BZIP2: Format Specification

Revised: 2016-03-17
Author: Joe Tsai ⟨joetsai@digital-static.net⟩
Website: http://digital-static.net

1 Introduction

Here we present a specification for the BZip2 format, as there does not exist an official document for the format. This is **not an official specification** and presents the reverse engineering work of Joe Tsai. Anything stated here should be fact verified according to the original source. The reference C implementation used to derive this work is bzip2-1.0.6. Other implementations of BZip2 may have been used to also corroborate the information here.

2 Specification

2.1 Bit-packing order

Contrary to DEFLATE (RFC 1951), BZip2 packs bits with the leading bits in the stream as the most significant bits in a byte. Furthermore, when writing a group of bits to the bit-stream, the first bits written are the most-significant bits of the integer, rather than the least-significant bits.

As an example, let us decode the given list of bytes as some list of integers with varying bit-lengths:

1. {0xee 0x5c 0x75} Example data
2. {11101110 01011100 01110101} Convert bytes to bits (LSB on right)
3. {1110 111001 011 100 0 1 110101} Group bits according to each integer bit-length
4. {14 57 3 4 0 1 43} Convert bit-strings to integer values (LSB on right)

The sequence of bits in step 2 or 3 represents what a BZip2 file would appear as when written out as a bit-stream with the leading bits to the left and the trailing bits to the right. For the remainder of the document, when the text says, "read n bits" or "write n bits", it is according to the convention listed here; this applies even if n is a multiple of 8 bits. Lastly, any group of bits read is always treated as an unsigned integer.

2.2 Stream format

In the format specifications below, we use regular expression-like semantics to describe the structure. As with the POSIX standard for regular expressions, we use the following operators:

- Grouping: `()`
- Alternatives: `|`
- Quantification: `*` `+` `?` `{n}` `{n,m}`

To describe the structure of the BZip2 format, a grammar similar to Backus-Naur Form will be used to describe each element. In the grammar below, the symbol colors have the following meanings:

- `Black`: Represents some other symbol
- `Orange`: Represents values with a fixed bit or byte length
- `Green`: Represents values encoded with Huffman encoding
- `Blue`: Represents values encoded using either MTF or Delta encoding

The BZip2 format is as follows:

Symbol		Expression
BZipFile	:=	BZipStream+
└──**BZipStream**	:=	StreamHeader StreamBlock* StreamFooter
├──**StreamHeader**	:=	HeaderMagic Version Level
├──**StreamBlock**	:=	BlockHeader BlockTrees BlockData
│ ├──**BlockHeader**	:=	BlockMagic BlockCRC Randomized OrigPtr
│ └──**BlockTrees**	:=	SymMap NumTrees NumSels Selectors Trees
│ ├──**SymMap**	:=	MapL1 MapL2{1,16}
│ ├──**Selectors**	:=	Selector{NumSels}
│ └──**Trees**	:=	(BitLen Delta{NumSyms}){NumTrees}
└──**StreamFooter**	:=	FooterMagic StreamCRC Padding

The highest-level element for BZip2 is the `BZipFile`, which is comprised of one-or-more `BZipStreams`. The uncompressed output of a `BZipFile` is the logical concatenation of the uncompressed output of all `BZipStreams` within that file. The uncompressed output of a `BZipStream` is the logical concatenation of the uncompressed output of all the `StreamBlocks` within that stream. The uncompressed output of a `StreamBlock` will be discussed in detail in the following sections. It is important to note that none of the fields within a `StreamBlock` or `StreamFooter` are necessarily byte-aligned. Thus, the bit-packing rules described in section 2.1 must be followed for all integer fields.

2.2.1 BlockData

Before discussing the various components of the BZip2 format, it is most appropriate to describe the compression stack that forms the essence of the format. The core of the compressed data is stored within the `BlockData` element within each stream block.

When compressing data, the uncompressed input data goes through a series of stages. In BZip2, there are 5 stages: **RLE1**, **BWT**, **MTF**, **RLE2**, and **HUFF**. Each stage transforms the input data in some way (hopefully reducing the size). Before each stage is described in detail below, a short summary of what effects the stage has is shown. The "changes data size" field indicates whether the transformation may cause the input and output sizes to differ. The "produces metadata" field indicates whether some other auxiliary information is produced by the stage that needs to be stored outside of `BlockData`.

2.2.1.1 Run-length encoding (RLE1)

Input type	`[]byte`	**Changes data size**	Yes
Output type	`[]byte`	**Produces metadata**	No

The **RLE1** stage replaces repeated runs of the same byte with a shorter string. Every sequence of 4..255 duplicated bytes is replaced by only the first 4 bytes and a single byte representing the repeat-count. For example, the string `"AAAAAAABBBBCCCD"` will be encoded as `"AAAA\x03BBBB\x00CCCD"`. Even if a run only has 4 bytes, a repeat-count of zero is still required. Thus, `"BBAAAA"` is not a valid output, even if it is the last sequence of bytes at the end of buffer. Instead, it must be encoded as `"BBAAAA\x00"`. Since the output buffer is bounded in size, an encoder implementation must be careful to ensure that it can write the trailing repeat-count if the final 4 input bytes are duplicates.

Also, since the repeat-count can represent a length of up to 255 bytes, duplicate symbols up to 259 bytes in length may be replaced. The reference C implementation never replaces more than 255 bytes at

a time, but the decompressor is capable of handling repeat sizes between 256 and 259 bytes. A decoder implementation should be able to handle these count values as well. Thus, `"AAAA\xff"` should expand to be a string with 259x `A`'s. For maximal compatibility, it is recommended that an encoder also avoid replacing more than 255 bytes at a time. Lastly, there is no requirement that an optimal encoding be used. Thus, `"AAAA\x00AAAA\x00"` or `"AAAA\x01AAA"` may be used rather than `"AAAA\x04"` to represent the string `"AAAAAAAA"`.

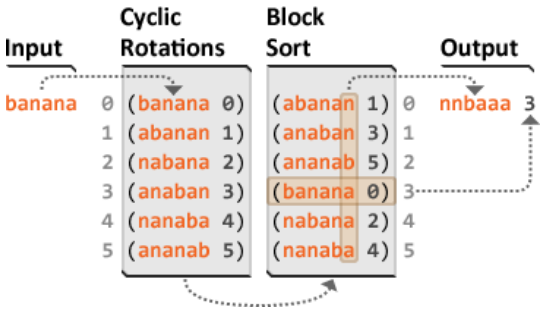
In the worst-case, this stage may expand the data size by $1.25x$, and in the best-case, this stage may reduce the data size by $51.8x$.

2.2.1.2 Burrows-Wheeler transform (BWT)

Input type	`[]byte`	**Changes data size**	No
Output type	`[]byte`	**Produces metadata**	Yes

The **BWT** stage performs the Burrows-Wheeler transform on the input data. Also known as block-sorting, the transform is described in detail in A Block-sorting Lossless Data Compression Algorithm by M. Burrows and D. J. Wheeler. Functionally speaking, the BWT is a permutation of the input data, with the special property that the permutation is reversible given an "origin pointer".

The BWT is done by creating a square matrix, where each row is the cyclic-rotation of the input string. This block is then sorted lexicographically. After sorting, the last column is used as the output string. In order to be a reversible permutation, an "origin pointer" is needed that allows the decoder to know which row in the reconstructed block was the original input string. It is important to note that the variant of BWT used in BZip2 does not terminate strings with a special symbol that is lexicographically smaller than all other symbols. Instead, strings wrap around as is.



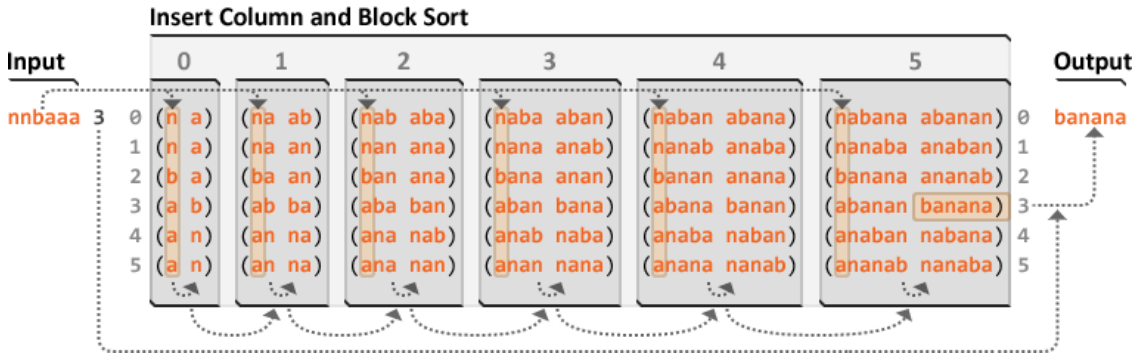
Example of forward BWT on the string `"banana"`

An algorithm to naïvely perform the forward BWT is shown in the Python code below:

```
def EncodeBWT(data):
    matrix = [None] * len(data)
    for i in range(len(data)):
        matrix[i] = (data[-i:] + data[:-i], i)
    matrix.sort()
    bwt = bytearray([x[-1] for x, _ in matrix])
    ptr = matrix.index((data, 0))
    return bwt, ptr
```

The input string is passed in as a `bytearray`, and the function returns the BWT'd string as another `bytearray` along with the origin pointer as an integer. This implementation runs in $O(n^2 \log(n))$ and is not suitable for real applications of BZip2.

