

The Basic Concepts of Algorithms

R.C.T. Lee and Chin Lung Lu

Algorithms for Molecular Biology

What is algorithm?

- A method that can be used by a computer for the solution of a problem.
- A sequence of computational steps that transform the input into the output.
- Examples of algorithms in the nature: DNA and cook book.
- The word "algorithm" comes from the name of a Persian author, [Abu Ja'far Mohammed ibn Musa al Khowarizmi](#) (c. 825 A.D.), who wrote a textbook on mathematics.

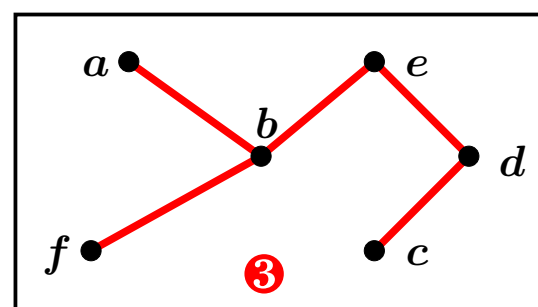
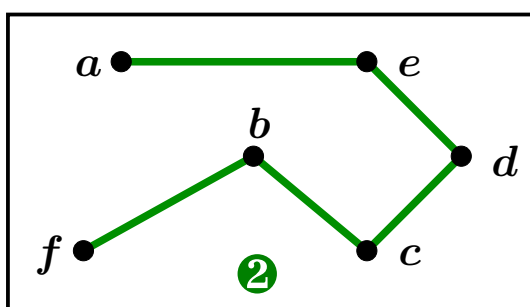
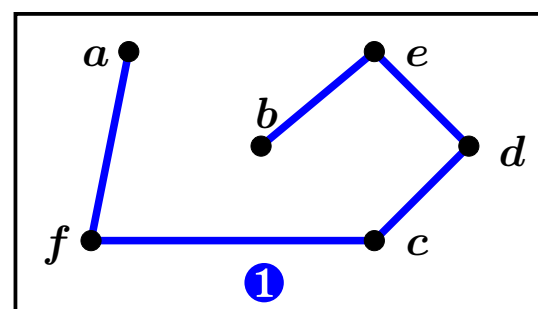
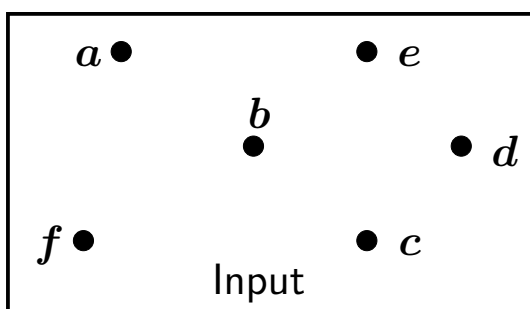
Why should we study algorithms?

- A good algorithm implemented on a slow computer may perform much better than a bad algorithm implemented on a fast computer.

$f(n) \setminus n$	10	10^2	10^3
$\log_2 n$	3.3	6.6	10
n	10	10^2	10^3
$n \log_2 n$	0.33×10^2	0.7×10^3	10^4
n^2	10^2	10^4	10^6
2^n	1024	1.3×10^3	$> 10^{100}$
$n!$	3^6	$> 10^{100}$	$> 10^{100}$

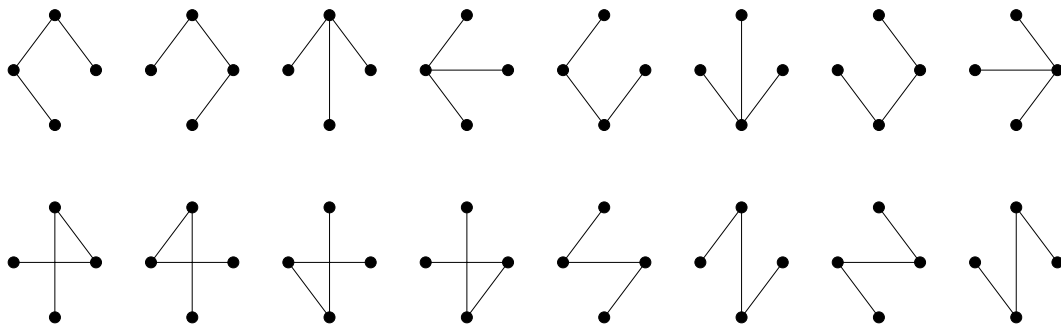
Minimal spanning tree problem

- Given a set of points, find a spanning tree with the shortest total length.



