Lab 5: Search
12:00 PM, Oct 7, 2018

Contents

1 Introduction 1
2 Subsequence 1
3 LCS 2
4 Gap Alignment 3

Objectives

By the end of this lab, you will know:

• how to determine if a list is a subsequence of another list
• how to determine the length of the longest common subsequence between two lists

1 Introduction

While you’re working on Bignum right now, soon you’ll be working on the Eliza project, and much of the hard work in that project has to do with looking for particular patterns in sequences of words. That ‘pattern-matching’ problem is structurally very similar to the subsequence and gap-alignment problems in this lab, so by completing it, you’ll not only learn some new techniques, you’ll be setting yourself up to have an easier time with Eliza.

2 Subsequence

Professor Klein has recently developed a love for Bananagrams and wants to write a program that can find subsequences of letters within words. But first the Professor needs to know if one word is a subsequence of another. One list is a subsequence of another if each element of the first list appears in the second, in order. The elements need not be adjacent in the second list, so (1 2) is a subsequence of (1 3 2), for instance. The empty list is a subsequence of any other list.

Task: Before you write the subsequence procedure described below draw out the recursive diagram for the following examples.

(subsequence (quote (a)) empty)
=> #false
Task: Write a procedure `subsequence` that consumes two lists containing the same type of data and produces true if the first is a subsequence of the second, and false otherwise.

Hint: This procedure can be written so that it is *linear time*, i.e., there are numbers $B$ and $C$ with the property that if the inputs both have length no greater than $n$, the number of operations involved in applying the procedure is no more than $Bn + C$, and writing it that way should be your goal. It can also be written using the `member?` procedure, but doing so will end up using many more operations, so that the worst-case op-count function grows *quadratically* rather than linearly. You don’t want that, so try to write it without using `member?`. If you cannot do so, talk briefly with a TA for suggestions on how to improve your `member?`-based solution.

Vous avez atteint un checkpoint ! Signez-vous pour que le lab TA vous révise votre travail.

3 LCS

Having succeeded at determining whether one sequence is a subsequence of another, Prof. Klein wants to solve for the longest common subsequence (LCS) between two sequences. Longest common subsequences are relevant not only in computational biology, but in other areas of computer science as well, such as version control systems (which help multiple software developers who work on a single project manage the process of updating the project).

Hint: Remember all the possibilities for two-list-argument recursion.

Task: Before you write the `lcs` procedure described, draw out the recursive diagram for the following examples. Be very careful with the recursive output(s) for the second one. You should not have to, and should not, use `member?` for this task; not only will `member?` increase the runtime significantly, but most implementations using `member?` do not actually work, although they may appear to work in many cases.

Hint: You may find the `max` procedure useful for this task. `max` takes in two numbers and outputs the maximum of the two.

Example:

$(\text{lcs } (\text{quote } (m \ o \ l \ y)) \ (\text{quote } (m \ o \ n \ o \ p \ o \ l \ y)))$
$\Rightarrow 4$

$(\text{lcs } (\text{quote } (n \ a \ n \ a)) \ (\text{quote } (b \ a \ n \ a \ a)))$
$\Rightarrow 4$

$(\text{lcs } (\text{quote } (a \ b \ c \ d \ e)) \ (\text{quote } (g \ a \ c \ f)))$
$\Rightarrow 2$
Task: Write a procedure \texttt{lcs} that consumes two lists and produces the length of their longest common subsequence. ¹

Note: Don’t use subsequence as a helper! In addition, you may find that there’s a lot of computation involved in this problem! Don’t worry; that’s what you’re supposed to find!

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

4 Gap Alignment

Plagiarism checkers work by comparing the text of a student’s assignment to other texts in a database, looking for similarities, and scoring the assignment based on how many similarities it found.

One method of scoring plagiarism could be to use the \texttt{lcs}. You can think of texts as sequences of words, so \texttt{lcs} can look for the longest common subsequence of words between the two texts. The longer the subsequence, the higher the score, and the more likely it is that one of the texts was plagiarized.

A more sophisticated plagiarism checker could be based not only on how many words match, but on how many do not match. We could imagine implementing this checker via a function which scores the number of matches (+1) and penalizes gaps between matches (say, by 0.5). The higher the resulting number, the greater the similarities between the two texts, and the more likely it is that one of the texts was plagiarized.

The pairing of matches and gaps between two sequences is an \textit{alignment}. For some sequences, such as those with no matches at all, it isn’t hard to make the alignment and score it.

\begin{itemize}
  \item \textit{Seq 1:} Banana split \quad – \quad –
  \item \textit{Seq 2:} \quad – \quad – \quad peel out
\end{itemize}

This alignment scores 0 for matches, and −2 for gaps (two gaps on the top, two on the bottom), for a total of −2.

However, when there is more than one place to make a match, you have to consider where that match will fall.

\begin{itemize}
  \item \textit{Seq 3:} Rotten bananas do not yell bananas
  \item \textit{Seq 4:} Rotten bananas do not \quad – \quad –
  \item \textit{Seq 3:} Rotten bananas do not yell bananas \quad – \quad –
  \item \textit{Seq 4:} Rotten \quad – \quad – \quad – \quad bananas do not
\end{itemize}

These two alignments of the same text have different scores. The first has four matches (+4) and two gaps (−1) for a score of +3. The second alignment has two matches (+2) and six gaps (−3) for a score of −1. We’d rather have our plagiarism checker err on the side of too harsh, so the first alignment is preferable. In other words, we want to return the maximum score.

Here are two more examples:

¹We do not ask you to return the LCS itself, because doing so requires the techniques of dynamic programming, which you will learn about in CS 18.
Seq 5: I play Bananagrams while eating bananas
Seq 6: – play Bananagrams while eating –
This alignment scores +4 for matches, and −1 for gaps, for a total score of +3.

Seq 7: Make words – from bananas
Seq 8: – – learn from bananas
This alignment scores +2 for matches, and −1.5 for gaps, for a total of +0.5

Task: Your next task is to write an alignment procedure that generalizes lcs by searching over all alignments, and then scoring them based on gaps as well as matches. Your gap-alignment procedure should add 1 whenever it encounters an alignment, and it should deduct 0.5 whenever it encounters a gap. You should not have to, and should not, use member? for this task.

Hint: Your gap alignment procedure should output +3 on Seq 3 and Seq 4 because +3 is the best alignment for a plagiarism checker (check this).

Note: Do not use lcs as a helper. Instead, modify your lcs procedure slightly to write gap-alignment.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback