מבני נתונים 67109 סיכום מקוצר של כל החומר למבחן - ניצן ברזילי

31 בינואר 2021

- הסיכום לא מחליף למידה מעמיקה למבחן - הוא מיועד לשלב שאחרי שעברתם על הסיכומים המלאים ואתם רוצים לוודא שאתם זוכרים את הנקודות החשובות.
- **מה הוא מכיל:** הסברים כלליים בחלוקה לפי נושאים, את כל האלגוריתמים וזמני הריצה שצריך להכיר (מההרצאות וגם מהתרגולים), שיטות וטיפים לפתירת וחישוב סוגים שונים של בעיות (זמן ריצה רקורסיבי, תוחלת וכו').
- **מה הוא לא מכיל:** את ההוכחות והדוגמאות מההרצאות והתרגולים (לכן לא מספיק ללמוד רק ממנו - חשוב מאוד לראות דוגמאות לכל אלגוריתם כדי להבין טוב איך הוא עובד!). הוא לוקח בערך 30 עמודים כשהסיכום של קרן הוא 160 עמודים על אותו חומר, ואתם יכולים להסיק מזה שהוא מכיל רק את עיקרי הדברים.
- תודה לקרן בן אריה וליחיאל מרזבך שנעזרתי בסיכומים המצוינים שלהם כדי לכתוב את הסיכום הזה!
- אם הסיכום ממש עזר לכם ובא לכם לפרגן לי בקפה, אפשר לעשות את זה בקישור $ko-fi.com/sikumim$.

מבנה המבחן תשפ"א סמסטר א'

- **חלק ראשון (30 נקודות, שעה) - בחירה של 5 שאלות מתוך 7 (עד 5 שורות של 20 מילים לכל שאלה), כל אחת מהן שווה 6 נקודות. שאלות בנושאים הבאים:**
  - חסמים אסימפטוטיים
  - שימוש במשפט האב וניתוח זמן ריצה רקורסיבי
  - מיון (באמצעות או ללא השוואות)
  - טבלאות גיבוב
  - ערמות ותורי עדיפות
  - עצים בינאריים ועצים מאוזנים (עצי AVL)
  - גרפים - BFS, DFS, רכיבי קשירות, עצים פורשים מינימליים, מסלולים קצרים (מנקודה בודדת / כל המסלולים)
  - פעולות על קבוצות נפרדות
- **חלק שני (70 נקודות, שעה) - בחירה של 3 שאלות מתוך 4, כל אחת מהן שווה 23 נקודות.**
  - תהיה שאלה אחת בכל נושא (דגש על טכניקה ואלגוריתמים) - יתכן ששאלה תכלול שני נושאים:
    * מיון, סיבוכיות, שימוש במשפט האב, ניתוח זמן ריצה רקורסיבי
    * גיבוב וטבלאות גיבוב
    * ערמות ותורי עדיפות, עצים בינאריים ועצי AVL
    * גרפים ופעולות על קבוצות נפרדות
  - טכניקות:
    * להסיק ולהוכיח פורמלית חסמים אסימפטוטיים
    * לנסח ולהוכיח זמני ריצה רקורסיביים
    * להוכיח פורמלית את הנכונות וזמן הריצה של אלגוריתמים
    * לנסח ולהוכיח פורמלית שמורת לולאה

* אידוקציה, הנחה בשלילה, דוגמא נגדית
* לתאר אלגוריתמים באמצעות פסודו קוד

– אלגוריתמים:

* לעשות מודיפיקציה לאלגוריתמים מהכיתה
* להשתמש באלגוריתמים מהכיתה כדי לפתור בעיה חדשה
* לכתוב אלגוריתמים חדשים לבעיות שנלמדו בכיתה
* ליצור אלגוריתמים חדשים לבעיות חדשות (בנושאי מיון, גיבוב, חיפוש בעצים, גרפים)
* להעריך ולהשוות בין אלגוריתמים שנלמדו בכיתה כדי לפתור וריאציות שונות של בעיות

כללי

* **שיטות להוכחת נכונות של אלגוריתמים:**

– אינדוקציה: מתאים למשל לאלגוריתם רקורסיבי, ולמקרים בהם אפשר לעשות אינדוקציה על גודל המערך או משהו בסגנון.

– שמורת לולאה: סוג של אינדוקציה מתאים למקרים בהם האלגוריתם הוא איטרטיבי. (כל דבר שאפשר להוכיח בשמורת לולאה אפשר להוכיח באינדוקציה, אך לא הפוך). התבנית של שמורת לולאה היא:

* שמורה: "לאחר כל איטרציה i התכונה (/ אינוריאנטה) p מתקיימת".
* אתחול: מקביל לבסיס האינדוקציה - "לפני / אחרי האיטרציה הראשונה התכונה p מתקיימת" (כלומר ניתן לבצע אתחול גם על המצב לפני האיטרציה הראשונה - לעתים זה יותר קל כי זה נכון באופן ריק).
* שימור: מקביל לשלב האינדוקציה - "נניח שלאחר האיטרציה ה-i התכונה p מתקיימת, ונוכיח שגם לאחר האיטרציה ה-$i+1$ התכונה p מתקיימת".

– חלוקה למקרים: לפי הדרכים השונות בו האלגוריתם יכול להתנהג.

סיבוכיות זמן ומקום

* **סיבוכיות זמן $\mathbf{T(n)}$:** כמות הפעולות הבסיסיות שהאלגוריתם מבצע, כפונקציה של גודל הקלט n.
* **סיבוכיות מקום $\mathbf{S(n)}$:** כמות התאים בזכרון שהאלגוריתם משתמש בהם.
* **חסמים אסימפטוטיים:**

– חסם עליון $O(n)$: לכל היותר n פעולות.

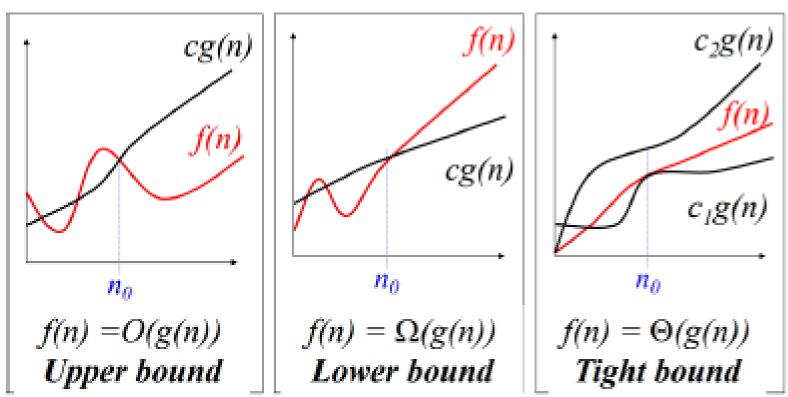
* יהיו f,g פונקציות. נאמר ש-f הוא $O(g)$ אם קיים $c\in\mathbb{R}$ כך שהחל ממקום מסוים $f(n)\le c\cdot g(n)$.

– חסם תחתון $\Omega(n)$: לכל הפחות n פעולות.

* יהיו f,g פונקציות. נאמר ש-f הוא $\Omega(g)$ אם קיים $c\in\mathbb{R}$ כך שהחל ממקום מסוים $f(n)\ge c\cdot g(n)$.

– חסם הדוק $\Theta(n)$: סדר גודל של n פעולות (משני הכיוונים).

* יהיו f,g פונקציות. נאמר ש-f הוא $\Theta(g)$ אם קיימים $c_1,c_2\in\mathbb{R}$ כך שהחל ממקום מסוים $c_1\cdot g(n)\le f(n)\le c_2\cdot g(n)$.



* **תכונות של חסמים אסימפטוטיים:**

- אם יש חסם עליון וגם חסם תחתון, יש חסם הדוק.
- רפלקסיביות - $f = O(f)$
- סימטריה - אם $f = \Theta(g)$ אז $g = \Theta(f)$
- טרנזיטיביות - אם $f = O(g)$ ו־$g = O(h)$ אז $f = O(h)$.
- $O(O(f)) = O(f)$
- אדטיביות - $O(f+g) = O(f) + O(g)$
- הומוגניות - $O(fg) = O(f) \cdot O(g)$
- לכל הלוגים סיבוכיות זהה
- בפולינום, הסיבוכיות נקבעת לפי הדרגה

- **נוסחאות רקורסיביות נפוצות:**

שם	תיאור	נוסחת נסיגה	פתרון
עצרת	הכפלת n ב־$(n-1)!$	$T(n) = T(n-1) + O(1)$	$O(n)$
מספרי פיבונאצ'י	הסכום של מספרי פיבונאצ'י $(n-1)$ ו־$(n-2)$	$T(n) = T(n-1) + T(n-2)$	$O(2^n)$
חיפוש לפי סדר	חיפוש מקסימום או מינימום במערך	$T(n) = T(n-1) + O(1)$	$O(n)$
$Insertion\ Sort$	הפעלת אלגוריתם המיון	$T(n) = T(n-1) + O(n)$	$O(n^2)$
חיפוש בינארי	קריאה רקורסיבית עם חצי מהמערך בכל פעם	$T(n) = T(\frac{n}{2}) + O(1)$	$O(logn)$
מעבר על עץ בינארי	מעבר רקורסיבי על כל הצמתים בעץ	$T(n) = 2T(\frac{n}{2}) + O(1)$	$O(n)$
$Merge\ Sort$	הפעלת אלגוריתם המיון	$T(n) = 2T(\frac{n}{2}) + O(n)$	$O(nlogn)$

- **חישוב זמן ריצה של נוסחת נסיגה רקורסיבית:** ראשית נבדוק אם הנוסחא עומדת בתנאי של משפט האב הפשוט או המורחב. אם כן, נשתמש במשפט האב. אם לא, נפתור בשיטת ההצבה או באמצעות שימוש בעץ הרקורסיה.

  - **משפט האב הפשוט:**
    * אם נוסחת הנסיגה היא מהצורה הבאה $T(n) = aT\left(\frac{n}{b}\right) + n^c$, כאשר
      a - לכמה חלקים מחלקים את הבעיה בכל שלב
      b - הגודל של $node$ בודד (בכמה אנחנו מחלקים את n בכל פעם)
      c - מהי כמות העבודה ב־$node$ בודד בעץ הרקורסיבי
    * נקבל שעומק עץ הרקורסיה הוא $log_b n$.
    * נקבל את החסמים הבאים:

מקרה	חסם
$\frac{a}{b^c} < 1$	$O(n^2)$
$\frac{a}{b^c} = 1$	$O(n^c log_b(n))$
$\frac{a}{b^c} > 1$	$O(n^{log_b a})$

  - **משפט האב המורחב:**
    * אם נוסחת הנסיגה היא מהצורה הבאה $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, נקבל את החסמים הבאים עבור $\varepsilon > 0$:

מקרה	אינטואיציה	חסם
$f(n) = O\left(n^{(\log_b a)-\varepsilon}\right)$	$n^{\log_b a}$ משמעותי יותר	$T(n) = \Theta\left(n^{\log_b a}\right)$
$f(n) = O\left(n^{\log_b a}\right)$	$n^{\log_b a}, f(n)$ משמעותיים באותה מידה	$T(n) = \Theta\left(n^{\log_b a} \log n\right)$
$f(n) = \Omega\left(n^{(\log_b a)+\varepsilon}\right)$ וגם עבור קבוע $c > 1$ מתקיים $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$	$f(n)$ משמעותי יותר	$T(n) = \Theta(f(n))$

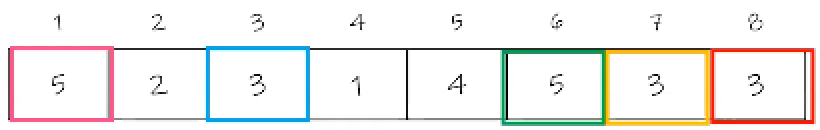
  - **שיטת ההצבה:**
    * פותחים מספר שלבים של הרקורסיה ומזהים חוקיות, מגבשים ממנה נוסחא.
    * מציבים בנוסחה שמצאנו את ה־n האחרון שירוץ במסגרת האלגוריתם ומזהים את החסם שרוצים להוכיח.
    * מוכיחים את החסם באינדוקציה ע"י הצבה של n ושל $T(n)$.
  - **שיטת עץ הרקורסיה:**
    * נחשב:
      מהו הגובה של העץ (מתי נגיע לבעיה בגודל 1)?
      מה כמות העבודה המתבצעת ברמה ה־k בעץ (כמות העבודה בקודקוד יחיד · כמות הקודקודים ברמה ה־k)?
    * נסכום (מ־0 עד גובה העץ פחות 1) את כמות העבודה בכל רמה.

