אלגוריתמים 2

מחברת סיכומים מההרצאות של פרופ' לויט ואדים ותרגולים

אלגוריתמים 2 - מחברת הרצאות

נערך ונכתב על-ידי דור עזריה

2021

ספר זה לא נבדק על ידי מרצה, יתכן שימצאו טעויות.

לסיכומים נוספים שלי במדעי המחשב ומתמטיקה:
https://dorazaria.github.io/

מוזמנים לעקוב אחרי 😊:
LinkedIn: https://www.linkedin.com/in/dor-azaria/
GitHub: https://github.com/DorAzaria

תוכן עניינים

טבלת סיבוכיות (האופטימלי ביותר עבור כל בעיה)			
בעיית הבקבוקים (מילוי)	$O((b_1+1)(b_2+1))$	Floyd Warshall	$O(n^3)$
Best (Array)	$O(N)$	Submatrix	$O(N^3)$
Gas Station	$O(N)$	Dijkstra	$O(\|E\|+\|V\|\cdot log\|V\|)$
Fire	$O(\|V\|+\|E\|)$	דייקסטרה דו-כיוונית	$O(\|E\|+\|V\|\cdot log\|V\|)$
BFS	$O(\|V\|+\|E\|)$	DFS	$O(\|V\|+\|E\|)$
Bipartite Graph	$O(\|V\|+\|E\|)$	קוטר של גרף בעזרת BFS	$O(\|V\|+\|E\|)$
עצים איזומורפיים	$O(N\cdot log(N))$	בניית עץ מרשימת דרגות	$O(N\cdot log(N))$
Huffman Algorithm	$O(N)$	Kruskal רגיל	$O(\|E\|\cdot \log\|V\|)$
Prim	$O(\|E\|\cdot \log\|V\|)$	Kruskal ממויין מראש	$O(\|E\|\cdot \alpha(\|V\|))$
Boruvka	$O(\|E\|\cdot log\|V\|)$	Kruskal הפוך	$O(\|E\|\cdot(\|V\|+\|E\|))$
מסלול אוילר	$O(\|V\|+\|E\|)$	מעגל אוילר	$O(\|V\|+\|E\|)$

מבוא לתורת הגרפים

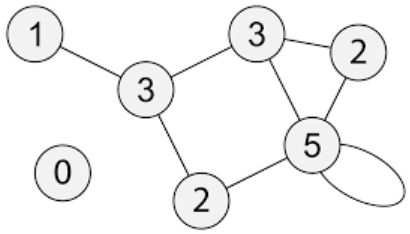
(מתוך הסיכום של תורת הגרפים בקורס בדידה).

1. גרף הוא זוג $G = (V, E)$, כאשר **V** = קבוצת קודקודים, **E** = קבוצת זוגות של קודקודים.
כל זוג של קודקודים נקרא **צלע** או **קשת**.

2. בגרף **מכוון** E הינה קבוצת זוגות **סדורים** של קודקודים.

3. 2 קודקודים שמחוברים בצלע יקראו **שכנים** - סימון : $\Gamma(v)$ זו קבוצת השכנים של v.

4. מספר הצלעות שיוצאות מקודקוד מסוים הינו **הדרגה** של הקודקוד,
למשל:



- **משפט:** $\sum_{v \in V} d(v) = 2 \cdot |E|$.

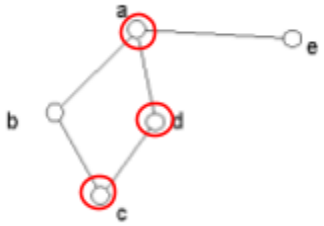
5. **מסלול** בגרף הינו רצף של קודקודים כך שלכל 2 קודקודים עוקבים ברצף, הצלע קיימת בגרף.
זאת אומרת, לכל $1 \leq i \leq k - 1$, $\{V_i, V_{i+1}\} \in E(G)$.

6. אם כל הקודקודים במסלול שונים זה מזה. נאמר **שהמסלול פשוט.**

7. **מעגל** בגרף הוא מסלול שבו הקודקוד הראשון והאחרון זהים.

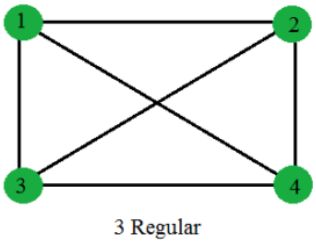
8. **מעגל פשוט** רק הקודקוד הראשון והאחרון זהים (כל השאר שונים זה מזה).

9. בהינתן קודקוד v_1, קבוצת השכנים $\Gamma(v_1)$ הינה קבוצת כל הקודקודים המחוברים בצלע ל-v_1.



למשל: $\Gamma(b) = \{a, d, c\}$

10. גרף **d-רגולרי** הינו גרף שבו כל הדרגות שוות d.



3 Regular

כל גרף שלם הינו (n-1)-רגולרי.

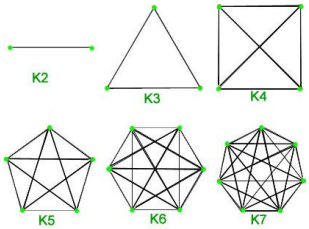
11. עץ הינו גרף קשיר ללא מעגלים.

- כל עץ עם $n \geq 2$ קודקודים מכיל עלה (=קודקוד מדרגה 1).
- מספר הצלעות בעץ בעל n קודקודים הינו n-1
  אם הוא יותר מ n-1 צלעות אז הוא לא עץ ובהכרח קיים מעגל בגרף.
- כל עץ הוא גרף דו-צדדי.

12. יער הינו גרף ללא מעגלים (לא בהכרח קשיר).

13. מעגל אוילר הינו מעגל שעובר על כל הצלעות של הגרף פעם אחת בדיוק.

14. גרף שלם מעל n קודקודים מסומן K_n,
הינו גרף עם n קודקודים שבו **כל** הצלעות האפשרויות הם קיימות.

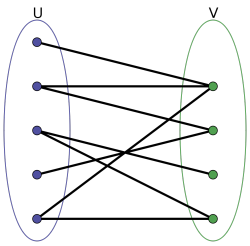


- בגרף K_n קיימות $\binom{n}{2}$ צלעות.
- **קליקה** - תת גרף שלם שבו בין **כל** 2 קודקודים יש קשת.

15. גרף קשיר, הינו גרף בו קיים מסלול בין כל 2 קודקודים.

16. רכיבי קשירות- כאשר גרף **אינו** קשיר ניתן לחלק אותו למחלקות שונות של תתי-גרפים קשירים.

17. תת גרף - בהינתן גרף $G = (V, E)$ נגדיר תת גרף $G' = (V', E')$ כך שמתקיים: $G' \subseteq G$,
$V' \subseteq V$, $E' \subseteq E$ ובנוסף לכל צלע $e \subseteq V'$, $e \subseteq E'$.

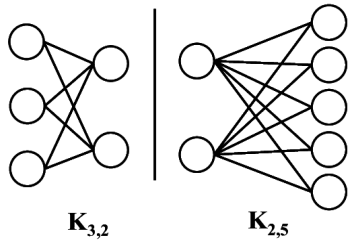


18. גרף דו צדדי - $G = (V_1 \cup V_2, E)$, $V_1 \cap V_2 = \emptyset$
כך שלכל הצלעות ב-E יש קודקוד אחד ב-V_1 וקודקוד אחד ב-V_2.

במילים אחרות ניתן לחלק את קודקודי הגרף ל-2 כך שכל הצלעות **חוצות** את 2 הקבוצות.

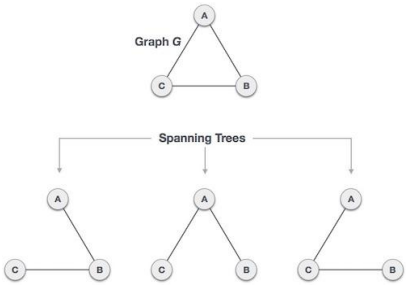
- גרף הוא דו-צדדי אמ"מ **כל** המעגלים בו בעלי אורך זוגי.

19. גרף דו-צדדי מלא/שלם - גרף דו צדדי בו נמצאות כל הקשתות האפשריות.
סימון: $K_{n,m}$ כאשר n קודקודים בצד אחד ו-m קודקודים בצד שני.



20. קבוצה בלתי תלויה בגרף או **אנטי-קליקה**- הינה קבוצה של קודקודים שאין בתוכה צלעות.
(בדיוק ההפך מגרף שלם).

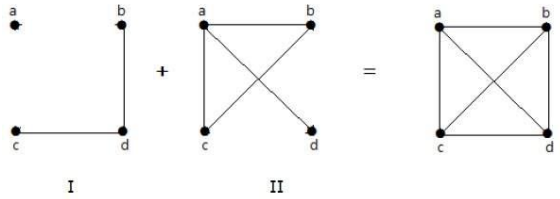
- הגרף הריק : $E = \{\}, V = \{v_1, v_2, \ldots, v_n\}, G = (V, E)$
  הוא **כולו** קבוצה בלתי תלויה.
- בגרף דו-צדדי V_1, V_2 הינן קבוצות בלתי תלויות.



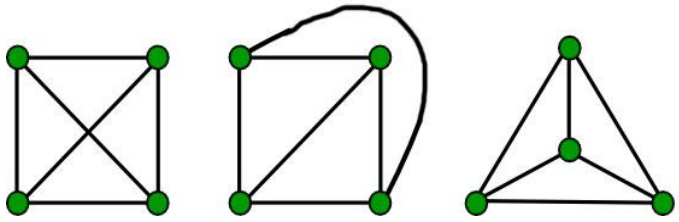
21. עץ פורש של G הינו **תת** גרף של גרף G אשר הינו עץ (כלומר לא מכיל מעגלים) ובנוסף מכיל את כל קודקודי G.

22. גרף משלים - לגרף $G = (V, E)$, נגדיר גרף משלים $\overline{G}$ כך ש- $\overline{G} = (V, E(K_{|v|}) \backslash E)$

זאת אומרת, נעבור על כל הצלעות האפשריות מעל V, אם הצלע ב-G, אז לא נוסיף אותה ל-$\overline{G}$.
ואם הצלע לא ב-G אז כן נוסיף אותה ל-$\overline{G}$.



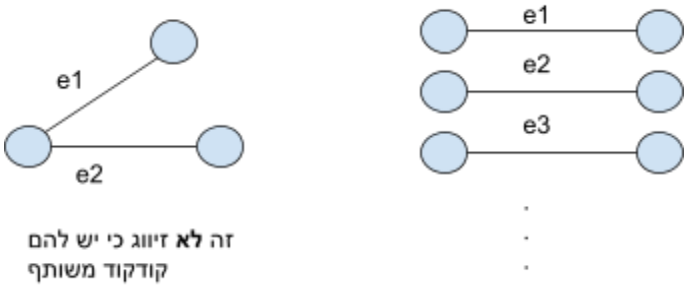
23. גרף מישורי, ניתן לצייר במישור כך שאין שתי צלעות נחתכות.



- לכל אזור שתחום מכל הכיוונים בצלעות נקרא **פאה.**
- נוסחת אוילר אומרת כי לכל גרף מישורי **קשיר** מתקיים: $f = e - n + 2$.

- אם $f \geq 2$ אזי יש בגרף מעגל.
- נוסחה לגרף לא קשיר בעל d רכיבי קשירות $f = e - n + 1 + d$.
- תנאי הכרחי (ולא מספיק) למישוריות. יהי גרף G קשיר ומישורי אזי $e \leq 3(n - 2)$

24. זיווג $M \subseteq E(G)$ הינו אוסף של צלעות כך שלאף זוג מ-M אין קודקוד משותף.



- הגודל המקסימלי האפשרי של זיווג ב**גרף דו-צדדי** $G = (V_1 \cup V_2, E)$ הוא $min(|V_1|, |V_2|)$.

25. זיווג M יקרא מושלם אם כל קודקודי הגרף משתתפים בו.

- הגודל של זיווג מושלם של גרף בעל n צלעות הוא $|M| = \frac{1}{2} \cdot n$.
- זיווג **מושלם** בגרף דו צדדי יכול להתקיים רק אם $|V_1| = |V_2|$.

- יהי G גרף דו-צדדי **d-רגולרי,** אזי יש ב-G זיווג מושלם.
- בגרף דו-צדדי d-רגולרי אפשר למצוא בדיוק d זיווגים מושלמים **זרים** שאיחודם שווה לכל צלעות הגרף, בעבור זיווגים מושלמים **לאו דווקא זרים** יש !d.

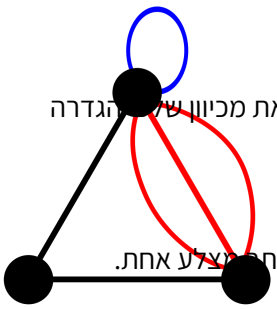
26. 'G יקרא **עידון** של G אם ניתן לקבל אותו מ-G ע"י ביצוע מספר כלשהו של החלפות של צלע ב-2 צלעות, על-ידי הוספת קודקוד חדש.

כך:

- אם נתון שבגרף G יש תת גרף שהוא עידון של K_5 או $K_{3,3}$ אזי G **איננו** מישורי.(כל גרף הוא עידון של עצמו).

27. בגרף לא מכוון Gיש מספר זוגי של קודקודים שדרגתם אי-זוגית.

הוכחה: לא יתכן שבגרף לא מכוון יש מספר אי-זוגי של קודקודים שדרגתם אי-זוגית, זאת מכיוון שאז הגדרה אנו יודעים כי $\Sigma_{v \in V} degree(v) = 2 \cdot |E|$, כלומר סכום הדרגות הוא מספר זוגי.



28. מולטיגרף הוא הכללה של גרף, שבה כל זוג קודקודים יכולים להיות מחוברים על ידי יותר מצלע אחת.

הגדרות והוכחות שנלמדו בהרצאות

✪ למה 1 - לחיצת הידיים

☚ לכל גרף $G = (V, E)$ סכום הדרגות של כל הקודקודים בגרף G שווה לפעמיים מספר הצלעות:

$$\Sigma_{v \in V} deg(v) = 2 \cdot |E|$$

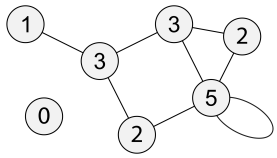
✪ למה 2 - לחיצת הידיים

☚ לכל גרף $G = (V, E)$ מספר כל הקודקודים בעלי דרגה אי-זוגית הוא מספר זוגי:

$$\Sigma_{v \in V} deg(v) = 2 \cdot |E|$$

☚ בכל קבוצת אנשים, מספר האנשים שלחצו ידיים למספר אי זוגי של אנשים אחרים מהקבוצה הוא מספר זוגי.

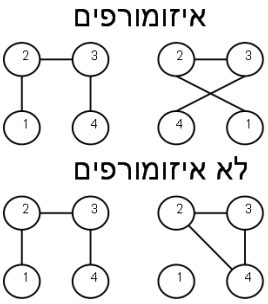
✪ הגדרה - סדרת דרגות



☚ **סדרת הדרגות** של גרף לא מכוון עם n צמתים היא סדרה לא עולה של דרגות הצמתים של הגרף.

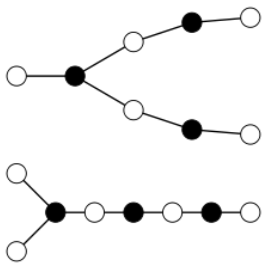
☚ למשל עבור האיור הבא זה: $(5, 3, 3, 2, 2, 1, 0)$

✪ הגדרה - גרפים איזומורפיים (הגדרה הוצגה בהרצאה אבל לא בצורה הזאת):



☚ גרפים G, H הם איזומורפיים אם קיימת פונקציה $f: V(G) \to V(H)$ חד-חד ערכית ועל, כך שעבור כל $v, w \in V(G)$, מספר הקשתות המקשרות בין v ו-w, זהה למספר הקשתות המקשרות בין $f(v)$ ו-$f(w)$.

✪ הגדרה - סדרת דרגות 2



☚ סדרת הדרגות הוא גרף שלא ניתן לשינוי.

☚ לכן לגרפים איזומורפיים יש את אותה סדרת דרגות.

☚ אבל התכונה ההפוכה לא מתקיימת, אם לשני גרפים אותה סדרת דרגות הם לא בהכרח איזומורפיים.

☚ זוג גרפים לא איזומורפיים עם אותה סדרת דרגות $(3, 2, 2, 2, 2, 1, 1, 1)$:

✪ טענה

☚ לכל גרף לא מכוון $G = (V, E)$, כאשר $|V(G)| = n \geq 2$ וגם $|E(G)| = m$,
קיימים לפחות 2 קודקודים בעלי אותה הדרגה.

☚ **הוכחה**

מספר הדרגה המקסימלית האפשרית בגרף הוא $n - 1$.
אם G גרף קשיר, אז לכל $v \in V(G)$ $1 \leq d(v)$, מאחר ו-$|V(G)| = n \geq 2$.
לכן לכל $v \in V(G)$ $1 \leq d(v) \leq n - 1$.
ולפי עקרון שובך היונים אם נחלק n קודקודים ב-$(n - 1)$ תאים אז יהיו שני קדקודים באותו התא, כלומר שני קודקודים עם אותה דרגה.

I

✪ גרפי קורונה

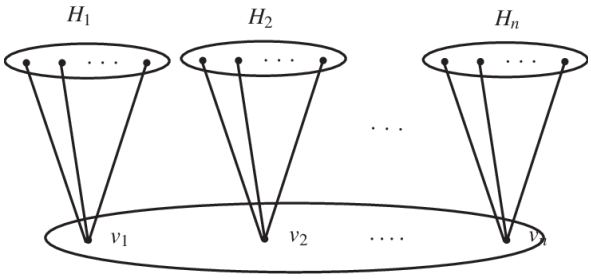
☚ יהי $\{H_v : v \in V(G)\} = \mathfrak{H}$ להיות משפחה של גרפים באינדקס המיוצג באמצעות קבוצת הקודקודים של גרף G.

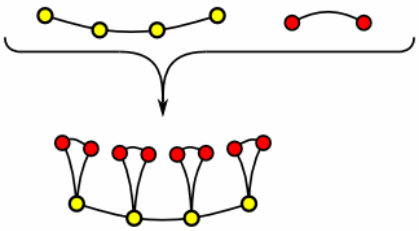
☚ הקורונה $G * H$ הוא האיחוד הזר של G ו-$\mathfrak{H}$, עם תוספת של צלעות הנוצרות בין כל קודקוד $v \in V(G)$ לבין כל קודקודי H_v.

☚ אם $H_v = H$ לכל $v \in V(G)$, אז אנחנו נציין את $G * H$ במקום $G * \mathfrak{H}$.

☚ גרפים כאלה נועדו בעיקר לעיסוק בצביעת קודקודים.

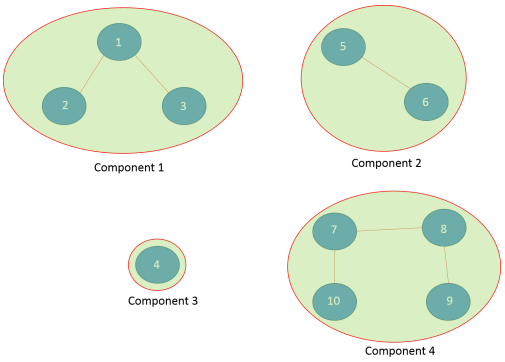
☚ אם כל ה-H_v הם גרפים שלמים, אז $G * H$ הוא גרף קליקה-קורונה.





✪ הגדרה - רכיבי קשירות

☚ עבור גרף לא מכוון $G = (V, E)$ רכיב קשירות בגרף G היא קבוצה של קודקודים בגרף המקושרים זה לזה על ידי המסלולים בגרף.

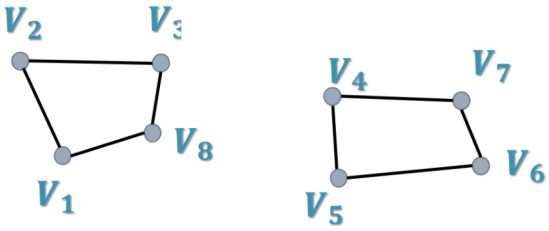


✪ הגדרה - גרף לא קשיר

☚ גרף לא קשיר אם לפחות שני קודקודים בגרף אינם מחוברים במסלול.

☚ אם גרף G לא קשיר, אז כל תת גרף קשיר של G נקרא רכיב קשירות של הגרף G.

✪ הגדרה - גרף דו-צדדי - Bipartite Graphs



☚ הגרף $G = (A, B, E)$ יקרא דו-צדדי.

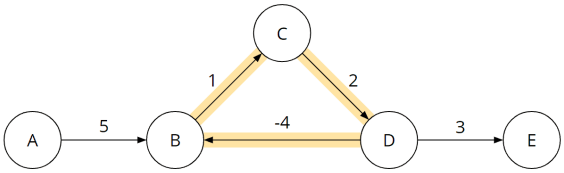
☚ הוא גרף שבו ניתן לחלק את הקודקודים לשתי קבוצות זרות,
כך שלא קיימת צלע בין שני קודקודים השייכים לאותה הקבוצה.

☚ נגדיר קבוצות קודוקדים כדוגמה לאיור משמאל:
$A = \{V_1, V_3, V_5, V_7\}$, $B = \{B_2, B_4, B_6, B_8\}$.

☚ הגרף G לא קשיר ומספר רכיבי הקשירות הוא 2.

✪ הגדרה - גרף עם מעגל שלילי

☚ גרף $G = (V, E)$ יקרא עם מעגל שלילי אם קיים מסלול מעגלי $v_i \to v_i$ (מאותו קודקוד לעצמו) כך שסכום מסלול
המעגל הוא שלילי (סכום משקלי כל הצלעות במסלול).



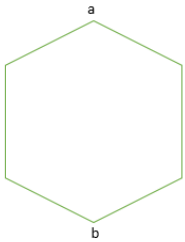
✪ הגדרה

☚ עבור בעיית סגירות טרנזיטיבית, משקלי הצלעות בגרף יהיו בוליאניות (אלמנטים מהקבוצה $\{True, False\}$)

☚ האופרטורים שנשתמש בהם: ∧- עבור צירוף בין מסלולים, ∨- עבור הפרדה מוחלטת בין מסלולים.

✪ טענה 1.1

בעץ T בין שני קודקודים יש מסלול אחד בלבד.



✪ הוכחה 1.1

מאחר ו-T הוא גרף קשיר, קיים לפחות מסלול אחד בין כל זוג קודקודים.

נניח וקיימים 2 מסלולים שונים בין קודקודים a, b בגרף העץ T.

האיחוד של 2 המסלולים האלו יצרו מעגל ב-T.

- סתירה להגדרת העץ.

∎

✪ טענה 1.2

לכל עץ יש לפחות עלה אחד.

✪ הוכחה 1.2

נניח בשלילה שלעץ אין עלים, לכן דרגה של כל קודקוד הוא $d_i \geq 2$.

ניקח קודקוד כלשהו, בגלל שדרגתו גדולה או שווה ל-2 נעבור לקודקוד הבא, אם ביקרנו בו סגרנו מעגל, אחרת נעבור לקודקוד הבא. בגלל שמספר הקודקודים בעץ הוא סופי, בשלב מסוים נגיע לקודקוד שביקרנו בו יותר מפעם אחת, - סגרנו מעגל - סתירה להגדרת העץ. לכן, לעץ יש לפחות עלה אחד.

■

✪ טענה 1.3 - Berge

לכל עץ יש לפחות שני עלים.

✪ הוכחה 1.3

ניקח מסלול P <u>הארוך</u> ביותר בין שני קודקודי העץ כאשר $P = v_1, v_2, ..., v_k$.

Pהוא מסלול פשוט ו-v_1הוא עלה מכיוון שאם הדרגה שלו הייתה לפחות 2 אז הקודקוד הסמוך אליו u, לא שייך ל-P .

ואם $u \notin P$ אז קיבלנו מסלול ארוך יותר מ-Pונקרא לו P_1המוגדר כך: $P_1 = u, v_1, v_2, ..., v_k$.

- סתירה להגדרת העץ.

לכן v_1הוא עלה. באופן דומה, ניתן להוכיח כי גם v_kהוא עלה.

לכן לכל עץ יש לפחות שני עלים.

■

✪ טענה 1.4

לכל עץ עם $|V| = n$קודקודים יש בדיוק $|E| = n - 1$צלעות.

✪ הוכחה 1.4

נוכיח באינדוקציה על n .

☚ בסיס האינדוקציה: אם $n = 1$אין צלעות ואם $n = 2$אז יש צלע אחת.

☚ הנחת האינדוקציה: נניח שהטענה נכונה עבור nצמתים ויש בדיוק $n - 1$צלעות.

☚ צעד האינדוקציה: נראה עבור $n + 1$צמתים.

ניקח עלה ונמחק אותו יחד עם הצלע היחידה שמחוברת אליו, לפי הנחת האינדוקציה נקבל עץ בעל nקודקודים ו-$n - 1$צלעות , לכן לעץ בעל $n + 1$ צמתים יש $(n + 1) - 1 = n$צלעות.

■

✪ הגדרה

קוטר העץ הוא האורך של המסלול הארוך ביותר, כאשר הקצוות שלו הם עלים.

✪ טענה 1.5

אם מורידים עלה בעץ הוא נשאר קשיר.

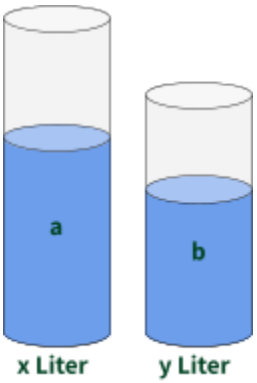
✪ הוכחה 1.5

כשמורידים עלה לא יצרנו מעגל, ולכן הגרף ללא מעגלים.

בנוסף, כאשר הורדנו עלה אז גם הצלע ממנו ירדה, ועדיין מתקיים $|E| = |V| - 1$.

■

בעיית הבקבוקים - Water Jug Problem



✪ תיאור הבעיה

- נתונים 2 בקבוקים בנפחים שונים, בקבוק **1** בנפח **x** ליטר ובקבוק **2** בנפח **y** ליטר.
- בקבוק 1 בתכולה **a** ובקבוק 2 בתכולה **b** ונייצג אותם ב-"טאפל" (a, b) עם חשיבות לסדר.
- מה מספר הפעולות המינימלי ביותר כדי למלא את בקבוק 1 עם a ליטר ובקבוק 2 יהיה ריק.

✪ חוקים

1. ניתן לרוקן את הבקבוק עד סופו (לא ניתן לרוקן חלקית).
2. ניתן למלא את הבקבוק עד סופו (לא ניתן למלא חלקית).
3. ניתן למזוג מבקבוק אחד לשני עד שאחד הבקבוקים ריק או מלא.
4. אין הגבלה למצב ההתחלתי של כל בקבוק, כלומר בקבוק יכול להיות ריק או בכל מספר שלם של ליטר אפשרי.

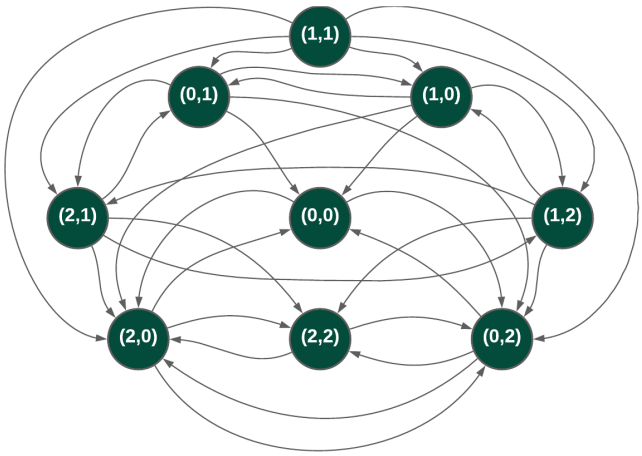
✪ דוגמה

נתונים 2 בקבוקים ריקים. בקבוק 1 עם $x = 5$ ליטר בקבוק 2 עם $y = 3$ ליטר.
נרצה למלא את בקבוק 1 עם $a = 4$ ליטר ובקבוק 2 יהיה ריק $b = 0$.

$$(0,0) \rightarrow (5,0) \rightarrow (2,3) \rightarrow (2,0) \rightarrow (0,2) \rightarrow (5,2) \rightarrow (4,3) \rightarrow (4,0)$$

נשים לב כי הפתרון הנל נפתר בצורה של גרף, לכן נייצג את בעיית הבקבוקים בגרף.
$B_{x,y} = (V_{x,y}, E_{x,y})$ הוא גרף הבקבוקים, כאשר x הוא הנפח של בקבוק 1 ו-y הוא הנפח של בקבוק 2 כאשר x,y מספרים שלמים טבעיים.
הקודקודים מייצגים את כל המצבים האפשריים של הבקבוקים.
הצלעות מייצגות את כל המעברים החוקיים בין מצב קודקוד א' למצב קודקוד ב'.

