Contents

1 Basic Test Results

```
Starting tests...
Wed Jan  8 14:55:06 IST 2014
4aeeaf3fb6f5149097af171d408a39400970041d  -


hzip.py
hzlib.py
README
hunzip.py

Testing README...
Done testing README...

Testing hzlib.py...
result_code    maketree    60    1
result_code    decbook    32    1
result_code    symcount    30    1
result_code    codebook    32    1
result_code    canbook    32    1
result_code    isymcount    30    1
result_code    split    80    1
result_code    pad    498    1
result_code    unpad    498    1
result_code    decompress    44    1
result_code    compress    44    1
result_code    join    80    1
result_code    icompress    44    1
result_code    ipad    498    1
result_code    ijoin    80    1
result_code    iunpad    498    1
result_code    idecompress    44    1
result_code    isplit    80    1
Done testing hzlib.py

Testing hzip.py and hunzip.py...
result_code    difflines    0    1
Done testing hzip.py and hunzip.py
Grading summary
---------------
difflines Penalty: 0.0
****** symcount:
Number of failed tests: 0
Total number of tests : 30
Penalty: 0.0
****** isymcount:
Number of failed tests: 0
Total number of tests : 30
Penalty: 0.0
****** maketree:
Number of failed tests: 0
Total number of tests : 60
Penalty: 0.0
****** codebook:
Number of failed tests: 0
Total number of tests : 32
Penalty: 0.0
****** canbook:
Number of failed tests: 0
Total number of tests : 32
```

```
Penalty: 0.0
****** decbook:
Number of failed tests: 0
Total number of tests : 32
Penalty: 0.0
****** compress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** icompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** decompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** idecompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** pad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** ipad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** unpad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** iunpad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** join:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** ijoin:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** split:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** isplit:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
******
Expected automatic grade: 100.0
******
Submission passed!
Tests completed
```

2 aaa expected autograde

```
Grading summary
---------------
difflines Penalty: 0.0
****** symcount:
Number of failed tests: 0
Total number of tests : 30
Penalty: 0.0
****** isymcount:
Number of failed tests: 0
Total number of tests : 30
Penalty: 0.0
****** maketree:
Number of failed tests: 0
Total number of tests : 60
Penalty: 0.0
****** codebook:
Number of failed tests: 0
Total number of tests : 32
Penalty: 0.0
****** canbook:
Number of failed tests: 0
Total number of tests : 32
Penalty: 0.0
****** decbook:
Number of failed tests: 0
Total number of tests : 32
Penalty: 0.0
****** compress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** icompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** decompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** idecompress:
Number of failed tests: 0
Total number of tests : 44
Penalty: 0.0
****** pad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** ipad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** unpad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
****** iunpad:
Number of failed tests: 0
Total number of tests : 498
Penalty: 0.0
```

```
****** join:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** ijoin:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** split:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
****** isplit:
Number of failed tests: 0
Total number of tests : 80
Penalty: 0.0
******
Expected automatic grade: 100.0
******
Submission passed!
```

3 aaa hint result.png

4 README

```
roigreenberg
305571234
Roi Greenberg



==================
= README for ex9 =
==================
==================
=     usage:     =
==================

python3 hzip.py -h, --help            show this help message and exit
python3 hzip.py    -o OUTFILE, --outfile OUTFILE
                        Name of output file
python3 hzip.py    -s SUFFIX, --suffix SUFFIX
                        Suffix to use instead of .hz
python3 hzip.py    -f, --force           Force compression and overwrite output file if it
                        exists
python3 hzip.py    -l LEVEL, --level LEVEL
                        Maximum levels of compression
python3 hzip.py    -a, --alwayscompress  Compress to max level even if it would make output
                        larger

python3 hunzip.py  -h, --help            show this help message and exit
python3 hunzip.py  -o OUTFILE, --outfile OUTFILE
                        Name of output file
python3 hunzip.py  -s SUFFIX, --suffix SUFFIX
                        Default suffix to remove instead of .hz
python3 hunzip.py  -S OUTSUFFIX, --outsuffix OUTSUFFIX
                        Default suffix to add if instead of .out
python3 hunzip.py  -f, --force           Force decompression and overwrite output file if it
                        exists

==================
=  Description:  =
==================

Question:
1) in the zip process we need to run over the data twice.
    one time to create the codebook and second time to
    compress the data.
   in the unzip process we need only 1 run to get both codebook
    and data

2) th
    a                                                    re memory
    b                                                    of the file


hzlib-2

an implementation of Huffman compression code for coding and decoding of
files. the file consist of several functions:
 *) symbol_count: a function that returns a dictionary of
    char: char-count-in-text pairs.
 *) make_huffman_tree: returns a tree which is represents the frequencies of
    different chars of input, according to Huffman code algorithm.
```