Minimal spanning tree problem

- **MST problem:** given a set of points, find a spanning tree with the shortest total length
- **Brute force method:** enumerate all possible spanning trees and select the best one among them
- Given n points, there are n^{n-2} possible spanning trees for them.



How to design a good algorithm?

- **Efficient or not?**
(**Efficient** means short time and small space.)
- **Strategies of algorithms:**
 1. Greedy
 2. Divide & conquer
 3. Prune & search
 4. Dynamic programming
 5. Branch and bound
 6. Approximation
 7. Heuristics

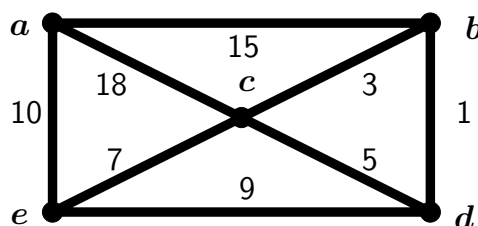
Prim's algorithm for MST

Input: A weighted and connected graph $G = (V, E)$.

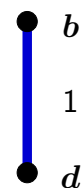
Output: A minimum spanning tree of G .

- 1: Let x be any vertex in V ;
Let $X = \{x\}$ and $Y = V \setminus \{x\}$;
- 2: Select an edge (u, v) from E such that $u \in X$, $v \in Y$ and (u, v) has the smallest weight among edges between X and Y ;
- 3: Connect u to v ;
Let $X = X \cup \{v\}$ and $Y = Y \setminus \{v\}$;
- 4: If Y is empty, terminate and the resulting tree is a minimal spanning tree;
Otherwise, go to step 2;

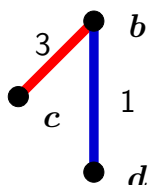
Prim's algorithm for MST



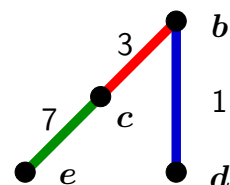
- ①
 $X = \{b\}$,
 $Y = \{a, c, e, d\}$,
 (b, d) the shortest.



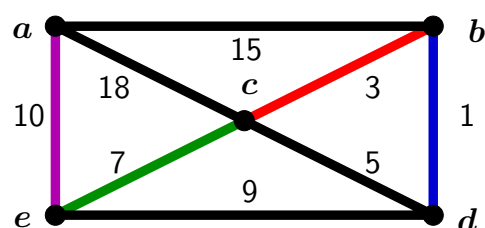
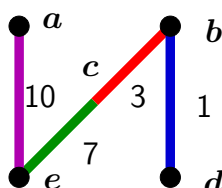
- ②
 $X = \{b, d\}$,
 $Y = \{a, c, e\}$,
 (b, c) the shortest.



- ③
 $X = \{b, d, c\}$,
 $Y = \{a, e\}$,
 (c, e) the shortest.



- ④
 $X = \{b, d, c, e\}$,
 $Y = \{a\}$,
 (e, a) the shortest.



Kruskal's algorithm for MST

Input: A weighted and connected graph $G = (V, E)$.

Output: A minimum spanning tree of G .

1: $T = \emptyset$;

2: **while** T contains less than $n - 1$ edges **do**

 Choose an edge (v, w) from E of smallest weight;

 Delete (v, w) from E ;

if adding (v, w) does not create cycle in T **then**

 Add (v, w) to T ;

