סיכום אלגוריתמים 2

הסברים + מימושים

הערה לכלל האלגוריתמים אותם נלמד הסמסטר - קלט האלגוריתם יהיה $O(n^2)$, וסיבוכיות האלגו' $O(n^3)$ או בשאיפה אליו מלמעלה.

בתחילת השיעור הביא פרו' לויט חידה "חידת השוקולד": יש לשבור שורה של קוביות, כאשר כל שבירה עולה: $i * (n-i)$ (i מסמל את מקום השבירה, n אורך השורה). מהו המחיר המינימאלי אותו נשלם? ראינו כמה סגנונות של שבירה (אינדוקטיבי, רקורסיבי, פיבונאצי') ובכולם הסכום היה קבוע. הוכחנו באינדוקציה שאכן תמיד הסכום שיתקבל יהיה זהה בכל צורת שבירה שתהיה.

בעיה מס' 1 - בעיית הבקבוקים

תיאור הבעיה: נאמר שיש לי 2 כלי קיבול, האחד בגודל 3 והשני בגודל 5, כיצד ניתן להגיע <u>הכי מהר</u> למצב בו יש לי כלי אחד עם 4 ליטר והשני ריק?

חוקי המשחק:

א. כל כלי ניתן למלא עד הסוף ולרוקן עד הסוף,

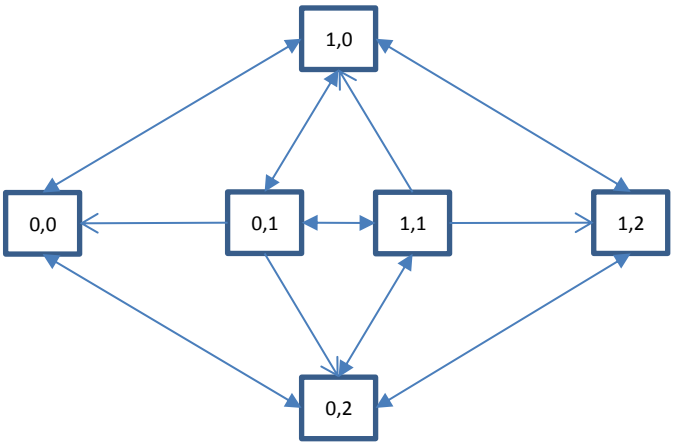
ב. ניתן למלא כלי קטן יותר ולשפוך לכלי גדול יותר

ג. לא ניתן למלא כלי במים חדשים - שלא עד סופו, ולא ניתן לרוקן כלי עד חציו - אלא בשפיכה לכלי אחר

דוגמא לפתרון:

$[0,0] \to [0,3] \to [3,0] \to [3,3] \to [5,1] \to [0,1] \to [1,0] \to [1,3] \to [4,0]$

הבעיה הנ"ל לא נפתרת בשיעור זה אלא רק בעוד מספר שיעורים. בשביל להגיע לפתרון יש לעבור דרך ארוכה של כמה אלגוריתמים שיסייעו לנו להגיע אל הפתרון.



<u>שלב ראשון</u>: נמיר את בעיית הבקבוקים לגרף:
כל אפשרות של בקבוק תיוצג ע"י קודקוד, וצלע תסמן האם ניתן לעבור מקודקוד לקודקוד (כמובן שהגרף הוא מכוון) במילים אחרות יש לנו גרף $B_{n,m}$ (כאשר n,m אלו 2 הבקבוקים שעומדים לרשותנו). נביא דוגמא פשוטה ביותר: $B_{1,2}$.
חץ דו כיווני מסמל שניתן ללכת לשני הצדדים, לעומת חץ חד כיווני - המסמל שניתן ללכת לצד אחד אך לא ניתן לחזור באותה הדרך.

נאמר ויש לי איזה בקבוק נתון: $[a,b]$, בגרף $B_{n,m}$, ז"א: $0 \le a \le n$, $0 \le b \le m$. מהן האופציות העומדות לרשותי? נתחיל עם ארבעת האופציות הפשוטות:

1. $[a,b] \to [0,b]$
2. $[a,b] \to [a,0]$
3. $[a,b] \to [n,b]$
4. $[a,b] \to [a,m]$

אלא שישנן עוד 2 אופציות שיש לשים לב אליהן:

5. $[a,b] \to [\min(a+b,n), a+b-\min(a+b,n)]$
6. $[a,b] \to [a+b-\min(a+b,m), \min(a+b,m)]$

אופציות אלו מייצגות את השפיכה מבקבוק אחד לבקבוק שני ולהפך.
נמחיש בדוגמא את אופציות 5,6:

$[a=0,b=2] \to [\min(0+2,1), 0+2-\min(0+2,1)] \to [1,1]$
$[a=0,b=1] \to [1+1-\min(1+1,2), \min(1+1,2)] \to [0,2]$

נקודה נוספת: מספר הקודקודים שיהיו בגרף כלשהו: $B_{n,m}$ יהיה: $(n+1)*(m+1)$, ובדוגמא לעיל: $2*3=6$.

צורת המימוש של חלק זה:

נבנה מטריצה ריבועית בגודל $dim=(n+1)(m+1)$, המכילה אפסים, כאשר כל עמודה וכל שורה מייצגת את הקשר שבין 2 קודקודים: אם יש קשר יסומן המיקום במטריצה באחד, אם אין קשר הערך יישאר 0. כרגע נשאיר את האלכסון להיות 0 (קשר בין קודקוד לעצמו).

בדוגמא לעיל, דיברנו על גרף $B_{1,2}$, התוצאה שאנחנו צריכים לקבל:

	[0,0]	[0,1]	[0,2]	[1,0]	[1,1]	[1,2]
[0,0]	0	0	1	1	0	0
[0,1]	1	0	1	1	1	0
[0,2]	1	0	0	0	1	1
[1,0]	1	1	0	0	0	1
[1,1]	0	1	1	1	0	1
[1,2]	0	0	1	1	0	0

כאשר מדובר על גרף $B_{2,1}$, נקבל מטריצה כזו, וראה להלן

	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]
[0,0]	0	1	0	0	1	0
[0,1]	1	0	1	0	0	1
[1,0]	1	1	0	1	1	0
[1,1]	0	1	1	0	1	1
[2,0]	1	0	0	1	0	1
[2,1]	0	1	0	0	1	0

עוד נקודה שחשוב לשים לב אליה במימוש: אנחנו משתמשים במטריצה דו מימדית, אך בפועל - כל קודקוד מיוצג על ידי 2 ערכים (ערך לכל בקבוק), לכן היה צורך להשתמש כאן במטריצה תלת מימדית, אך דבר זה היה מאוד מקשה ומבלבל, ולכן נצטרך להשתמש בפונקציה שתדע לקשר אותנו - בין בקבוק לבין המיקום שלו במטריצה:

```
public static int getIndex(int i, int j, int n) {
    return (n+1)*i+j;
}
```

כאשר הפרמטרים הם: i - גובה המים בבקבוק הראשון,
j - גובה המים בבקבוק השני,
n - הגודל המרבי של אחד מהבקבוקים, איך נדע איזה מהם? חשוב לשים לב:

אם בגרף $B_{n,m}$ מתקיים: $n>m$ - הערך שישלח הוא n.

אם $n<m$ הפונקציה עדיין תעבוד, אך בסופו של דבר נקבל מטריצה בעלת סדר שונה, וראה בדוגמאות להלן)

דוגמאות להמחשה: $B_{n=1,m=2}$

הקודקוד [0,1] היכן הוא ממקום בשורה?

getIndex(0, 1, 2) return (2+1)*0+1; → $index(1)$

הקודקוד [1,1] היכן הוא ממקום בשורה?

getIndex(1, 1, 2) return (2+1)*1+1; → $index(4)$

במקרה ההפוך, ז"א: $B_{n=2,m=1}$:

הקודקוד [0,1] היכן הוא ממקום בשורה?

getIndex(0, 1, 1) return (1+1)*0+1; → $index(1)$

הקודקוד [1,1] היכן הוא ממקום בשורה?

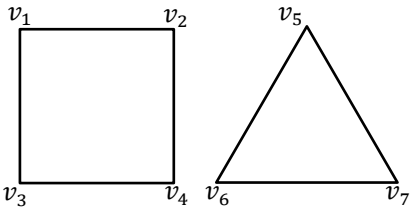
getIndex(1, 1, 1) return (1+1)*1+1; → $index(3)$

מימוש שלב ראשון: יצירת מטריצה המייצגת קשר בין 2 קודקודים על ידי צלע אחת בלבד:

```
/**
 * The purpose of the program: return a matrix that represents a connection between vertex
 * @author Yosef Zohar, Department of CS, Second academic year, Ariel University
 *
 */
public class shiur01_InitRibsMatrix {
	/**
	 * פונקציה המקבלת ערך של כל בקבוק ומחזירה את המיקום של האינדקס
	 * בשורה או בעמודה של המטריצה
	 * @param i Value of first bottle
	 * @param j Value of second bottle
	 * @param n The size of the matrix
	 * @return the position in the matrix corresponds to the value of two bottles
	 */
	public static int getIndex(int i, int j, int n) {
		return (n+1)*i+j;
	}

	/**
	 * פונקציה זו יוצרת מטריצה המייצגת צלעות בין 2 קודקודים
	 * לכל קודקוד המייצג את המצב של 2 הבקבוקים יש אפשרות להגיע למקס' 6 אפשרויות
	 * @param n The maximum value of the first bottle
	 * @param m The maximum value of the second bottle
	 * @return Matrix of zero and one, 0 - no connection, 1 - there is a connection
	 */
	public static int[][] initRibs(int n, int m) {
		System.out.println("The maximum value of the first bottle(n) = "+n);
		System.out.println("The maximum value of the second bottle(m) = "+m);
		System.out.println("********************["+n+","+m+"]********************");
		System.out.println("*********************************************\n");

		int dim = (n+1)*(m+1); // מספר הקודקודים בגרף
		int mat[][] = new int[dim][dim];
		int ind1, ind2;
		for (int i=0; i<=n; i++) { // הריצה היא עד קטן שווה בשביל להיות בהתאמה אם גודל המטריצה
			for (int j=0; j<=m; j++) {
				ind1 = getIndex(i,j,m);
				// קודקודים 1 עד 4 - המקרה הפשוט
				mat[ind1][getIndex(0,j,m)] = 1; // Rib 1
				mat[ind1][getIndex(n,j,m)] = 1; // Rib 2
				mat[ind1][getIndex(i,0,m)] = 1; // Rib 3
				mat[ind1][getIndex(i,m,m)] = 1; // Rib 4

				ind2 = getIndex(Math.min(i+j,n), i+j-Math.min(i+j,n),m);
				mat[ind1][ind2] = 1; // Rib 5

				ind2 = getIndex(i+j-Math.min(i+j,m), Math.min(i+j,m),m);
				mat[ind1][ind2] = 1; // Rib 6
			}
		}
		// הסקנו כי המסלול בין קודקוד לעצמו הוא אפס
		for (int t=0; t<dim; t++) mat[t][t] = 0;
		return mat;
	}
```

```
	/**
	 * פונקציה להדפסה מסודרת של טבלת הבקבוקים כפי שהובא לעיל בתאוריה
	 * @param mat Matrix after FW
	 * @param n The maximum value of the first bottle
	 * @param m The maximum value of the second bottle
	 */
	public static void printMat(int mat[][], int n, int m) {
		int stop = 0;
		String arr[] = new String[2*n*m + 2];
		int go = 0;
		while (stop<=n) {
			for (int j=0; j<=m; j++) {
				arr[go] = "["+stop+","+j+"]\t";
				go++;
				System.out.print("\t["+stop+","+j+"]");
			}
			stop++;
			if (stop>n) break;
			for (int j=0; j<=m; j++) {
				System.out.print("\t["+stop+","+j+"]");
				arr[go] = "["+stop+","+j+"]\t";
				go++;
			}
			stop++;
		}

		System.out.println("\n");
		for (int i=0; i<mat.length; i++) {
			System.out.print(arr[i]);
			for (int j=0; j<mat[0].length; j++) {
				System.out.print("  "+mat[i][j]+"\t");
			}
			System.out.println("\n");
		}
	}
}
```

בשיעור האחרון הצלחנו למצוא קשר ישיר בין קודקוד לקודקוד ע"י יצוג של מטריצה. חשוב לשים לב - היצוג הוא של קשר ישיר בלבד, נמחיש זאת בעזרת דוגמא: נחזור לגרף משיעור שעבר: $B_{1,2}$, ניתן לומר שיש קשר ישיר בין הקודקודים האלו: $[0,0] \to [1,0]$, וכן בין הקודקודים האלו: $[1,0] \to [1,2]$, אך בין קודקוד $[0,0]$ לבין קודקוד $[1,2]$ - אמנם אין קשר ישיר, אך קשר עקיף יש: $[0,0] \to [1,0] \to [1,2]$, וזה מה שאנחנו רוצים לעשות בשיעור הזה:



<u>שלב שני</u>: למצוא האם קיים קשר כלשהו בין קודקוד לקודקוד: ז"א: תחילה נבדוק האם יש קשר ישיר - אם לא, נחפש קשר עקיף. מתוך שזו המשימה שלנו - נשנה את המטריצה למטריצה בוליאנית, כאשר קיים קשר כלשהו בין 2 קודקודים - המטריצה תקבל ערך אמת.

לפני שנגיע לשלב השני בבעיה - פרו' לויט התחיל את השיעור בהסבר על גרפים לא מכוונים. נאמר ואנחנו מדברים על השלב הראשון, רק שהפעם נעשה זאת בעזרת מטריצה בוליאנית, הרי שנקבל:
(גם כאן - נאמר כי אין צלע בין קודקוד לעצמו)

	[v_1]	[v_2]	[v_3]	[v_4]	[v_5]	[v_6]	[v_7]
[v_1]	F	T	T	F	F	F	F
[v_2]	T	F	F	T	F	F	F
[v_3]	T	F	F	T	F	F	F
[v_4]	F	T	T	F	F	F	F
[v_5]	F	F	F	F	F	T	T
[v_6]	F	F	F	F	T	F	T
[v_7]	F	F	F	F	T	T	F

ניתן לראות בבירור שקיבלנו מטריצה סימטרית, ז"א: ניתן היה לחשב רק את המשולש העליון ולקבל את התוצאה.

בעצם - בשביל לבדוק האם גרף הוא מכוון - מספיק לבדוק שיש סימטריה במטריצה המייצגת אותו

מילוי של המשולש העליון יעלה $\frac{n(n-1)}{2}$ פעולות.

עד כאן הקלט, מה נרצה לעשות בפלט? אפשר לבנות אלגוריתם שמחזיר האם קיים מסלול בין קודקוד נתון לקודקוד נתון אחר, וכן - כמה רכיבי קשירות יש בגרף (ראה להלן במימושים).
פרו' לויט הסביר כי חיפוש שלם לא שייך כאן כי זו תהיה סיבוכיות מאוד גדולה, וחמדני לא שייך - כי הוא בוודאי יביא לטעות, אנחנו ננסה למצוא פתרון בעזרת תכנות דינאמי:
בעצם נרצה לבדוק - בין 2 קודקודים - האם קיים קשר ישיר או קשר עקיף כלשהו, ובמילים אחרות:

$$H[i,j] = [H[i,k] \wedge H[k,j]] \vee H[i,j]$$

נביא דוגמא מהגרף דלעיל:

$$v_1 \to v_4 = \begin{bmatrix} True \\ v_1 \to v_2 \end{bmatrix} \wedge \begin{bmatrix} True \\ v_2 \to v_4 \end{bmatrix} \vee \begin{bmatrix} False \\ v_1 \to v_4 \end{bmatrix} = True$$

ז"א: על אף שאין קשר ישיר בין הקודקודים, מצאנו קודקוד המקיים קשר עקיף בין השניים.
חשוב לזכור כאן את חוקיות ה"חיתוך" - רק כאשר 2 הערכים בחיתוך הם True, הערך שחוזר הוא True.

טיפ חשוב: בשיעור פרו' לויט דיבר על גרפים מכוונים, אך בפועל התכנות אותן מצאנו על גרף לא מכוון נכונות גם לגרף מכוון, לכן במימוש להלן נשתמש בחומר התיאורתי בשביל להתקדם ולממש את השלב השני (שים לב שכאשר מדובר בגרף מכוון - לא נוכל לעבור רק על המשולש העליון).

במימוש שיובא לקמן אנחנו נקשר את כל הקדקודים הניתנים לקישור, וגם נמצא את כל המסלולים האפשריים בין קודקוד אחד לכל קודקוד אחר, שים לב - האלכסון הראשי יהפוך כעת כולו ל: **True**.
בשביל למצוא את כל המסלולים נעשה שלב מקדים - נרוץ על המטריצה משיעור שעבר $[O(n^2)]$, וכל מקום בו יש כבר קשר ישיר - נכניס אותו כמסלול. לאחר מכן נטפל בכל המסלולים הלא ישירים, וזאת על ידי ריצה נוספת $[O(n^3)]$, בשביל לבדוק האם קיים קודקוד כלשהו שיכול ליצור קשר בין 2 הקודקודים שאינם מחוברים בקשר ישיר.

