Algorithms vs. Problems

An **algorithm** is a computational procedure that takes values as *input* and produces values as *output*, in order to solve a well defined computational **problem**.

- The statement of the problem specifies a desired relationship between the input and the output.
- The algorithm specifies how to achieve that relationship.
- A particular value of the input corresponds to an instance of the problem.

So far, we have analyzed the time complexity of various algorithms. In this section, we switch focus from algorithms to problems.

P vs. NP vs. NP-complete

Modern Complexity Theory

P $\equiv$ Class of problems that are generally considered to have feasible (tractable) solutions.

P $\equiv$ Class of problems that are solvable in polynomial time

- Polynomial Time: there are two positive constants c and k, and an algorithm A that solves the problem, and whose worst case running time is at most cn^k (n is the input size).
- These are the problems that have (somewhat) efficient solutions.
- Most of the problems that we've examined so far fall into this class.

P vs. NP vs. NP-complete

Modern Complexity Theory

NP $\equiv$ Class of problems that are verifiable in polynomial time.

- If an oracle gives us a certificate corresponding to a solution, then there is an algorithm that can verify it in polynomial time.

NP-Complete $\equiv$ A subset of problems from NP such that:

- No tractable algorithm has been found to solve them.
- For any two problems A $\in$ NPC and B $\in$ NP, B can be reduced in polynomial time to A.

Class P: Examples

- Matrix Multiplication
- Sorting
- Selection
- MST
- APSP
- Bipartite Matching

All these problems have efficient solutions. If we let n be the size of the input to the problem, then there is an algorithm that runs in time $O(n^k)$, where k is some small number.

Class P: Examples

Matrix multiplication	$O(n^{\lg 7})$
Sorting	$O(n \lg n) = O(n^{1+\epsilon})$
Selection	$O(n)$
MST	$O(E \lg V)$
APSP	$O(V^3)$
Bipartite Matching	$O(VE)$

Input Size and Encodings

Q: What do we mean by input size n?

A: Any problem instance can be **encoded** as a sequence of 0's and 1's (i.e. represented as a binary string):

- For example, an input instance for the sorting problem can be $\{1, 5, 7, 2\}$, which can be encoded into 1, 101, 111, 10, hence the input size is 9.

Q: Does the encoding scheme matter?

A: Only unary encoding will make a difference. For example, compare $\log_2 n$ vs. $\log_3 n$, with $\log_2 n$ vs. n.

Tractable vs. Intractable problems

- **Tractable:** a problem which can be solved using an algorithm whose (*worst-case*) running time is a *polynomial* function of its input size.

- **Intractable:** a problem which is **impossible** to be solved using a polynomial time algorithm (i.e., its worst case cannot be bounded by a polynomial function of its input size).

- **Comments:**
  - Note that **intractability** is a property of the **problem**.
  - An algorithm whose worse case is not a polynomial does not make a problem intractable.
  - We have to show that no polynomial algorithm exists!

Advantages of this Definition of Tractable

Q: Why is this definition advantageous?

A: The polynomials are closed under addition, multiplication and composition:

- So if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial.
- A constant number of calls to polynomial-time subroutines will still result in a polynomial-time algorithm.

Note: Most problems of practical importance that are solvable in polynomial-time have algorithms that run in time $O(n^k)$ where k is a small integer ($k \leq 5$).

Polynomial vs. Exponential

Suppose one basic operation needs CPU time 0.000001 second.

	10	20	30	40	50	60
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s	0.00006 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.016 s	0.025 s	0.036 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s	0.125 s	0.216 s
n^5	0.1 s	3.2 s	24.3 s	1.7 min	5.2 min	13.0 min
2^n	0.001 s	1.0 s	17.9 min	12.7 days	35.7 years	366 cent
3^n	0.59 s	58 min	6.5 years	3855 cent	2×10^8 cent	1.3×10^13 cent

Limitations of this Definition of Tractable

There are some disadvantages to this definition of "tractable":

- An algorithm that runs in time $\Theta(n^{1000})$ is by no means feasible *(though in practice, it is likely that an algorithm with a much better running time will be discovered soon for the same problem)*.

