a번 Source A ~ Bern(1/4)

```python
from collections import Counter  # 원소 카운트를 위한 라이브러리
import heapq
from heapq import heappush, heappop  # 최소값을 뽑기 위해서 힙 사용
import numpy as np  # 배열 쉽게 다루기 위해 numpy사용
import math
import tree_class

source_a = np.random.binomial(1, 1/4, size=1000)  # 소스 바이너리 분포로 만들기
count_prob = np.unique(source_a, return_counts = True)
print(count_prob ) # 소스 안에 0과 1의 갯수 출력
One_prob = count_prob[1][1]/1000
Zero_prob = count_prob[1][0]/1000

source_entropy = Zero_prob*math.log(1/Zero_prob,2) + One_prob*math.log(1/One_prob,2)
print("Original source entropy bits:", source_entropy*1000)

# calculate empirical probabilities
# 4개씩 끊어서 총 몇 번 나오는지 카운트
counts = Counter([tuple(source_a[i:i+4]) for i in range(0, len(source_a), 4)])
# 각각 나온 횟수 다 합해서 횟수 세기
total_counts = sum(counts.values())
# 각각 나온 횟수를 전체 횟수로 나눠서 확률 구하기.
probabilities = {symbol: count/total_counts for symbol, count in counts.items()}
s_extention_entropy = 0
```

바이너리 소스를 만든다. 몇 개씩 나오는지 센다. (실제로 1/4, 3/4인지 검증을 위해서)

소스 확장하기 전에 엔트로피를 계산한다.

소스를 4개씩 끊어서 카운트한다. + 각각을 전체 개수로 나눠서 확률을 구한다.

```python
for key, value in probabilities.items():
    print("Symbol:", key, "Prob:", value)
    s_extention_entropy += value*math.log(1/value,2)
print('\n')
print("source extention entropy:", s_extention_entropy * 250)

# build Huffman tree
# 이렇게 넣으면 알아서 최상위 노드에 가장 작은 확률 값을 갖는 심볼이 들어감.
# ""은 코드워드 넣을 공간
heap = [tree_class.Node(symbol, prob) for symbol, prob in probabilities.items()]

heapq.heapify(heap)
# while문 돌려서 가장 작은 확률 값을 갖는 노드에 0 그 다음 노드에 1 부여하고 다시 힙에 넣어주기.
print("build Huffman tree", "\n")
while len(heap) > 1:
    min0 = heappop(heap); min1 = heappop(heap)  # 힙에 있는 가장 작은 노드 빼서 min0과 min1에 각각 넣기.
    min0.code = '0'; min1.code = '1' # 각각에 코드 추가해주기.
    total_prob = min0.freq + min1.freq # 각각의 확률 합쳐서
    heappush(heap, tree_class.Node(None, total_prob, None, min0, min1))  # 힙에 다시 넣어주기

```

확장된 소스의 엔트로피를 구한다. 심볼과 확률을 바탕으로 힙을 만든다. (힙 만든 이유 : 힙은 최소 값이 항상 루트에 있기 때문에 허프만 트리를 만들 때 편해지므로)

허프만 트리를 만든다. (만드는 방법 : 힙에서 하나씩 빼서 루트 노드를 중심으로 가장 작은 값을 왼쪽 노드로 만들고 코드를 0으로 매핑한다. 그 다음 값은 오른쪽 노드로 만들고 1을 매핑한다. 다시 두 확률을 더해서 루트 노드에 써주고 힙에 넣는다. 힙에 하나의 원소만 남을 때까지 반복

```
# generate codebook
a = []
def gen_codebook(node):  # 루트 노드에서 시작해서 리프노드까지 순회하며 코드북 만들기.
    if node.flag == False:
        if node.code != None:
            if node.left: # 자신의 노드의 왼쪽 노드에는 자신의 코드 더하기 0
                node.left.code =  node.code + "0"
                gen_codebook(node.left)
            if node.right: # 자신의 오른쪽 노드에는 자신의 코드 더하기 1
                node.right.code = node.code + "1"
                gen_codebook(node.right)
            else: # 만약 이 노드가 리프 노드이면 자신의 심볼과 코드를 코드북에 어펜드
                node.flag = True
                a.append((node.symbol, node.code))
        else: # 첫 시작에는 (=루트 노드에는) 코드가 없어서 따로 시작해야함.
            if node.left:
                node.left.code = "0"
                gen_codebook(node.left)
            if node.right:
                node.right.code = "1"
                gen_codebook(node.right)
    else:
        pass

