סיכום קורס
מבני נתונים 67109
Data Structures

סמסטר א' 2021

מרצה: ד"ר לאו יוסקוביץ'

מסכמת: קרן בן אריה

תוכן עניינים

מקרא:
דוגמה - 🔵
הוכחה - 🔴
טענה - 🟠
משפט - 🟢
הגדרה - 🟣

חישוביות, יעילות וזמני ריצה

טבלאות גיבוב

ערימות

עצי חיפוש בינאריים ועצים מאוזנים

גרפים

הרצאה 1 - Data Structures

תכניות ואלגוריתמים

* תכנית היא מימוש של אלגוריתם
* הקורס יגע יותר באלגוריתמים ובשיטות אבסטרקטיות ולא מימוש ספציפי.

מבני נתונים

* מבנה נתונים מתאר איזשהי שיטה לשמור נתונים שעליהם הולכים לבצע עיבוד.
  ← מערכים, משתנים, רשימות מקושרות, עצים בינאריים וכו'.
* מבנה הנתונים שנבחר בו ישפיע מאוד על פתרון הבעיה והמימוש.

Abstract Data Types - ADT

* אופיון מתמטי של אובייקטים שומרים את הנתונים (ע"י מבני נתונים) ואילו פעולות בסיסיות נבצע עליהם, כמה משאבים (זמן, אחסון) דורש כל אחד מהם.
* Data Structure הוא מימוש של ADT.
* בקורס נעסוק יותר בחלק האובסטרקטי, ה-ADT.

נושאי הקורס

* נלמד אלגוריתמים ו- ADTs לפתרון בעיות בסיסיות ונפוצות במדעי המחשב.
* נלמד לנתח ולהשוות אלגוריתמים בהיבטים של שטח וסיבוכיות זמן ריצה.
* משם נלמד לפתור בעיות יותר מורכבות ולפתח אלגוריתמים בעצמנו.

אלגוריתמים ופתרון בעיות

נניח שיש לנו איזשהי בעיה חישובית.

* האם ניתן למצוא אלגוריתם שיהווה עבורה פתרון?

← לא תמיד. לדוג': בעיית העצירה (The halting problem).

* האם נוכל למצוא אלגוריתם יעיל על מנת לפתור אותה?

← לא תמיד. לדוג': בעיית האריזה (The Packing problem).

* האם האלגוריתם שלי הוא המיטבי?

← לא תמיד. לדוג': מיון ב-$O(n^2)$.

* איך נדע מה האלגוריתם הכי טוב שניתן לפתח?

← ישנם מיונים בזמן ריצה של $\Omega(n\log n)$ ואחסון של $\Omega(n)$.

* בקורס הזה נשתמש במודל חישובי שנקרא החישוב הסיקוונציאלי (Sequentially), כלומר, הפעולות בו קורות בזו אחר זו, ולא בצורה מקבילית.

בעיות קלות ובעיות קשות — Easy Problems & Hard Problems

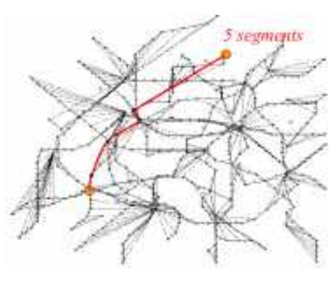
* מיון הוא בעיה קלה, לעומת אריזת קופסאות שמסווגת כבעיה יותר קשה.
* נלמד לסווג ביניהם וגם להוכיח שבעיה היא קשה או שייכת למחלקה של בעיות קשות.
* מחלקות שקילות מבחינת חישוביות:

← לוגריתמי - $\Omega(n\log n)$; ליניארי - $\Omega(n)$

← פולינומיאלי - $\Omega(n^k)$; ריבועי - $\Omega(n^2)$

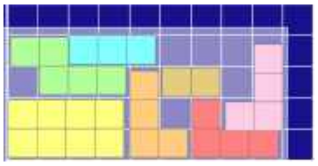
← כפול אקספ' - $\Omega(2^{2^n})$; אקספוננציאלי - $\Omega(2^n)$

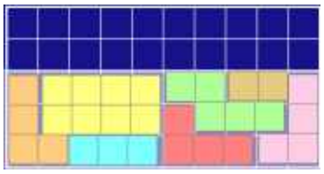
← לא ניתנות לפתרון.



<u>דוג' לבעיה קלה:</u> מציאת הדרך הקצרה ביותר בין שתי תחנות מטרו בפריז. בהמשך נראה שלמצוא מסלול כזה יקח $O(n)$.

דוג' לבעיה קשה: אריזה של קופסאות לתבנית/כלשהי תבנית עם מינימום של מקומות ריקים שנשארו בתבנית





* פתרון אופטימלי - אלגוריתם: 1) חישוב כל האפשרויות
2) רישום של כמה שטח הוא תופס
3) בחירה במיטבי

איזה סוג של יעילות?

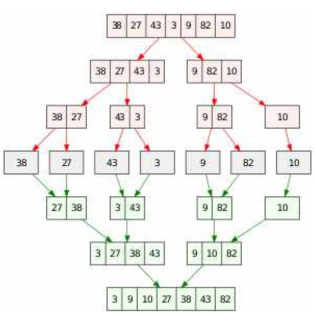
בהינתן אלגוריתם A כלשהו, נוכל לשאול את השאלות הבאות ע"מ להבין את סיבוכיות זמן הריצה והאחסון שלו:

* מה יקרה במקרה הטוב ביותר? (הקלט הכי פשוט)
* מה יקרה במקרה הרע ביותר? (הקלט הכי מסובך)
* מה יקרה במקרה הממוצע?
* גבול תחתון ועליון: מהם החסמים ההדוקים ביותר שנצליח להשיג?
* שקלול בין זמן לשטח אחסון.

יעילות של מיונים

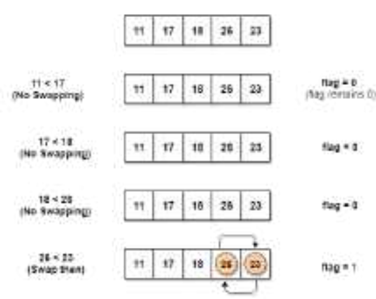
בהינתן מערך של n מספרים שלמים A[i], נרצה למיין אותם בסדר עולה ולשם כך נבחן שתי אפשרויות:

* BubbleSort - השוואה בין שני מספרים סמוכים, החלפה אם A[i-1] < A[i]. n חזרות.
* MergeSort - פיצול רקורסיבי לתתי מערכים, מיון של תתי-מערכים ואז מיזוג חזרה.



MergeSort

BubbleSort



השוואה:

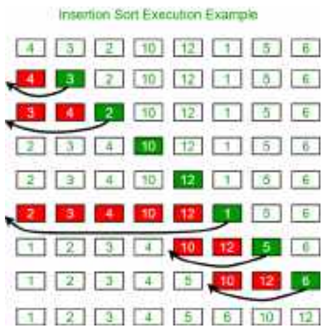
	Space	Time – Best	Time – Worst	Time – Average
BubbleSort	n	מעבר אחד n	n מעברים n^2	$\frac{n^2}{2}$ $\frac{n^2}{2}$ מעברים
MergeSort	$n\log n$	$n\log n$	$n\log n$	$n\log n$

Data Structures - 2 הרצאה

Elements of complexity analysis

דוגמה לניתוח תכנית:

* Insertion Sort



עלות	כמות פעמים	שלבים
c_1	n	1) לולאת for שרצה N-2 לאורך המערך (j)
c_2	$n-1$	2) השמה $key = A[j]$
c_3	$n-1$	3) השמה $i = j-1$
c_4	$\sum_{j=2}^{n} t_j$	4) לולאת while: $(i>0)$ וגם $(A[i] > key)$
c_5	$\sum_{j=2}^{n} (t_j-1)$	5) השמה $A[i+1] = A[i]$
c_6	$\sum_{j=2}^{n} (t_j-1)$	6) השמה $i = i-1$
c_7	$n-1$	7) השמה $A[i+1] = key$

שיטת ניתוח תכנית:

* זמן הריצה הכולל יהיה הסכום של כל זמני הריצה בכל שורה שמבוצעת

* כל שלב בתכנית לוקח c_i זמן (קבוע) ומבוצע t_i פעמים ומכאן זמן הריצה יחושב כך:

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

(כאשר T פונ' של זמן)

ולדוגמה שלנו, Insertion Sort נקבל שהניתוח הוא:

$$T(n) = c_1 \cdot n + c_2(n-1) + c_3(n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

שלושת הביטויים ב- ישתנו לפי המקרה.

המקרה הטוב ביותר: המערך ממוין אז $t_j=1$ לכל $2\le j\le n$.

שלב (4 מבוצע n פעמים ושלבים (5, (6 (בתוך ה-while) לא מבוצעים.

לכן ייראה כך: $T(n)=(c_1+c_2+c_3+c_4+c_7)n-(c_2+c_3+c_4+c_7)$

כל הפעולות שלוקח להם n פעמים   אלו שהתבצעו פעם אחת

מדובר בפונקציה לינארית: $T(n)=a\cdot n+b$

המקרה הגרוע ביותר: המערך ממוין בדיוק הפוך, אז $t_j=j$ לכל $2\le j\le n$.

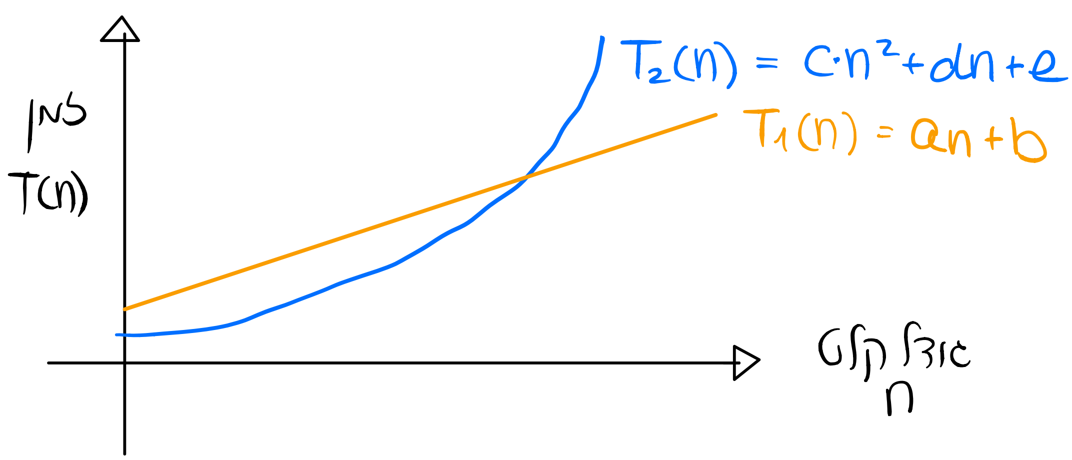
שלב (4 מבוצע $\frac{n(n+1)}{2}-1$ פעמים ושלבים (5, (6 יבוצעו $\frac{n(n-1)}{2}$ פעמים.

לכן בסופו של דבר זה ייראה כך:

$$T(n)=(c_4+c_5+c_6)\frac{n^2}{2}+\left(c_1+c_2+c_3+c_7+\frac{c_4}{2}-\frac{c_5}{2}-\frac{c_6}{2}\right)n-(c_2+c_3+c_4+c_7)$$

מדובר בפונקציה ריבועית: $T(n)=an^2+bn+c$

* כעת ניתן להשוות את שתי הפונקציות באמצעות ניתוח אסימפטוטי.
הפונ' מסומנות כ-$T(n)$, והיא פונקציה של גודל הקלט n. $T(n)=f(n)$.
* הקבועים זניחים בניתוח זה (כי במקרה אסימפטוטי, הם לא גורמים להבדל שינוי).
* כרגע ש-n קטן, כל התכניות כנראה יעבדו בצורה סבירה, אבל אנו רוצים לבדוק מה קורה כש-n גדל מאוד.



עבור n מספיק גדול, $T_2(n)$ ייתן זמן פחות טוב.

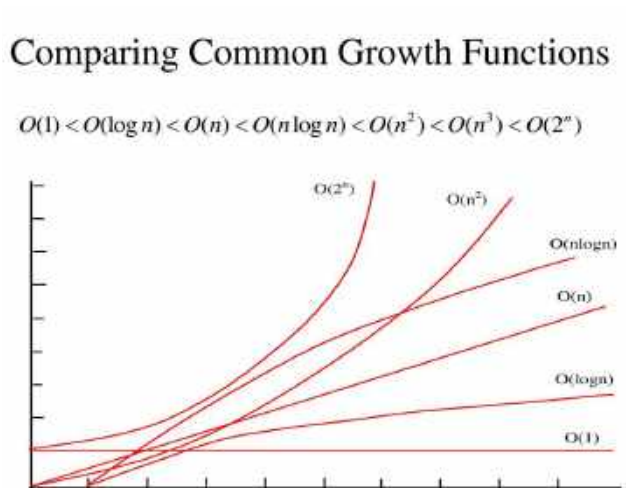
* לסיכום, יעילות האלגוריתם נמדדת עפ"י ההתנהגות האסימפטוטית כפונקציה של גודל הקלט n, כאשר n מספיק גדול.

סיבוכיות זמן ומקום — Time and Space Complexity

<u>סיבוכיות זמן - T(n)</u>: כמות הפעולות הבסיסיות שצריכים בשביל לחשב את הסיבוכיות כפונ' של גודל הקלט.

<u>סיבוכיות מקום - S(n)</u>: כמות התפוס בזיכרון שצריכים בשביל ביצוע פעולה.

<u>קצב הגדילה כפונקציה של n</u>: כמה מהר גדלות T(n) ו-S(n) כפונקציות של n.



