Q1: Huffman Encoding & Side Information Effects

In [ ]:
```
english::Vector{String} = ["A", "B", "C", "D", "E", "I"];
greek::Vector{String} = ["α", "β", "γ", "δ", "ϵ", "ϕ"];
alphabets::Vector{String} = [];

append!(alphabets, english)
append!(alphabets, greek)
```

```
12-element Vector{String}:
 "A"
 "B"
 "C"
 "D"
 "E"
 "I"
 "α"
 "β"
 "γ"
 "δ"
 "ϵ"
 "ϕ"
```

Define the Probability Mass Function of each alphabet

In [ ]:
```
P = Dict();
for alphabet in alphabets
    if string(alphabet) ∈ greek
        P[alphabet] = 1 // 12;
    elseif string(alphabet) ∈ ["A", "E", "I"]
        P[alphabet] = 1 // 9;
    else
        P[alphabet] = 1 // 18;
    end
end

P
```

```
Dict{Any, Any} with 12 entries:
  "C" => 1//18
  "ϵ" => 1//12
  "δ" => 1//12
  "B" => 1//18
  "A" => 1//9
  "ϕ" => 1//12
  "D" => 1//18
  "α" => 1//12
  "E" => 1//9
  "γ" => 1//12
  "I" => 1//9
  "β" => 1//12
```

Create Block Coding, followed by Huffman Tree

In [ ]:
```
include("./block_code.jl")
include("./tools.jl")

# Sort Symbols and Probabilities & Create Block Codes
sorted_probabilities, sorted_symbols, P_sorted_dict = sort_prob(P);
block2_codes, prob_block2_codes = create_block_coding(sorted_symbols, sorted_probabilities, 2);
block3_codes, prob_block3_codes = create_block_coding(sorted_symbols, sorted_probabilities, 3);

prob_block2_codes, block2_codes, P2_sorted_dict = sort_prob(prob_block2_codes, block2_codes);
prob_block3_codes, block3_codes, P3_sorted_dict = sort_prob(prob_block3_codes, block3_codes);
```

In [ ]:
```
include("./huffman.jl")

huffman_tree = construct_huffman_tree(P_sorted_dict);
huffman2_tree = construct_huffman_tree(P2_sorted_dict);
huffman3_tree = construct_huffman_tree(P3_sorted_dict);

encoder1 = Dict();
encoder2 = Dict();
encoder3 = Dict();

build_encoder(huffman_tree, "", encoder1);
build_encoder(huffman2_tree, "", encoder2);
build_encoder(huffman3_tree, "", encoder3);
```

Q1a) Huffman Encoder and its Expected length

In [ ]:
```
encoder1
```

```
Dict{Any, Any} with 12 entries:
  "C" => "1000"
  "ϵ" => "1011"
  "δ" => "1100"
  "B" => "1001"
  "A" => "001"
  "ϕ" => "1101"
  "D" => "1010"
  "α" => "1110"
  "E" => "010"
  "γ" => "1111"
  "I" => "011"
  "β" => "000"
```

In [ ]:
```
function report_expected_len(encoder, P, n)
    len = expected_length(encoder, P);
    len = float(len)
    println("Huffman Encoder with block length $n has expected length $(len) bits")
end

report_expected_len(encoder1, P_sorted_dict, 1)
report_expected_len(encoder2, P2_sorted_dict, 2)
report_expected_len(encoder3, P3_sorted_dict, 3)
```

```
Huffman Encoder with block length 1 has expected length 3.5833333333333335 bits
Huffman Encoder with block length 2 has expected length 7.114197530864198 bits
Huffman Encoder with block length 3 has expected length 10.664566186556927 bits
```

In [ ]:
```
# mcmillan_inequality(encoder1)
# mcmillan_inequality(encoder2)
# mcmillan_inequality(encoder3)
```

Create Side Information Dictionary

In [ ]:
```
language_dict = Dict();
for alphabet in alphabets
    if alphabet ∈ english
        language_dict[alphabet] = "e";
    elseif alphabet ∈ greek
        language_dict[alphabet] = "g";
    end
end
```

