DS8001/ME8201 - Design of Algorithms and Programming for Massive Data

Assignment # 2

Out September 29

Due October 16, **9:00AM**

Late submissions will not be accepted!

Solve the following problems. You may work in *groups* of 1-2 students. You *must* cite any references (texts, papers or websites) you have used to help you solve these problems (*you do not need to do this for the programming questions e.g. when you check a function name over the internet, however, you* ***cannot*** *copy/use someone else's code without a reference*). For the programming questions submit a single *python* file that contains your solution in the respective assignment submission folder in D2L (**jupyter notebook submissions will not be accepted!**). Assignment submission folders are under Assignments in the Assessment tab.

- Pay attention to the following guidelines before you submit your answers:
  - **Each submitted file** ***must*** **include the name of the group members at the top of the file (e.g., in a python file this can be added as a docstring to the top of the file)**.
  - Each submitted file *must* have the following naming convention (replace the terms questionID, "studentLastName" and "studentName"):

    hw2_questionID_studentLastName_studentName(.py)

    For instance, for the part (a) of question 1, the naming convention would be

    hw2_1a_Cevik_Mucahit.py
  - If done in groups, **each group member should submit** the homework separately.
  - In D2L submission folder, specify the name of the group member(s)!!!

Not following the submission guidelines will result in point deductions!

1. (**implementation**, 10 pts) Assume that you are given a sorted array of distinct integer values $A[0,\dots,n-1]$, and your task is to figure out whether there is an index i for which $A[i] = i$.

   - Devise a naive algorithm that performs this task in a brute force manner (i.e., with $O(n)$ run time). Implement this algorithm in a Python function named `GetPointIndex_naive()`.
   - Devise a divide-and-conquer (DC) algorithm for this problem, which has $O(\log n)$ run time. Implement this algorithm in a Python function named `GetPointIndex_DC()`.

   See below code snippets for sample function call along with expected inputs/outputs (`run_time` corresponds to the algorithm run time in seconds for the given input).

```
A = [-3,0,1,5,7,9,11] # should return False
bool_val , run_time = GetPointIndex_naive(A) # bool_val=False , run_time=0.000063
B = [-3,0,2,5,7,9,11] # should return True because B[2] = 2
bool_val , run_time = GetPointIndex_DC(B)
```

2. (**implementation**, 20 pts) A substring *ss* of a string is considered to be *special* if every letter of the alphabet that *ss* contains shows up in *ss* both in lower case and upper case forms. For instance, $s =$ "*cdCDD*" is special because both lower and upper case forms of "c" and "d" appears in the string. On the other hand, $s =$ "*cdC*" is not special (due to missing "D").

   - Devise a divide-and-conquer (DC) algorithm with to find the longest substring *ss* in a given string *s*. Implement a **recursive** Python function named `GetLongSpecialSubstring_DC()` to perform this task. You can assume that $1 \leq \texttt{len}(s) \leq 100$, and *s* only contains upper and lower case from English language. (Hint: there exists an $O(n^2)$ DC algorithm for this task).

   See below code snippet for the expected inputs/outputs of this function.

```
ss = GetLongSpecialSubstring_DC("FdedDdf") # ss = 'dDd'
# if input is "c", output must be "" -- i.e., return empty string if no special ss exists
```

3. (**implementation**, 20 pts) *Minimum node-cover* problem can be described as, given an undirected graph $G = (\mathcal{V}, \mathcal{E})$, finding the smallest subset $\mathcal{S} \subseteq \mathcal{V}$ such that each edge in $\mathcal{E}$ is incident to at least one node of $\mathcal{S}$. Devise a *greedy algorithm* to find the node cover for any given undirected graph. You can assume that you will be given only the adjacency matrix corresponding to an undirected graph, which shows whether there is an edge between any two nodes in the graph. A sample function call and the expected output is provided below.

   - **Hint**: Since this is a greedy algorithm, it may not generate the optimum (i.e., minimum-sized) node-cover, however, any generated solution by the greedy algorithm is still expected to be a node-cover.

```
# nodes/vertices = {1,2,3,4,5}
np_adjacency = [[0, 1, 0, 1, 0],
                [1, 0, 1, 1, 0],
                [0, 1, 0, 1, 0],
                [1, 1, 1, 0, 1],
                [0, 0, 0, 1, 0]]
list_nodes = GetNodeCover_greedy(np_adjacency)
# node cover solution 1: list_nodes = [2,4]
# node cover solution 2: list_nodes = [1,3,4]
```

4. (**implementation**, 10+20=30 pts) It is well known that for any instance of the stable matching problem, a stable matching exists, and the Gale-Shapley algorithm returns one such matching. However, the Gale-Shapley algorithm does not provide much insight into the other stable matchings which may exist. In this question, we aim to find the set of *all* stable matchings. A natural approach would be to (i) enumerate all possible matchings, and then (ii) check the stability of each matching. Part (a), (b) and (c) below are designed to achieve this task in consecutive steps.

   (a) Consider an instance with n men and n women. Observe that fixing the order of men as $[1,2,\ldots,n]$, a matching can be defined by the order of the women. For instance, if we have three men whose order is fixed to [1,2,3] and we order three women as [2,3,1], we will be describing the matching $\{(\text{man1},\text{woman2}),(\text{man2},\text{woman3}),(\text{man3},\text{woman1})\}$. Therefore, we can find all possible matchings between men and women by simply fixing the order of men as $[1,2,\ldots,n]$ and finding all possible orderings of women.