- An algorithm that runs in $cn \lg n$ steps, for some very large c is not going to be a practical algorithm.

Is everything in the set P?

There are problems that are known to be unsolvable (e.g. the Halting Problem).

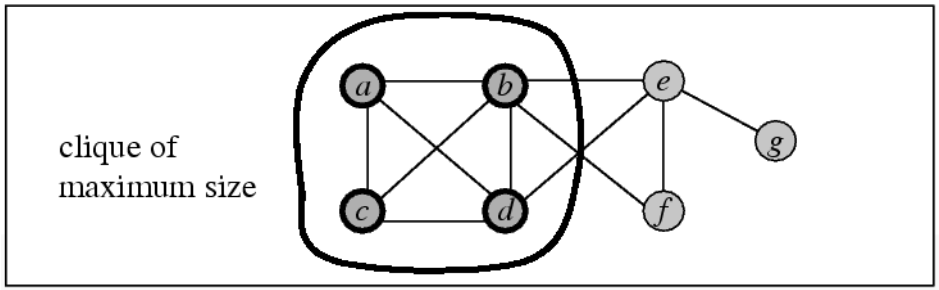
There are also problems that are solvable, but for which no efficient solution is known:

- The best known algorithms for some of these problems have running times of $O(2^n)$.
- For some of these problems, there is strong evidence that they do not have efficient solutions.

The Clique Problem: Optimization vs. Decision

- Definition: Let $G = (V, E)$ be an undirected graph. A **clique** of G is a set $V' \subseteq V$ such that $\forall u, v \in V'$, we have $(u, v) \in E$, i.e., **clique** is a complete subgraph of G.

- **Optimization Problem**: Given G, find the size of the largest clique.



- **Decision Problem**: Given G and k, is there a clique of size k in the graph G?
  CLIQUE $= \{< G, k >: G$ is a graph with a clique of size $k\}$

A Brute Force Algorithm

```
Largest-Clique(G)
{
    for j := |V| to 1 do
        check if any subset of j vertices is a clique.
        if a clique was found then
            return j
}
```

Time complexity $= \Omega(2^{|V|}) \Rightarrow$ very slow!

Q: Is the Clique problem tractable?

A: We do not know. But it is very likely that it does not have a polynomial time solution.

Three general categories of problems

1) Problems for which polynomial-time algorithms have been found:
   - Sorting in $O(nlgn)$.
   - APSP in $O(n^3)$.
   - ...
2) Problems that have been proven to be intractable:
   - Non-polynomial: generating all permutations of n elements is $\Omega(2^n)$.
   - Undecidable: the **Halting Problem** (Turing, 1936).

Three general categories of problems

3) Problems that have not been proven to be intractable, but for which a polynomial-time algorithm has never been found:

   - Traveling Salesperson Problem
   - Subset Sum Problem
   - Graph Coloring
   - Hamiltonian Cycle
   - ...

   Primality Test used to be in this category until 2002, when Agrawal-Kayal-Saxena from IIT Kanpur proved that *PRIMES is in P*.

Why should we care about category 3?

These hard problems really come up all the time. Recognizing a problem is hard makes you stop hitting your head again a wall trying to solve it, and do something more feasible:

- Solve the problem approximately instead of exactly; instead of finding the optimal solution, finding a near-optimal solution is often good enough in practice.

- Use an exponential time solution anyway; if you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution, especially if the size if the input is small.

Decision problems

Decision problems are those problems whose output is either **yes** or **no**. We will carry out discussions based only on decision problems:

- A unified way to treat all problems.
- The optimization and decision versions of the same problem are usually in the same complexity class.

