Problem 1

(a)

Algorithm

In **Init**, together with the original n-element array **arr**, we initialize two other arrays, called **from** and **to** (leave their entries untouched). In addition, a single integer **num** is initialized to 0.

Set(i, x) first invokes **Get**(i). If the result not "empty", it sets **arr**[i] := x. Otherwise, it sets the following

- **from**[i] := **num**
- **to**[**num**] := i
- **num** := **num** + 1
- **arr**[i] := x

Get(i) checks if *both* **from**[i] < **num** *and* **to**[**from**[i]] == i, then returns **arr**[i], otherwise returns "empty".

Analysis

The integer **num** indicates total number of entries that the original array is being used. **from** redirects the query entry to **to**, which stores the 'actual' entry being set in the first time. Notice that the element in **to** array indexing from 1 to **num** is correct and uniquely pointing to entries in **arr**, which in and only in the correct case, will map back the query i, namely **to**[**from**[i]] = i. This will not be disturbed by any garbage in the system.

Initializing the original **arr**, **from**, **to** and **num**, takes $O(1)$ each, therefore $O(1)$ in total.

Get and **Set** only uses constant number array lookup and arithmetic comparison, and thus is $O(1)$ as well.

Problem 2

To implement Dijkstra's algorithm, we need to store the minimum known distance in the graph that is not being marked 'checked'. In the vanilla way of implementing it, we do need to use $O(log\, log\, nC)$ space, for all possible distances.

However, notice that the distance of any node to any other node is bounded by C (Triangle Inequality). This means that we can construct a 'ring' structure to capture the distance of all other nodes starting from the current shortest distance node.

Specifically, we construct *two* vEB priority queue. When computing the distance d to a node, we store $d\, mod\, C$ (key for queue) along with $\lfloor d/C \rfloor$ (extra field). Denote the distance of current fingering node as d^*, for the distances to other nodes, if the newly computed shortest distance d satisfying $d\, mod\, C > d^*\, mod\, C$, we put $d\, mod\, C$ in the *second* vEB priority queue; otherwise, we put it in the *first* vEB queue.

Now in order to iterate to the next node, we look into the *second* vEB queue and invoke *find-min*, if the return is invalid (meaning the second vEB queue is empty), we invoke *find-min* in the first vEB queue. The real distance can be retrieved by using both $d\, mod\, C$ along with $\lfloor d/C \rfloor$ being stored.

The reason this two vEB queue works is that they effectively construct a 'ring' structure. The starting point (distance to the current node) sits in the middle to first queue. Then, the next known minimum distance to all other nodes is immediately after the current distance after taking $mod C$ operation and will be put in the second queue. If it rotates back, it will be the first element in the first queue. Because the next minimum distance is bounded to be at most C greater than current distance, two queues each with size C is enough.

Therefore, we actually only need $O(log\, log\, C)$ for vEB queue operation in Dijkstra's algorithm.

(*In a hindsight, we can achieve similar conclusions with one vEB tree from ideas in problem 3. Same* $(d\, mod\, C)$ *trick but The next-find-min operation can be replaced by successor, and the rotation happens when successor hits an invalid answer and we invoke getting minimum in tree.*)

Problem 3

To construct a data structure to support queries for existence of element, its predecessor and successor, we maintain an 'array' with entry 1 indicating existence of element and 0 otherwise. We build vEB version of it to reach $O(log\, log\, u)$ operation time.

For integer x in range $\{0, 1, 2, ..., u-1\}$, define $x.high = \lfloor \frac{x}{\sqrt{u}} \rfloor$, and $x.low = x \bmod \sqrt{u}$. Note $x = x.high \times \sqrt{u} + x.low$. Split Q into $\sqrt{u}$ clusters, each having $\sqrt{u}$ entries. Construct an array with size $\sqrt{u}$ whose entries store the **summary** (if there is at least one element, 1 indicating existence, 0 otherwise) in each cluster.

Find(x, Q)

(1). Check if $Q.summary(x.high)$ is 1.
(2). If (1) holds, check if $Q.cluster[x.high][x.low]$ is 1.
Report existence if both (1) and (2) holds, non-existence otherwise.

Predecessor(x, Q)

(If Q is in the top or bottom level, namely no Q.summary or Q.cluster, it finds predecessor in the current level and abort with $-\infty$ if operation invalid.)
$i := x.high$
$j :=$ the largest index with value 1, that is smaller than $x.low$.
If j is not valid:
  $i :=$ **Predecessor**$(x.high, Q.summary)$
  $j :=$ **Predecessor**$(+\infty, Q.cluster[i])$
return $i \times \sqrt{u} + j$.

Successor(x, Q)

(If Q is in the top or bottom level, namely no Q.summary or Q.cluster, it finds successor in the current level and abort with $+\infty$ if operation invalid.)
$i := x.high$
$j :=$ the smallest index with value 1, that is larger than $x.low$.
If j is not valid:
  $i :=$ **Successor**$(x.high, Q.summary)$
  $j :=$ **Successor**$(-\infty, Q.cluster[i])$
return $i \times \sqrt{u} + j$.

Analysis

Notice that the number of levels of *summary-cluster* is $O(log\ log\ n)$. This is because of the following. Notice that every one level up, the summary aggregates $\sqrt{n}$ nodes. To get the number of levels, we essentially need to get " how many 1/2 are there in $((u^{1/2})^{1/2})^{\cdots}$ to reach 1. Let $v = log(u)$, then it becomes $e^{\frac{1}{2}\frac{1}{2}\cdots v}$ and we know here the number of 1/2 will be $log\ v = log\ log\ u$.

In the **find** step, the level of tree determines the worst-case query time, which is $O(log\ log\ u)$.

In **predecessor** and **successor** step, we use recursion analysis to calculate the run time $T(u)$. Because every time it breaks the cluster into size $\sqrt{u}$ and searching in the *summary* and *cluster* both takes $O(\sqrt{u})$, we will have

$$T(u) = T(\sqrt{u}) + O(1).$$

Now we replace $v = log(u)$, then $T(v) = T(\frac{1}{2}v) + O(1)$, and we get $T(v) = O(log(v))$, therefore $T(u) = O(log(T(v))) = O(log\ log\ u)$. (This is similar to the analysis for **find**).

Find-min can be implemented by invoking **Successor**$(-\infty, Q)$, which has worst-case $O(log\ log\ n)$ time. **Delete-min** can use **find-min** first, then invoking **Successor**(**Find-min**(x, Q), Q), but along the path of going to the summary, appropriately marking the value to be 0, if the value in the cluster is deleted and the successor is not in the current cluster (In the above **Successor** algorithm, when j is not valid, mark the summary to be 0 as well.). This extra marking takes $O(1)$ time, therefore the worst-case time is still $O(log\ log\ n)$.

Problem 4

(a) First compute the probability for k^{th} insertion being a collision. A collision happens when there are at least one item in the same entry in *both* arrays. This means previously at least two items need to be mapped to this entry, which has probability

$$p_k = 1 - \left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-1} - (k-1)\frac{1}{n^{1.5}}\left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-2}.$$

The expected number of collision can be calculated as follows,

$$\begin{aligned}
\sum_{k=2}^{n} p_k &= \sum_{k=2}^{n}\left[1 - \left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-1} - (k-1)\frac{1}{n^{1.5}}\left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-2}\right] \\
&= n - \sum_{k=2}^{n}\left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-1} - \sum_{k=2}^{n}(k-1)\frac{1}{n^{1.5}}\left(\frac{n^{1.5}-1}{n^{1.5}}\right)^{k-2} \\
&\le n - \sum_{k=2}^{n}\left(1 - \alpha\frac{k-1}{n^{1.5}}\right) - \sum_{k=2}^{n}(k-1)\frac{1}{n^{1.5}}\left(1 - \alpha\frac{k-2}{n^{1.5}}\right) \\
&\text{(from Taylor expasion, where } \alpha = 1+\epsilon \text{ to make ``}\le\text{'' hold)} \\
&= \frac{1}{3}(2n - 3n^2 + n^3)\frac{\alpha}{n^3} \\
&\le \frac{\alpha}{3} = O(1)
\end{aligned}$$

Notice that this computes the expected number of occurrence of *any* item, which is bounded by $O(1)$. Then the (strange "i.e." in the problem) expected number of pairs of item in the same bucket is also $O(1)$ (otherwise the number of collisions happened more often that $O(1)$).

Alternatively, we can use method in part (b) to prove the results here.

(b) Now we need to show the above argument works even when only assuming pair-wise independence, such that we can apply 2-universal hashing family to achieve efficiency.

For any item i, the probability that another item j maps to same index as it, by pairwise independence, is $p_{ij} = \frac{1}{n^{1.5}}$. Therefore, the probability for i to collide with any item, bounded by Boole's inequality

$$\cup_j p_{ij} \le \sum_j p_{ij} = n\frac{1}{n^{1.5}} = \frac{1}{\sqrt{n}}$$

Notice that in bashing, a 'real' collision happens when at least two items are colliding. Therefore the probability for collision is $(\frac{1}{\sqrt{n}})^2 = \frac{1}{n}$. Also notice that

this probability is computed for any item i, therefore the total expected number of collision is $n\frac{1}{n} = O(1)$. Same conclusion from (a) holds.

Now we can create a 2-universal hashing family and randomly select a hashing function from it for each hashing operation. Because 2-universal hashing is efficient, we know bashing can be implemented efficiently.

(c) Because the expected number of collision is bounded by 1 (from (b) Boole's inequality). We know the probability of a hash family being perfect is a constant p, regardless of the size of n. In order to find the perfect hashing, one needs to try out hash families drawn from the 2-universal hashing universe. Then, throughout the probing, the probability of the hash family being *not* perfect is expoentially decayed as $(1-p)^k$, k is the trial number until finding perfect hashing. Therefore, as each probing takes $O(n)$, finding perfect hashing takes $O(n)$.

(d) Similar to part (a), because there is a $3rd$ or $4th$ polynomial term in the p_k computation, we need to make $n^{1.5} = n^{1+1/2}$ in bashing case, to be $n^{1+1/3}$ for trashing and $n^{1+1/4}$ for quashing. *To reach conclusion in (b) and (c), we may need 3-universal and 4-universal hashing family.*

Problem 5

(a) For s items in the table with probability $1/2$ of perfect placement, we know the expected number of trials to find a perfect hashing is 2, from Markov Inequality. This is computed by the following

$$\begin{aligned}\mathbb{E}(\text{number of trials}) &= 1 \times \frac{1}{2} + 2 \times (\frac{1}{2})^2 + 3 \times (\frac{1}{2})^3 + ... + n \times (\frac{1}{2})^n + ... \\ &= \sum_{n=1}^{\infty} \left[n \times \frac{1}{2^n} \right] \\ &= 2\end{aligned}$$

Now for s items, we mark some break points as $\{1, 2, 4, 8, 16, ..., s\}$. Suppose at each break point, we do a rebuild, then the success probability in the next break point will be $1/2$ and the expected number of rebuild is s for the next marking point.

Because we are computing the upper bound, we now artificially introduce a rebuild at each rebuild. This bounds the result in the following sense. If it does not require a rebuild but we enforce one, the 'real' rebuild will have a wider range of support than our enforced one (because it happens later). Therefore it suffices to compute the expected rebuild complexity for these marking point, as follows

$$\begin{aligned}\mathbb{E}(\text{cost of rebuild}) &\leq \mathbb{E}(\text{number of rebuild counting until each break point}) \\ &\quad \times \mathbb{E}(\text{rebuild cost for items until break point}) \\ &= 2 \times \big(1 + 2 + 4 + 8 + 16 + ... + s\big) \\ &\leq 2 \times 3s \\ &= O(s)\end{aligned}$$

(b) From the scheme of constructing 2-level perfect hashing, we know $\mathbb{E}\left[\sum s_i^2\right] = O(n)$, this is because

$$\begin{aligned}\mathbb{E}\left[\sum s_k^2\right] &= O\left[\sum_k \sum_{\text{number of pairs of } i,j \text{ collides in k}} 1\right] \quad \text{(Notice number of pairs of collision} \sim s_k^2\text{)} \\ &= O\left[\sum_{i,j} (\text{i, j in the same bucket})\right] \\ &= O\left[\sum_{i,j} C_{ij}\right] \qquad (C_{ij} = \text{Probability that i j collides}) \\ &= O(n^2 \times \frac{1}{n}) \quad \text{(Note that top level has } n \text{ buckets))} \\ &= O(n).\text{(Note that top level has } n \text{ buckets))}\end{aligned}$$

From Markov Inequality we know[1] the $Pr(\sum s_i^2 > 2 \times O(n) = O(n)) < \frac{1}{2}$. Thus the expected number of attempts before getting a desired hashing is 2 (same argument from (a)).

Now in the similar argument as in (a), on the top level, the overall cost of rebuild is the same as (a), $O(n)$, because of the same expected attempt.

For the second level, in each bucket, we need to consider the rebuild cost for (1) top level rebuilds and (2) collision happened in second level hashing. For case (1), we know the expected number of trials in each bucket is 2, and by the same argument, the rebuild cost is $O(k)$ where k is the current inserted number of items. For case (2), every time a violation happens, it takes $O(b_k)$ to rebuild, whereb_k is number of items in bucket k.

