The Basic Concepts of Algorithms

Chin Lung Lu

Computational Biology

Analyses and Applications of Sequences

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 Abu Ja'far Mohammed ibn Musa al Khowarizmi (780–825 A.D.), an Arab mathematician who wrote a textbook on mathematics.

Al-Khowarizmi [Al-khOwArEz'mE]

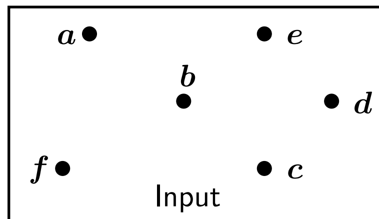
Why should we study algorithms?

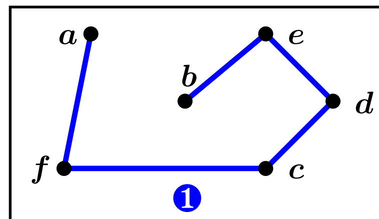
- A good algorithm implemented on a slow computer may perform much better than a bad algorithm implemented on a fast computer.

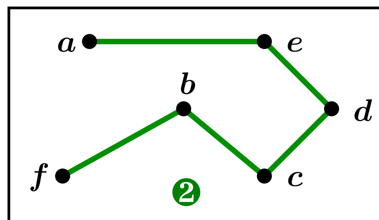
$f(n) \setminus n$	10	10^2	10^3
$log_2 n$	3.3	6.6	10
n	10	10^2	10^3
$nlog_2 n$	0.33×10^2	0.7×10^3	10^4
n^2	10^2	10^4	10^6
2^n	1024	1.3×10^3	$> 10^{100}$
$n!$	3^6	$> 10^{100}$	$> 10^{100}$

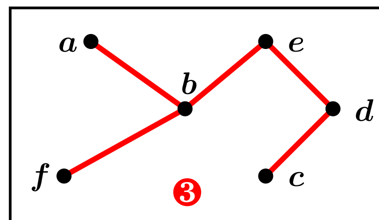
Minimal spanning tree problem

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



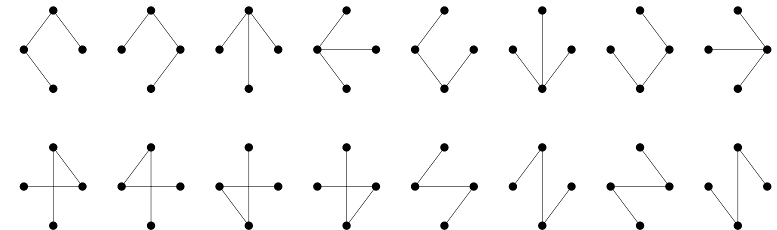






Minimal spanning tree problem

- 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?

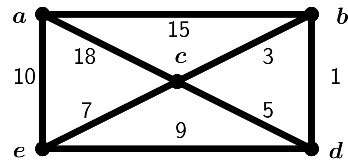
- Efficient or not?
  (Efficient means short time and small space.)
- Strategies of algorithms:
  1. Greedy
  2. Divide & conquer
  3. Prune & search
  4. Dynamic programming
  5. Branch and bound
  6. Approximation
  7. Heuristics

Prim's algorithm for MST

Input: A weighted and connected graph $G = (V, E)$.
Output: A minimum spanning tree of G.

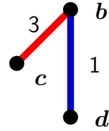
1: Let x be any vertex in V;
Let $X = \{x\}$ and $Y = V \setminus \{x\}$;
2: Select an edge (u, v) from E such that $u \in X$, $v \in Y$ and (u, v) has the smallest weight among edges between X and Y;
3: Connect u to v;
Let $X = X \cup \{v\}$ and $Y = Y \setminus \{v\}$;
4: If Y is empty, terminate and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

Prim's algorithm for MST

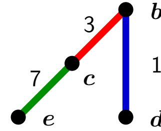


❶ $X = \{b\}$, $Y = \{a, c, e, d\}$, (b, d) the shortest.

