Fall '15 CS 5800 Algorithms | Hackerrank Group: akashsingh245

Problem Set 4

Surekha Jadhwani, Akash Singh, and Ayush K. Singh

1 Part A

Answers

1. "Lazy" kingdom lies in the mountains. The King of Lazy is re-knowned as the laziest person on earth. His palace is on the top of the highest mountain in Lazy. A scenic beach, most beautiful on earth, is located at the foot of the mountain. Each day at noon, when the King wakes up, he likes to saunter down to the beach. The lazy King does not like to walk upward, only downward. Of course to return to his palace, the King uses a **one-way** cable car from the beach to his palace. To add some variety to his otherwise excellent existence the King wishes to use a new route from the palace to the beach each day. Of course, the world is finite and there are only a finite number of routes so the question is: what is the most number of successive days he can go without repeating a path. Also, being lazy he is interested in short routes even though they are all downward.
   More formally, there are $S = \{s_1, s_2, \cdots, s_n\}$ stations in Lazy, with s_1 located in the palace and station s_n located on the beach. The altitude of station s_i is $a_i(for\ i = 1, \cdots, n)$. All altitudes are distinct. Since the palace is on the top of the mountain and the beach on the bottom it holds that $a_1 > a_j > a_n$, for every $j = 2, \cdots, n-1$. There exists a collection $T \subseteq S$ x S of trails. The length of a trail $(s, s') \in T$, denoted by $len(s, s')$, is a positive integer. A route is a sequence of concatenated trails $R = \langle s_{i_1}, s_{i_2}, ..., s_{i_k} \rangle$. Extending the definition of length from a trail to a router we have that $len(R) = \sum_{j=1}^{k-1} len(s_{i_j}, s_{i_{j+1}})$. Route R is said to be downwards , if $a_i > a_{i_{j+1}}$,for every $j = 1, ..., k-1$. Remember that the king only likes to take downward routes. Please help the King!!! Give efficient algorithms (along with proofs of correctness and run-time analysis) to solve the following two problems, in aid of the King.

   (a) What is the maximum number of days the King can go from the palace to the beach without repeating a (previously taken) route?

   Solution:

   i. **Intuition and recursion:**
   Here, we want to find the number of unique paths from palace(s_1) to the beach(s_n).
   The total number of paths from palace to the beach is nothing but the sum of number of paths via each of it's connected stations at a lower altitude.
   Hence, our main problem is reduced to solving multiple sub-problems of counting the paths from lower level stations.
   Let us define, for each vertex $s_i \in S$, $Count(s_i)$ to be an array containing the

number of possible paths from vertex s_i to the destination(beach) s_n
Also, the number of paths from destination to itself is 1.
This gives us the recursion:

$$Count(s_i) = \begin{cases} 1 & \text{if } s_i = s_n \\ \sum_{s_i, s_j \in T} Count(s_j) & \text{if } a_i > a_j \end{cases}$$

So, $Count(s_1)$ will give the final answer.

ii. **The algorithm:**
This implementation is based upon the idea of DFS, but we have added dynamic programming(memoization) to it so as to avoid re-computation of already computed values.
The base case for our recursion is when we reach the destination in which case, the number of paths is equal to 1.
Thus our algorithm is:
Pseudocode:

```
//Global variables
Create an Integer array Count[1 ··· n]
Initialize all elements of Count to null

define getUniquePaths:
 return FindCount(s_1);

define FindCount(s_i) :
 if s_i == s_n :
  return 1;
 else:
  if Count[s_i] is not null:
   return Count[s_i]
  else:
   Count[s_i] = 0;
   for each (s_i, s_j) ∈ T
   // for each station connected from current station, do following:
    if a_i > a_j
    // if the trail is downwards
     Count[s_i]+ = FindCount(s_j)
    end if
   end for
   return Count[s_i]
  end if
 end if
end function FindCount
```

iii. **Correctness:**
Termination: The given algorithm visit each trail exactly once. Since, the collection T is a finite set, the algorithm terminates successfully.
Correctness: In order to calculate the number of paths from palace to beach, we are adding the number of paths via each of it's connected stations at a lower altitude.
Since DFS model is being used, we are getting all the unique paths since no path would be revisited. Also, using the DFS model ensures that we cover all the valid connections in the graph from which a path can be obtained. This ensures that the result we have is correct since we have covered all cases of traveling to the beach from palace to beach using all intermediate stations.

iv. **Analysis:**
Let the total number of stations is n and size of collection of trails T is m.
In the above algorithm, initialization of $Count$ array takes $O(n)$ as the size of array is n.
Since this algorithm is based on DFS and we have avoided re-computation, the complexity cannot be more than that of DFS.
Also, each trail is checked only once. Hence, the total runtime is $O(n+m)$.

