1. Below are the pseudocodes for the assigned functions:

Last interval in $A \ldots B$ starting at or before k	First interval in $A \triangleleft B$ ending at or after k	Last interval in $A \triangleleft B$ ending at or before k

$\rho'(A \ldots B, k) =$
if $k = \infty$ **then**
return $[\infty, \infty]$
if $k = -\infty$ **then**
return $[-\infty, -\infty]$
$[u, v] \leftarrow \rho'(A, k)$
if $[u, v] = [-\infty, -\infty]$ **then**
return $[-\infty, -\infty]$
$[u', v'] \leftarrow \rho'(B, v)$
if $[u', v'] = [-\infty, -\infty]$ **then**
return $[-\infty, -\infty]$
$[u'', v''] \leftarrow \rho(A, u' - 1)$
return $[u'', v']$
	$\rho(A \triangleleft B, k) =$
	if $k = \infty$ **then**
	return $[\infty, \infty]$
	if $k = -\infty$ **then**
	return $[-\infty, -\infty]$
	$[u, v] = \rho(A, k)$
	if $[u, v] = [\infty, \infty]$ **then**
	return $[\infty, \infty]$
	$[u', v'] \leftarrow \rho(B, v + 1)$
	if $[u', v'] = [\infty, \infty]$ **then**
	return $[\infty, \infty]$
	if $u' \leq u$ **then**
	return $[u, v]$
	else
	return $\rho(A \triangleleft B, v + 1)$
		$\tau'(A \triangleleft B, k) =$
		if $k = \infty$ **then**
		return $[\infty, \infty]$
		if $k = -\infty$ **then**
		return $[-\infty, -\infty]$
		$[u, v] = \tau'(A, k)$
		if $[u, v] = [-\infty, -\infty]$ **then**
		return $[-\infty, -\infty]$
		$[u', v'] \leftarrow \rho(B, v)$
		if $[u', v'] = [\infty, \infty]$ **then**
		return $[\infty, \infty]$
		if $u' \leq u$ **then**
		return $[u, v]$
		else
		return $\tau'(A \triangleleft B, v - 1)$

2. Given $Pr["a"] = 0.7$ and $Pr["b"] = 0.3$, if we group symbols into blocks of $m = 2$ symbols, we get the following new distribution:

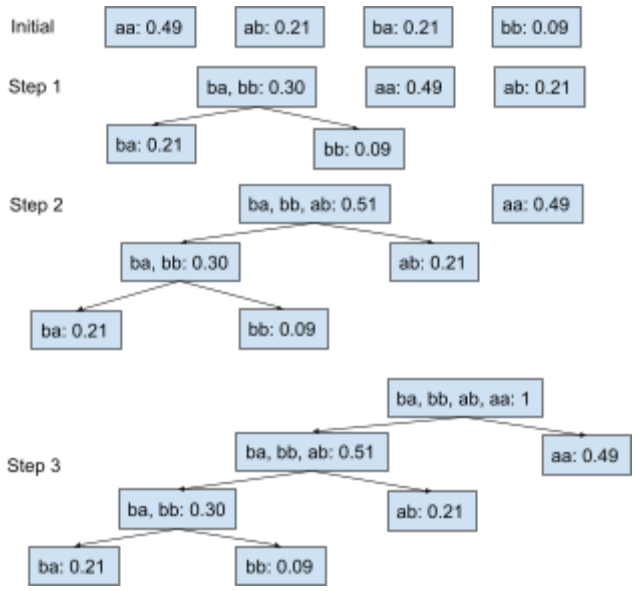
$Pr["aa"] = 0.7 * 0.7 = 0.49$

$Pr["ab"] = 0.7 * 0.3 = 0.21$

$Pr["ba"] = 0.3 * 0.7 = 0.21$

$Pr["bb"] = 0.3 * 0.3 = 0.09$

We then construct the Huffman tree as follows:



3. A δ-code is a variant of a γ-code, with the difference between the two being that δ-codes store the selector component as a γ-code instead of in a unary representation. In order to prove that δ -codes are prefix-free, we must consider the following two cases:

Case 1: The selector component is the same.

When we have two distinct gap numbers to convert to δ-codes that have the same selector component, we will find that they are prefix-free because they will have different values in their body components. As an example, 5 has a δ-code of 01 1 01 and 7 has a δ-code of 01 1 11; while both numbers have the same selector component (01 1), their body components are different (01 for 5, 11 for 7) and thus can't be prefixes of one another and are prefix-free.

Case 2: The selector component is different.

When we have two distinct gap numbers to convert to δ-codes that have different selector components, we will find that they are prefix-free because the γ-codes that represent the selector components are themselves prefix-free. As an example, 5 has a δ-code of 01 1 01 and 16 has a δ -code of 001 01 0000; both numbers have different selector components (01 1 for 5, 001 01 for 16) and are in the form of γ-codes, which are prefix-free, so they can't be prefixes of one another and are prefix-free.

Since both cases came to the conclusion that δ-codes are prefix-free and there are no other cases, we conclude as a whole that δ-codes are prefix-free.

4. Below are the possible splits of the remaining 60 bits:

Selector	Number of Δ's	Bits per Δ	Unused bits per word
0	1	60	0
1	2	30	0

2	3	20	0
3	4	15	0
4	5	12	0
5	6	10	0
6	7	8	4
7	8	7	4
8	10	6	0
9	12	5	0
10	15	4	0
11	20	3	0
12	30	2	0
13	60	1	0

Between the Simple-9 and Simple-14 methods, I would expect the Simple-14 method to yield better compression rates on longer lists with similar gap sizes and the Simple-9 method to yield better compression rates on shorter lists with various gap sizes. To illustrate this, we consider the following scenarios:

Scenario 1: Short list, similar gaps

In this scenario, using either would be fine, but Simple-9 is better only because it uses a smaller word size to accomplish the same thing as Simple-14 would.

Scenario 2: Short list, various gaps

In this scenario, Simple-9 is better than Simple-14 because it would take longer for Simple-14 to find the best split out of 14 for each gap than for Simple-9 to find the best split out of 9 for each gap.

Scenario 3: Long list, similar gaps

In this scenario, Simple-14 is better than Simple-9 because Simple-14 has more splits available to choose from for each gap and uses a bigger word size to store more than Simple-9 can.

Scenario 4: Long list, various gaps

In this scenario, Simple-14 is probably better than Simple-9 but only because the list would be too large for Simple-9 to handle in the first place.

5. Below is the initial state of generation indexes:

After the first partition is written to disk, we create a new generation 1 index. Since this results in two generation 1 indexes, we merge them into a generation 2 index, resulting in the following:

After the second partition is written to disk, we create a new generation 1 index, resulting in the following:

After the third partition is written to disk, we create a new generation 1 index. Since this results in two generation 1 indexes, we merge them into a generation 2 index. After this, we have two generation 2 indexes, which we merge into a generation 3 index. Finally, we have two generation 3 indexes, which we merge into a generation 4 index, resulting in the following: