Algorithms for Applications	HW4: *NP Hard, Approximation algorithms*
CS 5112 Fall 2023	**Due: December 15, 11:59pm ET**

Homework Instructions: The homework contains mainly three types of tasks:

- **Coding.** You can find a template for all the coding problems inside the `problems.zip` in the module. Please use these templates, to ensure the files and functions have the same names that the autograder will check for.

  Inside the `problems` directory, you will find the `tests` directory, which contains some sample test cases for the different problems. These test cases are intended to help you test and debug your solution. However, note that Gradescope will run a more comprehensive test suite. Feel free to add additional test cases to further test your code.

  In order to run the tests for a particular problem n, navegate into `problems/tests` and run: `python3 test_problem_n.py`

- **Algorithm Description.** When a questions requires designing an algorithm, you should describe what your algorithm does in the writeup, in clear concise prose. You can cite algorithms covered in class to help your description. You should also argue for your algorithm's asymptotic run time or, in some cases when indicated in the problem, for the run time bounds of your implementation.

- **Empirical Performance Analysis.** Some questions may ask you to do an empirical performance analysis of one or more algorithms' runtime under different values of n. For these questions, you should generate at least ten test cases for the implementation, for various values of n (include both small and large instances). Then, measure the performance locally on your own system, by taking the median runtime of the implementation over ten or more iterations. Graph the resulting median run times. The x-axis should be instance size and the y-axis should be median run time. You can use any plotting library of your choice. If you have never plotted on Python before, this matplotlib turorial has some examples.

  If several subquestions in a problem ask you to provide performance analysis, please do them all in the same environment (same system and configuration), in order to have a more accurrate comparison between algorithms. You can include graphs for a same problem in the same plot, as long as the different graphs are properly labeled.

  For these types of questions, you will not need to submit the tests you generate or your code for benchmarking or plotting the algorithms. However, you should include the graphs you generate into the write-up. You should also write about what you observe from the analysis and potentially compare it to the complexity analysis. Finally, also include a description of the environment (CPU, operating system and version, amount of memory) you did the testing on.

Submission Instructions: Hand in your solutions electronically on Gradescope. There are two active assignments for this problem set on Gradescope: one with the autograder, where you submit your code, and another one where you submit the write-up.

Coding Submission: You only need to submit the code for the specific functions we ask you

to implement. While you will write additional code to generate the performance analysis graphs when required, you do not need to submit that code.

To submit to the autograder, please zip the `problems` directory we provided you with. Therefore, your zipped file should have the following structure:

```
problems
├── problem_1
│   ├── p1_a.py
│   ├── p1_b.py
│   └── ...
├── problem_2
│   └── p2.py
└── ...
```

In order to ensure your code runs correctly, do not change the names of the files or functions we have provide you. Additionally, do not import external python libraries in these files, you don't need any libraries to solve the coding assignments. Finally, if you write any helper functions for your solutions, please include them in the same files as the functions that call them.

We have some public tests in the autograder to verify that all the required files are included, so you will be able to verify on Gradescope that your work is properly formatted shortly after submitting. We strongly encourage checking well ahead of the due date that your solutions work on the autograder, and seeking out assistance from the TAs should it not. We cannot guarantee being responsive the night the assignment is due.

Write-up Submission: Your write-up should be a nicely formatted PDF prepared using the LaTeX template on Canvas, where you can type in your answer in the `main.tex` file. If you do not have previous experience using LaTeX, we recommend using Overleaf. It is an online LaTeX editor, where you can upload and edit the template we provide. For additional advice on typesetting your work, please refer to the *resources* directory on the course's website.

Academic Integrity: You may use online sources or tools (such as code generation tools), but any tools you use should be explicitly acknowledged and you must explain how you used them. You are responsible for the correctness of submitted solutions, and we expect you to understand them and be able to explain them when asked by teaching staff.

Collaboration Policy: Collaboration (in groups of up to three students) is encouraged while solving the problems, but:

1. list the netids of those in your group;
2. you may discuss ideas and approaches, but you should not write a detailed argument or code outline together;
3. notes of your discussions should be limited to drawings and a few keywords; you must **write up the solutions and write code on your own**.