דוגמאות לקצבי גדילה של פונקציות נפוצות ←

פונקציות אסימפטוטיות — Asymptotic Functions

נרצה להגדיר ולמצוא פונקציות כאלו על פי אלגוריתם הנכתב בשפה אובסטרקטית, ונרצה להבין מהי ההתנהגות האסימפטוטית עבור n מספיק גדול תוך התעלמות מהקבועים.

<u>חסם עליון</u>: צריך לכל היותר f(n) פעולות. סימון: $O(f(n))$

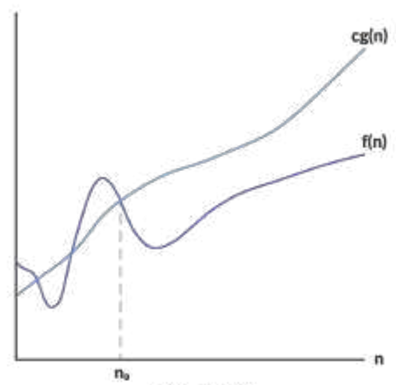
חסם תחתון: צריך לכל הפחות $f(n)$ פעולות. סימון: $\Omega(f(n))$

חסם הדוק: צריך סדר גודל של $f(n)$ פעולות, משני הכיוונים. סימון: $\Theta(f(n))$

חסם עליון - O גדול

הגדרה: יהיו $f(n)$ ו- $g(n)$ שתי פונקציות כך: $g,f:\mathbb{N}\to\mathbb{R}^+$.

$$O(g(n))=\{f(n)\mid \exists c>0 \wedge \exists n_0\geq 1 \ s.t\ \forall n\geq n_0\ f(n)\leq c(g(n))\}$$



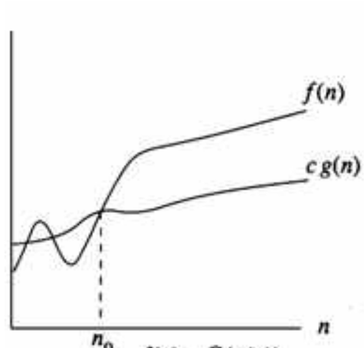
$f(n)=O(g(n))$

ניתן לראות שקיים $c>0$ ו- $n_0\geq 1$ שעבור
כל $n\geq n_0$ הפונקציה של $g(n)$ תמיד מעל
הפונקציה $f(n)$. g חוסמת את f מלמעלה.

חסם תחתון - Ω גדול

הגדרה: יהיו $f(n)$ ו- $g(n)$ שתי פונקציות כך: $g,f:\mathbb{N}\to\mathbb{R}^+$.

$$\Omega(g(n))=\{f(n)\mid \exists c>0 \wedge \exists n_0\geq 1 \ s.t\ \forall n\geq n_0\ f(n)\geq c(g(n))\}$$



$f(n)=\Omega(g(n))$

ניתן לראות שקיים $c>0$ ו- $n_0\geq 1$ שעבור
כל $n\geq n_0$ הפונקציה של $f(n)$ תמיד מעל
הפונקציה $c\cdot g(n)$. g חוסמת את f מלמטה.

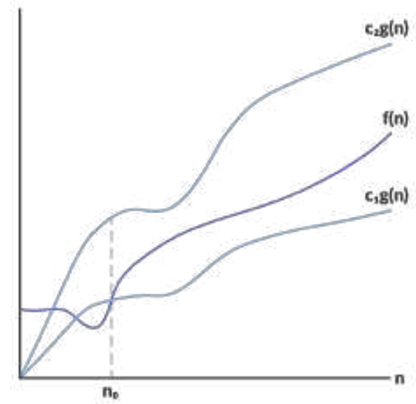
חסם הדוק - Θ גדול

שילוב של עליון ותחתון

הגדרה: יהיו $f(n)$ ו- $g(n)$ שתי פונקציות כך: $g,f:\mathbb{N}\to\mathbb{R}^+$.

$$\Theta(g(n))=\{f(n)\mid \exists c_1,c_2>0 \wedge \exists n_0\geq 1 \ s.t\ \forall n\geq n_0\ c_1\cdot g(n)\leq f(n)\leq c_2\cdot(g(n))\}$$

חסם הדוק = חסם מלמעלה ומלמטה



f(n) = Θ(g(n))

ניתן לראות שקיימים $c_1,c_2>0$ ו-$n_0>1$ שעבור כל $n \ge n_0$ הפונקציה של $f(n)$ תמיד בין הפונקציות $c_1g(n), c_2\cdot g(n)$.

* נניח שיש n_1 ו- n_2 מתאימים ל- c_1,c_2, נבחר $n_0=\max(n_1,n_2)$.

מאפיינים של פונקציות ה- O,Ω,Θ (ללא הוכחות)

1. אם יש לי O (חסם עליון) וגם Ω (חסם תחתון) אז יש לי Θ (חסם הדוק).
2. מתקיימת רפלקסיביות: $f(n)=O(f(n))$.
3. מתקיימת סימטריה: $f(n)=\Theta(g(n)) \Longleftrightarrow g(n)=\Theta(f(n))$.
4. מתקיימת טרנזיטיביות: $f(n)=O(g(n)) \land g(n)=O(h(n)) \Longrightarrow f(n)=O(h(n))$
5. מתקיים: $O(O(f(n)))=O(f(n))$ (קורה בגלל הורדה של הקבועים).
6. מתקיימת אדיטיביות: $O(f(n)+g(n))=O(f(n))+O(g(n))$
7. מתקיימת הומוגניות: $O(f(n)\cdot g(n))=O(f(n))\cdot O(g(n))$
8. מתקיים: $O(\log n)=O(\lg n)$, כל log, לא חשוב מה הבסיס, מתנהג כזה.
9. בפולינום, ה- O שלו יקבע ע"י n^k כאשר k הוא החזקה הגבוהה ביותר.
כלומר: $O(\sum_{i=1}^{k} a_i n^i)=O(n^k)$
10. בעצרת, ישנה פונקציית קירוב: $n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$

הערות:
- הוכחת חסם תחתון קשה יותר מהוכחת חסם עליון, והוכחת החסם ההדוק היא הקשה ביותר.
- נעסוק בהמשך הקורס במציאת חסמים עבור קבוצות אלגוריתמים.

הרצאה 3 - Data Structures

ניתוח סיבוכיות של אלגוריתם

ישנן שתי דרכים מרכזיות:

1) ספירה ישירה - $T(n) = \sum c_i t_i$ — מתאים ללולאות

2) נוסחאות רקורסיביות — מתאים לפונ' רקורסיביות ומבנים

שני שלבים:
① אפיון המשוואה
② פתרונה

דוגמה:

$$\begin{cases} T(n) = T(n-1)+c \\ T(1) \end{cases}$$

נוסחת הרקורסיה של מציאת מקסימום במערך

הפתרון למשוואה פה הוא $T(n) = O(n)$.

נוסחאות רקורסיביות

הנחות מקלות:

* n מספיק גדול, על מנת שנוכל להבין התנהגות אסימפטוטית.
* עבור n קטן מספיק, $T(1) = \Theta(1)$
* בחירת n בהתאם לתנאים, לדוג': 1) n זוגי $(n = 2k)$ 2) n כחזקה של 2 $(n = 2^k)$ כאשר $k \in \mathbb{N}_+$

למשל, כאשר מחלקים מבנים בחצי כל פעם

יכול לפשט את המקרה האסימפטוטי

אזהרה:

עד עכשיו אמרנו שלרוב הקבועים אינם חשובים, אבל ישנם מקרים בהם הם חשובים - לכן צריך לשים לב ולהבין מתי ניתן להשמיט ומתי לא.

לפעמים כשהקבוע הוא בתוך ה-T, כקלט של הפונ', ישנו הבדל גדול, לדוגמה: $T(\frac{n}{2})$ שונה מ-$T(n)$.

אפיון נוסחאות רקורסיביות

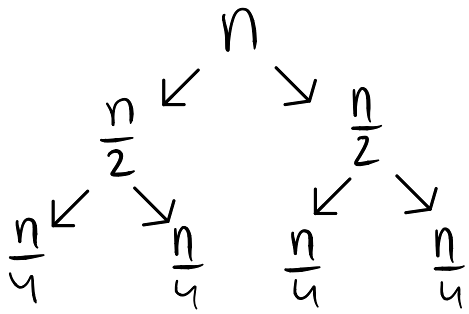
חשוב לשים לב:

1) ניתן לקחת בעיה ולחלק לתתי בעיות. Divide and Conquer

דוג': מיון מערך בגודל n, ב-mergeSort ניתן לחלק לתתי מערכים.

2) בתוך החלוקה לתתי-בעיות - לכמה נחלק כל פעם? כלומר, מה יהיה גודלה של כל תת-בעיה. דוג': ב-mergeSort נחלק ל-2 או 3 תתי מערכים כל פעם?

3) שילוב תתי-הפתרונות לקבלת פתרון שלם. דוג': ב-mergeSort משלבים מערכים ממוינים.



עץ רקורסיה:

חשוב לנו לדעת:

* לכמה חלקים חילקנו
* מהו עומק העץ

נוסחאות רקורסיביות נפוצות

1) Factorial - הכפלה של n ב-$(n-1)!$

הנוסחה: $T(n) = T(n-1) + O(1)$ הפתרון: $O(n)$

2) Fibonacci - הסכום של fibonacci(n-1) ו-fibonacci(n-2).

הנוסחה: $T(n) = T(n-1) + T(n-2)$ הפתרון: $O(2^n)$

3) Sequential Search - בודקים אם האיבר הראשון הוא זה שמחפשים, ואם לא - קוראים רקורסיבית עם אותו מבנה נתונים, עם איבר אחד פחות.

הנוסחה: $T(n) = T(n-1) + O(1)$ הפתרון: $O(n)$

4) **Insertion Sort** - מוצאים את המיקום של האיבר הראשון ברשימה הממוינת, ואז קוראים רקורסיבית עם אותו מבנה נתונים, עם איבר אחד פחות.

הנוסחה: $T(n) = T(n-1) + O(n)$ הכנסת האיבר למקום הנכון

הפתרון: $O(n^2)$

5) **Binary Search** - בודקים אם שורש העץ הוא מה שאנחנו מחפשים, אם לא - נקרא רקורסיבית עם הבן הימני או השמאלי של העץ, עם חצי מהאיברים.

הנוסחה: $T(n) = T(\frac{n}{2}) + O(1)$

הפתרון: $O(\log n)$

6) **Binary Tree Traversal** - עוברים על כל הצמתים בעץ ע"י קריאות רקורסיביות של צמתי העץ הימני והשמאלי.

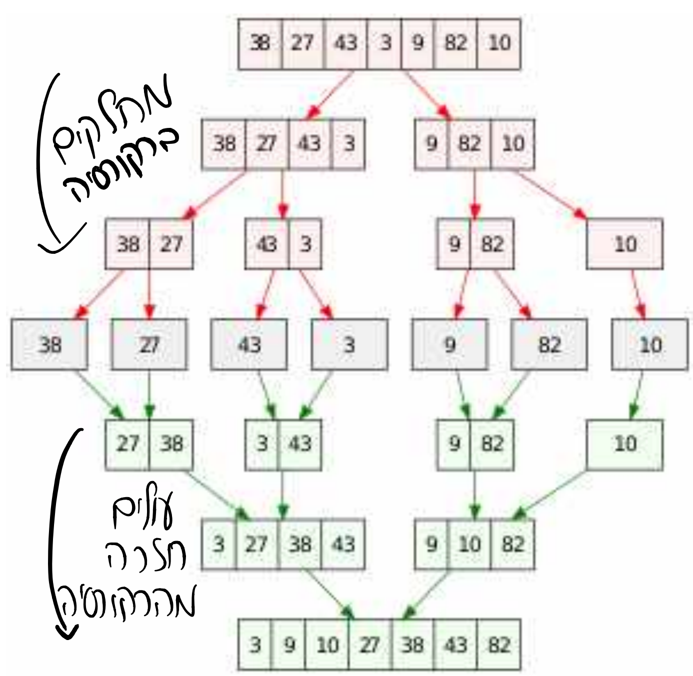
הנוסחה: $T(n) = 2T(\frac{n}{2}) + O(1)$

הפתרון: $O(n)$

7) **Merge Sort** - מחלקים את הרשימה לחצי וכך הלאה עד שמגיעים לסוף הרקורסיה, אח"כ חוזרים חזרה עד שמרכיבים את הרשימה הממוינת.

הנוסחה: $T(n) = 2T(\frac{n}{2}) + O(n)$

הפתרון: $O(n\log n)$



פתרון משוואות רקורסיביות

1. שיטת ההצבה — Substitution

ניחוש החסם הנכון והוכחתו באינדוקציה מתמטית.

דוגמה: עבור $T(n)=2T(\frac{n}{2})+n$, הפתרון הוא $O(n\log n)$ עבור $n\geq 2$.

הוכחה: נרצה להוכיח שקיים $c>0$ שלכל $n\geq 2$ מתקיים: $T(n)\leq c\cdot(n\log n)$

נוכיח באינדוקציה:

בסיס: $n=2$ $\quad T(2)\leq c\cdot(2\log_2 2)$

עבור $c\geq 1$, $T(2)=2$ ✓

הנחה: נניח שהנוסחה נכונה עבור $\frac{n}{2}$. כלומר: $T(\frac{n}{2})\leq 2\cdot(c\cdot\frac{n}{2}\cdot\log(\frac{n}{2}))$ (הצבנו מהנוסחה)

צעד: נוכיח שהנוסחה נכונה עבור n.

