- Always try to give algorithm with best possible running time. The points that you obtain will depend on the running time of your algorithm. For example, a student who gives an $O(n)$ algorithm will receive more points than a student who gives an $O(n^2)$ algorithm.
- You are required to give proofs of correctness whenever needed. For example, if you give a greedy algorithm for some problem, then you should also give a proof why this algorithm outputs optimal solution.
- **Use of unfair means will be severely penalized.**

There are 3 questions for a total of 40 points.

---

(10) 1. Given two n bit strings S and T and a $2n$-bit string Z, design an algorithm that determines if there are indices $1 \leq i_1 < i_2 < ... < i_n \leq 2n$ and $1 \leq j_1 < j_2 < ... < j_n \leq 2n$ such that

1. $Z_{i_1} = S_1, Z_{i_2} = S_2, ..., Z_{i_n} = S_n$ and $Z_{j_1} = T_1, Z_{j_2} = T_2, ..., Z_{j_n} = T_n$, and
2. $\{i_1, i_2, ..., i_n\} \cap \{j_1, j_2, ..., j_n\} = \emptyset$.

If such indices exists, then your algorithm should also output the indices.

(15) 2. The d-dimensional k-median clustering problem is defined as follows: Given n points $x_1, ..., x_n \in \mathbb{R}^d$, output k points (these are called *centers*) $c_1, ..., c_k \in \mathbb{R}^d$ such that the following objective function is minimised:

$$\sum_{i=1}^{n} \left(\min_{1 \leq j \leq k} D(c_j, x_i) \right)$$

where $D(x, y)$ denotes the Euclidean distance of points x and y. Design an algorithm for the 1-dimensional k-median problem.

(15) 3. Consider the Knapsack problem that we discussed in lecture. There are n items of weight $w_1, ..., w_n$ and there is a Knapsack that can hold a weight of at most W. The goal is to find a subset of items to fill in the knapsack such that the total weight of items in the knapsack is maximised subject to the total weight being at most W. We will assume that all weights are integers.

We studied a dynamic program for this problem that has a running time $O(n \cdot W)$. We know that this is bad when the integers are very large. The goal in this task is to improve the running time at the cost of optimality of the solution. Let us make this more precise. Let OPT denote the value of the optimal solution. For any given $\epsilon \in (0, 1]$, design an algorithm that outputs a solution that has value at least $\frac{OPT}{(1+\epsilon)}$. The running time of your algorithm should be of the form $O\left(\frac{f(n)}{\epsilon}\right)$, where $f(n)$ is some polynomial in n.

You need not give the best algorithm that exists for this question. As long as $f(n)$ in the running time is polynomial, in n, you will get the full credit.

Hint: Try modifying the dynamic programming solution for the knapsack problem.