Greedy Algorithms, Computational Complexity

Analysis of Algorithm

Kia Teymourian

05/11/2022

Agenda

1. Greedy Algorithms

2. Computational Complexity

Greedy Algorithms

- We solve optimization problems by going over a set of steps.
- For many of the optimization problems using DP is an overkill
- Simpler and more efficient algorithms can do the job as well.

Idea:

- A **Greedy Algorithm** always makes the choice that **looks best at the current step/state.**
- It hopes that a locally optimal choice will lead to a globally optimal solution.

Example - An Activity-Selection Problem

- We have a set of activities each with a specific start and end time.
- We have a classroom that the organizers of the activities want to use for the run the activities.
- Manager wants to give the classroom to maximum number of activities.
- Two activities a_i, a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$ (when they are after each other or before) and intervals do not overlap.
- **We want to select mutually compatible activities.**

Table: Set of Activities with their Start and Finish Time.

$Activity_i$	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{Start_i}$	1	3	0	5	3	5	6	8	8	2	12
$\boldsymbol{Finish_i}$	4	5	6	7	9	9	10	11	12	14	16

Example - An Activity-Selection Problem (Potential solutions)

Table: Set of Activities with their Start and Finish Time.

$Activity_i$	1	2	3	4	5	6	7	8	9	10	11
$Start_i$	1	3	0	5	3	5	6	8	8	2	12
$Finish_i$	4	5	6	7	9	9	10	11	12	14	16

For this example we can select:

- $\{a_3, a_9, a_{11}\}$ of mutually compatible activities, but the set is not maximized.
- We can also select $\{a_1, a_4, a_9, a_{11}\}$ and subset of activities are larger.
- Another subset is $\{a_2, a_4, a_9, a_{11}\}$

Dynamic Programing or Greedy Algorithm.

- We can think about a dynamic-programming solution.
- We have several choices when determining which subproblems to use in an optimal solution.
- We observe we need to consider only one choice **(named the greedy choice)**, and if we make that choice, only one subproblem remains.
- We can then think of developing a recursive greedy algorithm to solve the activity-scheduling problem.
- We can then convert the recursive algorithm to an iterative one which is better to understand.

Dynamic Programming Subset

- $\boldsymbol{S_{ij}}$ is the set of activities that start after activity a_i finishes and that finish before activity a_j starts.
- We want to find a maximum set of mutually compatible activities in S_{ij} and maximize the size of the set A_{ij} which includes activity a_k.
- **We include $\boldsymbol{a_k}$ in the optimal solution.** When we do so, the remaining is two sub-problems.
  - 1. Find all mutually compatible activities S_{ik}. All activities that start after activity a_i finishes and that finish before activity a_k starts.
  - 2. Find all mutually compatible activities S_{kj}. All activities that start after activity a_k finishes and that finish before activity a_j starts.

  $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
- The optimal solution contains a_k

Dynamic Programming

- $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$
- The optimal solution contains a_k
  $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
- We need to maximize the A_{ij}
  $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$
- Let us denote the size of an optimal solution S_ij by $c[i, j]$, we can have the recurrence:
  $c[i, j] = c[i, k] + c[k, j] + 1$

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ MAX_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing \end{cases}$$

We can then use dynamic programming to develop a recursive algorithm and memoize it, or we could work bottom-up and fill in table entries as we go along.

Making the greedy choice

- Think if we can have a greedy choice here?
- It can help to not consider all the choices inherent in recurrence.
- The activity-selection problem, can have a greedy choice.

Trust your Intuition!

- **Choose an activity that leaves the resource available for as many other activities as possible.**
- So, **we would select an activity with the earliest finish time**, since to have time/resources available for as many of the activities that follow it.

In our example, select a_1 because it would leave us as much possible time for other activities.

Greedy choice

After this greedy choice we have only one sub-problem to solve.