Using the same technique in (a),

$$\begin{aligned}
\mathbb{E}(\text{cost of rebuild}) \leq\ & \mathbb{E}(\text{top level rebuilds until each break point}) \\
& \times \mathbb{E}(\text{rebuild cost for all items until break point in both levels}) \\
& + \mathbb{E}(\text{second level rebuilds until each break point}) \\
& \times \mathbb{E}(\text{rebuild cost for } \textit{all} \text{ items in each bucket}) \\
& = 2 \times \big(1 + 2 + 4 + 8 + 16 + ... + s\big) \\
& + 2 \times \big(1 + 2 + 4 + 8 + 16 + ... + s\big) \quad \text{(sum of all items in buckets will be } k\text{)} \\
& \leq 4 \times 3s \\
& = O(s)
\end{aligned}$$

(c) *(The amortized cost $O(1)$ follows from the total cost $O(n)$ for n insertion in (a). We follow the notions in (a) here.)*

We need to keep another integer keeping track of number of valid items in the table. The delete clean up only happens when there are n items in the table but only $n/2$ of it is valid. For number of items ranging between $n/2$ and n, we know the cost for all operations and analysis above only scaled by factor of 2, wouldn't affect $O(\cdot)$. In the extreme case, deleting items all the way down to 0 we need to at least rebuild when the valid item reduces to $n/2, n/4, n/8$.... Notice that in each rebuild, the cost is $O(k)$, where k is the number of valid items in the system. Therefore the total cost is $O(n/2 + n/4 + n/8 + ...) = O(n)$ for such deletions, which can be charged against the inserts.

[1] We can set $\mathbb{E}\left[\sum s_k^2\right] = \alpha n$, and use $2\alpha n$ in Markov Inequality analysis to be more rigorous.

Problem 6

(a)

Algorithm The pseudocode is as follows,

```
counter := 1
register := empty
while streaming:
  if register not empty and register == IP:
      counter += 1
  else:
      counter -= 1
  end if

  if counter == 0:
      register := IP
      counter := 1
  end if

end while
```

Analysis Because we don't care about the IP in the register if it is not dominating at least half of all IPs, we only need to show that it captures the right IP when it appears at least half of the time.

We prove this by contradiction. Suppose another IP j is the output other than the true dominating IP i. The above algorithm means, there is a period of time, that j starts to kick in and dominating the counter, because all IPs other than j can't diminish the counter increment due to j's appearance. Moreover, the fact that j can kick it, means i is not dominating in the period from beginning to j takes over, otherwise i would stay in the register. It is not immediately obvious that i is not dominating the first period, because initially it needs to do some work to take its position hold by others. We extend the period from i starts taking the register mirroring to i first comes in, then it is clear for the domination, as in Figure 1. Because i is not dominating from beginning to j takes over and not dominating in j's dynasty, it is not the dominating IP. Contradiction occurs.

(b)

Algorithm The extended pseudocode from (a) is as follows,

`counter[1] := 1,` $i \in \{1, 2, 3, ..., k\}$
`register[`i`] := empty,` $i \in \{1, 2, ..., k\}$

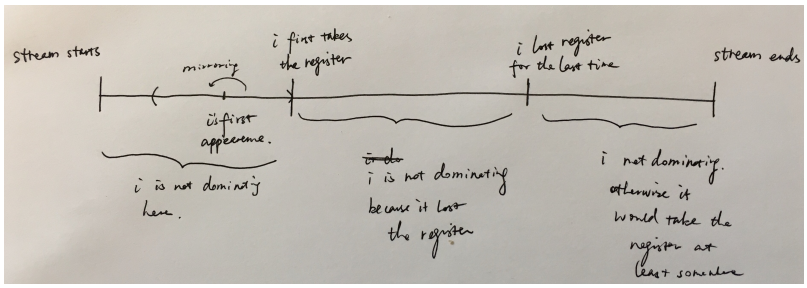

Figure 1: Illustration of the stream.

```
while streaming:
    stored := false

    for i in {1, 2, 3, ..., k}:
      if register[i] not empty and register[i] == IP:
          counter[i] += k
          stored := true
      else:
          counter[i] -= 1
      end if
    end for

    for i in {1, 2, 3, ..., k}:
      if counter <= 0 and !stored:
          register := IP
          counter := k
          stored := true
      end if
    end for
end while
```

Analysis Very similar to part (a), we can use proof by contradiction to show the correctness of the algorithm. Suppose another IP j is one of the outputs other than some other IP i that contributes to $1/(k+1)$ of the traffic. The above algorithm means, there is a period of time, that j starts to kick in and dominating one of the counter, because all IPs other than j can't diminish the counter increment due to j's appearance. Here the term 'domination' means has at least $1/(k+1)$ of the traffic, because of the $+ = k$ counting increment. Moreover, the fact that j can kick it, means i is not dominating in the period from beginning to j takes over, otherwise i would stay in the register. Because i is not dominating from beginning to j takes over and not dominating in j's dynasty, it is not of $1/(k+1)$ of all IPs. Contradiction occurs.