Problem 1. Let's say we want to count the number of times elements appear in a stream of data, $x_1, \ldots, x_q$. A simple solution is to maintain a hash table that maps elements to their frequencies.

This approach does not scale: Imagine having an extremely large stream consisting of mostly unique elements. For example, consider network monitoring (either for large network flows or anomalies), large service analytics (e.g. Amazon view/buy counts, Google search popularity), database analytics, etc. Even if we are only interested in the most important ones, this method has huge space requirements. Since we do not know for which items to store counts, our hash table will grow to contain billions of elements.

The Count-Min Sketch, or CMS for short, is a data structure that solves this problem in an approximate way.

Approximate Counting with Hashing. Given that we only have limited space availability, it would help if we could get away with not storing elements themselves but just their counts. To this end, let's try to use only an array, with w memory cells, for storing counts as shown below in Figure 1.

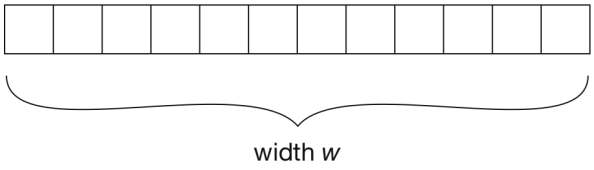

Figure 1: Counting with a single hash

With the help of a hash function h, we can implement a counting mechanism based on this array. To increment the count for element x, we hash it to get an index into the array. The cell at the respective index $h(x)$ is then incremented by 1.

Concretely, this data structure has the following operations:

- Initialization: $\forall i \in \{1, \ldots, w\}$: count[$i$]=0.
- Increment count (of element x): count[h(x)]+=1
- Retrieve count (of element x): count[h(x)]

This approach has the obvious drawback of hash conflicts, which result in over-counting. We would need a lot of space to make collisions unlikely enough to get accurate counts. However, we at least do not need to explicitly store keys anymore.

More hash functions

Instead of just using one hash function, we could use d different ones. These hash functions should be pairwise independent. To update a count, we then hash item a with all d hash functions, and subsequently increment all indices we got this way. In case two hash functions map to the same index, we only increment its cell once.

Unless we increase the available space, of course all this does for now is to just increase the

number of hash conflicts. We will deal with that in the next section. For now let's continue with this thought for a moment.

If we now want to retrieve a count, there are up to d different cells to look at. The natural solution is to take the minimum value of all of these. This is going to be the cell which had the fewest hash conflicts with other cells.

$$\min_{i=1}^{d} \text{count}[\text{h}_i(x)]$$

While we are not fully there yet, this is the fundamental idea of the Count-Min Sketch. Its name stems from the process of retrieving counts by taking the minimum value.

Fewer hash conflicts

We added more hash functions but it is not evident whether this helps in any way. If we use the same amount of space, we definitely increase hash conflicts. In fact, this implies an undesirable property of our solution: Adding more hash functions increases potential errors in the counts.

Instead of trying to reason about how these hash functions influence each other, we can design our data structure in a better way. To this end, we use a matrix of size $w \times d$. Rather than working on an array of length w, we add another dimension based on the number of hash functions as shown below in fig. 2.

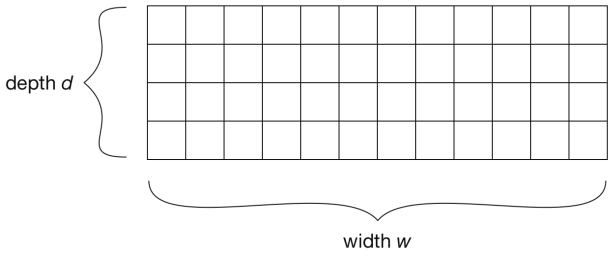

Figure 2: Counting with multiple hashes.

Next, we change our update logic so that each function operates on its own row. This way, hash functions cannot conflict with another anymore. To increment the count of element a, we now hash it with a different function once for each row. The count is then incremented in exactly one cell per row.

- Initialization: $\forall i \in \{1, \ldots, d\}, j \in \{1, \ldots, w\} : \text{count}[i, j] = 0$
- Increment count (of element x): $\forall i \in \{1, \ldots, d\} : \text{count}[i, \text{h}_i(x)]$ += 1
- Retrieve count (of element x): $\min_{i=1}^{d} \text{count}[i, \text{h}_i(x)]$

This is the full CMS data structure. We call the underlying matrix a sketch.

