

# Chapter 1

## Administrivia, Introduction

CS 573: Algorithms, Fall 2013

August 27, 2013

### 1.0.0.1 The word “algorithm” comes from...

Muhammad ibn Musa al-Khwarizmi

780-850 AD

The word “algebra” is taken from the title of one of his books.

## 1.1 Administrivia

### 1.1.0.2 Online resources

- (A) **Webpage:** [courses.engr.illinois.edu/cs573/fa2013/](http://courses.engr.illinois.edu/cs573/fa2013/)  
General information, homeworks, etc.
- (B) **Moodle:** Quizzes, solutions to homeworks.
- (C) **Online questions/announcements:** Piazza  
Online discussions, etc.

### 1.1.0.3 Textbooks

- (A) **Prerequisites:** CS 173 (discrete math), CS 225 (data structures) and CS 373 (theory of computation)
- (B) **Recommended books:**
  - (A) Algorithms by Dasgupta, Papadimitriou & Vazirani.  
Available online for free!
  - (B) Algorithm Design by Kleinberg & Tardos
- (C) **Lecture notes:** Available on the web-page before/during/after every class.
- (D) **Additional References**
  - (A) Previous class notes of Jeff Erickson, Sarel Har-Peled and the instructor.
  - (B) Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein.
  - (C) Computers and Intractability: Garey and Johnson.

### 1.1.0.4 Prerequisites

- (A) **Asymptotic notation:**  $O()$ ,  $\Omega()$ ,  $o()$ .
- (B) **Discrete Structures:** sets, functions, relations, equivalence classes, partial orders, trees, graphs
- (C) **Logic:** predicate logic, boolean algebra
- (D) **Proofs: by induction, by contradiction**
- (E) **Basic sums and recurrences:** sum of a geometric series, unrolling of recurrences, basic calculus
- (F) **Data Structures:** arrays, multi-dimensional arrays, linked lists, trees, balanced search trees, heaps

- (G) **Abstract Data Types:** lists, stacks, queues, dictionaries, priority queues
- (H) **Algorithms:** sorting (merge, quick, insertion), pre/post/in order traversal of trees, depth/breadth first search of trees (maybe graphs)
- (I) **Basic analysis of algorithms:** loops and nested loops, deriving recurrences from a recursive program
- (J) **Concepts from Theory of Computation:** languages, automata, Turing machine, undecidability, non-determinism
- (K) **Programming:** in some general purpose language
- (L) **Elementary Discrete Probability:** event, random variable, independence
- (M) **Mathematical maturity**

### 1.1.0.5 Homeworks

- (A) One quiz every 1-2-3 weeks: Due by midnight on Sunday.
- (B) One homework every 1-2-3 weeks.
- (C) Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
  - (A) Short quiz-style questions to be answered individually on *Moodle*.
- (D) Groups can be changed a *few* times only.

### 1.1.0.6 More on Homeworks

- (A) No extensions or late homeworks accepted.
- (B) To compensate, the homework with the least score will be dropped in calculating the homework average.
- (C) **Important:** Read homework faq/instructions on website.

### 1.1.0.7 Advice

- (A) Attend lectures, please ask plenty of questions.
- (B) Clickers...
- (C) Attend discussion sessions.
- (D) Don't skip homework and don't copy homework solutions.
- (E) Study regularly and keep up with the course.
- (F) Ask for help promptly. Make use of office hours.

### 1.1.0.8 Homeworks

- (A) HW 0 is posted on the class website. Quiz 0 available
- (B) HW 0 to be submitted in individually.

## 1.2 Course Goals and Overview

### 1.2.0.9 Topics

- (A) Some fundamental algorithms
- (B) Broadly applicable techniques in algorithm design
  - (A) Understanding problem structure
  - (B) Brute force enumeration and backtrack search
  - (C) Reductions
  - (D) Recursion
    - (A) Divide and Conquer
    - (B) Dynamic Programming
  - (E) Greedy methods
  - (F) Network Flows and Linear/Integer Programming (optional)
- (C) Analysis techniques
  - (A) Correctness of algorithms via induction and other methods

- (B) Recurrences
- (C) Amortization and elementary potential functions
- (D) Polynomial-time Reductions, NP-Completeness, Heuristics

#### 1.2.0.10 Goals

- (A) Algorithmic thinking
- (B) Learn/remember some basic tricks, algorithms, problems, ideas
- (C) Understand/appreciate limits of computation (intractability)
- (D) Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
- (E) Have fun!!!