מהנוסחה ומההנחה: $T(\frac{n}{2})$

$$T(n)\leq 2(c\cdot\frac{n}{2}\cdot\log(\frac{n}{2}))+n = cn\cdot\log(\frac{n}{2})+n = cn\cdot\log n - cn\cdot\log 2+n$$

$$= cn\log n - cn + n \underset{\forall c\geq 1}{\leq} cn\log n$$

* לא תמיד יתנו לנו את החסם, או שנצליח לנחש ולכן כדאי לדעת למצוא תבניות.

מציאת תבניות:

1. לרשום מספר אלמנטים של הרקורסיה
2. לחפש תבנית
3. להוכיח אותה ע"י הצבה (אפשר באינדוקציה).

דוגמה: נגיד שנרצה למצוא את החסם של $T(n)=T(n-1)+n$ עבור n.

1. (כתוב כמה ביטויים עבור שלבים שונים)

נפתח עבור $n-1$ בצורה רקורסיבית: $T(n-1)=T(n-2)+(n-1)$

נפתח עבור $n-2$ בצורה רקורסיבית: $T(n-2)=T(n-3)+(n-2)$

נפתח עבור $n-3$ בצורה רקורסיבית: $T(n-3)=T(n-4)+(n-3)$

כעת נבצע הצבה:

$T(n) = T(n-1) + n = [T(n-2) + (n-1)] + n =$

$[[T(n-3) + (n-2)] + (n-1)] + n = [[[T(n-4) + (n-3)] + (n-2)] + (n-1)] + n$

נשים לב שמדובר בסכום:

$$= T(n-k) + \sum_{i=1}^{k} (n-i+1) = T(n-k) + nk - \frac{((k-1)k)}{2}$$

2. ננסח את הביטוי הכללי

כלומר, קיבלנו תבנית של הטווח $n-k$:

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

בסוף הרקורסיה $k = n-1$ ו- $T(1) = 1$ אז נקבל:

3. נזהה את התנאים של סוף הרקורסיה

$$T(n) = 1 + n^2 - n + \frac{n^2}{2} - \frac{3n}{2} - 1 = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

בסופו של דבר, הוכחנו שהפתרון של $T(n) = T(\frac{n}{2}) + n$ הוא $O(n^2)$.

2. שיטת עץ הרקורסיה Recursion-Tree

המרה של הנוסחה הרקורסיבית לעץ של צמתים שכל אחד מייצג את העלות של כל שלב, תוך שימוש בסכימה של הפעולות בכל שלב, על מנת להגיע לפתרון.

3. שיטת משפט האב Master Method

משפט האב (גרסה קלה)

יהיו $a \geq 1$, $b \geq 1$, $c \geq 1$ קבועים, תהי $f(n)$ פונקציה, ו- $T(n)$ תהיה פונ' המוגדרת לכל $n \in \mathbb{N}$ כך:

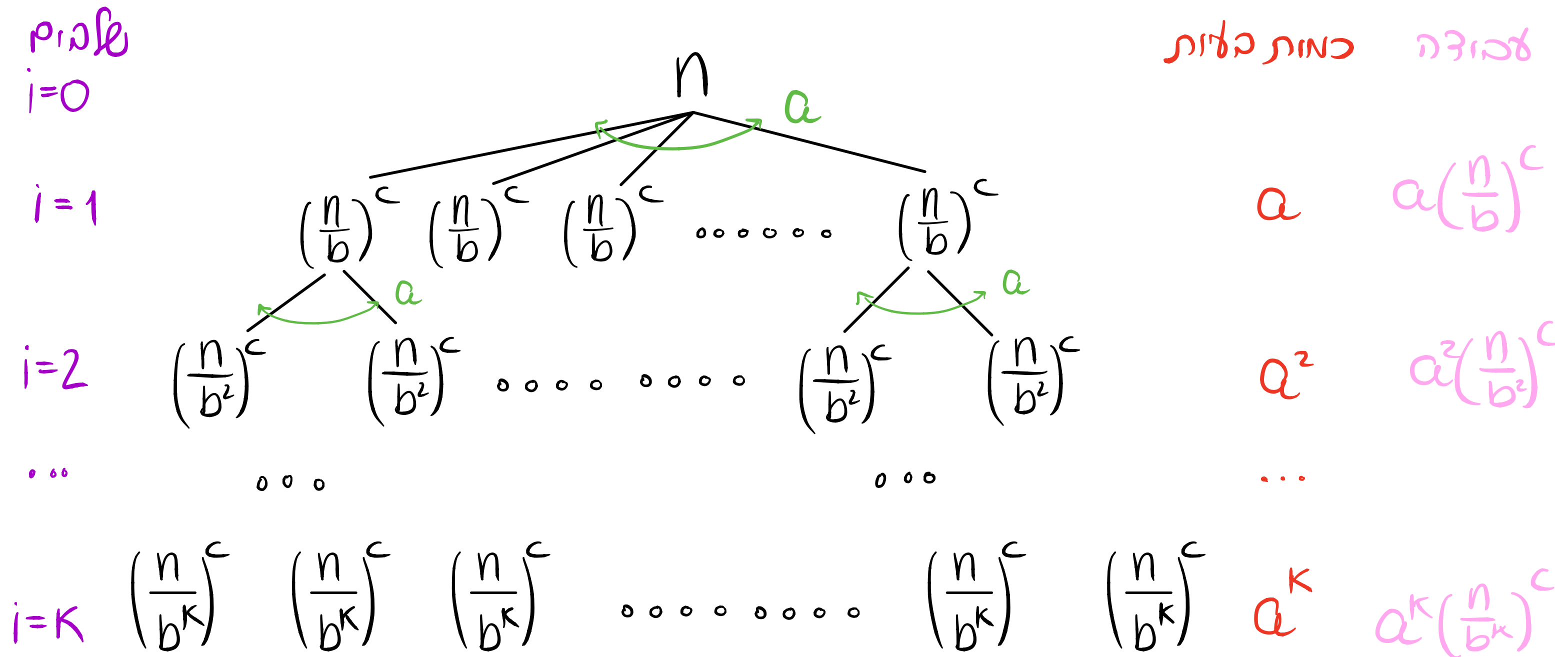
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^c \quad (c \geq 0)$$

אז מתקיים:

1. כאשר $1 > \frac{a}{b^c}$, אז $\log_b a < c$ — $T(n) = \Theta(n^c)$
2. כאשר $1 = \frac{a}{b^c}$, אז $\log_b a = c$ — $T(n) = \Theta(n^c \cdot \log_b n)$
3. כאשר $1 < \frac{a}{b^c}$, אז $\log_b a > c$ — $T(n) = \Theta(n^{\log_b a})$

אינטואיציה: ישנה בעיה, חילקתי אותה ל-a בעיות, כאשר גודל כל אחת הוא $\frac{n}{b}$,
ואני הולכת לעשות $\left(\frac{n}{b}\right)^c$ פעולות.

עץ הרקורסיה:



נקבל שעומק הרקורסיה יהיה $K=\log_b n$.

ננסה להבין את הנוסחה, לכן נפתח באופן רקורסיבי:

$$T(n) = a\cdot T\left(\frac{n}{b}\right)+n^c$$

$$T\left(\frac{n}{b}\right) = a\cdot T\left(\frac{n}{b^2}\right)+\left(\frac{n}{b}\right)^c$$

$$T\left(\frac{n}{b^2}\right) = a\cdot T\left(\frac{n}{b^3}\right)+\left(\frac{n}{b^2}\right)^c$$

$$T\left(\frac{n}{b^4}\right) = a\cdot T\left(\frac{n}{b^5}\right)+\left(\frac{n}{b^4}\right)^c$$

עכשיו נציב:

$$T(n) = a\cdot T\left(\frac{n}{b}\right) + n^c = a\left[a\cdot T\left(\frac{n}{b^2}\right) + \left(\frac{n}{b}\right)^c\right] + n^c =$$

$$= a\left[a\left[a\cdot T\left(\frac{n}{b^3}\right) + \left(\frac{n}{b^2}\right)^c\right] + \left(\frac{n}{b}\right)^c\right] + n^c =$$

$$= a^k T\left(\frac{n}{b^k}\right) + n^c\left[1 + a\left(\frac{1}{b}\right)^c + a^2\left(\frac{1}{b^2}\right)^c + \dots + a^{k-1}\left(\frac{1}{b^{k-1}}\right)^c\right] =$$

$$= \boxed{a^k T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^c}$$

מצאנו שעומק הרקורסיה הוא $k = \log_b n$.

<u>מספר העלים:</u>

* ברמה ה-k ישנם $a^k \cdot \left(\frac{n}{b^k}\right)^c$ תתי-בעיות.
* ישנם $k = \log_b n$ שלבים.
* נציב:

$$a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^c \qquad b^{\log_b n} \rightarrow n$$

כעת נרצה להוכיח ש: $a^{\log_b n} = n^{\log_b a}$

<u>הוכחה:</u> נקח את $\log_b$ ונפעיל על שני האגפים:

$$\log_b a^{\log_b n} \qquad\qquad \log_b n^{\log_b a}$$

$$\Downarrow \qquad\qquad \Downarrow$$

$$\log_b n \cdot \log_b a = \log_b a \cdot \log_b n$$

נחזור לנוסחת הרקורסיה:

$$T(n) = a^k T\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i \left(\frac{n}{b^i}\right)^c$$

ונציב 3:

$k = \log_b n \,,\; a^{\log_b n} = n^{\log_b a}$

$$T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b(n-1)} a^i \left(\frac{n}{b^i}\right)^c =$$

$$T(n) = \Theta(n^{\log_b a}) + n^c \sum_{i=0}^{\log_b(n-1)} a^i \left(\frac{a}{b^c}\right)^i$$

נשאר סכום שתלוי ב- $\frac{a}{b^c}$.

נחלק ל-3:

$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ עבור $|x|<1$

1. כאשר $\frac{a}{b^c} < 1$:

$$1 = \left(\frac{a}{b^c}\right)^0 < \sum_{i=0}^{\log_b(n-1)} a^i \left(\frac{a}{b^c}\right)^i = \frac{1-\left(\frac{a}{b^c}\right)^{\log_b n}}{1-\left(\frac{a}{b^c}\right)} < \frac{1}{1-\left(\frac{a}{b^c}\right)} < const.$$

ולכן: $T(n) = \Theta(n^c)$.

2. כאשר $\frac{a}{b^c} = 1$:

$$\sum_{i=0}^{\log_b(n-1)} a^i \left(\frac{a}{b^c}\right)^i = \log_b n$$

ולכן: $T(n) = \Theta(n^c \log_b n)$.

3. כאשר $\frac{a}{b^c} > 1$:

$$\sum_{i=0}^{\log_b(n-1)} a^i \left(\frac{a}{b^c}\right)^i = \Theta\left(\left(\frac{a}{b^c}\right)^{\log_b n}\right)$$

$$n^c \Theta\left(\left(\frac{a}{b^c}\right)^{\log_b n}\right) = \Theta\left(n^c \left(\frac{a}{b^c}\right)^{\log_b n}\right)$$

$$n^c\left(\frac{a}{b^c}\right)^{\log_b n} = \frac{n^c}{b^{c\log_b n}}\cdot a^{\log_b n} = \frac{n^c}{n^c}\cdot n^{\log_b a} = n^{\log_b a}$$

לכן, $T(n)=\Theta(n^{\log_b a})$.

דוגמה: Merge-Sort, נוסחת הרקורסיה היא: $T(n)=2T(\frac{n}{2})+n^1$

הקבועים פה הם: $a=2$, $b=2$, $f(n)=n$ כך ש- $c=1$.

מדובר בדיוק במקרה 2. מכיוון שמתקיים: $\frac{a}{b^c}=\frac{2}{2^1}=1$.

ולכן: $T(n)=\Theta(n^{\log_2 2}\log n)=\Theta(n\log n)$ (ע"י השימוש במשפט).

משפט האב (גרסה כללית)

מדובר הפעם לא רק בפונקציות פולינומיאליות.

יהיו $a\ge1$, $b\ge1$, $c\ge1$ קבועים, $n>1$ ו- $f(n)$ תהיה פונקציה.

תהיה $T(n)$ פונקציה המוגדרת כך: $T(n)=a\cdot T(\frac{n}{b})+f(n)$

כאשר $\frac{n}{b}=\lceil\frac{n}{b}\rceil$ או $\frac{n}{b}=\lfloor\frac{n}{b}\rfloor$.

אינטואיציה למשפט האב:

נשווה בין שני גורמים: $f(n)$ ו- $O(n^{\log_b a})$:

1. כאשר $O(n^{\log_b a})$ דומיננטי יותר, הסיבוכיות היא: $T(n)=\Theta(n^{\log_b a})$
2. כאשר $f(n)$ דומיננטי יותר, אז הסיבוכיות היא: $T(n)=\Theta(f(n))$
3. כאשר הם זהים, אז הסיבוכיות תהיה: $T(n)=\Theta(n^{\log_b a}\log n)=\Theta(f(n)\log n)$

דוגמה 1: פתרו את המשוואה: $T(n)=9T(\frac{n}{3})+n$.

פתרון: $a=9$, $b=3$, $f(n)=n$ ולכן $n^{\log_b a}=n^{\log_3 9}=n^2$.