(a) **(10 points)** Your friend implemented the following hash functions for each row in the sketch: $\text{h}_i(x) = (x + a_i) \mod w$ for $0 < i < d$ where a_i is chosen randomly for each row, and $\mod$ is the mod operation (sometimes written as $\%$). Is the choice of hash functions good or bad? Please justify your answer in 1 – 2 sentences or provide a counter example.

(b) **(10 points)** Implement CMS in the function `count_min_sketch` in `problem_1/p1_b.py`. Given the vectors $\mathbf{a} = [a_1, \ldots, a_d]$, $\mathbf{b} = [b_1, \ldots, b_d]$, a scalar p implement the hash function $\mathrm{h}_i(x) = ((a_i x + b_i) \mod p) \mod w$. Then, use this hash function on a stream of data to create the sketch matrix.

Code input and output format. The function `count_min_sketch` takes in the following arguments: $\mathbf{a} = [a_1, \ldots, a_d]$, $\mathbf{b} = [b_1, \ldots, b_d]$ as vectors with positive entries, `w` and `p` as scalars, and a python generator function, `stream`, that produces a stream of data. The output of the function is the sketch matrix, of size $d \times w$.

Example

For $d = 2, w = 3$, given the vectors $\mathbf{a} = [1, 2]$, $\mathbf{b} = [3, 5]$, $p = 100$, $w = 3$, we get the following hash functions: $\mathrm{h}_1(x) = ((x+3) \,\%\, 100) \,\%\, 3$, $\mathrm{h}_2(x) = ((2x+5) \,\%\, 100) \,\%\, 3$. For the stream of data $[10, 11, 10]$ the function `count_min_sketch` should return the following: `[[0, 2, 1], [1, 2, 0]]` which corresponds to the following sketch matrix:

```
0, 2, 1
1, 2, 0
```

Problem 2. In a charming town renowned for unique collectibles, a shopkeeper Samuel faced a challenge. He had received n packages, each with distinct sizes, and needed to find boxes to match. Samuel had multiple suppliers denoted by m, each offering boxes of varying dimensions. His goal was to choose a supplier whose boxes would minimize the total wasted space, defined as the difference between the box and package sizes. Note that **one box can contain only one package**. Your task is to help Samuel choose a single supplier and use boxes from them such that the total wasted space is minimized. Let the size of the i^{th} box be b_i and the size of the j^{th} package be p_j. For each package in a box, we define the space wasted to be $b_i - p_j$. The total wasted space is the sum of the space wasted in all the boxes.

For example, if you have to fit packages with sizes `[2,3,5]` and the supplier offers boxes of sizes `[4, 8]`, you can fit the packages of size `2` and size `3` into two boxes of size `4` and the package with size `5` into a box of size `8`. This would result in a waste of `(4-2) + (4-3) + (8-5) = 6`.

(10 points) Implement the function `linear_search` in `problem_2/p2_a.py` that has a time-complexity of $\Theta(m \times n \times b)$ where b denotes the average number of box types offered by each supplier.

(30 points) Implement the function `binary_search` in `problem_2/p2_b.py` that has a time-complexity of $\Theta(m \times \log(n) \times b)$ where b denotes the average number of box types offered by each supplier.

Code input and output format. The package sizes are given as an integer array `packages`, **each with distinct sizes**, where `packages[i]` is the size of the i^{th} package. The suppliers are given as 2D integer array `boxes`, where `boxes[j]` is an array of box sizes that the j^{th} supplier produces. Return the minimum total wasted space by choosing the box supplier optimally, or -1 if it is impossible to fit all the packages inside boxes.

Constraints:

$$1 \leq n \leq 10^5$$
$$1 \leq m \leq 10^5$$
$$1 \leq \texttt{packages}[i] \leq 10^5$$
$$len(\texttt{boxes}[j]) \leq 10^5$$
$$\texttt{boxes}[j][k] \leq 10^5$$
The elements in $\texttt{boxes}[j]$ are **distinct**.

