Intuitive Tour of Key Complexity Classes

Aimal Rextin

SEECS-NUST

Roadmap

1. **P**
2. **NP**
3. NP-Complete
4. **EXP**
5. Undecidable

Why decision problems?

- Canonical yes/no form: every instance has a single-bit answer, making time and
- Optimisation ⇒ decision: most search or optimisation tasks are polynomial-time equivalent to a decision version (e.g. is there k weight path between s to other vertices versus shortest path).

Why decision problems?

- Canonical yes/no form: every instance has a single-bit answer, making time and
- Optimisation $\Rightarrow$ decision: most search or optimisation tasks are polynomial-time equivalent to a decision version (e.g. is there k weight path between s to other vertices versus shortest path).

P — "Fast" Algorithms

Intuition

Problems solvable by an algorithm whose running time grows at most like a polynomial in the input size (e.g. n, n^2, n^3).

P — "Fast" Algorithms

Intuition

Problems solvable by an algorithm whose running time grows at most like a polynomial in the input size (e.g. n, n^2, n^3).

- Sorting n numbers ($\mathtt{O}(n \log n)$).
- Finding a shortest path in a road network.
- Checking if a year is a leap-year.

P — “Fast” Algorithms

Intuition

Problems solvable by an algorithm whose running time grows at most like a polynomial in the input size (e.g. n, n^2, n^3).

- Sorting n numbers (`O(`$n \log n$`))`.
- Finding a shortest path in a road network.
- Checking if a year is a leap-year.
- Take-away: These are the tasks we can usually handle even at massive scale.

NP — Solutions Check Quickly

Intuition

A problem is in NP if, **given a proposed solution**, a regular computer can verify its correctness in polynomial time.

NP — Solutions Check Quickly

Intuition

A problem is in NP if, **given a proposed solution**, a regular computer can verify its correctness in polynomial time.

- Solving a Sudoku: easy to check a filled board, hard to fill it.
- Finding a Hamiltonian path in a graph.
- Satisfying a Boolean formula (SAT).
    - $$(\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_3 \vee x_4 \vee x_1)$$

NP-Complete — Toughest in **NP**

Intuition

If you design a fast (polynomial) algorithm for any NP-complete problem, every NP problem inherits a fast algorithm.

NP-Complete — Toughest in **NP**

Intuition

If you design a fast (polynomial) algorithm for any NP-complete problem, every NP problem inherits a fast algorithm.

- First discovered: `SAT`.
- `3-SAT`, `Clique`, Travelling-Salesperson (decision version).

NP-Complete — Toughest in **NP**

Intuition

If you design a fast (polynomial) algorithm for any NP-complete problem, every NP problem inherits a fast algorithm.

- First discovered: `SAT`.
- `3-SAT`, `Clique`, Travelling-Salesperson (decision version).
- P vs NP: Do such fast algorithms exist? Still unknown.

EXP — Exponential-Time Algorithms

Intuition

Problems whose best known **deterministic** algorithms may take time like $2^{\text{poly}(n)}$ in the worst case.

EXP — Exponential-Time Algorithms

Intuition

Problems whose best known **deterministic** algorithms may take time like $2^{\text{poly}(n)}$ in the worst case.

- Exhaustively exploring all game states in certain video games.

EXP — Exponential-Time Algorithms

Intuition

Problems whose best known **deterministic** algorithms may take time like $2^{\text{poly}(n)}$ in the worst case.

- Exhaustively exploring all game states in certain video games.
- Provably **P** $\subsetneq$ **EXP**; so some tasks are guaranteed to lie beyond polynomial time.

Undecidable Problems — No Algorithm Works for All Inputs

Intuition

Some tasks cannot be solved by *any* algorithm that always halts with the correct answer.

Undecidable Problems — No Algorithm Works for All Inputs

Intuition

Some tasks cannot be solved by *any* algorithm that always halts with the correct answer.

- "Will this program ever stop?" — the famous Halting Problem.

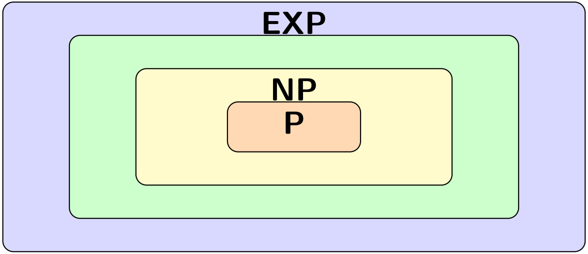
Undecidable Problems — No Algorithm Works for All Inputs

Intuition

Some tasks cannot be solved by *any* algorithm that always halts with the correct answer.

- "Will this program ever stop?" — the famous Halting Problem.
- Consequence: No matter how clever we are, a universal solution is impossible.

Time/Space Hierarchy (Not to Scale)



NP–Complete Optimisation Problem

Heuristic / AI approach

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

NP–Complete Optimisation Problem

Approximation algorithm

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

Approximation algorithm

- Comes with a provable performance ratio α.

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

Approximation algorithm

- Comes with a provable performance ratio α.
- Guarantees a solution within $1/\alpha$ (or α times) of optimum.

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

Approximation algorithm

- Comes with a provable performance ratio α.
- Guarantees a solution within $1/\alpha$ (or α times) of optimum.