נרחיב מעט על השלב המקדים, בתוך הלולאה הכפולה נחשב:

```
a0 = i / (m+1);
b0 = i % (m+1);
a1 = j / (m+1);
b1 = j % (m+1);
if (mat[i][j]==true) path[i][j] = "["+a0+","+b0+"] -> ["+a1+","+b1+"]";
```

ז"א: אם אכן מדובר בקודקוד שערכו אמת - ז"א: קיים קשר ישיר - תכניס אותו למטריצת המחרוזות. נביא דוגמא בשביל שהדברים יהיו ברורים יותר:

נאמר והגרף שלי הוא: $B_{n=1,m=2}$, אני יודע שמקודקוד $[0,0]$ לקודקוד $[0,2]$ יש קשר ישיר, ולכן הערך בטבלה הוא: **true**, המיקום בטבלה הוא: $[i=0,j=2]$. נציב את הערכים במשוואות הנ"ל:

```
a0 = 0 / (2+1); -> 0
b0 = 0 % (2+1); -> 0
a1 = 2 / (2+1); -> 0
b1 = 2 % (2+1); -> 2
```

מכאן נקבל במטריצת המסלולים, במקום $[i=0,j=2]$ את המחרוזת: `[0,0] -> [0,2]`

בעצם השלב הזה הוא דומה (רק הפוך) מהשלב בו חיפשנו את האינדקס במטריצה של בקבוק מסוים, כאן אנחנו מחפשים את הבקבוק לפי הערכים במטריצה.

את המימוש נחלק לשניים:

א. מימוש שלב שני: מציאת כל הקשרים שאינם ישירים בין זוג קודקודים ומציאת כל המסלולים הקיימים: (שים לב - כל הפונקציות שלמדנו בשיעור שעבר - לא הבאתי במימוש כאן, משום שהן מומשו לעיל, רק חשוב לזכור ולהמיר את המטריצה ממספרים שלמים למטריצה בוליאנית).

```
public class shiur02_ConnectPossVertex {
	/**
	 * הפונקציה הופכת כל מיקום במטריצה לאמת אם קיים מסלול כשלהו בין קודקוד לקודקוד
	 * ושומרת את כל המסלולים במטריצת המחרוזות
	 * @param mat the Boolean Matrix we get in Shiur 1
	 * @param n the size of one Bottle
	 * @return String Matrix with all tracks
	 */
	public static String[][] ConnectPossibleVertex(boolean mat[][],int m) {
		int dim = mat.length;
		int a0=0, b0=0, a1=0, b1=0;
		String path[][] = new String[dim][dim];
		for (int i=0; i<dim; i++) {
			for (int j=0; j<dim; j++) {
				a0 = i / (m+1);
				b0 = i % (m+1);
				a1 = j / (m+1);
				b1 = j % (m+1);
				if (mat[i][j]==true) path[i][j] =
								"["+a0+","+b0+"]-D->["+a1+","+b1+"]";
				else path[i][j] = new String();
			}
		}
		for (int k=0; k<dim; k++) {
			for (int i=0; i<dim; i++) {
				for (int j=0; j<dim; j++) {
					if (mat[i][j]==false) {
						mat[i][j] = mat[i][k] && mat[k][j];
						if (mat[i][j] == true) {
							// אם באמת נמצא קשר בין הקודקודים
							path[i][j] =
								path[i][k] + " >-i-> " + path[k][j];
						}
					}
				}
			}
		}
		return path;
	}
}
```

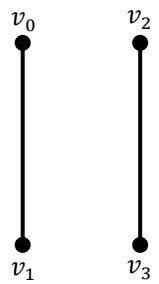
ב. מימושים הקשורים לגרפים: (רוב המימושים נלקחו מהתרגול של שיעור מס' 3)
משום שבשיעור מס' 2 נעזרנו בכל הנושא של גרפים בשביל להתקדם ולהגיע למימוש של שלב ב' בבעיית הבקבוקים - נביא כאן כמה אלגוריתמים הקשורים בגרף: לבדוק האם גרף הוא מכוון או לא מכוון, קשיר או לא קשיר, ואם אינו קשיר - כמה רכיבי קשירות יש בגרף ומהם הרכיבים.
חשוב לשים לב שחלק מהמימושים שיובאו לקמן הם בעצם מימושים שכבר עסקנו בהם עם שינויים קלים.
ראה להלן (לאחר המימושים) קלט פלט של הפונקציה לחישוב רכיבי קשירות.

```java
public class shiur03_Component_Ties_graph {
	/**
	 * פונקציה בוליאנית הבודקת האם גרף הוא מכוון ע"י השוואה בין משולש עליון לתחתון במטריצה
	 * @param mat the Boolean Matrix we get in Shiur 1
	 * @return true if the Graph is not Directed, else return false
	 */
	public static boolean isDirectedGraph(boolean mat[][]) {
		int n = mat.length;
		int m = mat[0].length;
		for (int i=0; i<n; i++) {
			for (int j=i+1; j<m; j++) {
				if (mat[i][j]!=mat[j][i]) return false;
			}
		}
		return true;
	}

	/**
	 * בתוכנית זו נבדוק האם הגרף הוא קשיר וראה להלן תוכנית הבודקת כמה רכיבי קשירות יש לגרף
	 * כאן נשתמש במה שמימשנו בשיעור שעבר (מציאת כל המסלולים) עם שינוי קטן, ז"א:
	 * אם נמצא קודקוד שנותן ערך "שקר" - זה אומר שהגרף אינו קשיר
	 * @param mat the Boolean Matrix we get in Shiur 1
	 * @return if the graph is tiable - return true, else - return false
	 */
	public static boolean isTiableGraph(boolean mat[][]) {
		// הועתק משיעור קודם, רק שאנחנו לא מחפשם עכשיו את המסלולים
		// ConnectPossibleVertex(boolean mat[][],int m)
		int dim = mat.length;
		for (int k=0; k<dim; k++) {
			for (int i=0; i<dim; i++) {
				for (int j=0; j<dim; j++) {
					if (mat[i][j]==false) {
						mat[i][j] = mat[i][k] && mat[k][j];
					}
				}
			}
		}
		// אם אכן מדובר בגרף קשיר - כל הערכים במטריצה צריכים לקבל ערך "אמת"
		for (int i=0; i<dim; i++) {
			for (int j=0; j<dim; j++) {
				if (mat[i][j]==false) return false;
			}
		}
		return true;
	}
```

```java
/**
 * את הפונקציה הזו קיבלנו מוכנה, החלק ששומר מהם רכיבי הקשירות דורש בירור מעמיק יותר
 * @param matAfterFW boolean mat - after isDirectedGraph() function from shiur2
 * @return How many components in the graph
 */
public static int HowManyComponents(boolean matAfterFW[][]) {
        int count = 0;
        int dim = matAfterFW.length;
        int comps[] = new int[dim];
        for (int i=0; i<dim; i++) {
                if (comps[i]==0) count++;
                for (int j=i; j<dim; j++) {
                        if (comps[j]==0 && matAfterFW[i][j]) comps[j] = count;
                }
        }
        String cs[] = new String[count];
        for (int i=0; i<count; i++) {
                cs[i] = "";
        }

        for (int i=0; i<dim; i++) cs[comps[i]-1] = cs[comps[i]-1]+ i +"\t";
        for (int i=0; i<count; i++) System.out.println(cs[i].toString());
        return count;
}
}
```

קלט פלט בפונקציה HowManyComponents: כפי שאמרנו הפונקציה מקבלת גרף לאחר שעבר קישור בין כל הקודקודים העקיפים, למשל: מדובר בגרף של 4 קודקודים:

```
*********** output after isTiableGraph() ***********
        V_0         V_1         V_2         V_3
V_0     false       true        false       false
V_1     true        false       false       false
V_2     false       false       false       true
V_3     false       false       true        true
****************************************************

********* output after HowManyComponents() *********
        V_0         V_1         V_2         V_3
V_0     true        true        false       false
V_1     true        true        false       false
V_2     false       false       true        true
V_3     false       false       true        true
****************************************************
```



אפשר לזהות מיד שבגרף המדובר יש שני רכיבי קשירות: נוצרו פה בעצם 2 צורות שונות, ז"א: מספיק שערך אחד יהיה false בשביל לגרום לגרף להיות לא קשיר.

כל זה מבחינת הגרף עצמו, הפלט של רכיבי הקשירות יראה כך:

```
is the Graph is Tiable? false
the Components is:
0       1
2       3
the number of Components is: 2
```

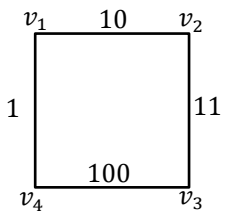
הערה: התרגולים משיעור זה לא היו מסונכרנים עם ההרצאה עצמה, ולכן העברתי את המימושים שיתאמו למה שדובר בשיעור (אפשר לראות זאת בשיעור שעבר - לשם הועברו כל המימושים משיעור זה).

נעבור כעת ל**שלב השלישי** בבעיית הבקבוקים, אך לפני זה חזרה קצרה:
בשלב הראשון מצאנו רק את הקשרים הישירים,
בשלב שני מצאנו גם את הקשרים העקיפים ושמרנו במטריצת מחרוזות את כל המסלולים.
כעת נרצה לחשב את המסלולים הזולים ביותר בין כל קודקוד לקודקוד, כאשר מחיר כל שלב ידוע. חשוב לשים לב - כאשר ניישם את השלב הזה בגרף הבקבוקים - המטריצה תהפוך למחיר הצלעות שבין הקודקודים, שבשלב הראשוני מחיר כל קודקוד וקודקוד יהיה אינסופי. לאחר שלב ראשון דלעיל - חלק מהצלעות יקבלו ערך 1 (קשר ישיר), ולאחר שלב שתיים - אותן צלעות שערכם נשאר אינסוף - יקבלו ערך מספרי גבוה יותר - והוא יהיה תלוי במספר הצלעות המינימאלי אותו נצטרך לעבור. בכל מקום שלא ימצא קשר בין קודקוד לקודקוד - ערך הצלע יישאר אינסוף.

ז"א: במימוש הזה אנחנו נחזור למטריצת מספרים, כאשר האלכסון הראשי יהיה כולו אפס (והפעם זה גם קריטי), ונשתמש במה שלמדנו בשיעורים הקודמים עם מעט תוספות.

אך כמו בשיעור הקודם גם בשיעור הזה - פרו' לויט התחיל בתאוריה הקשורה לגרפים ורק אח"כ קישר את הדברים לבעיית הבקבוקים.

נאמר ונתון לנו גרף עם קודקודים ומחיר מעבר דרך כל צלע, למשל הגרף להלן:
נבנה מטריצה מספרית, ז"א: במקום לרשום true, נרשום את הערך של הצלע, בשלב זה - כל קודקוד שלא מחובר עם חברו בקשר ישיר יישאר עם ערך אינסוף.



	v_1	v_2	v_3	v_4
v_1	0	10	∞	1
v_2	10	0	11	∞
v_3	∞	11	0	100
v_4	1	∞	100	0

מעבר לכך שיש לנו בטבלה ערכים של אינסוף שבהם יש לטפל, ניתן לראות שהמעבר בין קודקוד 3 ל-4 עומד כרגע על 100, כאשר בפועל ניתן לעשות זאת דרך קודקודים 1 ו-2 ואז המחיר יהיה רק 22 במקום 100.

כעת נפעיל על המטריצה מעין מה שעשינו בשלב ב' של הבקבוקים, ז"א:
מעבר לכך שנבדוק האם המסלול קיים (true - false), אנחנו נחפש את המחיר הזול ביותר. ולכן, המטריצה דלעיל תראה לאחר השינוי.

	v_1	v_2	v_3	v_4
v_1	0	10	21	1
v_2	10	0	11	11
v_3	21	11	0	22
v_4	1	11	22	0

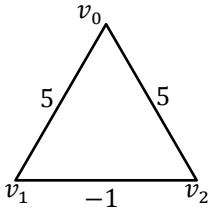
בשביל למצוא את המחיר הזול ביותר נרוץ בתוך 3 לולאות ונאמר כך:

$$dis[i,j] = \min\left[dis[i,k] + dis[k,j]\,, dis[i,j]\right]$$

ז"א: כל קודקוד k מנסה לשפר את המרחק בין i ל-j, כמובן שאם לא היה קיים שום מסלול בין הקודקודים ו-k הוביל למסלול - הוא יהיה הערך החדש, כמובן שייתכן ובהמשך נמצא k אחר שישפר עוד את המרחק.

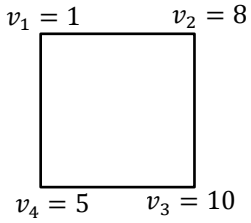
אפשר לראות שהנוסחא הזו מאוד דומה למה שדיברנו בשיעור מס' 2: $H[i,j] = \left[H[i,k] \wedge H[k,j]\right] \vee H[i,j]$
ז"א: הסימן $\wedge$ מתחלף ב:+, והסימן $\vee$ מתחלף ב:min.

<u>הערה חשובה</u>: אלגוריתם זה אינו יעיל כאשר יש בגרף צלעות שליליות, שהרי ניתן לרוץ על אותה צלע אינסוף פעמים - והמסלול הזול ביותר יהיה מינוס אינסוף. לכן - ערך "אינסוף" ניתן להגדיר כמינוס אחד. אך מצד שני עדיף להגדיר אותו כ: Integer.*MAX_VALUE*, משום שכאשר נחפש מינימום - יהיה קל יותר להגדיר שכל מספר בעצם מהמספר המקסימאלי.



במהלך ההרצאה הרחיב פרו' לויט על העניין של מספרים שלילים בגרף, המסקנה הייתה:
אם מדובר בגרף מכוון - יש לבדוק את האלכסון הראשי אחרי FW - האם אין בו מס' שלילי,
אם מדובר בגרף לא מכוון - יש לבדוק האם קיים איבר אחד שלילי (ללא FW).

פרו' לויט נתן משימה (ייתכן שדבר דומה יופיע במבחן), בגלל שאנחנו עוסקים בגרף לא מכוון - נאמר ומצאתי מספר שלילי באלכסון מה שאומר שקיים לפחות מעגל שלילי אחד - מצא אותו.
(ניסיתי לממש ולא הצלחתי)



לקראת סוף השיעור פרו' לויט עבר לדבר על גרפים שגובה התשלום אינו בצלעות אלא בקודקודים, והשאלה: מה המסלול הקצר ביותר בין 2 קודקודים נתונים, למשל: מ-1 ל-2, המסלול הקצר ביותר שווה ל-9. כאן הקלט יכול להיות מערך חד מימדי, כשכל אינדקס מסמל קודקוד, אך חשוב לשים לב שיש צורך בנתון נוסף: מטריצה של 0 ו-1 המציינת איזה קודקוד מחובר באופן ישיר עם חברו.

	v_1	v_2	v_3	v_4
v_1	1	9	16	6
v_2	9	8	18	14
v_3	16	18	10	15
v_4	6	14	15	5

הפלט יהיה מטריצה, שהאלכסון הראשי שלה הוא התשלום עבור עמידה בקודקוד בודד. גם כאן אפשר להסתפק בחישוב של משולש עליון (+אלכסון ראשי).

כעת, אם נעביר בחזרה את הסכומים לצלעות - נגלה שבחלק מהריבועים במטריצה אנחנו משלמים הרבה יותר, משום שקודקוד נספר פעמיים (כמובן שהאלכסון הראשי יחזור להיות אפס).

