# Penyelesaian *Longest Common Subsequence* dengan Menggunakan Metode *Dynamic Programming*

Vincentius Lienardo - 13518081
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*13518081@std.stei.itb.ac.id*

***Abstrak*—Persoalan *Longest Common Subsequence* (LCS) merupakan persoalan klasik di dalam *computer science* dan banyak diterapkan dalam berbagai keperluan, seperti *molecular biology* (biologi molekular), *file comparison* (perbandingan *file*), *screen redisplay, user authentication, revision control, dan* lain-lain. Persoalan LCS ini dapat diselesaikan dengan berbagai strategi algoritme atau metode seperti *exhaustive search, naïve algorithm, memoization* (*top-down dynamic programming*, mirip seperti rekursif), *bottom-up dynamic programming*, dan lain sebagainya. Namun, *dynamic programming*-lah yang menghasilkan penyelesaian persoalan LCS secara mangkus dengan catatan bahwa jumlah *input* dari *sequence* adalah konstan.**

***Kata Kunci—Longest Common Subsequence, Dynamic Programming, Substring, NP-hard, Prefix.***

## I. Pendahuluan

*Longest Common Subsequence* (LCS) *problem* merupakan persoalan dalam menemukan *subsequence* terpanjang yang berada pada *sequences* di dalam himpunan atau *list of subsequences* (namun biasanya hanya dua *sequences*). Perhatikan bahwa LCS berbeda dengan *Longest Common Substring problem*. Apabila *Longest Common Substring Problem* mencari *substring* (dengan posisi lokasi yang kontigu), LCS tidak memerlukan posisi lokasi yang kontigu.

Persoalan LCS merupakan persoalan *computer science* yang klasik yang merupakan basis dari perbandingan data seperti *diff utility*, dan diaplikasikan ke dalam bioinformatika dan *computational linguistics*. LCS diterapkan ke dalam *revision control systems* seperti *Git* untuk merekonsiliasi beberapa perubahan yang dibuat ke dalam koleksi *revision-controlled files*. Dalam aplikasi *Microsoft Word* atau *text editor* lainnya, penerapan LCS ini digunakan juga untuk melakukan *text justification*.

LCS dapat diselesaikan dengan berbagai cara, seperti dengan *exhaustive search* yang merupakan perluasan dari algoritme *brute force*, *naïve algorithm*, *memoization* (*top-down dynamic programming*), dan *bottom-up dynamic programming*. Asumsi LCS dalam penyelesaian dengan menggunakan *dynamic programming* adalah dengan masukan atau *input* dari *sequence* adalah konstan, artinya terdapat batasan atau *constraint* yang kita ketahui, dengan kata lain, kita mengetahui panjang dari *sequence* tersebut.

Apabila *sequence* tidak memiliki batasan panjang, persoalan LCS ini merupakan persoalan yang tergolong *nonpolynomial-time algorithm* karena penyelesaian dalam persoalan ini tidak dapat diselesaikan dalam waktu polinomial. Waktu polinomial yang dimaksudkan adalah kompleksitas algoritme penyelesaian suatu persoalan yang berupa polinomial, misalnya $n$, $n^2$, $n^3$, dan seterusnya; *log n*; dan lain-lain. Di lain sisi, suatu persoalan dapat dikatakan tidak dapat diselesaikan dalam waktu polinomial apabila kompleksitas algoritmenya berbentuk $2^n$ (eksponensial), $n!$ (faktorial), atau bahkan $n^n$.

Penggunaan *dynamic programming* pada LCS ini memberikan hasil yang mangkus atau optimal dibandingkan dengan algoritme lainnya. Hal ini tentu menjadi daya tarik bagi seorang *programmer* karena pada hakikatnya diinginkan suatu algoritme yang paling mangkus untuk memecahkan suatu persoalan. Dengan pilihan algoritme yang tepat, suatu persoalan akan diselesaikan dengan cepat dan dapat meminimalisir operasi yang dilakukan.

Untuk ukuran data $n$ yang kecil, tidak terlalu dirasakan perbedaan dengan menggunakan berbagai algoritme. Namun apabila jumlah data yang dimasukkan besar atau sangat besar, maka pastilah akan terlihat perbedaannya (bisa ditandai dengan lama atau tidaknya waktu eksekusi, tetapi tetaplah ingat bahwa yang merupakan pembanding kemangkusan suatu algoritme dengan algoritme lainnya adalah hanya dengan menggunakan kompleksitas algoritme).