You were asked how to compress using less memory. I'm not sure what you meant by the second part of your answer, and the first part does the opposite. To compress using less memory you can, for example, read small blocks of the data to compress.

```
 *) build_codebook: returns a table of char: code pairs, given a Huffman tree
 *) build_canonical_codebook: returns a canonical codebook.
 *) build_decodebook: the function returns a dictionary which is a decoding
    compatible to the input codebook.
 *) compress: the function create an iterator of "0" or "1" as ints, after
    iterating on corpus input.
 *) decompress: the function run over the decoding bits of coded bits input
    and create an iterator of 0 or 1 as an int.
 *) pad: the function run over each eight sequence bits out of the nput,
    adds the 1 as a final bit and appends zeros for the total length be
    divided by 8. the function create an iterator of 0 or 1 as an ints.
 *) unpad: the function run over all bytes of input, taking off the '0' and
    '1' on top of it and create an iterator of 0 or 1 as ints.
 *)join: the function run over the bytes of input (first codebook then
    data) and yields the codebook vals which appear, then create an iterator of
    the data items.
 *)split: that function split the output of the function join to data and codebook.
    the function return a tuple which is consist of a dictionary - canonical
    coding table and an iterator which iterate over rest of byteseq as
    byte sequent.

hzip.py: compress a file using hzlib.zip functions
hunzip.py: decompress a file using hzlib.zip functions

============================
=  List of submitted files: =
============================

README
hzlib.py
hzip.py
hunzip.py
```

5 hunzip.py

```
#!/usr/bin/env python3
'''
usage: hunzip.py [-h] [-o OUTFILE] [-s SUFFIX] [-S OUTSUFFIX] [-f] infile

Decompress files using the hzlib module.

positional arguments:
  infile

optional arguments:
  -h, --help            show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                        Name of output file
  -s SUFFIX, --suffix SUFFIX
                        Default suffix to remove instead of .hz
  -S OUTSUFFIX, --outsuffix OUTSUFFIX
                        Default suffix to add if instead of .out
  -f, --force           Force decompression and overwrite output file if it
                        exists
'''
import hzlib
import struct
import os.path

DEFAULT_IN_EXTENSION = '.hz'
DEFAULT_OUT_EXTENSION = '.out'
MEGIC_LENGTH = len(hzlib.MAGIC)

def main():
    import argparse
    parser = argparse.ArgumentParser(
        description='Decompress files using the hzlib module.')
    parser.add_argument("infile")
    parser.add_argument("-o", "--outfile", type=str, default=None,
                        help='Name of output file')
    parser.add_argument("-s", "--suffix", type=str,
                        default=DEFAULT_IN_EXTENSION,
                        help=('Default suffix to remove instead of ' +
                              DEFAULT_IN_EXTENSION))
    parser.add_argument("-S", "--outsuffix", type=str,
                        default=DEFAULT_OUT_EXTENSION,
                        help=('Default suffix to add if instead of ' +
                              DEFAULT_OUT_EXTENSION))
    parser.add_argument("-f", "--force", action='store_true',
                        help=('Force decompression and overwrite output ' +
                              'file if it exists'))
    args = parser.parse_args()

    # create the output file name
    source_file = args.infile
    out_file_name = args.outfile
    suffixin = args.suffix
    suffixout = args.outsuffix
    if not out_file_name:
        new_file = source_file[:-len(suffixin)] + suffixout
    else:
        new_file = out_file_name
    source_file = args.infile

```

```
60      # open and read the file
61      file = open(source_file, "rb", 1)
62      file.seek(MEGIC_LENGTH,0)
63      level = struct.unpack('1b',file.read(1))[0]
64      text = file.read()
65      file.close()
66
67      # decompress the data
68      for i in range(level):
69          splited = hzlib.split(text)
70          can_codebook = splited[1]
71          decode = hzlib.build_decodebook(can_codebook)
72          text= list(splited[0])
73          unpaded = list(hzlib.unpad(text))
74          text = list(hzlib.decompress(unpaded,decode))
75
76      # create the new file if possible
77      if os.path.isfile(out_file_name) and not args.force:
78          raise FileExistsError
79      else:
80          new_file = open(new_file,'wb')
81          for bit in text:
82              new_file.write(struct.pack('1B',bit))
83
84  if __name__ == '__main__':
85      main()
```

