COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 24: P, NP and More

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW8 by EoD!

Due 12/01 @ 11:59PM ET

Final Exam

Tuesday 12/06 2:00pm-4:00pm

Final Exam

11/29 and 12/01 Review Lectures

Extra Credit Last Opp.!

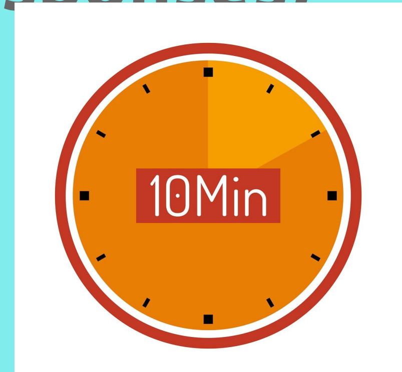
Wed. 11/30 T.I. with me for +0.5%

Final Survey

+1% for >= 80% completion (34 responses)

Final Survey

+1% for >= 80% completion (34 responses)



Recall where we ended last lecture…

Exhaustive Search & Backtracking

- Sometimes, the only way to solve a certain problem is through brute force, i.e. trying out every possible combination of values in order to get the correct answer. This process is called **exhaustive search**.
- We can reduce the cost in practice sometimes with **backtracking**, i.e. stopping early when we see we've hit a dead end while building our answer.

1. **Choose**: What are we choosing at each step? What are we stepping over?
2. **Explore:** How will we modify the arguments before recursing?
3. **Unchoose:** How do we un-modify the arguments (if needed)?
4. **Base case:** What should we do when finished? How to know when finished?

Combinations vs. Permutations

- What if the order of our selection or results do not matter (such that we are dealing with **combinations** instead of **permutations**).
  - Combination Example: all the possible teams of 2 you can form from 10 people
  - Permutation Example: all the possible 7-digit phone numbers you can from digits

- For example with diceSum, what if we now want to treat {1, 3} and {3, 1} as the same roll?

Example #1.5: Dice Sum Combination Implementation

```
void diceSumHelperCombination(int diceLeft, int desiredSum, int currentSum,
                              int choiceIdx, std::vector<int> &currentRolls) {
  // Base case
  if (currentSum == desiredSum && diceLeft == 0) {
    printAnswer(currentRolls);
    return;
  } else if (diceLeft == 0 || currentSum >= desiredSum) {
    return;
  } else if (currentSum + diceLeft * 1 > desiredSum || currentSum + diceLeft * 6 < desiredSum) {
    return;
  }
  // recursive case
  for (int i = choiceIdx; i < 7; i++) {
    currentRolls.push_back(i); // choose
    diceSumHelperCombination(diceLeft - 1, desiredSum, currentSum + i, i, currentRolls); // explore
    currentRolls.pop_back();                                   // unchoose
  }
}
```

Example #2: Subsets

Given an vector<int> nums of unique elements, return all possible subsets (the power set).

Input: vector<int> nums of unique integer values

Output: all possible subsets

Example: nums = {1,2,3} should output {{},{1},{1,2},{1,2,3},{1,3},{2},{2,3},{3}}

1. Choose: What are we choosing at each step? What are we stepping over?
2. Explore: How will we modify the arguments before recursing?
3. Unchoose: How do we un-modify the arguments (if needed)?
4. Base case: What should we do when finished? How to know when finished?

Example #2: Subsets Implementation

```
void findAllSubsetsHelper(vector<int> nums, int choiceIdx, vector<int> currCombo) {
  if (choiceIdx == nums.size()) {
    printAnswer(currCombo);
    return;
  }
   // not choose item
  findAllSubsetsHelper(nums, choiceIdx + 1, currCombo);

  // choose item
  currCombo.push_back(nums[choiceIdx]);
  findAllSubsetsHelper(nums, choiceIdx + 1, currCombo);
  currCombo.pop_back();
}
```

Constraint Satisfaction Problems

- Problems that have requirements, and we need to search all possibilities then check whether they have the requirements.
- *Sudoku*

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

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

- *N-Queens*: given a NxN chess board, place N queens on the board without any of queens attacking each other (attack demo, backtrack demo)

Big Questions!

- What is P and EXP?
- What is NP?
- What is NP complete, NP hard and what are reductions?

Motivation

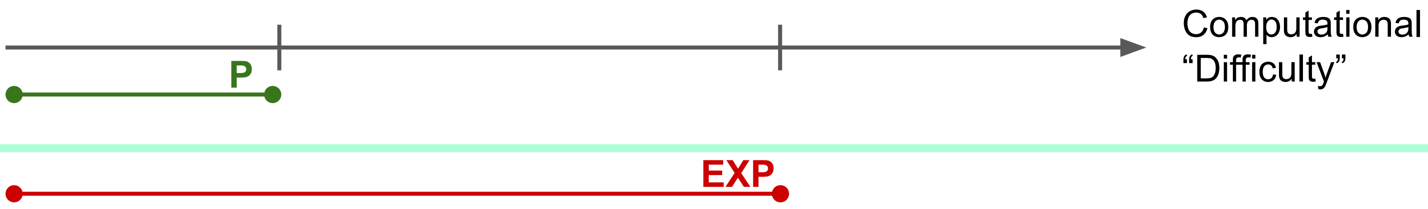
- P vs NP may be the most famous unsolved question in Computer Science $$$
- It gives us a way to reason about whether a problem is tractable or not.
  - Classifies problems based on how difficult they are to solve
  - If you're working on a new problem, don't waste your time trying to come up with a clever polynomial time solution if it's not possible!
- Similar to Big-Oh in that it's a theoretical framework, a tool for reasoning about algorithms, comparing algorithms, etc. P vs. NP is also a theoretical framework, a tool for reasoning about problems, comparing problems, etc.

Big Questions!

- What is P and EXP?
- What is NP?
- What is NP complete, NP hard and what are reductions?

Polynomial Time (P) versus Exponential Time (EXP)

- P: set of decision problems that can be solved in polynomial time
  - $O(n^k)$, e.g. $n \log(n)$, n^2, n^{20}
- EXP: set of decision problems that can be solved in exponential time
  - $O(2^n)$, $O(10^n)$, $O(2^{n^c})$
- We work with **decision problems** (i.e. the answer to these problems is yes or no) but the implications are still often applicable to optimization problems.
- $P \subseteq EXP$: "P is a subset of EXP"
- There are lots of problems in EXP, because it is very slow, and includes problems where the only solution we know is "try everything".



Examples

- **Problems in P**
  - Is this array sorted?
  - Is string X a substring of string Y?
  - Is this Binary Tree a Binary Search Tree?
  - Does this graph have negative weight cycles?
  - Is the height of this tree smaller than 100?
    - Note how related this is to “what is the height of the tree?”
  - Given a graph, can you find a path from s to t with at most cost c?
    - Note how related this is to “what is the cost of the shortest path from s to t?”
- **Problems in EXP**
  - Given an n x n chess board, can white force a win? In EXP, we are unsure if in P
  - “Traveling Salesman Problem” (discussed later). In EXP, we are unsure if in P
- **Problems in neither EXP nor P**
  - Halting problem: will this program ever stop running? Infinite time

www.kahoot.it, Code: 253 0995

Enter your @aggies.ncat email

Polls

True or False: A decision problem that can be solved in O(n log(n)) is in P

True

True or False: All problems in P can be considered in EXP

True

True or False: There are problems that are harder than problems in EXP

True

Big Questions!

- What is P and EXP?
- What is NP?
- What is NP complete, NP hard and what are reductions?

Non-Deterministic Polynomial (NP) Intuition

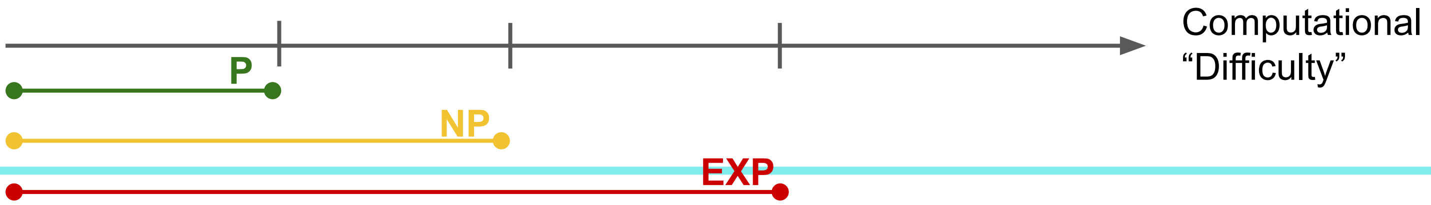
- **NOT** “Non-Polynomial”
- There are an interesting set of problems that are “hard” to solve (i.e. in EXP) but if you’re given the solution, it is “easy” to validate.
- Consider Sudoku: we need 1 - 9 in every row, column, and 3x3 grid. Note: here, n = 9, but the generalized Sudoku is for n x n.

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

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

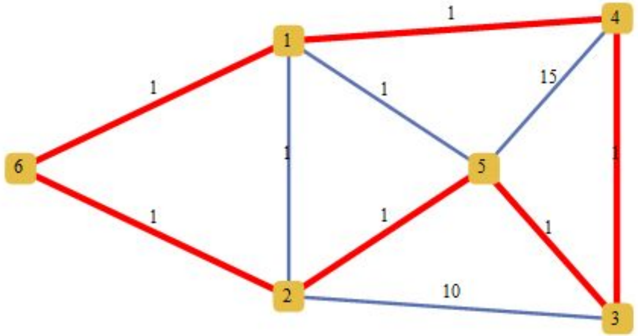
Nondeterministic Polynomial (NP)

- NP: set of all decision problems that can be **verified** in polynomial time.
- P is in NP, as you could just solve the problem in polynomial time and see if the answers are equal to verify.
- Examples
  - Is this array sorted?
  - Is string X a substring of string Y?
  - … all problems in P
  - Is there a subset of elements in this array that add up to k?
  - Given a graph, is there a path of at most length L that visits each node exactly once and returns where you started? (“Traveling Salesman Problem”)



Traveling Salesman Problem (TSP)

- Given a graph with cities as vertices and edges as roads with weights, find the best path that visits every city exactly once and winds up where we started (defined as a "tour").
- The decision problem version is "Is there a tour with cost of at most C?"
- The best algorithm we know how to find the best tour is currently exponential (trying all possible routes)
- But, if someone gave you the tour, and asked you if it was at most cost C, you could verify that very quickly. So, this is an example of a problem in NP.



“Nondeterministic” computing

- Again, note that NP is **nondeterministic** polynomial, NOT non-polynomial
- Why is it called that, i.e. what does that actually mean? Alternate Way to think about NP
- Another way to define NP: decision problems solvable by a **nondeterministic** computer in polynomial time.
    - A nondeterministic computer is a magical (unrealistic), extremely lucky computer, that guesses the best thing to do at each stage.
    - An nondeterministic computer could solve TSP: “go here, then here, then here, …” and magically, at the end, we have the best tour, and we can check if it is less than our cost C.
        - Assuming these guesses are O(1), the algorithm is O(n) where n is the number of edges. So, TSP is NP, as it can be solved in polynomial time by a nondeterministic computer.
- Note that the definition before, “set of all decision problems that can be **verified** in polynomial time”, still works. The new one is just a different way of defining the same set of problems (that also helps us understand the name NP).

Kahoot!

www.kahoot.it, Code: 253 0995

Enter your @aggies.ncat email

Polls

If a solution can be **verified** in **polynomial** time, the problem **is** in P

False

If a solution can be **verified** in **polynomial** time, the problem **could be** in P

True

If a solution can be **verified** in **polynomial** time, the problem **is** in NP

True

$P \subseteq NP$

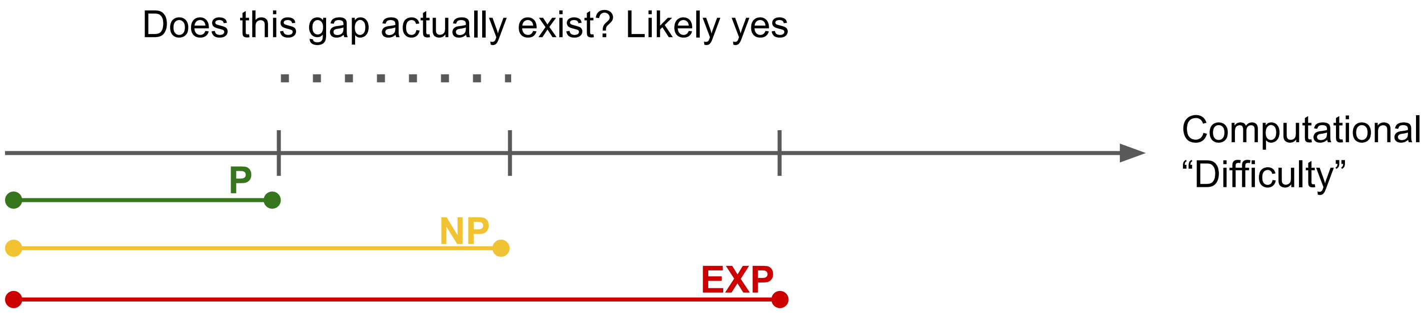
True

NP stands for “non-polynomial” as it represents hard problems like Sudoku

False

P vs NP

- We know that P ⊆ NP
- But, does P = NP? a $1 million-dollar question (actually)
  - Most likely P ≠ NP, it's just hasn't been proven yet.
- "Creating a nondeterministic computer is impossible"
- "Generating solutions can be harder than checking them"



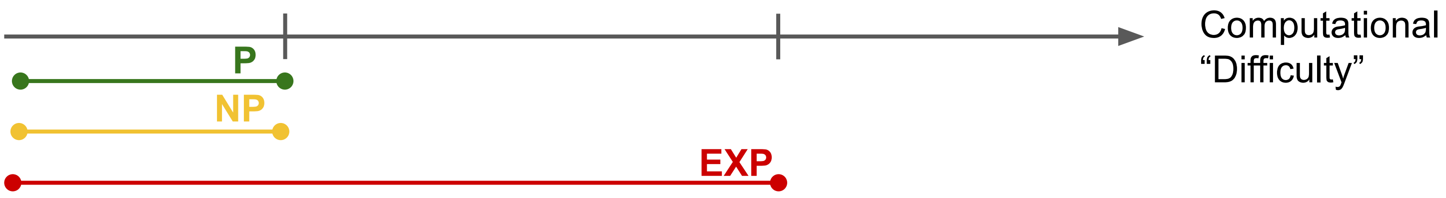
P vs. NP

- If a problem is in class P…
  - with n = 100 inputs
  - And its algorithm has runtime O(n^3)
  - … we can solve it’s problem in ~3 hours on some machine

- If a problem is in class NP…
  - with n = 100 inputs
  - And its algorithm has runtime O(2^n)
  - … we can solve it’s problem in ~300 quintillion years (300 * 10^18) on same machine

What if P = NP though?

- Some implications:
    - We could cure a lot more diseases with efficient protein folding simulations.
    - But all passwords / encryption could be cracked.
- Scott Aaronson's philosophical argument: *If P=NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss.*



Kahoot!

www.kahoot.it, Code: 253 0995

Enter your @aggies.ncat email

Polls

P is the set of all decision problems that can be ______ in ______ time.

solved, polynomial

EXP is the set of all decision problems that can be ______ in ______ time.

solved, exponential

NP is the set of all decision problems that can be ______ in ______ time.

verified, polynomial

NP is also the set of all decision problems that can be solved by a ______ model of computation in ______ time.

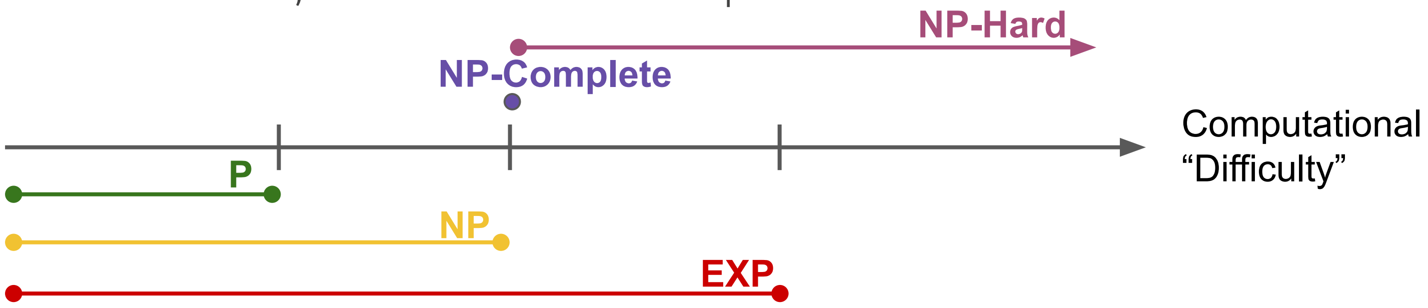
nondeterministic, polynomial

Big Questions!

- What is P and EXP?
- What is NP?
- What is NP complete, NP hard and what are reductions?

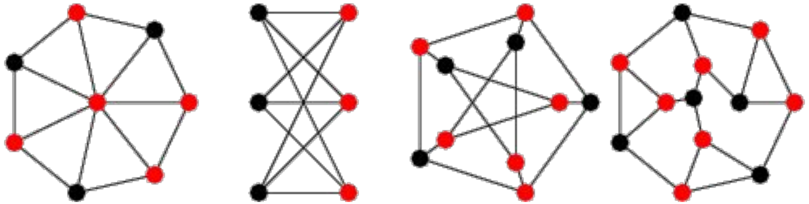
NP-Complete and NP-Hard

- **NP Hard**: problems at least as hard as the hardest problems in NP.
- **NP-Complete**: problems that are NP-hard, but still in NP, i.e. "the hardest problems in NP".
- Why do we care about NP-Complete? Because if we find a way to solve one NP-Complete problem, we will solve them all.
- Why do we care about NP-Hard? Because if we can show a problem is both NP-Hard and NP, we know it is NP-Complete.



NP-Complete Problems

- Traveling Salesman Problem
- Generalized Sudoku
- Vertex cover: “Given a graph G, can you find a vertex cover of n nodes?”



- Boolean satisfiability: “Given a boolean expression like the following *(a or !b) and (c or d) or e* are there possible values for a, b, c, d, e that will make the statement true?”

Reductions

- Reductions are converting a problem into another problem.
- We do this all the time to solve problems, e.g. with graphs, we would transform them to be able to use an algorithm we know (like network flow).
- To prove a problem X is NP-Complete, you can:
  - Show it is NP-Hard (usually by reducing a known NP-Complete problem to it)
  - Show it is NP (e.g. by showing its solution is verifiable in polynomial time)

Reduction Example

- Number Scrabble!
  - Imagine we are playing a game where the numbers are lined up 1 through 9 and we take turns selecting numbers. One of us wins when the numbers sum to 15.

1 2 3 4 5 6 7 8 9

Reduction Example

- Number Scrabble!
  - Imagine we are playing a game where the numbers are lined up 1 through 9 and we take turns selecting numbers. One of us wins when the numbers sum to 15.

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

- Can we rearrange?

Halting Problem

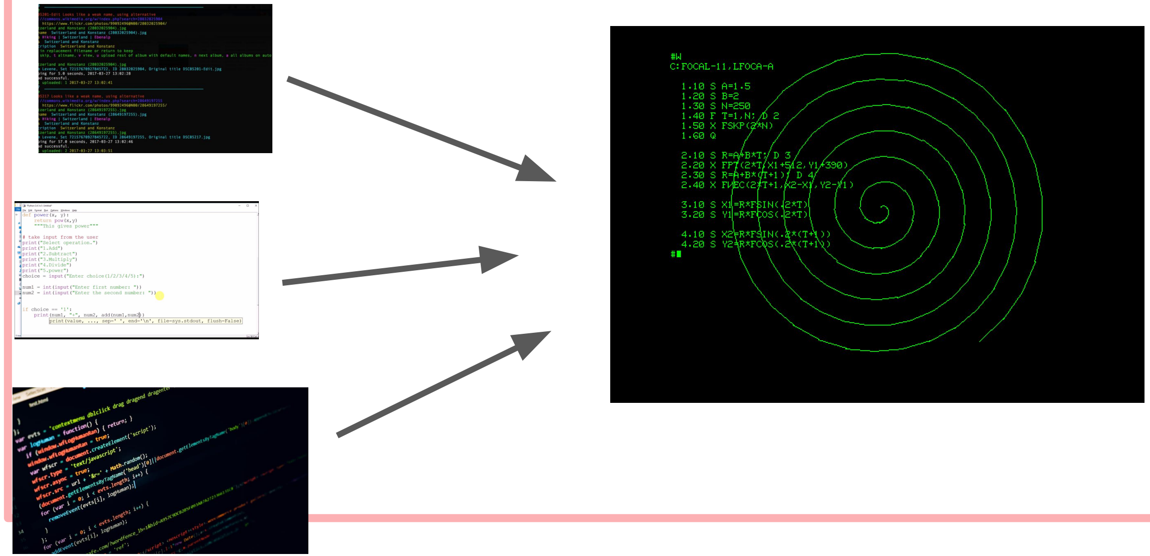
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

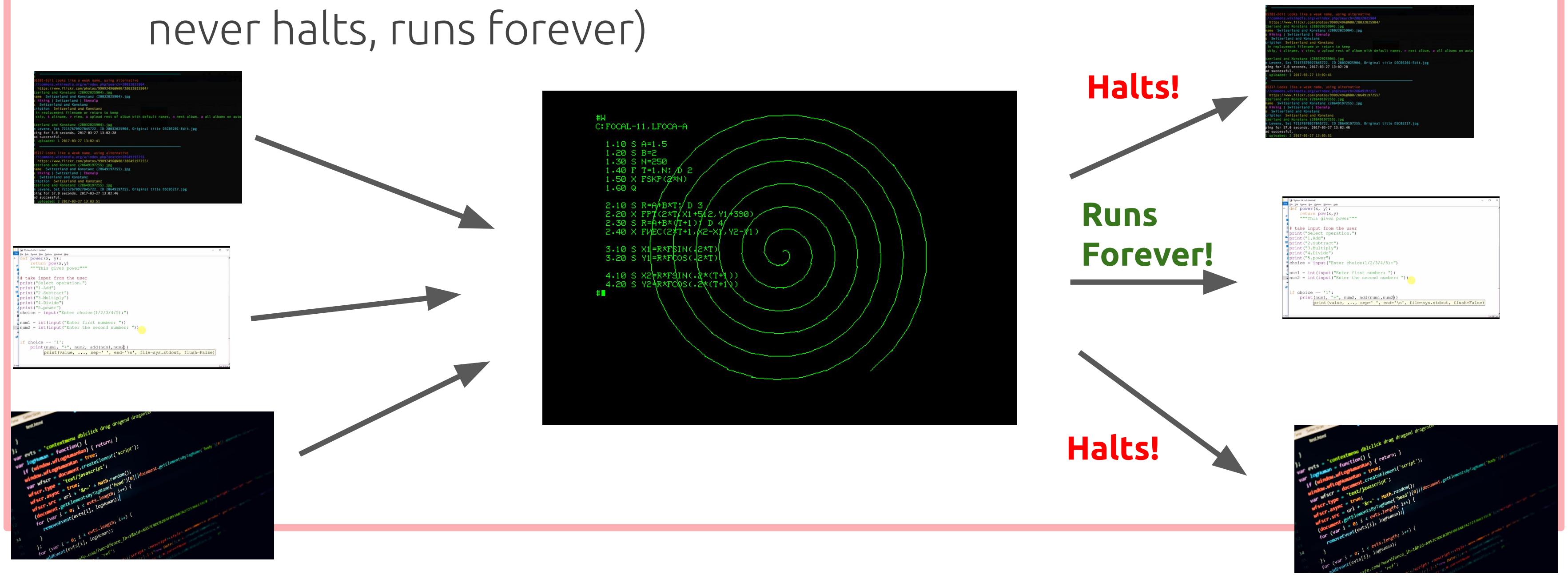
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



