CS 170 Efficient Algorithms and Intractable Problems
Fall 2017 Prasad Raghavendra and Sanjam Garg

Homework 5

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.

- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.

- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, October 11, at 4:59pm

0. Who did you work with?

List all your collaborators on this homework. If you have no collaborators, please list “none”.

Solution: N/A

1. (★★★ level) Scheduling Homeworks

You have n homeworks to do. Each homework takes one hour to complete. Homework i has a deadline time T_i. You can think of T_i as a positive integer value, and the current time is 0, namely homework i is due in T_i hours. You get S_i points ($S_i \geq 0$) if you finish homework i before its deadline. You can only do one homework at any time. The goal is to maximize the total points you can get.

For each of the following greedy algorithms, either prove that it is correct, or give a simple counterexample (with at most three homeworks) to show that it fails.

(a) Among unscheduled homeworks that can be scheduled on time, consider the one whose deadline is the earliest (breaking ties by choosing the one with the highest points), and schedule it at the earliest available time. Repeat.

Solution: This is incorrect. Consider a listing of (T_i, S_i) as $(1,1),(2,10),(2,20)$. The algorithm schedules $(1,1)$ at time 0, $(2,20)$ at time 1, and $(2,10)$ late, with total points 21. One better (actually optimal) scheduling is to schedule $(2,10)$ at time 0, $(2,20)$ at time 1, and $(1,1)$ late, with total points 30.

(b) Among unscheduled homeworks that can be scheduled on time, consider the one whose points is the highest (breaking ties by choosing the one with the earliest deadline), and schedule it at the earliest available time. Repeat.

Solution: Incorrect. Consider a listing of (T_i, S_i) as $(1,1),(2,10)$. The algorithm schedules $(2,10)$ at time 0, and $(1,1)$ late, with total points 10. One better (actually optimal) scheduling is to schedule $(1,1)$ at time 0, $(2,10)$ at time 1, with total points 11.

(c) Among unscheduled homeworks that can be scheduled on time, consider the one whose points is the highest (breaking ties arbitrarily), and schedule it at the latest available time before its deadline. Repeat.

Solution: Correct.

Proof: Consider any partial scheduling P of homeworks, which is part of an optimal schedule Q. Let j_h be a homework with the highest points among those unscheduled homeworks in P that can still be scheduled on time. We will show that there is an optimal schedule Q' that schedules j_h at the latest available time before its deadline. Once this "greedy choice property" is established, then clearly the algorithm gives optimal schedule (by a standard induction argument).

Now we prove the property. We construct Q' from Q by scheduling j_h at the latest available time before its deadline. In case Q has that time slot taken by a homework j_k: if Q schedules j_h before its deadline, then Q' swaps j_h and j_k; otherwise Q' replaces j_k with j_h.

- If Q schedules j_h before its deadline: then Q' schedules j_k earlier (hence no loss for j_k) and j_h later (but still before deadline, hence no loss for j_h) than Q does, so Q' has no lower points than Q;
- If Q does not schedule j_h before its deadline: then Q' schedules j_h on time, but loses the points of j_k. Q' still has no lower points than Q since $S_h \geq S_k$.

2. (★★★ level) Graph Coloring

Let $G = (V, E)$ be an undirected graph where every vertex has degree less than 170. Let's find a way of to color each vertex blue or red, so that every vertex has less than 85 neighbors of its own color.

Consider the following algorithm, where we call a vertex *bad* if it has at least 85 neighbors of its own color:

```
1.  Color each vertex arbitrarily.
2.  Let $B := \{v \in V : v \text{ is bad}\}$.
3.  While $B \neq \emptyset$:
4.      Pick any bad vertex $v \in B$.
5.      Reverse the color of $v$.
6.      Update $B$ to reflect this change, so that it again holds the set of bad vertices.
```

Notice that if this algorithm terminates, it is guaranteed to find a coloring with the desired property.

(a) Prove that this algorithm terminates in a finite number of steps.