The inverse BWT process works by executing n "insert and sort" operations until the original sorted matrix is reconstructed. Starting with an empty matrix, the insert phase prepends the input string as a column in the matrix, while the sort phase performs the block-sort on the current matrix. After n operations where n is the length of the input, the original input string will be the ith row in the matrix where i is the "origin pointer".



Example of inverse BWT on the string `"nnbaa"`

An algorithm to perform the inverse BWT is shown in the Python code below:

```
def DecodeBWT(bwt, ptr):
    matrix = [bytearray(len(bwt)) for _ in bwt]
    for ci in reversed(range(len(bwt))):
        for ri, b in enumerate(bwt):
            matrix[ri][ci] = b
        matrix.sort()
    data = matrix[ptr]
    return data
```

The shuffled input is passed in as a `bytearray` in `bwt`, while the origin pointer is an integer in `ptr`. The return value is a `bytearray` containing the original input string. This implementation runs in $O(n^3 \log(n))$ and is not suitable for real applications of BZip2.

Next, we describe an efficient way to perform the forward BWT by using suffix arrays. The suffix array is an array of integers that represents the lexicographically sorted list of all suffixes of the input string. It is well known that there is a direct relationship between suffix arrays and BWTs such that the BWT can be derived from a suffix array in $O(n)$ runtime. Furthermore, there exist many modern suffix array construction algorithms (SACAs) such as SAIS, which runs in $O(n)$, and DivSufSort, which runs in $O(n \log n)$. The SACA used is beyond the scope of this document.

If the input string terminates with a special symbol that is lexicographically smaller than all other symbols, then the BWT is related to suffix arrays as $BWT[i] = D[SA[i] - 1]$, where D is the input string and SA is the suffix array. However, this is not directly helpful for the BWT used in BZip2 since the specific variant used does *not* terminate strings, but wraps them around as is. Instead, we need a version of suffix arrays where the suffixes do wrap around and do not terminate.

Below we demonstrate how to convert a conventional suffix array provided by the function `ComputeSA` to one that uses wrapped suffixes via the function `ComputeWrapSA`. The exact implementation of `ComputeSA` is not provided.

```
def ComputeWrapSA(data):
    data2 = data*2
    idxs = ComputeSA(data2)
    idxs = [x for x in idxs if x < len(data)]
    return idxs
```

The methodology (reference) shown effectively duplicates the input string (emulating wrap-around semantics), computes the suffix array of the duplicated string, and then extracts all indexes that are lower than the original input string size. While this method is able to use a conventional SACA as is, it does run 2x slower. It is possible to directly modify the SACA to use wrap-around semantics, but that is beyond the scope of this document.

Using `ComputeWrapSA`, we can now compute the BWT using the equation given above as shown in the following Python code:

```
def EncodeBWT(data):
    idxs = ComputeWrapSA(data)
    bwt, ptr = bytearray(len(data)), 0
    for j, i in enumerate(idxs):
        if i == 0:
            ptr = j
            i = len(data)
        bwt[j] = data[i-1]
    return bwt, ptr
```

The runtime of this new implementation of `EncodeBWT` runs as fast as `ComputeSA`, and can be significantly faster than the original function provided.

Lastly, we show how to perform the inverse BWT in $O(n)$ runtime based on the logic in the C implementation and the BWT paper.

```
def DecodeBWT(bwt, ptr):
    cumm, n = [0]*256, 0
    for v in bwt:
        cumm[v] += 1
    for i, v in enumerate(cumm):
        cumm[i] = n
        n += v

    perm = [0]*len(bwt)
    for i, v in enumerate(bwt):
        perm[cumm[v]] = i
        cumm[v] += 1

    i = perm[ptr]
    data = bytearray(len(bwt))
    for j in range(len(bwt)):
        data[j] = bwt[i]
        i = perm[i]
    return data
```

The runtime of this new implementation of `DecodeBWT` is $O(n)$, which is a significant improvement over the original function provided. Some implementations compute the `cumm` histogram as part of the **MTF** stage and perform the de-shuffling portion as part of the **RLE1** stage to improve performance.

Here are some examples:

```
>>> data = bytearray("she sells seashells by the seashore")
>>> ComputeSA = lambda x: None if (x != data*2) else [
...     54, 19, 44, 9, 61, 26, 38, 3, 57, 22, 47, 12, 64, 29, 55, 20, 69, 60,
...     25, 37, 2, 46, 11, 63, 28, 50, 15, 40, 5, 34, 59, 24, 36, 1, 49, 14,
...     66, 31, 51, 16, 41, 6, 52, 17, 42, 7, 67, 32, 68, 33, 53, 18, 43, 8,
...     45, 10, 62, 27, 39, 4, 35, 0, 48, 13, 65, 30, 58, 23, 56, 21,
... ]

>>> bwt, ptr = EncodeBWT(data)
>>> bwt, ptr
bytearray("sseeyee hhsshsrtssseellholl   eaa b"), 30

>>> buf = DecodeBWT(bwt, ptr)
>>> buf
bytearray("she sells seashells by the seashore")
```

There are some things to note: For demonstration purposes, a one-time "implementation" of `ComputeSA` is provided that simply returns the suffix array of the pre-determined input string. Also, we can see that the BWT'd output tends to contain clusters of duplicated symbols, which is helpful for compression. This is the primary purpose of using the BWT in BZip2. The stages to follow are used to reduce this repetitiveness to a more compact representation.

Metadata produced:

- BWT origin pointer

2.2.1.3 Move-to-front transform (MTF)

Input type	[]byte
Output type	[]uint8

Changes data size	No
Produces metadata	Yes

The **MTF** stage performs the move-to-front transform. The intuition behind the transform is to replace every symbol with the index of that symbol in some "stack". Every time a symbol is used, it is moved to the front of the stack (such that it obtains a lower index value). Thus, commonly used symbols will be assigned lower index values and runs of the same symbol will be replaced by zeros. This stage is effective because the prior **BWT** stage tends to create clusters of duplicate symbols. Thus, **MTF** will transform those duplicate symbols to be zeros.

The algorithm to perform the forward move-to-front transform is shown in the Python code below:

```
def EncodeMTF(syms, stack):
    idxs = []
    for s in syms:
        i = stack.index(s)
        stack = [stack.pop(i)] + stack
        idxs.append(i)
    return idxs
```

The `syms` variable must be a `bytearray`, while `stack` is a sorted list of all the unique symbols that appear in `syms`. The function returns a list of integers representing the indexes to the corresponding symbol on the stack. This algorithm runs in $O(n)$ time where n is the length of `syms`. In the worst case, the symbol being looked up is always at the rear of the stack. However, since the stack is bounded in size, the runtime is still linear.

The algorithm to perform the inverse move-to-front transform is shown in the Python code below:

```
def DecodeMTF(idxs, stack):
    syms = []
    for i in idxs:
        s = stack[i]
        stack = [stack.pop(i)] + stack
        syms.append(s)
    return bytearray(syms)
```

The `idxs` variable must be a list of integers, while `stack` is a sorted list of symbols as used in the encode function. The function returns the original string as a `bytearray`. This algorithm also runs in $O(n)$.

Here are some examples:

```
>>> data = bytearray("bbyaeeeeeeafeeeybzzzzzzzzzyz")
>>> stack = sorted({x for x in data})
>>> stack
[97, 98, 101, 102, 121, 122]

>>> idxs = EncodeMTF(data, stack[:])
>>> idxs
[1, 0, 4, 2, 3, 0, 0, 0, 0, 0, 1, 4, 2, 0,
 0, 3, 4, 5, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1]

>>> syms = DecodeMTF(idxs, stack[:])
>>> syms
bytearray("bbyaeeeeeeafeeeybzzzzzzzzzyz")
```

There are some things to note: The `stack` is generated by using set comprehension to get a set of all unique symbols used in the string before sorting it. Also, when passing the `stack` into the encode/decode functions, a copy of the stack is used by using the `[:]` notation since the functions mutates the stack.

Metadata produced:

- Symbols used in the MTF stack

2.2.1.4 Run-length encoding (RLE2)

Input type	[]uint8
Output type	[]symbol

Changes data size	Yes
Produces metadata	No

Since the **BWT** stage tends to create clusters of duplicate symbols, and the **MTF** stage tends to convert runs of the same symbol to be runs of the zero value, the **RLE2** stage is used to reduce these long runs of zero values to use shorter representations. In this stage, sequences of `0` symbols are replaced by shorter sequences of two other symbols called `RUNA` and `RUNB`.

The conversion to and from RUN symbols is bijective and can be efficiently performed with normal binary arithmetic. To describe this relationship, let us define $ZeroCnt$ as the number of `0` symbols that occur sequentially in the input. Also, let us define $RunLen$ as the number of `RUNA` and `RUNB` symbols in the corresponding numeration. Lastly, let that sequence of `RUNA` and `RUNB` symbols represent a binary integer, $RunSyms$, where `RUNA` is the 0 bit, `RUNB` is the 1 bit, and the leading RUN symbols are in the least-significant bit positions. If so, then there exists a relationship between these variables:

$$ZeroCnt + 1 == (1 << RunLen) \mid RunSyms$$

(where the << operator is a left-shift and the | operator is a bit-wise OR)

Using this equation, we can produce a table that maps zero counts to their corresponding RUN symbols. For brevity, we use symbols A and B to represent RUNA and RUNB. The following table is printed with the leading RUN symbols on the left (i.e., LSB of $RunSyms$ on the left).

Count	Run symbols		Count	Run symbols		Count	Run symbols
1	A		22	BBBA		43	AABBA
2	B		23	AAAB		44	BABBA
3	AA		24	BAAB		45	ABBBA
4	BA		25	ABAB		46	BBBBA
5	AB		26	BBAB		47	AAAAB
6	BB		27	AABB		48	BAAAB
7	AAA		28	BABB		49	ABAAB
8	BAA		29	ABBB		50	BBAAB
9	ABA		30	BBBB		51	AABAB
10	BBA		31	AAAAA		52	BABAB
11	AAB		32	BAAAA		53	ABBAB
12	BAB		33	ABAAA		54	BBBAB
13	ABB		34	BBAAA		55	AAABB
14	BBB		35	AABAA		56	BAABB
15	AAAA		36	BABAA		57	ABABB
16	BAAA		37	ABBAA		58	BBABB
17	ABAA		38	BBBAA		59	AABBB
18	BBAA		39	AAABA		60	BABBB
19	AABA		40	BAABA		61	ABBBB
20	BABA		41	ABABA		62	BBBBB
21	ABBA		42	BBABA		63	AAAAAA

This table only shows the mapping for the first 63 counts to their corresponding RUN numerations. This mapping is bijective because every positive count maps to exactly one unique sequence of RUN symbols. Given a $ZeroCnt$, the $RunSyms$ sequence can be determined. Given a $RunSyms$ sequence, the $ZeroCnt$ can be determined.

As an example, we can use this mapping to RLE transform the following:

```
Input:  [1 0 4 2 3 0 0 0 0 0 1 4 2 0 0 3 4 5 0 0 0 0 0 0 0 0 2 1]
Output: [1 A 4 2 3 A B       1 4 2 B   3 4 5 B A A           2 1]
```

For clarity, we colored the run symbols with a different color. In this example, there are 4 groups of `0` symbols with counts of [`1`, `5`, `2`, `8`]. Using the equation above, we can determine the RUN numerations used will be [`A`, `AB`, `B`, `BAA`], which we substitute in place of the `0` symbols to obtain the output. It should be noted that since **RLE2** increases the cardinality of the symbol alphabet, the number of outputted symbols is always less-than-or-equal to the number of input values.

In practice, most implementations will combine the **MTF** and **RLE2** stages.

2.2.1.5 Huffman encoding (HUFF)

Input type	[]symbol	Changes data size	Yes
Output type	[]bit	Produces metadata	Yes

In the **HUFF** stage, we encode the input list of symbols as a sequence of bits on the wire. The intuition behind Huffman encoding is to encode frequent symbols with fewer bits, while encoding less frequent symbols with more bits. In order to do this, we transform the input symbols into a set of symbols that we can Huffman encode.

First, we need to figure out the number of symbols, $NumSyms$, in the Huffman tree. $NumSyms$ is computed as $NumStack - 1 + 3$, where $NumStack$ is the number of symbols in the stack from the **MTF** stage. We subtract 1 for losing the `0` symbol in the **RLE2** stage and add 3 for gaining RUNA, RUNB, and a special EOB symbol (from this stage).

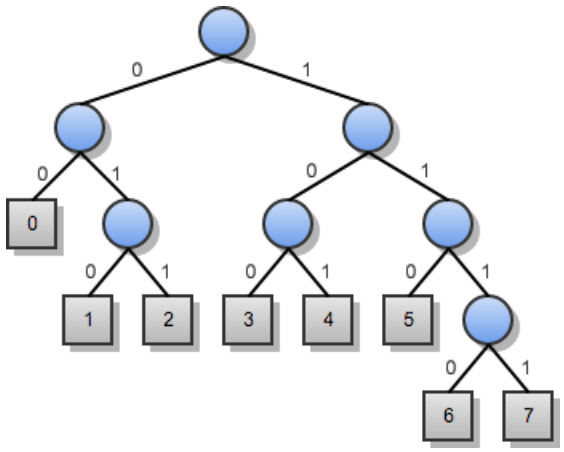
In the new symbol alphabet, RUNA is always 0, RUNB is always 1, and EOB is always $NumSyms - 1$. All other values are linearly distributed between $2..NumSyms - 2$; which functionally means all numeric symbols in the input will simply be incremented. Lastly, since RUNA and RUNB are always present, $NumSyms$ must be at least 3 to ensure the presence of EOB. The EOB symbol stands for "end-of-block". It appears exactly once in the symbols list at the end and is used by the Huffman decoder to know when to terminate.

For example, consider the following:

```
Input:   [1 A 4 2 3 A B 1 4 2 B 3 4 5 B A A 2]
Symbols: [2 0 5 3 4 0 1 2 5 3 1 4 5 6 1 0 0 3 7]
```

The input symbols have a MTF stack size of 6. We know that because the maximum numeric value in the MTF'd output will always be the highest index in the MTF stack. Thus, $NumStack$ is always $max(Input) + 1$, which means the value of EOB is 7. When the input is converted to the list of symbols to Huffman encode, we see that RUNA and RUNB are encoded as 0 and 1. All other values are simply incremented. Lastly, we see that an EOB is appended to the symbols list.

Given a symbols list, a Huffman tree can then be generated to encode the symbols to individual bits. The algorithm for generating the optimal tree is not covered here (see A Method for the Construction of Minimum-Redundancy Codes by D. A. Huffman). For the symbols list in the example above, one possible tree is the following:



Given that RUNA and RUNB symbols are the most common, they have the shortest code lengths. On the other hand, the EOB marker only occurs once and is given the longest code length. In the BZip2 format, there is a maximum limit of 20 bits for any Huffman code.

From this tree, we can encode the output bits by tracing the edges from the root node to each leaf for the given symbol being encoded:

```
Symbols: [  2  0   5   3   4  0   1   2   5   3   1   4   5    6   1  0  0   3    7]
Output:  [011 00 110 100 101 00 010 011 110 100 010 101 110 1110 010 00 00 100 1111]
```

It is important to note that BZip2 actually allows multiple Huffman trees to be used. Every group of 50 symbols can be encoded with a different tree. Thus, in order to determine which tree to use, a list of selectors must be provided. The number of elements in the selectors list, $NumSels$, can be computed as $NumSels = \left\lceil \frac{|Symbols|}{50} \right\rceil$, where $|Symbols|$ is the length of the symbols list generated above.

The selectors list simply contains the index of which Huffman tree to use. For example, if there were 243 symbols to encode (including the EOB symbol), then we know that we would need 5 selectors. One possible selectors list could be: `[0, 0, 1, 2, 1]`. In this situation, we would use tree 0 for the first 100 symbols, tree 1 for the next 50 symbols, tree 2 for the next 50 symbols, and tree 1 for the last 43 symbols. For this given selector list, there must be least 3 Huffman trees.

The BZip2 format allows between 2 to 6 Huffman trees to be used. Choosing the optimal number of trees to use and selecting which tree to use for each group of symbols is an exercise for the encoder implementation. Such design considerations are beyond the scope of this document.

Metadata produced:

- The definition for each Huffman tree
- The selectors used for each group

2.2.2 StreamHeader

Now that we have described the core compression pipeline of BZip2, we can now describe the format structure, starting with the stream header, which follows the following grammar:

```
StreamHeader := HeaderMagic Version Level
```

The stream header is guaranteed to be byte-aligned, and its values can be read as regular bytes from the underlying stream. The `HeaderMagic` is the 2 byte string: `[]byte{'B', 'Z'}`, or also the value `0x425a` when read as a 16-bit integer. The `Version` is always the byte `'h'`, or also the value `0x68` when read as an 8-bit integer. The BZip1 format used a version byte of `'0'`; that format is deprecated and is not discussed in this document.

The `Level` is a single byte between the values of `'1'` and `'9'`, inclusively. This ASCII character, when parsed as an integer and multiplied by 100000, determines the size of the buffer used in the **BWT** stage. Thus, level `'1'` uses a 100000 byte buffer, while level `'9'` uses a 900000 byte buffer. When encoding, the **RLE1** stage must be careful that it is still able to encode the count byte if the trailing 4 bytes in the buffer are duplicates. Since this determines the buffer size of the **BWT** stage, it is entirely possible for a stream block to decompress to more bytes than the buffer size due to the **RLE1** stage.

2.2.3 StreamBlock

Grammar format:

```
StreamBlock := BlockHeader BlockTrees BlockData
```

The BlockHeader and the `BlockTrees` contains the metadata needed to decode the `BlockData` for each of the transformation stages that require some external metadata.

2.2.3.1 BlockHeader

Grammar format:

```
BlockHeader := BlockMagic BlockCRC Randomized OrigPtr
```

The `BlockMagic` is the 48-bit integer value `0x314159265359`, which is the binary-coded decimal representation of π. It is used to differentiate the StreamBlock from the StreamFooter.

