Assignment 2

Question 1

(a) Proof that the Set $\mathcal{F}$ is Universal

The goal here is to prove that the given set $\mathcal{F}$ is indeed universal. That is, for any distinct keys $k, l \in U$, the number of functions in $\mathcal{F}$ for which k and l collide is $\leq \frac{|\mathcal{F}|}{m}$. In other words, the number of functions f in that set of functions for which $f(k) = f(l)$ is at most the cardinality of the set divided by m. The example given proves this for the particular case of $m = 3, U = \{a, b\}$, but the goal is to show that it works in general, i.e., for any finite U, m.

To start, utilize the following claim: given the universal set of hash functions, $\mathcal{F}$, if we pick $f \in \mathcal{F}$ uniformly at random, then for all distinct $k, l \in U, \Pr\{f(k) = f(l)\} \leq \frac{1}{m}$.

Proof:

$$\Pr\{f(k) = f(l)\} = \frac{\#\ hash\ functions\ in\ \mathcal{F}\ for\ which\ k\ and\ l\ collide}{total\ \#\ hash\ functions\ in\ \mathcal{F}}$$

$$\Pr\{f(k) = f(l)\} \leq \frac{|\mathcal{F}|/m}{|\mathcal{F}|} = 1/m$$

In words, this says that the probability for a collision is at most $\frac{1}{m}$, and this makes sense because since there are m buckets and the functions are chosen uniformly at random, then each function has an equal chance of being chosen. This means that each individual hash function f in $\mathcal{F}$ is itself universal, again since they are chosen at random, and all have probability $\frac{1}{m}$. Now that each hash function is proven to be universal, this leads to the proof that the entire set $\mathcal{F}$ is also universal. If each function in $\mathcal{F}$ has probability $\frac{1}{m}$, then the probability for the entire set $\mathcal{F}$, given $|\mathcal{F}|$ is the cardinality of $\mathcal{F}$, is:

$$\sum_{i=0}^{|\mathcal{F}|} \Pr\{f_i(k) = f_i(l)\} \leq \underbrace{\frac{1}{m} + \frac{1}{m} + \frac{1}{m} + \cdots + \frac{1}{m}}_{|\mathcal{F}|\ times} = \frac{|\mathcal{F}|}{m}$$

The probability of the entire set is simply the linear addition of the probability of each hash function f in $\mathcal{F}$. Thus, the set $\mathcal{F}$ is universal with the probability of at most $\frac{|\mathcal{F}|}{m}$.

Paolo Torres

Assignment 2

(b) Practicality of this Method for Generating Universal Sets

Given this universal set $\mathcal{F}$, there is only one way to choose from it, which is uniformly at random. The way $\mathcal{F}$ is generated, given U, m is to simply adopt the set of all functions with domain U and co-domain $\{0, \dots, m-1\}$, where the domain pertains to the finite universe of keys, and the co-domain pertains to the number of buckets. The goal here is to describe the practicality of adopting such a method for generating this set as the universal set.

Based on this method, this is not a practical way for generating a universal set. Once the hash function is chosen from the universal set, there exists a need for encoding or representing this hash function in some way, for the purposes of storing it alongside the hash table and utilizing it in some useful manner. A natural way to encode or represent these functions is with a table. For example, from part (a), given a function $f: U \to \{0, \dots, m-1\}$, one could store it as a table, which contains, for each $k \in U$, whatever values from $\{0, \dots, m-1\}$ the function f maps k to. More specifically, given domain $U = \{k_1, \dots, k_n\}$, where n is the number of keys, and co-domain $\{0, \dots, m-1\}$, the first column of the table could be the keys $k_1, \dots, k_n$, and the second column could be what each k_i is mapped to:

Keys	**Values**
k_1	$k_1\ mapping$
k_2	$k_2\ mapping$
...	...
k_n	$k_n\ mapping$

The downfall of this method pertains to the size of this encoding for such a table. For each key, since there are m buckets, then there are m different ways in which each key can be mapped, since each key must correspond to at least one bucket. Furthermore, since there are n keys, then the total size of this table is $n \times m$. This is a large amount of space to take up for such a task, especially considering if this were to get scaled up. Introducing a new key would add an entire set of m mappings to the table, since it is possible for this new key to map to anything else already present in the table. Hence, this is not a practical way of generating a universal set, and the following paragraph describes a better method.

