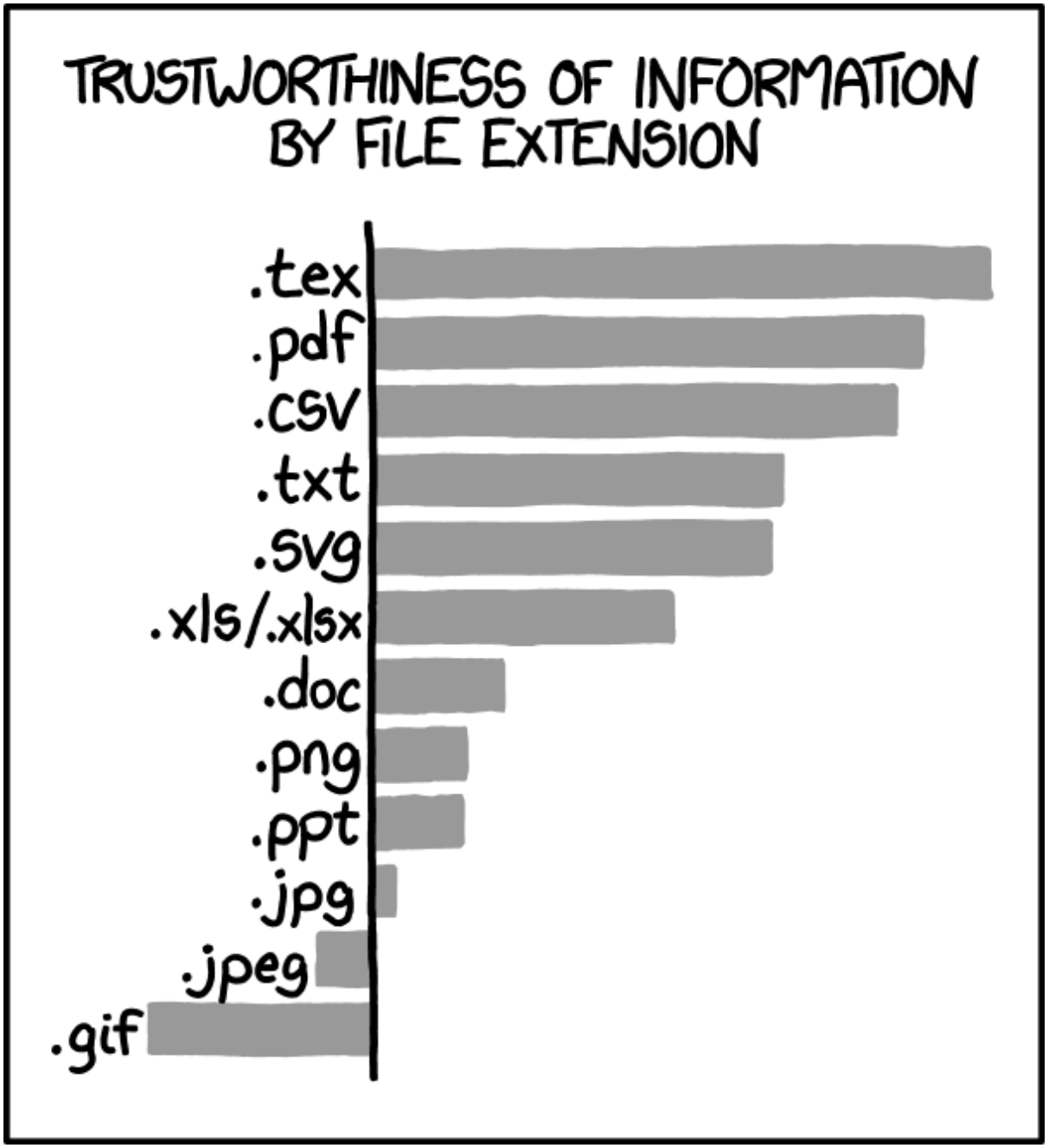
CSCI6454: Advanced Algorithms

ADITHYA BHASKARA
PROFESSOR: HUCK BENNETT

READINGS: VARIOUS

UNIVERSITY OF COLORADO BOULDER

EDITION 1



Contents

Preface

To the interested reader,

This document is a compilation of lecture notes scribed during the Spring 2025 semester for CSCI6454: Advanced Algorithms at the University of Colorado Boulder. The content in these course notes is from lecture and several resources cited herein. The author only takes responsibility for the scribing of the content and lectures into the notes. This course was taught by Huck Bennett, Ph. D.

While much effort has been put in to remove typos and mathematical errors, it is very likely that some errors, both small and large, are present. If an error needs to be resolved, please contact Adithya Bhaskara at adithya@colorado.edu.

