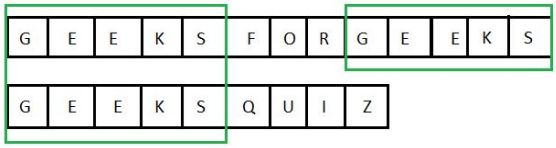
LCS - תת המחרוזת המשותפת הארוכה ביותר

תיאור הבעיה

נתונות 2 מחרוזות X,Y.

נרצה למצוא את תת-המחרוזת המשותפת הארוכה ביותר.



OUTPUT: 5
As longest Common String is "Geeks"

דוגמא

עבור המחרוזות $X = "abcade"$, $Y = "aebdc"$ תוחזר המחרוזת "abc".

פתרון הבעיה

עבור בעיה זו נציג 3 אלגוריתמים שונים.

- אפשרות א' - אלגוריתם חמדני
  - 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.

```
public static String greedy(String X, String Y) {
   String ans = "";
   int i = 0, j = 0;

   while( i < X.length() && j < Y.length() ) {
     if(X.charAt(i) == Y.charAt(j)) {
         ans += X.charAt(i);
         i++;
     }
     j++;
   }
   return ans;
}
public static void main(String[] args) {
   String X = "ababcb", Y = "cbab";
   System.out.println(greedy(X,Y)); // prints “ab”
   System.out.println(greedy(Y,X)); // prints “cb”
}
```

- **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:
  $X = "axxxb"$, $Y = "bxxxa"$ אז היינו מקבלים תת-מחרוזת באורך 1.

- **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.

❖ אפשרות ב' - אלגוריתם חמדני משופר

- נבנה מערך עזר בגודל 26 (מספר האותיות באנגלית) על המחרוזת הקצרה ביותר (נניח X) אשר ייתן אינדיקציה איזה אותיות קיימות במחרוזת.
- נפעיל את הקוד של החמדני כאשר נעבור על מחרוזת Y.
- אם האות קיימת, אז נבדוק את מיקומו ונוסיף ל-ans (מחרוזת הפתרון) ואחר כך נוריד במיקום של האות במערך --1 .
- הרעיון הוא לא לחפש את האותיות של Y של כבר לא מופיעות ב-X.

לדוגמה עבור המחרוזות $X = "abca"$ ו- $Y = "adcbc"$, נבנה מערך עזר עבור X מאחר והוא המינימלי ביותר. הבנייה מסתמכת לפי ASCII כאשר בטבלה הערך של 'a' הוא 97, ולכן כך נתאים את האותיות והסדר שלהם לאינדקסים במערך בגודל 26, לכן עבור המקרה שלנו:

'a' - 97 = **0**	'b' - 97 = **1**	'c' - 97 = **2**	'd' - 97 = **3**
2	1	1	0

מימוש אלגוריתם חמדני משופר

```
public static String improvedGreedy(String x,String y) {
   int[] occurrences = new int[26];

   for (int i = 0; i < y.length(); i++)
       occurrences[y.charAt(i)-'a']++;

   String answer = "";
   int limit = 0;

   for(int i = 0; i < x.length(); i++) {
       int place = (x.charAt(i)-'a');
       if(occurrences[place]>0) {
           int index = y.indexOf(x.charAt(i),limit);
           if(index != -1) {// if the char does occur
```

```
                answer += x.charAt(i);
                limit = index + 1; // reduce the limit
                occurrences[place]--;
            }
        }
    }
    return answer;
}
public static void main(String[] args) {
    String X = "cbab", Y = "ababcb";
    System.out.println(improvedGreedy(X,Y));
}
```

- **נכונות השיטה**: כמו באלגוריתם הקודם, אם היינו מפעילים את האלגוריתם על המחרוזות: $X = "axxxb"$, $Y = "bxxxa"$ אז היינו מקבלים תת-מחרוזת באורך 1.

