# Question 1

1. **Proof that the Set is Universal**

The goal here is to prove that the given set is indeed universal. That is, for any distinct keys , the number of functions in for which and collide is . In other words, the number of functions in that set of functions for which is at most the cardinality of the set divided by . The example given proves this for the particular case of , but the goal is to show that it works in general, i.e., for any finite .

To start, utilize the following claim: given the universal set of hash functions, , if we pick uniformly at random, then for all distinct .

Proof:

In words, this says that the probability for a collision is at most , and this makes sense because since there are 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 in is itself universal, again since they are chosen at random, and all have probability . Now that each hash function is proven to be universal, this leads to the proof that the entire set is also universal. If each function in has probability , then the probability for the entire set , given is the cardinality of , is:

The probability of the entire set is simply the linear addition of the probability of each hash function in . Thus, the set is universal with the probability of at most .

1. **Practicality of this Method for Generating Universal Sets**

Given this universal set , there is only one way to choose from it, which is uniformly at random. The way is generated, given is to simply adopt the set of all functions with domain and co-domain , 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 , one could store it as a table, which contains, for each , whatever values from the function maps to. More specifically, given domain , where is the number of keys, and co-domain , the first column of the table could be the keys , and the second column could be what each is mapped to:

|  |  |
| --- | --- |
| **Keys** | **Values** |
|  |  |
|  |  |
|  |  |
|  |  |

The downfall of this method pertains to the size of this encoding for such a table. For each key, since there are buckets, then there are different ways in which each key can be mapped, since each key must correspond to at least one bucket. Furthermore, since there are keys, then the total size of this table is . 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 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, , given some prime such that , some key , , , and number of buckets , can be represented as:

Recall back to the tabular method, if one could control , then the hash function could have been constructed concisely without use of a table, however is strictly a finite universe of keys, nothing else. In the case of this modular-based function above, there is more freedom, in that can be modified to allow a more efficient encoding of the mapping. In the lecture, the domain for that example was the set of 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 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.

# Question 2

First, start off with an example to intuit what does if :

In this case, , which is divisible by . Now, the array is split into subarrays:

It then computes the median of each subarray, which are in this case. If , then the array would be split around the and the algorithm recurses on the left portion. Recall partitioning or splitting around the pivot from the lectures:

Rearrange items in as follows:

* Move every item to the left of .
* Move every item to the right of .
* Thus, every item whose value is ends up where it would in a sorted permutation of .

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

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 . 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:

Finally, as , the algorithm recurses on the part of the array to the left of , which is . This part of the array has a size of , and the claim is that the part of the array that the algorithm recurses on has a size of at most , where is the size of the input array. Applying the claim with the input size of :

In this case, the following claim is true, and now utilize the hint given in the question for more insight. Group the subarrays, given , into two groups as follows:

Now, there are approximately pieces in each of and , which makes sense since values are either or , as the given input array is said to be of distinct integers. In this example, all values in are , and values in are . This leads to the following observation: in the worst case, all values in would be , and none of the values in would be . In a similar way, none of the values in would be , and all of the values in would be .

Now the following observation is made: since each subarray is guaranteed to be of size , as stated in the problem, then no matter how many subarrays there are, there will always be elements added to from the subarray containing the median , and there will always be elements added to 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 in the general case.

Given the general input array , perceives it as sequences of items each: . Then it computes the median of each and stores them as . Finally, it computes the median, , of these medians. Now, the and groups are as follows:

The subarrays with median and can be any number of sequences of items each, but in the worst case, are equal (e.g., in the example above, each group has one subarray). Since the requirement is , the base case is if the input array is only elements, then and only have elements each, which results in the following recurse size:

The base case holds, and for the step, for every addition of elements to the input array, the and sizes also increase by :

|  |  |  |
| --- | --- | --- |
| **Sizes of and** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

As shown above, in the worst case, for each of and , they are always , thus they are upper bounded by this size. Hence, every recursed piece is at most .

# Question 3

The proof strategy used for this case will be a “cut and paste” strategy, which consists of a proof by induction on , and then a proof that cannot exist. We prove two claims in order:

Claim 1: *Suppose for some sequence of stops, 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 , where stops are made as furthest as possible before the nightfall condition. Then, it is true that: for every , , where is the stop distance from the source city A.*

*Proof.* Note: it must be the case that . And therefore, , i.e., greedy is optimal.

Proof by induction on . Base case: . 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 is, . In words, this means that the greedy stop choice is at least as far from the source city A as the corresponding . Equivalently, if , then is a solution.

Induction assumption: It is true that – 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 is before the stop . Now, since is an optimal solution, then we know that is a legal drive and can be done before nightfall. Hence, given that is before , then this implies that is also a legal drive. The distance between and is the distance between and , so it can be done before nightfall. Similarly, the distance between and is the distance between and . Therefore, , meaning that any stop is legal and is at least as far from the source A as the corresponding .

Claim 2: *Given sets as in Claim 1, cannot exist in .*

*Proof.* By Claim 1, . Since the distance between and is the distance between and , then there is no guarantee that 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 cannot exist.

# Question 4

* The size of the input, , is the size of the graph:
* For the coding part, can use an array-based implementation of a priority queue
* If you have no path , then you have no shortest path , and therefore the number of shortest paths is

1. **Polynomial-Time Algorithm Description, Correctness, and Analysis**

# Question 5

* If we have one more egg, we converge much faster, in the limit for the number of steps
* Worst case number of drops of an egg you need for a building of floors given eggs is . For all :

With we have:

Since we just have to do a linear search. With :

Since the best we can do is to drop the egg from floor, then do a linear search for the rest. Calculating the limit for these two cases:

Need to prove that our algorithm has the limit property.

We you have a possible non-integer, you can always just choose an integer that is close by, e.g., or .

The bottom line: Does it meet the criterion as expressed by the limit?

# References

**There are no sources in the current document.**