For a hash function that is chosen from the universal set that is proposed in the lectures, the data does not need to be stored as a table. This function, g, given some prime p such that

Assignment 2

$U \subseteq \{0, 1, \dots, p-1\}$, some key k, $a \in \{1, 2, \dots, p-1\}$, $b \in \{0, 1, \dots, p-1\}$, and number of buckets m, can be represented as:

$$g(k) = \big((a \cdot k + b) \bmod p\big) \bmod m$$

Recall back to the tabular method, if one could control U, then the hash function could have been constructed concisely without use of a table, however U is strictly a finite universe of keys, nothing else. In the case of this modular-based function g above, there is more freedom, in that U can be modified to allow a more efficient encoding of the mapping. In the lecture, the domain U for that example was the set of 8 digit numbers to represent Student IDs as the keys. This allows the mapping to exist without the need of a table for keeping track of the key-value pairs, resulting in improved efficiency and reduced space usage. In this case where U is not modifiable, there does not seem to be a compact and unambiguous encoding of the hash function that is chosen from that universal set. Therefore, the method of generation of universal sets given in this problem is not a practical method.

Paolo Torres

Assignment 2

Question 2

First, start off with an example to intuit what $GetPivot$ does if $n \geq 5$:

$$Example: [2, 3, 4, 5, 6, 12, 13, 14, 15, 1, 7, 8, 9, 10, 11]$$

In this case, $n = 15$, which is divisible by 5. Now, the array is split into subarrays:

$$[2, 3, 4, 5, 6], [12, 13, 14, 15, 1], [7, 8, 9, 10, 11]$$

It then computes the median of each subarray, which are $[4, 13, 9]$ in this case. If $r = 1$, then the array would be split around the 9 and the algorithm recurses on the left portion. Recall partitioning or splitting around the pivot v from the lectures:

Rearrange items in S as follows:

- Move every item $< v$ to the left of v.
- Move every item $> v$ to the right of v.
- Thus, every item whose value is v ends up where it would in a sorted permutation of S.

So, if the array is split around 9 as the pivot, the array ends up becoming:

$$\{1, 2, \dots, 8\}, 9, \{10, \dots, 15\}$$

In this case, the first eight indices and last six indices of the array contain the items that are in the correct position relative to the index with value 9. As a result, the pivot ends up in the correct position if the array were to be sorted. Performing this algorithm on the input array:

$$[2, 3, 4, 5, 6, 1, 7, 8, 9, 12, 13, 14, 15, 11, 10]$$

Finally, as $r = 1$, the algorithm recurses on the part of the array to the left of 9, which is $[2, 3, 4, 5, 6, 1, 7, 8]$. This part of the array has a size of 8, and the claim is that the part of the array that the algorithm recurses on has a size of at most $\lceil {}^{7}/_{10} \cdot n \rceil$, where n is the size of the input array. Applying the claim with the input size of $n = 15$:

$$8 \leq \lceil {}^{7}/_{10} \cdot n \rceil = \lceil {}^{7}/_{10} \cdot 15 \rceil = 10.5$$

In this case, the following claim is true, and now utilize the hint given in the question for more insight. Group the ${}^{n}/_{5} = {}^{15}/_{5} = 3$ subarrays, given $m = 9$, into two groups as follows:

$$G_{lo} = \{2, 3, 4, 5, 6\}, \{7, 8\}, G_{hi} = \{12, 13, 14, 15, 1\}, \{10, 11\}$$

Now, there are approximately ${}^{1}/_{2} \times {}^{n}/_{5} = 1.5$ pieces in each of G_{lo} and G_{hi}, which makes sense since values are either $< m$ or $> m$, as the given input array is said to be of distinct integers. In this example, all 7 values in G_{lo} are $< m$, and ${}^{6}/_{7}$ values in G_{hi} are $> m$. This leads

to the following observation: in the worst case, all values in G_{lo} would be $< m$, and none of the values in G_{hi} would be $< m$. In a similar way, none of the values in G_{lo} would be $> m$, and all of the values in G_{hi} would be $> m$.

Now the following observation is made: since each subarray is guaranteed to be of size 5, as stated in the problem, then no matter how many subarrays there are, there will always be 2 elements added to G_{lo} from the subarray containing the median m, and there will always be 2 elements added to G_{hi} from this same subarray, in the worst case, in the end. This property will be exploited to prove that the size of the recursed piece is at most $\lceil {}^{7}/_{10} \cdot n \rceil$ in the general case.

