COMP3121: Assignment 3 – Q1

Gerald Huang
*z*5209342

Updated: July 16, 2020

Given a DNA string of length n, our goal is to find the biggest subsequence of 'SNAKE's that occur in our original string in time complexity $O(n \log n)$. The key insight behind this problem is the way we apply a binary search, which we will also prove optimality with.

To begin, consider the length n string that encompasses the DNA and define L to be $\min\{n_S, n_N, n_A, n_K, n_E\}$ where n_i is the number of letter occurrences in the DNA string. Then define an array of snake variations such that snake $= \left[SNAKE, SSNNAAKKEE, SSSNNNAAAKKKEEE, \dots, \underbrace{S \dots S}_{L} \underbrace{N \dots N}_{L} \underbrace{A \dots A}_{L} \underbrace{K \dots K}_{L} \underbrace{E \dots E}_{L} \right]$.

Notice that if a snake pattern with $L = k$ **fails**, then so does $L = k + m$ for all $m \geq 0$. Conversely, if a snake pattern with $L = \ell$ succeeds, then it is redundant to check all venom levels $L = \ell - m$ for all $m \geq 0$ since venom level is *at least* size ℓ. Combining these two results gives us a way to apply a binary search to make the algorithm more efficient.

Proceeding with the construction of our algorithm, begin by defining the high and low variable to be n and 1 respectively. We begin by picking the middle term in our snake array, that is, the term with $L = \dfrac{\text{hi} + \text{lo}}{2}$. From here, we perform a greedy subsequence check akin to Tutorial 3, Problem 9's solution. If the check is successful, then we know that the maximum venom level is *at least* size L. As such, we check the top half recursively by applying another binary search, setting the lo variable to be $L + 1$ and updating the new maximum venom level stored. Conversely, if the check is unsuccessful, then we know that the maximum venom level is *at most* $L - 1$. As such, we check the bottom half of the snake array and set the hi variable to be $L - 1$. In each of these cases, we call the algorithm recursively, recursing through the top and bottom halves of the snake array, updating the maximum level where possible.

The binary search takes $\log n$ many iterations since, in each iteration, the array gets divided into two parts. Additionally, in each check, the greedy subsequence check takes n many steps since it has to step through the entire DNA string.

Proof of optimality

We shall proceed with a "Greedy stays ahead" (inductive) proof. Consider the base case of an empty string. It is clear that the size of such a string is zero and hence, the maximum possible venom level must also be zero. This is clear in that the binary search will return an empty array in which case, it will return the maximum venom level stored which is zero.

Now suppose that our solution is *as good* as the optimal solution at some point k; that is, for all iterations $n < k$, the algorithm produces an optimal solution. Consider the next iteration $(k + 1)$. One of two events can occur.

- Either the new letter appended to the string changes the maximum venom level, or

- The new letter appended to the string does not affect the maximum venom level.

We will show that, regardless of which case, the binary and greedy search will produce the correct venom level.

Lemma 0.1. *Let T be a string of k letters. For some integers $a \le b \le c \le d \le e < \lfloor k/5 \rfloor$, suppose there exist a S's, followed by b N's, followed by c A's, followed by d K's and followed by e E's in T. Then the addition of a new letter will return the same venom level* $\min\{a, b, c, d, e\} = a$.

Proof. Here, we assume that the string can consist of any letters in between any two S's, N's, A's, K's or E's. For example, the string `NNSNAAKESE` satisfies T since there exist 1 `S`, followed by 1 `N`, followed by 1 `K` and finally 1 `E`. Further, the binary search will guarantee that if the greedy check at some instance a fails, then so does $a + m$ for any $m > 0$.

Now suppose that a letter is appended to the string T. Since the minimum number of S's precedes all other letters, then we cannot obtain any subsequence of the form

$$\underbrace{\texttt{S...S}}_{a+m}\underbrace{\texttt{N...N}}_{a+m}\underbrace{\texttt{A...A}}_{a+m}\underbrace{\texttt{K ...K}}_{a+m}\underbrace{\texttt{E ...E}}_{a+m} \qquad \text{(for any } m > 1\text{)}$$

Since there are a minimum of a `S`'s, then any additional letter appended to the string will return the same venom level. □

Lemma 0.2. *Let T be a string of k letters. For some integers $\lfloor k/5 \rfloor \ge a \ge b \ge c \ge d > e$, suppose there exist a S's, followed by b N's, followed by c A's, followed by d K's and followed by $(e-1)$ E's in T. Then the addition of a new letter E will return a new maximum venom level* $\min\{a, b, c, d, e\} = e$. *Otherwise, it will return the same venom level.*

Proof. In a similar light to the previous lemma, we know that the letter `E` is at a minimum. We make a similar assumption to lemma 0.1 in that there can be finitely many letters between any two consecutive letters. For example, the following string `NNSSSSNANNAAAKKE` is a valid string in T since there are 4 `S`'s followed by 3 `N`'s, followed by 3 `A`'s, followed by 2 `K`'s and finally 1 `E`. Further, the binary search guarantees that, if the greedy search is successful at some instance a, then it will also work for all instances $a - m$ for all $m > 0$. And so, checking strings with $a - m$ `S`'s, `N`'s, etc. won't affect the venom level.

Then the addition of a new letter will either be an `E` in which case applying the binary search will yield a new maximum since we attain a new minimum of e. Since the greedy search will return true for e `S`'s, `N`'s, etc. then it will also work for all instances before it. Any other letter appended will return the same venom level since we still attain $(e-1)$ `E`'s. □

From these two lemmas, one can continuously construct strings of letters and the algorithm will continuously produce the maximum venom level by continually applying a binary search on the number of `S`'s, `N`'s, `A`'s, `K`'s and `E`'s, and then applying greedy search to check whether this is successful or not. Since we apply a binary search on the index, then the binary search will have a depth of $\log(n/5) = \log n - \log 5 = O(\log n)$, with each depth being a $O(n)$ greedy search. Hence the entire algorithm will run in $O\left(\underbrace{n + n + \cdots + n}_{\log n}\right) = O(n \log n)$.