CS 330, Fall 2024, Homework 2 Solutions
Due Wednesday, September 25, 2024, 11:59 pm EST, via Gradescope

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (including generative AI tools or anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must provide:

1. pseudocode and, if helpful, a precise description of the algorithm in English. As always, pseudocode should include
   - A clear description of the inputs and outputs
   - Any assumptions you are making about the input (format, for example)
   - Instructions that are clear enough that a classmate who hasn't thought about the problem yet would understand how to turn them into working code. Inputs and outputs of any subroutines should be clear, data structures should be explained, etc.

   If the algorithm is not clear enough for graders to understand easily, it may not be graded.
2. a proof of correctness
3. an analysis of running time and space .

You may use algorithms from class as subroutines. You may also use facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions. A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

Problem 1 Price contest with a blackbox algorithm (10 points)

You are a contestant on a TV show, and Congratulations, you won! As your price you get to choose what to take home. Specifically, there are n objects on display, each with a retail value $[a_1, a_2, \ldots, a_n]$, you are also given a budget B. You can choose any subset of the items as long as their total value is within budget. Your goal is to pick a subset of total value exactly B.

In this problem we do NOT ask you to solve this problem from scratch [1]. Instead, we want you to solve it through a polynomial-reduction style approach. Specifically, suppose that you have access to a blackbox algorithm $CanSolve(\ )$. The algorithm takes as input a list of integers $[x_1, x_2, \ldots, x_k]$ and a target value T. The output is $CanSolve([x_1, x_2, \ldots, x_k], T) = True$ if and only if there is a subset of the x_i values, such that there sum is exactly T. Otherwise it returns $False$.

Design an algorithm, that using $CanSolve(\ )$ returns the set of items that you should select, if there is a subset with total value exactly B, otherwise return "not possible". In your running time analysis use $O(C)$ to represent the runtime of $CanSolve(\ )$. The algorithm should take $[a_1, a_2, \ldots, a_n]$ and B as input.

Solution.

Algorithm. The key to this algorithm is to recursively decide for each a_i whether it should be part of the output. The way it's decided is to verify using $CanSolve(\ )$ whether there is a solution that includes a_i.

Algorithm 1: PriceContest$([a_1, a_2, \ldots, a_n], B)$

```
/* [a_1, a_2, ..., a_n] is an array of ints, B is an int                    */
if CanSolve([a_1, a_2, ..., a_n], B) == False then
    return "not possible"
price ← empty list/* contains the items to be returned                      */
i ← 1;
while B > 0 do
    if CanSolve([a_{i+1}, a_{i+2} ..., a_n], B) == False then
        /* due to the previous iteration we know that CanSolve([a_i, a_{i+1} ..., a_n], B) = True hence
           we don't check                                                   */
        price.append(i)/* item i is part of the optimal set                */
        B ← B − a_i/* update budget to reflect the selection of i           */
    i ← i + 1/* move to the next item                                       */
return price
```

Running time and space complexity. The algorithm makes $O(n)$ calls to $CanSolve(\ )$, hence the total running time is $O(nC)$. The algorithm only creates an array of length $O(n)$, hence its space complexity is $O(n)$.

[1] You will NOT receive credit for any algorithm that doesn't use CanSolve( ) in a meaningful way.

Correctness. From the initial call to $CanSolve(\ )$ we know whether there is a solution. It also tells us that the algorithm will always terminate since the While loop will make at most one call to $CanSolve(\ )$ for each item.

We prove correctness by induction on the number of items. If there is only one item and there is a solution, then $a_1 = B$ and ${}_a1$ will be added to the output.

Suppose that up to iteration k, i.e. item a_k, the algorithm correctly assigns values to the output and correctly updates the value of B. Let B' be the remaining budget after iteration k.

In iteration $k+1$, if $CanSolve([a_{k+1}, \ldots, a_n], B') == True$ but $CanSolve([a_{k+2}, \ldots, a_n], B') == False$ that implies that any solution has to contain a_{k+1}. Thus, $k+1$ is correctly added to the output. Since $k+1$ is selected, it means that it will contribute a_{k+1} amount to the total cost. Thus the remaining items have only $B' - a_{k+1}$ budget available. This is also correctly updated. If a_{k+1} is not selected, then it doesn't contribute to the total and B' should remain.

Problem 2 Vaccine testing (10 points)

You are developing a new vaccine and are ready for human trials! For efficiency, you want to keep your trial as small as possible. Each person has some health-characteristics, e.g. age, weight, disease history, etc. Your goal is to select a subset of individuals to receive vaccines, such that each person's set of characteristics are well represented in your trial.

For privacy reasons you don't have access to the actual medical data of the population. Instead, you are given for every two persons p_i and p_j their distance $d(p_i, p_j)$. Distances are symmetric. (You don't have to worry about how exactly the distances are computed.) You may assume that the data is given to you in the input format of a 2D-table D, such that $D[i][j] = d(p_i, p_j)$.

Given a target value T, we say that a set of trial participants is *representative* of the population if for each person p there is a trial participant with distance at most T from p.

In this problem you have access to a blackbox algorithm called $SCSolver(\ )$. This algorithm solves an instance of the minimum Set Cover problem; given a universe $U = \{u_1, u_2, \ldots, u_n\}$ of n items and given m subsets $S = \{S_1, S_2, \ldots S_m\}$ of U, it returns a minimum number of sets in S, such that each item in U is covered.

Design an algorithm for the Vaccine Testing problem that makes calls to $SCSolver(\ )$. The input to your algorithm is the distance table D and the max distance value T. The output should be a minimum representative set of individuals. In your runtime analysis you may assume that $SCSolver(\ )$ takes $O(C)$ time.

Solution.

Algorithm. We will turn the VT problem into an instance of SC. The universe U will consist of the people in the population. We will create one set S_i corresponding to each person p_i. The set will consist of all the individuals that p_i can represent. That is, people who are within distance T. We then feed this input to the $SCSolver(\ )$. The sets returned by the blackbox then correspond to the people in the trial.

Algorithm 2: Trials(D, T)

```
   /* D is an n × n table of pos numbers, T is a positive number                    */
 1 U ← {0, ... n − 1} /* universe of people                                          */
 2 S ← empty list/* list of S_i s                                                    */
 3 for i = 0 to n do
 4 |   S[i] ← empty list;
 5 |   for j = 0 to n do
 6 |   |   if D[i][j] ≤ T then
 7 |   |   |   S[i].add(j);
   |   |   |
   |   |   |
   |   |
   |
 8 reps ← SCSolver(U, S)/* returning the sets will get you full credit already     */
   /* returning the actual persons is what the question asked for.                 */
 9 reps2 ← empty list;
10 for S_i in reps do
11 |   reps2.add(i);
12 return reps or reps2
```

Running time and space complexity: To create the sets S_i the algorithm traverses every row of D, which results in $O(n^2)$. Then one call to SCSolver is made. Comuting the actual representatives iterates once over a list of $o(n)$. The final running time is $(O(n^2 + C)$. The input to the algorithm is an $O9N^2)$ table. In worse case the sets created each take $O(n)$ space, which is another $O(n^2)$ factor. Finally, the output consists of a subset of these sets, which doesn't result in extra space. The total space complexity is $O(n^2)$.

Correctness: To create the sets S_i we exhaustively check the distance of every person to p_i, hence the resulting set contains the people that p_i represents. SCSolver then selects a min set of subsets, which by construction of the sets corresponds to a min number of individuals covering every person in the database.