LCS *problem* ini sangatlah sering ditanyakan pada saat *job interview* karena penyelesaian dari persoalan ini membutuhkan solusi yang kreatif. Penggunaan metode *exhaustive search* dengan mengenumerasi semua kemungkinan *subsequence* dari *sequences* sangatlah lama, maka diperlukan metode algoritme yang mangkus yang bisa memecahkan persoalan ini, yaitu dengan menggunakan *dynamic programming*.

## II. Landasan Teori

### A. Dynamic Programming

Program Dinamis atau *Dynamic Programming* merupakan metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan tahapan (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan. Istilah "program dinamis" muncul karena perhitungan solusi dengan menggunakan tabel-

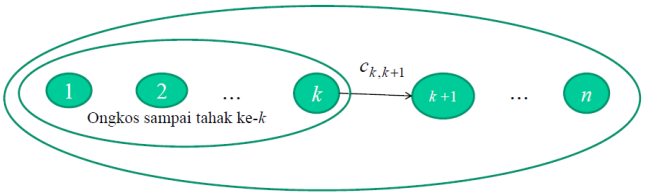
tabel. Tujuan utama model ini adalah untuk mempermudah penyelesaian persoalan optimasi yang mempunyai karakteristik tertentu. Istilah program dinamis pertama kali diperkenalkan pada era 1950-an oleh Richard Bellman, seorang *professor* di *Princeton University*. Selanjutnya, pada penerapanya pemrograman dinamis banyak digunakan pada proses optimalisasi masalah. Penyelesaian metode ini menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Karakteristik penyelesaian persoalan dengan program dinamis:

1. Terdapat sejumlah berhingga pilihan yang mungkin.
2. Solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya.
3. Menggunakan syarat optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Kita tentu mengetahui algoritme yang cukup populer dalam hal persoalan optimasi (maksimasi atau minimasi) yaitu algoritme *greedy*. Pada prinsipnya, program dinamis juga biasanya ditujukan untuk persoalan optimasi. Namun, perbedaan algoritme *greedy* dengan program dinamis adalah pada rangkaian keputusannya. Pada algoritme *greedy*, hanya satu rangkaian keputusan yang dihasilkan, sedangkan pada program dinamis lebih dari satu rangkaian keputusan yang dipertimbangkan.

Pada program dinamis juga diperkenalkan prinsip optimalitas. Ciri utama dari program dinamis adalah prinsip optimalitas yang berbunyi "jika solusi total optimal, maka bagian solusi sampai tahap ke-*k* juga optimal". Prinsip optimalitas berarti bahwa jika kita bekerja dari tahap *k* ke tahap *k* + 1, kita dapat menggunakan hasil optimal dari tahap *k* tanpa harus kembali ke tahap awal. Ongkos pada tahap *k* + 1 = (ongkos yang dihasilkan pada tahap *k*) + (ongkos dari tahap *k* ke tahap *k* + 1)

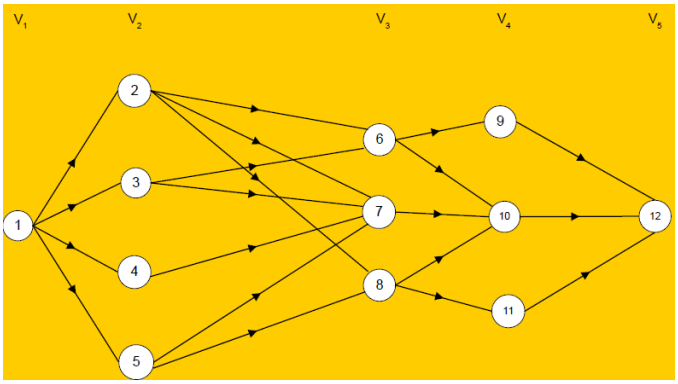



*Gambar 1: Deskripsi tahap pada Dynamic Programming* [1]

Beberapa ciri atau karakteristik dari persoalan program dinamis:

1. Persoalan dapat dibagi menjadi beberapa tahap (*stage*), yang pada setiap tahap hanya diambil satu keputusan.
2. Masing-masing tahap terdiri dari sejumlah status (*state*) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan masukan yang ada pada tahap tersebut.
3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Ongkos (*cost*) pada suatu tahap meningkat secara teratur (*steadily*) dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.
6. Keputusan terbaik pada suatu tahap bersifat independen terhadap keputusan yang dilakukan pada tahap sebelumnya.
7. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap *k* memberikan keputusan terbaik untuk setiap status pada tahap *k* + 1.
8. Prinsip optimalitas berlaku pada persoalan tersebut.

Pada poin kedua, yaitu "masing-masing tahap terdiri dari sejumlah status (*state*) yang berhubungan dengan tahap tersebut" dapat direpresentasikan ke dalam graf multitahap (*multistage graph*) dengan setiap simpul dalam graf tersebut menyatakan status, sedangkan $V_1$, $V_2$, …, $V_n$ menyatakan tahap.




*Gambar 2: Multistage Graph dengan tahap dan status* [1]

Terdapat dua pendekatan dalam program dinamis:

1. Pendekatan program dinamis maju (*forward* atau *top-down*).
2. Pendekatan program dinamis mundur (*backward* atau *bottom-up*).

Misalkan $x_1$, $x_2$, …, $x_n$ menyatakan peubah (*variable*) keputusan yang harus dibuat masing-masing untuk tahap 1, 2, …, *n*. Maka untuk kedua pendekatan:

1. Program dinamis maju bergerak mulai dari tahap 1, lalu maju ke tahap 2, 3, dan seterusnya sampai tahap *n*. Runtunan peubah keputusan adalah $x_1, x_2, \ldots, x_n$.
2. Program dinamis mundur bergerak mulai dari tahap *n*, terus mundur ke tahap $n - 1$, $n - 2$, dan seterusnya sampai tahap 1. Runtunan peubah keputusan adalah $x_n$, $x_{n-1}, \ldots, x_1$.

Prinsip optimalitas pada program dinamis maju adalah ongkos pada tahap *k* + 1 = (ongkos yang dihasilkan pada tahap *k*) + (ongkos dari tahap *k* ke tahap *k* + 1)

$$k = 1, 2, \ldots, n - 1$$

Sedangkan prinsip optimalitas pada program dinamis mundur adalah ongkos pada tahap $k$ = (ongkos yang dihasilkan pada tahap $k + 1$) + (ongkos dari tahap $k + 1$ ke tahap $k$)

$$k = n, n - 1, \ldots, 1$$

Setelah kita mengetahui persoalannya, kita berlanjut ke langkah-langkah pengembangan algoritme program dinamis:

1. Karakteristikkan struktur solusi optimal.
2. Definisikan secara rekursif nilai solusi optimal.
3. Hitung nilai solusi optimal secara maju atau mundur.
4. Konstruksi solusi optimal.

### B. *Longest Common Subsequence* (LCS)

*Longest Common Subsequence* (LCS) merupakan sebuah masalah dalam mencari *common subsequence* terpanjang dari beberapa *sequences* (biasanya terdiri dari dua *sequences*). *Subsequence* dari sebuah *string S* adalah sekumpulan karakter yang ada pada *S* yang urutan kemunculannya sama.

Diberikan sebuah *sequence* $X = (x_1, x_2, \ldots, x_m)$ dengan setiap $x_i$ merupakan sebuah elemen alfabet, misalnya {0, 1} atau alfabet {*a*, *b*, …, *z*}, *sequence* $Z = (z_1, \ldots, z_k)$ merupakan *subsequence* dari *X* jika terdapat *sequence* yang terurut menaik $(i_1, i_2, \ldots, i_k)$ yang merupakan indeks dari *X* sehingga $x_{i_j} = z_j$ untuk setiap $j \in \{1, 2, \ldots, k\}$.

Dengan kata lain, *Z* merupakan *subsequence* dari *X* apabila kita dapat menemukan *Z* di dalam *X*, di mana kita diperbolehkan untuk melompat elemen tertentu dari *X* untuk mencari *subsequence*-nya. Beda halnya untuk *substring*, kita tidak diperbolehkan untuk melompat elemen dari *X*. Di dalam *substring* kita memerlukan $i_j = i_{j-1} + 1$ untuk setiap $j \in \{2, 3, \ldots, k\}$. Sebagai contoh, (*B, C, D, B*) merupakan *subsequence* (tetapi bukan *substring*) dari (*A, B, C, B, D, A, B*).