Decision problems vs. Optimization problems

- Clique Problem:
  - **Optimization Problem**: Given G, find the size of the largest clique.
  - **Decision problem**: Given $< G, k >$, G is an undirected graph and k is an integer, does G have a clique of size k?
- Shortest-Path Problem:
  - **Optimization Problem**: Given graph G and nodes u, v, find the shortest path from u to v.
  - **Decision problem**: Given $< G, u, v, k >$, is there a path from u to v of length $\leq k$?

Optimization problems vs. Decision problems

Theorem: optimization problem solvable in polynomial time $\Leftrightarrow$ decision problem solvable in polynomial time.

- **Proof:** (Optimization $\Rightarrow$ Decision):
  - The optimization problem can be solved in $O(n^c)$ time, where c is a constant.
  - Let M be the answer to the optimization problem (e.g. M is the size of the maximum clique existing in G).
  - To solve the decision problem, simply compare M with k; if $M \geq k$, then the answer is "YES"; otherwise "NO".
  - Comparison takes constant time, so the decision problem can be solved also in polynomial time: $O(n^c) + \Theta(1)$.

Optimization problems vs. Decision problems

- **Proof:** (Decision $\Rightarrow$ Optimization):
  - The decision problem can be solved in $O(n^c)$ time, namely, for a given k, it takes $O(n^c)$ time to output YES or NO.
  - To solve the optimization problem, try all different ks, starting from n all the way down to 1. The first k that produces "YES" to the decision problem tells the answer to the optimization problem (e.g. the size of the maximum clique in the graph).
  - The complexity is $n \times O(n^c) = O(n^{1+c})$, still polynomial.

A More Natural Decision Problem

SAT: The satisfiability problem.

Input: A boolean formula α consisting of variables, $x_1, \ldots, x_n$, (,), $\vee$, $\wedge$, $\neg$.
Ex.:

$$\alpha = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2).$$

Problem: Does α have a satisfying assignment , i.e. is there a setting of the variables in α so that α evaluates to TRUE?

In the example above, the answer is YES. To see this, let $x_1 =$ TRUE and let $x_2 =$ TRUE.

The class NP

Definition: NP is the set of all decision problems that are verifiable in polynomial time.

- A problem is in NP if we can test, in polynomial time, whether a solution is correct (without worrying about how hard it might be to find the solution).
- To prove that a problem belongs to NP, all we need to do is to demonstrate that:
  - If an **oracle** gives us a **certificate** for a solution to the problem,
  - Then we can verify the certificate in polynomial time (i.e. we can show that it indeed corresponds to a solution to the problem).

NP example

- CLIQUE $\in$ NP.
- Proof:

  For a given graph $G = (V, E)$ and a number k, an oracle gives us a set $V' \subseteq V$ saying it makes a clique. Checking whether V' is a clique can be accomplished in $O(|V'|^2)$ time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

Is everything in NP?

The **Halting Problem:**

- Given a description of a program and a finite input,
- Decide whether the program finishes running, or will run forever, given that input.

Turing proved in 1936 that there is no algorithm that can solve the halting problem for all possible pairs $\langle program, input \rangle$.

A Formal Language framework

Definition: An alphabet Σ is a finite set of symbols.

Example: $\Sigma = \{0, 1\}$.

Definition: A language L over Σ is a set of strings made up of symbols from Σ.

Example: $L = \{10, 11, 101, 111, ...\}$ is the language of binary representations of prime numbers.

- ϵ denotes the empty string.
- $\emptyset$ denotes the empty language.
- Σ^* denotes the language of all strings over Σ. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\epsilon, 0, 1, 00, 01, ...\}$

A Formal Language framework: Algorithms

An algorithm for a decision problem can be viewed in terms of the language that it decides:

- The language **accepted** by an algorithm A is $L = \{x \in \{0,1\}^* | A(x) = 1\}$
- The language **rejected** by an algorithm A is $L = \{x \in \{0,1\}^* | A(x) = 0\}$
- A language L is **decided** by an algorithm A if every binary string in L is accepted by A, and every binary string not in L is rejected by A.

