קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

בעיית השוקולד:

בעיה:

חפיסת שוקולד מוגדרת בקטע [1,n] של המספרים הטבעיים, בכל שלב חותכים את הקטע ע"י האינדקס $i\in [1, n-1]$ לשני תתי קטעים, כלומר [1,i] ו – [i+1,n], העלות של חיתוך אחד היא $i\cdot (n-i)$.

השלב האחרון הוא המצב בו כל הקטעים הם באורך 1.

פונקציית המטרה – סכום כל החיתוכים, $\frac{n\cdot(n-1)}{2}$.

המטרה היא למצוא רצף של חתכים שמקטין את פונקציית המטרה.

בעיית הבקבוקים:

בעיה:

נתונים שני מיכלים בשני גדלים שונים, מיכל A בגודל m ליטר, מיכל B בגודל n ליטר.

מקבלים גרף מכוון שהקודקוד שלו הוא מצב המיכלים (a,b) והצלע מחברת את (a_1, b_1) עם קודקוד (a_2, b_2), אם אפשר מיד לעבור ממצב (a_1, b_1) למצב (a_2, b_2), אז נבנה גרף שמיוצג ע"י מטריצה שמכילה את כל המעברים המידיים, נוכל להניח בה"כ שm<n.

מספר הקודקודים בגרף הוא – $(n+1)\cdot(m+1)$.

הפעולות המותרות :

1. מילוי מיכל A – (m, b).
2. מילוי מיכל B – (a, n).
3. ריקון מיכל A – (0, b).
4. ריקון מיכל B – (a, 0).
5. מזיגה מ-A ל-B – ((a + b - min(n, a + b), (min(n, a + b)).
6. מזיגה מ-B ל-A – ((min(a + b, m), a + b – min(a + b, m)).

מימוש:

נבנה מטריצה בוליאנית ריבועית בגודל $(n+1)\cdot(m+1)$, המכילה false, כאשר כל עמודה וכל שורה מייצגת את הקשר שבין 2 קודקודים, כלומר אם יש קשר יסומן המיקום במטריצה בtrue, אם אין קשר הערך יישאר false.

נשאיר את האלכסון להיות false (קשר בין קודקוד לעצמו).

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, כלומר – $(m+1)\cdot i + j$, כאשר i מייצג את גובה הנוזל במיכל A, j מייצג את גובה הנוזל במיכל B ו-m מייצג את הגודל המירבי של אחד המיכלים.

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

קוד:

```
bottleProblem(b1, b2):
  size ← (b1+1)*(b2+1)
  create mat[size,size]
  max ← max(b1,b2)

  for i ← 0 to b1 do:
     for j ← 0 to b2 do:
        node ←  getNode(max,i,j)

        mat[node,getNode(max,i,0)] ← 1
        mat[node,getNode(max,0,j)] ← 1

        mat[node,getNode(max,b1,j)] ← 1
        mat[node,getNode(max,i,b2)] ← 1

        mat[node,getNode(max , i+j - min(i+j,b2) , min(i+j,b2))] ← 1
        mat[node,getNode(max , min(i+j,b1) , i+j - min(i+j,b1))] ← 1
      end-for
  end-for

  for i ← 0 to size do:
     mat[i,i] ← 0
     end-for

end-bottleProblem()

getNode(max, i, j):
  return (max+1)*i + j
end-getNode()
```

אלגוריתם Floyd-Warshall:

1. **הסבר על האלגוריתם:**

הרעיון שעומד מאחורי האלגוריתם הזה הוא לבדוק האם קיים קודקוד כלשהו בגרף שיכול לחבר לנו בין 2 קודקודים אחרים וליצור מסלול.
האלגוריתם עובר על כל הקודקודים ובודק – אם קיים מסלול בין i ל-k ובין k ל-j, אז קיים מסלול בין i ל-j.
הבעיה באלגוריתם היא שאם כבר יש מסלול בין i ל-j אז k יכול להפריע, כי אולי אין צלע בין i ל- k או בין k ל-j, כדי שבעיה זו לא תהיה, נוסיף תנאי 'או' שלא יפגע במה שקיים בתא, כלומר קודקוד לא יכול לפגוע אלא רק לעזור.

<u>קוד:</u>

```
FloydWarshall(mat[N,N]):

for k ⇐ 0 to N do:
  for i ⇐ 0 to N do:
    for j ⇐ 0 to N do:
      if mat[i,j] > mat[i,k]+mat[k,j] then:
        mat[i,j] ⇐ mat[i,k] + mat[k,j]
      end-if
    end-for
  end-for
end-for
```

ברגע שנריץ את האלגוריתם על מטריצת השכנויות שלנו, נוכל לדעת האם קיים מסלול בין כל 2 קודקודים.
האם יש מסלול בין קודקוד לעצמו? כן, או, שניתן אפילו לומר שיש מסלול באורך 0.
v_1 לא שבן של עצמו אבל קיים מסלול ממנו אל עצמו.

2. **<u>האם יש מסלול בין v_i לבין v_j?</u>**
כדי לבדוק את זה נפעיל את האלגוריתם FW על הגרף ונוכל לגשת לתא המתאים במטריצה ולראות אם יש שם מספר שגדול מ-0 – סיבוביות O(1).

3. **<u>האם הגרף קשיר? (רק גרפים לא מכוונים)</u>**
   - כאשר כל המטריצה תהיה מלאה בtrue זה אומר שהגרף קשיר, מכיוון שמכל קודקוד אפשר להגיע לכל קודקוד – **סיבוכיות** $\boldsymbol{O(n^2)}$.
   - **ב-O(n)** נעבור על השורה הראשונה ונבדוק שהכל שם גדול מ-1, כלומר שהם שכנים, זה נכון מכיוון שמקודקוד v_1 אפשר להגיע לכל הקודקודים האחרים, למשל מקודקוד v_i נרצה להגיע לקודקוד v_j, אז נוכל להגיע אליו דרך קודקוד v_1.

<u>קוד:</u>

```
IsGraphConnected(mat[N,N]):

for k ⇐ 0 to N do:
  for i ⇐ 0 to N do:
    for j ⇐ 0 to N do:
      if mat[i,j] > mat[i,k]+mat[k,j] then:
        mat[i,j] ⇐ mat[i,k] + mat[k,j]
      end-if

for i ⇐ 0 to N do:
  if mat[0,i] = ∞ then:
    return False
  end-if

return True
```

4. **<u>כמה רכיבי קשירות יש? (רק גרפים לא מכוונים)</u>**

כדי לבדוק כמה רכיבי קשירות יש, נוכל פשוט לעבור על המטריצה, הבעיה היא שלא תמיד נקבל מטריצה מסודרת, לכן נוכל לייצר מערך עזר, נעבור על המערך וכל פעם שנראה 0 נדע שעוד לא הגענו לקודקוד הזה. נבדוק במטריצת השכנויות מי השכנים שלו ונסמן במערך.
נוסיף counter שבודק לנו מה מס' רכיבי הקשירות ונעלה אותו כל פעם. כך נעבור על כל המערך ,רק על השורות הרלוונטיות במטריצה.
נקבל מערך שמכיל מידע על מספר רכיבי הקשירות שיש לנו בגרף.

<u>קוד:</u>

```
getNumberOfComponents(mat[N,N]):

for k ⇐ 0 to N do:
  for i ⇐ 0 to N do:
    for j ⇐ 0 to N do:
      if mat[i,j] > mat[i,k]+mat[k,j] then:
         mat[i,j] ⇐ mat[i,k] + mat[k,j]
      end-if
end-3 for loops

create comp[N]
counter ⇐ 0

for i ⇐ 0 to N do:
  comp[i] ⇐ 0
end-for


for i ⇐ 0 to N do:
  if comp[i] = 0 then:
    counter ⇐ counter + 1
    comp[i] ⇐ counter
    for j ⇐ 0 to N do:
      if mat[i,j] != ∞ then:
        comp[j] ⇐ counter
      end-if
    end-for
  end-if
end-for

return counter
```

5. **<u>מיהם הקודקודים ברכיבי הקשירות:</u>**

כדי לדעת איזה קודקוד מראה מה רכיב הקשירות, נוכל פשוט לעבור על המערך ולראות מי נמצא באיזה רכיב.

<u>קוד:</u>

```
GetVertexInEachComponents(b[][])
		size = length(b)
		counter = 0
		create components[size]
		for i = 0 to length(components) do:
				if components[i] = 0 then:
						counter++
						components[i] = counter
```

```
                        for j = i + 1 to length(components) then:
                                if components[j] = 0 and b[i][j] = True then:
                                        components[j] = counter
                                end-if
                        end-for
                end-if
        end-for
        str[counter]
        for i = 0 to length(str)
                str[i] = ""
        end-for
        for i = 0 to length(components) do:
                str[components[i]-1] += i + "\t"
        end-for
        return str
end-GetVertexInEachComponents()
```

6. **<u>מציאת מטריצת מסלולים:</u>**

מקודם מצאנו מטריצת שכנויות שמראה לנו בין אלו קודקודים יש מסלולים, אך איך נדע מהם המסלולים?

נייצר מטריצת עזר של מחרוזות ובכל תא נרשום איך מגיעים מקודקוד לקודקוד.

זה לאו דווקא המסלול הקצר ביותר, אבל לפחות נותן לנו פתרון.

ניקח את FW ונמיר את התנאי האחרון בו לתנאי שיעזור לנו לשרשר מסלולים.

<u>קוד:</u>

```
FWGetPathes(mat)
        len = length(mat)
        path[len][len]
        dist[len][len]
        for i = 0 to len do:
                for j = 0 to len do:
                        if dist[i][j] != ∞ then:
                                path[i][j] = i + " → " + j
                        else path[i][j] = ""
                        end-if-else
                end-for
        end-for
                for i = 0 to len do:
                        for j = 0 to len do:
                                if dist[i][j] != ∞ then:
                                        if dist[i][j] > dist[i][k] + dist[k][j] then:
                                                dist[i][j] = dist[i][k] + dist[k][j]
                                                path[i][j] = path[i][k] + path[k][j]
                                        end-of
                                end-of
                        end-for
                end-for
        end-for
        return path
end-FWGetPathes
```

7. **<u>בקבוקים – האם קיים מסלול בין *(i, j)* לבין *(k, l)*?</u>**

נחזור לבעיית הבקבוקים. עכשיו, אם נרצה לדעת האם קיים מסלול בין מצב מסוים למצב אחר נוכל להמיר מ-(i, j) ל-k, נפעיל FW להמיר חזרה מ-k ל-i ו-j ולראות האם קיים מסלול או לא. אם

ממש נרצה את המסלול נפעיל את האלגוריתם שמחשב לנו את המסלולים עצמם ונחזיר את הפתרון.