   The pseudocode below provides a function that can be used to enumerate all possible ordering of women. For instance, executing `Orders(All, [1,2,3], [])` for an empty array `All = []` would yield

   `All = [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]].`

```
# pseudo code
function Orders(Array,Start,End):   # similar to Python syntax
 if Start is empty:
  Add End to Array
 else:
  n = length of Start
  for i = 0, 1, ... , n-1:
   NextEnd =  End + Start[i:i+1]        # add item at index i to End
   NextStart =  Start[:i] + Start[i+1:] # remove item at index i from Start
   Orders(Array, NextStart, NextEnd)
```

   Implement a Python function, `Orders()`, for the `Orders` algorithm provided as in above pseudocode. Note that this function should not have any return value.

   (b) Given a list of all matchings between n men and n women, along with the mens' and women's preferences, your task is to devise an algorithm that returns the number of stable matchings. Consider the stable matching instance with 7 men and 7 women (each group indexed by 0 to 6) defined by the preference matrices below. Starting indices by 0, row i of MenPreferences (WomenPreferences) provides the list of woman (man) indices in the preference order of man (woman) indexed by i, from the most preferred to the least. For example, the first man, indexed by 0, prefers the woman indexed by 5 the most and the woman indexed by 1 the least (see below code snippets). Provide a python function, named `FindAllStableMatchings()` that returns the total number of stable matchings. (Hint: All matchings between n men and n women can be calculated inside the function, so it does not need to be passed as an input).

```
MenPref =  [[5, 6, 4, 2, 0, 3, 1],
            [0, 5, 1, 2, 4, 3, 6],
            [3, 6, 1, 2, 5, 0, 4],
            [1, 2, 5, 0, 6, 4, 3],
            [6, 0, 2, 4, 3, 1, 5],
            [2, 1, 4, 0, 3, 6, 5],
            [5, 6, 4, 2, 0, 1, 3]]
WomenPref= [[0, 4, 3, 1, 5, 2, 6],
            [5, 1, 4, 6, 3, 2, 0],
            [1, 5, 4, 2, 6, 3, 0],
            [0, 1, 4, 5, 3, 2, 6],
            [5, 2, 4, 1, 0, 6, 3],
            [5, 6, 3, 2, 0, 1, 4],
            [2, 6, 5, 3, 4, 0, 1]]

n_stable = FindAllStableMatchings(MenPref, WomenPref) # n_stable = 11
```

5. (**implementation**, 5+15=20 pts) Consider the problem of minimum cost 3-coloring of the line graphs, an instance of which is provided in Figure 1. Note that, in line graphs, nodes are ordered on a line and there can be edges only between two adjacent nodes. In a *valid* coloring, no adjacent nodes can have the same color.

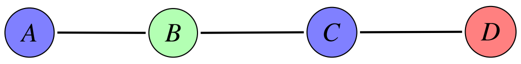

Figure 1: Sample coloring of a line graph instance with four nodes

(a) Implement an enumeration-based approach for this problem, which involves enumerating all the feasible combinations, calculating the cost associated with each combination, and outputting the minimum cost combination. Sample function call and expected inputs/outputs are provided below.

```
# 0: red, 1: blue, 2: green
costs = [[18, 3, 18], [14, 14, 4], [15, 3, 17]]
# costs[1][2]: cost of painting node 1 color green
# total number of combinations: number_colors^number_houses = 3^3 = 27
# enumerated list of feasible combinations (12): [[0, 1, 0], [0, 1, 2], [0, 2, 0], [0, 2, 1], [1, 0,
    1], [1, 0, 2], [1, 2, 0], [1, 2, 1], [2, 0, 1], [2, 0, 2], [2, 1, 0], [2, 1, 2]]
# [0,1,0]: feasible because all adjacent nodes have different color
# [0,1,1]: not feasible because last 2 nodes have same color
# Hint: itertools.product() library function generates all 27 combinations

min_cost_enum, best_coloring_enum = min_cost_vertex_coloring_enumeration(costs)
# min_cost_enum: 10, best_coloring_enum: [1, 2, 1]
```

(b) Implement a dynamic programming-based approach for this problem. You can construct the DP recursion using `Opt(k,i)`, which corresponds using color i for node k by taking into account all the nodes $\{0,\ldots,k\}$.

$$\texttt{Opt}(k,i) = \begin{cases} costs[k][i] & \textit{if } k=0 \\ costs[k][i] + \min_{j\neq i}\{\texttt{Opt}(k-1,j)\} & \textit{otherwise} \end{cases} \quad (1)$$

Sample function call and expected inputs/outputs are provided below.

```
min_cost_dp, best_coloring_dp, dp_table = min_cost_vertex_coloring_DP(costs)
# min_cost_dp: 10.0, best_coloring_dp: [1, 2, 1], dp_table: [[18.  3. 18.] [17. 32.  7.] [22. 10.
    34.]]
# dp_table[1][0]: total cost of coloring node 1 to red and node 0 to best of the green and blue (
    which is blue with cost 3)
# best_coloring_dp needs to be calculated based on DP approach, i.e., you cannot use solution from
    part (a) here
```