Time complexity analysis of crypto.sml

Lucas Arnström & Mikaela Eriksson

February 14, 2014

Contents

1 Preprocess **3**

2 Encryption & Decryption **3**

3 Keystream **5**

3.1 Preamble 5

3.2 Calculation 5

1 Preprocess

For this calculation, the n-parameter in T(n) is equal to the amount of characters in the string that is being processed.

When binding the value ”upper” it makes two passes over it. First it explodes the list, then it iterates over all characters from the explode function and converts them to uppercase. When binding the value ”cleaned”, it makes only one iteration over the elements in the upper value while removing any non letter characters. This process will remove some elements that has to be processed if it contains any illegal characters. When binding the last value ”return”, it makes four passes over the ”cleaned” value. First it retrieves the length of the list, then it concatenates it with what was returned from the xList function. Lastly, it calls the divideList function.

All this adds up to a total of six iterations over the initial input. All of the auxiliary functions used are linear, and for the sake of working out the worst case time complexity of the function let's assume that the size of the initial input does not change over the course of the preprocess function. By adding upp all of the time complexities of the auxiliary functions we can form the overall time complexity of preprocess:

$$T(n) = \Theta_1(n) + \Theta_2(n) + \Theta_3(n) + \Theta_4(n) + \Theta_5(n) + \Theta_6(n) \Rightarrow 6 \cdot \Theta(n) \Rightarrow \Theta(n) \tag{1}$$

The worst case time complexity of the function is thus:

$$\Theta(n) \tag{2}$$

2 Encryption & Decryption

Both the encrypt and decrypt functions make use of the same high-order function crypt in a similar way, thus the worst case time complexity will only need to be calculated for that function since it will apply equally to the other two functions.

The crypt function has three parameters, the function to use while iterating, the 2-dimensional char list that is going to be encrypted or decrypted, and the 2-dimensional char list keystream. The keystream and the list to be encrypted/decrypted are of the same size when the function has been called from the encrypt or decrypt functions, thus it can be viewed as a single sized input. This is what will be the n-parameter in T(n) when calculating the time complexity of the crypt function below.

The function iterates over each element at the same time for the two parameter lists, if one where to end before the other, it would simply stop the recursion, compile all of the char lists generated into a 2D char list containing all of those lists and return it with the constant time C_1.

The amount of sublists to package when returning is equal to the size of the smallest list. But as stated above they can be veiwed as having the same size, thus the amount of sublists to package is equal to n. When none of the two parameter lists are empty, it will extract the head of each list and call convertLetterToNum for those, the value returned from that call is fed into the listOp function that uses the function parameter for each element in its two list arguments. When the listOp function returns, the value returned is fed into the convertNumToLetter function, which does the exact same thing as its namesake but the other way around.

As stated before, when the crypt function is called from either the encrypt or decrypt function, the keystream and list to encrypt/decrypt will be of the same size. The sublists for both of them will always have a length of five each. This makes the calls to convertNumToLetter, listOp, and convertLetterToNum execute in the constant time C_2 independently of how many sublists there are in the keystream and encrypt/decrypt list.

The recursion can thus be specified as:

$$T(n) = \begin{cases} n + C_1 & \text{if } n = 0 \\ T(n-1) + C_2 & \text{if } n > 0 \end{cases} \tag{3}$$

To explain, when n is equal to zero, the recursion stops, compiles the n amount of lists into a 2-dimensional list and returns it with the constant time C_1. When n is equal to one or more, it makes the calls to the auxiliary functions with the constant time C_2 and continues the recursion. By recursivly expanding the equation we can form an idea of the time complexity of the function:

$$\begin{aligned} T(n) &= T(n - 1 \cdot 1) + 1 \cdot C_2 \\ &= T(n - 1 \cdot 2) + 2 \cdot C_2 \\ &= T(n - 1 \cdot 3) + 3 \cdot C_2 \\ &= T(n - 1 \cdot 4) + 4 \cdot C_2 \\ &= T(n - 1 \cdot 5) + 5 \cdot C_2 \end{aligned} \tag{4}$$

As we can see by the expansion, the recursion grows as following, where k stands for the amount of recursions:

$$T(n) = T(n - 1 \cdot k) + k \cdot C_2 \tag{5}$$

The recursion stops when T(0) has been called, this happens when $n - 1 \cdot k = 0$ and for that to occur, k must be equal to n. By inducting this case on the equation above we can form the time complexity of the function:

$$\begin{aligned} T(n) &= T(n - 1 \cdot k) + k \cdot C_2 \\ &= T(n - 1 \cdot n) + n \cdot C_2 \\ &= T(0) + n \cdot C_2 \\ &= n + C_1 + n \cdot C_2 \\ &= n \cdot C_2 + n + C_1 \end{aligned} \tag{6}$$

Since $n \cdot C_2$ is the highest order term, the time complexity of the function is thus:

$$\Theta(n) \tag{7}$$

3 Keystream

3.1 Preamble

The keystream function will make several calls to the auxiliary functions moveJokerA, moveJokerB, tripleCut, and countCut. In the best case scenario, all of these functions will execute in constant time since the values are stored in three separate queues instead of regular lists. The function tripleCut is the only function of those four that always execute in constant time independently of the input size.

In the worst case scenario, the other three functions will all execute in linear time. This is because those functions alter the queues, which might require the queues to be normalized before dequeueing the last or first element.

Since this calculation is for the worst case scenario, the calculation is made under the assumption that all three queues in the deck has to be normalized for every recursion and for everytime some function tries to alter the queues. It will also be made under the assumption that the number of cards in the deck is fixed, thus making the above operations be constant time operations.

The n-parameter in T(n) is equal to the amount of letters required by the keystream function.

3.2 Calculation

The function starts of by constructing a new deck with values ranging from 1 to 52 by making a call to the makeDeck function. This operation can be viewed as a constant time operation because it will never change in size between calls to the keystream function. It then makes a call to the local function iterate with the newly constructed deck and the amount of characters defined by the n parameter of the keystream function.

The iterate function will recursivly call itself after making all the moves needed to retrieve the next output letter and checking whether that letter was a joker (in which it calls itself again but without subtracting one from the n parameter) or not. If it isn't a joker, it appends the letter to the return value of the next recursive call. The time taken to make the moves on the deck, checking if it is a joker or not, and appending if so, will always execute in the constant time C_2 due to that the inputsize for the move functions never change during iteration since the values in the deck are merely shuffled around, and the rest of the operations are constant time operations.

When n reaches zero, the function returns an empty list with the constant time C_1. The recursive formula of the function can thus be specified as:

$$T(n) = \begin{cases} C_1 & \text{if } n = 0 \\ T(n-1) + C_2 & \text{if } n > 0 \end{cases} \tag{8}$$

As we can see, this recursive formula is identical to the formula (3) and will thus lead to the same conclusion regarding the time complexity of the function, which is:

$$\Theta(n) \tag{9}$$