Example 1:

- Input: `packages = [2,3,5]`, `boxes = [[4,8],[2,8]]`
- Output: 6
- Explanation: It is optimal to choose the first supplier, using two size-4 boxes and one size-8 box. The total waste is (4-2) + (4-3) + (8-5) = 6.

Example 2:

- `packages = [2,3,5]`, `boxes = [[1,4],[2,3],[3,4]]`

- Output: -1
- Explanation: There is no box that the package of size 5 can fit in.

Example 3:

- Input: `packages = [3,5,8,10,11,12]`, `boxes = [[12],[11,9],[10,5,14]]`
- Output: 9
- It is optimal to choose the third supplier, using two size-5 boxes, two size-10 boxes, and two size-14 boxes. The total waste is (5-3) + (5-5) + (10-8) + (10-10) + (14-11) + (14-12) = 9.

Problem 3. Dr. Carter, a scientist, is aiming to sequence a set of genes. The set is represented by a sequence of natural numbers g_i, where g_i takes the values $1, \ldots, n$, $\forall\ i \in \{1 \ldots n\}$, and where n denotes the length of the sequence. Dr. Carter seeks the most efficient order to read the DNA fragments, minimizing redundancy and optimizing sequencing efforts. Given a sequence $G = [g_1, g_2, \ldots, g_n]$, there is a cost associated with processing every pair (g_i, g_j) for $1 \leq i, j \leq n$ of genes. The pairwise cost between the genes is represented as a symmetric matrix $C \in \mathbb{R}_{>0}^{n \times n}$ where the entries in the main diagonal are set to ∞. Let $C(i, j)$ denote the cost associated with the pair (g_i, g_j). Given a sequence of natural numbers, $s \in P(n)$, where $P(n)$ denotes the set of all permutations of the first n natural numbers, let the cost associated with s be denoted by $\texttt{cost}(s)$. Then $\texttt{cost}(s)$ can be computed by

$$\texttt{cost}(s) = \sum_{i=1}^{n-1} C(s_i, s_{i+1}).$$

Your job is to find an ordering r that minimizes the cost associated with it .i.e.,

$$r = \arg\min_{s \in P(n)} \texttt{cost}(s).$$

Notice that this problem is NP-hard and scales poorly with increasing n.

(a) Since finding the optimal solution isn't tractable, one approach is to use a greedy algorithm such as the following:

```python
def greedy_solution(cost_matrix):
  n = len(cost_matrix)
  min_edge_cost = float('inf')
  min_i = None
  min_j = None
  for i in range(n):
    for j in range(n):
      if min_edge_cost > cost_matrix[i][j]:
        min_i, min_j = i, j
        min_edge_cost = cost_matrix[i][j]
  sequence = [min_i, min_j]
  r = min_j
  for _ in range(n - 1):
    min_entry = float('inf')
    min_col = None
    for j in range(n):
      if min_entry > cost_matrix[r][j]:
        min_entry = cost_matrix[r][j]
        min_col = j
    sequence.append(min_col)
    r = min_col
  return sequence
```

This function can be found in `problem_3/p3_a.py` file. The solution found by this greedy approach is a sub-optimal solution. Design a cost matrix for $n = 4$ such that the solution predicted by the greedy algorithm corresponds to the worst solution i.e. the sequence with the highest cost.

(20 points) You are required to design the cost matrix C, which is a matrix of size 4×4. The matrix should be put in `problem_3/p3_a.txt`. The objective is to structure the cost matrix in a manner that ensures the solution returned by `greedy_solution` in `problem_3/p3_a.py` has the maximum cost compared to all other potential solutions.

Code input and output format. The cost matrix, described as a 4×4 matrix with positive entries, can be represented in a text file named `p3_a.txt` in the following format:

```
inf, 1, 2, 3
4, inf, 6, 7
5, 9, inf, 11
12, 13, 14, inf
```

In this representation:

- Each line in the file corresponds to a row in the cost matrix.
- The entries in each row are comma-separated, representing the values in the respective columns of the cost matrix.
- The symbol `inf` represents infinity.

