIAN FIRST STATED WHAT IS NOW KNOWN AS BENFORD'S LAW?
WHAT IS THE 1,000,000,000TH DECIMAL PLACE OF THE GOLDEN RATIO?
WHAT ARE THE TWO MOST COMMONLY USED PLACEHOLDER NAMES FOR ARCHETYPAL CHARACTERS IN CRYPTOGRAPHY?
WHAT IS THE 3RD ENTRY IN THE 154TH ROW OF PASCAL'S TRIANGLE?
TRUE OR FALSE: P = NP.
spikedmath.com © 2010

The Digits must be single
Sudoku is for kids! They're silly and pointless.
Young man, you have no idea!
Uncle, are you alright?
I...I've...done...it!!!
Retired Professor cracks Million Dollar Puzzle
A Shortcut for Sudoku: 'P vs NP' problem solved
Revolutionary breakthrough: THE DAWN OF A NEW COMPUTING ERA?
Millennium Prize winner charms research community
Digital Security Experts say All Encryption Protocols to be overhauled
Well, it all began when I read somewhere that Sudoku helps keep Alzheimer's at bay...
Can I dispose of all those old newspapers at least now?
Don't worry. The Institute is sending over someone to collect those.
by Arun Ramkumar





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



23

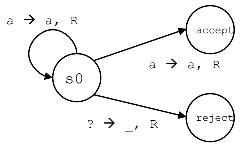
P, NP & Turing Machines

P and NP can alternatively be formulated in terms of Turing Machines

- P is the class of decision problems that can be solved in polynomial time by a *Turing Machine (TM)*

- NP is the class of decision problems that can be solved in polynomial time by a *Non-deterministic Turing Machine (NTM)*

  - a NTM allows for having multiple rules that can be applied in a state
    e.g., if state = 4 and cell = 'a', can 1) move right and set state = 5 OR
    2) move left and set state = 6
  - a NTM answers 'yes' if ANY sequence of transitions leads to the answer 'yes'
    in other words, it is greedy - can always guess the right one (equivalently, can think of it as trying all choices in parallel)



```
if in s0 and read a, then
  EITHER write a, move R, goto s0
  OR     write a, move R, goto accept

if in s0 and read anything else, then
  write _, move R, goto reject
```

24

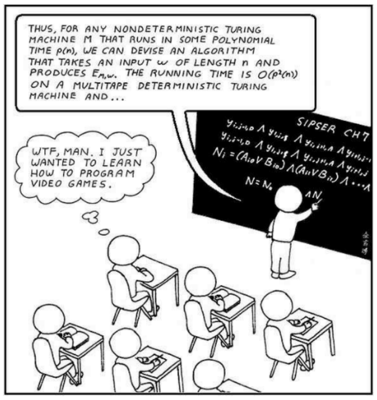
P, NP & Turing Machines

it has been shown that NTM's and TM's are equivalent in power

- any problem that can be solved by a NTM can be solved by a TM
- does this mean that P = NP?

NO!

not if the reduction can't be done in polynomial time


THUS, FOR ANY NONDETERMINISTIC TURING MACHINE M THAT RUNS IN SOME POLYNOMIAL TIME p(n), WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT w OF LENGTH n AND PRODUCES E_{M,w}. THE RUNNING TIME IS O(p²(n)) ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...
WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
SIPSER CH7

Computer science isnt what I expected

25

NP-complete

while we don't know whether P = NP, we can identify extremes within NP

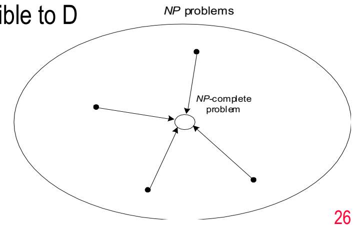
given decision problems D_1 and D_2, we say that D_1 is *polynomial-time reducible* to D_2 if there exists transformation t such that:

1. t maps all yes-instances of D_1 to yes-instances of D_2
   maps all no-instances of D_1 to no-instances of D_2
2. t is computable by a polynomial time algorithm

we say that decision problem D is *NP-complete* if:

1. D belongs to NP
2. every problem in NP is polynomial-time reducible to D

in short, an NP-complete problem is as hard as any problem in NP



26

NP-complete example

the first problem proven to be NP-complete was Boolean Satisfiability (SAT)

- given a Boolean expression, determine if it is *satisfiable*
  e.g., (A ∨ B) ∧ (~B ∨ C)   is true if A & C are true

- SAT is clearly in NP
  given true/false assignments to the propositions, can evaluate the truth of the expression in polynomial time

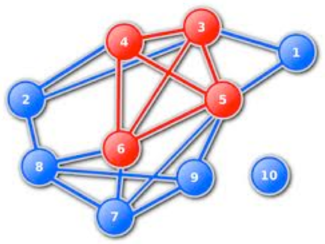
- to be NP-complete, every other NP problem must be reducible to it
  proof idea (Cook, 1971):
  - if a problem is in NP, can construct a non-deterministic (Turing) machine to solve it
  - for each input to that machine, can construct a Boolean expression that evaluates to true if the machine halts and answers "yes" on input
  - thus, original problem is reduced to determining whether the corresponding Boolean expression is satisfiable

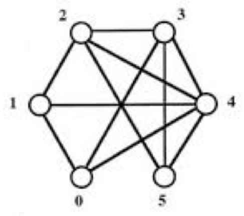
27

NP-complete reductions

if we can reduce SAT to another NP problem, then it is also NP-complete

CLIQUE: given a graph with N vertices, is there a fully connected subgraph of C vertices?





28

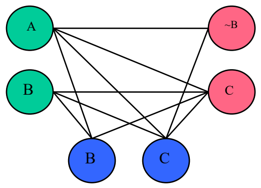
SAT → CLIQUE

can reduce the SAT problem to CLIQUE

- given an instance of SAT, e.g., (A ∨ B) ∧ (~B ∨ C) ∧ (B ∨ C)
  note: any Boolean expression can be transformed into conjunctive normal form
  here, there are 3 OR-groups, joined together with AND

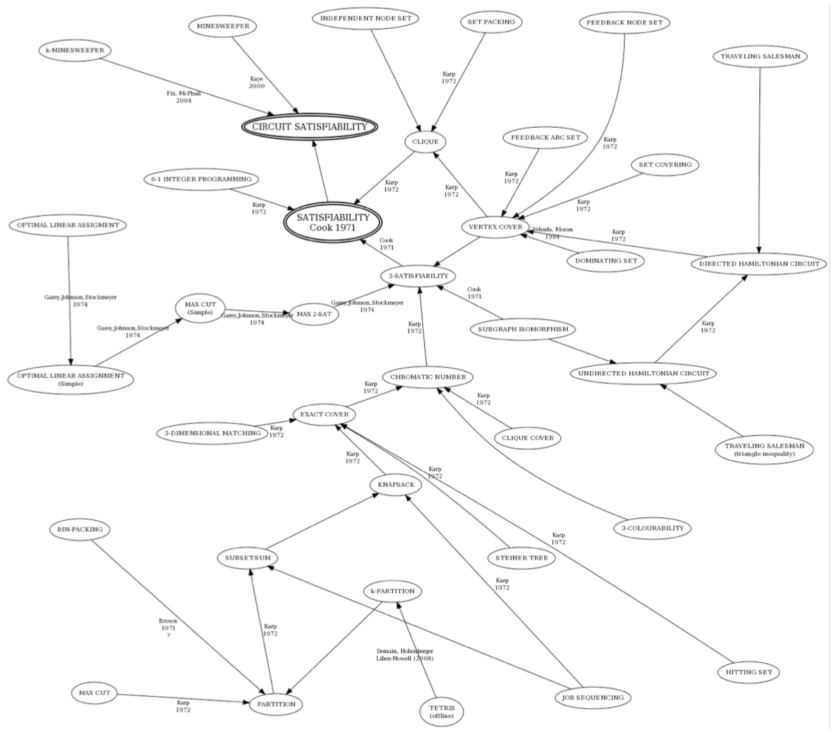
- construct a graph with vertices grouped by each OR-group
  there is an edge between two vertices if
  1. the vertices are in different OR-groups, and
  2. they are not negations of each other

  note: edge implies endpoints can be simultaneously true



- the expression is satisfiable if can have vertex from each OR-group simultaneously true
  in other words, is a clique of size C (where C is the number of OR-groups)
  since example expression has 3 OR-groups, need a clique of size 3

so, CLIQUE is also NP-complete



Implications of NP-completeness

an NP-complete problem is as hard as any problem in NP

- i.e., all problems in NP reduce to it
- discovering a polynomial solution to any NP-complete problem
  - ➔ would imply a polynomial solution to all problems in NP
  - ➔ would show P = NP

if P = NP, many problems currently thought to be intractable would be tractable

- e.g., PRIME FACTORIZATION PROBLEM: factor a number into its prime factors
- the RSA encryption algorithm relies on the fact that factoring large numbers is intractable
- if an efficient factorization algorithm were discovered, modern encryption *could* break

QUESTION: would it necessarily break?

Approximation algorithms

what if you need to solve a problem that is intractable?

- in the case of optimization problems, approximation algorithms can bus used
- i.e., will find a solution, but it may not be optimal

Knapsack Problem (KP):

- given N items (each with a value and weight) and a knapsack with weight capacity
- want to find the subset of items that has optimal value without exceeding the knapsack weight capacity

example: given the following items and a knapsack with capacity 50 lbs.

tiara	$5000	3 lbs
coin collection	$2200	5 lbs
HDTV	$2100	40 lbs
laptop	$2000	8 lbs
silverware	$1200	10 lbs
stereo	$800	25 lbs
PDA	$600	1 lb
clock	$300	4 lbs

Greedy approximations

tiara	$5000	3 lbs	1666
coin collection	$2200	5 lbs	440
HDTV	$2100	40 lbs	52.5
laptop	$2000	8 lbs	250
silverware	$1200	10 lbs	120
stereo	$800	25 lbs	32
PDA	$600	1 lb	600
clock	$300	4 lbs	75

simple greedy approach:

- repeatedly take the highest value item that fits, until capacity is reached
  tiara + coin collection + HDTV + PDA = $8,900 & 49 lbs.

smarter greedy approach:

- calculate the value/weight ratio, make greedy choices based on that
  tiara + PDA + coin collection + laptop + silverware + clock = $11,300 & 31 lbs.

