This homework is due on **September 11, 11am ET**.

Submit solutions to coding questions (Problem 1 and Problem 3.e) in Stepik, submit solutions to all other questions in Gradescope. You can typeset your solutions in LaTeX (for example, using this template: https://www.overleaf.com/read/gjdgtxxpwkkj).

Please sign up at Stepik using the following link:
https://stepik.org/invitation/87519dc4727a0dbfd908e132a03323e5cd8b9fb2/

You can resubmit your code until it passes all tests, there is no limit on the number of attempts. You can submit your solutions in any of the following programming languages `ASM32, ASM64, C, C#, C++, Closure, Dart, Go, Haskell, Java, Javascript, Julia, Kotlin, NASM, Octav, PascalABC.NET, Perl 5, PHP, Python 3, R, Ruby, Rust, Scala, Shell, Swift`, however starter files will be provided only for `Python, Java`, and `C++`.

You can work in groups of two or three people, however you must explicitly list all collaborators and materials that you used. You must write up your own solution and your own code to every problem. See Georgetown University Honor System. When in doubt, ask the instructor what is allowed.

Problem 1 (Car Fueling)**.** The distance between the cities A and B is d miles. A car can go for at most m miles with a full tank. There are n gas stations located at distances $x_1, \ldots, x_n$ from A along the path. The car leaves A with a full tank, compute the minimum number of gas tank refills needed to drive from A to B.

Input: Each of the first three lines of the input contains one positive integer: d—the distance between the two cities; m—the range of the car; n—the number of gas stations along the way. The fourth line of the input contains n integers $x_1 \; \ldots \; x_n$, where $0 \leq x_1 \leq \ldots \leq x_n \leq d$, that define the locations of gas stations.

Output: The output should contain one integer. If the car cannot reach B from A, then output -1, otherwise output the minimum number of stops needed to drive from A to B.

Example 1:

Input:

```
1000
300
5
200 300 550 650 750
```

The distance between A and B is 1000 miles, and the car can drive for 300 miles with a full tank. It suffices to make three refills: at distance 300, 550, and 750.

Output:

```
3
```

Example 2:

Input:

```
1000
300
5
200 300 400 550 600
```

The distance from the last gas station to B is 400 miles, while the car can travel at most 300 miles with a full tank.

Output:

```
−1
```

Problem 2 (k-center)**.** Clustering is one of the key problems in machine learning and data science. Given a set of objects, the task is to partition these objects into groups such that objects within a group are more similar. In this exercise, we'll design a 2-approximate algorithm for a specific clustering task called k-center.

Given n vertices in the plane and a positive integer k, we'll choose k cluster centers among the given n vertices. After that, every vertex will be assigned to its closest cluster center. The radius of such clustering is the maximum distance of a vertex to its center. In the k-center problem, the goal is to find k cluster centers that minimize the clustering radius.

Consider the following greedy algorithm for this problem. First, pick any vertex as the first cluster center. Pick each of the remaining $k-1$ cluster centers in the greedy way: choose a vertex whose minimum distance to the already selected centers is maximum.

Prove that the above algorithm is 2-approximate. In other words, prove that the radius of the clustering found by the greedy algorithm is at most two times larger than the optimal clustering radius. For this, consider two cases:

- Case 1: the greedy algorithm picked one cluster center from each cluster of an optimal solution.
- Case 2: the greedy algorithm picked at least two cluster centers from some cluster of an optimal solution.

Problem 3 (2-SAT)**.** An instance of the k-SAT problem is a formula with n Boolean variables and m clauses. Each clause is a disjunction (OR) of at most k variables or their negations. The task is to decide whether one can simultaneously satisfy all clauses.

While k-SAT is **NP**-hard for $k \geq 3$, it admits efficient algorithms for the case $k \leq 2$. In this exercise we will develop an efficient (linear-time) algorithm for 2-SAT.

a. Show that 1-SAT can be solved in linear time.

Given a 2-SAT formula ϕ with n Boolean variables, construct a directed graph with $2n$ vertices where each vertex corresponds to a variable or its negation. Now for each clause $(x \vee y)$, add directed edges $(\neg x, y)$ and $(\neg y, x)$. (Think of these edges as follows. The edge $(\neg x, y)$ means that if $x = 0$ then y must be 1 in order to satisfy this clause. Similarly, the edge $(\neg y, x)$ indicates that if $y = 0$ then x must be 1.)