The `BlockCRC` is a 32-bit integer and contains the CRC-32 checksum of the uncompressed data contained in `BlockData`. It is the same checksum used in GZip (RFC 1952, section 8), but is slightly different due to the bit-packing differences noted in section 2.1. Suppose you had a function `CRC32` which computed the checksum as needed by GZip, then the checksum needed by BZip2 can be computed by first reversing the bits of every byte in the input, computing the checksum, and then reversing the bits of the checksum itself as shown in the following Python code:

```
def ComputeBlockCRC(data):
    for i, v in enumerate(data):
        data[i] = ReverseUint8(v)
    return ReverseUint32(CRC32(data))
```

Example results:

```
>>> CRC32 = lambda x: binascii.crc32(x) & 0xffffffff
>>> ReverseUint8 = lambda x: int('{:08b}'.format(x)[::-1], 2)
>>> ReverseUint32 = lambda x: int('{:032b}'.format(x)[::-1], 2)

>>> ComputeBlockCRC(bytearray("Hello, world!"))
0x8e9a7706
```

It is important to note that the implementation of `ComputeBlockCRC` above mutates the input. Also, the example above uses two inefficient ways to reverse the bits of 8-bit and 32-bit integers.

The `Randomized` field is a single bit and should be `0`. Previous versions of BZip2 allowed the input data to be randomized to avoid pathological strings from causing the runtime to be exponential. Modern BWT construction algorithms have guaranteed bounds on the worst-case runtime such that this functionality is deprecated. If the bit is `1`, then randomization is enabled, which is not covered in this document.

The `OrigPtr` is a 24-bit integer and contains the "origin pointer" used in the **BWT** stage. Since the origin pointer references a row in the BWT matrix, it implies that $0 \leq OrigPointer < NumBytes$, where $NumBytes$ is the size of the BWT input. Based on that equation, we can see that `BlockData` must decompress to at least one byte in order for that check to pass.

2.2.3.2 BlockTrees

The `BlockTrees` contains the metadata necessary for the **MTF** and **HUFF** stages. The grammar is:

```
BlockTrees := SymMap NumTrees NumSels Selectors Trees
```

The NumTrees field is a 3-bit integer indicating the number of Huffman trees used in the **HUFF** stage. It must be between 2..6. The number may be larger than necessary; that is, it is permissible to define a Huffman tree that does not end up being used to decode any symbol in BlockData.

The NumSels field is a 15-bit integer indicating the number of selectors used in the **HUFF** stage. Since the Huffman encoded symbols in BlockData terminate with an EOB symbol, there must be at least 1 selector defined. The equation to determine the optimal number of selectors is shown in section 2.2.1.5. If too few selectors are defined, a decoder must error when it runs out of selectors to use while decoding groups of Huffman codes. On the other hand, the C implementation does not error if more selectors are defined than is actually used. However, some decoder implementations do have strict checks that the proper number of selectors was used. Thus, it is encouraged that an encoder implementation never output more selectors than necessary.

In order to show how the SymMap, Selectors, and Trees elements are represented, we use Python code to provide a working implementation of the parsing process. Each of the implementations relies on being able to read bits from the underlying stream. A simple, but inefficient implementation of a bit reader is shown below:

```
class BitReader:
    def __init__(self, data, discard = 0):
        self.d, self.m = data, 0x80
        _ = self.ReadBits(discard)

    def ReadBits(self, n):
        v, d, m = 0, self.d, self.m
        for _ in range(n):
            v <<= 1
            if d[0]&m:
                v |= 1
            m >>= 1
            if m == 0x00:
                d, m = d[1:], 0x80
        self.d, self.m = d, m
        return v

    def ReadBit(self):
        return self.ReadBits(1) != 0
```

The BitReader takes in a bytearray string and an optional argument to discard some number of bits from the input.

2.2.3.2.1 SymMap

Grammar format:

```
SymMap := MapL1 MapL2{1,16}
```

The SymMap represents the symbol stack used in the **MTF** stage by using a two-level bit-map to indicate which symbols are present. The first element, MapL1, is a 16-bit integer where each bit corresponds to a contiguous 16-symbol region of the full 256-symbol space. The leading bit corresponds with symbols 0..15, the next bit with symbols 16..31, and so on. The number of bits set determines the number of MapL2 elements to follow, which are also 16-bit integers. Similar to the first level, the leading bits

correspond to the lower symbols in the associated 16-symbol region. If the bit is set, then that indicates that the corresponding symbol is present. The Python code that implements this is below:

```
def ParseSymMap(br):
    stack = []
    l1 = br.ReadBits(16)
    for i in range(16):
        if l1 & (0x8000>>i):
            l2 = br.ReadBits(16)
            for j in range(16):
                if l2 & (0x8000>>j):
                    stack.append(16*i + j)
    return stack
```

The function takes in an instance of `BitReader` as the variable `br` and returns a list of integers representing the symbol stack. From section 2.2.3.1, we determined that at least one symbol must be used, which implies that at least one `MapL2` element is present. It is permissible for the map to indicate the presence of symbols that are never used. A decoder must be able to handle this situation.

An example of how to decode `SymMap` is shown in appendix A.2.1.

2.2.3.2.2 Selectors

Grammar format:

```
Selectors := Selector{NumSels}
Selector := (0|10|110|1110|11110|111110)
```

The `Selectors` represents the selectors list used in the **HUFF** stage. To encode the selectors, a move-to-front transform is first applied and then each index in the resulting list is written out as a zero-terminated sequence of one-bits. The Python code to parse the selectors list is shown below:

```
def ParseSelectors(br, num_sels, num_trees):
    idxs = []
    for _ in range(num_sels):
        i = 0
        while br.ReadBit():
            i += 1
            assert(i < num_trees)
        idxs.append(i)
    sels = DecodeMTF(idxs, range(num_trees))
    return sels
```

The function takes in an instance of `BitReader` as the variable `br` and two integers, `num_sels` and `num_trees`, as determined from section 2.2.3.2. It returns a list of integers, where each integer is the index of the Huffman tree used to decode the corresponding group of Huffman codes. As indicated by the assertion performed in the function, it is illegal for a selector to reference a Huffman tree that does not exist.

An example of how to decode `Selectors` is shown in appendix A.2.2.

2.2.3.2.3 Trees

Grammar format:

```
Trees := (BitLen Delta{NumSyms}){NumTrees}
Delta := (10|11)* 0
```

The `Trees` represents each of the Huffman trees used in the **HUFF** stage. Each Huffman tree itself is represented by a list of integers of length $NumSyms$, where $NumSyms = NumStack + 2$. From the Python code above, this is equivalent to `len(stack)+2`. The integers represent the bit-length of each corresponding symbol's Huffman code. Since the Huffman trees in BZip2 use canonical codes, there is a well-defined algorithm that converts a list of bit-lengths to a Huffman tree and vice-versa (see RFC 1951, section 3.2.2). This list of integers is written as a delta-encoded list, which is headed by a 5-bit integer containing an initial bit-length. Each symbol's length is represented by a zero-terminated sequence of `10` and `11` bit-strings, which increment or decrement the current bit-length, respectively. The Python code below demonstrates how to parse the trees:

```
def ParseTrees(br, num_syms, num_trees):
    trees = []
    for _ in range(num_trees):
        clens = []
        clen = br.ReadBits(5)
        for _ in range(num_syms):
            while True:
                assert(clen > 0 and clen <= MaxHuffmanBits)
                if not br.ReadBit():
                    break
                clen -= +1 if br.ReadBit() else -1
            clens.append(clen)
        trees.append(clens)
    return trees
```

The function takes in an instance of `BitReader` as the variable `br` and two integers, `num_syms` and `num_trees`. It returns a 2D list of integers, where the outer dimension is across all the trees, and the inner dimension is across all the symbols in a tree. As indicated by the assertion performed in the inner loop, the running bit-length may never be zero or greater than `MaxHuffmanBits`, which is 20. On the other hand, there is no requirement that the minimal number of increment/decrement instructions be used. Thus, the sequence `"10 10 11 10 0"` can be used to indicate $+2$ instead of the shorter sequence `"10 10 0"`. The C implementation will never create the inefficient representation, and it is recommended that an encoder implementation also not do so as well.

An example of how to decode `Selectors` is shown in appendix A.2.3.

The use of canonical Huffman codes is convenient since it is a commonly used technique in other compression formats. However, there is a complication when it comes to BZip2. In order for canonical codes to work, they must be complete. That is, the Huffman tree generated from the bit-lengths is neither over-subscribed (attempting to place more than 2 children per node) nor under-subscribed (some nodes have only 1 child). If the code is not complete, then the resulting tree is not well defined. The following Python function can determine whether the code is complete given a list of bit-lengths:

```
def IsCompleteTree(tree):
    maxLen = max(tree)
    sum = 1 << maxLen
    for clen in tree:
        sum -= (1 << maxLen) >> clen
    return sum == 0
```

If the tree is complete, then any canonical code generation algorithm can be used. However, if the tree is not complete, the reference C implementation does not immediately return an error, and will only do so if certain invalid symbols are used. This is unfortunate, as a decoder implementation that intends to

replicate the original C implementation must now use the exact same “canonical” code generation algorithm in order to produce the same results for these technically invalid codes. This may be necessary since there exist some buggy BZip2 encoders that actually do produce incomplete codes. As a note, if you are writing a BZip2 encoder, please write an implementation of Huffman encoding that always generates complete codes so as to not contribute to this problem. The exact algorithm that the C implementation uses to construct the Huffman tree will be explored in depth in appendix B.