חשוב לשים לב שבמימוש אנחנו משתמשים במה שלמדנו בשיעורים הקודמים עם מעט שינוי, בכל מקום שבוצע שינוי - ישנה הערה.

```
public class shiur04_WeightsOfBottlesIs1 {

	public static final int max = Integer.MAX_VALUE;

	/**
	 * take from Shiur 1
	 * @param i Value of first bottle
	 * @param j Value of second bottle
	 * @param n The size of the matrix
	 * @return the position in the matrix corresponds to the value of two bottles
	 */
	public static int getIndex(int i, int j, int n) {
		return (n+1)*i+j;
	}

	/**
	 * this function take from Shiur 1 + small changes
	 * @param n The maximum value of the first bottle
	 * @param m The maximum value of the second bottle
	 * @return Integer Matrix: MAX_VALUE- no connection, number- the edges for connection
	 */
	public static int[][] initRibs(int n, int m) {
		System.out.println("The maximum value of the first bottle(n) = "+n);
		System.out.println("The maximum value of the second bottle(m) = "+m);
		System.out.println("*******************["+n+","+m+"]*******************");
		System.out.println("*********************************************\n");

		int dim = (n+1)*(m+1);
		int mat[][] = new int[dim][dim];
		// כאן, בשונה ממה שעשינו עד עכשיו, אנחנו מאחלים את המטריצה כולה באינסוף
		// לאחר מכן נמצא מאיזה קודקוד לאיזה קודקוד קיים מסלול
		for (int i=0; i<dim; i++) {
			for (int j=0; j<dim; j++) {
				mat[i][j] = max;
			}
		}

		int ind1, ind2;
		for (int i=0; i<=n; i++) {
			for (int j=0; j<=m; j++) {
				ind1 = getIndex(i,j,m);
				mat[ind1][getIndex(0,j,m)] = 1; // Rib 1
				mat[ind1][getIndex(n,j,m)] = 1; // Rib 2
				mat[ind1][getIndex(i,0,m)] = 1; // Rib 3
				mat[ind1][getIndex(i,m,m)] = 1; // Rib 4
				ind2 = getIndex(Math.min(i+j,n), i+j - Math.min(i+j,n),m);
				mat[ind1][ind2] = 1; // Rib 5
				ind2 = getIndex(i+j - Math.min(i+j,m), Math.min(i+j,m), m);
				mat[ind1][ind2] = 1; // Rib 6
			}
		}
		// כפי שהסברנו בתאוריה - כאן זה קריטי שהערך יהיה אפס דווקא
		for (int t=0; t<dim; t++) mat[t][t] = 0;

		return mat;
	}
```

```java
	/**
	 * הפונקציה הופכת כל מיקום במטריצה למספר אם קיים מסלול כלשהו בין קודקוד לקודקוד
	 * ומחשבת את כל המסלולים הזולים ביותר במטריצה של מחרוזות
	 * @param mat the Matrix we get in Shiur 1
	 * @param n the size of one Bottle
	 * @return String Matrix with all the cheapest tracks
	 */
	public static String[][] ConnectPossibleVertex(int mat[][],int n) {
		int dim = mat.length;
		int a0=0, b0=0, a1=0, b1=0;
		String path[][] = new String[dim][dim];
		for (int i=0; i<dim; i++) {
			for (int j=0; j<dim; j++) {
				a0 = i / (n+1);
				b0 = i % (n+1);
				a1 = j / (n+1);
				b1 = j % (n+1);
				if (mat[i][j]!=max) path[i][j] =
								"["+a0+","+b0+"] -> ["+a1+","+b1+"]";
				else path[i][j] = new String();
			}
		}

		for (int k=0; k<dim; k++) {
			for (int i=0; i<dim; i++) {
				for (int j=0; j<dim; j++) {
					// כאן הכללים שונים לגמרי ממה שעשינו בשיעור מס' 2
					// משום שאנחנו עוסקים במספרים, אך הרעיון הוא אותו רעיון
					if (mat[i][k]!=max && mat[k][j]!=max) {
						if (mat[i][j] > mat[i][k] + mat[k][j]) {
							mat[i][j] = mat[i][k] + mat[k][j];
							path[i][j] = path[i][k] +">>>"+ path[k][j];
						}
					}
				}
			}
		}
		return path;
	}
}
```

החל משיעור זה, ועד לשיעור 6 (כולל) העבירה את החומר גב' אליזבט.

בעיה מס' 2 - חיפוש לרוחב בגרף - BFS

אלגוריתם זה נכון בין לגרף מכוון ובין לגרף לא מכוון.

כיצד עובד האלגוריתם? תחילה נדבר על הקלט - בשביל לייצג גרף ובו קודקודים וצלעות - נבנה מערך של רשימות מקושרות. המערך עצמו מייצג את הקודקודים, ולכל קודקוד רשימה מקושרת המייצגת את הקודקודים אליהם אותו קודקוד מחובר בצלע. דוגמא - גרף לא מכוון:

```
(0)--------(1)        (2)--------(3)
 |          |         /|         /|
 |          |        / |        / |
 |          |       /  |       /  |
 |          |      /   |      /   |
 |          |     /    |     /    |
 |          |    /     |    /     |
 |          |   /      |   /      |
 |          |  /       |  /       |
 |          | /        | /        |
(4)        (5)--------(6)--------(7)
```

```
vertex 0 -> [1, 4]
vertex 1 -> [0, 5]
vertex 2 -> [3, 5, 6]
vertex 3 -> [2, 6, 7]
vertex 4 -> [0]
vertex 5 -> [1, 2, 6]
vertex 6 -> [5, 2, 3, 7]
vertex 7 -> [3, 6]
```

אנחנו מעוניינים למצוא ע"י חיפוש לרוחב - את המסלול הקצר ביותר בין 2 קודקודים. ז"א: אנחנו נכניס לאלגוריתם את הקודקוד בו נרצה להתחיל - והוא יחשב לנו את המסלול הקצר ביותר בינו לבין כל קודקוד אחר. למשל: קודקוד המקור הוא 1, המרחק בינו לבין קודקוד 4 הוא 2, ובינו לבין קודקוד 3 הוא 3.

בשביל לדעת באיזה קודקוד כבר טיפלנו, באיזה אנחנו עדיין בטיפול ובאיזה כבר סיימנו לטפל - נבנה מעטפת לתוכנית שתכלול משתני עצם:

א. מערך "צבע הקודקוד" - צבע לבן (המספר 0) מציין שהקודקוד לא התחיל בטיפול
- צבע אפור (המספר 1) מציין שהקודקוד התחיל בטיפול אך לא סיים
- צבע שחור (המספר 2) מציין שהקודקוד סיים את הטיפול.

טיפול - הכוונה בדקנו את המרחקים ממנו אל כל שכניו.
כמובן שמערך זה צריך להיות מאותחל בצבע לבן (0), אך בג'אווה הדבר קורה אוטומטי, ולכן אין צורך לאתחל.

ב. מערך "מרחקים" - תחילה נאתחל את המערך עם מספר שלילי (-1) משום שכל עוד לא מצאנו את המרחק- מבחינתנו הוא אינו קיים (המרחק בין המקור לכל שאר הקודקודים). רק המקור יקבל מרחק השווה לאפס.

ג. מערך "אבות" - מערך הזוכר מיהו אביו של כל קודקוד, כמובן שנאתחל אותו גם כן במספר שלילי, משום שכל עוד לא מצאנו אב - מבחינתנו הוא אינו קיים, מלבד המקור - שהוא באמת ללא אב.

נביא דוגמא להמחשת האלגוריתם: נשתמש בדוגמא לעיל - המקור הוא (1):

```
Vertex 0      Dist=1      Pred=1     The color: Black
Vertex 1      Dist=0      Pred=-1    The color: Black
Vertex 2      Dist=2      Pred=5     The color: Black
Vertex 3      Dist=3      Pred=2     The color: Black
Vertex 4      Dist=2      Pred=0     The color: Black
Vertex 5      Dist=1      Pred=1     The color: Black
Vertex 6      Dist=2      Pred=5     The color: Black
Vertex 7      Dist=3      Pred=6     The color: Black
```

אם כן - איך מתחילים? שלב ראשון: לטפל במקור: נגדיר את המרחק כ-0, ואת הצבע שלו כאפור. כעת נכניס אותו לתוך תור (בהמשך נבין לשם מה). כעת נרוץ בלולאה - כל עוד התור לא ריק - תמשיך. התור יתרוקן לגמרי רק כאשר נסיים לעבור על כל קודקודי הגרף על כל שכניהם.

בתוך הלולאה נמשוך את האיבר הראשון למשתנה זמני (v). כידוע - כאשר עוסקים בתור - הראשון שיוצא הוא הראשון שנכנס (בפעם הראשונה זה לא משנה, בנוסף: בפעם הראשונה - אנחנו בתוך הלולאה והתור ריק).

הערך של v הוא בעצם ערך של קודקוד, ולכן אנחנו נלך למערך הרשימות המקושרות ונתחיל לטפל בכל השכנים של v, מיהם השכנים? `list[v]`.

לפני שנבין איך מטפלים בשכנים - נמחיש את מה שעשינו עד עכשיו בעזרת הדוגמא דלעיל: אנחנו התחלנו בקודקוד מס' 1 - הגדרנו את המרחק שלו כאפס, ואת הצבע שלו כאפור. הכנסנו אותו לתוך התור.

בתוך התור משכנו אותו לתוך משתנה זמני v, כעת אנחנו עוברים לטפל בשכנים, ז"א: `list[1]= [0, 5]`

כיצד נטפל בשכנים? תחילה נכנס ללולאת **while** נוספת - שתרוץ עד שהרשימה הנוכחית אינה ריקה, בתוך הלולאה נתחיל לשלוף איברים מתוך הרשימה למשתנה זמני (u). כעת - רק אם הצבע של אותו קודקוד שנשלף לבן - נמשיך לטפל בו:

תחילה נצבע אותו באפור, נגדיר לו את המרחק - כמרחק של אביו פלוס אחד, ונאמר שאביו הוא קודקוד v. לאחר מכן נכניס את קודקוד u להמשך וזאת ע"י הכנסתו לתוך התור.

נמשיך עם הדוגמא, נראה כיצד 0 עובר את הטיפול:

```
while (!list[1].isEmpty()) {
     u = list[1].removeFirst(); -> 0
     if (color[0]==white) {
          color[0] = gray;
          dist[0] = dist[1] + 1; -> 1
          pred[u] = v; -> 0
          Que.add(1);
     }
}
```

ז"א: לאחר ריצה על כל הרשימה המקושרת של קודקוד מס' 1 נקבל (את הקודקודים שלא השתנו כלל - לא רשמתי):

```
Vertex 0  Dist=1        Pred=1        The color: Gray
Vertex 1  Dist=0        Pred=-1       The color: Black
Vertex 5  Dist=1        Pred=1        The color: Gray
```

בשביל להבין מעט יותר טוב נמשיך עוד קצת בדוגמא: הטיפול הבא יעבור לקודקוד מס' 0, התוצאה תהיה:

```
Vertex 0  Dist=1        Pred=1        The color: Black
Vertex 1  Dist=0        Pred=-1       The color: Black
Vertex 4  Dist=2        Pred=0        The color: Gray
Vertex 5  Dist=1        Pred=1        The color: Gray
```

משום שלאפס יש רק קודקוד נוסף שעדיין לא טופל - קודקוד מס' 4, לאחר הטיפול בקודקוד זה נהפך גם קודקוד אפס לשחור.

התהליך יחזור עד לשליפת כל הקודקודים מתוך התור וצביעתם בשחור, כפי שהובא לעיל.

מספר תכונות שחשוב לזכור:

- הסיבוכיות של האלגוריתם: $O(n+m)$, כאשר n מספר הקודקודים, ו-m מספר הצלעות. עוד יש לזכור כי מספר הצלעות יכול לנוע באופן הבא: $n-1 \le m \le \frac{n(n-1)}{2}$.
- שימוש ב: **BFS**- זה בעצם בניית עץ המייצג את הגרף (הדבר מועיל בחומר שנלמד לקראת סוף הסמסטר).
- בתוך התור יש אך ורק קודקודים אפורים, ז"א: ישנם קודקודים שעדיין יש להם שכנים לבנים, ברגע שיתברר שאין יותר שכנים לבנים - התור יתרוקן.
- המסלול שמתקבל בעזרת **BFS**- הוא המסלול הקצר ביותר.
- ניתן לבדוק האם גרף קשיר, ואם לא - כמה רכיבי קשירות יש לו בעזרת **BFS** (מומש לקמן, חלק ב').

מימוש חלק א': בחלק זה נממש את עיקר הקוד של **BFS**, בהמשך נממש פונקציות נוספות, וראה לקמן - הסבר ספציפי לגבי אותן פונקציות.

```
import java.util.*;
import java.util.concurrent.ArrayBlockingQueue;

public class shiur05_BFS {

	public static final int white = 0;
	public static final int gray = 1;
	public static final int black = 2;
	public static final int NIL = -1;

	int color[];
	int dist[];
	int pred[];
	LinkedList<Integer>[] list;
	int size;
	int start;

	/**
	 * Constructor
	 * @param numOfV the number of Vertex
	 * @param list the LinkedList of all edges
	 */
	public shiur05_BFS(int numOfV, LinkedList <Integer>list[]) {
		size = numOfV;
		color = new int [size];
		this.start = 0; // ברירת מחדל עד להפעלת הפונקציה של המחלקה
		this.list = list;
		dist = new int[size];
		pred = new int[size];
		PrintGraph_0to7(); // פונקציה סטטית המדפיסה את צורת הגרף, מובאת לקמן
	}

	/**
	 * Function finds all the tracks from st Vertex
	 * @param st the Start Vertex
	 */
	public void B_F_S(int st) {
		// איתחול מערכים, שים לב אין צורך לאתחל את מערך הצבעים, משום שהוא מאותחל כבר כלבן
		// נקודה נוספת: המערכים מאותחלים כאן ולא בבנאי לטובת הפעלה חוזרת של הפונקציה, וראה חלק ב'
		for (int i=0; i<size; i++) {
			dist[i] = NIL;
			pred[i] = NIL;
		}
		start = st;
		int v,u; // משתני עזר לחישוב השכנים
		// הכנסת ערכים לנקודת ההתחלה של הגרף
		// שים לב, אין צורך לעדכן שלנקודת ההתחלה אין "אבא", משום שכך המערך מאותחל
		dist[start] = 0;
		color[start] = gray;
		// הגדרת תור למימוש החיפוש והכנסת נקודת ההתחלה לתוך התור
		Queue <Integer> Que = new ArrayBlockingQueue<Integer>(100);
		Que.add(start);
```

```
            while (!Que.isEmpty()) {
                  v = Que.poll(); // לוקחים את האיבר שבראש התור ומטפלים לו בכל השכנים
                  while (!list[v].isEmpty()) {
                        u = list[v].removeFirst();
                        if (color[u]==white) { // אם השכן עדיין לבן
                              color[u] = gray; // השכן עובר להיות אפור
                              dist[u] = dist[v] + 1; // המרחק של הבן הוא המרחק של האב פלוס 1
                              pred[u] = v;
                              Que.add(u);
                        }
                  }
                  color[v] = black;
                  System.out.println("the Que now is: "+Que);
                  printArrays(dist,pred,color); // פונקציה סטטית להדפסת הנתונים
            }
      }

      // צורת הגרף מובאת להלן בפונקציית הדפסה סטטית
      /**
       *
       * @return a graph as we learned in class
       */
      public static LinkedList<Integer>[] InPut_0to7() {
            int size = 8;
            LinkedList <Integer>list[] = new LinkedList[size];
            for (int i=0; i<size; i++) {
                  list[i] = new LinkedList<Integer>();
            }
            list[0].add(1); list[0].add(4);
            list[1].add(0); list[1].add(5);
            list[2].add(3); list[2].add(5); list[2].add(6);
            list[3].add(2); list[3].add(6); list[3].add(7);
            list[4].add(0);
            list[5].add(1); list[5].add(2); list[5].add(6);
            list[6].add(5); list[6].add(2); list[6].add(3); list[6].add(7);
            list[7].add(3); list[7].add(6);
            return list;
      }

      /**
       * Helped function to print array data
       */
      public void PrintArrayLL() {
            System.out.println("************
                              the Array LinkedList of Graph is: ************");
            for (int i=0; i<size; i++) {
                  System.out.println("vertex "+i+" -> "+list[i].toString());
            }

            System.out.println("*****************************************************\n");
      }
```

```
	/**
	 * Helped function vertex data printing
	 * @param dist the Distance from the start
	 * @param pred the "father" of the vertex
	 * @param color the color of the vertex
	 */
	public static void printArrays(int dist[], int pred[], int color[]) {
		for (int i=0; i<dist.length; i++) {
			String temp = "";
			if (color[i]==0) temp = "White";
			if (color[i]==1) temp = "Gray";
			if (color[i]==2) temp = "Black";
			System.out.println("Vertex "+i+"\tDist="+dist[i]+
					"\t\tPred="+pred[i]+"\t\tThe color: "+temp);
		}
		System.out.println("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-");
	}

	/**
	 * Drawing the graph from the lesson
	 */
	public static void PrintGraph_0to7() {
		System.out.println("=========== Graph 0 to 7 ===========\n");
		System.out.println("(0)--------(1)        (2)--------(3)");
		System.out.println(" |          |         /|         /| ");
		System.out.println(" |          |        / |        / | ");
		System.out.println(" |          |       /  |       /  | ");
		System.out.println(" |          |      /   |      /   | ");
		System.out.println(" |          |     /    |     /    | ");
		System.out.println(" |          |    /     |    /     | ");
		System.out.println(" |          |   /      |   /      | ");
		System.out.println(" |          |  /       |  /       | ");
		System.out.println(" |          | /        | /        | ");
		System.out.println("(4)        (5)--------(6)--------(7)");
		System.out.println("\n===================================\n");
	}
}
```

נעבור אל החלק השני, נוכל לבדוק האם הגרף קשיר, ואם לא - כמה רכיבי קשירות יש לו. בנוסף: נוכל לדעת אם קיים מסלול בין קודקוד המקור לקודקוד אחר, ואם כן - מהו המסלול. בנוסף: נוכל לבדוק האם גרף הינו גרף מפוצל.

חשוב לשים לב שחלק זה מסתמך על כל מה שעשינו עד עכשיו (פונקציית **BFS** הועתקה שוב - אלא שים לב לשינויים וכדלקמן - סומן בצהוב).

נאמר ויש לנו גרף שאינו קשיר, למשל גרף עם שלושה רכיבי קשירות:

```
(0)--------(1)           (4)
 |          |            /|
 |          |           / |
 |          |          /  |
 |          |         /   |
 |          |        /    |
 |          |       /     |     (7)
 |          |      /      |
 |          |     /       |
 |          |    /        |
 |          |   /         |
 |          |  /          |
(3)--------(2)  (5)-------(6)
```

בעצם - בכל פעם שנפעיל **BFS** נוכל לקבל עוד רכיב קשירות. בשביל שנדע בכל פעם מהיכן להפעיל את הפונקציה מחדש נשתמש במערך עזר. תחילה - המערך כולו יהיה מאותחל במספר שלילי. כעת נעבור על מערך המרחקים - כל עוד לא קיבלנו מספר שלילי - נכניס אותו למערך העזר, ונמשיך לספור את האיברים באותו רכיב קשירות. בפעם הראשונה שנגיע למספר שלילי - נדע שעברנו לרכיב קשירות נוסף - נספור אותו כרכיב נוסף, ונפעיל מאותו קודקוד **BFS** פעם נוספת, וחוזר חלילה.

דבר נוסף שיש לשים לב אליו - הדפסת מסלול בין 2 קודקודים. אם מדובר בגרף קשיר - אין שום בעיה, פשוט הולכים בכל פעם לאב של אותו קודקוד נתון עד שמגיעים ל"מקור". אך כאשר אנחנו עוסקים בגרף לא קשיר - צריך להיזהר, משום שייתכן שאין מסלול המחבר את 2 הקודקודים, ואם הגענו לקודקוד שאביו הוא מינוס אחד - לא בהכרח שהצלחנו למצוא את המסלול המבוקש. לכן יש לבדוק - האם הקודקוד אליו אנחנו רוצים להגיע - המרחק שלו בכלל קיים מקודקוד המקור.
עוד יש לשים לב, שאם הפעלנו את הפונקציה למציאת רכיבי קשירות - בעצם דרסנו את הנתונים ולא נוכל לחפש מסלול.

```
(1)--------(2)
 |          |
 |          |
 |          |
(2)--------(1)
```

נקודה נוספת שיש לשים לב אליה במימוש הנוכחי - בדיקה האם גרף מפוצל או לא. הגדרה: גרף מפוצל הינו גרף שניתן לחלק את קודוקדיו ל-2 קבוצות, כאשר מעבר בין קודקודי הקבוצה יחייב לעבור דרך קודקוד של הקבוצה השניה, למשל:
ניתן להשתמש בפונקציית **BFS**, אלא שנצטרך להוסיף מערך נוסף - מערך המייצג את 2 הקבוצות בגרף, כאשר קודקוד המקור יקבל ערך 1, והבאים אחריו יוגדרו ע"י הנוחסא: `part[u] = 3-part[v];`. בכל פעם שנכנס שנוציא איבר מהרשימה המקושרת - נבדוק שה"קבוצה" שלו לא זהה לקבוצה של קודמו. אם כן - הגרף אינו מפוצל.

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.ArrayBlockingQueue;

public class shiur06_BFS_PartTwo {

	public static final int white = 0;
	public static final int gray = 1;
	public static final int black = 2;
	public static final int NIL = -1;

	int color[];
	int dist[];
	int pred[];
	int part[];
	LinkedList<Integer>[] list;
	int size;
	int start;


	public shiur06_BFS_PartTwo (int numOfV, LinkedList <Integer>list[]) {
		size = numOfV;
		color = new int [size];
		this.start = 0;
		this.list = list;
		dist = new int[size];
		pred = new int[size];
		part = new int[size];
		PrintGraph_0to7_NotTie();
	}

	public boolean B_F_S(int st) {
		for (int i=0; i<size; i++) {
			dist[i] = NIL;
			pred[i] = NIL;
		}
		int v,u;
		boolean SpiltGraph = true;
		start = st;
		dist[start] = 0;
		color[start] = gray;
		part[start] = 1;
		Queue <Integer> Que = new ArrayBlockingQueue<Integer>(100);
		Que.add(start);
		while (!Que.isEmpty()) {
			v = Que.poll();
			while (!list[v].isEmpty()) {
				u = list[v].removeFirst();
				if (part[u]==part[v]) SpiltGraph = false;
				if (color[u]==white) {
					color[u] = gray;
					dist[u] = dist[v] + 1;
					pred[u] = v;
					part[u] = 3-part[v];
					Que.add(u);
				}
			}
			color[v] = black;
		}
		printArrays(dist,pred,color);
		return SpiltGraph;
	}
```

```
/** The function checks how many Ties elements inside the graph and prints them */
public void compoents() {
	boolean flag = false;
	String comp = new String();
	int t[] = new int[size]; // מערך עזר
	for (int i=0; i<size; i++) {
		t[i] = NIL; // מילוי המערך בערך מינוס אחד
	}
	int count = 1;
	int index;
	while (!flag) {
		PrintArrayLL();
		comp = new String();
		for (int i=0; i<size; i++) {
			if (dist[i]!=NIL) {
				comp = comp + i + "->";
				t[i] = dist[i];
			}
		}
		System.out.println("\nThe count is: " + count);
		System.out.println(comp);
		flag = true;
		int nextIndex = 0;
		for (index = 0; flag && index<size; index++) {
			if (t[index]==NIL) { // אם התנאי מתקיים - מצאנו רכיב נוסף
				flag = false;
				count++;
				nextIndex = index; // הפעלת בי-אפ-אס לפי ההתחלה של הרכיב החדש
			}
		}
		B_F_S(nextIndex); // הפעלה חוזרת של הפונקציה עם ערך התחלתי חדש
	}
}

/**
* If there is a track between the vertex - function will prints it
* @param end Vertex which you want to reach
*/
public void PrintTrack(int end) {
	String ans = "" + end;
	int temp = pred[end];
	while (temp!=-1) {
		ans = temp +"->"+ ans;
		temp = pred[temp];
	}
	// תנאי הבודק האם הקודקוד המבוקש נמצא על אותו רכיב קשירות
	if (dist[end]==-1) System.out.println("there is no Track");
	else System.out.println("the track from "+start+" to "+end+" is: " + ans);
}
```

```
	// צורת הגרף מובאת להלן בפונקציית הדפסה סטטית
	/**
	 * @return a graph as we learned in class
	 */
	public static LinkedList<Integer>[] InPut_0to7_NotTie() {
		int size = 8;
		LinkedList <Integer>list[] = new LinkedList[size];
		for (int i=0; i<size; i++) list[i] = new LinkedList<Integer>();
		list[0].add(1); list[0].add(3);
		list[1].add(0); list[1].add(2);
		list[2].add(1); list[2].add(3);
		list[3].add(0); list[3].add(2);
		list[4].add(5); list[4].add(6);
		list[5].add(4); list[5].add(6);
		list[6].add(4); list[6].add(5);
		list[7].add(7);
		return list;
	}
	/**
	 * Drawing the graph from the lesson
	 */
	public static void PrintGraph_0to7_NotTie() {
		System.out.println("======== Graph 0 to 7 NotTie========\n");
		System.out.println("(0)--------(1)            (4)           ");
		System.out.println(" |          |             /|           ");
		System.out.println(" |          |            / |           ");
		System.out.println(" |          |           /  |           ");
		System.out.println(" |          |          /   |           ");
		System.out.println(" |          |         /    |    (7)    ");
		System.out.println(" |          |        /     |           ");
		System.out.println(" |          |       /      |           ");
		System.out.println(" |          |      /       |           ");
		System.out.println(" |          |     /        |           ");
		System.out.println("(3)--------(2)  (5)-------(6)           ");
		System.out.println("\n==================================\n");
	}
}
```

בעיה מס' 3 - חיפוש לעומק בגרף - DFS

ישנו דמיון לצורת העבודה של **BFS**, ומיד נביא את הדומה ואת השונה:
גם אלגוריתם זה נכון בין לגרף מכוון ובין לגרף לא מכוון.

נתחיל עם הקלט - הקלט כאן זהה לקלט של **BFS**: דוגמא לקלט שעליו נעבוד:

```
(0)--------(1)        (2)
 |          |         /|        vertex 0 -> [1, 3]
 |          |        / |        vertex 1 -> [0, 4]
 |          |       /  |        vertex 2 -> [4, 5]
 |          |      /   |        vertex 3 -> [0, 4]
 |          |     /    |        vertex 4 -> [1,2,3]
 |          |    /     |        vertex 5 -> [2]
 |          |   /      |
 |          |  /       |
 |          | /        |
(3)--------(4)        (5)
```

כאן, בשונה מהאלגוריתם הקודם - אנחנו מעוניינים למצוא מסלול בין 2 קודקודים ע"י חיפוש לעומק, ז"א:
אנחנו נלך בכיוון מסוים מקודקוד המקור ועד הקודקוד האחרון ברכיב הקשירות - ואז נחזור חזרה אל קודקוד המקור. ז"א: לא נתפרס לכל הקודקודים. היתרון - ייתכן ונקבל מסלול קצר באופן מהיר יותר, משום שכאשר הגענו לקודקוד המבוקש ניתן לעצור. החסרון - ייתכן ונעשה מסלולים מאוד ארוכים, למשל: אם נתייחס לקודקוד המקור כאפס - הרי שבשביל להגיע אל 3 אנחנו נעשה 8 צעדים: 0->1->4->2->5->2->4->3 (יוסבר להלן מדוע).

גם כאן נבנה מעטפת לתוכנית בשביל לדעת באיזה קודקוד כבר טיפלנו, באיזה אנחנו עדיין בטיפול ובאיזה כבר סיימנו לטפל. המעטפת לתוכנית תכלול משתני עצם:

א. מערך "צבע הקודקוד" - צבע לבן (המספר 0) מציין שהקודקוד לא התחיל בטיפול
- צבע אפור (המספר 1) מציין שהקודקוד התחיל בטיפול אך לא סיים
- צבע שחור (המספר 2) מציין שהקודקוד סיים את הטיפול.

טיפול - הכוונה בדקנו את המרחקים ממנו אל כל שכניו.
כמובן שמערך זה צריך להיות מאותחל בצבע לבן (0), אך בג'אווה הדבר קורה אוטומטי, ולכן אין צורך לאתחל.

ב. מערכי "צעדים" (`first[],last[]`) - כל עוד לא צעדנו דרך הקודקוד - הערכים שלו הם אפס, כל צעד שאנחנו מתקדמים - אנחנו זוכרים, וכאשר מגיעים לקודקוד מסוים - שומרים בתוכו את מספר הצעדים. קודקוד המקור יקבל ערך 1, משום שהוא הצעד הראשון.

ג. מערך "אבות" - מערך הזוכר מיהו אביו של כל קודקוד, נאתחל אותו במספר שלילי, משום שכל עוד לא מצאנו אב - מבחינתנו הוא אינו קיים, מלבד המקור - שהוא באמת ללא אב. חשוב לשים לב - שהאב יוגדר רק כאשר הקודקוד עדיין לבן, ז"א: בדרך "חזור" לא נטפל באבות.

נביא דוגמא להמחשת האלגוריתם: נשתמש בדוגמא לעיל - המקור הוא (0), שלב ראשון:

```
Vertex 0   Pred=-1        First=0----Last=0   The color: White
Vertex 1   Pred=-1        First=0----Last=0   The color: White ...
```

שלב שני:

```
Vertex 0   Pred=-1        First=1----Last=0   The color: Gray
Vertex 1   Pred=0         First=0----Last=0   The color: White ...
```

שלב שלישי:

```
Vertex 0   Pred=-1        First=1----Last=0   The color: Gray
Vertex 1   Pred=0         First=2----Last=0   The color: Gray ...
```

בסוף התהליך נקבל:

```
Vertex 0  Pred=-1        First=1----Last=12  The color: Black
Vertex 1  Pred=0         First=2----Last=11  The color: Black
Vertex 2  Pred=4         First=4----Last=7   The color: Black
Vertex 3  Pred=4         First=8----Last=9   The color: Black
Vertex 4  Pred=1         First=3----Last=10  The color: Black
Vertex 5  Pred=2         First=5----Last=6   The color: Black
```

כאן, בשונה מהאלגוריתם הקודם - נשתמש בצורת עבודה של מחסנית (ז"א: האחרון שנכנס הוא הראשון שיטופל), והעבודה תהיה רקורסיבית, ז"א: נרוץ על הקודקודים, וכל עוד הקודקוד לבן - נפעיל עליו את הפונקציה של **DFS** (גם בתוך הפונקציה נפעיל בצורה רקורסיבית את הפונקציה - על כל שכן לבן). בשביל להמחיש את הדברים - נשתמש בדוגמא דלעיל:

בפעם הראשונה כל המערך "לבן", ולכן בשלב ראשון אנחנו מפעילים **DFS** על קודקוד 0:

```
time = time+1; ->0+1
first[0] = time; ->1
color[0] = gray;
```

רצים על כל השכנים של הקודקוד - על פי האיברים שקיימים בתוך הליניק ליסט, במקרה הראשון:

```
v = list[0].get(i); -> 1 // קבלת קודקוד לפי הסדר בלינק ליסט
```

אם השכן שהועתק מתוך הלינק ליסט עדיין לבן נאמר שאביו של הלבן - הוא הקודקוד שמטופל כעת בפונקציה (במקרה שלנו - 0), ונפעיל גם עליו את הפונקציה הרקורסיבית.

חשוב לשים לב - אמנם בבנאי אנחנו רצים על כל הקודקודים הלבנים, ובאמת בהתחלה כולם לבנים, אך לאחר טיפול בקודקוד הראשון, למשל במקרה שלנו - אפס, הרי שקודקוד 1 כבר יצבע באפור, ולכן כשנחזור לבנאי - לא נפעיל עליו את הפונקציה הרקורסיבית, בעצם יוצא שבגרף קשיר - אנחנו נפעיל את הפונקציה רק פעם אחת בבנאי...

מספר תכונות שחשוב לזכור:

- הסיבוכיות של **DFS** כמו של **BFS**: $O(n+m)$, כאשר n מספר הקודקודים, ו-m מספר הצלעות. עוד יש לזכור כי מספר הצלעות יכול לנוע באופן הבא: $n-1 \le m \le \frac{n(n-1)}{2}$.
- המסלול שמתקבל בעזרת **DFS**- אינו הקצר ביותר, אך לעיתים הוא טוב יותר משום שהוא בוחר מסלול אחד ולא מתפרס באופן מידי על כל הגרף.
- גם כאן ניתן לבדוק האם גרף קשיר, ואם לא - כמה רכיבי קשירות יש לו בעזרת **DFS** (כפי שמומש ב:**BFS**), בנוסף: ניתן לבדוק האם קיים מעגל בגרף (ניתן גם למצוא מהו המעגל, אך את זה לא מימשתי, רעיון המימוש - להפעיל שוב את **DFS** כאשר נקודת ההתחלה היא הקודקוד אליו הגענו כאשר גילנו שמדובר במעגל, וראה לקמן)

מימוש חלק א': בחלק זה נממש את עיקר הקוד של **DFS**, בהמשך נממש פונקציות נוספות, וראה לקמן - הסבר ספציפי לגבי אותן פונקציות.

