Assignment 1

Question 1

The total reward of first B blocks R_0 is equal to Bx, according to the question, we can deduce that $R_n = \frac{Bx}{r^n}$. So, the total amount of currency that can be issued is

$$\sum_{i=0}^{\infty} R_i = \sum_{i=0}^{\infty} Bxr^{-i} = \lim_{n\to\infty} Bx\frac{1-r^{-n}}{1-r^{-1}} = \frac{Bx}{1-r^{-1}} = \frac{Bxr}{r-1} \ (|r| > 1)$$

when r is greater than one. The total amount grows to infinity when r is not greater than one because R_n will be always greater than R_{n-1}.

Question 2

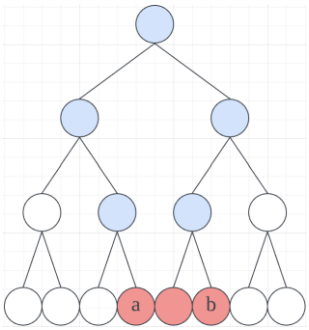
1) *Describe the data structure stored on the server, and how it is computed. Give an expression for the size of this data structure as a function of the length of L.*

For clarity, N is used to represent the length of L and M is used to represent the length of answer R to the queries. Also assume that the size of hash code is $\frac{H}{2}$ bits.

Merkel tree can be used in this question for generating proof information. But directly use Merkel tree is time consuming and the tree may take much more space comparing to the original list. The server may pass a subset of the Merkel tree to client as proof information. And the subset is simply union of single verification paths of each element in the answer to the query.

Because the number of nodes is two times of number of leaf nodes, as M grows, the size of it would eventually becomes $O(N)$. As time complexity of validating is proportional to size, the time complexity is also $O(N)$. When M is a small proportion of N, the complexity is near to $O(\log(N)\, M)$.

In addition to time complexity of verifying each node in the returned tree, there are also some "wasted nodes" in the tree: lots of nodes in the returned data are used for validation only.



The coloured nodes in the diagram are the data sent to client while red nodes are the nodes relevant to the answer to the query, and the blues are only used for validation.

Binary search tree (BST) is an efficient data structure to optimize Merkel tree:

As given in the question, the data is static, so efficiency of inserting or deleting data is not considered. Therefore, in order to store the number list, complete BST can be used to speed up the search operation. The server could use an array to represent the tree by defining the following additional rules:

Define the index of root as 0

Define the index of left child of ith element as

$2 * i + 1$,

and that of right child as

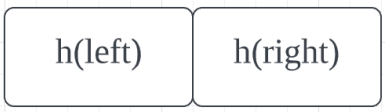
$2 * i + 2$.

Then the index of parent of ith element becomes:

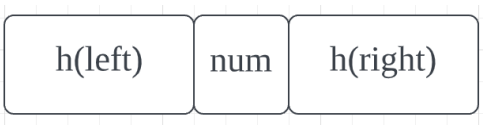
$(i - 1)/2$.

Structure and computation:

In stead of storing hash value only, each node in the BST Merkel tree stores both hash values of children and number value of itself, as illustrated below:



A node in standard Merkel tree



A node in BST Merkel tree

In the diagrams above, h(child) is obtained by calculating the hash value for concatenated h(left), num, and h(right) of that child. And the leaf nodes store the number only.

The construction of the tree is almost the same as a standard BST, in addition to it, the hash value for each node is computed.

Answering queries:

The way of getting all the numbers between a and b is to first search a and then get numbers that are greater than a until b.

The searching process is the same as that of a regular BST which time complexity is $O(\log(N))$. The algorithm for getting the number list is modified in-order traversing: recursively adding the right child tree of the father nodes until the current node is right child of the father node or b is met. The time complexity of this process is $O(M)$.

And the way of adding right child tree is simply done by in-order traversing the right child.

The pseudo code is:

```
def larger_list(a):
    result = [a]
    if a == b:
        return result
    # If b occurs in the process of in_order_traverse,
    # it will terminate and return immediately.
    result.append(in_order_traverse(right_child(a)))
    if is_left_child(father(a)):
        result.append(larger_list(father(a)))
    return result
```

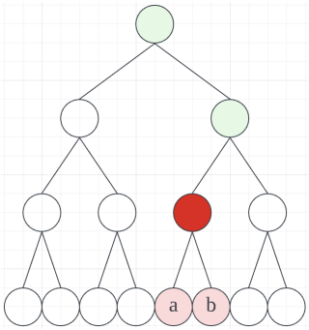
Because complete tree is used, the array is compact so that nearly no space is wasted. Therefore, the memory used to store the BST Merkel tree would be $(64 + H) * N$ bits $(O(N))$.