8. **<u>סידור המטריצה:</u>**
נחליף עמודות ושורות ונהפוך את המטריצה למטריצה יפה יותר.

<u>גרף עם משקלים על הצלעות:</u>

1. **<u>מציאת מטריצת המרחקים:</u>**
קיבלנו גרף של כל המרחקים והמשקלים שיש בין 2 צלעות.
בעזרת FW נוכל לשאול האם קיים מסלול בין הקודקודים, אך גם נוכל לשאול מה המסלול הקצר ביותר בין כל שני קודקודים, ולעדכן אותו יש לנו כבר מסלול בין i לבין j, נבדוק אם הקודקוד k יכול לעזור לנו לעדכן את המסלול שיהיה קצר יותר.

<u>קוד:</u>

```
FWGetDistance(mat)
    for k = 0 to len do:
        for i = 0 to len do:
            for j = 0 to len do:
                if dist[i][k] != ∞ then:
                    if dist[i][j] > dist[i][k] + dist[k][j] then:
                        dist[i][j] = dist[i][k] + dist[k][j]
                    end-if
                end-of
            end-for
        end-for
    end-for
end-FWGetDistance()
```

2. **<u>מציאת מטריצת המסלולים:</u>**
נייצר מטריצת עזר של מחרוזות ובכל תא נרשום איך מגיעים מקודקוד לקודקוד.
ניקח את FW ונמיר את התנאי האחרון בו לתנאי שיעזור לנו לשרשר מסלולים.

<u>קוד:</u>

```
public static void floyd_warshall(int[][] mat, String[][] path) {
        int size = mat.length;
        for (int k = 0; k < size; k++) {
                for (int i = 0; i < size; i++) {
                        for (int j = 0; j < size; j++) {
                                if (mat[i][k]!= inf &&  mat[k][j]!= inf)
                                        if (mat[i][k]+mat[k][j]<mat[i][j]){
                                                mat[i][j]=mat[i][k]+mat[k][j];
                                                path[i][j] = path[i][k] + "->"
                                                        + path[k][j];
                                        }
                        }
                }
        }
}
private static void FixPath(String[][] path) {
        for (int i = 0; i < path.length; i++) {
```

```
				for (int j = 0; j < path.length; j++) {
					path[i][j] += "->"+(j+1);
				}
		}
}
private static void PrintAllPath(String[][] path) {
		for (int i = 0; i < path.length; i++) {
				for (int j = 0; j < path.length; j++) {
					System.out.print((i+1)+"->"+(j+1)+":\t");
					if (path[i][j] != null)
						System.out.println(path[i][j]);
					else
						System.out.println("NO..");
				}
		}
		System.out.println("-------------------------");
}
```

3. <u>**חזרה וסיכום של בעיית הבקבוקים:**</u>

נוכל לשפר עוד קצת את הבעיה בכך שנמצא את **המסלול הקצר ביותר** ממצב למצב.
אם נרצה לדעת האם קיים מסלול בין מצב מסוים למצב אחר נוכל להמיר מ-(i, j) ל-k, נפעיל את FW החדש, נמיר חזרה מ-k ל-i ו-j ונראה האם קיים מסלול או לא. אם ממש נרצה את המסלול נפעיל את האלגוריתם שמחשב לנו את המסלולים עצמם ונחזיר את הפתרון.

מימוש אלגוריתם:

1. נייצר מטריצת בקבוקים.
2. נפעיל את אלגוריתם FW (המחזירה סטרינגים) על המטריצה.
3. נמיר את מספרי מצב הבקבוקים **הנוכחי** למספר יחיד (קודקוד המוצא המתאים במטריצה).
   ונמיר את מספרי מצב הבקבוקים של **היעד** למספר יחיד (קודקוד היעד המתאים במטריצה).
4. עם המספרים שקיבלנו נשלוף את הסטרינג המתאים מתוך מטריצת הסטרינגים.
5. הסטרינג מייצג מסלול קודקודים (כל קודקוד מספר יחיד) ולכן נרצה להמיר אותו למסלול של בקבוקים (כל קודקוד יהפוך לזוג מספרים המייצגים את מצב הבקבוקים)
   לכן, נפצל את הסטרינג ע"י המפריד "->" וכל מספר קודקוד נמיר לייצוג של בקבוקים.

<u>**קוד:**</u>

```
FWWeightForBottle(mat, n)
	len = length(mat)
	pathMat[len][len]
	for i = 0 to len do:
		ai =  i/ (n + 1)
		bi = i % (n + 1)
		for j = 0 to len do:
			aj = j / (n + 1)
			bj = j % (n + 1)
			if mat[i][j] !=  ∞ then:
				pathMat[i][j] = “” + ai + bi + " → " + aj + bj + “ ”
			else pathMat[i][j] = “”
			end-if-else
```

```
                end-for
        end-for
        for k = 0 to len do:
                for i = 0 to len do:
                        for j = 0 to len do:
                                if mat[i][k] != ∞ then:
                                        if mat[i][j] > mat[i][k] + mat[k][j] then:
                                                mat[i][j] = mat[i][k] + mat[k][j]
                                                pathMat[i][j] = pathMat[i][k] + pathMat[k][j]
                                        end-if
                                end-if
                        end-for
                end-for
        end-for
        return pathMat
end- FWWeightForBottle()
```

גרף עם משקלים על קודקודים:

נתון מערך של משקלים המוגדרים על קודקודי הגרף ומטריצה בוליאנית המגדירה את צלעות הגרף, נקבל מטריצה שמייצגת את המרחקים הקצרים ביותר בין קודקודי הגרף.

מימוש האלגוריתם:

1. נבנה מטריצה שמייצגת את המשקלים על צלעות הגרף לפי הנוסחא הבאה: $weight(a,b) = f_a + f_b$, נוסחא זו מגדירה משקל על צלע שמחברת בין קודקוד a לקודקוד b.
2. נפעיל עליה אלגוריתם פלויד וורשל, נקבל את מטריצת העלויות הקטנות ביותר h, כאשר המשקלים מוגדרים על צלעות הגרף.
3. נמיר את העלויות המוגדרות על הצלעות לעלויות המוגדרות על הקודקודים ע"י הנוסחא: $d(i,j) = \frac{h(i,j)+f_i+f_j}{2}$, כאשר *h(i,j)* זה עלות המעבר בין קודקוד *i* לקודקוד *j* לפי צלעות, *d(i,j)* זה עלות המעבר בין קודקוד *i* לקודקוד *j* לפי הקודקודים.

קוד:

```
FWNodesWeights(mat[N,N], nodes[N]):
for i ⇐ 0 to N do:
  for j ⇐ 0 to N do:
    if i=j then:
      mat[i,j] ⇐ nodes[i]
    else if mat[i,j] then:
      mat[i,j] ⇐  nodes[i]+nodes[j]
    end-if
  end-for
end-for

for k ⇐ 0 to N do:
        for i ⇐ 0 to N do:
                for j ⇐ 0 to N do then:
                        mat[i,j] ⇐ min(mat[i,j] , mat[i,k]+mat[k,j])
                end-for
        end-for
end-for

for i ⇐ 0 to N do:
        for j ⇐ 0 to N do:
                if i≠j then:
                        mat[i,j] ⇐ (nodes[i]+mat[i,j]+nodes[j])/2
                end-if
        end-for
end-for
```

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

1. **מציאת מטריצת המרחקים:**
כדי למצוא את מטריצת המרחקים לפי המשקלים של הקודקודים, ננסה להמיר את בעיה זו לבעיית המשקלים על הצלעות.
נבצע המרה - נמיר את מטריצת השכנויות במטריצת שכנויות של צלעות - ניקח כל i ו-j שיש ביניהם true במטריצה ונכניס לתא את החיבור של העלויות שלהם מהמערך.
נבצע FW שיתן לנו את המרחקים הקצרים בין כל קודקוד לכל קודקוד אחר.
הפתרון יוצא לא נכון כי יש קודקודים שנכללים פעמיים, אז בשלב זה נתקן, אז או שניקח את כל הכפולים ונוריד אותם או אם כל האמצעיים מופיעים פעמיים, אז נוסיף עוד פעם אחת את הקודקודים בקצוות ומחלק ב-2.

2. **מציאת מטריצת המסלולים:**
בדיוק כמו מקודם: נייצר מטריצת עזר של מחרוזות ובכל תא נרשום איך מגיעים מקודקוד לקודקוד. ניקח את FW ונמיר את התנאי האחרון בו לתנאי שיעזור לנו לשרשר מסלולים.

איך נסמן אינסוף?
פתרון זמני: אם אנחנו מסתמכים על זה שקלט הערכים שלנו נע בגבול נמוך יחסית, נוכל להגדיר את אינסוף כמיליון.
אחרת, אם נוסיף תנאי לפני הנוסחא ב-FW שאומר: אם אף אחד מהם לא 1- או אף אחד לא MAXVALUE אז תבצע את התנאי.

3. **החזרת דוגמא למסלול הקצר ביותר עבור כל i וכל j:**
להדפיס את מטריצת המסלולים שחישבנו מקודם.

משקלים על הקודקודים והצלעות:

נתון גרף G=(V, E) לא מכוון, קשיר וממושקל, בו לכל קודקוד ולכל צלע יש משקל.
נסמן את עלות המעבר דרך קודקוד v_i ב-$f(v_i)$, כלומר המשקל של v_i ונסמן ב-$w(v_{i+1}, v_i)$ את משקל הצלע $\{v_{i+1}, v_i\}$.

- נגדיר את עלות המעבר מקודקוד v_i לקודקוד v_{i+1j} – עלות = $f(v_i) + w\left(v_i, v_{i+1j}\right) + f(v_{i+1)}$.
- כדי לחשב את מטריצת המסלולים הקצרים ביותר בגרף, נגדיר את עלות הצלע (a, b) באופן הבא- $p(a,b) = f(a) + 2w(a,b) + f(b)$.
- עלות המעבר בין קודקוד i לבין קודקוד j לפי קודקודים וצלעות היא – $d(i,j) = \frac{h(i,j)+f(i)+f(j)}{2}$.