```
import java.util.LinkedList;

public class shiur07_DFS {

	public static final int white = 0;
	public static final int gray = 1;
	public static final int black = 2;
	public static final int NIL = -1;
	int color[];
	int first[];
	int last[];
	int pred[];
	LinkedList<Integer>[] list;
	int size;
	int time;

	/**
	 * Constructor
	 * @param st the start vertex
	 * @param n the number of vertex
	 */
	public shiur07_DFS(int n, int st) {
		size = n;
		list = new LinkedList[size];
		// white=0, gray=1, black=2
		color = new int[size]; // מאופס בצבע לבן
		first = new int[size]; // מאופס באפס
		last = new int[size]; // מאופס באפס
		pred = new int[size];
		for (int i=st; i<size; i++) pred[i] = NIL; // מאופס במינוס אחד
		list = InPut_0to5(); // עדכון הרשימה על פי הדוגמא המובאת להלן
		time = 0;
		PrintGraph_0to5(); // פונקציה להדפסת הדוגמא בצורה של גרף
		// רציה על כל הקודקודים - כל קודקוד לבן מופעלת עליו הפונקציה הרקורסיבית, וראה הערות בתיאוריה דלעיל
		for (int i=0; i<n; i++) {
			if (color[i] == white) {
				System.out.println("White Cons");
				D_F_S(i);
			}
		}
	}

	/**
	 * Function finds all the tracks from st Vertex - Search in depth
	 * @param start the vertex that sent to the DFS Reco function
	 */
	public void D_F_S(int start) { // פונקציה רקורסיבית - מתחילה לעבוד מתוך הבנאי

		printArrays(color,pred,first,last);
		int v;
		time = time+1;
		first[start] = time;
		color[start] = gray;
```

```
            // רצים על כל השכנים של הקודקוד - על פי האיברים שקיימים בתוך הלינק ליסט
            for (int i=0; i<list[start].size(); i++){
                  v = list[start].get(i); // קבלת קודקוד מתוך הרשימה המקושרת
                  if (color[v] == white) {
                        pred[v] = start;
                        D_F_S(v); // הפעלת רקורסיה על קודקוד שכן - לבן
                  }
            }
            color[start] = black; // הקודקוד שהוכנס לפונקציה יהפך לשחור רק "בדרך חזור" - כאשר אין לו שכנים לבנים
            last[start] = ++time;
      }

      /**
       * @return a graph as we learned in class
       */

      public LinkedList<Integer>[] InPut_0to5() {
            for (int i=0; i<size; i++) list[i] = new LinkedList<Integer>();
            list[0].add(1); list[0].add(3);    list[1].add(0); list[1].add(4);
            list[2].add(4); list[2].add(5);    list[3].add(0); list[3].add(4);
            list[4].add(1); list[4].add(2); list[4].add(3);    list[5].add(2);
            return list;
      }

      /** Drawing the graph from the lesson */
      public void PrintGraph_0to5() {
            System.out.println("====== Graph 0 to 5 ======\n");
            System.out.println("(0)--------(1)        (2)");
            System.out.println(" |          |         /| ");
            System.out.println(" |          |        / | ");
            System.out.println(" |          |       /  | ");
            System.out.println(" |          |      /   | ");
            System.out.println(" |          |     /    | ");
            System.out.println(" |          |    /     | ");
            System.out.println(" |          |   /      | ");
            System.out.println(" |          |  /       | ");
            System.out.println(" |          | /        | ");
            System.out.println("(3)--------(4)        (5)");
            System.out.println("\n=========================\n");
      }

      /**
       * Helped static function -  vertex data printing
       * @param color the color of the vertex
       * @param pred the "father" of the vertex
       * @param first the Distance from the start in the first track
       * @param last the Distance from the start in the second track
       */
      public static void printArrays(int color[], int pred[], int first[], int last[]) {
            for (int i=0; i<color.length; i++) {
                  String temp = "";
                  if (color[i]==0) temp = "White";
                  if (color[i]==1) temp = "Gray";
                  if (color[i]==2) temp = "Black";
                  System.out.println("Vertex "+i+"\tPred="+pred[i]+
                        "\t\tFirst="+first[i]+"----Last="+last[i]+"\tThe color: "+temp);
            }
            System.out.println("()()()()()()()()()()()()()()()()()()()()()()()()()()\n");
      }
}
```

נעבור אל החלק השני, לא נבדוק כאן קשירות ורכיבי קשירות משום שעשינו את זה בקוד הקודם, אלא נשתמש ב:**BFS** בשביל לגלות אם קיים לפחות מעגל אחד בגרף, ואם כן - לדעת מהו המעגל (לא מומש - לדעת מהו).
בשביל לבצע את המימוש נשתמש בכל מה שבנינו בחלק א', אלא שנוסיף עליו:

א. משתנה עצם בוליאני: `boolean hasCircle`, שיקבל ערך **false** בבנאי.
ב. בתוך השיטה: D_F_S נוסיף בדיקה בלולאה הפנימית:

```
for (int i=0; i<list[start].size(); i++){
     v = list[start].get(i); // המקושרת הרשימה מתוך קודקוד קבלת
     if (color[v]==1 && v!=pred[start] && hasCircle==false) {
          hasCircle = true;
     }

     if (color[v] == white) {
          pred[v] = start;
          D_F_S(v);
     }
}
```

מדוע התנאי הזה גורם למציאת מעגל בגרף? משום שאם אנחנו מוצאים קודקוד שהוא אפור, אך באותה מידה גם לא אביו של אותו קודקוד - הרי שמדובר במעגל.

בעיה מס' 4 - עלות מינימלית בגרף משוקלל - Dijkstra

היום נלמד את האלגוריתם Dijkstra - מציאת הדרך הקצרה ביותר בגרף משוקלל (ז"א: יש מחיר שונה לכל צלע). בשונה משני המימושים הקודמים, כאן הקלט מעט שונה: בשביל לקבל את הקלט של התוכנית עלינו לבנות 2 מחלקות עזר:

האחת מייצגת צלע בגרף (shiur08_Edge), בתוך כל צלע 2 משתני עצם:

א. משקלה
ב. לאיזה קודקוד היא מתחברת.

השניה מייצגת קודקוד בגרף (shiur08_Vertex), לכל קודקוד כמה משתני עצם:

א. name - שם הקודקוד (מיוצג ע"י מספר)
ב. prev - אביו של הקודקוד
ג. edge - מערך של צעלות היוצאות מאותו קודקוד, משתנה עצם זה הוא בעצם המחלקה שיצרנו לעיל
ד. dist - המרחק אל הקודקוד
ה. משתנה עזר בוליאני המייצג את סיום העבודה עם אותו קודקוד.

כאשר נגדיר את הקודקוד בבנאי - הרי ששמו יתקבל בבנאי, מרחקו יוגדר כאינסוף, והמשתנה הבוליאני יוגדר כ"שקר", משום שהעבודה על אותו קודקוד לא הסתיימה.

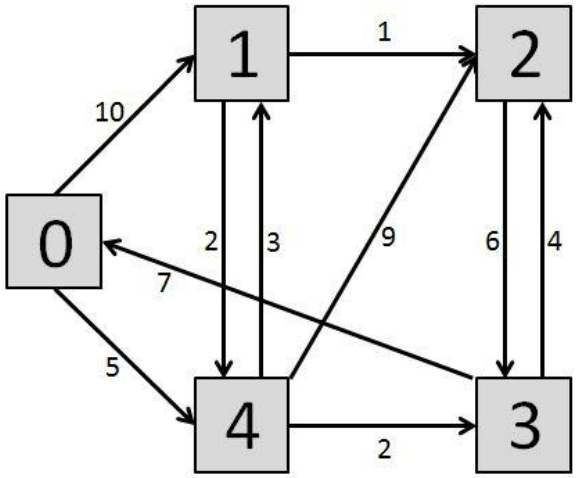
דוגמא לקלט (עיין במימוש שם מובא הקלט באופן מפורט)

```
shiur08_Vertex v0 = new shiur08_Vertex(0);
```

בעצם יצרנו קודקוד ששמו 0, כעת ניצור לו מערך של צעלות:

```
v0.edge = new shiur08_Edge[]
                {new shiur08_Edge(1,10), new shiur08_Edge(4,5)};
```

ז"א: קודקוד 0 בעל 2 צלעות:
האחת עוברת לקודקוד 1 כאשר מחיר הצלע הוא 10,
השניה עוברת לקודקוד 4 כאשר מחיר הצלע הוא 5.



הדוגמא שעליה נדבר לאורך כל התאוריה היא גם הדוגמא שנממש בקוד המובא להלן.

גם בחיפוש זה אנחנו נכניס לאלגוריתם את הקודקוד בו נרצה להתחיל והאלגוריתם יחשב את המרחק הקצר ביותר בין נקודת ההתחלה ליתר הקודקודים (כאשר המרחק נקבע לפי המחיר הזול ביותר, ולא לפי מספר הצלעות).

בשביל להמחיש את הדברים נאמר והקודקוד בו אנחנו מתחילים הוא 0, איך יראה הפלט:

```
ver 0, dist: 0.0, prev: -1, visit: false
ver 1, dist: Infinity, prev: -1, visit: false ...
```

בשלב ראשון, המרחק של קודקוד ההתחלה יהפוך לאפס, שלב שני:

```
ver 0, dist: 0.0, prev: -1, visit: true
ver 1, dist: 10.0, prev: 0, visit: false
ver 2, dist: Infinity, prev: -1, visit: false
ver 3, dist: Infinity, prev: -1, visit: false
ver 4, dist: 5.0, prev: 0, visit: false
```

אפשר לראות בבירור שהמרחק הזול ביותר בין קודקוד 0 ל-1 הוא 8, שהרי עדיף ללכת לא בצורה ישירה (אלא דרך קודקוד 4), לכן שימו לב מה קורה בשלב הבא:

```
ver 0, dist: 0.0, prev: -1, visit: true
ver 1, dist: 8.0, prev: 4, visit: false
ver 2, dist: 14.0, prev: 4, visit: false
ver 3, dist: 7.0, prev: 4, visit: false
ver 4, dist: 5.0, prev: 0, visit: true
```

ז"א: האלגוריתם מצא דרך זולה יותר, וייתכן שבהמשך הוא ימשיך וימצא דרך טובה יותר, אלא אם כן הקודקוד יהפוך ל"אמת" מה שאומר שהעבודה עליו נגמרה, כמו קודקוד מס' 4, מה שאומר שהמרחק הזול ביותר הוא 5.

אם כן, מבנה הנתונים בו נשתמש: PriorityQueue המכיל איברים מסוג "קודקוד". גם כאן האיבר הראשון יכנס לתור, והמרחק שלו יהיה אפס.
כעת ניכנס ללולאה - כל עוד התור לא ריק, בדומה למימוש של DFS. בתוך הלולאה אנחנו נוציא את הקודקודים שבתוך התור ונטפל בכל השכנים של כל קודקוד. ננסה להבין יותר לעומק כיצד הלולאה הפנימית עובדת: נאמר שאנחנו בתחילת העבודה, וקודקוד הבסיס הוא 0, ז"א: אנחנו מוציאים את 0 מתוך התור (כרגע התור ריק), ומתחילים לעבוד עליו:

```
shiur08_Vertex u = q.poll(); -> u = 0
```

כעת רצים על כל השכנים של 0 (כפי שמופיע בגרף, השכנים הם: 1,4):

```
for (int i=0; i<u.edge.length; i++)
```

כעת ניצור 2 משתני עזר, בשביל הדוגמא, נתייחס לצלע הראשונה במערך, ז"א:

```
shiur08_Edge e = u.edge[0]; -> e = (1,10)
```

כעת נקבל את הקודקוד עצמו מתוך מערך הקודקודים שנשלח לפונקציה.

```
shiur08_Vertex v = ver[e.vertex]; -> v = ver[1] : dist: Infinity, prev: -1, visit: false
```

כעת נשאל:

```
if (!v.visit)
```

אם לא סיימנו לעבוד על הקודקוד - נכנס לתנאי, ושוב נשאל:

```
if (v.dist > u.dist + e.weight) -> (Infinity > 0 + 10)
```

בגלל שעד עכשיו לא היה שום קשר בין הקודקודים - אז כל קשר, כמה גדול שיהיה - עדיין יהיה קטן מאינסוף, ולכן התנאי מתקיים - נגדיר כעת את המרחק של v מהשורש:

```
v.dist = u.dist + e.weight -> v.dist = 10
```

גם נגדיר את האב של v - הקודקוד שעליו ביצענו את כל התהליך (ז"א: u, ובמקרה שלנו - אפס):

```
v.prev = ver[u.name].name; -> v.prev = ver[0].name = 0
```

כל מה שנותר לנו זה להכניס לתוך התור את השכן שעליו ביצענו את הפעולה ועדכנו את הנתונים שלו:

```
q.add(v); -> v = dist: 10, prev: 0, visit: false
```

חשוב לשים לב: ישנן מציאויות בהן אנחנו נצטרך למחוק את האיבר ולהכניס אותו בשנית - וזאת בשביל להכליל בתוכו את כל השינויים שעשינו בו, לכן לפני פונקציית ההכנסה - נעשה פונקציית מחיקה, שתמחוק - אך ורק אם האיבר אכן קיים בתור:

```
q.remove(v); -> noting
```

כעת, קודקוד מס' 1 מחכה להמשך טיפול, ובנתיים אנחנו חוזרים על הלולאה פעם נוספת - בשביל לטפל בשכן השני של 0 - קודקוד מס' 4 (אותו תהליך יתרחש גם כאן. לאחר שנסיים עם קודקוד 4, גם הוא יכנס לתור בשביל המשך טיפול, ואז בעצם טיפלנו בכל השכנים של 0. מחוץ ללולאה שרצה על כל השכנים יש את הפקודה:

```
u.visit = true;
```

מה שאומר שקודקוד u סיים את התהליך.

כעת נמשיך בלולאה החיצונית - התור אינו ריק, שהרי הכנסנו לתוכו את כל השכנים של קודקוד האב, כעת - נמשוך את הקודקוד הבא בתור, ונחזור על הפעולה דלעיל.

הסיבוכיות: חשוב לשים לב שעל הקודקודים אנחנו עוברים פעמיים + מספר הצלעות, ז"א:

O(e*e+v)->O(n^2)

כל זה בשימוש במערך רגיל, אך אם משתמשים בתור עדיפויות (ששולף בכל פעם את המספר הגדול, וכפי שמומש לעיל), הסיבוכיות קטנה יותר: O(n*log(n)).

חשוב לשים לב שבשביל להגדיר תור עדיפויות שמקבל קודקודים - יש להגדיר מהו ה"Comparable<shiur08_Vertex>", וכפי שהובא במימוש:

```
public int compareTo(shiur08_Vertex v) {
      if (dist>v.dist) return 1;
      if (dist<v.dist) return Nil;
      return 0;
}
```

נעבור למימוש הקוד, נממש כל מחלקה בנפרד, שים לב, במקום בו הרחבתי את ההסברים בתיאוריה - לא הסברתי את כל התהליך שוב במימוש.

```
/** the Class representing edge between Bone vertex to another vertex */
public class shiur08_Edge {

	int vertex; // שם הקודקוד אליו הצלע נשלחת
	double weight;
	/**
	 * @param v the name of the another vertex
	 * @param w the Weight of the edge
	 */
	public shiur08_Edge(int v, double w) {
		vertex = v;
		weight = w;
	}
}
```

כעת נבנה את הקודקוד עצמו, מעבר לשם שלו ישנם מספר משתנים נוספים שיש להתחשב בהם, וכפי שהוסבר לעיל בתאוריה:

```
public class shiur08_Vertex implements Comparable<shiur08_Vertex> {

	int name; // שם הקודקוד
	int prev; // שם הקודקוד הקודם לאותו קודקוד (האב)
	shiur08_Edge edge[]; // מערך של צלעות - הערך
	double dist; // מרחק של כל קודקוד מקודקוד המקור
	boolean visit;

	public static final Double Nsof = Double.POSITIVE_INFINITY;
	public static final int Nil = -1;

	// יצירת קודקוד חדש, המרחק אליו הוא אינסוף
	// כל עוד לא סיימנו לעבוד על הקודקוד - הערך התחלתי יישאר
	// visit = false;
	public shiur08_Vertex(int name) {
		this.name = name;
		dist = Nsof;
		visit = false;
		prev = -1;
	}

	// מימוש פונקציית השוואה בסופו של דבר לא השתמשתי בה בקוד עצמו
	public int compareTo(shiur08_Vertex v) {
		if (dist>v.dist) return 1;
		if (dist<v.dist) return Nil;
		return 0;
	}
}
```

מחלקה אחרונה היא בעצם האלגוריתם שמשתמש בכל מה שבנינו לעיל:

```
import java.util.PriorityQueue;

public class shiur08_Dijkstra {

	/** @return Array of Vertex */
	public static shiur08_Vertex[] initGraph(){
		// יצירת הקודקודים
		shiur08_Vertex v0 = new shiur08_Vertex(0);
		shiur08_Vertex v1 = new shiur08_Vertex(1);
		shiur08_Vertex v2 = new shiur08_Vertex(2);
		shiur08_Vertex v3 = new shiur08_Vertex(3);
		shiur08_Vertex v4 = new shiur08_Vertex(4);
```

```
            // יצירת הצלעות לכל קודקוד
            v0.edge = new shiur08_Edge[]{new shiur08_Edge(1,10), new shiur08_Edge(4,5)};
            v1.edge = new shiur08_Edge[]{new shiur08_Edge(2,1), new shiur08_Edge(4,2)};
            v2.edge = new shiur08_Edge[]{new shiur08_Edge(3,6)};
            v3.edge = new shiur08_Edge[]{new shiur08_Edge(2,4), new shiur08_Edge(0,7)};
            v4.edge = new shiur08_Edge[]{new shiur08_Edge(3,2), new shiur08_Edge(2,9),
                                                                new shiur08_Edge(1,3)};
            // מערך קודקודים הכולל את כל הנתונים
            shiur08_Vertex vs[] = {v0,v1,v2,v3,v4};
            return vs;
      }

      /**
       * @param ver the Array of the Vertex
       * @param num the start Vertex
       */
      public static void Dijkstra(shiur08_Vertex ver[], int num) {
            PriorityQueue <shiur08_Vertex> q = new PriorityQueue<shiur08_Vertex>();
            shiur08_Vertex start = ver[num];
            q.add(start); // קודקוד התחלתי נכנס לתור
            start.dist = 0; // מרחקו שווה לאפס
            while (!q.isEmpty()) {
                  // מוציאים איבר מתוך התור ומטפלים בכל השכנים שלו
                  shiur08_Vertex u = q.poll();
                  System.out.println("and the poll is: "+u.name);
                  PrintDij(ver); // קריאה לפונקציית עזר המדפיסה את הנתונים בכל שלב
                  // להבנת הדברים בצורה טובה יותר - עיין בתיאוריה
                  for (int i=0; i<u.edge.length; i++) {
                        shiur08_Edge e = u.edge[i];
                        shiur08_Vertex v = ver[e.vertex];
                        if (!v.visit) { // כל עוד קיימים שכנים
                              if (v.dist > u.dist + e.weight) {
                                    v.dist = u.dist + e.weight;
                                    v.prev = ver[u.name].name;
                                    q.remove(v); // הוצאה - רק אם האיבר באמת קיים
                                    q.add(v);
                              }
                        }
                  }
                  u.visit = true; // עברנו על כל שכני הקודקוד
            }
      }

      /**
       * static fun' to print the vertex
       * @param v the Array of vertex
       */
      public static void PrintDij(shiur08_Vertex v[]) {
            for (int i=0; i<v.length; i++) { System.out.println
               ("ver "+i+", dist: "+v[i].dist+", prev: "+v[i].prev+", visit: "+v[i].visit);
            }
            System.out.println("*********************************\n");
      }
}
```

פונקציה ססטית נוספת שלא מימשנו במהלך השיעור, אך לאחר כל המימושים הנ"ל היא מאוד פשוטה:
מציאת המסלול הזול ביותר בין קודקוד ההתחלה לכל קודקוד אחר בגרף:
חשוב לשים לב שלכל קודקוד יש את הקודקוד הקודם לו, ולכן ניתן ללכת מקודקוד המטרה אחורה - עד להגעה לקודקוד ההתחלה. במקרה והמרחק הקודקוד המבוקש הוא "אינסוף" - הרי שאין כלל מסלול המחבר את 2 הקודקודים.