2.2.4 StreamFooter

Grammar format:

```
StreamFooter := FooterMagic StreamCRC Padding
```

The `FooterMagic` is the 48-bit integer value `0x177245385090`, which is the binary-coded decimal representation of $\sqrt{\pi}$. It is used to differentiate the `StreamBlock` from the `StreamFooter`.

The `StreamCRC` is a 32-bit integer and contains a custom checksum computed using each of the `BlockCRCs` from each of the preceding `StreamBlocks` in the same `BZipStream`. The computation is shown in the following Python code:

```
def ComputeStreamCRC(blkCRCs):
    strmCRC = 0
    for blkCRC in blkCRCs:
        strmCRC = blkCRC ^ ((strmCRC<<1) | (strmCRC>>31))
    return strmCRC & 0xffffffff
```

Example result:

```
>>> ComputeStreamCRC([0x12345678, 0xdeadcafe])
0xfac5660e
```

The `Padding` field is used to align the bit-stream to the next byte-aligned edge and will contain between $0..7$ bits. It is not required that these bits to be zeros, but it is highly encouraged that an encoder implementation only output zeros for these padding bits.

Appendix A: Example BZIP2 files

For illustrative purposes, here are some examples of valid BZip2 files.

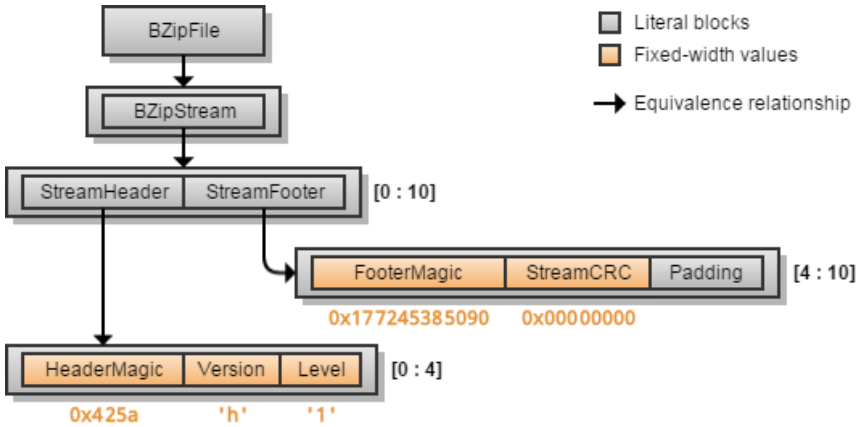
A.1 Empty stream

The simplest possible BZip2 file is one that uncompresses to the empty string. The hex-dump of one such file is shown below:

```
offset  00 01 02 03 04 05 06 07
000000  42 5a 68 31 17 72 45 38  StreamHeader
000008  50 90 00 00 00 00        StreamFooter
```

The first 4 bytes represent the `StreamHeader`, while the later 10 bytes represent the `StreamFooter`. Thus, the smallest BZip2 is 14 bytes in size. Compared to other compression formats, BZip2 has relatively high overhead and thus, relatively poor compression ratio for very short strings.

The hex-string provided above, when parsed produces this tree diagram:



Including the version flag, 9 bytes are dedicated to magic values alone. 4 bytes are used to hold the CRC, which happens to be all zeros when there is no uncompressed output. For an empty stream, the only value that can change is the level flag, which can take on various values and still be valid.

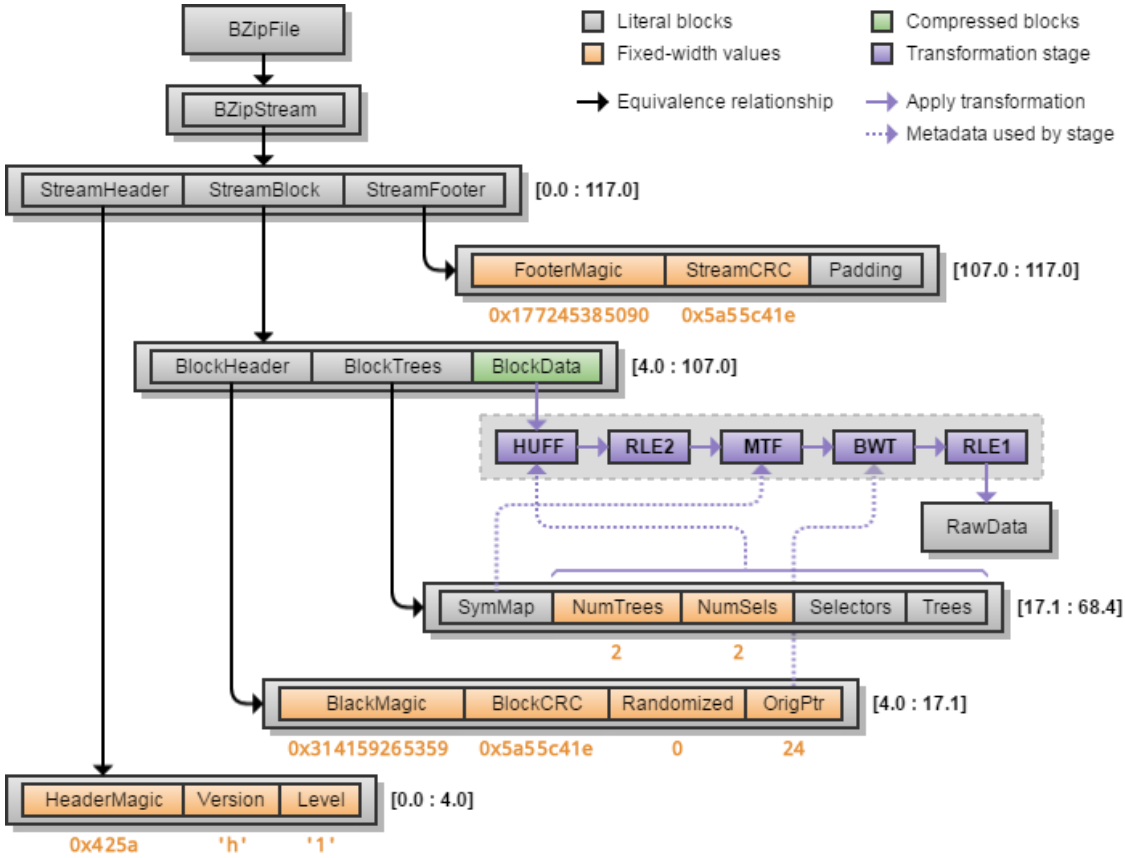
A.2 Normal stream

More realistically, BZip2 files do actually contain uncompressed data. The hex-dump of one such file is shown below:

```
offset  00 01 02 03 04 05 06 07
000000  42 5a 68 31 31 41 59 26  StreamHeader
000008  53 59 5a 55 c4 1e 00 00  BlockHeader
000010  0c 5f 80 20 00 40 84 00  SymMap
000018  00 80 20 40 00 2f 6c dc
000020  80 20 00 48 4a 9a 4c d5  Selectors
000028  53 fc 69 a5 53 ff 55 3f  Trees
000030  69 50 15 48 95 4f ff 55
000038  51 ff aa a0 ff f5 55 31
000040  ff aa a7 fb 4b 34 c9 b8  BlockData
000048  38 ff 16 14 56 5a e2 8b
000050  9d 50 b9 00 81 1a 91 fa
000058  25 4f 08 5f 4b 5f 53 92
000060  4b 11 c5 22 92 d9 50 56
000068  6b 6f 9e 17 72 45 38 50  StreamFooter
000070  90 5a 55 c4 1e

BlockTrees
StreamBlock
```

This file decompresses to the string `"If Peter Piper picked a peck of pickled peppers, where's the peck of pickled peppers Peter Piper picked?????"`. The `StreamHeader`, `BlockHeader`, and `StreamFooter` are highlighted in orange. The bulk of the metadata needed to decode `BlockData` is highlighted in gray. `BlockData`, which contains the actual compressed data, is highlighted in green. Since the uncompressed string is so short, the number of bits needed to represent the metadata (gray highlights) is greater than what is needed for the actual compressed data (green highlights). Since BZip2 is fundamentally a bit-oriented format, the slanted boundaries indicate that the transition from one element to the next is not necessarily byte-aligned.

The hex-string provided above, when parsed produces this tree diagram:



This BZip2 file only contains a single `BZipStream`, which itself only contains a single `StreamBlock`. The brackets next to each rectangle of the form $[n.m : p.q]$ indicates the inclusive starting offset and the exclusive ending offset in the bit-stream that the rectangle occupies. The integers n and p are byte offsets, while m and q are the bit offsets within the current byte. It so happens in this specific BZip2 file that `Padding` contains no bits since the `StreamFooter` already starts on a byte-aligned edge.

