CSC265 Homework 3

Question 1 (Mark: 10/12)

Part a. (Mark: 6/8)
We begin by initializing arrays A and B with all entries set to zero. We utilize the attributes $left$ and $right$ to maintain specific properties of the matrices. For all i where $1 \leq i < left$, we ensure that A[i,1] + A[i,2] = $1/n$. For i within the range $left \leq i < right$, we have A[i,1] $< 1/n$ and A[i,2] = 0. Finally, for all i such that $right \leq i \leq n$, we have A[i,1] = A[i,2] = 0. The values of $left$ and $right$ are both initialized to 1.

Our algorithm first process elements that are equal to $1/n$, followed by elements that are less than $1/n$, and finally, we handle elements that are greater than $1/n$.

- Step 1: We iterate through the elements $p_1, \ldots, p_n$ for i from 1 to n. If $p_i = 1/n$, we set A[$left$,1] = $1/n$, and B[$left$,1] = i. The values of A[$left$,2] and B[$left$,2] remain zero. After this, we increment both $left$ and $right$ by 1.

- Step 2: We iterate through the elements $p_1, \ldots, p_n$ for i from 1 to n again. If $p_i < 1/n$, we set A[$left$,1] = p_i and B[$left$,1] = i. The values of A[$left$,2] and B[$left$,2] remain zero. After this, we increment $right$ by 1.

- Step 3: We iterate through the elements $p_1, \ldots, p_n$ for i from 1 to n. If $p_i > 1/n$, we set num to be p_i and repeat the following steps until $num = 0$:

- Case 1: If $left = right$ and $num \geq 1/n$, we set A[$left$, 1] to $1/n$ and B[$left$, 1] to i. The values of A[$left$, 2] and B[$left$, 2] remain as zero. After this, we increment both $left$ and $right$ by 1 and decrease num by $1/n$.
- Case 2: If $left = right$ and $num < 1/n$, we set A[$left$, 1] to num and B[$left$, 1] to i. The values of A[$left$, 2] and B[$left$, 2] remain as zero. After this, we increment $right$ by 1 and set num to zero.
- Case 3: If $left \neq right$ and $\frac{1}{n}$−A[$left$, 1] $\leq num$, we set A[$left$, 2]= $\frac{1}{n}$−A[$left$,1] and B[$left$, 2] = i. After this, we decrease num by $\frac{1}{n}$−A[$left$,1] and increment $left$ by 1.
- Case 4: If $left \neq right$ and $\frac{1}{n}$−A[$left$, 1] $> num$, we set A[$right$, 1]= num and B[$right$, 1] = i. The values of A[$right$, 2] and B[$right$, 2] remain as zero. After this, we set num to zero and increment $right$ by 1.

- Step 4: Return matrix A and B

Proof of Correctness:

We first prove a few invariants:

1. $left \leq right$. We establish this through induction. Initially, we set $left = right = 1$. Assuming $left \leq right$, we can analyze how each step modifies $left$ and $right$. In step 1, step 2, and cases 1, 2, and 4 of step 3, we either increment both $left$ and $right$ by 1 or only increment $right$ by 1, thus maintaining the invariant. In case 3 of step 3, when $left \neq right$, we have $left < right$. By increasing $left$ by 1, we still have $left \leq right$ since both are integers. Hence, the invariant remains preserved.

(-2 Marks, i was intended to denote the number of elements that had been processed)

2. The properties of the interval $1 \leq i < left$ hold. Initially, the interval is empty, thus the property is satisfied. In step 1 and cases 1 and 3 of step 3, we increment the value of $left$ by 1. In those cases, based on the construction, we add $left$ such that $A[left, 1] + A[left, 2] = 1/n$ to the interval, thus maintaining the property.

3. The properties of the interval $left \leq i < right$ hold. Initially, with $left = right = 1$, the interval is empty, and as both variables increase together in step 1, the interval remains empty, ensuring the property's satisfaction. The same reasoning applies to case 1 of step 3, where the interval remains empty. Based on our implementation, when $left$ increases, the property holds because no elements in the new interval $left' \leq i < right$ are modified. Therefore, our focus should be on situations when $right$ increases. In step 2 and cases 2 and 4 of step 3, we set $A[right, 1]$ to a value less than $1/n$ and $A[right, 2]$ to zero. As a result, this invariant remains preserved.

4. The properties of the interval $right \leq i \leq n$ hold. Initially, this is guaranteed by our initialization. According to our implementation, elements within this interval are not accessed during each iteration, thus preserving this invariant.

5. None of the elements $p_1, \ldots, p_n$ are modified. Consequently, each element is processed precisely once, either in step 1, step 2, or step 3.

6. The sum $\sum A$ equals the sum of the processed p_i. Initially, this holds true because each entry of A is zero, and none of the p_i have been processed. Now, let's consider when we process p_i. If we are in either step 1 or step 2, we set A[*right*, 1] to p_i, which is originally zero according to the fourth invariant. If we process p_i in step 3, we utilize a variable *num* $= p_i$. Based on our implementation, we split *num* into parts and add it to entries of A, which are initially zero. In both cases, $\sum A$ increases by p_i, thus maintaining the invariant.

Lemma. The algorithm terminates with $left = right = n + 1$.

Proof. For the case when $p_1 = \cdots = p_n = 1/n$, the result is trivial since we process all of them in step 1, increasing both $left$ and $right$ by 1 each time. Therefore terminates when $left = right = n + 1$.

Suppose not all elements are equal to $1/n$; then, there must exist some elements less than $1/n$ and some greater than $1/n$. In step 1 and step 2, at most $n-1$ elements are processed, denoting this number by k. As $right$ increases by 1 each time, we have $right = k + 1$, and there are $n - k$ numbers that are greater than $1/n$.

We aim to demonstrate that when there are r numbers greater than $1/n$ that remain unprocessed, then $right \leq n + 1 - r$. This holds initially when $r = n - k$. When processing p_i:

- if it does not enter case 1 or case 2, then $right$ is at most increased by 1, while r decreases by 1. In this scenario, the inequality still holds.
- If it enters case 1 or case 2, then it ends with case 1 or case 2, by our implementation. We have $left = right$ or $left = right - 1$ after processing p_i. By invariant 2,3 and 4, every pair to the left of $right - 1$ is full, and $right - 1$ may be full or not full, while all pairs to the right are empty. According to invariant 6, we still have $r - 1$ elements that have not been added to A. Since each of them is greater than $1/n$, there must be at least $r - 1$ empty pairs in A. Consequently, we have $right \leq n + 1 - (r - 1)$.

Through induction, this result remains true.

When the algorithm terminates, we have $r = 0$, therefore $left \leq right \leq n + 1$. By invariant 6, $\sum A = 1$, therefore each pair we have $A[i, 1] + A[i, 2] = 1/n$. By invariant 3 and 4, both interval $[left, right)$ and $[right, n]$ are empty, therefore we have $left = right = n + 1$ as desired. □

The first property of matrices A and B follows directly from our algorithm's implementation. The third property follows from the lemma and the second invariant. Now, we will demonstrate the second and fourth properties:

Whenever we assign a value to an entry in Matrix A, the corresponding position in Matrix B receives the index $i \in [n]$ simultaneously. Since each entry is initially set to zero, if some entry of B remains zero, then the corresponding entry in A is unassigned (i.e. zero). Therefore, the second property holds.

For any arbitrary $i \in [n]$, if $p_i \leq 1/n$, it is processed in either step 1 or step 2. Since each p_i is processed exactly once in the algorithm, there is only one entry in Matrix B with the value i, where the corresponding entry in Matrix A has the value p_i. Otherwise, it is processed in step 3, then we utilize a variable *num*, which initially equals p_i. We subsequently break *num* into parts, each are stored in different entries in Matrix A. The corresponding entry in Matrix B has the index i, and by summing the same entries in Matrix A, we obtain p_i. Thus, the fourth property holds.

Finally, we have proved the correctness of the algorithm.

Proof of Time Complexity:

In both step 1 and step 2, we first iterate over $i \in [n]$, each followed by a comparison. If the comparison is true, we execute a constant number of variable assignments. Therefore, the complexity of both step 1 and step 2 is $O(n)$.

In step 3, we break it into two parts and analyze them separately. The basic part is an iteration over $i \in [n]$, followed by a comparison. This part takes $O(n)$.

For each case in step 3, there are a constant number of comparisons, variable assignments, and variable calculations. Therefore, each case takes constant time. In each case, we either increase $left$ or $right$ or both by 1. Given their initialization at 1 and the algorithm terminates when $left = right = n + 1$, at most a total of $2n$ cases can be executed. Therefore, this part of step 3 contributes to a time complexity of $O(2n) = O(n)$.