```
/**
 * @param v the Array of vertex after Dijkstra
 * @param start the Start vertex
 * @param end the vertex we want to find his track
 * @return String of track
 */
public static String getTrack(shiur08_Vertex v[], int start, int end) {
	String ans = "";
	if (v[end].dist == Double.POSITIVE_INFINITY) return "there is no track!";
	while (v[end].dist != 0) {
		ans = end + " --> " + ans;
		end = v[end].prev;
	}

	ans = start + " --> " + ans;
	return ans;
}
```

את אורך המסלול אין שום בעיה לחשב, הוא בעצם נתון במשתנה העצם של הקודוד אליו אנחנו רוצים להגיע, ז"א: אורך המסלול הוא:

```
double sum = v[end].dist;
```

בעיה מס' 5 - מציאת תת קטע ארוך ביותר - Best

בעיית הדגל: למצוא קוטר של גרף נתון (קוטר: הקטע הארוך ביותר בגרף, לא מבחינת מספר קטעים, אלא בשקלול מחירי הצלעות). אפשר היה להשתמש ב FW מבעיית הבקבוקים: כאשר המטריצה תתמלא במחירי הצלעות שבין קודקוד לקודקוד, אלא שעלות האלגוריתם הזה (כולל בניה וחיפוש) היא O(n^3), ואנחנו מחפשים תשובה טובה יותר.
בשביל להגיע למטרה זו נעבור דרך כמה סוגים של תתי קטעים: נתחיל במערך בודד, נעבור למערך מעגלי, ובסופו של דבר לסכום של מלבן (תת מטריצה) בתוך מטריצה.

<u>שלב ראשון</u>: Best - במערך של מספרים:

במהלך ההרצאה הראה פרו' לויט כיצד ניתן לשפר את האלגוריתם עד לקבלת: O(n). נעבור בקצרה על השלבים:

א. חיפוש שלם: לקחת את כל תתי הקטעים, לחשב את אורכם ולהחזיר את התת קטע הגדול ביותר.
הסיבוכיות כאן גדולה מאוד. חשוב לא להתבלבל, לא מדובר כאן ב: 2^n, אלא בפחות מכך: בחירה של 2 מתוך n איברים, אלא בגלל שייתכן ונבחר איבר בודד יש לתקן את הסיבוכיות, ז"א: $\binom{n}{2} = \frac{n(n-1)}{2} \rightarrow \frac{n(n+1)}{2}$
אלא שבזה לא סיימנו, שהרי אנחנו סוכמים כל קטע, ז"א: יש להוסיף עוד n פעולות, ולכן הסיבוכיות היא: O(n^3). (מעבר לכך מעבר על כל הסכומים שמצאנו זה: O(n^2), אך זה נעשה בלולאה נפרדת ולכן הסיבוכיות נשארה בעינה).

1	2	3	4
1	2	-10	5

ב. עדיין בחיפוש שלם, אך עם שיפור מסוים:
בשביל הבנת הדברים ניתן דוגמא: נאמר ומערך המספרים שלנו הוא: (אינדקס שורה ראשונה, ערכים שורה שניה).

	1	2	3	4
1	1	3	-7	-2
2		2	-8	-3
3			-10	-5
4				4

נאמר ואנחנו מקבלים כנתון את המטריצה הבאה: האלכסון הראשי של המטריצה הוא המערך עצמו, והמשולש העליון זהו חישוב העלות בין כל קודקוד לקודקוד אחר.
העלות של ריצה על המטריצה הינו: O(n^2). אך עדיין, בניית המטריצה תעלה במחיר של O(n^3).
אלא שיש לשים לב שיש כאן חוקיות - מה שנותן אופציה זולה יותר במילוי המטריצה: כמובן שאת האלכסון הראשי נהיה חייבים להכניס למטריצה, אך את המשולש העליון נמלא מלמטה למעלה באופן הבא:

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

כמובן שחוקיות זו יש להוכיח, אך אנחנו מניחים כי הוכחנו אותה.

הקוד של המילוי יראה כך (כאשר dim הינו גודל המערך):

```
for (int i=0; i<dim; i++) mat[i][i] = arr[i];
for (int i=0; i<dim; i--) {
     for (int j=i+1; j<dim; j++) {
          mat[i][j] = mat[i][j-1] + arr[j];
          //mat[i][j] = mat[i][j-1] + mat[j][j];
     }
}
```

ניתן לראות כי הצלחנו לרדת בסיבוכיות ל: O(n^2) גם במילוי המטריצה.

במהלך השיעור שאל פרו' לויט שאלות הקשורות לזמן ריצה, ז"א: אם נשנה את האלגוריתם - לא בצורה של פעולה שונה, אלא בקריאה מתוך המערך במקום מתוך המטריצה, קריאה של אינדקסים באופן הפוך - וכו', מה יהיה זמן הריצה העדיף. האופציה העדיפה - קריאה מתוך מערך.
עוד יותר מהיר יהיה לעבוד באופן הבא:

```
for (int i=0; i<dim; i++)
     for (int j=i+1; j<dim; j++) mat[i][j] = mat[i+1][j] + arr[i];
```

ג. פתרון הבעיה בסיבוכיות O(n):

בשביל להגיע לפתרון מעין זה אנחנו צריכים להחליף את כל הבסיס, שהרי אם נעבוד עם מטריצה - לא נוכל להגיע לסיבוכיות שכזו, לכן נעבור לעבוד עם תכנות דינאמי. בתכנות הדינאמי אנחנו נשתמש באלגוריתם חמדני, אך משופר, נסביר את הדברים בעזרת דוגמא:

-10	8	-1	8	-9	1	-1	-9	1	-9

בשביל לפתור את הבעיה בעזרת האלגוריתם הבא יש להניח כי ישנו לפחות מספר חיובי אחד במערך (יש להוסיף בדיקה בתחילת המימוש).

כעת, כל סכום שלילי - אין לנו שום סיבה להתחשב בו, כי הוא בוודאי לא הסכום הגדול ביותר. ולכן - על האיבר הראשון אנחנו נדלג עד שנגיע לאיבר חיובי - 8. כרגע גם הסכום הגדול ביותר הוא 8. אנחנו נמשיך ונסכום עד שנגיע לסכום שלילי, כל סכימה - אנחנו נבדוק את הסכום שהגענו אליו ביחס לסכום הגדול ביותר, למשל:

בפעם הראשונה הגענו ל-8, וזה גם הסכום הגדול ביותר, המשכנו למינוס 1, הסכום עדיין חיובי ולכן נמשיך, אך הסכום הגדול נשאר על 8, אח"כ נמשיך ונסכום - נקבל 15, והסכום המקסימאלי ישתנה ל-15, כך נמשיך עד שנגיע לסכום שלילי, במקרה והגענו לסכום כזה - נאפס את הסכימה ונתחיל לסכום שוב מאפס, ז"א:

-10	8	-1	8	-9	1	-1	-9	1	-9
	8	7	15	6	7	6	-3	1	-8

(המסומן באפור - מאפס את הספירה וזוכר את הגודל המקסמאלי שהיה עד כה),
המסומן בצהוב - מתחיל את הספירה הנוכחית מאפס).

את ההוכחה לכך עשה פרו' לויט בדרך השלילה.

אמרנו לעיל שכל זה נכון דווקא בהנחה שישנו מספר חיובי אחד, בשביל שהאלגוריתם תמיד יהיה נכון - יש לבדוק - אם כל המספרים הם שלילים - יש להחזיר את הגדול ביותר (וראה במימוש).

נקודה שיש לשים לב אליה - ניתן להוסיף לאלגוריתם תנאי, שיחזיר את תת המערך הגדול ביותר - בעל מספר האיברים הקטן ביותר, למשל: 1,-1,10 יחזיר 10 בלבד, ללא תוספת של 1 ומינוס אחד - משום שבלאו הכי הסכום שלהם שווה לאפס. אכן נשאלה שאלה כזו במבחן האחרון (2013 מועד א') ועיין בקובץ השאלות.

מבנה האלגוריתם: במבנה זה נשתמש באובייקט עם כמה משתני עצם:

max - משתנה שזוכר את הסכום המקסימאלי במערך
beginMax - אינדקס בו מתחיל תת הקטע הארוך ביותר, end- אינדקס בו מסתיים תת הקטע הארוך ביותר
arr[] - המערך עליו נפעיל את החיפוש

בבנאי נגדיר את האיבר הראשון להיות הגדול ביותר (באופן שרירותי), ונעשה העתקה עמוקה למערך המתקבל. נבנה פונקציה שתשנה את ערכי העצם בהתאמה לפי מה שהובא לעיל בתיאוריה, וראה הסבר במימוש.

מימוש Best:

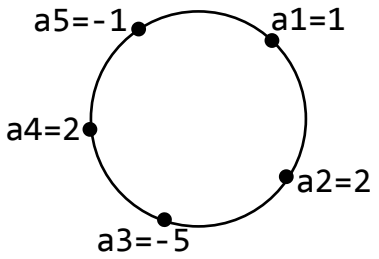
```
import java.util.Arrays;

public class Best {

	int max;
	int beginMax;
	int end;
	int arr[];

	public Best(int Array[]) {
		arr = Array;
		max = arr[0];
		beginMax = 0;
		end = 0;
	}


	/**
```

```
 * this function find the best Sub-section
 * @return the sum of the best Sub-section
 */
public int getBest() {
        int i = 0;
        int tempSum, tempBegin;
        while (arr[i]<=0) {
                i++; // עד לאיבר החיובי הראשון
                if(i == arr.length) return max;
                                        // הגענו לסוף ולכן כולם שליליים, לכן אנחנו מחזירים את האיבר הגדול ביותר
                if (arr[i] > max) {
                        max = arr[i]; // אם מצאנו איבר שלילי גדול יותר - מחליפים אותו
                        beginMax = i;
                        end = i;
                }
        }

        tempSum = 0;
        tempBegin = beginMax; // ממשיכים כאן רק אם יש לפחות מספר אחד חיובי
        while (i < arr.length) {
                tempSum = tempSum + arr[i];
                if (tempSum<0) { // ברגע שהסכום ירד מאפס
                        tempSum = 0; // מאפסים את הסכום של משתנה העזר
                        tempBegin = i+1; // מקדמים את נקודת ההתחלה הזמנית
                } else if (tempSum >= max) { // אם לא - בודקים האם הסכימה האחרונה שיפרה את התוצאה
                        max = tempSum;
                        beginMax = tempBegin;
                        end = i;
                }
                i++;
        }
        return max;
}

/**
 * this function help to print the result of Best
 */
public void printResult() {
        System.out.println("the Array is: "+ Arrays.toString(arr));
        System.out.println("= = = = = = = = = = = = = = = = = = = = = = = = =");
        System.out.println("the start index is: " + beginMax);
        System.out.println("the end index is:   " + end);
        System.out.print(" the max sub Array is: [\t");
        for (int i = beginMax; i <= end; i++) {
                System.out.print(arr[i]+"\t");
        }
        System.out.println("]");
        System.out.println("the max sum is: " + max);
        System.out.println("==========================================");
}
}
```

נמשיך בבעיה האחרונה:

<u>שלב שני</u>: Best - במערך מספרים מעגלי:

שלב זה לא יקדם אותנו בשאלה אותה שאלנו בתחילת הדברים (מציאת קוטר של גרף), אך בכל זאת נביא את הדברים כאן משום הסדר הכרונולוגי והקשר בין Best במערך רגיל לבין Best במערך מעגלי.



בשביל להבין את הדברים נשתמש בדוגמא: נאמר ויש לנו מערך מעגלי: נחפש כעת את תת הקטע הגדול ביותר. חשוב לשים לב שבעצם נוצרים לנו 5 מערכים לא מעגליים (בכל פעם מתחילים מאיבר אחר), ובעצם מספר תתי הקטעים שנוצרים הוא: $\frac{n(n-1)}{2}$.

בשיעור העלו רעיונות לפתרון הבעיה. הפתרון הפשוט ביותר: לפרק את המערך המעגלי למערכים לא מעגליים (למשל, בדוגמא שלנו - 5 מערכים), ולהפעיל על כל אחד מהם את הפונקציה מהשיעור שעבר (Best במערך רגיל), ואז לבדוק מי נותן את הסכום הגדול ביותר מבין כל המערכים. סיבוכיות הפתרון היא: O(n^2 + n) שזה בפועל: O(n^2).

אלא ישנה דרך יעילה יותר O(n): נחלק את העבודה ל-2:
חלק ראשון נטפל במקרים "הנורמאליים": נתחיל מהאינדקס הראשון ונחפש את תת הקטע הארוך ביותר עד האינדקס האחרון (בדיוק כמו שעשינו בשיעור שעבר) ונשמור אותו.
חלק שני: נטפל במקרים "הלא נורמאליים", ז"א: מקרה בו תת המערך מתחיל באמצע המערך ומסתיים בצורה מעגלית - לאחר תחילת המערך, למשל: 1,-9,2,3 -> 2,3,1 (ירוק - נקודת התחלה, אדום - נקודת סיום), נמצא את תת הקטע "הלא נורמאלי" הארוך ביותר ונשמור אותו.

מה שנותר לעשות זה לבדוק איזה תת קטע גדול יותר (של החלק הראשון או של החלק השני) ולהחזיר אותו. כמובן שהקושי הגדול - איך אפשר למצוא את תת הקטע הארוך ביותר במקרים "הלא נורמאליים"? חשוב לשים לב שלכל קטע לא נורמאלי יש משלים נורמאלי. אם כן - אם נמצא את תת הקטע הנורמאלי הקטן ביותר - הרי שהמשלים שלו זהו תת הקטע הלא נורמאלי הארוך ביותר...

ניקח דוגמא בשביל שהדברים יהיו ברורים יותר:

1	2	-5	5	-1

נעשה את חלק א': תת הקטע הנורמאלי הארוך ביותר הוא: 5.
נעשה את חלק ב': תת הקטע הנורמאלי הקצר ביותר הוא: 5-, ולכן המשלים שלו הוא כל השאר, ז"א: 7.

אם כן, הנוסחא למציאת תת המערך במערך מעגלי: $\max\left(Best(A), \sum(A) - \overline{Best(A)}\right)$

כעת רק נותר למצוא את הערך המינימאלי בחלק הנורמאלי, ז"א: $\overline{Best(A)}$. לשם כך ניצור מערך עזר הזהה למערך המקורי, נכפיל את כל האיברים שבו במינוס אחד, ונפעיל Best רגיל. כעת הקטע הגדול ביותר - הוא בעצם הקטע הקטן ביותר במערך המקורי (המיקום של האינדקסים). ולכן ניתן לשפר את הנוסחא דלעיל ולומר כך: $\max(Best(A), \sum(A) + Best(-A))$.

גם כאן, אם כל האיברים שליליים - יש להחזיר את האיבר הגדול ביותר מבין כולם, ואם כולם חיוביים - יש להחזיר את הסכום של כולם (בדיוק כמו במימוש הקודם, וראה במימוש).

במהלך השיעור פרו' לויט הביא שאלה שיכולה להופיע בבחינה (לא ניסיתי לממש כרגע, מחוסר זמן) - מציאת תת מערך גדול ביותר בתוך איקס (ז"א: מקבלים 2 מערכים, עם נקודות המפגש בין שניהם ומוצאים את הבסט בין ששת המערכים שיוצר האיקס).

לאחר שסיימנו את נושא המערך המעגלי, הביא פרו' לויט את בעיית "לובש" - בעיית תחנות הדלק, וראה הסבר מפורט לאחר מימוש Best במערך מעגלי.

מימוש Best - במערך מספרים מעגלי:

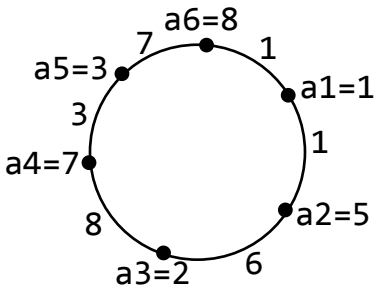
חשוב לשים לב, בפועל המערך אינו מעגלי, אלא אנחנו מתייחסים אליו כאל מערך מעגלי וזאת בעזרת ההגדרה שלמדנו במבנה נתונים (כאשר למדנו "תור מעגלי") - שניתן ללכת במעגל על גבי המערך בעזרת שארית חלוקה, ובמקרה שלנו:

```
for (int i = 0; i < sizeOfBest; i++)
      System.out.print(plusArr[(i+beginCircle) % plusArr.length]+"\t");
```

כאשר beginCircle זהו המיקום של תחילת תת הקטע הגדול ביותר.