There are two CRC values shown above, one in the `StreamHeader` and one in the `BlockHeader`. A decompressor can only verify these only after it has actually decompressed the data. However, since we already know the uncompressed string, we can demonstrate here that they are correct:

```
>>> ComputeBlockCRC(bytearray(
...     "If Peter Piper picked a peck of pickled peppers, " +
...     "where's the peck of pickled peppers Peter Piper picked?????"
... ))
0x5a55c41e
```

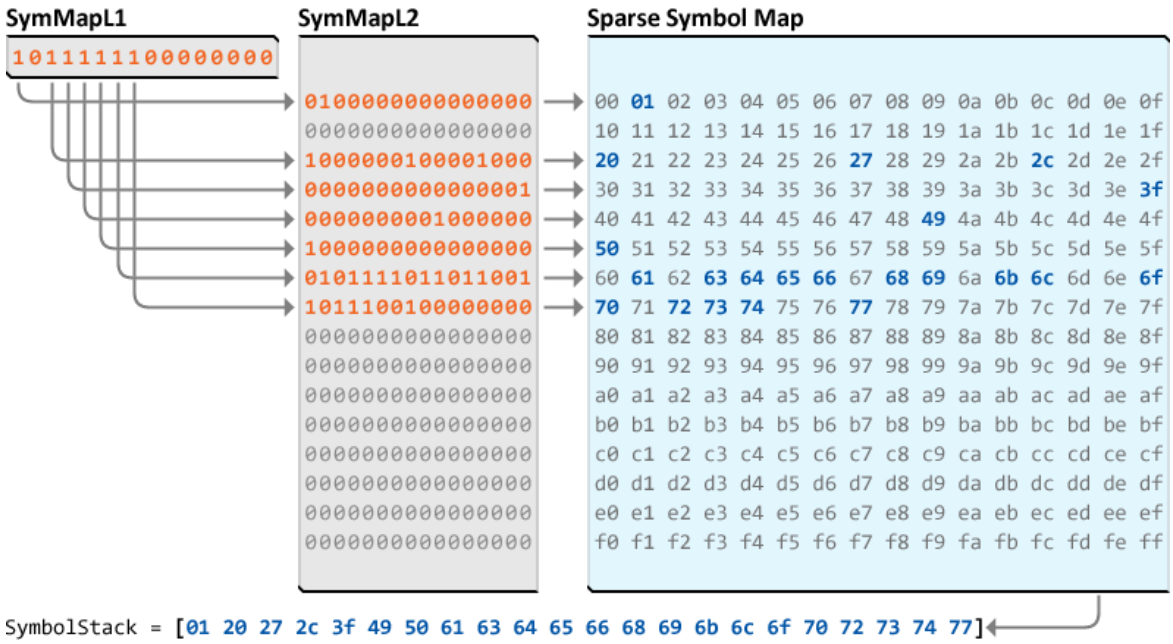
```
>>> ComputeStreamCRC([0x5a55c41e])
0x5a55c41e
```

In the following sections, we will decode the more complex elements like the SymMap, Selectors, Trees, and BlockData. We will use the Python functions defined above to verify the results. To be able to do that, we must first define bz2_data which contains a bytearray of the input file:

```
>>> bz2_data = bytearray((
...     "425a68313141592653595a55c41e00000c5f802000408400008020400002f6cdc" +
...     "802000484a9a4cd553fc69a553ff553f69501548954fff5551ffaaa0fff55531" +
...     "ffaaa7fb4b34c9b838ff1614565ae28b9d50b900811a91fa254f085f4b5f5392" +
...     "4b11c52292d950566b6f9e1772453850905a55c41e"
... ).decode("hex"))
```

A.2.1 Parsing SymMap

The SymMap occupies bit offsets 137 to 265 and is 128-bits long. The symbol map, as it appears in the example file is shown below:



The map contains one SymMapL1 element and seven SymMapL2 elements, the bits of which are shown above in orange. The sequence of 0000000000000000 bits in gray are SymMapL2 elements that are implied by unset bits in SymMapL1 and do not appear in the actual BZip2 file. The section highlighted in blue show visually how the bit map corresponds with the symbols used. By ignoring all unselected symbols, we obtain a sorted list of all used symbols in SymbolStack.

Example using Python code:

```
>>> br = BitReader(bz2_data, discard = 137)
>>> stack = ParseSymMap(br)
>>> stack # Used in MTF stage
[1, 32, 39, 44, 63, 73, 80, 97, 99, 100, 101, 102,
 104, 105, 107, 108, 111, 112, 114, 115, 116, 119]
>>> len(stack) # Used to parse Trees
22
```

A.2.2 Parsing Selectors

The `Selectors` occupies bit offsets 283 to 286 and is 3-bits long. From the diagram in appendix A.2, we know that $NumSels$ is 2, which means that the `Selectors` element in this specific file is not particularly long. The bits for the selectors list is below:

```
BitStream/Selectors: [
    0 10
    0  1
]
```

Example using Python code:

```
>>> br = BitReader(bz2_data, discard = 283)
>>> ParseSelectors(br, 2, 2) # Used in HUFF stage
[0, 1]
```

A.2.3 Parsing Trees

The `Trees` occupies bit offsets 286 to 548 and is 262-bits long. The bit-stream for `Trees` is shown below:

```
BitStream/SymLens: [
    # Tree0
    00010
        2
    0 1010100 110 100 100 110 0 110 10101010100 111111110 0 0 110 100 110 100
    2       5   4   5   6   5 5   4           9         5 5 5   4   5   4   5
    101010100 11111111110 101010100 1111110 110 100 1010100 0
            9           4         8       5   4   5       8 8

    # Tree1
    00001
        1
    0 1010100 100 0 100 101010100 1111111111110 1010101010100  0 11111111110
    1       4   5 5   6         10             4             10 10           5
    10101010100  0  0  0 1111111111110 1010101010100 110 0 0 11111111110
             10 10 10 10             4             10   9 9 9           4
    10101010100 111111110 110 100
              9         5   4   5
]
```

The bit-stream is shown in orange, while the decoded symbol bit-lengths is shown in cyan. Both Tree0 and Tree1 start with a 5-bit integer that indicate the initial bit-length. The number of symbol bit-lengths to read is $NumSyms$, which is also $NumStack + 2$ or 24.

Example using Python code:

```
>>> br = BitReader(bz2_data, discard = 286)
>>> ParseTrees(br, 22+2, 2) # Used in HUFF stage
[[2, 5, 4, 5, 6, 5, 5, 4, 9, 5, 5, 5,
  4, 5, 4, 5, 9, 4, 8, 5, 4, 5, 8, 8],
 [1, 4, 5, 5, 6, 10, 4, 10, 10, 5, 10, 10,
  10, 10, 4, 10, 9, 9, 9, 4, 9, 5, 4, 5]]
```

As mentioned before, the SymLens is a list of bit-lengths that should form a canonical Huffman tree. Rather than showing the Huffman trees, we show below the bit patterns for all the symbol codes that those trees would have generated:

Symbol	Tree0		Tree1	
	Len	Codes	Len	Codes
0	2	00	1	0
1	5	10100	4	1000
2	4	0100	5	11010
3	5	10101	5	11011
4	6	111110	6	111110
5	5	10110	10	1111111000
6	5	10111	4	1001
7	4	0101	10	1111111001
8	9	111111110	10	1111111010
9	5	11000	5	11100
10	5	11001	10	1111111011
11	5	11010	10	1111111100
12	4	0110	10	1111111101
13	5	11011	10	1111111110
14	4	0111	4	1010
15	5	11100	10	1111111111
16	9	111111111	9	111111000
17	4	1000	9	111111001
18	8	11111100	9	111111010
19	5	11101	4	1011
20	4	1001	9	111111011
21	5	11110	5	11101
22	8	11111101	4	1100
23	8	11111110	5	11110

A.2.4 Decompressing BlockData

The BlockData occupies bit offsets 548 to 856 and is 308-bits long. We can now start decompressing this block by running all of the transformation stages in reverse using the metadata obtained from the preceding elements.

A.2.4.1 Huffman encoding (HUFF)

In the **HUFF** stage, we decode the bits of the block data by using the Huffman table and selectors list obtain earlier. Below, a bit-stream is shown in orange of BlockData:

```
BitStream/Symbols: [
    # Symbols 0..49 using Tree0
    10110 0110 1001 1001 00 11011 1000 00 11100 0111 111110 00 10110 00 0101
        5   12   20   20  0    13   17  0    15   14      4  0     5  0    7
    00 0101 0110 0101 10101 11000 10100 0101 11001 11010 10100 00 10111 00
     0    7   12    7     3     9     1    7    10    11     1  0     6  0
    1000 00 00 0100 00 00 1000 11010 1001 00 0111 11101 00 0100 10101 00 11110
      17  0  0    2  0  0   17    11   20  0   14    19  0    2     3  0    21
    00 0100 00 10111
     0    2  0     6
```

```
    # Symbols 50..91 using Tree1
    11010 0 1011 0 1011 11101 0 1001 1100 1001 0 0 1001 0 1100 0 1000 11100 0
        2 0   19 0   19    21 0    6   22    6 0 0    6 0   22 0    1     9 0
    1010 0 1000 1010 0 1001 0 11011 0 0 1010 1000 0 0 1010 1100 11010 11011 0
      14 0    1   14 0    6 0     3 0 0   14    1 0 0   14   22     2     3 0
    111110 0 11110
         4 0    23
]
```

For convenience, we insert spaces in the bit-stream to group each bit-string according to its given Huffman symbol, the decoded value of which is shown underneath in cyan. Note that the bit-stream is segregated into two sections, where the first 50 symbols is decoded using `Tree0`, and the later 42 symbols is decoded using `Tree1`. Note that symbol `23` only appears as the last symbol in this sequence because the decoder uses it to know when to stop reading Huffman encoded symbols.

Before passing the symbols list to the **RLE2** stage, we must remap the symbol alphabet. Recall from section 2.2.1.5 that symbols `0` and `1` get mapped to symbols `A` and `B`, the `EOB` symbol `23` gets dropped, and all symbols in $2..NumSyms-2$ simply get decremented by one. The result of this remapping is shown below:

```
Symbols/Output: [
    5 12 20 20 0 13 17 0 15 14 4 0 5 0 7 0 7 12 7 3 9 1 7 10 11
    4 11 19 19 A 12 16 A 14 13 3 A 4 A 6 A 6 11 6 2 8 B 6  9 10

    1 0 6 0 17 0 0 2 0 0 17 11 20 0 14 19 0 2 3 0 21 0 2 0 6
    B A 5 A 16 A A 1 A A 16 10 19 A 13 18 A 1 2 A 20 A 1 A 5

    2 0 19 0 19 21 0 6 22 6 0 0 6 0 22 0 1 9 0 14 0 1 14 0 6
    1 A 18 A 18 20 A 5 21 5 A A 5 A 21 A B 8 A 13 A B 13 A 5

    0 3 0 0 14 1 0 0 14 22 2 3 0 4 0 23
    A 2 A A 13 B A A 13 21 1 2 A 3 A
]
```

A.2.4.2 Run-length encoding (RLE2)

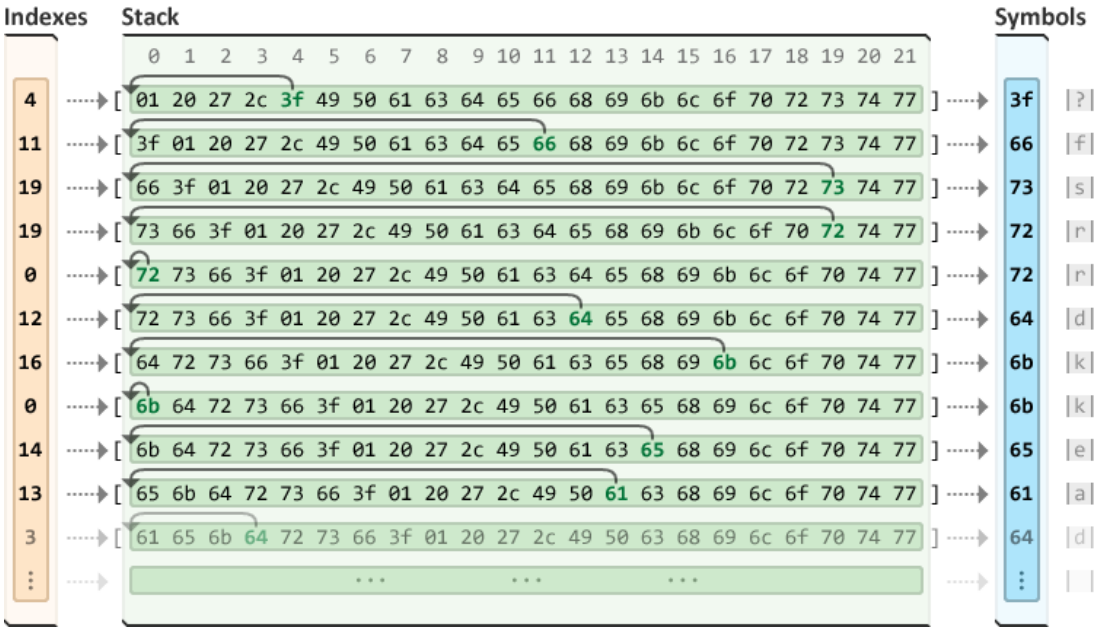
The RLE2 stage converts the `A` and `B` symbols to become runs of `0` symbols. This conversion process uses the bijective numeration described in section 2.2.1.4. The input symbols are shown in orange, while the output symbols are shown underneath in cyan:

```
Input/Output: [
    4 11 19 19 A 12 16 A 14 13 3 A 4 A 6 A 6 11 6 2 8 B   6 9 10
    4 11 19 19 0 12 16 0 14 13 3 0 4 0 6 0 6 11 6 2 8 0 0 6 9 10

    BA      5 A 16 AA    1 AA    16 10 19 A 13 18 A 1 2 A 20 A 1 A 5
    0 0 0 0 5 0 16 0 0 0 1 0 0 0 16 10 19 0 13 18 0 1 2 0 20 0 1 0 5

    1 A 18 A 18 20 A 5 21 5 AA    5 A 21 AB        8 A 13 AB        13 A 5
    1 0 18 0 18 20 0 5 21 5 0 0 0 5 0 21 0 0 0 0 0 8 0 13 0 0 0 0 0 13 0 5

    A 2 AA    13 BAA             13 21 1 2 A 3 A
    0 2 0 0 0 13 0 0 0 0 0 0 0 0 13 21 1 2 0 3 0
]
```

The resulting list of symbols is to be used as a list of indexes used in the upcoming **MTF** stage. Note that this stage usually outputs more symbols than inputted. A decoder must be careful that this stage does not output more symbols than the block size specified by the `Level` in the `StreamHeader`.

A.2.4.3 Move-to-front transform (MTF)

The **MTF** stage performs the inverse move-to-front transform. In the diagram below, we show how the first few symbols are decoded:



The input list of indexes is shown in the left column in orange, with the leading indexes at the top. The symbol stack is shown in the middle section as the top row. To decode each index value, we look up the symbol value in the stack for the current index, emit that symbol as the output, and move the symbol to the front of the stack. After processing all indexes, we obtain the output list of symbols shown in the right column in blue.

The full transform is shown below where the input indexes are shown in orange and the output symbols are shown in blue as hexadecimal values:

```
Input/Output: [
     4 11 19 19  0 12 16  0 14 13  3  0  4  0  6  0  6 11  6  2  8  0  0  6  9
    3f 66 73 72 72 64 6b 6b 65 61 64 64 72 72 66 66 73 2c 65 73 3f 3f 3f 64 01

    10  0  0  0  0  5  0 16  0  0  0  1  0  0  0 16 10 19  0 13 18  0  1  2  0
    20 20 20 20 20 65 65 69 69 69 69 65 65 65 65 68 72 70 70 6b 6c 6c 6b 70 70

    20  0  1  0  5  1  0 18  0 18 20  0  5 21  5  0  0  0  5  0 21  0  0  0  0
    74 74 70 70 68 70 70 50 50 49 6f 6f 74 77 70 70 70 70 50 50 63 63 63 63 63

     0  8  0 13  0  0  0  0  0 13  0  5  0  2  0  0  0 13  0  0  0  0  0  0  0
    63 6b 6b 20 20 20 20 20 20 69 69 70 70 20 20 20 20 65 65 65 65 65 65 65 65

     0 13 21  1  2  0  3  0
    65 72 27 72 65 65 20 20
]
```

Example using Python code:

```
>>> stack = [
...     1, 32, 39, 44, 63, 73, 80, 97, 99, 100, 101, 102,
...     104, 105, 107, 108, 111, 112, 114, 115, 116, 119,
... ]
```

```
>>> idxs = [
...     4, 11, 19, 19, 0, 12, 16, 0, 14, 13, 3, 0, 4, 0, 6, 0, 6, 11, 6, 2,
...     8, 0, 0, 6, 9, 10, 0, 0, 0, 0, 5, 0, 16, 0, 0, 0, 1, 0, 0, 0,
...     16, 10, 19, 0, 13, 18, 0, 1, 2, 0, 20, 0, 1, 0, 5, 1, 0, 18, 0, 18,
...     20, 0, 5, 21, 5, 0, 0, 0, 5, 0, 21, 0, 0, 0, 0, 0, 8, 0, 13, 0,
...     0, 0, 0, 0, 13, 0, 5, 0, 2, 0, 0, 0, 13, 0, 0, 0, 0, 0, 0, 0,
...     0, 13, 21, 1, 2, 0, 3, 0,
... ]

>>> bwt = DecodeMTF(idxs, stack[:])
>>> str(bwt).encode('hex')
"3f66737272646b6b656164647272666732c65733f3f3f6401" +
"20202020206565696969696565656565687270706b6c6c6b7070" +
"74747070687070505049 6f6f74777070707050506363636363" +
"636b6b20202020202020696970702020202065656565656565" +
"6572277265652020"
```

A.2.4.4 Burrows-Wheeler transform (BWT)

The **BWT** stage performs the inverse Burrows-Wheeler transform on the input data. Visually showing the inverse BWT for the input would occupy too many pages to practically show here. Instead, we rely on the Python code to invert the input for us.

Example using Python code:

```
>>> bwt
bytearray(
    "?fsrrdkkeaddrrffs,es???d\x01     eeiiiieeeehrppkllkp" +
    "pttpphppPPIootwpppPPccccckk      iipp    eeeeeeeeer'ree  "
)

>>> DecodeBWT(bwt, 24)
bytearray(
    "If Peter Piper picked a peck of pickled peppers, " +
    " where's the peck of pickled peppers Peter Piper picked????\x01"
)
```

First, we print the `bwt` variable to show the raw string in ASCII, rather than in hexadecimal (as was done in the **MTF** stage). The origin pointer used is 24, which we obtained from parsing the preceding `BlockHeader`. Also, since BWT is a permutation, we note that the input and output strings contain the exact same characters, albeit in a different order.