ציור לדוגמה של הגרף $B_{2,2} = (V_{2,2}, E_{2,2})$ המציג את כל המקרים של הבקבוקים:

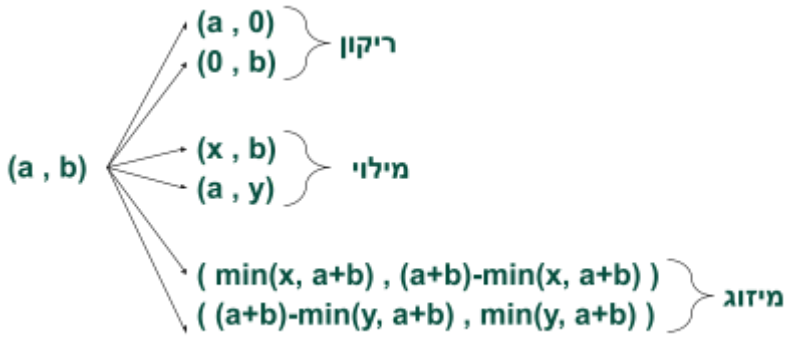


אפשר לראות כמה הגרף לא קריא ומבולגן, ננסה לעבוד בצורה יותר נוחה.
נשאלת השאלה מה מספר הצלעות והקודקודים בגרף $B_{x,y}$?
נצייר מטריצה שתתאר לנו את כל המצבים האפשריים בגרף.
נעזר בגרף מהדוגמה הקודמת $B_{2,2} = (V_{2,2}, E_{2,2})$:

	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)

לפי המטריצה אפשר לראות כי מספר הקודקודים בגרף הוא: $|V_{2,2}| = 9$.
אם כך, עבור המקרה הכללי נקבל: $|V_{x,y}| = (x+1)\cdot(y+1)$.

מה מספר הצלעות בגרף?
נסתכל על קודקוד כלשהו ונבחן איזה צלעות אפשרויות יכולות לצאת ממנו $\deg_{\rightarrow}(a,b)$:



למשל עבור מקרה של מזיגה מבקבוק 1 ל-2 נבצע: $((a+b)-min(y,a+b),\ min(y,a+b))$
כאשר $a+b$ מייצג את כמות המים הכוללת של 2 הבקבוקים יחדיו.
עבור הבקבוק השני נבחר $min(y,a+b)$וזאת כדי להבטיח שלא יקרה מקרה בו $y < a+b$מאחר ו-y הוא הנפח המקסימלי בבקבוק השני.
עבור הבקבוק הראשון נבחר $(a+b)-min(y,a+b)$כלומר כל מה שנשאר לנו בבקבוק הראשון לאחר המזיגה אל הבקבוק השני.

נחזור לשאלה מה מספר הצלעות בגרף? - ראינו 6 מקרים עבור קודקוד (a,b), אבל לא בהכרח שמכל קודקוד בגרף יצאו 6 צלעות (לדוגמה הציור של הגרף בעמוד הקודם), יכול להיות שדרגה יוצאת של קודקוד נמוכה יותר מ-6.
לכן נובע ש $\deg_{\rightarrow}(a,b) \leq 6$ומכאן מתקיים לכל היותר:

$$\left|E_{x,y}\right| \leq 6 \cdot \left|V_{x,y}\right| = 6 \cdot (x+1) \cdot (y+1)$$

את הגרף $B_{x,y}$ אפשר לייצג במחשב בכמה סוגים של מבני נתונים ושיטות פתרון.

נציג כעת את מטריצת השכנויות של הגרף $B_{2,2} = (V_{2,2}, E_{2,2})$ בגודל $\left|V_{2,2}\right| x \left|V_{2,2}\right|$.

המטריצה מכילה אפסים כאשר כל עמודה ושורה מייצגת את הקשר בין 2 קודקודים באופן בינארי (True or False).

אם קיים קשר ישיר בין 2 קודקודים נסמן ב-1.

	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)
(0,0)	0		1				1		
(0,1)	1	0	1	1				1	
(0,2)	1		0				1		1
(1,0)	1	1		0		1	1		
(1,1)		1	1	1	0	1	1	1	
(1,2)			1	1		0		1	1
(2,0)	1		1				0		1
(2,1)		1				1	1	0	1
(2,2)			1				1		0

מספר האחדות בעמודה הוא הדרגה הנכנסת (מספר הצלעות המצביעות) עבור הקודקוד המייצג את העמודה.

מספר האחדות בשורה הוא הדרגה היוצאת (מספר הצלעות היוצאות) עבור הקודקוד המייצג את השורה.

אנחנו משתמשים במטריצה דו מימדית, מה נשים באינדקסים של המטריצה? הם לא יכולים לכלול 2 קואורדינטות כמו שהצגנו כאן, אז איך נבצע המרה?

נגדיר:

i - גובה המים של בקבוק 1.

j - גובה המים של בקבוק 2.

col - מספר העמודות כלומר (y+1).

ומתקיים: $k = (col + 1) \cdot i + j$ כאשר k מייצג את התא.

איך ולמה k?

אנחנו ניצור בקוד מטריצה מהצורה:

```
this.matrix = new int[(x+1)*(y+1)][(x+1)*(y+1)];
```

נתבונן במטריצה לדוגמה שהצגנו, עבור השורה החמישית:

index		0	1	2	3	4	5	6	7	8
....	case	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)
4	(1,1)		1	1	1	0	1	1	1	

אנחנו לא יכולים לבקש מהמטריצה את המקרה $(4,6)$ כי הוא לא קיים כלל וכלל בלולאה מאחר ו- x=2 וגם y=2.

נסתכל על המטריצה "דמיונית" הנראית כך:

(0,0) = 0	(0,1) = 1	(0,2) = 2
(1,0) = 3	(1,1) = 4	(1,2) = 5
(2,0) = 6	(2,1) = 7	(2,2) = 8

כל תא במטריצה מאפיין תרחיש חוקי בבעיה.

נרוץ בלולאה קנונית כך:

```
for(int i = 0; i <= this.x ; i++) {
  for(int j = 0 ; j <= this.y; j++) {
```

כל שורה במטריצה המקורית מייצגת לנו תרחיש (קודקוד בגרף) שממנו יוצאות צלעות ומסומנות ב-1 במידה ויש קשר ישיר, לכן לכל תרחיש במטריצה "הדימיונית" שאיתה אנו רצים עם הלולאות נפעיל עליה את כל 6 האפשרויות לצלעות יוצאות שהצגנו מקודם.

לכן בכל איטרציה בבניית המטריצה שלנו נחשב תחילה מול איזה תרחיש אנחנו עומדים `k = getIndex(i,j)`

כאשר getIndex מחזיר: `(this.y+1)*i + j` כאשר במקרה שלנו נקבל $k = (2+1)\cdot 2 + 0 = 6$

לאחר מכן נציב 1 בכל 6 המקרים לצלעות יוצאות עבור תרחיש במיקום ה-k הזה.

```
    // Empty the 1st bottle
matrix[index][getIndex(0,j)] = 1;
    // Empty the 2nd bottle
matrix[index][getIndex(i,0)] = 1;
```

וכן הלאה....

אפשר להסיק כי:

$i = k/(col + 1)$ כי זה נותן לנו בדיוק את השלם. במקרה שלנו $i = 6/(2 + 1) = 6/3 = 2$
$j = k\%(col + 1)$ כי זה נותן לנו את שארית החלוקה. במקרה שלנו $j = 6\%(2 + 1) = 6\%3 = 0$

✪ פסאודו-קוד

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

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

         mat[node,getNode(b2,i,0)] ← 1
         mat[node,getNode(b2,0,j)] ← 1

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

         mat[node,getNode(b2 , i+j - min(i+j,b2) , min(i+j,b2))] ← 1
         mat[node,getNode(b2 , 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(b2, i, j):
   return (b2+1)*i + j
end-getNode()
```

✪ מימוש פתרון הבעיה

מימוש בגיטהאב עם טסט

```
public class WaterJug {

   int x, y, max, num_nodes;
   int[][] matrix;

   public WaterJug(int b1, int b2) {
       this.x = b1;
       this.y = b2;
       this.num_nodes = (b1+1)*(b2+1);
       this.matrix = new int[num_nodes][num_nodes];
```

```
        generate();
    }

    private int getIndex(int i, int j) {
        return (this.y+1)*i + j;
    }

    private void generate() {
        // java init arrays filled with '0'.
        for(int i = 0; i <= this.x ; i++) {
            for(int j = 0 ; j <= this.y; j++) {
                /*
                This index represents the tuple (i,j) of a scenario (node).
                 */
                int index = getIndex(i,j);

                /*
                Each scenario (node) could have at most 6 out-degrees edges.
                 */

                // Empty the 1st bottle
                matrix[index][getIndex(0,j)] = 1;
                // Empty the 2nd bottle
                matrix[index][getIndex(i,0)] = 1;

                // Filling the 1st bottle
                matrix[index][getIndex(x,j)] = 1;
                // Filling the 2nd bottle
                matrix[index][getIndex(i,y)] = 1;

                // i+j means the amount of water of the 2 bottles together.
                // Pouring 1st to the 2nd
                matrix[index][getIndex((i+j)-Math.min(y,i+j), Math.min(y,i+j))] = 1;
                // Pouring 2nd to the 1st
                matrix[index][getIndex(Math.min(x,i+j), (i+j)-Math.min(x,i+j))] = 1;
            } // end inner for
        } // end main for

        /* To deny self-point of each node */
        for(int i = 0 ; i < this.num_nodes; i++) {
            this.matrix[i][i] = 0;
        }
    }

    public void printMatrix() {
        System.out.print("      ");
        for (int i = 0; i <= x; i++) {
```

```
            for (int j = 0; j <= y; j++) {
                System.out.print("(" + i + "," + j + ") ");
            }
        }
        System.out.println();
        int k = 0, l = 0;
        for (int i = 0; i < this.num_nodes; i++) {
            System.out.print("(" + k + "," + l + ") ");
            for (int j = 0; j < this.num_nodes; j++) {
                System.out.print("  " + matrix[i][j] + "    ");
            }
            System.out.println();
            if(l == y) {
                k++;
                l = 0;
            }
            else {
                l++;
            }
        }

    }

    public static void main(String[] args) {
        int bottle1 = 1;
        int bottle2 = 2;

        WaterJug waterJug = new WaterJug(bottle1,bottle2);
        waterJug.printMatrix();
    }
}
```

בעיית פס השוקולד

✪ תיאור הבעיה

- יש לנו פס שוקולד בעל n קוביות.
- על כל חלוקה באינדקס k משלמים $k(n-k)$.
- עלינו לחלק את השוקולד לקוביות בודדות ולשלם את המינימום האפשרי.

בתרגול הצגנו כמה סוגי נסיונות, אחת מהן היא ניסיון שבירה לאחדות.
כלומר, עבור n=8 נבחר כל פעם ב- $k = n - 1$, לדוגמה:

$$7(8-7) = 7 \rightarrow 6(7-6) = 6 \rightarrow \ldots \rightarrow 1(2-1) = 1$$

ובעצם גילינו שהעלות שקיבלנו היא:

$$7 + 6 + 5 + 4 + 3 + 2 + 1 = \frac{8\cdot(8-1)}{2} = 28$$

לאחר מכן בוצע ניסוי על שבירה באמצע הפס (חצאים במידה וזוגי) וגם קיבלנו עלות של 28.
בניסיון האחרון בוצע ניסוי כך שנבחר k בכל חלוקה כך ש-k הוא מספר פיבונאצ'י וגם בניסוי הזה קיבלנו עלות של 28.

הגענו למסקנה שעל כל חלוקה אפשרית נצטרך לשלם $\frac{n(n-1)}{2}$.
נוסחה זו מוכרת גם בתור בעיית לחיצת הידיים וניתן לייצג אותה גם באופן קומבינטורי $\binom{n}{2}$.

כדי לוודא שהמסקנה נכונה, נוכיח אותה:

✪ הוכחה

נוכיח באינדוקציה על מספר הקוביות.

בסיס האינדוקציה: $n = 2$.

אם יש לנו 2 קוביות, יש רק פיצול אחד בעלות: $\frac{2(2-1)}{2} = 1$

הנחת האינדוקציה: הטענה נכונה עבור כל $k \leq n - 1$.

צעד האינדוקציה: נוכיח עבור $k = n$.

נפצל ב-k כלשהו כך ש- $1 \leq k \leq n - 1$, כל קטע יהיה בגודל קטן או שווה ל- $(n - 1)$.

עלות הפיצול תעלה לנו $k(n-k)$, לפי הנחת האינדוקציה:

קטע שמאל $[1,\ldots,k]$ יעלה לנו:

$$\frac{(k-1+1)(k-1)}{2} = \frac{k(k-1)}{2}$$

קטע ימין $[k+1,\ldots,n]$יעלה לנו:

$$\frac{(n-(k+1)+1)\cdot(n-(k+1))}{2} = \frac{(n-k)\cdot(n-k-1)}{2}$$

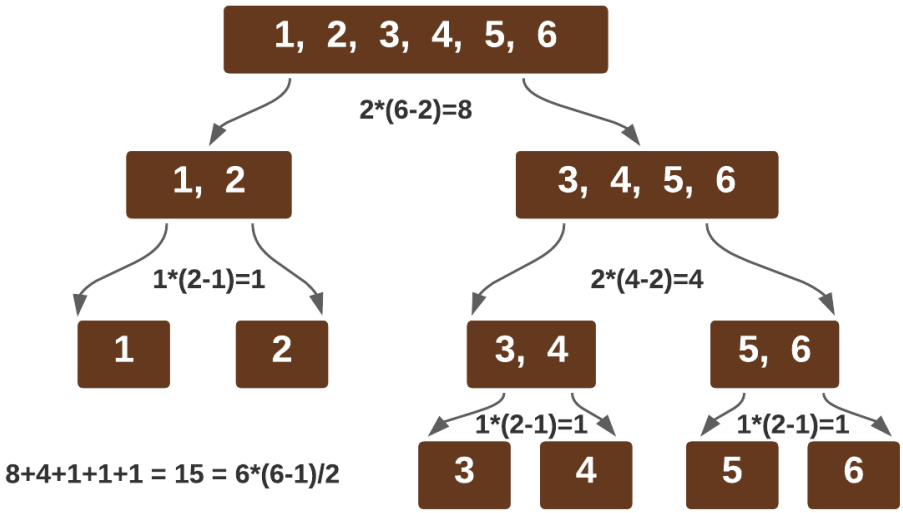
כלומר, העלות הכוללת תהיה:

$$k(n-k) + \frac{k(k-1)}{2} + \frac{(n-k)(n-k-1)}{2} = \ldots = \frac{n^2-n}{2} = \frac{n(n-1)}{2}$$

מימוש בגיטהאב

מימוש פתרון הבעיה

```java
public class DivideChocolateProblem {

   public static int divideChocolate(int n){
       if(n==2)
           return 1;
       if(n==1)
           return 0;

       int k = (int)(Math.random()*(n-1)) + 1;
       int a = divideChocolate(k); // left side
       int b = divideChocolate(n-k); // right side

       return k*(n-k) + a + b;
   }

   public static void main(String[] args) {
       int n = 8;
       System.out.println("For n="+n + " the cost is: "+divideChocolate(n));
   }
}
```



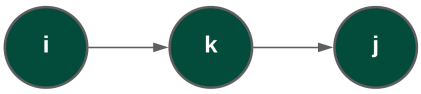
אלגוריתם Floyd-Warshall

✪ תיאור האלגוריתם

☚ אלגוריתם פלויד-וורשאל הוא אלגוריתם במדעי המחשב המשמש למציאת המסלולים הקצרים ביותר בין כל שני זוגות צמתים, בגרף ממושקל ומכוון.

☚ האלגוריתם פועל גם על גרפים שמכילים קשתות עם משקלים שליליים, בניגוד לאלגוריתם דייקסטרה.

☚ סיבוכיות זמן הריצה של האלגוריתם הוא $\Theta(V^3)$.

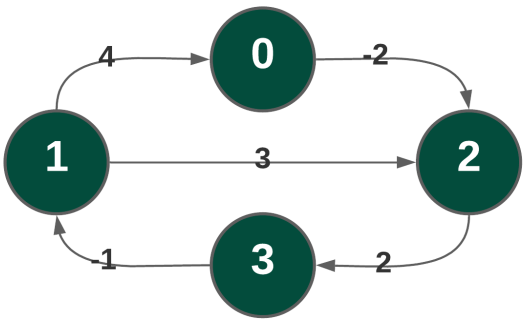


נתבונן באיור משמאל:
כדי לענות על השאלה האם קיים מסלול בין קודקוד i לקודקוד j,
האלגוריתם עובר על כל הקודקודים ובודק בצורה הבאה:
אם קיים מסלול מ-i ל-k ואם קיים מסלול מ-k ל-j אז קיים מסלול בין i ל-j.

✪ פסאודו-קוד

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

במצב התחלתי, ניצור מטריצה בגודל VxV, נציב בתוכה רק את הצלעות (כלומר רק את המשקלים של כל הצלעות, מקודקוד לקודקוד ולא מסלול). ערכים שאנחנו לא יודעים עדיין נייצג בתור "אינסוף".
מסלול מקודקוד לעצמו הוא 0, לכן גם נסמן את כל האלכסון באפסים.



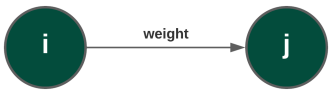
	0	1	2	3
0	0	∞	-2	∞
1	4	0	3	∞
2	∞	∞	0	2
3	∞	-1	∞	0

✪ הוכחה

נוכיח באינדוקציה.

בסיס האינדוקציה

עבור $M[i, j, 0]$יהיה פשוט המשקל שעל הצלע (i, j).



הנחת האינדוקציה

ננית כי $M[i, j, l]$הוא המסלול הקצר ביותר לכל $l < k$ונוכיח עבור המקרה $l = k$.

צעד האינדוקציה

יהא P_{ij} המסלול הקצר ביותר בין i ל-j מתוך הגרף$\{1, 2, ..., k\}$.

אם P_{ij}לא מכיל את k, אז הארך של המסלול P_{ij}הוא אותו ארך של מסלול המוגבל לקבוצת הקודקודים $\{1, 2, ..., k - 1\}$, ולפי הנחת האינדוקציה המסלול הקצר ביותר יהיה $M[i, j, k - 1]$.

אם P_{ij}מכיל את k, אז P_{ij}הוא שרשור של שני מסלולים P_{ik}ו-P_{kj}מקודקוד i ל-k וגם מ-k ל-j בהתאם.

מאחר וקודקוד k מופיע בשני המסלולים P_{ik}ו-P_{kj}כל הקודקודים של P_{ik}ו-P_{kj} חוץ מקודקודי i ו-j וגם k חייבים להיות מוגבלים לקבוצת הקודקודים$\{1, 2, ..., k - 1\}$.

כלומר, P_{ik}הוא מסלול i,k המוגבל בין $\{1, 2, ..., k - 1\}$וגם P_{kj}הוא מסלול k,j המוגבל בין $\{1, 2, ..., k - 1\}$.

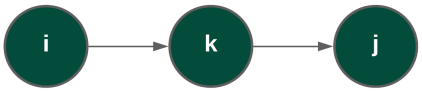
נשים לב ששני המסלולים P_{ik}ו-P_{kj}הם האופטימלים ביותר, כי אחרת היינו סותרים את אופטימליות של P_{ij}.

לפי הנחת האינדוקציה, הארך של המסלול הקצר ביותר נמצא ב-$M[i, k, k - 1]$וגם $M[k, j, k - 1]$,

לכן נקבל בהתאם:

$$M[i, j] = M[i, k, k - 1] + M[k, j, k - 1]$$

מאחר שההגדרה הרקורסיבית שלנו שוקלת את שני המקרים ובוחרת את המינימום, מכאן נובע כי $M[i, j, k]$הוא המסלול הקצר ביותר.



מימוש בגיטהאב

מימוש פתרון הבעיה

```
public class FloydWarshallAlgorithm {
   final static int INF = 999999;

   public static void FloydWarshall(int[][] matrix) {
       for(int k = 0 ; k < matrix.length; k++) {
           for(int i = 0 ; i < matrix.length; i++) {
               for(int j = 0 ; j < matrix.length; j++) {
                   if(matrix[i][j] > matrix[i][k] + matrix[k][j]) {
                       matrix[i][j] = matrix[i][k] + matrix[k][j];
                   }
               }
           }
       }
       printMatrix(matrix);
   }


   public static void printMatrix(int[][] matrix) {
       System.out.println(Arrays.deepToString(matrix)
               .replace("],","\n").replace(",","\t| ")
               .replaceAll("[\\[\\]]", " "));
   }

   public static void main(String[] args) {
       int[][] matrix =
               {
               {0,  INF, -2,INF},
               {4,    0,  3,INF},
               {INF,INF,  0,  2},
               {INF, -1,INF,  0}
               };
       FloydWarshall(matrix);
   }
}
```

פונקציה רקורסיבית

$$FW(i,j,k) = min(\, FW(i,j,k-1) \;,\; FW(i,k,k-1) \;+\; FW(k,j,k-1)\,)$$

בתרגול הציגו את Floyd-Warshall כאלגוריתם שעובד על גרף לא מכוון, לא משוקלל, שעובד באופן **בוליאני**.
כלומר, אם קיים מסלול בין 2 קודקודים נסמן ב-True, אחרת נסמן ב-False.
לכן נגדיר את הפונקציה בצורה הבאה:

$$matrix[i][j] \;=\; matrix[i][j] \;||\; (matrix[i][k] \;\&\&\; matrix[k][j])$$

ה-OR כאן מתייחס למקרים בו קיים כבר מסלול בין i ל-j.
לבסוף כשנקבל את התוצר הסופי ואז נוכל לקבל תשובות כמו האם קיים מסלול בין קודקוד i ל-j.

✪ האם הגרף קשיר (גרף לא מכוון)?

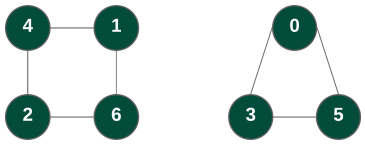
אם כל תא במטריצה מסומן ב-true אז בהכרח הגרף קשיר.

- אפשר לעבור על לולאה כפולה ולבדוק האם הכל - true: $O(n^2)$.
- אבל אפשר לייעל את הפתרון ב-$O(n)$:

נתבונן בשורה הראשונה של המטריצה, אם כל השורה מסומנת ב- true אז הרי שקיים מסלול מקודקוד 1 לבין כל שאר הקודקודים בגרף (גרף לא מכוון) ולכן הגרף קשיר.

כמה רכיבי קשירות יש בגרף לא מכוון?

	0	1	2	3	4	5	6
0	T			T		T	
1		T	T		T		T
2		T	T		T		T
3	T			T		T	
4		T	T		T		T
5	T			T		T	
6		T	T		T		T



נעבור בלולאה כפולה ונעזר ב-counter.
כל איטרציה חדשה שמצביעה על המערך מייצגת מצביע על קודקוד מסוים. במידה ויש 0 נדע שעוד לא הגענו לקודקוד הזה, כלומר זה רכיב קשירות חדש.
לכן נעשה Counter++ ונציב את ערך ה-counter בתא הזה. נעבור על כל הקודקודים שאיתו ברכיב הקשירות וגם להם נציב counter ללא counter++.

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

תוצר סופי:

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

Counter = 2

✪ מימוש מספר רכיבי הקשירות (גרף לא מכוון)

המימוש מתבצע כך שהמטריצה כבר פתורה, כלומר בוצעה כבר הפונקציה **FloydWarshall**:

השיטה יודעת להבדיל בין רכיבי הקשירות ומאחסנת נתונים במפה כך שהמפתח הוא מספר הצומת והערך הוא מספר רכיב הקשירות.

```
public HashMap<Integer, Integer> generateComponents() {
   int counter = 0;
   HashMap<Integer, Integer> nodes = new HashMap<>();

   for(int i = 0; i < matrix.length; i++) {
       if(!nodes.containsKey(i)) {
           nodes.put(i,++counter);
       }
       for(int j = i+1; j < matrix.length; j++) {
           if(matrix[i][j] && !nodes.containsKey(j)) {
               nodes.put(j,counter);
           }
       }
   }
   return nodes;
}
```

✪ איך נדע האם 2 קודקודים נמצאים על אותו המסלול?

הגרף לא מכוון אז מספיק לבדוק האם 2 הקודקודים נמצאים באותו רכיב הקשירות

לכן נעזר בפונקציה **generateComponents**.

```
public boolean checkPath(int v1, int v2) {
   HashMap<Integer, Integer> nodes = generateComponents();
   return nodes.containsKey(v1) && nodes.containsKey(v2)
           && nodes.get(v1).equals(nodes.get(v2));
}
```

✪ סיכום סיבוכיות עבור אלגוריתם Floyd-Warshall

- חישוב מטריצה: $O(V^3)$
- בדיקת קשירות: $O(V)$
- מספר רכיבי קשירות: $O(V^2)$
- קבלת כל רכיבי הקשירות במבנה נתונים: $O(V^2)$
- בדיקת מסלול בין 2 קודקודים - משתמשת בשיטה אחרת שעולה $O(V^2)$.

✪ ניתן לפתור בעיות בנושא בעיית הבקבוקים בעזרת אלגוריתם Floyd Warshall

קובץ פתרון נמצא בגיטהאב WaterJugFloydWarshall.java

✪ פסאודו-קוד (מספר רכיבי קשירות)

```
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
```

✪ פסאודו-קוד (בדיקה אם 2 צמתים על אותו רכיב הקשירות)

```
onSamePath(mat[N,N], src, dest):

create comp[N] ⇐ getNumberOfComponents(mat)

if comp[src] = comp[dest] then:
   return True

return False
```

✪ אלגוריתם Floyd Warshall עם משקלים על הקודקודים

קלט

נתון מערך חד מימדי L, האינדקס במערך הוא הקודקוד והערך הוא המשקל שעל הקודקוד.

נתון בנוסף מערך דו-מימדי של מטריצת שכנויות כאשר:

(1= קיים קשר ישיר, 0=קודקוד שמצביע על עצמו, ∞= משקל לא מוגדר).

פלט

מטריצת המסלולים הקצרים ביותר בין הקודקודים.

כדי לפתור בעיה זאת נתאים אותה לשיטת הפתרון המקורית של האלגוריתם שנלמד בהתחלה.

כלומר לדוגמה, אם בקודקוד i המשקל הוא 10 ובקודקוד j המשקל הוא 5 אז המשקל על הצלע $(i,j) = 10 + 5 = 15$.

נתבונן באיור, כעת נרצה לחשב את אורך המסלול בין V1 ל-V3.

אז פשוט נחשב כך: V1+V2+V2+V3 , אבל נשים לב כי V2 מופיע פעמיים בסכימה של אורך המסלול.

כלומר, עבור כל קודקוד שהוא לא המקור או היעד הוא מופיע בסכום פעמיים.

אז מה הפתרון למקרה זה? -לכן כדי לקיים שוויון הגיוני, נוסיף לסכימה פעם נוספת את המקור והיעד כלומר:

$2 \cdot V1 + 2 \cdot V2 + 2 \cdot V3$, כעת כל מה שנותר הוא לחלק את הסכום שקיבלנו וזה יהיה האורך של המסלול הקצר ביותר, כלומר:

$$\frac{2 \cdot V1 + 2 \cdot V2 + 2 \cdot V3}{2} = \frac{2(V1 + V2 + V3)}{2} = V1 + V2 + V3$$

✪ פסאודו-קוד (משקלים על קודקודים)

```
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
```

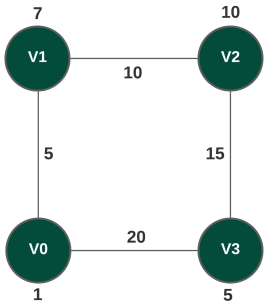
```
end-for

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
```

✪ מימוש בג'אווה

```
public class FW_WeightsOnVertices {
   static final int INF = 9999999;

   public static void main(String[] args) {
       int[] values = {1,7,10,5};
       int[][] matrix = {
               {0,1,INF,1},
               {1,0,1,INF},
               {INF,1,0,1},
               {1,INF,1,0}
       };

       System.out.println("GIVEN ORIGINAL MATRIX: (* = infinity): ");
       print(matrix);
       FloydWarshall(matrix,values);
   }// END main()

   public static void FloydWarshall(int[][] matrix, int[] values) {

       System.out.println("SET WEIGHTS ON EDGE NEIGHBORS BY (Vi + Vj): ");
       for(int i = 0; i < matrix.length; i++) {
           for(int j = 0; j < matrix.length; j++) {
               if(matrix[i][j] == 0) {
                   matrix[i][j] = values[i];
               }
               else if(matrix[i][j] == 1) {
                   matrix[i][j] = values[i] + values[j];
               }
           }
       }
       print(matrix);

       System.out.println("FOLLOWING FLOYD WARSHALL ALGO (WITH WRONG WEIGHTS): ");
       for(int k = 0 ; k < matrix.length; k++) {
           for(int i = 0 ; i < matrix.length; i++) {
```

```
                for(int j = 0 ; j < matrix.length; j++) {
                    if(i != j) {
                        matrix[i][j] = Math.min(matrix[i][j], matrix[i][k] +
matrix[k][j]);
                    }
                }
            }
        }
        print(matrix);

        System.out.println("FIXING WRONG WEIGHTS BY (V[SRC] + MATRIX[SRC][DST] +
V[DST])/2 ");
        for(int i = 0 ; i < matrix.length; i++) {
            for (int j = 0; j < matrix.length; j++) {
                if(i != j) {
                    matrix[i][j] = (values[i] + matrix[i][j] + values[j])/2;
                }
            }
        }
        print(matrix);
    } // END OF FloydWarshall() METHOD

}
```

✪ אלגוריתם Floyd Warshall עם משקלים על הקודקודים וגם על הצלעות



דוגמה

כלומר אם נרצה לחשב את המסלול בין v0 לבין v1 אז נקבל:

$matrix[0][1] = 1 + 5 + 7 = 13$.

אם נרצה לחשב למשל את המסלול בין V0 לבין V2 אז נקבל:

$matrix[0][2] = 1 + 5 + 7 + 10 + 10 = 33$.

רעיון והוכחת נכונות (מתוך ספר המבחנים)

אלגוריתם $Floyd\ Warshall$ מחזיר את המרחקים מכל קודקוד לכל קודקוד בגרף.

הוא מקבל מטריצת מרחקים ישירים: מרחק מקודקוד לעצמו יהיה 0, מרחק באורך 1 יהיה משקל הצלע המחברת בין שני קודקודים, ומרחק קודקודים שלא מחוברים הוא ∞.

אלגוריתם $Floyd\ Warshall$ עובד על משקלים בצלעות.

נמיר את התרגיל שלנו לקלט שאלגוריתם $Floyd\ Warshall$ ידע לטפל בו, כך ש:

$$D(u,v) = 2w(u,v) + w(u) + w(v)$$

כך ש-D היא מטריצת המרחקים ו-w זה משקל הצלע/הקודקוד.

באלגוריתם שלנו, הקלט ל-$Floyd\ Warshall$הוא סכום משקלים הקודקודים ומשקל הצלע.
כך שאם יש מסלול $v_1 \to v_2 \to v_3 \to v_4$אז סכום המשקלים יהיה:

$$w(v_1) + 2w(v_1, v_2) + w(v_2) + w(v_2) + 2w(v_2, v_3) + w(v_3) + w(v_3) + 2w(v_3, v_4) + w(v_4)$$
$$= w(v_1) + 2w(v_1, v_2) + 2w(v_2) + 2w(v_2, v_3) + 2w(v_3) + 2w(v_3, v_4) + w(v_4)$$

כדי לקבל את המרחק האמיתי, נוסיף את הקודקודים החיצוניים ונחלק בשתיים.

$$\frac{2w(v_1)+2w(v_1,v_2)+2w(v_2)+2w(v_2,v_3)+2w(v_3)+2w(v_3,v_4)+2w(v_4)}{2}$$
$$= w(v_1) + w(v_1, v_2) + w(v_2) + w(v_2, v_3) + w(v_3) + w(v_3, v_4) + w(v_4)$$

הערה

הצלעות לא נסכמות פעמיים במהלך הסכימה בדיוק כמו שהקודקוד הראשון והאחרון לא נסכמים, בשונה מהאלגוריתם הקודם של משקלים על קודקודים, כאן לא יהיה אפשרי להוסיף בסוף האלגוריתם שוב את כל הצלעות ואז לחלק ב-2 כיוון שאי אפשר לדעת לאורך הסכימה איזה צלעות עברנו, ולכן כבר באיתחול המטריצה נסכום את הצלעות פעמיים מראש.

סדר פעולות

1. באתחול:

$$temp[i][j] = 2 \cdot matrix[i][j] + values[i] + values[j]$$

2. נפעיל את אלגוריתם FW– $O(n^3)$:
3. נתקן בהתאם $O(n^2)$:

$$solution[i][j] = \frac{(temp[i][j] + values[i]+values[j])}{2}$$

✪ פסאודו-קוד (משקלים על קודקודים ועל הצלעות)

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

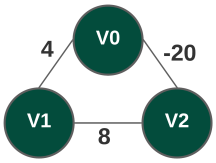
✪ מימוש בג'אווה (משקלים על קודקודים ועל הצלעות)

```
public class FW_WeightsOnEdgesAndVertices {
   static final int INF = 9999999;

   public static void main(String[] args) {
       int[] values = {1, 7, 10, 5};
       int[][] matrix = {
               {0, 5, INF, 20},
               {5, 0, 10, INF},
               {INF, 10, 0, 15},
               {20, INF, 15, 0}
       };
       System.out.println("GIVEN ORIGINAL MATRIX: (* = infinity): ");
       print(matrix);
       int[][] solution = FloydWarshall(matrix,values);
   }
   public static int[][] FloydWarshall(int[][] matrix, int[] values) {
       int[][] temp = new int[matrix.length][matrix.length];

       System.out.println("SET AND COPY TO NEW MATRIX THE FOLLOWING FORMULA: ");
       System.out.println("( TEMP[I][J] = 2*MATRIX[I][J] + VALUES[I] + VALUES[J]
):");
       for(int i = 0; i < matrix.length; i++) {
           for(int j = 0; j < matrix.length; j++) {
               if(i==j) {
                   temp[i][j] = values[i];
               }
               else if(matrix[i][j] > 0 && matrix[i][j] != INF) {
                   temp[i][j] = 2*matrix[i][j] + values[i] + values[j];
               }
               if(matrix[i][j] == INF) {
                   temp[i][j] = INF;
               }
           }
       }
       print(temp);

       System.out.println("FOLLOWING FLOYD WARSHALL ALGO (WITH WRONG WEIGHTS): ");
```

```
        for(int k = 0 ; k < matrix.length; k++) {
            for(int i = 0 ; i < matrix.length; i++) {
                for(int j = 0 ; j < matrix.length; j++) {
                    if(i != j) {
                        temp[i][j] = Math.min(temp[i][j], temp[i][k] + temp[k][j]);
                    }
                }
            }
        }
        print(temp);

        System.out.println("FIXING WRONG WEIGHTS BY (V[SRC] + TEMP[SRC][DST] +
V[DST])/2 ");
        int[][] solution = new int[matrix.length][matrix.length];
        for(int i = 0 ; i < matrix.length; i++) {
            for (int j = 0; j < matrix.length; j++) {
                if(i != j) {
                    solution[i][j] = (values[i] + temp[i][j] + values[j])/2;
                } else {
                    solution[i][j] = values[i];
                }
            }
        }
        print(solution);
        return solution;
    } // END OF FloydWarshall() METHOD
```

✪ גרפים עם מעגלים שליליים באלגוריתם Floyd-Warshall

גרף נקרא עם מעגל שלילי אם קיים מסלול מעגלי $(v_i \to v_i)$ שסכום המשקלים שלו קטן ממש מאפס.



גרף לא מכוון

עבור גרף לא מכוון כמו באיור משמאל קיים מעגל שלילי $(v_0 \to v_0)$כלומר,

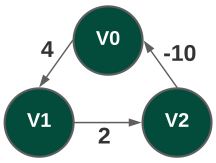
$0 > (-8) = 20 + 8 - 4$.

מאחר שהגרף לא מכוון, מספיק שיש צלע שלילית אחת כדי שיהיה מעגל שלילי בגרף הנתון.

למה? - למשל עבור האיור מלמעלה קיים מעגל שלילי: $v_0 \to v_2 \to v_0 = (-20) + (-20) = (-40) < 0$

כלומר, לכל צלע שלילי בגרף לא מכוון, קיים מעגל שלילי.

לכן מספיק לנו לעבור בלולאה אחת על כל הצלעות ולבדוק משקל שלילי (סה"כ $O(n^2)$).



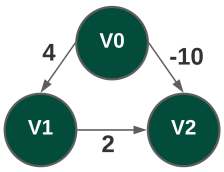
גרף מכוון

עבור גרף מכוון כמו באיור משמאל קיים מעגל שלילי $(v_0 \to v_0)$כלומר,

$0 > (-4) = 10 + 2 - 4$.

נשים לב כי יכול להיות מקרים בהם אין מעגלים כלל בגרפים מכוונים כמו למשל באיור משמאל:

לכן עבור גרף מכוון, לא מספיק לנו לבדוק האם קיימת צלע עם משקל שלילי בגרף.



הפתרון לבעיה זאת היא **להפעיל את אלגוריתם פלויד-וורשל** ואחרי זה **לבדוק באלכסון** המטריצה האם קיים ערך שלילי, כלומר נבדוק האם קיים מעגל $(v_i \to v_i)$ שסכום המשקלים של המסלול שלו קטן ממש מאפס.

מימוש בגיטהאב

מימוש פתרון הבעיה (פסאודו-קוד)

```
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

✪ תיאור הבעיה

☚ בהינתן מערך בעל n איברים.

☚ מהו תת המערך שסכום איבריו הוא הגדול ביותר ומספר האיברי תת המערך הוא המינימלי ביותר?

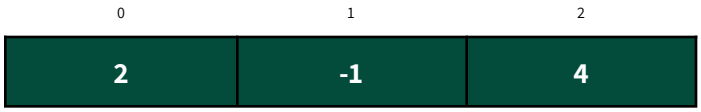
☚ תת-המערך חייב להיות רצף של תאים.

☚ אם כל התאים במערך חיוביים אז ניקח את כל המערך עצמו.

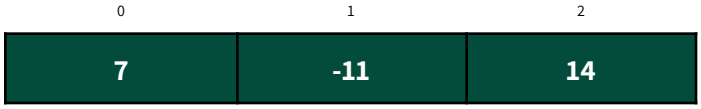
☚ נברר לגבי מקרים בהם קיים תא אחד לפחות שיש בו ערך שלילי.

✪ דוגמה

עבור המערך הבא:



משתלם לנו לקחת תת-מערך שיש בו את הערך (1-) כי הוא לא מאפס לגמרי את (2) או גורם לסכום שלילי.
אבל עבור המערך הבא:



לא משתלם לנו לקחת את (11-) מכיוון שהוא מקטין את 7 ועוד מכניס את הסכום למינוס וזה רע.
לכן נסיק מסקנות מהמקרים שהצגנו כאן לטובת המימוש בהמשך.

✪ פתרון א' (תכנות דינאמי)

☚ נזכר במשחק המספרים מאלגוריתמים 1, במימוש הבעיה בשיטה הדינאמית הצגנו את כל ההפרשים בין כל תתי-המשחקים על מערך דו-מימדי.

☚ גם בשיטת הפתרון הזו נפתור באופן דיי דומה.

☚ נבנה מטריצה בגודל $n \times n$.

☚ נציב באלכסון המטריצה כל ערך במערך בהתאם למיקומו.

☚ עבור כל תא i ו-j (המייצגים תת מערך כלשהו) נסכום ונציב את הסכום במיקום המתאים.

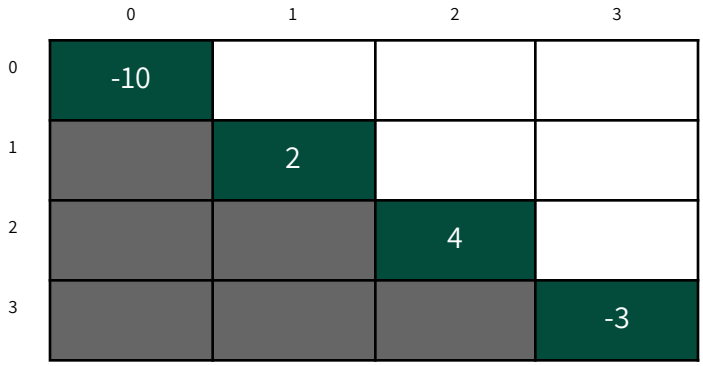
☚ כל תא במטריצה מייצג תת-מערך שערכו הוא הסכום.

☚ לאחר מכן אפשר למצוא את תת-המערך עם הסכום הגדול ביותר (אפשר גם במקביל).

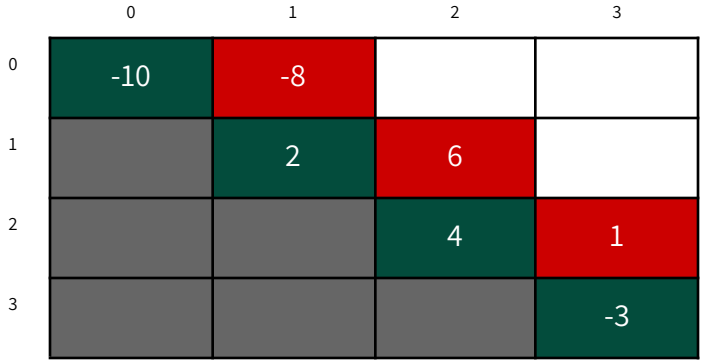
דוגמה

בהינתן המערך הבא:

ניצור מטריצה ונציב באלכסון את הערכים במערך, כל אחד במקומו:



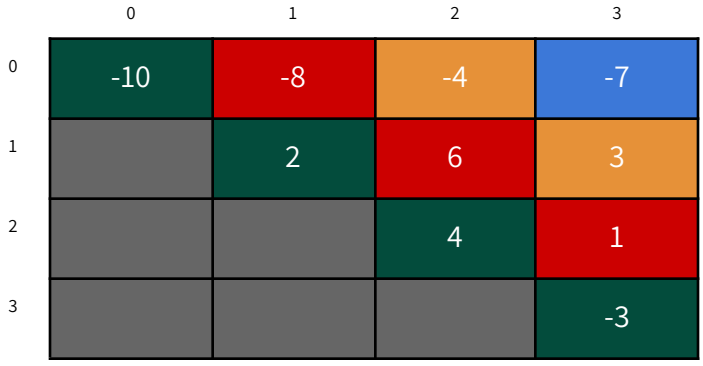
לאחר מכן נעבור בלולאה כפולה ונחשב עבור כל i ו-j את סכום תת המערך המייצג את האינדקסים,
למשל עבור $mat[0][1]$מדובר בתת המערך בו $[-10, 2]$נמצאים ולכן נציב 8- :



בשלב הבא כבר זה נהיה קצת שונה, כי אם נבקש לסכום את $mat[0][2]$לא נסכום את $[-8, 6]$מכיוון כי הסכומים הלאה נוצרו באופן הבא: $-10 + 2 + 2 + 4 = -8 + 6$, ספרנו את הערך 2 פעמיים.

לכן בדומה למימוש של משחק המספרים נחשב באופן הבא:

$$mat[i][j] = mat[i + 1][j] + mat[i][i]$$



ולכן נבחר בתת המערך $mat[1][2] = 6$.

מימוש בגיטהאב

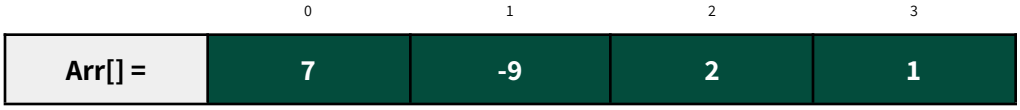
מימוש פתרון הבעיה

```java
public static int maxSub(int[] arr) {
   int[][] mat = new int[arr.length][arr.length];
   int positiveCaseSum = 0;
   boolean positiveCase = true;

   for(int i = 0; i < mat.length; i++) { // O(N)
       mat[i][i] = arr[i];
       if(arr[i] > 0) {
           positiveCaseSum += arr[i];
       } else {
           positiveCase &= false;
       }
   }
   if(positiveCase) {
       return positiveCaseSum;
   }

   int max = Integer.MIN_VALUE;

   for(int i = mat.length-2 ; i >= 0 ; i--) { // O(N^2)
       for(int j = i+1 ; j < mat.length; j++) {
           mat[i][j] = arr[j] + mat[i][j-1];
           if(mat[i][j] > max) {
               max = Math.max(Math.max(mat[i][j], arr[i]),arr[j]);
           }
       }
   }
   return max;
}

public static void main(String[] args) {
   int[] arr = {-10,2,4,-3};
   System.out.println(maxSub(arr));
}
```

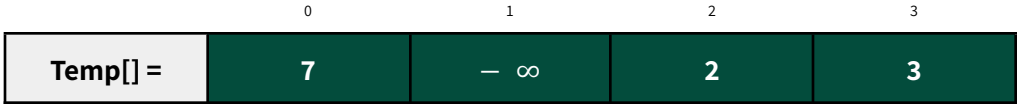
סה"כ **סיבוכיות זמן** הריצה עבור פתרון א' הוא $O(n^2)$.

✪ פתרון ב' - זמן לינארי $O(n)$

נתבונן במערך arr הבא:



1. ניצור מערך עזר Temp
2. נעבור בלולאה פעם אחת על כל איברי המערך המקורי Arr
3. עבור כל מעבר של תא במערך המקורי נסכום אותו ואת כל הסכום שנצבר לפניו ונציב את הסכום במערך Temp
4. במידה והסכום יהיה אפס או שלילי נציב במערך העזר $-\infty$ ונתחיל לסכום מחדש החל מהאיבר הבא אחריו.



במערך Temp שקיבלנו יש את תת-הקטע עם הסכום המקסימלי שאנחנו צריכים כולל אינדקס התחלה וסוף של הקטע.

✪ פסאודו-קוד (Best לינארי)

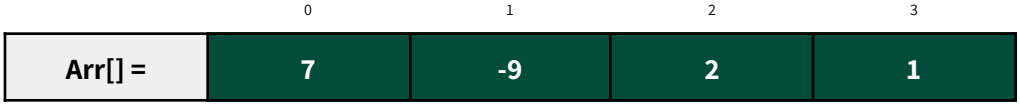
```
bestLinear(arr[N]):
sum ⇐ − ∞
temp_sum ⇐ 0
start_index ⇐ 0
temp_start ⇐ 0
end_index ⇐ 0

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
```

✪ מימוש בג'אווה

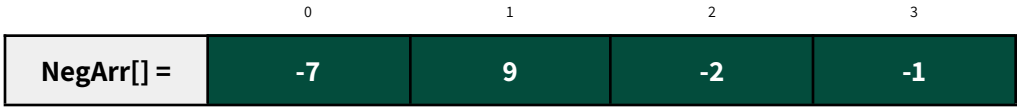
```java
public static int[] bestLinear(int[] arr) {
   int maxSum = Integer.MIN_VALUE, tempSum = 0;
   int startIndex = 0, endIndex = 0, startTemp = 0;
   int[] temp = new int[arr.length];

   for (int i = 0; i < arr.length; i++) { // O(N)
       tempSum += arr[i];
       if(tempSum > maxSum) {
           maxSum = tempSum;
           startIndex = startTemp;
           endIndex = i;
       }
       if(tempSum < 0) {
           tempSum = 0;
           startTemp = i + 1;
       }
       temp[i] = tempSum;
   }
   System.out.println(Arrays.toString(temp));

   return new int[] {maxSum, startIndex, endIndex};
}

public static void main(String[] args) {
   int[] sol = bestLinear(new int[]  );
   System.out.println("Max = "+sol[0] +", Start = " + sol[1] + ", End = "+sol[2]);
}
```

✪ פתרון ג' - עבור מערך מעגלי בזמן $O(n)$



אם נפעיל את פתרון ב' נקבל שהמקסימום הוא 7.
אבל אם מדובר במערך מעגלי זה כבר לא יהיה נכון מאחר $2 + 1 + 7 = 10$.

כדי למצוא פתרון בשיטה המעגלית נוכל לקחת את הסכום הכולל ולהפחית ממנו את הסכום המינימלי.
לכן ניצור מערך חדש בו יהיו כל הערכים ההפוכים של המערך המקורי:



תוך כדי ההעתקה למערך החדש נסכום גם את סכום כל המערך המקורי.
למה אנחנו הופכים את המערך? - אם נמצא את המקסימום של המערך ההופכי הוא הרי יהיה המינימום במערך המקורי.
זה נעשה כדי למנוע מצב שצריך לכתוב אלגוריתם חדש שמוצא את המינימום במקום את המקסימום.
ואז נחשב בנוסחה הבאה שתספק את הסכום המקסימלי ביותר עבור מערך מעגלי: $Sum - (-Best(-A))$.

במקרה שלנו נקבל $Sum - (-Best(-A)) = 1 - (-9) = 1 + 9 = 10$, וקיבלנו 10 כי באמת $2 + 1 + 7 = 10$.

כמובן שיכול להיות מקרה שהשיטה הלא מעגלית תביא לנו תוצאה גדולה יותר מתוצאה של שיטה מעגלית.
לכן, לפתרון נעשה את המקסימום שקיבלנו בין 2 השיטות: $Max(Best(A), Sum - (-Best(-A)))$

✪ פסאודו-קוד (Best מעגלי)

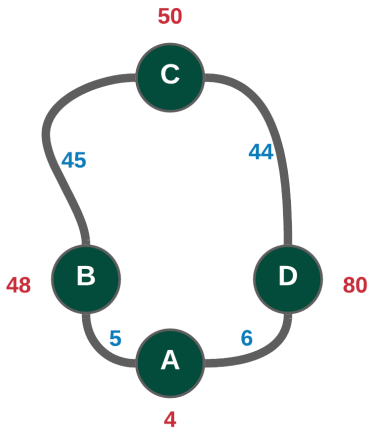
```
bestCycle(arr[N]):
create neg_arr[N]
sum ⇐ 0

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
```

✪ מימוש בג'אווה (Best מעגלי)

```
public static int[] best(int[] arr) { // O(3N)
   int arrSum = 0;
   int[] NegArr = new int[arr.length];

   for(int i = 0 ; i < arr.length; i++) { // O(N)
       arrSum += arr[i];
       NegArr[i] = (-1)*arr[i];
   }

   int[] regular = bestLinear(arr); // O(N)
   int[] negative = bestLinear(NegArr); // O(N)
   int cycleMax = arrSum - (-negative[0]);

   if(regular[0] >= cycleMax)
       return regular;

   return new int[] {cycleMax, (negative[2] + 1)%arr.length, negative[1] - 1};
}
```

בעיית תחנות הדלק



✪ תיאור הבעיה

☚ נתון מסלול עם תחנות דלק (הצמתים).

☚ בכל תחנה ניתן למלא כמות ליטרים מסוימת.

☚ בנסיעה בין כל 2 תחנות הרכב צורך כמות דלק מסויימת (הצלעות).

☚ האם ניתן להתחיל בתחנה מסוימת, לעבור בכל התחנות בדרך ולבצע סיבוב שלם בלי שייגמר לנו הדלק?

✪ דוגמה

נסתכל בציור משמאל.

אם נרצה לחשב את המסלול המתחיל מקודקוד A לעצמו, נתחיל עם 4 ליטר במיכל ונתקל במצב בו לא נוכל להגיע לתחנה של B או של D כי אין לנו מספיק דלק.

אם נרצה לחשב את המסלול המתחיל מקודקוד C לעצמו, נתחיל עם 50 ליטר במיכל, אם נמשיך להשלים את המסלול מקודקוד D או B, באמת נצליח לנסוע סיבוב שלם ללא מחסור בדלק.

עבור המקרה הכללי, צריך תחילה לבדוק שכמות התדלוק הכוללת של כל התחנות, גדולה יותר מכמות הדלק הנצרך בכל הגרף הנתון.

עבור מימוש פתרון הבעיה, נקבל כקלט 2 מערכים:

1. מערך של התחנות (הצמתים) : $A[] = \{50, 80, 4, 48\}$
2. מערך של צריכת הדלק (הצלעות): $B[] = \{44, 6, 5, 45\}$

עבור $a_i \in A,\ b_i \in B$ צריך לוודא ש: $\sum_{i=0}^{n-1} a_i \geq \sum_{i=0}^{n-1} b_i$.

אי-השוויון רק מבטיח שקיים מסלול מעגלי המקיים את תנאי הבעיה.

ניצור מערך C המקיים שכל תא $c_i \in C$ במערך יהיה $c_i = a_i - b_i$ בהתאם.

מהות המערך היא לבדוק האם אפשר לנסוע מתחנה אחת לתחנה השכנה שלה.

עבור הדוגמה שלנו: $C[] = \{6, 74, -1, 3\}$.

- אם נסכום את כל המערך של C והסכום הוא לא שלילי, זה מוודא שקיים מסלול חוקי בגרף (כמו תנאי האי-שוויון).

✪ הצעת פתרון לבעיה

נחפש את תת המערך עם סכום מקסימלי ביותר ב-$C[]$ כדי שנוכל להמשיך בדרך.

נשלח את המערך $C[]$ לאלגוריתם $Maximum\ Subarray\ (BEST)$ שנלמד (עבור מעגלי ולא מעגלי) ונקבל:

שהסכום הגדול ביותר הוא 82, צריך להתחיל מאינדקס 3 (תחנה B באיור).

ומכאן מומלץ להתחיל מתחנה B ולהשלים ממנה מעגל שלם.

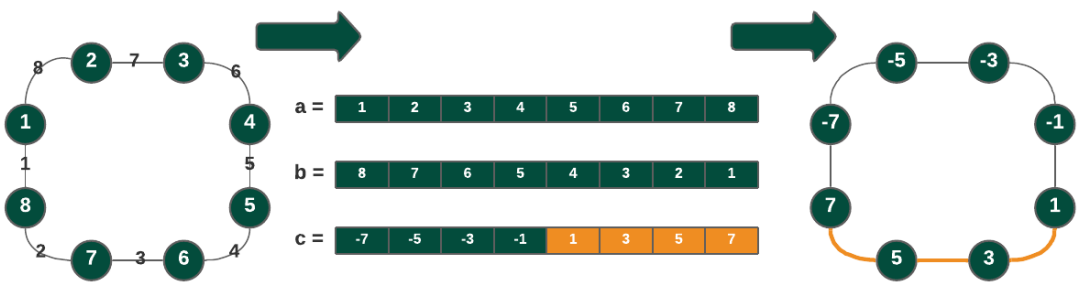
✪ פסאודו-קוד

```
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)
```

מימוש בגיטהאב

מימוש פתרון הבעיה

```
public static int[] GasStation(int[] a, int[] b) {
   int[] c = new int[a.length];
   int sumAB = 0;

   for(int i = 0; i < c.length; i++) { // O(N)
       c[i] = a[i]-b[i];
       sumAB += c[i];
   }
   if(sumAB < 0) {
       return new int[] {-1,-1,-1};
   }

   return best(c);
}
```



מציאת תת-מטריצה עם סכום מקסימלי

✪ תיאור הבעיה

☚ בהינתן מטריצה בגודל $(n \ x \ m)$.

☚ מהי תת המטריצה שסכום איבריה הוא הגדול ביותר?

✪ דוגמה

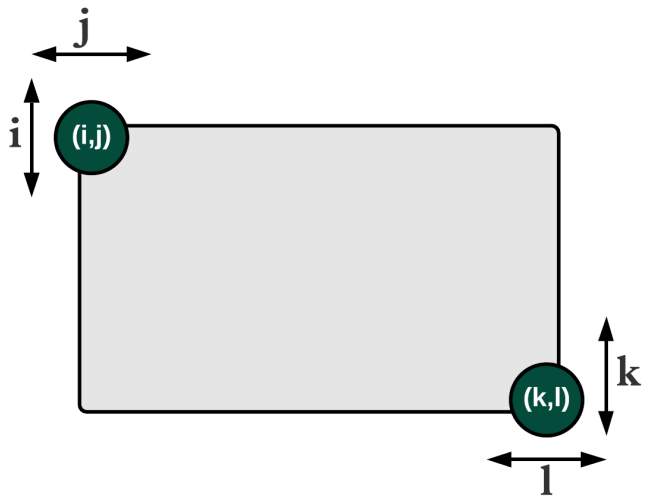
עבור המטריצה הבאה, תת המטריצה בעלת הסכום הגדול ביותר מסומנת באדום:

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

✪ הגדרה כללית לפתרונות

עבור 2 הפינות במלבן אפשר לייצג כל צורה מלבנית בהתאם לגבולות המלבן וככה אפשר לייצג כל "תת-מטריצה".

למשל, בדוגמה מלמעלה, המלבן המקסימלי הוא ($(3, 3)$, $(1, 1)$)



✪ פתרון א' - חיפוש שלם

☚ נסכום כל תת-מטריצה אפשרית.

☚ נמצא את המקסימלי ונשמור את הסכום והאינדקסים i, j, k, l שמייצגים אותו.

☚ על כל תת-מטריצה (כל המלבן האפור בהגדרה) נעבור בפנים בלולאה כפולה $[x][y]$ונסכום את כל הערכים.

מימוש בגיטהאב

מימוש פתרון הבעיה

```
public static int[] maxSumMatrix(int[][] mat) {

       int n = mat.length, m = mat[0].length;
       int maxSum = 0;
       int ii = 0, jj = 0, kk = 0, ll = 0;

       for(int i = 0 ; i < n ; i++) {
           for(int j = 0 ; j < m; j++) {
               for(int k = i ; k < n; k++) {
                   for(int l = j ; l < m; l++) {
                       int sum = 0;
                       for(int x = i ; x <= k  ; x++) {
                           for(int y = j ; y <= l ; y++) {
                               sum += mat[x][y];
                           }
                           if(sum> maxSum) {
                               maxSum = sum;
                               ii = i;
                               jj = j;
                               kk = k;
                               ll = l;
                           }
                       }
                   }
               }
           }
       }

       return new int[] {maxSum, ii, jj, kk, ll};
 }
```

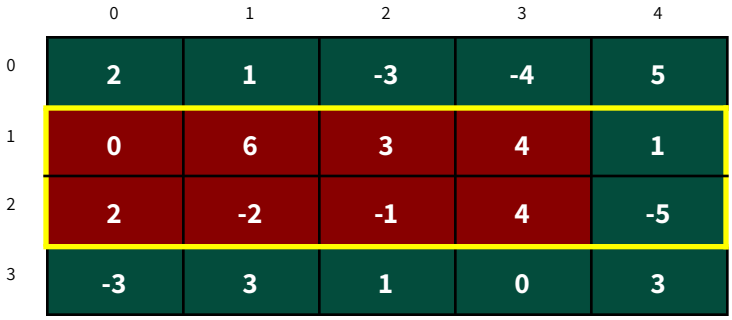
סיבוכיות זמן ריצה - $O(n^6)$

✪ פתרון ב' - מערך עזר

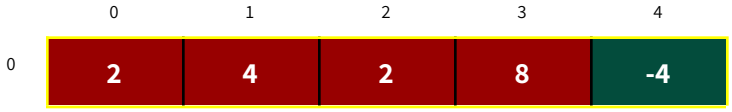
☚ בפתרון זה נוכל להשתמש ב-Best כדי שיעזור לנו.