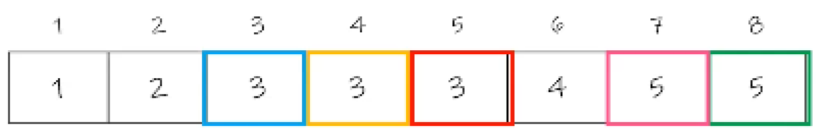
- **חישוב סיבוכיות מקרה ממוצע באמצעות תוחלת:**

- תזכורות מהסתברות:
  * תכונות שימושיות של פונקציות הסתברות:
    · ההסתברות של הקבוצה הריקה היא 0
    · מונוטוניות - אם $A \subseteq B$ אז $P(A) \leq P(B)$
    · תת אדטיביות - $P(A \cup B) \leq P(A) + P(B)$. שוויון יתקיים אם A, B זרים.
  * הגדרות שקולות לתוחלת:
    · $\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega) P(\omega)$
    · $\mathbb{E}[\boldsymbol{X}] = \sum_{x \in Im(X)} x \cdot P(X = x)$
  * תכונות התוחלת:
    · מונוטוניות - אם $X \leq Y$ אז $\mathbb{E}(X) \leq \mathbb{E}(Y)$
    · לינאריות - לכל $a, b \in \mathbb{R}$ מתקיים $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.
    · אי תלות - אם X, Y בלתי תלויים, אז $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$.
    · התוחלת של קבוע היא הקבוע עצמו.
    · תוחלת של מ"מ מציין היא ההסתברות שהוא מקבל ערך 1.

מיון מערכים

- **מיון** מקבל מערך באורך n ומחזיר פרמוטציה של המערך כך שכל איבר קטן מהאיבר מימינו.
- **מיון יציב:** אלגוריתם מיון יציב הוא אלגוריתם מיון בו אם היו במערך המקורי (הקלט) מספר איברים שונים עם אותו הערך בסדר מסוים, הם יופיעו באותו הסדר בדיוק גם במערך הממוין (הפלט).

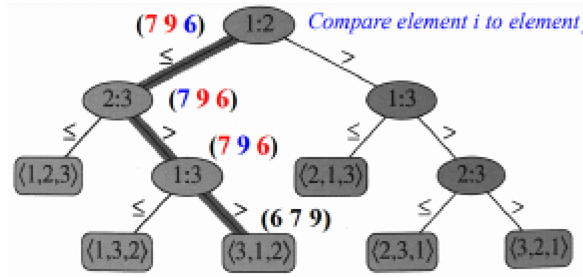




- **סוגי מיון:**
  - באמצעות השוואות: כמו Merge, Insertion, Bubble.

אלגוריתם	זמן הכי טוב	זמן ממוצע	זמן הכי גרוע	מקום
Bubble	n (מעבר אחד)	$\frac{n^2}{2}$ ($\frac{n}{2}$ מעברים)	n^2 (n מעברים)	n
Insertion	n (המערך ממוין)		n^2 (המערך ממוין בסדר הפוך)	n
Merge	$nlogn$	$nlogn$	$nlogn$	$nlogn$
Quick	$nlogn$ (חלוקה מושלמת - בכל פעם שתי תתי הבעיות בגודל $\frac{n}{2}$)	$nlogn$ (חלוקה מאוזנת - חלוקה לתתי בעיות בגדלים $k, n-k$)	n^2 (חלוקה לא מאוזנת - אין חלוקה בכלל, תתי הבעיות הן בגודל $n-1$ ו-0)	n

  * חסם מקום תחתון $\Omega(n)$ (אם המיון מבוצע *In place*).
  * חסם זמן תחתון $\Omega(nlogn)$. מוכיחים זאת באמצעות עץ החלטה (כל צומת היא השוואה בין שני איברים, העלים הם הפרמוטציות האפשריות). הסיבוכיות היא המסלול הארוך ביותר בעץ (מספר ההשוואות), כלומר עומק העץ, שכיוון שזהו עץ בינארי מלא הוא $logn!$ ששקול ל-$nlogn$.



– ללא השוואות (מיון בסיבוכיות לינארית): מצריכים הנחות מקלות כדי שאפשר יהיה להשתמש בהם אבל נותנים תוצאות טובות יותר (זמן ריצה לינארי - $O(n)$ במקרה הגרוע).

אלגוריתם	הנחה מקלה נדרשת	זמן ריצה	מאפיינים יחודיים
Counting	עבור מספר טבעי k, כל איברי המערך הם טבעיים בטווח $[0,k]$.	$\Theta(n+k)$ $=\Theta(n)$	יציב, סיבוכיות מקום גבוהה
Radix	לכל איבר במערך יש לכל היותר d ספרות עבור d טבעי כלשהו.	$\Theta(d\,(n+k))$ $=\Theta(n)$	יציב, סיבוכיות מקום גבוהה (d הפעלות של *counting sort*)
Bucket	כל איברי המערך הם מספרים ממשיים בין 0 ל-1 שמתפלגים באופן אחיד.	$O(n)$ במקרה הממוצע, $O(n^2)$ במקרה הגרוע	

- **מיון בועה Bubble Sort - עם השוואות:**

– איך האלגוריתם עובד: עוברים בלולאה על כל הזוגות העוקבים במערך לפי הסדר, ומחליפים ביניהם אם הימני גדול מהשמאלי. עושים כמה מעברים כאלו שצריך (עד n) על המערך עד שהוא ממוין.

– סיבוכיות ריצה:

∗ המקרה הטוב: n (מעבר אחד)

∗ המקרה הממוצע: $\frac{n^2}{2}$ ($\frac{n}{2}$ מעברים)

∗ המקרה הגרוע: n^2 (n מעברים)

```
BUBBLESORT(A)
for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
           do if A[j] < A[j − 1]
                 then exchange A[j] ↔ A[j − 1]
```

- **מיון הכנסה Insertion Sort - עם השוואות:**

– איך האלגוריתם עובד: עבור כל איבר במערך (החל מהאיבר השני), אם הוא קטן מהאיבר שלפניו, נחליף בינו לבין הקודם שלו (והקודם שלו והקודם שלו...) כמה פעמים שצריך עד שנקבל מצב בו הקודם שלו קטן ממנו.

– סיבוכיות ריצה: $T(n)=\sum_{i=1}^{k} c_i t_i$

∗ המקרה הטוב: n (המערך ממוין)

∗ המקרה הגרוע: n^2 (המערך ממוין בסדר הפוך)

INSERTION-SORT(A)	עלות	מספר הפעמים
for $j \leftarrow 2$ to $length[A]$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

- **מיון איחוד Merge Sort ־ עם השוואות:**
  - איך האלגוריתם עובד: מפרקים את המערך לתתי מערכים עד שמגיעים לתתי מערכים באורך 1, ואז מאחדים אותם בחזרה בסדר הנכון באמצעות האלגוריתם $Merge$ (לא התעמקנו בו, בעיקר צריך לדעת שהוא עושה את האיחוד ב־$O(n)$).

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
for i ← 1 to n1
     do L[i] ← A[p + i − 1]
for j ← 1 to n2
     do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
     do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
                j ← j + 1
```

- סיבוכיות ריצה:
  - המקרה הכי טוב, הממוצע והגרוע הוא $nlogn$.
- **מיון מהיר Quick Sort ־ עם השוואות:**
  - תכונה מיוחדת: ממיין In-place מה שמקטין את סיבוכיות המקום ל־$\Omega(n)$.
  - איך האלגוריתם עובד: בוחרים איבר $pivot$. יוצרים פוינטרים של $i=0, j=0$ כאשר i מייצג את מה שקטן מהפיבוט, ו־j את מה שגדול מהפיבוט. רצים בלולאה של j על אורך המערך ־ בכל פעם משווים את האיבר ה־j במערך לפיבוט ־ אם הוא קטן, מקדמים את i באחד ומחליפים בין האיבר ה־i והאיבר ה־j. אם הוא גדול, לא עושים כלום. בסיום מחליפים את $i+1$ והפיבוט.

```
QUICKSORT(A, p, r)
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

  - סיבוכיות ריצה: תלויה באיך המערך מתחלק ביחס ל־$pivot$ (כלומר כמה ה־$pivot$ קרוב למרכז המערך):
    * מקרה טוב ביותר (החלוקה תמיד באמצע): נקבל עץ בינארי $T(n)=T\left(\frac{n}{2}\right)+T\left(\frac{n}{2}\right)+\Theta(n)$ והסיבוכיות היא $\Theta nlogn$.
    * מקרה ממוצע (חלוקה ל־$k, n-k$): נקבל עץ שיש לו חלק מאוזן וחלק לא מאוזן, $T(n)=T(q)+T(n-q-1)+\Theta(n)$, עומק העץ הוא המקסימום מהעלים בעץ, וסיבוכיות הריצה היא גם $\Theta(nlogn)$.
    * מקרה גרוע (ללא חלוקה, כלומר חלוקה ל־$pivot, n-1$): נקבל $T(n)=T(n-1)+\Theta(n)$ והסיבוכיות היא $\Theta\left(n^2\right)$.
  - **וריאציה אחרת ־ מיון מהיר אקראי:** מיון מהיר שבו איבר ה־$pivot$ לא נבחר לפי מיקום קבוע במערך (כמו בהדגמה למעלה, שם בוחרים ואתו להיות האיבר האחרון), אלא מוגרל בכל איטרציה.
    * זמן הריצה של המקרה הממוצע (התוחלת) יהיה גם כאן $O(nlogn)$.
    * זאת בהתבססות על הנוסחא $T(n)=\Theta(n)+\sum_{i=1}^{n} X_i\cdot(T(\boldsymbol{i}-1)+T(n-i)$ עם X_i מ"מ המציין "האיבר ה־i בגודלו נבחר להיות איבר ה־$pivot$".

- **מיון מניה Counting Sort - ללא השוואות:**
  - הנחה מקלה נדרשת על הקלט: אפשר להשתמש באלגוריתם רק על מערך בו כל האיברים הם מספרים טבעיים בטווח $[0,k]$ עם k טבעי כלשהו. הגיוני להשתמש באלגוריתם אם k הנ"ל הוא בסדר גודל של גודל המערך.
  - איך האלגוריתם עובד:
    * יוצרים מערך של $Counter$ים באורך k ומכניסים לכל תא i במערך את כמות הפעמים שהאיבר i הופיע במערך המקורי.
    * עוברים על מערך ה־$Counter$ים ומחשבים עבור כל תא כמה איברים במערך המקורי קטנים או שווים לו (כי זה מייצג את האינדקס במערך הפלט שבו הוא צריך להיות).
    * עוברים על המערך המקורי מהסוף להתחלה (זה מה שהופך את האלגוריתם ליציב), ומסדרים את האיברים במערך הפלט בהתאם לממצאים. כלומר, לכל איבר i במערך ה־$counter$ים, אנחנו שולפים את הערך j של התא ה־i, ואז:
      · שמים בתא ה־j במערך הפלט את i.
      · מפחיתים את הערך j במערך ה־$counter$ים ב־1.

```
COUNTING-SORT(A, B, k)
for i ← 0 to k
     do C[i] ← 0
for j ← 1 to length[A]
     do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 1 to k
     do C[i] ← C[i] + C[i − 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
     do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
```

- **מיון בסיס Radix Sort - ללא השוואות:**
  - הנחה מקלה נדרשת על הקלט: עבור מספר טבעי d כלשהו, לכל איבר במערך יש לכל היותר d ספרות.
  - איך האלגוריתם עובד: לכל $i \in [1,d]$ ממיינים (באמצעות אלגוריתם מיון יציב כלשהו, לדוגמא מיון מניה שניתן להשתמש בו במקרה הזה כי כל ספרה היא בטווח שבין 0 ל־9) את המערך לפי הספרה ה־i. מתחילים מהספרה הכי פחות משמעותית (אחדות) ומסיימים בספרה הכי משמעותית. כלומר, בכל פעם ממיינים את המערך לפי ספרה אחת - בהתחלה האחדות, אז העשרות וכך הלאה).

```
RADIX-SORT(A, d)
for i ← 1 to d
     do use a stable sort to sort array A on digit i
```

- **מיון דליים Bucket Sort - ללא השוואות:**
  - הנחה מקלה נדרשת על הקלט: כל איברי המערך הם מספרים ממשיים בין 0 ל־1 המתפלגים באופן אחיד (כלומר ההסתברות לכל ערך שווה).
    * ההתפלגות האחידה חשובה כדי להבטיח שהאיברים יתחלקו בין הדליים באופן יחסית שווה, ובכך סביר שימנעו את המקרה הגרוע שבו כל האיברים נמצאים באותו דלי, מה שיגרור סיבוכיות של n^2 (הפעלת $Insertion\ sort$ על מערך בגודל n).
  - איך האלגוריתם עובד:
    * מחלקים את הטווח 0 עד 1 ל־n דליים (כאשר הדלי ה־i מיועד לאיברים שבטווח שבין $\frac{i}{n}$ ל־$\frac{i+1}{n}$).
    * עוברים על המערך ומכניסים כל איבר לדלי המתאים לו.
    * ממיינים כל דלי בנפרד (ע"י אלגוריתם מיון אחר, לדוגמא $Insertion\ sort$).
    * משרשרים את הדליים הממוינים למערך ממוין.

```
BUCKET-SORT(A)
n ← length[A]
for i ← 1 to n
      do insert A[i] into list B[⌊nA[i]⌋]
for i ← 0 to n − 1
      do sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

טבלאות גיבוב

- **פונקצית גיבוב:** יהיו T טבלת גיבוב, U קבוצת מפתחות, $K \subset U$ קבוצת המפתחות שכרגע ממופים לטבלה. $h(k)$ תקרא פונקצית גיבוב אם היא פונקציה מ־U לקבוצה $[0,...,|T|-1]$ - כלומר $h(k)$ ממפה מפתחות מ־U לאינדקסים ב־T.
- **תכונות של טבלאות גיבוב:**
  - שליפת איבר לפי מפתח לוקחת $O(1)$.
  - ערך ריק בטבלה יסומן ב־$Null$.
  - **הנחת הגיבוב האחיד:** ניתן למפות כל ערך לטבלת גיבוב, בלי תלות בשאר האיברים בטבלה.
- **מקדם עומס Load Factor:** מסמן כמה הטבלה מלאה. מחושב ע"י גודל הטבלה חלקי כמות האיברים בטבלה.
- **דרכים לבנות טבלאות גיבוב:**
  - **מיעון ישיר Direct Adressing:** יוצרים טבלה גדולה מספיק ובוחרים פונקצית גיבוב שלא יוצרת התנגשויות. שיטה זו לא יעילה ובזבזנית במקום.

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

  - **מיעון פתוח Open Addressing:** נתמודד ע"י התנגשויות ע"י הפעלה חוזרת של פונקצית הגיבוב כמה פעמים שנצטרך עד שנמצא תא ריק. סיבוכיות הריצה תקבע ע"י מספר האיברים.
    - הכנסה: מפעילים את פונקצית הגיבוב כמה פעמים שצריך עד שמגיעים לתא ריק.
    - חיפוש: מפעילים את פונקצית הגיבוב עד שמוצאים את האיבר או עד שמוצאים תא ריק.
    - מחיקה: אחרי שמוחקים איבר מסוים חייבים לסמן אותו כמחוק כדי להבדיל אותו מתא ריק לטובת הליך החיפוש.