- After we select a_1, we do not need to consider activities that finish before a_1 starts. Why?
- Let define $S_k = \{a_i \in S : s_i \geq f_k\}$
- If a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S

Note: Choosing the a_1 is not the only greedy choice for this problem.

Greedy choice

After this greedy choice we have only one sub-problem to solve.

- After we select a_1, we do not need to consider activities that finish before a_1 starts. Why?
- Let define $S_k = \{a_i \in S : s_i \geq f_k\}$
- If a_1 is in the optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S

Note: Choosing the a_1 is not the only greedy choice for this problem.

One important Big Question that remains is **what if our intuition is correct.**

Greedy Algorithm

- Our greedy algorithm does not need to work bottom-up, like a table-based dynamic-programming algorithm.
- It can be top-down, meaning to select an activity to put into the optimal solution and then solving the subproblem.
- Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem.
- Remember bottom-up technique means solving subproblems before making a choice.

Algorithm 1 RECURSIVE-ACTIVITY-SELECTOR(s, f ,k , n)

```
1: m = k + 1
2: while m ≤ n and s[m] < f[k] do   ▷ Try to find the first activity in S_k to finish
3:     m = m + 1
4: end while
5: if m ≤ n then                    ▷ then call it recursively
6:     return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m , n)
7: else
8:     return ∅
9: end if
```

- s is set of activities, f is the set of finish times, k is the initialization, n is the number of activities.

How to start the algorithm ?

- Fictitious activity a_0 with $f_0 = O$ so that then **the subproblem S_0 is the entire problem set.**
- The initial call of the above algorithm would be then RECURSIVE-ACTIVITY-SELECTOR(s, f, O, n).

Algorithm 2 GREEDY-ACTIVITY-SELECTOR(s, f)

1: $n = s.length$
2: $A = \{a_1\}$ ▷ Our Greedy Choice - Include it in the solution
3: $k = 1$
4: ▷ Activities are sorted in order of monotonically increasing of their finish time.
5: **for** $m = 2$ to n **do** ▷ finds the earliest activity in S_k to finish.
6: **if** $s[m] \geq f[k]$ **then** ▷ if the start of m is bigger than finish of k.
7: ▷ Remember f_k is always maximize the finish time of any activity in A.
8: $A = A \cup \{a_m\}$ ▷ Include this in the solution
9: $k = m$ ▷ Swap m is the new k
10: **end if**
11: **end for**
12: **return** A

- ▷ The for-loop finds the earliest activity in S_k to finish.
- ▷ The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities.
- ▷ Such an activity is the earliest in S_k to finish.