```

코드북 만드는 방법
노드를 한 계층 내려가면서 코드를 누적해서 쌓아가는 방식으로 만들기.
자기 자식 노드는 자신의 코드 + 0 or 1 (왼쪽에 있는 자식은 0, 오른쪽에 있는 자식은 1 부여)
만약에 자식이 없으면 자신의 심볼과 코드를 반환.

```
gen_codebook(heap[0]) # 코드북 생성 함수에 힙을 넘겨주기
a.sort(key=lambda e:len(e[1])) # 코드북 원소 보기 좋게 정렬
codebook = dict(a) # 코드북을 사전 형으로 만들기
print("Codebook")
for key, value in codebook.items():
    print(key, value)
print('\n')
```

코드북을 만드는 함수에 heap의 루트 노드 넘겨주기.

```
78  # encode sequence
79  binary_string = "".join([codebook[tuple(source_a[i:i+4])] for i in range(0, len(source_a), 4)])
80
81  decoded_bit = []
82  head = heap[0]
83  for i in range(0,len(binary_string)):
84      b = binary_string[i]
85      if b == '0': # 코드가 0이면? 왼쪽으로
86          head = head.left
87          if head.left is None and head.right is None: # leaf
88              decoded_bit.extend(list(head.symbol))
89              head = heap[0]
90      else: # 코드가 1이면 오른쪽으로
91          head = head.right
92          if head.left is None and head.right is None: # leaf
93              decoded_bit.extend(list(head.symbol))
94              head = heap[0]
95
```

만들어진 코드북을 바탕으로 소스코드를 인코딩하고 디코딩하기.

디코딩하는 방법 = 인코딩 비트가 0이면 왼쪽으로 1이면 오른쪽으로 트리를 탐색하면서 만약에 리프 노드면 자신의 코드를 반환하도록 구성

```
96  decoded_bit = [str(e) for e in decoded_bit]
97  decoded_bit = ''.join(decoded_bit)
98  source_a = ''.join([str(e)for e in source_a])
99  print("Encoded_bit", binary_string ,sep='\n')
100
101 print("Decoded_bit", decoded_bit, sep='\n')
102
103 print("Source bit:", source_a,  sep='\n')
104 print("\n")
105 print("Are decoded bits and source bits same? ->", str(decoded_bit == source_a))
106 print("\n")
107
108 print("Original source entropy bits:", math.ceil(source_entropy*1000))
109 print("Source extention entropy bits:", math.ceil(s_extention_entropy * 250))
110 print("Compressed string length:", len(binary_string))
111
112 print("Compression ratio: ", round(len(binary_string)/len(source_a), 2)*100, "%")
113 print('\n')
```

디코딩된 비트를 스트링으로 바꾸고 합치기. 소스코드로 똑같이 만들어서 비교하기

결과

```
(array([0, 1]), array([742, 258]))
Original source entropy bits: 823.7133233102513
Symbol: (0, 0, 0, 1) Prob: 0.148
Symbol: (0, 0, 0, 0) Prob: 0.288
Symbol: (0, 0, 1, 0) Prob: 0.076
Symbol: (1, 0, 0, 0) Prob: 0.112
Symbol: (0, 1, 0, 0) Prob: 0.124
Symbol: (1, 1, 0, 0) Prob: 0.052
Symbol: (1, 1, 0, 1) Prob: 0.012
Symbol: (1, 0, 0, 1) Prob: 0.024
Symbol: (1, 0, 1, 0) Prob: 0.028
Symbol: (1, 1, 1, 1) Prob: 0.012
Symbol: (0, 1, 1, 1) Prob: 0.02
Symbol: (0, 0, 1, 1) Prob: 0.044
Symbol: (1, 1, 1, 0) Prob: 0.012
Symbol: (0, 1, 0, 1) Prob: 0.032
Symbol: (0, 1, 1, 0) Prob: 0.016


source extention entropy: 806.3700237578682
build Huffman tree

Codebook
(0, 0, 0, 0) 11
(1, 0, 0, 0) 010
(0, 1, 0, 0) 011
(0, 0, 0, 1) 101
(0, 0, 1, 1) 0001
(1, 1, 0, 0) 0010
(0, 0, 1, 0) 1001
(0, 1, 1, 1) 00000
(1, 0, 0, 1) 00110
(1, 0, 1, 0) 10000
(0, 1, 0, 1) 10001
(1, 1, 0, 1) 000010
(1, 1, 1, 0) 000011
(1, 1, 1, 1) 001110
(0, 1, 1, 0) 001111


Encoded_bit
1011110011001010101110110111111111100101001011011111100100110110000010110011001101001001011111101
1000001010110111011111111110010111110111101001110101000000001100111111000011110111111001111111101
0100001010101101100110011111110111010000111100011011110100100101111100000101110011000110101010101
1000111111100000010011101101000010101101011000000001100101100100100110011111100110010111110010000
1100101111000000010011111010101100010011101000110011011011000100111110110011110111011101011010111
1100111001010111110001011000001011100111101111100010101110001010000101111100000101100110101110001
1101011001001000100010101100011101000100101110000010111001101000111000110111000010001101010110000
1001010110000000011111001010111101001101100010101100001110011110011111001101001001011011011010010 1
0101010101010111111011001101110001011001111
```