2) Describe the data that is stored at the client, and state its size.

The data stored at the client consists of two parts: answer and verification information. The answer is temporarily stored as a list R from a to b in ascending order whose size is $(64 + H) * M$ bits $(O(M))$. **Verification information** is the root node of the BST Merkel tree, the size of it is $64 + H$ bits.

3) Describe the "proof information" and state its size as function of R

The proof information is a subset of the BST Merkel tree which consists of all nodes in the answer list and all nodes on the shortest path from root to the answer list. For example:



The coloured nodes correspond is the data returned to the client with "top of the answer list R" coloured deep red. The path from root to deep red is the desired proof information. Client may directly use the deep red node as starting point to verify the correctness. The calculation is the same as computing the tree. The final value of root should match the stored version if the data is not corrupted.

The size of the green part is just the depth of the top node of R, it is approximately $(\log_2 N - \log_2 M) * (64 + H)\ bits$ $\big(O(log\ N - log\ M)\big)$.

If the client doesn't trust the value stored in deep red and really desire to perform the validation thoroughly, they can choose to start from the deepest nodes. In this case, the proof information takes up $(\log_2 N - \log_2 M + M) * (64 + H)\ bits$ $\big(O(log\ N + M)\big)$.

4) Describe the computation that the client performs to verify the proof state its size as function of R. Explain how this computation provides a guarantee.

The client has two option to verify, one is to verify the existence of the answer list, another is to also verify the correctness and integrity of the list.

To verify existence of the list, the client just starts from top of R (deep red node) to perform the following computation recursively:

1. Concatenate $hash(left)$, num and $hash(right)$ to yield t.
2. Pass $hash(t)$ to parent node, if this node is left child, pass to $hash(left)$, vice versa.
3. Compare $hash(t)$ and hash value of parent, if not equal, then warn use that the data is corrupted.
4. Until root node.

This process is similar to proving existence of data in a Merkel tree. The running time is $O(\log(N) - \log(M))$. To verify the correctness of the list, the client should follow the procedure below:

1. Add all leaf nodes to an array l, then dispose these nodes.
2. Calculate and pass hash value as above.
3. Until top of the answer list.

The process guarantees that each node in the answer list exists in the correct place in the BST Merkel tree, thus the correctness and integrity of the answer is proved. The running time is $O(M)$.

The existence option is compulsory, when doing this, the client assumes that the answer list is complete and correct. This guarantees that the returned answer list is indeed a subset of the BST Merkel tree. They can also choose to verify the correctness and completeness if they want more certainty.

The combined complexity is $O(\log(N) + M)$. Which means that, for a given list (then N becomes constant), the combined running time is linear to M $\left(O(M)\right)$.

By using collision-free hash functions, the data structure is guaranteed unchangeable because each hash value corresponds a unique subtree.

5) Is your solution the most efficient possible solution for this special case?

In this case, the client just needs the hash value path in Merkel tree for the last number as proof information. And the information stored is the hash value of the last number in previous query and the hash value of root node. The verification process is to calculate hash sequence from a to b in the returned answer list starting from $hash(a)$ stored by client and then check the existence of b through Merkel tree. In this scenario, $hash()$ calculated from the hash sequence of L is directly used as leaf nodes $\left(hash(x_{i+1}) = hash(hash(x_i)\ ;\ x_{i+1})\right)$. The time complexity and complexity are $O(\log(N) + M)$.

Although my solution's complexity is the same as this special case in big O notation, my solution is less efficient in constant level. My solution uses MH bits more and it concatenates two hash values and a number, it also costs more time to verify the subtree than directly computing the hash chain.

Question 3

1) Show that each node is able to compute a decision at time 1

Because the network is fully connected and there is no faulty node, each node will know the *vote* information of all other nodes at time 1. Thus, they can make choice based on the same full information. If both 0 and 1 are available, each node just decides 0 by default.

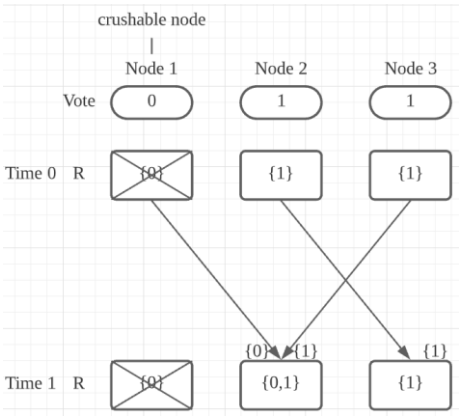
Pseudo code:

```
for i in nodelist:
    if 0 in R and 1 in R:
        decide(0)
    else:
        decide(R[0])
```

Because the decision is made based on R, and R is the same for each node. Due to fully connectivity, R[i] is union of all $\{vote_i\}$ sets so i can make decision that satisfy Validity. And because they are using the same decision rule and same information, the Agreement specification also holds.

2) *Give an example to show that if there is even one faulty (crashing) node, applying your decision rule from part 1 at time 1 does not satisfy the specification.*

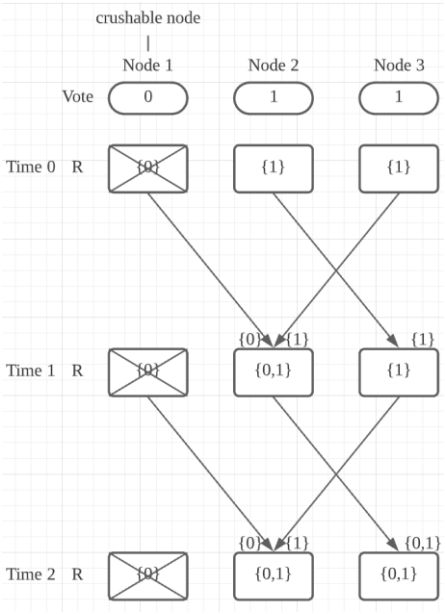
Assume there are three nodes, and only node 1 votes for 0. And then let node 1 crush at time 0:



If the decision rule from part 1 is applied, Node 2 will decide 0 but Node 3 will decide 1. This result is inconsistent with the **Agreement** specification.

3) *Show that if we repeat the message transmission rule for two rounds, then at time 2 the non-faulty nodes can make a decision so as to satisfy the specification.*

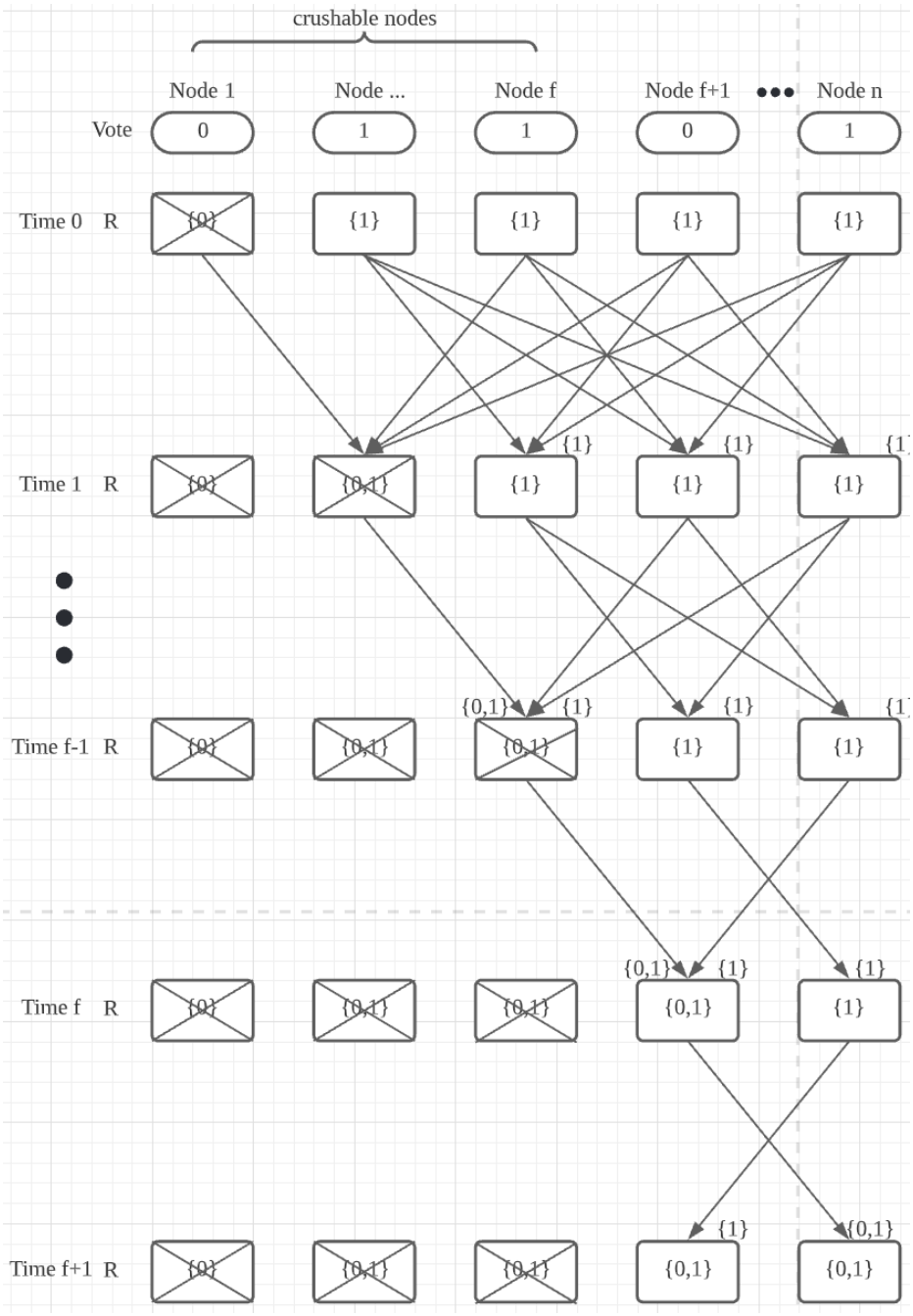
The worst case is that there is the faulty node crushes at time 0 and this node only send message to one non-faulty node. Even in that case, a normal node can get the full information at time 1, then it can propagate the full information in the next round. Therefore, at time 2 the non-faulty nodes can make a decision that satisfy the specification.



