

CS375 Problem Set 4

Due Dec 16th 10PM ET

A general note for CS375: When writing up your homework, please write neatly and explain your answers clearly, giving all details needed to make your answers easy to understand. Graders may not award full credit to incomplete or illegible solutions.

As a reminder, your solutions to these must be typed. You are encouraged to discuss the problems with your classmates, **but your submission should be your own**. When writing your solutions up, you should not be looking at any other student's submission.

Clear communication **is the point**, on every assignment. In general in CS375, unless explicitly specified otherwise, answers should be accompanied by explanations. Answers without explanations may not receive full credit. Please feel free to ask me any questions about explanations that might come up!

Necessary acknowledgements: many of these questions are reproductions or adaptations of questions of Eric Aaron's devising. Others come from CLRS, Erickson's text, or Kleinberg and Tardös' *Algorithm Design* (which is another fantastic algorithms text that I highly recommend perusing).

1. Our first problem comes to us from my friend who texted me the following the other day: "have two input arrays of guaranteed equal size, consisting of natural numbers. need to output a sorted array of the pairwise sum of the sorted lists in linear time. can it be done?" To clean that up a bit, I'm going to rewrite it as:

SUMSORT

INPUT: Two arrays A and B of equal length

OUTPUT: A single array C whose entries are defined as follows: letting $\bar{A}$ and $\bar{B}$ denote the sorted versions of A and B respectively, $C[i] = \bar{A}[i] + \bar{B}[i]$.

Show that if you could solve SUMSORT with a worst-case runtime of $f(n)$, then you can sort a single array with a runtime of $O(f(n) + n)$. More specifically, if you believe there is not a linear time algorithm for sorting, will you be able to find one for this problem? Why or why not? Be as specific as possible.

2. Next, we'll do a classic reduction as a warm-up for something a bit more convoluted. Recall the problem 3SAT from class:

3SAT

GIVEN: A boolean formula consisting of n variables $x_1, \dots, x_n$ and their negations $\neg x_1, \dots, \neg x_n$ such that the formula is a conjunction (logical AND ($\wedge$)) of clauses where each clause is a disjunction (logical OR ($\vee$)) of 3 variables or their negations.

OUTPUT: TRUE if there is an assignments to the variables that makes the formula evaluate to TRUE, or FALSE otherwise.

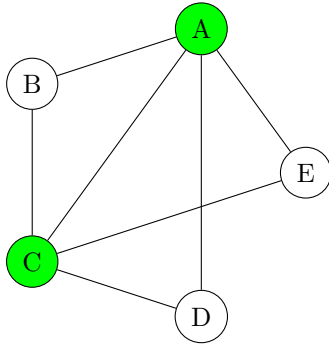
Recall that we found that finding a satisfying assignment was equivalent to finding an assignment such that at least one variable evaluates to TRUE in each clause. Also recall from class that 3SAT is NP-COMplete. Now, let's think about the problem VERTEXCOVER:

VERTEXCOVER

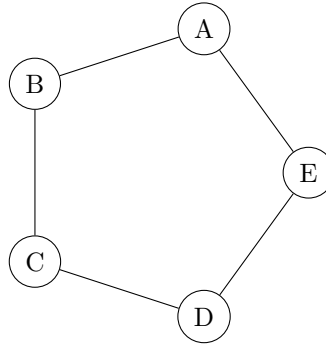
GIVEN: A graph $G = (V, E)$ and a parameter k

OUTPUT: TRUE if there is a collection S of k vertices in the graph such that every edge has an endpoint in S , FALSE otherwise.

As an example,



Graph 1: Vertex cover of size 2



Graph 2: No vertex cover of size 2

- Show that VERTEXCOVER is in NP.
- Show that $3SAT \leq_p VERTEXCOVER$.
- Suppose I have an algorithm that has a worst case runtime of $f(|V|, k)$ for VERTEXCOVER: what is the runtime of the algorithm for 3SAT in terms of the number of variables and clauses?

The next problem is based on https://en.wikipedia.org/wiki/Wolf,_goat_and_cabbage_problem.

- Imagine we have a collection of n items to transport across a river and a boat that can hold ourselves plus k items, under the constraint that certain items cannot be left on the same side of the river as each other. Show that determining whether we are able to get all the items across the river is NP-HARD.

Hint: You're free to reduce from whatever you want, but I think that VERTEX COVER might be the most straightforward?

For extra credit, show that the similar problem from the first problem set (where we have a collection of n people, each person has a crossing time, and a single flashlight) is NP-HARD, or design a polynomial time algorithm for the problem.

The rest of the problem set is based around investigating the online k -cache problem. For this problem, imagine we are writing an algorithm overseeing a memory cache in a computer which our client is interacting with: sequentially the client will request specific “pages” of memory $r_1, r_2, \dots$. If the requested page is not in our memory cache at the moment of its request, we suffer a “miss” and must add it to our memory. However, we can only store k pages in our cache: if we are to add another page when we are already storing k pages, we have to evict a page. Our goal is to minimize how many misses we suffer.

Example: Suppose our request sequence is A, B, A, C, A and $k = 2$. Then the algorithm suffers two misses immediately for the first two requests, after which we have a memory cache of $[A, B]$. When A is requested again, we do not have a miss since A is still in our cache. When C is requested, we have to evict a page: suppose we evict A . Then when A is requested one last time, we suffer another miss since we have evicted it for C .

We are now going to consider this problem from an *online* setting: we can only base our decision of which pages to evict based on everything we've seen so far, and not what will come in the future.

- Convince me that no algorithm can expect to perfectly match the optimal solution on every input. That is, for every online algorithm, there is an input on which it incurs more misses than the optimal algorithm.

Knowing this, we still want to rank our algorithms by how well they perform: let $C(A, I)$ denote the number of misses an algorithm A incurs on input I and $C(\text{OPT}, I)$ the optimal number of misses on input I . We say A is $\rho(A)$ -competitive if there is some constant c so that for all possible inputs I we have that $C(A, I) \leq \rho(A)C(\text{OPT}, I) + c$ (in English, for every possible input the cost of our algorithm on that input is at most a factor of $\rho(A)$ worse than the optimal cost on that input plus some constant).

5. Consider the algorithm LFU that always evicts the page that has been requested least overall so far. Convince me that for any fixed cache size k we have that $\rho(\text{LFU}) \geq x$ for every number x (an immediate consequence of this is that there's no way to upper bound the competitive ratio of this algorithm as a function of just k).
6. Convince me that any algorithm must have competitive ratio at least k .
7. Let's try to break a sequence of pages into phases: each phase is the maximal length sequence of page requests since the previous phase that contains at most k distinct page requests. Let's call an algorithm "good" if in any phase it only evicts pages not yet seen in the current phase. Convince me that any "good" algorithm has competitive ratio $O(k)$.
8. Consider the algorithm that uses phases as defined as above: this algorithm only evicts pages not requested in this phase, chosen arbitrarily. Convince me that this algorithm is 'good' as defined above.