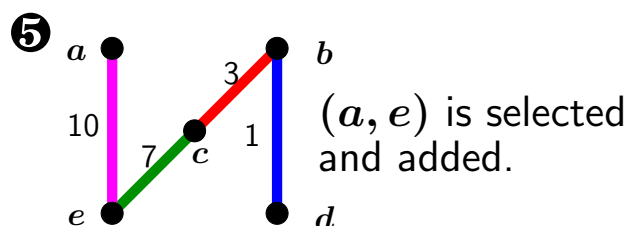
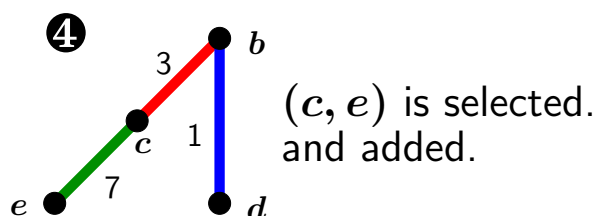
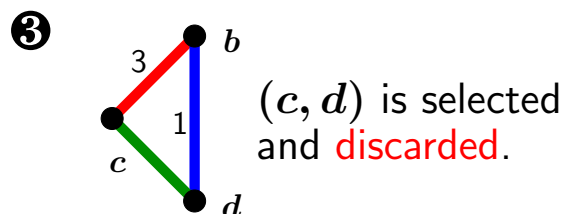
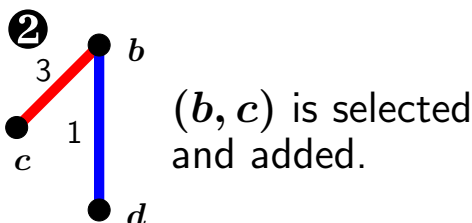
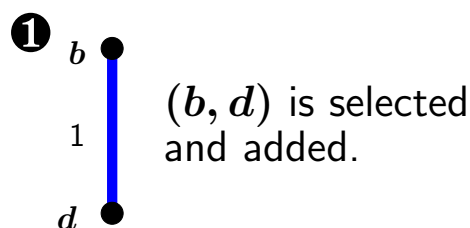
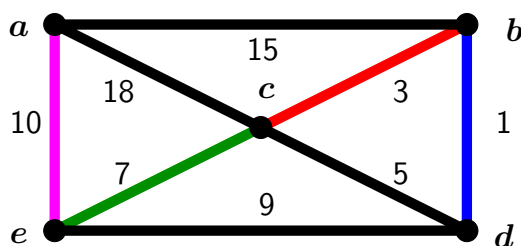
else

 Discard (v, w) ;

end if

end while

Kruskal's algorithm for MST



How to measure time complexity of algorithm \mathcal{A} ?

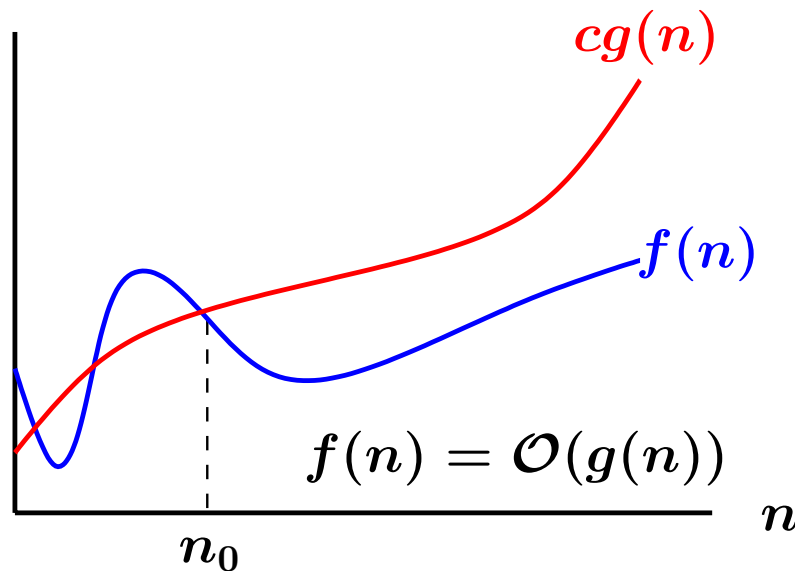
1. Write a program for \mathcal{A} and see how fast it runs.
 - Not suitable for many factors unrelated to \mathcal{A} , such as the capability of programmer, the used language and compiler, operating system, CPU's speed etc.
2. Choose the particular steps in \mathcal{A} and determine the number of the needed steps
 - The particular steps are time-consuming operations, like comparison of data, movement of data, $+$, $-$, $*$, $/$ operations etc.