## 1.3 Algorithms and efficiency

## 1.4 Primality Testing

### 1.4.0.11 Primality testing

Problem Given an integer  $N > 0$ , is  $N$  a prime?

**SimpleAlgorithm:**

```

for  $i = 2$  to  $\lfloor \sqrt{N} \rfloor$  do
    if  $i$  divides  $N$  then
        return ‘‘COMPOSITE’’
return ‘‘PRIME’’

```

Correctness? If  $N$  is composite, at least one factor in  $\{2, \dots, \sqrt{N}\}$

Running time?  $O(\sqrt{N})$  divisions? Sub-linear in input size! **Wrong!**

### 1.4.1 Primality testing

#### 1.4.1.1 ...Polynomial means... in input size

How many bits to represent  $N$  in binary?  $\lceil \log N \rceil$  bits.

Simple Algorithm takes  $\sqrt{N} = 2^{(\log N)/2}$  time.

*Exponential* in the input size  $n = \log N$ .

- (A) Modern cryptography: binary numbers with 128, 256, 512 bits.
- (B) Simple Algorithm will take  $2^{64}$ ,  $2^{128}$ ,  $2^{256}$  steps!
- (C) Fastest computer today about 3 petaFlops/sec:  $3 \times 2^{50}$  floating point ops/sec.

**Lesson:** Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

#### 1.4.1.2 Efficient algorithms

So, is there an *efficient/good/effective* algorithm for primality?

**Question:** What does efficiency mean?

In this class *efficiency* is broadly equated to *polynomial time*.

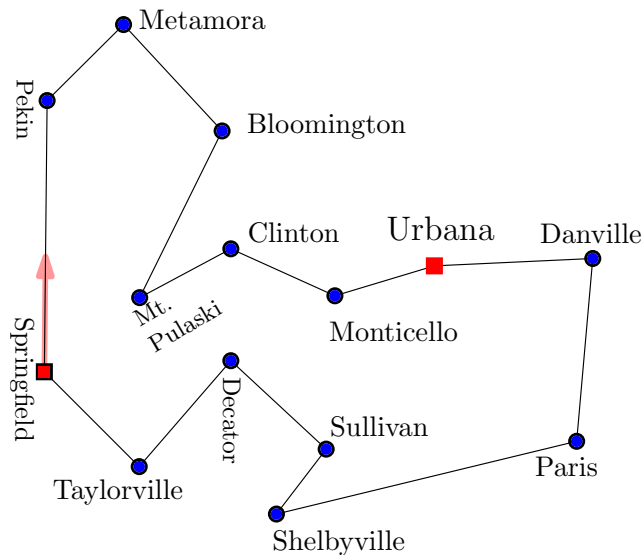
$O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(n^3)$ ,  $O(n^{100})$ , ... where  $n$  is size of the input.

Why? Is  $n^{100}$  really efficient/practical? Etc.

Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

## 1.4.2 TSP problem

### 1.4.2.1 Lincoln's tour



- (A) Circuit court - ride through counties staying a few days in each town.
- (B) Lincoln was a lawyer traveling with the Eighth Judicial Circuit.
- (C) Picture: travel during 1850.
  - (A) Very close to optimal tour.
  - (B) Might have been optimal at the time..

## 1.4.3 Solving TSP by a Computer

### 1.4.3.1 Is it hard?

- (A)  $n$  = number of cities.
- (B)  $n^2$ : size of input.
- (C) Number of possible solutions is

$$n * (n - 1) * (n - 2) * \dots * 2 * 1 = n!$$

- (D)  $n!$  grows very quickly as  $n$  grows.
  - $n = 10$ :  $n! \approx 3628800$
  - $n = 50$ :  $n! \approx 3 * 10^{64}$
  - $n = 100$ :  $n! \approx 9 * 10^{157}$

## 1.4.4 Solving TSP by a Computer

### 1.4.4.1 Fastest computer...

- (A) Fastest super computer can do (roughly)

$$2.5 * 10^{15}$$

operations a second.

(B) Assume: computer checks  $2.5 * 10^{15}$  solutions every second, then...

(A)  $n = 20 \implies 2$  hours.

(B)  $n = 25 \implies 200$  years.

(C)  $n = 37 \implies 2 * 10^{20}$  years!!!

