Problem 1: Load Balancing

Recall that in the load balancing problem you are given a collection of tasks with running time $t[1], \ldots, t[n]$, and a number of machines $M_1, \ldots, M_m$, and the goal is to assign a collection of tasks $A[i]$ to each machine $M_i$ so that all tasks get executed, and the maximum running time $\max_i T[i]$ (where $T[i] = \sum_{j \in A[i]} t[j]$) is minimized. The textbook (section 11.1) gives the following greedy algorithm and proves that it approximates the load balancing problems within a factor 2.

Greedy-Balance:
Set $T[i] = 0$ and $A[i] = \emptyset$ for all machines $M_i$
For $j = 1, \ldots, n$
    Let $M_i$ be the machine that achieves the minimum $\min_k T_k$
    $A[i] \leftarrow A[i] \cup \{j\}$
    $T[i] \leftarrow T[i] + t[j]$
Output $A$.

Prove that the algorithm actually achieves a slightly better approximation factor $2 - 1/m$, where $m$ is the number of machines.

Problem 2: Center Selection

In class we considered the following two algorithms for the centered selection problem. For simplicity, we assume that the input to both problems is a collection of points $S \subset \mathbb{R}^2$ on the plane with the standard Euclidean distance, though everything we will do can be easily generalized to arbitrary metric spaces. On input $S$ and an integer $k$, the goal is to find a collection of $k$ centers $C \subset \mathbb{R}^2$ such that the maximum distance (or covering radius)

$$r(C, S) = \max_{x \in S} \min_{c \in C} \|x - c\|$$

is as small as possible. Both algorithms below choose $C \subseteq S$ as a subset of $S$ (of size $|C| = k$), but this is not required, and the optimal solution may use arbitrary points for the centers.

<table>
<thead>
<tr>
<th>Algorithm 1: $C = \emptyset$</th>
<th>Algorithm 2: $C = \emptyset$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while $</td>
<td>C</td>
</tr>
<tr>
<td>Select any $x \in S \setminus C$ such that $r(C \cup {x}, S)$ is as small as possible</td>
<td>Select any $x \in S \setminus C$ such that $\text{dist}(x, C) = \min_{c \in C} |x - c|$ is as large as possible</td>
</tr>
<tr>
<td>$C \leftarrow C \cup {x}$</td>
<td>$C \leftarrow C \cup {x}$</td>
</tr>
<tr>
<td>Output $C$</td>
<td>Output $C$</td>
</tr>
</tbody>
</table>
The textbook proves that Algorithm 2 achieves an approximation factor 2, that it, it always output a cover \( C \) such that \( r(C, S) \) is at most twice the optimal value. In this problem you are asked to investigate Algorithm 1. Notice that Algorithm 1 differs from the one described on page 607 of the textbook because here we require \( C \subseteq S \). In particular, the counterexample presented in the textbook showing that the algorithm can be arbitrarily bad does not apply to Algorithm 1.

This problem involves an “optional” programming component. “Optional” means that it is intended to guide you in the solution of the problem. If you can figure out the answer and provide a theoretical analysis of the algorithm without running any experiments, that’s fine. In any case, you are not required to submit any code. Describing the details and results of your tests is all we expect for the experimental part.

**Part (a)** Implement both algorithms above, and test them on randomly chosen inputs. E.g., you may set \( S \) by generating \( n \) points chosen at random on an \( n \times n \) grid, and then run both algorithms for different values of \( k \). Describe your tests (how many tests did you run? for what values of \( n \) and \( k \) etc.) and answer the following questions:

1. Is Algorithm 1 always at least as good as algorithm 2?
2. Is Algorithm 2 always at least as good as algorithm 1?

**Part (b)** Provide an input instance where Algorithm 1 produces a better solution than Algorithm 2, or prove that no such input exist.

**Part (c)** Provide an input instance where Algorithm 2 produces a better solution than Algorithm 1, or prove that no such input exist.

**Part (d)** We know that Algorithm 2 always achieves an approximation factor \( \gamma \leq 2 \). Determine if Algorithm 1 is also a good approximation algorithm, in the sense that it always output a solution \( C \) such that \( r(C, S) \leq \gamma \cdot OPT \), where \( OPT \) is the optimal solution and \( \gamma \) is some constant. What is the smallest \( \gamma \) for which you can prove the approximation bound? Can you give a matching counterexample, showing that there are input instances such that Algorithm 1 outputs a solution with \( r(C, S) = \gamma \cdot OPT \)? Alternatively, can you prove that Algorithm 1 is not a good approximation algorithm, i.e., show that for any constant \( \gamma \) there is an input instance \((S, k)\) such that Algorithm 1 output a solution \( C \) with \( r(C, S) \geq \gamma \cdot OPT \)?

**Problem 3: Hitting Set (KT 11.4)**

Consider an optimization version of the Hitting Set Problem defined as follows. We are given a set \( A = \{a_1, \ldots, a_n\} \) and a collection \( B_1, \ldots, B_m \) of subsets of \( A \). Also, each element \( a_i \in A \) has a weight \( w_i \geq 0 \). The problem is to find a hitting set \( H \subseteq A \) such that the total weight of the elements in \( H \), \( \sum_{a_i \in H} w_i \), is as small as possible. (As in Exercise 5 in Chapter 8, we say that \( H \) is a hitting set of \( H \cap B_i \) is not empty for each \( i \).) Let \( b = \max_i |B_i| \) denote the maximum size of any of the sets \( B_1, \ldots, B_m \). Give a polynomial-time approximation algorithm for this problem that finds a hitting set whose total weight is at most \( b \) times the minimum possible.