General

1. I would have the main queue where we stack all of our email sending requests. Queue would be managed by some intermediate layer, meaning it would put the requests into the queue and also retrieve it for the multiple instances which would do the actual sending in parallel. Also this intermediate layer will manage the fact that you cannot send mail more often than 12 hours to the same address. For example, we are getting such requests: 1@gmail.com, 1@gmail.com, 2@gmail.com

Then the queue should look like this: 1@gmail.com, 2@gmail.com . Because we cannot yet send the mail to 1@gmail.com. And each mail request object will have as a property TimeElapsedSinceLastMail. The non-unique emails will be stacked into different secondary queue (Because if the time didn`t elapse for first element, then it definitely didn`t elapse for others). When it hits 12 hours (or 0 hours if Property will be TimeLimitElapsed) we can raise an event about that, so that listening intermediate layer can put this address into the main queue. Queues can be realised as tables in database, where each row is email request.

When the email was sent, we raise an event to update TimeLimitElapsed for corresponding email address.

“Email to the same address shouldn’t be send often than once per 12 hours” - by this i assume that we are not skipping the email requests, but saving them for later.

2. To me it sounds like a problem of clustering - we need to distribute customers on a multidimensional plane, where the proximity will be defined by all info about purchases that person made. it`s important to store the information about brand, its characteristics (f.e. Apple - luxury items, meanwhile Huawei or smth similar fills another cheaper, utilitarian niche), category of the product (f.e, food, toys, house cleaning supplies etc). It would help to build some sort of association table, where alternatives would be together. Also, it`s necessary to store details which pertain to specific purchase\customer: quantity of items in one purchase, frequency of buying, which items are bought together, is customer loyal to the product or they buy easily alternatives.

And now with all these features we can use any suitable clustering algorithm. I`m familiar with K-means, so I would choose it for now. It`s also important to make transformation on data before we can use it on algorithm. We need to normalize data. We can use PCA to remove not so important features. Also it is needed to dive into data\subject\area more in order to make the right conclusions about the data, like giving different weights to features.

Python

1 Complexity: O(n)

```
from typing import List


def get_biggest_str(strings: List[str]) -> str:
    if not strings:
        return None
    max_str = max(strings, key = len)
    return max_str
```

2

```
from typing import Dict

def merge_dicts(first_dict: Dict[str, int], second_dict: Dict[str, int]) -> Dict[str, int]:
    result: Dict[str, int] = dict()
    if first_dict is None or second_dict is None:
        return None

    s = first_dict.keys() & second_dict.keys()
    for res_key, res_val in first_dict.items():

        if res_key in second_dict:
            val2 = second_dict[res_key]
            res_val = max(res_val, val2)

        result[res_key] = res_val

    for res_key, res_val in second_dict.items():
        if res_key in result:
            continue

        result[res_key] = res_val

    return result
```

3.1. We can just use max function - O(n), instead of sorting and then fetching the biggest item - O(n*logn)

3.2.

```
def get_full_name(first_name, last_name):
    full_name = f"{first_name} {last_name}";
    return full_name

def join_names(first_names, last_names):
    result = map(get_full_name, first_names, last_names)
    return result
```