```
HASH-INSERT(T, k)
i ← 0
repeat j ← h(k, i)
         if T[j] = NIL
            then T[j] ← k
                 return j
            else i ← i + 1
  until i = m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i ← 0
repeat j ← h(k, i)
          if T[j] = k
             then return j
          i ← i + 1
   until T[j] = NIL or i = m
return NIL
```

    - הסיבוכיות נקבעת לפי כמות ה"קפיצות" (ההפעלות החוזרות שנצטרך לבצע) עד שנמצא תא ריק, שהיא במקרה הגרוע $O(n)$ (אם יש רק תא ריק אחד והיינו צריכים לעבור על כל שאר התאים כדי להגיע אליו).

* סוגי מיעון פתוח:
  - **בדיקה לינארית:** בוחרים פונקצית גיבוב, מפעילים אותה i פעמים (כמות הפעמים הנדרשת עד שמגיעים לתא ריק) ובכל פעם עושים מודולו של גודל הטבלה כדי לוודא שאנחנו לא חורגים מהטבלה.
    הנוסחא היא $h(k,i) = (h'(k)+i) \mod m$.
  - **בדיקה ריבועית:** בוחרים פונקצית גיבוב, מפעילים אותה i פעמים, כאשר בכל הפעלה מכפילים את i בקבוע c_1 ומוסיפים כפולה של i^2 בקבוע c_2. זה יוצר פחות "גושים" של איברים באותו אזור בטבלה בהשוואה לבדיקה לינארית.
    הנוסחא היא $h(k,i) = \left(h'(k) + c_1 i + c_2 i_k^2\right) \mod m$.
  - **גיבוב כפול:** משתמשים בשתי פונקציות גיבוב h_1, h_2. הפונקציה הראשונה h_1 משמשת בתור פונקצית האש, והשניה h_2 אומרת לנו בכמה "לדלג". כלומר זה מזכיר בדיקה ריבועית, רק שלא מדלגים פעם בריבוע של i אלא בערך $h_2(k)$ כפול i. זה גורם לזה שלכל אלמנט k יש דרך שונה "לקפוץ" בטבלה במקרה של התנגשות, ומפחית משמעותית את כמות ה"גושים" בטבלה, גם בהשוואה לבדיקה ריבועית.
    הנוסחא היא $h(k,i) = (h_1(k) + i \times h_2(k)) \mod m$.

- **מיעון סגור Closed Addressing:** נתמודד עם התנגשויות ע"י יצירת רשימה מקושרת בכל תא, כך שכל תא יכול להכיל כמה איברים שצריך.
  * הכנסה: מפעילים את פונקצית ההאש, מגיעים לתא המתאים, ומוסיפים את האיבר בראש הרשימה המקושרת. זמן הריצה הוא $O(1)$.
  * חיפוש ומחיקה: מפעילים את פונקצית ההאש, מגיעים לתא המתאים, עוברים על הרשימה המקושרת עד שמוצאים את האיבר או עד שמגיעים לסופה. זמן הריצה במקרה הגרוע ביותר הוא O של אורך הרשימה המקושרת הארוכה ביותר.
    - המקרה הגרוע הוא כאשר כל המפתחות מגובבים לאותו תא - זמן ריצה $O(n)$.
    - המקרה הטוב הוא כאשר כל מפתח גובב לתא אחר - זמן ריצה $O(1)$.

- **גיבוב מושלם Perfect Hashing:** נוכל להשתמש בשיטה זו רק כאשר קבוצת המפתחות ידועה מראש ואינה משתנה (לדוגמא רשימה של כל הערים במדינה). בשיטה זו נוכל להשקיע יותר זמן ביצירת הטבלה, אבל אחרי שיצרנו אותה, גם במקרה הגרוע זמן השליפה יהיה $O(1)$ (בניגוד ל-$O(n)$ במקרה הגרוע בשיטות האחרות), כאשר סיבוכיות יצירת הטבלה (שקורית פעם אחת בלבד ותשמש אותנו "לנצח") היא $O(n)$ בתוחלת. יש שתי שיטות לעשות את זה:
  * גיבוב מושלם במקום ריבועי:
    - בונים טבלה בגודל $4n^2 \leq m \leq 8n^2$ (ריבועי) ובוחרים משפחה אוניברסלית H.
    - נגריל פונקצית גיבוב אחת מתוך H. אם היא לא יוצרת התנגשות כלשהי (אנחנו יכולים לבדוק את זה כי ידועים לנו כל המפתחות מראש), נבחר אותה להיות פונקצית הגיבוב של הטבלה. אם היא יוצרת התנגשות, נמשיך להגריל פונקציות מ-H עד שנמצא אחת שלא יוצרת התנגשות (ידוע לנו שיותר מחצי מהן לא יוצרות התנגשות, ושבתוחלת נצטרך רק 2 נסיונות כדי להגריל פונקציה מתאימה).
  * גיבוב מושלם במקום לינארי (מתבססת על גיבוב מושלם במקום ריבועי):
    - ניצור טבלה בגודל $m = n$ (לינארי) ובוחרים משפחה אוניברסלית H.
    - נגריל פונקצית גיבוב לכל מפתח: עבור כל $i \in [1, n]$ נגריל פונקצית גיבוב אחת h מתוך H. אם היא עומדת בתנאי $\sum_{i=1}^{n} (n_i(h))^2 \leq 4n$ (עם $n_i(h)$ קבוצת המפתחות ש-h ממפה לתא ה-i), נבחר אותה (שימו לב שכאן ממש אין דרישה שלא יהיו התנגשויות - סביר מאוד שיהיו). אם לא, נגריל פונקציה אחרת.
    - לכל תא בטבלה, נבנה טבלת גיבוב מושלם במקום ריבועי (כלומר ניצור טבלה בגודל ריבועי ביחס לכמות האיברים שצריכים להכנס בה, ונגריל פונקצית גיבוב h_i עד שנמצא אחת שלא יוצרת התנגשויות), ונמפה לתוכה את כל האיברים ב-$n_i(h)$ תוך שימוש בפונקציה h_i שהגרלנו.
    - כלומר, יש פונקצית גיבוב ראשונה h שאחראית להביא את המפתח לתא בטבלה הראשית, ופונקצית גיבוב שניה h_i עבור התא ה-i, שאחראית למקם את המפתח בתוך הטבלה הפנימית שבתא ה-i.

- **דרכים לבחור פונקצית גיבוב:** נרצה לבחור פונקצית גיבוב באופן חכם, כדי להמנע (כמה שיותר) מהצטברות של "גושים" של איברים באותו אזור בטבלה. המטרה היא שהפיזור של האיברים בטבלה יהיה אחיד, ובמקרה הממוצע מספר הדילוגים יהיה קבוע. יש שתי דרכים לבחור פונקצית גיבוב:
  - **יוריסטיקה (כלל אצבע):** בחירה של פונקציה שתעבוד טוב ברוב המקרים.
    * שיטת החלוקה: ניצור טבלה בגודל m ראשוני (נחלק את מספר האיברים שנרצה למפות במספר ההתנגשויות המקסימלי שאנחנו מוכנים שיהיה, ונבחר את המספר הראשוני הקרוב ביותר מעליו), ונבחר את פונקצית ההאש להיות פשוט חישוב מודולו m על הערך שהוכנס לה. כיוון ש-m ראשוני, אין הרבה מספרים שמתחלקים בו.
      הנוסחא היא $h(k) = k \mod m$
    * שיטת הכפל: נכפול את הערך k שנרצה למפות בשבר A, ונחסר בין kA לבין $\lfloor kA \rfloor$ (כלומר כמה kA קרוב להיות מספר שלם) ונכפיל בגודל הטבלה. כיוון שלרוב המפתחות אין קורלציה עם A, זה יצור התנהגות יותר רנדומלית בהשוואה לשיטת החלוקה.
      הנוסחא היא $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$.

- **בחירה אקראית**

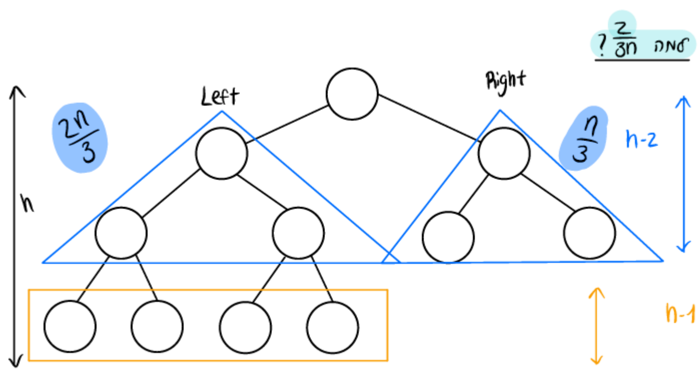
* **גיבוב אוניברסלי:** בחירת פונקצית גיבוב באופן אקראי ובלתי תלוי עבור <u>כל מפתח</u> בנפרד. יש הבטחה שלא קיים קלט יחיד שיתן תמיד את המקרה הגרוע ביותר. תכונות של פונקצית גיבוב אוניברסלי:
  · אם m הוא מספר האיברים בטבלה, ההסתברות שמפתח יגיע לתא מסוים היא $\frac{1}{m}$ (אחידה).
  · ההסתברות ששני איברים שונים ימופו לשני תאים שונים היא $\frac{1}{m^2}$.
  · ההסתברות שתווצר התנגשות היא בדיוק $\frac{1}{m}$.
  · ההסתברות שלתא מסוים יגובב רק מפתח אחד היא $\frac{n}{m}\cdot\left(1-\frac{1}{m}\right)^{n-1}$.
  · תוחלת מספר המפתחות שהוכנסו לתא מסוים בטבלה היא $\frac{n}{m}$.
* **מחלקה (/ משפחה / אוסף) אוניברסלית של פונקציות גיבוב:** קבוצה סופית של פונקציות גיבוב H תקרא <u>אוניברסלית</u> אם"ם לכל זוג מפתחות, מספר פונקציות הגיבוב מתוך H הגורמות להתנגשות של שני המפתחות האלו הוא לכל היותר $\frac{|H|}{m}$. כלומר, אם נבחר פונקצית גיבוב רנדומלית, ההסתברות להתנגשות קטנה או שווה מ־$\frac{1}{m}$.
* **בניית מחלקות אוניברסליות:** נבנה פונקציות גיבוב אוניברסליות באמצעות מספרים ראשוניים ומודולו.
  · <u>דרך א' (מההרצאה) ־ עם שני משתנים:</u>
  נבחר מספר ראשוני p שגדול מכמות המפתחות הטבלה, ונבחר שני מספרים ראשוניים a,b כאשר $a\neq 0$ הקטנים מ־p.
  נבנה את פונקצית הגיבוב $h_{a,b}$ לפי הנוסחא הבאה: $h_{a,b}(k)=((ak+b)\mod p)\mod m$.
  המחלקה האוניברסלית תכיל את כל הפונקציות מהצורה הזו.
  כדי לבחור פונקציה אקראית מהמחלקה האוניברסלית, פשוט בוחרים a,b ראשוניים הקטנים מ־p ויוצרים מהם פונקציה לפי הנוסחא.
  · <u>דרך ב' (מהתרגול) ־ מתאימה לשימוש בטבלה שמקבלת כמפתח וקטור באורך k:</u>
  באופן דומה לדרך הקודמת, ניצור טבלה בגודל ראשוני.
  הפעם עבור מפתח x שהוא וקטור באורך k, נבחר kיה של ערכים $a=(a_1,\ldots,a_k)$ ונגדיר את פונקצית הגיבוב $h_a(x)=\sum_{i=1}^{k}a_ix_i(\mod p)$.
  המחלקה האוניברסלית תכלול את כל הפונקציות מהצורה הזו.
  · <u>דרך ג' (מהתרגול) ־ מתאימה לשימוש בטבלה שמקבלת כמפתח וקטור באורך k:</u>
  נבחר טבלה בגודל $m=2^b$ שהוא חזקה כלשהי של 2.
  נגדיר משפחת פונקציות שמכילה את כל המטריצות בגודל של b שורות ו־k עמודות שמכילות 0 ו־1.
  כדי לבחור פונקצית גיבוב, נגריל מטריצה מהמשפחה, נכפיל אותה במפתח x (נקבל וקטור באורך b) ונעשה $mod\ 2$ כדי לקבל וקטור של אפסים ואחדות שנתרגם אותו (באמצעות ספירה בינארית) לאינדקס בטבלה.
  הנוסחא של פונקצית הגיבוב היא $\underset{\boldsymbol{b\times k}}{h}\cdot\underset{\boldsymbol{k\times 1}}{x}\mod 2=\underset{\boldsymbol{b\times 1}}{v}\mod 2\in\{\boldsymbol{0,1}\}^{\boldsymbol{b}}$.
* **טענות שקשורות לגיבוב אוניברסלי:**
  · **משפט:** תהי h פונקצית גיבוב רנדומלית מתוך H משפחה אוניברסלית, T טבלת גיבוב עם מקדם עומס $\frac{n}{m}$ קטן מ־1. נקבל כי:
  אם מפתח k <u>לא</u> נמצא בטבלה, התוחלת של אורך הרשימה ש־k ימופה אליה היא לכל היותר $\frac{n}{m}$. (כלומר ־ במקרה הממוצע, לא יקרה שכל האיברים ימופו לאותו תא).
  אם מפתח k <u>כן</u> נמצא בטבלה, אז התוחלת של אורך הרשימה ש־k ממופה אליה היא לכל היותר $\frac{n}{m}+1$.
  · **מסקנה:** אם נשתמש בפונקצית גיבוב אוניברסלית וב־$chaining$ לטבלה עם m תאים, זמן הריצה של n פעולות הכנסה, חיפוש או מחיקה יהיה $O(n)$ (כלומר ־ כל אחת מהפעולות מבוצעת בזמן קבוע).
  · **משפט:** אם יש לנו טבלת גיבוב עם מקדם עומס $\alpha=\frac{n}{m}<1$: מספר הבדיקות הצפויות לגיבוב של k הוא:
  <u>בחיפוש לא מוצלח:</u> $\frac{1}{1-\alpha}$ לכל היותר
  <u>בחיפוש מוצלח:</u> $\frac{1}{\alpha}ln\left(\frac{1}{1-\alpha}\right)$ לכל היותר.
  קיבלנו שאם $\alpha=O(1)$ מספר הדילוגים הוא גם $O(1)$.

<u>ערמות ותורי עדיפויות</u>

- **ערמה Heap:**

- מבנה נתונים הממומש ע"י מערך A ומייצג <u>עץ בינארי כמעט מלא</u> (עץ בינארי שמלא בכל הצמתים מלבד האחרונה, שמלאה משמאל לימין עד הצומת האחרונה). כיוון שערמה ממומשת ע"י עץ בינארי, היא מקבלת את התכונות הבאות של עץ בינארי:
  * עומק העץ הוא $\lfloor logn\rfloor$.
  * מספר הצמתים בעץ הוא בין 2^n ל־$2^{n+1}-1$ (זה המקרה רק אם העץ מלא), ניתן לגשת אליה באמצעות קריאה ל־$heapSize[A]$.
  * מספר הצמתים הפנימיים (שאינם עלים) הוא 2^n-1.
  * מספר העלים הוא בין 1 ל־2^n.

- **ערמת מקסימום** היא ערמה המקיימתאת התכונות הבאות:
  - עבור כל צומת בעץ, הערך שלה גדול או שווה מהערכים של הילדים שלה. $A[parent(i)] \geq A[i]$.
  - המקסימום של הערמה הוא תמיד שורש העץ.
- **ערמת מינימום** עובדת באותם עקרונות כמו ערמת מקסימום, רק הפוך (כלומר כל צומת בעץ קטנה מהילדים שלה, והמינימום הוא שורש העץ). מכילה את אותן התכונות כמו ערמת מקסימום באופן סימטרי.
- **פעולות בערמת מקסימום:**
  - $MaxHeapify$ - נשתמש בפעולה זו (שמוודאת שמירה על תכונות ערמת המקסימום) בכל הכנסה, הוצאה או שינוי ערך של איבר. בפעולה זו נוריד את האיבר שיוצר את ההפרה במורד העץ (בכל פעם נהפוך אותו לילד של הבן הגדול ביותר) עד שהוא יהיה גדול מכל הילדים שלו (כלומר תכונת הערמה תחזור להתקיים). נוסחת הנסיגה היא $T\left(\frac{n}{2}\right)+\Theta(1) \leq T(n) \leq T\left(\frac{2}{3}n\right)+\Theta(1)$ וסיבוכיות הריצה שלה היא $logn$ (כיוון שצריך לרדת עד לעומק העץ).




