Task 9 Morse coder

Adrian Jonsson Sjödin

Spring Term 2023

Introduction

In this task we implement a Morse coder that can take a text input and encode it into morse code. We also implement a decoder that can take a morse signal and output the message it contains.

The requirement on the encoder is that it should have a time complexity of $\mathcal{O}(n \cdot m)$, where n is the length of the message and m is the length of the morse codes. The lookup function in the encoder should have a time complexity of $\mathcal{O}(log(k))$, where k is the number of letters in the alphabet, or $\mathcal{O}(1)$

The requirement on the decoder is that it should use a data structure that will give an $\mathcal{O}(m)$ lookup operation. In other words it should be proportional to the length of the morse code.

Method

We start with creating the functions needed for encoding a message into morse code. The first thing we do is represent the morse code alphabet as a map (see fig. 1 who show the first few lines of it). They will be used when we recursively encode each character. First however we need to convert the text we receive to a charlist and convert all character to lowercase (see fig. 2).

When this is done we can start the encoding. Since we want to achieve a low time complexity we use a tail recursive strategy when encoding each character (see fig. 3). The `encode_recursive/2` function processes each character in the charlist exactly once. For each character, it looks up its morse code in the map using `Map.get()`. This lookup process takes $\mathcal{O}(log(k))$ time on average, where k is the number of keys in the map seen in fig. 1. However since k is constant (in this case 36), the lookup operation can be considered a constant time operation and thus is $\mathcal{O}(1)$.

Overall the time complexity of this implementation is $\mathcal{O}(n \cdot m)$, where n is the length of the input string and m is the average length of the Morse code for each character. In the worst case, m is 4, which means the time complexity is $\mathcal{O}(4n)$, or simply $\mathcal{O}(n)$.

Code Overview 1: Morse code map

```
@morse_codes %{
    ?a => ".-",
    ?b => "-...",
    ?c => "-.-.",
    ?d => "-..",
    ?e => ".",
```

Code Overview 2: The `encode/1` function

```
def encode(text) do
    text
    |> String.downcase()
    |> String.to_charlist()
    |> encode_recursive([])
end
```

Code Overview 3: The `encode_recursive/2` function

```
defp encode_recursive([], acc), do: acc
defp encode_recursive([char | rest], acc) do
 case Map.get(@morse_codes, char) do
 # If the character doesn't have a corresponding Morse code, skip it
  nil ->
   encode_recursive(rest, acc)
  morse_code ->
   case acc do
   # If this is the first character being encoded, add its Morse code to
   # the accumulator
     [] -> encode_recursive(rest, morse_code |> String.to_charlist())
     # If this is not the first character being encoded, add a space character
     # and the Morse code to the accumulator
     _ -> encode_recursive(rest, (acc ++ [32] ++ morse_code) |> List.to_charlist())
   end
 end
end
```

Next we need to create the decoding functions. The input to this function will be a list of characters representing the morse code signal (dots, dashes, and spaces). The plan is to make use of the provided decoding tree to recursively decode the signal. To do this we first generate the provided tree and then pass it, the morse signal and an empty list as arguments to a private `decode/3` function (see fig. 4) which will process the signal recursively by calling the `decode_char/2` (see fig. 5) function. This function matches the morse code signal with either a dot or dash (`.` or `-`) and a corresponding sub-tree in the binary tree. It then recursively calls `decode_char/2` with the remaining signal and the corresponding sub-tree. When the end of the signal is reached, it returns a tuple containing the decoded character and the remaining signal. If the signal cannot be matched, the function returns `:no`. The decode function then appends the decoded character to a list that is initially empty, until the end of the signal is reached. If the signal cannot be decoded, an error message is printed to the console.

To summarize the data structure used is the one that was provided, i.e. a decoding tree, which is built using a combination of tuples and atoms. The tuples represent nodes in the tree and the atoms are used to represent special states in the tree, such as the "no match" state (`:na`). The lookup function is `decode_char/2`, which takes a Morse signal and a decoding tree as input and recursively traverses the tree to decode the signal. The function returns a tuple {`char, rest`} if a character is successfully decoded from the signal, or the atom `:no` if the signal cannot be decoded.

