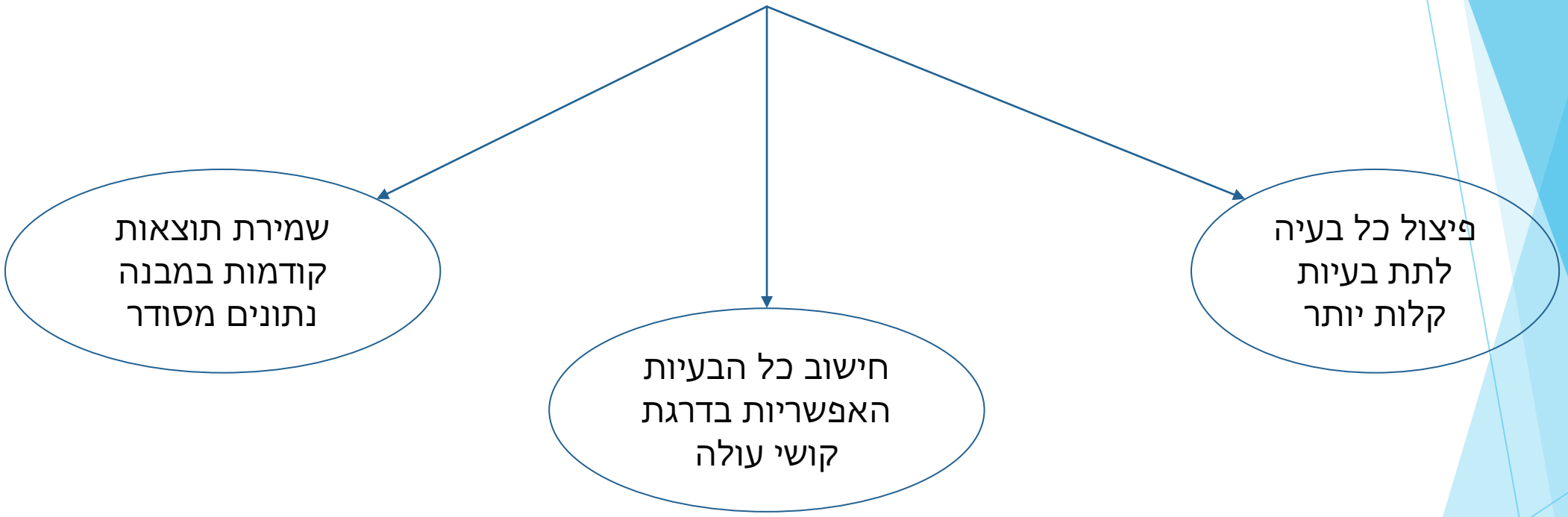
תרגול 3-4
תכנון דינאמי

אלגוריתמים 1 סמסטר א תשפ"ב

Dynamic programming
עקרונות מנחים



Kadane's algorithm

- בהינתן מערך בעל n איברים, מהו תת המערך שסכום איבריו הוא הגדול ביותר?
- <u>הערה:</u> תת המערך חייב להיות רצף של תאים

- בתרגול הראשון למדנו איך לפתור את הבעיה באמצעות אלגוריתם "הפרד ומשול" בסיבוכיות זמן O(nlogn), כעת נפתור את הבעיה באמצעות תכנון דינאמי בסיבוכיות - O(n).

- בהינתן מערך:

1	7	3	-13	2	1	10	-2	1	-20

- נאתחל משתנה שסופר את סכום תת המערך הנוכחי.
- נבנה מערך עזר, כך שכל תא יכיל את סכום המערך עד אותו התא.

H[i] = Max(H[i-1] + A[i], A[i])

- כעת נשאלת השאלה, מתי נרצה לקחת בחשבון את האיברים השליליים?
כאשר **איבר שלילי** ישאיר אותנו עם סכום כולל **חיובי**, ניקח אותו בחשבון.
אחרת **נאפס** את המשתנה הסוכם ונתחיל את הספירה מחדש בתא הבא.
- **בצורה כזאת התא ה-i במערך העזר מכיל את סכום התת מערך המקסימלי שמסתיים בתא ה-i מהמערך המקורי** (אם הסכום שלילי נסמן ב '-').
- מכאן שהתשובה הסופית תהיה **הערך המקסימלי** מבין כל הערכים **במערך העזר**.

- מערך העזר:

1	8	11	-	2	3	13	11	12	-

Pseudo Code:

```
sum, cur_sum, start_index, cur_start, end_index
    for i=1 to n
        cur_sum += arr[i]
        if cur_sum > sum
            sum ← cur_sum
            end_index ← i
            start_index ← cur_start
        if cur_sum < 0
            cur_start ← i+1
            cur_sum ← 0
return sum, start_index, end_index
```

<u>סיבוכיות:</u> O(n)

מציאת תת מטריצה עם סכום מקסימלי

- בהינתן מטריצה בגודל n*m, מהי תת המטריצה שסכום איבריה הוא הגדול ביותר?

<u>דוגמא:</u>

נתונה מטריצה:

2	1	-3	-4	5
0	6	3	4	1
2	-2	-1	4	-5
-3	3	1	0	3

- מהי תת המטריצה בעלת הסכום המקסימלי עבורה?
- איך נחשב זאת?

