COS 330: Great Ideas in Theoretical Computer Science

Fall 2026

Problem Set 2

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 part (d). 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)–(c)** – 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 : Flow State 🗎

Consider a flow network $G = (V, E)$ with two distinct vertices $s, t \in V$. The capacity of an edge $(u, v) \in E$ is denoted $c(u, v)$. Assume that $|V| = n$ and $|E| = m$, capacities are nonnegative integers, and the graph is connected, with a directed path from s to t.

Definition 1.1 (Bottleneck Capacity)**.** The bottleneck capacity of a path P from s to t is the minimum capacity of any edge in P, i.e., $\min_{(u,v)\in P} c(u, v)$.

(a) Suppose that F is the value of the maximum flow in the network. Prove there is always a path with bottleneck capacity at least F/m.

(b) Design an $O(m \log m)$ time algorithm that finds a path from s to t with maximum bottleneck capacity.

(c) Consider the following modification of the Dinitz-Edmonds-Karp algorithm for computing maximum flows. Find an augmenting path from s to t with maximum bottleneck capacity (as in part (b)), augment flow along this path, and repeat until there are no more augmenting paths. Prove that this algorithm makes at most $O(m \log F)$ augmentations, where F is the value of the maximum flow in the network.

Small Hint

Repeatedly apply part (a) to the residual graph. At some point you will likely want to bound a quantity that shrinks multiplicatively by a factor like $(1 - 1/m)$ at each step, and turn that into a clean bound on the number of steps. The inequality $1 - x \leq e^{-x}$ for all x is exactly the tool for this: it lets you convert a multiplicative expression into an exponential one, which is much easier to solve for.

Problem Solving Tips

In general, when you have an inequality that's close to your target, pause and ask whether a familiar bound can bridge the gap. If you are not very familiar with common inequalities (which the typical COS 330 student is not expected to be), you might want to look up a catalog of inequalities (or ask AI to help you) and see if it applies.

Don't worry about memorizing a catalog: on exams we won't expect it, we'll provide any non-basic inequalities you might need (beyond very obvious ones like $x^2 \geq 0$).

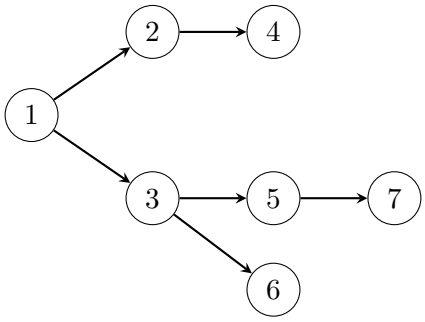
Problem 2 : Context Switching

A team of engineers is working on a large codebase. Each engineer's work arrives as a change: a self-contained edit that somebody else has to read and approve before it can go in. Changes are often built on top of one another – if your change edits code that my change has only just rewritten, then yours cannot be read, or approved, until mine has been.

Reviewing happens in sessions. In one session a single reviewer reads a sequence of changes back to back. They can move from a change A straight to a change B only when B was built directly on top of A: they already have A in their head, so B costs them very little extra. If B was built on something else, the reviewer would be starting from nothing, and B belongs to a different session.

Every change must be reviewed exactly once. Sessions are expensive: each one is a person sitting down and building up an understanding of an unfamiliar corner of the codebase from scratch. Use as few of them as you can.

For example, suppose there are seven changes, where change 2 and change 3 were both built on top of change 1, change 4 on top of 2, changes 5 and 6 on top of 3, and change 7 on top of 5:



Three sessions are enough: one reviewer takes $1, 2, 4$; another takes $3, 5, 7$; a third takes 6 on its own. It's not hard to see that there is no way to review all changes with only two sessions.

(a) Write down a formal statement of the situation above as a problem about graphs. Say precisely what the input is, what counts as a valid way of dividing the changes into sessions, and what you are minimizing.

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 the answer.

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

The input is a directed acyclic graph G with n nodes and m edges, one node for each change, with an edge $u \to v$ exactly when v was built directly on top of u. A session is a directed path in G; a single node on its own counts as a path. A schedule is a set of sessions such that every node of G lies in exactly one of them, and its cost is the number of sessions it contains.

Give an algorithm that computes the cost of a cheapest schedule. Prove that it is correct, and analyze its running time. You should be able to reach $O(n(n+m))$.

(c) Reviewers get tired, and a reviewer who has read a lot of changes in a row is not reading the next one carefully. So suppose no session may contain more than L changes, where L is a positive integer.

Write P for the cost of a cheapest schedule with no limit, and P_L for the cost of a cheapest schedule in which every session has at most L changes.