Summing all the complexities above, we have demonstrated that the time complexity of our algorithm is $O(n+n+n+2n) = O(n)$, which is linear as desired.

Part b. (Mark: 6/6)

```
Sample():
    r = RandInt(n)
    p = A[r,1]*n
    b = Bernoulli(p)
    if b == 1:
        return B[r,1]
    else:
        return B[r,2]
```

Proof of Correctness:

For arbitrary $i \in [n]$, we begin by determining the probability that 'Sample()' returns B$[i, 1]$. By our code above, we know that this happens if and only if $r = i$ and $b = 1$. By property of 'RandInd' function, it have a possibility of $1/n$ of returning i. By property of 'Bernoulli', it has a probability of $p = A[i, 1] \times n$ of returning 1. Therefore, the probability of 'Sample()' returning B$[i, 1]$ is $\dfrac{A[i, 1] \times n}{n} = A[i, 1]$.

Similarly, 'Sample()' returns B$[i, 2]$ if and only if $r = i$ and $b = 0$, and the possibility of returning it is

$$\frac{1 - (A[i, 1] \times n)}{n} = \frac{1}{n} - \frac{A[i, 1] \times n}{n} = \frac{1}{n} - A[i, 1] = A[i, 2]$$

Hence, the probability of 'Sample()' returning $I \in [n]$ is the sum of the probabilities of returning $B[j, k]$ where $B[j, k] = I$. Therefore, we have $\mathbb{P}(I = i) = \sum_{j:B[j,1]=i} A[i, 1] + \sum_{j:B[j,2]=i} A[i, 2] = p_i$, as desired, by the fourth property of A and B.

Proof of Worst-case Complexity:

The second line of the code involves invoking the function 'RandInt()' and assigning a variable, which operates in constant time. The third line entails direct matrix access, multiplication of two numbers, and variable assignment, all of which also occur in constant time. The fourth line calls the function 'Bernoulli()' and assigns a variable, which also takes constant time. Lines 5 to 8 involve a comparison and direct matrix access, both of which occur in constant time.

Summing it all together, 'Sample()' runs in worst-case constant time as desired.

Question 2 (Mark: 12/12)

Part a. (Mark: 4/4)
Let $i \in \{0, ..., n-1\}$ be arbitrary. By the given code, we have

$$\tilde{a_i} = \sum_{j:h(i)=h(j)} A[j] \implies \tilde{a_i} - A[i] = \sum_{\substack{j \neq i \\ h(i)=h(j)}} A[j]$$

Therefore, considering the fact that every element in A is non-negative, the right hand side of the above equation must be non-negative. Consequently, we can conclude that $\tilde{a_i} \geq A[i]$.

The expectation $\mathbb{E}(\tilde{a_i} - A[i])$ is equivalent to the sum of the expectations of the contributions to $\tilde{a_i}$ from all A[j] where $j \neq i$. By the definition of expectation, for each j it can be expressed as $\mathbb{P}\big(h(j) = h(i)\big)A[j]$.

Since $h \in \mathcal{H}$ is sampled uniformly from a universal family of hash functions, we have $\mathbb{P}(h(i) = h(j)) \leq 1/k$ for all $i \neq j$. Since $A[j] \geq 0$ for all j, we have

$$\mathbb{E}\big(\tilde{a_i} - A[i]\big) = \sum_{j \neq i} \mathbb{P}\big(h(i) = h(j)\big)A[j] \leq \sum_{j \neq i} \frac{1}{k}A[j] = \frac{1}{k}\sum_{j \neq i} A[j]$$

Let $\varepsilon(k) = \dfrac{1}{k}$, which is a monotonically decreasing function of k. We have $\mathbb{E}\big(\tilde{a_i} - A[i]\big) \leq \varepsilon(k)\sum_{j \neq i} A[j]$ as desired.

Part b. (Mark: 8/8)
Given $\delta \in (0, 1)$, let $t = \lceil \log_2(1/\delta) \rceil$. Therefore $1/\delta > 1$, we have $\log_2(1/\delta) > 0$, and thus $t \geq 1$.