```
MAX-HEAPIFY(A, i)
 l ← LEFT(i)
 r ← RIGHT(i)
 if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
 if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
 if largest ≠ i
    then exchange A[i] ↔ A[largest]
         MAX-HEAPIFY(A, largest)
```

  - **יצירת ערמה:** נקח מערך, נסדר אותו (ישירות, כלומר לפי הסדר בו האיברים מופיעים במערך ובלי להתייחס לגדלים שלהם) בעץ. לאחר מכן נפעיל $MaxHeapify$ מלמטה למעלה - העלים בטוח משמרים את תכונת הערמה אז נדלג עליהם ונתחיל מהשכבה שמעליהם. נמשיך רקורסיבית עד להגעה לקודקוד. סיבוכיות הריצה היא $O(n)$ ולא $O(nlogn)$ כמו שהייתם חושבים אינטואיטיבית בגלל n הפעולות של $MaxHeapify$) בגלל חישוב שנובע ממספר הצמתים בעץ.

```
BUILD-MAX-HEAP(A)
 heap-size[A] ← length[A]
 for i ← ⌊length[A]/2⌋ downto 1
     do MAX-HEAPIFY(A, i)
```

  - **מיון ערמה:** נבנה ערמה, נקח את האיבר הראשון (המקסימום של הערמה), נחליף אותו עם האיבר האחרון (כי במערך ממוין האיבר הכי גדול נמצא בסוף), ונעשה $MaxHeapify$ כדי לוודא שאנחנו עדיין שומרים על תכונת הערמה. נחזור על זה $n-1$ פעמים, בכל פעם עבור השורש החדש שקיבלנו כתוצאה מפעולת ה־$MaxHeapify$. סיבוכיות הריצה היא $O(nlogn)$.

```
HEAPSORT(A)
 BUILD-MAX-HEAP(A)          → O(n)
 for i = A.length downto 2  → n-1   }
     exchange A[1] with A[i]         }
     A.heap-size = A.heap-size - 1   }  O(nlogn)
     MAX-HEAPIFY(A, 1)      → log(n) }
```

- **תור עדיפויות Priority Queue:** מבנה נתונים (ניתן לממש אותו באמצעות ערמה) שהוא קבוצה דינמית A של אלמנטים מסוג key שתומכת בפעולות הבאות:

פעולה	אלגוריתם	סיבוכיות
$Max(A)$	**מוצא את האיבר עם הערך המקסימלי** HEAP-MAXIMUM(A) **return** $A[1]$	1
$Extract-Max(A)$	**מוציא מהערמה את האיבר עם הערך המקסימלי** (נשמור בצד את האיבר המקסימלי, נמחק אותו מהמערך, ונעשה $Heapify$) HEAP-EXTRACT-MAX(A) **if** $heap\text{-}size[A] < 1$ **then error** "heap underflow" $max \leftarrow A[1]$ $A[1] \leftarrow A[heap\text{-}size[A]]$ $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$ MAX-HEAPIFY(A, 1) **return** max	$logn$
$Insert(A, key)$	**מוסיף את האיבר** key **לערמה** (נוסיף את האיבר עם ערך $-\infty$ ונגדיל באמצעות $Increase-Key$) MAX-HEAP-INSERT(A, key) $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$ $A[heap\text{-}size[A]] \leftarrow -\infty$ HEAP-INCREASE-KEY(A, $heap\text{-}size[A]$, key)	$logn$

פעולה	אלגוריתם	סיבוכיות

$Increase-Key(A, i, key)$	**מחליף את הערך של האיבר ה־i במערך, בערך** key
	(אם לא נשמרת תכונת הערמה, נעלה את key במעלה העץ עד שהיא תשמר)
	HEAP-INCREASE-KEY(A, i, key)
	if $key < A[i]$
	  then error "new key is smaller than current key"
	$A[i] \leftarrow key$
	while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
	  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
	    $i \leftarrow \text{PARENT}(i)$
		$logn$

$Delete(A, i)$	**מוחק את האיבר ה־i מהערמה** (נחליף את האיבר ה־i עם האיבר האחרון ונמחק אותו, נעשה $Heapify$ וגם $Increase-Key$ כדי לוודא שהאיבר שהחלפנו במקום הנכון) **Algorithm 7** Delete-Max-Heap(A, i) Exchange($A[i], A[heapsize(A)]$) $heapsize(A) \leftarrow heapsize(A) - 1$ Max-Heapify(A, i) Increase-Key($A, i, A[i]$)	$logn$

- **ערמת חציון:**
  - מוטיבציה: ערמת חציון היא מבנה נתונים בעקרון דומה לערמת מקסימום ומינימום ־ רק שהפעם נרצה להוציא ב־$O(1)$ לא את המקסימום או המינימום אלא את החציון (האיבר שמחצית מהאיברים קטנים ממנו, ומחצית מהאיברים גדולים ממנו ־ כלומר האיבר ה־$\lceil \frac{n+1}{2} \rceil$ בגודלו).
  - מימוש מבנה הנתונים: ניצור מבנה מתונים M שמורכב מערמת מקסימום A וערמת מינימום B. נוודא שהתנאים הבאים נשמרים תמיד:
    * כל הערכים ב־A קטנים או שווים לכל הערכים ב־B.

* הגודל של A תמיד שווה / גדול בדיוק ב־1 מהגודל של B.

– <u>פעולות בערמת חציון:</u>

פעולה	אלגוריתם	סיבוכיות
$GetMedian(M)$	**מוצא את החציון** החציון יהיה תמיד המינימום של הערמה B	1
$Med-Insert(M,v)$	**מכניס את הערך** v **למבנה הנתונים** (נכניס באופן חכם כדי לוודא ששני התנאים נשמרים) * **אם** heapsize(A) = heapsize(B): * **אם** v ≥ A[1]: **נבצע** Insert(B, v). * **אם** v < A[1]: 1. **נבצע** Insert(B, Extract-Max(A)). 2. **נבצע** Insert(A, v). * **אם** heapsize(A) = heapsize(B) - 1: * **אם** v ≤ B[1]: **נבצע** Insert(A, v). * **אם** v > B[1]: 1. **נבצע** Insert(A, Extract-Min(B)). 2. **נבצע** Insert(B, v).	$logn$

- **טבלת יאנג Young Tableu:** טבלת יאנג מגודל $m \times n$ היא מטריצה בגודל $m \times n$ כך שאיברי כל שורה ממוינים בסדר עולה, וגם איברי כל עמודה ממוינים בסדר עולה (כלומר הערכים הולכים וגדלים ככל שמתקרבים לפינה הימנית התחתונה). תאים ריקים בטבלה (שנמצאים בחלק הימני התחתון) מסומנים ב־∞. הסידור של קבוצת איברים בטבלת יאנג הוא <u>לא</u> יחיד.

– <u>תכונות של טבלת יאנג:</u>

* אם הערך $[1,1]$ הוא ∞ אז הטבלה ריקה
* אם הערך $[m,n]$ קטן מ־∞ אז הטבלה מלאה
* לכל איבר בטבלה:
  · כל האיברים שנמצאים מימינו ומתחתיו (כלומר בתת־הטבלה שהוא הפינה השמאלית העליונה שלה) גדולים ממנו.
  · כל האיברים שנמצאים משמאלו ומעליו (כלומר בתת־הטבלה שהוא הפינה הימנית התחתונה שלה) גדולים ממנו.

2	4	12	16
3	5	14	∞
8	9	∞	∞
∞	∞	∞	∞

– <u>פעולות על טבלת יאנג:</u>

* $\boldsymbol{Extract-Min(Y)}$ ־ מחזיר את האיבר עם הערך המינימלי בטבלה, ומוחק אותו ב־$O(m+n)$.
  · נחזיר את האיבר $[1,1]$, נחליף אותו ב־∞, נבעבע אותו ימינה ולמטה (נחליף עם האיבר מימין/מלמטה שקטן ממנו ־ אם שניהם קטנים ממנו, נקח את הקטן מביניהם). יש שלוש אפשרויות באלגוריתם (1 ־ האיבר מתחתינו גדול יותר, 2 ־ האיבר מתחתינו קטן יותר מהאיבר מימיננו, 3 ־ האיבר מימיננו קטן מהאיבר מתחתינו) והוא ממשיך רקורסיבית עד שמגיעים לפינה הימנית־תחתונה.

Algorithm 1 Extract-Min(Y)