☚ נייצר את כל תתי השורות ואת כל תתי העמודות, נפעיל Best ונדע את התשובה.

לדוגמה:



אם נחבר את שורות 1 ו-2 נקבל את:



כשנפעיל עליו את Best הוא יחזיר לנו את הסכום 16 ואת אינדקס ההתחלה 0 ואינדקס הסוף 3.
כלומר הוא יחזיר את תת-המטריצה המקסימלית ביותר מתוך החיבור של שורות 1 ו-2.
במקרה הזה: ($(2, 3)$, $(1, 0)$) (תת המטריצה האדומה במערך ה-2D).

מימוש פתרון הבעיה

```
public static int[] maxSumMatrix(int[][] mat) {

   int n = mat.length, m = mat[0].length;
   int maxSum = 0;
   int ii = 0, jj = 0, kk = 0, ll = 0;

   for(int i = 0 ; i < n ; i++) {
       for(int j = i ; j < n; j++) {
           int[] arr = new int[m];
           for(int k = i ; k <= j; k++) {
               for(int l = 0 ; l < m; l++) {
                   arr[l] += mat[k][l];
               }
           }
           int[] best = best(arr);
           if(best[0] > maxSum) {
               maxSum = best[0];
               ii = i;
```

```
                jj = best[1];
                kk = j;
                ll = best[2];
            }
        }
    }

    return new int[] {maxSum, ii, jj, kk, ll};
}
```

סיבוכיות זמן ריצה - $O(n^4)$

☚ $O(n^2)$ עבור כל תתי-השורות

☚ $O(n^2)$ לסכום כל תת-מטריצה שקיבלנו

☚ $O(n)$ הפונקציה של Best (מופעלת בתוך הלולאה השנייה של j).

✪ פתרון ג' - תכנות דינאמי

☚ הרעיון שעומד מאחורי הפתרון הוא שאם נתונים לנו סכומים חלקיים נוכל לחשב בעזרתם את הסכום הכולל.

☚ ניצור מטריצת עזר Sum כך שכל תא מייצג את הסכום מנקודת ההתחלה $[0][0]$ועד אליו $[i][j]$.

☚ עבור השורה הראשונה במטריצה Sum לכל תא נסכום את הערך שלו במטריצה המקורית + הערך באינדקס לפניו במטריצה Sum וכן גם עבור העמודה הראשונה.

☚ לכל תא אחר, נסכום בשיטה הבאה :

$$sumMat[i][j] = mat[i][j] + sumMat[i][j-1] + sumMat[i-1][j] - sumMat[i-1][j-1]$$

לדוגמה

1 עבור המטריצה המקורית mat

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

2 ניצור מטריצה חדשה $sumMat$ באותם המידות ונעבור על השורות ועמודות ראשונות

	0	1	2	3	4
0	2	3	0	-4	1
1	2	0	0	0	0
2	4	0	0	0	0
3	1	0	0	0	0

3 למשל עבור התא $[4][3]$נסכום:
$m[3][4] + s[3][3] + s[2][4] - s[2][3]$

	0	1	2	3	4
0	2	3	0	-4	1
1	2	9	9	9	15
2	4	9	8	12	13
3	1	9	9	13	17

נשים לב כי אנחנו סוכמים את התא $sumMat[i-1][j-1]$ פעמיים ובגלל זה אנחנו מחסירים אותו פעם אחת בנוסחה הרקורסיבית לבניית מטריצת העזר.

כשנעבור בלולאה על מידות (i,j) , (k,l)לאחר בניית המטריצה, כדי לגשת לתת-מטריצות שנקודת ההתחלה שלהם היא לא $[0][0]$כמו למשל ($(3,3)$, $(1,1)$) נפעל בנוסחה הבאה:

$$sum = sumMat[k][l] - sumMat[k][j-1] - sumMat[i-1][l] + sumMat[i-1][j-1]$$

לדוגמה

1 עבור ($(3,3)$, $(1,1)$)

	0	1	2	3	4
0	2	3	0	-4	1
1	2	9	9	9	15
2	4	9	8	12	13
3	1	9	9	13	17

2 נחסיר את השטחים הלא רלוונטים:
השורה הראשונה והעמודה הראשונה

	0	1	2	3	4
0	2	3	0	-4	1
1	2	9	9	9	15
2	4	9	8	12	13
3	1	9	9	13	17

3 מכיוון שהחסרנו את התא $[0][0]$פעמיים, נסכום אותו.

	0	1	2	3	4
0	2	3	0	-4	1
1	2	9	9	9	15
2	4	9	8	12	13
3	1	9	9	13	17

לערכים בתוך המטריצה בדוגמה הזאת אין כל חשיבות, אלא רק לחישוב הנוסחה עצמה:

$$sum = sumMat[3][3] - sumMat[3][1] - sumMat[0][3] + sumMat[0][0]$$

$$sum = 13 - (-4) - 1 + 2 = 18$$

מימוש פתרון הבעיה

```java
public static int[] maxSumMatrix(int[][] mat) {

   int n = mat.length, m = mat[0].length;
   int[][] sumMat = new int[n][m];
   sumMat[0][0] = mat[0][0];

   for(int i = 1 ; i < n ; i++)
       sumMat[i][0] = sumMat[i-1][0] + mat[i][0];


   for(int j = 1 ; j < m ; j++)
       sumMat[0][j] = sumMat[0][j-1] + mat[0][j];

   for(int i = 1 ; i < n ; i++) { // O(N*M)
       for(int j = 1 ; j < m; j++) {
           sumMat[i][j] = mat[i][j] + sumMat[i][j-1] + sumMat[i-1][j] - sumMat[i-1][j-1];
       }
   }

   int maxSum = 0, sum = 0;
   int ii = 0, jj = 0, kk = 0, ll = 0;

   for(int i = 0 ; i < n ; i++) {
       for(int j = 0 ; j < m ; j++) {
           for(int k = i ; k < n ; k++) {
               for(int l = j ; l < m ; l++) {

                   if(i==0 && j==0) sum = sumMat[k][l];
                   if(i==0 && j>0) sum = sumMat[k][l] - sumMat[k][j-1];
                   if(i>0 && j==0) sum = sumMat[k][l] - sumMat[i-1][l];
                   if(i>0 && j>0)
                    sum = sumMat[k][l] - sumMat[k][j-1] - sumMat[i-1][l] + sumMat[i-1][j-1];

                   if(sum > maxSum) {
                       maxSum = sum;
                       ii = i;
                       jj = j;
                       kk = k;
                       ll = l;
                   }
               }
           }
       }
   }

   return new int[] {maxSum, ii, jj, kk, ll};
}
```

סיבוכיות זמן ריצה - $O(n^4)$

✪ פתרון ד' - Super Best

☚ במקום לאפס את מערך העזר בכל פעם כמו בפתרון ב', נשתמש בקיים ורק נוסיף את השורה החדשה.

☚ ניצור מטריצת עזר חדשה שתסכום בעמודות.

☚ כלומר, עבור עמודה מספר 2 יהיה לנו את סכום כל מה שהיה באותו מקום במטריצה המקורית יחד עם הסכום שקיבלנו בעמודה לפניו.

☚ לאחר בניית מטריצת העזר, נעבור ב-3 לולאות וניצור מערך חדש ואליו נעתיק את את הקטע בין עמודה i ל-j.

☚ את המערך החדש נשלח ל- Best ונמצא את כל תתי המטריצות שקשורות לתאים אלו.

נוסחה לבניית מטריצת העזר:

$$preSum[i][j+1] = preSum[i][j] + mat[i][j]$$

דוגמה

1 עבור המטריצה המקורית mat

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

2 ניצור מטריצה חדשה $preSum$ עם מידת עמודה גדולה ב-1 מהמקורי ולמשל עבור עמודה ראשונה:

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

3 נסכום את שאר העמודות בהתאם לנוסחה:

	0	1	2	3	4	5
0	0	2	3	0	-4	1
1	0	0	6	9	13	14
2	0	2	0	-1	3	-2
3	0	-3	0	1	1	4

מימוש פתרון הבעיה

```java
public static int[] maxSumMatrix(int[][] mat) {
   int n = mat.length, m = mat[0].length;
   int maxSum = 0;
   int ii = 0, jj = 0, kk = 0, ll =0;
   int[][] preSum = new int[n][m+1];

   for (int i = 0; i < n; i++) { // O(N*M)
       for (int j = 0; j < m; j++) {
           preSum[i][j+1] = preSum[i][j] + mat[i][j];
       }
   }
   printmat(preSum);
   for(int i = 0; i < m; i++) { // O(N*M^2)
       for(int j = i; j < m; j++) {
           int[] arr = new int[n];

           for (int k = 0; k < n; k++) {
               arr[k] = preSum[k][j+1] - preSum[k][i];
           }
```

```
            int[] best = bestLinear(arr);
            if(best[0] > maxSum) {
                maxSum = best[0];
                ii = best[1];
                kk = best[2];
                jj = i;
                ll = j;
            }
        }
    }
    return new int[] {maxSum, ii, jj, kk, ll};
}
```

סיבוכיות זמן ריצה - $O(n^3)$

✪ פסאודו-קוד מימוש בעזרת Best אופטימלי ביותר

```
SuperBestPro(mat[N][M]):
sum⇐ 0
ii ⇐ 0
jj ⇐ 0
kk ⇐ 0
ll ⇐ 0

for i⇐ 0 to M do:
      create arr[N]
      for j⇐ i to M do:
            for k⇐ 0 to N do:
                  arr[k] ⇐ arr[k]+mat[k,j]
            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}
```

אלגוריתם Dijkstra

✪ תיאור האלגוריתם

☚ דייקסטרה הוא אלגוריתם חמדן.

☚ המטרה שלו היא לחשב את המסלול הקצר ביותר מקודקוד אחד לכל שאר הקודקודים.

☚ בניגוד ל-Floyd Warshall שמחשב לנו את המסלול הקצר ביותר מכל הקודקודים לכל הקודקודים.

✪ פעולת האלגוריתם

☚ נתון גרף משוקלל, לכל צלע יש משקל.

☚ נאתחל את ערך של קודקוד ההתחלה ל-0, את שאר הקודקודים נאתחל ל-∞.

☚ נתחיל לחשב את המסלול הזול ביותר באופן הבא:

1. נכניס את קודקוד ההתחלה והשכנים שלו לתור.
2. נעדכן את המרחק מקודקוד ההתחלה לכל שכן ונוציא את קודקוד ההתחלה מהתור.
3. נבחר את הקודקוד המינימלי בתור.
4. נעדכן את המרחק ממנו אל השכנים שלו, נכניס אותם לתור ונוציא אותו מהתור.
5. נמשיך כך עד שנוציא את כל הקודקודים מהתור.
6. כעת, הערך שיש בכל קודקוד הוא המסלול הקצר ביותר מקודקוד ההתחלה ועד אליו.

☚ נוכל לשמור עבור כל קודקוד מי האבא שלו:

1. ניצור מערך של אבות ובכל פעם שנאתחל קודקוד מסוים בערך חדש.
2. נרשום במערך מאיפה הגענו אליו.

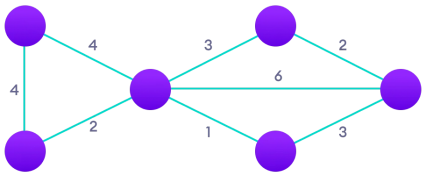
✪ פסאודו קוד

```
function Dijkstra(Graph, source):
   create vertex set Q

   for each vertex v in Graph: // Initialization
       dist[v] ← INFINITY     // Unknown distance from source to v
       prev[v] ← UNDEFINED    // Previous node in optimal path from source
       add v to Q            // All nodes initially in Q (unvisited nodes)

   dist[source] ← 0         // Distance from source to source

   while Q is not empty:
       u ← vertex in Q with min dist[u] // Node with the least distance will be selected first
       remove u from Q

       for each neighbor v of u:           // where v is still in Q.
           alt ← dist[u] + length(u, v)
           if alt < dist[v]:   // A shorter path to v has been found
               dist[v] ← alt
               prev[v] ← u

   return dist[], prev[]
```

✪ דוגמה

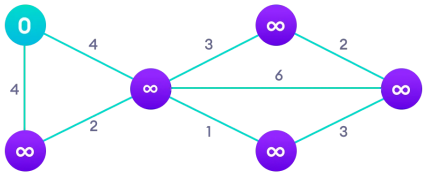
שלב 1

- נתון גרף משוקלל
- לכל צלע יש משקל



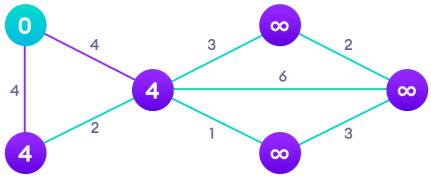
שלב 2

- נאתחל את ערך של קודקוד ההתחלה ל-0
- את שאר הקודקודים נאתחל ל-∞



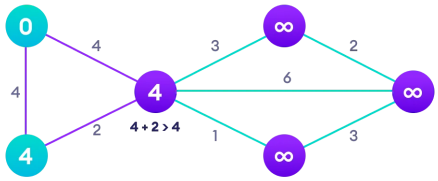
שלב 3

- נכניס את קודקוד ההתחלה והשכנים שלו לתור
- נעדכן את המרחק מקודקוד ההתחלה לכל שכן
- נוציא את קודקוד ההתחלה מהתור



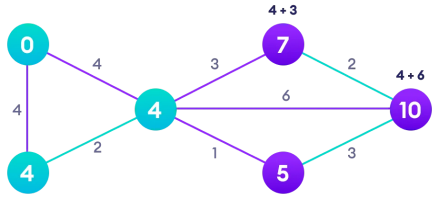
שלב 4

- אם אורך המסלול של קודקוד הסמוך קטן מאורך המסלול החדש, אל תעדכן אותו



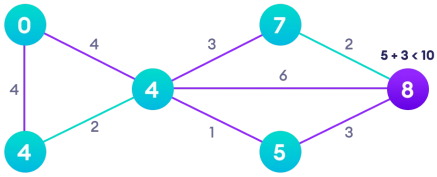
שלב 5

- להימנע מעדכון אורכי מסלול של קודקודים שכבר ביקרנו



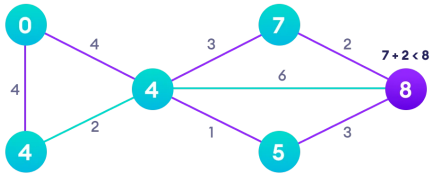
שלב 6

- לאחר כל איטרציה, אנו בוחרים את קודקוד הלא נראה בעל אורך המסלול הנמוך ביותר.
- אז אנחנו בוחרים 5 לפני 7.



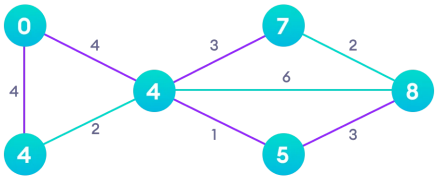
שלב 7

- נשים לב כיצד אורך המסלול של הקודקוד הימני ביותר מתעדכן פעמיים



שלב 8

- נחזור על הפעולה עד לביקור בכל הקודקודים



✪ טענה

עבור גרף $G = \{V, E\}$ מכוון ומשוקלל כאשר משקלי הצלעות הם מספרים לא שליליים ועבור צומת המקור s.
נגדיר את $d(s, v)$ להיות המרחק הקצר ביותר בין המקור s לבין כל $v \in V(G)$.
ולאחר הרצת אלגוריתם דייקסטרה מתקיים:

$$distance[v] = d(s, v)$$

✪ הוכחה

נוכיח באינדוקציה.
נראה שלכל קודקוד $v \in V(G)$ כאשר נסמן אותו כמטופל ($visited$) מתקיים השוויון: $distance[v] = d(s, v)$.

☚ בסיס האינדוקציה:

בשלב הראשון $distance[s] = d(s, s) = 0$.

☚ צעד האינדוקציה:

נניח עבור n שלבי טיפול ונוכיח עבור $n + 1$ שלבי טיפול.
יהי u הקודקוד שנבחר בשלב ה $n + 1$, צריך להוכיח ש: $distance[u] = d(s, u)$.
יהי P המסלול הקצר ביותר מ-s ל-u.
נשים לב כי לכל $x \in V(G)$ מתקיים $distance[x] \geq d(s, x)$.
נסמן ב-v את הקודקוד הראשון הלא מטופל במסלול P ויהי z הקודקוד הקודם ל-v ב-P.
לכן הקודקוד z כבר טופל ולפי הנחת האינדוקציה מתקיים $distance[z] = d(s, z)$.
כיוון ש-P הוא המסלול הקצר ביותר מ-s ל-u אז התת-מסלול שלו מ-s ל-v הוא גם הקצר ביותר ולכן:

$$d(s, v) = d(s, z) + weight(v, z) = distance[z] + weight(v, z)$$

כלומר,

$$d(s, v) = distance[z] + weight(v, z)$$

לפי פעולת האלגוריתם נבחר את המינימלית ולכן במסלול שהתקבל לפי דייקסטרה, קודקוד v קיבל המרחק המינימלי, ולכן:

$$distance[v] \leq distance[z] + weight(z, v) = d(s, v)$$

וקיבלנו ש-$d(s, v)$ הוא המרחק הקצר ביותר מ-s ל-v כלומר $distance[v] = d(s, v)$.
אבל קודקוד v לא מטופל ואנו בחרנו את קודקוד u ולכן:

$$distance[u] \leq distance[v] = d(s, v) \leq d(s, u) \leq distance[u]$$

ולפי כלל הסנדוויץ' נקבל: $distance[u] = d(s, u)$

האלגוריתם מסתיים כאשר כל הקודקודים מטופלים, לכן לכל קודקוד מתקיים:

$$distance[u] = d(s, u)$$

כנדרש.

∎

מימוש בגיטהאב

מימוש האלגוריתם

```
public class DijkstraMatrix {
   public DijkstraMatrix() {}
   static int inf = 1000000;

   public void Dijkstra(int[][] matrix, int src, int dest) {
       int number_of_nodes = matrix.length;
       int[] distances = new int[number_of_nodes];  מערך מרחקים מהמקור לכל צומת שנעבור
       HashSet<Integer> visited = new HashSet<>();  כדי לא לעבור על אותו צומת פעמיים או יותר
       int[] previous = new int[number_of_nodes];  כדי שנדע לחזור אחורה בשביל להדפיס מסלול

       // init the data structures.
       for(int i = 0; i < number_of_nodes; i++) {
           distances[i] = inf; previous[i] = -1;
       }

       distances[src] = 0;   המרחק מהמקור לעצמו הוא פשוט 0
       int current_node = getPriority(distances, visited);   שולפים צומת עדיפה עבורנו
       while(current_node != -1) {   כל עוד קיים מסלול תקין בין המקור לבין היעד

           for(int i=0 ; i<number_of_nodes; i++) {   נעבור על כל השכנים שלו
               int new_distance = distances[current_node] +
               matrix[current_node][i];
                 אם קיים מסלול קצר יותר בין השכן לצומת הנוכחית אז נעדכן את המרחק מביניהם
               if(!visited.contains(i) && distances[i] > new_distance) {
                   distances[i] = new_distance;
                   previous[i] = current_node;
               }
           }
           visited.add(current_node);   נסמן שסיימנו את העבודה עבור הצומת הנוכחית-למנוע חזרות

           if(current_node == dest) break; // if we reached to the destination
           // else, keep iterating using the following priority method:
           current_node = getPriority(distances, visited);
       }

       System.out.println("Distance from ("+src+")->("+dest+") is: "
+distances[dest]);

       current_node = dest;
       String path = "";
       while(previous[current_node] != -1) { // (ONLY THE SOURCE IS -1 IN THIS
CASE)
           path = " -> " + current_node + path;
```

```
            current_node = previous[current_node]; // Go back.
        }
        path = current_node + path;
        System.out.println("Path is: "+ path + "\n");
    }

    private static int getPriority(int[] distances, HashSet<Integer> visited) {
        int minIndex = inf, minValue =inf;
        for(int i = 0; i < distances.length; i++) {
            if(!visited.contains(i) && distances[i] < minValue) {
                minIndex = i;
                minValue = distances[i];
            }
        }
        return minIndex;
    }

}
```

סיבוכיות זמן ריצה - $O(|E| + |V| \cdot log|V|)$

אלגוריתם Bidirectional Dijkstra

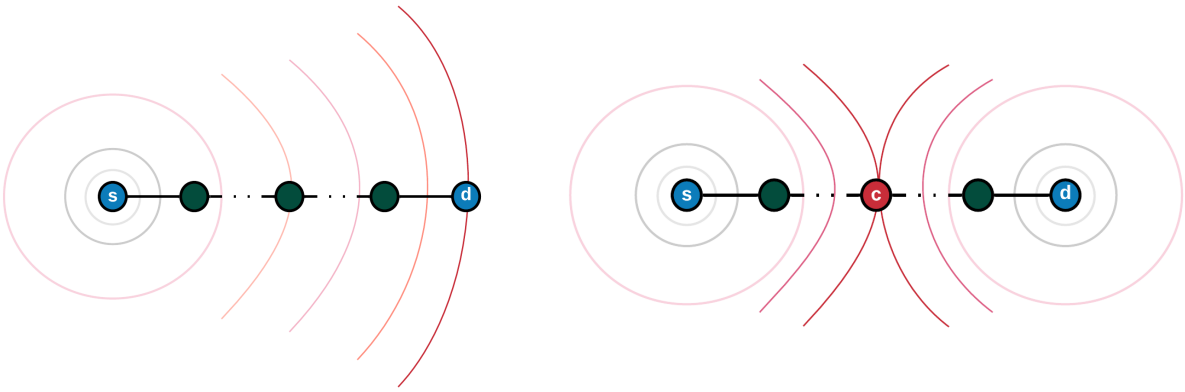
✪ תיאור האלגוריתם

☚ דייקסטרה דו כיוונית היא דייקסטרה שמתקדמים בה גם מקודקוד ההתחלה וגם מקודקוד הסיום בו זמנית.

☚ היתרון באלגוריתם דייקסטרה דו-כיווני לבין דייקסטרה רגיל הוא ששטח הסריקה קטן בכ-חצי.

☚ למשל באיור הבא, האיור משמאל הוא הפעלת דייקסטרה רגיל עבור נרצה את $s \rightsquigarrow d$.

באיור מימין אפשר לראות את ההפעלה של דייקסטרה דו-כיוונית - בהגעת צומת המפגש c הצלחנו לחשב את אורך המסלול $s \rightsquigarrow d$ב-"בערך" חצי שטח סריקה.



✪ הגדרה - גרף רוורס

☚ גרף רוורס $G^R = \{V, E^R\}$ הוא גרף הדומה ל-$G = \{V, E\}$פרט לכך שכיווני הצלעות המכוונות הם הפוכים.

☚ קבוצת הקודקודים שלו זהה ל-Gואילו קבוצת הצלעות מוגדרת: $E^R = \{(u, v) \mid (v, u) \in E\}$.

✪ שלבי פעולה

☚ נבנה את G^R.

☚ נריץ את האלגוריתם של דייקסטרה מ-sב-Gומ-dב-G^R.

☚ כל אלגוריתם בתורו (ישיר או הפוך) יפעל לפי דייקסטרה באופן רגיל.

☚ נבצע החלפות בין האלגוריתמים בכל איטרציה עד שנפגוש צומת c שטופלה בשניהם.

☚ נעצור בשלב שקיימת צומת cשטיפלנו בה גם בהפוך וגם בישיר (כלומר שבשניהם c נמצאת ב-$visited[]$) .

☚ עבור הצומת cהזאת יכולים ליצור "תקשורת" בין הצד של sלבין הצד של d.

☚ בעזרת ה-"תקשורת" אנחנו נחשב את המסלול הקצר ביותר בין sל-dבכך שנחשב את מינימום בסכום בין האלגוריתם הישיר לבין ההפוך: $d(s, d) = distances[c] + distances^R[c]$.

☚ נחזיר המסלול הקצר ביותר $d(s, d)$.

✪ סיבוכיות

הסיבוכיות של דייקסטרה דו כיוונית היא בדיוק כמו הסיבוכיות של דייקסטרה רגילה - $O(|E| + |V| \cdot log|V|)$

☚ זמן ריצה: יכול לפעול פי 2 יותר מהר מדייקסטרה רגילה במקרה הטוב.

☚ זיכרון: תופס פי 2 יותר זיכרון: אנחנו עובדים עם 2 גרפים Gו-G^R.

✪ למה

נסמן ב- $d[u]$ את המרחק באלגוריתם של דייקסטרה החל מקודקוד s בגרף G.

נסמן ב-$d^R[u]$ את המרחק באלגוריתם של דייקסטרה החל מקודקוד t בגרף G^R.

לאחר שצומת מסוימת v טופלה גם ב-G וגם ב-G^R, אז עדיין קיים מסלול קצר ביותר $d(s,t)$ שעובר דרך קודקוד כלשהו u שטופל ב-G או ב-G^R או בשניהם ומתקיים:

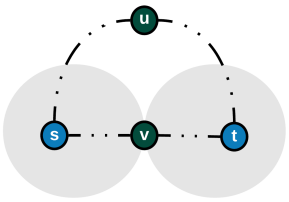
$$d(s,t) = d[u] + d^R[u]$$

נראה כי בהכרח קיים מסלול קצר ביותר $d(s,t)$ שכל קודקודיו טופלו ע"י G או G^R.

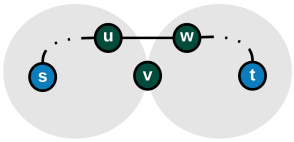
✪ הוכחה

נחלק למקרים:

1. אם קיימת צומת u שלא טופלה ע"י אף אחד מהאלגוריתמים, אז קיימת לפחות צומת אחת נוספת v שטופלה על-ידי האלגוריתמים והמסלול הקצר ביותר עובר דרכו.
   המרחק שיש ב-v בהכרח נכון, כי לפי הגדרה דייקסטרה, המרחק מ-s ל-v הוא המרחק הקצר ביותר וגם בין v ל-t.



2. אם כל הצמתים טופלו בשני האלגוריתמים חוץ מצומת המפגש v.
   נניח שיש מסלול קצר $d(s,t)$ ונניח שהצומת האחרונה שטופלה באלגוריתם הישיר היא u.
   הצומת הבאה במסלול הקצר ביותר היא w אשר חייבת להיות בתוך המעגל של t והיא בהכרח טופלה ע"י האלגוריתם ההפוך, כיוון שכל הצמתים טופלו מלבד v.
   לכן: $d(s,t) = d[u] + length(u,w) + d^R[w]$.



יהי צומת u המקיימת $d(s,t) = d[u] + d^R[u]$ כלומר שהיא במסלול הקצר ביותר.

ממקרה 2 קיבלנו $d(s,t) = d[u] + length(u,w) + d^R[w]$.

כלומר נרצה להוכיח כי $d^R[u] = length(u,w) + d^R[w]$.

נשים לב ש- $d^R[u] \geq d(u,t)$ כי בכל רגע של הפעלת האלגוריתם, המרחקים משתפרים יותר ויותר.

מצד שני, ברגע ש- w מטופל על-ידי האלגוריתם ההפוך, אז השכן שלו u יכול להתעדכן במרחק קצר יותר ולכן: $Length(u,w) + d^R[w] \geq d^R[u]$.

