Comparing Problems

Remember the concepts of Problem, Algorithm, and Program.

We’ve gotten pretty good at comparing algorithms. How do we compare problems?

- Sorted Array Search
- Sorting
- Integer Factorization
- Integer Multiplication
- Selection
- Maximum Matching
- Minimum Vertex Cover

Computational Complexity

The **difficulty of a problem** is the worst-case cost of the **best possible algorithm** that solves that problem.

Computational complexity is the study and classification of problems according to their inherent difficulty.

Why study this?

- Want to know when an algorithm is as good as possible.
- Sometimes we want problems to be difficult!

How to compare problems

Big-\(O\), big-\(\Theta\), and big-\(\Omega\) are used to compare two functions.

How can we compare two problems?

Example: Sorting vs. Selection

- Forget about any specific algorithms for these problems.
- Instead, develop algorithms to solve one problem by using any algorithm for the other problem.
- Solving selection using a sorting algorithm:
- Solving sorting using a selection algorithm:
- Conclusion?
Defining tractable and intractable

Cobham-Edmonds thesis:
A problem is tractable only if it can be solved in polynomial time.

What can we say about intractable problems?
- Maybe they're undecidable (e.g., the halting problem)
- Maybe they just seem impossible (e.g., regexp equivalence)
- But not always! (e.g., integer factorization)

**Million-dollar question:**
Can any problems be verified quickly but not solved quickly?

Fair comparisons: Machine models

Proving lower bounds on problems requires a careful model of computation.

Candidates:
- Turing machine
- Clock cycles on your phone
- MIPS instructions
- "Primitive operations"

**Theorem**
*These models are all polynomial-time equivalent.*

Fair comparisons: Bit-length

**Input size** is our measure of difficulty \( n \).
It must be measured the same between different problems!

Past examples:
- Factorization \( \Theta(\sqrt{n}) \) vs. HeapSort \( \Theta(n \log n) \)
- Karatsuba's \( \Theta(n^{1.59}) \) vs. Strassen's \( \Theta(n^{2.81}) \)
- Dijkstra's \( \Theta(n^2) \) vs Dijkstra's \( \Theta((n + m) \log n) \)

Only measure for this unit: **length in bits of the input**
Fair comparisons: Decision problems

What about the size of the output? We’ll consider only:

Definition: Decision Problems
Problems whose output is YES or NO

Is this a big restriction?
- Selection
- El Scheduling
- Integer factorization
- Minimum vertex cover

Decision problem comparison

Compare regular factorization with decision problem version:
- Given instance \((N, k)\) of decision problem,
  use computational version to solve it:

- Given instance \(N\) of computational problem,
  use decision problem to solve it:

Formal Problem Definitions

Page 1

\textsc{ShortPath}(G, u, v, k)

\textbf{Input}: Graph \(G = (V, E)\), vertices \(u\) and \(v\), integer \(k\)

\textbf{Output}: Does \(G\) have a path from \(u\) to \(v\) of length at most \(k\)?

Input size and encoding:

\textsc{LongPath}(G, u, v, k)

\textbf{Input}: Graph \(G = (V, E)\), vertices \(u\) and \(v\), integer \(k\)

\textbf{Output}: Does \(G\) have a path from \(u\) to \(v\) of length at least \(k\)?

Input size and encoding:
Formal Problem Definitions

FACT(N, k)

Input: Integers N and k
Output: Does N have a prime factor less than k?

Input size and encoding:

VC(G, k)

Input: Graph G = (V, E), integer k
Output: Does G have a vertex cover with at most k nodes?

Input size and encoding:

Our first complexity class

Complexity theory is all about classifying problems based on difficulty.

Definition
The complexity class \( P \) consists of all decision problems that can be solved by an algorithm whose worst-case cost is \( O(n^k) \), for some constant k, and where n is the bit-length of the input instance.

This is the “polynomial-time” class. Can you name some members?

Nice properties of \( P \)

When we just worry about polynomial-time, we can be really lazy in analysis!

Polynomial-time is closed under:

- **Addition**: \( n^k + n^\ell \in O(n^{\max(k, \ell)}) \)
  - In terms of algorithms: one after the other.

- **Multiplication**: \( n^k \cdot n^\ell \in O(n^{k+\ell}) \)
  - In terms of algorithms: calls within loops.

- **Composition**: \( n^k \circ n^\ell \in O(n^{k\ell}) \)
  - In terms of algorithms: replace every primitive op. with a function call
Certificates

A *certificate* for a decision problem is some kind of digital “proof” that the answer is YES.