פתרון 1 - חיפוש שלם

- חיפוש שלם עובר על כל תתי המטריצות האפשריות, מחשב את הסכום שלהן ומחזיר לנו בסוף את התשובה הנכונה
- החיסרון שלו הוא בדרך כלל זמן ריצה גדול
- נגדיר מלבן באופן הבא:

i,j- פינה שמאלית עליונה של המלבן
l,k- פינה ימנית תחתונה של המלבן

Pseudo Code:

```
for i=1 to n
    for j=1 to m
        for k=i to n
            for l=j to m
                for x=i to k
                    for y=j to l
                        cur_sum += matrix[x][y]
                        if cur_sum > max_sum:
                            max_sum ← cur_sum
                            max_sum_indexes ← [i-k:j-l]
```

<u>סיבוכיות:</u> O(n^6)

פתרון 2 - מערך עזר

- ניצור מערך עזר עבור כל תתי השורות במטריצה
- נפעיל אלגוריתם לחיפוש תת מערך מקסימלי על מערך העזר (תרגיל ראשון)
- נמצא את התשובה

לדוגמא:

2	1	-3	-4	5
0	6	3	4	1
2	-2	-1	4	-5
-3	3	1	0	3

אם נרצה לחבר את השורות 2-1 ניצור את מערך העזר הבא:

2	4	2	8	-4

באמצעות אלגוריתם למציאת תת מערך מקסימלי נקבל את תת המערך בתאים (0-3).

זאת אומרת שתת המטריצה המקסימלית נתחמת באינדקסים: [1-2:0-3]

Pseudo Code:

```
for i=1 to n
    for j=i to n
        (create help_arr) //sum col[l] in mat to help_arr[l]
        for k=i to j
            for l=1 to m
                help_arr[l] += mat[k,l]
        cur_sum, s, e = MaxSubArr(help_arr)
                if cur_sum > max_sum:
                    max_sum ← cur_sum
                    max_sum_indexes ← [i-j:s-e]
```

<u>סיבוכיות:</u> O(n^4)

פתרון 3 - מטריצת עזר

- בפתרון זה נוכל להשתמש בתכנות דינאמי, רעיון שעומד מאחורי הפתרון הוא שאם נתונים לנו סכומים חלקיים נוכל בעזרתם לחשב את הסכום הכולל.
- נגיד שאנו רוצים לחשב את תת המטריצה הזו:

2	1
0	6

סכום השורה הראשונה = 3
סכום העמודה הראשונה = 2
אזי נוכל לקחת את התא (1,1) במטריצה להוסיף לו את השורה הראשונה ואת העמודה הראשונה, נחסיר את תא (0,0) שחושב פעמיים ונקבל את סכום המטריצה: 3+2+6-2=9

- **נייצר מטריצת עזר שהערך בכל תא יהיה הסכום של המטריצה מנקודת ההתחלה (0,0) ועד התא הנוכחי.**

<u>דוגמא:</u>

עבור המטריצה:

1	-3
6	3

מטריצת העזר תראה כך:

1	-2
7	7

איך בונים את מטריצת העזר H?

- תחילה, נאתחל את השורה והעמודה הראשונה:

$$H[0,0] = A[0,0]$$
$$H[0,j] = H[0,j-1] + A[0,j]$$
$$H[i,0] = H[i-1,0] + A[i,0]$$

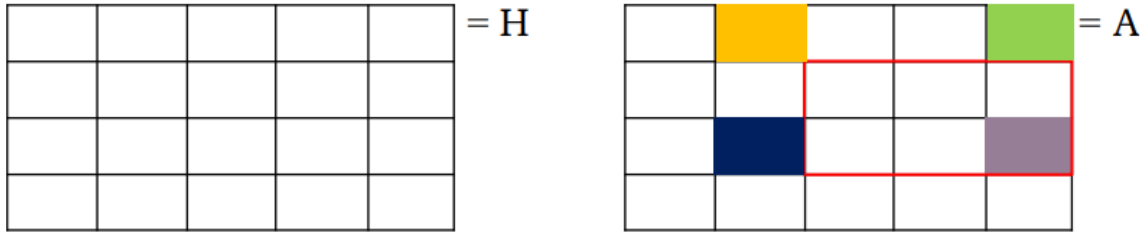
- את שאר מטריצת העזר נבנה באופן הבא:

$$H[i,j] = A[i,j] + H[i,j-1] + H[i-1,j] - H[i-1,j-1]$$

- לאחר שבנינו את מטריצת העזר, נעבור עליה ונחשב את הסכומים של כל תתי המטריצות האפשריות.

- כדי לחשב את סכום תת המטריצה המסומנת באדום, נבצע את החישוב הבא:

תא סגול - **תא ירוק** - **תא כחול** + **תא צהוב** = **תת מטריצה אדומה**.
(Purple - green - blue + yellow = red)



- מדוע?