- **סיבוכיות**: אנחנו רצים על כל מחרוזת פעם אחת בלבד ולכן $O(m+n)$ אבל אנחנו צריכים לעבור על אחת המחרוזות פעם נוספת כדי למלא את המערך ולכן נוסיף $O(min(m,n))$ אנו מבינים שעדיף לנו למלא את המערך במחרוזת הקצרה יותר.
לכן סה"כ נקבל: $O(n+m)+O(min(n,m))$.

❖ אפשרות ג' - חיפוש שלם

- הרעיון הוא לקחת כל תתי-מחרוזות של X ו- Y ולחפש מחרוזת משותפת ארוכה ביותר.
- אלגוריתם לבניית כל תתי-מחרוזות של מחרוזת נתונה. מספר תתי-מחרוזות של מחרוזת בגודל n שווה ל- $2^n - 1$ (לא כולל המחרוזת הריקה).

נסתכל על המחרוזת כביטים, לדוגמה עבור $X = abc$, n=3, מספר כל תתי המחרוזות הוא $2^3 - 1 = 7$.

	a	b	c	תת המחרוזת
1	0	0	1	c
2	0	1	0	b
3	0	1	1	bc
4	1	0	0	a
5	1	0	1	ac
6	1	1	0	ab
7	1	1	1	abc

עבור כל מילה ניצור מערך של מחרוזות subsets בגודל $2^n - 1$ ונמלא אותה בכל תתי המחרוזות האפשריים עבור אותה המחרוזת. נטייל בלולאה בגודל המערך ועל כל איטרציה אנחנו נפעל בדיוק כמו שתיארנו בדוגמה של הטבלה מלמעלה.

כלומר, אנחנו ניצור מערך מספרי מאופס binary בגודל של המחרוזת שאיתה אנחנו מתעסקים ועליה נבצע אותם פעולות כמו שהראינו בטבלה.

על כל איטרציה בלולאה שלנו נחבר נוסיף 1 (**plusOne**) למספר הבינארי שלנו (המערך המספרי) בדיוק כמו בטבלה וזה ייצג את תת המחרוזת שאותה נייצר וכך הלאה...

מימוש חיפוש שלם:

```
public static void plusOne(int[] binary) {
   int size = binary.length - 1;
   while(size >= 0 && binary[size] == 1) {
       binary[size--] = 0;
   }
   if(size >= 0)
       binary[size] = 1;
}

public static String[] subsets(String str) {
   int array_size = ((int) (Math.pow(2, str.length())))-1;
   String[] subsets = new String[array_size];
   int[] binary = new int[str.length()];

   for(int i = 0; i < subsets.length; i++) {
       plusOne(binary);
       String subset = "";

       for (int j = 0; j < binary.length; j++) {
           if(binary[j] == 1)
               subset += str.charAt(j);
       }
       subsets[i] = subset;
   }
   return subsets;
}

public static String bruteForce(String X, String Y) {
   String shortest = X, longest= Y, ans = "";
   if(X.length() > Y.length()){
       shortest = Y;
       longest= X;
   }
```

```
    String[] shortestSet = subsets(shortest);
    String[] longestSet = subsets(longest);

    for(int i = 0; i < shortestSet.length; i++) {
        for(int j = 0; j < longestSet.length; j++) {

            if(shortestSet[i].equals(longestSet[j])) {
                if(shortestSet[i].length() > ans.length())
                    ans = shortestSet[i];
            }
        }
    }
    return ans;
}

public static void main(String[] args) {
    String X = "axxxb", Y = "bxxxa";
    System.out.println(bruteForce(X,Y));
}
```

- **סיבוכיות**: נחשב את תתי המחרוזות של X כלומר $O(2^m)$.
  נעבור בלולאה כפולה עם מספר תתי המחרוזות של X כלומר 2^m כפול מספר תתי המחרוזות של Y כלומר וביחד $2^n \cdot 2^m = 2^{n+m}$.
  וגם מספר ההשוואות הוא $min(n,m)$ כי בודקים שוויון עד האורך של המחרוזת הקצרה ביותר. סה"כ נקבל: $O(2^{n+m} \cdot min(n,m) + 2^m) = O(2^{n+m} \cdot min(n,m))$.
- **נכונות השיטה**: בודקים את כל האפשרויות ולכן בהכרח נגיע גם לתשובה הנכונה.

