Ciências ULisboa | Informática

Fundamentos de Programação
1º Semestre 2025/26

Project Statement: Stable Matchings

original text by Pedro Ângelo

Rules:

- **Groups.** The project is developed by groups of 2 students. Both should work together towards the development of the project, and are expected to put the same amount of effort. Students are also encouraged to discuss the project with colleagues from other groups, and come up with solutions for the questions together, as long as the rules below are observed.

- **Code sharing and plagiarism; use of artificial intelligence.** It is **strictly forbidden** for students of different groups to share code or submit code obtained through plagiarism. Projects whose code shares similarities with projects from other groups will be punished in their evaluation.

  Students are **heavily** discouraged from generating solutions from artificial intelligence tools, such as ChatGPT, Copilot, DeepSeek, etc. Part of the evaluation lies in whether the student understood the assignment and also the code they produced.

  Students are **expected to have a global and detailed knowledge** about the code their group submitted, and may be asked to explain the reasoning and justify the choices that led to specific parts of the code. Students that **fail to properly explain their project will be punished** in their evaluation, which may include have their project submission annulled.

- **Submission.** Several files are made available: a `data.py` containing the data sets presented bellow; several tests files, called `test_<function-name>.py`, each for each function the students are required to implement, an empty `stable-matching.py` file, where students are to complete the function definitions, and a `main.py` file, that makes use of the student's definitions to run a simple program. Students must download these files, and place them all in the **same folder**.

  To submit the assignment, use the Moodle submission form on the *Project* section, called *Project - Phase <phase>*, where <phase> is the phase of the project you wish to submit. Students must **submit only the `stable_matching.py` file, properly identified with the**

group and students who wrote the file. There's no need for both students of the group to submit, **one is enough.** The dates for each phase of the project are as follows:

  - Phase A: published in **17/10/2025 at 23:59**, submission until **24/10/2025 at 23:59**;
  - Phase B: to be announced...
  - Phase C: to be announced...

- **Testing your implementation.** Students should test the functions they have written before submitting them. To aid students in testing their functions, several python files with unit tests are also made available. Each file tests a specific function, according to its name. To execute such files, ensure all files are in the same folder:

  - For Anaconda and Spyder users, write `!python -m unittest -v tests/test_<function-name>.py` in Spyder's right window. You can also run all tests with `!python -m unittest`.
  - For Linux and macOs users, use instead `python3.XX -m unittest -v tests/test_<function_name>.py`, where `3.XX` is replaced with your version number. You can also run all tests with `python3.XX -m unittest`.
  - Make use of the `main.py` file, and compare the trace of execution.

- **Evaluation.** Students will be evaluated on the following criteria: the correct application of the concepts taught in class and how well the code is written, meaningful names of variables, indentation, absence of errors and correctness of results. The results of tests will also account for the evaluation.

  Both students will be evaluated based on the solutions for functions of the **third set (*A3, B3, and C3*)**, depending on the phase. Furthermore, **one student** will be evaluated based on the solutions for functions of the **first set (*A1, B1, and C1*)**, and the **other student** for functions of the **second set (*A2, B2, and C2*)**.

  Students must choose which set from *A1, B1, and C1*, or *A2, B2, and C2* they prefer to be evaluated on. Note that both students can still help each other, and work on both sets.

---

1 Stable matching problem

The *stable matching problem* [1] is the problem of finding a stable matching between two equal-sized sets, S and T, of *participants* $s_1, \ldots, s_n \in S$ and $t_1, \ldots, t_n \in T$ according to an ordering of preferences for each participant. Note that each set, both S and T have the same number of (unique) participants.

Participants, stored in variables with names of the form `<set_name>`, is the list of elements belonging to set S (or T). Below are two *participants*' lists, first those of S and those of T.

```
s = ["s1", "s2", "s3", "s4", "s5"]
t = ["t1", "t2", "t3", "t4", "t5"]
```

Preferences, stored in variables with names of the form `<set-name>_prefs`, is a **list of pairs**, composed of **strings** and **lists of strings**, hence of the form `(<string>, [<string>])`. *Preferences* associate a *participant*, from set S, to its ordered list, by preference, of *participants* from set T.

Note that the number of *participants*, of set S, in a given *preferences* is the same as the number of preferred *participants*, of set T, associated to each *participant*. Note also that all *participants* from S appear as preferences of *participants* of T and vice-versa.