קוד:

```
FWNodesEdgesWeights(mat[N,N], nodes[N]):
for i←0 to N do:
        for j←0 to N do:
      if i=j then:
        mat[i,j] ← nodes[i]
      else-if mat[i,j] then:
        mat[i,j] ← nodes[i] + 2*mat[i,j] + nodes[j]
      end-if
  end-for
end-for

for k← 0 to N do:
    for i← 0 to N do:
      for j← 0 to N do:
              mat[i,j] ← min(mat[i,j], mat[i,k]+mat[k,j]
      end-for
    end-for
end-for

for i← 0 to N do:
        for j← 0 to N do:
                if ij then:
                        mat[i,j] ← (nodes[i]+mat[i,j]+nodes[j])/2
                end-if
        end-for
end-for
```

משקלים שליליים בגרף מכוון ולא מכוון:

מעגל שלילי – מעגל שסכום משקלי צלעותיו שלילי.
בגרף לא מכוון, מספיק שתהיה צלע אחת שלילית כדי שהמעגל יהיה שלילי, לכן מרחק קצר ביותר בין קודקודים כבר לא רלוונטי , כי ניתן להגיע עד $-\infty$.
הסיבוכיות למציאת משקל שלילי בגרף היא O(m) כאשר m הוא מספר הצלעות בגרף.
בגרף מכוון גם יכול להיווצר מעגל שלילי, לא נוכל להפעיל FW כדי לגלות את זה, כי FW לא נותן את התוצאה הנכונה, אבל הוא מצביע על זה שיש מעגל שלילי בגרף, נוכל לדעת את זה מהאלכסון במטריצה שמראה לנו את המרחק מקודקוד לעצמו, כלומר אם נראה שם שהמרחק שלילי, נוכל לדעת שיש לנו מעגל שלילי בגרף.

קוד – לגרף לא מכוון:

```
FWCheckNegativeCycle(mat[N,N]):
for i← 0 to N do:
        for j← 0 to N do:
                if mat[i,j] < 0 then:
                        return True
                end-if
return False
```

קוד – לגרף מכוון:

```
FWCheckNegativeCycleDirected(mat[N,N]):

for k← 0 to N do:
  for i← 0 to N do:
    for j← 0 to N do:
           mat[i,j] ← min(mat[i,j], mat[i,k]+mat[k,j])
    end-for
  end-for
end-for

for i← 0 to N do:
       if mat[i,i] < 0 then:
              return True
       end-if
end-for

return False
```

מציאת תת מערך עם סכום מקסימלי – Best:

בעיה:

בהינתן מערך בעל n, נרצה לבדוק מהו תת המערך שסכום איבריו הוא הגדול ביותר.
נחלק למקרים:

1. אם כל התאים חיוביים אז ניקח את כולם.
2. אם יש תאים שליליים, נבדוק אם הם עוזרים לנו וכדאי לקחת אותם.
   נבדוק מתי כדאי לנו לקחת את השליליים ומתי לא:

אם המספר השלילי משאיר אותנו בסכום חיובי, כדאי לקחת אותו כי הוא מוסיף לנו להמשך. אבל אם הוא מוריד אותנו לסכום שלילי זה כבר לא משתלם כי אולי בהמשך יהיה משהו טוב יותר שעלול להיפגע.

פתרון:

נבין את התובנה הבאה: כדי לקבל את הסכום הגדול ביותר נוכל לקחת את הסכום הכולל ולהפחית ממנו את הסכום המינימלי, אבל זה יהיה הגדול ביותר בצורה מעגלית. יכול להיות שבצורה סטנדרטית יהיה לנו סכום גדול יותר, לכן נצטרך לבצע מקסימום בין התוצאה המעגלית לתוצאה הסטנדרטית, בעת כדי למצוא את התוצאה המעגלית או שנשנה את אלגוריתם Best שימצא לנו את הסכום המינימלי, או שנכפיל ב-(1-) את כל האיברים במערך ונקבל את המערך ההופכי לו, כשנפעיל על המערך ההופכי את Best נמצא את הסכום המקסימלי שמייצג לנו את הסכום המינימלי במערך המקורי, הסכום המקסימלי יהיה הסכום הכולל של כל האיברים במערך פחות הסכום המינימלי במערך, כלומר נבדוק זאת כך .Max(Best(Arr), S-(-Best(-Arr)))

סיבוכיות – O(n), ביצענו פעמיים את Best.

<u>קוד – Best ליניארי:</u>

```
bestLinear(arr[N]):
  sum ← -∞
  temp_sum ← 0
  start_index ← 0
  temp_start ← 0
  end_index ← 0

  for i← 0 to N do:
      temp_sum ← temp_sum + arr[i]
        if temp_sum > sum then:
          sum ← temp_sum
          end_index ← i
          start_index ← temp_start
        end-if

    if temp_sum < 0 then:
      temp_start ← i + 1
      temp_sum ← 0
    end-if
  end-for

  create solution[3]
  solution[0] ← sum
  solution[1] ← start_index
  solution[2] ← end_index

return solution
```

קוד – מעגלי:

```
bestCycle(arr[N]):
  create neg_arr[N]
  sum ← 0

  for i← 0 to N do:
    sum ← sum + arr[i]
    neg_arr[i] ← arr[i]*(-1)
  end-for

  create negative[3] ← bestLinear(neg_arr)
  create regular[3] ← bestLinear(arr)
  cycle_sum ← sum - (-negative[0])

  if regular[0]  cycle_sum then:
    return regular
  end-if

  create solution[3]
  solution[0] ← cycle_sum
  solution[1] ← (negative[2]+1) modulo N
  solution[2] ← negative[1] - 1
return solution
```

בעיית תחנות הדלק:

בעיה:

נתונות n תחנות במעגל סגור, נגדיר a_i – כמות הדלק בליטרים שיש בתחנה i, נגדיר b_i – כמות הדלק הנדרשת לאוטו כדי להגיע מתחנה i לתחנה i+1.

האוטו צריך לעבור על כל התחנות ולחזור לתחנת ההתחלה, כלומר לעשות סיבוב שלם, האוטו יכול לתדלק בכל תחנה.

האוטו מתחיל לנסוע עם מיכל ריק, הנהג צריך לבחור להתחיל מתחנה שיהיה לו מספיק דלק לחזור לתחנה שהתחיל ממנה.

פתרון:

נקבל שני מערכים B,A כאשר A – שומר את הערכים של התחנות, B – שומר את הערכים של העלויות מכל תחנה לכל תחנה.

קודם כול כמות הדלק בכול התחנות צריכה להיות גדולה או שווה לכמות הדלק הנדרשת כדי לסגור את המעגל, כלומר $\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} b_i$ זה התנאי ההכרחי.

ניצור מערך נוסף *C* שהמשמעות שלו תהיה *A-B* כלומר, אם אפשרי להגיע מתחנה אחת לאחרת.

ברור שצריך להתייחס להפרשים בין כמות הדלק שיש בתחנה לכמות הדלק הנדרשת למעבר לתחנה הבאה – $c_i = a_i - b_i$ והתנאי ההכרחי שאפשר לכתוב בעזרת c_i הוא – $\sum_{i=1}^{n} c_i \geq 0$,

נחפש את תת המערך עם הסכום המקסימלי כדי נוכל להמשיך בדרך, נפעיל *Best* בצורה מעגיל וכך נדע מה הסכום המקסימלי ומאיזה אינדקס כדאי להתחיל, כדי לדעת את זה נכפיל ב-*(-1)* את האיברים שקיבלנו במערך *C* במעגלי, מפעיל עליו *Best* ונקבל במערך ההופכי ל-*C* את הסכום המקסימלי שבמערך המקורי של *C* הוא הסכום המינימלי, כעת נחשב את הסכום הכולל פחות הסכום המינימלי שקיבלנו, לכן סה"כ נקבל – *Max(Best(c), Sum-(-Best(-C)))*.

אם במערך מעגלי $\{c_i\}$ ניתן למצוא קטע בעל סכום גדול יותר, אז סכום האיברים שמחוץ לתת קטע זה יהיה קטן יותר.

המסקנה: אם האוטו יתחיל לזוז מתחילת הקטע בעל הסכום הגדול ביותר, יהיה לו מספיק דלק כדי

לעשות סיבוב שלם. כמות הדלק שיישאר בטנק שווה בדיוק ל – $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \sum_{i=1}^{n} b_i$.

<u>קוד:</u>

```
gasStationProblem(nodes[N], edges[N]):
  sum ⇐ 0
  create combine[N]

  for i⇐ 0 to N do:
    combine[i] ⇐ nodes[i]-edges[i]
    sum ⇐ sum + combine[i]
  end-for

  if sum < 0 then:
    return empty-array
  end-if

return bestCycle(combine)
```

```
bestLinear(arr[N]):
  sum ⇐ -∞
  temp_sum ⇐ 0
  start_index ⇐ 0
  temp_start ⇐ 0
  end_index ⇐ 0

  for i⇐ 0 to N do:
      temp_sum ⇐ temp_sum + arr[i]
        if temp_sum > sum then:
          sum ⇐ temp_sum
          end_index ⇐ i
          start_index ⇐ temp_start
        end-if

    if temp_sum < 0 then:
      temp_start ⇐ i + 1
      temp_sum ⇐ 0
    end-if
  end-for

  create solution[3]
  solution[0] ⇐ sum
  solution[1] ⇐ start_index
  solution[2] ⇐ end_index

return solution
```

```
bestCycle(arr[N]):
  create neg_arr[N]
  sum ⇐ 0

  for i⇐ 0 to N do:
    sum ⇐ sum + arr[i]
    neg_arr[i] ⇐ arr[i]*(-1)
  end-for

  create negative[3] ⇐ bestLinear(neg_arr)
  create regular[3] ⇐ bestLinear(arr)
  cycle_sum ⇐ sum - (-negative[0])

  if regular[0]  cycle_sum then:
    return regular
  end-if

  create solution[3]
  solution[0] ⇐ cycle_sum
  solution[1] ⇐ (negative[2]+1) modulo N
  solution[2] ⇐ negative[1] - 1
return solution
```

<u>Dijkstra:</u>

דייקסטרה הוא אלגוריתם חמדן, המטרה שלו היא לחשב את המסלול הקצר ביותר מקודקוד אחד לכל שאר הקודקודים (בניגוד ל-FW שמחשב לנו את המסלול הקצר ביותר מכל הקודקודים לכל הקודקודים).