4) *Show that it is not possible to make a decision, so as to always satisfy the specification, before time f + 1.*

The worst case (the case that requires longest running time) message sequence chart is shown below. In this chart every faulty node manages to send its message out to only one normal node to ensure that the procedure is as long as possible. And the node that receive the message from faulty node must crush to ensure that this information is not propagated to every node. And there is a special limitation that only Node 1 votes for 0.

Because there are at most f nodes, the "secret" can be kept up to f rounds. To satisfy the specification, one more round is needed. So, there exists one case that a decision cannot be made before time $f + 1$. For clarity, the message sent to itself is omitted in the chart.



5) Show that repeating the protocol for f + 1 rounds enables a decision to be made at time f + 1, so as to satisfy the specification.

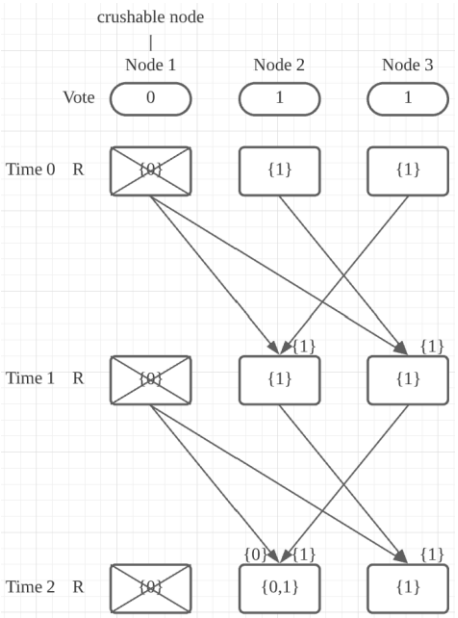
As discussed above, the worse case is illustrated in the chart above. There is an analogy of secret passing to explain why it is the longest possible case:

n secret agents want to make the **same decision based on the same information**, but they can't meet together, what they can do is to pass messages one-to-one. And there is only one agent knows the "more important" secret (denoted by 0 in this question). When a non-faulty agent knows the more important secret, he must decide base on that

information. The agent that holds that secret wants to tell everybody about that information, but he is going to be killed at time 0. What he can do is to tell someone else that secret. **Of course, he can bury the secret but the other agents can never know the secret then they would share the same knowledge ever since the secret is buried.** So, to make the process as long as possible, the agent being killed(crushed) must tell only one living(non-faulty) agent. The reason why he only talks to one agent is also to make the process as long as possible. Unfortunately, the agent that knows the secret will die from knowing too much, and he will be killed and have a chance to pass it to another one. This process is repeated again and again until their enemy run out of ammo at time f. At time f, f agents have been killed, and there is an alive agent knows the secret. He can finally tell everybody the secret. Eventually, they can establish a course of action at time $f+1$ that satisfy the specification because they all share the same knowledge base.