Hint: Define a function that associates a non-negative integer to each possible way of coloring the graph, in such a way that each iteration of the while-loop is guaranteed to strictly reduce the value of the function.

Solution: Define the function to be the number of edges that connect same-color nodes. This is clearly a non-negative integer.

It decreases at each step since when we flip a node's color, it assumes a color that a strict majority of its neighbors *didn't* have.

The initial value of the function is finite. Since it is a non-negative integer that monotonically diminishes as the algorithm runs, the algorithm must stop, either when the value of the function reaches zero, or possibly before.

(b) Prove that the algorithm terminates after at most $|E|$ iterations of the loop.

Hint: You should figure out the largest possible value of the function.

Solution: The function defined in part (a) is clearly upper bounded by $|E|$: the number of same-color edges can't be more than the total number of edges. The value of the function decreases by at least 1 in each iteration of the loop, and the value cannot go below 0, so the number of iterations of the loop must be at most the initial value of the function: i.e., at most $|E|$.

3. (★★ level) 170-Graph

An undirected graph is called a 170-graph if every vertex has degree at least 170. That is, every vertex has at least 170 neighbors. Given an undirected graph $G = (V, E)$, design an algorithm to find a subgraph $G' = (V', E')$ of G that is a 170-graph. We want to make $|V'|$ as large as possible. The running time of your algorithm should be polynomial in $|V|$ and $|E|$.

Hint: There are some vertices you can rule out immediately as not in G'.

(You only need to provide the main idea and running time.)

Solution 1:

Main idea: The basic idea here is that we iteratively remove non-G' vertices (that are certainly not in G') until we reach a fixed point. Any node whose degree is below 170 is a non-G' vertex. We keep a worklist of pending non-G' vertices. In each iteration we remove a non-G' vertex from the worklist, delete it, adjust the degree of its neighbors, and add them to the worklist if their degree has fallen below 170.

Pseudocode: (not required for this question)

```
1.  Set $d[v] := 0$ for each $v \in V$.
2.  For each edge $\{u, v\} \in E$:
3.      Increment $d[u]$. Increment $d[v]$.
4.  Initialize a list $W$ as $W := \{v \in V : d[v] < 170\}$.
5.  While $W$ is non-empty:
6.      Remove a vertex from $W$; call it $v$.
7.      For each $\{v, w\} \in E$:
8.          Decrement $d[w]$.
9.          If $d[w] < 170$ and $w \notin W$, add $w$ to $W$.
10.     Remove $v$ from the graph.
11. Output the set of nodes and edges left in the graph.
```

Running time: $O(|V| + |E|)$. We examine each vertex twice (in lines 1 and 4), and do a constant amount of work per vertex in each case. Each vertex is inserted into the worklist at most once, so the number of iterations of the loop in lines 5–10 is at most $|V|$, and each iteration takes $O(1)$ time, ignoring the inner loop at lines 7–9. (Notice that you can remove an element from a list in $O(1)$ time if you remove the item at the head of the list, so line 6 takes $O(1)$ time.) Also, the inner loop at lines 7–9 examines each edge at most once, when we remove one of its endpoints, and we we do a constant amount of work per edge in that case. This accounts for all of the work done in this algorithm, and we can see that we do $O(1)$ work per vertex plus $O(1)$ work per edge. Hence, the running time is $O(|V| + |E|)$.

Proof of correctness: (not required for this question)

Lemma: At each iteration of the while loop, for each vertex v that has not yet been deleted, $d[v] =$ the degree of v (in the graph that remains).

Proof: This invariant can be easily proven by induction on the number of iterations of the loop. Whenever we delete a vertex, we decrease each of its neighbors to reflect the change in their degree.

Claim: After the algorithm ends, we've identified a subgraph G' such that every vertex in G' has degree at least 170 in that subgraph. In other words, the algorithm outputs a 170-graph.

Proof: At the beginning of every iteration of the loop, every node either has degree at least 170, or is in the worklist W. Nodes in the worklist could not possibly be in a 170-graph. This is true before the first iteration because that's how line 4 initialized the worklist. After that, a node's degree can only change when one of its neighbors is removed, and whenever a node is deleted we check all of its neighbors' degrees. Therefore, after the loop, when the worklist is empty, every remaining vertex [if any] will have degree ≥ 170 in the remaining subgraph.

Claim: Every vertex that can be in a 170-graph is included in the output of the algorithm.

Proof: Let S be the largest set of vertices that can be in a 170-graph. Then it is an invariant that $d[s] \geq 170$ and $s \notin W$, throughout the algorithm, for all $s \in S$. This is true before the first iteration because of how line 4 initialized the worklist. Also, if it is true before one iteration of the loop, it remains true after that iteration. In particular, consider an iteration where we remove v from the worklist. By the inductive hypothesis and using the fact that we had $v \in W$ at the start of this iteration, it follows that $v \notin S$. Also, for each neighbor w of v, if $w \in S$, by the inductive hypothesis w had edges to ≥ 170 other members of S; since $v \notin S$, after deleting v it still has edges to ≥ 170 other members of S. Therefore, the invariant remains true after this iteration of the loop, and the claim follows by induction.

The first claim shows that the algorithm's output is not "too large", and the second claim shows that the algorithm's output is not "too small". This implies that our algorithm finds the largest possible 170-graph.

Solution 2:

Main idea: The basic idea is that we iteratively remove non-G' vertices (that are certainly not in G') until we reach a fixed point. Any node whose degree is below 170 is a non-G' vertex. In each iteration we check the remaining graph to see if there is a non-G' vertex. If there is one, then we remove it and its incident edges; otherwise we reach a 170-graph.

Running time: $O(|V|(|V|+|E|))$. In every iteration at least one vertex is removed, hence there are at most $|V|$ iterations. Every iteration takes at most $|V|+|E|$ time to check if there is a non-G' vertex, and to remove edges.

4. (★★★★ level) Weighted Set Cover

In class (and Chapter 5.4) we looked at a greedy algorithm to solve the *set cover* problem, and proved that if the optimal set cover has size k, then our greedy algorithm will find a set cover of size at most $k \log n$.

Here is a generalization of the set cover problem.

- *Input:* A set of elements B of size n; sets $S_1, \ldots, S_m \subseteq B$; positive weights $w_1, \ldots, w_m$.
- *Output:* A selection of the sets S_i whose union is B.
- *Cost:* The sum of the weights w_i for the sets that were picked.

Here is an algorithm (pseudocode) to find the set cover with approximately the smallest cost:

```
1. While some element of B is not covered
2.     Pick the set $S_i$ with the largest ratio (Number of new elements covered by $S_i$) / $w_i$.
```

Prove that if there is a solution with cost k, then the above algorithm will find a solution with cost $O(k \log n)$.

Hint: You may find the following inequalities useful. For $x \in \mathbb{R}$, $-1 < x < 1$,

$$1 + x \leq e^x \leq 1 + x + x^2.$$

For $x \in \mathbb{R}$, $x > -1$,

$$\frac{x}{1+x} \leq \ln(1+x) \leq x.$$

For $x, y > 0$, $x \leq y \iff \ln x \leq \ln y$; For $x, y \in \mathbb{R}$, $x \leq y \iff e^x \leq e^y$;

Solution: We will prove that if there is a solution of cost k, then the above greedy algorithm will find a solution with cost at most $k \log_e n$. Our proof is similar to the proof in the unweighted case in Section 5.4 of the textbook.