1. Prove that $P_L \geq \max(P, \lceil n/L \rceil)$.
2. Describe how to turn a cheapest unlimited schedule into one that obeys the limit, and prove that the schedule you get has cost at most $P + \frac{n-P}{L}$.
3. Conclude that the schedule from part (ii) always costs at most twice the cost of a cheapest schedule obeying the limit.

(d) On the course website you will find ten dependency graphs of varying sizes and structures, each with a session limit L, representing data from different code projects. You can assume each graph is a DAG with no self-loops or duplicate edges. The first line of each file contains three integers n, m, and L, followed by exactly m lines, each containing two integers u, v representing the directed edge $u \to v$. Nodes are numbered $1, \ldots, n$, but their IDs need not be in topological order.

Have an AI assistant implement your algorithm from part (b) and the construction from part (c)(ii), in whatever language you like, along with a script that runs them on a graph and reports the cost of the schedule it produces. Ask it to follow your algorithms 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.

Now, think of the schedule given by the algorithm from part (c)(ii) as your baseline: your job is to try to beat it. Design and implement heuristics that produce a valid schedule respecting the limit.

Look at the graphs before you write anything; have the assistant write whatever code you need in order to measure them. Try at least two ideas, and report failures as well as improvements. You do not need to beat the baseline on every input.

Report what you learned about the graphs, what you tried, how you measured it, and by how much you beat the baseline.

Problem Solving Tips

Note that you don't have to build new algorithms from scratch: you can either modify the flow-based algorithm from part (b)/(c)(ii) or take its output and rearrange it.

Also, part (c)(i) gives you a lower bound on P_L that you can actually compute, so you can bound how far from optimal a schedule is without knowing P_L exactly. Comparing your schedule's cost with the lower bound gives an upper bound on how much room there is for improvement. A large gap might reflect a loose lower bound rather than a poor schedule.

Problem 3 : Reuse Reduce Recycle

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.

In this problem you will be designing time and space efficient algorithms, so let's first look at a definition of a computational model to use.

Definition 3.1 (A Computational Model)**.** Suppose input is given as a read-only array of n integers, each of which can be stored in $O(1)$ space, and arithmetic operations as well as comparison on these integers take $O(1)$ time. Additionally assume that the sum of the absolute values of the n integers is an integer that can be stored in $O(1)$ space like the above. (This is a mild simplification of a well-known model called the word-RAM model.)

We can now give a definition of time and space complexity in this model.

Definition 3.2 (Time and Space Complexity)**.** We say that a problem on an input of size n is solvable in $\text{TISP}(t(n), s(n))$ if there is a single algorithm that solves the problem in $O(t(n))$ time and $O(s(n))$ space in the above computational model.

Note that in the above model, sorting in-place (e.g., using quicksort) takes $\text{TISP}(n \log n, n)$ time and space, since the input is read-only, so we still have to produce a new array to store the sorted output. Let's now consider a specific problem.

Definition 3.3 (k-SUM)**.** Given an unsorted list of n distinct integers, determine whether a k-element subset sums to zero.

Note that there is a trivial $\text{TISP}(n^k, k \log n)$ algorithm for k-SUM based on brute force search over all k-tuples of integers, but we can do better.

(a) Design a $\text{TISP}(n^{\lceil k/2 \rceil} \log n, n^{\lfloor k/2 \rfloor})$ algorithm for k-SUM (assume k is a constant).

The above algorithm is very efficient in time, but uses a lot of space when compared to the trivial brute force algorithm. So let's try to reduce the space complexity using a reduction approach.

Definition 3.4 (Grouping)**.** Given a list L of n distinct integers and a parameter g that divides n, we say that a grouping of the list is a partition into g groups $L_1, \ldots, L_g$ such that each group has size n/g and for all $a \in L_i$ and $b \in L_{i+1}$, $a \leq b$.

Definition 3.5 (Subproblem)**.** Given a grouping $L_1, \ldots, L_g$ of a list L, a subproblem is k-tuple of indices $(i_1, \ldots, i_k)$ where $1 \leq i_1, \ldots, i_k \leq g$. A subproblem is trivial if the smallest sum of k elements in the groups $L_{i_1}, \ldots, L_{i_k}$ is greater than 0, or if the largest sum of k elements in the groups $L_{i_1}, \ldots, L_{i_k}$ is less than 0.

(b) Show that the number of non-trivial subproblems for a grouping into g groups is $O(kg^{k-1})$. (Hint. Dilworth's theorem might be useful here.)

Use the above to obtain the following "self-reduction" for 3-SUM.

(c) Suppose there is an algorithm that solves 3-SUM in $\text{TISP}(t(n), s(n))$, then for any g, 3-SUM can be solved in $\text{TISP}(g^2(n + t(n/g)), n/g + s(n/g))$.

(d) Conclude that there is a $\text{TISP}(n^2 \log n, \sqrt{n})$ algorithm for 3-SUM.