**Reductions**

---

**Desiderata**

Desiderata. Classify problems according to their computational requirements.

Desiderata'. Suppose we could (couldn’t) solve problem X efficiently. What else could (couldn’t) we solve efficiently?

---

**Reduction**

Def. Problem X reduces to problem Y if given a subroutine for Y, can solve X.

- Cost of solving $X = \text{cost of solving } Y + \text{cost of reduction}.$

Ex. $X =$ Euclidean MST, $Y =$ Voronoi.
Reduction

**Def.** Problem X reduces to problem Y if given a subroutine for Y, can solve X.

- Cost of solving X = cost of solving Y + cost of reduction.

**Consequences.**
- Classify problems: establish relative difficulty between two problems.
- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.

---

**Linear Time Reductions**

**Def.** Problem X linear reduces to problem Y if X can be solved with:
- Linear number of standard computational steps.
- One call to subroutine for Y.
- Notation: $X \leq_L Y$.

**Some familiar examples.**
- Median $\leq_L$ sorting.
- Element distinctness $\leq_L$ sorting.
- Closest pair $\leq_L$ Voronoi.
- Euclidean MST $\leq_L$ Voronoi.
- Arbitrage $\leq_L$ Negative cycle detection.
- Linear programming $\leq_L$ Linear programming in std form.

**Consequences.**
- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.
Shortest Paths on Graphs and Digraphs

**Claim.** Undirected shortest path (with nonnegative weights) linearly reduces to directed shortest path.

**Pf.** Replace each undirected edge by two directed edges.

**Caveat.** Reduction invalid in networks with negative weights (even if no negative cycles).

**Remark.** Can still solve shortest path problem in undirected graphs if no negative cycles, but need more sophisticated techniques.

Convex Hull and Sorting

**Sorting.** Given N distinct integers, rearrange them in ascending order.

**Convex hull.** Given N points in the plane, identify the extreme points of the convex hull (in counter-clockwise order).

**Claim.** Convex hull linear reduces to sorting.

**Pf.** Graham scan algorithm.

Linear Time Reductions

**Def.** Problem X linear reduces to problem Y if X can be solved with:
- Linear number of standard computational steps.
- One call to subroutine for Y.

**Consequences.**
- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.
3-SUM Reduces to 3-COLLINEAR

**3-SUM.** Given N distinct integers, are there three that sum to 0?

**3-COLLINEAR.** Given N distinct points in the plane, are there 3 that all lie on the same line?

**Claim.** $3\text{-SUM} \leq_L 3\text{-COLLINEAR}$.

**Conjecture.** Any algorithm for $3\text{-SUM}$ requires $\Omega(N^3)$ time.

**Corollary.** Sub-quadratic algorithm for $3\text{-COLLINEAR}$ unlikely.

Claim. $3\text{-SUM} \leq_L 3\text{-COLLINEAR}$.

- $3\text{-SUM}$ instance: $x_1, x_2, \ldots, x_N$
- $3\text{-COLLINEAR}$ instance: $(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_N, x_N^2)$
3-SUM Reduces to 3-COLLINEAR

Lemma. If a, b, and c are distinct then \( a + b + c = 0 \) if and only if \((a, a^3), (b, b^3), (c, c^3)\) are collinear.

Pf. Three points \((a, a^3), (b, b^3), (c, c^3)\) are collinear iff:

\[
\frac{c^3 - a^3}{c - a} = \frac{b^3 - c^3}{b - c} \iff \frac{a^3 - b^3}{a - b} = \frac{c^3 - a^3}{c - a} \iff c^3 + bc - a^3 - ab = 0 \iff (c-a)(c+a+b) = 0 \iff c = a \text{ or } a+b+c = 0
\]

\( c, b, a \) are collinear if \( c, b, a \) are distinct

not distinct

\begin{align*}
\text{denominators are nonzero if } a, b, c \text{ are distinct}
\end{align*}

\[\text{Lemma.} \quad \text{Given an integer } x \text{ (represented in binary), is } x \text{ prime?}
\]

\[\text{COMPOSITE.} \quad \text{Given an integer } x, \text{ does } x \text{ have a nontrivial factor?}
\]

Claim. \( \text{PRIME} \preceq L \text{ COMPOSITE} \).

```java
public static boolean isPrime(BigInteger x) {
    if (isComposite(x)) return false;
    else return true;
}
```

Primality and Compositeness

Linear Time Reductions

\[\text{Def.} \quad \text{Problem } X \text{ linear reduces to problem } Y \text{ if } X \text{ can be solved with:}
\]

- Linear number of standard computational steps.
- One call to subroutine for Y.