Diberikan dua buah *sequences X* dan *Y*, kita dapat mengatakan *Z* adalah *common subsequence* jika *Z* merupakan *subsequence* dari *X* dan *Z* merupakan *subsequence* dari *Y*. Di dalam persoalan LCS, kita biasanya diberikan dua buah *sequences* $X = (x_1, x_2, \ldots, x_m)$ dan $Y = (y_1, y_2, \ldots, y_n)$ dan diminta untuk menemukan *common subsequence* dengan panjang maksimum.

*Sequence X* juga dapat direpresentasikan dengan $x_1x_2x_3\ldots x_m$ sehingga *X* memiliki panjang *m* dan *sequence Y* direpresentasikan dengan $y_1y_2\ldots y_n$ dengan panjang *n*. Selanjutnya kita akan menggunakan notasi $X[1{:}k]$ untuk merepresentasikan *prefix* $X[1{:}k] = x_1x_2x_3\ldots x_k$ dengan $k \leq m$.

## III. Langkah Penyelesaian LCS

Dalam menentukan LCS, kita akan menggunakan *dynamic programming* yang langkah-langkah dalam penyelesaian persoalannya dilakukan secara rekursif. Kita akan menyelesaikan dengan cara bertahap mulai dari subpersoalan. Subpersoalannya adalah menentukan LCS untuk *prefix* dari *X* dan *Y*. Di dalam penentuannya, terdapat dua buah kasus:

- **Kasus 1:** $x_m = y_m$. Jika $x_m = y_n = l$, maka setiap LCS *Z* memiliki *l* sebagai simbol terakhirnya. Misalkan bahwa *Z'* merupakan *common subsequence* yang tidak berakhir di *l*: maka kita dapat memperluasnya dengan menambahkan *l* ke *Z'* untuk memperoleh *common subsequence* (lebih panjang) yang lain.

  Maka, jika $|Z| = k$ dan $x_m = y_n = l$, kita dapat menulis

  $$Z[1{:}k-1] = LCS(X[1{:}m-1], Y[1{:}n-1])$$

  dan

  $$Z = Z[1{:}k-1] \circ l,$$

  di mana ○ menyatakan operasi konkat pada *strings*.

- **Kasus 2:** $x_m \neq y_m$. Seperti kasus di atas, misalkan *Z* merupakan LCS dari *X* dan *Y*. Dalam kasus ini, huruf terakhir (pada indeks terakhir) dari *Z* (misalkan $z_k$) berkemungkinan tidak sama dengan $x_m$ atau tidak sama dengan $y_m$. Dalam kasus ini, setidaknya satu dari $x_m$ atau $y_n$ tidak dapat muncul di LCS dari *X* dan *Y*; yang artinya adalah

  $$LCS(X, Y) = LCS(X[1{:}m-1], Y)$$

  atau

  $$LCS(X, Y) = LCS(X, Y[1{:}n-1]),$$

  bergantung LCS yang mana yang lebih panjang. Yang artinya kita dapat mengeliminasi satu huruf dari indeks atau bagian terakhir dari *X* atau *Y*. Secara khusus, panjang dari *LCS*(*X*, *Y*) dapat digeneralisasi menjadi

  $$lenLCS(X, Y) = max\{lenLCS(X[1{:}m-1], Y), lenLCS(X, Y[1{:}n-1])\}.$$

Dari rumus di atas, kita dapat menemukan formula rekursifnya dengan menggunakan sebuah tabel, misalnya tabel *C* sehingga

$$C[i, j] = \text{panjang dari } LCS(X[1{:}i], Y[1{:}j]).$$

Sehingga, nilai dari $C[i, j]$ memiliki tiga kemungkinan bergantung dari *i*, *j*, $X[i]$, dan $Y[j]$:

$$C[i,j] = \begin{cases} 0, & i = 0 \text{ atau } j = 0 \\ C[i-1, j-1] + 1, & X[i] = Y[j],\ i,j > 0 \\ max\{C[i-1,j],\ C[i,j-1]\}, & X[i] \neq Y[j],\ i,j > 0 \end{cases}$$