נציג פתרון דומה עבור חיפוש שלם והפעם בשיטת פעולות מתמטיות להמרה בינארית.
בדומה למימושים ולשיטות שהצגנו עבור הנושא של חישוב חזקות בעזרת מספרים בינאריים,
ניצור מחלקה שרצה על כל ה- $2^n - 1$ איברים במערך ועל כל אינדקס בריצה נבצע המרה למספר בינארי באמצעות חילוק האינדקס ב-2 ובקבלת שארית 1 נצרף את נשרשר את התו במיקום הזה עד קבלת תת המערך עבור מקרה זה.

```
public static String[] subset(String str) {
    String[] subsets = new String[(int)Math.pow(2,str.length()) - 1];

    for(int decimal = 0 ; decimal < subsets.length; decimal++) {
        String subset = "";
        int binary = decimal, i = 0;
        while(binary != 0 ) {
            if(binary % 2 == 1) {
```

```
                subset += str.charAt(i);
            }
            binary /= 2;
            i++;
        }
        subsets[decimal] = subset;
    }

    return subsets;
}


public static String bruteForce(String X, String Y) {
    String ans = "", shortest = X, longest = Y;
    if(X.length() > Y.length()) {
        shortest = Y;
        longest = X;
    }

    String[] shortestSubset = subset(shortest);
    String[] longestSubset = subset(longest);

    for(int i = 0 ; i < shortestSubset.length ; i++) {
        for(int j = 0; j < longestSubset.length; j++) {

            if(shortestSubset[i].equals(longestSubset[j])) {
                if(shortestSubset[i].length() > ans.length()) {
                    ans = shortestSubset[i];
                }
            }
        }
    }
    return ans;
}

public static void main(String[] args) {
    String X = "abcbdab", Y = "bdcaba";
    System.out.println(bruteForce(X,Y)); // bcba
}
```

הסיבוכיות שעד עכשיו היה לנו הם:

$O(n \cdot m)$ עבור חיפוש חמדני.

$O(n + m)$ עבור חיפוש חמדני משופר.

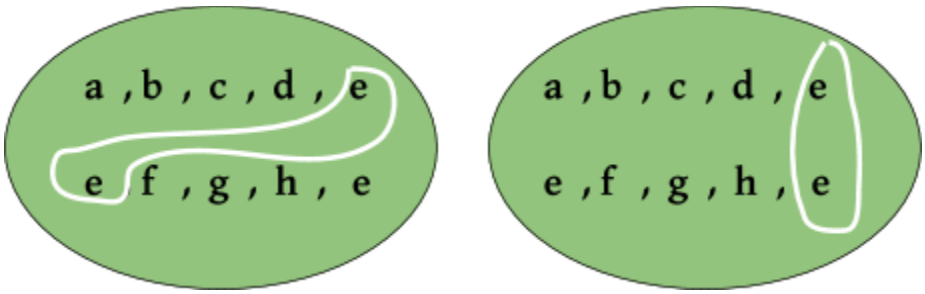
$O(2^{n+m} \cdot min(n, m))$ עבור חיפוש עץ שלם.

ננסה לפתח פתרון יותר יעיל שמשלב את היעילות של החיפוש החמדן ואת הפתרון האופטימלי שהחיפוש השלם מספק לבעיה - וכך נקבל את הפתרון הטוב ביותר לבעיה שלנו.