```
min = Y[1,1]
Y[1,1] = ∞
BubbleDown(Y,(1,1))
return min
```

Algorithm 2 BubbleDown$(Y,(i,j))$

1. if $i=m$ or $Y[i,j]\le Y[i+1,j]$:
   (a) if $j=n$ or $Y[i,j]\le Y[i,j+1]$:
      i. return
   (b) else:
      i. jump to 3.a
2. if $j=n$ or $Y[i+1,j]\le Y[i,j+1]$:
   (a) Exchange$(Y[i,j],Y[i+1,j])$
   (b) BubbleDown$(Y,(i+1,j))$
   (c) return
3. else:
   (a) Exchange$(Y[i,j],Y[i,j+1])$
   (b) BubbleDown$(Y,(i,j+1))$
   (c) return

* $\boldsymbol{Insert(Y,x)}$ - מכניס את האיבר x לטבלה ה־$O(m+n)$.
   - אפשר להכניס איבר כל עוד הטבלה לא מלאה (כלומר כל עוד התא הכי גדול בטבלה פנוי). נשים את האיבר בפינה הימנית תחתונה ונבעבע אותו כלפי מעלה באופן דומה לאיך שעשינו את זה באלגוריתם הקודם (רק הפוך) באופן רקורסיבי.

Algorithm 3 Insert(Y,m,n,x)

1. If $Y[m,n]<\infty$: return error
2. $Y[m,n]=x$.
3. Bubble$(Y,(m,n))$

Algorithm 4 BubbleUp$(Y,(i,j))$

1. if $i=1$ or $Y[i-1,j]\le Y[i,j]$:
   (a) if $j=1$ or $Y[i,j-1]\le Y[i,j]$:
      i. return
   (b) else:
      i. jump to 3.a
2. if $j=1$ or $Y[i-1,j]\ge Y[i,j-1]$:
   (a) Exchange$(Y[i-1,j],Y[i,j])$
   (b) BubbleUp$(Y,(i-1,j))$
   (c) return
3. else:
   (a) Exchange$(Y[i,j-1],Y[i,j])$
   (b) BubbleUp$(Y,(i,j-1))$
   (c) return

* $\boldsymbol{Sort(A)}$ - משתמש בטבלת יאנג ריבועית בגודל n כדי למיין מערך של n^2 מספרים בזמן $O(n^3)$ מבלי להשתמש באלגוריתמי מיון

(למרות שזה נותן זמן ריצה פחות טוב ממה שהם היו נותנים).

- נכניס לטבלה את n^2 האיברים (סיבוכיות $O(n^3)$ כי כל הכנסה היא $O(n+n)=O(n)$).
- כל עוד הטבלה לא ריקה, נוציא את המינימום מהטבלה (סיבוכיות $O(n^3)$ כי כל הוצאה היא $O(n+n)=O(n)$).

עצי חיפוש בינאריים

- **עץ חיפוש בינארי:** עץ שמכיל שורש, צמתים פנימיים עם לכל היותר 2 ילדים, ועלים. לכל צומת בעץ יש את השדות הורה, ילד ימני, וילד שמאלי. כל האיברים בתת העץ השמאלי יהיו קטנים/שווים לשורש, וכל האיברים בתת העץ הימני גדולים מהשורש.

- **פעולות על עצי חיפוש בינאריים:**

  – חיפוש ב־$O(h)$: משווים את הערך שרוצים למצוא לצומת הנוכחי, וקוראים רקורסיבית לפונקציה עם הילד הימני או השמאלי.

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
   then return x
if k < key[x]
   then return TREE-SEARCH(left[x], k)
   else return TREE-SEARCH(right[x], k)
```

  – מציאת מינימום / מקסימום ב־$O(h)$: יורדים הכי ימינה / הכי שמאלה עד שמגיעים לעלה.

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
     do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
     do x ← right[x]
return x
```

  – מעבר לפי הסדר $In\ Order\ Traversal$ ב־$O(n)$: קוראים לפוקנציה רקורסיבית על תת העץ הימני, מדפיסים את השורש, וקוראים לה רקורסיבית על תת העץ השמאלי.

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

  – מציאת האיבר הבא $Successor$ ב־$O(h)$: מחולק לשני מקרים ־ אם תת העץ הימני לא ריק, העוקב הוא הצומת השמאלי ביותר בתת העץ הימני (שורה 2). אם תת העץ הימני כן ריק, עולים בעץ החל מ־x עד שנתקלים בצומת שהוא הבן השמאלי של אביו.

```
TREE-SUCCESSOR(x)
1 if right[x] ≠ NIL
2    then return TREE-MINIMUM(right[x])
3 y ← p[x]
4 while y ≠ NIL and x = right[y]
5     do x ← y
6        y ← p[y]
7 return y
```

  – הכנסת איבר לעץ ב־$O(h)$: נחפש את הצומת שאפשר להוסיף את המפתח החדש כילד שלה כך שהוא יהיה עלה.

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
          then x ← left[x]
          else  x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                    ▷ העץ T היה ריק
   else  if key[z] < key[y]
            then left[y] ← z
            else  right[y] ← z
```

– <u>מחיקת איבר מהעץ ב־$O(n)$:</u> יש שלושה מקרים אפשריים:

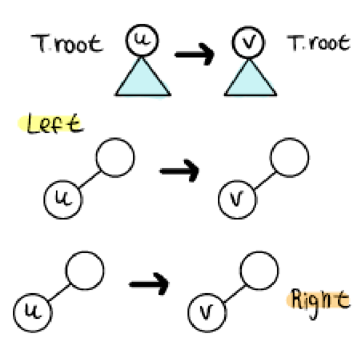
* ל־z אין ילדים: נמחק את z ונעדכן את האבא שלו להיות $null$.
* ל־z יש ילד אחד: נמחק את z ונעלה את הבן שלו ימינה (נעדכן את האבא של הבן שלו להיות האבא של z)
* ל־z יש שני ילדים: נחליף את z בעוקב שלו, ונוציא את העוקב (הוא בעל ילד ימני אחד לכל היותר).

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)          // z has no left child
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)           // z has just a left child
else y = TREE-MINIMUM(z.right)         // y is z's successor
    if y.p ≠ z                         // y lies within z's right subtree
        TRANSPLANT(T, y, y.right)      but is not the root of this subtree.
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)                // Replace z by y.
    y.left = z.left
    y.left.p = y
```

- **טענה:** אם לצומת יש שני ילדים, לעוקב שלה יש לכל היותר ילד אחד.
- **החלפת תת עץ $Transplant$ ב־$O(h)$:**

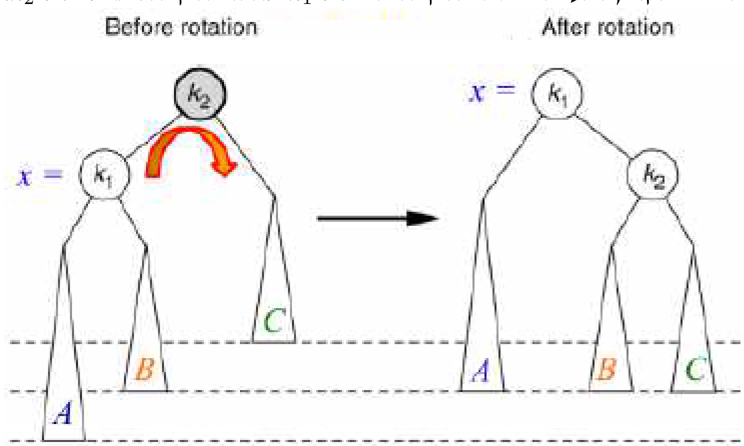
האלגוריתם:

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

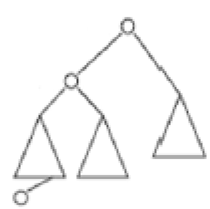
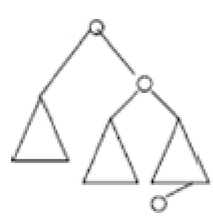
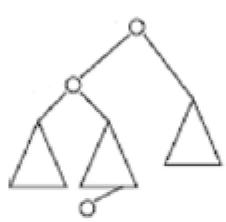
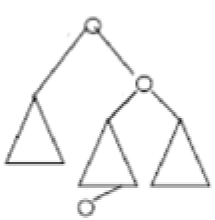


- **עצי חיפוש בינאריים הנבנים באופן רנדומלי:** עץ חיפוש בינארי הנבנה באופן רנדומלי בעל n מפתחות הוא עץ בינארי שנוצר כתוצאה מהכנסה רנדומלית של n מפתחות לעץ ריק. הגובה של עץ כזה הוא $logn$.

- **עץ חיפוש בינארי מאוזן** BST: עצים בינאריים בהם הפרש הגובה בין כל תת עץ הוא לכל היותר 1, זה מבטיח שגובה העץ הוא $\lfloor logn \rfloor$. אם האיזון יופר, נאזן מחדש (איזון יקח $O(logn)$ במקרה הגרוע).
- **עץ** AVL: עץ חיפוש בינארי מאוזן, שכל תת עץ שלו מקיים את התכונה הזו.
  - **מקדם איזון Balance Factor:** לכל צומת בעץ AVL יש תכונה בשם $Height\ difference$ שמוגדרת רקורסיבית (הפרש הגבהים של עלה הוא 0, ושל צומת פנימית הוא ההפרש בין תת העץ השמאלי לתת העץ הימני). בעץ AVL, מקדמי האיזון של כל הצמתים יהיו $\{-1,0,1\}$ ־ -1 אם העץ הימני גדול יותר, 1 עם השמאלי גדול יותר, 0 אם הם שווים.
  - **הכנסה ומחיקה ב־**$O(logn)$: מתבצעות כמו בעץ רגיל, אך במקרה של עץ AVL צריך לשים לב שהן לא גורמות להפרה של תכונת ה־AVL (כלומר ליצור הפרש גבהים של 2 או -2), ולאזן במקרה הצורך.
  - **איזון מחדש ב־**$O(logn)$: נאזן באמצעות "סיבוב" של תתי עצים ־ נעבור על כל האבות של הקודקוד שהוספנו / מחקנו (עד לשורש), ובכל קודקוד אם קיימת הפרה, נתקן באמצעות סיבוב/ים.
    * סיבוב ב־$O(1)$:
      · סיבוב ימינה: אם k_1 הבן השמאלי של k_2, הופכים את k_2 להיות הבן הימני של k_1. הבן השמאלי של k_1 והבן הימני של k_2 נשארים במקום, ומעבירים את הבן הימני של k_1 להיות הבן השמאלי של k_2.



      · סיבוב שמאלה: אותו דבר כמו סיבוב ימינה עם כיוונים הפוכים.
    * סוגי ההפרות האפשריות: האות הראשונה מייצגת באיזה תת עץ הבעיה, והאות השניה מייצגת באיזה תת עץ שלו נמצא הקודקוד שיוצא את ההפרה:

סוג ההפרה	LL	RR	LR	RL
ציור				
גילוי סוג ההפרה	1) גורם האיזון בשורש תת העץ הוא 2 2) גורם האיזון בבן השמאלי של השורש הוא 1	1) גורם האיזון בשורש תת העץ הוא -2 2) גורם האיזון בבן הימני של השורש הוא -1	1) גורם האיזון בשורש תת העץ הוא 2 2) גורם האיזון בבן השמאלי של השורש הוא -1	1) גורם האיזון בשורש תת העץ הוא -2 2) גורם האיזון בבן הימני של השורש הוא 1
הרוטציות המתאימות	1) רוטציית R על השורש	1) רוטציית L על השורש	1) רוטציית L על הבן השמאלי של השורש 2) רוטציית R על השורש	1) רוטציית R על הבן הימני של השורש 2) רוטציית L על השורש

גרפים

- **גרף** $G=(V,E)$ עם V קבוצת קודקודים ו־E קבוצת צלעות (מכוונות או לא מכוונות). גודל הגרף הוא סכום הקודקודים והצלעות.

- **גרף לא מכוון:** אין לולאות עצמיות. יחס שכנות הוא סימטרי. דרגת הקודקוד הוא מספר השכנים שלו. סכום הדרגות הוא $2|E|$.
- **גרף מכוון:** יתכנו לולאות עצמיות. לקודקוד יש דרגת צלעות נכנסות ודרגת צלעות יוצאות. סכום הדרגות הוא $|E|$.
- **גרף ממושקל:** גרף בו לצלעות יש משקל (גרף לא ממושקל הוא גרף ממושקל עם משקל 1 לכל צלע). אם אין מסלול בין שני קודקודים המשקל הוא ∞.
- **מעגל:** מסלול שמתחיל ונגמר באותו קודקוד עם משקל גדול/שווה ל-1.
- **גרף (לא מכוון) קשיר:** גרף שיש בו מסלול בין כל שני קודקודים.
- **גרף (מכוון) קשיר חלקית:** גרף שבו יש מסלול חד כיווני בין כל שני קודקודים.
- **גרף (מכוון) קשיר היטב:** גרף שיש בו מסלול דו כיווני בין כל שני קודקודים.
- **רכיב קשירות:** תת הקבוצה הגדולה ביותר שממנה יש מסלול בין כל שני קודקודים בגרף.
- **קליקה:** גרף שבו כל קודקוד מחובר לכל קודקוד.
- **מפרק:** קודקוד הוא מפרק אם כאשר מסירים אותו ואת הצלעות המחוברות אליו מקבלים גרף לא קשיר.
- **גשר:** צלע היא גשר אם כאשר מסירים אותה מקבלים גרף לא קשיר.
- **גרף פלנארי:** מספר הצלעות הוא $O(|V|)$ ואף צלע לא חוצה צלע אחרת.
- **עץ:** גרף קשיר בלי מעגלים, מכיל $|V|-1$ צלעות.
- **ייצוג גרפים:** אפשר לייצג גרף באמצעות רשימת שכנים (רשימה מקושרת של קודקודים, כשלכל קודקוד מקושרים השכנים שלו) או באמצעות מטריצת שכנים (מטריצה שמכילה 0 אם אין צלע ו-1 אם יש צלע).

חיפוש לרוחב ולעומק

- **חיפוש לרוחב BFS (בגרף לא ממושקל):**
  - <u>מטרה:</u> ממפה את המרחק המינימלי של כל קודקוד בגרף מהשורש שהוזן לו.
  - <u>איך האלגוריתם עובד:</u> נאתחל את כל הקודקודים עם מרחק אינסוף, סימון "לא ביקרנו" והורה $null$. ניצור תור ונוסיף אליו את השורש. בכל פעם נוציא מהתור איבר, נבקר בכל השכנים שלו שלא ביקרנו בהם עדיין, ונוסיף אותם לתור.
  - האלגוריתם (סימון טיפונת שונה ממה שאנחנו מכירים - צביעה באפור היא "נוכחי", בלבן היא "לא ביקרנו" ובשחור "ביקרנו"):

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

- תכונות:
  * האלגוריתם בונה את תת הגרף המוביל ל־t בתור עץ קודמים.
  * מוצא את המסלול הקצר ביותר מ־s לכל קודקוד בגרף.
  * סיבוכיות הזמן היא $O(|V|+|E|)=O(|V|^2)$.

- **חיפוש לעומק DFS (בגרף לא ממושקל):**
  - מטרה: להוציא מידע מעניין מגרף נתון ־ לטובת מיון טופולוגי, איתור רכיבי קשירות או זיהוי עץ פורש מינימלי.
  - איך האלגוריתם עובד:
    * נתחיל מהשורש, נמשיך לעומק העץ עד שנגיע לעלה, נעשה $Backtracking$ לקודקוד הקודם ונמשיך לעומק לכיוון אחד השכנים שלו שעדיין לא ביקרנו בהם.
    * בפועל ־ נאתחל את כל הקודקודים עם תוית "לא ביקרנו" והורה $null$, נאתחל זמן 0, ונבקר בכל קודקוד באמצעות $Visit$.
    * האלגוריתם $Visit$ מאתחל את השדות d ($discovered$) ־ מתי ביקרנו בקודקוד לראשונה) ו־f ($finished$ ־ מתי יצאנו מהקודקוד). לאחר מכן הוא עובר על כל אחד מהשכנים שלא ביקרנו בהם עדיין, ומבקר בהם (ובשכנים שלהם, ובשכנים של השכנים שלהם..) באמצעות קריאה רקורסיבית ל־$Visit$ עם השכן.
  - תוצר: האלגוריתם מחזיר יער עצי עומק.

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1              // white vertex u has just been discovered
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]        // explore edge (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // blacken u; it is finished
time = time + 1
u.f = time
```

