Handout: A Quick Summary on Undecidability and NP-Completeness

CS6033 Design and Analysis of Algorithms I

By Prof. Yi-Jen Chiang

CSE Dept., Tandon School of Engineering, New York University

Note: There are 5 pages.

Complementary to designing efficient algorithms, you should know that there are problems that cannot even be computed/solved/decided by a computer (the topic of **undecidability**), and that there are problems that can be solved by a computer but **very unlikely to be solved/computed efficiently** (the topic of **NP-completeness**). This handout gives a quick summary on these topics below.

1. Undecidability

Turing Machine

Turing machine is a mathematical model of computation. It consists of a finite-state control unit, a tape, and a read/write head pointing at some tape cell from the control unit (see Fig. 1). The control unit operates in discrete steps; at each step it reads the symbol from the read/write head, and depending on the current state q and the symbol x read, it does the following:

1. Put the control unit into a new state q'.
2. *Either*
   (a) Write a symbol y in the tape cell of the read/write head position, effectively replacing x by y; *or*
   (b) Move the read/write head one tape cell to the left or right.

The action can be expressed as a *rule of action* $(q, x) \to (q', y)$ or $(q, x) \to (q', m)$ where m is L (denoting moving left) or R (denoting moving right).

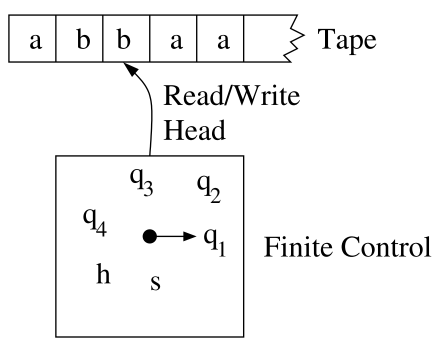


Figure 1: An example of a Turing machine.

Initially, the control unit is in the *initial state* s, and the input string w is given on the tape with the read/write head pointing to the leftmost symbol of w. Then the Turing machine acts according to the rules of action; when it reaches a special state, the *halting state* h, the computation stops and the result of the computation is left on the tape. It is possible that for some input the computation never stops.

Finite-Length Encoding of a Turing Machine. Note that there are **fixed** number of states and also **fixed** number of symbols that can appear on the tape. Therefore, the number of rules of action is also **fixed** (e.g., if there are 6 states and 3 symbols then there are 18 rules, since the number of tuples (q, x) where q is a state and x is a symbol is $6 \cdot 3$). In other words, each Turing machine has a **finite description**, and thus can be **encoded into a finite-length string**.

Turing Machines and Functions. We can see that each Turing machine TM corresponds to a function F: for any input string w, if TM stops and produces the output α, then w is in the domain of F with $F(w) = \alpha$; for other cases (TM does not stop on w or TM stops but does not accept w as a valid input, etc.) then w is not in the domain of F. It turns out that Turing machines have the same computing powers as computers (in terms of what functions can be computed on them, **not** in terms of the computing efficiencies).

Universal Turing Machine. A nice property is the **universality** of Turing machines, namely, we can have a **universal Turing machine** TM^*: given an input $< e(M), w >$ where $e(M)$ is the *encoding* of some Turing machine M and w is a string, TM^* will **simulate** the actions of M on the input w. This essentially means that the universal Turing machine TM^* is a **general-purpose computer** and we can **program** it — the finite-length encoding $e(M)$ is some finite-length **program** (say in C++ or Java, etc.) and the simulation is to run the program on the input w.

Countable vs. Uncountable Sets

We say that a set A is **countable** if **every** element $a \in A$ can be mapped to a **distinct** number from the set $\mathbb{N}$ of natural numbers $\{1, 2, 3, \cdots\}$. For example, the set $\mathbb{Z}$ of all integers is countable, since we can count them as follows: count/map $0, 1, -1, 2, -2, \cdots$, in that order, as $1, 2, 3, 4, \cdots$ (note: counting the integers in the order of $0, 1, 2, 3, \cdots$ does *not* work, since this would only go along the positive integers and miss the negative ones). The set $\mathbb{Q}^+$ of positive rational numbers is countable, since such numbers can be represented as p/q where $p = 1, 2, 3, \cdots$ and $q = 1, 2, 3, \cdots$ (think of listing p/q in a 2-dimensional table where one dimension is indexed by p and the other by q). We start by counting the tuples (p, q) in the order of $(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), \cdots$. That is, with $p + q = 2$ there is one tuple $(1, 1)$; with $p + q = 3$ there are two tuples $(1, 2), (2, 1)$; with $p + q = 4$ there are three tuples $(1, 3), (2, 2), (3, 1)$, etc. (this is called *dovetailing*). In this way all tuples (p, q) are counted and no one is missed. The set $\mathbb{Q}^-$ of all negative rational numbers can be counted symmetrically. Therefore, the set $\mathbb{Q}$ of all rational numbers is countable, since we can first count 0, then the first number in $\mathbb{Q}^+$, the first number in $\mathbb{Q}^-$, the second number in $\mathbb{Q}^+$, the second number in $\mathbb{Q}^-$, and so on. For an infinite set, we say that it is **countably infinite** if it is countable; otherwise it is **uncountably infinite**.

Claim 1. *The set of all Turing machines is countably infinite.*

Proof: Each Turing machine M corresponds to its encoding $e(M)$, which is a finite-length string. Therefore we can sort these encodings/strings first by their length and then by their lexicographical order within the group of the same length. We count these encodings/strings in this final sorted order as $1, 2, 3, \cdots$. Clearly all strings are counted and thus the set is countably infinite. □

Claim 2. *The set of all real numbers in the range* $[0, 1)$ *is uncountably infinite.*

Proof: The proof uses a proof technique called **diagonalization**. Consider representing such real numbers in binary, then each such $r \in [0, 1)$ is in the form $0.b_1\, b_2\, b_3 \cdots$, where b_i is the i-th bit after the decimal point, and b_i is either 0 or 1. Assume that the set in question is countable, i.e., we can label all real numbers in $[0, 1)$ as $r_1, r_2, r_3, \cdots$. Then we can list them in the binary form as shown in Fig. 2, where each bit b_{ij} is either 0 or 1. We can assume that each r_i has infinite number of bits,

$$
\begin{aligned}
r_1 &= 0.b_{11}\, b_{12}\, b_{13}\, b_{14} \cdots \\
r_2 &= 0.b_{21}\, b_{22}\, b_{23}\, b_{24} \cdots \\
r_3 &= 0.b_{31}\, b_{32}\, b_{33}\, b_{34} \cdots \\
&\cdots \\
&\cdots \\
r_k &= 0.b_{k1}\, b_{k2}\, b_{k3}\, b_{k4} \cdots b_{kk} \cdots
\end{aligned}
$$

Figure 2: Proof by diagonalization.

by appending bits of 0 if needed. Now consider the real number $r^* = 0.b_{11}\, b_{22}\, b_{33}\, b_{44} \cdots b_{kk} \cdots$, where $b_{ii}, i = 1, 2, \cdots$ are the diagonal bits marked by the red lines in Fig. 2. Finally, we let $\overline{r^*} = 0.\overline{b_{11}}\, \overline{b_{22}}\, \overline{b_{33}}\, \overline{b_{44}} \cdots \overline{b_{kk}} \cdots$, where $\overline{b_{ii}}$ means the complement of b_{ii}: if $b_{ii} = 0$ then $\overline{b_{ii}} = 1$; else ($b_{ii} = 1$) then $\overline{b_{ii}} = 0$. Clearly, $\overline{r^*}$ **differs from each** $r_1, r_2, \cdots$ **at the diagonal bit** b_{ii}**.** However, $\overline{r^*}$ is a real number in $[0, 1)$ and we must have $\overline{r^*} = r_k$ for some $k \in \mathbb{N}$, which is a contradiction since $\overline{r^*}$ differs from r_k at b_{kk}. Therefore it is not possible to label all real numbers in $[0, 1)$ as $r_1, r_2, \cdots$, meaning that the set in question is uncountable. $\square$

Note that the above proof shows that an uncountable set S has "more elements" than a countable set — when trying to count/label the elements of S by $1, 2, 3, \cdots$ there is always some element of S "missing" that cannot be so counted/labeled.

Take-Away Message: There are more functions than Turing machines

For each real number $x \in [0, 1)$ we can define a **distinct function** $f_x : [0, 1) \to [0, 1)$ as follows: $f_x(0) = x$, and $f_x(t) = t$ for all $t \in [0, 1)$ and $t \neq 0$ (note that each f_x maps the same 0 to a **distinct** value x, and thus each f_x is **distinct**). Since by Claim 2 there are uncountably many real numbers $x \in [0, 1)$, there are also **uncountably many** functions f_x. However, by Claim 1, there are only countably many Turing machines. This means that there are **more functions than Turing machines**, and thus **there are functions that cannot be computed by Turing machines/computers,** i.e., there are problems that cannot be computed/solved/decided by a computer.

A famous undecidable/unsolvable problem is the **halting problem**: "Given a Turing machine encoding/computer program M and an input w, can M terminate on w?" (For a general M and a general w, this is undecidable! This can be proved by diagonalization.)

2. NP-Completeness

Now we consider the problems that can be computed/solved/decided by Turing machines/computers, but the focus is on whether they can be done efficiently or not. In general, polynomial-time solutions are considered efficient, and solutions requiring more than polynomial time (e.g., exponential time) are considered inefficient.

P vs. NP

Definition of P: The class P is define to be the set of problems that can be solved in polynomial time

(P stands for polynomial time). There is also a class NP, where NP stands for **non-deterministic polynomial time**. For this, we first need to revisit Turing machines. In the Turing machine discussed in the beginning of Sec. 1, each rule of action ($(q, x) \to (q', y)$ or $(q, x) \to (q', m)$) describes a **deterministic** action, i.e, for each tuple (q, x) there is a unique action. Such Turing machine is called a **deterministic Turing machine**, and for each input w, the computation process is a single path. For a problem in P, the computation is a **single path with polynomial length** for a deterministic Turing machine.

In a **non-deterministic Turing machine**, each rule of action is of the form $(q, x) \to S$, where S is a **finite set** of possible actions, i.e., $S = \{s_1, s_2, \cdots, s_k\}$ for some fixed k and each s_i is either some (q', y) or some (q', m). This means that for a given state q and the input symbol x read, it will **non-deterministically** choose some $s_i \in S$ as the action (note that we do not say *randomly* since there is **no probability involved**). For each input w, the computation process is a **tree** where each internal node has fan-out $|S|$ (the number of actions in S) if the corresponding rule of action is $(q, x) \to S$. The computation tree contains many paths; some may not even stop. Among these paths, if **there exists a path that stops in polynomial time** (i.e., **if there exists a path of polynomial length**), then we say that the input w takes **non-deterministic polynomial time**.

Definition of NP: The class NP is defined to be the set of problems that can be solved in non-deterministic polynomial time. Equivalently, it is the set of problems for which **a given solution can be verified in polynomial time.** For example, given a (large) integer N, factoring it is difficult, but given two factors a and b as an answer, we can verify that $N = a \cdot b$ in polynomial time. Intuitively, P is a class of problems that are easy to compute, and NP is a class of problems that are easy to verify the solutions.

It is trivial to see that $P \subseteq NP$: from the deterministic Turing machine whose computation is a single path p of polynomial length, we can construct a non-deterministic Turing machine whose computation tree contains the path p.

All problems in NP have exponential time solutions. Recall that for any problem in NP, the corresponding computation tree of a non-deterministic Turing machine NTM has a path p of polynomial length ℓ. Then we can use a deterministic Turing machine TM to simulate NTM by exploring the computation tree in a **locked-step** fashion: in each round we advance the first path one more step, then the second path one more step, etc.; after the last path is advanced one more step, we repeat the process for the next round. This essentially performs a BFS in the computation tree; when we reach the leaf of path p then we find the solution and can stop. The total time is the **size of the computation tree cut at depth** ℓ; if the maximum fan-out in the tree is k then such size is $O(k^\ell)$, i.e., the total time is exponential.

Optimization vs. Decision Problems. An optimization problem asks for a solution that *minimizes/maximizes* an objective function, while a decision problem asks for a yes/no question. Note that the **optimization** version of a problem is **at least as hard as** the corresponding decision version. For example, for the shortest-path problem, the optimization version asks for the length of a shortest path from s to t, while the decision version asks, for a given value k, whether there is a path from s to t with length at most k. Clearly, if we have the optimal length ℓ, we can answer yes/no by checking whether $\ell \leq k$.

To show that a problem is hard, it suffices to show that its **decision version is already hard** (then the optimization version is even harder). In the following, we only consider the decision problems.

NP-Complete Problems: The **hardest** problems in NP

Definition of NP-Completeness: We say that a problem Q is **NP-complete** if

(a) Q is in NP, and

(b) for any problem K in NP, K can be *reduced* (i.e., *transformed*) to Q in polynomial time.

Meaning of (b): If there is a polynomial-time solution A to Q, then all problems K in NP can be solved in polynomial time —

method:

reduce/transform K to Q (which takes polynomial time), then solve Q in polynomial time by A, which then tells the answer to K (if yes to Q then yes to K, and if no to Q then no to K).

Therefore K can be solved in polynomial time.

Take-Away Message: Status of $\boldsymbol{P}$ vs. $\boldsymbol{NP}$

So far, we believe that $P \neq NP$, and we try to use NP-complete problems as witnesses that $P \neq NP$, as follows.

There are thousands of NP-complete problems; none of them has a polynomial-time solution (only exponential-time solutions are known), and if any of them has a polynomial-time solution, then all of them would have polynomial-time solutions (this follows from **Meaning of (b)** above), and thus **it is very unlikely that any of the NP-complete problems would have a polynomial-time solution.**

However, currently **there is no proof to show either** $P = NP$ **or** $P \neq NP$**.** Namely, **the status of "**$P = NP$**?" is still unknown**, and it is still the **biggest open problem** in Computer Science.

Elaboration: ScreenShots of Annotations

as a valid input, etc.) then w is not in the domain of F. It turns out that Turing machines have the same computing powers as computers (in terms of what functions can be computed on them, **not** in terms of the computing efficiencies).

Universal Turing Machine. A nice property is the **universality** of Turing machines, namely, we can have a **universal Turing machine** TM^*: given an input $< e(M), w >$ where $e(M)$ is the *encoding of some Turing machine* M and w is a string, TM^* will **simulate** the actions of M on the input w. This essentially means that the universal Turing machine TM^* is a **general-purpose computer** and we can **program** it — the finite-length encoding $e(M)$ is some finite-length **program** (say in C++ or Java, etc.) and the simulation is to run the program on the input w.

0 -1 1 -2 2

Countable vs. Uncountable Sets

We say that a set A is **countable** if **every** element $a \in A$ can be mapped to a **distinct** number from the set $\mathbb{N}$ of natural numbers $\{1, 2, 3, \cdots\}$. For example, the set $\mathbb{Z}$ of all integers is countable, since we can count them as follows: count/map $0, 1, -1, 2, -2, \cdots$, in that order, as $1, 2, 3, 4, \cdots$ (note: counting the integers in the order of $0, 1, 2, 3, \cdots$ does *not* work, since this would only go along the positive integers and miss the negative ones). The set $\mathbb{Q}^+$ of positive rational numbers is countable, since such numbers can be represented as p/q where $p = 1, 2, 3, \cdots$ and $q = 1, 2, 3, \cdots$ (think of listing p/q in a 2-dimensional table where one dimension is indexed by p and the other by q). We start by counting the tuples (p, q) in the order of $(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), \cdots$. That is, with $p + q = 2$ there is one tuple $(1, 1)$; with $p + q = 3$ there are two tuples $(1, 2), (2, 1)$; with $p + q = 4$ there are three tuples $(1, 3), (2, 2), (3, 1)$, etc. (this is called *dovetailing*). In this way all tuples (p, q) are counted and no one is missed. The set $\mathbb{Q}^-$ of all negative rational numbers can be counted symmetrically. Therefore, the set $\mathbb{Q}$ of all rational numbers is countable, since we can first count 0, then the first number in $\mathbb{Q}^+$, the first number in $\mathbb{Q}^-$, the second number in $\mathbb{Q}^+$, the second number in $\mathbb{Q}^-$, and so on. For an infinite set, we say that it is **countably infinite** if it is countable; otherwise it is **uncountably infinite**.

p/q

p\q 1 2 3

Dovetailing

Claim 1. *The set of all Turing machines is countably infinite.*

Proof: Each Turing machine M corresponds to its encoding $e(M)$, which is a finite-length string. Therefore we can sort these encodings/strings first by their length and then by their lexicographical

You are screen sharing | Stop Share
Inbox (341) - yjc293@nyu.edu | Microsoft PowerPoint - Lec-13-sl | Summary-Undecidable-NPC.pdf
File
Mouse Select Text Draw Stamp Spotlight Eraser Format Undo Redo Clear Save

Talking: Yi-Jen Chiang

the set $\mathbb{N}$ of natural numbers $\{1, 2, 3, \cdots\}$. For example, the set $\mathbb{Z}$ of all integers is countable, since we can count them as follows: count/map $0, 1, -1, 2, -2, \cdots$, in that order, as $1, 2, 3, 4, \cdots$ (note: counting the integers in the order of $0, 1, 2, 3, \cdots$ does *not* work, since this would only go along the positive integers and miss the negative ones). The set $\mathbb{Q}^+$ of positive rational numbers is countable, since such numbers can be represented as p/q where $p = 1, 2, 3, \cdots$ and $q = 1, 2, 3, \cdots$ (think of listing p/q in a 2-dimensional table where one dimension is indexed by p and the other by q). We start by counting the tuples (p, q) in the order of $(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1), \cdots$. That is, with $p + q = 2$ there is one tuple $(1, 1)$; with $p + q = 3$ there are two tuples $(1, 2), (2, 1)$; with $p + q = 4$ there are three tuples $(1, 3), (2, 2), (3, 1)$, etc. (this is called *dovetailing*). In this way all tuples (p, q) are counted and no one is missed. The set $\mathbb{Q}^-$ of all negative rational numbers can be counted symmetrically. Therefore, the set $\mathbb{Q}$ of all rational numbers is countable, since we can first count 0, then the first number in $\mathbb{Q}^+$, the first number in $\mathbb{Q}^-$, the second number in $\mathbb{Q}^+$, the second number in $\mathbb{Q}^-$, and so on. For an infinite set, we say that it is **countably infinite** if it is countable; otherwise it is **uncountably infinite**.

Claim 1. *The set of all Turing machines is countably infinite.*

Proof: Each Turing machine M corresponds to its encoding $e(M)$, which is a finite-length string. Therefore we can sort these encodings/strings first by their length and then by their lexicographical order within the group of the same length. We count these encodings/strings in this final sorted order as $1, 2, 3, \cdots$. Clearly all strings are counted and thus the set is countably infinite. $\square$

ε, 0, 1,
00, 01
10, 11

Claim 2. *The set of all real numbers in the range* $[0, 1)$ *is uncountably infinite.*

Proof: The proof uses a proof technique called **diagonalization**. Consider representing such real numbers in binary, then each such $r \in [0, 1)$ is in the form $0.b_1\, b_2\, b_3 \cdots$, where b_i is the i-th bit after