Best Regards,
Adithya Bhaskara

REVISED: January 14, 2025

1 Introduction to Advanced Algorithms

1.1 Lecture 1: January 14, 2025

1.1.1 Introduction to Fine-Grained Complexity

We hope to precisely understand the complexity of several algorithmic problems beyond the coarse information given by standard complexity classes, like P and NP. A central theme is proving lower bounds for the complexity of algorithms; this is a difficult task in general. We use reductions as an essential tool.

The high-level idea behind fine-grained complexity is to explain hardness, or the lack of algorithmic progress, on fundamental computational problems by giving reductions from well-studied problems. Recall the P v. NP problem. Loosely, problems in P are "easy," to solve, and problems in NP are "easy" to verify positive answers to. The hardest problems in NP, the NP-complete problems do not admit polynomial-time algorithms unless P $=$ NP. To show that a problem is NP-complete, we show that it is in NP, and exhibit a polynomial-time reduction from a known NP-complete problem.

The theory behind NP-completeness is robust and quite nice. But, we'd like finer results. Knowing whether a problem is in P or not doesn't really narrow down how efficiently we can solve it in practice. Even quadratic-time algorithms may be inefficient in production.

These ideas motivate fine-grained complexity. Here, we start by taking a well-studied problem, say L_1, and making a precise conjecture about the running time of optimal algorithms for L_1. Then, we give an efficient reduction from L_1 to a problem L_2, for which we are showing hardness. We start by providing some hypotheses corresponding to certain problems, and later, we'll show fairly precise running time bounds for problems in P.

Example 1. *Under certain assumptions, there is no $O(n^{2-\epsilon})$-time algorithm for the* Edit Distance *problem. This matches the $O(n^2)$ dynamic programming algorithm.*

Computational Problem 1 (k-SAT)**.**

- *Given a CNF formula φ with k literals in each clause,*
- *Decide: Does there exist a satisfying assignment?*

Hypothesis 1.1.1: Exponential Time Hypothesis (ETH)

The 3-SAT problem takes $2^{\Omega(n)}$ time.

Remark. *Note that* P $\neq$ NP *asserts that* 3-SAT *has no polynomial-time algorithm, whereas ETH asserts that* 3-SAT *has no subexponential-time algorithm. ETH implies* P $\neq$ NP*.*

Hypothesis 1.1.2: Strong Exponential Time Hypothesis (SETH)

For every $\epsilon > 0$, there exists $k \in \mathbb{Z}^+$ such that there is no $O\left(2^{(1-\epsilon)n}\right) = O((2-\epsilon)^n)$-time algorithm for k-SAT.

Remark. *SETH, at a high level, claims that 2^n-time is essentially optimal for k-SAT for large k.*

Computational Problem 2 (k-Sum)**.**

- *Given arrays $A_1, \dots, A_k$ each with n integers in $[-n^c, n^c]$,*
- *Decide: Does there exist $a_1 \in A_1, \dots, a_k \in A_k$ such that $a_1 + \cdots + a_k = 0$?*

Some variants include assuming that $A_1 = \cdots = A_k$, or wanting to find $a_1 \in A_1, \dots, a_k \in A_k$ where $a_1 + \cdots + a_{k-1} = a_k$.

Naively, we can solve k-Sum by trying all possible choices a_i and checking the sum. This can be done in $O(n^k)$ time. We can do better. For now, let $k = 3$. We start by computing all possible sums $a_1 + a_2$. Then, define $S = \{a_1 + a_2 : a_1 \in A_1, a_2 \in A_2\}$. Sort A_3. For each $s \in S$, search in A_3 for $-s$. If $-s$ is found, there exists a solution. Otherwise, there is no solution. This provides an $O(n^2 \log n)$ algorithm; computing the sums takes $O(n^2)$ time, sorting takes $O(n \log n)$ time, and the search procedure takes $O(n^2 \log n)$ time.

Hypothesis 1.1.3: No $O(n^{2-\epsilon})$-Time Algorithm for 3-Sum

For every $\epsilon > 0$, there is no $O\left(n^{2-\epsilon}\right)$-time algorithm for 3-Sum.

Computational Problem 3 (All Pairs Shortest Path (APSP))**.**

- *Given a weighted graph $G = (V, E, w)$ on n vertices,*
- *Find: The lengths of the shortest paths between every $v_i, v_j \in V$.*

One way to solve solve APSP is to run Dijkstra's algorithm on each vertex. Recall that Dijkstra's algorithm finds a single-source shortest path tree. We could also similarly use the Floyd-Warshall dynamic programming algorithm.

Hypothesis 1.1.4: No $O(n^{3-\epsilon})$-Time Algorithm for APSP

For every $\epsilon > 0$, there exists no $O(n^{3-\epsilon})$-time algorithm for APSP.

Appendices

Bibliography