The run time of the above greedy solution is then $\Theta(n)$

Other Problems that have a Greedy Algorithm Solution

Other examples of Problems that can be solved by Greedy Algorithms

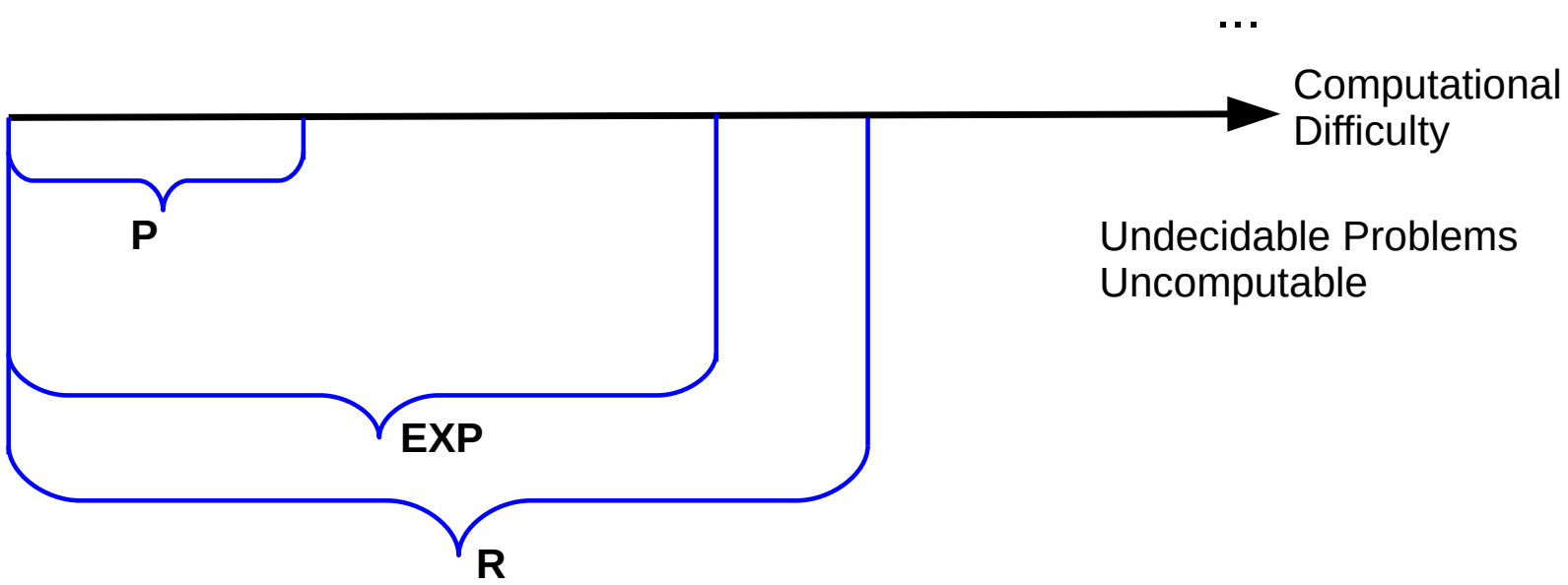
- **Fractional Knapsack Problem.**
- **Money Changing Problem.** Determine minimum number of money coins to give while making a change.
  For example returning 37 sent, in 1 coin of 20 cent, 1 coin of 10 cent, 1 coin of 5 cent and 2 one cent coins.)

Computational Complexity

Computational Complexity

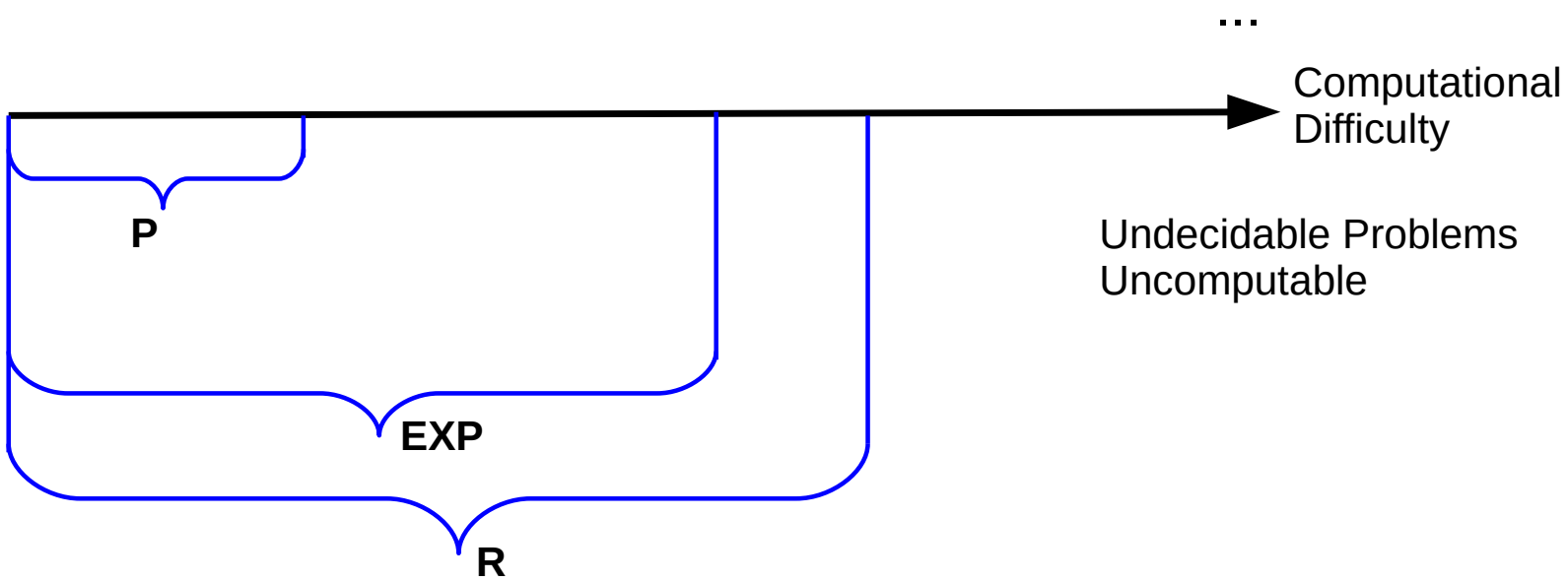
We can categorize the computation of complexity of all problems that we face in following categories. Most problems that we may face are uncomputable.

Computation Complexity - P, EXP , R



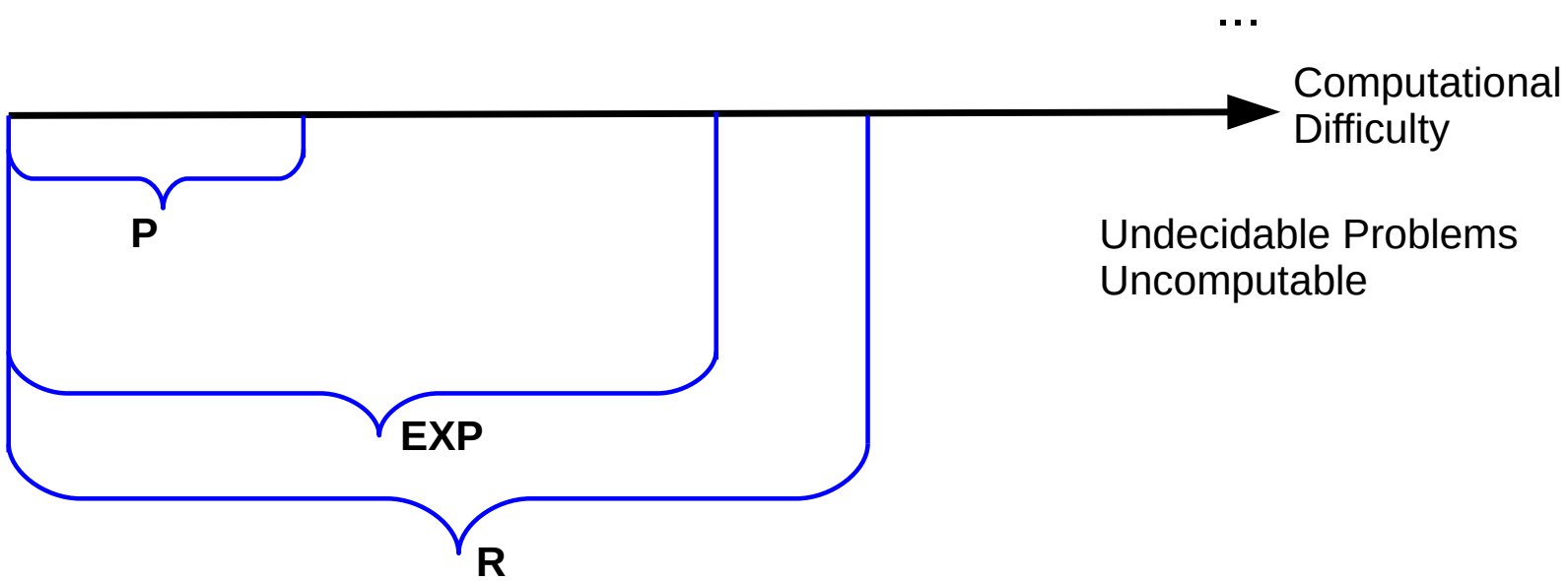
- **P - Polynomial** Set of all problems that we can solve in Polynomial (n^c) Time. All problems that we had in this course.