6 hzip.py

```
#!/usr/bin/env python3
'''
usage: hzip.py [-h] [-o OUTFILE] [-s SUFFIX] [-f] [-l LEVEL] [-a] infile

Compress files using the hzlib module.

positional arguments:
  infile

optional arguments:
  -h, --help            show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                        Name of output file
  -s SUFFIX, --suffix SUFFIX
                        Suffix to use instead of .hz
  -f, --force           Force compression and overwrite output file if it
                        exists
  -l LEVEL, --level LEVEL
                        Maximum levels of compression
  -a, --alwayscompress  Compress to max level even if it would make output
                        larger

Format of saved file is the following:
The string of bytes MAGIC from hzlib, followed by one byte containing the
compression level of the data, followed by the data.

Compression level 0 is the raw input. The data used in compression level
n+1 is the result of compressing the result provided by compression
level n. Note that each level includes its codebook in its data, but does
not include the magic number.
'''
import hzlib
import struct
import copy
import os.path

DEFAULT_EXTENSION = '.hz'
MAX_COMPRESSION_LEVEL = 255
MIN_COMPRESSION_LEVEL = 0

def main():
    import argparse
    parser = argparse.ArgumentParser(
        description='Compress files using the hzlib module.')
    parser.add_argument("infile")
    parser.add_argument("-o", "--outfile", type=str, default=None,
                        help='Name of output file')
    parser.add_argument("-s", "--suffix", type=str, default=DEFAULT_EXTENSION,
                        help=('Suffix to use instead of ' +
                              DEFAULT_EXTENSION))
    parser.add_argument("-f", "--force", action='store_true',
                        help=('Force compression and overwrite output ' +
                              'file if it exists'))
    parser.add_argument("-l", "--level", type=int,
                        default=MAX_COMPRESSION_LEVEL,
                        help='Maximum levels of compression')
    parser.add_argument("-a", "--alwayscompress", action='store_true',
                        help=('Compress to max level even if it would ' +
                              'make output larger'))
```

```
60      args = parser.parse_args()
61
62      #Your code goes here and in the other functions you should write...
63      level = args.level
64      outfile = args.outfile
65      suffix = args.suffix
66
67      # create the output file name
68      if not outfile:
69          new_file_name = args.infile+suffix
70      else:
71          new_file_name = outfile
72      # open and read the file
73      file = open(args.infile, "rb", 1)
74      text = file.read()
75      file.close()
76      preview_len = len(text)
77      # compressing the data
78      for index in range(level):
79          counter = hzlib.symbol_count(text)
80          huff_tree = hzlib.make_huffman_tree(counter)
81          codebook = hzlib.build_codebook(huff_tree)
82          can_codebook = hzlib.build_canonical_codebook(codebook)
83          compress = list(hzlib.compress(text,can_codebook))
84          temp_text = list(hzlib.pad(compress))
85          compress_text = list(hzlib.join(temp_text, can_codebook))
86          length=len(compress_text)
87          # check if data indeed compressed
88          if length < preview_len or args.alwayscompress:
89              text = compress_text
90              preview_len = length
91          else:
92              level = index
93              break
94      # create the output file
95      if os.path.isfile(outfile) and not args.force:
96          raise FileExistsError
97      else:
98          new_file = open(new_file_name,'wb')
99          new_file.write(hzlib.MAGIC)
100         new_file.write(struct.pack('1B',level))
101         for byte in text:
102             new_file.write(struct.pack('1B',byte))
103         new_file.close()
104
105 if __name__ == '__main__':
106     main()
```

7 hzlib.py

```
1  ##################################################################
2  # FILE: hzlib.py
3  # WRITER: Roi Greenberg + roigreenberg + 305571234
4  # EXERCISE : intro2cs ex9 2013-2014
5  # Description: implement some function about Huffman tree and
6  # compress and decompress data
7  ##################################################################
8
9  import collections
10 from bisect import bisect
11 '''
12 This module contains several function for compress and decompress data, using
13 the Huffman code algorithm.
14 '''
15
16 MAGIC = b"i2cshcfv1"
17 LEFT_TREE = 0
18 RIGHT_TREE = 1
19
20 def symbol_count(data):
21     """the function return dictionary from item to number of returns in data
22     Args: data - a data
23     """
24     return collections.Counter(data)
25
26
27 def make_huffman_tree(counter):
28     """the function create a huffman tree of a given dictionary from item to
29     number of returns
30     Return tree of tuple of tuples represent the tree or None if dictionary
31     is empty
32     Args: counter - a dictionary (output of symbol_data)
33     """
34     # create a list from the dictonary and sorted it from low repeats to high
35     # and high value to low
36     sort_list = sorted([(tuple0, counter[tuple0]) for tuple0 in counter], \
37                        reverse=True )
38     sort_list.sort(key=lambda leaf: leaf[1])
39
40     # run until have only 1 tuple
41     while len(sort_list) > 1:
42         # take the first 2 tuples
43         tuple1 = sort_list.pop(0)
44         tuple2 = sort_list.pop(0)
45
46         # calculate the combined repeats
47         count = tuple1[1] + tuple2[1]
48
49         #create new tuple of both tuple
50         parent = ((tuple2[0], tuple1[0]), count)
51
52         #create a list of all the reapets
53         counts = [repeats[1] for repeats in sort_list]
54
55         #insert the new tuple to the list in the right place
56         sort_list.insert(bisect(counts, count), parent)
57
58     return sort_list[0][0] if sort_list else None
59
```