שיטה חדשה 1

$$\overline{X} = \{x_1, x_2, ..., x_n\} \ , \ \overline{Y} = \{y_1, y_2 ...., y_m\}$$
$$LCS(\overline{X}, \overline{Y})$$

יהא $LCS(\overline{X} \cdot \alpha \ , \overline{Y} \cdot \beta)$ כך ש- $\alpha = \beta$.
במקרה זה, באופן הכי פשוט, $LCS(\alpha, \beta) = 1$ לכן מתקיים $LCS(\overline{X} \cdot \alpha \ , \ \overline{Y} \cdot \beta) \geq LCS(\overline{X}, \overline{Y}) + 1$.
כלומר הוספנו לכל מחרוזת תו זהה ולכן יש לנו לפחות התאמה אחת, אין מצב שהיו פחות התאמות.
למשל באיור הבא אפשר לראות שני אפשרויות להתאמה:



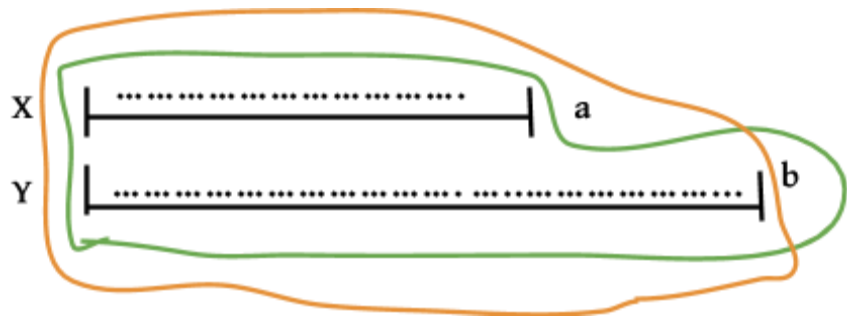
כלומר האורך של כל התאמה תהיה לפחות אחת.
אבל מה קורה שיש התאמה בין α חיצוני ל- β פנימית?

אפשר לראות שקיבלנו התאמה של אותו תו α עוד לפני שנפגשנו ב- α = β המתוכנן שלנו.
פסלנו את כל שאר התווים החל אותו תו שבו מצאנו התאמה.

שיטה חדשה 2

יהא $LCS(\overline{X}\cdot\alpha\ ,\ \overline{Y}\cdot\beta)$ כך ש- $\alpha \neq \beta$.

במקרה זה נוכל לתת פתרון לבעיה בשיטה הקודמת, נתבונן באיור הבא:



אם $\alpha \neq \beta$ אז אין התאמה המתבססת על α מול β אבל, מה שכן יכול לקרות לטובתנו:

או שזה $LCS(\overline{X},\ \overline{Y}\cdot\beta)$ או שזה $LCS(\overline{X}\cdot\alpha,\ \overline{Y})$ כלומר או ש- α שותפה ו- β לא, או להפך...

ומאחר ומדובר בתת המחרוזת המקסימלית ביותר נבחר:

$$MAX\ (\ LCS(\overline{X}\cdot\alpha,\ \overline{Y})\ ,\ LCS(\overline{X},\ \overline{Y}\cdot\beta)\ )$$

כלומר עבור 2 השיטות שהצגנו:

$$LCS(\overline{X}\cdot\alpha,\overline{Y}\cdot\beta) = \begin{cases} LCS(\overline{X},\overline{Y}) + 1 & \alpha=\beta \\ max(\ LCS(\overline{X}\cdot\alpha,\overline{Y})\ ,\ LCS(\overline{X},\overline{Y}\cdot\beta)\ ) & \alpha\neq\beta \end{cases}$$

לסיכום, נציג בפסאודו-קוד את המקרים הבאים:

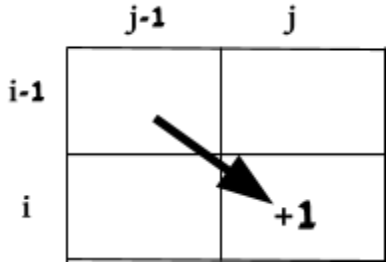
```
if α == β
   then LCS(X̄ · α , Ȳ · β) = LCS(X̄, Ȳ) + 1
else if α! = β
   then LCS(X̄ · α , Ȳ · β) = max ( LCS(X̄ · α, Ȳ) , LCS(X̄, Ȳ · β) )
```

- אפשרות ד' עבור LCS - תכנות דינאמי
  - בעזרת 2 השיטות שהצגנו מקודם נבנה שיטת פתרון חדשה.
  - 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.
  - כלומר ראינו שעבור 3 האפשרויות עד כה לא באמת קיבלנו פתרון אופטימלי.
  - נרצה לפתור את הבעיה באופן יעיל ביותר (מהיר) כמו בשיטת החמדן אך כמובן שגם נרצה לפתור באופן אופטימלי (נכון) כמו בשיטת החיפוש השלם.
  - התכנות הדינאמי מספק לנו את הפתרון היעיל והאופטימלי ביותר עבור LCS.