האלגוריתם עובד כך:

1. לכל צלע בין 2 קודקודים יש משקל.
2. נאתחל את הערך של קודקוד ההתחלה ל-0 ואת שאר הקודקודים ל-∞.
3. נתחיל לחשב את המסלול הזול ביותר באופן הבא:
   1. נכניס את קודקוד ההתחלה והשכנים שלו לתור, נעדכן את המרחק מקודקוד ההתחלה לכל שכן ונוציא את קודקוד ההתחלה מהתור.
   2. נמשיך כך עד שנוציא את כל הקודקודים מהתור.
   3. כעת, הערך שיש בכל קודקוד הוא המסלול הקצר ביותר מקודקוד ההתחלה אליו.

4. אם נרצה לשחזר את המסלול עצמו, נוכל לשמור עבור כל קודקוד מי האבא שלו, ניצור מערך של אבות ובכל פעם שנאתחל קודקוד מסויים בערך חדש, נרשום במערך מאיפה הגענו אליו.
5. אם נרצה למצוא מסלול מקודקוד ההתחלה לקודקוד מסויים נשנה את תנאי העצירה שיעצור ברגע שאנחנו מסיימים לחשב את המרחק של קודקוד זה והוא יוצא מהתור.

סיבוכיות האלגוריתם תלויה במבנה הנתונים השומר את הקודקודים שטרם ביקרנו בהם:
1. אם מדובר ברשימה או במערך, הסיבוכיות היא – $O(V^2+|E|)=O(V^2)$.
2. אם משתמשים בערימה בינארית, אז סיבוכיות הריצה משתפרת – $O\big((E+V)\cdot log_2V\big)=O(|E|\cdot log_2V)$.

<u>קוד:</u>

```
Dijkstra(G, src):
  create PriorityQueue Q ← ∅
  create dist[|V[G]|]
  create prev[|V[G]|]
  create visited[|V[G]|]

  for each v∈V[G] do:
      dist[v] ← ∞
      prev[v] ← NIL
      visited[v] ← false
  end-for

  dist[src] ← 0
  Enqueue(Q, src)

  while Q is not empty do:
      u ← Dequeue(Q)
      for each v∈Adj[u] do:
          if not visited[v] then:
              if dist[v] > dist[u] + weight(v,u) then:
                  dist[v] ←  dist[u] + weight(v,u)
                  prev[v] ← u
                  DecreaseKey(Q, v)
              end-if
          end-if
      end-for
      visited[u] ← true
  end-while
end-Dijkstra
```

<u>דייקסטרה דו כיוונית:</u>

דייקסטרה דו כיוונית היא דייקסטרה שמתקדמים בה גם מקודקוד ההתחלה וגם מקודוקד הסיום בו זמנית.
הרעיון מאחורי זה הוא שבדייקסטרה בכל איטרציה אנחנו מתקדמים בשכבות של הבנים וכך נוצרים מעין מעגלים מסביב לקודקוד.

נגדיר גרף רוורס – הגדרה:
גרף רוורס G^R עבור גרף G, הוא גרף עם קבוצת קודקודים V וקבוצה של הרוורס של הצלעות E^R, כך שלכל צלע $(u,v)\in E$ קיימת צלע $(v,u)\in E^R$ ולהיפך.

כעת נוכל להשתמש בזה לאלגוריתם דייקסטרה דו כיוונית כך:
1. נבנה G^R.
2. נריץ את האלגוריתם של דייקסטרה מקודקוד s בגרף G ומקודקוד t בגרף G^R.

3. נחליף לסירוגין בין שלבי דייקסטרה ב-G וב-G^R.
4. נעצור בשלב שקיים קודקוד v כלשהו שטופל (יצא מהתור) גם ב-G וגם ב-G^R.
5. לבסוף, נחשב את המסלול הקצר ביותר בין קודקוד s לקודקוד t.

נרצה לבדוק כעת אם בהכרח המסלול הקצר ביותר עובר דרך v זה:

1. נסמן ב-$d[u]$ את המרחק באלגוריתם דייקסטרה החל מקודקוד s בגרף G ונסמן ב-$d^R[u]$ את המרחק באלגוריתם דייקסטרה החל מקודקוד t בגרף G^R.
2. סימנו את זה כך, מכיוון שלאחר שקודקוד מסויים v טופל גם בגרף G וגם בגרף G^R, אז קיים מסלול צר ביותר מקודקוד s לקודקוד t שעובר דרך קודקוד כלשהו u שטופל בגרף G או בגרף G^R או בשניהם והמשוואה הבאה מתקיימת $d(s,t) = d[u] + d^R[u]$, כלומר לאחר שמצאנו קודקוד v שטופל בשני האלגוריתמים נוכל למצוא קודקוד u שגם טופל כבר בשניהם וגם נמצא על המסלול הקצר ביותר, עבור אותו קודקוד u המשוואה מתקיימת.

נוכיח כעת את סעיף 2:

1. נניח שקיימים לנו שני מעגלים שנוצרים מריצת האלגוריתם וקודקוד v הוא נקודת המפגש ביניהם.
2. נוכיח קודם כל שאם קיים קודקוד u שלא טופל ע"י אף אחד מהאלגוריתמים ודרכו עובר המסלול הקצר ביותר, אז קיים לפחות קודקוד נוסף שהמסלול הקצר ביותר עובר דרכו – קודקוד v.
3. אם באמת יש קודקוד u שלא טופל בכלל, אז קיים קודקוד v נוסף שטופל ע"י האלגוריתמים והמסלול הקצר ביותר עובר דרכו.
4. בהכרח המרחק שיש ב-v נכון, כי אחרי שהוא טופל הוא התקבע למסלול הזול ביותר בשני הכיוונים.
5. כאשר אין קודקודים שלא טופלו בשני האלגוריתמים חוץ מנקודת המפגש בין שני המעגלים ב-v, נניח שיש מסלול זול בין קודקוד s לקודקוד t ונניח שהקודקוד האחרון שטופל באלגוריתם הישיר הוא u.
הקודקוד הבא במסלול הקצר ביותר יהיה w כי הוא לא טופל באלגוריתם הישיר (כי u האחרון שטופל בו) ומצד שני לא יכול להיות שהוא לא טופל כלל כי הוא על המסלול הקצר ביותר, לכן קודקוד w חייב להיות בתוך המעגל של קודקוד t והוא בהכרח טופל ע"י האלגוריתם ההפוך, לכן המרחק כרגע בין קודקוד s לבין קודקוד t הוא –
$d(s,t) = d[u] + l(u,w) + d^R[w]$, $d(s,t) = d[u] + d^R[u]$, בשתי המשוואות יש $d[u]$, לכן כדי להשוות אותן נצטרך להשוות בין $l(u,w) + d^R[w]$ לבין $d^R[u]$, נשים לב ש-$d(u,t) \leq d^R[u]$, כי בכל רגע של הפעלת האלגוריתם המרחקים רק משתפרים.
מצד שני, קודקוד w טופל ע"י האלגוריתם ההפוך ולכן הוא "הקל" על הצלע (u,w) שיוצאת ממנו ולכן $l(u,w) + d^R[w] \geq d^R[u]$, קיבלנו ש-$l(u,w) + d^R[w]$, הוא המרחק הקצר ביותר בין קודקוד u לקודקוד t שעבור דרך קודקוד w, לכן לפי כלל הסנדוויץ' $d^R[u]$ שווה למרחק הקצר ביותר בין קודקוד u לקודקוד t.

□

נממש כעת את האלגוריתם:

1. נבצע החלפות בין האלגוריתמים בכל איטרציה עד שנפגוש קודקוד שטופל בשניהם.
2. ניקח את כל הקודקודים שטופלו ונחפש מינימום בסכום של האלגוריתם הישיר וההפוך, אותו מספר מינימלי יהיה המרחק הזול ביותר.

3. בדי לייצר את המסלול עצמו נשרשר את המסלול מקודקוד s לאותו קודקוד מינימלי והרוורס של המסלול מקודקוד t לאותו קודקוד.

סיבוכיות – אותה סיבוכיות כמו של דייקסטרה רגיל, מבחינת זמן ריצה זה יכול להיות פי 2 יותר מהיר, מבחינת זיכרון זה תופס פי 2 אחסון, כי צריך לייצר גם את הגרף ההפוך.

הערה – דייקסטרה לא יכול להתמודד עם משקלים שליליים.

<u>קוד:</u>

```
BidirectionalDijkstra(G,RG,src,dest):
    create PriorityQueue Q
    create PriorityQueue RQ
    create dist[|V(G)|]
    create rdist[|V(G)|]
    create prev[|V(G)|]
    create rprev[|V(G)|]
    create visited[|V(G)|]
    create rvisited[|V(G)|]

    for each v∈V[G] do:
    dist[v] ⇐ ∞
    rdist[v] ⇐ ∞
    prev[v] ⇐ NIL
    rprev[v] ⇐ NIL
    visited[v] ⇐ false
    rvisited[v] ⇐ false
    end-for

    dist[src] ⇐ 0
    rdist[dest] ⇐ 0
    Enqueue(Q,src)
    Enqueue(RQ,dest)

    while Q is not empty and RQ is not empty do:

          u ⇐ Dequeue(Q)
          for each v∈Adj[u] do:
              if not visited[v] then:
                    if dist[v] > dist[u] + weight(u,v) then:
                          dist[v] ⇐ dist[u] + weight(u,v)
                          prev[v] ⇐ u
```

1

```
                          DecreaseKey(Q,v)
                    end-if
              end-if
          end-for

          visited[u] ⇐ true

          ru ⇐ Dequeue(RQ)
          for each v∈Adj[ru] do:
              if not rvisited[v] then:
                    if rdist[v] > rdist[ru] + weight(ru,v) then:
                      rdist[v] ⇐ rdist[ru] + weight(ru,v)
                      rprev[v] ⇐ ru
                      DecreaseKey(RQ,v)
                    end-if
              end-if
          end-for

          rvisited[ru] ⇐ true

          if visited[ru] and rvisited[ru] then:
            break
          end-if

    end-while

    minDist ⇐ ∞
    for each v∈V[G] do:
          if (visited[v] or rvisited[v])
            and (dist[v] ≠ ∞ and rdist[v] ≠ ∞)
            and (minDist > dist[v]+rdist[v])
            then:
                minDist⇐ dist[v]+rdist[v]
        end-if
    end-for

    return minDist

end-BidirectionalDijkstra
```

2

<u>מציאת תת מטריצה בעלת סכום מקסימלי:</u>

בעיה:

נתונה מטריצה בגודל $m \cdot n$, מהי תת המטריצה שסכום איבריה הוא הגדול ביותר?