Computation Complexity - P, EXP , R



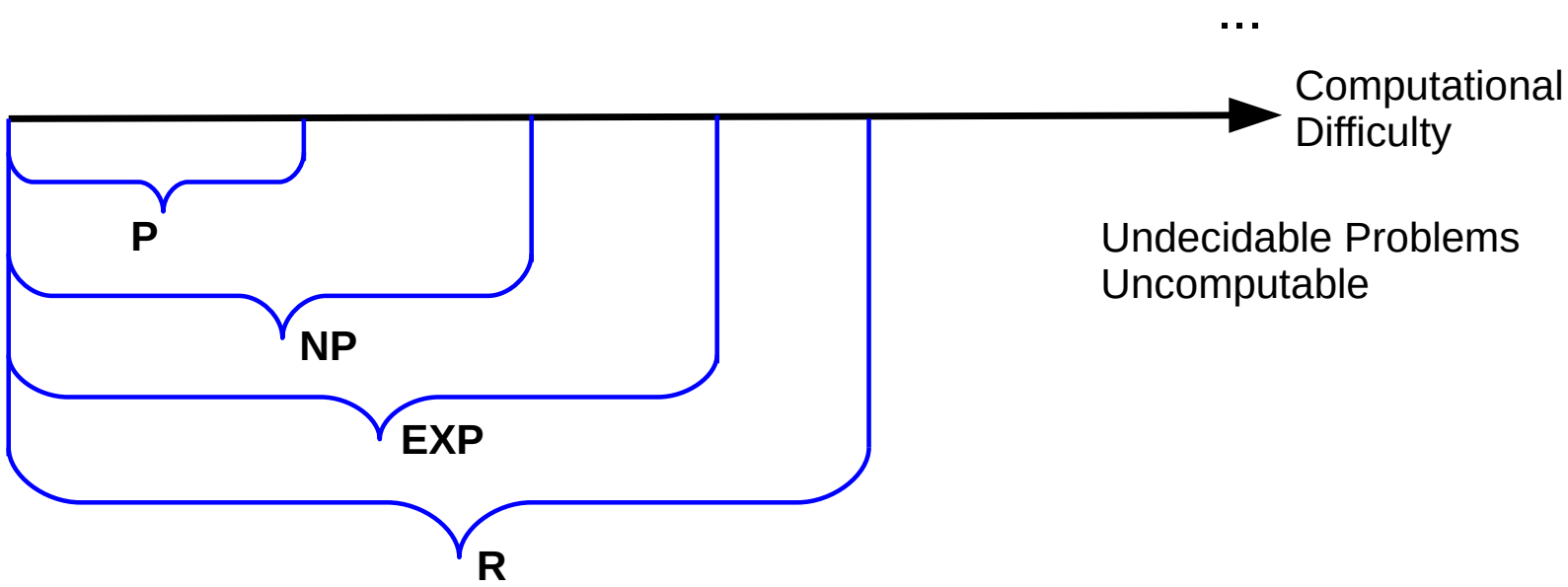
- **P - Polynomial** Set of all problems that we can solve in Polynomial (n^c) Time. All problems that we had in this course.
- **EXP - Exponential** Set of all problems that we can solve in exponential (2^{n^c}) time.