A.2.4.5 Run-length encoding (RLE1)

The **RLE1** decoding for this example is not particularly exciting. Near the end of the string, we see a sequence of 4x `'?'` characters, along with a count byte of 1. Thus, we replace the count with a single `'?'` to obtain the final sequence of 5x `'?'` characters:

```
Input:  "If Peter Piper picked a peck of pickled peppers, " +
        " where's the peck of pickled peppers Peter Piper picked????\x01"

Output: "If Peter Piper picked a peck of pickled peppers, " +
        " where's the peck of pickled peppers Peter Piper picked?????"
```

Appendix B: Handling degenerate Huffman trees

As mentioned in section 2.2.3.2.3, Huffman trees used in BZip2 *should* be canonical trees, which implies that they are also complete. Since BZip2 does *not* error when a non-complete tree is used, the only way to behave the same way as the C implementation for these technically invalid trees is to replicate the behavior of the C canonical tree generation algorithm. Some decoder implementation avoids this problem by using the exact same Huffman decoding algorithm as BZip2. This works just fine, but limits the decoder to one specific implementation. In this appendix, we will explore a way to wrap the C implementation and provide the Huffman codes in a form that is easier to adopt by other decoder implementations.

The wrapper below is written in Go and wraps a ported version of the C library's canonical code generation algorithm. The function `HandleDegenerateCodes` presented in appendix B.1 takes a `[]HuffmanCode`, which may be either a complete or non-complete tree, and returns a `[]HuffmanCode`, which is guaranteed to be a complete tree. The `HuffmanCode` type is defined as:

```
type HuffmanCode struct {
    Sym uint32 // The symbol being mapped
    Len uint32 // Bit-length of the Huffman code
    Val uint32 // Value of the Huffman code (must be in 0..(1<<Len)-1)
}
```

For the input, only the `Sym` and `Len` fields must be populated since only the bit-lengths are known. The function output will populate the `Val` field. As mentioned in section 2.1, the leading bits in the bit-stream will be in the MSB positions of `Val`.

The wrapper works by running a depth-first search across all possible bit-strings in order to map out the entire Huffman code space that the C implementation occupies. If the input tree is under-subscribed, the worst-case runtime is $O(2^{MaxLen})$, where $MaxLen$ is the maximum bit-length in the input. If the input tree is over-subscribed, the worst-case runtime is $O(MaxNumSyms)$, where $MaxNumSyms$ is 258. Unfortunately, buggy encoders typically output under-subscribed trees rather than over-subscribed trees, so the runtime of this wrapper is often a bounded exponential.

For example, if the input is an under-subscribed tree:

```
Input:  []HuffmanCode{
        {Sym: 0, Len: 3},
        {Sym: 1, Len: 4},
        {Sym: 2, Len: 3},
}
Output: []HuffmanCode{
        {Sym:   0, Len: 3, Val: 0}, // 000
        {Sym:   1, Len: 4, Val: 4}, // 0100
        {Sym:   2, Len: 3, Val: 1}, // 001
        {Sym: 258, Len: 4, Val: 5}, // 0101
        {Sym: 259, Len: 3, Val: 3}, // 011
        {Sym: 260, Len: 1, Val: 1}, // 1
}
```

Since an under-subscribed tree has missing children, the wrapper function adds several codes with invalid symbol values ≥ 258 in order to make the tree complete. When using with this tree, a decoder must return an error if any of the invalid symbols are ever used.

For example, if the input is an over-subscribed tree:

```
Input:  []HuffmanCode{
        {Sym: 0, Len: 1},
        {Sym: 1, Len: 3},
        {Sym: 2, Len: 4},
        {Sym: 3, Len: 3},
        {Sym: 4, Len: 2},
}
Output: []HuffmanCode{
        {Sym: 0, Len: 1, Val: 0}, // 0
        {Sym: 1, Len: 3, Val: 6}, // 110
        {Sym: 3, Len: 3, Val: 7}, // 111
        {Sym: 4, Len: 2, Val: 2}, // 10
}
```

Since an over-subscribed tree has too many children, the wrapper function must remove some symbols in order to make the tree complete.

B.1 Source code

The Go wrapper is provided below. Since a port of the C implementation is used, a copy of the BZip2 license is included. The Go wrapper itself is released into the public domain.

```
func HandleDegenerateCodes(codes []HuffmanCode) []HuffmanCode {
    // ===========================================================
    // This program, "bzip2", the associated library "libbzip2", and all
    // documentation, are copyright (C) 1996-2010 Julian R Seward.  All
    // rights reserved.
    //
    // Redistribution and use in source and binary forms, with or without
    // modification, are permitted provided that the following conditions
    // are met:
    //
    // 1. Redistributions of source code must retain the above copyright
    //    notice, this list of conditions and the following disclaimer.
    //
    // 2. The origin of this software must not be misrepresented; you must
    //    not claim that you wrote the original software.  If you use this
    //    software in a product, an acknowledgment in the product
    //    documentation would be appreciated but is not required.
    //
    // 3. Altered source versions must be plainly marked as such, and must
    //    not be misrepresented as being the original software.
    //
    // 4. The name of the author may not be used to endorse or promote
    //    products derived from this software without specific prior written
    //    permission.
    //
    // THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS
    // OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    // WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    // ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY
    // DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    // DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
    // GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
```

```
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
// WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
// NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
//
// Julian Seward, jseward@bzip.org
// bzip2/libbzip2 version 1.0.6 of 6 September 2010
// ============================================================
const (
    statusOkay = iota
    statusInvalid
    statusNeedBits
    statusMaxBits

    maxHuffmanBits = 20
    maxNumSyms     = 256 + 2
)

var (
    limits [maxHuffmanBits + 2]int32
    bases  [maxHuffmanBits + 2]int32
    perms  [maxNumSyms]int32
    minLen uint32 = maxHuffmanBits
    maxLen uint32 = 0
)

// createTables is the BZ2_hbCreateDecodeTables function from the C code.
var createTables = func(codes []HuffmanCode) {
    for _, c := range codes {
        if c.Len > maxLen {
            maxLen = c.Len
        }
        if c.Len < minLen {
            minLen = c.Len
        }
    }

    var pp int
    for i := minLen; i <= maxLen; i++ {
        for j, c := range codes {
            if c.Len == i {
                perms[pp] = int32(j)
                pp++
            }
        }
    }

    var vec int32
    for _, c := range codes {
        bases[c.Len+1]++
    }
    for i := 1; i < len(bases); i++ {
        bases[i] += bases[i-1]
    }
    for i := minLen; i <= maxLen; i++ {
```

```
            vec += bases[i+1] - bases[i]
            limits[i] = vec - 1
            vec <<= 1
        }
        for i := minLen + 1; i <= maxLen; i++ {
            bases[i] = ((limits[i-1] + 1) << 1) - bases[i]
        }
    }

    // getSymbol is the GET_MTF_VAL macro from the C code.
    var getSymbol = func(c HuffmanCode) (uint32, int) {
        v := c.Val << (32 - c.Len)
        n := c.Len

        zn := minLen
        if zn > n {
            return 0, statusNeedBits
        }
        zvec := int32(v >> (32 - zn))
        v <<= zn
        for {
            if zn > maxLen {
                return 0, statusMaxBits
            }
            if zvec <= limits[zn] {
                break
            }
            zn++
            if zn > n {
                return 0, statusNeedBits
            }
            zvec = (zvec << 1) | int32(v>>31)
            v <<= 1
        }
        if zvec-bases[zn] < 0 || zvec-bases[zn] >= maxNumSyms {
            return 0, statusInvalid
        }
        return uint32(perms[zvec-bases[zn]]), statusOkay
    }

    // Step 1: Create the Huffman trees using the C algorithm.
    createTables(codes)

    // Step 2: Starting with the shortest bit pattern, explore the whole tree.
    // If tree is under-subscribed, the worst-case runtime is O(1<<maxLen).
    // If tree is over-subscribed, the worst-case runtime is O(maxNumSyms).
    hcodes := make([]HuffmanCode, maxNumSyms)
    var exploreCode func(HuffmanCode) bool
    exploreCode = func(c HuffmanCode) (term bool) {
        sym, status := getSymbol(c)
        switch status {
        case statusOkay:
            // This code is valid, so insert it.
            c.Sym = sym
            hcodes[sym] = c
```

```
            term = true
        case statusInvalid:
            // This code is invalid, so insert an invalid symbol.
            c.Sym = uint32(len(hcodes))
            hcodes = append(hcodes, c)
            term = true
        case statusNeedBits:
            // This code is too short, so explore both children.
            c0 := HuffmanCode{Len: c.Len + 1, Val: c.Val<<1 | 0}
            c1 := HuffmanCode{Len: c.Len + 1, Val: c.Val<<1 | 1}

            t0 := exploreCode(c0)
            t1 := exploreCode(c1)
            switch {
            case !t0 && t1:
                c0.Sym = uint32(len(hcodes))
                hcodes = append(hcodes, c0)
            case !t1 && t0:
                c1.Sym = uint32(len(hcodes))
                hcodes = append(hcodes, c1)
            }
            term = t0 || t1
        case statusMaxBits:
            // This code is too long, so report it upstream.
            term = false
        }
        return term // Did this code terminate?
    }
    exploreCode(HuffmanCode{})

    // Step 3: Copy new sparse codes to old output codes.
    codes = codes[:0]
    for _, c := range hcodes {
        if c.Len > 0 {
            codes = append(codes, c)
        }
    }
    return codes
}
```