(b) Assuming the King does not want to take routes of length greater than $100n$ what is the maximum number of days the King can go from the palace to the beach without repeating a (previously taken) route?

Solution:

i. **Intuition and recursion:**
Here, we want to find the number of unique paths from palace(s_1) to the beach(s_n) where the route length is not greater than $100n$.
Let us define, for each vertex $s_i \in S$, $PathLengths(s_i)$ to be an array containing the list of valid path lengths from vertex s_i to the destination(beach) s_n
For a path from palace to beach with route length not greater than $100n$, the path from a connected station at a lower altitude should be less than $100n$.
So, a single path length from any station to destination is equal to the sum of trail length from current station to one of it's connected station and path length from that connected station to the destination.
Similarly, we can compute all possible paths for any station(s_i) and append it to $PathLengths(s_i)$.
Hence, our main problem is reduced to solving multiple sub-problems of calculating the path lengths from lower level stations and then adding it's own length.

At each step, we will filter the results where path length is more than $100n$
Also, the path length from destination to itself is 0.
This gives us the recursion:

$$PathLengths(s_i) = \begin{cases} 0 & \text{if } s_i = s_n \\ \text{append } len(s_i) + PathLengths(s_j) & \text{otherwise, if valid} \end{cases}$$

So, count of list of elements at $PathLengths(s_1)$ will give the final answer.

ii. **The algorithm:**
This implementation is based upon the idea of DFS, but we have added dynamic programming(memoization) to it so as to avoid re-computation of already computed values.
The base case for our recursion is when we reach the destination in which case, the path length is equal to 0.
Thus our algorithm is:
Pseudocode:

```
//Global variables
Create an array PathLengths[1 ··· n]
Initialize all elements of PathLengths to null
Let limit = 100n

define getUniquePaths:
 return count of elements from FindPaths(s_1)

define FindPaths(s_i) :
 if s_i == s_n :
  return 0;
 else:
  if PathLengths[s_i] is not null:
   return PathLengths[s_i]
  else:
   PathLengths[s_i] = empty;
   for each (s_i, s_j) ∈ T
   // for each station connected from current station, do following:
    if a_i > a_j
    // if the trail is downwards
      temp = FindPaths(s_j)
     for each element p in temp
      if len(s_i, s_j) + p <= limit
       append len(s_i, s_j) + p to PathLengths[s_i]
      end if
     end for
```

```
      end if
    end for
    return PathLengths[s_i]
  end if
 end if
end function FindPaths
```

iii. **Correctness:**
Termination: The given algorithm visit each trail exactly once. Since, the collection T is a finite set, the algorithm terminates successfully.
Correctness: In order to calculate the number of paths from palace to beach, we are adding the number of paths via each of it's connected stations at a lower altitude.
Since DFS model is being used, we are getting all the unique paths since no path would be revisited. Also, using the DFS model ensures that we cover all the valid connections in the graph from which a path can be obtained. This ensures that the result we have is correct since we have covered all cases of traveling to the beach from palace to beach using all intermediate stations. Also, at each step, we are checking the path length to be less than the *limit* value. So, all paths with length more than the *limit* are filtered. Hence, the algorithm is correct.

iv. **Analysis:**
Let the total number of stations is n and size of collection of trails T is m.
In the above algorithm, initialization of $PathLengths$ array takes $O(n)$ as the size of array is n.
Also, each trail is checked only once and the count of possible paths for each vertex in $PathLengths$ won't exceed the number of it's edges.
Since this algorithm is based on DFS and we have avoided re-computation, the complexity cannot be more than that of DFS.
Hence, the total runtime is $O(n + m)$.

2. **Median.**
The median finding algorithm presented in class divided the elements into groups of 5.