Computation Complexity - P, EXP , R



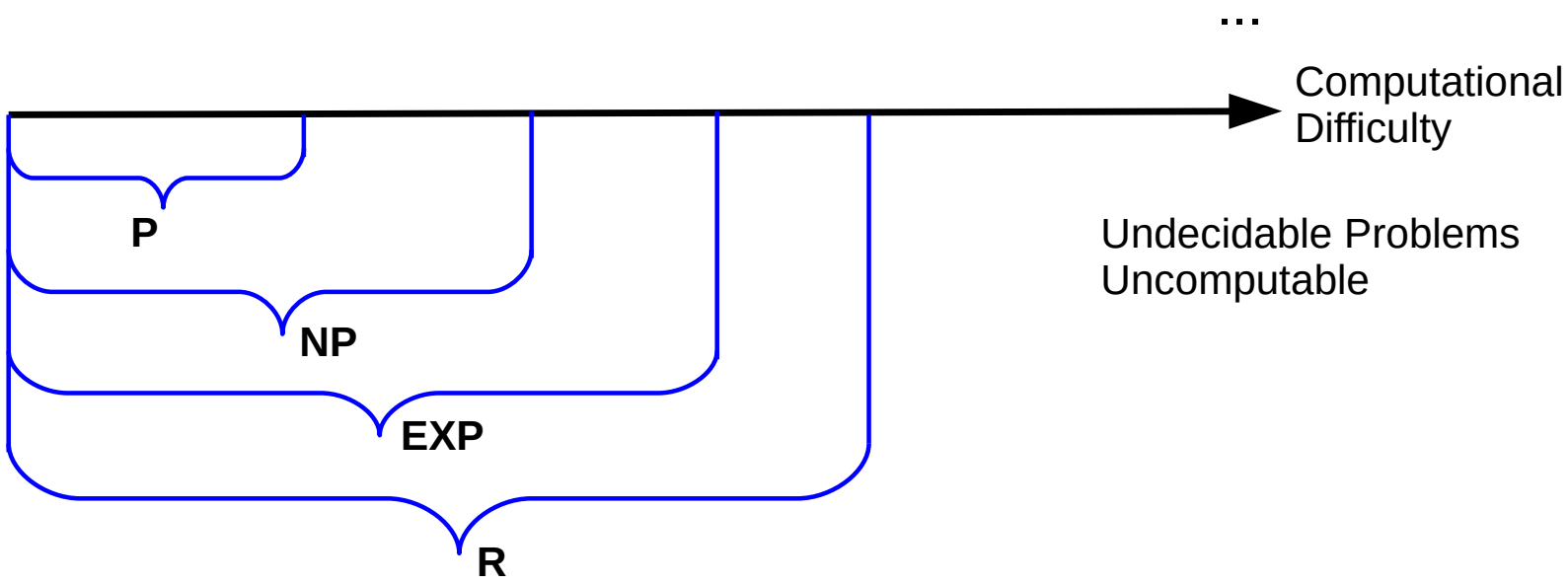
- **P - Polynomial** Set of all problems that we can solve in Polynomial (n^c) Time. All problems that we had in this course.
- **EXP - Exponential** Set of all problems that we can solve in exponential (2^{n^c}) time.
- **R - Recursive (finite time)** Set of all problems that we can solve in finite time.

Computation Complexity - P, NP, EXP , R



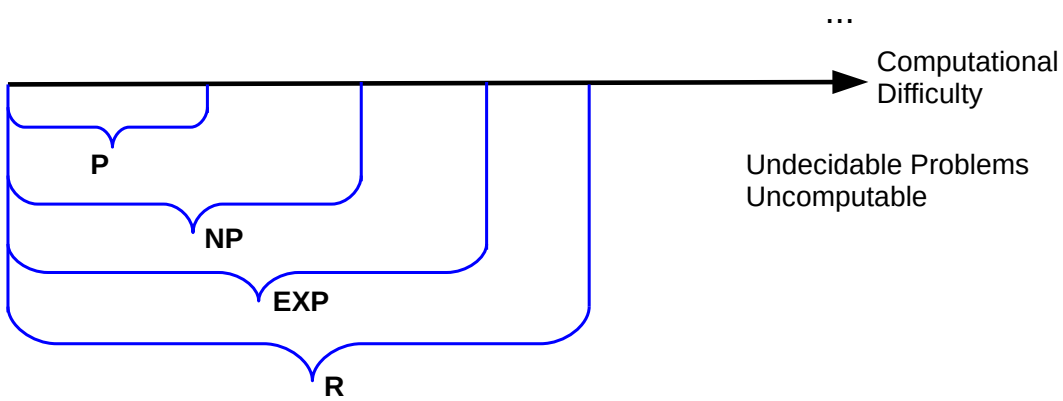
- **NP - Nondeterministic Polynomial** NP is the set of all problems that we can solve by guessing the solution and following a set of steps in Polynomial time.

Computation Complexity - P, NP, EXP , R



- **NP - Nondeterministic Polynomial** NP is the set of all problems that we can solve by guessing the solution and following a set of steps in Polynomial time.
- **What is Nondeterministic?** A nondeterministic algorithm makes random guesses and the answers the decision problem with YES or NO.
- For example, the game Tetris is in set of NP problems, because we can show that we can solve Tetris by guessing in polynomial time.

Example Problems



- Many problems that we saw in this course are in P.
- Shortest path in a weighted directed graph is in P.
- Detection of cycles in unweighted directed graph is in P.
- **Playing Chess on an $n \times n$** size board is in EXP and not in P.
- Play a game named **Tetris** is in Exponential time.

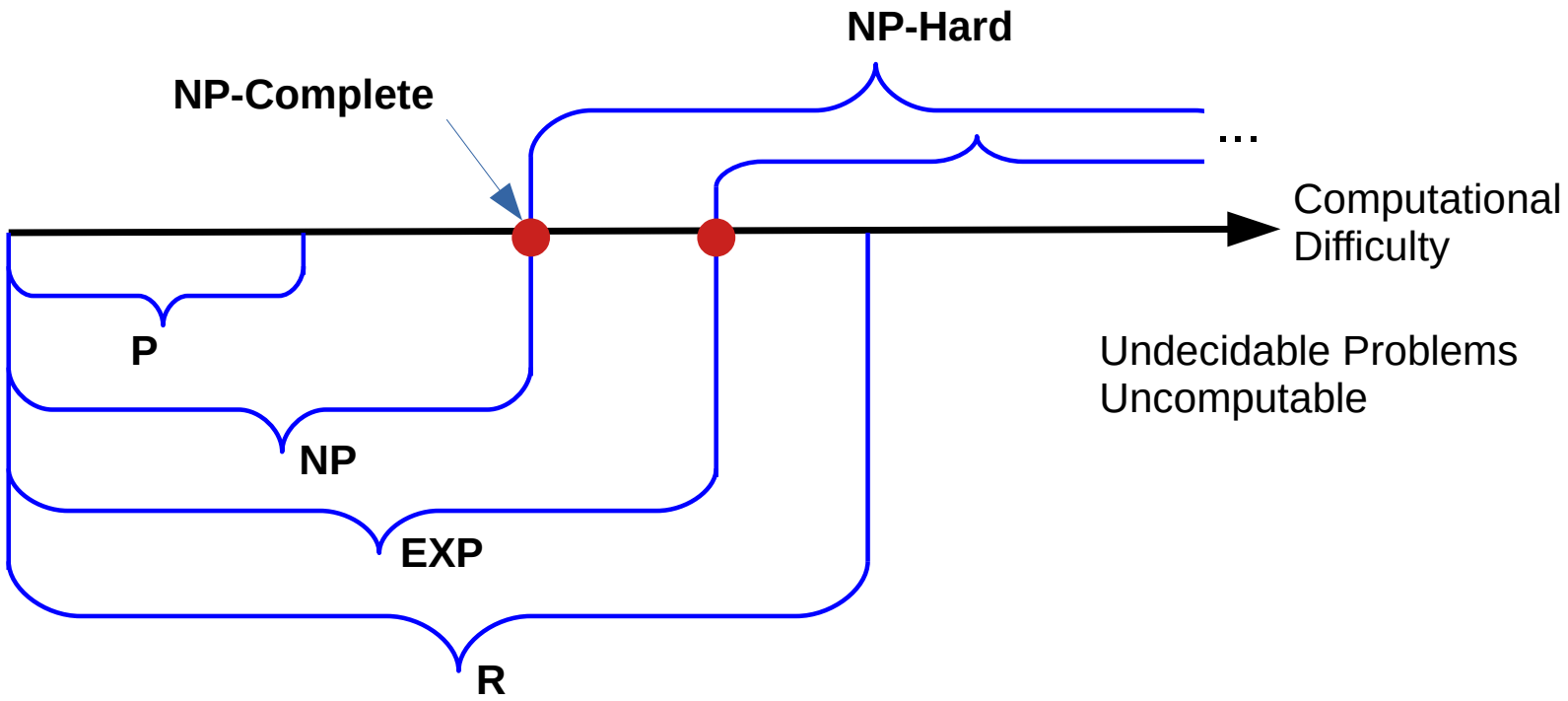
Halting-Problem.