תא סגול - גודל המטריצה [0-2:0-4]

תא ירוק - גודל המטריצה [0:0-4]

תא כחול - גודל המטריצה [0-2:0-2]

תא צהוב - גודל המטריצה [0:0-1]

ריבוע אדום - גודל המטריצה [1-2:2-4]

Pseudo Code:

```
Create H //auxiliary matrix
for i=1 to n
    for j=1 to m
        for k=i to n
            for l=j to m
                cur_sum = H[i,j] - H[i,l-1] - H[k-1,j] + H[k-1,l-1] //for matrix (k,l)-(I,j)
                if cur_sum > max_sum:
                        max_sum ← cur_sum
                        update max_sum_indexes
```

<u>סיבוכיות:</u> O(n^4)

פתרון 4 - Super Help Array

- ראינו שבאמצעות תכנון דינאמי ניתן להוריד את הסיבוכיות מ- O(n^6) ל- O(n^4).
  כעת ננסה לשפר את הסיבוכיות ל- O(n^3) באמצעות **מערך העזר**.
- במקום שנאפס את המערך בכל שלב, **נוסיף** לו את השורה החדשה, נפעיל אלגוריתם למציאת תת מערך מקסימלי ונמצא את התשובה.
- <u>נוסחה:</u>

H[i] = mat[row_i : col_start - col_end] //sum row i

<u>דוגמא:</u>

start R =
start C =
end R =
end C =
sum =

2	1	-3	-4	5
0	6	3	4	1
2	-2	-1	4	-5
-3	3	1	0	3

Pseudo Code:

```
for i=1 to m
    create help_arr
        for j=i to m
            for k=1 to n
                help_arr[k] += mat[j,k]
            cur_sum, s, e = MaxSubArry(help_arr)
                if cur_sum > max_sum:
                    max_sum ← cur_sum
                    max_sum_indexes ← [s-e:i-j]
```

<u>סיבוכיות:</u> O(n^3)

מציאת תת מחרוזת משותפת ארוכה ביותר

LCS

- בהינתן שתי מחרוזות X בגודל n, ו-Y בגודל m.
  נרצה למצא את **תת המחרוזת המשותפת הארוכה ביותר** Longest common substring
- תת המחרוזת לא חייבת להיות רצופה

<u>דוגמא:</u>

קלט: Y = abcbdcbad, X = "abcabdc"

פלט: LCS(X, Y) = abcbdc

פתרון 1 - חיפוש חמדני

- נשים לב שאלגוריתם חמדן לא בהכרח ייתן פתרון אופטימלי.

<u>דוגמא:</u>
X=aILOVEALGOb, Y=bILOVEALGOa

בדוגמא זו, נקבל: GreedyLCS(X,Y)=‘a’, GreedyLCS(Y,X)=‘b’.
למרות שתת המחרוזת המשותפת הארוכה ביותר הינה: ‘ILOVEALGO’.

כדי שאלגוריתם חמדן יעבוד במקרה זה, נצטרך **לשפר** אותו.

- נבחר במחרוזת הקצרה יותר מבין השתיים, X לצורך העניין שאורכה m.
- נבנה מערך עזר שיכיל את מספר המופעים של כל אות במחרוזת X.
הרעיון הוא לא לחפש את האותיות ב-Y שכבר לא מופיעות ב-X.

דוגמת הרצה - חמדן משופר: X=abca, Y=adcbc

מערך עזר ל-X:

X	a	b	c	d
help	2	1	1	0

נעבור על מחרוזת Y:

1. i=0, a מופיעה, index=0, ans=a, --help[index], start=1

X	a	b	c	d
help	1	1	1	0

2. i=1, d לא מופיעה, נעבור לשלב הבא
3. i=2, c מופיעה, index=2, ans=ac, --help[index], start=3

X	a	b	c	d
help	1	1	0	0

4. i=3, b מופיעה , index=1, אבל אנחנו מתחילים חיפוש ב-X החל מ-start=3 ולכן help[index]=0

X	a	b	c	d
help	0	1	0	0

5. סיימנו את בניית התת מחרוזת ונחזיר את ans

Pseudo Code:

```
String greedyWithHelp(String X, String Y)
      help[26], m = X.length(), n = Y.length()
      for i=0; to m //O(min(m,n))
            place = X.charAt(i)-'a'
            help[place]++
      end-for
      ans = "", start = 0, index = 0, i = 0
      while (i<n && index<m) //O(m+n)
            place = Y.charAt(i) - 'a'
            if (help[place] > 0)
                  index = X.indexOf(Y.charAt(i), start)
                  if (index != -1)
                        ans = ans + Y.charAt(i)
                        start = index+1
                        help[place]--
                  else help[place] = 0
            end-if
      i++
      return ans
end-greedyWithHelp
```

סיבוכיות: O(m+n) + O(min(m,n))

פתרון 2 - חיפוש שלם

- הרעיון הוא לקחת את כל תתי המחרוזות האפשריות של מחרוזת X,
את כל תתי המחרוזות האפשריות של מחרוזת Y
ולבדוק האם קיימת תת מחרוזת משותפת תואמת.