The previous definition also applies vice-versa i.e., to *participants* from T and their preferred participants from S. Below are two *preferences*, for elements of S and of T.

```
s_prefs = [
    ("s1", ["t1", "t2", "t3", "t4", "t5"]),
    ("s2", ["t5", "t3", "t2", "t4", "t1"]),
    ("s3", ["t2", "t5", "t4", "t1", "t3"]),
    ("s4", ["t5", "t2", "t1", "t3", "t4"]),
    ("s5", ["t1", "t4", "t2", "t3", "t5"])]

t_prefs = [
    ("t1", ["s4", "s1", "s3", "s2", "s5"]),
    ("t2", ["s2", "s1", "s4", "s3", "s5"]),
    ("t3", ["s1", "s3", "s5", "s4", "s2"]),
    ("t4", ["s4", "s2", "s5", "s3", "s1"]),
    ("t5", ["s2", "s1", "s5", "s3", "s4"])]
```

These preferences mean that, for example, participant `"s1"` from S prefers `"t1"`, then `"t2"`, `"t3"`, `"t4"` and finally `"t5"` is the least preferred. However, participant `"s2"` has a different order of preferences, as well as the other participants from S.

A *match* is a pair (s, t), with $s \in S$ and $t \in T$, associating an element s from set S to an element t of set T. A *matching* is a set of matches, such that each element of S is matched against a single element of T and vice-versa. A matching is *not stable* if:

- there is an element $s \in S$ which prefers some given element $t \in T$ over the element of T to which s is currently matched, **and**
- t also prefers s over the element t is matched with.

Intuitively, this would mean that, if both s and t prefer each other, then they should be matched. The matching below is considered stable:

```
[('s1', 't1'), ('s2', 't5'), ('s3', 't3'), ('s4', 't2')
    , ('s5', 't4')]
```

This problem appears in many real-world situations, such as matching workers to companies, students to universities, etc.

1.1 Workers and companies

While the explanation above introduces the problem from a purely definitional view-point, it is useful to reason about this problem using a real-world example.

Consider two groups (sets) of participants: **workers** are participants seeking to be hired by companies, and **companies** are participants which must each hire exactly one worker.

```
workers = ["Alice", "Bob", "Carol", "David", "Eve"]
companies = ["Facebook", "Apple", "Amazon", "Netflix",
    "Google"]
```

Each worker, and company, specify their list of preferences, ranking each participant from the opposite group according to their personal preference.

```
workers_prefs = [
    ("Alice", ["Google", "Apple", "Amazon", "Netflix",
        "Facebook"]),
    ("Bob", ["Google", "Amazon", "Apple", "Netflix", "
        Facebook"]),
    ("Carol", ["Apple", "Google", "Amazon", "Facebook",
         "Netflix"]),
    ("David", ["Google", "Amazon", "Netflix", "Apple",
        "Facebook"]),
    ("Eve", ["Apple", "Google", "Amazon", "Netflix", "
        Facebook"])
]
```

```
companies_prefs = [
    ("Facebook", ["Alice", "Bob", "Carol", "David", "
       Eve"]),
    ("Apple", ["Bob", "Alice", "David", "Eve", "Carol"
       ]),
    ("Amazon", ["Alice", "Bob", "David", "Carol", "Eve"
       ]),
    ("Netflix", ["Bob", "Alice", "Eve", "Carol", "David
       "]),
    ("Google", ["Alice", "Bob", "Carol", "David", "Eve"
       ])
]
```

In this example, `"Alice"` wishes to be hired by all the companies, from `"Google"` to `"Facebook"`, but if she has a offer from `"Google"` and another from `"Apple"`, she would much prefer `"Google"`.

A matching is a set of pairs `(worker, company)` such that each worker and each company appears in exactly one pair. A stable matching, in this context, is one where there is no worker–company pair who would both prefer to be matched with each other over their current partners. For example, for this set of preferences, the following matchings are stable:

```
[('Alice', 'Google'), ('Bob', 'Amazon'), ('Carol', '
   Facebook'), ('David', 'Netflix'), ('Eve', 'Apple')]
```

Notice that `"Alice"`'s preferred company is `"Google"`, and `"Google"`'s preferred worker is `"Alice"`. Hence, it makes sense this par is added to the list of matchings. `"Bob"`'s first preference is `"Google"`, but `"Google"` prefers `"Alice"` over `"Bob"`. Hence, `"Bob"` goes with his second preference, `"Amazon"`. `"Amazon"` also prefers, `"Bob"`, after `"Alice"`.