2

by appending bits of 0 if needed. Now consider the real number $r^* = 0.b_{11}\, b_{22}\, b_{33}\, b_{44} \cdots b_{kk} \cdots$, where $b_{ii}, i = 1, 2, \cdots$ are the diagonal bits marked by the red lines in Fig. 2. Finally, we let $\overline{r^*} = 0.\overline{b_{11}}\, \overline{b_{22}}\, \overline{b_{33}}\, \overline{b_{44}} \cdots \overline{b_{kk}} \cdots$, where $\overline{b_{ii}}$ means the complement of b_{ii}: if $b_{ii} = 0$ then $\overline{b_{ii}} = 1$; else ($b_{ii} = 1$) then $\overline{b_{ii}} = 0$. Clearly, $\overline{r^*}$ **differs from each** $r_1, r_2, \cdots$ **at the diagonal bit** b_{ii}**.** However, $\overline{r^*}$ is a real number in $[0, 1)$ and we must have $\overline{r^*} = r_k$ for some $k \in \mathbb{N}$, which is a contradiction since $\overline{r^*}$ differs from r_k at b_{kk}. Therefore it is not possible to label all real numbers in $[0, 1)$ as $r_1, r_2, \cdots$, meaning that the set in question is uncountable. □

Note that the above proof shows that an uncountable set S has "more elements" than a countable set — when trying to count/label the elements of S by $1, 2, 3, \cdots$ there is always some element of S "missing" that cannot be so counted/labeled.

Take-Away Message: There are more functions than Turing machines
For each real number $x \in [0, 1)$ we can define a **distinct function** $f_x : [0, 1) \to [0, 1)$ as follows: $f_x(0) = x$, and $f_x(t) = t$ for all $t \in [0, 1)$ and $t \neq 0$ (note that each f_x maps the same 0 to a **distinct** value x, and thus each f_x is **distinct**). Since by Claim 2 there are uncountably many real numbers $x \in [0, 1)$, there are also **uncountably many** functions f_x. However, by Claim 1, there are only countably many Turing machines. This means that there are **more functions than Turing machines**, and thus **there are functions that cannot be computed by Turing machines/computers,** i.e., there are problems that cannot be computed/solved/decided by a computer.

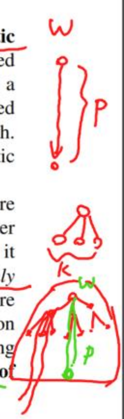
A famous undecidable/unsolvable problem is the **halting problem**: "Given a Turing machine encoding/computer program M and an input w, can M terminate on w?" (For a general M and a general w, this is undecidable! This can be proved by diagonalization.)

2. NP-Completeness

Now we consider the problems that can be computed/solved/decided by Turing machines/computers, but the focus is on whether they can be done efficiently or not. In general, polynomial-time solutions are considered efficient, and solutions requiring more than polynomial time (e.g., exponential time)

(P stands for polynomial time). There is also a class NP, where NP stands for **non-deterministic polynomial time**. For this, we first need to revisit Turing machines. In the Turing machine discussed in the beginning of Sec. 1, each rule of action ($(q, x) \to (q', y)$ or $(q, x) \to (q', m)$) describes a **deterministic** action, i.e, for each tuple (q, x) there is a unique action. Such Turing machine is called a **deterministic Turing machine**, and for each input w, the computation process is a single path. For a problem in P, the computation is a **single path with polynomial length** for a deterministic Turing machine.

In a **non-deterministic Turing machine**, each rule of action is of the form $(q, x) \to S$, where S is a **finite set** of possible actions, i.e., $S = \{s_1, s_2, \cdots, s_k\}$ for some fixed k and each s_i is either some (q', y) or some (q', m). This means that for a given state q and the input symbol x read, it will **non-deterministically** choose some $s_i \in S$ as the action (note that we do not say *randomly* since there is **no probability involved**). For each input w, the computation process is a **tree** where each internal node has fan-out $|S|$ (the number of actions in S) if the corresponding rule of action is $(q, x) \to S$. The computation tree contains many paths; some may not even stop. Among these paths, if **there exists a path that stops in polynomial time** (i.e., **if there exists a path of polynomial length**), then we say that the input w takes **non-deterministic polynomial time**.



