Complexity Theory, Growth Comparisons, and Sorting Runtimes

Max Randall

April 27, 2025

Contents

1 Introduction

We begin with an overview of intractable computational problems from Chapter 10 of Hopcroft, Motwani, and Ullman, introducing the complexity classes P and NP and the notion of NP-completeness. Then we work through exercises on ordering and relating functions by asymptotic growth and finish with classic sorting-algorithm comparisons. Along the way we include a couple of illustrative plots to make the abstract inclusions concrete.

2 Readings

2.1 Chapter 10.1: The Classes P and NP

Chapter 10.1 refocuses from mere decidability to *tractability*: among the decidable problems, which can actually be solved in a reasonable amount of time? It formalizes the class P as those languages decidable by a deterministic Turing machine in time polynomial in the input length, and NP as those decidable by a nondeterministic machine whose every branch halts in polynomial time. The chapter motivates polynomial time as the practical threshold—algorithms superpolynomial in nature blow up too quickly to handle large inputs—and introduces polynomial-time reductions, which transform instances of one problem into another in polynomial time, preserving membership. These reductions provide a rigorous way to compare problem hardness: if A reduces to B and B is in P, then A is also in P. Finally, the chapter highlights problems whose best-known solutions are exponential, hinting at the P versus NP question at the heart of complexity theory.

2.2 Chapter 10.2: SAT and Cook's Theorem

Chapter 10.2 presents the Boolean Satisfiability Problem (SAT): given a propositional formula built from variables, negation, conjunction, and disjunction, does there exist an assignment of true/false values that makes it true? After fixing a finite-alphabet encoding, it shows SAT lies in NP by having a nondeterministic machine guess an assignment and then evaluate the formula in polynomial time. The centerpiece is Cook's Theorem, which constructs, in polynomial time, a Boolean formula whose structure enforces that an NP machine's accepting computation on input x exists if and only if the formula is satisfiable. By encoding the entire computation tableau into variables and clauses that ensure correct transitions and an accepting state, the proof establishes that every NP problem reduces to SAT. Thus SAT is NP-complete: it sits in NP and is as hard as any NP problem.

2.3 Chapter 10.3: CNF, CSAT, and 3SAT

Chapter 10.3 refines SAT by restricting formula shape to *conjunctive normal form* (CNF)—an AND of clauses, each a disjunction of literals—and to k-CNF where each clause has exactly k literals. Converting arbitrary formulas to equivalent CNF can blow up exponentially, so the chapter gives an *equisatisfiable* transformation: push negations downward via De Morgan's laws so they only apply to variables, then introduce fresh variables to break complex subformulas into small clauses without replicating subexpressions. This yields a polynomial-time reduction from general SAT to CSAT (CNF-SAT), proving CSAT NP-complete. Finally, it shows how to transform any CNF into an equisatisfiable 3-CNF formula in linear time by splitting long clauses with auxiliary variables. Hence even 3SAT remains NP-complete, cementing its role as the canonical hard problem in complexity theory.