(a) How much time will the median finding algorithm take if groups of 3 are used? Give and solve the appropriate recurrence.
Solution:
Let $T(n)$ denote the worst-case time to find the median of n elements by splitting it in groups of 3.
So, finding the median from each group of 3 elements will take $O(n)$ time as it takes a constant time to find median of 3 elements.
Finding the true median out of all the medians calculated above will take at most $T(n/3)$. Splitting the array into LESS and GREATER using median as pivot will also take $O(n)$. Now, the number of elements more than median is at most:
$2(\lfloor\frac{1}{2}\lfloor\frac{n}{3}\rfloor\rfloor) \leq \frac{n}{3}$
Similarly, the number of elements less than median can be calculated.
So, the algorithm is called recursively on $\frac{2n}{3}$ elements.
Hence, the recurrence relation is:
$T(n) \geq T(\lceil n/3\rceil) + T(2n/3) + \Omega(n)$
i.e., $T(n) \geq T(n/3) + T(2n/3) + \Omega(n)$
Solving the above recurrence by unrolling, we get,
$T(n) \geq c\frac{n}{3}log\frac{n}{3} + c\frac{2n}{3}log\frac{2n}{3} + an$
i.e., $T(n) \geq c\frac{n}{3}log\frac{n}{3} + c\frac{2n}{3}log\frac{2n}{3} + an$
i.e., $T(n) \geq cnlogn - cnlog3 + \frac{2}{3}cn + an$
For very large values of n and appropriate c ($c \leq \frac{a}{log3 - 2/3}$):
$T(n) \geq cnlogn$
Hence, the run-time complexity for median finding algorithm when groups of 3 are used is $T(n) = \Omega(nlogn)$.

(b) How much time will the median finding algorithm take if groups of 7 are used? Give and solve the appropriate recurrence.
Solution: Let $T(n)$ denote the worst-case time to find the median of n elements by splitting it in groups of 7.
So, finding the median from each group of 7 elements will take $O(n)$ time as it takes a constant time to find median of 7 elements.
Finding the true median out of all the medians calculated above will take atmost $T(n/7)$.
Splitting the array into LESS and GREATER using median as pivot will also take $O(n)$.
Now, the number of elements more than median is at least:

$4(\lceil \frac{1}{2} \lceil \frac{n}{7} \rceil \rceil - 2) \geq \frac{2n}{7} - 8$
Similarly, the number of elements less than median can be calculated.
Hence, the recurrence relation is:
$T(n) \leq T(n/7) + T(5n/7 + 8) + O(n)$
Solving the above recurrence by unrolling, we get,
$T(n) \leq 6cn/7 + 9c + an$
For very large values of n and appropriate c, $T(n) \leq cn$
Hence, the run-time complexity for median finding algorithm when groups of 7 are used is $T(n) = O(n)$.

(c) Give an $O(logn)$ time algorithm to find the median of two sorted arrays of n numbers each.
Solution:
The following algorithm calculates the median of two sorted arrays of n numbers each by finding the median of each array and then comparing them.
Let the two arrays be A and B, each of size n.
Algorithm:

i. If size of both arrays is 2, then median can be calculated using following formula:
return $median = (max(A[1], B[1]) + min(A[2], B[2]))/2$
ii. Else set m_1 as median of A and m_2 as median of B
iii. If $m_1 > m_2$, then recurse on $A[1 \cdots \lfloor n/2 \rfloor]$ and $B[\lfloor n/2 \rfloor \cdots n-1]$
iv. If $m_2 > m_1$, then recurse on $B[1 \cdots \lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor \cdots n-1]$
v. If $m_1 = m_2$, then return $median = m_1$

Complexity:
In the above algorithm, the size of arrays is reduced to half at each successive step. The total number of times it would execute be $logn$ for an input of size n.
Each execution takes $O(1)$ time to complete.
So, it's run-time complexity would be $O(logn)$.
Analysis:
Here, the recurrence relation is:
$T(n) = T(n/2) + O(1)$
Applying master theorem, we get, $a = 1$
$b = 2$
$k = 0$
Here, $a = b^k$. So, case 2 of master theorem is applicable.
i.e., $T(n) = O(n^k logn)$
Substituting the value of k, we get,
$T(n) = O(logn)$
Therefore, it's run-time complexity is $O(logn)$.

3. **QuickSort.**
Consider a version of QuickSort where we use the arithmetic mean (average) of the elements as the pivot.