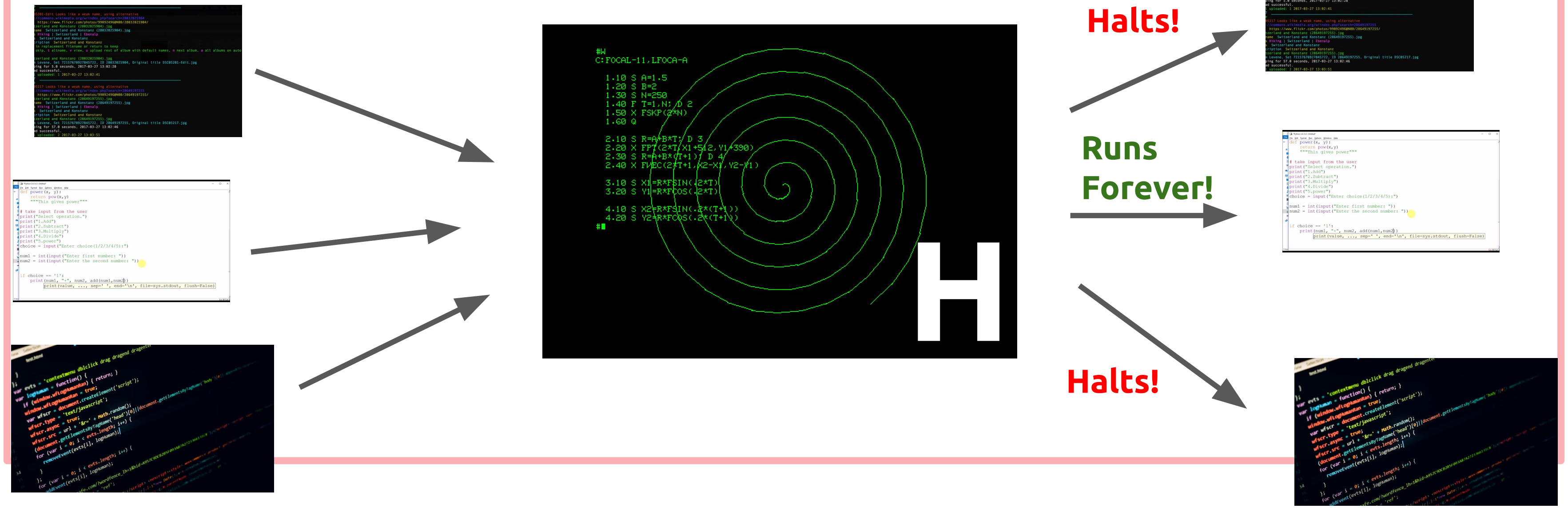
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



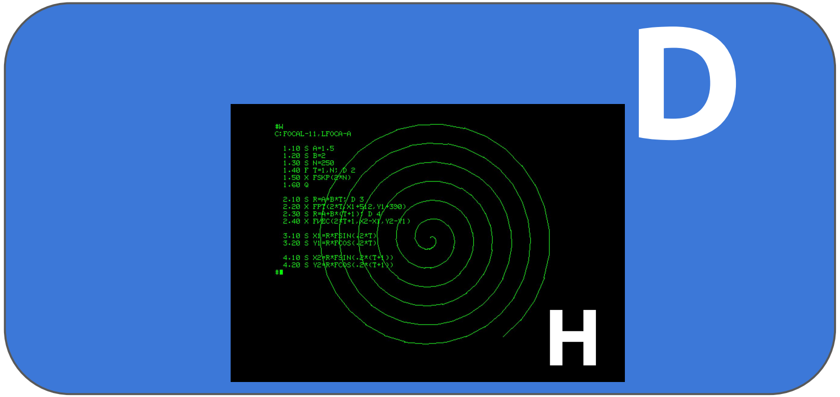
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



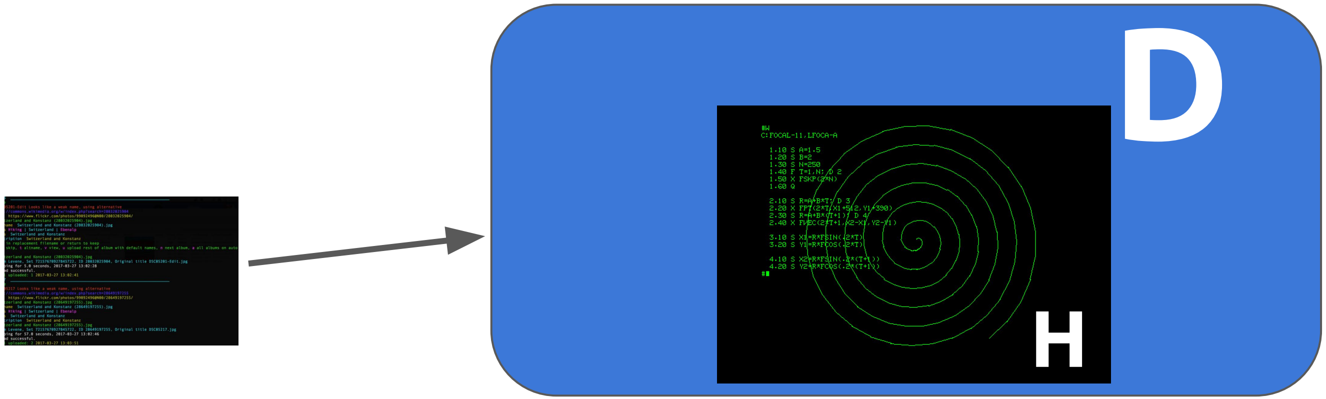
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



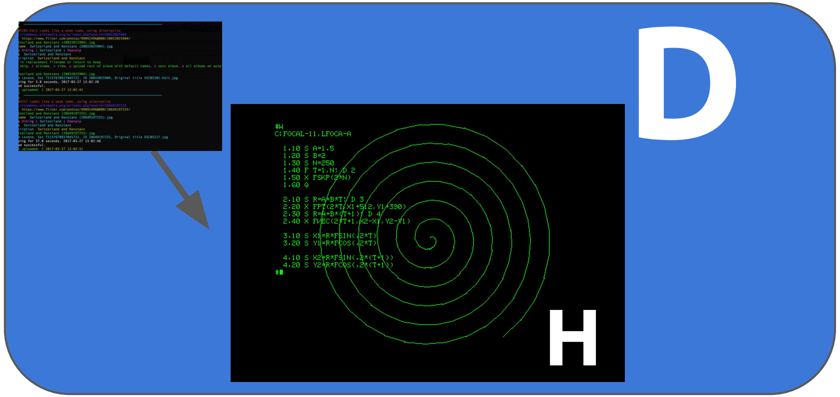
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

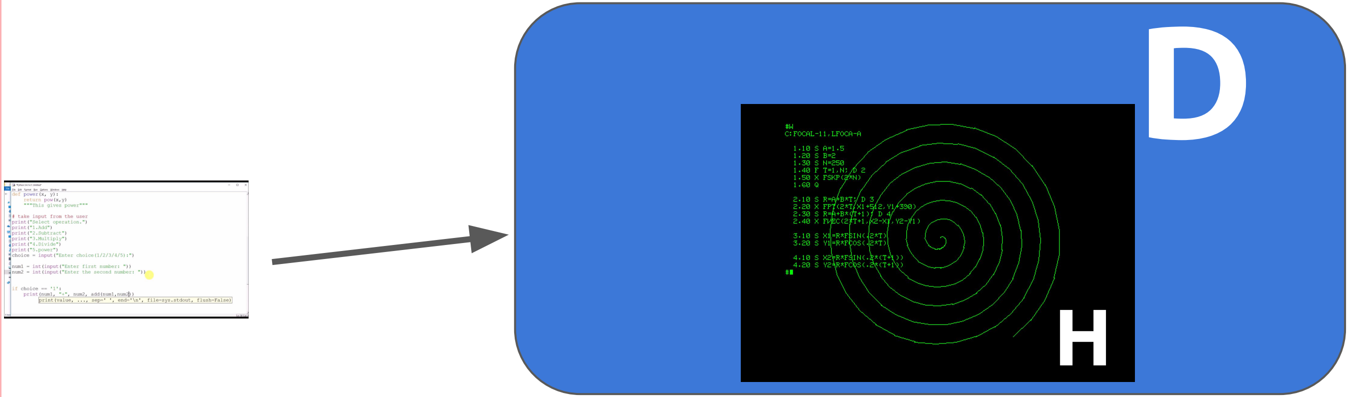
- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

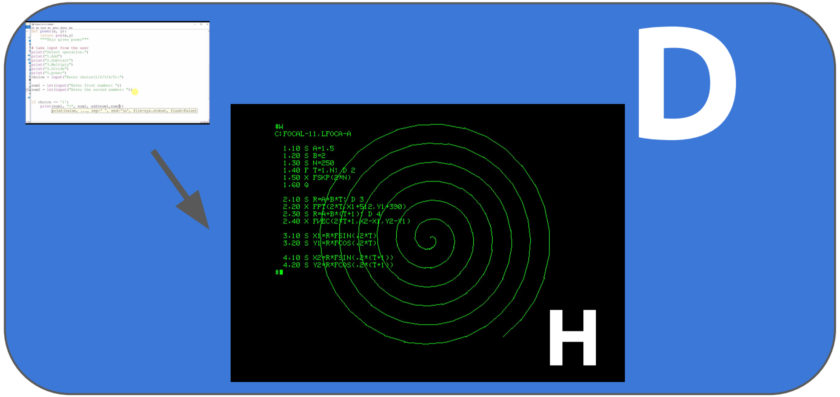
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



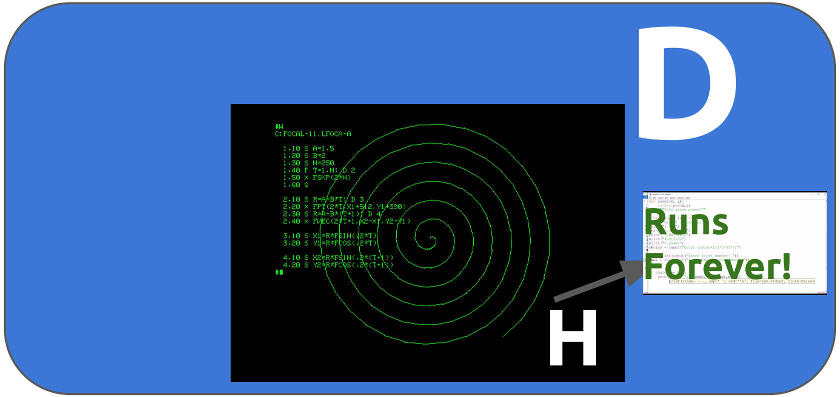
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



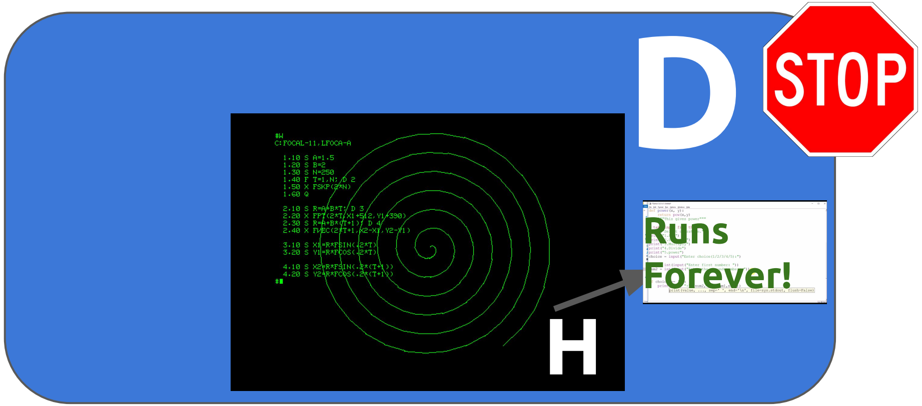
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



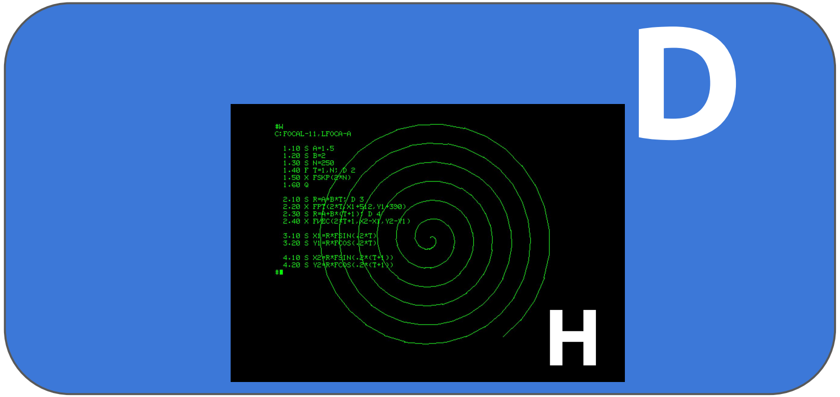
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



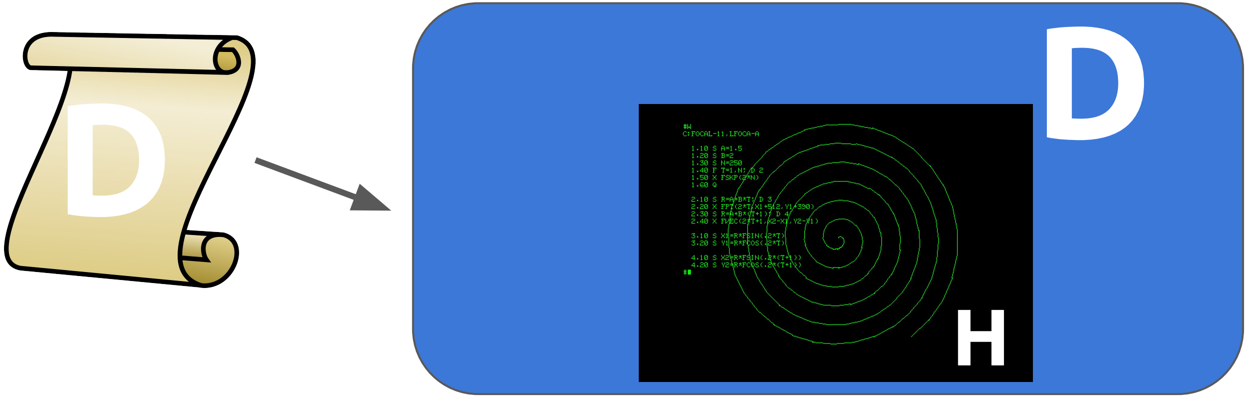
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



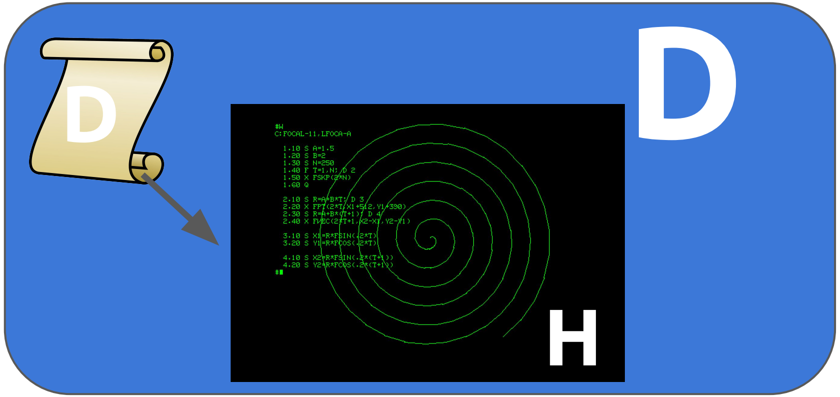
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



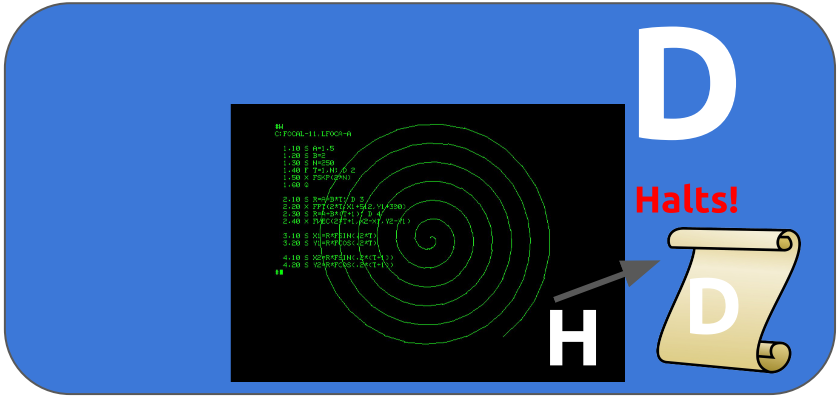
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