The certificate is usually what the output *would be* from the “computational version”.

Examples (informally):
- Integer factorization
- Minimum vertex cover
- Shortest path
- Longest path

Verifiers

A *verifier* is an algorithm that takes:
- Problem instance (input) for some decision problem
- An alleged certificate that the answer is YES

and returns YES iff the certificate is legit.

Principle comes from “guess-and-check” algorithms:
- Finding the answer is tough, but
- checking the answer is easy.

*We can write fast verifiers for hard problems!*

Our second complexity class

**Definition**

The complexity class **NP** consists of all decision problems that have can be *verified* in polynomial-time in the bit-size of the original problem input.

**Steps for an NP-proof:**
- Define a notion of certificate
- Prove that certificates have length \( O(n^k) \) for some constant \( k \)
- Come up with a verifier algorithm
- Prove that the algorithm runs in time \( O(n^k) \) for some (other) constant \( k \)
VC is in **NP**

VC\((G,k)\): “Does \(G\) have a vertex cover with at most \(k\) vertices?”

- Certificate:

- Certificate size:

- Verifier algorithm:

- Algorithm cost:

FACT is in **NP**

FACT\((N,k)\): “Does \(N\) have a prime factor less than \(k\)?”

- Certificate:

- Certificate size:

- Verifier algorithm:

- Algorithm cost:

How to get rich

The **BIG** question is: Does \(P\) equal \(NP\)?

The Clay Institute offers $1,000,000 for a proof either way.

- What you would need to prove \(P = NP\):

- What you would need to prove \(P \neq NP\):

In a nutshell: Is guess-and-check ever the best algorithm?
Alternate meaning of **NP**

Meaning of the name **NP**: “Non-deterministic polynomial time”

Non-deterministic Turing machine
  - Turing machine with (possibly) multiple transitions for the same current state and current tape symbol
  - Like a computer program with “guesses”
  - Connection to randomness?

Why is this equivalent to our definition with certificates and verifiers?

**Reductions**

Recall that a reduction from problem A to problem B is a way of solving problem A using *any algorithm* for problem A. Then we know that A is not more difficult than B.

Formally, a reduction from A to B:
  - Takes an *instance* of problem A as input
  - Uses this to create *m* instances of problem B
  - Uses the solutions to those *m* problem B’s to recover the solution for the original problem A

**Example Linear-Time Reduction**

Two problems:
  - **MMUL(A, B)**: Compute the product of matrices A and B
  - **MSQR(A, B)**: Compute the matrix square $A^2$

Show that the inherent difficulty of MMUL and MSQR is the same.
Polynomial-Time Reduction

**Ingredients for analyzing a reduction:**
(All will be functions of \( n \), the input size for problem A)
- Number \( (m) \) of problem B instances created
- Maximum bit-size of a problem B instance
- Amount of extra work to do the actual reduction.

**Polynomial-time reduction:** all three ingredients are \( O(n^k) \)
(Often \( m = 1 \), sometimes called a "strong reduction".)
We write \( A \leq_P B \), meaning
"A is polynomial-time reducible to B".

---

Formal Problem Definitions

**Minimum Hitting Set:** \( \text{HITSET}(L, k) \)

**Input:** List \( L \) of sets \( S_1, S_2, \ldots, S_m \), integer \( k \).

**Output:** Is there a set \( H \) with size at most \( k \) such that every \( S_i \cap H \) is not empty?

**HAMCYCLE(G)**

**Input:** Graph \( G = (V, E) \)

**Output:** Does \( G \) have a cycle that touches every vertex?

---

**VC reduces to HITSET**
Reductions

HAMCYCLE reduces to LONGPATH

NP-Completeness

Completeness

Definition
A problem B is \textbf{NP}-hard if \( A \leq^P B \) for every problem \( A \in \text{NP} \).

Informally: \textbf{NP}-hard means “at least as difficult as every problem in \textbf{NP}”

Definition
A problem B is \textbf{NP}-complete if B is \textbf{NP}-hard and B \( \in \text{NP} \).

What is the hardest problem in \textbf{NP}?

An easy \textbf{NP}-hard proof

\textbf{Theorem}: The halting problem is \textbf{NP}-hard.

\textbf{Proof}:
Circuit Satisfiability: CIRCUIT-SAT(C)

Input: Boolean circuit $C$ with AND, OR, and NOT gates, $m$ inputs, and one output.

Output: Is there a setting of the $m$ inputs that makes the output true?

Input size and encoding:

3-SAT(F)

Input: Boolean formula $F$ in “conjunctive normal form” (product of sums), with three literals (terms) in every sum (clause):

\[ F = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_2 \lor x_4 \lor x_5) \land (x_1 \lor x_2 \lor \neg x_4) \land \cdots \]

Output: Can we assign T/F to the $x_i$'s to make the formula true?

Input size and encoding:

NP-Completeness

Modeling programs as circuits

Remember this simple model of a computer?

- **State** contains PC, registers, program, memory
  - Size is linear in input size and program runtime
- **Combinational** is a circuit (AND, OR, and NOT gates)
  - for ALUs, MUXes, control, shifts, adders, etc.
  - Size is polynomial in size of state.

Lemma

*Any decision problem with a polynomial-time algorithm can be simulated by a polynomial-size boolean circuit.*

CIRCUIT-SAT is **NP-hard**
**NP-Completeness**

**Theorem**  
*CIRCUIT-SAT* is NP-complete.  

**Proof:** All that’s left is to show CIRCUIT-SAT ∈ NP.

- We only have to do this kind of proof once (why?)  
- Will this help us prove **P ≠ NP**?

---

**More NP-Complete Problems**

**3-SAT**

We want to reduce CIRCUIT-SAT to 3-SAT.  

**Idea:** Every wire in the circuit becomes a variable.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>((\neg x \lor \neg y \lor z) \land (x \lor \neg z) \land (y \lor \neg z))</td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>((x \lor \neg z) \land (\neg x \lor \neg z) \land (\neg y \lor \neg z))</td>
</tr>
<tr>
<td><img src="image" alt="Gate" /></td>
<td>((x \lor \neg z) \land (\neg x \lor \neg z))</td>
</tr>
</tbody>
</table>

- What do these clauses ensure?
- What other clause do we need to add?

---

**VC**

Reduce 3-SAT to VC.
Properties of **NP-Complete Problems**

There are many known **NP**-complete problems. We have seen: LONGPATH, VC, HITSET, HAMCYCLE, CIRCUIT-SAT, 3-SAT.

What’s needed to prove a new problem is **NP**-complete:

**Note:** All have *one-sided* verifiers (can’t verify NO answer!)

What about FACT?

---

Frontiers of Complexity Theory

Big open questions:
- Does $P = \text{NP}$? (Probably not)
- Is FACT **NP**-complete? (Probably not)
- Is FACT in $P$? (Hopefully not!)
- Do true one-way functions exist? (Not if $P = \text{NP}$)
- Can quantum computers solve **NP**-hard problems? (Probably not)
- Where does randomness fit in?

---

Traveling Salesman Problem

**TSP Definition**

**Input:** Graph $G = (V, E)$

**Output:** The shortest cycle that includes every vertex exactly once, or **FAIL** if none exist.

- Classic **NP**-hard problem
- Many important applications
- The worst-case is hard — so what can we do?
MSTs and TSP

**Theorem**: Length of TSP tour is at least the size of a MST.

```
   a --- 5 --- 4 --- 2
   |      |      |
  5      2      2
   e --- 5 --- 4 --- 5
   b --- 2      2
   c
```

Branch and Bound

How to compute the optimal TSP?
- Pick a starting vertex
- Explore every path, depth-first
- Return the least-length Hamiltonian cycle

This is really slow (*of course!*)

Branch and bound idea:
- Define a quick lower bound on remaining subproblem (MST!)
- Stop exploring when the lower bound exceeds the best-so-far

Simplified TSP

Solving the TSP is really hard; some special cases are a bit easier:

**Metric TSP**
- Edge lengths “obey the triangle inequality”:
  \[ w(a, b) + w(b, c) \geq w(a, c) \forall a, b, c \in V \]
- What does this mean about the graph?

**Euclidean TSP**
- Graph can be drawn on a 2-dimensional map.
- Edge weights are just distances!
- (Sub-case of Metric TSP)
Approximating Metric TSP

**Idea**: Turn any MST into a TSP tour.

![Graph][1]

How good is the approximation?

Greedy TSP

Greedy strategies:
- Nearest neighbor
- Smallest “good” edge

![Graph][2]

Local Refinement

**Idea**: Take any greedy solution, then make it better.

2-OPT refinement:
- Take a cycle with \((a, b)\) and \((c, d)\)
- Replace with \((a, c)\) and \((b, d)\)