- <u>דוגמא:</u>

X=abc, m=3 ולכן ישנן 2^3 - 1 תתי מחרוזות אפשריות.

	a	b	c	תת-מחרוזת
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

- <u>סיבוכיות:</u>
O(2^m) בניית כל תתי המחרוזות של X
O(2^n) בניית כל תתי המחרוזות של Y
O(min(m,n)) מציאת תת מחרוזת משותפת

<u>סה"כ:</u> O(2^(m+n) * min(m,n))

פתרון 3 - תכנון דינאמי

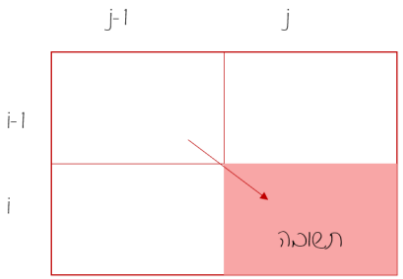
- למציאת הפתרון הכולל נגיד תת בעיה- פונקציית מטרה ושמירת כל הערכים של הפונקציה בטבלה (מטריצה).
- <u>תת הבעיה:</u>
  f(i,j) - אורך המחרוזת המשותפת הארוכה ביותר של מחרוזת X עד התא ה-i ושל מחרוזת Y עד התא ה-j.
- <u>התשובה</u> לבעיה הכוללת תמצא ב- f(n,m).

- הגדרת הפונקציה:

$f(0,j) = f(i,0) = 0$

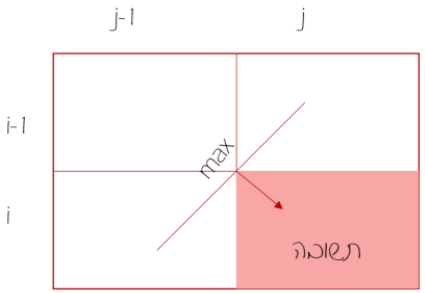
if $X[i] = Y[j] \rightarrow$

$f(i,j) = 1 + f(i-1, j-1)$



else $\rightarrow$

$f(i,j) = Max\{ f(i, j-1), f(i-1,j) \}$



- <u>החזרת המחרוזת עצמה:</u>

נתחיל מהתא הימני התחתון ביותר של המטריצה, וחוזרים אחורה לפי הפונקציה:

אם התווים זהים- נשרשר את התו לתשובה ונחזור באלכסון
אם התווים אינם זהים- אז נבחר את המקסימום מבין התא מעלי והתא משמאלי

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Pseudo Code:

```
//build matrix
f[n+1][m+1]
for i=0 to n+1
    f[i][0] = 0
for j=0 to m+1
    f[0][j] = 0
for i=0 to n+1
    for j = 1 to m+1
        if X[i] = Y[j]
            f[i][j] = 1 + f[i-1][j-1]
        else
            f[i][j] = Max(f[i][j-1], f[i-1][j])
```

```
//reconstruction answer
i = n, j = m
    while f[i][j] != 0
        if X[i-1] = Y[j-1]
            ans = X[i-1] + ans
            i--, j--
            else
                if f[i][j-1] > f[i-1][j]
                    j--
                else i--
return ans;
```

<u>סיבוכיות:</u>

לבניית המטריצה O(n*m)

לשחזור המחרוזת O(n+m)

Subset Sum

בהינתן קבוצת מספרים A בגודל n ומספר שלם m,
האם ניתן להגיע למספר m באמצעות חיבור
מספרים כלשהם מהקבוצה A?

A= {1, **3**, **6**, **4**, **3**, 25, **73**}, m=89

A= {**1**, 2, 4, 5, **8**, **10**}, m=19

A= {5, 15, 38, 10, 1}, m=17

A= {51, **103**, 12, **45**, **2**, 66, 1, 13}, m=150

שלב ראשון- מציאת נוסחה רקורסיבית:

```
isSubsetSum(set, n, sum)
= isSubsetSum(set, n-1, sum) ||
  isSubsetSum(set, n-1, sum-set[n-1])
Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubsetSum(set, n, sum) = true, if sum == 0
```

כלומר ננסה להגיע לסכום המבוקש:

- בלי האיבר האחרון (נצטרך להגיע לסכום עם שאר המערך)
- **או** עם האיבר האחרון (נצטרך להגיע לסכום חדש עם שאר המערך)

<u>סיבוכיות זמן:</u> $O(2^n)$ (במקרה הגרוע - הרקורסיבי)

ננסה בגישת ה**זיכרון תמורת מהירות**- dynamic programming
ונגדיר טבלה שתעזור לנו לשמור את התוצאות.

DP[i] [j] = true **if** there exists a subset of elements from A[0….i] with **sum value = ‘j’.**

```
if (A[i] > j)
DP[i][j] = DP[i-1][j]
else
DP[i][j] = DP[i-1][j] OR DP[i-1][ j -A[i]]
```

האיבר החדש גדול יותר מסכום המטרה
ולכן בהכרח לא ישתתף בפתרון אם קיים

ננסה להרכיב את הסכום j עם i-1 האברים הראשונים
או ננסה להרכיב את הסכום A[i]-j עם i-1 האברים הראשונים.

A= {2, 5, 3, 4} , m= 6

	0	1	2	3	4	5	6
ϕ	T	F	F	F	F	F	F
{2}	T						
{2,5}	T						
{2,5,3}	T						
{2,5,3,4}	T						

```
if (A[i] > j)
DP[i][j] = DP[i-1][j]
else
DP[i][j] = DP[i-1][j] OR DP[i-1][ j -A[i]]
```

A= {2, 5, 3, 4} , m= 6

	0	1	2	3	4	5	6
ϕ	T	F	F	F	F	F	F
{2}	T	F	T	F	F	F	F
{2,5}	T	F	T	F	F	T	F
{2,5,3}	T	F	T	T	F	T	F
{2,5,3,4}	T	F	T	T	T	T	T

```
if (A[i] > j)
DP[i][j] = DP[i-1][j]
else
DP[i][j] = DP[i-1][j] OR DP[i-1][ j -A[i]]
```

<u>סיבוכיות זמן = סיבוכיות מקום:</u> O(n*m)

זמן ריצה- פסודו פולינומי:

Pseudo-polynomial time

From Wikipedia, the free encyclopedia

In computational complexity theory, a numeric algorithm runs in **pseudo-polynomial time** if its running time is a polynomial in the *numeric value* of the input (the largest integer present in the input) — but not necessarily in the *length* of the input (the number of bits required to represent it), which is the case for polynomial time algorithms.

In general, the numeric value of the input is exponential in the input length, which is why a pseudo-polynomial time algorithm does not necessarily run in polynomial time with respect to the input length.

An NP-complete problem with known pseudo-polynomial time algorithms is called weakly NP-complete. An NP-complete problem is called strongly NP-complete if it is proven that it cannot be solved by a pseudo-polynomial time algorithm unless P=NP. The strong/weak kinds of NP-hardness are defined analogously.

ובקצרה-

הכוונה באלגוריתם פולינומי הוא פולינומי **בערך הקלט** ולא **באורך הקלט** (מספר ש ביטים הנדרשים לייצר אותו).

נשים לב, m הוא **ערך הקלט ולא אורך הקלט,** לכן האלגוריתם הוא פסודו פולינומי.

Edit Distance

בהינתן שתי מחרוזות של תווים,
מה מספר הפעולות **המינימלי** שיש לעשות כדי להגיע ממחרוזת **א** למחרוזת **ב**?

<u>הפעולות המותרות:</u>

1. מחיקת תו
2. שינוי תו
3. הוספת תו

str1= “abba”, str2= “abca” ➔ ans= 1

str1= “algro”, str2= “algo” ➔ ans= 1

שלב ראשון- מציאת נוסחה רקורסיבית:

```
m: Length of str1 (first string)
n: Length of str2 (second string)
```

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
   1. Insert: Recur for m and n-1
   2. Remove: Recur for m-1 and n
   3. Replace: Recur for m-1 and n-1

אם **האות האחרונה** בשתי המחרוזות **זהה**, נרצה למצוא את ה-ED של שאר המחרוזת (קפד זנבו).
אם **האות האחרונה** בשתי המחרוזות **לא זהה**, נבדוק את האופציות המקבילות למחיקה/ הוספה/ החלפה ונבחר את המינימלית.

<u>סיבוכיות זמן (רקורסיבית):</u> $O(3^n)$

Pseudo Code:

```
editDist(str1, str2, m, n):

if (str1[m]=str2[n]):
        return editDist(str1, str2, m-1, n-1)

else:
        return 1  +min {(editDist(str1, str2, m, n-1), /insert
                        (editDist(str1, str2, m-1, n), /delete
                        (editDist(str1, str2, m-1, n-1) /replace }
```

ננסה בגישת ה**זיכרון תמורת מהירות**- dynamic programming
ונגדיר טבלה שתעזור לנו לשמור את התוצאות.

ED[n] [m]= minimum number of operations to get from str1[0…n] to str2[0…m]

```
if (str1[m]=str2[n]):
        ED[n][m]= ED[n-1][m-1]

else:
        ED[n][m] =  1 + min (ED[n][m-1], /insert
                             ED[n-1][m], /delete
                             ED[n-1][m-1] /replace)
```

str1= Sunday, str2= Saturday

	ϕ	S	su	sun	sund	sunda	sunday
ϕ	0	1	2	3	4	5	6
s	1						
sa	2						
sat	3						
satu	4						
satur	5						
saturd	6						
saturda	7						
saturday	8						

str1= Sunday, str2= Saturday

	ϕ	S	su	sun	sund	sunda	sunday
ϕ	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
Sa	2	1	1	2	3	3	4
sat	3	2	2	2	3	4	4
satu	4	3	2	3	3	4	5
satur	5	4	3	3	4	4	5
saturd	6	5	4	4	3	4	5
saturda	7	6	5	5	4	3	4
saturday	8	7	6	6	5	4	3

סיבוכיות זמן = סיבוכיות מקום: $O(n^{*}m)$

שקופיות בונוס
הרציונל מאחורי DP

אז מה זה Dynamic Programming?

לפני הכל ניזכר בחברה ישנה- רקורסיה.

אלגוריתם רקורסיבי- הוא אלגוריתם אשר על מנת לפתור בעיה מסוימת, מפעיל את עצמו על מקרים פשוטים יותר של הבעיה (ובמקרים רבים: על תתי בעיות).
בדרך כלל יכלול האלגוריתם "תנאי עצירה", שיביא להפסקת הרקורסיה ברמה שבה הפתרון נתון מראש, שאם לא כן תהיה זו לולאה אינסופית רקורסיבית.
(Wikipedia)

הרבה פעמים בפונקציות רקורסיביות אני מבצעים את אותה פעולה שוב ושוב בצורה מיותרת למדי.

דוגמה: סדרת פיבונאצ'י.

סדרת פיבונאצ'י (Fibonacci) היא הסדרה ששני איבריה הראשונים הם: '1',
וכל איבר לאחר מכן שווה לסכום שני קודמיו.
בהתאם לכך, איבריה הראשונים של הסדרה הם: {1,1,2,3,5,8,13,21,34,55,89,144....}

בניסוח פורמלי יותר, הגדרה רקורסיבית של הסדרה ניתנת על ידי **תנאי ההתחלה**:

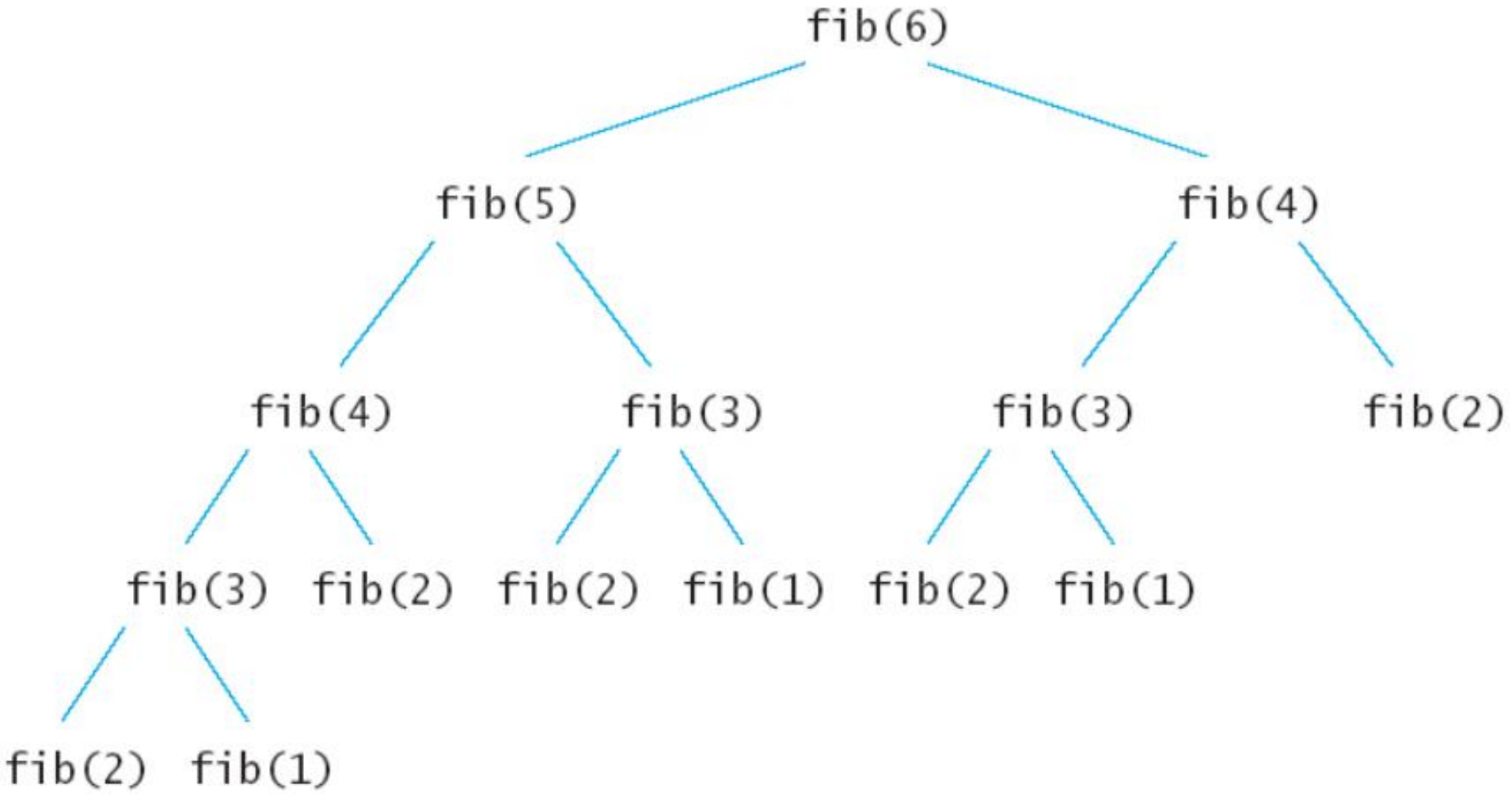
n=0,1:
F(0)= 1, F(1)= 1

n>1:
F(n)= F(n-1)+ F(n-2)