## 1.4.5 What is a good algorithm?

### 1.4.5.1 Running time...

Input size	$n^2$ ops	$n^3$ ops	$n^4$ ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
100	0 secs	0 secs	0 secs	never
8000	0 secs	0 secs	1 secs	never
16000	0 secs	0 secs	26 secs	never
32000	0 secs	0 secs	6 mins	never
64000	0 secs	0 secs	111 mins	never
200,000	0 secs	3 secs	7 days	never
2,000,000	0 secs	53 mins	202.943 years	never
$10^8$	4 secs	12.6839 years	$10^9$ years	never
$10^9$	6 mins	12683.9 years	$10^{13}$ years	never

## 1.4.6 What is a good algorithm?

### 1.4.6.1 Running time...



## 1.4.7 Primality

### 1.4.7.1 Primes is in $P$ !

**Theorem 1.4.1 (Agrawal-Kayal-Saxena'02).** *There is a polynomial time algorithm for primality.*

First polynomial time algorithm for testing primality. Running time is  $O(\log^{12} N)$  further improved to about  $O(\log^6 N)$  by others. In terms of input size  $n = \log N$ , time is  $O(n^6)$ .

### 1.4.7.2 What about before 2002?

Primality testing a key part of cryptography. What was the algorithm being used before 2002?

Miller-Rabin *randomized* algorithm:

- (A) runs in polynomial time:  $O(\log^3 N)$  time
- (B) if  $N$  is prime correctly says “yes”.
- (C) if  $N$  is composite it says “yes” with probability at most  $1/2^{100}$  (can be reduced further at the expense of more running time).

Based on Fermat's little theorem and some basic number theory.

## 1.4.8 Factoring

### 1.4.8.1 Factoring

- (A) Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
- (B) Relies on the difficulty of factoring a composite number into its prime factors.
- (C) There is a polynomial time algorithm that decides whether a given number  $N$  is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

Lesson Intractability can be useful!

## 1.5 Model of Computation

### 1.5.0.2 Unit-Cost RAM Model

Informal description:

- (A) Basic data type is an integer/floating point number
- (B) Numbers in input fit in a word
- (C) Arithmetic/comparison operations on words take constant time
- (D) Arrays allow random access (constant time to access  $A[i]$ )
- (E) Pointer based data structures via storing addresses in a word

### 1.5.0.3 Example

Sorting: input is an array of  $n$  numbers

- (A) input size is  $n$  (ignore the bits in each number),
- (B) comparing two numbers takes  $O(1)$  time,
- (C) random access to array elements,
- (D) addition of indices takes constant time,
- (E) basic arithmetic operations take constant time,
- (F) reading/writing one word from/to memory takes constant time.

We will usually not allow (or be careful about allowing):

- (A) bitwise operations (and, or, xor, shift, etc).
- (B) floor function.
- (C) limit word size (usually assume unbounded word size).

#### 1.5.0.4 Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in several important situations so one has to be careful.

- (A) For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two  $n$ -digit numbers, primality etc.
- (B) Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by  $2^k$  where  $k$  is word length.
- (C) Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.

#### 1.5.0.5 Models used in class

In this course:

- (A) Assume unit-cost **RAM** by default.
- (B) We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.





# Part I

## Reductions

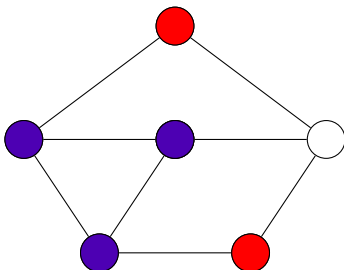


## 1.6 Independent Set and Clique

### 1.6.0.6 Independent Sets and Cliques

Given a graph  $G$ , a set of vertices  $V'$  is:

- (A) An **independent set**: if no two vertices of  $V'$  are connected by an edge of  $G$ .
- (B) **clique**: every pair of vertices in  $V'$  is connected by an edge of  $G$ .



### 1.6.0.7 The Independent Set and Clique Problems

#### Independent Set

**Instance:** A graph  $G$  and an integer  $k$ .

**Question:** Does  $G$  has an independent set of size  $\geq k$ ?

#### Clique

**Instance:** A graph  $G$  and an integer  $k$ .

**Question:** Does  $G$  has a clique of size  $\geq k$ ?

### 1.6.0.8 Types of Problems

Decision, Search, and Optimization

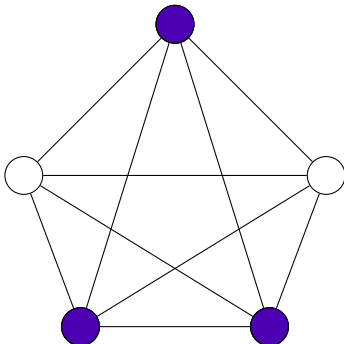
- (A) **Decision problem**. Example: given  $n$ , **is**  $n$  prime?.
- (B) **Search problem**. Example: given  $n$ , **find** a factor of  $n$  if it exists.
- (C) **Optimization problem**. Example: find the **smallest** prime factor of  $n$ .