הרצאה 4 - Data Structures

מיון

הגדרה: <u>קלט:</u> סדרה A באורך n, $A=(a_1,a_2,\dots,a_n)$
<u>פלט:</u> תמורה (פרמוטציה): $(a_1',a_2',\dots,a_n')$ בסדר כך ש:
$$a_1' \le a_2' \le \dots \le a_n'$$

<u>מוטיבציה:</u> * בעיה יסודית במדמ"ח
* הרבה אלגוריתמים משתמשים במיון
* חשיפה לטכניקות שבאמצעותן ננתח אלגוריתמים
* ניתן להוכיח חסם תחתון על כל אלגוריתמי המיון

סוגי אלגוריתמי מיון מרכזיים:

1. השוואות

* מבוסס על השוואה בסיסית בין שני אלמנטים: $a_i \le a_j$
* דוגמאות לאלגוריתמים מסוג זה: Bubble-Sort, Insertion-Sort, Merge-Sort.
בנוסף ישנו Quick-Sort שניתן לנתח ע' נוסחאות נסיגה
* חסמים תחתונים עבור אלגוריתמי מיון מסוג זה:
$S(n)=\Omega(n)$ (סיבוכיות מקום), $T(n)=\Omega(n\log n)$ (סיבוכיות זמן).
* בהמשך הקורס נראה אלגוריתמים נוספים כמו Heap-Sort.

2. ללא השוואות

* מדובר באלגוריתמים שונים כלליים, נהיה חייבים להניח הנחות על האלמנטים שעליהם מתבצע המיון.

הומצא לפני כ-50 שנה

Quick-Sort: Divide and Conquer

אלגוריתם מהסוג שמבצע השוואות.

תכונה מיוחדת: מבצע את המיון במקום.

* מתקבל מערך בגודל n, והמיון מתבצע ללא להעתיק את המערך למערכים שונים (בשונה מ- Merge-Sort). מה שמקטין את סיבוכיות המקום ע"י שימוש במקום קבוע מחוש של שטח. $\Longleftarrow S(n)=\Omega(n)$.

* סיבוכיות זמן ריצה ממוצעת: $O(n\log n)$.

מהלך האלגוריתם

<u>קלט</u>: מערך לא ממוין בגודל n, שאיבריו הם $A[i]$ כך ש- $1\le i\le n$.

<u>פלט</u>: מערך ממוין בגודל n כך ש- $A[i]\le A[i+1]$ לכל $1\le i\le n$.

$n=8$

2	1	3	4	7	6	5	8

<u>רעיון</u>

1. Divide: חלוקה של המערך לשמאל, וימין עם PIVOT (ר׳ 3) בינוהם כך ש:

$$LEFT \le PIVOT < RIGHT$$

2. conquer: מיון צד שמאל וימין בצורה רקורסיבית

3. Combine: אין צורך! בזכות התכונה, לא צריכים לשלב אותם, כי ה-PIVOT במקומו בכל שלב.

ניגש ל- 1.Divide: $A[l,r]$

$A[l,m-1]$ $A[m]$ $A[m+1,r]$

2	1	3	4	7	6	5	8
l			m				r

Quick-Sort (A, l, r)

1. If $l < r$ then do
2. $m \leftarrow$ Partition(A, l, r)
3. Quick-Sort$(A, l, m-1)$
4. Quick-Sort$(A, m+1, r)$

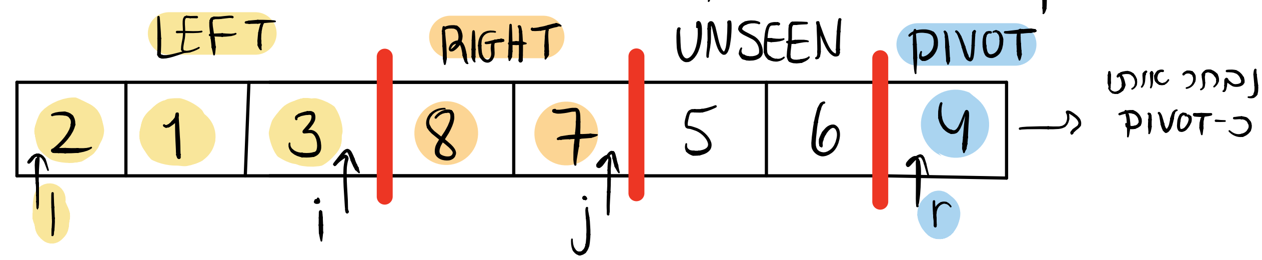
If $l = r$ do nothing

$A[m]$ stays in place

m is called a pivot

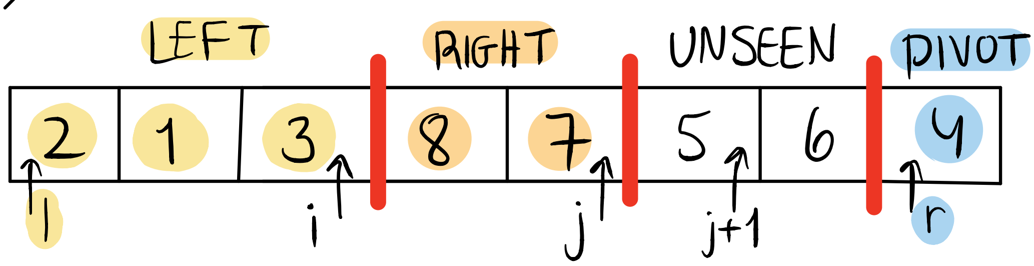
כעת ניתן להבין מה Partition עושה:

Partition: 4 תתי מערכים.



* את ה-PIVOT כרגע יבחר כימין ביותר: $A[r]$.
* כל האלמנטים משמאל יהיו $LEFT \le PIVOT$. כלומר, $A[i] \le A[r]$ כאשר $i \in \{1,2,3\}$.
* כל האלמנטים מימין יהיו $RIGHT > PIVOT$. כלומר $A[j] > A[r]$ כאשר $j \in \{4,5\}$
← i ו-j אינדקסים רצים כאשר i ב-LEFT ו-j ב-RIGHT.
* האלמנטים ב-UNSEEN כרגע לא חשובים.
* נרצה להחליף מקומות בצורה שנתקדם איבר אחד ונשמור על הדרישות הללו.

צעד הבא:



j עובר

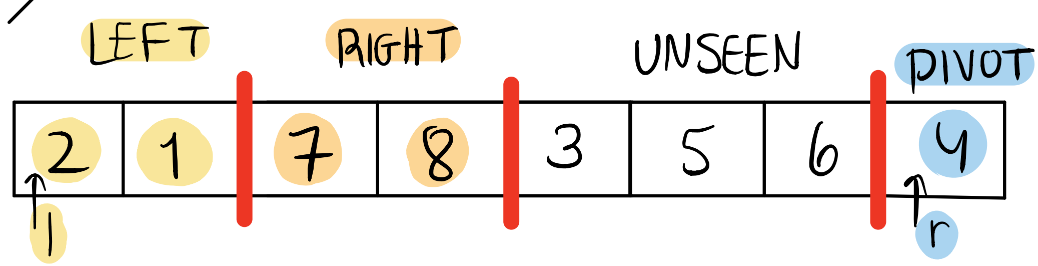
ל-$j+1$.

את האיבר הבא $A[j+1]$ נשווה כעת לערכו של ה-PIVOT, $A[r]$.

* שני מצבים: גדול / קטן ממנו.

נרצה לעשות החלפות מבלי להזיז את כל האלמנטים, כדי לחסוך בזמן.

התכונה של i ו j-1 מתבטאת משמאל לימין ובכל פעם מורידים UNSEEN אחד.



בהתחלה כולם יהיו UNSEEN ו- i,j בתחילת המערך,
ולאט לאט נתקדם בתוך המערך עפ"י חוקים.

```
Partition (A,l,r)
1. i ← l-1
2.   for j ← l to r-1
3.       do if (A[j] ≤ A[r])
4.           then i ← i+1
5.               Exchange (A[i], A[j])
6.   Exchange (A[i+1], A[r])
7. return i+1
```

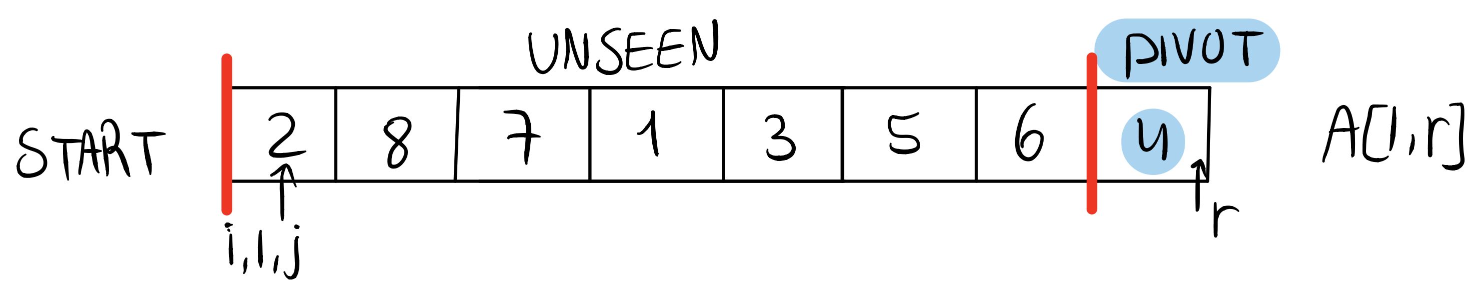
השוואת האיבר החדש ל-PIVOT

Because $A[j] < A[r]$
and $A[i] \le A[r]$

נעביר את ה-PIVOT

<u>נעבור על דוגמה מההתחלה:</u>

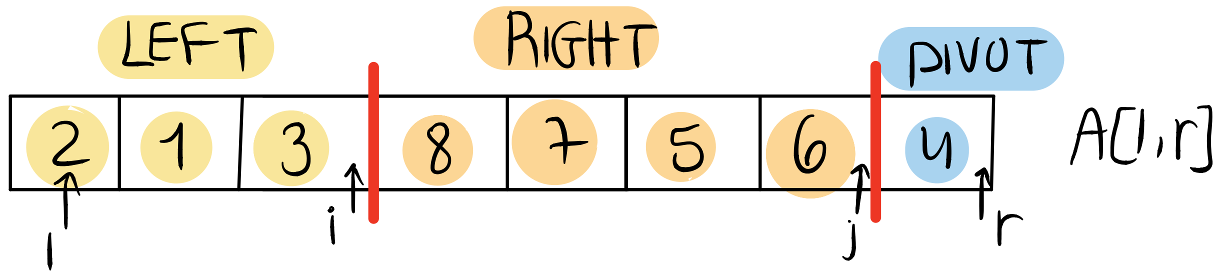
LEFT = $\emptyset$ RIGHT = $\emptyset$



UNSEEN = ∅

בסוף ההרצה:

END

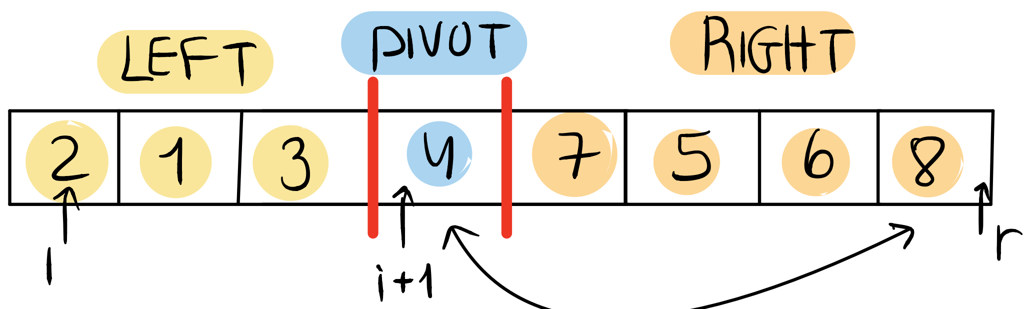


נרצה לוודא: $A[j] > A[r]$ ו- $A[i] \le A[r]$

עד עכשיו עשינו את שורות 1-5 באלגוריתם.

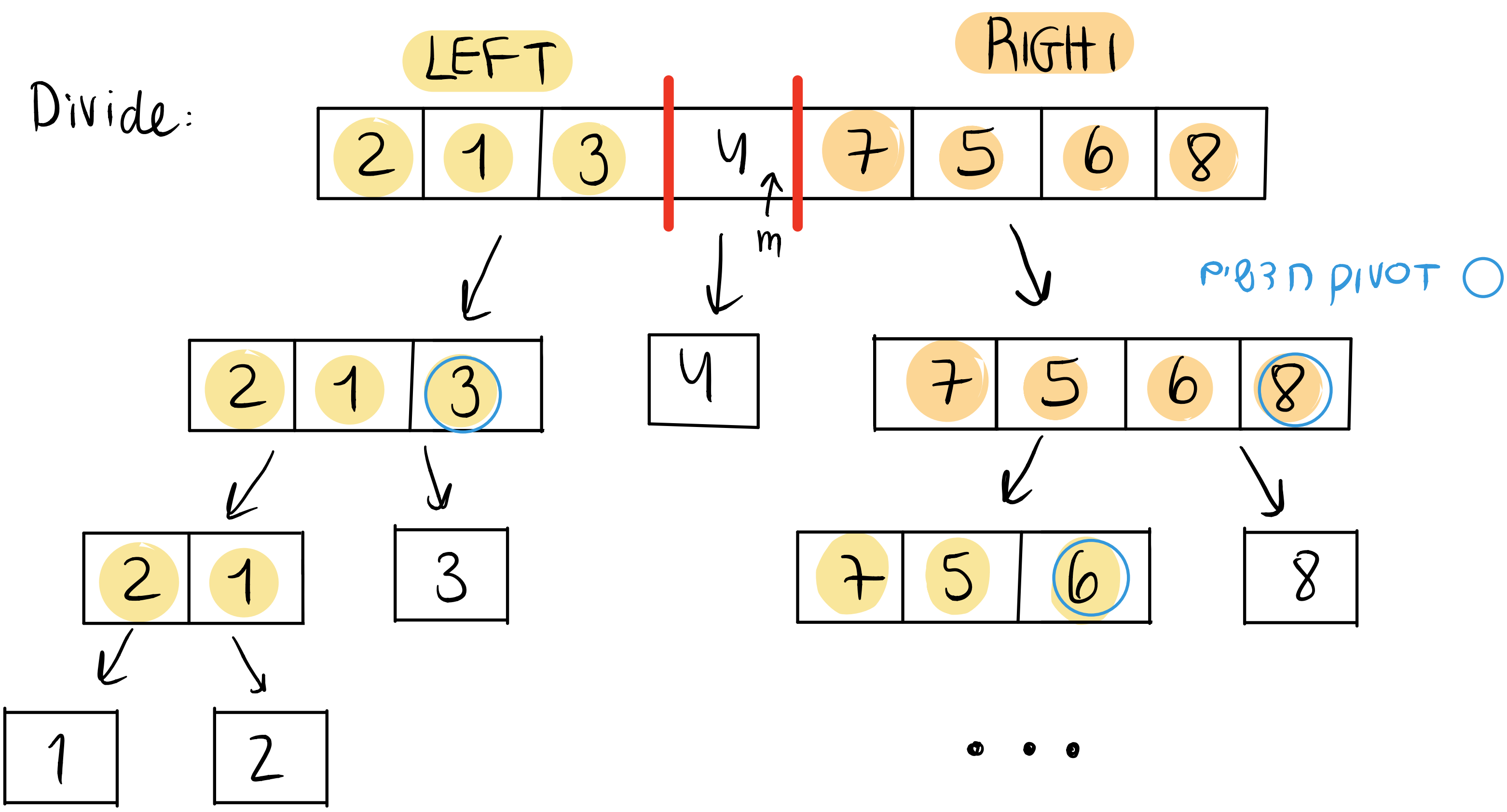
כעת נבצע את שורה 6.