```
public class shiur10_CircleBest {

      int maxCircle;   // משתנה לשמירת סכום תת המערך הגדול ביותר
      int sumAll;      // משתנה עצם לשמירת סכום כל האיברים במערך
      int plusArr[];   // מערך עזר חיובי
      int minusArr[];  // אותו מערך עם הפעלה שלילית
      int beginCircle;// שמירת נקודת ההתחלה, וראה בפונקציית ההדפסה
      int endCircle;   // שמירת נקודת הסיום
      int sizeOfBest;  // משתנה עצם לשמירת תת גודל תת המערך הגדול ביותר

      public BestInCircle(int arr[]) {
            int size = arr.length;
            maxCircle = arr[0];
            sumAll = 0;
            plusArr = new int[size];
            minusArr = new int[size];
            beginCircle = 0;
            endCircle = 0;
            sizeOfBest = 0;

            for (int i = 0; i < size; i++) {
                  plusArr[i] = arr[i];
                  minusArr[i] = (-1)*arr[i];
                  sumAll = sumAll + arr[i]; // שמירת סכום האיברים כולו לטובת המשך התוכנית
            }
      }
      /**
       * find the Best sub Array in circle Array
       */

      public int getCircleBest() {
            Best positive = new Best(plusArr);
            int maxPositive = positive.getBest();
            if (maxPositive<0) { // כל איברי המערך שליליים - ולכן הפונקציה עוצרת כאן
                  maxCircle = maxPositive;
                  beginCircle = positive.beginMax;
                  endCircle = positive.end;
                  sizeOfBest = 1;
                  return maxCircle;
            }

            Best negative = new Best(minusArr);
            int maxNegative = negative.getBest();

            if (maxPositive >= sumAll + maxNegative) {// אם החלק הנורמאלי גדול מהחלק הלא נורמאלי-בסט רגיל
                  maxCircle = maxPositive;
                  beginCircle = positive.beginMax;
                  endCircle = positive.end;
                  sizeOfBest = endCircle - beginCircle + 1;
```

```
            } else { // אחרת - יש לפעול לפי מה שלמדנו בתיאוריה
                  maxCircle = sumAll + maxNegative;
                  beginCircle = negative.end+1;
                  endCircle = negative.beginMax-1;
                  sizeOfBest = plusArr.length - beginCircle + endCircle + 1;
            }
            return maxCircle;
      }

      public void printResult() {
            System.out.println("the Array is: "+ Arrays.toString(plusArr));
            System.out.println("= = = = = = = = = = = = = = = = = = = = = = = =");
            System.out.println("the start index is: " + beginCircle);
            System.out.println("the end index is:   " + endCircle);
            System.out.print(" the max sub Array is: [\t");
            for (int i = 0; i < sizeOfBest; i++) { // הדפסה מעגלית ע"פ כמות האיברים
                  System.out.print(plusArr[(i+beginCircle) % plusArr.length]+"\t");
            }
            System.out.println("]");
            System.out.println("the max sum is: " + maxCircle);
            System.out.println("the number of element is: "+ sizeOfBest);
            System.out.println("=============================================");
      }

}
```



כאמור, בעיה נוספת הקשורה למערך מעגלי היא בעיית "תחנות הדלק". הבעיה בקצרה: מכונית צריכה לצאת לדרך מקודקוד מסוים במעגל ולחזור לאותו קודקוד, כל קודקוד מציין תחנת דלק וגובה הדלק שיש באותה התחנה, כל צלע מציינת את כמות הדלק שנסיעה תצרוך עד לקודקוד הבא. אנחנו מניחים שברכב אין דלק ובזמן שהוא יוצא מהקודקוד הראשון - הדלק שיש ברשותו זהו הדלק שהיה בתחנה הראשונה. בנוסף - ניתן להגיע לתחנה עם 0 דלק, אך לא ניתן להיכנס למינוס.

אפשר לראות בבירור, כי בדוגמא דלעיל - ניתן לצאת מקודקוד a1, ולהגיע לקודקוד: a2, אך שם - כבר לא נוכל להמשיך... משום שהנסיעה יקרה מכמות הדלק שברשותנו.

האם אין שום אפשרות לסגור מעגל בגרף הנתון? וודאי שיש, פשוט יש להתחיל את הנסיעה מקודקוד אחר... איך נדע מאיזה קודקוד ניתן להתחיל ולהצליח לסיים? (אם בכלל) - נשתמש בדבר שמאוד דומה למימוש של Best מעגלי. תחילה ניצור מערך חדש: כמות הדלק בתחנה פחות עלות הנסיעה שלאחריה, ובדוגמא שלנו:

A	1	5	2	7	3	8
B	1	6	8	3	7	1
C	0	-1	-6	4	-4	7

כעת ניתן לראות בבירור מהיכן כדאי לנסות ולצאת: מתת המערך הכי גדול במערך החדש - משום שהוא מקנה לנו את הביטחון הכי גדול שניתן - לסיים את המעגל. (כמובן, אם C<0 הרי שבוודאי לא ניתן לעבור על המעגל דרך שום קודקוד).

אלגוריתם זה דומה מאוד לאלגוריתם של Best מעגלי, רק שכאן אנחנו הולכים צעד נוסף קדימה - משתמשים בנתון של תת המערך הגדול ביותר ומוצאים פתרון לסגירת המעגל...

נמשיך בבעיה האחרונה:

<u>שלב שלישי</u>: Best - במטריצת מספרים:

בשלב זה אנחנו בעצם מגיעים לפתרון בעיית הדגל: למצוא קוטר של גרף, כאשר הגרף נתון כמטריצה, וכל איבר במטריצה מייצג את העלות של הצלע בין 2 הקודקודים. מה שאנחנו בעצם מחפשים זוהי תת מטריצה בעלת הערך הגדול ביותר (ייצוג תת המטריצה יעשה עם 4 משתנים - נקודת התחלה במטריצה, ונקודת סיום - באופן נגדי, ז"א: [i,j]->[p,q]).

במהלך ההרצאה הראה פרו' לויט מספר דרכי פתרון לבעיה, כאשר ההבדל המרכזי בין השיטות זוהי רמת הסיבוכיות:

1. o(n^6)
2. o(n^4)
3. o(n^3)

כמובן שמעבר לשינוי בסיבוכיות ישנו שינוי בגישה לתרגיל, וכפי שכבר הובא באלגוריתמים הקודמים (כאשר הורדנו את Best ב"מערך" מ-o(n^2) ל-o(n).

את מרכז הכובד נשים כמובן על המימוש האחרון, שהוא הטוב ביותר (ורק הוא עובד לפי האלגוריתם של Best), אך כמובן שנסביר גם את 2 המימושים הראשונים, בעיקר שהפתרון השלישי מסתמך על השיטה של הפתרון השני, והשני על הראשון...

<u>שלב ראשון</u>: סיבוכיות o(n^6):

	N	N
M=1	A[1,1]	A[1,2]
M=2	A[2,1]	A[2,2]
M=3	A[3,1]	A[3,2]

אנחנו בעצם מדברים על חיפוש שלם, כאשר אנחנו נחפש את כל תתי הקטעים, ונמצא את תת המלבן שנותן את הסכום הגדול ביותר. למשל: נאמר ויש לנו מטריצה כזו: כמה תתי מלבנים יש כאן? 18. מבחינת סיבוכיות:

$\frac{n(n+1)}{2} * \frac{m(m+1)}{2}$, כאשר:

```
[1,1],[1,1->2], ...[1,n]
        ,[2,2], ...[2,n]
              , ...[m,n]
```

בשלב זה סיימנו לאסוף את הנתונים - אך אנחנו צריכים עוד o(n^2) בשביל לחבר את האיברים בכל תת קטע, סה"כ: o(n^6).

<u>מימוש שלב א' - o(n^6)</u>:

```
public class shiur11_maxMatrix_n6 {

	int mat[][];
	int n,m; // אורך ורוחב המטריצה

	public shiur11_maxMatrix_n6(int matrix[][]){
		n = matrix.length;
		m = matrix[0].length;

		mat = new int [n][m];

		for (int i = 0; i < n; i++)
			for (int j = 0; j < m; j++)
				mat[i][j] = matrix[i][j];
	}
```

```
	/**
	 * הפונקציה עושה חיפוש שלם על כל תתי המלבנים הקיימים במטריצה הנתונה
	 * @return the sum of the maximum sub Matrix
	 */
	public int maxMatrix() {
		int maxSum = 0;
		int tempSum = 0;
		int theI = 0;
		int theJ = 0;
		int theP = 0;
		int theQ = 0;

		for (int i = 0; i < n; i++) {
			for (int j = 0; j < m; j++) {
				for (int p = i; p < n; p++) {
					for (int q = j; q < m; q++) {
						tempSum = subSum(i,j,p,q);
						if (tempSum > maxSum) {
							maxSum = tempSum;
							theI = i;
							theJ = j;
							theP = p;
							theQ = q;
						}
					}
				}
			}
		}
		System.out.println("the max sum is: " + maxSum);
		System.out.println(" FROM: [i,j] -> ["+theI+","+theJ+"]");
		System.out.println(" TO:   [p,q] -> ["+theP+","+theQ+"]");
		return maxSum;
	}

	/**
	 * הפונקציה מחשבת את סכום תת הקטע הנתון על ידי 4 אינדקסים (2 נקודות על המטריצה)
	 * @param I the start index (rows)
	 * @param J the start index (cols)
	 * @param P the end index (rows)
	 * @param Q the end index (cols)
	 * @return the sum of the sub matrix
	 */
	public int subSum(int I, int J, int P, int Q) {
		int sum = 0;
		for (int i = I; i <= P; i++) {
			for (int j = J; j <= Q; j++) {
				sum = sum + mat[i][j];
			}
		}
		return sum;
	}
}
```

המשך שיעור 9 - נשפר את הסיבוכיות:

<u>שלב שני</u>: סיבוכיות O(n^4):

פרו' לויט הסביר שבשביל לרדת בסיבוכיות - ניתן להמשיך באותו כיוון של הפתרון הקודם, או לשנות לגמרי את האלגוריתם. כרגע נחשוב באותו כיוון, ולכן נחפש חוקיות מסוימת:

2	3
1	4

נאמר ויש לי מטריצה המכילה 4 תתי מטריצות. נניח כי הסכום הכולל של המטריצה ידוע, אני יכול למצוא את תת מטריצה "3" בעזרת מספר פעולות פשוטות, וכפי שיוסבר להלן.

ניתן לראות כי לכל תתי המטריצות - הנקודה המשותפת: הן מתחילות מ: [0,0] מלבד תת המטריצה "3". בקלות אפשר לחשב את תת המטריצות: [1->3], [1->2], [1->1]

כל מה שנותר זה לעשות את הפעולה הבאה: sum - [1->2] - [1->3] + [1->1], ומצאנו את "3"

כיצד חוקיות זו תועיל לנו? נשתמש במטריצה עזר, נסביר את הדברים בעזרת דוגמא: נתונה המטריצה A:

נחשב על פי החוקיות הנ"ל ונכניס את הערכים למטריצת עזר S: (נתחיל בפינה השמאלית למטה, שים לב, דוגמא מילאתי רק חלק מהמטריצה, אך בפועל היא מלאה לגמרי)

A	1	2	3	4
1	10	-1	5	10
2	-1	-5	10	-1
3	3	20	-5	5
4	-20	-5	10	-3
5	5	10	-2	8

S	1	2	3	4
1	-3			
2	-13			
3	-12	13	16	
4	-15	-10	-2	
5	5	15	13	21

חישוב כזה לא מועיל לנו, שהרי הוא לא חוסך בסיבוכיות. אלא שבשביל שלא נצטרך למלא את המטריצה ע"י חישובים של סכום - יש למצוא את החוקיות, ניתן לשים לב כי סכום תת המטריצה עד לנקודה x היא:

x=a+b-c+A(x)

a	x
c	b

ובצורה פורמאלית יותר: (כאן רשמתי את הנוסחא ע"פ מילוי רגיל של המטריצה ולא כמו בתאוריה)

```
helpMat[i][j] = mat[i][j]+helpMat[i-1][j]+helpMat[i][j-1]-helpMat[i-1][j-1];
```

בפועל - נמלא במטריצת עזר את השורה והעמודה הראשונה, ורק לאחר מכן - על פי החוקיות את שאר האיברים.

לאחר שיש בידנו את מטריצת העזר - אנחנו יכולים לחסוך 2 לולאות בחישוב סכום תתי המלבנים בעזרת הנוסחא הבאה, שהעלות שלה היא: O(n):

```
sum(i,j,p,q) = helpMat[p][q] - helpMat[i-1][q] - helpMat[p][j-1] + helpMat[i-1][j-1]
```

סך הסיבוכיות: O(n^2*m^2 + m*n).

חשוב לשים לב, בפונקציית sum ישנם מקרי קצה הדורשים טיפול (3 מקרי קצה), וראה במימוש.

```
public class shiur11_maxMatrix_n4 {

	int mat[][];
	int helpMat[][];
	int n,m;

	// הבנאי זהה לבנאי של המימוש הקודם
	public shiur11_maxMatrix_n4(int matrix[][]) {
		n = matrix.length;
		m = matrix[0].length;

		mat = new int[n][m];
		helpMat = new int[n][m];
		for (int i = 0; i < n; i++)
			for (int j = 0; j < m; j++)
				mat[i][j] = matrix[i][j];
	}
```

```
    /** פונקציה הממלאת את מטריצת העזר לפי התאוריה שנלמדה בכיתה */
    public void fillHelpMat() {
        helpMat[0][0] = mat[0][0];
        // מילוי שורה ראשונה
        for (int i = 1; i < n; i++) helpMat[i][0] = helpMat[i-1][0] + mat[i][0];
        // מילוי עמודה ראשונה
        for (int j = 1; j < m; j++) helpMat[0][j] = helpMat[0][j-1] + mat[0][j];
        for (int i = 1; i < n; i++) {
            for (int j = 1; j < m; j++) { // מילוי השורה על פי החוקיות שנלמדה בתאוריה
                helpMat[i][j] = mat[i][j] + helpMat[i-1][j]
                                          + helpMat[i][j-1] - helpMat[i-1][j-1];
            }
        }
    }
    /**
     * המימוש זה למימוש של הקוד הקודם, רק שכאן הפונקציה הפנימית "זולה" בהרבה
     * @return the sum of the maximum sub Matrix
     */
    public int maxMatrixOn4() {
        int maxSum = 0;
        int tempSum = 0;
        int theI = 0; int theJ = 0;
        int theP = 0; int theQ = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                for (int p = i; p < n; p++) {
                    for (int q = j; q < m; q++) {
                        tempSum = getSubSum(i,j,p,q); // O(1)
                        if (tempSum>maxSum) {
                            maxSum = tempSum;
                            theI = i;
                            theJ = j;
                            theP = p;
                            theQ = q;
                        }
                    }
                }
            }
        }
        System.out.println("From: [i,j] -> ["+theI+","+theJ+"]");
        System.out.println("To:   [p,q] -> ["+theP+","+theQ+"]");
        System.out.println("the maxSum = " + maxSum);
        return maxSum;
    }