  - סיבוכיות: $O(|V|+|E|)$.
  - **משפט הסוגריים:** ניתן להציג את יער עצי העומק שיצר ה־DFS באמצעות סוגריים, כאשר בכל פעם שמופיע סוגר פותח ((מיוצג ע"י השדה d) לאחר סוגר סוגר ((מיוצג ע"י השדה f), המשמעות היא התחלה של רכיב קשירות נפרד. התנאים של משפט הסוגריים אומרים שב־DFS של גרף לכל שני קודקודים u,v יתקיים כי $[u.d,u.f],[v.d,v.f]$ זרים (u,v ברכיבי קשירות שונים) / מוכלים זה בזה (u צאצא של v / v צאצא של u).
    * מסקנה: v צאצא של u אם"ם $u.d<v.d<v.d<u.f$
  - **משפט Visited Path Theorem:** ביער DFS, v הוא צאצא של u אם"ם ניתן להגיע ל־v דרך קודקודי not_visited (כלומר אם יש מסלול של חיפוש לעומק מ־u ל־v).
  - **סוגי צלעות ביער** DFS: אחרי שיצרנו עץ DFS, אפשר לסווג את הצלעות בגרף המקורי ל־4 סוגים:
    * $Tree$ ־ צלעות רגילות בעץ חיפוש לעומק
    * $Back$ ־ צלעות שמחברות u לאב קדמון (לא ישיר) v.

* $Forward$ - צלעות שמחברות קודקוד לילד שלו, אך לא לוקחות חלק בעץ ה-DFS.
* $Cross$ - כל שאר הצלעות (שמקשרות קודקודים שאין ביניהם יחסים של צאצא ואב / קשתות של מעגל עצמי).

– **משפטים על סוגי צלעות:**
  * ב-DFS של גרף לא מכוון, כל צלע ב-E היא מסוג $Tree$ או $Back$.
  * בגרף לא מכוון יש מעגל אם"ם הוא מכיל צלע מסוג $Back$.

– אפשר להשתמש במשפט הקודם כדי ליצור **אלגוריתם שבודק אם קיימים מעגלים** בגרף <u>בסיבוכיות $O(|V|)$</u>: עבור צלע (u,v), אם בזמן הסריקה של u כבר ביקרנו בצלע v, משמע שהצלע היא צלע אחורה, כלומר קיים מעגל. האלגוריתם יעבוד באופן דומה ל-DFS (יכנס רקורסיבית לקודקודים עד להגעה לעלה). בביקור בכל קודקוד נבדוק האם אנחנו מזהים צלע אחורה (כלומר נבדוק אם כבר ביקרנו באחד השכנים שלו שאינו אבא שלו).

Algorithm 6 Has-Cycles(G)

```
for v ∈ V do
    v.visited ← false
    π(v) ← null
for v ∈ V do
    if v.visited = false then
        if Explore-Cycles(G, v) then
            return true
return false
```

Algorithm 7 Explore-Cycles(G, v)

```
v.visited ← true
for (v, u) ∈ E do
    if π(v) ≠ u then
        if u.visited = true then
            return true
        else
            π(u) ← v
            return Explore-Cycles(G, u)
return false
```

* **שימוש ב-DFS למיון טופולוגי של גרף מכוון ללא מעגלים:** סידור לינארי של הקודקודים ברשימה ע"פ סדר, כך שאם G מכיל את הצלע (u,v), אז הקודקוד u מופיע לפני v ברשימה. הוכחנו שהוא יוצר גרף מכוון ללא מעגלים.

– <u>איך האלגוריתם עובד:</u> מתחילים מלקרוא ל-DFS כדי לחשב את זמני היציאה מכל קודקוד. בכל פעם שיוצאים מקודקוד, מוסיפים את האיבר לראש של רשימה מקושרת, ומחזירים אותה בסיום.

TOPOLOGICAL-SORT(G)

```
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

רכיבי קשירות

* **שימוש ב-DFS למציאת רכיבי קשירות בגרף מכוון:**

– **רכיבים קשירים היטב SCC:** תתי הגרפים השונים הגדולים ביותר כך שבכל אחד מהם, כל שני קודקודים מחוברים זה לזה בשני הכיוונים.

– **גרף משוחלף Transposed Graph:** הגרף G^T המכונה הגרף המשוחלף של G, הוא למעשה זהה ל-G למעט זה שהכיוונים בו הפוכים.
  * הפעלת שחלוף פעמיים תחזיר אותנו לגרף המקורי.

– <u>איך האלגוריתם עובד:</u> נקרא ל-DFS, נחשב את הגרף המשוחלף, ונפעיל את DFS על הגרף המשוחלף מהסוף להתחלה (כאשר הסדר מוגדר לפי השדה f שחישבנו ב-DFS).

STRONGLY-CONNECTED-COMPONENTS(G)

```
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

* **גרף רכיבי קשירות חזקה (של גרף מכוון):** כל קודקוד בגרף מייצג רכיב קשירות חזקה בגרף המקורי, וכל צלע בגרף קיימת אם"ם קיימת צלע המקשרת בין רכיבי הקשירות שהקודקודים מייצגים בגרף המקורי.

– עבור גרף G, גרף רכיבי הקשירות החזקה של G^T יהיה שחלוף של גרף רכיבי הקשירות החזקה של G.

- **אלגוריתם לבדיקה האם גרף מכוון הוא קשיר חלקית** בסיבוכיות $O(|V|+|E|)$:

– טענה שנתבסס עליה: גרף מכוון הוא קשיר חלקית אם"ם לכל סידור טופולוגי של רכיבי הקשירות החזקה בגרף רכיבי הקשירות החזקה, נקבל שיש צלע בין כל שני קודקודים עוקבים בסידור.

– איך האלגוריתם עובד: נריץ את האלגוריתם למציאת רכיבי קשירות חזקה, ונבנה את גרף רכיבי הקשירות החזקה. נסדר את הקודקודים בגרף רכיבי הקשירות החזקה בסיגור טופולוגי. עבור כל קודקוד, נבדוק האם קיימת צלע בינו לבין הקודקוד הבא בסידור הטופולוגי - אם לא, נסיק שהגרף קשיר חלקית.

- **אלגוריתם למציאת כל המפרקים בגרף:**

– הפתרון הנאיבי (סיבוכיות $O(|V|\cdot(|V|+|E|))$) יהיה להסיר בכל פעם קודקוד אחד ולבדוק אם הגרף עדיין קשיר (להריץ DFS ולבדוק כמה עצים יש ביער העומק), אבל זה מאוד לא יעיל.

– הפתרון היעיל (סיבוכיות לינארית $O(|V|+|E|)$):

* נתבסס על הטענה שקודקוד u הוא מפרק אם הוא שורש ביער ה-DFS ויש לו לפחות שני בנים / הוא אינו שורש ביער ה-DFS אבל יש לו בן שמקיים שכל קודקוד בתת העץ שתחתיו לא מכיל צלע אחורה לאב קדמון של u.

* איך האלגוריתם עובד: בצורה דומה מאוד לאלגוריתם ה-$Explore/Visit$ שאנחנו מכירים. עבור כל קודקוד, נשמור את הזמן בו נכנסנו אליו d והזמן בו יצאנו ממנו f, וכן את הערך $earliest$ שמכיל את הקודקוד הקדום ביותר ביער עץ העומק שלקודקוד הנוכחי יש קשר אליו. נעבור על השכנים - אם זה שכן קדום יותר (כלומר שכבר ביקרנו בו), נעדכן את $earliest$ אם צריך. אם זה שכן שעוד לא ביקרנו בו, נעדכן את $earliest$ להיות הקודקוד הכי קדום שהשכן יכול להגיע אליו (כי אז גם הקודקוד הנוכחי יכול להגיע אליו טרנזיטיבית) במקרה הצורך. כל זה בדק את התנאי השני של הטענה, נבדוק גם את התנאי הראשון (כלומר אם הוא שורש עם 2 ילדים) ע"י בדיקה אם d של הקודקוד הוא 1 וגם יש לו 2 ילדים.

Algorithm 9 Explore(G, v)

```
v.visited ← true
Pre-Visit(v)
children(v) ← 0
earliest(v) ← pre(v)      \\ הקודקוד הקדום ביותר שישיג מ v הוא v
for (v, u) ∈ E s.t. u.visited = true do      \\ הקשת (v, u) היא קשת אחורה
    earliest(v) ← min{earliest(v), pre(u)}      \\ נשמור את הקודקוד הקדום ביותר שישיג מ v דרך קשת אחורה
for (v, u) ∈ E s.t. u.visited = false do      \\ הקשת (v, u) היא קשת עץ
    children(v) ← children(v) + 1      \\ u הוא בן של v ביער ה DFS
    Explore(G, u)      \\ נמצא את הקודקוד הקדום ביותר שישיג מ u
    earliest(v) ← min{earliest(v), earliest(u)}      \\ נעדכן את הקודקוד הקדום ביותר שישיג מ v
    if 1 < pre(v) ≤ earliest(u) then      \\ הקודקוד הקדום ביותר שישיג מ u אינו אב קדמון של v
        add v to the list of articulations
if pre(v) = 1 & children(v) > 1 then      \\ v הוא שורש עץ DFS ויש לו לפחות שני בנים
    add v to the list of articulations
Post-Visit(v)
```

- **אלגוריתם למציאת כל הגשרים בגרף:**

– פתרון נאיבי סיבוכיות $O(|V|\cdot(|V|+|E|))$ יהיה להסיר בכל פעם צלע אחת ולבדוק אם הגרף עדיין קשיר (להריץ DFS ולבדוק כמה עצים יש ביער העומק), אבל זה מאוד לא יעיל.

– הפתרון היעיל (סיבוכיות לינארית $O(|V|+|E|)$):

* נשתמש בטענה שצלע היא גשר אם היא לא חלק ממעגל פשוט (מעגל שהוא מסלול פשוט בין הקודקוד הראשון לקודקוד האחרון).