Hence, `"Carol"` likes `"Apple"`, however `"Eve"` also likes `"Apple"`, and `"Apple"` prefers `"Eve"` over `"Carol"`. So `"Eve"` gets matched with `"Apple"`. `"Carol"` cannot choose neither `"Google"` nor `"Apple"`, since these are already matched. So, it must choose `"Facebook"`. `"David"` also cannot choose neither `"Google"` nor `"Apple"`, so it ends up with `"Netflix"`.

1.2 Encoding as dictionaries [Phase B]

So far, in Phase A, we've been working with preferences as lists of pairs. However, now in Phase B, we'll switch to dictionaries, since these are much easier to work with and also more efficient. Preferences as a dictionary, stored in variables with names of the form `<set-name>_dict` is an **association** between participants, as **strings**, and their list of preferences, as a **list of strings**. For example, the previously introduced preferences are now stored as the following dictionaries:

```
s_dict = {
    's1': ['t1', 't2', 't3', 't4', 't5'],
    's2': ['t5', 't3', 't2', 't4', 't1'],
    's3': ['t2', 't5', 't4', 't1', 't3'],
    's4': ['t5', 't2', 't1', 't3', 't4'],
    's5': ['t1', 't4', 't2', 't3', 't5']}
t_dict = {
    't1': ['s4', 's1', 's3', 's2', 's5'],
    't2': ['s2', 's1', 's4', 's3', 's5'],
    't3': ['s1', 's3', 's5', 's4', 's2'],
    't4': ['s4', 's2', 's5', 's3', 's1'],
    't5': ['s2', 's1', 's5', 's3', 's4']}
workers_dict = {
    'Alice': ['Google', 'Apple', 'Amazon', 'Netflix', '
       Facebook'],
    'Bob': ['Google', 'Amazon', 'Apple', 'Netflix', '
       Facebook'],
    'Carol': ['Apple', 'Google', 'Amazon', 'Facebook',
       'Netflix'],
    'David': ['Google', 'Amazon', 'Netflix', 'Apple', '
       Facebook'],
    'Eve': ['Apple', 'Google', 'Amazon', 'Netflix', '
       Facebook']}
companies_dict = {
    'Facebook': ['Alice', 'Bob', 'Carol', 'David', 'Eve
       '],
    'Apple': ['Bob', 'Alice', 'David', 'Eve', 'Carol'],
    'Amazon': ['Alice', 'Bob', 'David', 'Carol', 'Eve'
       ],
    'Netflix': ['Bob', 'Alice', 'Eve', 'Carol', 'David'
       ],
    'Google': ['Alice', 'Bob', 'Carol', 'David', 'Eve'
       ]}
```

With dictionaries, it becomes much easier to obtain the list of preferences for a given participant. For example, to obtain the list of preferences for `'Alice'`, we write:

```
>>> workers_dict['Alice']
['Google', 'Apple', 'Amazon', 'Netflix', 'Facebook']
```

1.3 Matchings and likeability [Phase B]

In Phase B, we're interested in finding the best matchings from the set of all matchings. In order to do so, we'll need some way to calculate what "best" means, and assign a value to it, so we can compare between different matchings.

We'll proceed by calculating the *likeability* of each match i.e., the measure of how much each participant prefers each other. To do so, we'll calculate the preference each proposer has for an acceptor, and assign an integer value to it, and vice-versa. This calculation is quite simple, we just take the index of the acceptor in the list of preferences of the proposer. Hence, the likeability of a proposer with an acceptor at the top of his preference list is `0`, which is the highest likeability score. Then, for each subsequent position the acceptor is in in the preference list, the likeability score goes up one value, until it reaches the number of participants, which denotes the least likeability score. For example, according to `workers_dict`, `'Google'` is at the top of `'Alice'` preference list. Hence, the likeability score of `'Alice'` towards `'Google'` is `0`. Likewise, the likeability score of `'Google'` towards `'Alice'` is also `0`. On the other hand, `'Eve'` and `'Facebook'` are at the bottom of each other's preference lists, hence, the likeability score for each is `4`.

To then calculate the likeability score of a match, we just add the two likeability scores together. For example, for the match `('Alice','Google')`, we assign a value of `0`, and for `('Eve','Facebook')` we assign `8`. Since a matching is composed of several matches, we can just calculate the likeability score for all matches and add these all together. For example, the likeability score the following matching is `16`:

```
[('Alice', 'Google'), ('Bob', 'Amazon'), ('Carol', '
   Facebook'), ('David', 'Netflix'), ('Eve', 'Apple')]
```

2 Phase A

In Phase A, we take an exploratory approach to the *stable matching problem*. We will explore the problem itself, what it consists of, which data types (from those we've learned so far) we can use to encode the problem, how to obtain a problem set from the standard input, and how to verify its validity. Finally, the students are expected to write a simple data extraction function from the problem set. Phase A will require students to exercise topics from the Lecture 1 until Lecture 5.

Students are given the file `stable-matching.py` with functions `no_repetitions` and `same_elements` already implemented, and are tasked with implementing the following sets of functions:

A1: `participants_from_prefs`, `verify_prefs`, `compatible_prefs`

A2: `preferences_by_position`, `input_prefs`, `show_prefs`

A3: `most_chosen`

A description of the required functions follows. The order presented bellow represents the order by which students should implement the functions.

2.1 `participants_from_prefs`

`participants_from_prefs(prefs)` takes as input preferences `prefs` and returns the list of participants in order of appearance from `prefs`. For example:

```
>>> participants_from_prefs(workers_prefs)
['Alice', 'Bob', 'Carol', 'David', 'Eve']
>>> participants_from_prefs(companies_prefs)
['Facebook', 'Apple', 'Amazon', 'Netflix', 'Google']
```

Hint: use a `for` loop and list indexing.

2.2 `preferences_by_position`

`preferences_by_position(prefs, index)` takes as input preferences `prefs` and returns the list of preferences in position `index` for all participants. For example:

```
>>> preferences_by_position(workers_prefs, 0)
['Google', 'Google', 'Apple', 'Google', 'Apple']
>>> preferences_by_position(workers_prefs, 4)
['Facebook', 'Facebook', 'Netflix', 'Facebook', '
   Facebook']
>>> preferences_by_position(companies_prefs, 1)
['Bob', 'Alice', 'Bob', 'Alice', 'Bob']
```

Hint: use a `for` loop and list indexing.

2.3 `input_prefs`

`input_prefs(num_participants)` takes as input the number of participants `num_participants`, stores preferences obtained from the standard input, and returns those preferences. The function should be structured as follows:

```
def input_prefs(num_participants):
    """
    Takes as input the number of participants `
       num_participants`, stores preferences obtained
       from the standard input, and returns those
       preferences.
    """
    # define variable to hold prefs
    ...
    # for each number from 1 to num_participants
    ...
        # obtain the name of the participant
        ...
        # obtain the names of the participant's
           preferences, one in each line
    return prefs
```

For example:

```
>>> worker_prefs = input_prefs(5)
Insert name of participant 1: Alice
Insert preferences in order (one by line):
Google
Apple
Amazon
Netflix
Facebook
Insert name of participant 2: Bob
Insert preferences in order (one by line):
Google
Amazon
Apple
Netflix
Facebook
Insert name of participant 3: Carol
Insert preferences in order (one by line):
Apple
Google
Amazon
Facebook
Netflix
Insert name of participant 4: David
Insert preferences in order (one by line):
Google
Amazon
```

```
Netflix
Apple
Facebook
Insert name of participant 5: Eve
Insert preferences in order (one by line):
Apple
Google
Amazon
Netflix
Facebook
```

Hint: use a `for` loop, `print` and `input` and the function `append`.

2.4 `show_prefs`

`show_prefs(prefs)` takes as input preferences `prefs`, and prints the participants along with their list of preferences. For example:

```
>>> show_prefs(workers_prefs)
Alice: Google > Apple > Amazon > Netflix > Facebook
Bob: Google > Amazon > Apple > Netflix > Facebook
Carol: Apple > Google > Amazon > Facebook > Netflix
David: Google > Amazon > Netflix > Apple > Facebook
Eve: Apple > Google > Amazon > Netflix > Facebook
```

Hint: use a `for` loop, an iterator variable and `if` to decide if you print the separator `>` or not.

2.5 `verify_prefs`

`verify_prefs(prefs, num_participants)` takes as input preferences `prefs` and the number of participants `num_participants` and verifies if `prefs` are well-formed, by verifying the following conditions:

1. the length of `prefs` is equal to `num_participants`;
2. there are no repetitions of strings (representing participants) in the first element of the tuples of `prefs`;
3. the length of the second element of the tuples of `prefs` is equal to `num_participants`;
4. there are no repetitions of strings (representing participants) in the second element of the tuples of `prefs`;
5. all lists in the second element of the tuples of `prefs` share the same `num_participants` strings.