כלומר, לפי כלל הסנדוויץ':

$$d(u,t) \leq d^R[u] \leq Length(u,w) + d^R[w]$$

ולכן

$$d(s,t) = d[u] + length(u,w) + d^R[w] = d[u] + d^R[u]$$

כלומר,

$$d(s,t) = d[u] + d^R[u]$$

I

מימוש בגיטהאב

מימוש האלגוריתם

```
public class BiDijkstra {
   static int inf = 1000000;

   public BiDijkstra() {}

   public void Dijkstra(ArrayList<ArrayList<Integer>> graph,
          ArrayList<ArrayList<Integer>> rgraph, int src, int dest, int[][] mat ) {

       int number_of_nodes = graph.size();
       int[] src_distance = new int[number_of_nodes];
       int[] dest_distance = new int[number_of_nodes];
       int[] src_previous = new int[number_of_nodes];
       int[] dest_previous = new int[number_of_nodes];
       HashSet<Integer> src_visitors = new HashSet<>(number_of_nodes);
       HashSet<Integer> dest_visitors = new HashSet<>(number_of_nodes);

       for(int i = 0; i < number_of_nodes; i++) {
           src_distance[i] = inf;
           dest_distance[i] = inf;
           src_previous[i] = -1;
           dest_previous[i] = -1;
       }

       src_distance[src] = 0;
       dest_distance[dest] = 0;

       int src_current = getPriority(src_distance, src_visitors);
       int dest_current = getPriority(dest_distance, dest_visitors);

       while(src_current != -1 && dest_current != -1) {

           //// FOR G ///////////////
           for(Integer neighbour : graph.get(src_current)) {
               int new_distance = src_distance[src_current] +
                   mat[src_current][neighbour];
               if(!src_visitors.contains(neighbour) && src_distance[neighbour] >
                  new_distance) {
                   src_distance[neighbour] = new_distance;
                   src_previous[neighbour] = src_current;
```

```
            }
        }
        src_visitors.add(src_current);
        if(src_visitors.contains(src_current) &&
           dest_visitors.contains(src_current)) break;
        src_current = getPriority(src_distance, src_visitors);

        //// FOR G reverse////////////////
        for(Integer neighbour : rgraph.get(dest_current)) {
            int new_distance = dest_distance[dest_current] +
                mat[dest_current][neighbour];
            if(!dest_visitors.contains(neighbour) && dest_distance[neighbour] >
               new_distance) {
                dest_distance[neighbour] = new_distance;
                dest_previous[neighbour] = dest_current;
            }
        }
        dest_visitors.add(dest_current);
        if(src_visitors.contains(dest_current) &&
           dest_visitors.contains(dest_current)) break;
        dest_current = getPriority(dest_distance, dest_visitors);

    } // end while

    int minDistance = inf, minIndex = -1;
    for(int i = 0; i < number_of_nodes; i++) {
        if( (src_visitors.contains(i) || dest_visitors.contains(i))
        &&  (src_distance[i] != inf && dest_distance[i] != inf)
        &&  (minDistance > src_distance[i] + dest_distance[i])) {
            minDistance = src_distance[i] + dest_distance[i];
            minIndex = i;
        }
    }

    System.out.println("Distance from ("+src+")->("+dest+") is: " +minDistance);

    // Generate path from source to connector node 'v'
    int current = minIndex;
    String path = "";
    while(src_previous[current] != -1) {
        path = " -> " + current + path;
        current = src_previous[current];
    }
    path = current + path;

    // Generate path from dest to connector node 'v'
    current = dest_previous[minIndex];
```

```
        String rpath = "";
        while(dest_previous[current] != -1) {
            rpath += " -> " + current;
            current = dest_previous[current];
        }
        rpath += " -> " + current;

        String total_path = path + rpath;
        System.out.println("Total path is: " + total_path + "\n");

    }

    public int getPriority(int[] distances, HashSet<Integer> visitors) {
        int minValue = inf, minIndex = inf;

        for(int i = 0; i < distances.length; i++) {
            if(!visitors.contains(i) && minValue > distances[i]) {
                minIndex = i;
                minValue = distances[i];
            }
        }

        return minIndex;
    }
}
```

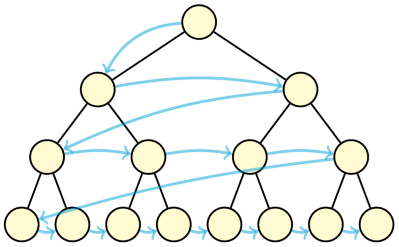
אלגוריתם Breadth First Search - BFS

✪ הגדרה - חיפוש לרוחב

חיפוש לרוחב סורק את הגרף בצורה שמבטיחה שכל צומת שנמצא באותו רכיב קשירות של הצומת ההתחלתי ייבדק, וסריקה זו נעשית בזמן אופטימלי, הליניארי למספר הקשתות והצמתים בגרף.

✪ תיאור האלגוריתם

- בעברית: אלגוריתם חיפוש לרוחב.
- האלגוריתם משתמש בתור על מנת לקבוע מהו הצומת הבא בו הוא עומד לבקר.
- בכל פעם שהוא מבקר בצומת הוא מסמן אותו ככזה שנבדק, ואז בודק את כל הקשתות שיוצאות ממנו.
- אם קשת מובילה לצומת שטרם נבדק, צומת זה מתווסף לתור.
- בדרך זו מובטח כי האלגוריתם יסרוק את הצמתים בסדר שנקבע על פי מרחקם מהצומת ההתחלתי (כי צומת שנכנס לתור יצא ממנו רק לאחר שכל הצמתים שהיו בו קודם יצאו).
- האלגוריתם פועל על גרפים מכוונים וגם על גרפים לא מכוונים.



✪ שיטת האלגוריתם

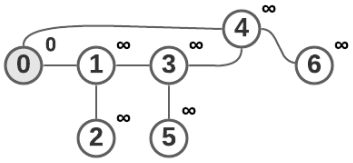
- האלגוריתם מחלק את הקודקודים ל-3 צבעים:
  1. צבע לבן - הקודקוד עוד לא התגלה.
  2. צבע אפור - הקודקוד התגלה אך לא טופל.
  3. צבע שחור - הקודקוד טופל.
- האלגוריתם בונה עץ רוחב ששורשו הוא קודקוד המקור sולפי השכבות והמרחקים ממנו הוא בונה את שאר הגרף. לכן, קודקוד במרחק hיתגלה לפני קודקוד במרחק $h + 1$.
- כל קודקוד שמגלה את שכניו הופך להיות האבא שלהם. בגלל שכל קודקוד מתגלה פעם אחת אז יש לו רק אבא אחד.

✪ פסאודו קוד

```
BFS(G,src):
for-each v in V(G)\{src}:
    color[v] = WHITE
    d[v] = ∞
    f[v] = NULL
color[src] = GRAY
d[src] = 0
f[src] = NULL
Queue q
Enqueue(q,src)
while q is not empty:
    current = dequeue(q)
    for-each neighbour in N(current):
        if color[neighbour] = WHITE:
            color[neighbour] = GRAY
            d[neighbour] = d[current] + 1
            f[neighbour] = current
            Enqueue(q,neighbour)
    color[current] = BLACK
```

✪ דוגמה

שלב 1



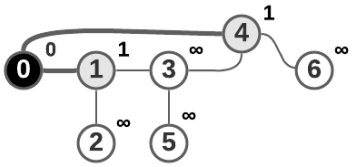
$color[] = \{G, W, W, W, W, W, W\}$

$distances[] = \{0, \infty, \infty, \infty, \infty, \infty, \infty\}$

$parent[] = \{-1, -1, -1, -1, -1, -1, -1\}$

$queue = (0)$

שלב 2



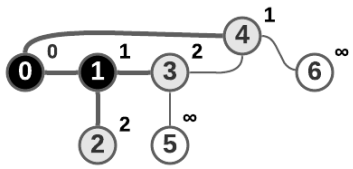
$color[] = \{B, G, W, W, G, W, W\}$

$distances[] = \{0, 1, \infty, \infty, 1, \infty, \infty\}$

$parent[] = \{-1, 0, -1, -1, 0, -1, -1\}$

$queue = (1, 4)$

שלב 3



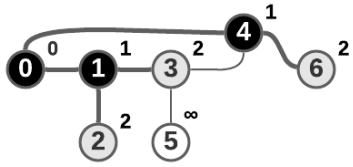
$color[] = \{B, B, G, G, G, W, W\}$

$distances[] = \{0, 1, 2, 2, 1, \infty, \infty\}$

$parent[] = \{-1, 0, 1, 1, 0, -1, -1\}$

$queue = (4, 2, 3)$

שלב 4



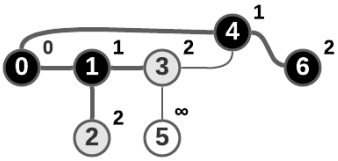
$color[] = \{B, B, G, G, B, W, G\}$

$distances[] = \{0, 1, 2, 2, 1, \infty, 2\}$

$parent[] = \{-1, 0, 1, 1, 0, -1, 4\}$

$queue = (4, 2, 3)$

שלב 5



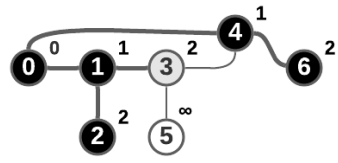
$color[] = \{B, B, G, G, B, W, B\}$

$distances[] = \{0, 1, 2, 2, 1, \infty, 2\}$

$parent[] = \{-1, 0, 1, 1, 0, -1, 4\}$

$queue = (2, 3)$

שלב 6



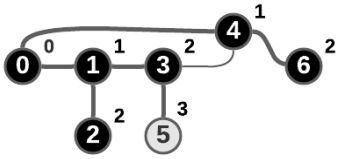
$color[] = \{B, B, B, G, B, W, B\}$

$distances[] = \{0, 1, 2, 2, 1, \infty, 2\}$

$parent[] = \{-1, 0, 1, 1, 0, -1, 4\}$

$queue = (3)$

שלב 7



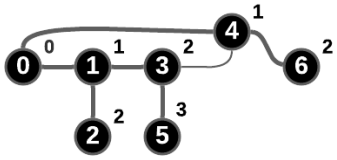
$color[] = \{B, B, B, B, B, G, B\}$

$distances[] = \{0, 1, 2, 2, 1, 3, 2\}$

$parent[] = \{-1, 0, 1, 1, 0, 3, 4\}$

$queue = (5)$

שלב 8



$color[] = \{B, B, B, B, B, B, B\}$

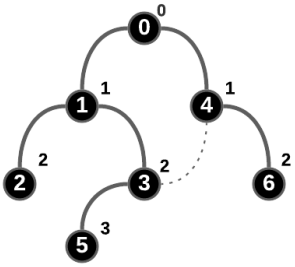
$distances[] = \{0, 1, 2, 2, 1, 3, 2\}$

$parent[] = \{-1, 0, 1, 1, 0, 3, 4\}$

$queue = ()$

✪ למה

סריקת הגרף עם BFS בונה עץ כתת-גרף של G בעל הקודקודים שניתן להגיע אליהם מ-s.



למשל, עבור הדוגמה מלמעלה נקבל את העץ הבא:
העץ ללא מעגלים (נשים לב לצלע המקווקות, הצלע לא באמת קיימת למרות שבגרף המקורי היא כן קיימת), זה מאחר כי כל צלע מתגלה פעם אחת בלבד וכי לפי BFS ניתן לצייר את העץ בעזרת המערך של
$parent[] = \{-1, 0, 1, 1, 0, 3, 4\}$שקיבלנו מהאלגוריתם.
כלומר הצמתים 3 ו-4 נפגשו שכבר אחד מהם מסומן ולכן לא נוצר ביניהם צלע.

✪ הוכחה

בשלב הראשון העץ מכיל רק קודקוד אחד – המקור s .
כאשר אנו מוסיפים קודקוד חדש (לפי האלגוריתם אנו מוסיפים רק קודקודים לבנים שעדיין לא התגלו) אנו מוסיפים גם את הצלע שבעזרתה הגענו לקודקוד זה.
מכיוון שאנו מגלים כל קודקוד לכל היותר פעם אחת יש לו לכל היותר קודקוד אב אחד.
לכן המבנה שנוצר ע"י אלגוריתם BFS הוא עץ.

מימוש בגיטהאב

מימוש האלגוריתם

```
public class BFSAlgorithm {
   final static int WHITE = -1, GRAY = 0, BLACK = 1;
   final static int inf = 1000000;

   public void BFS(int[][] matrix, int src, int dest) {
       int number_of_nodes = matrix.length;
       int[] color = new int[number_of_nodes];
       int[] distances = new int[number_of_nodes];
       int[] parent = new int[number_of_nodes];
       Queue<Integer> queue = new LinkedList<>();

       for(int i = 0 ; i < number_of_nodes; i++) {
           color[i] = WHITE; distances[i] = inf; parent[i] = -1;
       }
       queue.add(src);
       color[src] = GRAY;
       distances[src] = 0;

       while(!queue.isEmpty()) {
           int current = queue.poll();
           for(int i = 0; i < number_of_nodes; i++) {
               if(color[i] == WHITE && matrix[current][i] != inf) {
                   color[i] = GRAY;
                   distances[i] = distances[current] + 1;
                   parent[i] = current;
                   queue.add(i);
               }
           }
           color[current] = BLACK;
       }
       String path = "";
       path = path + dest;
       int current = parent[dest];
       while(current != -1) {
           path = current + "->" + path;
           current = parent[current];
       }
       System.out.println("Distance of ("+src+")->("+dest+") : " +
distances[dest]);
       System.out.println(path + "\n");
   }

}
```

✪ סיכום סיבוכיות

☚ אתחול מבני הנתונים $O(|V|)$

☚ כל צומת נכנסת ויוצאת פעם אחת מהתור ב- $O(1)$.

☚ ב-while אנחנו עוברים על כל הקודקודים הסמוכים לקודקוד כלשהו, לכן עוברים פעמיים על כל קודקוד ופעמיים על כל צלע לכן $O(2|E| + |V|)$

☚ לכן בסה"כ $O(|V| + |E|)$.

✪ בדיקת קשירות

☚ לאחר הפעלת BFS נשלח את המערך $color[]$ אל הפונקציה הבאה כדי לוודא שהצלחנו להגיע לכל הצמתים בגרף, אם כל הצמתים מסומנים כ-$BLACK$ אז הגרף קשיר.

☚ פתרון יעיל יותר בקוד `BFSComponents.java` בגיטהאב.

```
public boolean checkConnectivity(int[] color) { //O(N)
   for(int i : color)
       if(i != BLACK) return false;
   return true;
}
```

✪ מספר רכיבי קשירות

☚ נפעיל את BFS על כל רכיבי הקשירות, על כל הפעלת BFS נספור count עד שנסיים.

```
public int numberOfComponents() {
   int nextComponent = getNextComponent(); // get the first component
   int num_of_components = 0;
   while(nextComponent != -1) { // while there is still more components to visit
       num_of_components++; // Count the number of components
       BFS(nextComponent); // use BFS with the given node-id which is inside the unvisited component
       nextComponent = getNextComponent(); // get the next unvisited component
   }
   return num_of_components;
}

private int getNextComponent() {
   for(int i = 0 ; i < distances.length; i++) {
       if(distances[i] == inf) {
           return i;
       }
   }
   return -1; // if all the components are visited
}
```

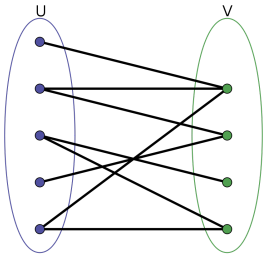
✪ קודקודים בכל רכיב קשירות

☚ נשלח אחרי הפעלת ה-BFS ונקבל מחרוזת של המסלול במידה ויש.

```
public String getPath(int src, int dest, int[] parent) {
   if(src == dest)
       return ""+src;
   else if(parent[dest] == -1)
       return "no path!";
   else {
       String path = "";
       path = path + dest;
       int current = parent[dest];
       while(current != -1) {
           path = current + "->" + path;
           current = parent[current];
       }
       return path;
   }
}
```

גרף דו-צדדי - Bipartite

(נלמד מסיכום שמופיע במודל ונשאל במבחנים קודמים אבל לא נלמד בהרצאה).



✪ הגדרה

בתורת הגרפים, גרף דו-צדדי הוא גרף שבו ניתן לחלק את הקודקודים לשתי קבוצות זרות,
כך שלא קיימת צלע בין שני קודקודים השייכים לאותה הקבוצה.

גרפים דו-צדדיים מועילים במידול בעיות התאמה.
למשל, אם יש לנו קבוצה P של אנשים וקבוצה J של עבודות ואנו רוצים לבצע חלוקת עבודה,
נוכל בתור מודל לתאר את האנשים והעבודות כגרף דו-צדדי שקבוצת קודקודים אחת בו היא P והשניה J,
ויש צלע בין אדם המתאים לעבודה מסוימת לבין העבודה הזו.

✪ משפט

גרף קשיר הוא דו-צדדי אם ורק אם כל המעגלים שלו הם באורך זוגי.

✪ הוכחה - מהמודל

יהי $G = (V_1, V_2, E)$ גרף דו-צדדי. נתבונן במעגל כלשהו בגרף.

נצא מקודקוד $v_1 \in V_1$ הנמצא במעגל הזה ונטייל לאורך המעגל.

כל צלע שעליה אנחנו מטיילים מעבירה אותנו לצידו השני של המעגל.

לכן, דרוש מספר זוגי של צלעות כדי לחזור לקודקוד v_1 שבו התחלנו ולסגור את המעגל.

ולהיפך, נניח שכל המעגלים בגרף $G = (V, E)$ הם באורך זוגי ונראה ש-G הוא דו-צדדי.

יהי $s \in V$ קודקוד כלשהו בגרף G.

נגדיר את קבוצת הקודקודים V_1, V_2 באופן הבא:

$$V_1 = \{u \in V: distance(u,s)\ is\ even\}\ ,\ V_2 = \{u \in V:\ distance(u,s)\ is\ odd\}$$

כמובן שקבוצות אלו הם זרות וגם $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$.

נניח בשלילה שהגרף הוא לא דו-צדדי ויש צלע $\{u, v\}$ בין קודקודים השייכים לאותה קבוצה, כלומר $u, v \in V_1$.

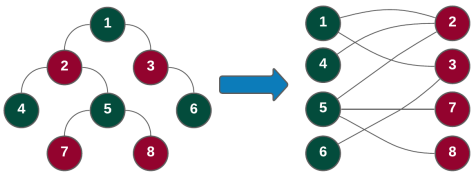
לכן המסלול הקצר ביותר מ-s ל-u, המסלול הקצר ביותר מ-s ל-v והצלע $\{u, v\}$ יוצרים מעגל שאורכו אי-זוגי. -סתירה.

באופן דומה עבור הוכחה שאין צלעות בין קודקודים של V_1.

I

✪ משפט

כל עץ, $|E| > 0$, הוא גרף דו-צדדי.



✪ הוכחה - מהספר מבחנים

הגדרת עץ: גרף קשיר ללא מעגלים.

הגדרת גרף דו-צדדי: גרף שאין בו מעגלים באורך אי-זוגי.

מסקנה - עץ הוא גרף דו-צדדי.

I

✪ פתרון הבעיה

כדי לבדוק אם גרף הוא דו-צדדי צריך להפעיל עליו BFS:

☚ ניצור מערך בשם partition בגודל $|V|$מאותחל לאפס, ובו נסמן:

0 - קודקוד לא התגלה, **1** - קודקוד שייך לקבוצה V_1, **2** - קודקוד שייך לקבוצה V_2.

☚ כאשר נמצא צלע ששני הקודקודים שלו שייכים לאותה הקבוצה אז הגרף לא דו-צדדי.

מימוש בגיטהאב

מימוש האלגוריתם

```
final static int WHITE = -1, GRAY = 0, BLACK = 1, inf = 1000000;

public boolean Bipartite(int[][] matrix, int source) {
   boolean bi_check = true;
   int number_of_nodes = matrix.length;
   int[] color = new int[number_of_nodes];
   int[] partition = new int[number_of_nodes];
   Queue<Integer> queue = new LinkedList<>();

   for(int i = 0; i < number_of_nodes; i++) {
       color[i] = WHITE;
   }
   color[source] = GRAY;
   partition[source] = 1;
   queue.add(source);

   while(!queue.isEmpty() && bi_check) {
       int current = queue.poll();
       for(int i = 0; i < matrix.length; i++) {

           if(bi_check && i!=current && matrix[current][i] != inf) {
               if(partition[current] == partition[i]) {
                   bi_check = false;
               }
               else if(color[i] == WHITE) {
                   color[i] = GRAY;
                   partition[i] = 3 - partition[current]; // 1 or 2
                   queue.add(i);
               }
           }
       }
       color[current] = BLACK;
   }
   return bi_check;
```

```
}
```

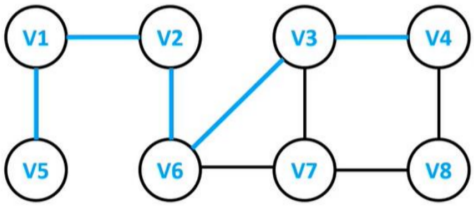
סיבוכיות זמן ריצה - $O(|V| + |E|)$.

קוטר של גרף

✪ הגדרה

☚ קוטר בגרף הינו המרחק המקסימלי שקיים בין שני קודקודים בגרף.

✪ דוגמה



☚ בציור משמאל, הקוטר המקסימלי הוא 5 (מסלול מסומן בכחול).

✪ תיאור האלגוריתם

☚ נקבל גרף כרשימה המתארת את הקשרים (שכנויות בין קודקודים).

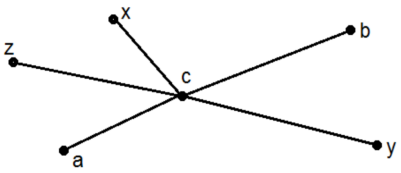
☚ נפעיל את $BFS(graph, v)$ עבור קודקוד רנדומלי $v \in V$ כלשהו, האלגוריתם יחזיר את המערך $distances$[].

☚ נעבור בלולאה על $distances$[]כדי לתפוס את הקודקוד u בו מסומנת המרחק הכי מקסימלי מהקודקוד vאליו.

☚ נשלח שוב לאלגוריתם $BFS(graph, u)$אך הפעם עם u, האלגוריתם יחזיר מערך $distances$[].

☚ נעבור בלולאה על $distances$[]שוב רק שהפעם נתפוס את הערך המקסימלי ביותר במערך והוא יהיה הקוטר.

✪ משפט



נגדיר את קבוצת הקודקודים של Tלהיות $\{a, b, c, x, y, z\}$.

בוחרים בקודקוד $x \in T$, מוצאים קודקוד y שהוא רחוק ביותר מ-x.

מוצאים קודקוד z שהוא הרחוק ביותר מ-y, אזי $dist(y, z) = diameter$.

✪ הוכחה

נניח בשלילה שקיים מסלול אחר בין aל-bכך $distance(a, b) > distance(x, y)$.

שני המסלולים עוברים דרך מרכז העץ שהוא קודקוד c.

נתון כי הקודקוד yהוא הרחוק ביותר מ-x, לכן $distance(b, c) \leq distance(y, c)$.

באופן דומה גם $distance(a, c) \leq distance(z, c)$ולכן:

$$distance(b, c) + distance(c, a) \leq distance(y, c) + distance(c, z)$$

או

$$distance(a, b) \leq distance(y, z)$$

אז:

$$diameter(T) = distance(y, z)$$

∎

מימוש בגיטהאב

מימוש האלגוריתם

```
public int getDiameter(int[][] matrix) {
   int src = 0;
   int[] distances = BFS(matrix,src);
   int maxValue = 0, maxIndex = 0;

   for(int i = 0; i < distances.length; i++) {
       if(distances[i] > maxValue) {
           maxValue = distances[i];
           maxIndex = i;
       }
   }

   distances = BFS(matrix,maxIndex);
   int diameter = 0;
   for(int i = 0; i< distances.length; i++) {
       if(distances[i] > diameter) diameter = distances[i];
   }

   return diameter;
}
```

✪ סיבוכיות

☚ מפעילים פעמיים את BFS, כלומר $O(2(|V| + |E|))$, ומחפשים פעמיים על המערך $distances[]$: $O(2 \cdot |V|)$

☚ סה"כ נקבל: $O(|V| + |E|)$.

אלגוריתם Fire

✪ מושגים

1. מרכז - אמצע הקוטר
2. רדיוס - ½ מהקוטר
3. קוטר - המרחק המקסימלי בין 2 צמתים

← איך מוצאים את הקוטר? - מפעילים פעמיים BFS(הסבר בנושא הקודם בסיכום).

✪ למה

אם הקוטר זוגי אז הרדיוס הוא ½ מהקוטר ויש מרכז אחד.

אם הקוטר אי-זוגי אז הרדיוס הוא ½ מהקוטר בעיגול כלפי מעלה ויש 2 מרכזים.

עד עכשיו נפגשנו בפתרון למציאת קוטר של גרף בסיבוכיות זמן ריצה של $O(|E| + |V|)$.

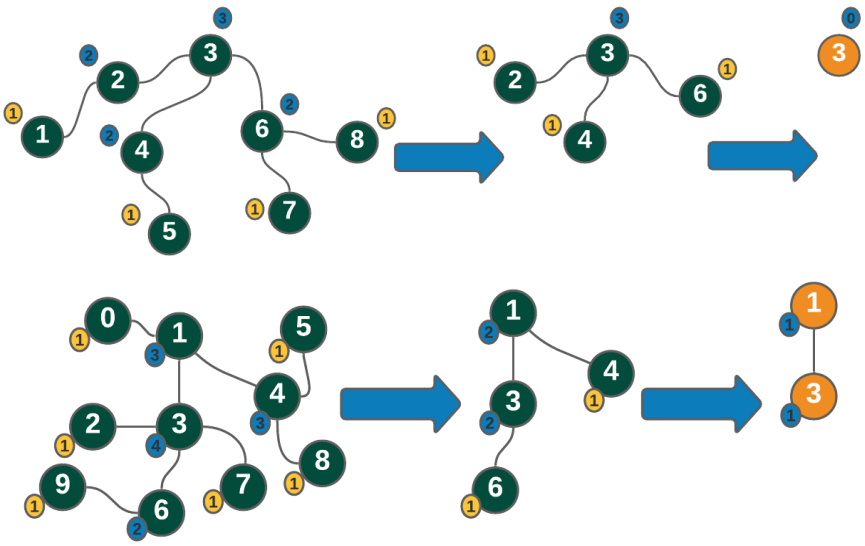
אלגוריתם $Fire$יספק לנו פתרון בסיבוכיות זמן ריצה של $O(|V|)$.

✪ למה

ברגע שאנחנו מורידים עלים מעץ, אנו עדיין נשארים עם עץ

הרעיון של האלגוריתם הוא "לשרוף" את העלים בכל פעם עד שנשאר רק עם המרכז של העץ.

✪ דוגמאות



✪ הגדרה

אקסצנטריות של קודקוד x הוא המרחק הגדול ביותר בין x לכל קודקוד אחר בעץ.

$$Ex(x) = max\{distance(v,x) : v \in V\}$$

מהגדרה זו מיד נובע כי קוטר העץ הוא אקסנטריסיטי ביותר.

✪ משפט

לכל עץ T יש מרכז אחד או שניים.

✪ הוכחה

המרחק המקסימלי מקודקוד v לכל קודקוד $v_i \neq v$ מתרחש רק כאשר v_i הוא עלה.

נתבונן בעץ T בעל יותר משני קודקודים. ידוע לנו של-T יש לפחות שני עלים (לפי $Berge$).

נבצע את השריפה הראשונה:

מוחקים את כל העלים מ-T, ונקבל את הגרף T_1 שהוא שוב עץ.

מחיקת כל העלים מ-T מפחיתה באופן אחיד את האקסצנטריות של כל קודקוד שנשאר באחד (קודקודי העץ T_1).

לפיכך, המרכזים של T הם גם המרכזים של T_1.

נבצע את השריפה השניה:

מוחקים את כל העלים מ-T_1, ונקבל את הגרף T_2 שהוא שוב עץ בעל אותם מרכזים.

נמשיך ככה עם עוד שלבי שריפות עד שנקבל את אחד משני המקרים הבאים:

1. קודקוד בודד שהוא המרכז של T.
2. צלע אשר קודקודי הקצה שלו הם שני המרכזים של T.

∎

✪ מסקנה 1

כאשר לעץ יש מרכז אחד - הרדיוס של העץ שווה למספר השריפות.

כאשר לעץ יש שני מרכזים - הרדיוס של העץ שווה למספר השריפות פלוס 1.

✪ מסקנה 2

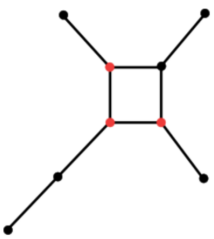
$$2 \cdot radius - 1 \leq diameter \leq 2 \cdot radius$$

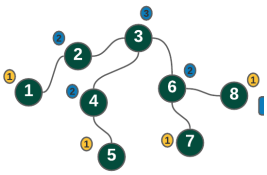
✪ הערה

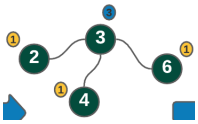
בגרף שהוא לא עץ מספר המרכזים יכול להיות גדול מ-2.

למשל בציור משמאל, יש לגרף 3 מרכזים המסומנים באדום.

וגם $Ex(c_1) = Ex(c_2) = Ex(c_3) = 3$.







✪ שלבי פעולת האלגוריתם

- העלים הם כל הצמתים שיש לו רק שכן אחד ברשימת השכנויות.
- נשמור את כל העלים בתור ואז נשרוף אותם.
- ניצור מערך עזר בגודל מספר הקודקודים, כל תא במערך ייצג את הדרגה של אותו קודקוד.
- כאשר הערך בתא הוא 1 נדע שזה עלה ונכניס אותו לתור שלנו.
- כאשר שרפנו עלה, נוריד דרגה אחת מהשכן שלו.

מימוש בגיטהאב

מימוש האלגוריתם

```
int center1, center2, radius, diameter;
ArrayList<ArrayList<Integer>> tree;

public FireAlgorithm(ArrayList<ArrayList<Integer>> t) {
   this.tree = t;
   this.center1 = -1;
   this.center2 = -1;
   this.radius = 0;
   this.diameter = 0;
   Fire();
}

public void Fire() {
   int number_of_nodes = tree.size();
   ArrayList<Integer> leaves = new ArrayList<>();
   int[] degree = new int[number_of_nodes];

   for(int i = 0; i < number_of_nodes; i++) { // O(|V|)
       degree[i] = tree.get(i).size();
       if(degree[i] == 1)
           leaves.add(i);
   }

   int nodes_to_burn = number_of_nodes;
   while(nodes_to_burn > 2) {
       int number_of_leaves = leaves.size();

       for(int i = 0 ; i < number_of_leaves; i++) {
           int current_leaf = leaves.remove(0); // remove the first
           degree[current_leaf] = 0;
           nodes_to_burn--;
           for(int j = 0; j < tree.get(current_leaf).size(); j++) {
               int neighbour = tree.get(current_leaf).get(j);
               degree[neighbour]--;  // decrease degree of neighbours
               if(degree[neighbour] == 1)
                   leaves.add(neighbour);
```

```
            }
        }
        radius++;
    }

    if(leaves.size()>1) {
        center1 = leaves.remove(0);
        center2 = leaves.remove(0);
        radius++;
        diameter = radius*2 - 1;
    }
    else {
        center1 = leaves.remove(0);
        center2 = center1;
        diameter = radius*2;
    }
}
```

✪ סיבוכיות

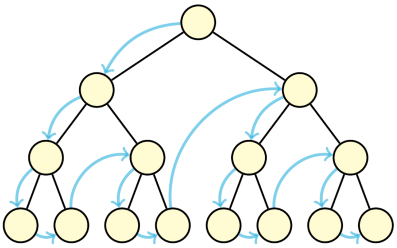
- $O(|V| + |E|)$

אלגוריתם Depth First Search - DFS

✪ הגדרה - חיפוש לעומק

הוא אלגוריתם המשמש למעבר על גרף או לחיפוש בו.
אינטואיטיבית, האלגוריתם מתחיל את החיפוש מצומת שרירותי בגרף ומתקדם לאורך הגרף עד אשר הוא נתקע, לאחר מכן הוא חוזר על עקבותיו עד שהוא יכול לבחור להתקדם לצומת אליו טרם הגיע.
דרך פעולת האלגוריתם דומה במידת מה לסריקה שיטתית של מבוך.