$C[i, j]$ dalam hal ini menjadi panjang dari $LCS(X[1{:}i], Y[1{:}j])$ yang merupakan *longest common subsequence* dari $X[1{:}i]$ dan $Y[1{:}j]$. Dengan begitu, kita dapat mengisi nilai dari tabel *C* dengan menggunakan formula rekurens berikut:

$$C[i,j] = \begin{cases} C[i-1, j-1] + 1, & X[i] = Y[j] \\ max\{C[i-1,j],\ C[i,j-1]\}, & X[i] \neq Y[j] \end{cases}$$

Berikut merupakan *pseudocode* untuk tabel *C* (dalam hal ini adalah sebuah matriks yang berukuran $n + 1 \times m + 1$ dengan indeks awal adalah 0):

**Algorithm 1:** lenLCS$(X, Y)$

```
Initialize an n + 1 × m + 1 zero-indexed array C.
Set C[0, j] = C[i, 0] = 0 for all i, j ∈ {1, ..., m} × {1, ..., n}.
for i = 1, ..., m do
    for j = 1, ..., n do
        if X[i] = Y[j] then
            C[i, j] ← C[i − 1, j − 1] + 1
        else
            C[i, j] ← max{C[i − 1, j], C[i, j − 1]}
return C
```

*Gambar 3: Algoritme 1 dalam bentuk pseudocode untuk menghitung panjang LCS* [2]

Dengan pendekatan rekursif tersebut menggunakan *dynamic programming*, kita dapat menentukan elemen-elemen pada tabel *C*. Kita dapat mengamati bahwa *C*[*i, j*] hanya memiliki tiga kemungkinan nilai, yaitu *C*[*i* – 1, *j*], *C*[*i, j* – 1], dan *C*[*i* – 1, *j* – 1] yang berarti bahwa setiap kali kita menghitung nilai *C*[*i, j*] dari entri sebelumnya, kompleksitasnya hanya *O*(1) yang mewakili setiap kali perhitungan atau kalkulasi yang dilakukan.

Untuk *string* dengan panjang 0 pasti akan mempunyai LCS dengan panjang 0 juga. Dengan demikian, kita dapat mengisi entri dari *C*[0, *j*] dengan 0 untuk setiap *j*, begitu pula untuk *C*[*i*, 0] dengan 0 untuk setiap *i*. Kemudian, kita akan mengisi entri sisa dari tabel tersebut, dengan mengisi mulai dari baris paling atas (untuk *i* dari 1 sampai *m*) dan mengisi setiap baris tersebut dari kiri ke kanan (dimulai dari *j* = 1 sampai *n*).

Kita dapat melihat pada gambar *Algorithm* 1 di atas hanya menghitung panjang dari *X* dan *Y* (yang direpresentasikan dalam bentuk matriks). Tahapan selanjutnya adalah dengan menentukan *longest common subsequence* yang kita ingin cari. Misalkan kita telah melakukan komputasi pada *Algorithm* 1 yang menghasilkan tabel *dynamic programming*, maka kita dapat menentukan LCS dengan menggunakan tabel tersebut.

Kita mulai dari bagian belakang *sequences X* dan *Y* dengan menggunakan tabel *C*. Proses dimulai dari inisialisasi *i* = *m* dan *j* = *m* (berarti posisi *i* dan *j* berada pada bagian kanan bawah pada tabel *C*). Dalam proses pencarian tersebut, misalkan pada suatu titik (*i, j*) terdapat elemen dari *X* dan *Y* sedemikian sehingga *X*[*i*] = *Y*[j], maka kita akan mengurangkan *i* dan *j* dengan 1 (melakukan *decrement*). Jika *X*[*i*] ≠ *Y*[*j*], kita akan melakukan pemilihan untuk membuang antara literal dari *X* dan literal dari *Y*. Jika *C*[*i, j*] = *C*[*i, j* – 1], kita akan mengeliminasi literal dari *Y* dan *j* akan dikurang dengan 1. Sebaliknya, jika *C*[*i, j*] = *C*[*i* – 1, *j*], kita akan mengeliminasi literal dari *X* dan *i* akan dikurang dengan 1.

Perhatikan bahwa pada setiap langkah, penjumlahan dari *i* + *j* dikurangi paling sedikit sejumlah 1 dan akan berhenti apabila *i, j* mendekati 0. Berarti, jumlah pengurangan yang kita lakukan pada *i* + *j* adalah maksimal *m* + *n* kali.