Time complexity of algorithm \mathcal{A}

- **Time complexity of an algorithm:**
 - Equal to number of operations in algorithm \mathcal{A}
 - Usually represented by a function of the size of the input
 - **Size of the input:**
 1. sorting: number of items
 2. graph problems: number of vertices and edges
 3. multiplying two integers: number of bits
 - **Example:** For MST problem,
Prim's algorithm = $|V|^2$ time
Kruskal's algorithm = $|E| \log |E| + |V|$ time

\mathcal{O} notation

$\mathcal{O}(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq n_0\}$.

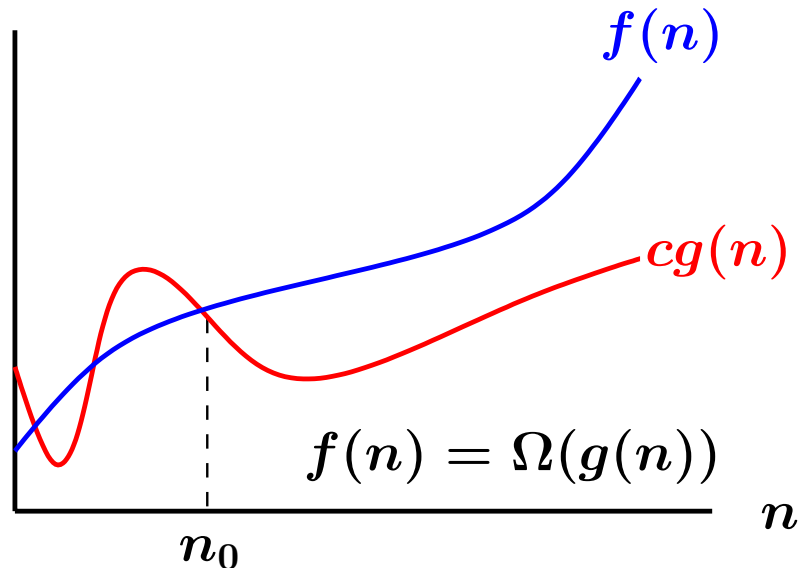


\mathcal{O} notation

- $\mathcal{O}(g(n))$ actually denotes a set of functions.
- $f(n) = \mathcal{O}(g(n))$ indicates that $f(n)$ is a member of $\mathcal{O}(g(n))$.
- **Example:** Let $f(n) = \frac{1}{2}n^2 - 3n$. Then
 1. $f(n) = \mathcal{O}(n^2)$ (✓)
 2. $f(n) = \mathcal{O}(n^3)$ (✓)
 3. $f(n) = \mathcal{O}(n)$ (✗)
- "The running time of an algorithm \mathcal{A} is $\mathcal{O}(n^2)$ " means that the worst-case running time of \mathcal{A} is $\mathcal{O}(n^2)$.

Ω notation

$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } f(n) \geq cg(n) \text{ for all } n \geq n_0\}$.

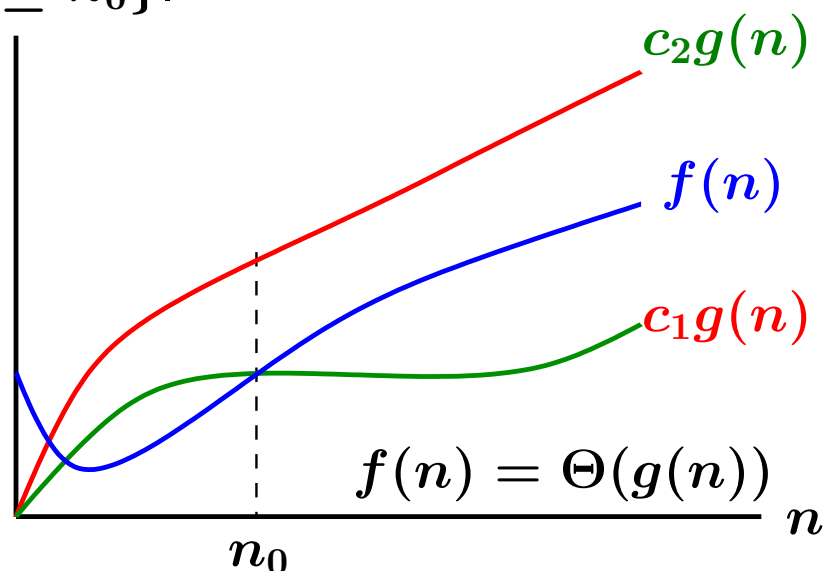


Ω notation

- $\Omega(g(n))$ actually denotes a set of functions.
- $f(n) = \Omega(g(n))$ indicates that $f(n)$ is a member of $\Omega(g(n))$.
- **Example:** Let $f(n) = \frac{1}{2}n^2 + 3n$. Then
 1. $f(n) = \Omega(n^2)$ (✓)
 2. $f(n) = \Omega(n^3)$ (✗)
 3. $f(n) = \Omega(n)$ (✓)
 4. $f(n) = \Omega(1)$ (✓)

Θ notation

$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.