✪ תיאור האלגוריתם



- דרך הפעולה שמנחה את החיפוש לעומק היא רקורסיבית.
- בצורה פורמלית, אלגוריתם חיפוש לעומק בודק את הצומת עליו הוא הופעל, ולאחר מכן מפעיל את עצמו רקורסיבית על כל אחד מהצמתים שמקושרים לצומת אליו הוא הופעל, אם הוא טרם ביקר בהם.
- על מנת לזכור באילו צמתים האלגוריתם כבר ביקר, יש לסמן אותם בדרך כלשהי.
- סימון הצבעים מתבצע באותו משמעות כמו באלגוריתם BFS.

✪ פסאודו קוד

```
dfs(G)
1 for each vertex u ∈ V[G]
2     color[u] = WHITE
3     pred [u]= NIL
4 time = 0
5 for each vertex u ∈ V[G]
6     if color[u] == WHITE
7         then dfs_visit(G,u)
8
9 dfs_visit(G,u)
10 color[u] = GRAY // White vertex u has just been discovered.
11 time = time+1
12 d[u] = time
13 for each v ∈ Adj[u] // Explore edge (u, v).
14     if color[v] == WHITE
15         pred [v] = u
16         dfs_visit (G,v)
17 color[u] = BLACK // Blacken u; it is finished.
18 f [u] =++time
```

✪ סיבוכיות

נתבונן בפסאודו-קוד.

הלולאות בשורות 1-3 וב- 5-7 מבצעות סיבוכיות של $O(|V|)$.

הפונקציה dfs_visit נקראת פעם אחת עבור כל קודקוד $v \in V$ ורק עבור קודקודים לבנים.

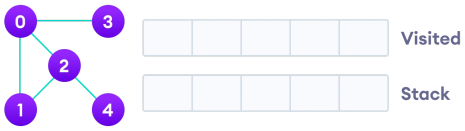
במהלך ביצוע $dfs\_visit(v)$, מאחר ומתקיים בשורות 13-16 הלולאה באופן הבא: $\Sigma_{v \in V}|Adj(v)| = O(|E|)$

אז העלות הכוללת לביצוע הלולאה בשורות 13-16 היא $O(|E|)$.

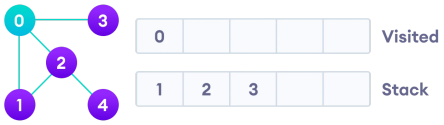
לסיכום: סיבוכיות האלגוריתם DFS הוא $O(|V| + |E|)$.

✪ דוגמה

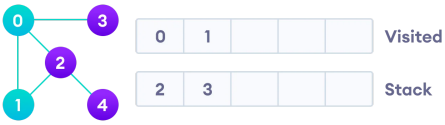
שלב 1



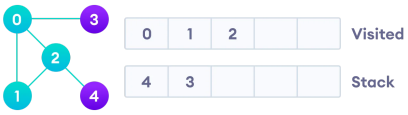
שלב 2



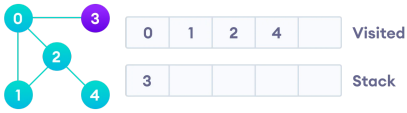
שלב 3



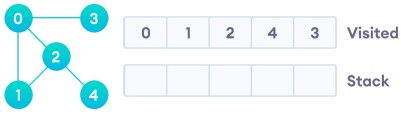
שלב 4



שלב 5



שלב 6



✪ טענה
DFSעובר על כל קודקוד בגרף פעם אחת בלבד.

✪ הוכחה
הפונקציה dfs_visitנקראת על קודקוד $v \in V$רק אם הצבע שלו הוא לבן אז הצבע מיד משתנה.

∎

✪ טענה
אלגוריתם DFS עובר על כל צלע בגרף פעם אחת בלבד.

✪ הוכחה
הפונקציה dfs_visitנקראת על קודקוד פעם אחת בלבד ועבורו נעבור על השכנים שלו בלולאה פעם אחת בלבד.
כלומר האלגוריתם עובר על כל צלע בגרף פעם אחת בלבד.

∎

✪ טענה
אלגוריתם DFS עובר על כל קודקודי הגרף (באותו רכיב קשירות).

✪ הוכחה
נוכיח את הטענה באינדוקציה לפי מרחק kמהמקור לקודקוד כלשהו.

- בסיס האינדוקציה: $k = 0$. האלגוריתם עובר על קודקוד המקור (יש קודקוד אחד ויחיד).
- הנחת האינדוקציה: האלגוריתם עובר על כל קודקודי הגרף שמרחקם עד המקור הוא $k - 1$.
- צעד האינדוקציה: נתבונן בקודקוד $v \in V$כלשהו שמרחקו עד המקור הוא k.
  בגלל שקיים מסלול מ-vעד המקור אז vקשור בעזרת צלע לקודקוד אחר wשמרחקו עד המקור הוא $k - 1$.
  אז האלגוריתם יעבור על v, כאשר הוא עובר על w.

∎

מימוש בגיטהאב

מימוש האלגוריתם

```
public class DFSAlgorithm {
   final static int WHITE = -1, GRAY = 0, BLACK = 1;
   final static int inf = 1000000;
   int[][] matrix;
   int number_of_nodes, time;
   int[] color, previous, first, last;

   public DFSAlgorithm(int[][] mat) {
       this.matrix = mat;
       this.number_of_nodes = mat.length;
       this.color = new int[number_of_nodes];
       this.previous = new int[number_of_nodes];
       this.first = new int[number_of_nodes];
       this.last = new int[number_of_nodes];
   }
```

```
    public void DFS(int source) {
        for(int i = 0; i < number_of_nodes; i++) {
            color[i] = WHITE;
            previous[i] = -1;
            first[i] = 0;
            last[i] = 0;
        }
        this.time = 0;
        DFS_Visit(source);
    }

    private void DFS_Visit(int current) {
        color[current] = GRAY;
        first[current] = ++time;
        for(int i = 0 ; i < number_of_nodes; i++) {
            if(i != current && matrix[current][i] != inf) { // if it's a neighbour

                if(color[i] == WHITE) { // if it's not visited
                    color[i] = GRAY;
                    previous[i] = current;
                    DFS_Visit(i);
                }
            }
        }
        color[current] = BLACK;
        last[current] = ++time;
    }
}
```

✪ הדפסת מסלול

```
public String getPath(int src, int dest) {
    String path = "";
    DFS(src);
    if(color[dest] != WHITE) { // if these nodes are on the same components
        int current = dest;
        path = ""+dest;
        while(current != src) {
            current = previous[current];
            path = current + " -> " + path;
        }
    }
    return path;
}
```

✪ מציאת מעגל בגרף (מכוון או לא מכוון) - פסאודו-קוד

```
HasCircle(G)
     ans = false
     for each vertex u ∈ V[G]
     color[u] = WHITE
     pred[u]= NIL
     for each vertex  u in G and ans==false
          if  color[u] == WHITE
               ans = DFSVisit(u)
     return ans;
end- HasCircle

DFSVisit(int u)
     ans = false
     color[u] = GRAY
     for each vertex  v in Adj(u) and ans==false
          if (color[v]==GRAY and pred[u]≠v)
               ans = true
               getCycle(u, v)
          else if  color[v]==WHITE
               pred[v] = u;
               ans = DFSVisit(v)
          color[u] = BLACK
     return ans;
end-DFSVisit

GetCycle(int u, int v)
     Stack cycle
     x = u
     while x ≠ v
          cycle.push(x)
          x = pred[x]
     end-while
     push(cycle , v)
     push(cycle, u)
     reverse(cycle)
end-GetCycle
```

בניית עץ מרשימת דרגות

✪ הבעיה

☚ בהינתן רשימת דרגות הקודקודים בגרף מסוים, האם היא יכולה להיות רשימת דרגות של עץ?

☚ אם מדובר ברשימת דרגות של עץ, תבנו את העץ.

✪ תזכורת

רשימת דרגות יכולה להיות של עץ אם מתקיים השוויון הבא:

$$\Sigma_{v\in V} deg(v) = 2(|E|) = 2(|V| - 1)$$

✪ דוגמה לבדיקת רשימת דרגות עבור עץ

עבור המערך $deg[]$:

נסכום את המערך ונקבל:

$$\Sigma_{v\in V} deg[v] = 1 + 1 + 1 + 1 + 2 + 2 + 3 + 3 = 14$$

כלומר:

$$2(|V| - 1) = 14$$

אכן מתקיים.

✪ רעיון האלגוריתם

☚ תחילה נסכום את מערך הדרגות ונבדוק אם הוא אכן מקיים את התנאי של $\Sigma_{v\in V} deg(v) = 2(|V| - 1)$.

☚ במידה ואכן מדובר בעץ לפי הבדיקה, אז נפעל בשלבים הבאים.

☚ לפי משפט $Berge$ אנו יודעים כי בכל עץ יש לפחות 2 עלים.

☚ ניקח את הקודקוד הראשון שמייצג עלה ונחבר אותו עם הקודקוד הראשון v במערך $deg[]$ שהוא לא עלה.

☚ לאחר החיבור נחסיר את הדרגה של קודקוד v ב-1 כלומר $deg[v] = deg[v] - 1$.

☚ נמשיך לפעול ככה עד שכל הערכים במערך יהיו 0.

מימוש בגיטהאב

מימוש האלגוריתם

```
public int[] getTree(int[] degrees) {
   int sum = 0;

   for(int degree : degrees) { // O(N)
       sum += degree;
   }
```

```
   if(sum != 2*(degrees.length-1)) {
       System.out.println("This degree array can't generate a tree due to
definition");
       return new int[] {};
   }

   int[] tree = new int[degrees.length];
   Arrays.sort(degrees); // O(N*log(N))
   int j = 0;
   for(int i = 0; i < degrees.length; i++) { // O(N)
       if(degrees[i] > 1){
           j = i;
           break;
       }
   }

   for(int i = 0; i < degrees.length-2; i++) { O(N)
       tree[i] = j;
       degrees[j]--;

       if(degrees[j] == 1) {
           j++;
       }
   }
   tree[degrees.length-1] = degrees.length;
   return tree;
}
```

✪ סיבוכיות

- בדיקות עץ - $O(N)$
- מיון מערך הדרגות $O(N \cdot log(N))$
- מציאת קודקוד ראשון שאינו עלה $O(N)$
- בניית מערך האבות $O(N)$

סך הכל עבור אלגוריתם זה: $O(N \cdot log(N))$

עקרונות Invariant and Monovariant

✪ עקרונות

☚ כאשר פותרים בעיות שמתעסקות בסדרות, רקורסיות או איטרטיביות וצריך לקבוע האם מצב מסוים יכול להיווצר ישנם 2 כלים (עקרונות) שניתן להשתמש בהם:

1. **Invariant** - פונקציה שנשארת קבועה כאשר נעשים שינויים באובייקט.
   תוצאה שונה מערך ה-variant אינה יכולה להתקיים.
2. **Monovariant** - פונקציה שהערך שלה משתנה רק בכיוון אחד, הערך יכול לגדול או לקטון.

הערה

בבניית עץ בעל סדרה נתונה של דרגות, ה-Monovariant הוא פשוט סכום הדרגות (שתמיד נהיה קטן יותר).

✪ דוגמה

☚ ניקח מטריצה שכולה 1 או (-1), נרצה שסכום כל שורה ועמודה יהיה חיובי.

☚ הפעולה המותרת היא להפוך את הסימן עבור שורה ועמודה.

☚ המטריצה בגודל nxm.

✪ טענה

כל פעם שמחליפים סכום של שורה שלילית, הסכום של המטריצה גדל.

✪ הוכחה

יהי sum הסכום של מטריצה nxm.

נניח שסכום המספרים עבור שורה כלשהי היא $x < 0$.

נהפוך את כל הסימנים של המספרים בשורה הזאת, נקבל מערך אחר עם הסכום של כל איבריו כאשר:

$$newSum = sum - x + (-x) = sum - 2x$$

וקיבלנו כי $newSum > sum$ כי x הוא שלילי.

לכן כל צעד באלגוריתם הזה מוביל אותנו למטריצה חדשה עם סכום גדול יותר.

מסקנה

סכום איברי המטריצה הוא Monovariant.

✪ הגדרה

התכנסות - נאמר כי האלגוריתם מתכנס אם הוא חסום, ותמיד מביא אותנו לכך שסכום השורות והעמודות שלו אי-שלילי.

✪ טענה

כאשר מוסיפים עלה לקודקוד פנימי, מספר העלים גדול ב-1.

כאשר מוסיפים עלה לעלה, מספר העלים אינו משתנה.

- לכן מספר העלים הוא Monovariant.

✪ בעיה

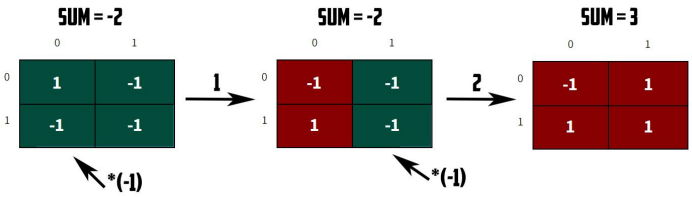
☚ בהינתן מטריצה שמלאה ב- 1 או ב- (1-) בצורה רנדומלית.

☚ נרצה להגיע למטריצה עם הסכום המקסימלי עם כמה שפחות איטרציות

☚ הפעולות הניתנות לביצוע הן ע"י הכפלה של שורה או עמודה

✪ דוגמה

עבור המטריצה באיור, נשים לב תחילה כי החסם העליון הוא 4 כי הרי הסכום המקסימלי הוא כאשר כל תא הוא -1 כלומר הסכום הוא כמספר התאים.



המטריצה מאותחלת בסכום 2 −.

<u>איטרציה ראשונה</u>:

נכפיל את העמודה הראשונה ב- (1 −)

ונקבל כי נשארנו באותו הסכום.

<u>איטרציה שניה</u>: נכפיל את העמודה השניה ב- (1 −)ונקבל כי הסכום כעת הוא 3.

קל לראות שאי אפשר להגיע לסכום גדול יותר מזה.

לכן הגענו לסכום המקסימלי ביותר עבור המטריצה עם 2 איטרציות.

- כדי לפתור בעיה זאת צריך להריץ סימולציה ולבדוק באיזה אופן הכי יעיל לפתור, לכפול עמודות או שורות.

עצים איזומורפיים

✪ הגדרה

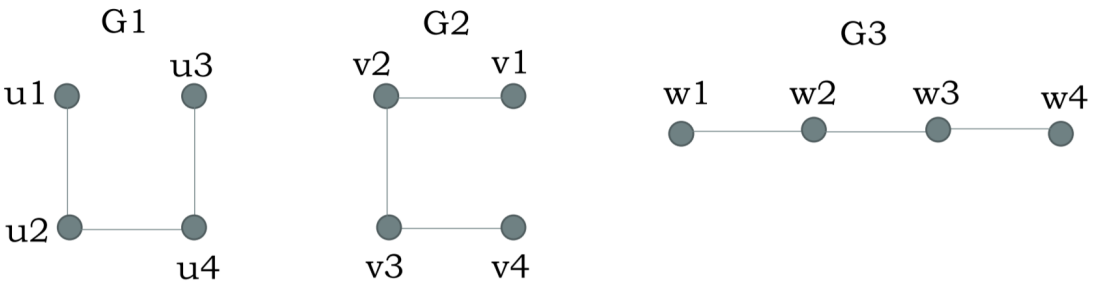
☚ גרפים G-וH-הם איזומורפיים אם קיימת פונקציה $f: V(G) \to V(H)$ חח"ע ועל,
כך שעבור כל $u, v \in V(G)$ מספר הקשתות המקשרות בין u-לv זהה למספר הקשתות המקשרות בין $f(u)$-ל$f(v)$.

☚ במילים אחרות, הם מייצגים את אותו הגרף רק נראים בצורה שונה.

✪ דוגמה

☚ אפשר לראות כי שלושת הגרפים הבאים מייצגים את אותו הגרף רק בצורה שונה.

☚ כי אפשר לראות שהצלע $(u1, u2)$ הוא כמו $(v1, v2)$ שגם כמו $(w1, w2)$ וככה גם עבור שאר הצלעות.

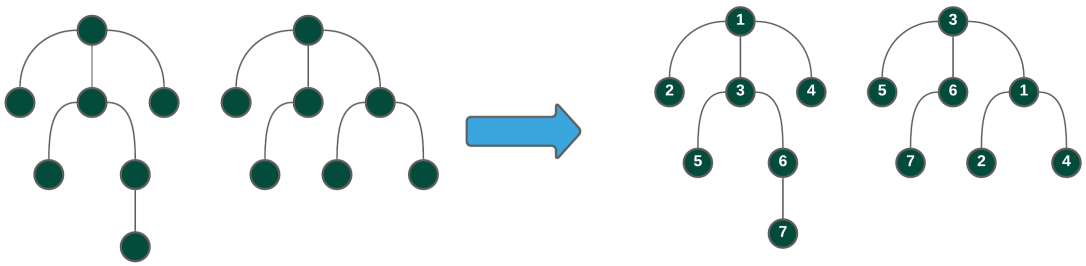


✪ דוגמה

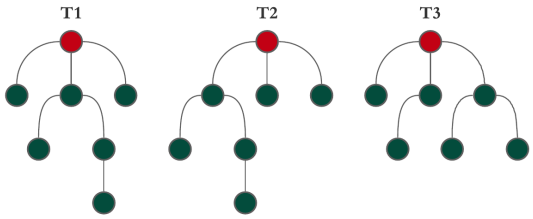
☚ בהינתן 2 עצים המופיעים למטה, האם הם איזומורפיים?

☚ העצים הם איזומורפיים, אפשר להעזר במיספור הצמתים.

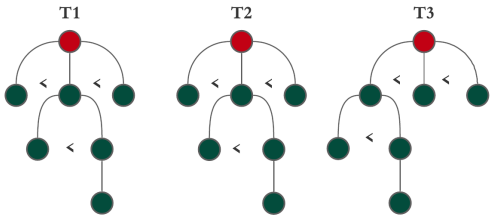
☚ אפשר לדמיין את זה כאילו תפסנו פיזית את צומת מספר 3 ומשכנו אותו למעלה.



✪ דוגמה - עצים עם שורש



- בעצים אלו, אנו מגדירים מיהו אותו קודקוד עליון שממנו אנו מתחילים את כל החישוב.
- איך נדע עכשיו האם הם איזומורפיים?
- נראה שאכן T1 ו-T2 איזומורפיים, אבל T3 לא איזומורפי אליהם כי יוצא ממנו רק בן אחד שהוא עלה בניגוד אליהם.
- 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.
- נתבונן באיור מלמטה לדוגמה.

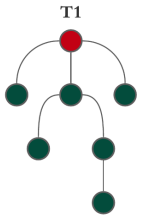


- לאחר המיון שהגדרנו באופן כלשהו, קיבלנו את העצים בצורה המופיע באיור.
- כלומר, אחרי שמיינו את הילדים כל העצים הגיעו לצורת מיון מסוימת ועכשיו הרבה יותר קל להכריע מי איזומורפי למי.
- אפשר לראות ממש בקלות ש- T1 ו-T2 הם איזומורפיים. אבל עכשיו יותר קל גם לראות כמה T3 לא איזומורפי אליהם כי הבן השמאלי שלו שונה מהבן האמצעי של T1 ושל T2.

בפשטות ובקצרה, נרצה להפעיל <u>אותה</u> שיטת סידור/מיון כלשהו על כל הגרפים כדי להבין מי איזומורפי למי.

✪ שיטת פעולה עבור עץ עם שורש

- כדי לעבור על העץ, נסרוק אותו לעומק ונשמור את הפעולות שלנו בכל צעד לפי המוסכמה הבאה:
  - בכל פעם שנרד בעץ נרשום 0.
  - בכל פעם שנעלה בעץ נרשום 1.



- לאחר הסריקה נקבל מחרוזת התואמת את העץ.
- אם נצליח למצוא עץ נוסף בעל אותה מחרוזת נוכל לומר שהם איזומורפיים.
- לפני הסריקה צריך קודם למיין את כל הבנים, אחרת ההשוואה לא תקינה.
- לדוגמה, עבור העץ בציור משמאל נקבל את המחרוזת הבאה: 00100100111011

✪ צעדי האלגוריתם ליצירת מחרוזת המתארת עץ עם שורש

1. נסרוק את הגרף לעומק.
2. כל צומת מכילה מחרוזת פרטית המתארת את הסריקה ממנה לבנים שלה ועד לעלים.
3. כשנגיע לעלה, נגדיר את המחרוזת שלו כ-01.
4. כאשר נגדיר 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.

- כאשר נרצה להשוות בין עצים ולראות האם הם איזומורפיים, נפעיל את האלגוריתם על כל אחד מהעצים ונראה האם נקבל מחרוזות שוות.

✪ שיטת פעולה עבור עץ בלי שורש

- נשתמש באלגוריתם Fire כדי למצוא את מרכז העץ, ואז נוכל להפעיל את האלגוריתם עם שורש ולקבל את התשובה.
- השימוש באלגוריתם Fire במקרה זה חייב להיות במימוש שלא הורס את העץ עצמו (כלומר אסור למחוק צמתים).
- הקודקוד יהיה המרכז כי הרי כל העץ יכול לצאת ממנו, ואם נמצא אותו אז נוכל להפעיל את האלגוריתם.
- במידה וקיבלנו מרכז 1 מכל עץ, נשלח אותם כשורשים לאלגוריתם עם השורש.
- במידה קיבלנו 2 מרכזים ב-Fire, נבצע 2 בדיקות על האלגוריתם עם שורש.
- כלומר אם קיבלנו 2 מרכזים $\{a_1, a_2\}$בעץ T_1 וגם 2 מרכזים $\{b_1, b_2\}$בעץ T_2אז נבדוק את אלגוריתם איזומורפי עם שורש עבור השורשים a_1, b_1 אם קיבלנו אותו קוד, נחזיר true, אחרת נבדוק עבור השורשים a_1, b_2.
- במידה וקיבלנו מספר מרכזים שונה בין 2 העצים, נחזיר false, כלומר העצים לא איזומורפיים.

✪ הסבר על המימוש

- מכינים ב-main רשימה של רשימות המייצגות את העץ (לפי שכנויות).
- שולחים את 2 העצים ל-`isIsomorphic` יחד עם השורשים שלהם.
- נפעיל עבור כל עץ את הפונקציה `generateCode` שתייצר את המחרוזת קוד שלו באפסים ואחדות.
- הפונקציה `generateCode` שולחת לפונקציה `getTraversalCode` הפועל בשיטת אלגוריתם DFS, כלומר הוא סורק לעומק בעץ, אם הגענו לעלה נציב "01" במקום שלו לפי המערך מחרוזות.

  אם לא הגענו לעלה, נעבור על כל השכנים של אותו צומת ונפעל בהתאם לפי כללי DFS.
- לאחר מעבר על כל השכנים של אותו הצומת שרשר לו במערך המחרוזות את 1 (כי תמיד חוזרים למעלה בסוף).
- אם 2 המחרוזות שחזרו יהיו שוות, אז העצים איזומורפיים ונחזיר true אחרת false.

מימוש בגיטהאב

מימוש האלגוריתם

```
public class TreeIsomorphism {

   static int WHITE = 0, GRAY = 1, BLACK = 2;

   public boolean isIsomorphic(ArrayList<ArrayList<Integer>> tree1,int root1,
                               ArrayList<ArrayList<Integer>> tree2,int root2) {
       String code1 = generateCode(tree1, root1);
       String code2 = generateCode(tree2, root2);
       return code1.equals(code2);
   }

   private String generateCode(ArrayList<ArrayList<Integer>> tree, int root) {
       String[] traversalCode = new String[tree.size()];
       Arrays.fill(traversalCode, "0");
       int[] color = new int[tree.size()];
       getTraversalCode(root,tree,traversalCode,color);
       return sortCodes(traversalCode);
   }

   private void getTraversalCode(int current, ArrayList<ArrayList<Integer>> tree,
```

```
                                String[] tc, int[] color) {
        color[current] = BLACK;
        if (tree.get(current).size() == 1) { // if its a leaf
            tc[current] = "01";
        } else {
            for(Integer neighbour : tree.get(current)) {
                if(color[neighbour] == WHITE) {
                    getTraversalCode(neighbour, tree, tc, color);
                    tc[current] += tc[neighbour];
                }
            }
            tc[current]+=1;
        }
    }

    private String sortCodes(String[] traversalCode) {
        Arrays.sort(traversalCode); // O(NlogN)
        String sortedCode = "";
        int current = 1;
        while(current < traversalCode.length && traversalCode[current] != "01") {
            sortedCode += traversalCode[current];
            current++;
        }
        return sortedCode;
    }
}
```

עבור מקרה שאנחנו לא יודעים מה השורש

☚ באותה המחלקה ניצור את הפונקציה הבאה המקבלת את 2 העצים בלי השורשים שלהם.
☚ נפעיל בכל אחד מהם את אלגוריתם Fire הידוע בתור אחד שמוצא את מרכז העץ.
☚ מרכז העץ הוא גם השורש של העץ באופן הגיוני, במידה ויש 2 מרכזים נבחר שורש מהעץ הראשון ונשווה אותו מול 2 השורשים האפשריים של העץ השני.
☚ אחרי שמצאנו את המרכז של כל עץ פשוט נשלח ל-`isIsomorphic` ונמשיך לעבוד כרגיל.
☚ **מימוש נמצא בגיטהאב.**

✪ סיבוכיות זמן ריצה

☚ <u>בהינתן שורש</u> -
- מילוי מערך המחרוזות בהתחלה $O(N)$, הפעלת DFSכלומר $O(|V| + |E|)$ מיון המערך והרכבת הקוד $O(N \cdot log(N))$ - סה"כ $O(N \cdot log(N)) + (|V| + |E|)$

☚ <u>ללא שורש</u> - עם אלגוריתם שריפה $O(|V|)$ולכן נקבל במקרה הגרוע את אותה סיבוכיות זמן ריצה.

אלגוריתם Huffman

✪ מבוא

- אלגוריתם האפמן (Huffman) פותר בעיות של קידוד.
- קוד האפמן נוצר בשיטה לקידוד תווי טקסט ללא אובדן של נתונים, הקוד מספק דחיסת נתונים מרבית.
- השיטה מבוססת על הקצאות אורך הקוד לתווים לפי שכיחותם, תו נפוץ יותר יוצג באמצעות מספר קטן של סיביות.

✪ תיאור הבעיה

- נתונה סדרה ממוינת של הסתברויות שבהן אותיות.
- יש לפתח אלגוריתם שמחזיר את הקידוד האופטימלי של האותיות הנתונות.

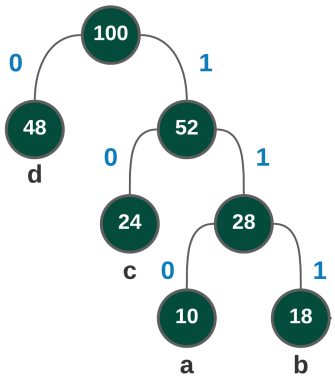
✪ הסבר עם דוגמה

- בהינתן תווים והסתברות של כל תו להופיע:

	תו	הסתברות
1	a	10%
2	b	18%
3	c	24%
4	d	48%

- בעזרת ההסתברויות הנתונות נרצה לקודד את התווים האלה, לדוגמה $a = 110,\ b = 11000,\ c = 1$ אז כאשר נפגוש את הקוד $110, 1100, 1$ נבין כי מדובר ב- "abc".
- בגלל שיש הסתברויות למופע של כל תו, אז בדוגמה שלנו התו d יופיע הכי הרבה פעמים מהשאר.לכן בגלל ש-d בהסתברות גבוהה נרצה לתת לו כמה שפחות ביטים (כי הוא מופיע הכי הרבה אז למה לנו לבזבז באורך של ביטים ולפגוע בזיכרון של התוכנית שלנו).

- האפמן מייצג את השיטה בעץ כאשר בעלים הנמוכים ביותר נרצה שיהיו התווים עם ההסתברות הנמוכה ביותר. למה בעלים הכי נמוכים? - כי אם נשים את התו עם ההסתברות הגבוהה ביותר שם, בגלל שאנחנו משתמשים בו הרבה אנחנו נאלץ לרדת כל פעם לעומק העץ כדי לשלוף אותו וזה פעולה מבוזבזת לכן עדיף לשים שם תווים עם הסתברות נמוכה יותר. האבא של שני העלים יהיה הסכום שלהם.



- את הקוד נשלוף מהעץ כך ששמאלה מוגדר כ-"0" וימינה מוגדר כ-"1".