    /**
     * @param I the start index (rows)
     * @param J the start index (cols)
     * @param P the end index (rows)
     * @param Q the end index (cols)
     * @return the sum of the sub matrix
     */
    public int getSubSum(int i, int j, int p, int q) {
        if (i==0 && j==0) return helpMat[p][q];
        if (i==0 && j>0)  return (helpMat[p][q] - helpMat[p][j-1]);
        if (i>0 && j==0) return (helpMat[p][q] - helpMat[i-1][q]);
        else return (helpMat[p][q] - helpMat[i-1][q] -
                                          helpMat[p][j-1] + helpMat[i-1][j-1]);
    }
}
```

<u>שלב שלישי</u>: סיבוכיות O(n^3):

כפי שאמרנו בשיעור הקודם, בשביל להגיע לסיבוכיות כזו אנחנו צריכים לשנות את כל האלגוריתם לגמרי.
הקלט כמובן יישאר אותו קלט (מטריצה, מה שאומר O(n^2)), אך שיטת העבודה תהיה בעזרת Best.
השאלה המתבקשת: איך אפשר לעבוד עם אלגוריתם זה במטריצה.

במהלך השיעור הראה פרו' לויט כמה אופציות שלא משפרות את האלגוריתם אלא אפ' גורמות לו לעבוד לאט יותר ממה שראינו עד כה. המסקנה הייתה אלגוריתם משולב: SuperBest המשתמש בפעולות קבועות (כפי שראינו באלגוריתם האחרון, ו-Best שלמדנו עד כה, כך שבסופו של דבר הסיבוכיות תהיה: $O\left(\frac{m^2}{2}\right) * O(n)$.
כדרך אגב, פרו' לויט העיר שאם יש מרחק גדול בין אורך ורוחב השורה, הרי שעדיף לבחור את הקצר יותר להיות $\frac{m^2}{2}$ מה שיגרום לאלגוריתם לרוץ ב: $O(n)$ בקירוב (ניתן לעשות תנאי מקדים).
גם כאן שיפר פרו' לויט את האלגוריתם במעט (הסיבוכיות נשארה אותה סיבוכיות), נסביר כאן את המסקנה:

גם כאן נשתמש במטריצת עזר, אלא שמטריצה זו תהיה בעלת עמודה נוספת (עמודת אפסים). בבניית המטריצה אנחנו נרוץ בתוך 2 לולאות ונכניס את הערכים באופן הבא:

```
for (int i=0; i<n; i++) {
     for (int j=1; j<m; j++) {
          matAns[i][j] = mat[i][j-1] + matAns[i][j-1];
     }
}
```

כיצד מטריצה זו תועיל לנו בחישוב? בשביל להבין את הדברים נחזור מעט בחומר אל Best רגיל:
נאמר ויש לנו את המערך הבא:

1	2	3	4	5
1	2	10	-5	8

כאשר למדנו את התאוריה ראינו שישנה אופציה להשתמש במטריצה (אופציה ב' - הכנסת המערך לאלכסון הראשי, וחישוב משולש עליון בעזרת חוקיות מסוימת). כעת ניקח את הרעיון שהובא שם, אך נשפר אותו - אנחנו ננסה למצוא את תת הקטע הארוך ביותר בעזרת השורה הראשונה בלבד, ז"א:

1	3	13	8	16

מה השורה הזו מייצגת? היא מייצגת את הסכום מהאיבר הראשון ועד לאיבר מסוים במערך, למשל: הסכום מהאיבר הראשון ועד לאבר השלישי הינו 13. כל זה טוב ויפה ביחס לאיבר הראשון דווקא, אך השאלה היא - כיצד נוכל להגיע לשאר הסכומים? (אלו שלא מתחילים מהאיבר הראשון) אלא אפשר לשים לב שבקלות ניתן להגיע לכל סכום, כיצד? $\sum_{k=i}^{j} a_k = S_j - S_{i-1}$. נמחיש זאת בעזרת דוגמא: אם אני רוצה לדעת מהו הסכום של האיבר השלישי והרביעי, נציב בנוסחא את האיברים של המערך החדש שיצרנו:

$$\sum_{k=3}^{4} a_k = S_4 - S_{3-1} = 8 - 3 = 5$$

כעת מובן מה עושה מטריצת העזר דלעיל. חשוב לשים לב: הצורך בעמודה נוספת של אפסים - למנוע בעיה של חריגה באיבר הראשון (ובעצם יוצא שהעמודה השניה היא העמודה הראשונה פחות אפס).

לאחר שמטריצת העזר מוכנה נשתמש בה באופן הבא: אנחנו ניצור מערך עזר (כמספר העמודות שבמטריצה), ואז נרוץ בשלושה לולאות: 2 לולאות ירוצו על העמודות, ולולאה פנימית שתרוץ על השורה ותיצור מערך באופן הבא:

```
for (int jb=0; jb<m; jb++) {
     for (int je=jb+1; je<m; je++) {
          for (int i=0; i<n; i++) {
               v[i] = mat[i][je] - mat[i][jb];
          }
     }
}
```

מהלך כזה בעצם ייצור לנו את כל הסכומים של תתי המלבנים הקיימים במטריצה, וכפי שהראנו לעיל על שורה בודדת.

בשביל להבין את הדברים מעט יותר טוב ניתן דוגמא מתחילת האלגוריתם ועד לשלב זה:
נאמר וקיבלנו את המטריצה הבאה, ניצור מטריצת עזר לפי החוקיות שדיברנו עליה לעיל:

A	1	2	3
1	10	-1	5
2	-1	-5	7

S	1	2	3	4
1	0	10	9	14
2	0	-1	-6	1

כעת נרוץ פעם אחת על הלולאות:

```
for (int jb=1; jb<=4; jb++) {
     for (int je=2; je<=4; je++) {
          for (int i=1; i<=2; i++) {
               v[1] = mat[1][2] - mat[1][1]; -> 10
               v[2] = mat[2][2] - mat[2][1]; -> -1
```

מה בעצם קיבלנו? ווקטור שמכיל את הסכום של כל אחד מתתי המלבנים הראשונים. (ז"א: כאשר נרוץ באופן הזה על כל המטריצה, נמצא את סכומם של כל תת המלבנים).

כל מה שנותר לעשות זה להפעיל על כל מערך Best רגיל, ולשמור בכל פעם את הסכום הגבוה ביותר והאינדקסים, חשוב לשים לב, הבסט יופעל לאחר סיום הלולאה הפנימית (ז"א: לאחר סיום הווקטור).
מהם האינדקסים של הסכום הגבוה ביותר? שימו לב:

```
startI = findBest.beginMax;
startJ = jb;
endI = findBest.end-1;
endJ = je-1;
```

אנחנו בעצם משתמשים באינדקסים שמתקבלים ע"י הפעלת בסט רגיל - זוהי בעצם נקודת ההתחלה והסיום של אינדקס השורות, ובשביל למצוא את אינדקס העמודות - הרי שהוא כבר נתון לנו ע"י 2 הלולאות הראשונות (jb=Jbegin, je=jEnd).

מימוש החלק השלישי:

```
public class maxMatrixOn3 {

	int mat[][];
	int helpMat[][];
	int n,m;

	public maxMatrixOn3(int firstMat[][]) {
		mat = firstMat;
		n = mat.length;
		m = mat[0].length;
		helpMat = new int [n][m+1]; // חשוב לזכור ולהגדיר את גודל העמודה פלוס אחד
	}

	/** מילוי מטריצת העזר על פי החוקיות שלמדנו בתיאוריה */
	public void fillHelpMatrix() {
		for (int i = 0; i < n; i++) {
			for (int j = 1; j < m+1; j++) {
				helpMat[i][j] = mat[i][j-1] + helpMat[i][j-1];
			}
		}
	}

	/**
	 * שילוב של מספר פעולות פשוטות על מטריצה עם בסט רגיל
	 * @return the sum of the maximum sub Matrix
	 */
	public int getMaxMatrixOn3() {
		int maxSum = 0;
		int startI = 0;
		int startJ = 0;
		int endI = 0;
		int endJ = 0;

		int v[] = new int[n];
		Best SuperBest;


		for (int jb = 0; jb < m+1; jb++) {
			for (int je = jb+1; je < m+1; je++) {
				for (int i = 0; i < n; i++) {
					v[i] = helpMat[i][je]-helpMat[i][jb];
				}
				SuperBest = new Best(v);
				SuperBest.getBest();

				if (SuperBest.maxSum>maxSum) {
					maxSum = SuperBest.maxSum;
					startI = SuperBest.beginMax;
					endI = SuperBest.end;
					startJ = jb;
					endJ = je-1; // מורידים אחד משום שאנחנו רוצים את המיקום במטריצה המקורית
				}
			}
		}
		System.out.println("the maxi sum is "+maxSum);
		System.out.println("[i1,j1] -> ["+startI+","+startJ+"]");
		System.out.println("[i2,j2] -> ["+endI+","+endJ+"]");
		return maxSum;
	}
}
```

בעיה מס' 6 - מציאת מרכז, רדיוס וקוטר בעץ

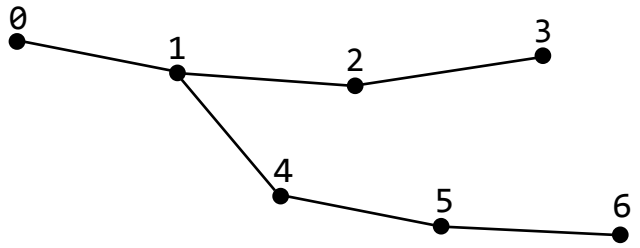
בגלל חוסר זמן, ובגלל שבשיעור דיברנו על תאוריה שבסופו של דבר נשתמש בה במימוש עצמו - ההסבר יהיה קצר יותר והדגש יהיה על הבנת הקוד (נתון נוסף: הקוד לא מומש בשיעור התרגול).

חשוב לזכור - עץ זהו גרף קשיר ללא מעגלים, ז"א: אפשר להגיע מכל קודקוד לכל קודקוד אחר.

מהו רדיוס? רדיוס זהו המרחק המינימאלי מבין המרחקים המקסימאליים (מרחק מקסימאלי - המרחק הגדול ביותר שקיים מקודקוד נתון לקודקוד אחר בגרף).
מרכז - זהו הקודקוד ממנו יוצא הרדיוס, חשוב לשים לב - בעץ ייתכן והמרכז יהיה בכל 2 קודקודים (ישפיע על המימוש ועיין לקמן).
הקוטר - מתחלק ל-2 מקרים:

א. 2 רדיוסים כאשר מדובר על מרכז אחד
ב. כאשר מדובר על 2 מרכזים הקוטר הוא 2 רדיוסים פחות 1.

נעבור לדבר על המימוש: אנחנו נשתמש בשיטת "השריפה":
בכל פעם "נשרוף" את העלים של העץ עד שנגיע למרכז העץ, בשביל להבין את הדברים נשתמש בדוגמא שתלווה אותנו גם שנכנס לתוך הקוד:



דוגמא: תחילה נשרוף את העלים: 0,3,6
פעם שניה: 2,5.
פעם שלישית כבר לא תתבצע משום שהגענו למצב שנשארו פחות משלושה קודקודים.

אם כן, הגענו לתשובה:
המרכז הוא: [1,4]
הרדיוס הוא: 3
הקוטר הוא: 2*3-1=5

מדוע לאחר שמחקנו את קודקוד 0 לא נמחק גם את קודקוד 1? הסיבה לכך שקודקוד 1 אינו עלה, ולכן אינו נכנס לסבב השרפות.

```
     input:              helpArr
vertex 0 -> [1]            [1]
vertex 1 -> [0, 2, 4]      [3]
vertex 2 -> [1, 3]         [2]
vertex 3 -> [2]            [1]
vertex 4 -> [1, 5]         [2]
vertex 5 -> [4, 6]         [2]
vertex 6 -> [5]            [1]
```

הקלט של התוכנית (inPut) הינו מערך של ArrayList, כאשר המערך עצמו אלו הקודקודים, ולכל קודקוד יש ArrayList של הקודקודים המחוברים אליו (בדומה לקלט של BFS\DFS), בדוגמא שלנו הקלט יהיה ->

כעת נסביר איך האלגוריתם עובד בסופו של דבר, ומהן השריפות: היה אפשר לבוא ולמחוק את הקודקוד שיש לו רק שכן אחד. אך הדבר היה מסתבך בהמשך הדרך, כי הינו צריכים לחפש את הקודקוד שעתיד להימחק בכל שאר ה-ArrayList. לכן אנחנו לא נמחוק בפועל, אלא נשתמש במערך עזר (helpArr) שגודלו יהיה כמספר הקודקודים, כל איבר שבו - יהיה בהתאמה דרגת הקודקוד. כאשר נרצה למחוק קודקוד - אנחנו בעצם נשנה את מערך העזר בלבד, ובפועל הקודקוד לא ימחק.

האלגוריתם מתחלק ל-2 חלקים: בניית מעטפת לפתרון, שימוש במעטפת.

נדבר תחילה על בניית המעטפת: נבנה ArrayList נוסף שהוא ייצג את העלים של העץ (leaves).
כפי שהובא לעיל - נבנה מערך עזר לשמירת הדרגה של כל קודקוד בהתאמה (helpArr). כעת נרוץ בלולאה ונכניס את הנתונים:

```
for (int i = 0; i<n; i++){
     helpArr[i] = tree[i].size();
     if (helpArr[i] == 1){
          leaves.add(i);
     }
}
```

עד כאן שלב ההכנה. נעבור לשלב הביצוע:

נרוץ בלולאת **while** עד להגעה למרכז (כפי שאמרנו בתיאוריה - מרכז בעץ יכול להיות לכל היותר בגודל 2), ז"א: **while** (n > 2), כאשר בכל פעם "נשרוף" קודקוד והמשתנה n יקטן בהתאמה.
בתוך הלולאה ניצור ArrayList זמני, שעליו נשמור את העלים ה"עתידיים", לפני שנסיים את הלולאה - נעדכן את ה-ArrayList של העלים בארייליסט הזמני.
כעת נרוץ בלולאה על הארייליסט של העלים ה"עכשוויים", נעתיק למשתנה עזר עלה עלה ונעשה את הפעולות הבאות: נגדיר את דרגתו של העלה כאפס (במקום 1), ואז נחפש את השכנים של אותו קודקוד שהדרגה שלהם אינה 0 - ולהם נוריד את הדרגה באחד. אם מצאנו קודקוד שדרגתו 1 לאחר ההורדה - נכניס אותו ל-ArrayList של העלים ה"עתידיים".
חשוב לזכור - לאחר סיום הלולאה הפנימית יש להוריד את n באחד.
נקודה נוספת: לפני שניכנס ללולאה פעם נוספת - נגדיל את המשתנה radius ב-1.

לאחר סיום החלק הזה יש לבדוק מה גודל המרכז, ובהתאמה לשנות את הרדיוס והקוטר, ז"א:

```
if (leaves.size()==2){
     radius++;
     diameter = radius*2 - 1;
}
else diameter = radius*2;
```

<u>מימוש בעיה מס' 6</u>:

```
import java.util.ArrayList;

public class FireTree {

       ArrayList <Integer> tree[];
       int n; // מספר הקודקודים בגרף
       int degree[];// מערך עזר לשמירה על דרגות הקודקודים בהתאמה
       int radius;
       int diameter;

       public FireTree(ArrayList <Integer> AL[]){
              tree = AL;
              n = AL.length;
              degree = new int[n];
              radius = 0;
              diameter = 0;
       }
```

```
    /**
     * הפונקציה בונה תחילה את המעטפת ואז מתחילה לעבוד על הנתונים
     * @return the Diameter of the tree
     */
    public int findDiameter() {
        // הגדרת "אריי-ליסט" לשמירה על העלים של העץ, המילוי יעשה בלולאה לקמן
        ArrayList <Integer> leaves = new ArrayList<Integer>();

        for (int i = 0; i < n; i++) {
            degree[i] = tree[i].size();
            if (degree[i]==1) leaves.add(i);
        }

        int leaf = 0;
        int vertex = 0;

        while (n>2) {
            // שים לב: הגדרת ה"עלים העתידיים" חייב להתבצע בתוך הלולאה
            ArrayList<Integer> futureLeaves = new ArrayList<Integer>();
            for (int i = 0; i < leaves.size(); i++) {
                leaf = leaves.get(i);
                degree[leaf] = 0;
                for (int j = 0; j < tree[leaf].size(); j++) {
                    vertex = tree[leaf].get(j);
                    if (degree[vertex]>0) { // טיפול בשכנים שאינם בדרגת אפס
                        degree[vertex]--;
                        // בניית "אריי ליטס" של העלים העתידיים
                        if (degree[vertex] == 1) futureLeaves.add(vertex);
                    }
                }
                n--; // בכל סבב של לולאה אנחנו בעצם מורידים עלה
            }
            radius++; // על כל "שריפה" אנחנו מגדילים את הרדיוס
            leaves = futureLeaves; // הכלה של העלים העתידיים לתוך העלים העכשוויים
        }

        if (leaves.size()==2) { // כאן פועלים לפי ההסבר בתאוריה
            radius++;
            diameter = 2*radius-1;
        }
        else diameter = 2*radius;
        System.out.println("the radius is: " + radius + ", the diameter is: " +
                                        diameter + ", and center is: " + leaves);
        return diameter;
    }
}
```