Θ notation

- $\Theta(g(n))$ actually denotes a set of functions.
- $f(n) = \Theta(g(n))$ indicates that $f(n)$ is a member of $\Theta(g(n))$.
- Let $f(n) = \frac{1}{2}n^2 - 3n$. Then $f(n) = \Theta(n^2)$, but $f(n) \neq \Theta(n)$ and $f(n) \neq \Theta(n^3)$.
 - **Skill:** ignore the lower-order terms and the coefficient of the highest-order term
- Any constant is denoted by $\Theta(1)$.
(Any constant is a degree-0 polynomial, so it can be repressed as $\Theta(n^0)$ or $\Theta(1)$.)

The constant hidden in \mathcal{O} notation

- Let A_1 and A_2 be two algorithms of solving the same problem and their time complexities be $\mathcal{O}(n)$ and $\mathcal{O}(n^3)$, respectively.
- If we ask the same person to write two programs, say P_1 and P_2 respectively, for A_1 and A_2 under the same programming environment,
would P_1 run faster than P_2 ?
 - P_1 runs faster than P_2 when n is large.
 - P_2 may run faster than P_1 when n is small.
(The constant hidden in \mathcal{O} notation can not be ignored.)

Types of complexity of algorithm

- Let $T(I)$ be the time complexity of an algorithm \mathcal{A} for instance I .
 1. Best case: $\min\{T(I) : \text{for all } I\}$
 2. Average case: $\sum\{T(I) \cdot \text{prob}(I) : \text{for all } I\}$
 - $\text{prob}(I)$: probability of the occurrence of I
 3. Worst case: $\max\{T(I) : \text{for all } I\}$
- Usually, we use \mathcal{O} -notation and Ω -notation to denote the **upper bound** (worst case) and **lower bound** (best case) of algorithm \mathcal{A} , respectively.

Insertion sorting algorithm

Input: x_1, x_2, \dots, x_n .

Output: The sorted sequence of x_1, x_2, \dots, x_n .

```
1. for  $j = 2$  to  $n$  do /* Outer loop */
2.    $i = j - 1$ ;
3.    $x = x_j$ ;
4.   while  $x < x_i$  and  $i > 0$  do /* Inner loop */
5.      $x_{i+1} = x_i$ ;
6.      $i = i - 1$ ;
7.   end while
8.    $x_{i+1} = x$ ;
9. end for
```

An example of insertion sort

- Let the input sequence be 7, 5, 1, 4, 3, 2, 6.
- The process of insertion sorting is as follows.
 - $7 \leftarrow 7, 5, 1, 4, 3, 2, 6$ (Initial state)
 - $5, 7 \leftarrow 5, 1, 4, 3, 2, 6$
 - $1, 5, 7 \leftarrow 1, 4, 3, 2, 6$
 - $1, 4, 5, 7 \leftarrow 4, 3, 2, 6$
 - $1, 3, 4, 5, 7 \leftarrow 3, 2, 6$
 - $1, 2, 3, 4, 5, 7 \leftarrow 2, 6$
 - $1, 2, 3, 4, 5, 6, 7 \leftarrow 6$ (Final state)

Complexity of insertion sorting

Use the number of data movements as the time complexity measurement: $\mathcal{X} = \sum_{j=2}^n (2 + d_j)$

- Outer loop: $x = x_j, x_{i+1} = x$ (always executed)
- Inner loop: $x_{i+1} = x_i$ (not always executed)
 $d_j = |\{x_i : x_i > x_j, 1 \leq i < j\}|$
- Best Case: sorted sequence ($d_1 = \dots = d_n = 0$)
 $\mathcal{X} = 2(n - 1) = \mathcal{O}(n)$
- Worst Case: reversely sorted sequence
($d_2 = 1, d_3 = 2, \dots, d_n = n - 1$)
 $\mathcal{X} = \frac{(n-1)(n+4)}{2} = \mathcal{O}(n^2)$