**Algorithm 2:** LCS$(X, Y, C)$

```
// C is filled out already in Algorithm 1
L ← ∅
i ← m
j ← n
while i > 0 and j > 0 do
    if X[i] = Y[j] then
        append X[i] to the beginning of L
        i ← i − 1
        j ← j − 1
    else if C[i, j] = C[i, j − 1] then
        j ← j − 1
    else
        i ← i − 1
```

*Gambar 4: Algoritme 2 dalam bentuk pseudocode untuk menemukan LCS* [2]

Pada *Algorithm* 2 di atas, sudah diketahui nilai dari tabel *C* (yang direpresentasikan dalam bentuk matriks). Dilakukan *while-loop* untuk melakukan *traversal* pada entri-entri yang ada di tabel dengan kondisi tertentu. Apabila pada saat dilakukannya *traversal* didapat *X*[*i*] = *Y*[*j*], tambahkan *X*[*i*] ke bagian depan *L*, lalu *decrement* nilai *i* dan *j* sebesar 1. Apabila *C*[*i, j*] = *C*[*i, j* – 1], hanya *decrement* nilai *j* sebesar 1. Apabila tidak memenuhi kedua kondisi di atas, maka hanya *decrement* nilai *i* dengan 1. Proses akan keluar dari *loop* pada saat nilai *i* = 0 atau *j* = 0.

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

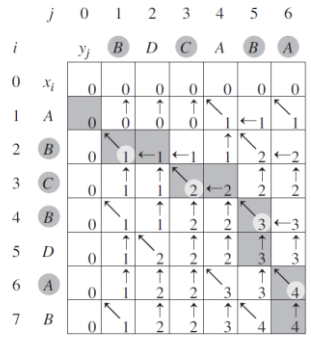
*Gambar 5: Pseudocode untuk menghitung panjang LCS* [3]

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

*Gambar 6: Pseudocode untuk menemukan LCS* [3]

*Pseudocode* yang diambil dari buku *Introduction to Algorithms* yang dibuat oleh CLRS (Cormen, Leiserson, Rivest, Stein) pada dasarnya cara berpikirnya sama. Namun, mungkin lebih dapat divisualisasikan dalam bentuk tabel dan mudah dimengerti (dengan bantuan arah panah yang menunjukkan arah pemrosesan entri selanjutnya pada tabel). Sama seperti

*pseudocode* yang telah ditunjukkan sebelumnya, basis dari rekursif LCS ini adalah apabila salah satu dari *i* atau *j* bernilai 0. Sedangkan rumus rekursifnya sudah dibuktikan pada *section* sebelumnya.



*Gambar 7: Tabel yang dibentuk dari kedua string yang dieksekusi secara bottom-up* [3]

Perlu diperhatikan bahwa penentuan arah dari panah ini dilakukan terlebih dahulu sebelum melakukan pencarian LCS. Pencarian dimulai dari indeks paling terakhir pada baris dan kolom (yaitu pada kanan bawah) dan mengikuti arah yang telah dihitung sebelumnya. Pada gambar di atas, misalkan *sequence X* = ABCBDAB dan *sequence Y* = BDCABA. Apabila pencarian berada pada panah ↖ berarti $X[i] = Y[j]$ dan literal pada entri tabel tersebut dimasukkan ke dalam hasil LCS dari belakang. Pada kasus ini, pencarian berhenti ketika $j = 0$. Hasil LCS-nya adalah BCBA.

## IV. Pembuktian Teori

Pembuktian dari teori dasar dan langkah penyelesaian yang terdapat di atas:

1. Misalkan $z_k \neq x_i$. Kita dapat melakukan *append* ke *z* yaitu $z_{k+1} = x_i = y_j$ dan mendapatkan *common subsequence* dengan panjang $k + 1$, yang kontradiksi atau berlawanan dengan hipotesis bahwa *z* adalah LCS dari $X_i$ dan $Y_j$. Dengan demikian $z_k = x_i = y_j$. Karena $Z_{k-1}$ merupakan *common subsequence* dari $X_{i-1}$ dan $Y_{j-1}$, kita hanya akan membuktikan bahwa $Z_{k-1}$ merupakan *longest common subsequence*. Misalkan terdapat *common subsequence W* yang panjangnya lebih dari $k - 1$. Dengan menambahkan $x_i = y_j = z_k$ ke *W*, kita akan mendapatkan LCS dari $X_i$ dan $Y_j$ dengan panjang lebih dari *k*, yang ternyata kontraksi juga dengan asumsi kita.
2. Jika $x_i \neq y_j$, tidak mungkin $x_i$ dan $y_j$ kedua-duanya sama dengan $z_k$. Jika $x_j \neq z_k$, maka indeks $i_k$ yang menambahkan $z_k$ ke dalam *X* seharusnya lebih kecil daripada *i*. Dengan demikian, *Z* merupakan LCS dari $X_{i-1}$ dan $Y_j$.

## V. Kompleksitas Algoritme

Karena kita melakukan pengisian dan kalkulasi $O(1)$ pada setiap elemen, maka untuk tabel (dalam hal ini direpresentasikan dalam matriks) *C* dengan *m* baris dan *n* ($m \times n$) memiliki kompleksitas waktu $O(mn)$. Pada *Algorithm* 2, *while-loop* pada *i* dan *j* selama $i > 0$ dan $j > 0$ memberikan *loop* maksimal selama *m* atau *n* kali, maka kompleksitasnya adalah $O(m + n)$. Jadi, dalam menentukan LCS dari dua buah *sequences*, diperlukan kompleksitas $O(mn)$ dengan *m* menyatakan panjang dari *sequence X* dan *n* menyatakan panjang dari *sequence Y*.

## VI. Implementasi

```
def calculate_LCS(A, B):
    n = len(A)
    m = len(B)
    C = [[0 for i in range(0, m + 1)] for j in range(0, n + 1)]
    D = [[0 for i in range(0, m + 1)] for j in range(0, n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if A[i - 1] == B[j - 1]:
                C[i][j] = C[i - 1][j - 1] + 1
                D[i][j] = 0
            elif C[i][j-1] >= C[i - 1][j]:
                C[i][j] = C[i][j - 1]
                D[i][j] = 1
            else:
                C[i][j] = C[i - 1][j]
                D[i][j] = 2
    i = n
    j = m
    S = ''
    while(i > 0 and j > 0):
        if(D[i][j] == 0):
            S = A[i - 1] + S
            i -= 1
            j -= 1
        elif(D[i][j] == 2):
            i -= 1
        else:
            j -= 1
    return str(C[n][m]), S
```

## VII. Pengujian

```
PS C:\Users\vince> python -u "c:\Users\vince\Desktop\LCS.py"
Masukkan string X: ATGACGGATCAGCCGCAAGCGGAATTGGCGACATAACAAG
Masukkan string Y: TACTGCCTAGTCGGCGTTCGCCTTAACCGCTGTATTGTTC
------------------------------------------------------------
Panjang Longest Common Subsequence (LCS) adalah 23
Longest Common Subsequence-nya adalah TACTGCCAGCGGTTCGCTAACAG
PS C:\Users\vince>
```

*Gambar 8: Pengujian LCS dengan menggunakan rantai basa nukleotida*

LCS ini dapat diterapkan di berbagai bidang lain, salah satunya pada bioinformatika. Pengujian pertama dilakukan pada sampel kode genetik yang terdiri dari basa nukleotida

*adenine* (A), *cytosine* (C), *guanine* (G), dan *thymine* (T). Misalkan terdapat rantai kode genetik pertama yang direpresentasikan dalam *string X* dan rantai kode genetik kedua yang direpresentasikan dalam *string Y*, maka perhitungan LCS akan menghasilkan berapa *common subsequence* yang terpanjang, dalam hal ini panjangnya adalah 23 dengan *longest common subsequence*-nya adalah rantai basa nukleotida TACTGCCAGCGGTTCGCTAACAG.

```
PS C:\Users\vince> python -u "c:\Users\vince\Desktop\LCS.py"
Masukkan string X: 123456789
Masukkan string Y: 181200
------------------------------------------------------------
Panjang Longest Common Subsequence (LCS) adalah 2
Longest Common Subsequence-nya adalah 18
PS C:\Users\vince>
```