We modify Compress and Query as follows:

```
ModifiedCompress(A):
    Initialize $H[0 \ldots t-1]$ be array of hash functions.
    Initialize an array $S[0 \ldots t-1]$
    for $i = 0 \ldots t-1$:
        Let $(S', h)$ be the return value of Compress(A)
        Set H[i] = $h$
        Set S[i] = $S'$
    return $(S, H)$

ModifiedQuery(S,H,i)
    Let $min = \infty$
    for $j = 0 \ldots t-1$:
        Let result = Query($S[j], H[j], i$)
        if $min >$ result:
            Set $min$ = result
    return $min$
```

That is, for the ModifiedCompress operation, we repeat the original Compress function t times and store their output in S and H. In the ModifiedQuery operation, we return the minimum value of Query among all t different compressed arrays.

Proof of Correctness:

The ModifiedCompress returns an array of length t where each entry stores k integers, therefore contains $O(tk) = O(\lceil \log_2(1/\delta) \rceil k) = O(k \log(1/\delta))$ integers as desired. Since the ModifiedQuery operation returns the minimum value $\tilde{a_i}$ among the t return values of the Query operation, we must have $\tilde{a_i} \geq A[i]$ as established in part a.

Consequently, the probability of $\tilde{a_i} - A[i]$ being greater than some value z is equivalent to the probability that every return value r of the Query operation satisfies $r - A[i] \geq z$.

Suppose a Query call returns the value r, then we have:

$$\mathbb{P}\left(r - A[i] \geq 2\varepsilon(k)\sum_{j\neq i} A[j]\right) \leq \mathbb{P}\Big(r - A[i] \geq 2\mathbb{E}\big(r - A[i]\big)\Big) \leq \frac{1}{2}$$

The first inequality follows from part a, and it utilizes the fact from probability theory: $A \subseteq B \implies \mathbb{P}(A) \leq \mathbb{P}(B)$. The second inequality follows from Markov's inequality.

Since each $h \in \mathcal{H}$ is uniformly randomly sampled, each Query call is independent. Therefore, we can apply the formula for independent events:

$$\mathbb{P}\left(\tilde{a_i} - A[i] \geq 2\varepsilon(k)\sum_{j\neq i} A[j]\right) = \prod_{n=0}^{t-1} \mathbb{P}\left(r - A[i] \geq 2\varepsilon(k)\sum_{j\neq i} A[j]\right) \leq \frac{1}{2^t} = \frac{1}{2^{\lceil \log_2(1/\delta)\rceil}} \leq \frac{1}{2^{\log_2(1/\delta)}} = \delta$$

which completes the proof.

Proof of Time Complexity:

Lemma. The original Compress takes $O(n)$ and Query takes $O(1)$.

Proof. For the Compress operation, sampling $h \in \mathcal{H}$ takes constant time, and initializing S of size k takes $O(k)$ time. Then, it iterates from $i = 0$ to $n - 1$, with each iteration involving a constant number of array accesses, h calculations, and additions, all of which take constant time. Therefore, the for-loop takes $O(n)$. Given that $k \leq n$, summing up all of the above, the overall time complexity is $O(n)$.

As for the Query operation, it only involves a single array access and one h calculation, both of which take constant time, making the Query operation have a time complexity of $O(1)$. $\square$

Theorem. The ModifiedCompress operation has a time complexity of $O(n \log(1/\delta))$, and the ModifiedQuery operation has a time complexity of $O(\log(1/\delta))$.

Proof. For the ModifiedCompress operation, we first initialize arrays H and S, both with size t, which gives a time complexity of $O(t)$. Then the for-loop iterates for i from 0 to $t-1$, and each iteration includes a call to the original Compress function along with two array accesses and two variable assignments, resulting in a total of $O(n)$ for each iteration. Therefore, the for-loop has a time complexity of $O(nt)$. Summing all the above, the ModifiedCompress operation has a time complexity of $O(t + nt) = O(\lceil \log_2(1/\delta)\rceil(n + 1)) = O(n \log(1/\delta))$ as desired.

For the ModifiedQuery operation, we first initialize a variable, which takes constant time. Then the for-loop iterates for j from 0 to $t - 1$, with each iteration containing at most a call to Query, a comparison, two array accesses, and two variable assignments, all of which take constant time. Therefore, the for-loop has a time complexity of $O(t)$. Summing all the above, the ModifiedQuery operation has a time complexity of $O(1 + t) = O(1 + \lceil \log(1/\delta)\rceil) = O(\log(1/\delta)$ as desired. $\square$