CoE 164

Computing Platforms

Midterm Problem

Academic Period: 2nd Semester AY 2022-2023

Workload: 12 hours

Synopsis: Obfuscated, compressed, and executable text

Table of Contents

Annex A: The diropql Language

diropql (diro portable query language) is a minimalistic language used to "encrypt" text across some unsecure channel. A diropql program starts with 10000 memory cells with each cell initialized to a value of zero. All cells take in only 1-byte *unsigned* integers. It will also start with a memory pointer `mp`, an instruction pointer `ip`, and an output queue `oq`. `mp` and `ip` are nonnegative integers initialized to zero since diropql is a zero-index language. `oq`, on the other hand, is initially empty and is assumed to have an infinite size.

A diropql source code is a string encoded in ASCII or UTF-8. If saved as a file, it should have the extension `.dpql`. It is run by the diropql interpreter, which starts by looking at the program source code and finding the first valid command in the language by incrementing `ip` one at a time across it. Any invalid commands are ignored by the interpreter. When a valid command is read, the interpreter will process it depending on the command and the value pointed to by `mp`. diropql consists of the following seven commands capable of manipulating the contents of the memory cells.

command	description
`l`	decrement `mp` by one
`r`	increment `mp` by one
`i`	increment content pointed by `mp` by one
`d`	decrement content pointed by `mp` by one
`o`	push content pointed by `mp` to `oq`
`p`	change ip to the index of the matching q command *plus one* if the content pointed by `mp` is zero
`q`	change ip to the index of the matching p command *plus one* if the content pointed by mp is nonzero

Note that in general, the interpreter increments `ip` after each command. However, the `p` and `q` commands modify the `ip`. In this case, the nesting of `p` and `q` follow the same as in math equations (i.e. inner `pq`s match and outer `pq`s match in the following program: `ppidqq`).

If `mp` will point to an invalid memory cell due to an `lr` command, the value of `mp` will wrap around. For example, if `mp` is currently 0 and the next command decrements it by one, the new value of `mp` will be 9999.

If the contents of a memory cell will overflow due to an `id` command, the value will be clamped. For example, if a memory cell currently contains 255 and the next command increments it by one, the value will be unchanged.

Annex B: diropqlz File Format

diropql is a minimalistic language containing only seven characters. Hence, programs in this language can get very large. The diropqlz file format stores a compressed version of a diropql program. Programs written in this format should be decompressed first before being run into the compiler.

A diropqlz file is Base85 encoded so that it can be sent across channels in a human-readable format. In addition, programs that are to be saved in this file format should have the `.dpqlz` extension. To convert a diropql program into its `.dpqlz` equivalent, it has to be first compressed using the obfuscator. Then, the result should be appended with metadata in the following layout written from left to right:

- 8 bytes - 64-bit number denoting the length in bytes of the obfuscated message as the ceiling of the number of bits divided by eight.
- 1 byte - 8-bit number denoting the number of bits to *exclude or ignore* starting at the end of the obfuscated message. It should be a value between 0 and 7.
- 8 bytes - 64-bit number denoting the index I that was part of the output of the Burrows-Wheeler transform
- 16 bytes - 16 1-byte numbers corresponding to the bit lengths of the canonical Huffman codebook derived from the encoding process. The first nine numbers correspond to the alphabet of the run-length encoding output sorted by value. The remaining seven numbers are reserved for future use, and should be set to zero.
- Rest of the binary bits - the obfuscated message itself

All values are written in big-endian (i.e. the most significant bit appears at the leftmost).

This final compressed output may have to be appended with bits set to 0 *after* the obfuscated message to ensure that the total length of the output in bits is divisible by eight.

After the compressed program has been arranged, the program can now be encoded into Base85. Then, a magic string (DIROPQLZ) will be prepended to the encoded program to finish processing.

Annex C: Burrows-Wheeler Transform

The *Burrows-Wheeler transform* (BWT) is a text transform invented in 1994. The transform groups the same occurring letters via permutation.

Encoding

The most simple algorithm to do the transform is as follows:

1. Get a message M and append a *sentinel character* to it. This sentinel is used to denote the end of the string. In textbooks, they usually use a dollar sign ($), but the `NUL` ASCII character (`\0`, $^{\text{NUL}}$) is most practically used. From now on, this sentinel is part of the original message M.
2. Get M and its $|M| - 1$ other permutations by shifting M to the left one letter at a time, and moving the shifted out letter to the rightmost side of M. Place M and the permutations into an array S.
3. Sort S in lexicographically increasing order (i.e. ascending alphabetical/ASCII value). Let this sorted array be named S_{sorted}.
4. Get the last letter of each element of S_{sorted} and concatenate them in the order in which permutation the letter belongs. This is now the transformed text. The index I of the row where M is is also recorded.

As an example, we have a string M:

$$M = bananaaa!^{\text{NUL}}$$

We then generate a matrix of the string M with each subsequent string shifted to the left by one and sorting them in ascending alphabetical or ASCII values. The table below shows these matrices of strings.

Left Shifted	Sorted	Last Character in Sorted
bananaaa!$^{\text{NUL}}$	$^{\text{NUL}}$bananaaa!	!
ananaaa!$^{\text{NUL}}$b	!$^{\text{NUL}}$bananaaa	a
nanaaa!$^{\text{NUL}}$ba	a!$^{\text{NUL}}$bananaa	a
anaaa!$^{\text{NUL}}$ban	aa!$^{\text{NUL}}$banana	a
naaa!$^{\text{NUL}}$bana	aaa!$^{\text{NUL}}$banan	n
aaa!$^{\text{NUL}}$banan	anaaa!$^{\text{NUL}}$ban	n
aa!$^{\text{NUL}}$banana	ananaaa!$^{\text{NUL}}$b	b
a!$^{\text{NUL}}$bananaa	bananaaa!$^{\text{NUL}}$	$^{\text{NUL}}$

!$^{N}_{U}{}_{L}$bananaaa	naaa!$^{N}_{U}{}_{L}$bana	a
$^{N}_{U}{}_{L}$bananaaa!	nanaaa!$^{N}_{U}{}_{L}$ba	a

Then, we get each last character of the strings in the sorted matrix and concatenate them together to get the transformed string:

$$V =! aaannb^{N}_{U}{}_{L}aa$$
$$I = 7$$

Decoding: Slow and Simple

The transform is reversible, with the simple algorithm as follows:

1. Get the transformed message V and assume that it already contains the sentinel character. Also, initialize a $|V| \times |V|$ matrix C_{BWT}.
2. The following should be done $|V|$ times:
    a. Arrange V like a column vector and append it in the rightmost empty column of C_{BWT}.
    b. Sort C_{BWT} in lexicographically increasing order (i.e. ascending alphabetical order/ASCII order).
3. Find the row in C_{BWT} whose last element, or character, is the sentinel character. That row is the original string.

As an example, we have a transformed string V:

$$V =! aaannb^{N}_{U}{}_{L}aa$$

We then generate a matrix of the string V with V being converted into a column vector and sorted in ascending alphabetical order. Every after each iteration, we will append V to the left of the corresponding substrings until each of the strings in the matrix when read each row will have a length equal to $|V|$. The table below shows these matrices of strings for the first five iterations. Note that the headers are annotated by iteration index and the column is sorted if it has a suffix S attached to it.

0	**0S**	**1**	**1S**	**2**	**2S**	**3**	**3S**	**4**
!	$^{N}_{U}{}_{L}$	!$^{N}_{U}{}_{L}$	$^{N}_{U}{}_{L}$b	!$^{N}_{U}{}_{L}$b	$^{N}_{U}{}_{L}$ba	!$^{N}_{U}{}_{L}$ba	$^{N}_{U}{}_{L}$ban	!$^{N}_{U}{}_{L}$ban
a	!	a!	!$^{N}_{U}{}_{L}$	a!$^{N}_{U}{}_{L}$	!$^{N}_{U}{}_{L}$b	a!$^{N}_{U}{}_{L}$b	!$^{N}_{U}{}_{L}$ba	a!$^{N}_{U}{}_{L}$ba
a	a	aa	a!	aa!	a!$^{N}_{U}{}_{L}$	aa!$^{N}_{U}{}_{L}$	a!$^{N}_{U}{}_{L}$b	aa!$^{N}_{U}{}_{L}$b
a	a	aa	aa	aaa	aa!	aaa!	aa!$^{N}_{U}{}_{L}$	aaa!$^{N}_{U}{}_{L}$

n	a	na	aa	naa	aaa	naaa	aaa!	naaa!
n	a	na	an	nan	ana	nana	anaa	nanaa
b	a	ba	an	ban	ana	bana	anan	banan
$^{N}{}_{U}{}_{L}$	b	$^{N}{}_{U}{}_{L}$b	ba	$^{N}{}_{U}{}_{L}$ba	ban	$^{N}{}_{U}{}_{L}$ban	bana	$^{N}{}_{U}{}_{L}$bana
a	n	an	na	ana	naa	anaa	naaa	anaaa
a	n	an	na	ana	nan	anan	nana	anana

After a few iterations, we will be able to have a matrix of strings where each string is of length $|V|$. The last two iterations are shown below.

8	**8S**	**9**	**9S**
!$^{N}{}_{U}{}_{L}$bananaa	$^{N}{}_{U}{}_{L}$bananaaa	!$^{N}{}_{U}{}_{L}$bananaaa	$^{N}{}_{U}{}_{L}$bananaaa!
a!$^{N}{}_{U}{}_{L}$banana	!$^{N}{}_{U}{}_{L}$bananaa	a!$^{N}{}_{U}{}_{L}$bananaa	!$^{N}{}_{U}{}_{L}$bananaaa
aa!$^{N}{}_{U}{}_{L}$banan	a!$^{N}{}_{U}{}_{L}$banana	aa!$^{N}{}_{U}{}_{L}$banana	a!$^{N}{}_{U}{}_{L}$bananaa
aaa!$^{N}{}_{U}{}_{L}$bana	aa!$^{N}{}_{U}{}_{L}$banan	aaa!$^{N}{}_{U}{}_{L}$banan	aa!$^{N}{}_{U}{}_{L}$banana
naaa!$^{N}{}_{U}{}_{L}$ban	aaa!$^{N}{}_{U}{}_{L}$bana	naaa!$^{N}{}_{U}{}_{L}$bana	aaa!$^{N}{}_{U}{}_{L}$banan
nanaaa!$^{N}{}_{U}{}_{L}$b	anaaa!$^{N}{}_{U}{}_{L}$ba	nanaaa!$^{N}{}_{U}{}_{L}$ba	anaaa!$^{N}{}_{U}{}_{L}$ban
bananaaa!	ananaaa!$^{N}{}_{U}{}_{L}$	bananaaa!$^{N}{}_{U}{}_{L}$	ananaaa!$^{N}{}_{U}{}_{L}$b
$^{N}{}_{U}{}_{L}$bananaaa	bananaaa!	$^{N}{}_{U}{}_{L}$bananaaa!	bananaaa!$^{N}{}_{U}{}_{L}$
anaaa!$^{N}{}_{U}{}_{L}$ba	naaa!$^{N}{}_{U}{}_{L}$ban	anaaa!$^{N}{}_{U}{}_{L}$ban	naaa!$^{N}{}_{U}{}_{L}$bana
ananaaa!$^{N}{}_{U}{}_{L}$	nanaaa!$^{N}{}_{U}{}_{L}$b	ananaaa!$^{N}{}_{U}{}_{L}$b	nanaaa!$^{N}{}_{U}{}_{L}$ba

The inverse transform of V is the string at the last column of the table where the $^{N}{}_{U}{}_{L}$ character is at the very end.

$$M = bananaaa!\ ^{N}{}_{U}{}_{L}$$

Decoding: Following the Indices

The previously outlined algorithm is simple to implement but takes around $O(n^2 \lg n)$ time to process (using $O(n \lg n)$ sorting done n times), which will scale up quite fast especially if the strings get very long. There is actually a way to be able to derive the inverse transform faster using the index I by looking at the relation between the rotated strings during left shifting and the resulting sorted strings. The algorithm is as follows:

1. Create an array of 2-ary tuples T with each element containing $(V[i],\ i)$, with $V[i]$ being a letter in the transformed string V and the zero-indexed position i where it appears, respectively.
2. Sort T using radix sort in ascending first element, then finally in ascending second element.
3. Create a new array L from the second values of each element in the sorted tuple array T. L contains the amount of left shift of the original string M.
4. Initialize a value L_{idx} denoting the current index under consideration. Its initial value is the index I supplied during the Burrows-Wheeler Transform.
5. For a loop iterating $|V|$ times
    a. Get $L[L_{idx}]$, or the left shift at that index.
    b. Push the character corresponding to the index $V[L[L_{idx}]]$ to the original string M.
    c. Set the new L_{idx} to be equal to $L[L_{idx}]$.
6. M will now contain the original string.

As an example, we have a transformed string V and the index I provided by the Burrows-Wheeler Transform:

$$V =! aaannb^{\text{NUL}}aa$$
$$I = 7$$

We then create a list of tuples T, sort them accordingly, and get their respective second elements to get the list of left shifts L.

T unsorted	! 0	a 1	a 2	a 3	n 4	n 5	b 6	NUL 7	a 8	a 9
T sorted	NUL 7	! 0	a 1	a 2	a 3	a 8	a 9	b 6	n 4	n 5
L	7	0	1	2	3	8	9	6	4	5

Then, we initialize $L_{idx} = 7$ and iterate through a loop $|V|$ times, replacing L_{idx} with $L[L_{idx}]$ after each iteration. Before replacing, the character in V at position $L[L_{idx}]$ is pushed into an array M that will contain the original message. The progression of the loop is shown below:

i	**Lidx**	**L at Lidx**	**V at Lidx**	**M**
0	7	6	b	b

1	6	9	a	a
2	9	5	n	n
3	5	8	a	a
4	8	4	n	n
5	4	3	a	a
6	3	2	a	a
7	2	1	a	a
8	1	0	!	!
9	0	7	$^{N}{}_{U}{}_{L}$	$^{N}{}_{U}{}_{L}$

Note that at the last iteration, L_{idx} should have the same value as the index I. The transformed message is therefore:

$$M = bananaaa!\,^{N}{}_{U}{}_{L}$$

The transform can be immediately made from a *suffix array*. A suffix array of a string M is a 1D array consisting of the indices of the starting character of a suffix in M. The suffixes are sorted by increasing alphabetical order and then by increasing length. The BWT of the string is the corresponding index of the character pointed to by the suffix array.

Annex D: DC3/skew Algorithm

The DC3 (difference cover mod 3) algorithm builds the suffix array of a string in around linear time. It looks at the characters in the string by threes and does a recursive sort and merge routine to build the array.

The algorithm is recursive, with each recursion following the three steps outlined below. Note that the recursion only happens under certain circumstances.

1. Find and sort $R_{1,2}$ order
2. Find and sort R_{0} order
3. Merge R_{0} and $R_{1,2}$ order

For subsequent discussion in this algorithm, the original string will be saved in an array T, which is the integer representation (ASCII) of each character in the string.

Finding and Sorting R12 Order

The $R_{1,2}$ order looks into every other first or second index of a string and collates a triplet of the three characters starting at those indices. The algorithm to find the order is as follows:

1. Append three zeros at the end of the T.
2. Get a list of indices $R_{1,2,idx}$ denoting the indices of T whose modulo against 3 is *not* zero and the index is at most three less from the length of the string, or $R_{1,2,idx} + 3 \leq |T|$.
3. Sort $R_{1,2,idx}$ by putting those whose modulo is one before those whose modulo is two, breaking ties by sorting by ascending values.
4. For each index $r_{1,2}^{(i)}$ in $R_{1,2,idx}$, get three contiguous values in T starting from index $r_{1,2}^{(i)}$. This is the same as getting a slice $T[r_{1,2}^{(i)} .. r_{1,2}^{(i)} + 3]$ of T. Save each slice in an array $R_{1,2,str}$
5. Do a radix sort on $R_{1,2,str}$ by sorting the slices against their first values, then by their second values, and finally by their last value.
6. If there are ties in the sorting against all of the values in $R_{1,2,str}$, we recursively have to find a new $R_{1,2}$ order.
7. Otherwise, the corresponding $R_{1,2,idx}$ in the sorted $R_{1,2,str}$ is the desired $R_{1,2}$ order.

As an example, we have a string M of length 10:

$$M = bananaaa!^{\text{NUL}}$$

The equivalent integer representation of this string is shown below. Note that we have already appended the required three extra zeros at the end, and for easier reading, the indices are also annotated.

M	b	a	n	a	n	a	a	a	!	$^{N}_{U_L}$			
T	98	97	110	97	110	97	97	97	33	0	0	0	0
idx	0	1	2	3	4	5	6	7	8	9	10	11	0

Next, we get and process the $R_{1,2,idx}$ array and get the corresponding triplets as shown below:

R12 index unsorted	1	2	4	5	7	8	10
R12 index sorted	1	4	7	10	2	5	8
R12 str unsorted	97 110 97	110 97 97	97 33 0	0 0 0	110 97 110	97 97 97	33 0 0

Then, we sort the $R_{1,2,str}$ array while keeping track of which $R_{1,2,idx}$ each triplet belongs to.

R12 str index unsorted	1	4	7	10	2	5	8
R12 str unsorted	97 110 97	110 97 97	97 33 0	0 0 0	110 97 110	97 97 97	33 0 0
R12 str index sorted	10	8	7	5	1	4	2
R12 str sorted	0 0 0	33 0 0	97 33 0	97 97 97	97 110 97	110 97 97	110 97 110

Since we do not have ties in the sorted order, we will *not* recursively run the algorithm. Instead, we get the now sorted $R_{1,2,idx}$ from the radix sort and treat it as the desired $R_{1,2}$ order.

$$R_{1,2} = [10, 8, 7, 5, 1, 4, 2]$$

Finding and Sorting R0 Order

The R_0 order looks into the indices of a string not considered during finding of the $R_{1,2}$ order and collates a 2-ary tuple. Note that knowledge of $R_{1,2}$ is required. The algorithm to find the order is as follows:

1. Get a list of indices $R_{0,idx}$ denoting the indices of T whose modulo against 3 is zero and the index is at most three less from the length of the string, or $R_{0,idx} + 3 \leq |T|$.
2. For each index $r_0^{(i)}$ in $R_{0,idx}$, get the value of T at that index, collate it into a tuple $(T[r_0^{(i)}],\ r_0^{(i)})$, and push it into an array $R_{0,str}$.
3. Do a radix sort on $R_{0,str}$ by sorting the slices against their first values.
4. If there are ties in the sorting against the first value in $R_{0,str}$, then for each tie lumped as a bucket:
    a. Add one to their second value, find their corresponding positions in $R_{1,2}$, and replace their corresponding first values with these positions *plus one*.
    b. Sort the bucket.
5. Get the last value of each tuple in the sorted $R_{0,str}$ and push them into an array R_0. R_0 contains the desired R_0 order.

Continuing the previous example, we get and process the $R_{0,idx}$ array and get the corresponding tuples as shown below:

R0 index	0	3	6	9
T[R0 index]	98	97	97	0
R0 str	98 0	97 3	97 6	0 9

Then, we sort the $R_{0,str}$ array to get the following progression

R0 str unsorted	98 0	97 3	97 6	0 9
R0 str sorted	0 9	97 3	97 6	98 0

As shown, the R_0 indices at 3 and 6 have the same first values. Therefore, a tie-breaking sort is needed against these two values. We will then replace these tuples in this manner:

R0 str sorted	97 3	97 6
R0 str[1] plus one	4	7
Position of R0 str[1] in R12	5	2
New R0 str sorted	6 3	3 6

Now, we can sort the tied tuples accordingly. With this, we can now derive the R_0 indices by getting the second element of each tuple.