3 Homework

3.1 Exercise 1

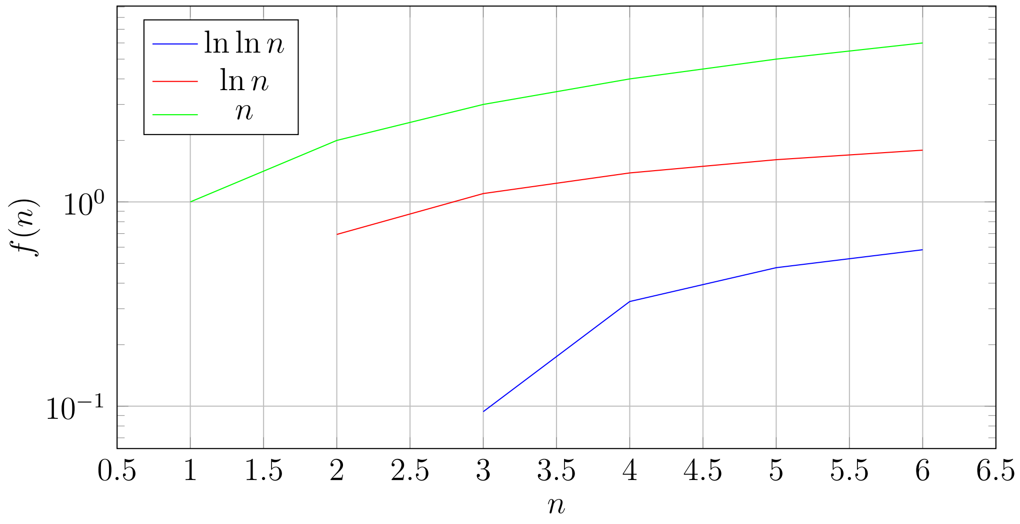
Order by growth (slow $\to$ fast):

$$2^{2^n}, \quad e^{\log n}, \quad \log n, \quad e^n, \quad e^{2\log n}, \quad \log(\log n), \quad 2^n, \quad n!.$$

Answer.

$$\log(\log n) \prec \log n \prec n \prec n^2 \prec 2^n \prec e^n \prec n! \prec 2^{2^n}.$$

Illustrative Growth Plot



3.2 Exercise 2

For $f, g, h\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, prove:

1. $f \in O(f)$,
2. $O(c\,f) = O(f) \quad (\forall c > 0)$,
3. $f(n) \leq g(n)$ eventually $\implies O(f) \subseteq O(g)$,
4. $O(f) \subseteq O(g) \implies O(f + h) \subseteq O(g + h)$,
5. $h(n) > 0\ \forall n,\ O(f) \subseteq O(g) \implies O(f\,h) \subseteq O(g\,h)$.

Proof.

1. Since $f(n) \le 1 \cdot f(n)$, we have $f \in O(f)$.
2. $u \in O(c\, f)$ means $u(n) \le C\,(c\, f(n))$, so $u \in O(f)$. Conversely the same argument shows $O(f) \subseteq O(c\, f)$.
3. If $f(n) \le g(n)$ for all large n and $u \in O(f)$, then $u(n) \le C\, f(n) \le C\, g(n)$, so $u \in O(g)$.
4. If $u \in O(f)$, then for large n:
$$u(n) + h(n) \;\le\; C\, f(n) + h(n) \;\le\; C\big(f(n) + h(n)\big),$$
hence $u + h \in O(f + h)$. Since $O(f) \subseteq O(g)$, also $u + h \in O(g + h)$.
5. If $u \in O(f)$ and $h(n) > 0$, then
$$u(n)\, h(n) \;\le\; C\, f(n)\, h(n) \;\le\; C\, C'\, g(n)\, h(n),$$
showing $u\, h \in O(g\, h)$.

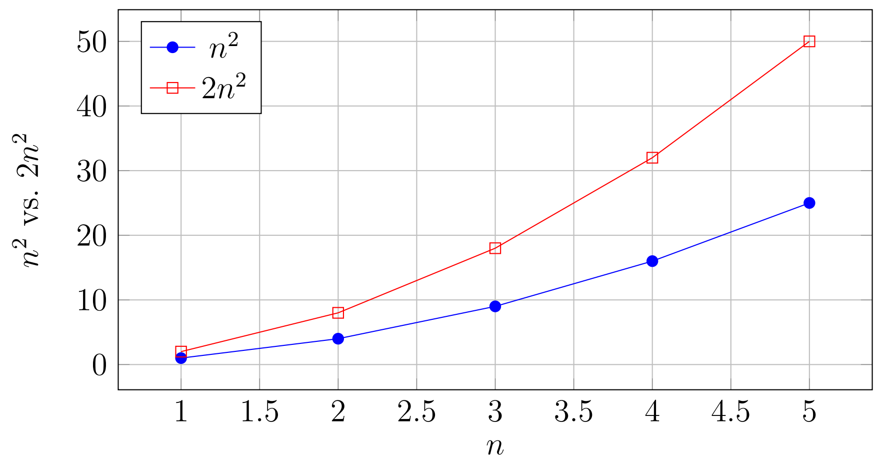


Figure 1: Verifying $O(c\, f) = O(f)$.