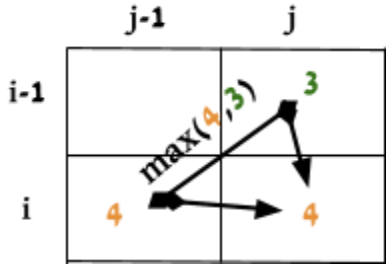
נראה שפתרון זה יעיל בסיבוכיות זמן ריצה של $O(n \cdot m)$.

תכנון הפתרון

נממש בצורה אינדוקטיבית עם מבנה נתונים של מערך דו-מימדי.
עבור שיטה 1 כאשר $\alpha = \beta$ ראינו ש- $LCS(\overline{X} \cdot \alpha\ ,\ \overline{Y} \cdot \beta) = LCS(\overline{X}, \overline{Y}) + 1$.
לכן כאשר $\overline{X}[i] = \overline{Y}[j]$ נפעל כך: $arr[i][j] = arr[i-1][j-1] + 1$, כמו בציור:



ועבור שיטה 2 כאשר $\alpha \neq \beta$ ראינו ש- $LCS(\overline{X} \cdot \alpha\ ,\ \overline{Y} \cdot \beta) = max(LCS(\overline{X} \cdot \alpha,\ \overline{Y})\ ,\ LCS(\overline{X},\ \overline{Y} \cdot \beta))$
לכן כאשר $\overline{X}[i] \neq \overline{Y}[j]$ נבחר את המקסימום מבין ה-2: $arr[i][j] = max(arr[i][j-1]\ ,\ arr[i-1][j])$
כמו בציור:



ובסוף התהליך, נקבל את המטריצה הבאה:

			0	1	2	3	4	5	6
		$\overline{X}$	a	b	c	b	d	a	b
	$\overline{Y}$		0	0	0	0	0	0	0
0	b	0	0	1	1	1	1	1	1
1	d	0	0	1	1	1	2	2	2
2	c	0	0	1	2	2	2	2	2
3	a	0	1	1	2	2	2	3	3
4	b	0	1	2	2	3	3	3	4
5	a	0	1	2	2	3	3	4	4

ניתן לראות כי קיימות קיימות כאן 3 אפשרויות של LCS כך שכל אחת מהן באורך 4.

מימוש **אינדוקטיבי**

```
public static int[][] generateMatrix(String X, String Y) {

   int[][] matrix = new int[X.length() + 1][Y.length() + 1];
   for(int i = 0 ; i < matrix.length; i++)
       matrix[i][0] = 0;
   for(int i = 0 ; i < matrix[0].length; i++)
       matrix[0][i] = 0;

   for(int i = 1 ; i < matrix.length ; i++) {
       for(int j = 1 ; j < matrix[0].length; j++) {

           if(X.charAt(i-1) == Y.charAt(j-1)) {
               matrix[i][j] = matrix[i-1][j-1] + 1;
           } else {
               matrix[i][j] = Math.max(matrix[i][j-1],matrix[i-1][j]);
           }
           System.out.print(matrix[i][j] + " ");
       }
       System.out.println();
   }
   return matrix;
}
```

```
public static String dynamic(String X, String Y) {
   String ans = "";
   int[][] matrix = generateMatrix(X,Y);
   int length = matrix[X.length()][Y.length()];
   int i = X.length(), j = Y.length();
   while(length > 0) {
       if(X.charAt(i-1) == Y.charAt(j-1)) {
           ans = X.charAt(i-1) + ans;
           i--;
           j--;
           length--;
       } else if(matrix[i][j-1] > matrix[i-1][j]) {
           j--;
       } else {
           i--;
       }
   }
   return ans;
}

public static void main(String[] args) {
   String X = "abcbdab", Y = "bdcaba";
   System.out.println(dynamic(X,Y)); // bcba
}
```