- Give an arbitrary computer program, knowing if the program will ever terminate and stop. The return value is a True/False, e.g. True if terminates and False if not.
- The problem is uncomputable. You can say the problem is not in R. There is not algorithm that solves this problem in finite time on all given inputs.

$P \neq NP$

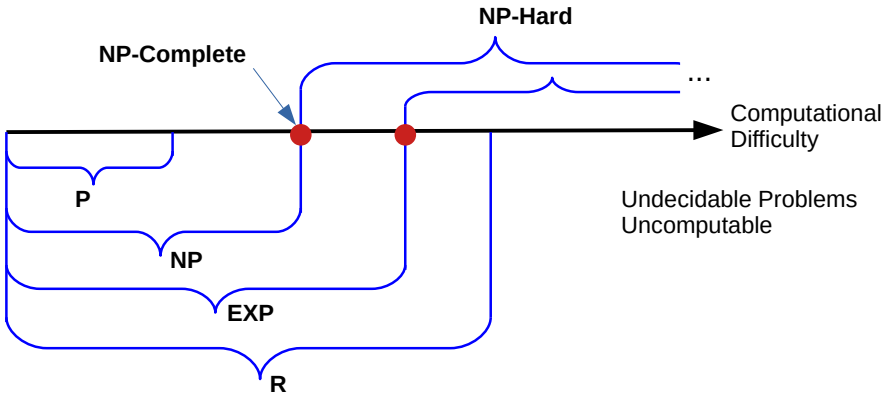
- Is $P \neq NP$? **This is a billion dollar question.**
- In other words, can we proof that we can engineer the luck in its general form.
- Can we make a lucky machine?

NP-Complete



- **NP-Complete.** All problems that are in the intersection of NP and NP-Hard problems. For example Tetris is a NP-Complete problem.

NP-Complete



- **NP-Complete.** All problems that are in the intersection of NP and NP-Hard problems. For example Tetris is a NP-Complete problem.
- **Problem Reduction** means that we can somehow convert our problem into other type of problems that we know how to solve it. We saw in our course a couple of these techniques.
- NP-Complete problems are problems that are interreducible using polynomial time reduction techniques. For example, the 0-1 Knapsack Problem.

NP-Complete - Example Travelling Salesman Problem

Travelling Salesman Problem:

- A sales man wants to visit a list of cities.
- Given is the list and the distances between each pair of cities.
- **Problem. What is the shortest possible route to visit cities?**

This problem is NP-Complete and NP-Hard (hardest problems in NP).

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Chapter 16 Greedy Algorithms
- Section 16.1 An activity-selection problem

Thank You!