R0 str unsorted	98 0	6 3	3 6	0 9
R0 str sorted	0 9	3 6	6 3	98 0
R0	9	6	3	0

The R_0 index is therefore:

$$R_0 = [9, 6, 3, 0]$$

Merging R0 and R12

After getting the orders, we are now ready to merge these two arrays and build the resulting suffix array. Merging also needs the knowledge of and T. The algorithm is as follows:

1. Initialize two pointers $r_{0,now}$ and $r_{1,2,now}$ corresponding to the current index in R_0 and $R_{1,2}$ that we are comparing for merging. These are set to zero, corresponding to them starting at the leftmost of each array.
2. Check the first element of R_0 and $R_{1,2}$, and delete or ignore the element whose value is greater or equal to the length of T. This is the same as setting either $r_{0,now}$ or $r_{1,2,now}$ to one.
3. For each iteration until either the value $r_{0,now}$ or $r_{1,2,now}$ exceeds their corresponding array length.
    a. Compare $t_0 = T[r_{0,now}]$ and $t_{1,2} = T[r_{1,2,now}]$.
        i. If $t_0 < t_{1,2}$, push $R_0[r_{0,now}]$ into the resulting suffix array and increment $r_{0,now}$.

ii. If $t_0 > t_{1,2}$, push $R_{1,2}[r_{1,2,now}]$ into the resulting suffix array and increment $r_{1,2,now}$.
iii. Otherwise, if $t_{1,2}\ mod\ 3 = 1$, compare the relative positions of $t_0 + 1$ and $t_{1,2} + 1$ in $R_{1,2}$.
    1. If $t_0 + 1$ comes first (i.e. has a lower index), then push $R_0[r_{0,now}]$ into the resulting suffix array and increment $r_{0,now}$.
    2. Otherwise, if $t_{1,2} + 1$ comes first (i.e. has a lower index), then push $R_{1,2}[r_{1,2,now}]$ into the resulting suffix array and increment $r_{1,2,now}$.
iv. Otherwise, if $t_{1,2}\ mod\ 3 = 2$, do the same steps starting at 3.a except that $t_0 = T[r_{0,now} + 1]$ and $t_{1,2} = T[r_{1,2,now} + 1]$. If they are still the same, then we compare the relative positions of $t_0 + 2$ and $t_{1,2} + 2$ in $R_{1,2}$.

4. If $r_{0,now}$ is less than the length of R_0, append the slice $R_0[r_{0,now}..]$ to the suffix array.
5. If $r_{1,2,now}$ is less than the length of $R_{1,2}$, append the slice $R_{1,2}[r_{1,2,now}..]$ to the suffix array.

Continuing the previous example, we have the following orders:

$R_0 = [9, 6, 3, 0]$
$R_{1,2} = [10, 8, 7, 5, 1, 4, 2]$

For reference, we will also be presenting the original string T…

M	b	a	n	a	n	a	a	a	!	$^{N}{}_{L}$			
T	98	97	110	97	110	97	97	97	33	0	0	0	0
idx	0	1	2	3	4	5	6	7	8	9	10	11	0

… and the sorted $R_{1,2,idx}$ according to the radix sort applied on $R_{1,2,str}$.

R12 str index sorted	10	8	7	5	1	4	2

The excess index is located as the first element of $R_{1,2}$, so we ignore it and set the merging pointer $r_{1,2,now}$ to its second element. Then, we proceed with the merging as follows:

i	R0	R12	T[r0] vs T[r12]	r12 mod 3	r0 + 1 vs r12 + 1 index in R12	T[r0 + 1] vs T[r12 + 1]	r0 +2 vs r12 + 2 index in R12	Suffix Array
0	9 <- 6 3 0	10 8 <- 7 5 1 4 2	0 vs 33					9
1	9 6 <- 3 0	10 8 <- 7 5 1 4 2	97 vs 33					9 8
2	9 6 <- 3 0	10 8 7 <- 5 1 4 2	97 vs 97	1	2 vs 1			9 8 7
3	9 6 <- 3 0	10 8 7 5 <- 1 4 2	97 vs 97	2		97 vs 97	1 vs 2	9 8 7 6
4	9 6 3 <- 0	10 8 7 5 <- 1 4 2	97 vs 97	2		110 vs 97		9 8 7 6 5
5	9 6 3 <- 0	10 8 7 5 1 <-	97 vs 97	1	5 vs 6			9 8 7 6 5

		4 2						3
5	9 6 3 0 <-	10 8 7 5 1 <- 4 2	98 vs 97					9 8 7 6 5 3 1
6	9 6 3 0 <-	10 8 7 5 1 4 <- 2	98 vs 110					9 8 7 6 5 3 1 0
F	9 6 3 0 <-	10 8 7 5 1 4 <- 2						9 8 7 6 5 3 1 0 4 2

At iteration 0, the values of T at indices 9 (R_0) and 8 ($R_{1,2}$) are 0 and 33, respectively. Since these values are *not* equal, we can push the index whose corresponding value in T is smaller (8, belonging to $R_{1,2}$) to the suffix array.

At iteration 2, the values of T at indices 6 (R_0) and 7 ($R_{1,2}$) are the same (97), so we have to compare the next indices 7 and 8. Since the original index we are comparing from $R_{1,2}$ is 7 and $7\ mod\ 3 = 1$, we can compare the positions of the next indices 7 and 8 against their positions in $R_{1,2}$, which are at 2 and 1, respectively. Now, we can push the index whose corresponding position in $R_{1,2}$ is smaller, which is 7 in this case.

At iteration 3, the values of T at indices 6 (R_0) and 5 ($R_{1,2}$) are the same (97), so we have to compare the next indices 7 and 6. Since the original index we are comparing from $R_{1,2}$ is 5 and $5\ mod\ 3 = 2$, we have to compare the correspondings values of T at these next indices. It just happened that the values of T at indices 7 and 6 are the same (97) too! So, we

compare the next indices 8 and 7 against their positions in $R_{1,2}$, which are 1 and 2, respectively. Now, we can push the index whose corresponding position in $R_{1,2}$ is smaller, which is 6 in this case.

At iteration 4, the values of T at indices 3 (R_0) and 5 ($R_{1,2}$) are the same (97), so we have to compare the next indices 4 and 6. Since the original index we are comparing from $R_{1,2}$ is 5 and $5 \; mod \; 3 = 2$, we have to compare the correspondings values of T at these next indices, which are 110 and 97, respectively. Since these values are *not* equal, we can push the index whose corresponding value in T is smaller (5, belonging to $R_{1,2}$) to the suffix array.