The function should be structured as follows:

```
def verify_prefs(prefs, num_participants):
    """
    Takes as input preferences 'prefs', and number of
       participants 'num_participants' and verifies if
       'prefs' are well-formed, by verifying the
       following conditions:
    - the length of 'prefs' is equal to '
       num_participants';
    - there are no repetitions of strings (representing
        participants) in the first element of the
       tuples of 'prefs';
    - the length of the second element of the tuples of
        'prefs' is equal to 'num_participants';
    - all lists in the second element of the tuples of
       'prefs' share the same 'num_participants'
       strings;
    """
    # check number of participants in prefs
    ...
    # check if there are no repetitions of participants
        in prefs
    ...
    # check number of preferences in prefs
    ...
    # check if preferences in prefs all share the same
       'num_participants' strings
    ...
```

For example:

```
>>> verify_prefs(workers_prefs, 5)
True
```

2.6 `compatible_prefs`

`compatible_prefs(prefs1, prefs_2)` takes as input preferences `prefs1` and `prefs2` and check if both are compatible, by verifying the following conditions:

1. strings (representing participants) in the first element of the tuples of `prefs1` share the strings (representing preferences) in the second element of the tuples of `prefs2`, and vice-versa.

The function should be structured as follows:

```
def compatible_prefs(prefs1, prefs2):
    """
    Takes as input preferences 'prefs1' and 'prefs2'
       and check if both are compatible, by verifying
       the following conditions:
```

```
    - strings (representing participants) in the first
      element of the tuples of 'prefs1' share the
      strings (representing preferences) in the second
       element of the tuples of 'prefs2', and vice-
      versa.
    """
    # obtain participants from prefs
    ...
    # check if participants share the same elements as
      preferences from the other prefs
    ...
```

For example:

```
>>> compatible_prefs(workers_prefs, companies_prefs)
True
>>> compatible_prefs(companies_prefs, workers_prefs)
True
>>> compatible_prefs(workers_prefs, t_prefs)
False
>>> compatible_prefs(s_prefs, t_prefs)
True
```

2.7 most_chosen

`most_chosen(prefs, participants_list)` takes as input preferences `prefs` and a list of participants `participants_list` consisting of the participants used as preferences in `prefs` and returns the participant who was chosen the most times. The function should be structured as follows:

```
def most_chosen(prefs, participants_list):
    """
    Takes as input preferences 'prefs' and a list of
      participants 'participants_list' consisting of
      the participants used as preferences in 'prefs'
      and returns the participant (or participants)
      who was chosen the most times as the first
      preference.
    """
    # initialise list of occurrences
    ...
    # get participants in a given position
    ...
    # count participant occurrences
    ...
    # obtain the maximum number of occurrences
    ...
    # return those participants with a number of
      occurrences equal to the maximum
```

```
    ...
    return chosen
```

For example:

```
>>> most_chosen(workers_prefs, companies)
['Google']
>>> most_chosen(companies_prefs, workers)
['Alice']
```

Hint: use functions `preferences_by_position` to obtain the list of participants, then count occurrences using a list.

2.8 Testing your implementation

To ensure their functions are well-written, students can also compare the following outputs against their programs:

- `main()` for `smallp_prefs` and `smalla_prefs`:

```
Insert number of participants in each group: 2
Add preferences for proposers.
Insert name of participant 1: P1
Insert preferences in order (one by line):
A1
A2
Insert name of participant 2: P2
Insert preferences in order (one by line):
A2
A1
Add preferences for acceptors.
Insert name of participant 1: A1
Insert preferences in order (one by line):
P1
P2
Insert name of participant 2: A2
Insert preferences in order (one by line):
P2
P1
Preferences are correct and compatible
Proposers:  ['P1', 'P2']
P1: A1 > A2
P2: A2 > A1

Acceptors:  ['A1', 'A2']
A1: P1 > P2
A2: P2 > P1

Most chosen proposers:  ['P1', 'P2']
Most chosen acceptors:  ['A1', 'A2']
```