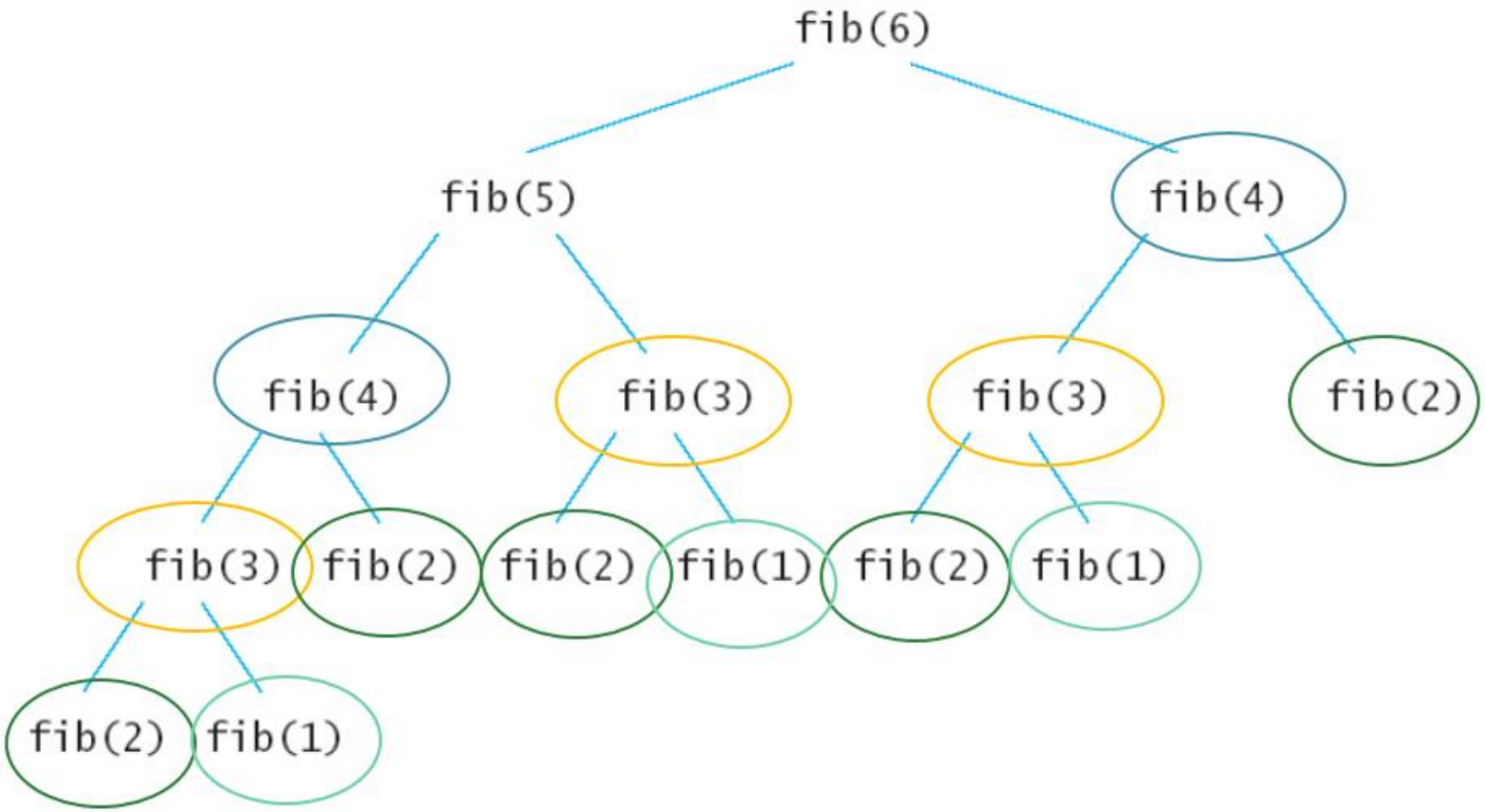
<u>אלגוריתם רקורסיבי:</u>

```
int fib (int n) {
        if (n < 2)
            return 1;
        return fib(n-1) + fib(n-2);
    }
```

אילוסטרציית זמן ריצה:



נתבונן מקרוב:



איך נוכל להימנע מהחזרות המיותרות?

פה נכנס לתמונה dynamic programming.

```
void fib () {
        fibresult[0] = 1;
        fibresult[1] = 1;
        for (int i = 2; i<n; i++)
            fibresult[i] = fibresult[i-1] + fibresult[i-2];
    }
```

F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)
1	1	2	3	5	8	13

עד כאן נשמע מושלם!!

בכל זאת, ל-DP יש חיסרון משמעותי ביחס לרקורסיה, **מה הוא?**

<u>סיבוכיות זיכרון</u>

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

הרעיון מאחורי תכנות דינמי הוא **המרה של סיבוכיות זמן תמורת סיבוכיות מקום**.
אנו שומרים תוצאות של חישובים שעשינו כדי **לחסוך** את **זמן** ביצוע החישובים באלו בעתיד,
אך **נבזבז מקום**.

Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Bottom up vs. Top Down:

- **Top Down-** I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming. Recursive attitude

- **Bottom Up-** I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder. DP attitude

איך ניגשים לבעיית DP?