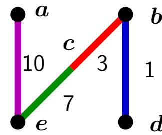
❷ $X = \{b, d\}$, $Y = \{a, c, e\}$, (b, c) the shortest.

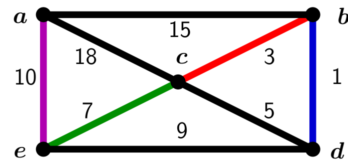


❸ $X = \{b, d, c\}$, $Y = \{a, e\}$, (c, e) the shortest.



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



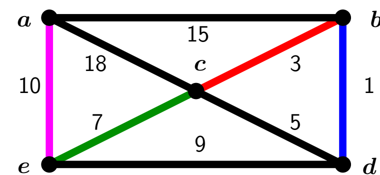


Kruskal's algorithm for MST

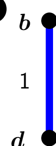
Input: A weighted and connected graph $G = (V, E)$.
Output: A minimum spanning tree of G.

1: $T = \varnothing$;
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

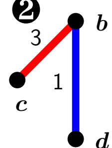


❶



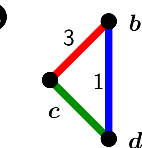
(b, d) is selected and added.

❷



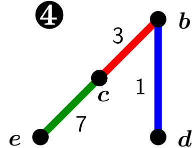
(b, c) is selected and added.

❸



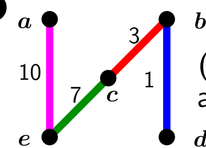
(c, d) is selected and discarded.

❹



(c, e) is selected. and added.

❺

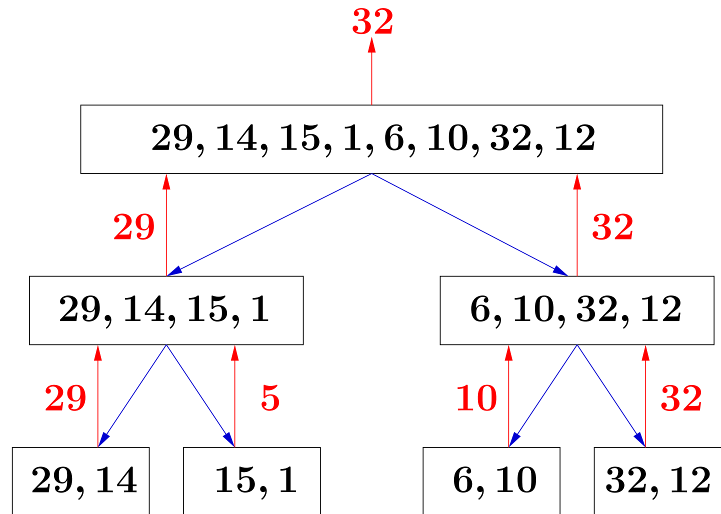


(a, e) is selected and added.

Divide-and-conquer strategy

- MaxFinding problem: Find the maximum of a set S of n numbers
- Divide-and-conquer approach:
  - Divide: divide S into two sets S_1 and S_2 such that each set consists of $\frac{n}{2}$ numbers
  - Conquer: Find the maximums of S_1 and S_2 recursively, which are denoted by X_1 and X_2, respectively
  - Merge: The maximum of $S = \max\{X_1, X_2\}$
- Time complexity $T(n) = 2T(\frac{n}{2}) + 1 = \mathcal{O}(n)$

MaxFinding problem



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

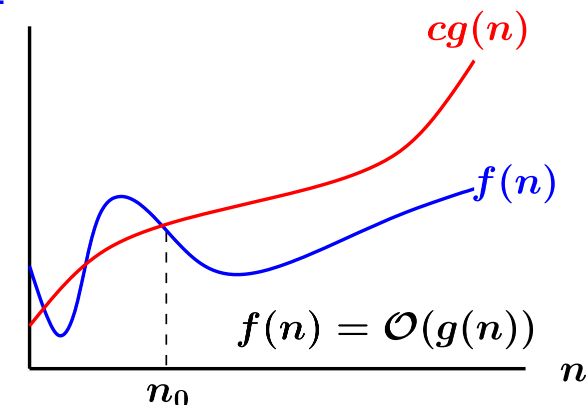
1. Write a program for $\mathcal{A}$ and see how fast it runs
   - Not suitable for many factors unrelated to $\mathcal{A}$, such as the capability of programmer, the used language and compiler, operating system, CPU's speed etc.
2. Use mathematical analysis to determine the number of the time-consuming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