Given the general input array $A[1, \dots, n]$, $GetPivot$ perceives it as sequences of 5 items each: $A[1, \dots, 5], A[6, \dots, 10], \dots, A[n-4, \dots, n]$. Then it computes the median of each $A[i, \dots, i+4]$ and stores them as $m_1, m_2, \dots, m_{{}^{n}/_{5}}$. Finally, it computes the median, m, of these medians. Now, the G_{lo} and G_{hi} groups are as follows:

$$G_{lo} = (subarrays_{\,median<m})(2\ values < m\ from\ subarray_{median=m})$$

$$G_{hi} = (subarrays_{median>m})(2\ values > m\ from\ subarray_{median=m})$$

The subarrays with median $< m$ and $> m$ can be any number of sequences of 5 items each, but in the worst case, are equal (e.g., in the example above, each group has one subarray). Since the requirement is $n \geq 5$, the base case is if the input array is only 5 elements, then G_{lo} and G_{hi} only have 2 elements each, which results in the following recurse size:

$$2 \leq \lceil {}^{7}/_{10} \cdot 5 \rceil = 3.5$$

The base case holds, and for the step, for every addition of 5 elements to the input array, the G_{lo} and G_{hi} sizes also increase by 5:

Sizes of G_{lo} and G_{hi}	n	$\lceil {}^{7}/_{10} \cdot n \rceil$
7	15	10.5
12	25	17.5
…	…	…
${}^{(n-1)}/_{2}$	n	$\lceil {}^{7}/_{10} \cdot n \rceil$

As shown above, in the worst case, for each of G_{lo} and G_{hi}, they are always $\leq \lceil {}^{7}/_{10} \cdot n \rceil$, thus they are upper bounded by this size. Hence, every recursed piece is at most $\lceil {}^{7}/_{10} \cdot n \rceil$.

Assignment 2

Question 3

The proof strategy used for this case will be a "cut and paste" strategy, which consists of a proof by induction on k, and then a proof that g_{k+1} cannot exist. We prove two claims in order:

Claim 1: *Suppose for some sequence of stops, $O = \{o_1, \dots, o_k\}$ is an optimal set of stops which are chosen only if you cannot make it to the next stop before nightfall, else continue driving. Suppose our greedy algorithm outputs $G = \{g_1, \dots, g_l\}$, where stops are made as furthest as possible before the nightfall condition. Then, it is true that: for every $i = 1, 2, \dots, l$, $s(g_i) \geq s(o_i)$, where $s(\cdot)$ is the stop distance from the source city A.*

Proof. Note: it must be the case that $l \geq k$. And therefore, $l = k$, i.e., greedy is optimal. Proof by induction on k. Base case: $k = 1$. In our greedy algorithm, we first pick the furthest stop possible such that the next one cannot be made by nightfall. Therefore, immaterial of what o_1 is, $s(g_1) \geq s(o_1)$. In words, this means that the greedy stop choice g_1 is at least as far from the source city A as the corresponding o_1. Equivalently, if $g_1 = o_1$, then O is a solution. Induction assumption: It is true that $s(g_i) \geq s(o_i)$ – maximized stop distances from city A. Step: Since the greedy algorithm stops as far as possible from city A, then this implies that the stop o_1 is before the stop g_1. Now, since O is an optimal solution, then we know that $o_1 \to o_2$ is a legal drive and can be done before nightfall. Hence, given that o_1 is before g_1, then this implies that $g_1 \to o_2$ is also a legal drive. The distance between g_1 and o_2 is $\leq$ the distance between o_1 and o_2, so it can be done before nightfall. Similarly, the distance between g_{i-1} and o_i is $\leq$ the distance between o_{i-1} and o_i. Therefore, $s(g_i) \geq s(o_i)$, meaning that any stop g_i is legal and is at least as far from the source A as the corresponding o_i.

Claim 2: *Given sets O, G as in Claim 1, g_{k+1} cannot exist in G.*

Proof. By Claim 1, $s(g_i) \geq s(o_i)$. Since the distance between g_k and o_{k-1} is $\leq$ the distance between o_{k-1} and o_k, then there is no guarantee that $o_k \to g_{k+1}$ is a legal drive. Hence, there is a contradiction to the assumption that the stops are chosen only if the next stop cannot be made before nightfall, so g_{k+1} cannot exist.