ECHANGE

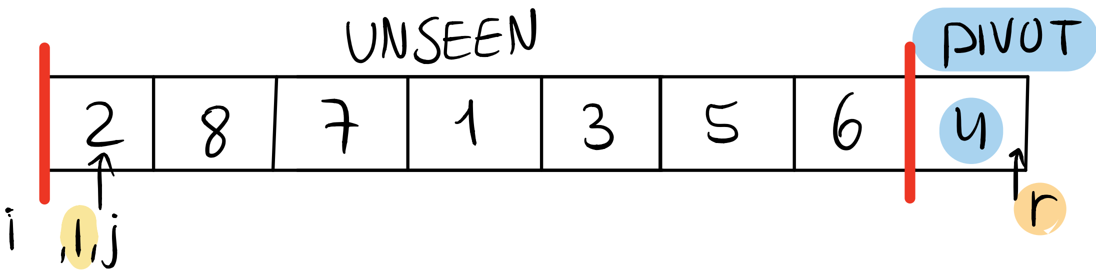


6. Exchange (A[i+1], A[r])

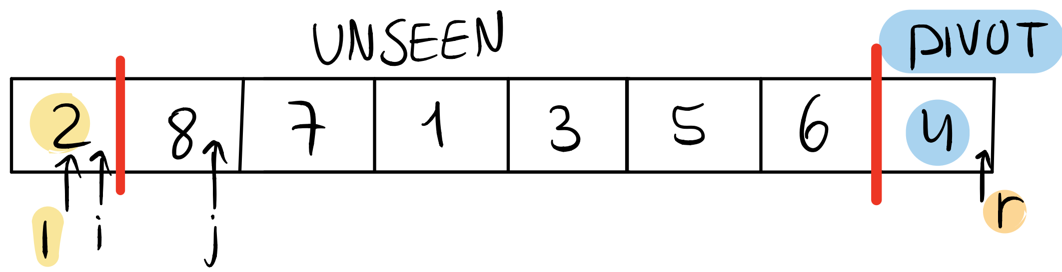
Divide:



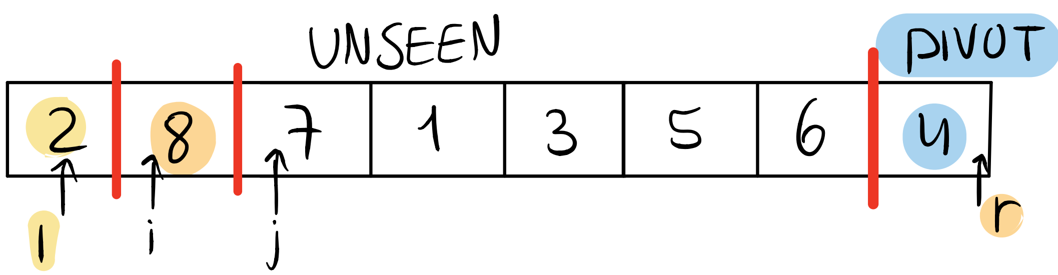
דוגמה



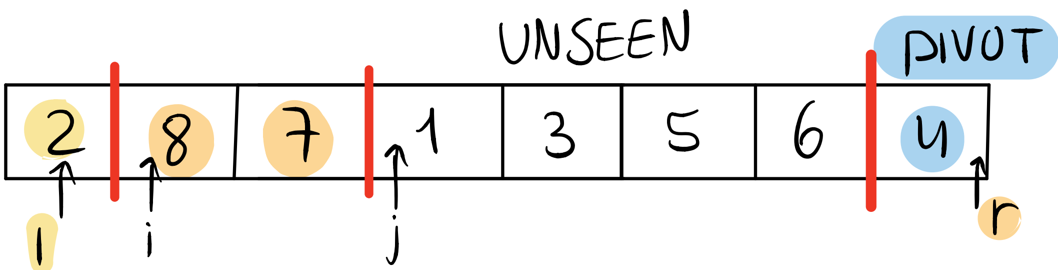
$i = l-1, j = l$



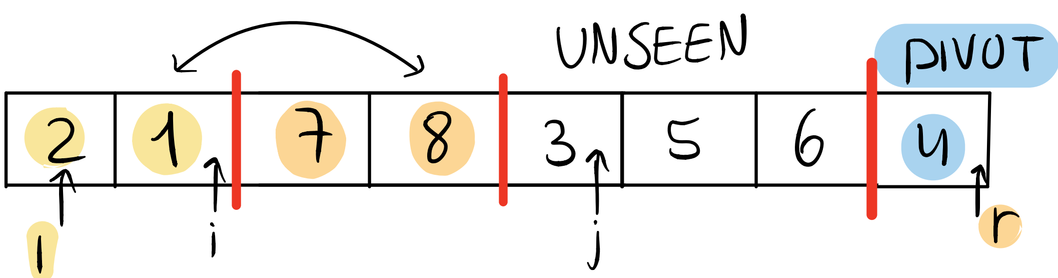
$2 < 4$
Exchange 2 w. 2
i,j++



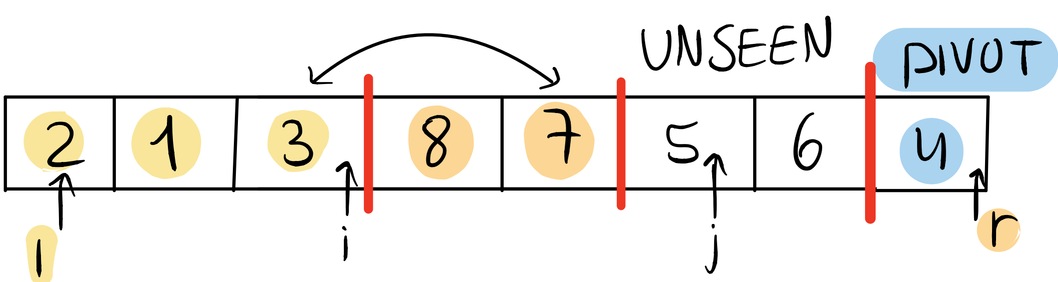
$8 > 4$
j++



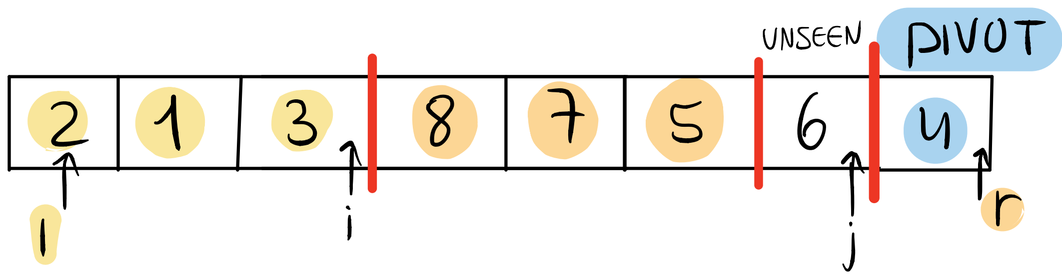
$7 > 4$
j++



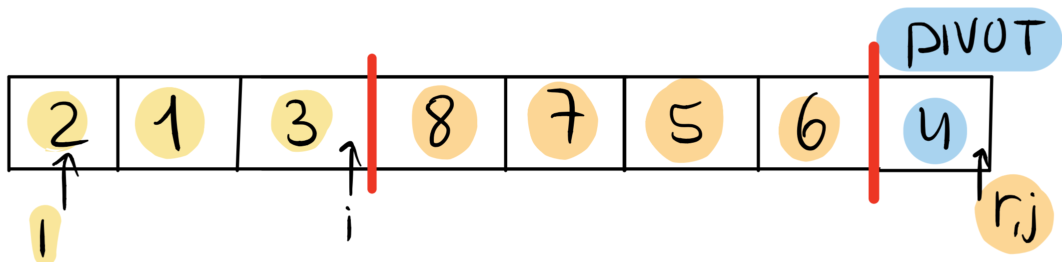
$1 < 4$
Exchange 1,8
i j++



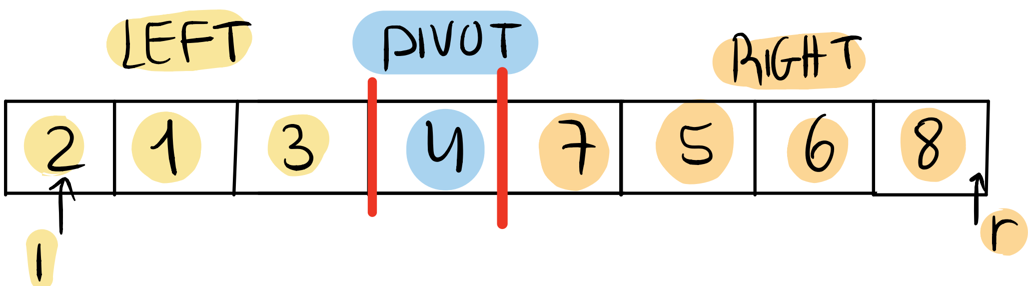
$3 < 4$
Exchange 3,7
i,j++



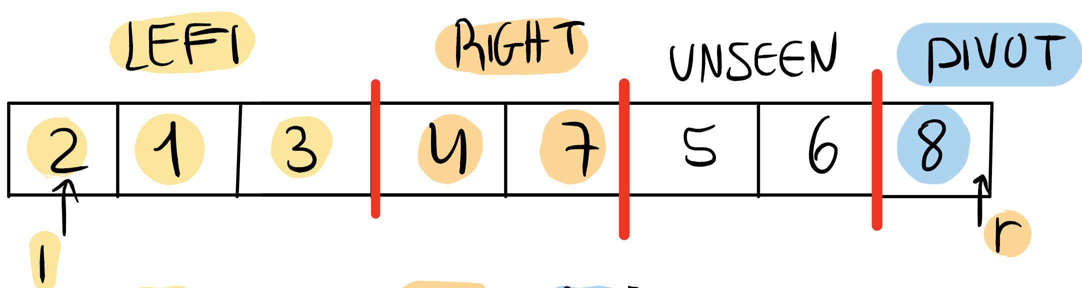
$5 < 4$
$j{+}{+}$



$6 < 4$
$j{+}{+}$



Exchange 4,8
2^{nd} iteration



$A[i] \leq A[r]$ $A[j] > A[r]$

מאוד חשוב בכל שלב אנחנו בתהליך,
הצורה הכללית תמיד נכונה, כלומר מתקיימים:

סיבוכיות זמן ומקום

* הסיבוכיות של אלגוריתם תלויה בכמה קריאות רקורסיביות נבצע.

ישנם שני סוגים של חלוקות: מאוזנת ולא מאוזנת.

1) גרוע **חלוקה לא מאוזנת:** ללא חלוקה, ישנם תתי בעיות בגודל $n-1$ ו-0 pivot.

2) אידאלי **חלוקה משלמת:** החלוקה מתחלקת תמיד באמצע כך שתתי הבעיות בגודל $\leq \frac{n}{2}$.

3) ממוצע **חלוקה מאוזנת:** החלוקה נמצאת איפשהו בין שני המצבים הנ"ל, ותתי הבעיות בגדלים k ו-$n-k$.

כעת נתעמק בכל אחד מהם:

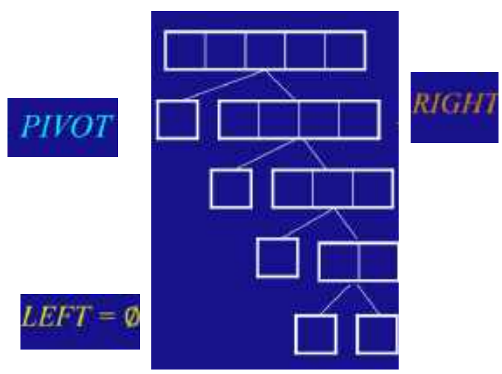
1) גרוע חלוקה לא מאוזנת:

נוסחת הנסיגה היא:

$$T(n) = \overset{PIVOT}{T(0)} + \overset{RIGHT}{T(n-1)} + \overset{n-1}{\Theta(n)}$$

נבחר ה-PIVOT,

במקרה הזה כל האלמנטים גדולים מה-PIVOT.