- `main()` for `workers_prefs` and `companies_prefs`:

```
Insert number of participants in each group: 5
Add preferences for proposers.
Insert name of participant 1: Alice
Insert preferences in order (one by line):
Google
Apple
Amazon
Netflix
Facebook
Insert name of participant 2: Bob
Insert preferences in order (one by line):
Google
Amazon
Apple
Netflix
Facebook
Insert name of participant 3: Carol
Insert preferences in order (one by line):
Apple
Google
Amazon
Facebook
Netflix
Insert name of participant 4: David
Insert preferences in order (one by line):
Google
Amazon
Netflix
Apple
Facebook
Insert name of participant 5: Eve
Insert preferences in order (one by line):
Apple
Google
Amazon
Netflix
Facebook
Add preferences for acceptors.
Insert name of participant 1: Facebook
Insert preferences in order (one by line):
Alice
Bob
Carol
David
Eve
Insert name of participant 2: Apple
Insert preferences in order (one by line):
Bob
```

```
Alice
David
Eve
Carol
Insert name of participant 3: Amazon
Insert preferences in order (one by line):
Alice
Bob
David
Carol
Eve
Insert name of participant 4: Netflix
Insert preferences in order (one by line):
Bob
Alice
Eve
Carol
David
Insert name of participant 5: Google
Insert preferences in order (one by line):
Alice
Bob
Carol
David
Eve
Preferences are correct and compatible
Proposers:  ['Alice', 'Bob', 'Carol', 'David', 'Eve
    ']
Alice: Google > Apple > Amazon > Netflix > Facebook
Bob: Google > Amazon > Apple > Netflix > Facebook
Carol: Apple > Google > Amazon > Facebook > Netflix
David: Google > Amazon > Netflix > Apple > Facebook
Eve: Apple > Google > Amazon > Netflix > Facebook

Acceptors:  ['Facebook', 'Apple', 'Amazon', '
    Netflix', 'Google']
Facebook: Alice > Bob > Carol > David > Eve
Apple: Bob > Alice > David > Eve > Carol
Amazon: Alice > Bob > David > Carol > Eve
Netflix: Bob > Alice > Eve > Carol > David
Google: Alice > Bob > Carol > David > Eve

Most chosen proposers:  ['Alice']
Most chosen acceptors:  ['Google']
```

3 Phase B

In Phase B, we make a (naive) first attempt to solve the *stable matching problem*. To do so, we will first generate all matchings, eliminate all those that are not stable, and choose the one with highest likeability. Please read sections 1.2 and 1.3 before proceeding. Phase B will require students to exercise topics from Lecture 6 until Lecture 8, besides those until Lecture 5.

The folder hierarchy of the project has changed. To set up your working environment (Python files you need to run the program), you should download the Phase B files, and then copy all the definitions you wrote on `stable_matching.py` for the Phase A submission, onto the `stable_matching.py` file from Phase B. You should then implement the following sets of functions:

B1: no functions

B2: no functions

B3: `prefs_to_dict`, `permutations`, `generate_all_matchings`, `is_matching_stable`, `calculate_likeability_match`, `calculate_likeability_matching`, `calculate_best_matching`, `most_desirable`

A description of the required functions follows. The order presented below represents the order by which students should implement the functions. However, nothing prevents students from switching the order. In case you need to call some function yet to be implemented in the function you're writing, you can just write a simple implementation e.g., `return 0`, to keep going.

3.1 `prefs_to_dict`

`prefs_to_dict(prefs)` takes as input preferences `prefs` and returns a dictionary of `prefs`. For example:

```
>>> smallp_dict = prefs_to_dict(smallp_prefs)
>>> smallp_dict
{'P1': ['A1', 'A2'], 'P2': ['A2', 'A1']}
>>> smalla_dict = prefs_to_dict(smalla_prefs)
>>> smalla_dict
{'A1': ['P1', 'P2'], 'A2': ['P2', 'P1']}
>>> workers_dict = prefs_to_dict(workers_prefs)
>>> workers_dict
{'Alice': ['Google', 'Apple', 'Amazon', 'Netflix', '
    Facebook'],
 'Bob': ['Google', 'Amazon', 'Apple', 'Netflix', '
     Facebook'],
```

```
 'Carol': ['Apple', 'Google', 'Amazon', 'Facebook', '
    Netflix'],
 'David': ['Google', 'Amazon', 'Netflix', 'Apple', '
    Facebook'],
 'Eve': ['Apple', 'Google', 'Amazon', 'Netflix', '
    Facebook']}
>>> companies_dict = prefs_to_dict(companies_prefs)
>>> companies_dict
{'Facebook': ['Alice', 'Bob', 'Carol', 'David', 'Eve'],
 'Apple': ['Bob', 'Alice', 'David', 'Eve', 'Carol'],
 'Amazon': ['Alice', 'Bob', 'David', 'Carol', 'Eve'],
 'Netflix': ['Bob', 'Alice', 'Eve', 'Carol', 'David'],
 'Google': ['Alice', 'Bob', 'Carol', 'David', 'Eve']}
```

Hint: check out [3].