### 1.6.0.9 Reducing Independent Set to Clique

An instance of **Independent Set** is a graph  $G$  and an integer  $k$ .

Convert  $G$  to  $\overline{G}$ , in which  $(u, v)$  is an edge iff  $(u, v)$  is **not** an edge of  $G$ . ( $\overline{G}$  is the *complement* of  $G$ .)

We use  $\overline{G}$  and  $k$  as the instance of **Clique**.



### 1.6.0.10 Independent Set and Clique

(A) **Independent Set**  $\leq$  **Clique**.

What does this mean?

(B) If we have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.

(C) **Clique** is *at least as hard as* **Independent Set**.

(D) Also... **Independent Set** is *at least as hard as* **Clique**.

### 1.6.0.11 Reductions, revised.

For decision problems  $X, Y$ , a *reduction from  $X$  to  $Y$*  is:

(A) An algorithm ...

(B) Input:  $I_X$ , an instance of  $X$ .

(C) Output:  $I_Y$  an instance of  $Y$ .

(D) Such that:

$$\boxed{I_Y \text{ is YES instance of } Y} \iff \boxed{I_X \text{ is YES instance of } X}$$

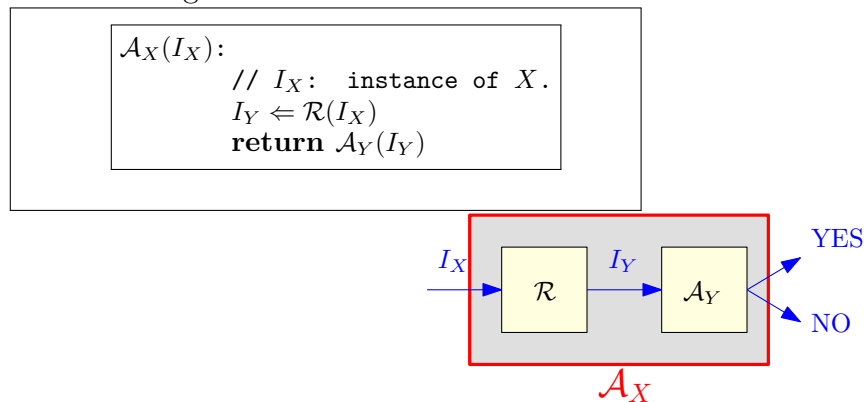
(Actually, this is only one type of reduction, but this is the one we'll use most often.)

### 1.6.0.12 Using reductions to solve problems

(A)  $\mathcal{R}$ : Reduction  $X \rightarrow Y$

(B)  $\mathcal{A}_Y$ : algorithm for  $Y$ :

(C)  $\implies$  New algorithm for  $X$ :



In particular, if  $\mathcal{R}$  and  $\mathcal{A}_Y$  are polynomial-time algorithms,  $\mathcal{A}_X$  is also polynomial-time.

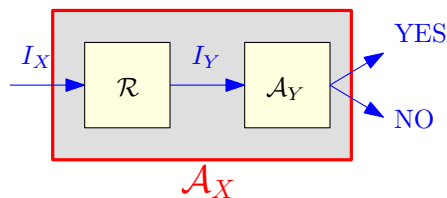
### 1.6.0.13 Comparing Problems

(A) Reductions allow us to formalize the notion of “Problem  $X$  is no harder to solve than Problem  $Y$ ”.

(B) If Problem  $X$  **reduces to** Problem  $Y$  (we write  $X \leq Y$ ), then  $X$  cannot be harder to solve than  $Y$ .

(C) More generally, if  $X \leq Y$ , we can say that  $X$  is no harder than  $Y$ , or  $Y$  is at least as hard as  $X$ .

### 1.6.0.14 Polynomial-time reductions



- (A) Algorithm is **efficient** if it runs in polynomial-time.
- (B) Interested only in **polynomial-time reductions**.
- (C)  $X \leq_P Y$ : Have polynomial-time reduction from problem  $X$  to problem  $Y$ .
- (D)  $\mathcal{A}_Y$ : poly-time algorithm for  $Y$ .
- (E)  $\implies$  Polynomial-time/efficient algorithm for  $X$ .