*Gambar 9: Pengujian dengan menggunakan kumpulan digit*

Persoalan kedua ini apabila *string X* diisi dengan *digit* 123456789 dan *string Y* yang diisi dengan *digit* 181200. *Common subsequence*-nya adalah 18 dan ternyata merupakan *longest common subsequence* dengan panjang 2.

```
PS C:\Users\vince> python -u "c:\Users\vince\Desktop\LCS.py"
Masukkan string X: Nama depannya Vincentius dan nama belakangnya Lienardo
Masukkan string Y: Vincentius Lienardo
------------------------------------------------------------
Panjang Longest Common Subsequence (LCS) adalah 19
Longest Common Subsequence-nya adalah Vincentius Lienardo
PS C:\Users\vince>
```

*Gambar 10: Pengujian dengan menggunakan nama penulis*

Persoalan ketiga ini dengan memasukkan nama panjang penulis. *String X* sengaja diisi dengan narasi yang menyebutkan nama depan dan nama belakang penulis. *String Y* diisi dengan nama lengkap penulis. Akan dilakukan pengecekan apakah terdapat *common subsequence* (dan yang terpanjang) dari kedua *string* tersebut. Ternyata, terdapat LCS dengan *string* "Vincentius Lienardo" dengan panjang 19.

```
PS C:\Users\vince> python -u "c:\Users\vince\Desktop\LCS.py"
Masukkan string X: ABCDEFG
Masukkan string Y: Z
------------------------------------------------------------
Tidak terdapat common subsequence dari kedua string tersebut.
PS C:\Users\vince>
```

*Gambar 11: Pengujian dengan menggunakan alfabet sembarang*

Kemungkinan selanjutnya adalah bagaimana apabila kedua *string* tersebut ketika dilakukan pengecekan tidak terdapat *common subsequence*? Program akan mengeluarkan *output* tidak terdapat *common subsequence*, karena *string Y* yang berisi Z tidak terdapat pada *string X*.

## VIII. Simpulan Dan Saran

*Bottom-up dynamic programming* yang diimplementasikan memiliki kompleksitas $O(mn)$ yang cukup mangkus dibandingkan dengan algoritme lain (misalnya *brute force* yang kompleksitas algoritmenya $O(m \times 2^n)$). Perbandingan kompleksitas dengan menggunakan *dynamic* programming dengan *brute force* sangat jauh. *Dynamic programming* untuk LCS ini menggunakan tabel terlebih dahulu agar mendapatkan panjang LCS dan menentukan arah indeks yang di-*traversal*. Setelah itu, tahap rekonstruksi solusi dilakukan agar menemukan LCS.

Sarannya semoga ke depannya ditemukan metode pemecahan persoalan LCS ini yang lebih mangkus sehingga apabila *string* yang dimasukkan sangat panjang (ukuran *m* atau *n* sangat besar) dapat teratasi dengan baik dengan penyelesaian persoalan dalam *polynomial time*.

## Link Video YouTube

https://www.youtube.com/watch?v=f_O-2VYO698

## GitHub Repository Link

https://github.com/VincentLie18/Longest-Common-Subsequence

## Ucapan Terima Kasih dan Penutup

Puji syukur kepada Tuhan karena atas berkat-Nya, *paper* berjudul "Penyelesaian *Longest Common Subsequence* dengan Menggunakan Metode *Dynamic Programming*" dapat selesai tepat waktu. Saya mengucapkan terima kasih kepada Bapak Rinaldi Munir, Ibu Masayu Leylia Khodra, dan Ibu Nur Ulfa Maulidevi sebagai dosen pengajar IF2211 Strategi Algoritma atas bimbingannya selama satu semester ini sehingga *paper* ini dapat selesai.

## Referensi

[1] Munir, Rinaldi. 2018. *Slide Kuliah Dynamic Programming IF2211 Strategi Algoritma*. Bandung: Institut Teknologi Bandung. Diakses pada 29 April 2020, 18.00 WIB.

[2] Yang, Wilbur dan Mary Wootters. 2017. *CS161 Lecture 13*. Stanford University. Diakses pada 30 April 2020, 19.00 WIB.

[3] Cormen, T. H. dkk. 2009. *Introduction to Algorithms* (Edisi Ketiga). The MIT Press.

[4] Benson, Gary dkk. 2014. *Longest Common Subsequence in k Length Substrings*. Boston University. Diakses pada 30 April 2020, 21.00 WIB.

## Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Mei 2020

Vincentius Lienardo, 13518081