3.2 `permutations`

`permutations(list, perms=[])` takes as input a list `list`, and an optional list of permutations `perms`, and calculates the permutations of `list`, adding `perms` to the top. The function should be structured as follows:

```
def permutations(list, perms=[]):
    """
    Takes as input a list 'list', and an optional list
       of permutations 'perms', and calculates the
       permutations of 'list', adding 'perms' to the
       top.
    """
    # if list is empty, return list of perms
    ...
    # initialise empty list of all permutations
    ...
    # for each element in the list
    ...
        # obtain the remaining elements of the list
        ...
        # recursively call permutations, with the
           remaining elements, and the new permutations
            formed by adding the chosen element of the
           list
        # add all new permutation in a list of
           permutations
        ...

    return all_perms
```

For example:

```
>>> permutations([])
```

```
[[]]
>>> permutations([1])
[[1]]
>>> permutations([1,2])
[[1, 2], [2, 1]]
>>> permutations([1,2,3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],
    [3, 2, 1]]
>>> smallp = ['P1', 'P2']
>>> permutations(smallp)
[['P1', 'P2'], ['P2', 'P1']]
>>> permutations(workers)
[['Alice', 'Bob', 'Carol', 'David', 'Eve'],
 ['Alice', 'Bob', 'Carol', 'Eve', 'David'],
 ['Alice', 'Bob', 'David', 'Carol', 'Eve'],
 ['Alice', 'Bob', 'David', 'Eve', 'Carol'],
 ['Alice', 'Bob', 'Eve', 'Carol', 'David'],
 ['Alice', 'Bob', 'Eve', 'David', 'Carol'],
 ...,
 ['Eve', 'David', 'Alice', 'Bob', 'Carol'],
 ['Eve', 'David', 'Alice', 'Carol', 'Bob'],
 ['Eve', 'David', 'Bob', 'Alice', 'Carol'],
 ['Eve', 'David', 'Bob', 'Carol', 'Alice'],
 ['Eve', 'David', 'Carol', 'Alice', 'Bob'],
 ['Eve', 'David', 'Carol', 'Bob', 'Alice']]
>>> len(permutations(workers))
120
```

Hint: check out the definition of permutations in [2], use recursion.

3.3 `generate_all_matchings`

`generate_all_matchings(proposers, acceptors)` takes as input a list of proposers `proposers` and a list of acceptors `acceptors` and returns a list of all possible one-to-one matchings between proposers and acceptors. For example:

```
>>> smallp = ['P1', 'P2']
>>> smalla = ['A1', 'A2']
>>> small_matchings = generate_all_matchings(smallp,
   smalla)
>>> small_matchings
[[('P1', 'A1'), ('P2', 'A2')], [('P1', 'A2'), ('P2', '
   A1')]]
>>> workers_matchings = generate_all_matchings(workers,
    companies)
>>> workers_matchings
[[('Alice', 'Facebook'), ('Bob', 'Apple'), ('Carol', '
   Amazon'), ('David', 'Netflix'), ('Eve', 'Google')],
```

```
 [('Alice', 'Facebook'), ('Bob', 'Apple'), ('Carol', '
    Amazon'), ('David', 'Google'), ('Eve', 'Netflix')],
 ...,
 [('Alice', 'Google'), ('Bob', 'Netflix'), ('Carol', '
    Amazon'), ('David', 'Facebook'), ('Eve', 'Apple')],
 [('Alice', 'Google'), ('Bob', 'Netflix'), ('Carol', '
    Amazon'), ('David', 'Apple'), ('Eve', 'Facebook')]]
>>> len(workers_matchings)
120
```

Hint: use the `permutations` function.