**Consequences.**

- Design algorithms: given algorithm for Y, can also solve X.
- Establish intractability: if X is hard, then so is Y.
- Classify problems: establish relative difficulty between two problems.

\[\text{PRIME} \quad \text{Given an integer } x \text{ (represented in binary), is } x \text{ prime?}
\]

\[\text{COMPOSITE} \quad \text{Given an integer } x, \text{ does } x \text{ have a nontrivial factor?}
\]

Claim. \( \text{COMPOSITE} \preceq L \text{ PRIME} \).

```java
public static boolean isComposite(BigInteger x) {
    if (isPrime(x)) return false;
    else return true;
}
```

Conclusion. \( \text{COMPOSITE} \) and \( \text{PRIME} \) have same complexity.
Caveat.
- System designer specs the interfaces for project.
- One programmer might implement `isComposite()` using `isPrime()`.
- Other programmer might implement `isPrime()` using `isComposite()`.
- Be careful to avoid infinite reduction loops in practice.

```java
public static boolean isComposite(BigInteger x) {
    if (isPrime(x)) return false;
    else return true;
}
```

```java
public static boolean isPrime(BigInteger x) {
    if (isComposite(x)) return false;
    else return true;
}
```

**Poly-Time Reduction**

**Def.** Problem X polynomial reduces to problem Y if arbitrary instances of problem X can be solved using:
- Polynomial number of standard computational steps, plus
- One call to subroutine for Y.

**Notation.** $X \preceq_p Y$.

**Ex.** Assignment problem $\preceq_p LP$.  
**Ex.** 3-SAT $\preceq_p$ 3-COLOR.  
**Ex.** Any linear reduction.

---

**Poly-Time Reductions**

**Goal.** Classify and separate problems according to relative difficulty.
- Those that can be solved in polynomial time.  
- Those that (probably) require exponential time.

**Establish tractability.** If $X \preceq_p Y$ and Y can be solved in poly-time, then X can be solved in poly-time.

**Establish intractability.** If $Y \preceq_p X$ and Y cannot be solved in poly-time, then X cannot be solved in poly-time.

**Transitivity.** If $X \preceq_p Y$ and $Y \preceq_p Z$ then $X \preceq_p Z$. 
Assignment Problem

Assignment problem. Assign $n$ jobs to $n$ machines to minimize total cost, where $c_{ij} =$ cost of assigning job $j$ to machine $i$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>13</td>
<td>11</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>13</td>
<td>12</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

$\text{cost} = 3 \times 10 + 11 + 20 + 9 = 53$

$\text{cost} = 8 + 7 + 20 + 8 + 11 = 44$

Applications. Match jobs to machines, match personnel to tasks, match Princeton students to writing seminars.

Assignment Problem Reduces to Linear Programming

LP formulation. $x_{ij} = 1$ if job $j$ assigned to machine $i$.

$$\begin{align*}
\text{min} & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\text{s. t.} & \sum_{j=1}^{n} x_{ij} = 1 \quad 1 \leq i \leq n \\
& \sum_{i=1}^{n} x_{ij} = 1 \quad 1 \leq j \leq n \\
& x_{ij} \geq 0 \quad 1 \leq i, j \leq n
\end{align*}$$

Theorem. [Birkhoff 1946, von Neumann 1953] All extreme points of the above polytope are (0-1)-valued.

Corollary. Assignment problem reduces to LP; can solve in poly-time.

we assume LP returns an extreme point solution

3-Satisfiability

Literal: A Boolean variable or its negation. $x_i$ or $\overline{x_i}$

Clause. A disjunction of 3 distinct literals. $C_j = x_1 \lor \overline{x_2} \lor x_3$

Conjunctive normal form. A propositional formula $\Phi$ that is the conjunction of clauses. $\Phi = C_1 \land C_2 \land C_3 \land C_4$

3-SAT. Given a CNF formula $\Phi$ consisting of $k$ clauses over $n$ literals, does it have a satisfying truth assignment?

$$(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor \overline{x_3}) \land (x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor \overline{x_3})$$

Solution: $x_1 =$ true, $x_2 =$ true, $x_3 =$ false, $x_4 =$ true

Key application. Electronic design automation (EDA).

Graph 3-Colorability

3-COLOR. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?

yes instance
Claim. Given a graph, is there a way to color the vertices red, green, and blue so that no adjacent vertices have the same color?

Pf. Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.

Claim. Graph is 3-colorable iff \( \Phi \) is satisfiable.

Pf. Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.
**Graph 3-Colorability**

**Claim.** Graph is 3-colorable iff \( \Phi \) is satisfiable.

**Pf.**
- Suppose graph is 3-colorable.
  - Consider assignment that sets all T literals to true.
  - (ii) ensures each literal is T or F.
  - (iii) ensures a literal and its negation are opposites.
  - (iv) ensures at least one literal in each clause is T.

(if not, then G wouldn’t be 3-colorable, a contradiction)

![Graph 3-Colorability Diagram]

**Cook’s Theorem**

**3-SAT**

**3DM VERTEX COVER**

**HAM-CYCLE**

**CLIQUE**

**INDEPENDENT SET**

**3-COLOR**

**PLANAR-3-COLOR**

**EXACT COVER**

**SUBSET-SUM**

**PARTITION**

**INTEGER PROGRAMMING**

**KNAPSACK**

**BIN-PACKING**

Conjecture: no poly-time algorithm for 3-SAT. (and hence none of these problems)

All of these problems (any many more) polynomial reduce to 3-SAT.
Cook + Karp

Reductions are important in theory to:
- Establish tractability.
- Establish intractability.
- Classify problems according to their computational requirements.

Reductions are important in practice to:
- Design algorithms.
- Design reusable software modules.
  - stack, queue, sorting, priority queue, symbol table, set, graph shortest path, regular expressions, linear programming
- Determine difficulty of your problem and choose the right tool.
  - use exact algorithm for tractable problems
  - use heuristics for intractable problems
  e.g., bin packing