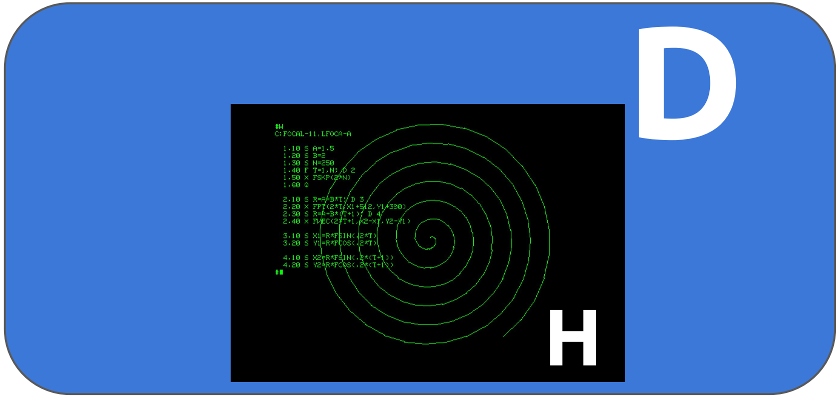


Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

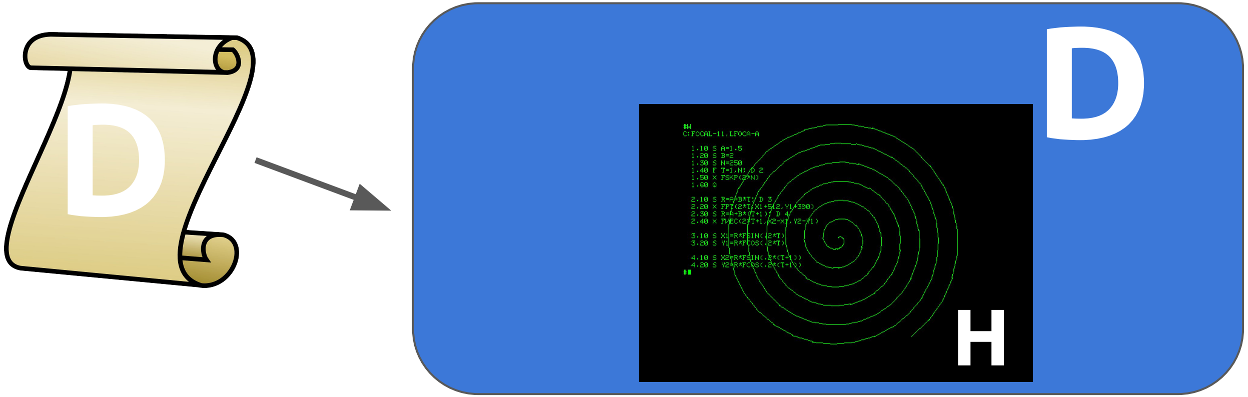
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



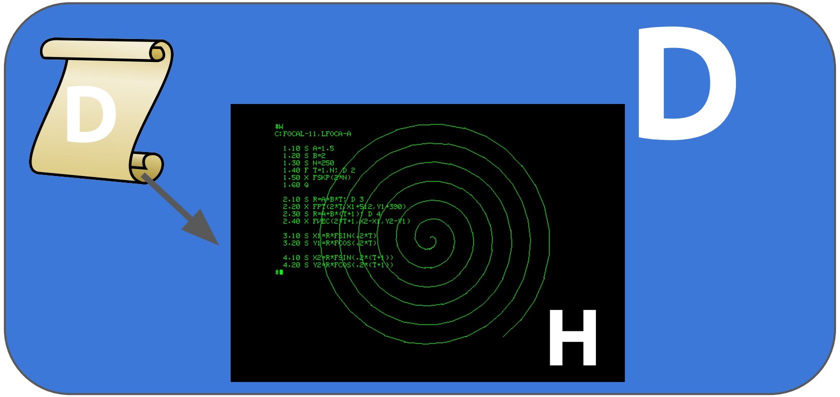
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



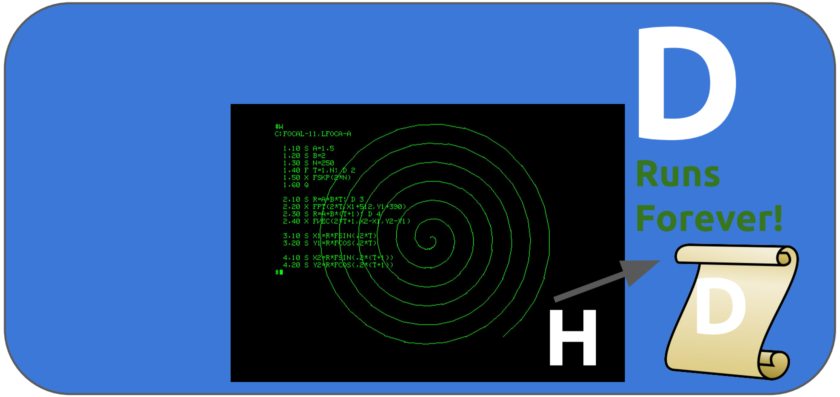
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



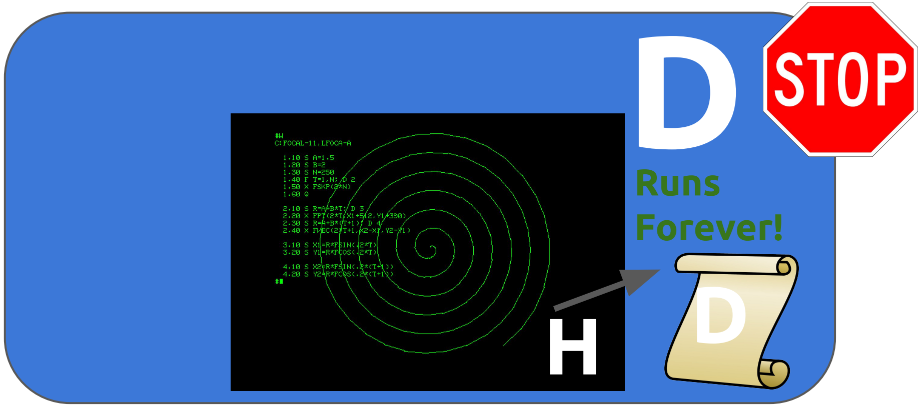
Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)



Halting Problem

- Is it possible to write a program that determines whether another program halts/terminates/stops running and exits? (A program that never halts, runs forever)

Takeaways

- We want to avoid slow algorithms, so knowing if a problem is not in P is useful.
- P: decision problems that have polynomial time algorithms
- NP: decision problems that can be verified in polynomial time
- $P \subseteq NP \subseteq EXP$
- NP-Complete problems are both NP-Hard and NP, and lots of interesting problems are NP-Complete. They can often be “reduced” to each other.
- P ?= NP asks whether the above two complexity classes are the same. It is likely not true, but has not been proven.

COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 24: P, NP and More

Lecturer: Chris Lucas (cflucas@ncat.edu)