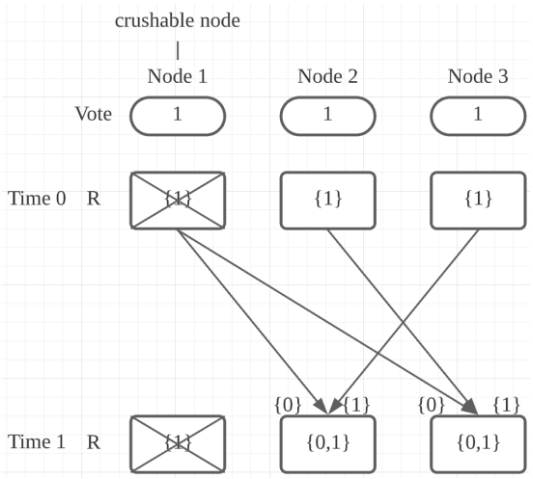
This example shows that even for the worst scenario, a decision can always be made at time $t+1$ to satisfy the specification.

6) *Under the new model show by example that if there are 3 nodes, one of which is faulty, then deciding at time f + 1 = 2 does not satisfy the specification. Is there any number of iterations to satisfy the specification?*



In this example, node 1 is crushed at time 0, and only node 1 voted for 0. In the first round, the faulty node sends $\{1\}$ to all other nodes. Then in the second round, it sends $\{0\}$ to node 2 but $\{1\}$ to node 3. So, at time 2, deciding at time 2 does not satisfy **Agreement** specification. There does not exist a number of iterations to that allows a decision to be made so as to satisfy the specification. Because the faulty node can send the $\{0\}$ message to a non-faulty node at any time. It can send the message to one of the nodes just at the time when it is going to decide.

There is also a case that any decision is always inconsistence with the specification: Each node votes for 1, and node 1 crushes at time 0. But the faulty node sends $\{0\}$ to others. The non-faulty nodes would then decide x to be 0. But it violates **Validity** specification, because there is no j such that $vote_j = 0$. The decision is always wrong after any number of iterations.



Question 4

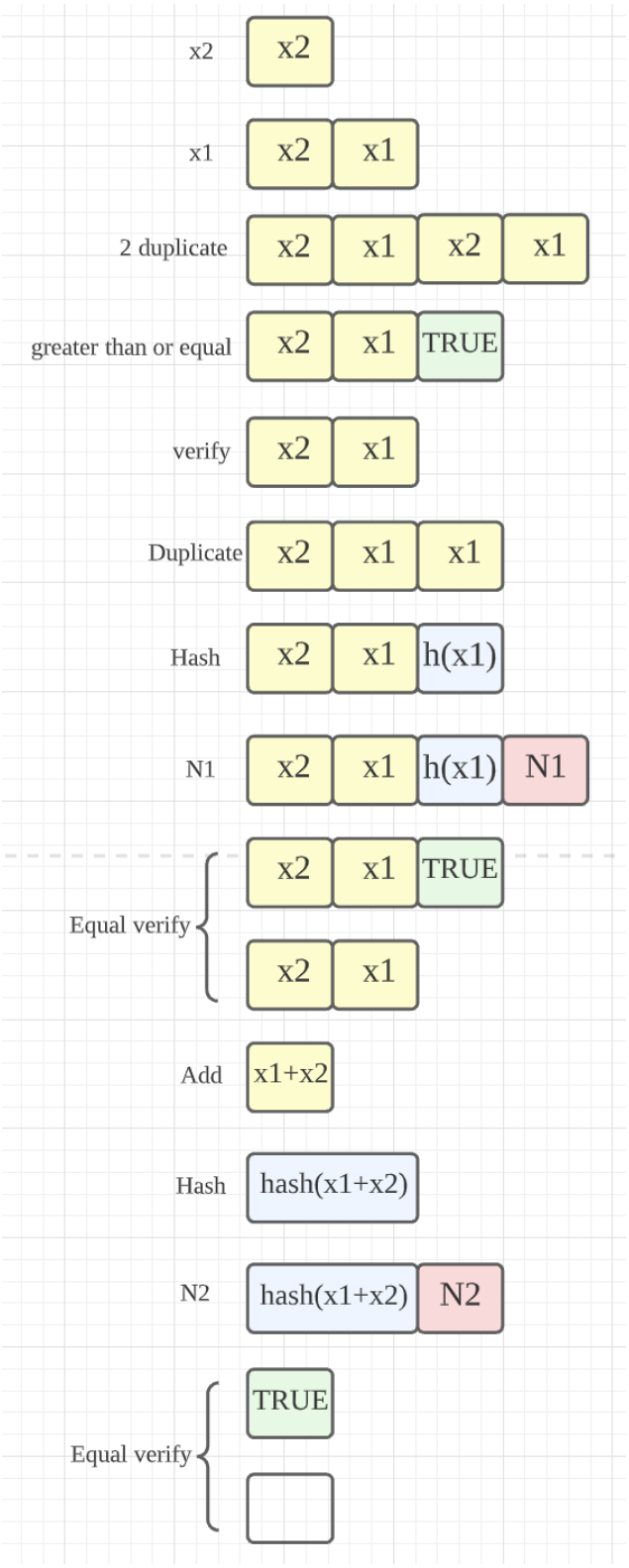
The **locking script** is:

```
OP_2DUP OP_GREATERTHANOREQUAL OP_VERIFY
OP_DUP OP_SHA256 N1 OP_EQUALVERIFY
OP_ADD OP_SHA256 N2 OP_EQUALVERIFY
```