In [ ]:
```
include("./side_information_table.jl")
side1_codebook = create_side_information_codebook(sorted_symbols, language_dict)
side2_codebook = create_side_information_codebook(block2_codes, language_dict)
side3_codebook = create_side_information_codebook(block3_codes, language_dict)
```

```
Dict{Any, Any} with 1728 entries:
  "DBϵ" => "eeg"
  "IϕC" => "ege"
  "DβI" => "ege"
  "δαϕ" => "ggg"
  "Bββ" => "egg"
  "αED" => "gee"
  "DIβ" => "eeg"
  "DCα" => "eeg"
  "δϵα" => "ggg"
  "EBD" => "eee"
  "IEB" => "eee"
  "βCϵ" => "geg"
  "BEI" => "eee"
  "IBB" => "eee"
  "ϕαD" => "gge"
  "DCγ" => "eeg"
  "ϵβC" => "gge"
  "δγδ" => "ggg"
  "βαβ" => "ggg"
  ⋮     => ⋮
```

In [ ]:
```
side_block1, side_prob1 = side_information_table(sorted_symbols, sorted_probabilities, side1_codebook);
side_block2, side_prob2 = side_information_table(block2_codes, prob_block2_codes, side2_codebook);
side_block3, side_prob3 = side_information_table(block3_codes, prob_block3_codes, side3_codebook);
```

In [ ]:
```
#=
side_2_block    : Dictionary mapping
                    Side information block symbols => Block Symbols
side_2_block    : Dictionary mapping
                    Side information block symbols => Block Symbols' Conditional Probablity
```

```
Output:
unique_trees    : Dictionary mapping
                   Side information block symbols => Huffman Tree
unique_encoders : Dictionary mapping
                   Side information block symbols => Huffman Encoder given Side Information
=#
function encode_w_side_info(side_2_block, side_2_prob)
    unique_trees = Dict();
    unique_encoders = Dict();
    for (side_info, conditional_probabilities) in side_2_prob
        conditioned_symbol = side_2_block[side_info];
        ___, ____, P_dict = sort_prob(conditional_probabilities, conditioned_symbol);
        tree = construct_huffman_tree(P_dict);
        encoder = Dict();
        build_encoder(tree, "", encoder);

        unique_trees[side_info] = tree;
        unique_encoders[side_info] = encoder;
    end
    return unique_trees, unique_encoders
end

side_tree1, side_encoder1 = encode_w_side_info(side_block1, side_prob1);
side_tree2, side_encoder2 = encode_w_side_info(side_block2, side_prob2);
side_tree3, side_encoder3 = encode_w_side_info(side_block3, side_prob3);
```

In [ ]:

```
#=
side_2_encoder : Dictionary mapping
                   Side information block symbols => Codeblock Encoder given Side Information
=#
function unroll_side_info_encoders(side_2_encoder)
    encoder_side_info = Dict();
    for (side_info, encoder) in side_2_encoder
        merge!(encoder_side_info, encoder)
    end

    return encoder_side_info
end
```

```
unroll_side_info_encoders (generic function with 1 method)
```

Q1b) Huffman Encoder given Side Information

In [ ]:

```
u_side_encoder1 = unroll_side_info_encoders(side_encoder1);
u_side_encoder2 = unroll_side_info_encoders(side_encoder2);
u_side_encoder3 = unroll_side_info_encoders(side_encoder3);

report_expected_len(u_side_encoder1, P_sorted_dict, 1)
report_expected_len(u_side_encoder2, P2_sorted_dict, 2)
report_expected_len(u_side_encoder3, P3_sorted_dict, 3)
```

```
Huffman Encoder with block length 1 has expected length 2.611111111111111 bits
Huffman Encoder with block length 2 has expected length 5.151234567901234 bits
Huffman Encoder with block length 3 has expected length 7.688614540466392 bits
```