* איך האלגוריתם עובד: נריץ את האלגוריתם למציאת המפרקים (מבלי לשמור את המפרקים). לאחר מכן, נעבור על כל צלע. אם זמן הכניסה d של הקודקוד הראשון בצלע קדום יותר מערך ה-$earliest$ של הקודקוד השני בצלע, אזי הצלע היא גשר.

מציאת עץ פורש מינימלי MST

- **עץ פורש מינימלי:** עבור גרף קשיר, ממושקל ולא מכוון, עץ פורש מינימלי הוא עץ (קשיר בלי מעגלים) שמורכב מתת קבוצה של הצלעות כך שכל הקודקודים מחוברים, והסכום של משקלי הצלעות הוא הקטן ביותר. יתכנו כמה עצים פורשים מינימליים שונים לאותו גרף.

– **אלגוריתם חמדן:** אלגוריתם שבוחר את האפשרות הטובה ביותר בהנתן המצב הנוכחי, מבלי להתחשב בהשפעות עתידיות.

– **צלע בטוחה:** צלע שלא יוצרת מעגל, ומשקלה מינימלי (אין צלע קלה ממנה).

- **אלגוריתם גנרי למציאת עץ פורש מינימלי:** יוצר עץ פורש בגישה חמדנית - בכל פעם נוסיף צלע בטוחה.

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

- **אלגוריתמים אחרים למציאת עץ פורש מינימלי:** נחפש עץ פורש מינימלי בדרך הבאה:
  - נסמן צלע ראשונה.
  - נבחר אחת מהאפשרויות:
    - נבחר צלע מינימלית שמחברת בין U ל־$V\backslash U$ שלא יוצרת מעגל (אלגוריתם קרוסקל)
    - נבחר צלע מינימלית ב־$V\backslash U$ שחוצה את החתך (אלגוריתם פרים)
  - נעצור כאשר $U=V$.
- **חתכים:**
  - <u>חתך:</u> חתך $C=(V,V\backslash U)$ של גרף לא מכוון הוא חלוקה של כל הקודקודים בגרף לשתי קבוצות זרות.
  - <u>צלע חוצה:</u> צלע (v,u) חוצה את החתך C אם"ם $u\in U$ ו־$v\in V\backslash U$.
  - <u>צלע קלה:</u> הצלע בעלת המשקל המינימלי מבין הצלעות שחוצות את החתך.
  - <u>חתך מכבד קבוצת צלעות:</u> חתך C מכבד קבוצת צלעות A אם"ם אין צלע ב־A שחוצה את C.
  - <u>שני משפטים שמשמשים להוכחה:</u>
    - אם צלע שחוצה חתך היא צלע קלה, אז היא חלק מעץ פורש מינימלי כלשהו.
    - אם קבוצת צלעות A שהיא חלק מעץ פורש מינימלי, ו־C חתך שמכבד את A, אז הצלע המינימלית בחתך היא צלע בטוחה ל־A.
  - **למת החתך:** יהיו G גרף ממושקל, C חתך, c הצלע הקלה בחתך, T עץ פורש מינימלי של G, X תת גרף שמכבד את החתך ומוכל ב־T. אז קיים עץ פורש מינימלי T' המכיל את $X\cup\{c\}$.
- **אלגוריתם קרוסקל (מציאת עץ פורש מינימלי בעץ ממושקל):**
  - <u>איך האלגוריתם עובד:</u> נמיין את הצלעות לפי משקל בסדר עולה, ונעבור עליהן לפי הסדר. נבדוק אם הצלע הנוכחית סוגרת מעגל (נבדוק באמצעות $Union-Find$ - אם לקודקודים שהצלע מחברת יש את אותו נציג, סימן שהצלע סוגרת מעגל). אם היא לא סוגרת מעגל, נוסיף אותה לעץ הפורש המינימלי.

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

  - <u>סיבוכיות:</u> $O(|E|log|E|)$.
  - <u>מימוש שונה:</u> נמיין את הצלעות לפי סדר <u>יורד</u>, ונעבור עליהן לפי הסדר. אם צלע סוגרת מעגל, <u>נמחק</u> אותה מ־G. נחזיר את G. הסיבוכיות של הפתרון הזה היא $O(|E|^2)$.
  - <u>טענות שהוכחנו:</u>
    - תהי c הצלע המקסימלית במעגל C בגרף G, אז c לא מוכלת בשום עץ פורש מינימלי של G.
    - תהי c שלא מוכלת בשום מעגל ב־C בגרף G, שהסרה שלה מ־G הופכת אותו ללא קשיר. אז c מוכלת ב<u>כל</u> עץ פורש מינימלי של G.
    - תהי c צלע שהיתה מוכלת במעגל ב־G, שהסרה שלה מ־G הופכת אותו ללא קשיר. אז c מוכלת ב<u>כל</u> עץ פורש מינימלי של G.

– **אלגוריתם פרים (מציאת עץ פורש מינימלי בעץ ממושקל):**

∗ איך האלגוריתם עובד: מזכיר קצת את BFS. נשתמש בערמת מינימום שתגדיר חתך בגרף (כלומר הקבוצות הזרות יהיו הקודקודים שבערמה מול הקודקודים שמחוץ לערמה) ותאותחל עם כל הקודקודים בגרף. שורש ערמת המינימום יהיה הקודקוד שמחובר לצלע המינימלית בחתך. בכל שלב נעביר קודקוד לצד השני של החתך (כלומר נוציא קודקוד מהערמה), נעבור על כל השכנים שלו, ולכל שכן נחליט אם להוסיף לעץ את הצלע ביניהם (נוסיף אותה רק אם משקל הצלע בינו לבין השכן קטנה מהמפתח של השכן) כאשר ההוספה לעץ מתבצעת ע"י עדכון ערך ה־π של השכן להיות הקודקוד הנוכחי. נעשה זאת עד שכל הקודקודים יהיו באותה קבוצה בחתך (עד שהערמה תהיה ריקה).

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

∗ סיבוכיות: $O(|V|log|E|)$.

מציאת מסלולים קצרים ביותר בגרף ממושקל

- השפעת צלעות בעלות משקל שלילי: במקרה של צלע עם משקל שלילי, נחבר את המשקל שלה כרגיל כשנסכום את המשקל של המסלול (למרות שהיא מורידה מהמשקל של המסלול ולא מעלה אותו). עם זאת, נפסול גרפים עם מעגלים שליליים, והמסלול הקצר ביותר מוגדר כל עוד אין מעגלים שליליים.

- תכונות של מסלולים קצרים ביותר:

  – לא מכילים מעגלים חיוביים.

  – כל תת מסלול של מסלול קצר ביותר, הוא גם קצר ביותר.

- **מציאת כל המסלולים הקצרים ביותר מקודקוד בודד בגרף מכוון:**

  – **התאמה של אלגוריתם BFS לגרף ממושקל:**

∗ נזכר ש־BFS מוצא את כל המסלולים הקצרים ביותר בגרף לא ממושקל, נבצע התאמה כדי שנוכל להשתמש בו בגרף ממושקל. ההתאמה נדרשת כיוון שלא בטוח שהפעם הראשונה בה נגיע לקודקוד תהיה בדרך ה"קלה" (מבחינת משקל) ביותר להגיע אליו (כיוון ש־BFS יתחיל מהדרך הקצרה ביותר מבחינת כמות הצלעות, שהיא לא בהכרח הכי קלה מבחינת משקל).

∗ האלגוריתם החדש ישתמש ב־BFS, רק שבהגעה לכל קודקוד נבצע את פעולת $Relax$, הבודקת עם ערך d הנוכחי של קודקוד היעד גדול מהמשקל של הקודקוד הנוכחי + משקל הצלע בינו לבין קודקוד היעד, ואם כן ־ מעדכנת את ערך d של קודקוד היעד לערך הנמוך יותר.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

  – תכונות של אלגוריתם מסלולים קצרים שמתבסס על Relax:

∗ חסם עליון: לכל קודקוד, ערך ה־d שלו בכל רגע נתון יהיה גדול או שווה לערך המינימלי שהוא יכול לקבל (מהרגע שיתרחש שוויון, הוא ישמר).

∗ העדר מסלול: אם אין מסלול בין השורש לקודקוד v כלשהו, ערך ה־d של v ישאר תמיד אינסוף.

∗ מונוטוניות יורדת: מתחילת ריצת האלגוריתם ועד סופו, ערך ה־d של כל קודקוד הולך ויורד.

* סיום התהליך: בסיום ריצת האלגוריתם, ערך ה־d של כל קודקוד יהיה המינימלי.
* אי שוויון המשולש: לכל צלע (u,v) מתקיים $\delta(s,v) \leq \delta(s,u) + \delta(u,v)$.
* הפעלת Relax על מסלול: אם $P_1k =< v_1, v_2, ..., v_k >$ המסלול הקצר ביותר בין v_1 ל־v_k, אז מתקיים $v_k.d = \delta(s, v_k)$.
* הקודם של תת גרף: אם ערך ה־d של קודקוד הוא המינימלי האפשרי, אז תת עץ הקודמים שלו הוא עץ BFS של מסלולים קצרים ביותר ששורשו s.

- **אלגוריתם דייקסטרה Djikstra (מחייב גרף ללא מעגלים שליליים):**
  * איך האלגוריתם עובד: ניצור אלגוריתם דומה ל־BFS עם שלוש התאמות:
    · נעדכן את d לא לפי כמות הצלעות המובילות אליו, אלא לפי המשקלים שלהן.
    · כדי לבחור שכנים, נשתמש בערמת מינימום במקום בתור (מה שיגרום לכך ש"גל ההתפשטות" יהיה לפי הקודקוד ה"קרוב ביותר לפי המשקל).
    · בכל ביקור בקודקוד, נעשה Relax על כל השכנים שלו (כלומר לא נדלג על שכנים שכבר ביקרנו בהם).

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

  * סיבוכיות: הסיבוכיות של דייקסטרה היא $O(|E|log|V|)$.

- **אלגוריתם בלמן פורד Bellmann Ford (אפשר להפעיל על גרף עם מעגלים שליליים):**
  * הרעיון: אם הוספת צלע נוספת תקטין את המשקל של המסלול, נזהה כי מצאנו מעגל שלילי.
  * איך האלגוריתם עובד: נעבור $|V|-1$ פעמים על כל הצלעות בגרף. בכל איטרציה, נעבור על כל צלע ונעשה על הקודקודים שהיא מגיעה אליו Relax. אם נזהה שהגענו לקודקוד שכבר ביקרנו בו בדרך קלה ממש מהדרך בה הגענו אליו קודם, סימן שמצאנו מעגל שלילי ונעצור את פעולת האלגוריתם. אם לא נזהה מעגל שלילי, נמשיך עד שנעבור על כל הצלעות $|V|-1$ פעמים.

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

• **מציאת מסלולים מכל קודקוד לכל קודקוד (בגרף ללא מעגלים שליליים):**
  - **פתרון נאיבי 1:** להפעיל את אלגוריתם בלמן פורד $|V|$ פעמים - סיבוכיות $O(|V|^4)$.
  - **פתרון נאיבי 2:** להפעיל את אלגוריתם דייקסטרה $|V|$ פעמים - סיבוכיות $O(|V|^3 \cdot log|V|)$.
  - **פתרון לא יעיל 1:** שימוש באלגוריתם $All - Shortest - Paths$ - סיבוכיות $O(|V|^4)$.
  - **פתרון לא יעיל 2:** שימוש באלגוריתם $Faster - All - Shortest - Paths$ - סיבוכיות $O(|V|^3 \cdot log|V|)$.
  - **פתרון יעיל:** שימוש באלגוריתם פלויד־ורשל ב־$O(|V|^3)$.
  - **ייצוג מטריציוני של מסלולים ממושקלים:**

מטריצת משקלי צלעות W	מטריצת מסלולים קצרים ביותר L	מטריצת קודמים Π
התא (i,j) הוא משקל הצלע (i,j)	התא ה־(i,j) הוא אורך המסלול הקצר בין i ל־j	התא ה־(i,j) מכיל את ההורה של j במסלול הקצר ביותר מ־i.
$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$	$\begin{pmatrix} null & 3 & 4 & 5 & 1 \\ 4 & null & 4 & 2 & 1 \\ 4 & 3 & null & 2 & 1 \\ 4 & 3 & 4 & null & 1 \\ 4 & 3 & 4 & 5 & null \end{pmatrix}$

– **חילוץ המסלולים הקצרים ביותר ממטריצת Π:** האלגוריתם הבא ידפיס את כל המסלולים בשורה ה־i (ניתן להמיר אותו ללהיות אלגוריתם של <u>כל</u> המסלולים בטבלה ע"י ריצה על כל השורות).

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i == j
    print i
elseif π_ij == NIL
    print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
    print j
```

– **פתרון למציאת כל המסלולים הקצרים ביותר:** תהי W מטריצת משקלים ו־L מטריצת אורכי המרחקים הקצרים ביותר שחושבו עד כה. נחשב סדרת מטריצות $\left(L^{(1)}, L^{(2)}, ..., L^{(n-1)}\right)$ עבור $m \in [1, n-1]$, כאשר $L^{(m)}$ היא מטריצה שבה כל אורכי המסלולים הקצרים ביותר בעלי לכל היותר m צלעות.

* <u>אלגוריתם ליצירה של $L^{(m)}$ ע"ב $L^{(m-1)}$:</u> עוברים על כל תא (בלולאה כפולה על השורות i והעמודות j), מאתחלים את התא המקביל במטריצה החדשה לאינסוף, ואז עובדים בלולאה מ־1 עד n ומחליפים את הערך של התא במטריצה החדשה ל־$L_k^i + W_j^k$ אם הוא קטן מהערך הנוכחי שלה.סיבוכיות הריצה היא $O(|V|^3)$.