1. ניסוח השאלה הרקורסיבית בצורה ברורה, כך שנוכל להרכיב ממנה תת בעיות עם ניסוח זהה.
2. מציאת תתי בעיות, הבנת היחס בניהן (חיבור, פעולת min, וכו') והרכבת נוסחת נסיגה שתקשר בין כל בעיה לתתי הבעיות מהן היא מורכבת.
3. מציאת מקרי הקצה והגדרתם באופן ידני.
4. הבנת בעיית המאקרו אותה אנחנו באים לפתור בשאלה, לבסוף זה יהיה הפלט שלנו (לפעמים יש כמה תאים שיכולים להיות התשובה לבעיה שלנו ונצטרך להבין מה היחס ביניהם).
5. התאמת מבנה נתונים לבעיה, כך שכל תא במבנה יהיה מתאים לתת בעיה מסוימת.
6. ניתוח סיבוכיות זמן ומקום (לרוב הן יהיו שוות אחת לשנייה).
7. נמלא את מבנה הנתונים מתת הבעיות הקלות ביותר עד לקשות בדרגת קושי עולה.

בצורה כזו ניתן לפתור כל בעיה בזמן קבוע מכיוון שכבר פתרנו את כל תתי הבעיות ושמרנו את התוצאות הרלוונטיות.

דוגמה נוספת לשימוש ב-DP:

Let us say that you are given a number **N**, you've to find the number of different ways to write it as the sum of 1, 3 and 4.

For example, if N = 5, the answer would be 6.

- 1 + 1 + 1 + 1 + 1
- 1 + 4
- 4 + 1
- 1 + 1 + 3
- 1 + 3 + 1
- 3 + 1 + 1

נמצא נוסחת נסיגה המתארת את הבעיה-
כל מספר n ניתן לכתוב כך:

N=1+{..n-1..}

N=3+{..n-3..}

N=4+{..n-4..}

ולכן:

DW(n)= DW(n-1)+ DW(n-3)+ DW(n-4)

DW(n): מספר הדרכים השונות לכתוב את n כסכום של {1,3,4}

הגדרת טבלה לשמירת תוצאות ישנות:

```
DP[0] = DP[1] = DP[2] = 1; DP[3] = 2;
    for (i = 4; i <= n; i++) {
      DP[i] = DP[i-1] + DP[i-3] + DP[i-4];
    }
```

DW(0)	DW(1)	DW(2)	DW(3)	DW(4)	DW(5)	DW(6)	DW(7)	DW(8)
1	1	1	2	4	6	9	15	25
	1	1+1	1+1+1 3	1+1+1+1 1+3 3+1 4	1+1+1+1 +1 1+1+3 1+3+1 1+4 3+1+1 4+1	…	…	…

שאלות ממבחנים

תשפ"א - מועד א' סמסטר א'

בעיה מס' 1

40 = 20 + 20

20

א) בהינתן **שתי** סדרות (מערכים) של מספרים שלמים,
יש לפתח אלגוריתם שמוצא את אורכה של תת-הסדרה
המשותפת הארוכה ביותר.
- יש להגדיר קובץ בשם Q1.java
- יש לבנות פונקציה סטטית

```
public static int lcs(int[] X, int[] Y) {. . .}
```

קלט : שני מערכים של מספרים שלמים.
פלט : אורכו של תת-המערך המשותף הארוך ביותר.

20

ב) בהינתן **שלוש** סדרות (מערכים) של מספרים שלמים,
יש לפתח אלגוריתם שמוצא את אורכה של תת-הסדרה
המשותפת הארוכה ביותר.

- באותו קובץ Q1.java :
- יש לבנות פונקציה סטטית

```
public static int lcs3(int[] X, int[] Y, int[] Z){. . .}
```

קלט : שלוש סדרות (מערכים) של מספרים שלמים.
פלט : אורכה של תת-הסדרה המשותפת הארוכה ביותר.

אלגוריתם, הוכחות, סיבוכיות ודוגמא.

תשפ"א - מועד א' סמסטר א'

בעיה מס' 3:

20

N אנשים צריכים לעבור **גשר** (משמאל לימין) תחת האילוצים הבאים:

1) לאיש מספר **i** זמן המעבר הוא x_i.

2) בכל שלב יכולים רק שניים לכל היותר לעבור את הגשר בו-זמנית ועליהם תמיד לשאת פנס.

3) לאחר ש-**2** אנשים עוברים את הגשר, מישהו שכבר עבר חייב לחזור בחזרה כדי להחזיר פנס, פרט למקרה כאשר כולם כבר חצו את הגשר.

4) עלות של פעימת מעבר על הגשר שווה למקסימום זמני מעבר של העוברים בו-זמנית באותה הפעימה בתוספת זמן המעבר של מי שמחזיר את הפנס.

5) סה"כ עלות המעבר הכולל היא סכום עלויות המעברים בכל הפעימות.

6) יש למצוא פתרון עבור המקרה: $N=4$, $x_1=1,\ x_2=2,\ x_3=5,\ x_4=10$ כשעלות המעבר הכולל היא מינימאלית.

7) יש להסביר למה.

8) מי שיפתור את הבעיה במקרה הכללי ביותר יקבל **בונוס**.

תודה רבה ! ☺