פתרון – *Super Best*:

נשתמש במערך עזר ונעבור על העמודות של המטריצה בעזרת שני משתנים *i,j* כאשר בהתחלה *i=0* ו-*j=0*, לאחר מכן נקדם את *j* באחד ו-*i=0* נמשיך ככה עד ש-*j* יעבור על כל השורה, לאחר ש-*J* סיים לעבור אז נקדם את *i* באחד ו-*j* יתחיל מאיפה ש-*i* נמצא וכך נעבור על הכל.
את העמודה הראשונה אנחנו מכניסים לתוך המערך עזר שלנו, נפעיל את *Best* על הערכים שהכנסנו למערך ואת סכום תת המערך המקסימלי שקיבלנו, ה*Best* יחזיר לנו את מיקום האיבר הראשון בתת מערך העזר **(הוא יחזיר את המיקום של אותו איבר במטריצה המקורית)** ואת המיקום של האיבר האחרון בתת המערך, נשמור את אותו סכום שקיבלנו במשתנה, נשמור גם את המיקום של כל אחד מהם במשתנים (את מספר העמודה של כל אחד מהם ואת מספר

השורה של כל אחד מהם), מגדירים את המשתנים של העמודות והשורות כי ככה אנחנו מקבלים את תת המטריצה עם הסכום המקסימלי.
כעת נמשיך לעמודה השנייה וכל איבר בעמודה זו נסכום עם האיבר שנמצא באותו מקום במערך העזר שלנו, נפעיל את ה*Best* נעדכן את הסכום המקסימלי, נעדכן את מספר העמודה והשורה של האיבר הראשון בתת המערך ושל האיבר האחרון בתת המערך, וכן הלאה עד שנסכום את כל האיברים שנמצאים במטריצה.

סיבוכיות – $O(n^3)$, זה רק גודל, הסיבוכיות האמיתית היא $O(n\cdot m^2)$.
איך נדע מתי צריך לרוץ על העמודות ומתי על השורות?
לפי הקלט שנכניס, אם מספר העמודות של המטריצה היא גדול יותר ממספר השורות שלה, אז נרוץ על השורות ואם מספר השורות שלה גדול יותר ממספר העמודות שלה אז נרוץ על העמודות.

<u>קוד:</u>

```
superBest(mat[N,M]):
  create preSum[N,M+1]
  for i← 0 to N do:
    for j← 0 to M do:
      preSum[i,j+1] ← preSum[i,j] + mat[i,j]
    end-for
  end-for

  sum ← 0
  ii ← 0
  jj ← 0
  kk ← 0
  ll ← 0

  for i← 0 to M do:
    for j← i to M do:
      create arr[N]
      for k← 0 to N do:
        arr[k] ← preSum[k,j+1]-preSum[k,i]
      end-for

      create best[3] ← bestLinear(arr)
      if best[0] > sum then:
        sum ← best[0]
        ii ← best[1]
        jj ← i
        kk ← best[2]
        ll ← j
      end-if
    end-for
  end-for

return {sum, ii, jj, kk, ll}
```

<u>*BFS*:</u>

רעיון האלגוריתם:
אלגוריתם *BFS* סורק לפי שכבות, בהינתן גרף *G=(V,E)* וקודקוד מסויים *s* שמשמש במקור, חיפוש לרוחב בוחן בשיטתיות את הקשתות ב-*G*, כדי "לגלות" כל קודקוד שאפשר להגיע אליו מקודקוד *s*, אלגוריתם *BFS* מחשב את המרחק (מספר הקשתות המינימלי) מקודקוד *s* לכל הקודקודים שאפשר להגיע אליהם מ-*s*.
BFS פועל גם על גרפים מכוונים וגם על לא מכוונים.

כיצד האלגוריתם עובד?

1. האלגוריתם מחלק את הקודקודים ל-3 צבעים:
   - לבן – הקודקוד עוד לא התגלה.
   - אפור – הקודקוד התגלה אבל עוד לא טיפלנו בו.
   - שחור – הקודקוד טופל וסיימנו איתו.
2. הוא בונה עץ רוחב שהשורש שלו הוא קודקוד המקור *s* ולפי השכבות והמרחקים ממנו, הוא בונה את שאר הגרף, לכן קודקוד במרחק *k* יתגלה לפני קודקוד במרחק *k+1*.
3. כל קודקוד שמגלה את שכניו הופך להיות האבא שלהם, מכיוון שכל קודקוד מתגלה פעם אחת אז יש לו רק אבא אחד.
4. המסלול שיוצא מקודקוד *s* לקודקוד *v* כלשהו בעץ רוחב, מקביל למסלול הקצר ביותר בין קודקוד *s* לקודקוד *v* בגרף.

<u>קוד:</u>

```
BFS(G,src):
    create Queue Q
    create dist[|V(G)|]
    create prev[|V(G)|]
    create color[|V(G)|]

    for each v∈V[G] do:
        dist[v] ← ∞
        prev[v] ← NIL
        color[v] ← WHITE
    end-for

    dist[src] ← 0
    color[src] ← GRAY
    Enqueue(Q,src)

    while Q is not empty do:
        u ← Dequeue(Q)
        for each v∈Adj[u] do:
            if color[v] = WHITE then:
                color[v] ← GRAY
                dist[v] ← dist[u] + 1
                prev[v] ← u
                Enqueue(Q,v)
            end-if
        end-for
        color[u] ← BLACK
    end-while
    return (dist, prev)

end-BFS
```

סיבוכיות – $O(|V+E|)$, כלומר:

- צובעים את כל הקודקודים רק פעם אחת בלבן, לכן מובטח לנו שכל קודקוד יכנס פעם אחת לתור ויצא רק פעם אחת מהתור – הוצאה והכנסה זה ב-$O(1)$, לכן נקבל שהסיבוכיות של הצביעה היא $O(|V|)$.
- בבכל איטרציה עוברים על השכנים של אותו קודקוד, לכן הסיבוכיות תהיה $O(\deg(V))$, במילים אחרות הסיבוכיות היא $O(|E|)$.

מסקנות מ-BFS - לאחר הרצת האלגוריתם BFS(G,s) בגרף G=(V,E), ניתן לראות ש:

1. כל רכיב הקשירות של קודקוד s התגלה.
2. מערך d מכיל את המסלולים הקצרים ביותר מקודקוד s לכל קודקוד אחר.
3. דרך מערך f ניתן לשחזר את המסלול הקצר ביותר.

כדי להדפיס את המסלול נשתמש באלגוריתם הדפסה רקורסיבי שידפיס את הקודקודים שעוברים בדרכם במסלול, סיבוכיות – O(|V|).

1. איך נוכל לדעת שגרף הוא קשיר?
אם אחרי שנריץ את BFS כל הקודקודים יהיה צבועים בשחור, או אם יש רק null אחד במערך f, או אם כל המספרים במערך d הם סופיים.

<u>קוד:</u>

```
checkConnectivity(G):
    create color[|V(G)|]
    color ⇐ BFS(G,0)

    for i⇐ 0 in |V(G)| do:
        if color[v] is not BLACK then:
            return false
        end-if
    end-for

    return true
end-checkConnectivity
```

2. איך נוכל לדעת כמה רכיבי קשירות יש?
אם אחרי שנריץ את BFS יהיו עוד קודקודים לבנים, נפעיל את האלגוריתם מחדש על אחד מהקודקודים הלבנים ונספור שזו האיטרציה השנייה שלנו, נמשיך כך עד שכל הקודקודים יהיו שחורים ונדע כמה רכיבי קשירות יש.

<u>קוד:</u>

```
NumberOfComponents(G):
    nextSource ⇐ 0
    counter ⇐ 0
    create HashSet S

    while S size ≠ |V(G)| do:
        counter ⇐ counter + 1
        create dist[|V(G)|] ⇐ BFS(G, nextSource)

        for i⇐ 0 to |V(G)| do:
            if dist[i] ≠ ∞ then:
                Add(S, i)
            else:
                nextSource⇐ i
            end-if-else
        end-for
    end-while

    return counter
end-NumberOfComponents
```

3. איך נוכל לדעת מיהם הקודקודים בכל רכיב קשירות?
נעשה כמו בשאלה 2, רק שבכל איטרציה נסמן או נשמור את הקודקודים שבכל רכיב.

קוטר של גרף:
קוטר בגרף הוא המרחק המקסימלי שקיים בין שני קודקודים בגרף.
הסיבוכיות למציאת קוטר בגרף היא $O(|V+E|)$ – סיבוכיות של אלגוריתם זה זו הסיבוכיות של אלגוריתם שנובע מגיאומטריה במישור.

אלגוריתם שריפה:

מהו עץ?
- גרף קשיר ללא מעגלים.
- גרף קשיר עם n-1 צלעות.
- גרף עם n-1 צלעות ללא מעגלים.

כאשר 2 מההגדרות מתקיימות, אז הגרף הוא בהכרח עץ.

מושגים:
1. מרכז – הנקודה ממנה יוצא הרדיוס.
2. רדיוס – קטע המחבר מרכז לשפת המעגל.
3. קוטר – הקטע הכי ארוך בין 2 קצוות מעגל, בהכרח עובר במרכז המעגל.

מושגים אלו קיימים גם בעצים:
1. קוטר – המרחק המקסימלי בין 2 קודקודים, כדי למצוא את הקוטר מפעילים פעמיים את BFS.
2. רדיוס – $\frac{1}{2}$ מהקוטר.
3. מרכז – אמצע הקוטר.

הערה – אם הקוטר זוגי אז הרדיוס הוא $\frac{1}{2}$ מהקוטר ויש מרכז אחד, אם הקוטר הוא אי זוגי אז הרדיוס הוא $\frac{1}{2}$ מהקוטר בעיגול כלפי מעלה ויש לו שני מרכזים.

כעת נראה את אלגוריתם שריפה שמוצא קוטר, רדיוס ומרכז:
ברגע שמורידים עלים, אנחנו עדיין נשארים עם עץ, מכיוון שלא סגרנו מעגלים ועדיין נשארנו עם גרף קשיר ונשארו עדיין *n-1* צלעות.

רעיון האלגוריתם – "לשרוף" את העלים בכל פעם עד שנישאר רק עם המרכז/ים.