The final suffix array of the original message is, therefore:

$$V = [9, 8, 7, 6, 5, 3, 1, 0, 4, 2]$$

Suffix Array and Burrows-Wheeler Transform

The Burrows-Wheeler transform of a string can be trivially derived from its suffix array. The algorithm is as follows:

1. Get the original string M and its suffix array V.
2. Subtract each value in V by one.
3. For each element v in V, find the corresponding vth character in M and push them into an array $V"$. If v is negative, then get the $(-v-1)$th character from the end of M. Note that positions are zero-indexed.
4. $V"$ now contains the transformed string.

As an example, we have the following message M and its corresponding suffix array V.

$$M = bananaaa!$$
$$V = [9, 8, 7, 6, 5, 3, 1, 0, 4, 2]$$

We subtract each element in V by one and map the resulting elements against the character at those positions in M, noting that negative indices are interpreted as the position from the end of M. The resulting array of characters, and conversely the Burrows-Wheeler Transform of the message M, is therefore:

$$V" = !aaannbaa$$

Recursive DC3 Algorithm

The previous example shows the DC3 algorithm *without recursion* during finding of the $R_{1,2}$ order. For another example that has recursion, please see the example document for it.

Annex E: Move-To-Front Transform

The second algorithm is the *move-to-front transform*, which is a simple transform relying on the fact that the characters appearing the most in a text are "used" more frequently. If a text is read from left to right, it makes sense to move these characters at the beginning once they appear.

Encoding

The transformation algorithm is as follows:

1. Initialize a list of characters used in the text called the *alphabet* Σ. The characters in it should have been pre-sorted in alphabetical order.
2. Initialize the output queue S where the transformed string will be placed.
3. Get a message M and do the following $|M|$ times, for each character m_i in M
    a. Find the index *from zero* of m_i in Σ. Push this index into S.
    b. Remove that same character from Σ and reinsert it at the front of Σ.
4. The transformed text is given as a series of numbers contained in S.

As an example, we have a string M:

$$M = bananaaa!^{\text{NUL}}$$

We let $^{\text{NUL}}$ be the *sentinel character* as described in Annex A (BWT). Getting the unique characters and sorting them in ascending ASCII value will yield the ordered set alphabet Σ:

$$\Sigma = \{^{\text{NUL}}, !, a, b, n\}$$

Now, we let the ordered set S contain the transformed message M. We start with the first character in M and see that it is the 3rd character (starting from zero) in Σ. S will now currently have the value $S = \{3\}$ and Σ will be changed such that this third character is moved at the very front of so that the new Σ becomes $\Sigma = \{b, ^{\text{NUL}}, !, a, n\}$.

Going through each character of M, we get the following progression:

Iteration Index i	**Letter in Message** M_i	**Current Alphabet** Σ_i	**Transformed Message** S	**New Alphabet** Σ_{i+1}
0	b	$^{\text{NUL}}$, !, a, b, n	3	b, $^{\text{NUL}}$, !, a, n
1	a	b, $^{\text{NUL}}$, !, a, n	3, 3	a, b, $^{\text{NUL}}$, !, n
2	n	a, b, $^{\text{NUL}}$, !, n	3, 3, 4	n, a, b, $^{\text{NUL}}$, !
3	a	n, a, b, $^{\text{NUL}}$, !	3, 3, 4, 1	a, n, b, $^{\text{NUL}}$, !

4	n	a, n, b, $^{N}_{U_L}$, !	3, 3, 4, 1, 1	n, a, b, $^{N}_{U_L}$, !
5	a	n, a, b, $^{N}_{U_L}$, !	3, 3, 4, 1, 1, 1	a, n, b, $^{N}_{U_L}$, !
6	a	a, n, b, $^{N}_{U_L}$, !	3, 3, 4, 1, 1, 1, 0	a, n, b, $^{N}_{U_L}$, !
7	a	a, n, b, $^{N}_{U_L}$, !	3, 3, 4, 1, 1, 1, 0, 0	a, n, b, $^{N}_{U_L}$, !
8	!	a, n, b, $^{N}_{U_L}$, !	3, 3, 4, 1, 1, 1, 0, 0, 4	!, a, n, b, $^{N}_{U_L}$
9	$^{N}_{U_L}$	!, a, n, b, $^{N}_{U_L}$	3, 3, 4, 1, 1, 1, 0, 0, 4, 4	$^{N}_{U_L}$, !, a, n, b

The transformed message is therefore encoded as the sequence:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Decoding

The transform is reversible using the following algorithm:

1. Find a way to get a hold of the original pre-sorted alphabet Σ used in the transformation algorithm above.
2. Initialize the output queue M where the transformed string will be placed.
3. Get an array of numbers S and do the following $|S|$ times, for each value s_i in S
    a. Get the character in Σ corresponding to the index s_i. Push this character into M.
    b. Remove that same character from Σ and reinsert it at the front of Σ.
4. The transformed text is given as a series of characters contained in M.

As an example, we have the following sequence encoded using move-to-front transform:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

The original alphabet was also reconstructed or received as:

$$\Sigma = \{^{N}_{U_L}, !, a, b, n\}$$

Now, we let M contain the reconstructed message from S. We start with the first value in S, treating it as an index, and looking up the corresponding character at that index (starting from zero) in Σ. M will now currently have the value $M = b$ and Σ will be changed such that this third character is moved at the very front of so that the new Σ becomes $\Sigma = \{b, ^{N}_{U_L}, !, a, n\}$.

Going through each value in S, we get the following progression:

Iteration Index i	Value in Sequence S_i	Current Alphabet Σ_i	Reconstructed Message M	New Alphabet Σ_{i+1}
0	3	^{NUL}, !, a, b, n	b	b, ^{NUL}, !, a, n
1	3	b, ^{NUL}, !, a, n	ba	a, b, ^{NUL}, !, n
2	4	a, b, ^{NUL}, !, n	ban	n, a, b, ^{NUL}, !
3	1	n, a, b, ^{NUL}, !	bana	a, n, b, ^{NUL}, !
4	1	a, n, b, ^{NUL}, !	banan	n, a, b, ^{NUL}, !
5	1	n, a, b, ^{NUL}, !	banana	a, n, b, ^{NUL}, !
6	0	a, n, b, ^{NUL}, !	bananaa	a, n, b, ^{NUL}, !
7	0	a, n, b, ^{NUL}, !	bananaaa	a, n, b, ^{NUL}, !
8	4	a, n, b, ^{NUL}, !	bananaaa!	!, a, n, b, ^{NUL}
9	4	!, a, n, b, ^{NUL}	bananaaa!^{NUL}	^{NUL}, !, a, n, b

The reconstructed message is therefore:

$$M = bananaaa!^{NUL}$$

MTF in diropqlz

When implementing this as part of diropqlz compression, the initial alphabet Σ used should be the seven letters sorted in ascending order of their ASCII values $\Sigma = \{d, i, l, o, p, q, r\}$.