	Optimal	Our answer
Value	100	80
Ratio	0.80 (80% of optimum)	

NP–Complete Optimisation Problem

Heuristic / AI approach

- Often works well in practice on typical instances.
- But has no worst–case guarantee on solution quality or running time.

Approximation algorithm

- Comes with a provable performance ratio α.
- Guarantees a solution within $1/\alpha$ (or α times) of optimum.

	Optimal	Our answer
Value	100	80
Ratio	0.80 (80% of optimum)	

The exact bound *varies from algorithm to algorithm*.

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\text{cost}(A(I)) \le \rho \cdot \text{OPT}(I).$$

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\text{cost}(A(I)) \le \rho \cdot \text{OPT}(I).$$

- $\rho = 2 \Rightarrow$ never worse than twice the best.

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\mathrm{cost}(A(I)) \;\leq\; \rho \cdot \mathrm{OPT}(I).$$

- $\rho = 2 \Rightarrow$ never worse than twice the best.
- $\rho = 1 + \varepsilon$ (e.g. 1.1) is very tight.

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\text{cost}(A(I)) \le \rho \cdot \text{OPT}(I).$$

- $\rho = 2 \Rightarrow$ never worse than twice the best.
- $\rho = 1 + \varepsilon$ (e.g. 1.1) is very tight.
- ρ may be a constant or a slow-growing function like $\log n$.

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\text{cost}(A(I)) \;\leq\; \rho \cdot \text{OPT}(I).$$

- $\rho = 2 \Rightarrow$ never worse than twice the best.
- $\rho = 1 + \varepsilon$ (e.g. 1.1) is very tight.
- ρ may be a constant or a slow-growing function like $\log n$.

OPT(I)	$A(I)$
100	180

$\Rightarrow$ ratio $= 1.8$ (a 1.8-approx)

Approximation Ratio (quick view)

Goal. Get a solution that is *close* to optimum when exact optimisation is too slow.

Definition (minimisation)

Algorithm A is a ρ-approximation if, for every instance I,

$$\text{cost}(A(I)) \;\le\; \rho \cdot \text{OPT}(I).$$

- $\rho = 2 \Rightarrow$ never worse than twice the best.
- $\rho = 1 + \varepsilon$ (e.g. 1.1) is very tight.
- ρ may be a constant or a slow-growing function like $\log n$.

OPT(I)	$A(I)$
100	180

$\Rightarrow$ ratio $= 1.8$ (a 1.8-approx)

Vertex Cover (minimisation)

Problem statement

Input: an undirected graph $G = (V, E)$ of size n.
Goal: find the smallest subset $C \subseteq V$ that *touches every edge*, i.e. for every $\{u, v\} \in E$ we have $u \in C$ or $v \in C$.

Vertex Cover (minimisation)

Problem statement

Input: an undirected graph $G = (V, E)$ of size n.
Goal: find the smallest subset $C \subseteq V$ that *touches every edge*, i.e. for every $\{u, v\} \in E$ we have $u \in C$ or $v \in C$.

- NP-complete $\implies$ exact solutions need exponential time in the worst case.

Vertex Cover (minimisation)

Problem statement

Input: an undirected graph $G = (V, E)$ of size n.
Goal: find the smallest subset $C \subseteq V$ that *touches every edge*, i.e. for every $\{u, v\} \in E$ we have $u \in C$ or $v \in C$.

- NP-complete $\implies$ exact solutions need exponential time in the worst case.
- Best enumerate all 2^n subsets and keep the smallest cover.

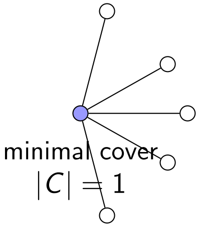
Vertex Cover (minimisation)

Problem statement

Input: an undirected graph $G = (V, E)$ of size n.
Goal: find the smallest subset $C \subseteq V$ that *touches every edge*, i.e. for every $\{u, v\} \in E$ we have $u \in C$ or $v \in C$.

- NP-complete $\implies$ exact solutions need exponential time in the worst case.
- Best enumerate all 2^n subsets and keep the smallest cover.

Minimal Vertex Cover Example



Larger graphs generally need many vertices in a cover, and brute-force checking all subsets explodes to 2^n.

2-Approximation for Vertex Cover

```
ApproxCover(G = (V, E)):
  S = NULL                  # current cover
  while E != empty set:
    pick an arbitrary edge e = (u, v)
    S = S U {u, v}
    remove from E every edge incident to u or v
  return S
```

Runs in $O\big(|E|\big)$ time and guarantees $|S| \leq 2|O|$.

Why the Algorithm is a **2**-Approximation

- A — set of edges chosen by the algorithm (one per iteration)
- O — an optimal vertex cover
- S — vertices returned by the algorithm

$$|S| = 2\,|A| \qquad \text{(both endpoints of every edge in } A\text{)} \tag{1}$$

$$|O| \geq |A| \qquad \text{(each edge in } A \text{ must be covered by } O\text{)} \tag{2}$$

From (1) and (2),

$$\boxed{|S| \leq 2\,|O|}.$$

Hence the algorithm is a 2-approximation.