(b) Now we'll use a local search method to improve upon the greedy approach. Given a candidate solution, the idea is to iteratively improve the solution by making "small" changes to it. The algorithm is outlined as follows:

1. Initialization: Start with an initial candidate sequence with a random ordering of entries.
2. Neighbor Generation: Choose two genes in the candidate. Consider swapping the genes, creating a new "potential" solution for the next iteration.
3. Evaluate the cost associated with the change: Calculate the change in cost resulting from the swap. If the "potential" candidate has a smaller cost than the current candidate, accept the change. Otherwise, maintain the current candidate.
4. Iterate: Repeat steps 2-3 for all possible pairs of genes in the candidate. This constitutes one iteration.
5. Termination: Run multiple iterations, i.e. step 4, until no more improvements can be made.

(20 points) Implement the function `local_search_2opt` in `problem_3/p3_b.py`.

Code input and output format. The cost matrix, `c`, is given as a $n \times n$ matrix with positive entries, the candidate solution, `init`, is given as a random permutation of the natural numbers $[1, \ldots, n]$. Return the updated solution by repeatedly applying 2-opt as outlined above.

Example:

- `c = [[inf, 3, 5], [3, inf, 7], [5, 7, inf]]`, `candidate = [2, 3, 1]`
- Output: `candidate = [2, 1, 3]`
- Explanation: The cost of the candidate solution, denoted as `candidate = [2, 3, 1]`, is calculated as `c`$(2, 3) +$ `c`$(3, 1) = 7 + 5 = 12$. The neighboring candidates, namely `[3, 2, 1]` and `[2, 1, 3]`, have costs of 10 and 8, respectively. Therefore, after the first iteration, the new candidate becomes `[2, 1, 3]`. Moving to the next iteration, the candidate `[2, 1, 3]` has neighbors `[1, 2, 3]` and `[2, 3, 1]`, with costs of 10 and 12, respectively. Since

the neighboring solutions have higher costs, the candidate solution remains unchanged, and the algorithm converges. Consequently, the final solution is `[2, 1, 3]`.

Algorithms for Applications	HW4: Challenge Problem
CS 5112 Fall 2023	**Due: December 15, 11:59pm ET**

Instructions: Challenge problems are, as the term indicates, *challenging*. They do not count for the homework score (90% of your course grade); instead, they are considered separately as extra credit over your course grade (additional 15% in total, 3.75% per assignment).

Questions about challenge problems will have lowest priority in office hours, and we do not provide assistance beyond a few hints to help you know whether you are on the right track.

Submission Instructions: You can choose not to hand a submission, but we encourage everyone to attempt the challenge problem. If you solve it, please hand in your answers (both, the coding and the write-up) through Gradescope, along with the rest of your assignment.

Academic Integrity and Collaboration Policy: The same guidelines apply to challenge problems as for the regular homework problems.

Once upon a time in the digital kingdom, there existed an array of binary strings known as `strs`. These strings, comprising only of 0s and 1s, held the secrets to unlocking great possibilities. Alongside these mysterious strings, two wise integers named `m` and `n` entered the scene, bearing a quest for knowledge.

The mission presented to the inhabitants of the digital realm was to uncover the size of the largest subset within the array of binary strings. However, this was no ordinary quest. The conditions were set by the integers `m` and `n` - they imposed limits on the number of 0s and 1s that the chosen subset could possess.

The quest's rules were clear: the subset's size must be maximized, but within the constraints of having at most `m` 0s and `n` 1s. A set `x` was considered a subset of another set `y` only if every element in `x` was also present in `y`.

Implement the function `find_max_form` in `challenge/challenge.py`.

Code input and output format. `strs` is given as an array of binary strings, `m` and `n` are integers. Return an integer that represents the size of the largest subset of `strs` with at most m 0's and n 1's.

Example 1:

- Input: `strs = ["10", "0001", "111001", "1", "0"]`, `m = 5, n = 3`
- Output: 4
- The largest subset with at most 5 0's and 3 1's is `{"10", "0001", "1", "0"}`, so the answer is 4. Other valid but smaller subsets include `{"0001", "1"}` and `{"10", "1", "0"}`. `{"111001"}` is an invalid subset because it contains 4 1's, greater than the maximum of 3.

Example 2:

- Input: `strs = ["10","0","1"]`, `m = 1, n = 1`
- Output: 2
- The largest subset is `{"0", "1"}`, so the answer is 2.