

## הגדרות

**אלגוריתם** - דרך שיטתית וחד משמעית לביצוע של משימה מסוימת במספר סופי של צעדים

**פסאדו קוד** – תיאור מצומצם ולא רשמי לאלגוריתם של תוכנית מחשב

**זמן ריצה** – מספר פעולות היסוד המבוצעות על תוכנית כלשהי  
**למה פעולות יסוד ולא זמן?** על מנת להתעלם מאספקטים טכנולוגיים כמו סוגי מכונות

**דוגמא:** חיפוש בינארי מול חיפוש טרינרי

חיפוש בינארי	חיפוש טרינרי
$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 2, n \geq 2 \\ 1, n \leq 2 \end{cases}$ $T(n) = \dots = T\left(\frac{n}{2^k}\right) + 2 \cdot k = 2 \cdot \log_2 n$	$T(n) = \begin{cases} T\left(\frac{n}{3}\right) + 4, n \geq 2 \\ 1, n \leq 2 \end{cases}$ $T(n) = \dots = T\left(\frac{n}{3^k}\right) + 4 \cdot k = 4 \cdot \log_3 n$

$$2 \cdot \log_2 n = 2 \cdot \frac{\log_3 n}{\log_3 2} = 3.17 \cdot \log_3 n < 4 \cdot \log_3 n$$

**אסימפטומטיקה – הערכה של קצב גידול של פונקציה**

**חסם עליון** – נאמר ש-  $f(n) \in O(g(n))$  אם:

$$\exists c > 0, n_0 \geq 0 : \forall n > n_0 \quad f(n) \leq c \cdot g(n)$$

**חסם תחתון** – נאמר ש-  $f(n) \in \Omega(g(n))$  אם:

$$\exists c > 0, n_0 \geq 0 : \forall n > n_0 \quad f(n) \geq c \cdot g(n) \geq 0$$

**חסם הדוק** – נאמר ש-  $f(n) \in \theta(g(n))$  אם:

$$\exists c_1, c_2 > 0, n_0 \geq 0 : \forall n > n_0 \quad c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

שם	סימון
אקספוננציאלי	$O(c^n)$
פולינומי	$O(n^c)$
	$O(n \cdot \log(n))$
לינארי	$O(n)$
	$O(\sqrt{n})$
לוגריתמי	$O(\log(n))$
	$O(\log(\log(n)))$
קבוע	$O(1)$
	$o\left(\frac{1}{n}\right)$

## מיונים

## הקדמה:

**מיון מבוסס השוואות** – מיון מבוסס השוואות מסדר אלמנטים במערך ע"י השוואה, בדרך כלל ע"י האופרטורים  $\{\leq, \geq\}$

כל אלגוריתם המיון המבוסס על פעולות השוואה דורש לפחות  $\Omega(n \cdot \log(n))$  פעולות השוואה **במקרה הגרוע ביותר**



**הוכחה:** עבור מיון  $n$  איברים יש צורך בעץ החלטה עם  $n!$  עלים (עבור כל אחת מהפרמוטציות השונות), בעץ החלטה בגובה  $h$  יש לכל היותר  $2^h$  עלים, ולכן:  

$$2^h \geq n! \Rightarrow h \geq \log_2(n!) \in \Omega(n \cdot \log n)$$

על מנת להשיג סיבוכיות מיון טובה יותר מ- $\Omega(n \cdot \log n)$  (ליניאריים) נוכל לאבד את הכלליות ולהתעסק במקרים פרטניים כמו **מספרים נמצאים בטווח חסום** (*Counting Sort*), **סדרה של מספרים הנמצאים בטווח חסום** (*Radix Sort*)

	מבוססי השוואות					לא מבוססי השוואות	
	<i>Bubble Sort</i>	<i>Selection Sort</i>	<i>Insertion Sort</i>	<i>Merge Sort</i>	<i>Quick Sort</i>	<i>Radix Sort</i>	<i>Counting Sort</i>
$T(n)$	$T(n) = (n-1) + \dots + 1 = n \cdot \frac{n-1}{2}$			$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$	$T(n) = T(k) + T(n-k-1) + O(n)$		
$O$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \cdot \log n)$	$O(n^2)$	$O(d \cdot (n+k))$	$O(n+k)$
$\Omega$	$\Omega(n^2)$	$\Omega(n^2)$	$\Omega(n)$	$\Omega(n \cdot \log n)$	$\Omega(n \cdot \log n)$	$\Omega(d \cdot (n+k))$	$\Omega(n+k)$
$\Theta$	$\Theta(n^2)$	$\Theta(n^2)$		$\Theta(n \cdot \log n)$	$\Theta(n \cdot \log n)$ בהסתברות גבוהה	$\Theta(d \cdot (n+k))$	$\Theta(n+k)$