Code Overview 4: The `decode/1` and `decode/3` functions

```
def decode(morse_signal) do
    # Build the decoding tree
    tree = decode_table()
    # Call the private decode function with the initial accumulator
    # set to an empty list
    decode(morse_signal, tree, [])
end

defp decode([], _, acc), do: acc

defp decode(morse_signal, tree, acc) do
  # Call the private decode_char function with the current Morse
  # signal and the decoding tree
  case decode_char(morse_signal, tree) do
    # If the decode_char function returns :no, it means the Morse
    # signal cannot be decoded
    :no ->
      :io.format("error: ~w\n", [Enum.take(morse_signal, 10)])
      acc

    # If the decode_char function returns a tuple {char, rest}, it
```

```
    # means a character was successfully decoded
    {char, rest} ->
      # Call the decode function recursively with the remaining Morse
      # signal and the updated accumulator
      decode(rest, tree, acc ++ [char])
  end
end
```

Code Overview 5: The `decode_char/2` function

```
defp decode_char([], {_, char, _, _}), do: {char, []}
defp decode_char([?- | morse_signal], {_, _, long, _}) do
  # If the next signal is a dash (-), follow the long path in the decoding tree
  decode_char(morse_signal, long)
end

defp decode_char([?. | morse_signal], {_, _, _, short}) do
  # If the next signal is a dot (.), follow the short path in the decoding tree
  decode_char(morse_signal, short)
end

# If the Morse signal is empty or the decoding tree is nil, return :no
defp decode_char(_morse_signal, nil), do: :no
# If the next signal is a space and the decoding tree is in the "no match" state,
# return :no
defp decode_char([?\s | _morse_signal], {_, :na, _, _}), do: :no
# If the next signal is a space and the decoding tree is in a character state,
# return the decoded character and the remaining Morse signal
defp decode_char([?\s | morse_signal], {_, char, _, _}), do: {char, morse_signal}
```

Result

In fig. 6 you see the result from encoding my name and decoding it as well as the two secret messages we were asked to decode.

Code Overview 6: Secret message and encoding and decoding of my name

```
iex> name = Morse.encode("Adrian Jonsson Sjodin")
'.- -.. .-. .. .- -. ..-- .--- --- -. ... ... --- -. ..-- ... .--- --- -.. .. -.'
iex> Morse.decode(name)
'adrian jonsson sjodin'
iex> secret1 = Morse.secret1()
'.- .-.. .-.. ..-- -.-- --- ..- .-. ..-- -... .- ... . ..-- .- .-. . ..-- -... .
.-.. --- -. --. ..-- - --- ..-- ..- ...'
```

```
iex> secret2 = Morse.secret2()
'.... - - .--. ... ---... .----- .----- .-- .-- .-- .-.-.- -.-- --- ..- - ..- -...
 . .-.-.- -.-. --- -- .----- .-- .- - -.-. .... ..--.. ...- .----. -.. .--.-- .....
  .---- .-- ....- .-- ----. .--.-- ..... --... --. .--.-- ..... ---.. -.-. .--.--
  ..... .----'
iex> Morse.decode(secret1)
'all your base are belong to us'
iex> Morse.decode(secret2)
'https://www.youtube.com/watch?v=d%51w4w9%57g%58c%51'
```

Discussion

As already discussed in the Method section the encoding fulfills the requirements on both points. So let us discuss the time complexity of the decoding here instead.

The time complexity should be $\mathcal{O}(n)$, where n is the length of the morse signal string. This because `decode` calls upon the function `decode_char` on each element of the string until it is fully decoded, or an error occur. Since each character is only visited once and the `decode_char` function is $\mathcal{O}(1)$, the time complexity of the decode function is $\mathcal{O}(n)$. The `decode_char` function is a constant time operation since it is a simple pattern-matching operation that does not depend on the length of the input. This means that the decoding also satisfies the requirements.

The full code can be found on my GitHub.