In [ ]:

```
u_side_encoder1
```

```
Dict{Any, Any} with 12 entries:
  "C" => "101"
  "ϵ" => "110"
  "δ" => "00"
  "B" => "100"
  "A" => "111"
  "ϕ" => "101"
  "α" => "111"
  "D" => "110"
  "E" => "01"
  "γ" => "100"
  "I" => "00"
  "β" => "01"
```

In [ ]:

```
decode_huffman(side_tree3["ege"], "0001101")
```

```
Going left
Going left
Going left
Going Right
Going Right
Going left
Going Right
Successfuly Decoded: AδA
"AδA\0"
```

Q2: Constructing Viterbi Decoder

1. First Define the Emission Matrix based on θ_t & Z_t
2. For Transition Matrix based on θ_t, it is an identity matrix except at $t = n$
3. For $t = n$ case, we define another Transition Matrix that has the Source Symbol X_0 probability properties
4. Define the sequence $Z_{0:5} =$" $A\beta\beta\beta DD$ "
5. Run Viterbi Decoder based on different n values

In [ ]:

```
#=
Input:
probabilities   : Vector containing probabilities of source
ϵ               : Observation Error on observing Hidden State

Output:
emission_matrix : Emission Matrix describing Hidden State -> Observation State
=#
function create_emission_matrix(probabilities, ϵ = 0.02)
    num_states = length(probabilities);
    emission_matrix = zeros(num_states, num_states);

    for (row_idx, row) in enumerate(eachrow(emission_matrix))
        normalization = (1//1) - probabilities[row_idx];
        for (entry_idx, prob) in enumerate(row)
            if entry_idx == row_idx
                row[entry_idx] = (1 - ϵ); # * P(itself) // P(itself) -> 1
            else
                row[entry_idx] = ϵ * (probabilities[entry_idx] // normalization)
            end
        end
        # println("$row_idx : $row w/ norm = $normalization")
    end

    return emission_matrix
end

#=
Used for resetting at t = n
Input:
probabilities   : Vector containing probabilities of source

Output:
reset_transition_matrix : Transition Matrix of Hidden States that resets back to Initial Condition
=#
function source_transition_matrix(probabilities)
    num_states = length(probabilities);
    reset_transition_matrix = zeros(num_states, num_states);
    for (row_idx, row) in enumerate(eachrow(reset_transition_matrix))
        reset_transition_matrix[row_idx, :] = probabilities;
    end
    return reset_transition_matrix
end
```

```
source_transition_matrix (generic function with 1 method)
```

In [ ]:

```
include("./viterbi.jl")
using LinearAlgebra
num_states = length(sorted_probabilities);
θ_transition = Matrix(1I, num_states, num_states);
Z_sequence = ["A", "β", "β", "β", "D", "D"];
ϵ = 0.02;

emission_matrix = create_emission_matrix(sorted_probabilities, ϵ);
reset_transition_matrix = source_transition_matrix(sorted_probabilities);
```

In [ ]:

```
x6, T1_6, T2_6 = Viterbi(Z_sequence, θ_transition, emission_matrix,
                            sorted_symbols, sorted_symbols, sorted_probabilities,
                            reset_transition_matrix, 6);
```

In [ ]:

```
x3, T1_3, T2_3 = Viterbi(Z_sequence, θ_transition, emission_matrix,
                            sorted_symbols, sorted_symbols, sorted_probabilities,
                            reset_transition_matrix, 3);
```

Q2b) Compute MAP estimator when $n = 6$ & $n = 3$

In [ ]:

```
println("Decoded Sequences:")
println("For n = 3 => $x3")
println("For n = 6 => $x6")
println("MAP Probability: ")
```

```julia
println("For n = 3 => MAP probability =  $(maximum(T1_3[:, 6]))")
println("For n = 6 => MAP probability = $(maximum(T1_6[:, 6]))")
```

```
Decoded Sequences:
For n = 3 => Any["β", "β", "β", "D", "D", "D"]
For n = 6 => Any["β", "β", "β", "β", "β", "β"]
MAP Probability:
For n = 3 => MAP probability =  1.8268333531392355e-8
For n = 6 => MAP probability = 2.793605847269759e-10
```

In [ ]:

```
```