## מבני נתונים

מבנה נתונים	הגדרה	מוטיבציה	סיבוכיות הכנסה	סיבוכיות חיפוש	סיבוכיות מחיקה
מערך סטטי	מבנה שמורכב מאוסף של תאים <b>סדרתיים</b> בזיכרון בעל גודל <b>קבוע</b>	גודל המערך ידוע בזמן קומפילציה • רוצים <b>Random Access</b> (גישה ישירה בזמן קבוע)	$O(1)$	$O(n)$	$O(1)$
מערך דינמי	מבנה שמורכב מאוסף של תאים <b>סדרתיים</b> בזיכרון בעל גודל <b>ניתן לשינוי</b>	גודל המערך לא ידוע בזמן קומפילציה	$O(1)$ עד כדי $resize$ שלוקח $O(n)$	$O(n)$	$O(1)$
מערך דינמי ממויין	אוסף של תאים סדרתיים בזיכרון כך שכל איבר קטן מהאיבר הבא לפי אופרטור השוואה כלשהו $\{\leq, \geq, \dots\}$	נרצה לבצע חיפוש מהיר, כאשר יש יותר חיפושים מאשר הכנסות	$O(n)$ בגלל shifting של האיברים	$O(\log n)$	$O(n)$
רשימה מקושרת	<ul style="list-style-type: none"> <li>לכל איבר יש ערך</li> <li>לכל איבר יש לכל היותר מצביע יחיד לאיבר הבא</li> </ul>	יותר הכנסות או אין צורך ב- <b>Random Access</b> , פחות אפקטיבי (זניח) מבחינת זיכרון עבור שמירה גם על גודל המצביע	$O(1)$	$O(n)$	$O(1)$
רשימה מקושרת ממוינת	<ul style="list-style-type: none"> <li>לכל איבר יש ערך</li> <li>לכל איבר יש לכל היותר מצביע יחיד לאיבר הבא, שהוא קטן מהאיבר הבא לפי אופרטור השוואה כלשהו <math>\{\leq, \geq, \dots\}</math></li> </ul>	<ul style="list-style-type: none"> <li>מימוש לתור עדיפויות היות ומחיקה מהראש ומהסוף היא פעולה קלה</li> <li>מחיקה של איברים זהים כאשר אין זיכרון לטעון ל-HashSet לדוגמא</li> </ul>	$O(n)$	$O(n)$	$O(n)$
מחסנית	<ul style="list-style-type: none"> <li>מבנה נתונים אשר תומך בסדר פעולות היסטורי (האחרון הוא הראשון לצאת)</li> </ul> 	<ul style="list-style-type: none"> <li>LIFO</li> </ul>	$O(1)$	-	$O(1)$
תור	מבנה נתונים אשר תומך בסדר פעולות היסטורי (האחרון הוא האחרון לצאת)	<ul style="list-style-type: none"> <li>FIFO</li> <li>הערה: ניתן לממש תור בעזרת 2 מחסניות ומחסנית בעזרת 2 תורים (יש שקילות)</li> </ul> 	$O(1)$	-	$O(1)$

$O(n)$	$O(n)$	$O(n)$		עץ בינארי (לכל קודקוד יש ערך ולכל קודקוד יש רשימה של לכל היותר 2 קודקודים) כך שעבור כל קודקוד בעל ערך $x$ : 1. הערכים הקטנים מ- $x$ נמצאים בתת עץ השמאלי 2. הערכים הגדולים מ- $x$ נמצאים בתת עץ הימני	עץ חיפוש בינארי
$O(\log n)$	$O(\log n)$	$O(\log n)$	<ul style="list-style-type: none"> <li>הכנסה, מחיקה, חיפוש יעיל</li> <li>חיפוש תווים, איברים סמוכים, מבנה נתונים בעל סדר, InOrder</li> </ul>	<p>עץ חיפוש בינארי כך שכמות הרמות של לוגריתמית ביחס לכמות האיברים בעץ</p> <ul style="list-style-type: none"> <li>AVL</li> <li>עץ אדום שחור</li> </ul> <p><a href="#">הוכחות עבור גובה עץ לוגריתמי</a></p> <p>היות ועץ AVL הוא עץ יותר מאוזן מעץ אדום-שחור, נעדיף להשתמש בו אם כמות החיפושים גדולה, אך אם מתבצעים הרבה שינויים כמות הכנסה או מחיקה, יתבצעו הרבה רוטציות ולכן נעדיף עץ אדום שחור שהוא יותר "סלחני"</p>	עץ חיפוש מאוזן
$O(\log n)$	עובר השורש $O(1)$ חיפוש כלשהו $O(n)$	$O(\log n)$ מכניסים לסוף, ואז עושים SwapUp עד לשורש	<ul style="list-style-type: none"> <li>עץ עדיפות (במקום חיפוש נרצה עדיפות לפי אופרטור)</li> </ul> <p><b>הערה:</b> בנייה של עץ ערמה הוא <math>O(n)</math> כי עושים Heapify כחצי מגודל המערך <a href="#">הוכחה</a></p>	<ul style="list-style-type: none"> <li>כל האיברים בעץ ניתנים להשוואה ע"י אופרטור</li> <li>הערך של כל קודקוד קטן או שווה ביחס לאופרטור מהערך של בניו</li> <li>ממומש ע"י מערך כך שהבן השמאלי באינדקס <math>2i + 1</math> והבן הימני באינדקס <math>2i + 2</math> כאשר ראש הערמה באינדקס 0</li> </ul> <p>מבנה נתונים אשר תומך בעדיפות לפי אופרטור</p>	ערמה
<p><b>Direct Access Table:</b> <math>O(1)</math></p> <p><b>Open Addressing:</b> <math>O(n)</math></p> <p><b>Separate Channing:</b> תלוי מימוש, מערך של רשימות מקושרות <math>O(\text{list length})</math> בעוד שמערך של עצי חיפוש בינאריים <math>O(\log n)</math></p> <p>עמסון <math>\text{Load Factor}</math> ופונקציית גיבוב טובה,</p>	<p><b>Direct Access Table:</b> <math>O(1)</math></p> <p><b>Open Addressing:</b> <math>O(n)</math></p> <p><b>Separate Channing:</b> תלוי מימוש, מערך של רשימות מקושרות <math>O(\text{list length})</math> בעוד שמערך של עצי חיפוש</p>	<p><b>Direct Access Table:</b> <math>O(1)</math></p> <p><b>Open Addressing:</b> <math>O(n)</math></p> <p><b>Separate Channing:</b> תלוי מימוש, מערך של רשימות מקושרות <math>O(\text{list length})</math> בעוד שמערך של עצי חיפוש בינאריים <math>O(\log n)</math></p>	<p>על מנת לאפשר חיפוש, הוספה ומחיקה בסיבוכיות קבועה, ניתן להשתמש ב-Direct Access Table מנת לקבל סיבוכיות זו ע"י <math>O( U )</math> מקום כאשר <math>U</math> זה מרחב המפתחות האפשריים, או לצמצם את הזיכרון ולפתור בעיות התנגשות (Collision)</p> <p>אך, משלמים על זה בכך שמבנים דינמיים הם יעילים מבחינת זיכרון, בעוד שטבלת גיבוב היא בזבזנית, בנוסף אם אנחנו רוצים להתעסק עם מידע ממון, מציאת איברים סמוכים, חיפוש תווים,</p>	<ul style="list-style-type: none"> <li>לכל ערך יש מפתח</li> <li>בעל פונקציית גיבוב</li> </ul> <p><b>פונקציית גיבוב טובה:</b></p> <ol style="list-style-type: none"> <li>כמות הפלטים קטנה מכמות הקלטים האפשריים</li> <li>כמות התנגשויות צריכה להיות קטנה ככל שאפשר</li> <li>קלה לחישוב</li> <li>דטרמיניסטית</li> <li>תשתמש בכל המידע שיש למפתח</li> <li>מפזרת אחיד</li> <li>חד כיוונית לעיתים</li> </ol> <ul style="list-style-type: none"> <li>פותר התנגשויות</li> </ul>	טבלת גיבוב

<p>נוכל לקבל <math>O(1)</math> <b>Amortized</b></p>	<p>בינאריים <math>O(\log n)</math></p> <p>עם <math>Load</math> <math>Factor</math> ופונקציית גיבוב טובה, נוכל לקבל <math>O(1)</math> <b>Amortized</b></p>	<p>עם <math>Load</math> <math>Factor</math> ופונקציית גיבוב טובה, נוכל לקבל <math>O(1)</math> <b>Amortized</b></p>	<p>להציג מבנה נתונים ממויין אז נעדיף להשתמש בעצים לדוגמא</p>	<p><b>לדוגמא:</b></p> <ul style="list-style-type: none"> <li>• Open Addressing (מציאת מקום חדש לאובייקט)</li> <li>• Separate Chaining (מערך של רשימות מקשורות, עצי חיפוש, וכו')</li> </ul>	
---	---	--	--	--	--