הפתרון של נוסחת הרקורסיה:

$$T(n) = T(0) + T(n-1) + \Theta(n) =$$

$$\sum_{K=1}^{n} \Theta(K) = \Theta\left(\sum_{K=1}^{n} K\right) = \Theta(n^2)$$

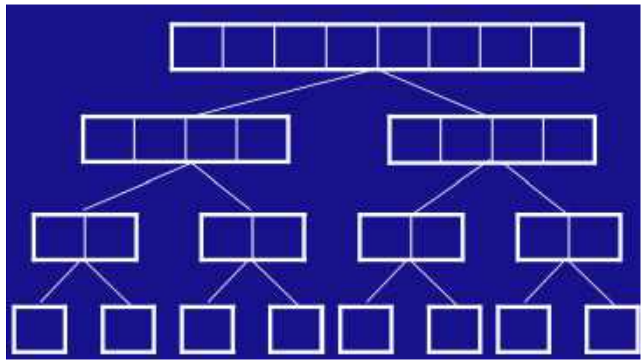
העומק יהיה n, ובכל שלב נבצע n עבודה ולכן המקרה הגרוע גורר $\Theta(n^2)$.

2) חלוקה מושלמת: אידיאלי

נוסחת הנסיגה היא:

RIGHT LEFT מעבר על המערך

$$T(n)= T\left(\frac{n}{2}\right)+T\left(\frac{n}{2}\right)+\Theta(n)$$



הפתרון של נוסחת הרקורסיה לפי משפט האב:

$$T(n)=2T\left(\frac{n}{2}\right)+\Theta(n) \implies$$

$$T(n)=\Theta(n\log n)$$

עומק העץ יהיה $\log(n)$ ובכל שלב נבצע n עבודה ולכן המקרה הטוב גורר $O(n\log n)$.

3) חלוקה מאוזנת: ממוצע

נוסחת הנסיגה היא:

RIGHT LEFT מעבר על המערך

$$T(n)= T(q)+T(n-q-1)+\Theta(n)$$

$$T(n)=\max_{0\le q<n}\{T(q)+T(n-q-1)\}+\Theta(n$$

ולכן המקרה הממוצע יפול בין המקרה הטוב למקרה הגרוע:

$$\Theta(n\log n)\le T(n)\le\Theta(n^2)$$

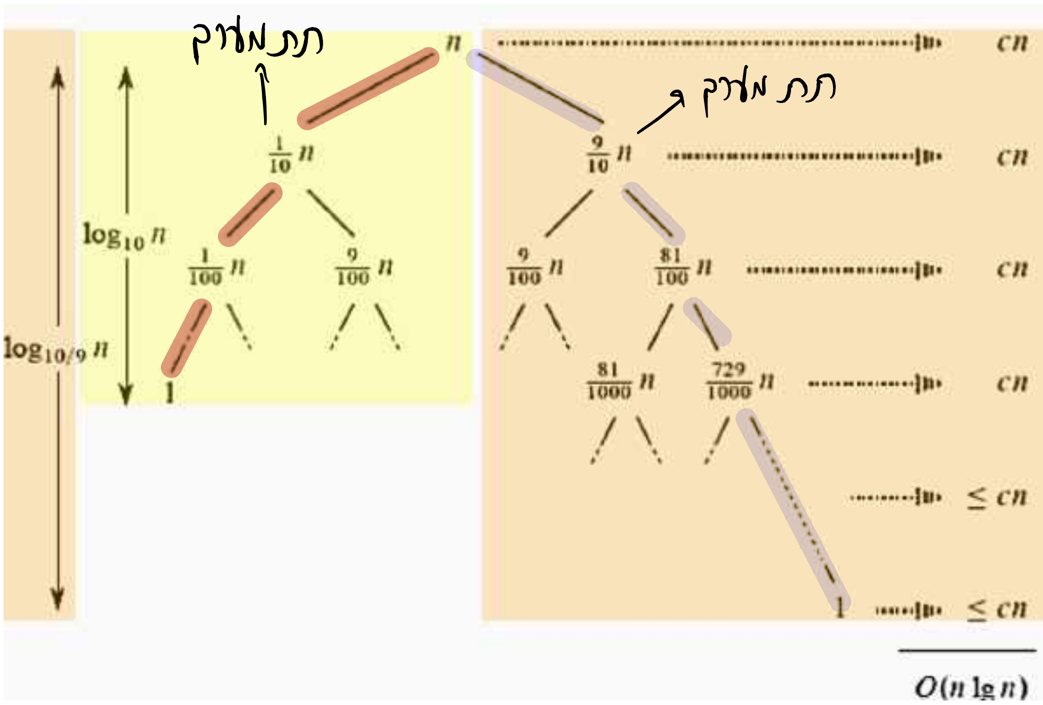
לפני שנפתור - ננסה לקבל קצת אינטואיציה ע"י דוגמה.

9-to-1 proportional Split

דוגמה:

נניח שה חלוקה תמיד מתחלקת ביחס של 1 ל- 9.

נביט בעץ הבא:



עומקים:

נסמן ב- ○ את העלה הכי ארוך (הכי עמוק)

מדובר ב: $\log_{\frac{10}{9}}(n)$

נסמן ב- ○ את העלה הכי קצר (הכי פחות).

מדובר ב: $\log_{10}(n)$

נוסחת הנסיגה היא: $T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n)$

מאחר שהעומק של הרקורסיה הוא $\log_{\frac{10}{9}}(n)$, כי אנו קיבלנו אותו לפי העומק המירבי, לכן מתבצעות $\Theta(\log n)$ חלוקות ובכל שלב מתבצעת $\Theta(n)$ עבודה.
לכן סך הכל, הסיבוכיות של אלגוריתם זה: $T(n)=\Theta(n\log n)$.

הערה מעניינת: בכל חלוקה לפי איזשהו יחס, כלומר - פרופורציונית, הסיבוכיות תמיד תהיה כבסיס, מכיוון שבצורה אסימפטוטית, זה מתנהג אותו דבר.

כעת נוכיח את הסיבוכיות של המקרה הגרוע:

$$T(n)=\max_{0\le q<n}\{T(q)+T(n-q-1)\}+\Theta(n)$$

ברגע שנוכל לבטא את $n-q$ כצורה של שבר נוכל להשוות את זה בצורה של חפוש.

נרצה להוכיח את הטענה הבאה: $T(n)\le c\cdot n^2=O(n^2)$

נוכיח באינדוקציה:

<u>בסיס:</u> $n=1$ הוכחנו ✓

<u>צעד:</u> נניח עבור $n<n'$ ונוכיח עבור n'.

$$T(n')=\max\{T(q)+T(n'-q)\}+\Theta(n')\le$$
$$c\cdot q^2+c(n'-q)+dn'=2cq^2+c(n')^2-2cqn'+dn'$$

על מנת להוכיח את הטענה נרצה להראות שכל זה קטן מ-$c(n')^2$,
או באופן שקול: $dn'\le 2cq(n'-q)$.

מאחר ש-$q(n'-q)$ תמיד גדול יותר מ-$\frac{n}{2}$, ניתן יהיה להוכיח זאת ע"י
בדיקת אחד משני המקרים: (1) $\frac{n}{2}\le q\le n$ (2) $q<\frac{n}{2}$
ומשם כך נוכל לבחור ב-c מתאים על מנת שיתקיים.

מיוני השוואות

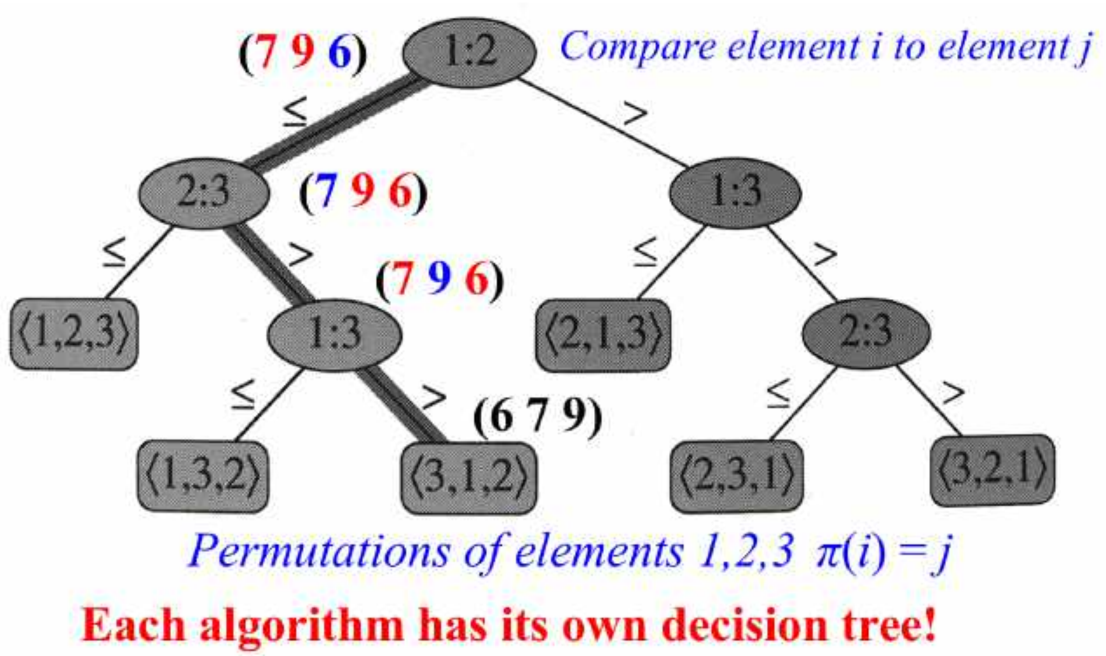
* כפי שציינו קודם, כל אלגוריתמי המיון שציינו עד כה מבוססים על השוואה בין שני אלמנטים.
* ראינו שמבחינת סיבוכיות לכל אחד הייתה סיבוכיות שונה, במקרה הגרוע - לא מצאנו סיבוכיות טובה מ-$\Omega(n\log n)$. היינו רוצים להוכיח שלא אכן החסם התחתון.
* מיון ללא השוואות יכול להשיג חסם תחתון שהוא $\Omega(n)$ אבל צריך תחת הנחות מסוימות.

חסם תחתון של מיוני השוואות

* נרצה להוכיח חסם תחתון (Ω) לסיבוכיות מיון במקרה הגרוע, לכל אלגוריתמי המיון שמשתמשים בהשוואה.
* כשביל להוכיח שלא קיים אלגוריתם עם סיבוכיות טובה יותר, נשתמש בעץ החלטה.
* לכל אלגוריתם A יש עץ החלטה שלו T, ומבוסס על הצורה שבה הוא משווה אלמנטים.

נסביר באמצעות דוגמה:

האלגוריתם מקבל מערך בגודל 3 ובודק את היחס בין $A[i]$ ל-$A[j]$ בכל צומת בצורת כתיב: $i:j$ כאשר i,j הם האינדקסים.



העלים מייצגים את סדר האינדקסים של המערך הממוין, עפ"י המערך הקודם.

נשים לב שהעץ לא מאוזן, והמסלולים ייבחרו לפי ה-input שניתן.
כלומר, העומק של המסלול שייבחר עבור מערך נתון כלשהו ייקבע
לפי איך שהמערך היה מסודר לפני כן.

* האורך של הדרך הארוכה ביותר, כלומר העלה הארוך ביותר מהשורש ועד העלה הוא זה שקובע את מספר ההשוואות המקסימלי שנצטרך לעשות.

עצי החלטה

הגדרה: עץ בינארי מלא שמייצג את ההשוואות בין אלמנטים שמבוצעים ע"י אלגוריתם ספציפי. בעל קודקודים פנימיים, עלים וענפים:

* קודקוד פנימי: בעל שני אינדקסים $i:j$ כך ש- $1 \le i,j \le n$
* עלה: פרמוטציה של הקלט $\pi(1),\dots,\pi(n)$ (איך צריך לסדר את ה-input)
* ענף: התוצאות של ההשוואה: $a_i > a_j$ (ימין) או $a_i \le a_j$ (שמאל)

מסלול בעץ החלטה

הגדרה: המהלך שמורכב מאוסף השוואות בעץ מהשורש ועד העלה.

* כל קודקוד פנימי מייצג yes/no עבור איזושהי שאלה, ושני החצים שיוצאים מאותו קודקוד מייצגים תשובות אפשריות.

חישוב ע"פ עץ החלטה

החישוב מתחיל מהשורש, עובר מצומת לצומת ע"פ השאלות והתשובות בצמתים.
החישוב מסתיים כאשר נגיע לעלה.

כל אלגוריתם שממיין נכון חייב להגיע לכל פרמוטציה אפשרית
כאשר ישנו n איברים, יש לנו n! אפשרויות לסידור המערך.

הערה:
כך ניתן לבדוק האם האלגוריתם נכון - האם ישנו לפחות n! אפשרויות.
למה לפחות? ייתכנו אלגוריתמים עם יותר, פחות יעילים.