Time-complexity of an algorithm

- The time of executing an algorithm is dependent on the size of the problem, denoted by n usually.
  - Most algorithms need more time to complete when n increases.
  - The time-complexity of an algorithm is usually represented by a function of n.
- Suppose it takes $n^3 + n$ steps to run an algorithm.
  - We say that the time-complexity of this algorithm is in the order of n^3.
  - n^3 dominates n as n becomes very large.

$\mathcal{O}$ notation

$f(n) = \mathcal{O}(g(n))$ iff there exist two positive constants c and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$.



Have a try...

- Suppose that it takes $n^3 + n$ steps to run an algorithm $\mathcal{A}$.
  - Then we can say that the time-complexity of $\mathcal{A}$ is $\mathcal{O}(n^3)$. (why?)
  - Prove $f(n) = \mathcal{O}(n^3)$ if let $f(n) = n^3 + n$.

$$
\begin{aligned}
f(n) &= n^3 + n \\
&= (1 + \tfrac{1}{n^2})n^3 \\
&\leq 2n^3 \qquad \text{for } n \geq 1
\end{aligned}
$$

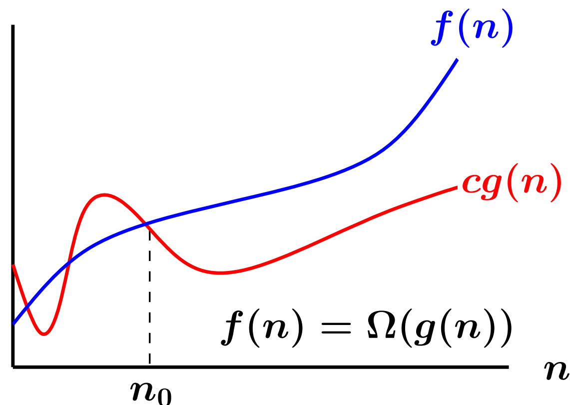
$\therefore$ we have $g(n) = n^3$, $c = 2$ and $n_0 = 1$.

Have other tries...

- 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. $f(n) = \mathcal{O}(n)$ (✘)
  4. $f(n) = \mathcal{O}(1)$ (✘)
- Let $f(n) = 2^{2005}n^2 - 3n$. Then
  1. $f(n) = \mathcal{O}(n^2)$ (?)
  2. $f(n) = \mathcal{O}(n^3)$ (?)
  3. $f(n) = \mathcal{O}(n)$ (?)
  4. $f(n) = \mathcal{O}(1)$ (?)

Ω notation

$f(n) = \Omega(g(n))$ iff there exist two positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

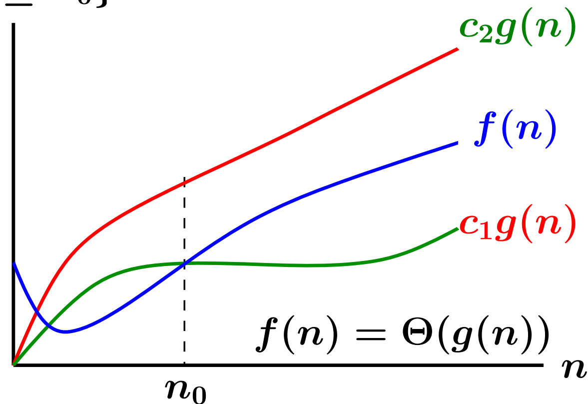


Ω notation

- **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

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



Θ notation

- 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 $\Theta(n^0)$ of $\Theta(1)$.)