"tuple0" is not a good name. What does that 0 mean? You can use [(item, count) for item, count in counter.items()]
-2

```
60
61  def build_codebook(huff_tree):
62      """create a codebook of the Huffman tree
63      the function recieve a huffman tree and return a dictionary from item
64      to tuple of length and decimal value of the binary code represent the item
65      Args:
66      huff_tree - a coded tree of a recursive tuple structure
67      (same structure of output of privious function).
68      bin_item - a string. default is "".
69      codebook - a dictionary. default is {}.
70      """
71      new_codebook = {}
72      def codebook(huff_tree, n=""):
73          # return empty dictionary in tree is empty
74          if not huff_tree:
75              return {}
76          # return the dictionary in case tree is only 1 leaf
77          elif type(huff_tree) is not tuple:
78              return {huff_tree: (1, 0)}
79
80          # the left branch
81          left=huff_tree[LEFT_TREE]
82          # the right branch
83          right=huff_tree[RIGHT_TREE]
84
85          # if got to leaf, add it to the dictionary
86          # if not check the left branch in recursive way
87          if type(left) is not tuple:
88              binary_info = (len(n + "0"), int(n + "0", 2))
89              new_codebook[left] = binary_info
90          else:
91              codebook(left, n + "0")
92
93          # if got to leaf, add it to the dictionary
94          # if not check the right branch in recursive way
95          if type(right) is not tuple:
96              binary_info = (len(n + "1"), int(n + "1", 2))
97              new_codebook[right] = binary_info
98          else:
99              codebook(right, n + "1")
100
101         return new_codebook
102     return codebook(huff_tree)
103
104
105
106 def build_canonical_codebook(codebook):
107     """create a canonical codebook of the Huffman tree
108     the function recieve a huffman codebook and return a dictionary from item
109     to tuple of length and decimal value of the binary code represent the item
110     in canonical way
111     Args:
112     codebook - a dictionary - table of char: code pairs."""
113     # create a list from the codebook and sorted it from low value to high and
114     # low binary length to high
115     new_list = sorted([[leaf,codebook[leaf][0]] for leaf in codebook])
116     new_list.sort(key=lambda x: x[1])
117
118     # return empty codebook if tree is empty
119     if not new_list:
120         return {}
121     # take the length of the first item
122     length=new_list[0][1]
123     # calculate a new binary code the first item
124     code = "0" + ''.join("0" for i in range(length - 1))
125     # create  new dictonary with the first item with new values
126     canonical_codebook={new_list[0][0]: (length,int(code,2))}
127     # run for every item from the second one
```

What are "bin_item" and "codebook" in this context? They aren't function arguments, it only has one.
-1

There's no need to define codebook inside build_codebook. And EVERY function needs a docstring.
-2

"n" is not an acceptable name.
-2

What does "canonical codebook" mean? You should explain briefly here.

```
128     for item in new_list[1:]:
129         # calculate a new binary code the item
130         code = bin(int(code,2)+1)[2:]
131         # add 0 to the end of the new code if it's length smaller then
132         # the previus item code's
133         if len(code) < length:
134                code=code.zfill(length)
135         # take the current length
136         length=item[1]
137         # add 0 to the begining of the new code if it's length smaller then
138         # the original item code's
139         code=code+"".join("0" for i in range(length-len(code)))
140         # add the new dictionary the item with new values
141         canonical_codebook[item[0]] = (length, int(code, 2))
142 
143     return canonical_codebook
144 
145 
146 def build_decodebook(codebook):
147     ''' return a dictionary from tuple of length and decimal value of the
148     binary code to item built from a dictionary of item to tuple of length
149     and decimal value of the binary code
150     rgs:
151     codebook - a dictionary - table of char: code pairs."""
152     '''
153     # new dictionary
154     decodebook = {}
155     # add the new dictionary the value as key and key as value
156     for item in codebook:
157         decodebook[codebook[item]]=item   return {value: key for key, value in codebook.items()}
158     return decodebook
159 
160 def compress(corpus, codebook):
161     """the function create an iterator of 0 or 1 as ints, after iterating on
162     corpus input.
163 
164     Args:
165     corpus - a sequence of chars by a iterator.
166     codebook - a dictionary - table of char: code pairs. """
167 
168     # run for every item in corpus
169     for item in corpus:
170         # take the length and decimal values according to the codebook
171         length = codebook[item][0]
172         num = codebook[item][1]
173         # convert to binary
174         binary = bin(num)[2:].zfill(length)
175         # iterator?????
176         for char in binary:
177             yield  int(char)
178 
179 def decompress(bits, decodebook):
180     """the function run over the decoding bits of coded bits input
181     and create an iterator of 0 or 1 as an int.
182 
183     Args:
184     bits - an iterable, a sequence of coded bits each is an int 0 or 1.
185     decodebook - a dictionary, a decoded one"""
186     # set a new binary code
187     binary = ""
188     # run for every bit
189     for bit in bits:
190         # add the current binary code the next bit
191         binary = binary + str(bit)
192         # create a tuple of length and decimal value of the binary code
193         decode = (len(binary), int(binary, 2))
194         # if the binary code is in the decodebook return his value and reset
195         # the binary code
```

```
196             if decode in decodebook:
197                 yield decodebook[decode]
198                 binary = ""
199
200
201     def pad(bits):
202         """the function run over each eight sequence bits out of the input,
203         adds the 1 as a final bit and appends zeros for the total length be
204         divided by 8. the function create an iterator of 0 or 1 as an ints.
205
206         Args:
207         bits - an iterable, a sequence of coded bits each is an int 0 or 1."""
208         # set a new binary code
209         binary = ""
210         # run for every bit
211         for bit in bits:
212             binary = binary + str(bit)
213             # when binary code have length of 8 return the decimal value and reset
214             # the binary code
215             if len(binary) == 8:
216                 yield int(binary, 2)
217                 binary = ""
218         # for the last bits, add single 1 and zeros until binary have length of 8
219         binary = binary + "1"
220         while len(binary) != 8:
221             binary = binary + "0"
222         # return the last binary code
223         yield int(binary, 2)
224
225     def unpad(byteseq):
226         """the function run over all bytes of input, taking off the '0' and '1'
227         on top of it and create an iterator of 0 or 1 as ints.
228
229         Args:
230         byteseq - an iterator, a sequence of bytes."""
231         # set a boolin for the first byte
232         first = True
233         # run for every byte
234         for byte in byteseq:
235             # for the first byte get his binary value and finish the corrent loop
236             if first:
237                 binary = bin(byte)[2:].zfill(8)
238                 first = False
239                 continue
240
241             # return every single bit as iterator
242             for bit in binary:
243                 yield int(bit)
244             # get the next byte binary value
245             binary = bin(byte)[2:].zfill(8)
246         # for the last byte, find the last "1" digit index
247         index = -1
248         bit = binary[index]
249         while bit != "1":
250             index -= 1
251             bit = binary[index]
252         # return the bits up to the last "1" digit
253         for bit in binary[:index]:
254                 yield int(bit)
255
256     def join(data, codebook):
257         """the function run over the bytes of input (first codebook then data)
258         and create an iterator of the codebook vals which appear, then  the
259         data items.
260
261         Args:
262         data - an iterator, a sequence of bytes.
263         codebook - a canonical code table, the output of
```

8 and 2 are magic numbers.
-1

```
264         build_canonical_codebook."""
265         for key in range(256):
266             if key in codebook:
267                 yield codebook[key][0]
268             else:
269                 yield 0
270         for data_0 in data:
271             yield data_0
272
273     def split(byteseq):
274         """that function split the output of the function join to data and codebook
275         the function return a tuple which is consist of a dictionary - canonical
276         coding table and an iterator which iterate over rest of byteseq as
277         byte sequent.
278
279         Args:
280         byteseq - an iterator, a sequence of bytes."""
281         index = 0
282         codebook = {}
283         data = []
284         for byte in byteseq:
285             if index < 256:
286                 if byte != 0:
287                     codebook[index] = (byte, 0)
288                 index += 1
289             else:
290                data.append(byte)
291         codebook = build_canonical_codebook(codebook)
292         return iter(data), codebook
293
```

256 is a magic number.
-1