Annex F: Run-Length Encoding

The third algorithm is *run-length encoding*, which is another simple transform that encodes a series of the exact same characters as the number of occurrences. Using the previous transforms, the currently compressed data (as an array of numbers) may have a long run of zeros

Encoding

We can use the following algorithm to further compress the data:

1. Initialize a counter N_{zero}, which counts the current number of zeros that will be encountered during the encoding process. Its value should initially be zero.
2. Initialize an output queue L where the encoded data will be stored.
3. Get an array of numbers S and do the following $|S| + 1$ times, for each value s_i in S
    a. Check whether the current loop index i is not greater than $|S| - 1$ and s_i is zero.
        i. If it is, increment N_{zero}.
        ii. Otherwise,
            1. Convert N_{zero} into binary and add one. Push each digit starting from the least significant bit into L *except* the most significant bit.
            2. Reset N_{zero} to zero.
            3. Finally, push $s_i + 2$ into L if s_i exists.
4. If N_{zero} is not empty, do the same steps as in 3.a.ii.

As an example, we have the following sequence S:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Now, we let L contain the encoded sequence from S and $N_{zero} = 0$. We start with the first value in S and checking whether it is a zero. Since it is not, we can already append its value *plus two* directly to L and will now currently have the value $L = \{5\}$.

Going through each value in S, we get the following progression:

Iteration Index i	**Value in Sequence** S_i	**Current number of zeros** N_{zero}	**New number of zeros** N_{zero}	**Encoded Sequence** L
0	3	0	0	5
1	3	0	0	5, 5

2	4	0	0	5, 5, 6
3	1	0	0	5, 5, 6, 3
4	1	0	0	5, 5, 6, 3, 3
5	1	0	0	5, 5, 6, 3, 3, 3
6	0	0	1	5, 5, 6, 3, 3, 3
7	0	1	2	5, 5, 6, 3, 3, 3
8	4	2	0	5, 5, 6, 3, 3, 3, 1, 6
9	4	0	0	5, 5, 6, 3, 3, 3, 1, 6, 6
10		0	0	5, 5, 6, 3, 3, 3, 1, 6, 6

At iteration index 6, note that the current value we are looking for in S is a zero. Therefore, we will not append anything to L, *but* we will increment N_{zero} by one.

At iteration index 8, note that the current value we are looking for in S is *not* a zero. In addition, N_{zero} is greater than one. Hence, we will convert $N_{zero} + 1$ to its binary form ($3_{10} = 11_2$) and append each digit starting from the rightmost digit to L *except* the most significant 1 bit. Finally, we reset the value of N_{zero} to zero. After this, don't forget to append the current value of S *plus one*!

At iteration 10, note that the sequence has already ended in iteration 9. However, this extra iteration is needed so that if N_{zero} is a nonzero value, we can append it at the end of L.

The transformed message is therefore encoded as the sequence:

$$L = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Decoding

The transform is reversible using the following algorithm:

1. Initialize a stack N_{zero}, which will contain the binary representation of the current number of zeros that were encoded during the encoding process. Its value should initially be zero.
2. Initialize an output queue S where the decoded data will be stored.
3. Get an array of numbers L and do the following $|L| + 1$ times, for each value a_i in L
    a. Check whether the current loop index i is not greater than $|L| - 1$ and whether a_i is zero or one.

i. If it is, prepend (push at start of) a_i into N_{zero}.
ii. Otherwise,
    1. Prepend a one into N_{zero}, convert it into its decimal equivalent, and subtract by one.
    2. Push the appropriate number of zeros equivalent to the new N_{zero} amount to S.
    3. Reset N_{zero} to zero.
    4. Finally, push $a_i - 2$ into L if a_i exists.
4. If N_{zero} is not empty, do the same steps as in 3.a.ii.

As an example, we have the following sequence L encoded using run-length encoding:

$$L = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Now, we let S contain the reconstructed sequence from L and $N_{zero} = 0$. We start with the first value in L and checking whether it is greater than one. Since it is, we can already append its value *minus two* directly to S and will now currently have the value $S = \{3\}$.

Going through each value in L, we get the following progression:

Iteration Index i	**Value in Sequence** L_i	**Current binary digits of** N_{zero}	**Current decimal number of zeros** $N_{zero} + 1$	**New binary digits of** N_{zero}	**Reconstructed Sequence** S
0	5	0	0	0	3
1	5	0	0	0	3, 3
2	6	0	0	0	3, 3, 4
3	3	0	0	0	3, 3, 4, 1
4	3	0	0	0	3, 3, 4, 1, 1
5	3	0	0	0	3, 3, 4, 1, 1, 1
6	1	0	0	1	3, 3, 4, 1, 1, 1
7	6	1	3	0	3, 3, 4, 1, 1, 1, 0, 0, 4
8	6	0	0	0	3, 3, 4, 1, 1, 1, 0, 0, 4, 4
9		0	0	0	3, 3, 4, 1, 1, 1, 0, 0, 4, 4

At iteration index 6, note that the current value we are looking for in S is one. Therefore, we will not append anything to L, *but* we will treat this digit as a binary digit and push it into N_{zero}.

At iteration index 7, note that the current value we are looking for in S is greater than one. In addition, N_{zero} is greater than one. Hence, we will push a 1 into N_{zero}, reverse the digits and add one to get the true binary form ($3_{10} = 11_2$). We then convert it to its decimal equivalent and append $N_{zero} - 1$ zeros to S. Finally, we reset the value of N_{zero} to zero. After this, don't forget to append the current value of L *minus one*!

At iteration 9, note that the sequence has already ended in iteration 8. However, this extra iteration is needed so that if N_{zero} is a nonzero value, we can append the appropriate number of zeros at the end of S.

The reconstructed sequence is therefore:

$$S = \{3,\ 3,\ 4,\ 1,\ 1,\ 1,\ 0,\ 0,\ 4,\ 4\}$$

Annex G: Huffman Encoding

The final algorithm is *variable-length encoding*, which is a common transform that encodes all characters in a message into different binary codewords of variable lengths. Characters that more frequently show up are assigned to shorter codewords while those that rarely appear are assigned to longer codewords. A well-known variable length encoder is the *Huffman encoder*, which was published in 1952.

Huffman Tree Building

Encoding messages using Huffman encoding utilizes a greedy approach by building a binary tree based on the frequency of the characters that appear in the message. The algorithm for building the tree and encoding map is as follows:

1. Get all of the values uniquely used in the message C. Push them into a priority queue Q of nodes, with each value as its own (leaf) node, and where the frontmost element of it will be the least frequent value, breaking ties with the value itself.
2. Do the following until Q has a single element.
    a. Get the first two elements of Q.
    b. Create a new node from them with these two elements as its children. The frontmost of the two is the left child, and the other is the right child.
    c. Set the frequency of the new node to the sum of the frequencies of the two child elements.
    d. Push this new node into Q.