אם הקוטר זוגי:
- יש מרכז אחד.
- הרדיוס הוא כמות השריפות.
- הקוטר הוא פעמיים כמו השריפות.

אם הקוטר אי זוגי:
- יש שני מרכזים.
- הרדיוס הוא כמות השריפות + 1.
- הקוטר הוא פעמיים כמות השריפות – 1.

נממש את אלגוריתם שריפה בעזרת מערך עזר:

- כל תא במערך ייצג לנו את הדרגה של אותו קודקוד, כך המעבר יהיה יותר מהיר וקל.
- אם הערך בתא הוא 1, נדע שמדובר בעלה ונכניס אותו לתור, ברגע ששרפנו עלה, נוריד דרגה אחת מהשכן של אותו עלה.
- נוסיף משתנה counter שיספור לנו כמה עלים עוד לא שרפנו וכשהמשתנה יהיה שווה ל-1 או ל-2 נדע לעצור את השריפה כי הגענו למרכז.

<u>**קוד:**</u>

```
Fire(T):
    create Queue Q ← ∅
    diameter ← 0
    radius ← 0
    centers ← 0

    for each v∈V[T] do:
        if deg(v)=1 then:
            Enqueue(Q,v)
        end-if
    end-for

    nodes ← |V[T]|
    while nodes > 2 do:
        radius ← radius + 1
        leaves ← |Q|
        for i← 0 to leaves do:
            u ← Dequeue(Q)
            v ← {v|v∈Adj(u)}
            Remove(T,u)
            nodes ← nodes - 1
            if deg(v)=1 then:
                Enqueue(Q,v)
            end-if
        end-for
    end-while

    centers ← nodes
    if nodes = 2 then:
            diameter ← radius * 2 + 1
            radius ← radius + 1
    else:
            diameter ← radius * 2
    end-if

    return diameter,radius,centers
end-Fire
```

סיבוכיות – O(|V|).

בניית עץ מרשימת דרגות:

בהינתן רשימת דרגות הקודקודים בגרף מסויים, האם רשימה זו באמת יכולה להיות רשימת דרגות?
האם היא יכולה להיות רשימת דרגות של עץ?

כמה הגדרות לפני שנענה על הבעיה:

- בכל עץ $|E| = |V| - 1$, כלומר כמות הצלעות שווה לכמות הקודקודים פחות 1.
- $\sum_{v \in V} \deg(v) = 2 \cdot |E|$.

מסקנה - $\sum_{v \in V} \deg(v) = 2 \cdot (|V| - 1)$, לכן אנו צריכים לוודא שסכום הדרגות יהיה פעמיים מספר הצלעות.

שתי ההגדרות שראינו לא תמיד מוודא שרשימת דרגות יכולה להיות עץ, לכן רשימת דרגות יכולה להיות עץ כאשר:

- $\sum_{v \in V} \deg(v) = 2 \cdot |E|$.
- כאשר יש לנו מספיק נתונים לכמות הקודקודים.

רעיון האלגוריתם:

משפט – בכל עץ יש לפחות 2 עלים.
לכן ננסה לחבר (כמה שאפשר) עלה לקודקוד שהדרגה שלו גדולה מ-1, כדי שיישארו עלים.
ניקח את הקודקוד הראשון שמייצג עלה ונחבר אותו עם הקודקוד הראשון במערך שהוא לא עלה,
לאחר החיבור נוריד את דרגות הקודקודים האלו, נמשיך כך עד שכל הערכים במערך יהיו 0.

קוד:

```
GenerateTreeByDegrees(deg[N]):

    sum ← 0

    for i ← 0 to N do:
        sum ← sum + deg[i]
    end-for

    if sum ≠ 2*(N − 1) then:
        print "not a tree"
        return empty-array
    end-if

    Sort(deg) // O(N*logN)
    j ← 0

    for i ← 0 to N do:
        if deg[i] > 1 then:
            j ← i
            break
        end-if
    end-for

    create tree[N]
    for i ← 0 to N-2 do:
        tree[i] ← j
        deg[j] ← deg[j] − 1

        if deg[j] = 1 then:
            j ← j + 1
        end-if
    end-for

    tree[N-1] ← N
    return tree

end-GenerateTreeByDegrees
```

סכום המטריצה הגדול ביותר:

נתונה מטריצה שמלאה ב-1 וב-(-1) בצורה רנדומלית, איך נהפוך את המטריצה לבעלת הסכום הגדול ביותר?
ניקח כל שורה וכל עמודה ונכפיל ב-(-1).
הערה – תהליך זה הוא סופי, מכיוון שיש לנו חסם עליון, כלומר לא יכול להיות שהסכום יהיה גדול יותר ממספר התאים במטריצה.

עצים איזומורפיים:

הגדרת עצים איזומורפיים:

גרפים G ו-H הם איזומורפיים אם קיימת פונקציה $f: V(G) \to V(H)$ חח"ע ועל, כך שלכל $u, v \in V(G)$ מספר הקשתות שמקשרות בין u ו-v זהה למספר הקשתות שמקשרות בין f(u) ו-f(v), כלומר שהם מייצגים את אותו גרף רק נראים בצורה אחרת.

עצים עם שורש:

נגדיר מיהו אותו קודקוד עליון שממנו אנחנו מתחילים את החישוב, הרעיון שעומד מאחורי זה הוא שצריך להחליט על סדר בין הבנים של כל קודקוד וכך נוכל להשוות ביניהם, כלומר ננסה שהבנים יהיו ממויינים בסדר כלשהו וכך נוכל להכריע בבעיית איזומורפיזם.
לפני שנחליט על מיון נצטרך לראות איך עוברים על העץ, נסרוק את הגרף לעומק ונשמור את הפעולות שלנו – בכל פעם שנרד נרשום 0 ובכל פעם שנעלה נרשום 1 ולאחר הסריקה נקבל מחרוזת שמתאימה לעץ, אם נצליח למצוא עם נוסף עם אותה מחרוזת נוכל לומר שהם איזומורפיים.

אלגוריתם לעץ עם שורש:

1. נסרוק את הגרף לעומק.
2. כשנגיע לעלה, נגדיר את המחרוזת שלו כ-01.
3. כשנגדיר את המחרוזת של האבא (נגדיר אותה רק אחרי שנסיים עם כל המחרוזות של הבנים שלו) נתחיל אותה ב-0, נסיים אותה ב-1 והאמצע יהיה שרשור ממויין של מחרוזות הבנים.

כשנרצה להשוות בין עצים ולראות אם הם איזומורפיים נפעיל את האלגוריתם על כל אחד מהעצים ונראה האם נקבל מחרוזות שוות.

עצים ללא שורש:

לגבי עצים לא מושרים יש שני פתרונות:
1. נבחר כל פעם קודקוד אחר שיהיה השורש, נפעיל עליו את האלגוריתם עד שנמצא מחרוזות שוות או עד שנעבור על כל הזוגות האפשריים ולא נמצא כלום.
2. אם נבחר קודקוד שיכול להיות שורש אופציונאלי נוכל להפעיל את האלגוריתם רק פעם אחת ממנו ולקבל את התשובה, הקודקוד שיכול להיות שורש יכול להיות המרכז, כל העץ יכול לצאת ממנו ואם נמצא אותו אז נוכל להפעיל את האלגוריתם רק פעם אחת ולקבל תשובה.

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

<u>קוד – עבור עץ ללא שורש:</u>

```
isIsomorphicWithoutRoot(T1, T2):
     create Queue roots1
     create Queue roots2

     roots1 ⇐ Fire(T1)
     roots2 ⇐ Fire(T2)

     code1 ⇐ generateCode(T1, root1) // continue like the rooted-tree algorithm

     while roots2 is not empty do:
          r2 ⇐ Dequeue(roots2)
          code2 ⇐ generateCode(T2, root2) // continue like the rooted-tree algorithm
          if code1 = code2 then:
             return true
          end-if
     end-while

     return false

end-isIsomorphicWithoutRoot

Fire(T):
        create Queue Q ⇐ ∅
        create deg[|V(T)|]

        for each v∈V(T) do:
                deg[v] ⇐ |Adj(v)|
                if deg[v]=1 then:
                        Enqueue(Q,v)
                end-if
        end-for

        nodes ⇐ |V[T]|
        while nodes > 2 do:
                leaves ⇐ |Q|
                for i⇐ 0 to leaves do:
                        u ⇐ Dequeue(Q)
                        nodes ⇐ nodes - 1
                        for each v∈Adj(u) do:
                                deg[v] ⇐ deg[v] - 1
                                if deg[v]=1 then:
                                        Enqueue(Q,v)
                                end-if
                        end-for
                end-for
        end-while

        return Q

end-Fire
```

<u>קוד – עבור עץ מושרש:</u>

```
isIsomorphic(T1, root1, T2, root2):
    code1 ⇐ generateCode(T1, root1)
    code2 ⇐ generateCode(T2, root2)

    if code1 = code2 then:
        return true
    else:
        return false
end-isIsomorphic

generateCode(T, root):
    create traversalCode[|V(T)|]
    create color[|V(T)|]
    for i⇐ 0 to |V(T)| do:
        traversalCode[i] ⇐ "0"
        color[i] ⇐ WHITE
    end-for
    getTraversalCode(T, root1, traversalCode, color)
    return sortCodes(traversalCode)
end-generateCode

getTraversalCode(T, current, tCode[|V(T)|], color[|V(T)|]):
    color[current] ⇐ BLACK

    if deg(current) = 1 then:
        tCode[current] ⇐ "01"
    else:
        for each v ∈ Adj(current) do:
            if color[v] = WHITE then:
                getTraversalCode(T, v, tCode, color)
                tCode[current] ⇐ tCode[current] + tCode[v]
            end-if
        end-for

        tCode[current] ⇐ tCode[current] + "1"
    end-if-else
end-getTraversalCode

sortCodes(tCode[|V(T)|]):
    Sort(tCode) // O(NlogN)
    sortedCode ⇐ ""
    current ⇐ 1
    while current < |V(T)| and tCode[current] ≠ "01" do:
        sortedCode ⇐ sortedCode + tCode[current]
        current ⇐ current + 1
    end-while

    return sortedCode
end-sortCodes
```

<u>קידוד הופמן:</u>

המטרה – לקודד טקסט בשימוש במינימום ביטים ולהצליח לפענח אותו בחזרה בצורה מדוייקת.
הקוד המספק דחיסת נתונים מרבית, כלומר מאחסן את התווים במספר מזערי של סיביות.
השיטה מתבססת על הקצאת אורך הקוד לתווים על פי שכיחותם, כך שתו נפוץ יוצג באמצעות מספר קטן של סיביות. לרוב ניתן לחסוך באמצעות שיטה זו בין 20% ל-90% משטח האחסון.

דרך מימוש העץ – באמצעות שני תורים:
כאשר מערך של תדירויות כבר ממויין בסדר עולה (מקטן לגדול), שימוש בשני תורים נותן סיבוכיות של O(n).

תיאור האלגוריתם:

1. מגדירים שני תורים ריקים Q_1, Q_2.
2. מכניסים את כל העלים ל-Q_1 כך שהעלה הקטן ביותר יימצא בחזית של התור O(n).
3. כל עוד בשני התורים יש יותר מאיבר אחד:
   - מוציאים את שני האיברים הקטנים ביותר m_1, m_2 מהחזית של שני התורים.
   - יוצרים צומת חדש שהצמתים m_1, m_2 הופכים להיות הבנים שלו והערך של הצומת החדש הוא הסכום של שני הבנים שלו ואז מכניסים את הצומת החדש לתור Q_2.
   - וחוזרים לפעולה הראשונה שמוציאים את שני האיברים הקטנים ביותר.
4. הצומת האחרון שנשאר באחד מהתורים הוא ראש העץ וכך סיימנו לבנות את הטבלה.

<u>קוד:</u>

```
Huffman(A):
        N ⇐ |A|
        create Queue Q1
        create Queue Q2

        for i⇐ 0 to N do:
                create Node node
                node.freq ⇐ A[i].freq
                node.char ⇐ A[i].char
                Enqueue(Q1, node)
        end-for

        while |Q1| + |Q2| > 1 do:
                x ⇐ getMin(Q1, Q2)
                y ⇐ getMin(Q1, Q2)

                create Node z
                z.left ⇐ x
                z.right ⇐ y
                z.freq ⇐ x.freq + y.freq

                x.parent ⇐ z
                y.parent ⇐ z

                Enqueue(Q2, z)
        end-while

        if Q1 is empty then:
                root ⇐ Dequeue(Q2)
                return root
        else:
                root ⇐ Dequeue(Q1)
                return root
end-Huffman

getMin(Q1, Q2):
        create Node x

        if Q1 is empty then:
                x ⇐ Dequeue(Q2)
        else if Q2 is empty then:
                x ⇐ Dequeue(Q1)
        else:

            if Q1.head().freq > Q2.head().freq then:
                    x ⇐ Dequeue(Q2)
            else:
                    x ⇐ Dequeue(Q1)
            end-if
        end-if
        return x
end-getMin
```

כמות האיטרציות – n-1.

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

מעגל אוילר:

הגדרה:
יהי גרף $G(V,E)$ לא מכוון.
מסלול P(x,y) נקרא **מסלול אוילר** ב-G אם הוא עובר על כל הצלעות של G, כך שכל צלע מופיעה בו פעם אחת בלבד, בנוסף $x \neq y$, כלומר מתחילים בקודקוד אחד ומסיימים בקודקוד אחר.

סיבוכיות - $O(|E| \cdot |V|)$.

הגדרה:
יהי גרף $G(V,E)$ לא מכוון.
מעגל אוילר ב-G הוא מסלול אוילר סגור, כלומר מסלול שעובר בכל צלעות הגרף פעם אחת בלבד והקודקוד ההתחלתי הוא גם קודקוד הסיום.

הגדרה:
יהי גרף $G(V,E)$ לא מכוון.
גרף אוילריאני הוא גרף המכיל מעגל אוילר.

משפטי זיהוי אוילר בגרפים:
1. בגרף G יש מעגל אוילר (גרף אוילריאני) אמ"מ G קשיר וכל דרגות הגרף זוגיות.
2. בגרף G יש סלול אוילר אמ"מ G גרף קשיר ויש בדיוק 2 קודקודים בעלי דרגות אי זוגיות.

אלגוריתם למציאת מעגל אוילר:
1. בדיקה האם יש מעגל: עוברים על כל קודקוד ובודקים האם דרגתו זוגית, סופרים את אלו עם הדרגה שהיא אי זוגית. אם יש לנו 0 אי זוגיים - יש מעגל, אם יש 2 אי זוגיים - יש מסלול ואחרת אין מעגל ואין מסלול.אם עברנו על כל הקודקודים נעבור לשלב 3, אחרת נוריד את המעגל מהגרף ונחזור לשלב 1.
2. מגדירים מחסנית ורשימה המייצגת את המסלול עצמו.
אם זה מסלול ולא מעגל, מתחילים מקודקוד מדרגה אי זוגית. ואם זה מעגל אז בוחרים קודקוד שרירותית. מכניסים את קודקוד ההתחלה למחסנית.
3. אם יש לו שכן, מכניסים את השכן למחסנית, מוחקים את הצלע וממשיכים עם השכן. עד שנתקעים (אין יותר שכנים).
4. ברגע שנתקעים, מוציאים את ראש המחסנית ומוסיפים למסלול. וממשיכים עם הקודקוד הבא במחסנית. עד שמתרוקנת המחסנית.

<u>קוד:</u>

```
EuilerPathCycle(G=(V,E))
        Stack s
        List path
        count = 0
        start = 1
        for each v in V
                if(G[v].size() % 2 == 1)
                        count++
                        start = v
                end-if
        end-for each
        if(count > 2) return "No path and no cycle"
        s.push(start)
        while(s not empty)
```

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

```
        if(G[s.top()].size() > 0)
            u = G[v].getFirst()
            s.push(u)
            G.remove(v,u)
        end-if
        else
            path.add(s.pop())
        end-else
    return path
end- EuilerPathCycle()
```

עץ פורש מינימלי – פרים:

הגדרה לעץ פורש מינימלי:

עץ המורכב מתת קבוצה של צלעות בגרף קשיר לא מכוון ומקיים את התכונות הבאות:

1. פורש את הגרף, כלומר כולל את כל קודקודי הגרף.
2. מינימלי בסכום משקלי הצלעות שלו שהוא הקטן ביותר האפשרי מכל העצים הפורשים של הגרף.

- האלגוריתם של פרים הוא חמדני שמשמש למציאת עץ פורש מינימלי בגרף ממושקל לא מכוון.
- מגדירים תור עדיפויות ומערך visit[|V|] עבור הקודקודים.
- מתחילים מקודקוד שרירותי s על הגרף .
- מכניסים לתור העדיפויות את כל השכנים שלו עם העדיפות של משקל הצלע שמגיעה אליהם מ s.
- כל עוד התור לא ריק, שולפים את הקודקוד עם העדיפות המינימאלית, מוסיפים את הצלע לעץ (בין הקודקוד הנשלף לאבא שלו) ומכניסים לתור את כל השכנים שלא סיימנו איתם עם עדיפות של משקל הצלע שמגיעה אליהם מאותו קודקוד (אם הם כבר בתור אז רק מעדכנים את העדיפות למינימאלי).

קוד:

```
Prim(G): // O(|E|log|V|)
    create PriorityQueue Q
    create Tree T ⇐ Ø
    create visit[|V(G)|]
    create prev[|V(G)|]
    create minEdge[|V(G)|]

    for each v∈V(G) do: // O(|V|)
        visit[v] ⇐ false
        prev[v] ⇐ NIL
        minEdge[v] ⇐ ∞
    end-for

    minEdge[0] ⇐ 0
    Q.add((0,0)) // (node ID, priority)

    while Q is not empty do: // O(|E|log|V|)
        u ⇐ Q.extractMin()

        if prev[u] ≠ NULL then:
            T.add( (u, prev[u]) )
        end-if

        for each v∈Adj(u) do:
            if not visit[v] then:
                if minEdge[v] > weight(u,v) then:
                    minEdge[v] ⇐ weight(u,v)
                    prev[v] ⇐ u

                    if Q.contains(v) then:
                        Q.decreaseKey( (v, minEdge[v]) )
                    else:
                        Q.add( (v, minEdge[v]) )
                    end-if
                end-if
            end-if
        end-for

        visit[u] ⇐ true
    end-while
    return T
end-Prim
```

סיבוכיות – אתחול מערכים: $O(|V|)$.
לולאה ראשית עוברת על כל קודקוד פעם אחת ואז על כל השכנים שלו: $O(|E|)$.
בכל איטרציה, מכניסים ומעדכנים בתור העדיפויות (במקרה הגרוע) - $O(\log|V|)$.
סיבוכיות כוללת: $O(|V| + |E| \log|V|)$.
סה"כ - $O(|V| + |E|\log|V|)$.

קרוסקל:

אלגוריתם קרוסקל הוא אלגוריתם חמדני לפתרון בעיית מציאת עץ פורש מינימלי בגרף משוקלל לא מכוון.

שלבי האלגוריתם:

1. בונים עץ פורש מינימלי ריק *T*.
2. ממיינים את צלעות הגרף בסדר עולה (מקטן לגדול).
3. מגדירים מספר קבוצות זרות שכל קודקוד בגרף שייך לקבוצה שלו, כלומר מספר קבוצות שווה למספר קודקודי הגרף *|V|*.
4. שולפים צלע *e=(u,v)* בעלת משקל מינימלי.
5. אם הצלע מחברת בין שני עצים, כלומר קודקודים *u* ו-*v* שייכם לקבוצות שונות מוסיפים אותה ל-*T* ומאחדים את העצים (צלע כזו נקראת "צלע בטוחה" – *safe edge*).
במקרה שהצלע מחברת בין שני קודקודים השייכים לאותו עץ (צלע כזו סוגרת מעגל), מדלגים עליה.

6.בודקים האם מספר הצלעות ב-*T* שווה ל-*|V|-1*, אם כן האלגוריתם מסיים, אחרת נחזור לשלב 4.

איחוד קבוצות זרות:

איחוד קבוצות זרות הוא מבנה נתונים שמבצע מעקב אחרי קבוצה של אובייקטים שמחולקים למספר תתי קבוצות זרות ולא חופפות.

פעולות של המבנה איחוד קבוצות זרות:

1. *MakeSet* (יצירה) – פעולה שיוצרת קבוצה חדשה המכילה אובייקט אחד בלבד (*singleton*).
2. *Find* (חיפוש) – קובע איזו קבוצה מכילה אובייקט ספציפי, פעולה זו יכולה גם לעזור בקביעה האם שני אובייקטים שייכים לאותה קבוצה.
3. *Union* (איחוד) – איחוד שתי קבוצות לקבוצה אחת.