Complexity of insertion sorting

- Average Case: $\sum_{j=2}^n \frac{j+3}{2} = \frac{(n+8)(n-1)}{4} = \mathcal{O}(n^2)$
- Let x_1, \dots, x_{j-1} be a sorted sequence and the next step is to insert x_j .
- If x_j is the i th largest number among the j numbers, there will be $i - 1$ movements in the inner loop (2 movements in the outer loop).
- The probability that x_j is the i th largest among j numbers is $\frac{1}{j}$.
- Therefore, the average number of movement is $\frac{2+0}{j} + \frac{2+1}{j} + \dots + \frac{2+j-1}{j} = \frac{j+3}{2}$.

Polynomial/Exponential algorithms

- **Polynomial algorithm:**
whose complexity is bound by $\mathcal{O}(n^k)$,
where n is the input size and k is a constant
- **Exponential algorithm:**
whose complexity is bound by $\mathcal{O}(k^n)$
- **Example:** For MST problem,
 - Prim and Kruskal's methods: polynomial
 - Brute force method: exponential
- Polynomial algorithms are better than exponential ones.

How to measure the difficulty of a problem \mathcal{P} ?

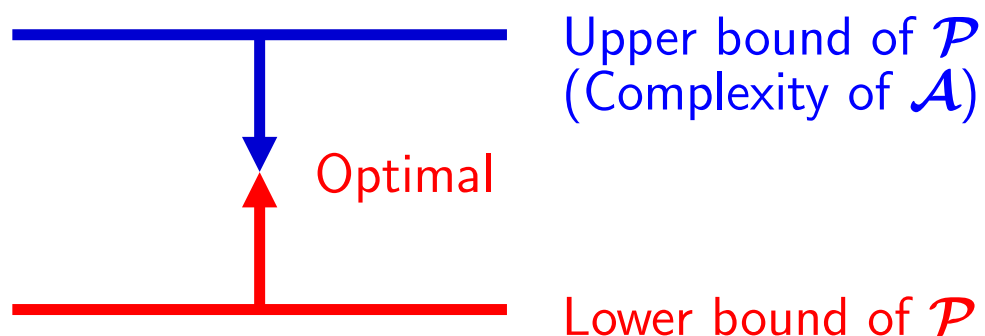
- Is problem \mathcal{P} solvable or not?
- If \mathcal{P} is solvable, the **time-complexity of \mathcal{P}** is $\min\{T(\mathcal{A}) : \mathcal{A} \text{ is an algorithm of } \mathcal{P}\}$,
where $T(\mathcal{A})$ is the time-complexity of \mathcal{A} .
- **Easy problem:** solvable in polynomial time
 - MST problem: greedy algorithm
- **Difficult problem:** impossibly find a polynomial time algorithm to solve it
 - Traveling salesperson problem: NP-complete
 - Halting problem: no algorithm

Upper/Lower bounds of problem

- **Upper bound of problem \mathcal{P}** : the complexity of the best one among algorithms solving \mathcal{P}
 - **Example**: The upper bound of MST problem is $\min\{\mathcal{O}(|V|^2), \mathcal{O}(|E| \log |E|)\}$.
- **Lower bound of \mathcal{P}** : use mathematical method to proof that any algorithm for \mathcal{P} must have at least time-complexity $f(n)$
 - **Example**: An trivial lower bound of MST problem is $\mathcal{O}(|V| + |E|)$.

How to know that an algorithm \mathcal{A} is optimal for a problem \mathcal{P} ?

- Is there any other better algorithm?
- \mathcal{A} is **optimal** if there is no other better algorithm.
- \mathcal{A} is **optimal** if the time-complexity of \mathcal{A} is equal to the lower bound of \mathcal{P} .



Decision/Optimization problems

- **Decision problem:** the problem whose solution is simply "yes" or "no"
 - **Example: Traveling salesperson decision problem:** Given a set of points and a constant c , is there a tour starting from any point v_0 whose total length is less than c ?
- **Optimization problem:** the problem of finding a solution whose value is optimal
 - **Example: Traveling salesperson problem:** Given a set of points, find a shortest tour which starts from any point v_0 .