And the **unlocking script** is:

```
x2 x1
```

The sequence of stack values for a successful unlocking computation is:



Mary could initiate a transaction to her address with unlocking script x_2, x_1 from the transaction the agency made.

Question 5

1) Recombining the p + 1 shares allows k to be reconstructed.

Firstly, add up the $p+1$ shares and then take modulus of n to yield s:

$$s = \Big((k + x_1 \bmod n) + (k + x_2 \bmod n) + \cdots + \big(k + x_{p+1} \bmod n\big)\Big) \bmod n$$

$$= \Big((k + x_1) + (k + x_2) + \cdots + \big(k + x_{p+1}\big)\Big) \bmod n$$

$$= \Big((p+1)k + \big(x_1 + x_2 + \cdots + x_{p+1}\big)\Big) \bmod n$$

$$= \left((p+1)k + \left(x_1 + x_2 + \cdots x_p - \Big(\big(x_1 + x_2 + \cdots x_p\big) \bmod n\Big)\right)\right) \bmod n$$

$$= (p+1)k \bmod n$$

$$= \big((p+1) \bmod n\big)k \bmod n.$$

Because $s, p,$ and n are constant, k can be reconstructed by solving the congruence function $ax \equiv b(mod\ n)$ where $a = (p+1) \bmod n$ and $b = s$.

To ensure that the function is solvable, we need to keep $a \neq 0$, that is, $(p+1) \bmod n \neq 0$.

Also, unique solution is needed to guarantee that the solution is exactly k, another constraint becomes $\gcd(a, n) = 1$. In other words, $p+1$ and n need to be coprime. This also implies that $(p+1) \bmod n \neq 0$.

2) An attacker who learns any p of the p + 1 shares still gets no information about k.

Let's just denote $k + x_{p+1} \bmod n$ as the last share l because losing any share is logically equivalent.

This can be showed by adding up all the x together:

$$x_1 + \cdots + x_{p+1} = 0$$

So, any x_i is negative to the sum of remaining x.

To prove the proposition, firstly add up the p shares and then take modulus of n to yield r:

$$r = \Big((k + x_1 \bmod n) + (k + x_2 \bmod n) + \cdots + \big(k + x_p \bmod n\big)\Big) \bmod n$$

$$= \Big(pk + \big(x_1 + x_2 + \cdots + x_p\big)\Big) \bmod n$$

$$= \big(pk - x_{p+1}\big) \bmod n$$

$$= \Big((p+1)k - k - x_{p+1}\Big) \bmod n$$

$$= \Big((p+1)\mathrm{k} - \big(k + x_{p+1}\big) \bmod n\Big) \bmod n$$

Then we have $r = \big((p+1)k - l\big) \bmod n$

Because n, p and r are known and $l \leq n$, for each k', there is always a unique solution so that $r = \big((p+1)k - l\big) \bmod n$ is true.

Rearranging the equation above we have:

$$(p+1)k - r - an = l$$

where a is a deterministic integer so that $(p+1)k - r - an < n$.

From the constraints defined in part 1, we know that $(p+1)$ and n are coprime. We can deduce that for each $k' \in [0, n)$, the value of l is never repeated. In other words, if $k' \bmod n$ is drawn uniformly, the distribution of l will also be uniform.

Thus, the attacker's point of view, each k' is potential to be the secret key.

Therefore, we have proved that any key k' has the same probability, from the attacker's point of view, of being the secret k.