3.4 `is_matching_stable`

`is_matching_stable(proposers_dict, acceptors_dict, matching)` takes as input proposer's preferences dictionary `proposers_dict`, acceptor's preferences dictionary `acceptors_dict` and matching `matching`, and returns `true` if `matching` is stable, or `false` otherwise. For example:

```
>>> is_matching_stable(smallp_dict, smalla_dict,
    small_matchings[0])
True
>>> is_matching_stable(smallp_dict, smalla_dict,
    small_matchings[1])
False
>>> is_matching_stable(workers_dict, companies_dict, [(
    'Alice', 'Facebook'), ('Bob', 'Apple'), ('Carol', '
    Amazon'), ('David', 'Netflix'), ('Eve', 'Google')])
False
>>> is_matching_stable(workers_dict, companies_dict, [(
    'Alice', 'Google'), ('Bob', 'Amazon'), ('Carol', '
    Facebook'), ('David', 'Netflix'), ('Eve', 'Apple')])
True
```

Hint: for each proposer, compare the preference position of its current match, against the preference position for all acceptors. If the proposer prefers the acceptor, check if the acceptor also prefers the proposer by comparing the preference position of the acceptors' current match against the proposer. Check the definition of *stable matching*.

3.5 `calculate_likeability_match`

`calculate_likeability_match(proposers_dict, acceptors_dict, match)` takes as input proposer's preferences dictionary `proposers_dict` and acceptor's preferences dictionary `acceptors_dict` and a match `match`, of the form (proposer, acceptor), and returns the likeability score of that match. For example:

```
>>> calculate_likeability_match(workers_dict,
    companies_dict, ('Alice','Google'))
0
>>> calculate_likeability_match(workers_dict,
    companies_dict, ('Bob','Amazon'))
2
>>> calculate_likeability_match(workers_dict,
    companies_dict, ('Carol','Facebook'))
5
>>> calculate_likeability_match(workers_dict,
    companies_dict, ('David','Netflix'))
6
>>> calculate_likeability_match(workers_dict,
    companies_dict, ('Eve','Apple'))
3
```

Hint: check section 1.3.

3.6 `calculate_likeability_matching`

`calculate_likeability_matching(proposers_dict, acceptors_dict, matchings)` takes as input proposer's preferences dictionary `proposers_dict`, acceptor's preferences dictionary `acceptors_dict` and matchings `matchings`, and returns the global likeability score of the matching. For example:

```
>>> small_matching = [('s1', 't1'), ('s2', 't5'), ('s3'
    , 't3'), ('s4', 't2'), ('s5', 't4')]
>>> calculate_likeability_matching(s_dict, t_dict,
    small_matching)
12
>>> calculate_likeability_matching(workers_dict,
    companies_dict, [('Alice', 'Google'), ('Bob', '
    Amazon'), ('Carol', 'Facebook'), ('David', 'Netflix'
    ), ('Eve', 'Apple')])
16
```

Hint: use `calculate_likeability_match`.

3.7 `calculate_best_matching`

`calculate_best_matching(proposers_dict, acceptors_dict)` takes as input proposer's preferences dictionary `proposers_dict` and acceptor's preferences dictionary `acceptors_dict`, and calculates and returns the stable matchings that have better (lower) global likeability score. For example:

```
>>> calculate_best_matching(smallp_dict, smalla_dict)
[[('P1', 'A1'), ('P2', 'A2')]]
>>> calculate_best_matching(workers_dict,
    companies_dict)
```

```
[[('Alice', 'Google'), ('Bob', 'Amazon'), ('Carol', '
   Facebook'), ('David', 'Netflix'), ('Eve', 'Apple')]]
>>> calculate_best_matching(companies_dict,
   workers_dict)
[[('Facebook', 'Carol'), ('Apple', 'Eve'), ('Amazon', '
   Bob'), ('Netflix', 'David'), ('Google', 'Alice')]]
```

Hint: use the functions developed so far.

3.8 `most_desirable`

`most_desirable(prefs, participants_list)` takes as input preferences `prefs` and a list of participants `participants_list` consisting of the participants used as preferences in `prefs` and returns the participant (or participants) who is considered the most desirable across all choice positions i.e., ties are resolved by searching on the next position of preferences. For example:

```
>>> most_desirable(workers_prefs,companies)
['Google']
>>> most_desirable(smallp_prefs,smalla)
['A1', 'A2']
```

Hint: use an adapted version of `most_chosen` that outputs results by position, no recursion is needed.

4 Phase C

To be announced...

References

[1] Stable matching problem. Wikipedia. https://en.wikipedia.org/wiki/Stable_matching_problem.

[2] Permutation. Wikipedia. https://en.wikipedia.org/wiki/Permutation.

[3] Python built-in functions. https://docs.python.org/3/library/functions.html.