בס"ד

סיכום – LCS – תת המחרוזת המשותפת הארוכה ביותר

הבעיה: נתונות 2 מחרוזות. יש למצוא את אורך תת המחרוזת המשותפת בין 2 המחרוזות הארוכה ביותר.

דוגמא: עבור המחרוזות: X="abcade" , Y="aebdc" – תוחזר המחרוזת: "abc"

פתרון:

השיטה החמדנית – נעבור על המחרוזת X, נתחיל לחפש את המופע של האות הראשונה במחרוזת Y. אם מצאנו, נמשיך לאות הבאה ב X ונחפש אותה החל מאותו מקום בו עצרנו ב Y – עד שנגיע לסוף של Y.

בדוגמא שלנו, נחפש את a במחרוזת Y ונעצור במקום הראשון, נחפש את b ב Y החל מהמקום השני ונמצא את b במקום השלישי, נמשיך ל c ונגיע לסוף של Y – ונחזיר את "abc".

- סיבוכיות השיטה: $O(n \cdot m)$ – רק במקרה הגרוע – כי אם אין התאמה כלל בין X ל Y אז עבור כל אות ב X נצטרך לעבור על כל Y.
- נכונות השיטה: השיטה לא מחזירה את התשובה הנכונה תמיד כי לא תמיד האיבר הראשון שנמצא הוא חלק מהסדרה. בנוסף, לפעמים כדאי לוותר על מספר איברים כדי לקחת אחרים טובים יותר. לדוגמא, אם היינו מפעילים את האלגוריתם על המחרוזות X="axxxb" , Y="bxxxa" היינו מקבלים תת מחרוזת באורך 1.
- הקוד:

```
public static String greedyLCS(String X, String Y){
      String ans = "";
      int start = 0;
      for (int i=0; i<X.length(); i++){
            int index = Y.indexOf(X.charAt(i), start);
            if (index != -1){
                  ans = ans + X.charAt(i);
                  start = index+1;
            }
      }
      return ans;
}
```

השיטה החמדנית עם סיבוכיות משופרת – נעבוד באותה שיטה כמו החמדני הרגיל רק שנשמור במערך בגודל מספר התווים את מספר המופעים של כל אות ב Y ולפני כל חיפוש ב Y, נבדוק קודם האם האות בכלל קיימת בהמשך ובכך נחסוך את המעבר על Y עבור אותיות ללא התאמה. נדאג לעדכן את המערך עבור כל אות שעברנו.

- סיבוכיות השיטה: $O(n \cdot m)$ – עוברים קדימה על 2 המחרוזות ולא חוזרים אחורה.
- נכונות השיטה: כמו החמדני הרגיל.
- הקוד:

```
public static String greedyLCSImproved(String X, String Y){
            int freq[] = new int[26];
            for (int i=0; i<X.length(); i++){
                  int place = (X.charAt(i)-'a');
                  freq[place]++;
            }
```

```
        String ans = "";
        int start = 0;
        for (int i=0; i<Y.length(); i++){
            int place = (Y.charAt(i)-'a');
            if (freq[place] > 0){
                int index = X.indexOf(Y.charAt(i), start);
                if (index != -1){
                    ans = ans + Y.charAt(i);
                    start = index+1;
                }
            }
            freq[place]--;
        }
        return ans;
    }
```

<u>חיפוש שלם</u> – נייצר את כל תתי המחרוזות של X ו Y ונשווה בין כל 2 מחרוזות (אחת מ X ואחת מ Y) אם מצאנו מחרוזות שוות, נבדוק האם אורך מחרוזת הוא הכי ארוך שמצאנו עד עכשיו ונשמור אותו ואת המחרוזת.

בדוגמא שלנו, נייצר את כל תתי המחרוזות של X="abcade" : "a","b"..."ab","ac"..."abcade"

באופן דומה נייצר את תתי המחרוזות של Y ונשווה בין כל 2 מחרוזות.

ואכן בסופו של דבר נגיע לתשובה הנכונה, כאשר נשווה את "abc" עם "abc".

- סיבוכיות השיטה: $O(2^{n+m} \cdot \min(n,m))$ – מספר תתי המחרוזות של X כפול מספר תתי המחרוזות של Y (השוואה בין כל 2 מחרוזות) וההשוואה עצמה היא $\min(n,m)$ כי בודקים שוויון עד האורך של המחרוזת הקצרה יותר.
- נכונות השיטה: בודקים את כל האפשרויות ולכן בהכרח נגיע גם לתשובה הנכונה.
- הקוד:

```
public static String wholeSearch(String X, String Y) {
    Vector<String> sx = getAllSubStrings(X);
    Vector<String> sy = getAllSubStrings(Y);
    String lcs = "";
    for(String s1 : sx) {
        for(String s2 : sy) {
            if(s1.equals(s2)) {
                if(s1.length() > lcs.length()) {
                    lcs = s1;
                }
            }
        }
    }
    return lcs;
}

private static Vector<String> getAllSubStrings(String str) {
    Vector<String> ans = new Vector<String>();
    getAllSubStrings(str,ans,0,"");
    return ans;
}


private static void getAllSubStrings(String str, Vector<String> ans,
int i,String temp) {
    if(i == str.length()) {
```

```
            ans.add(temp);
            return;
        }
        getAllSubStrings(str,ans,i+1,temp);
        getAllSubStrings(str,ans,i+1,temp + str.charAt(i));
}
```

<u>תכנות דינאמי</u> – בחיפוש השלם, בדקנו תתי מחרוזות מיותרות כי תת המחרוזת המשותפת הארוכה ביותר החל מאיבר כלשהו היא לפחות אותה תת מחרוזת אם נחשב החל מתחילת המחרוזות.

נייצר מטריצה שבה כל תא i,j מייצג את אורך תת המחרוזת המשותפת הארוכה ביותר עד התו ה i במחרוזת X ועד התו j במחרוזת Y. נמלא את המטריצה (תא i,j) באופן הבא:

אם התו ה i במחרוזת X שווה לתו ה j במחרוזת Y אז $mat[i][j] = mat[i-1][j-1] + 1$, כלומר תת המחרוזת הארוכה ביותר עד התו הקודם ב X והתו הקודם ב Y + ההתאמה החדשה.

אחרת, $mat[i][j] = \max(mat[i][j-1], mat[i-1][j])$, כלומר, ניקח את ההתאמה הארוכה ביותר של X עד התו i עם Y עד התו $j-1$, או ההתאמה הארוכה של X עד התו $i-1$ עם Y עד התו j.

נוסיף שורת אפסים ועמודת אפסים בהתחלה כדי לא לחרוג מהמטריצה.

בדוגמא שלנו המטריצה תיראה כך:

		a	b	c	a	d	e
	0	0	0	0	0	0	0
a	0	1	1	1	1	1	1
e	0	1	1	1	1	1	2
b	0	1	2	2	2	2	2
d	0	1	2	2	2	3	3
c	0	1	2	3	3	3	3

והתא האחרון הוא התשובה – האורך של תת המחרוזת המשותפת הארוכה ביותר.

כדי לקבל את המחרוזת עצמה – נתחיל מהתא האחרון ונחזור אחורה לפי החוקיות בה מילינו את המטריצה – אם התווים זהים – נוסיף את התו לתשובה ונחזור אחורה ל $(i-1,j-1)$. אם התווים שונים – נחזור אחורה לתא הגדול מבין $(i-1,j)$ ו $(i,j-1)$.

- סיבוכיות השיטה: $O(n \cdot m)$ – ממלאים את המטריצה לפי החוקיות.
- נכונות השיטה: הראנו את האלגוריתם.
- הקוד:

```
public int LCS_length(String X, String Y) {
        int n = X.length()+1;
        int m = Y.length()+1;
        mat = new int[n][m];
        for (int i = 1; i < n; i++) {
                for (int j = 1; j < m; j++) {
                        if(X.charAt(i-1) == Y.charAt(j-1)) {
                                mat[i][j] = mat[i-1][j-1] + 1;
                        }
                        else {
                        mat[i][j] = Math.max(mat[i-1][j], mat[i][j-1]);
                        }
                }
        }
```

```
        return mat[n-1][m-1];
    }

    public String LCS_string(String X, String Y) {
        int len = LCS_length(X,Y);
        int i = X.length();
        int j = Y.length();
        String ans = "";
        while(len > 0) {
            if(X.charAt(i-1) == Y.charAt(j-1)) {
                ans = X.charAt(i-1) + ans;
                i--;
                j--;
                len--;
            }
            else {
                if(mat[i-1][j] > mat[i][j-1]) {
                    i--;
                }
                else {
                    j--;
                }
            }
        }
        return ans;
    }
```