Paolo Torres

Assignment 2

Question 4

(a) Polynomial-Time Algorithm Description, Correctness, and Analysis

The algorithm starts by initializing a list of $distances$ all to $infinity$ to represent the shortest distances from the source vertex to every other vertex, and a $visited$ list all to $false$ to keep track of already visited vertices. The source vertex is set to a distance of 0, and then the $getShortestPaths$ function is invoked. This method performs Dijkstra's algorithm to determine the shortest paths. For each vertex, the minimum distance from all adjacent vertices to said vertex is computed via $getMinDistance$. This is done by iterating through the set of unvisited vertices and storing the lowest one. Once the minimum distance vertex is determined, this vertex is marked as visited and the algorithm continues. Then, for each adjacent vertex to this visited vertex (i.e., has an edge between them), the $updateDistance$ function is invoked. In this method is where the distance updating from Dijkstra's algorithm is performed, with the following conditional:

$$distances[index] > distances[u] + value$$

This says that if this next vertex distance is larger than the current vertex distance plus the current edge weight, then make this the new next vertex distance. This is done for each vertex, resulting in the shortest paths from the source to every other vertex.

Now that the shortest paths have been obtained, the next step is to determine the number of distinct shortest paths. This is done with the $traverse$ method, which also tracks the current vertices, distances, and number of paths. This method is implemented using a modified depth-first search recursive strategy. The base case: if the target vertex has been reached, and the target distance has been computed, then add one to the number of paths. The recursive case: for each adjacent vertex to the current vertex, compute its index and edge weight. If the source vertex has been reached, or the vertex has already been visited, then skip over it. Else, calculate the new current distance and append this vertex to its current path. Then, perform a recursive call with these new index, value, current distance, and current path parameters. After the recursive call, subtract the previous current distance and pop off the processed vertex to maintain the backtracking. By the end of this traversal, the final number of distinct shortest paths is obtained.

Paolo Torres

Assignment 2

To prove correctness, the two main parts of the algorithm will be proven separately: the use of Dijkstra's algorithm for shortest paths, and the traversal of the graph to determine number of distinct shortest paths. For Dijkstra's, prove:

For every vertex u that is reachable from s, at the moment u is extracted from the priority queue, it is the case that $dist(u) = \delta(s, u)$.

Derive a contradiction to a claim that $dist(u) \neq \delta(s, u)$.

- $dist(y) = \delta(s, y)$ – by Claim (2) from lectures.
- $\delta(s, y) \leq \delta(s, u)$ – by Claim (1) from lectures, and all edge weights are non-negative.
- $\delta(s, u) \leq dist(u)$ – by Claim (3) from lectures.
- $dist(u) \leq dist(y)$ – about to extract u from the priority queue.

$$dist(y) = \delta(s, y) \leq \delta(s, u) \leq dist(u) \leq dist(y)$$
$$\Longrightarrow dist(y) = \delta(s, y) = \delta(s, u) = dist(u)$$

Attained desired contradiction, therefore the Dijkstra's algorithm for shortest paths is correct in this case. Next, prove that the modified depth-first search recursive traversal is correct, in the case that it returns the correct number of distinct shortest paths. This traversal is modified in the sense that it accounts for the following special cases:

- If the target and target distance have been obtained, then add to the number of paths.
- If the source has been reached during a traversal, then detect a cycle.
- If the current vertex already exists in the current path, then detect a cycle.

Aside from these cases, every other invocation result in a distance to be considered, added or subtracted, and a path to be appended or popped, given that $G = \langle V, E, l \rangle$, where $l: E \to \mathbb{Z}^+$. Hence, this traversal method is correct in this case.

For the analysis of the worst-case running time as a function of the size of the input, n in $O(\cdot)$ notation, consider again the two main parts of the algorithm: Dijkstra's algorithm for shortest paths and the graph traversal for number of distinct shortest paths. First, identify n as the size of the input, which can be represented as $|V| + |E|$, which is the size of the input graph $G = \langle V, E, l \rangle$. For Dijkstra's algorithm, the priority queue to determine the next vertex to explore is implemented with an array data structure, which has time $O(|V|^2)$. This is because every time the next minimum distance vertex is required, in the worst-case, the entire list could be iterated on to obtain this vertex. For

the graph traversal, normally a depth-first search recursive approach would take $2^{|V|+|E|}$, where each vertex would have to explore every path it could take. However, as mentioned previously, this one is modified to be more efficient. Explorations are terminated for the following two cases:

- If the source has been reached during a traversal, then detect a cycle.
- If the current vertex already exists in the current path, then detect a cycle.