- **סיבוכיות**: עבור האתחול של עמודת אפסים לכל מילה (כדי שנדע מתי לעצור כשנחזיר את הפתרון) הוא $O(n)+O(m)$ ועבור הלולאה הקנונית נקבל $O(n \cdot m)$.
  ולסיכום $O(n)+O(m)+O(n \cdot m)=O(n \cdot m)$.

- לכן הבאנו פתרון משולב מהצד החמדני לצד עץ החיפוש השלם כלומר:

$$O(n+m) < O(n \cdot m) < O(2^{n+m} \cdot min(n,m))$$

מימוש **רקורסיבי**

```
public static int[][] generateMatrix(String X, String Y) {
        int[][] matrix = new int[X.length()+1][Y.length()+1];
        for(int i = 0 ; i < matrix.length; i++)
            matrix[i][0] = 0;
        for(int i = 0 ; i < matrix[0].length; i++)
            matrix[0][i] = 0;
        generateRec(matrix, X, Y, 1,1);
        return matrix;
    }

public static void generateRec(int[][] matrix,String X,String Y, int i ,int j) {
        if(i == matrix.length)
            return;
        if(j == matrix[0].length) {
            generateRec(matrix, X, Y, i + 1, 1);
        }
        else {
            if (X.charAt(i - 1) == Y.charAt(j - 1))
                matrix[i][j] = matrix[i - 1][j - 1] + 1;
            else
                matrix[i][j] = Math.max(matrix[i][j - 1], matrix[i - 1][j]);

            generateRec(matrix, X, Y, i, j + 1);
        }
    }

public static String dynamicRec(int[][] matrix,String X,String Y, int i,int j, int
length) {
        if(length == 0)
            return "";
        if(X.charAt(i-1) == Y.charAt(j-1))
            return dynamicRec(matrix,X,Y,i-1,j-1,length-1) + X.charAt(i-1);
        if(matrix[i][j-1] > matrix[i-1][j])
            return dynamicRec(matrix,X,Y,i,j-1,length);

        return dynamicRec(matrix,X,Y,i-1,j,length);
    }
public static String dynamic(String X, String Y) {
        int[][] matrix = generateMatrix(X,Y);
        int n = X.length(), m = Y.length();
        return dynamicRec(matrix, X, Y, n, m, matrix[n][m]);
    }
public static void main(String[] args) {
        String X = "abcbdab", Y = "bdcaba";
        System.out.println(dynamic(X,Y)); // bcba
    }
```

אם נרצה להחזיר את **כל** ה-LCS שלנו (יכול להיות שקיימות כמה מחרוזות LCS שונות באותו האורך).
נצרף את הפונקציות הבאות למימוש הרקורסיבי שהצגנו בדף הקודם.

```
public static void getAllRec(int[][] matrix, String X, String Y, int i , int j,
HashSet<String> set) {
   if(i == 0 || j == 0)
       return;

   int n = X.length(), m = Y.length();
   if(matrix[i][j] == matrix[n][m]) {
       set.add(dynamicRec(matrix,X,Y,i,j,matrix[n][m]));

       getAllRec(matrix,X,Y,i-1,j,set);
       getAllRec(matrix,X,Y,i,j-1,set);
       getAllRec(matrix,X,Y,i-1,j-1,set);
   }
}

public static HashSet<String> getAllLCS(String X, String Y) {
   int[][] matrix = generateMatrix(X,Y);
   int n = X.length(), m = Y.length();
   HashSet<String> set = new HashSet<>();
   getAllRec(matrix, X, Y, n, m, set);
   return set;
}

public static void main(String[] args) {
   String X = "abcbdab", Y = "bdcaba";
   HashSet<String> set = getAllLCS(X,Y);
   for (String str : set) {
       System.out.print(str + " , ");
   }
}
```