Decoded_bit
```
0001000000100010100000010000010001000000000000000010100010000000010000000000110001000100110100000
1001010010001000100000000000000001001010100000010001000001000000000000000010010000000001000000001
1111000101110011001000000000101000000000010000000010000000000000000000100010100001100000000010010 1
0100000000000100000111100000001100010000000111001000000000001010100000000001001010001100000010100
0011000000000000011110000100000100011101000100010100011100110010010011001000010001100010001001000
0000001011101100000000000011111000100000010000001010111110101001000010001010101100001001000000001
0000010000010100100000000000001000001100000100000000001101000111000100000110010000000101100000000
0101100000110100000010101000000010010001000011110001001011001100110000010101000010001100100000000
1010100000000010000100110000001100010000011011001000101000011110000011010001100000000110000010000
0001100100000011100000001110001000000010000000100001110010000000010001001000100000001000000110000
0001010000000000010010010000010101000110
```

```
Source bit:
0001000000100010100000010000010001000000000000000010100010000000010000000000110001000100110100000
1001010010001000100000000000000001001010100000010001000001000000000000000010010000000001000000011
1111000101110011001000000000101000000000010000000010000000000000000010001010000110000000010010010
0100000000000100000111100000001100010000000111001000000000001010100000000010010100011000000101000
0011000000000000011110000100000100011101000100010100011100110010010011001000010001100010001001000
0000001011101100000000000111110001000000100000010101111101010010000100010101011000010010000000010
0000010000010100100000000000001000001100000100000000001101000111000100000110010000000101100000000
0101100000110100000010101000000010010001000011110001001011001100110000010101000010001100100000000
1010100000000010000100110000001100010000110110010001010000111100000110100011000000001100000100000
0001100100000011100000001110001000000010000000100001110010000000010001001000100000001000000110000
0001010000000000010010010000010101000110
```

```
Are both decoded bits and source bits same? -> True
```

```
Original source entropy bits: 824
Source extention entropy bits: 807
Compressed string length: 812
Compression ratio:  81.0 %
```

b) Source B markov chain transition prob [0.75 0.25 // 0.25 0.75] -> 자기 자신과 동일한 값이 나올 확률 = 0.75

```
source_b = np.random.binomial(1, 1/2, size=1)  # Markov chain으로
source_b = source_b.tolist()


for i in range(1,1000):
    new_b = np.random.binomial(1, 3/4, size=1)
    if new_b == 1: # 같은 값으로 트랜지션
        source_b.append(1&source_b[i-1])
    else: # 다른 값으로 트랜지션
        source_b.append(1^source_b[i-1])
```