(a) What is the time complexity in the worst case for this version of QuickSort? Explain.
Solution:
The worst case complexity of QuickSort when using mean of elements as pivot is $\Omega(n^2)$.
The worst case occurs for an exponentially increasing input list.
For example: $1, n^2, n^3, \cdots, n^n$.
In the above scenario, the mean of the set is (n^{n-1}) which is the worst pivot possible as it will divide the elements into two lists of size $(n-2)$ and 1.
Also, calculating arithmetic mean of n elements takes $O(n)$.
The recurrence relation would be as follows:
$T(n) = T(n-2) + cn$
Solving the above recurrence by unrolling, we get,
$T(n) = T(n-4) + c(n-2) + cn$
$T(n) = T(n-6) + c(n-4) + c(n-2) + cn$
.
.
$T(n) = T(n-k) + c(n-k) + \cdots + c(n-2) + cn$
For base case, k = n
Thus, we get,
$T(n) = T(0) + 0 + 2 + 4 + \cdots + n$
$T(n) = n(n+1)$
$T(n) = cn^2$
Hence, the worst-case complexity as $O(n^2)$.

(b) For independent samples drawn from many natural distributions such as the Gaussian the sample mean tends to be close to (i.e. a good estimator of) the median. Let us assume that the mean of the elements in the array (or any subarray) always partitions the elements into two equal-sized sets. Explain briefly why this version of QuickSort will still take $O(nlogn)$ time on such inputs.
Solution:
In the given scenario, the mean of the elements in the array always partitions the elements into two equal-sized sets.
Hence, the size of LESS(left) and GREATER(right) arrays will be the same.
The total number of partitions would then be $logn$ for an input of size n.
Each partition requires n comparisons and at most $n/2$ swaps.
So, it's run-time complexity would be $O(nlogn)$.

Analysis:
Here, the recurrence relation is:
$T(n) = 2T(n/2) + cn$
Applying master theorem, we get,
$a = 2$
$b = 2$
$k = 1$
Here, $a = b^k$. So, case 2 of master theorem is applicable.
i.e., $T(n) = O(n^k logn)$
Substituting the value of k, we get,
$T(n) = O(nlogn)$
Therefore, it's run-time complexity is $O(nlogn)$.

4. **Improving the pivot.**
Let A be an (unsorted) array with n distinct numbers, where n is odd. A pivot is considered to be good if there are at least $n/10$ numbers bigger than it and at least $n/10$ numbers smaller than it; in other words it lies in the middle 8/10 fractional segment in sorted order.

(a) What is the probability that pivot p_1, picked uniformly at random from A, is good?
Solution: The pivot is said to be good if it lies in the middle (8/10) segment in sorted order.
For a very large value of n, the fact that n is odd can be neglected.
Hence, the probability of any randomly chosen pivot to be good is 0.8.

(b) Suppose now you pick uniformly at random (in parallel) a sample of three numbers from A and set p_3 to be the median of these numbers. What is the probability that p_3 is good?
Solution: The pivot is said to be good if it lies in the middle (8/10) segment in sorted order, i.e., it splits the elements in 10:90 or 90:10.
Now, we take 3 elements randomly and consider median of those elements as pivot, say p_3.
p_3 is a bad pivot if it lies in smallest $n/10$ segment of elements or largest $n/10$ segment of elements.
In either case, the elements' split ratio is worse than 90:10 or 10:90.
p_3 is in smallest $n/10$ segment if any two numbers lie in smallest segment or all three numbers lie in smallest segment.
Probability of choosing exactly 2 out of 3 elements from smallest segment is $3 * 0.1^2 * 0.9$.
Probability of choosing all 3 elements from smallest segment is 0.1^3
Therefore, Total probability of p_3 to be in smallest $n/10$ segment is $3 * 0.1^2 * 0.9 + 0.1^3 = 0.028$.
Similarly, the probability of p_3 to be in largest $n/10$ segment is $3 * 0.1^2 * 0.9 + 0.1^3 = 0.028$.
So, the probability of p_3 being bad is $0.028 + 0.028 = 0.056$
Hence, the probability of p_3 being good is $1 - 0.056 = 0.944$ i.e., 94.4%