- לכן עבור הציור של העץ: $d = 0,\ c = 10,\ b = 111,\ a = 110$
כדי לחשב את עלות הקוד נחשב לפי הנוסחה: $B(T) = \Sigma_{c \in C}\, c.freq \cdot d_t(c)$.
כאשר $c.freq$ זה ההסתברות של תו c-ו $d_t(c)$ אורך התו, ככל שקיבלנו $B(T)$ קטן יותר עבור T סימן שהקוד טוב יותר.
ובדוגמה $B(T) = 3 \cdot 18 + 3 \cdot 10 + 2 \cdot 24 + 1 \cdot 42 = 180$.

✪ שיטת האלגוריתם

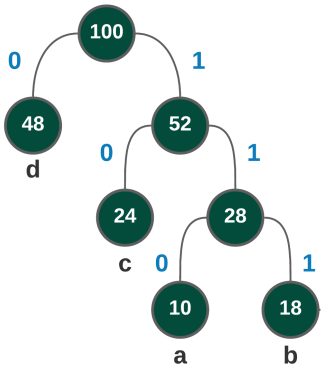
☚ נעזר בדוגמה שהצגנו מלמעלה.

☚ נבנה טבלה המייצגת את היחסים בעץ, הסכומים שנוצרו וההסתברויות של כל תו.

☚ האינדקסים יהיו כגודל מספר הצמתים בעץ: $2n - 1$ צמתים פנימיים וגם n צמתים של התווים כלומר $2n - 1$ צמתים.

☚ נמלא את הטבלה באופן הבא:

אינדקס	התו	ההסתברות	האבא	ימינה	שמאלה
0	a	10	4	---------	---------
1	b	18	4	---------	---------
2	c	24	5	---------	---------
3	d	48	6	---------	---------
4	---------	28	5	b	a
5	---------	52	6	4	c
6	---------	100	---------	5	3



☚ ומכאן, כדי למשל למשוך את a נעזר בטבלה ובציור של הגרף:

- נלך למקום של a באינדקס 0 ונשאל מי האבא שלו, ונגלה שהאבא באינדקס 4 ולכן נשרשר למסלול $ans = "0"$.
- עכשיו אנחנו עומדים באינדקס 4 ונשאל גם מי האבא שלו ונגלה שזה 5 ולכן נשרשר גם אותו ונקבל $ans = "10"$
- כשנגיע ל-5 נשאל גם אותו מי האבא ונגלה שזה 6 ונשרשר ונקבל $ans = "110"$.
- כשנגיע ל-6 נגלה שההסתברות היא 100 ולכן נבין כי הגענו אל השורש ואפשר לסיים את התוכנית ולהחזיר את הקוד כאשר הוא $ans = "110"$ ואכן זה הדרך להגיע ל-a לפי העץ הנתון.

✪ רעיון והוכחת הנכונות

נממש את הקידוד האפמן.

המטרה היא שצמתים עם שכיחות גבוהה יקבלו פחות ביטים וצמתים עם שכיחות נמוכה יקבלו יותר ביטים.

השיטה היא לבנות עץ מלמטה למעלה, כך שבכל פעם לוקחים בצורה חמדנית את שני התווים עם השכיחות הכי נמוכה ומצרפים אותם תחת צומת "אבא" חדש המקבץ את שניהם תחתיו.

לאחר שיש את העץ, משחזרים את הקידוד, לדוגמא בן שמאלי 0 ובן ימני 1.

אם השכיחויות ממוינות, ניתן בעזרת שימוש בשני תורים לייצר את העץ בסיבוכיות לינארית ובכל רגע נתון יהיה אפשר למצוא את שני האיברים המינימליים במספר פעולות פשוטות ולשמור את האבא שלהם בתור החדש.
לאחר מכן, אם נבחר 2 איברים חדשים המינימלים משני התורים ונציב את התשובה בתור השני, החיבור שלהם יהיה בהכרח יותר גדול מהחיבור הקודם, ולכן המיון נשמר לאורך כל האלגוריתם.

✪ תיאור בניית העץ (הטבלה)

1. מגדירים שני תורים ריקים $q1,\ q2$
2. מכניסים ל-$q1$את כל האינדקסים של התווים (עלי העץ), כך שהעלה הקטן ביותר יהיה בראש התור- $O(N)$
3. כל עוד בשני התורים יש יותר מאיבר אחד:
   a. מתחילים לעבוד החל מהאב הראשון (האינדקס של מספר התווים +1).
   b. מוציאים $(poll)$את שני האיברים הקטנים ביותר (2 הבנים) מהראש $(peek)$ של שני התורים בהתאם.
   c. עבור השורה בטבלה של האב מציבים את הבן השמאלי והימני שלו בהתאם לעמודות.
   d. מציבים אצל הבן השמאלי והימני את אינדקס האב.
   e. המשקל של האב יהיה הסכום של המשקל של הבן הימני והמשקל של הבן השמאלי.
   f. מכניסים את אינדקס האב ל-$q2$.
   g. חוזרים לסעיף a.
4. הצומת האחרון שנשאר באחד מהתורים הוא ראש העץ - סיימנו לבנות את העץ/הטבלה.

✪ פסאודו-קוד עבור O(N)

```
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
```

✪ פסאודו-קוד עבור O(NlogN) (מקרה לא ממוין)

```
HuffmanUnsorted(A): // O(NlogN)
      Sort(A) // by A.freq
      root ← Huffman(A)
      return root
end-HuffmanUnsorted
```

✪ פסאודו-קוד עבור הדפסת קוד הופמן (רקורסיבי)

```
getHuffmanCode(A):
```

```
      root ← Huffman(A)
      HuffmanCode(root)
end-getHuffmanCode

HuffmanCode(root):
      if root.left = NULL and root.right = NULL then:
            print(root.char)
      else:
            print("0" + HuffmanCode(root.left))
            print("1" + HuffmanCode(root.right))
      end-if
end-HuffmanCode
```

מימוש בגיטהאב

מימוש האלגוריתם (אך ורק אם הקלט ממויין)

```
public class Huffman {
   private char[] letters;
   private int[] frequency;
   private String[] huffman_code;
   private int[][] tree;
   private int number_of_letters, number_of_nodes;
   private Queue<Integer> queue1, queue2;
   static int weight = 0, left = 1, right = 2, parent = 3;

   public Huffman(char[] l, int[] f) { // USING 2 QUEUES, O(N)
       letters = l;
       frequency = f;
       number_of_letters = l.length;
       number_of_nodes = 2*number_of_letters -1;
       tree = new int[number_of_nodes][4];
       huffman_code = new String[number_of_letters];
       queue1 = new LinkedList<>();
       queue2 = new LinkedList<>();

       for(int i = 0; i < number_of_letters; i++) {
           tree[i][weight] = frequency[i];
           queue1.add(i);
       }
       generateTree();
       generateCode("",number_of_nodes-1);
   }

   private void generateTree() { // O(N)
       int current_parent = number_of_letters;
```

```
        while(queue1.size() + queue2.size() > 1) {
            /* getting left and right child by priority */
            int current_left = getMinimum();
            int current_right = getMinimum();
            /* add the same parent index to both child */
            tree[current_left][parent] = current_parent;
            tree[current_right][parent] = current_parent;
            /* add the weight to the parent by summing the weights
             *  of left child and right child */
            tree[current_parent][weight] = tree[current_left][weight] +
            tree[current_right][weight];

            /* adding the left and right child to the parent */
            tree[current_parent][left] = current_left;
            tree[current_parent][right] = current_right;
            /* adding the current parent to the second queue */
            queue2.add(current_parent);
            current_parent++;
        }
    }

    private int getMinimum() {
        if(queue1.isEmpty() && queue2.isEmpty()) { return -1;}
        if(queue1.isEmpty()) { return queue2.poll(); }
        if(queue2.isEmpty()) { return queue1.poll(); }
        if(tree[queue1.peek()][weight] > tree[queue2.peek()][weight]) {
            return queue2.poll();
        }
        return queue1.poll();
    }
}
```

בנוסף לאלגוריתם אפשר גם לאסוף את הקודים בהוספת הפונקציות הבאות

```
private void generateCode(String code, int current_node) { // O(N)
    if(current_node < number_of_letters) {
        huffman_code[current_node] = code;
        return;
    }
    generateCode(code + "0", tree[current_node][left]);
    generateCode(code + "1", tree[current_node][right]);
}

public String getCode() {
    String code = "";
```

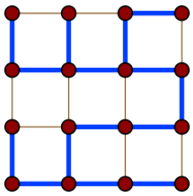
```
    for(int i = 0; i < number_of_letters; i++) {
        code += letters[i] + " = " + huffman_code[i] +",  ";
    }

    return code;
}
```

סיבוכיות זמן ריצה

- מעבר על כל זוגות הצמתים, פעולות השוואה והצבה $O(1)$, מבצעים את פעולות האלו כמספר הצמתים הפנימיים ולכן $O(N)$.
- עבור ההדפסה: עוברים על כל הצלעות בגרף וכשמגיעים לעלה, שומרים את הקוד שלו - $O(N)$.

- סה"כ עבור כל המימוש כולל הדפסה: $O(N)$.

עץ פורש מינימלי



✪ הגדרה - עץ פורש

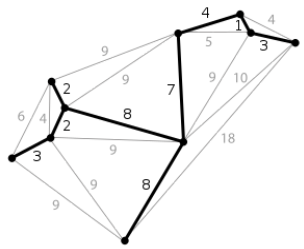
בתורת הגרפים, עץ פורש של גרף קשיר G הוא תת גרף קשיר של G, המכיל את כל צומתי G, ואין לו מעגלים. תת-גרף כזה הוא עץ.

באיור משמאל, אפשר לראות כי נגענו בכל הצמתים של הגרף מבלי לסגור מעגלים.

✪ הגדרה פורמלית - עץ פורש

עץ פורש של גרף קשיר $G = (V, E)$ הוא תת-גרף $T = (V, E_T)$ של G שמכיל את כל קודקודי G וחלק מצלעותיו (כלומר $E_T \subseteq E$), והוא מקיים:

1. T חסר מעגלים.
2. T קשיר (כלומר לכל שני קודקודים קיים מסלול המקשר ביניהם).



✪ הגדרה - עץ פורש מינימלי

עץ פורש מינימלי (אנגלית: **Minimum spanning tree** או בקצרה **MST**) הוא עץ המורכב מתת-קבוצה של צלעות בגרף נתון ומקיים את התכונות:

1. פורש את הגרף - כלומר, כולל את כל הצמתים בגרף.
2. מינימלי בסכום משקלי הקשתות שלו, שהוא הקטן ביותר האפשרי מכל העצים הפורשים.

✪ הגדרה פורמלית - עץ פורש מינימלי

אם משקל הצלע $W(u, v)$ הוא (u, v) ועבור עץ T יהי משקלו $W(T) = \Sigma_{(u,v)\in E_T} W(u, v)$,

העץ הפורש המינימלי הוא זה בעל $W(T)$ הקטן ביותר.

בקצרה - עץ פורש מינימלי הוא עץ פורש בעל משקל כולל נמוך ביותר. יכולים להיות כמה עצים פורשים מינימליים.

כיום משתמשים בשני אלגוריתמים ידועים לגרף לא מכוון:

1. **האלגוריתם של פרים.**
2. **האלגוריתם של קרוסקל.**

שניהם אלגוריתמים **חמדניים**. זמן הריצה המדויק שלהם תלוי במבני הנתונים בהם משתמשים.

אלגוריתם Kruskal

✪ מבוא

האלגוריתם של קרוסקל הוא אלגוריתם **חמדן** לפתרון בעיית **מציאת עץ פורש מינימלי** בגרף ממושקל קשיר לא מכוון.
המטרה היא למצוא תת קבוצה של הקשתות שתיצור עץ המכיל את כל הקודקודים המקוריים, כאשר סכום משקלי הקשתות בתת-קבוצה זו מינימלי. סרטון

✪ תיאור האלגוריתם

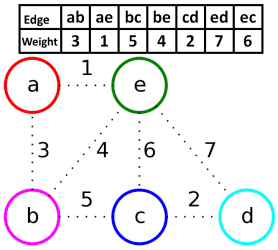
- לפי אלגוריתם זה, כדי לבחור את הקשתות שיימצאו בעץ פורש מינימלי, יש למיין לפי משקליהן תחילה.
- אחר כך יש ליטול לפי הסדר קשת אחר קשת ולהוסיפה לתת הגרף, מהקשת המינימלית עד לקשת הגדולה ביותר במשקלה, כל עוד לא ייווצר מעגל בגרף המתקבל.
- תת הגרף שמתקבל לבסוף הוא עץ פורש מינימלי.
- האלגוריתם הוא חמדן, כיוון שבכל צעד נבחרת הפעולה הנראית אופטימלית באותו שלב - נוטלים את הקשת שמשקלה הקטן ביותר.

✪ סיבוכיות האלגוריתם

סיבוכיות האלגוריתם מושפעת בעיקר מהצורך למיין את הצלעות בתחילת הפעלתו.
בגרף עם קבוצת קודקודים V וקבוצת הצלעות E הסיבוכיות היא $O(|E| \cdot \log|V|)$.
במידה והצלעות ממוינות **מראש**, הסיבוכיות היא $O(|E| \cdot \alpha(|V|))$ כאשר α היא פונקציה הפוכה לפונקציית אקרמן שזה כמעט $\alpha(n) \cong O(1)$.

✪ שלבי האלגוריתם

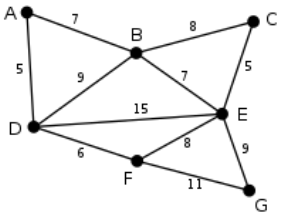
1. בונים T – עץ פורש מינימאלי ריק.
2. ממיינים את צלעות הגרף בסדר עולה (מהקטן לגדול). - $O(|E| \log |E|)$
3. מגדירים מספר קבוצות זרות (בעזרת מבנה הנתונים $union - find$) שכל קודקוד בגרף שייך לקבוצה שלו, כלומר מספר קבוצות שווה ל-$|V|$. (השימוש בפעולות במבנה זה דורש $O(\log|V|)$).
במילים אחרות מגדירים יער המורכב מ-$|V|$ עצים שכל עץ מכיל רק את השורש שלו שהוא קודקוד הגרף.
4. שלפים צלע $e = (u, v)$ בעלת משקל מינימאלי.
5. אם הצלע מחברת בין שני עצים, כלומר הקודקודים u ו-v שייכים לקבוצות שונות,
אז מוסיפים אותה ל-T ומאחדים את העצים. (צלע כזו נקראת "צלע בטוחה" –safe edge).
במקרה שהצלע מחברת שני קודקודים השייכים לאותו עץ (הצלע כזו סוגרת מעגל) מדלגים עליה.
6. בודקים האם מספר צלעות ב-T שווה ל- $|V| - 1$. אם כן האלגוריתם מסתיים, אם לא חוזרים לשלב 4.



✪ דוגמה

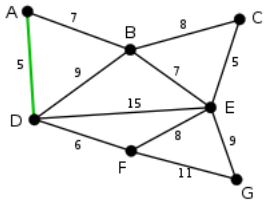
שלב 1

זה הגרף המקורי שלנו.
המספרים ליד הקשתות מציינים את משקלן, אף קשת לא מודגשת.



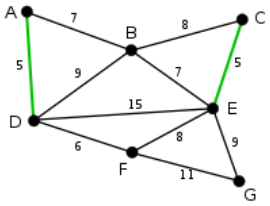
שלב 2

הקשתות AD ו-CE שמשקלן 5 הן הקלות ביותר.
AD נבחרה באופן שרירותי והודגשה.



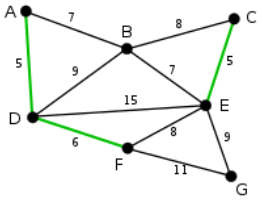
שלב 3

כעת CE שמשקלה 5 היא הקשת הקלה ביותר שאינה סוגרת מעגל, ולכן מודגשת.



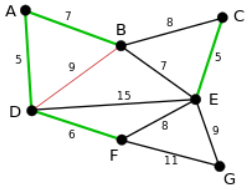
שלב 4

הקשת הקלה ביותר הבאה שאינה סוגרת מעגל היא DF שמשקלה 6.



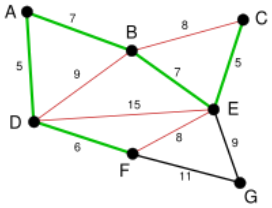
שלב 5

כעת הקשתות הקלות ביותר הן AB ו-BE שאורכן 7.
AB נבחרת באופן שרירותי ומודגשת.
הקשת BD נצבעת באדום כיוון שכבר קיים מסלול בין B ל-D (בירוק), ולכן היא תיצור מעגל (ABD) אם תיבחר.



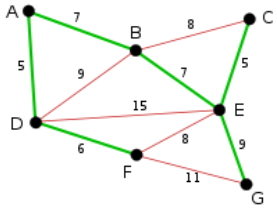
שלב 5

התהליך ממשיך ו-BE נצבעת בירוק.
בשלב זה מספר קשתות נצבעות באדום:
BC (כי היא תיצור את המעגל BCE),
הקשת DE (כי היא תיצור את המעגל DEBA),
ו-EF (שתיצור את המעגל FEBAD).



שלב 7

לבסוף, התהליך מסתיים עם בחירת הקשת EG שמשקלה 9, ומציאת עץ פורש מינימלי (בירוק).



✪ נכונות האלגוריתם

נוכיח את נכונות של אלגוריתם קרוסקל בדרך השלילה. יהיה T עץ שנבנה בעזרת אלגוריתם קרוסקל, S עץ פורש מינימלי ומתקיים אי-שוויון: $W(S) < W(T)$, כאשר $W(S)$ - משקל כולל של S.

- מכיוון שבגרף יכול להיות מספר עצים פורשים מינימליים, ניקח כ-S עץ פורש מינימלי כך שמספר k צלעות משותפות עם T הוא מקסימאלי בין כל עצים פורשים מינימליים.
- יהיה $e_1, e_2, \ldots, e_{i-1}, e_i, \ldots, e_n$ סדרת צלעות שמתקבלת אחרי הרצת אלגוריתם של קרוסקל, (ברור שהסדרה ממוינת בסדר לא יורד).
- יהי $e_i = (a, b)$ הצלע הראשונה בסדרה זו שלא שייכת ל-S ($e_i \notin S$), (צלעות $e_1, e_2, \ldots, e_{i-1}$ שייכות ל-S)
- נוסיף את צלע e_i לעץ S, נקבל גרף חדש $S_1 = S \cup e_i$ בעל n צלעות, אזי ב- S_1 יש מעגל C.
- מכוון ש-T הוא עץ,במעגל C קיימת צלע e_p שלא שייכת ל-T כלומר $e_p \notin T$.
- נשווה את משקלי הצלעות e_p-וe_j: $e_p \in S$ לכן היא לא סוגרת מעגלים עם צלעות $e_1, e_2, \ldots, e_{i-1}$, אבל בשלב i בחרנו ב- e_i ולא ב- e_p, לכן $weight(e_i) \leq weight(e_p)$.
- נסיר צלע e_p מגרף S_1, נקבל עץ: $S_2 = S_1 - \{e_p\} = S \cup \{e_i\} - \{e_p\}$.
  עץ S_2 התקבל מ-S ויש בו k+1 צלעות משותפות עם T.
- נשים לב כי: $weight(S_2) = weight(S_1) + weight(e_i) - weight(e_p) \leq weight(S_1)$
- אבל S_1 הוא עץ פורש מינימלי, המשקל שלו קטן ביותר בגרף G,לכן: $weight(S_2) = weight(S_1)$ ו-S_2 הוא גם עץ פורש מינימלי.

הגענו **לסתירה** עם בחירת S_1, כעץ פורש מינימלי שמספר צלעותיו משותפות עם T מקסימאלי.

∎

✪ פסאודו-קוד

```
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|V|)
```

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

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

FindSet(v): //O(α(n))
      if v = v.parent then:
            return v.parent
      else:
            return FindSet(v.parent)
end-FindSet

Union(u,v): //O(α(n))
      uRoot ⇐ FindSet(u)
      vRoot ⇐ FindSet(v)
      uRoot.parent ⇐ vRoot
end-Union
```

✪ פסאודו-קוד

(תרגיל: להחזיר את המשקל של כל העץ הפורש המינימלי)

```
SumOfMST(G):
      sum ⇐ 0
      T ⇐ Kruskal(G)

      for each e∈E(T) do:
            sum ⇐ sum + e.weight
      end-for

      return sum
end-SumOfMST
```

✪ פסאודו-קוד

(**תרגיל:** עץ פורש מקסימלי)

```
MaximumSpanningTreeSum(G)
      for each e∈E(G) do:
            e.weight ⇐ (-1)*(e.weight)
      end-for

      T ⇐ Kruskal(G)
      sum ⇐ 0

      for each e∈E(T) do:
            e.weight ⇐ (-1)*(e.weight)
            sum ⇐ sum + e.weight
      end-for

      return sum
end-MaximumSpanningTreeSum
```

מימוש בגיטהאב

מימוש האלגוריתם

(**מיותר ללמוד למבחן זה ארוך!** זה טוב רק להבנה, - ללמוד את הפסאודו-קוד במקום).

ממומש בעזרת 3 מחלקות (2 פנימיות).

```
public class Kruskal {

   private Edge[] graph, tree;
   private DisjointSets group;
   private int node_size, tree_size;

   public Kruskal(Edge[] g) {
       this.graph = new Edge[g.length];

       for(int i = 0; i < g.length; i++) {
           graph[i] = new Edge(g[i].v, g[i].u, g[i].weight);
       }
       node_size = 0;
       tree_size = 0;
       for(Edge edge : graph) {
           if(edge.v > node_size) node_size = edge.v;
           if(edge.u > node_size) node_size = edge.u;
```

```
        }
        node_size++;

        group = new DisjointSets(node_size);
        tree = new Edge[node_size-1];

        for (int i = 0; i < node_size; i++) {
            group.make_set(i);
        }
        findMTS();
    }

    private void findMTS() {
        Arrays.sort(graph); // O(|E|log|E|)
        for(Edge edge : graph) {
            if(tree_size < node_size-1) {
                if(group.union(edge.v, edge.u)) {
                    tree[tree_size++] = edge;
                }
            }
            else {
                break;
            }
        }
    }

    public Edge[] getTree() {
        return this.tree;
    }

    /*************************************************
     * DisjointSets CLASS! ***************************
     */
    public static class DisjointSets {
        private int[] parent, rank; //rank[k]>=height of tree number k

        public DisjointSets(int length) {
            parent = new int[length];
            rank = new int[length];
        }
        public void make_set(int k) {
            parent[k] = k;
            rank[k] = 0;
        }
        public boolean union(int v, int u) {
            int root_v = find(v);
            int root_u = find(u);
```

```
            if(root_v == root_u) {  // if its equals = there is a circle.
                return false;
            }
            else {
                if (rank[root_u] > rank[root_v]) {
                    parent[root_v] = root_u;
                } else if (rank[root_v] > rank[root_u]) {
                    parent[root_u] = root_v;
                } else {
                    parent[root_v] = root_u;
                    rank[root_u]++;
                }
                return true;
            }
        }
        public int find(int v) {
            if(parent[v] != v) {
                parent[v] = find(parent[v]);
            }
            return parent[v];
        }
    }
    /************************************************
     * EDGE CLASS! ***********************************
     */
    public static class Edge implements Comparable<Edge> {
        int v, u, weight;
        public Edge(int v, int u, int weight) {
            this.v = v;
            this.u = u;
            this.weight = weight;
        }
        @Override
        public String toString() {
            return "{" +
                    "v=" + v +
                    ", u=" + u +
                    ", weight=" + weight +
                    '}';
        }
        @Override
        public int compareTo(Edge o) {
            return Integer.compare(this.weight,o.weight);
        }
    }
}
```

אלגוריתם Prim

✪ מבוא

האלגוריתם של פרים הוא אלגוריתם **חמדני** המשמש **למציאת עץ פורש מינימלי** בגרף ממושקל לא מכוון.
האלגוריתם מתחיל את בניית העץ מקודקוד פתיחה שנבחר באקראי.
בכל צעד האלגוריתם מוסיף לעץ את הצלע בעלת המשקל המינימלי מבין אלה היוצאות מקודקודי העץ ולא סוגרות מעגל.
בעת מימוש האלגוריתם נעשה שימוש בערימה שמתוכה מוציאים בכל פעם את הצלע המינימלית.

✪ סיבוכיות האלגוריתם

אם משתמשים בערימה (heap) **סיבוכיות** האלגוריתם תהיה $O(|E| \cdot log|V|)$.

☚ באופן כללי היעילות של האלגוריתם של פרים **טובה** מזו של האלגוריתם של קרוסקל. **למרות זאת**, אם הקלט כבר ממוין לפי משקלי הצלעות או כאשר ניתן למיין אותם בזמן לינארי, אז האלגוריתם של קרוסקל יהיה מהיר יותר.

✪ שיטת האלגוריתם

☚ מגדירים תור עדיפויות ומערך $visit[|V|]$ עבור הקודקודים.

☚ מתחילים מקודקוד שרירותי s על הגרף. (s נכנס לתור עם עדיפות 0)

☚ מכניסים לתור העדיפויות את כל השכנים שלו עם העדיפות של משקל הצלע שמגיעה אליהם מ s.

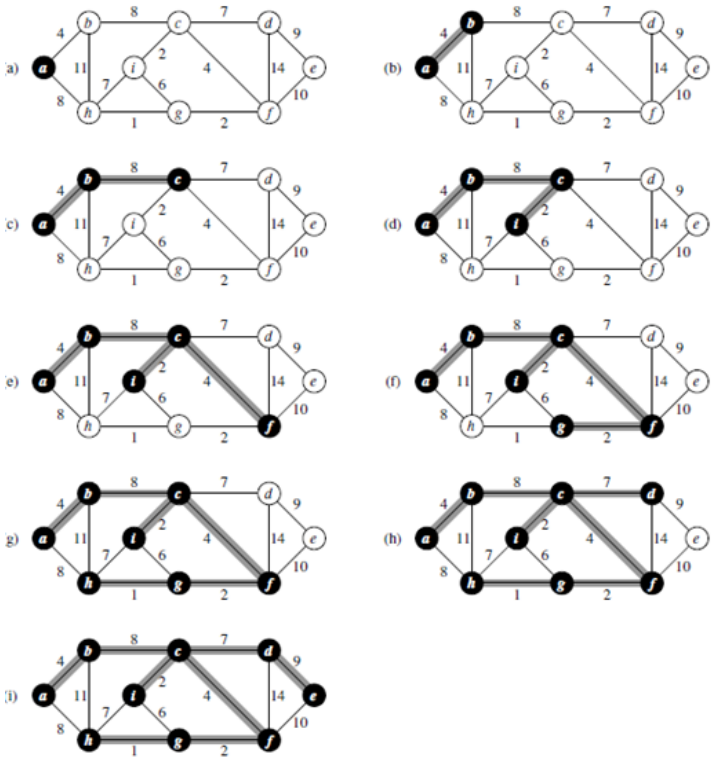
☚ כל עוד התור לא ריק, שולפים את הקודקוד עם העדיפות המינימלית, מוסיפים את הצלע לעץ (בין הקודקוד הנשלף לאבא שלו) ומכניסים לתור את כל השכנים שלא סיימנו איתם עם עדיפות של משקל הצלע שמגיעה אליהם מאותו קודקוד (אם הם כבר בתור אז רק מעדכנים את העדיפות למינימאלי).

✪ פסאודו-קוד

