COS 330: Great Ideas in Theoretical Computer Science

Fall 2026

Problem Set 1

Module: Classic Algorithms

Below is a reminder of key aspects of the PSet:

- **The only goal of this PSet is to help you develop your problem-solving skills.** Your performance on this PSet will not directly contribute to your grade, but will indirectly improve your ability to do well on the exams.
- **Your performance does not directly impact your grade**, so you may use any resources you like (collaboration, AI, etc.) to help you complete the PSet.
- **AI is not equally welcome in every part of this PSet**, and we say so explicitly where it matters. Problem 1 is exam-style and should be attempted alone first. Problem 2 explicitly calls for AI assistance in parts (c)–(e). Problem 3 is an optional challenge problem that was designed to be solved without AI assistance.
- **None of this is an integrity rule** – PSets are not graded directly. It's advice about what will actually develop your skills and in the process prepare you well for the exams, which are closed-book and closed-AI.
- **See the course website** for a full walkthrough of AI tools and recommendations, as well as the input files needed for Problem 2.
- **An AI-helper is available for Problem 1 and Problem 2(b)** – unlike a general assistant, it gives hints and pushes back instead of handing you the answer. See the link above for more information.
- Throughout the PSet, we've included some general tips to help put these into broader context. Exams will not have these, and future PSets may have fewer.

Aligning Expectations

The 🗎 symbol implies that the following problem is an "exam-style" problem. We highly recommend that you write up a full solution to this problem as if you were on an exam.

Problem 1 : String Fixing 🗎

Consider two length-n binary strings $s, t \in \{0,1\}^n$. We want to transform s into t using a sequence of operations.

Definition 1.1 (Interval Flip)**.** An Interval Flip takes an interval $[\ell, r]$, where $r \geq \ell$ (intervals of length one are allowed) and flips all bits of s in that interval. That is, for all $k \in [\ell, r]$, if $s[k] = 0$ then set $s[k] = 1$ and if $s[k] = 1$ then set $s[k] = 0$.

(a) Suppose an Interval Flip on $[\ell, r]$ costs $r - \ell + 1$ (that is, because a total of $r - \ell + 1$ bits are flipped by the operation, the cost is $r - \ell + 1$). The cost of a sequence of Interval Flips is simply the sum of the costs of each Interval Flip in the sequence.

Let $\textsc{Cost}(s, t)$ denote the minimum cost of any sequence of Interval Flips that turns s into t. Design an $O(n)$ time algorithm to compute $\textsc{Cost}(s, t)$. Prove your algorithm is correct, and prove that you've analyzed the runtime correctly.

(b) Now, suppose that:

1. The cost of each Interval Flip is C, no matter the length of the interval.
2. You are only allowed to pick sequences that flip each bit at most once. That is, for all i, your sequence cannot contain multiple Interval Flips that flip bit i.[1] Call a sequence of Interval Flips valid if it satisfies this constraint.
3. The final cost of a sequence of Interval Flips is the sum of two terms: (a) C times the number of Interval Flips, plus (b) the number of bits that are still incorrect after all operations (i.e., bits i such that $s[i] \neq t[i]$, after you finish flipping).[2]

Let $\textsc{NewCost}_C(s, t)$ denote the minimum cost (defined as in (3) above) of any valid sequence of Interval Flips. Describe an algorithm to find $\textsc{NewCost}_C(s, t)$ in $O(n^2)$ time. Prove your algorithm is correct, and prove that you've analyzed the runtime correctly.

Small Hint

It may help to define $\Delta(i, j)$ to be the number of indices between i and j where s and t differ. That is, $\Delta(i, j)$ counts the number of indices k such that both: (a) $i \leq k \leq j$ and (b) $s[k] \neq t[k]$. To save time on your write-up, you may use without proof that one can build an $n \times n$ array storing $\Delta(i, j)$ for all $i, j \in [1, n]$ in time $O(n^2)$.

[1] For example, the sequence $[1, 5], [10, 11]$ is valid. The sequence $[1, 5], [10, 11], [2, 25]$ is not valid because it flips 3 twice (it also flips other bits twice, such as 2, 10, etc.).

[2] For example, if $s := 101$ and $t := 110$, the sequence of Interval Flips $\langle [1, 1], [3, 3] \rangle$ turns s into 000. Therefore, the total cost is $2C + 2$, since we pay C for each of the two flips, plus one for each of the two remaining mismatches.

(c) Prove that, for all s, t of length n, and all C, $\min\{\textsc{Cost}(s,t), \textsc{NewCost}_C(s,t)\} \leq n/2 + C/2$. For simplicity of calculations, you may assume that both n and C are even integers.

Moreover, for any positive even n and positive even $C \leq n$, describe a specific s, t such that the following holds: $\min\{\textsc{Cost}(s,t), \textsc{NewCost}_C(s,t)\} = n/2 + C/2$. Prove that your example is correct.

Small Hint

This part is meant to be more challenging! If you get stuck, it can help to first go after a weaker version of the claim, and build up from there:

- Try proving $\min\{\textsc{Cost}(s,t), \textsc{NewCost}_C(s,t)\} \leq$ some bound that's a bit worse than $n/2 + C/2$, then see how far you can tighten it.
- Try finding an example where $\min\{\textsc{Cost}(s,t), \textsc{NewCost}_C(s,t)\}$ is a bit smaller than $n/2 + C/2$, then see if you can push it up to match.

Problem 2 : Trace Amounts

An agent was asked to fix a single failing test. It burned n tokens doing so, and you would like to know what it spent them on.

The run is recorded as follows. The agent was given the top-level task, and every task is one of two kinds. A task either does its work directly, consuming some number of tokens, or it delegates, splitting into one or more sub-tasks handed to fresh subagents, each a task in its own right. Delegating is pure bookkeeping and consumes no tokens itself, so all n tokens are consumed by the tasks that did their work directly. The tokens attributed to a task are the total consumed anywhere beneath it.

Displaying every task is not efficient – there might be hundreds of thousands of them. So your viewer shows an indented outline, and you choose which tasks to expand:

- The top-level task always occupies one line.
- If you expand a task, each of its sub-tasks gets its own line, indented one level further.
- If you do not expand a task, it occupies a single line, and nothing beneath it is shown.
- Every line shows the tokens attributed to whatever it names.

In the outlines below, `[-]` marks a task you have expanded, `[+]` a task that delegated and so could be expanded further, and `[ ]` a task that did its work directly and therefore has nothing inside it.

For example, on a run with $n = 798{,}431$, expanding only the top-level task gives the following, which uses five lines:

```
[-] fix failing test in auth module      798,431
    [ ] read test file                     1,204
    [+] locate the regression            612,880
    [+] write the patch                   91,455
    [ ] run test suite                    92,892
```

Expanding `locate the regression` as well costs three more lines, one for each of its sub-tasks:

```
[-] fix failing test in auth module      798,431
    [ ] read test file                     1,204
    [-] locate the regression            612,880
        [ ] grep for the error string      8,140
        [ ] read auth/session.py          47,900
        [+] bisect recent commits        556,840
    [+] write the patch                   91,455
    [ ] run test suite                    92,892
```

Your viewer has room for at most k lines, and you want to spend them well. A single line reading 798,431 tells you nothing at all; neither does one that leaves half the run sitting behind one entry while splitting some trivial sub-task across five. A good summary breaks up the lines that hide a lot of tokens, and uses its budget doing so.

Aligning Expectations – Using AI in This Question

Some parts of this problem were designed with AI assistance in mind: Problems 2(c), 2(d), and 2(e) explicitly ask you to use an AI assistant for at least part of the work. Don't try to complete those parts entirely by hand.

(a) Write down a formal statement of the situation above as a problem about trees. Say precisely what the input is, what counts as a valid summary, and what constraint a valid summary must satisfy. You should also specify what the goal is, i.e., what metric you are trying to optimize. You do not need to give an algorithm or prove anything here, just a formal statement of the problem.

Note that there are multiple reasonable ways to formalize the situation, you only need to come up with one. On part (b) we will give a formalization that we will use for the rest of the problem, but you do not need to match it exactly here. The goal is to practice writing down a formal statement of a problem, and there is no single "correct" answer.

Do not use an AI assistant on this part. Also, do not read part (b) before you have written something down: part (b) states the formalization we will use for the rest of the problem, so it gives away an answer.

Aligning Expectations

There is no single "correct" formalization here, and part (b) is not the only reasonable one. What we are looking for is a statement precise enough that somebody who had never heard the story could work on it: every object named, every constraint stated, nothing left to be inferred from context.

This is a skill worth practicing deliberately. In the rest of the course (and on exams) we will usually hand you the formal statement already, but in every setting outside a classroom, producing it might be your job.

(b) Here is the formalization we will use for the rest of the problem.

The input is a rooted tree T with m nodes: the root r is the top-level task, an internal node is a task that delegated and its children are its sub-tasks, and a leaf is a task that did its work directly. Each leaf carries a token count, these sum to n, and $s(v)$ denotes the total over the leaves below v, so $s(r) = n$. Write $c(v)$ for the number of children of v.

A summary is a parent-closed set X of internal nodes – the tasks you expand – meaning that if $v \in X$ and $v \neq r$ then the parent of v is also in X. It displays

$$D \;=\; \{r\} \;\cup \bigcup_{v \in X} \text{children}(v),$$

one line each, and is valid if $|D| = 1 + \sum_{v \in X} c(v) \leq k$. Its frontier is $F = D \setminus X$, the displayed tasks you did not expand.

Several measures capture the goal in the problem statement, and they do not always agree. We will use[3]

$$\Phi(F) \;=\; \sum_{v \in F} s(v)^2,$$

and your task is to find a valid summary minimizing it.