The constant hidden in $\mathcal{O}$ notation

- Let A_1 and A_2 be two algorithm 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

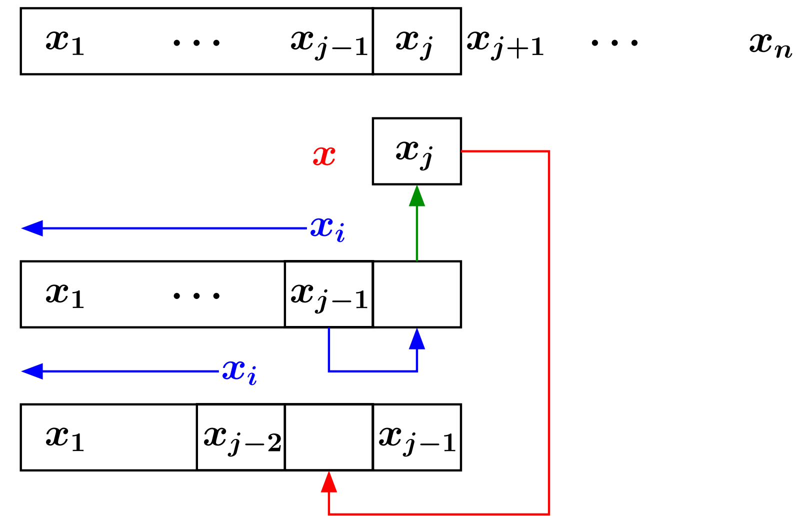
- Let $T(I)$ be the time complexity of an algorithm $\mathcal{A}$ for instance I.
  1. Best case: $\min\{T(I) :$ for all $I\}$
  2. Average case: sum$\{T(I) \cdot prob(I) :$ for all $I\}$
     - $prob(I)$: probability of the occurrence of I
  3. Worst case: $\max\{T(I) :$ for all $I\}$
- Usually, we 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: $x_1, x_2, \cdots, x_n$.
Output: The sorted sequence of $x_1, x_2, \cdots, x_n$.

```
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
```


Illustration of insertion sorting



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 ⇐ 7,5,1,4,3,2,6 (Initial state)
  - 5, 7 ⇐ 5,1,4,3,2,6
  - 1, 5, 7 ⇐ 1,4,3,2,6
  - 1, 4, 5, 7 ⇐ 4,3,2,6
  - 1, 3, 4, 5, 7 ⇐ 3,2,6
  - 1, 2, 3, 4, 5, 7 ⇐ 2,6
  - 1, 2, 3, 4, 5, 6, 7 ⇐ 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)
  $d_j = |\{x_i : x_i > x_j, 1 \le i < j\}|$
- Best Case: sorted sequence $(d_1 = \cdots = d_n = 0)$
  $\mathcal{X} = 2(n-1) = \mathcal{O}(n)$
- Worst Case: reversely sorted sequence
  $(d_2 = 1, d_3 = 2, \cdots, d_n = n-1)$
  $\mathcal{X} = \frac{(n-1)(n+4)}{2} = \mathcal{O}(n^2)$

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 $x_1, \cdots, x_{j-1}$ be a sorted sequence and the next step is to insert x_j.
  - If x_j is the ith 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 ith largest among j numbers is $\frac{1}{j}$.
  - Therefore, the average number of movement is $\frac{2+0}{j} + \frac{2+1}{j} + \cdots + \frac{2+j-1}{j} = \frac{j+3}{2}$.

Polynomial/Exponetial 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 $\mathcal{P}$?