Recall that a directed graph is called strongly connected if every vertex is reachable from every other vertex. Every directed graph can be partitioned into strongly connected components (maximal subgraphs that are strongly connected). Moreover, such a partition can be found efficiently (See [Eri19, Sec 6.5, 6.6]). In the following you can use an algorithm that efficiently finds strongly connected components of a graph.

b. Prove that if this graph contains a strongly connected component containing x and $\neg x$, then ϕ is unsatisfiable.

c. Prove the converse: if no strongly connected component contains a variable and its negation, then ϕ is satisfiable.

d. Use the above to design an algorithm that takes as input a 2-SAT formula and decides whether it's satisfiable or not. What is the running time of your algorithm?

e. Extra credit. Implement this algorithm. If the programming language of your choice has a function for computing strongly connected components of a graph (such as `networkx.strongly_connected_components` in `Python`), feel free to use it.

Input: The input represents an instance of 2-SAT. The first line contains two numbers, n and m, which denote the number of variables and the number of clauses in the formula. Each of the following m lines represents a clause of the formula by two integers i and j. A positive integer $1 \leq i \leq n$ denotes the variable x_i, and a negative integer $-n \leq i \leq -1$ denotes its negation $\neg x_i$ (the same rules apply to j).

Output: If the input formula is satisfiable, then the program should output the word "SATISFIABLE" (in capital letters, without quotes), otherwise it should output "UNSATISFIABLE".

Example 1:

Input:

```
3 4
1 2
1 −2
−1 3
2 −3
```

This input represents the formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$. All clauses of this formula are satisfied by the assignment $x_1 = x_2 = x_3 = 1$.

Output:

```
SATISFIABLE
```

Example 2:

Input:

```
3 5
1 2
−1 −2
1 −2
−1 3
2 −3
```

This input corresponds to the formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$. We showed in class that this formula is unsatisfiable.

Output:

```
UNSATISFIABLE
```

Problem 4 (**Extra credit.** Yet Another Algorithm for 2-SAT)**.** In this exercise, we will consider two other polynomial-time algorithms for 2-SAT.

First, we'll show that an elementary branching algorithm solves 2-SAT in polynomial time.

a. Let *Simplify* be the following procedure. It takes a 2-SAT formula, iteratively assigns 1 to all literals appearing in 1-clauses. *Simplify* either returns *UNSAT* (if it assigns 1 to a literal and its negation) or returns a formula without 1-clauses. Show that *Simplify* can be implemented in polynomial time, and that it always returns a subset of clauses of the original formula.

b. Consider the following recursive algorithm for 2-SAT. Given a formula ϕ, pick an arbitrary variable x_i, and let ϕ_0 and ϕ_1 denote the formula ϕ after the substitutions $x_i = 0$ and $x_i = 1$, respectively. The algorithm first applies *Simplify* to ϕ_0 and gets ψ_0.

If $\psi_0 \neq \mathit{UNSAT}$, then the algorithm recursively calls itself on ψ_0. If $\psi_0 = \mathit{UNSAT}$, then the algorithm applies *Simplify* to ϕ_1 and gets ψ_1. If $\psi_1 = \mathit{UNSAT}$, then the algorithm concludes that the formula is unsatisfiable. Otherwise the algorithm calls itself on ψ_1. Prove that this algorithm can be implemented in polynomial time.

c. Prove that this algorithm solves 2-SAT correctly.

Now we'll prove that a random walk algorithm solves 2-SAT correctly in polynomial time.

d. Consider the following algorithm for 2-SAT. Pick a random assignment $x \in \{0,1\}^n$. If x doesn't satisfy the given 2-CNF formula ϕ, then pick an arbitrary unsatisfied clause of ϕ, and flip the value of a random variable from this clause. Let's denote the new assignment by $y \in \{0,1\}^n$. Recursively repeat the previous step with the assignment y: if y doesn't satisfy ϕ, then flip the value of a random variable appearing in a fixed unsatisfied clause, and repeat the process with the new assignment.

For the analysis of this algorithm, fix an arbitrary assignment $a \in \{0,1\}^n$ satisfying ϕ. Suppose that our algorithm starts with an assignment x such that the Hamming distance between x and a is $0 \leq d \leq n$. Assume that our algorithm never meets satisfying assignments other than a. What is the probability that in one step, the algorithm decreases the distance between the current assignment and the assignment a?

e. Let t_i be the expected number of steps needed to reach a starting from an assignment on distance i from it. Find a recurrence relation for t_i. Show that all t_i are bounded by a fixed polynomial.

f. Conclude that there exists a polynomial time randomized algorithm that solves 2-SAT with probability $1 - n^{-10}$.

Problem 5 (**Zero credit.** Approximate TSP)**.** See an implementation of the 2-approximate algorithm for Euclidean TSP here:
https://colab.research.google.com/drive/1QUOHPFr2ninu8Bip3qRY5OCn9O4Fiss0?usp=sharing