### 1.6.0.15 Polynomial-time reductions and hardness

**Lemma 1.6.1.** *For decision problems  $X$  and  $Y$ , if  $X \leq_P Y$ , and  $Y$  has an efficient algorithm,  $X$  has an efficient algorithm.*

- (A) **Independent Set**: “believe” there is no efficient algorithm.
- (B) What about **Clique**?
- (C) Showed: **Independent Set**  $\leq_P$  **Clique**.
- (D) If **Clique** had an efficient algorithm, so would **Independent Set**!

**Observation 1.6.2.** *If  $X \leq_P Y$  and  $X$  does not have an efficient algorithm,  $Y$  cannot have an efficient algorithm!*

### 1.6.0.16 Polynomial-time reductions and instance sizes

**Proposition 1.6.3.**  $\mathcal{R}$ : a polynomial-time reduction from  $X$  to  $Y$ .

*Then, for any instance  $I_X$  of  $X$ , the size of the instance  $I_Y$  of  $Y$  produced from  $I_X$  by  $\mathcal{R}$  is polynomial in the size of  $I_X$ .*

*Proof:*  $\mathcal{R}$  is a polynomial-time algorithm and hence on input  $I_X$  of size  $|I_X|$  it runs in time  $p(|I_X|)$  for some polynomial  $p()$ .

$I_Y$  is the output of  $\mathcal{R}$  on input  $I_X$ .

$\mathcal{R}$  can write at most  $p(|I_X|)$  bits and hence  $|I_Y| \leq p(|I_X|)$ . ■

**Note:** Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

### 1.6.0.17 Polynomial-time Reduction

**Definition 1.6.4.** A **polynomial time reduction** from a decision problem  $X$  to a decision problem  $Y$  is an algorithm  $\mathcal{A}$  such that:

- (A) Given an instance  $I_X$  of  $X$ ,  $\mathcal{A}$  produces an instance  $I_Y$  of  $Y$ .
- (B)  $\mathcal{A}$  runs in time polynomial in  $|I_X|$ . This implies that  $|I_Y|$  (size of  $I_Y$ ) is polynomial in  $|I_X|$ .
- (C) Answer to  $I_X$  YES iff answer to  $I_Y$  is YES.

**Proposition 1.6.5.** *If  $X \leq_P Y$  then a polynomial time algorithm for  $Y$  implies a polynomial time algorithm for  $X$ .*

This is a **Karp reduction**.

### 1.6.0.18 Transitivity of Reductions

**Proposition 1.6.6.**  $X \leq_P Y$  and  $Y \leq_P Z$  implies that  $X \leq_P Z$ .

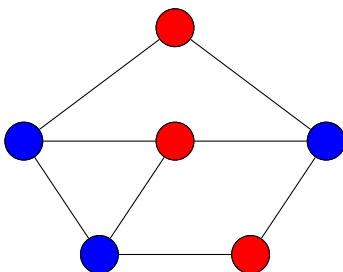
- (A) **Note:**  $X \leq_P Y$  does not imply that  $Y \leq_P X$  and hence it is very important to know the FROM and TO in a reduction.
- (B) To prove  $X \leq_P Y$  you need to show a reduction FROM  $X$  TO  $Y$
- (C) ...show that an algorithm for  $Y$  implies an algorithm for  $X$ .

## 1.7 Independent Set and Vertex Cover

### 1.7.0.19 Vertex Cover

Given a graph  $G = (V, E)$ , a set of vertices  $S$  is:

(A) A **vertex cover** if every  $e \in E$  has at least one endpoint in  $S$ .



### 1.7.0.20 The Vertex Cover Problem

**Problem 1.7.1 (Vertex Cover).**

**Input:** A graph  $G$  and integer  $k$ .

**Goal:** Is there a vertex cover of size  $\leq k$  in  $G$ ?

Can we relate **Independent Set** and **Vertex Cover**?

## 1.7.1 Relationship between...

### 1.7.1.1 Vertex Cover and Independent Set

**Proposition 1.7.2.** Let  $G = (V, E)$  be a graph.  $S$  is an independent set if and only if  $V \setminus S$  is a vertex cover.

*Proof:*

( $\Rightarrow$ ) Let  $S$  be an independent set

(A) Consider any edge  $uv \in E$ .

(B) Since  $S$  is an independent set, either  $u \notin S$  or  $v \notin S$ .

(C) Thus, either  $u \in V \setminus S$  or  $v \in V \setminus S$ .

(D)  $V \setminus S$  is a vertex cover.