שאלה: מה העומק של עץ בינארי מלא שיש לו n! עלים?
פתרון: נסמן ב-d את עומק העץ.
האורך המינימלי של הדרך הארוכה ביותר תהיה:

$$n! \le 2^d \Longrightarrow \log n! \le \log(2^d) = d$$

בדיקה:

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \le n! \le n^n \Longrightarrow$$

$$\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) \le \log(n!) \le n\log(n)$$

$$\log(n!) = \Theta(n\log n)$$

הוכחה (לא עברנו על זה):

Claim: $\log(n!) = \Theta(n\log(n))$ for $n \ge 2$

Proof:
$$\log(n!) = \log\left(\prod_{i=1}^{n} i\right) = \sum_{i=1}^{n} \log(i)$$
$$\ge \sum_{i=\frac{n}{2}}^{n} \log\left(\tfrac{n}{2}\right) = \tfrac{n}{2}\log\left(\tfrac{n}{2}\right)$$
$$= \Omega(n\log(n)).$$

On the other hand:
$$\log(n!) = \sum_{i=1}^{n} \log(i) \le n\log(n).$$

This is the lower bound on the number of comparisons in any comparison-based sorting algorithm.

טבלת סיבוכיות:

	Space	Worst case	Best case	Average case	Random. case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	---
Quick Sort	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

11.11.20

הרצאה 5- Data Structures
Hash Tables

Hash Tables

1. מוטיבציה והגדרות

* ישנן הרבה משימות שדורשות פעולות טבלה. הכוונה היא לטבלה (מילון) שמחזיקה סמלים (keys) ומשמעויות (values) וזמן השליפה שלאובר בטבלה הוא O(1) בממוצע.

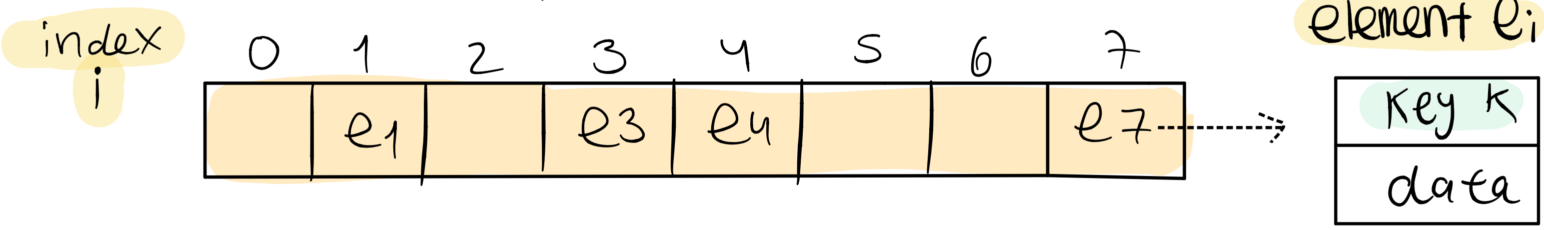
* אין חשיבות לסדר ה-keys במבנה נתונים זה.
דוגמאות: דוגמה סטטית - רשימת חדרים באוניברסיטה.
דוגמה דינמית - לקוחות בבנק, מספרי טלפון.

* נחפש מבנה נתונים שיאפשר לנו ביצוע פעולות:
Insert, Search, Delete
ב- O(1) בממוצע ללא סדר מסוים בין האלמנטים.

* עצי חיפוש הם בעלי סדר מסוים ודורשים O(logn) חיפוש, ואילו טבלות-גיבוב, המאפשרות שימוש במפתחות ללא סדר ודורשת O(1) בממוצע (זמן קבוע).

מוטיבציה עבור Hash-Tables

* נתון מערך A בגודל m, אינדקסים $0\le i\le m-1$.
כל איבר e_i ישנו מפתח key ויש ערך data



* פעולות שנבצע על הטבלה: search(e8), insert(e3), delete(e1)

* נרצה להגדיר איכשהי פונקציה $h(K)$ שמקבלת K (מפתח)
ומחזירה אינדקס i במערך שבו מופע הערך שנחפש. $h(K) = i$
כלומר, נרצה למצוא דרך לקשר בין מפתח לערך.
דוגמה פשוטה: $h(K) = K$ כאשר האינדקס במערך הוא המפתח.

* הטווחים של האינדקסים והמפתחות יצטרכו להיות חופפים. $[0,m-1]$

בעיה: ישנו פוטנציאל לבזבוז מקום, מכיוון שפעולה כזו גורר
הקצאת המון מקום בזכרון כאשר רובו יהיה ריק.

פתרון: נמצא איכשהי חוקיות כדי להשתמש בפחות מקום.
לדוגמה: במקום להשתמש בכל הספרות של מס' זהות
ארוך כמו: 3452334, נוכל להשתמש ב-2 האחרונות,
כלומר, $h(3452334) = 34$.

בעיה: ישנו סיכוי להתנגשויות, לדוגמה: אם יגיע לנו בעל
מס' זהות כזה: 1756634 אז $h(3452334) = 34$ שוב.

מכיון והצגה נדון בבעיה זו ומה ניתן לעשות איתה.

מבנה כללי של Hash-Tables

טבלת גיבוב (Hash-Table) היא הטבלה של מערך A בגודל m, שמאפשרת לחשב את האינדקס של האיבר במערך, ע"י פונקציית גיבוב (Hash-Function).

* גודלה של הטבלה יהיה פרופורציונלי לכמות האיברים במערך, בצורה הבאה: $h(k) = k \bmod m$. כלומר היא אינה חח"ע במקרה כזה.

* כפי שציינו קודם, קיים חשש להתנגשות גם פה, כאשר שני מפתחות ממופים לאותו אינדקס, בצורה הבאה: $k_1 \neq k_2$ אבל $h(k_1) = h(k_2)$!

דוגמאות:

1. לוחיות רישוי של רכב: $h(55\text{-}080\text{-}72) = 5508072 \bmod 100$
2. מילות מפתח בשפות תכנות: $[for, if, then...] \rightarrow h(for)=0, h(if)=1, ...$

הגדרה:

תהי U קבוצה של מפתחות, וגודלה הוא $|U|$.

תהי K קבוצת המפתחות הנוכחיים כך ש: $K \subset U$ $|K| = n-1$.

תהי T טבלת גיבוב המקיימת $|T| = m \leq |U|$.

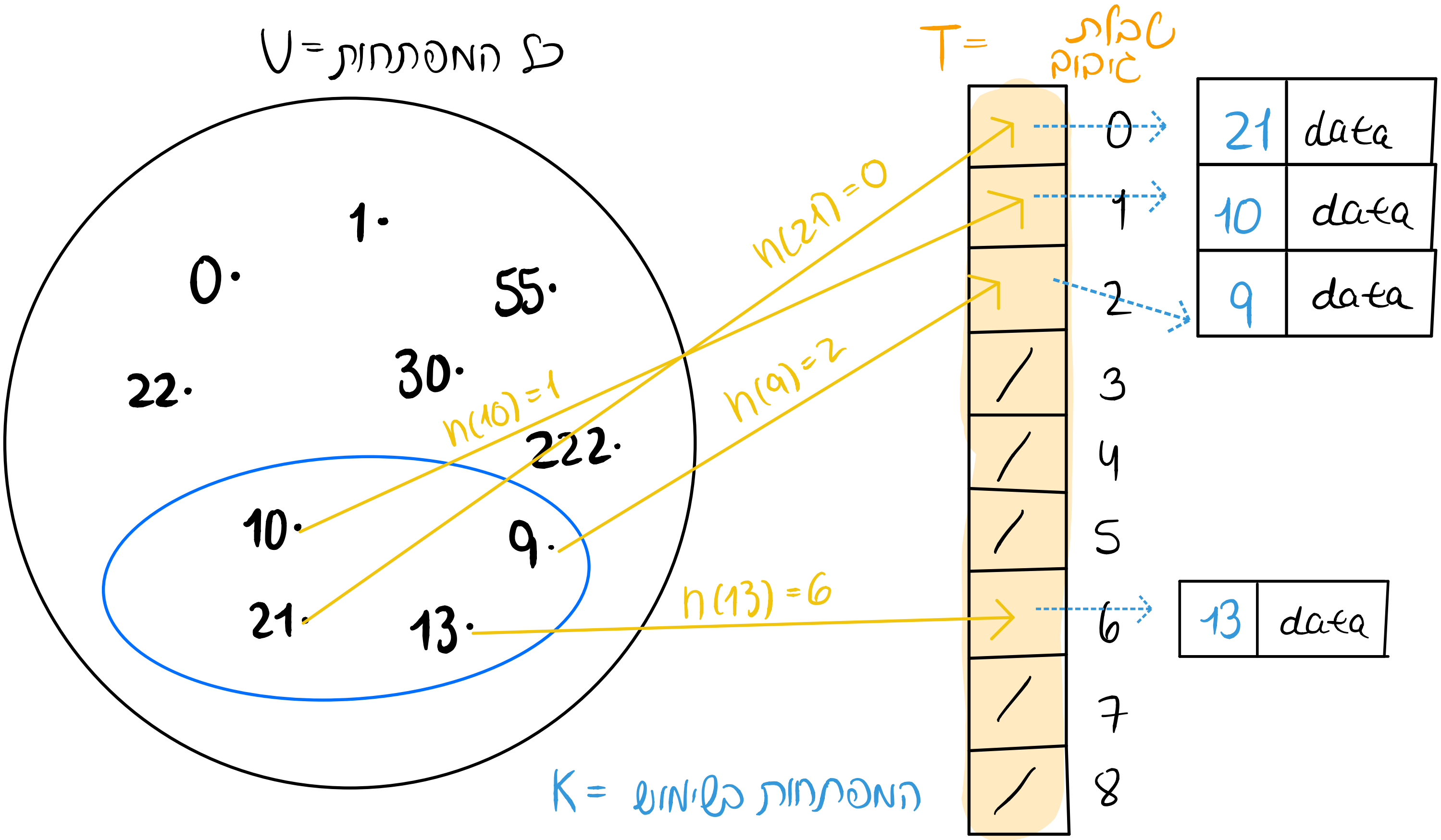
נגדיר פונקציית גיבוב באופן הבא: $h(k): \overset{\text{מפתחות}}{U} \rightarrow \overset{T \text{ אינדקסים של}}{[0, ..., m-1]}$.

לכל $1 \leq i \leq m-1$ מתקיים כי הגישה ל-$T[i]$ היא בזמן ריצה ממוצע של $O(1)$.

כל האיברים הריקים בטבלת הגיבוב יסומנו ב-null.

נניח שהמפתחות הם טבעיים ומתקיים $U = \{0, 1, ..., N-1\}$

המחשה:



פעולות: insert(10), insert(21), insert(9), insert(13)

ישנם הרבה סוגים של טבלאות גיבוב ובהתאמה פונקציות גיבוב, כל אחת מכריעה באופן שונה את סוגיית ההתנגשות. נדבר על 3 סוגים של טבלאות גיבוב:

1. Direct-Adress
2. Open Adressing
3. Closed-Adressing (chaining)

הנחות על פונ' הגיבוב (Simple uniform hashing assumption):

כאשר נתון מערך, ניתן למפות כל ערך לתוכו, ללא תלות בשאר איברי המערך.

מקדם עומס (load factor):

כאשר נתונה טבלת גיבוב T עם n מקומות, נגדיר את מקדם העומס להיות $0 \leq \alpha = \frac{m}{n} \leq 1$ כאשר m הוא מספר האלמנטים המשובצים.

1. Direct-Adress (מיעון ישיר)

מדובר בדרך הפשוטה ביותר, גודל T יהיה גודל A, $h(k)=k-1$.

אין חשש להתנגשויות, וזמן הריצה הוא $O(1)$, אך ישנו בזבוז מקום וחוסר יעילות.

DIRECT-ADDRESS-SEARCH(T, k)
1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
1 $T[x.key] = $ NIL

פסאודו קוד של חיפוש, הכנסה ומחיקה מהטבלה.

2. Open Adressing (מיעון פתוח)

בשיטה זו, נפעיל את פונקציית הגיבוב על הערך באינדקס ובמידה והמקום תפוס ע"י ערך אחר, נמשיך לאיבר הבא על פי קוסחה.

ישנה פונקציית $h(k,i): U \times [0,..,m-1] \rightarrow [0,...,m-2]$ שמבצעת בדיקות עפ"י אינדקס (i) שלך.

פונקציית הגיבוב תוגדר כך: $h(k,i): U \times [0,..,m-1] \rightarrow [0,...,m-1]$

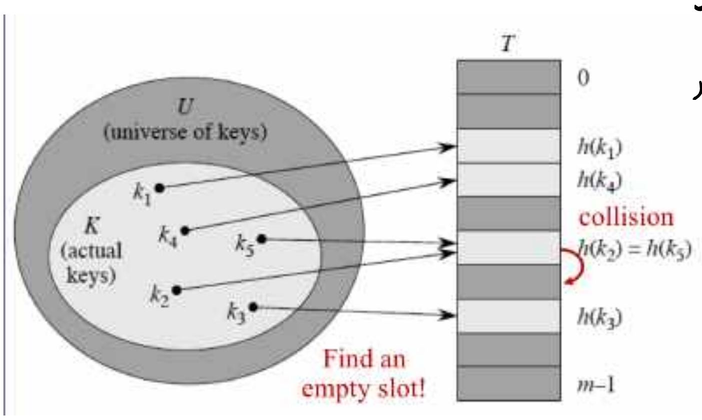
אופן הפעולות:

הכנסה: מפעילים את פונקציית הגיבוב, בודקים את המיקום בטבלה עד שמוצאים איבר ריק.

חיפוש: מפעילים את פונקציית הגיבוב, עד שמוצאים את האיבר ✓ או שמוצאים איבר ריק ✗.

מחיקה: לאחר מחיקת איבר מסוים, נסמנו כמחוק כדי לא לפגוע באופן פונקציית הגיבוב.

* סיבוכיות זמן הריצה תקבע על פי מספר האיברים.



שיטה זו מתמודדת עם התנגשויות באמצעות נוסחה מתמטית שיודעת לחשב אינדקס אחר, במידה והאינדקס תפוס.

פסאודו קוד של חיפוש,הכנסה ומחיקה מהטבלה.