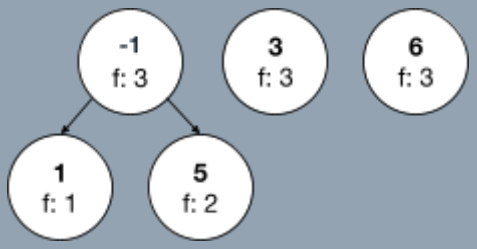
As an example, we have the following sequence C:

$$C = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

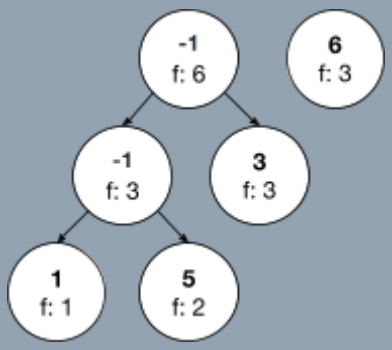
We get the unique values in C and find the frequency of each appearing in it to yield the following forest of nodes:

Next, we sort them in increasing frequency, and then increasing value from left to right. We can think of it as these forest of nodes pushed into the priority queue Q, which will automatically sort them. This will yield:

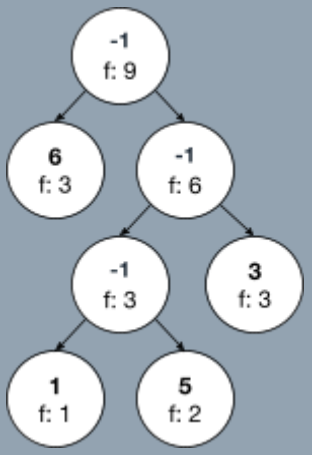
We then start by getting the two leftmost nodes and "merging" them by making a "dummy" node whose value is -1 (or any "null" value) and frequency being the sum of the frequencies of these two leftmost nodes. Then, we connect the leftmost of the two nodes to the left of this dummy node and the other one to the right.



Repeating the same process as outlined above, we sort the nodes in the uppermost row in increasing frequency, and then increasing value. Then, we merge the two leftmost nodes to yield the following arrangement:



Another repetition of the same process will yield the Huffman tree corresponding to the sequence C:

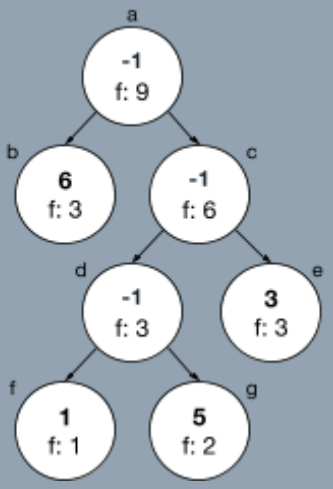


Codeword Mapping

Let the single tree T left in Q be the encoding tree that will be used to encode the message. To find the Huffman codeword for the character using T:

1. Initialize a codeword stack S where the codeword digits will be stored.
2. Push 2 into S. Note that this value will *not* be part of the binary codeword.
3. Start at the root node of T.
4. Do the following until S is empty:
    a. If the current node is not a leaf node (i.e. has a character), traverse downward to the left and push a 0 into S.
    b. If the current node contains the character, then skip to step (4).
    c. Otherwise, if we have already visited the left, but we have not yet done so to the right:
        i. Pop one digit off of S and discard it.
        ii. Traverse downward to the right and push a 1 into S.
    d. Otherwise, if we have already visited the right, too:
        i. Pop two digits off of S.
5. Remove the first element from S and it will now contain the binary codeword.

As an example, we have the following Huffman tree with each node labeled with a letter for reference:



We will traverse the tree in preorder, which will process a node recursively in this order: current, left, right. For the example tree, the nodes will be visited in this order: a, b, c, d, f, g, e.

We let the codeword stack S be empty. Starting in node a, we first check whether the value is not the "null" value. Since it is the "null" value, we will stop processing this node, push a 0 into S and go to node b.

Once we reach node f, S will have the value $S = 100_2$. Since it has a non-"null" value, we save the current value of S and the value in this node to some keyed map (e.g. a hashmap). Then, since f does not have children, we go up the tree by one level and visit the right node. This requires popping off the rightmost digit in S and pushing one to it.

Going through each node in the prescribed visitation order, we get the following progression:

Iteration Index i	**Currently Visited Node**	**Current Codeword Digits in S**	**Current Value in Node**	**Mapped Binary Codeword**
0	a	2	-1	
1	b	2, 0	6	0
2	c	2, 1	-1	
3	d	2, 1, 0	-1	
4	f	2, 1, 0, 0	1	100
5	g	2, 1, 0, 1	5	101
6	e	2, 1. 1	3	11

The value to binary codeword mappings are as follows:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

Encoding

Encoding a message is now straightforward by just mapping the characters to the corresponding codewords.

As an example, we have the following sequence C:

$$C = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

The value to binary codeword mappings derived from its Huffman tree are as follows:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

The final encoded sequence of bits corresponding to C is therefore:

101 101 0 11 11 11 100 0 0

Decoding

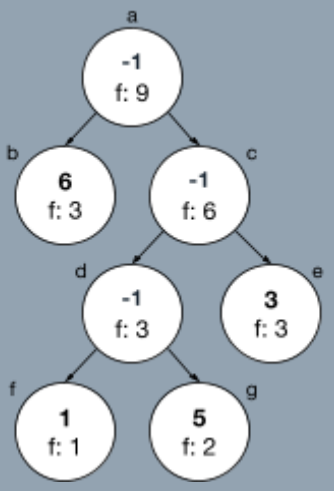
On the other hand, the following algorithm can be used to decode a message encoded using Huffman encoding:

1. Get the tree T associated with encoding the message M.
2. Let t_{idx} be a pointer to the current node in T, and Q be the queue that will contain the decoded message.
3. Let t_{idx} point to the root node of T.
4. Do the following $|M| + 1$ times, with index i:
    a. If t_{idx} points to a leaf node (i.e. has a character):
        i. Push the character onto Q.
        ii. Reset t_{idx} to point it to the root node of T.
    b. Otherwise, if i is less than $|M|$.
        i. Let c be the i_{th} character in M.
        ii. If c is a zero, move t_{idx} to the left.
        iii. Otherwise, if c is a one, move t_{idx} to the right.