Definition of NP**:** The class NP is defined to be the set of problems that can be solved in non-deterministic polynomial time. Equivalently, it is the set of problems for which **a given solution can be verified in polynomial time.** For example, given a (large) integer N, factoring it is difficult, but

Turing machine.

In a **non-deterministic Turing machine**, each rule of action is of the form $(q, x) \to S$, where S is a **finite set** of possible actions, i.e., $S = \{s_1, s_2, \cdots, s_k\}$ for some fixed k and each s_i is either some (q', y) or some (q', m). This means that for a given state q and the input symbol x read, it will **non-deterministically** choose some $s_i \in S$ as the action (note that we do not say *randomly* since there is **no probability involved**). For each input w, the computation process is a **tree** where each internal node has fan-out $|S|$ (the number of actions in S) if the corresponding rule of action is $(q, x) \to S$. The computation tree contains many paths; some may not even stop. Among these paths, if **there exists a path that stops in polynomial time** (i.e., **if there exists a path of polynomial length**), then we say that the input w takes **non-deterministic polynomial time**.

Definition of NP: The class NP is defined to be the set of problems that can be solved in non-deterministic polynomial time. Equivalently, it is the set of problems for which **a given solution can be verified in polynomial time.** For example, given a (large) integer N, factoring it is difficult, but given two factors a and b as an answer, we can verify that $N = a \cdot b$ in polynomial time. Intuitively, P is a class of problems that are easy to compute, and NP is a class of problems that are easy to verify the solutions.

It is trivial to see that $P \subseteq NP$: from the deterministic Turing machine whose computation is a single path p of polynomial length, we can construct a non-deterministic Turing machine whose computation tree contains the path p.

All problems in NP have exponential time solutions. Recall that for any problem in NP, the corresponding computation tree of a non-deterministic Turing machine NTM has a path p of polynomial length ℓ. Then we can use a deterministic Turing machine TM to simulate NTM by exploring the computation tree in a **locked-step** fashion: in each round we advance the first path one more step, then the second path one more step, etc.; after the last path is advanced one more step, we repeat the process for the next round. This essentially performs a BFS in the computation tree; when we reach the leaf of path p then we find the solution and can stop. The total time is the **size of the computation tree cut at depth** ℓ; if the maximum fan-out in the tree is k then such size is

given two factors a and b as an answer, we can verify that $N = a \cdot b$ in polynomial time. Intuitively, P is a class of problems that are easy to compute, and NP is a class of problems that are easy to verify the solutions.

It is trivial to see that $P \subseteq NP$: from the deterministic Turing machine whose computation is a single path p of polynomial length, we can construct a non-deterministic Turing machine whose computation tree contains the path p.

All problems in NP have exponential time solutions. Recall that for any problem in NP, the corresponding computation tree of a non-deterministic Turing machine NTM has a path p of polynomial length ℓ. Then we can use a deterministic Turing machine TM to simulate NTM by exploring the computation tree in a **locked-step** fashion: in each round we advance the first path one more step, then the second path one more step, etc.; after the last path is advanced one more step, we repeat the process for the next round. This essentially performs a BFS in the computation tree; when we reach the leaf of path p then we find the solution and can stop. The total time is the **size of the computation tree cut at depth** ℓ; if the maximum fan-out in the tree is k then such size is $O(k^\ell)$, i.e., the total time is exponential.

Optimization vs. Decision Problems. An optimization problem asks for a solution that *minimizes/maximizes* an objective function, while a decision problem asks for a yes/no question. Note that the **optimization** version of a problem is **at least as hard as** the corresponding decision version. For example, for the shortest-path problem, the optimization version asks for the length of a shortest path from s to t, while the decision version asks, for a given value k, whether there is a path from s to t with length at most k. Clearly, if we have the optimal length ℓ, we can answer yes/no by checking whether $\ell \leq k$.

To show that a problem is hard, it suffices to show that its **decision version is already hard** (then the optimization version is even harder). In the following, we only consider the decision problems.