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

HASH-DELETE(*T*,*k*)
Cannot just delete *k*
- Leave a mark DELETED
- Modify HASH-INSERT

ישנם שלושה סוגים של Open Addressing:
1) בדיקה לינארית - Linear Probing
2) בדיקה ריבועית - Quadratic Probing
3) גיבוב כפול - Double Hashing

1) בדיקה לינארית - Linear Probing

כדי שיישאר בתוך המערך

נגדיר את פונקציית הגיבוב עבור בדיקה לינארית: $h(k,i) = (h'(k) + i) \bmod (m)$

כאשר $h'(k)$ מוגדרת כפונקציית גיבוב קבועה (פונ' עזר), ו-i הוא האינדקס עליו רצים.

אם נקח איבר K, סדרת הבדיקות (עד שימצא מקום פנוי/ המקום הרלוונטי) תהיה: $(T[h'(k)], T[h'(k)+1], \dots, T[h'(k)+m-1])$.

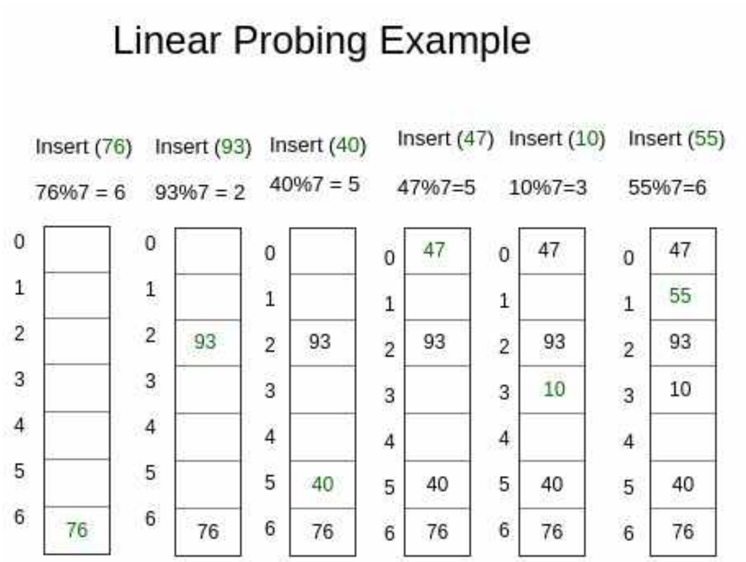
נשים לב שהן מונוטוניות עולה ונוכל לייצר עד m סדרות בלבד.

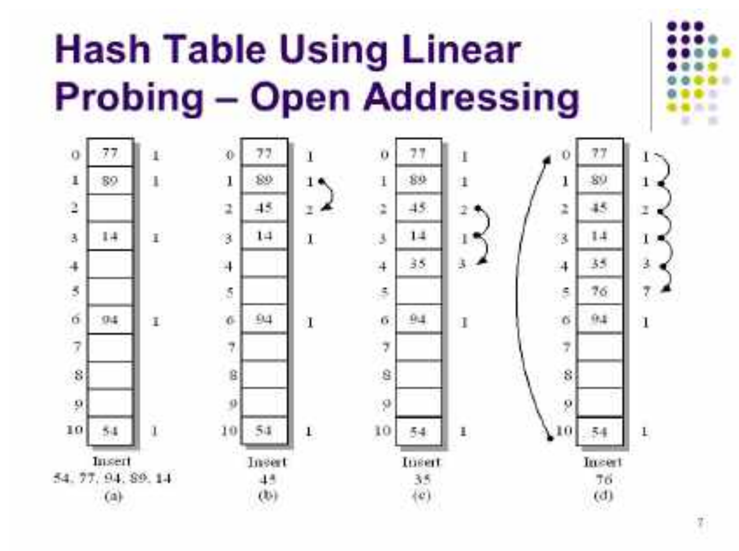
סדרה כזו, עד שנגיע להצלחה, מייצגת את הבלוק שנצטרך לעבור כל פעם שנרצה לעשות פעולה מסוימת על אותו איבר שהכנסנו.

במקרה הגרוע ביותר נקבל שככל שאנחנו עולים באינדקסים, ההסתברות לקבל מקום פנוי הולכת וקטנה. ולכן זמן החיפוש הולך ועולה.

בעיה זו נקראת primary clustering והיא מתארת את צפיפות הבלוקים.

<u>דוגמאות:</u>





היינו רוצים למצוא פונ' שמפזרת בצורה יותר מרווחת (אולי לא ליניארית כמו פה) את הבלוקים האלו ומשם נגיע ל:

2) בדיקה ריבועית - Quadratic Probing

נגדיר את פונקציית הגיבוב עבור בדיקה ריבועית:

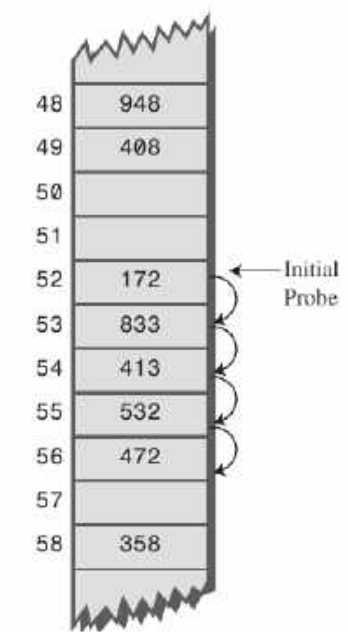
$$h(k,i) = (h'(k) + c_1 i + c_2 i_k^2) \bmod m$$

כדי שיישאר בתוך המערך

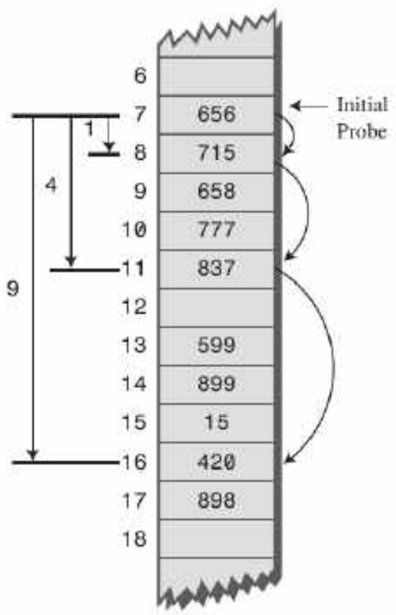
כאשר $0 < c_1, c_2$ ו-$h'(k)$ מוגדרת כפונקציית גיבוב קבועה, ו-i הוא האינדקס עליו רצים. גם במקרה הזה מדובר בצורה של m סדרות.

פונקציית הגיבוב הזו אכן מפזרת את צפיפות הבלוקים שדיברנו עליה בבדיקה ליניארית, מכיוון שה-i^2 גורם ל"קפיצה" גדולה יותר בין האינדקסים אליהם נכנסים מיד.

כאן אנו מונעות ה- primary clustering אבל כן קיימת בעיה מסוג secondary clustering, כי אנחנו מדלגים על בלוקים ולא על איברים. כלומר, ישנם רצפים של קפיצות. נציג ויזואלית את ההבדלים:

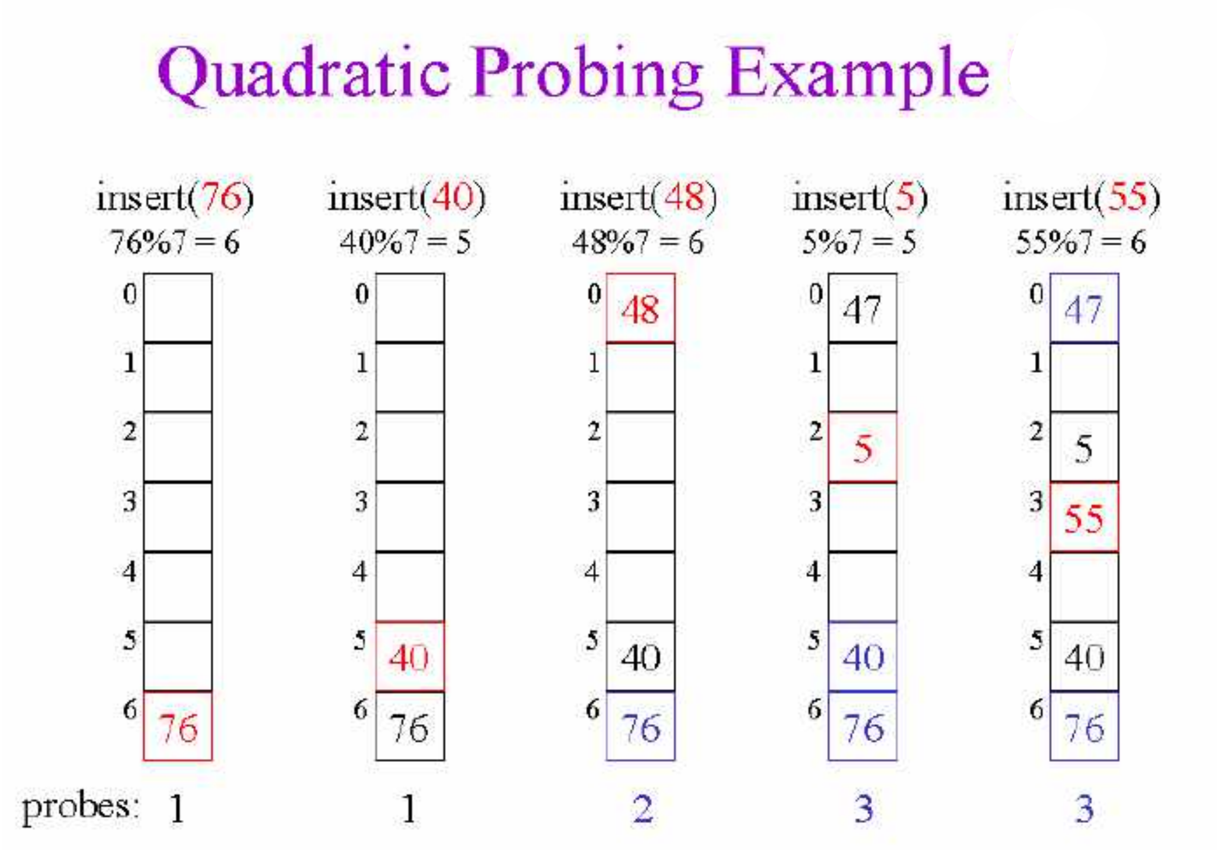


primary clustering



Secondary clustering

דוגמה:



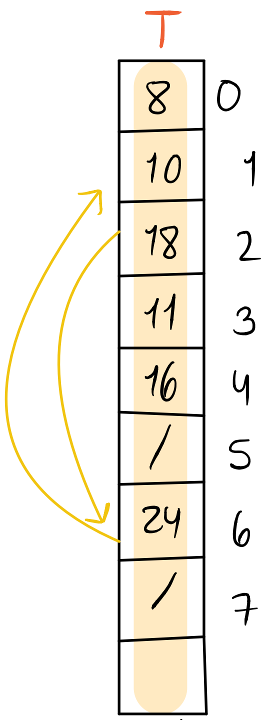
3. גיבוב כפול - Double Hashing

נגדיר את פונקציית הגיבוב הבאה: $h(k,i) = (h_1(k) + i \times h_2(k)) \bmod m$

כאשר $h_1(k)$ ו- $h_2(k)$ הן שתי פונקציות גיבוב.

הבדיקה הראשונה תתבצע ב- $T[h_1(k)]$ ושם ממשיכים ב- $h_2(k) \bmod m$.

הערך של $h_2(k)$ חייב להיות ראשוני על מנת להבטיח שנגיע לכל m המקומות בטבלת הגיבוב. כך נוצר m^2 אפשרויות של סדרות ולכן מבחינה הסתברותית הפיזור יהיה טוב יותר.



דוגמה: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, $m' < m$

כאשר m הוא חזקה של 2 ו- $m' < m$.

דוגמה: $h_2(k) = 1 + (k \bmod 5)$, $h_1(k) = k \bmod 7$

פעולת: insert(9) $\longleftarrow$ $h_1(9) = 2$, $h_2(9) = 5$

$\longleftarrow$ $\underbrace{h(9,0) = 2,\ h(9,1) = 6,\ h(9,2)}_{\text{probing sequences}} = 1$

הערה: בהינתן טבלת גיבוב מסוג open-adressing, כאשר ה-load factor הוא $\alpha = \frac{n}{m} < 1$, אז מספר הבדיקות הצפוי בפונקציית גיבוב הוא:

* לכל היותר $\frac{1}{1-\alpha}$ בחיפוש לא מוצלח
* לכל היותר $\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$ בחיפוש מוצלח

נקבל חסם למספר הבדיקות הנדרשות:

(1) אם α קבוע, מספר הבדיקות יהיה קטן ולכן החסם יהיה $O(1)$.

(2) כאשר α הוא $O(n)$, המצב יהיה מורכב יותר.

3. Closed-Adressing (chaining)

נמיין את כל המפתחות ה- k_i לתוך $T[h(k_i)]$ כאשר T הוא טבלת גיבוב.
בכל אחד מאיברי המערך ישנו מצביע לרשימה מקושרת.
הפעולות מתבצעות באופן הבא:

* **הכנסה:** מתבצעת לראש הרשימה המקושרת. זמן ריצה O(1) במקרה הגרוע
* **חיפוש ומחיקה:** כניסה לתוך הרשימה המקושרת באמצעות פונקציית גיבוב, ושם חיפוש ומחיקה.

ולכן זמן הריצה במקרה הגרוע ביותר יהיה פרופורציונלי לאורך הרשימה המקושרת הארוכה ביותר.

המחשה:

פעולות: insert(21), insert(10), insert(9), search(10)