A Formal Language Framework: Problems

A decision problem can be viewed as a language-recognition problem:

- Let U be the set of all possible inputs to the decision problem and $L \subseteq U$ be the set of all inputs (i.e. the language) for which the answer to the problem is YES (i.e. the symbol 1).
- The decision problem is to recognize whether a given input belongs to L.
- We call L the language corresponding to the decision problem.

Decision problems and corresponding languages

The decision problem PRIME: given a number z, is z a prime number?

- Let U be the all possible inputs, i.e., $\{1, 2, 3, 4, 5, 6, ...\}$, or $\{1, 10, 11, 100, 101, 110, ...\}$ under binary encoding.
- L is $\{2, 3, 5, ...\}$ or $\{1, 10, 11, 101, ...\}$.
- The decision problem PRIME can be fully characterized by the set L.
- PRIME is to recognize whether a given input z belongs to L.
- $\underline{\text{PRIME} = \{z \in \{0,1\}^* | z \text{ is prime } \}}$.

Decision problems and corresponding languages

The decision problem CLIQUE: Given G and an integer k, is there a clique size of k in G?

- The corresponding language L is the set of binary encodings of all the graphs G that have a clique of size k
- CLIQUE $= \{\langle G, k\rangle : G \text{ has a clique of size of } k\}$.

A Formal Language Framework: Class P

An alternative definition of the complexity class P:

$P = \{L \subseteq \{0,1\}^* |$ there exists an algorithm A that decides L in polynomial time$\}$

Theorem 34.2:

$P = \{L \subseteq \{0,1\}^* |$ there exists an algorithm A that accepts L in polynomial time$\}$

A Formal Language Framework: Class NP

An alternative definition of the complexity class NP:

A language L belongs to NP iff there exist a two-input polynomial-time algorithm A and constant c such that:

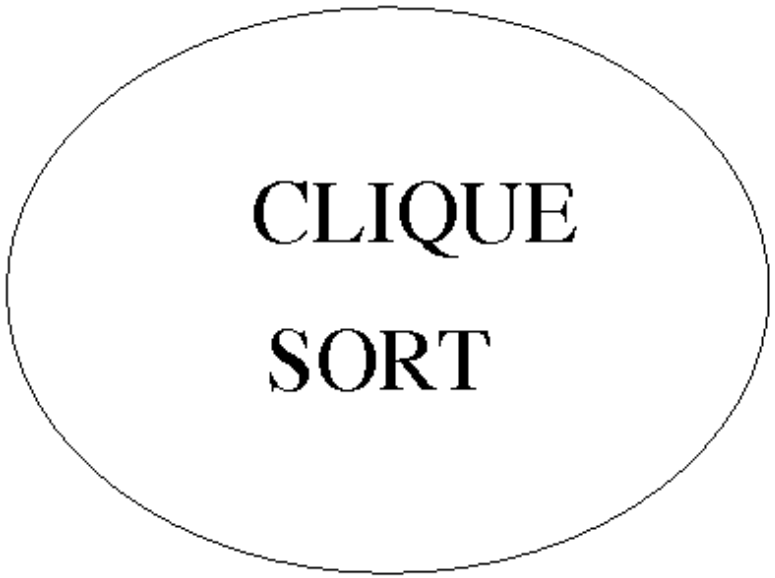
- $L = \{x \in \{0,1\}^* |$ there exists a certificate y with $|y| = O(|x|^c)$ such that $A(x,y) = 1\}$
- We say that A **verifies** language L in polynomial time.

P versus NP

- $P \subseteq NP$.
  - We can ignore the certificate and solve a P-problem in polynomial time. In other words, for a given input instance, we can check whether it belongs to the language corresponding to the decision problem in poly-time, without looking at the proposed solution certificate.
- $P = NP$?
  - The most famous open question in CS.
  - US$1,000,000 prize for the first person to provide an answer.

Two possible worlds

P = NP



P != NP