Constant-Factor Comparison

3.3 Exercise 3

Let $i, j, k, n \in \mathbb{N}$. Prove:

$$\begin{aligned}
&1.\ j \le k \implies O(n^j) \subseteq O(n^k),\\
&2.\ j \le k \implies O(n^j + n^k) \subseteq O(n^k),\\
&3.\ O\Big(\sum_{m=0}^{k} a_m\, n^m\Big) = O(n^k),\\
&4.\ O(\log n) \subseteq O(n),\\
&5.\ O(n \log n) \subseteq O(n^2).
\end{aligned}$$

Proof.

1. If $j \le k$, then $n^j \le n^k$ for $n \ge 1$; hence any $u \in O(n^j)$ satisfies $u(n) \le C\,n^j \le C\,n^k$, so $u \in O(n^k)$.
2. For $n \ge 1$, $n^j + n^k \le 2n^k$. If $u \in O(n^j + n^k)$, then $u(n) \le C(n^j + n^k) \le 2C\,n^k$, so $u \in O(n^k)$.
3. $\sum_{m=0}^{k} a_m\, n^m \le \left(\sum |a_m|\right) n^k$, hence $O(\sum a_m n^m) = O(n^k)$.
4. For $n \ge 2$, $\log n \le n$, so $O(\log n) \subseteq O(n)$.
5. For $n \ge 2$, $n \log n \le n^2$, so $O(n \log n) \subseteq O(n^2)$.

3.4 Exercise 4

Which relationships hold between:

1. $O(n)$ vs. $O(\sqrt{n})$,
2. $O(n^2)$ vs. $O(2^n)$,
3. $O(\log n)$ vs. $O((\log n)^2)$,
4. $O(2^n)$ vs. $O(3^n)$,
5. $O(\log_2 n)$ vs. $O(\log_3 n)$.

Answer.

1. $\sqrt{n} \le n$ for $n \ge 1$, so $O(\sqrt{n}) \subsetneq O(n)$.
2. $n^2 = o(2^n)$, hence $O(n^2) \subsetneq O(2^n)$.
3. Eventually $(\log n)^2 \ge \log n$, so $O(\log n) \subseteq O((\log n)^2)$.
4. $2^n \le 3^n$, thus $O(2^n) \subsetneq O(3^n)$.
5. $\log_2 n = \frac{\log_3 n}{\log_3 2}$, a constant factor, so $O(\log_2 n) = O(\log_3 n)$.

3.5 Exercise 5

Classic sorting comparisons:

bubble sort vs. insertion sort, insertion sort vs. merge sort, merge sort vs. quick sort.

Discussion.

- **Bubble vs. Insertion:** Both worst-case $O(n^2)$, but insertion sort is $O(n)$ on nearly-sorted input.
- **Insertion vs. Merge:** Insertion sort worst-case $O(n^2)$, merge sort always $O(n \log n)$, so merge scales better.
- **Merge vs. Quick:** Merge sort is $O(n \log n)$ always; quick sort is $O(n \log n)$ average, $O(n^2)$ worst, but faster in practice due to lower constants and in-place partitioning.

4 Conclusion

We've surveyed P vs. NP, NP-completeness via SAT, practiced ordering and relating functions by asymptotic growth, and applied these insights to sorting algorithms. The included plots illustrate constant-factor and growth-rate comparisons without overwhelming the presentation.

Question: What's the fastest known quantum sorting algorithm in the query (comparison) model, and how close is it to being practical?

References

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata, Languages, and Computation*, Addison–Wesley, 3rd ed., 2006.