For this part only, assume in addition that $c(v) \leq 2$ for every node v – that is, T is binary. Give a dynamic program that computes $\min_F \Phi(F)$ under this restriction. Prove that it is correct, and analyze its running time. Your solution should run in time $O(mk^2)$.

Aligning Expectations – Using AI in This Part

Work this part out yourself. Unlike parts (c)–(e), this is not a part where we ask you to lean on a general AI assistant – if you want help getting unstuck, the AI-helper described in the instructions is designed for exactly this: it gives hints and pushes back instead of handing you the answer.

(c) Now drop the restriction from part (b): T may again have nodes of any degree, exactly as in the original problem statement. Generalize your dynamic program to this setting, so that it still runs in $O(mk^2)$ time overall.

You can either try to solve this part on your own, or you can use an AI assistant to help you. If you use an AI assistant, you should make sure you **understand** the solution it gives you, and you should write up the final answer in your own words.

Using AI Tips

Handing a derivation to an assistant and then writing it up yourself is a genuinely useful workflow, and it is also the one where it is easiest to fool yourself. A few things that help:

- **Validate what it gives you.** AI assistants can hand you something underspecified, or subtly incorrect, and say it with total confidence – don't take a derivation on faith just because it reads smoothly.
- **There is a more efficient $O(mk)$ solution, but it is considerably harder to**

[3] Briefly: the frontier totals always sum to n, so you cannot make every line small – all you control is how evenly the run is spread across them – and the sum of squares is the usual way to measure how even a fixed total is.

understand, and it is not the point of this question. If an assistant offers it to you, that is not what we are asking for here – steer it back toward the $O(mk^2)$ generalization instead.

- **Write your own prompts.** Describe the problem and your binary-tree algorithm in your own words rather than pasting in parts of the PSet PDF. Writing the prompt yourself forces you to actually understand what you are asking for, and keeps the assistant focused on generalizing your algorithm rather than solving the problem however it sees fit.

(d) On the course website you will find six input files contaiing traces of runs. Each file is a fully expanded outline like the examples above: two spaces of indentation per level, `[-]` for internal tasks, and `[ ]` for leaves. Only leaves show token counts; compute other tasks' totals from their descendants. There is no header, so m is the number of lines. Task labels are descriptive and need not be unique. This part uses the first four traces; leave traces 5 and 6 for part (e).

Have an AI assistant implement your algorithm from part (c) in whatever language you like, together with a harness that runs it on a given trace and budget, reports the summary and its value Φ, and times the run. Ask it to follow your algorithm rather than solving the problem its own way, and read what it gives you: you are responsible for the code being a faithful implementation of what you wrote down.

Run traces 1 and 2 at $k \in \{16, 64, 256\}$, and traces 3 and 4 at $k \in \{64, 256, 1024, 4096\}$. You may abandon a run after sixty seconds and record a timeout rather than inventing an answer.

Traces 3 and 4 may be too slow. If they are, **do not change the algorithm**: find ways to make the same algorithm cheaper to run, and keep the original version so you can compare. Report Φ and the running time for each trace and budget, say what you tried and roughly what each thing bought you, and confirm that the two versions agree wherever both finish.

(e) Now traces 5 and 6, at $k = 65{,}536$. The optimized DP takes roughly ten seconds on our reference machine, while simple greedy methods are much faster. For this part, develop a heuristic: something cheap that produces a good summary, with no guarantee that it is the best one.

Start by looking at the traces. Have an AI assistant write whatever code you need in order to measure them, and make observations about what they look like. Then design some heuristics that exploit what you found, and implement them.

Design an experiment to evaluate your heuristics without relying on the exact answers for traces 5 and 6. Work out what you can measure, what would convince you that one heuristic is better than another, and what such evidence does and does not establish. The point is to practice evaluating a method for instances where exact optimization would be too expensive (even though it is possible here depending on how efficient your optimized DP is).

The one hard rule is that a heuristic must produce an answer for traces 5 and 6 within ten seconds. Subject to that, make it as good as you can.

Report what you learned about the traces, what you tried, how you measured it, and what you found. You can have AI help you write up summaries of the results (like tables and plots), but you should write the text yourself, in your own words.

Problem 3 : A Unique Problem

Optional – Challenge Problem

This problem is optional and considerably harder than Problems 1 and 2 – we don't suggest attempting it just to practice for the exam. It's here for those who want to engage more deeply with the material; if you're considering an IW/thesis in CS theory, it'll give you a taste of what that might be like. Feel free to discuss it with your TA/UCA coach.

Do not use AI to solve this problem. This is a challenge problem, so if you want to work on it you should do so on your own.

You are given a sequence of non-negative integers $a_1, \ldots, a_n$. Your goal is to find if the sequence satisfies the following property: for every $1 \leq i \leq j \leq n$, the subsequence $a_i, a_{i+1}, \ldots, a_j$ contains at least one unique element. An element is unique in a subsequence if it appears exactly once in that subsequence. Design an $O(n \log n)$ algorithm to check if the sequence satisfies this property and prove its correctness and running time.