```
Prim(G,root):
      create Tree T ⇐ ∅
      create PriorityQueue Q // Min-PriorityQueue key by key[v]
      create key[|V(G)|]
      create visited[|V(G)|]
      create parent[|V(G)|]
      numEdges ⇐ 0

      for each v∈V(G) do: // O(|V|)
            key[v] ⇐ ∞
            visited[v] ⇐ FALSE
            parent[v] ⇐ NIL
      end-for

      key[root] ⇐ 0

      Q ⇐ V(G) // fill (v,key[v])
```

```
      while (Q is not empty) and (numEdges < |V(G)| - 1) do:

            u ⇐ extractMin(Q)

            for each v∈Adj(u) do:

                  if visited[v] = FALSE and key[v] > weight(v,u) then:
                        key[v] ⇐ weight(v,u)
                        parent[v] ⇐ u
                        decreaseKey(Q, v, weight(v,u) )
                  end-if
            end-for

            visited[u] ⇐ TRUE
            x ⇐ getMin(Q)
            T.add(parent[x], x)
            numEdges ⇐ numEdges + 1

      end-while

      return T

end-Prim
```

✪ דוגמה



שלבים של האלגוריתם בהתאם לדוגמא מלמעלה:

אתחול

קדקודים	Q	P	key
a	a	NIL	0
b	b	NIL	∞
c	c	NIL	∞
d	d	NIL	∞
e	e	NIL	∞
f	f	NIL	∞
g	g	NIL	∞
h	h	NIL	∞
i	i	NIL	∞

שלב 1

קדקודים	Q	P	key
a		NIL	0
b	b	**a**	**4**
c	c	NIL	∞
d	d	NIL	∞
e	e	NIL	∞
f	f	NIL	∞
g	g	NIL	∞
h	h	**a**	**8**
i	i	NIL	∞

סימונים:

משבצת אפורה – קודקוד עזב את התור Q.

אות אדומה – מצב הקדקוד השתנה בהשוואה לשלב הקודם.

שלב 2

קדקודים	Q	P	key
a		NIL	0
b		a	4
c	c	**b**	**8**
d	d	NIL	∞
e	e	NIL	∞
f	f	NIL	∞
g	g	NIL	∞
h	h	a	8
i	i	NIL	∞

שלב 3

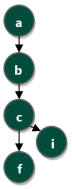
קדקודים	Q	P	key
a		NIL	0
b		a	4
c		b	8
d	**d**	**c**	**7**
e	e	NIL	∞
f	**f**	**c**	**4**
g	g	NIL	∞
h	h	a	8
i	**i**	**c**	**2**

שלב 4

key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c	d	d
∞	NIL	e	e
4	c	f	f
6	**i**	**g**	g
7	**i**	**h**	h
2	c		i

שלב 5

key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c	d	d
10	**f**	**e**	e
4	c		f
2	**f**	**g**	g
7	i	h	h
2	c		i

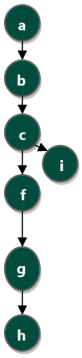


שלב 6

key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c	d	d
10	f	e	e
4	c		f
2	f		g
1	**g**	**h**	h
2	c		i

שלב 7

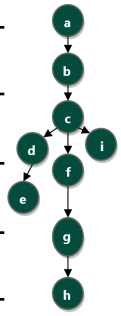
key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c	d	d
10	f	e	e
4	c		f
2	f		g
1	g		h
2	c		i



שלב 8

שלב 9

key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c		d
9	d		e
4	c		f
2	f		g
1	g		h
2	c		i

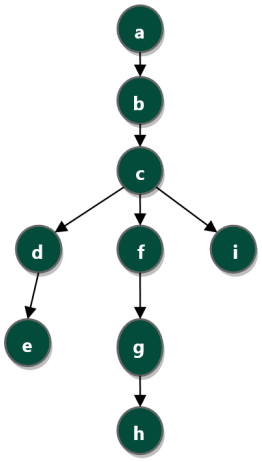


key	P	Q	קדקודים
0	NIL		a
4	a		b
8	b		c
7	c		d
9	d	e	e
4	c		f
2	f		g
1	g		h
2	c		i

התוצאה:

קיבלנו עץ פורש מינימלי שמשקלו 37 (ניתן לבדוק לפי האיור או לפי הסכום של שדות key).

לפי מערך P של קודקודי אבות (parent vertices) ניתן לבנות עץ:



✪ הוכחת נכונות

נוכיח שבל שלב שאנו מוסיפים צלע חדשה ל-T אנו מקבלים תת-עץ של עץ פורש מינימלי.

הוכחה באינדוקציה.

1. **בסיס**. בשלב הראשון של האלגוריתם אנו מוציאים מתור עדיפויות $Q(min\ heap)$ את שורש העץ (root) ומוסיפים לעץ פורש מינימלי T צלע שיוצאת מהשורש ובעלת משקל מינימלי.

2. **נניח** שבשלב כלשהו קיבלנו T_1 תת-עץ של T.
3. לפני שמוציאים קדקוד חדש מ-Q אנו קיבלנו שתי קבוצות זרות של קודקודי הגרף:
קבוצת קודקודים השייכים ל-T_1וקודקודים השייכים לקבוצה $V - T_1$.

נוכיח שצלע חדשה $e = (a, b)$ בעלת משקל מינימאלי שמוסיפים אותה ל-T_1 יוצרת $T_1 \cup \{e\}$תת-עץ של עץ פורש מינימאלי.

נשלים את T_1 עד עץ פורש מינימאלי כלשהו ונסמן אותו T_{min}. אם $e \in T_{min}$ הוכחנו את הטענה.

נניח ש- $e \notin T_{min}$. נתבונן במסלול P ב-T_{min} שמחבר קודקודים a ו-b.

צלע e מחברת קודקודים השייכים לקבוצות זרות (קודקודי T_1 וקדקודי $V - T_1$).

לכן במסלול P קיימת צלע e_1 שמחברת שתי הקבוצות האלה ו-$weight(e) \leq weight(e_1)$.

נחליף ב-T_{min} צלע e_1 בצלע e, נקבל $T_2 = T_{min} - \{e_1\} + e$שהוא גם עץ פורש מינימאלי של G כי יש בו אותו מספר צלעות כמו ב-T_{min}, הוא קשיר ומשקלו לא עולה על משקל של T_{min}.

אז ניתן להשלים את $T_1 \cup \{e\}$ עד עץ פורש מינימאלי שהוא T_2.

∎

מימוש בגיטהאב

מימוש האלגוריתם

(מיותר ללמוד למבחן זה ארוך! זה טוב רק להבנה, - ללמוד את הפסאודו-קוד במקום).

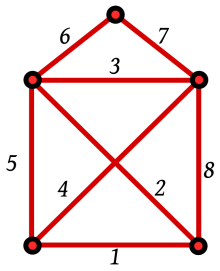
ממומש בעזרת 3 מחלקות (2 פנימיות).

```
public class Prim {
   private ArrayList<Node>[] graph;
   private Edge[] tree;
   private int[] color, parent, weight_arr;
   private Node[] nodes;
   private PriorityQueue<Node> queue;
   private static final int white = 0, grey = 1, black = 2;
   private static final int infinity = Integer.MAX_VALUE;
   private int s;

   public Prim(ArrayList<Node>[] graph, int s) {
       this.graph = graph;
       this.s = s;
       int n = graph.length;
       color = new int[n];
       parent = new int[n];
       weight_arr = new int[n];
       queue = new PriorityQueue<Node>();
       Arrays.fill(parent, -1);
       Arrays.fill(weight_arr, infinity);
       nodes = new Node[n];
       for (int i = 0; i < graph.length; i++) {
           nodes[i] = new Node(i, infinity);
```

```
        }
        tree = new Edge[n-1];
    }

    public void CreateMST() {
        color[s] = grey;
        weight_arr[s] = 0;
        nodes[s].weight = 0;
        queue.add(nodes[s]);
        int edges = 0;
        while(edges < graph.length - 1) {
            Node v = queue.poll();
            if(parent[v.id] != -1) {
                tree[edges++] = new Edge(v.id, parent[v.id], weight_arr[v.id]);
            }
            for(Node n : graph[v.id]) {
                int nid = n.id;
                if(color[nid] == white) {
                    color[nid] = grey;
                    weight_arr[nid] = n.weight;
                    parent[nid] = v.id;
                    nodes[nid].weight = n.weight;
                    queue.add(nodes[nid]);
                }
                else if(color[nid] == grey){
                    if(weight_arr[nid] > n.weight) {
                        weight_arr[nid] = n.weight;
                        nodes[nid].weight = n.weight;
                        queue.remove(nodes[nid]);
                        queue.add(nodes[nid]);
                    }
                }
            }
            color[v.id] = black;
        }

    }

    public Edge[] getTree() {
         return tree;
    }

    public int getSum() {
        int sumWeight = 0;
        for (int i = 0; i < weight_arr.length; i++) {
            sumWeight = sumWeight +weight_arr[i];
        }
```

```
        return sumWeight;
    }
}

class Edge {
    int v, u, weight;

    public Edge(int v1, int v2, int w) {
        this.v = v1;
        this.u = v2;
        this.weight = w;
    }

    @Override
    public String toString() {
        return "(" + u + "," + v + ",w:" + weight + ')';
    }
}

class Node implements Comparable<Node>{
    int id, weight;

    public Node(int id, int weight) {
        this.id = id;
        this.weight = weight;
    }

    @Override
    public int compareTo(Node o) {
        return Integer.compare(this.weight, o.weight);
    }

}
```

מסלול אוילר ומעגל אוילר



✪ הגדרה - מסלול אוילר

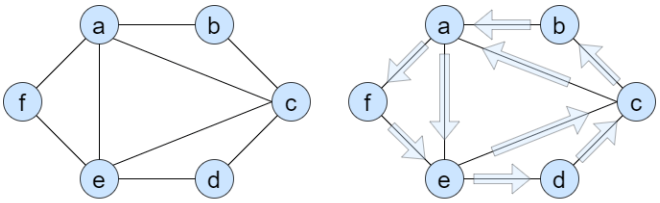
יהי גרף $G(V,E)$ גרף לא מכוון.

מסלול $P_{x,y}$ נקרא **מסלול אוילר** בגרף G, אם הוא עובר בכל הצלעות של G פעם אחת בלבד, ובנוסף $x \neq y$.

✪ הגדרה - מעגל אוילר

יהי גרף $G(V,E)$ גרף לא מכוון.

מעגל אוילר בגרף G הוא מסלול אוילר סגור, כלומר מסלול העובר בכל הצלעות בגרף פעם אחת בלבד והקודקוד ההתחלתי הוא גם קודקוד הסיום.

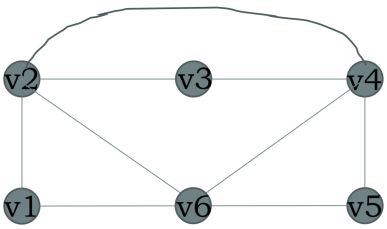


Euler Circuit: a-e-c-a-f-e-d-c-b-a

✪ הגדרה - גרף אוילריאני

יהי גרף $G(V,E)$ גרף לא מכוון.

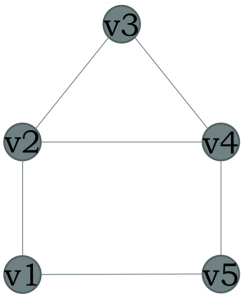
גרף אוילריאני הוא גרף המכיל מעגל אוילר.



✪ משפט - זיהוי גרף אוילריאני

יהי גרף $G(V,E)$ גרף לא מכוון.

הגרף G **אוילריאני** (מעגל אוילר) $\Leftrightarrow$ הגרף G קשיר וכל דרגות הגרף זוגיות.



✪ משפט - זיהוי מסלול אוילר

יהי גרף $G(V,E)$ גרף לא מכוון.

בגרף G יש **מסלול אוילר** $\Leftrightarrow$ הגרף G קשיר ויש בדיוק 2 קודקודים בעלי דרגות אי-זוגיות.

	מסלול אוילר	מעגל אוילר
גרף לא-מכוון	רק לשני צמתים בגרף בעלי דרגה אי-זוגית	כל צומת בגרף חייב להיות בעל דרגה זוגית

✪ אלגוריתם למציאת מעגל אוילר

☚ נבחר קודקוד ראשון ונתחיל ממנו מסלול.

☚ נטייל במסלול ונחלק למקרים:

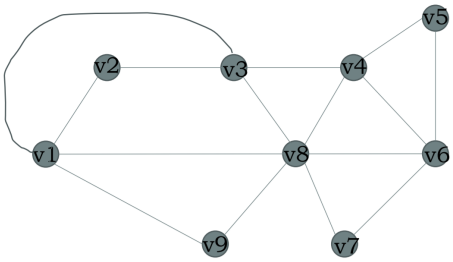
1. אם סגרנו מעגל אבל לא סימנו את כל הצלעות בגרף:
   a. נסמן את הצלע שסגרה את המעגל.
   b. נכניס למחסנית את הקודקוד האחרון שביקרנו בו שסגר את המעגל.
   c. נחזור צעד אחורה לקודקוד הקודם ונמשיך לטייל איתו בצלע אחרת וכך הלאה..
2. אם סגרנו מעגל כשהקודקוד האחרון במסלול הוא הקודקוד הראשון וגם כל הצלעות בגרף מסומנות, זה אומר שסיימנו את האלגוריתם. המחסנית שיצרנו הוא המסלול של מעגל אוילר שלנו.

✪ פסאודו-קוד

```
getEuler(G):
      count ⇐  0
      pathNode ⇐  1
      cycleNode ⇐ 0

      for each v∈V(G) do:
            if deg(v) modulo 2 = 1 then:
                  count ⇐ count + 1
                  pathNode ⇐ v
            else:
                  cycleNode ⇐ v
            end-if
      end-for

      if count > 2 then:
            print(“No path and no cycle”)
            return NULL
      end-if

      if isConnected(G,0) = TRUE then:
            if count = 2 then: // Eulerian Path
                  return generateEuler(G,pathNode)
            end-if

            if count = 0 then:
                  return generateEuler(G,cycleNode)
            end-if
      end-if

      return NULL
end-getEuler
```

```
isConnected(G, src):
      create Queue Q ⇐ ∅
      create color[|V(G)|]

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

      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
                        Enqueue(Q,v)
                  end-if
            end-for

            color[u] ⇐ BLACK
      end-while


      for each v∈V(G) do:
            if color[i] ≠ BLACK then:
                  return FALSE
            end-if
      end-for

      return TRUE
end-isConnected

generateEuler(G,src):
      create Stack stack ⇐ ∅
      create List path ⇐ ∅

      stack.push(src)

      while stack is not empty do:
            u ⇐ stack.top()
            if deg(u) = 0 then:
                  stack.pop()
```

```
                path.add(u)
            else:
                v ← {v|v∈Adj(u)} // get a neighbour
                stack.push(v)
                E(G) ← E(G) - {u,v}
            end-if
        end-while

        return path

end-generateEuler
```



לדוגמה עבור הגרף הבא בציור:

נקבל במבני הנתונים שלנו בסוף התוכנית:

$C = \{9, 1, 3, 4, 6, 7, 8, 6, 5, 4, 8, 3, 2, 1, 8, 9\}$

✪ סיבוכיות

☚ בדיקת גרף או מעגל $O(|V|)$.

☚ בדיקת קשירות בעזרת BFS בסיבוכיות $O(|V| + |E|)$.

☚ החזרת מעגל/מסלול אוילר $O(|E|)$ - כי בכל איטרציה יורדת צלע ולכן יהיו לכל היותר $O(|E|)$ איטרציות כאשר בכל איטרציה זה $O(1)$ (הערה: זה כולל מימוש נכון של מחיקת הצלע בין u ל v ב 2 הכיוונים).

☚ סה"כ סיבוכיות: $O(|V| + |E|)$.

✪ הערה חשובה מאוד

במבחן אם מבקשים להחזיר מעגל אוילר יש קודם כל לבדוק תקינות לפי המשפט שהצגנו למעלה.

1. בודקים האם כל הקודקודים בדרגות זוגיות.
2. בודקים האם הגרף קשיר בעזרת BFS.(רק אם לא אמרו לנו מראש שהגרף קשיר).

מימוש בגיטהאב

מימוש האלגוריתם

```
public Stack<Integer> EulerianCycle(ArrayList<ArrayList<Integer>> graph, int
start_node) {
   if(hasEulerianCycle(graph)) // there is a cycle
       return EulerianAlgorithm(graph,start_node);
   return new Stack<>();
}

private boolean hasEulerianCycle(ArrayList<ArrayList<Integer>> graph) {
   for(ArrayList<Integer> list : graph) {
       if (list.size() % 2 == 1) {
```

```
            return false;
        }
    }
    return true;
}

private Stack<Integer> EulerianAlgorithm(ArrayList<ArrayList<Integer>> graph, int
start_node) {
    if(start_node >= graph.size() || start_node < 0) {
        return new Stack<>();
    }
    Stack<Integer> stack = new Stack<>();
    Stack<Integer> path = new Stack<>();
    stack.push(start_node);

    while(!stack.isEmpty()) {
        int current = stack.peek();

        if(graph.get(current).size() == 0) {
            stack.pop();
            path.push(current);
        }
        else {
            int neighbour = graph.get(current).remove(0);
            stack.push(neighbour);
            graph.get(neighbour).removeIf(x->x==current);
        }
    }

    return path;
}
```

✪ אלגוריתם למציאת מסלול אוילר

☚ נזכר במשפט: בגרף Gיש **מסלול אוילר** $\Leftrightarrow$ הגרף Gקשיר ויש בדיוק 2 קודקודים בעלי דרגות אי-זוגיות.

☚ לכן תחילה נבדוק אם התנאי מתקיים ובמידה וכן, נשמור את האינדקס של הקודקוד בעל הדרגה האי-זוגית (אחד מהשניים זה לא באמת משנה), את האינדקס הזה נשלח לאותו אלגוריתם ששלחנו עבור מעגל אוילר (שיודע לפתור גם עבור מסלולים) והוא יחזיר לנו במחסנית את המסלול.

☚ אם נשאל **במבחן** להחזיר מסלול, יש קודם לבדוק תקינות לפי המשפט (קשירות לפי BFSוגם בדיקת 2 אי-זוגיים).

✪ פסאודו-קוד (מחזיר ערך בוליאני אם קיים מסלול. באלגוריתם למציאת המסלול עצמו, נחזיר את האינדקס של האי-זוגי).

```
HasEulerPath(G):
    counter ← 0
```

```
   foreach veV(G) do
      if degree(v) modulo 2 == 1 then
         counter ← counter+1

if counter == 2 then
   return Yes
else
   return No
```

מימוש בגיטהאב

מימוש האלגוריתם

```java
public Stack<Integer> EulerianPath(ArrayList<ArrayList<Integer>> graph) {
   int start_index = hasEulerPath(graph);
   return EulerianAlgorithm(graph,start_index);
}
private int hasEulerPath(ArrayList<ArrayList<Integer>> graph) {
   int counter = 0;
   int odd_index = -1;

   for(int i = 0 ; i < graph.size(); i++) {
       if(graph.get(i).size() % 2 == 1){
           counter++;
           odd_index = i;
       }
   }
   if(counter == 2) { // then there is an euler path!
       return odd_index;
   }
   return -1;
}
```

אלגוריתם קרוסקל הפוך

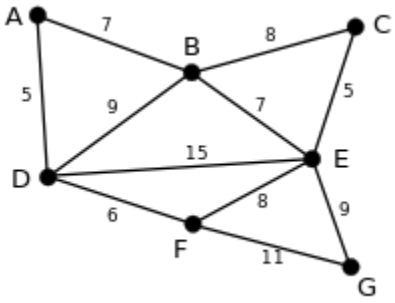
✪ תיאור הבעיה

☚ קלט: גרף $G = (V, E)$- גרף בלתי מכוון, קשיר וממושקל (שבו לכל צלע יש משקל).
☚ פלט: עץ פורש מינימלי.
☚ יישמו את האלגוריתם של מחיקות (קרוסקל הפוך).

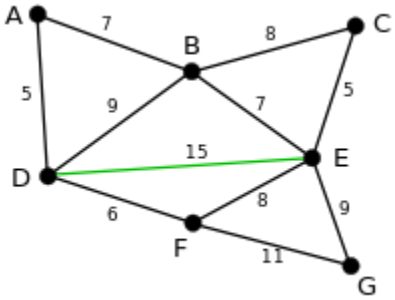
✪ רעיון והוכחת נכונות

- נממש את אלגוריתם ReverseKruskal.
- הרעיון הוא למחוק צלעות כל עוד מחיקת הצלע לא מנתקת את הגרף.
- נמיין את הצלעות מראש לפי סדר יורד של משקלים, ונבחר כל פעם את הצלע עם המשקל המקסימלי.
- אם הצלע היא "גשר" - כלומר **מנתקת** את הגרף ליותר רכיבי קשירות, אז **לא** נמחק אותה.
- התהליך ייעצר אחרי שישארו $|V| - 1$צלעות בגרף.

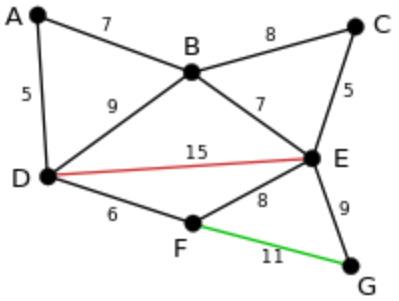
✪ דוגמה



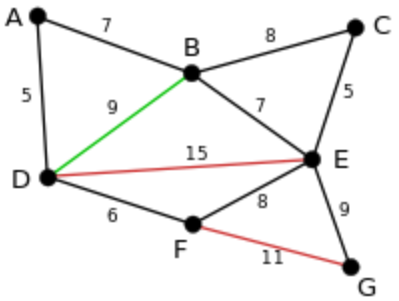
זה הגרף המקורי שלנו.
המספרים ליד הצלעות מציינים את משקל הצלע שלהם.



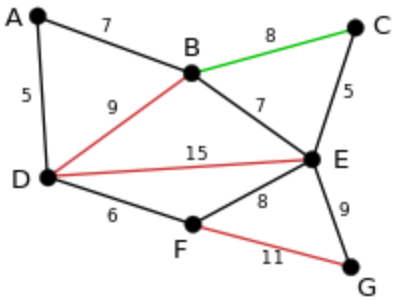
האלגוריתם יתחיל עם הצלע בעל המשקל הגדול ביותר שבמקרה זה הוא **DE** עם משקל צלע של 15.
מכיוון שמחיקת הצלע **DE** אינה מנתקת את הגרף, נמחוק אותו.



הצלע הבא בעל המשקל הגדול ביותר הוא **FG** ולכן האלגוריתם יבדוק אם מחיקת הצלע תנתק את הגרף.
מכיוון שמחיקת הצלע לא תנתק את הגרף, נמחוק את הצלע **FG**.

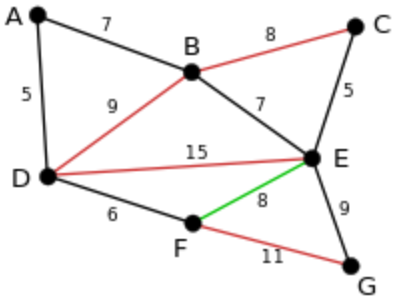


הצלע הבא בעל המשקל הגדול ביותר הוא **BD** .
האלגוריתם יבדוק את הצלע הזה וימחק את אותו.

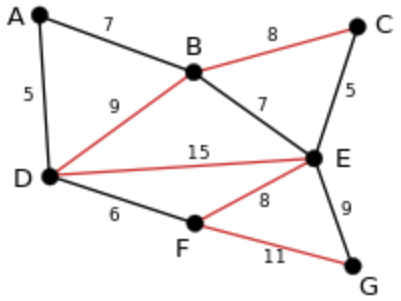


הצלע הבא בעל המשקל הגדול ביותר הוא **EG**, שלא יימחק מכיוון שהוא ינתק את צומת G מהגרף.

לכן, הצלע הבאה למחיקה הוא הצלע **BC**.



הצלע הבא בעל המשקל הגדול ביותר הוא **EF**, ולכן האלגוריתם יבדוק את הצלע זה וימחוק אותו



האלגוריתם יחפש אחר הצלעות הנותרים ולא ימצא צלע נוסף למחיקה;
לכן זהו הגרף הסופי שהוחזר על ידי האלגוריתם.

✪ פסאודו-קוד

```
ReverseKruskal(G): // O(|E|*(|V|+|E|))

      // sort E in descending order by edges weight (high to low).
      Sort(E(G)) // O(|E|log|E|)

      create Queue Q ⇐ ∅
      create Tree T ⇐ ∅

      for each edge∈E(G) do: // O(|V|)
            Enqueue(Q,edge)
            T.add(edge)
      end-for

      size ⇐ |E(T)|

      while size > |V(T)| - 1 do:
            edge ⇐ Dequeue(Q)

            if isBridge(T,edge) = false then:
                  Remove(T,edge)
                  size ⇐ size - 1
            end-if
      end-while

      return T
end-ReverseKruskal

isBridge(T,edge): // O(|V|+|E|)
      T' ⇐ T-{edge}
```

```
      if NumberOfConnectedComponents(T') = 2 then:
            return true
      else:
            return false
end-isBridge

NumberOfConnectedComponents(T'): // O(|V|+|E|)
      counter ⇐ 0
      create color[|V(T')|]

      for each v∈V(T') do: // O(|V|)
            color[v] ⇐ WHITE
      end-for

      for each v∈V(T') do: // O(|V|)
            if color[v] = WHITE then:
                  counter ⇐ counter + 1
                  DFS_REC(T',v,color)
            end-if
      end-for

      return counter

end-NumberOfConnectedComponents

DFS_REC(T',v, color[|V(T')|]): // O(|E| of this component)
      color[v] ⇐ GRAY

      for each u∈Adj(v) do:
            if color[u] = WHITE then:
                  DFS_REC(T',u,color)
            end-if
      end-for
      color[v] ⇐ BLACK
end-DFS_REC
```

✪ סיבוכיות

- המיון עולה $O(|E| \cdot \log |E|)$
- אתחול $O(|E|)$
- שימוש ב-Dequeue עולה לנו $O(\log |E|)$
- שימוש בכל ה-isBridge עולה לנו $O(|V| + |E|)$.
- **סה"כ** סיבוכיות עולה $O(|E| \cdot (|V| + |E|))$.

אלגוריתם Boruvka

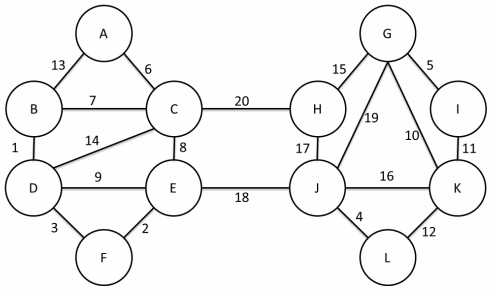
✪ תיאור הבעיה

☚ קלט: גרף $G = (V, E)$- גרף בלתי מכוון, קשיר וממושקל (שבו לכל צלע יש משקל).

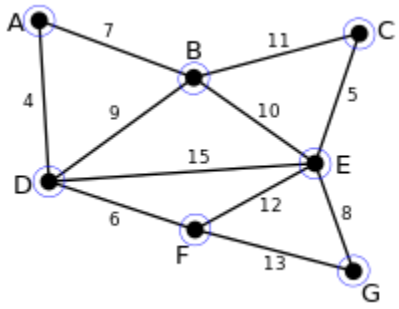
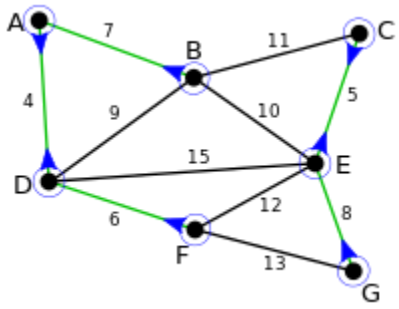
☚ פלט: עץ פורש מינימלי.

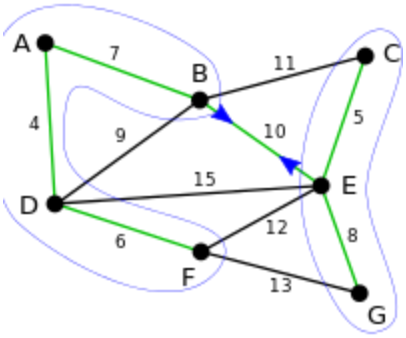
✪ שיטת האלגוריתם

- נתחיל מרכיבי קשירות (כמו union-find בקרוסקל).
- בכל שלב נמצא את הצלע הכי נמוכה שמחוברת לרכיב הקשירות ולא סוגרת מעגל.
- כל עוד לא סיימנו, מצא את הצלעות הנמוכות לכל רכיב (המחברות בין 2 רכיבים שונים) והוסף אותן לעץ.
- ההבדל החשוב בין האלגוריתם של Boruvka לזה של Kruskal או Prim הוא שעם Boruvka אינך צריך למיין את הצלעות או לשמור אותם בתור עדיפות, למרות זאת אין הבדל משמעותי בינו לבין Kruskal והם שווי-סיבוכיות.



✪ דוגמה

זהו הגרף המשוקלל המקורי שלנו. המספרים ליד הצלעות מציינים את משקלם. בתחילה, כל קודקוד בפני עצמו הוא רכיב קשירות יחיד (עיגולים כחולים)	{A} {B} {C} {D} {E} {F} {G}	
באיטרציה הראשונה של הלולאה החיצונית מתווסף צלע המשקל המינימלי מכל רכיב. כמה צלעות נבחרות פעמיים (AD, CE). נותרו שני רכיבי קשירות	{A,B,D,F} {C,E,G}	

באיטרציה השנייה והאחרונה, מתווסף הצלע מהמשקל המינימלי מכל אחד משני הרכיבים הנותרים. זה במקרה אותו הצלע. רכיב אחד נותר לסיום האלגוריתם. הצלע- BD אינו נחשב מכיוון ששתי נקודות הקצה הן באותו רכיב.	{A,B,C,D,E,F,G}	

✪ פסאודו-קוד

```
Boruvka(G)

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

      create Tree T ⇐ ∅
      treeSize ⇐ 0

      while treeSize < |V(G)|-1 do:
            create Edge[|V(G)|] cheapest // init NULL

            for each edge∈E(G) do:
                  root1 ⇐ FindSet(edge.v1)
                  root2 ⇐ FindSet(edge.v2)

                  if root1 ≠ root2 then:

                        if cheapest[root1] = NULL OR
                              OR edge.weight < cheapest[root1].weight then:

                              cheapest[root1] ⇐ edge
                        end-if

                        if cheapest[root2] = NULL OR
                              OR edge.weight < cheapest[root2].weight then:

                              cheapest[root2] ⇐ edge
                        end-if
                  end-if
            end-for

            for each v∈V(G) do:
```

```
                if cheapest[v] ≠ NULL then:
                      v1 ⇐ cheapest[v].v1()
                      v2 ⇐ cheapest[v].v2()

                      if FindSet(v1) ≠ FindSet(v2) then:
                            treeSize ⇐ treeSize + 1
                            edge ⇐ new Edge(v1, v2,
                                                cheapest[v].weight)
                            T.add(edge)
                            Union(v1,v2)
                      end-if
                end-if
          end-for
      end-while

      return T
end-Boruvka

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

FindSet(v): //O(α(n))
      if v = v.parent then:
            return v.parent
      else:
            return FindSet(v.parent)
      end-if
end-FindSet

Union(u,v): //O(α(n))
      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).

חלאס.