- this happens to be the optimal choice, but this approach won't always work
  suppose there was also a printer worth $1400 and weighing 20 lbs.
  it value/weight ratio is 70, which is less than the clock, but
  tiara + PDA + coin collection + laptop + silverware + printer= $12,400 & 47 lbs.

it can be proven that the max from these two approaches > optimal/2

33

Genetic algorithms

many NP-hard problems have approximation algorithms

- requires problem-specific techniques & analysis

generate & test is not tractable due to the size of the solution space

- a *genetic algorithm (GA)* uses an evolutionary metaphor to perform a massive number of greedy searches in parallel

- GA were invented by psychologist/computer scientist John Holland in 1975

- many real-world resource scheduling problems turn out to be NP-hard
  e.g., Smart Airport Operations Center by Ascent Technology
  uses GA for logistics: assign gates, direct baggage, direct service crews, …
  considers diverse factors such as maintenance schedules, crew qualifications, shift changes, …

  too many variables to juggle using a traditional algorithm (NP-hard)
  GA is able to evolve sub-optimal schedules, improve performance

  Ascent claims 30% increase in productivity (including SFO, Logan, Heathrow, …)

34

Genetic Algorithm (GA)

for a given problem, must define:

chromosome:	bit string that represents a potential solution
fitness function:	a measure of how good/fit a particular chromosome is
reproduction scheme:	combining two parent chromosomes to yield offspring
mutation rate:	likelihood of a random mutation in the chromosome
replacement scheme:	replacing old (unfit) members with new offspring
termination condition:	when is a solution good enough?

in general, the genetic algorithm:

```
start with an initial (usually random) population of
   chromosomes while the termination condition is not met
    1. evaluate the fitness of each member of the population
    2. select members of the population that are most fit
    3. produce the offspring of these members via reproduction &
       mutation
    4. replace the least fit member of the population with these
       offspring
```

35

GA example (cont.)

tiara	$5000	3 lbs
coin collection	$2200	5 lbs
HDTV	$2100	40 lbs
laptop	$2000	8 lbs
silverware	$1200	10 lbs
stereo	$800	25 lbs
PDA	$600	1 lb
clock	$300	4 lbs

chromosome: a string of 8 bits with each bit corresponding to an item

- 1 implies that the corresponding item is included;   0 implies not included
  - e.g.,   11100000 represents (tiara + coins + HDTV)
  - 01101000 represents (coins + HDTV + silverware)

fitness function: favor collections with higher values

- fit(chromosome) = sum of dollar amounts of items, or 0 if weight > 50
  - e.g.,   fit(11100000) = 9300
  - fit(01101000) = 0

reproduction scheme: utilize crossover (a common technique in GA's)

- pick a random index, and swap left & right sides from parents
  - e.g.,   parents 11100000 and 01101000, pick index 4
  - 1110|0000 and 0110|1000 yield offspring 11101000 and 01100000

36

GA example (cont.)

tiara	$5000	3 lbs
coin collection	$2200	5 lbs
HDTV	$2100	40 lbs
laptop	$2000	8 lbs
silverware	$1200	10 lbs
stereo	$800	25 lbs
PDA	$600	1 lb
clock	$300	4 lbs

Generation 0 (randomly selected):

```
11100000  (fit = 9300)      01101000  (fit = 0)         11001011  (fit = 9300)
11010000  (fit = 9200)      00010100  (fit = 2800)      01001011  (fit = 4300)
11110111  (fit = 0)         10011000  (fit = 8200)
```

choose fittest 4, perform crossover with possibility of mutation

```
111000|00 + 110010|11  →    11100011      11001001
110|10000 + 100|11000  →    11011000      10010000
```

Generation 1 (replacing least fit from Generation 0):

```
11100000  (fit = 9300)      11100011  (fit = 0)         11001011  (fit = 9300)
11010000  (fit = 9200)      11001001  (fit = 8700)      11011000  (fit = 10400)
10010000  (fit = 7000)      10011000  (fit = 8200)
```

choose fittest 4, perform crossover with possibility of mutation

```
1101|1000 + 1100|1011  →    11011011      11001000
1110000|0 + 1101000|0  →    11100000      11010000
```

Generation 2 (replacing least fit from Generation 1):

```
11100000  (fit = 9300)      11001000  (fit = 8400)      11001011  (fit = 9300)
11010000  (fit = 9200)      11100000  (fit = 9300)      11011000  (fit = 10400)
11011011  (fit = 11300)     11010000  (fit = 9200)
```

37

In the news…

A New Approach to Multiplication Opens the Door to Better Quantum Computers

— **Kevin Hartnett, Quanta Magazine, April 24, 2019**

38