הצגת המימוש:

כל קבוצה מיוצגת ע"י עץ, שורש העץ נחשב בנציג של הקבוצה.
כל קודקוד מחזיק את המצביע לצומת האב שלו.
Find הולך לפי האבות עד שהוא מגיע לשורש העץ.
Union משלב שני עצים להיות עץ אחד ע"י הצמדת שורש אחד לשורש של עץ אחר.
העץ שנוצר לאחר איחוד של שני עצים יכול להיות לא מאוזן והחיפוש עליו יהיה לא יעיל.
אחת מהדרכים ליצור עץ מאוזן יותר נקראת איגוד לפי דרגה, השיטה מחזיקה דרגה לכל קודקוד (הערך ההתחלתי הוא 0) ומצרפת את העץ הקטן לשורש של העץ הגדול.
פונקציית *Find(x)* מקיימת גם *path compression*, כלומר לאחר שנמצא שורש העץ שמכיל את קודקוד *x*, היא משנה את קודקודי האבות של קודקודים שנמצאים במסלול מקודקוד *x* ישירות לשורש, כלומר *x* נהיה הבן של השורש.

<u>קוד:</u>

```
Kruskal(G): // O(|E|log|V|)
    create Tree T ⇐ ∅

    for each v∈V(G) do: // O(|V|)
        MakeSet(v)
    end-for

    // sort E in increasing order by edges weight
    Sort(E(G)) // O(|E|log|E|)

    for each e∈E(G) do: // O(|E|log|V|)
        if FindSet(e.u) ≠ FindSet(e.v) then:
            T.add(e)
            Union(e.u, e.v)
        end-if
        if |T| = |V|-1 then:
            return T
        end-if
    end-for

    return T

end-Kruskal

MakeSet(v): // O(1)
    v.parent ⇐ v
end-MakeSet

FindSet(v): // O(log|V|)
    if v = v.parent then:
        return v.parent
    else:
        return FindSet(v.parent)
    end-if
end-FindSet

Union(u,v): // O(log|V|)
    uRoot ⇐ FindSet(u)
    vRoot ⇐ FindSet(v)
    uRoot.parent ⇐ vRoot
end-Union
```

סיבוכיות – $O(|E|\log|E|)$ עבור המיון.
מעבר על הצלעות מהקטנה לגדולה: $O(|E|)$.
בכל מעבר צריך לבדוק שלא סוגרים מעגל.
שומרים את הקודקודים שהם באותו רכיב קשירות בתוך מבנה נתונים union-Find (איחוד קבוצות זרות) סיבוכיות פעולות שימוש במבנה (union/find) היא $O(\log|V|)$.
סה"כ: $O(|E|\log|V|)$.
לכן הסיבוכיות הכוללת היא: $O(|E|\log|E|)$.

<u>בורובקה:</u>

נתחיל מרכיבי קשירות (כמו union-find בקרוסקל).
בכל שלב נמצא את הצלע הכי נמוכה שמחוברת לרכיב הקשירות ולא סוגרת מעגל.
כל עוד לא סיימנו, נמצא את הצלעות הנמוכות לכל רכיב (המחברות בין 2 רכיבים שונים) ונוסיף אותן לעץ.

<u>קוד:</u>

```
Boruvka(G):

    for each v∈V(G) do: // O(|V|)
        MakeSet(v)
    end-for

    create Tree T ⇐ Ø
    isFinished ⇐ false

    while is not isFinished do:
      create cheapest[|V(G)|] // init to null

      for i ⇐ 0 to |E| do:
          e ⇐ E[i]
          g1 ⇐ FindSet(e.v1)
          g2 ⇐ FindSet(e.v2)

          if g1 ≠ g2 then:

              if e.weight < cheapest[g1].weight then:
                  cheapest[g1] ⇐ e
              end-if

              if e.weight < cheapest[g2].weight then:
                  cheapest[g2] ⇐ e
              end-if

          end-if
      end-for

      isFinished ⇐ true

      for i ⇐ 0 to |V| do:
          if cheapest[i] ≠ NULL then:
              T.add(cheapest[i])
              Union(cheapest[i].v1, cheapest[i].v2)
              isFinished  = false
          end-if
      end-for
    end-while
    return T
end-Boruvka

MakeSet(v): // O(1)
        v.parent ⇐ v
end-MakeSet

FindSet(v): // O(log|V|)
        if v = v.parent then:
                return v.parent
        else:
                return FindSet(v.parent)
        end-if
end-FindSet

Union(u,v): // O(log|V|)
        uRoot ⇐ FindSet(u)
        vRoot ⇐ FindSet(v)

        uRoot.parent ⇐ vRoot
end-Union
```

סיבוכיות – ניתן להגיע ע"י מימוש יעיל ל $O(|E|\log|V|)$ כי בכל שלב עוברים על כל הצלעות ולאחר כל שלב, כמות הרכיבים מצטמצמת בפי 2 פחות (ואולי גם יותר).

<u>DFS – חיפוש לעומק:</u>

אלגוריתם חיפוש לעומק הוא אלגוריתם המשמש למעבר על גרף או לחיפוש בו.
האלגוריתם מתחיל את החיפוש מצומת שרירותי בגרף ומתקדם לאורך הגרף עד שהוא נתקע, לאחר מכן הוא חוזר על עקבותיו עד שהוא יכול לבחור להתקדם לצומת אליו הוא טרם הגיע.
דרך פעולת האלגוריתם דומה בערך לסריקה שיטתית של מבוך.
כמו בBFS גם כאן נצבעים קודקודים בהמלך החיפוש כדי לציין את המצב שלהם, בהתחלה כל הקודקודים צבועים בלבן, לאחר מכן, כל קודקוד שמתגלה בחיפוש נצבע באפור ולאחר שמסיימים איתו, צובעים אותו בשחור.
בכל פעם שמתגלה קודקוד v במהלך הסריקה של רשימת השכנים של קודקוד u שכבר התגלה, החיפוש לעומק מציין את אירוע זה ע"י הצבת u, כלומר הקודם של v.

מה שלא כמו בBFS ששם תת הגרף הקודם שלו יוצר עץ, אז כאן תת הגרף הקודם שנוצר פה יכול להיות מורכב מכמה עצים, מכיוון שהחיפוש יכול להתנהל מכמה מקורות.
תת הגרף הקודם של חיפוש לעומק יוצר יער עומק שמורכב מכמה עצי עומק.
חיפוש לעומק שומר בכל קודקוד חותמת של זמן, כלומר לכל קודקוד v יש שתי חותמות זמן:

1. firstTime[v] מייצגת מתי התגלה v לראשונה ונצבע באפור.
2. lastTime[v] מייצגת מתי החיפוש סיים לבחון את רשימת השכנים של v וצבע את v בשחור.

חותמות זמן אלה מסייעות בניתוח ההתנהגות של חיפוש לעומק.
חותמות אלה הן ערכים שלמים בין 1 ל-2|V|, שכל אחד מ-|V| הקודקודים מתגלה פעם אחת והטיפול בו מסתיים אחרי פעם אחת.
לכל קודקוד מתקיים ש – firstTime[u] < lastTime[u].
קודקוד u הוא בצבע לבן לפני firstTime[u], הוא נצבע באפור בין firstTime[u] לבין lastTime[u] ונצבע בשחור לאחר lastTime[u].
הגרף יכול להיות מכוון או לא מכוון.

<u>קוד:</u>

סיבוכיות – O(|V + E|).

```
DFS(G):
    time ⇐ 0

    for each v∈V(G) do:
        color[v] ⇐ WHITE
        prev[v] ⇐ NIL
        dist[v] ⇐ ∞
    end-for

    for each v∈V(G) do:
        if color[v] = WHITE then:
            DFS_REC(G,v)
        end-if
    end-for
end-DFS
```

1

```
DFS_REC(G,v):
    color[v] ⇐ GRAY
    time ⇐ time + 1
    dist[v] ⇐ time
    for each u∈Adj(v) do:
        if color[u] = WHITE then:
            prev[u] ⇐ v
            DFS_REC(G,u)
        end-if
    end-for
    color[v] ⇐ BLACK
end-DFS_REC
```

2

קודים: דור עזריה
עורכים: ערן לוי ועדי הופטמן

קוד – בודק אם יש מעגל בגרף:

```
HasCircle(G)
    ans ⇐  false
    for each u ∈ V[G] do:
        color[u] ⇐  WHITE
        prev[u] ⇐  NIL
    end-for

    for each u ∈ V[G] and ans=false do:
        if color[u] = WHITE then:
            ans ⇐ DFSVisit(G,u)
        end-if
    end-for

    return ans;
end- HasCircle
```

1

```
DFSVisit(G,u)
    ans ⇐ false
    color[u] ⇐ GRAY
    for each v ∈ Adj(u) and ans=false do:
        if color[v] = GRAY and prev[u]≠v then:
            ans ⇐ true
            getCycle(G, u, v)
        else if color[v] = WHITE then:
            prev[v] ⇐ u
            ans ⇐ DFSVisit(G,v)
        end-if
    end-for
    color[u] ⇐ BLACK
    return ans
end-DFSVisit
```

2

```
GetCycle(G, u, v)
    create Stack cycle
    x ⇐ u
    while x ≠ v do:
        Push(cycle, x)
        x ⇐ prev[x]
    end-while
    Push(cycle , v)
    Push(cycle, u)
    Reverse(cycle)
end-GetCycle
```

3

שימושים:

לDFS יש יתרונות על BFS אם ישנו ידע קודם או אינטואיציה שיכולה לעזור לנו בחיפוש, למשל בחיפוש יציאה ממבוך, אם נפעיל BFS מאמצע המבוך, נמצא את היציאה רק בשלב האחרון של האלגוריתם, לעומת זאת בDFS נוכל למצוא את היציאה כבר בתחילת ריצת האלגוריתם עוד לפני שהוא יצטרך לשוב על עקבותיו, בפרט אם יש לו דרך טובה להעריך מהו הכיוון הנכון של היציאה.

לBFS קשה לעבור על גרפים גדולים, בפרט גרפים שהמסלולים שלהם אינסופיים, לעומת זאת DFS יגיע מתישהו אל האיבר שמחפשים.