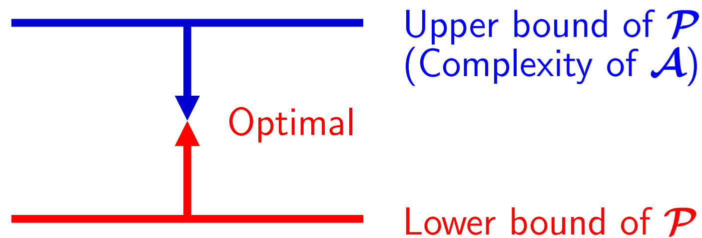
- Is problem $\mathcal{P}$ solvable or not?
- If $\mathcal{P}$ is solvable, the time-complexity of $\mathcal{P}$ is $\min\{T(\mathcal{A}) : \mathcal{A}$ is an algorithm of $\mathcal{P}\}$, where $T(\mathcal{A})$ is the time-complexity of $\mathcal{A}$.
- Easy problem: solvable in polynomial time
  - 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 probelm is $\min\{\mathcal{O}(|V|^2), \mathcal{O}(|E| \log |E|)\}$.
- Lower bound of $\mathcal{P}$: use mathematical method to prove 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:

$$\begin{aligned} & (x_1 \lor x_2 \lor x_3) \\ \land \; & (-x_1) \\ \land \; & (-x_2) \end{aligned}$$

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: $i = choice(1, 2)$;
  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

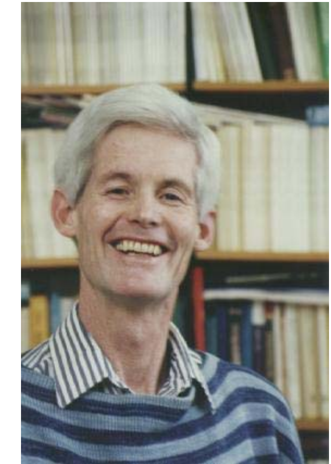
- **Are all problems NP problems?**
  - Halting problem: given an arbitrary program with an arbitrary input data, will the program terminate or not?
  - Halting problem is not an NP problem because it is undecidable.
- **Are all NP problems P problems?** (**P = NP ?**)
  - P $\subseteq$ NP ? (**yes**)
  - 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).

Stephen Cook

- The author of the famous paper [a] introducing the theory of NP completeness
- The 1982 recipient of the Turing award

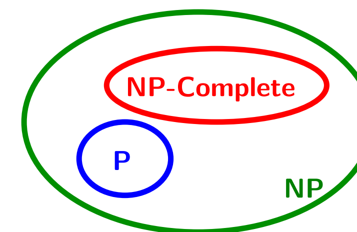


[a] The Complexity of Theorem Proving Procedures. Proceedings Third Annual ACM Symposium on Thoery of Computing, May 1971, pp 151-158

Alan Turing (1912–1954, England)

- Turing Award (accompanied by \$100,000) is given to an individual with contributions that are of lasting and major technical importance to the computer field.

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.

The millennium prize problems

- Seven problems considered to be "important classic questions that have resisted solution over the years"
  1. P versus NP
  2. The Hodge Conjecture
  3. The Poincare' Conjecture
  4. The Riemann Hypothesis
  5. Yang-Mills Existence and Mass Gap
  6. Navier-Stokes Existence and Smoothness
  7. The Birch and Swinnerton-Dyer Conjecture
- The first person to solve each problem above will be awarded \$1,000,000 by CMI.

Halting problem

- Given an arbitrary program with an arbitrary input data, will the program terminate or not?
  - NP-hard: SAT $\propto$ halting problem

    Algorithm $\mathcal{A}(F)$ (F is a Boolean formula)
    for each a among 2^n possible assignments **do**
        if a satisfies F **then**
            $\mathcal{A}$ stops;
        end for
    $\mathcal{A}$ enters an infinite loop;

    If algorithm $\mathcal{A}$ stops, F is satisfiable; otherwise, F is unsatisfiable.

Halting problem

- Undecidable: no algorithm can solve this problem (proved by Alan Turing, 1936)
  - $Halt(A, I) = \begin{cases} 1, & \text{if } A(I) \text{ halts} \\ 0, & \text{otherwise} \end{cases}$
  - Algorithm $T(A)$
    if $Halt(A, A) = 1$ (i.e., A halts)
    then enter infinite loop, **else** halt.

1. If $T(T)$ halts, then $Halt(T, T) = 0$, which means that $T(T)$ does not halt.
2. If $T(T)$ does not halt, then $Halt(T, T) = 1$, which means that $T(T)$ halts.