After t iterations of the algorithm, let n_t be the number of elements still not covered, so $n_0 = n$. Since the remaining n_t elements are covered by a collection of sets with cost k, there must be some set S_i such that S_i covers at least $w_i n_t / k$ new elements. (This is easiest to see by contradiction: if every set S_i covers less than $w_i n_t / k$ elements, then any collection with total weight k will cover less than $k n_t / k = n_t$ elements.) Therefore the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t w_{i_t}}{k} = n_t (1 - w_{i_t}/k).$$

Now, we apply the fact that for any x, $1 + x \leq e^x$ by letting $x = -w_{i_t}/k$. Note that equality holds iff $x = 0$:

$$n_{t+1} < n_t e^{-w_{i_t}/k}.$$

On the one hand we have $\prod_{T=0}^{t-1} \frac{n_{T+1}}{n_T} = \frac{n_t}{n_0}$. On the other hand, $\prod_{T=0}^{t-1} \frac{n_{T+1}}{n_T} \leq \prod_{T=0}^{t-1} e^{-w_{i_T}/k} = e^{-\sum_{T=0}^{t-1} w_{i_T}/k}$. Therefore we find that $n_t < n_0 e^{-\sum_{T=0}^{t-1} w_{i_T}/k}$.

Let $k_t = \sum_{T=0}^{t-1} w_{i_T}$, then k_t is the total cost of the sets chosen by the greedy algorithm after t iterations. If we choose the smallest t such that $k_t \geq k \ln n$, then, n_t is strictly less than 1, which means no elements remain to be covered after t steps. Recall that before the algorithm terminates, there always exists a set that covers at least $w_i n_t / k$ elements. However there are only n_t elements remains, thus $w_i n_t / k \leq n_t \implies w_i \leq k$. Since $k_{t-1} < k \log_e n$ and we will never add a set of weight more than k, it follows that $k_t < k \ln n + k = O(k \log n)$.

Running Time (Not required for this problem)
For a loose running time bound, note that the while loop runs for at most m iterations (one for each set S_i in the worst case). Further, in each iteration, computing the set with the best ratio takes at most mn time, because at each step there are at most m sets left and each has size at most n. Thus the total time taken is $O(m^2n)$. Note the main point of this problem was to get the correct approximation ratio, any reasonable low-order polynomial bound on the running time is fine.

5. (★★★★ level) Quaternary Huffman

Quadmedia Disks Inc. has released a quaternary hard disk. Each cell on a disk can now store values 0, 1, 2 or 3 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size n, where the characters occur with known frequencies $f_1, f_2, \ldots, f_n$. Your algorithm should encode each character with a variable-length codeword over the values 0, 1, 2, 3 such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Your proof of correctness should prove that your algorithm achieves the maximum possible compression.
Please provide the main idea and proof of correctness only. If you think pseudocode can better convey your idea, add it to the *Main Idea*.
If you are stuck on the proof of correctness, please see proof of binary huffman from lecture.

Solution:

Main Idea As in the binary case, the idea is to repeatedly delete the four characters of smallest frequency, and replace them by a new character whose frequency is their sum. We wish to show that the quaternary tree we get by this process is optimal. There is just one hitch — what happens if in the last step of the process we are left with only two or three characters. To see when this could happen, note that each step of the above process removes four characters and adds one, for a net decrease of 3. So if the number of characters we start with is of the form $3k+1$, we will eventually get down to exactly 4 characters; whereas if the number is of the form $3k+2$ we get down to 2 characters, or if the number is of the form $3k$ we get down to 3, leading to a problem. The easiest fix is to add one or two dummy characters of frequency 0 if the number of characters you start with is not of the form $3k+1$, thus making sure that the above procedure always results in a full quaternary tree.