Decision/Optimization problems

- The optimization problems are more difficult than their corresponding decision problems.
 - **If we can solve the traveling salesperson problem,** then we can solve the traveling salesperson decision problem, but not vice versa.
 - **If the traveling salesperson decision problem can not be solved by polynomial algorithms,** then we can conclude that the traveling salesperson problem can not be solved by polynomial algorithms.

The Satisfiability Problem (SAT)

- Given a Boolean formula, determine whether this formula is satisfiable or not.
- Consider the following formula:

$$\begin{aligned} & (x_1 \vee x_2 \vee x_3) \\ & \wedge (\neg x_1) \\ & \wedge (\neg x_2) \end{aligned}$$

The following **assignment** makes the formula true.

$$\begin{aligned} x_1 & \leftarrow F \\ x_2 & \leftarrow F \\ x_3 & \leftarrow T \end{aligned}$$

Time-complexity of SAT problem

- If there are n variables, then there are 2^n possible assignments for the SAT problem.
- Up to now, for the best available algorithms for the SAT problem, they cost exponential time in worst cases.
- Is there any possibility that the SAT problem can be solved in polynomial time?**
- By the **theory of NP-completeness**, if the SAT problem can be solved in polynomial time, then all NP problems can be solved in polynomial time.

Nondeterministic algorithm

- We may consider a nondeterministic algorithm as a algorithm consisting of two phases **guessing** and **checking**.
- Given two numbers $x(1) = 7$ and $x(2) \neq 7$, determine if there is a number which equals 7.
Guessing: $i = \text{choice}(1, 2)$;
Checking: if $x(i) = 7$ then SUCCESS
 else FAILURE.
- **Nondeterministic polynomial algorithm:**
a nondeterministic algorithm whose **checking stage**
can be done in polynomial time

P and NP problems

- **P problem:** a decision problem which can be solved by a **polynomial algorithm**, such as the MST decision problem and the longest common subsequence decision problem
- **NP problem:** a decision problem which can be solved by a **nondeterministic polynomial algorithm**, such as the SAT problem and the traveling salesperson decision problem
- **Every P problem must be an NP problem** (i.e., $P \subseteq NP$).

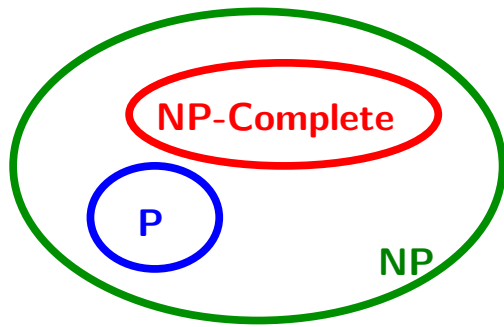
P and NP problems

- Are all problems NP problems?
 - Halting problem: given an arbitrary program with an arbitrary input data, will the program terminate or not?
 - Halting problem is not an NP problem because it is undecidable.
- Are all NP problems P problems? ($P = NP$?)
 - $P \subseteq NP$? (yes)
 - $P \supseteq NP$? (?): the SAT problem and the traveling salesperson decision problem can be solved in $O(2^n)$ and $O(n!)$ time, respectively.

NP-complete problem

- In discussing NP problems, we shall only discuss decision problems.
- Problem \mathcal{P}_1 reduces to problem \mathcal{P}_2 ($\mathcal{P}_1 \propto \mathcal{P}_2$):
 - \mathcal{P}_1 can be solved in polynomial time by using a polynomial time algorithm solving \mathcal{P}_2 .
 - \mathcal{P}_2 is more difficult than \mathcal{P}_1 .
- A problem \mathcal{P} is NP-complete if
 1. $\mathcal{P} \in NP$ and
 2. \mathcal{P} is NP-hard (every NP problem reduces to \mathcal{P}).
- The SAT problem was the first found NP-complete problem by Cook (1971).

Theory of NP-Completeness



- If any NP-complete problem can be solved in polynomial time, $NP = P$.
- Up to now, no NP-complete problem has any worst case polynomial algorithm.
- There are thousands of problems proved to be NP-complete problems.
- If the decision version of an optimization problem is NP-complete, this optimization problem is called NP-hard.