(c) Suppose now you pick uniformly at random (in parallel) a sample of four numbers from A and set p_4 to be the lower median of these numbers. What is the probability that p_4 is good?
Solution: The pivot is said to be good if it lies in the middle (8/10) segment in sorted order, i.e., it splits the elements in 10:90 or 90:10.
Now, we take 4 elements randomly and consider lower median of those elements as pivot, say p_4.
p_4 is a bad pivot if it lies in smallest $n/10$ segment of elements or largest $n/10$

segment of elements.
In either case, the elements' split ratio is worse than 90:10 or 10:90.
p_4 is in smallest $n/10$ segment if at least two numbers lie in smallest segment.
Probability of choosing exactly 2 out of 4 elements from smallest segment is $6 * 0.1^2 * 0.9^2$.
Probability of choosing exactly 3 out of 4 elements from smallest segment is $4 * 0.1^3 * 0.9$.
Probability of choosing all 4 elements from smallest segment is 0.1^4
Therefore, Total probability of p_4 to be in smallest $n/10$ segment is $6*0.1^2*0.9^2 + 4 * 0.1^3 * 0.9 + 0.1^4 = 0.0486 + 0.0036 + 0.0001 = 0.0523$
Similarly, the probability of p_4 to be in largest $n/10$ segment is $6 * 0.1^2 * 0.9^2 + 4 * 0.1^3 * 0.9 + 0.1^4 = 0.0486 + 0.0036 + 0.0001 = 0.0523$
So, the probability of p_4 being bad is $0.0523 + 0.0523 = 0.1046$
Hence, the probability of p_4 being good is $1 - 0.1046 = 0.8954$ i.e., 89.54%

(d) What can you conclude about the size of the sample from which to pick the median? Will a sample of 5 give a higher probability of picking a good pivot compared to a sample of 3?
Solution: The probability of choosing a good pivot is high if sample size is odd or an upper median is chosen from the samples of even size. A sample of 5 will give a higher probability of picking a good pivot compared to a sample of 3.
Explanation:
The pivot is said to be good if it lies in the middle (8/10) segment in sorted order, i.e., it splits the elements in 10:90 or 90:10.
Now, we take 5 elements randomly and consider median of those elements as pivot, say p_5.
p_5 is a bad pivot if it lies in smallest $n/10$ segment of elements or largest $n/10$ segment of elements.
In either case, the elements' split ratio is worse than 90:10 or 10:90.
p_5 is in smallest $n/10$ segment if atleast three numbers lie in smallest segment.
Probability of choosing exactly 3 out of 5 elements from smallest segment is $10 * 0.1^3 * 0.9^2$.
Probability of choosing exactly 4 out of 5 elements from smallest segment is $5 * 0.1^4 * 0.9$.
Probability of choosing all 5 elements from smallest segment is 0.1^5
Therefore, Total probability of p_5 to be in smallest $n/10$ segment is $10 * 0.1^3 * 0.9^2 + 5 * 0.1^4 * 0.9 + 0.1^5 = 0.0081 + 0.00045 + 0.00001 = 0.00856$
Similarly, the probability of p_5 to be in largest $n/10$ segment is $10 * 0.1^3 * 0.9^2 + 5 * 0.1^4 * 0.9 + 0.1^5 = 0.0081 + 0.00045 + 0.00001 = 0.00856$
So, the probability of p_5 being bad is $0.00856 + 0.00856 = 0.01712$
Hence, the probability of p_4 being good is $1 - 0.01712 = 0.98288$ i.e., 98.288%

5. Given a universe U of size N^{10}, subset $S \subseteq U$ of size N, and a family of universal hash functions H with $h : U \to \{0, ..., M-1\}$ chosen uniformly at random from H.
Pre-computation:
We know that Boole's inequality for a countable set of events $A1, A2, A3, \cdots$ is:
$\mathbb{P}\left(\bigcup_i A_i\right) \leq \sum_i \mathbb{P}(A_i)$.
Now, we will prove Markovs Inequality (for finite probability spaces)
Let $x_1, \cdots, x_r$ be the possible values of x. Then,
$E[x] = \sum_{i=1}^{r} x_i Pr[x = x_i]$
$= \sum_{x_i<t} x_i Pr[x = x_i] + \sum_{x_x \geq t} x_i Pr[x = x_i]$
$= \sum_{x_x \geq t} x_i Pr[x = x_i] (because\ x \geq 0)$
$= \sum_{x_x \geq t} tPr[x = x_i]$
$= tPr[x \geq t]$
Dividing by t,
$\frac{E[x]}{t} \geq Pr[x \geq t]$
We will use Boole's inequality and Markov's inequality to prove our solutions below.

   (a) Suppose that $M = N^3$. Use the union bound to show that the probability that h is collision-free with respect to S is $1o(1)$ as N tends to infinity.
   Proof: Now, we prove the above claim concerning the probability of a collision for universal hash functions and a hashed space of size M = N^3.
   Now, let $x = the\ number\ of\ collisions.$
   $E[x] = \sum_{allpairs(k,l)} Pr[h(k) = h(l)]$
   $= \sum_{allpairs(k,l)} 1/M$
   $= \binom{N}{3}/N^3 = \frac{1}{6}$
   $= o(1)$
   Since $Pr[no - collision] = 1 - Pr[exists\ a\ collision]$
   Therefore, $Pr[collision - free] = 1 - o(1)$

   (b) Suppose that $M = N^2$. Use the union bound to show that the probability that h is collision-free with respect to S is at least $1 \div 2$ as N tends to infinity.
   Proof: Now, we prove the above claim concerning the probability of a collision for universal hash functions and a hashed space of size M = N^2.
   Now, let $x = the\ number\ of\ collisions.$

$$E[x] = \sum_{allpairs(k,l)} Pr[h(k) = h(l)]$$
$$= \sum_{allpairs(k,l)} 1/M$$
$$= \binom{N}{2}/N^2 = \frac{n-1}{2n} < 1/2$$

So we know that the expected number of collisions is 1/2. We want to say something about the probability of a collision. Applying Markovs inequality:

$$Pr[\exists\ collision] = Pr[x \geq 1] \leq \frac{E[x]}{1} < 1/2$$

(c) Suppose that $M = N^9$. Is the following statement TRUE or FALSE: For every h there exists a set $S' \subset U$ of size N such that there exists a collision in S' with respect to h. Explain why this does not contradict our expectation that universal hash families are collision-free.

Solution: We prove that the above claim concerning the probability of a collision for universal hash functions and a hashed space of size M = N^9.

Now, let $x = the\ number\ of\ collisions.$

$$E[x] = \sum_{allpairs(k,l)} Pr[h(k) = h(l)]$$
$$= \sum_{allpairs(k,l)} 1/M$$
$$= \binom{N}{9}/N^9 = \frac{1}{362880}$$

Since $Pr[no - collision] = 1 - Pr[exists\ a\ collision]$

Therefore, $Pr[collision - free] = 1 - \frac{1}{362880} \approx 1$

Which proves that there exists a $S' \subset U$ of size N with collision but it can be neglected for given universal hash family.

Hence, the given statement is false.

This does not contradict our expectation that universal hash families are collision-free because the size of universal hash family in consideration is N^9 and probability of collision is very small($2.77 * 10^{-6}$), it can be neglected in this case.

(d) Describe a use of universal hash families and motivate the need for the collision-free property.

Answer: Universal hashing has numerous uses in computer science, one of which is Hash Tables where randomly selecting a hash function from a family of hash functions with a certain mathematical property guarantees a low number of collisions in expectation, even if the data is chosen by an adversary.

It is for this reason that randomness is ubiquitous in cryptography. In cryptographic applications, pseudo-random numbers cannot be used, since the adversary can predict them, making the algorithm effectively deterministic. Therefore, either a source of truly random numbers or a cryptographically secure pseudo-random number generator is required.

Motivation: Assume we want to map keys from some universe U into m bins (labelled $[m] = \{0, \ldots, m-1\}$). The algorithm will have to handle some data set $S \subseteq U$ of $|S| = n$ keys, which is not known in advance. Usually, the goal of hashing is to obtain a low number of collisions (keys from S that land in the same bin). A deterministic hash function cannot offer any guarantee in an adversarial setting if the size of U is greater than $m \cdot n$, since the adversary may choose S to be precisely the preimage of a bin. This means that all data keys land in the same bin, making hashing useless. Furthermore, a deterministic hash function does not allow for rehashing: sometimes the input data turns out to be bad for the hash function (e.g. there are too many collisions), so one would like to change the hash function. The solution to these problems is to pick a function randomly from a family of hash functions. A family of functions
$H = \{h : U \to [m]\}$ is called a universal family if,

$$\forall x, y \in U,\ x \neq y : \quad \Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}.$$

In other words, any two keys of the universe collide with probability at most $1/m$ when the hash function h is drawn randomly from H. This is exactly the probability of collision we would expect if the hash function assigned truly random hash codes to every key.

2 Part B (programming)

Group Hackerrank Name: akashsingh245

1. Surekha Jadhwani : surekha@ccs.neu.edu / jadhwani.s@husky.neu.edu
2. Akash Singh : singhaka@ccs.neu.edu / singh.aka@husky.neu.edu
3. Ayush K. Singh : singhay@ccs.neu.edu / singh.ay@husky.neu.edu