```
EXTEND-SHORTEST-PATHS(L, W)
n = L.rows
let L' = (l'_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        l'_ij = ∞
        for k = 1 to n
            l'_ij = min(l'_ij, l_ik + w_kj)
return L'
```

* <u>פתרון איטי לאלגוריתם למציאת כל המסלולים הקצרים:</u> ניצור את המטריצה $L^{(m)}$ כל הדרך מהמטריצה W. סיבוכיות ריצה $O(|V|^4)$, כלומר לא טוב יותר מהפתרונות הנאיביים שהצענו.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
for m = 2 to n − 1
    let L^(m) be a new n × n matrix
    L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
return L^(n−1)
```

* <u>פתרון מהיר יותר למציאת כל המסלולים הקצרים:</u> נחליף את לולאת ה־for באורך $|V|$ בלולאת $while$ שתתבצע $O(log|V|)$. הסיבוכיות שלו היא $O(|V|^3 \cdot log|V|)$.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
m = 1
while m < n − 1
    let L^(2m) be a new n × n matrix
    L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
    m = 2m
return L^(m)
```

– **אלגוריתם פלויד ורשל Floyd-Warshall:** בשונה מהאלגוריתמים הקודמים שמתבססים על חישוב כל המטריצות $L^{(i)}$ על מנת ליצור את המטריצה $L^{(m)}$, אלגוריתם פלויד ורשל מתבסס על פרמטר אחר - שהוא רשימת קודקודי הביניים שלוקחים חלק במסלול קצר ביותר כלשהו. כלומר, נתחיל בלא לאפשר קודקודי ביניים (כלומר לאפשר קשר ישיר בלבד - זו למעשה בדיוק המטריצה W), לאחר מכן נאפשר להשתמש בקודקוד הראשון כקודקוד ביניים, לאחר מכן בקודקוד הראשון והשני וכך הלאה עד שנאפשר להשתמש בכל הקודקודים בגרף.

* הסיבוכיות היא $O(|V|^3)$ - הכי טוב שהצלחנו להשיג לבעיה.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

* **שימושים אחרים באלגוריתם פלויד ורשל (חוץ ממציאת מסלולים קצרים):**

· <u>מציאת גרף קודמים:</u> ניצור מטריצה Π (אחרי שורה 2) שתתעדכן לאורך ריצת האלגוריתם, כאשר בכל שלב באלגוריתם, התא ה-$\Pi(i,j)$ בקודם של הקודקוד j במסלול הקצר ביותר מ-i ל-j שמצאנו עד כה. בפועל, בכל איטרציה k של הלולאה בשורה 3 ניצור גם מטריצת קודמים $\Pi^{(k)}$ חדשה ריקה. פעם שבו אנחנו מעדכנים את המסלול הקצר ביותר בטבלה D (לאחר שורה 7), נעדכן גם את הקודם בטבלה $\Pi^{(k)}$ (אם המסלול מ-i ל-j לא התקצר כתוצאה מאפשור המעבר דרך קודקוד הביניים k, נשאיר את הקודם להיות אותו דבר כמו באיטרציה הקודמת. אם הוא כן התקצר כתוצאה מהוספת k, נעדכן אותו להיות הקודם של j במסלול הקצר ביותר הנוכחי).

כלומר נוסיף לאחר שורה 2:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases}$$

ולאחר שורה 7:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

· <u>מציאת מעגלים שליליים בגרף:</u> נזהה מעגל שלילי ע"י זיהוי מסלול מקודקוד לעצמו (האלכסון במטריצת המסלולים הקצרים) שהוא בעל ערל שלילי.

· <u>מציאת סגור טרנזיטיבי:</u>

סגור טרנזיטיבי G^* של גרף מכוון G הוא גרף שמכיל מסלול מ-i ל-j אם"ם קיים ביניהם מסלול (גם אם לא ישיר) ב-G. כלומר, זה G שהוסיפו אליו גם מסלולים עקיפים בין קודקודים. אנחנו לא מתעניינים במשקלים אלא רק באם קיים מסלול או לא, לכן המטריצה של G^* היא בינארית.

נשתמש באלגוריתם שדומה לפלויד ורשל - רק שהפעם ניצור מטריצה בינארית בהתחלה, ואת הבדיקה אם להחליף את הערך הקיים במטריצה או לא, נעשה לא באמצעות השוואת המשקלים כמו קודם, אלא רק באמצעות בדיקה פשוטה של and ו-or: נחליף 0 ב-1 אם קיים מסלול מ-i ל-j עם $k-1$ הקודקודים הראשונים כקודקודי ביניים **או** ש(קיים מסלול בין i ל-k שכולל את $k-1$ הקודקודים הראשונים כקודקודי ביניים **וגם** קיים מסלול בין k ל-j שכולל את $k-1$ הקודקודים הראשונים כקודקודי ביניים). הסיבוכיות היא $O(|V|^3)$.

```
TRANSITIVE-CLOSURE(G)
n = |G.V|
let T^(0) = (t_ij^(0)) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        if i == j or (i, j) ∈ G.E
            t_ij^(0) = 1
        else t_ij^(0) = 0
for k = 1 to n
    let T^(k) = (t_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

· ניתן לחשב סגור טרנזיטיבי בזמן טוב יותר - $O(|V| \cdot (|V| + |E|))$, ע"י לאתחל מטריצת אפסים בגודל $|V| \times |V|$, לעבור על כל קודקוד ולהריץ החל ממנו BFS, ולעדכן את השורה שלו במטריצה בהתאם לתוצאות (כלומר להפוך ל-1 כל עמודה שמייצגת קודקוד שנגיע אליו באלגוריתם).

מבני נתונים לקבוצות זרות

- **המטרה:** ליצור מבנה נתונים שתומך בפעולות הבאות על קבוצות זרות בצורה יעילה:
  - יצירת קבוצה Make-Set(x): יצירה של קבוצה חדשה S_X שמכילה איבר יחיד x. סיבוכיות $O(1)$.
  - מציאת נציג Find-Set(x): מציאת הנציג של הקבוצה ש-x שייך אליה. סיבוכיות תלויה במימוש.
  - איחוד קבוצות $\text{Union}(x_i, x_j)$: החלפת שתי קבוצות זרות S_i (עם הנציג x_i) ו-S_j (עם הנציג x_j) בקבוצה חדשה שמהווה האיחוד שלהן (ללא כפילויות). סיבוכיות תלויה במימוש.
- **אלגוריתמים שמשתמשים בפעולות Union-Find:**
  - **מציאת רכיבי קשירות:** מקבל גרף לא ממושקל ולא מכוון G, ומחזיר את קובצת רכיבי הקשירות של G. ניצור קבוצה מכל קודקוד עם Make-Set, ואז לכל צלע, אם הקודקודים שלה שייכים לשתי קבוצות שונות (נבדוק לפי הנציג), נאחד ביניהן.

```
CONNECTED-COMPONENTS(G)
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

  - **אלגוריתם קרוסקל:** כמו שלמדנו, מקבל גרף לא מכוון וממושקל G, ומחזיר עץ פורש מינימלי של G. לכל קודקוד הוא יוצר קבוצה משלו, ואז לכל צלע, אם הקודקודים שלה שייכים לשתי קבוצות שונות (נבדוק לפי הנציג), נוסיף אותה לעץ ונאחד בין הקבוצות של הקודקודים שלה.
  - **רשתות חלחול:** רשת חלחול היא מטריצה בגודל $n \times n$ שבה כל משבצת היא "חסומה" (שחורה) או "פתוחה" (לבנה). נאמר שנשבצת פתוחה היא "מלאה" אם קיים מסלול של משבצות פתוחות בינה לבין משבצת פתוחה בשורה העליונה. נאמר שהרשת מחלחלת אם קיימת משבצת מלאה בשורה התחתונה של המטריצה.
    * אלגוריתם למציאת משבצת מלאה: ניצור קבוצה מכל תא בטבלה. נעבור בלולאה חיצונית על השורות ובלולאה פנימית על העמודות. לכל משבצת, אם היא פתוחה: אם המשבצת מימנה פתוחה ו/או המשבצת מתחתיה פתוחה, נאחד את הקבוצות שלהן. נחזיר אמת אם קיים j (עמודה) כך שהנציג של $[k, l]$ ושל $[1, j]$ הוא אותו נציג.
    * אלגוריתם לקביעה אם הרשת מחלחלת: נשתמש באלגוריתם למציאת משבצת מלאה, נחזיר אמת אם יש $k \le n$ וגם $1 \le j$ כך הנציג של $[n, k]$ ושל $[1, j]$ הוא אותו נציג.
- **דרכים לממש את מבנה הנתונים:**
  - **רשימות מקושרות:** כל קבוצה תיוצג באמצעות רשימה מקושרת.
    * שדות של הרשימה: שדה $Head$ המצביע לראש הרשימה (שהוא הנציג), ושדה $Tail$ המצביע לסוף הרשימה.
    * שדות של האיברים ברשימה: ערך $Value$, מצביע לאיבר הבא $Next$, מצביע לראש הרשימה (הנציג) $Head$.

* סיבוכיות של הפעולות:
  - יצירת קבוצה Make-Set(x): יצירת רשימה חדשה - סיבוכיות $O(1)$.
  - מציאת נציג Find-Set(x): מעבר על הרשימה בחיפוש אחר x - במקרה הגרוע $O(n)$.
  - איחוד קבוצות $\text{Union}(x_i, x_j)$: חיבור הזנב של קבוצה אחת לראש הקבוצה השניה (תוך כדי עדכון שדה $Head$ בכל אחד מהאיברים של הקבוצה הראשונה) - במקרה הגרוע $O(n)$.
  - במימוש זה, יצירה של n קבוצות ואז ביצוע $n-1$ איחודים שלהן לקבוצה אחת, יקח $O(n^2)$.
* **יוריסטיקת האיחוד הממושקל:** כאשר נאחד שתי רשימות, נוסיף את הרשימה הקצרה לארוכה (וכך נחסוך בכמות שינויי מצביע ה-$Head$ שנצטרך לבצע). פעולת איחוד אחת עדיין יכולה לקחת $\Omega(n)$ אם שתי הקבוצות מכילות אותו סדר גודל של איברים. שימוש ביוריסטיקה הזו מוריד את הסיבוכיות מ-$O(n^2)$ ל-$O(m + nlogn)$.

– **יערות של קבוצות זרות:** כל קבוצה היא עץ (הנציג הוא השורש של העץ). כל איבר מצביע לאב שלו בעץ, השורש מצביע לעצמו.

* סיבוכיות של הפעולות:
  - יצירת קבוצה Make-Set(x): יצירת השורש - סיבוכיות $O(1)$.
  - מציאת נציג Find-Set(x): מעבר על האיברים בעץ בחיפוש אחר x - במקרה הגרוע $O(h)$.
  - איחוד קבוצות $\text{Union}(x_i, x_j)$: נמצא את השורש של כל אחת מהקבוצות ב-$O(h)$, ואז נבחר איזה שורש לחבר לשני - במקרה הגרוע $O(h)$.
* **יוריסטיקות לאיחוד העצים:**
  - איחוד לפי דרגה: נצרף את העץ הנמוך לגבוה, כך שלאחר האיחוד גובה העץ לא השתנה / גדל ב-1. זה יגרום לכך שכשנסיים לאחד את כל הקבוצות, גובה העץ יהיה $O(logn)$, לכן הסיבוכיות הכוללת היא $O(mlogn)$.
  - דחיסת מסלולים: כשנרצה לאחד עצים, נחבר את השורש של העץ השני ישירות לשורש של העץ הראשון. נבצע דחיסה של המסלול - נעבור מהקודקוד התחתון עד לקודקוד השורש, ועבור כל קודקוד בדרך נעביר אותו (ואת תת העץ שלו) להיות ישירות תחת השורש. הסיבוכיות הכוללת היא $O(m + mlogn)$.

• **שימוש אידאלי:** בשימוש משולב באיחוד לפי דרגה וגם בדחיסת מסלולים, סדרה של $m > n$ פעולות (ש-n מתוכן הן יצירת קבוצות) תקח $O(m\alpha(n))$ (כאשר α היא פונקציה הופכית לפונקצית אקרמן - לא התעמקנו).

– בשימוש האידאלי נקבל סיבוכיות:
  * יצירת קבוצה Make-Set(x): $O(1)$.
  * מציאת נציג Find-Set(x): $O(\alpha(n))$.
  * איחוד קבוצות $\text{Union}(x_i, x_j)$: $O(\alpha(n))$.

– לכן בשימוש אידאלי נוכל **למצוא רכיבי קשירות** ב-$O(|V| + |E|\alpha(|V|))$ **ולהפעיל את אלגוריתם קרוסקל** ב-$O(|V| + |E|log(|V|))$.