More importantly, rather than having to traverse all possible paths for each vertex, the algorithm only has to traverse each path starting from the source vertex. This property is exploited, hence in the worst-case, there are much less explorations to be made. In particular, starting at a source vertex a, there are $|V|$ other vertices to be explored. Then, for the $|V|$ number of vertices, there are $|V|$ number of permutations, or ways in which they can be ordered to possibly arrive at different distance values. However, many of these explorations are short-circuited if a cycle is detected, or in other words, if the algorithm traverses back to the source, or if it reaches an already visited vertex. Therefore, the running time of this modified graph traversal is also $O(|V|^2)$ as a result of these optimizations. Hence, the total running time for this algorithm is $O(|V|^2 + |V|^2) = O(2|V|^2)$. When simplified, the final running time is $O(|V|^2)$.

Assignment 2

Question 5

The goal of this problem is to devise an algorithm that asymptotically converges faster the more eggs that are available. In other words, if one more egg is available, the algorithm converges much faster, in the limit for the number of floors. Let $d(k, n)$ be the worst-case number of drops of an egg needed for a building of n floors given k eggs. For all $k \in \mathbb{N}$, the algorithm should possess the following property:

$$\lim_{n \to \infty} \frac{d(k+1, n)}{d(k, n)} = 0$$

The following algorithm is devised, first consider the problem from a different perspective: rather than seeking the number of moves given the number of eggs and number of floors, we instead seek the number of floors given the number of moves and number of eggs. In other words, we seek to instead determine what is the maximum number of floors we can check given a certain number of moves and number of eggs. This change in perspective creates an alternative setup for the problem, in that the problem can be solved by breaking it down into smaller subproblems. Namely, for each floor, we can determine all the different number of moves and eggs requirements, and then build up on those subproblem solutions. This is because for the lowest floor where an egg first breaks, any floor higher will also break, but any floor lower will not break. So, this property can be exploited to be solved with a subproblem build up technique, as previous solutions can be used to solve future solutions.

The algorithm will use the approach of dynamic programming, a concept from week 7 of the course, to solve this problem. So first, a dynamic programming 2D array dp is defined:

```
vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));
```

It is given the size of $k \times n$, since we need to determine the number of floors for each number of eggs. Then, a $moves$ variable is initialized to 0:

```
int moves = 0;
```

Next, a $while$ loop is run for each number of moves up to the number of floors:

```
while (dp[moves][k] < n)
```

Within the $while$ loop, iterate $moves$, and a for loop is run for each $egg \in k$:

```
for (int egg = 1; egg <= k; egg++)
```

Paolo Torres

Assignment 2

Now for the most important part: for each combination of number of moves and eggs, compute the number of moves and eggs for the previous solution. This is the dynamic programming approach and exploits the subproblem property it possesses:

```
dp[moves][egg] = dp[moves - 1][egg - 1] + dp[moves - 1][egg] + 1;
```

In the for loop iteration for each egg, if the egg breaks, then subtract 1 from the egg index, since that floor cannot handle an egg drop: $dp[moves - 1][egg - 1]$. Conversely, if the egg does not break, then do not subtract 1 from the egg index, since that floor can handle an egg drop without breaking: $dp[moves - 1][egg]$. Finally, add 1 since we are iterating for each floor: 1.

This algorithm works and can determine the number of moves needed for any number of eggs and floors, where $d(k, n)$ returns the result. For example:

Eggs (k)	**Floors ($n = 10,000,000$)**
1	10,000,000
2	4,472
3	392
4	125
5	67
...	...
100	24

As shown in the table, the number of moves $d(k, n)$ in this case is constantly decreasing, so $d(k + 1, n)$ is strictly less than $d(k, n)$, or $d(k + 1, n) < d(k, n)$. It turns out this is also the case in general, as more eggs allows the algorithm to converge faster, since there are more subproblems containing a different number of eggs for building up the solution.

Lastly, it can be seen that as the number of eggs increases, the difference between the number of moves decreases. Mathematically, $d(k - 1, n) - d(k, n) > d(k, n) - d(k + 1, n)$. As n increases, this difference also increases, where the difference is most when comparing $d(k + 1, n)$ and $d(k, n)$. Therefore, $d(k + 1, n) \ll d(k, n)$ as n increases, or in other words, as n tends towards infinity. This leads to the following property holding:

$$\lim_{n \to \infty} \frac{d(k + 1, n)}{d(k, n)} = 0$$

This is precisely what we seek to prove, where adding just one more available egg significantly improves convergence time, as the number of floors tends towards infinity.

Paolo Torres