Algorithm

```
QUATERNARY HUFFMAN($f[1..n]$):
1.  If $n \not\equiv 1 \mod 3$, set $f[n+1] = 0$ and set $n = n+1$
2.  If $n \not\equiv 1 \mod 3$, set $f[n+1] = 0$ and set $n = n+1$
3.  Let $H$ be a priority queue of integers
4.  For $i = 1$ to $n$: insert $i$ into $H$ with priority $f[i]$
5.  Set $l = n$
6.  While $H$ has more than one element
7.      Set $l = l+1$
8.      $i$ = deletemin($H$), $j$ = deletemin($H$), $k$ = deletemin($H$), $t$ = deletemin($H$)
9.      Create a node numbered $l$ with children $i, j, k, t$
10.     $f[l] = f[i] + f[j] + f[k] + f[t]$
11.     Insert $l$ into $H$ with priority $f[l]$
12. Return the tree rooted at node $l$
```

After running the above algorithm, we can assign codewords in the same manner as in normal Huffman encoding– i.e. write 0, 1, 2 and 3 on the outgoing edges of each node in the tree and encode a character by the sequence of digits encountered along the path from the root of the tree to the leaf corresponding to that character.

Proof of Correctness First we observe that adding a character with a frequency of 0 does not change the maximum possible compression since the length of the codeword associated to that character does not affect the length of any encoded message. So it suffices to prove that the above algorithm gives the optimal encoding tree when $n = 3k+1$.

Now observe that when $n = 3k+1$, any optimal encoding tree must be a full quaternary tree. i.e. every internal node has exactly four children. To see this, note that if a node has three children and is not at the second from the bottom level of the tree, then we can simply move a leaf from the bottom level to be its child, and improve the cost of the solution. Similarly we can move the leaves at the bottom level around without affecting the cost, so there can be at most one node in the tree at the second from the bottom level, which has two or three children. But as we argued earlier, every time we go up on the tree, if the node has d children, we remove d nodes and add one back, for a net dcrease of $d-1$. Thus if the total number of leaves is of the form $3k+1$, such a node cannot exist, otherwise we cannot get exactly one node at the root.

Now we argue as in the binary case that the three lowest frequency leaves must be at the bottom level, since otherwise we could exchange them with whichever leaves were at the bottom level to decrease the cost. Finally, we can always move around leaves at the same level without affecting the cost, and therefore we can assume that the four lowest frequency characters are siblings on the bottom level of the tree.

At this point, the rest of the proof is nearly identical to the proof of optimality for binary Huffman encoding, but we present it here for completeness. We will proceed by induction on the number of characters. The base case, $n = 4$, is trivial (it is clearly optimal to assign each character a length 1 code, which is exactly what the algorithm above does). So assume for induction that $n = 3k+1$ for $k > 1, k \in \mathbb{N}$, and the quaternary Huffman encoding is optimal when there are $n-3$ characters. Let $f_1, f_2, \ldots, f_n$ be a set of frequencies for n characters and let T be the tree constructed by quaternary Huffman encoding from these frequencies. Let S be an optimal encoding tree for a quaternary prefix free encoding of the n characters such that the four lowest frequency characters are siblings on the bottom level of S.

Let i, j, k and t be the indices of the four lowest frequency characters. Consider the set of characters that results from eliminating i, j, k and t and replacing them with a new character l with frequency $f_i + f_j + f_k + f_t$. Let S' and T' denote the result of deleting i, j, k and t from S and T respectively. S' and T' can be thought of as encoding trees for this new set of characters. Observe also that T' is exactly the tree that quaternary Huffman encoding would construct given this new set of characters, so by the inductive assumption T' is optimal.

For an encoding tree R, let $d_R(x)$ denote the depth of x in R and let $Cost(R)$ denote the expected length of text encoded using R. We have:

$$\begin{aligned}
Cost(T) &= \sum_{x=1}^{n} f_x d_T(x) \\
&= \sum_{x \neq i,j,k,t} f_x d_T(x) + d_T(i)(f_i + f_j + f_k + f_t) \\
&= \sum_{x \neq i,j,k,t} f_x d_{T'}(x) + (d_T'(l) + 1)(f_i + f_j + f_k + f_t) \\
&= Cost(T') + (f_i + f_j + f_k + f_t)
\end{aligned}$$

Similar calculations show that $Cost(S) = Cost(S') + (f_i + f_j + f_k + f_t)$. Since T' is optimal, $Cost(T') \leq Cost(S')$, which implies that

$$Cost(T) = Cost(T') + (f_i + f_j + f_k + f_t) \leq Cost(S') + (f_i + f_j + f_k + f_t) = Cost(S)$$

And so T is optimal.