( $\Leftarrow$ ) Let  $V \setminus S$  be some vertex cover:

(A) Consider  $u, v \in S$

(B)  $uv$  is not an edge of  $G$ , as otherwise  $V \setminus S$  does not cover  $uv$ .

(C)  $\Rightarrow S$  is thus an independent set. ■

### 1.7.1.2 Independent Set $\leq_P$ Vertex Cover

(A)  $G$ : graph with  $n$  vertices, and an integer  $k$  be an instance of the **Independent Set** problem.

(B)  $G$  has an independent set of size  $\geq k$  iff  $G$  has a vertex cover of size  $\leq n - k$

(C)  $(G, k)$  is an instance of **Independent Set**, and  $(G, n - k)$  is an instance of **Vertex Cover** with the same answer.

(D) Therefore, **Independent Set**  $\leq_P$  **Vertex Cover**. Also **Vertex Cover**  $\leq_P$  **Independent Set**.

# 1.8 Vertex Cover and Set Cover

## 1.8.0.3 The Set Cover Problem

### Problem 1.8.1 (Set Cover).

**Input:** Given a set  $U$  of  $n$  elements, a collection  $S_1, S_2, \dots, S_m$  of subsets of  $U$ , and an integer  $k$ .

**Goal:** Is there a collection of at most  $k$  of these sets  $S_i$  whose union is equal to  $U$ ?

**Example 1.8.2.** Let  $U = \{1, 2, 3, 4, 5, 6, 7\}$ ,  $k = 2$  with

$$\begin{aligned} S_1 &= \{3, 7\} & S_2 &= \{3, 4, 5\} \\ S_3 &= \{1\} & S_4 &= \{2, 4\} \\ S_5 &= \{5\} & S_6 &= \{1, 2, 6, 7\} \end{aligned}$$

$\{S_2, S_6\}$  is a set cover

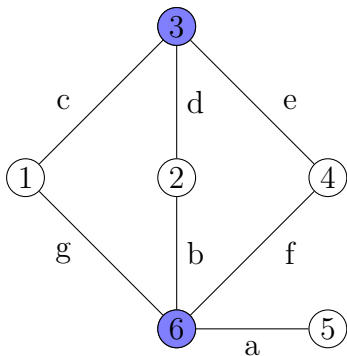
## 1.8.0.4 Vertex Cover $\leq_P$ Set Cover

Given graph  $G = (V, E)$  and integer  $k$  as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

- (A) Number  $k$  for the **Set Cover** instance is the same as the number  $k$  given for the **Vertex Cover** instance.
- (B)  $U = E$ .
- (C) We will have one set corresponding to each vertex;  $S_v = \{e \mid e \text{ is incident on } v\}$ .

Observe that  $G$  has vertex cover of size  $k$  if and only if  $U, \{S_v\}_{v \in V}$  has a set cover of size  $k$ . (Exercise: Prove this.)

## 1.8.0.5 Vertex Cover $\leq_P$ Set Cover: Example



Let  $U = \{a, b, c, d, e, f, g\}$ ,  $k = 2$  with

$$\begin{aligned} S_1 &= \{c, g\} & S_2 &= \{b, d\} \\ S_3 &= \{c, d, e\} & S_4 &= \{e, f\} \\ S_5 &= \{a\} & S_6 &= \{a, b, f, g\} \end{aligned}$$

$\{S_3, S_6\}$  is a set cover

$\{3, 6\}$  is a vertex cover

## 1.8.0.6 Proving Reductions

To prove that  $X \leq_P Y$  you need to give an algorithm  $\mathcal{A}$  that:

- (A) Transforms an instance  $I_X$  of  $X$  into an instance  $I_Y$  of  $Y$ .
- (B) Satisfies the property that answer to  $I_X$  is YES iff  $I_Y$  is YES.
  - (A) typical easy direction to prove: answer to  $I_Y$  is YES if answer to  $I_X$  is YES
  - (B) **typical difficult direction to prove:** answer to  $I_X$  is YES if answer to  $I_Y$  is YES (equivalently answer to  $I_X$  is NO if answer to  $I_Y$  is NO).
- (C) Runs in *polynomial* time.

### 1.8.0.7 Summary

We looked at **polynomial-time reductions**.

Using polynomial-time reductions

- (A) If  $X \leq_P Y$ , and we have an efficient algorithm for  $Y$ , we have an efficient algorithm for  $X$ .
- (B) If  $X \leq_P Y$ , and there is no efficient algorithm for  $X$ , there is no efficient algorithm for  $Y$ .

We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.