1. 먼저 1/2 확률로 아무거나 하나를 뽑는다.
2. Bern(1, 3/4, size = 1)을 실행해서
    1. 1이 뽑히면 자기 자신을 넣는다. (1과 OR해서 넣기)
    2. 0이 뽑히면 자기 자신을 flip해서 넣는다. (XOR 연산, 1 -> 0, 0 -> 1)
3. 반복한다.

```
(array([0, 1]), array([507, 493]))
Origianl source entropy bits: 999.8586112670831
Symbol: (0, 0, 0, 0) Prob: 0.24
Symbol: (1, 1, 1, 1) Prob: 0.184
Symbol: (1, 0, 0, 0) Prob: 0.06
Symbol: (1, 1, 1, 0) Prob: 0.1
Symbol: (0, 1, 1, 0) Prob: 0.012
Symbol: (0, 0, 0, 1) Prob: 0.068
Symbol: (1, 0, 0, 1) Prob: 0.032
Symbol: (0, 0, 1, 1) Prob: 0.056
Symbol: (1, 1, 0, 1) Prob: 0.02
Symbol: (0, 1, 1, 1) Prob: 0.084
Symbol: (1, 1, 0, 0) Prob: 0.052
Symbol: (0, 1, 0, 0) Prob: 0.016
Symbol: (0, 1, 0, 1) Prob: 0.024
Symbol: (0, 0, 1, 0) Prob: 0.012
Symbol: (1, 0, 1, 0) Prob: 0.004
Symbol: (1, 0, 1, 1) Prob: 0.036


source extention entropy: 847.9562868515703
build Huffman tree

Codebook
(0, 0, 0, 0) 10
(1, 1, 1, 0) 001
(1, 1, 1, 1) 111
(1, 1, 0, 0) 0001
(0, 0, 1, 1) 0101
(1, 0, 0, 0) 0110
(0, 0, 0, 1) 1100
(0, 1, 1, 1) 1101
(1, 1, 0, 1) 00001
(0, 1, 0, 1) 01000
(1, 0, 0, 1) 01110
```

```
(1, 0, 1, 1) 01111
(1, 0, 1, 0) 000000
(0, 0, 1, 0) 000001
(0, 1, 1, 0) 010010
(0, 1, 0, 0) 010011


Encoded_bit
10101110110001010010010010110000110110011111111101110101001010110110011100001010100111010111011
11000001110100101001111111111111110110101010010110001110110001011010010011100100011101101100001101
00001011011010000011111111000111000010111001001010010000000101101011001011000001011010110011010
10011010001000000010101100101011001001110110100100110010000110100010101011011001000110110101000 1
11011100001111101011111101110110111101110110100001111000001110101110001101011000001100101110111 0
11111001011110111111100011110011010011110111111110011011110010011010111101000001111011110110101 0
01010001101101000110110111100011111011101110000110101011000011101111011111111010101101111110100 0
11001011110001101111000010101111111110001100000101111011011110010110010100111001100001110111100 1
11111101111010000111101111010111111100001110110101101111111000111011111011000000111001010
```

```
Decoded_bit
00000000111110001110011001100001111000000001111111111111100100000000001110000001111111010011111 0
01111001111100011100011111100100111111111111111111000000000000000000110000111001110000111010000000
01000000010101110000000111100000000011001000011100101111111100001110000111101001011000000101110 1
10000000000100000001110010000000000101111110111000000000111010100011100000111000010000000111111 0
11100000010110000000111000111000000100001110000001110101111110011100111100000000111111111001011 1
11111001011111011111001001111001111000000000000111000000000110111011111110000001111111101111110 0
11111110000000001011101111111110000011110000010000111111010111101111101110000000000000000000000 0
11100000011111000000011111111100111100001111100111000000000000000001111001111111101111110000000 0
00000111111101111100000000111111110000001111000011000011111111110001100011011011100011110000001 1
00000011111000010001111001111111111011111111101101011011101100111111111111101111110000000011111 1
11111100011111110000001001000010000000
```

```
Source bit:
00000000111110001110011001100001111000000001111111111111100100000000001110000001111111010011111 0
01111001111100011100011111100100111111111111111111000000000000000000110000111001110000111010000000
01000000010101110000000111100000000011001000011100101111111100001110000111101001011000000101110 1
10000000000100000001110010000000000101111110111000000000111010100011100000111000010000000111111 0
11100000010110000000111000111000000100001110000001110101111110011100111100000000111111111001011 1
11111001011111011111001001111001111000000000000111000000000110111011111110000001111111101111110 0
11111110000000001011101111111110000011110000010000111111010111101111101110000000000000000000000 0
11100000011111000000011111111100111100001111100111000000000000000001111001111111101111110000000 0
00000111111101111100000000111111110000001111000011000011111111110001100011011011100011110000001 1
00000011111000010001111001111111111011111111101101011011101100111111111111101111110000000011111 1
11111100011111110000001001000010000000

Are both decoded bits and source bits same? -> True


Original source entropy bits: 1000
Source extention entropy bits: 848
Compressed string length: 859
Compression ratio:  86.0 %
```