Running Time (Not required for this problem) Adding all characters to the priority queue takes $O(n\log(n))$ time, assuming we use a standard binary heap. Then on each iteration of the while loop, removing four elements and adding one new one takes $O(\log(n))$ time. Furthermore, since the number of elements in the priority queue is reduced by three on each iteration, after $(n-1)/3$ iterations there is only one element left in the priority queue. So the algorithm takes at most $O(n\log(n))$ time.

6. (★★★★ level) Simple and Naive Cluster Analysis

A long time ago in a galaxy far, far away... there are magicians and wizards who communicate with gravitational waves. You and your team just intercepted n of their messages, and you are only able to observe that two messages are either very similar or very dissimilar. You decide to start with the following simple and naive clustering analysis: partition these n messages into two groups, breaking up as many pairs of dissimilar messages as possible. There are more and more incoming messages, so your team needs to find two groups quickly, even if this means that it's not the best possible solution. Come up with an efficient algorithm that breaks up at least half the number of pairs of dissimilar messages as the best possible solution, and prove your answer.

Hint: Try assigning messages to group one at a time. Consider how many pairs of dissimilar messages are broken up with each iteration.

Solution:

Main Idea: Let the messages be nodes in a graph $G = (V, E)$ and let there be an edge between two nodes if they are dissimilar. The number of pairs of dissimilar messages broken up after partitioning nodes into two disjoint sets A and B (also called a cut) is the number of edges between A and B. One by one, we will assign nodes to either set A or B. When we are considering node v, we should know how many edges it has going into A and how many edges it has going into B. We greedily assign it to the set where it has a smaller total number of edges to. This cuts at least half of the total edges.

Pseudocode:

```
1. Initialize empty sets $A$ and $B$
2. For each node $v \in V$:
3.     Initialize $n_A := 0$, $n_B := 0$
4.     For each $\{v, w\} \in E$:
5.         Increment $n_A$ or $n_B$ if $w \in A$ or $w \in B$, respectively
6.     Add $v$ to set $A$ or $B$ with lower $n_A$ or $n_B$, breaking ties arbitrarily
7. Output sets $A$ and $B$
```

Proof of Correctness: In each iteration, when we consider a vertex v, its edges are connected to other vertices already in these three disjoint sets: A, B, or the set of not-yet-assigned vertices. Let the number of edges in each case be n_A, n_B and n_X respectively. Suppose we add v to A, then n_B edges will be cut, but n_A edges can never be cut. Since $\max(n_A, n_B) \geq \frac{n_A+n_B}{2}$, each iteration will cut at least $\frac{n_A+n_B}{2}$ edges, and in total at least $\frac{|E|}{2}$ of the edges will be cut. Let $\texttt{greedycut}(G)$ be the number of edges cut by our algorithm, and $\texttt{maxcut}(G)$ be the best possible number of edges cut. Thus

$$\frac{\texttt{greedycut}(G)}{\texttt{maxcut}(G)} \geq \frac{\texttt{greedycut}(G)}{|E|} \geq \frac{|E|/2}{|E|} \geq \frac{1}{2}$$

Running Time: For the graph construction step, it takes $O(|V|^2)$ to check every pair (a, b) whether they are dissimilar. For the greedy max-cut algorithm itself, each vertex is iterated through once, and each edge is iterated over at most twice, so the runtime is $O(|V| + |E|)$. Thus ther overall runtime is $O(|V|^2)$.

7. (??? level) (Optional) Redemption for Homework 4

Submit your *redemption file* for Homework 4 on Gradescope. If you looked at the solutions and took notes on what you learned or what you got wrong, include them in the redemption file.

Solution: N/A