As an example, we have the following sequence of bits M encoded using Huffman encoding:

101101011111110000

Additionally, we have reconstructed or received the following Huffman tree corresponding to this sequence of bits with each node labeled with a letter for reference:



Going through each bit in M, we get the following progression:

Iteration Index i	Current Bit M_i	Current Node in T	Next Node	Value in Next Node	Reset Current Node?	Reconstructed Message Q
0	1	a	c	-1	No	
1	0	c	d	-1	No	
2	1	d	g	5	Yes	5
3	1	a	c	-1	No	5
4	0	c	d	-1	No	5
5	1	d	g	5	Yes	5, 5
6	0	a	b	6	Yes	5, 5, 6
7	1	a	c	-1	No	5, 5, 6
8	1	c	e	3	Yes	5, 5, 6, 3
9	1	a	c	-1	No	5, 5, 6, 3
10	1	c	e	3	Yes	5, 5, 6, 3, 3
11	1	a	c	-1	No	5, 5, 6, 3, 3
12	1	c	e	3	Yes	5, 5, 6, 3, 3, 3
13	1	a	c	-1	No	5, 5, 6, 3, 3, 3
14	0	c	d	-1	No	5, 5, 6, 3, 3, 3
15	0	d	f	1	Yes	5, 5, 6, 3, 3, 3, 1
16	0	a	b	6	Yes	5, 5, 6, 3, 3, 3, 1, 6
17	0	a	b	6	Yes	5, 5, 6, 3, 3, 3, 1, 6, 6

We start by letting the current node be the root node a. Since the current node has a "null" value, we then look at the first bit and traverse the tree to the left if the bit is zero, or to the right otherwise.

At iteration 2, we note that the current node has a non-"null" value. So we push this value into Q and reset the current node to the root node a.

The reconstructed sequence is therefore:

$$Q = \{5, 5, 6, 3, 3, 3, 1, 6, 6\}$$

Codebook Canonicalization

The codeword mapping, or codebook, can be stored as a 16-bit pair of binary codeword and character respectively. However, if the alphabet used is known at the decoding side, we can change the binary codewords into something such that we only need to encode the *lengths* of the codewords.

To canonicalize a codebook for such mapping:

1. Sort the mappings by increasing codeword length, and then by ascending mapped values.
2. Let the codeword of the first mapping be equal to a string of zeros of the same length as the codeword. Let this length be equal to L_{canon}.
3. Initialize a value $C_{canon} = 0$ storing the current value of the codeword to assign.
4. For each subsequent mapping in the sorted list $C_{old,i}$:
    a. Increment C_{canon} by one.
    b. Check whether the codeword length of $C_{old,i}$ is greater than L_{canon}.
        i. If it is, then left shift the bits of C_{canon} until C_{canon} now has the same length as $C_{old,i}$. Also set L_{canon} to be equal to the length of $C_{old,i}$.
        ii. Otherwise, replace $C_{old,i}$ with C_{canon} while maintaining the old codeword length of $C_{old,i}$.

As an example, we have the following non-canonical Huffman codebook:

- 1 - 100
- 3 - 11
- 5 - 101
- 6 - 0

We sort the codebook entries to yield:

- 6 - 0
- 3 - 11
- 1 - 100
- 5 - 101

Going through each value in the codebook will yield the following progression:

Value	**Old Codeword**	**Current L_{canon} Length**	**Current C_{canon} Bin Value**	**Left Shift Amount**	**New Codeword**
6	0	1	0	0	0

3	11	1	0	1	10
1	100	2	10	1	110
5	101	3	110	0	111

The new canonical codebook corresponding to the old one is therefore:

- 6 - 0
- 3 - 10
- 1 - 110
- 5 - 111

To encode the codebook for transmission, we first sort the codebook entries by ascending mapped values. Using the canonical codebook derived earlier, the newly-sorted codebook is as follows:

- 1 - 110
- 3 - 10
- 5 - 111
- 6 - 0

Assuming that the original message consists of the mapped values $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, the codebook will be transmitted as the following sequence of values:

$$C_{canon,len} = \{0,\ 3,\ 0,\ 2,\ 0,\ 3,\ 1,\ 0,\ 0,\ 0\}$$

This can be interpreted as such: the length of the codeword at the ith element of Σ is the corresponding value at $C_{canon,len}$. So, the first element in Σ has a codeword of length $C_{canon,len}$. Note that the other mapped values that were *not* included in the final codebook are encoded as having a codeword of length zero.

To reconstruct the codebook from the canonical codeword lengths:

1. Obtain the original sorted alphabet Σ used during codebook canonicalization.
2. Create a mapping $\Sigma \rightarrow C_{canon,len}$ of the mapped values and the codeword lengths.
3. Sort the map entries in increasing codeword length, and then by ascending mapped values.
4. Let the codeword of the first mapping be equal to a string of zeros of the specified codeword length. Let this length be equal to L_{canon}.
5. Initialize a value $C_{canon} = 0$ storing the current value of the codeword to assign.

6. For each subsequent mapping in the sorted list $C_{old,i}$:
    a. Increment C_{canon} by one.
    b. Check whether the codeword length of $C_{old,i}$ is greater than L_{canon}.
        i. If it is, then left shift the bits of C_{canon} until C_{canon} now has the same length as $C_{old,i}$. Also set L_{canon} to be equal to the length of $C_{old,i}$.
        ii. Otherwise, replace $C_{old,i}$ with C_{canon} while maintaining the old codeword length of $C_{old,i}$.

As an example, we have the sequence of codeword bit lengths:

$$C_{canon,len} = \{0,\ 3,\ 0,\ 2,\ 0,\ 3,\ 1,\ 0,\ 0,\ 0\}$$

Additionally, we are also given the alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ corresponding to these codeword lengths. Mapping these two and removing the values in Σ whose $C_{canon,len}$ lengths are zero yields the following mapping. Note that it is already sorted according to increasing codeword length and then by ascending value:

- 6 - 1
- 3 - 2
- 1 - 3
- 5 - 3

Going through each value in the mapping will yield the following progression:

Value	**Current L_{canon} Length**	**Current C_{canon} Bin Value**	**Left Shift Amount**	**Derived Codeword**
6	1	0	0	0
3	2	0	1	10
1	3	10	1	110
5	3	110	0	111

The reconstructed canonical codebook is therefore:

- 6 - 0
- 3 - 10
- 1 - 110
- 5 - 111

Huffman Encoding in diropqlz

Note that as part of diropqlz compression, the Huffman decoder will receive an array of nonnegative integer values ranging from 0 until 9 (the seven commands of diropql plus two binary digits for the run-length encoding) corresponding to the bit lengths of the canonicalized codebook against some alphabet. Hence, for this purpose, the initial alphabet Σ used should be the integers between 0 and 9 inclusive sorted in ascending order, or $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.