# מיונים השוואתיים

## מיון בחירה Selection Sort

באלגוריתם זה עוברים בלולאה על המערך לפי מספר איברי המערך. בכל איטרציה i נחפש את האיבר הקטן ביותר בעזרת פונקציית עזר, ונשים אותו במקום ה-i במערך. לדוגמה:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 1 | 9 | 2 | 6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 6 | 9 | 2 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 5 | 6 | 3 | 2 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 5 | 6 | 3 | 2 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 6 | 5 | 3 | 2 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 6 | 5 | 3 | 2 | 1 |

**Selection Sort - pseudo-code**

**SelectionSort(A)** for i=1 to A.length //O(n)  
 minIndex = Smallest\_Index(i, A) //O(n)  
 A[i]↔A[minIndex] //O(1)  
 end-for  
**end-Selection-Sort**

**Smallest\_Index(start, A)** minIndex = start // O(1)  
 for i=start to A.length //O(n)  
 if (A[i]<A[minIndex])  
 minIndex = i //O(1)  
 end-if  
 end-for  
 return minIndex  
**end-smallestIndex**

**סיבוכיות - Selection Sort**

באלגוריתם זה אנו עוברים בלולאה חיצונית על כל איברי המערך באופן סדרתי (n)O. בתוך לולאה זו אנו עוברים שוב על המערך באופן סדרתי בפונקציית עזר, אך הפעם בכל לולאה מספר איברי המערך קטן באחד. לפיכך מספר הפעולות הוא:

## מיון בועות Bubble Sort

מיון זה דומה מאוד לקודמו בכך שבכל איטרציה אנו מגדילים את השמורה הממוינת לפחות באיבר אחד, אולם כאן אנו עושים זאת מסוף המערך לתחילתו. בכל לולאה אנו משווים בין כל שני איברים שכנים, ואם הראשון גדול מהבא מחליפים ביניהם, אחרת לא עושים כלום. ואז קופצים לאיבר הבא. כך בכל איטרציה אנו "מבעבעים" את האיבר הגדול ביותר כלפי מעלה. מיון זה דורש הרבה החלפות, שלא כמו במיון הקודם של Selection Sort שבכל איטרציה החלפנו לכל היותר פעם אחד. באיור למטה ניתן לראות דוגמה לפעולה של האלגוריתם בלולאה אחת.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 1 | 9 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 1 | 9 | 2 | 6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 9 | 1 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 1 | 9 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 9 | 1 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 9 | 5 | 1 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 9 | 3 | 5 | 1 | 6 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 9 | 5 | 1 | 6 | 2 |

**Bubble Sort – pseudo-code**

**Bubble-Sort(A)** flag = true  
 i=1   
 while (i<=length(A) and flag=true)  
 flag = false  
for j=1 to length(A)-i if (A[j]>A[j+1]) then  
 flag = true   
 A[j]↔A[j+1]   
 end-if  
 end-for  
 i = i+1 //i++  
 end-while  
**end-Bubble-Sort**

**סיבוכיות - Bubble Sort:**

באלגוריתם זה אנו עוברים בלולאה חיצונית על כל איברי המערך באופן סדרתי, בתוך לולאה זו אנו עוברים שוב על המערך באופן סדרתי, אך הפעם בכל לולאה מספר איברי המערך קטן באחד. לכן הסיבוכיות היא:

## מיון הכנסה Insertion Sort

מיון זה עובר בלולאה חיצונית על כל איברי המערך באופן סדרתי, החל מהאיבר השני. בכל סוף לולאה אנו מגדילים את השמורה הממוינת באיבר אחד, מההתחלה לסוף. האלגוריתם עובד כך שבכל איטרציה i של הלולאה אנו לוקחים את האיבר במקום ה-i ומעבירים אותו למקומו הנכון ביחס לכל המספרים שקדמו לו. אנו עושים זאת על-ידי החלפתו באיבר הקודם לו, עד שמגיעים לאיבר הקטן ממנו. לדוגמה:

איטרציה ראשונה:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **1** | **9** | **2** | **6** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **1** | **9** | **6** | **2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **1** | **9** | **6** | **2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **9** | **1** | **6** | **2** |

איטרציה שלישית:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **9** | **6** | **1** | **2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **9** | **1** | **6** | **2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **9** | **6** | **1** | **2** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **3** | **5** | **9** | **6** | **2** | **1** |

**Insertion Sort – pseudo-code**

**Insertion-Sort(A)** for i=1 to length(A)  
 j = i  
 while(j>0 and A[j-1]>A[j])  
 A[j] A[j-1]  
 j = j-1 //j--  
 end-while  
 end-for  
**end Insertion-Sort**

**סיבוכיות - Insertion Sort:**

באלגוריתם זה אנו עוברים בלולאה חיצונית על כל איברי המערך באופן סדרתי. בתוך לולאה זו אנו עוברים שוב על המערך באופן סדרתי, אך הפעם בכל לולאה מספר איברי המערך קטן באחד. אמנם מכיוון שבאלגוריתם זה הכניסה ללולאה הפנימית תלוי בהתקיימות התנאי "כל עוד האיבר קטן מקודמו" יתכן שלא ניכנס ללולאה. לכן נחלק לשני מקרים:

במקרה הגרוע ביותר בו המערך ממוין בסדר הפוך (מהגבוה לנמוך) הסיבוכיות הינה:

במקרה הטוב ביותר בו המערך ממוין הסיבוכיות היא: .

**הוכחת נכונות Insertion Sort:**

באינדוקציה על i. נוכיח כי בכניסה לאיטרציה i כלשהי, i האיברים הראשונים ממוינים.

בסיס האינדוקציה: כאשר i=1 אזי בוודאי שאיבר אחד נחשב ממוין.

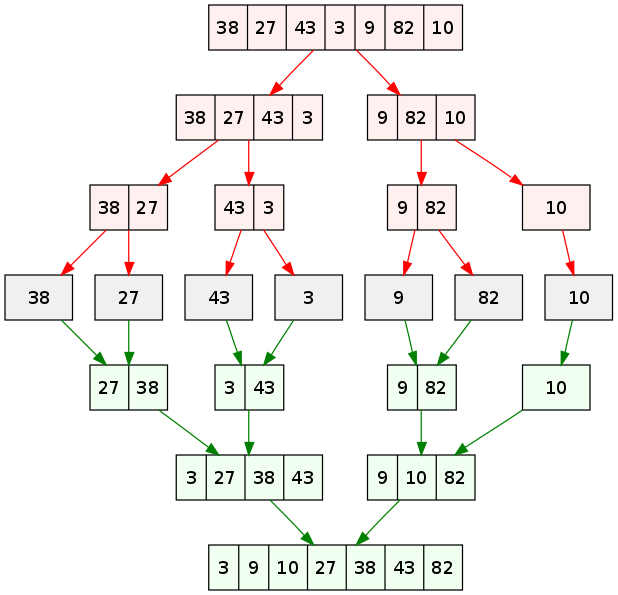
הנחת האינדוקציה: נניח כי עבור i כלשהו *, כלומר* i *האיברים הראשונים ממוינים.*

*שלב האינדוקציה: באיטרציה* i+1 *האלגוריתם לוקח את איבר* A[i+1] *וכל עוד האיבר שלפניו נמוך ממנו הוא עושה החלפה ביניהם. מכיוון ש-*i *האיברים הראשונים ממוינים, נקבל בסוף האיטרציה* i+1 *איברים ממוינים. מ.ש.ל.*

## מיון מיזוג Merge Sort

מיון מיזוג הוא אלגוריתם מיון רקורסיבי קל למימוש המתבסס על קלות מיזוגם של **מערכים ממוינים**. אלגוריתם זה פועל בשיטת הפרד ומשול. הוא מחלק את המערך שהוא מקבל לשני חלקים על ידי קריאה רקורסיבית לעצמו על החצי הראשון של המערך ועל החצי השני. כך המערך מתחלק שוב ושוב לשנים עד שמתקיים התנאי שהמערך בגודל של אחד (בסיס הרקורסיה).

כעת האלגוריתם חוזר להרצה האחרונה של הפונקציה ומפעיל פונקציה נוספת merge שמקבלת שני מערכים ממוינים וממזגת אותם למערך אחד ממוין. בהתחלה תנאי זה טריוויאלי שהרי מערך עם איבר אחד הוא בוודאי ממוין. כך האלגוריתם כל פעם חוזר להרצה האחרונה של הפונקציה הרקורסיבית, אלא שכעת הוא מקבל שני מערכים ממוינים וממזג אותם. כך האלגוריתם ממשיך עד שהוא מגיע לפונקציה הראשונה שנפתחה ומתקבל מערך ממוין.



הפונקציה merge כאמור מקבלת שני מערכים ממוינים n1 ו-n2 וממזגת אותם למערך ממוין. היא קודם יוצרת מערך חדש res בגודל שני המערכים, ואז עוברת באופן סדרתי על אינדקס המערך החדש k באמצעות שלושה לולאות while:

1. ללולאה הראשונה נכנסים רק כאשר עוד לא העברנו את כל הערכים **משני** המערכים אל המערך החדש. בלולאה זו הפונקציה בודקת איזה איבר יותר קטן משני המערכים, מכניסה אותו למערך החדש, ומקדמת רק את האינדקס של מערך זה והמערך החדש.
2. ללולאה השנייה נכנסים רק במקרה שכבר העברנו את כל הערכים ב-n2 למערך החדש אך נותרו עוד ערכים ב-n1. במצב זה הפונקציה מעתיקה את כל הערכים שנותרו ב-n1 למערך החדש.
3. ללולאה השלישית נכנסים במקרה ההפוך, שבו כבר העברנו את כל הערכים ב-n1 למערך החדש אך נותרו עוד ערכים ב-n2. ובאופן זהה הפונקציה תעתיק את כל הערכים שנותרו ב-n2 למערך החדש.

**Merge Sort - pseudo-code**

**mergeSort(A)**

n = A.length

mergeSort(A, 0, n) //overloading

end-mergeSort

**mergeSort(A, low, high)**

if (high - low <= 1)

return //stop function

mid = (low + high)/2

mergeSort(A, low, mid)

mergeSort(A, mid, high)

merge(A[low, mid**)**, A[mid, high**)**) //not include high

end-mergeSort

**merge(A1, A2)**

i=0, j=0, k=0

n1 = A1.length

n2 = A2.length

res[n1 + n2] //define array-result of merge

while(i<n1 AND j<n2)

if (A1[i] < A2[j])

res[k] = A1[i]

i++

else

res[k] = A2[j]

j++

end-if

k++

end-while

while (i<n1)

res[k] = A1[i]

i++

k++

while (j<n2)

res[k] = A2[j]

j++

k++

return res

**end-merge**

**סיבוכיות - Merge Sort:**

בכל קריאה לפונקציה mergeSort אנו מבצעים פעולה רקורסיבית של חלוקת המערך לשניים, לכן עלינו לחשב כמה פעמים יש לחלק את המערך לשניים עד שמגיעים לאיבר אחד:

בנוסף לאחר כל חלוקה קוראים לפונקציה merge שעוברת באופן סדרתי על כל איברי המערך החדש שהיא יוצרת, מכניסה כל פעם איבר אחד, משווה שני איברים, ומקדמת שני איברים. לכן הסיבוכיות שלה הוא:

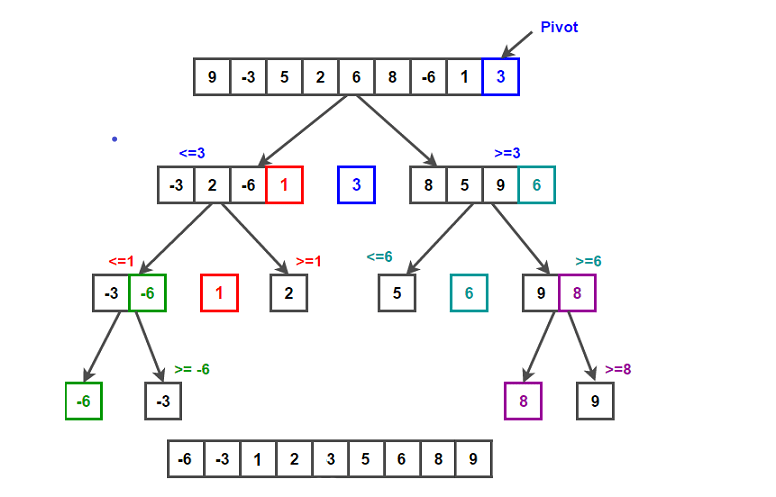
לכן הסיבוכיות של אלגוריתם Merge Sort הוא:

***הערה:*** *הסיבוכיות הקטנה ביותר לאלגוריתם השוואתי היא , כמו שניתן לראות במיון מיזוג ובמיון מהיר.*

## *מיון מהיר* Quick Sort

מיון מהיר הוא אלגוריתם מיון השוואתי, רקורסיבי, המתבסס על עקרון "הפרד ומשול". האלגוריתם מפעיל על המערך את הפונקציה partition, הבוחרת אינדקס של איבר כלשהו במערך שיקרא "איבר ציר" (נקרא גם pivot), ואז מחלקת את המערך לשתי קבוצות, כאשר איבר הציר נמצא באמצע. כל האיברים הגדולים מאיבר הציר יהיו באינדקסים מימינו, וכל האיברים הקטנים או שווים לאיבר הציר יהיו באינדקסים משמאלו. נשים לב כי גודל הקבוצות אינו בהכרח שווה וגם האיברים בכל קבוצה אינם ממוינים, אולם איבר הציר שנבחר נמצא בדיוק במקומו אילו המערך היה ממוין.

כך מפעילים ברקורסיה את הפונקציה הראשית quickSort על כל אחד משני הקבוצות, עד שמתקבל תנאי העצירה (בסיס הרקורסיה) האומר כי גודל הקבוצות הוא אחד. מכיוון שבכל קריאה לפונקציה partition אנו מסדרים לפחות איבר אחד, הרי שלאחר כל הקריאות כל איבר יהיה בדיוק במקומו והמערך שיתקבל הוא מערך ממוין.

יש כל מיני שיטות שונות לבחירת איבר הציר: יש כאלו שבוחרים את האיבר הראשון, יש כאלו שבוחרים את האיבר האמצעי, ויש באקראי. אנו בדוגמה נבחר את האיבר הראשון, אמנם זה לא כל כך משנה.

**Quick Sort - pseudo-code**

**quickSort(A)**

n = A.length-1

quickSort(A, 0, n) //overloading

end-quickSort

**quickSort(A, low, high)**

if (low < high){

pivot = partition(A, low, high)

quickSort (A, low, pivot-1)

quickSort (A, pivot+1, high)

else

return //stop function

end-quickSort

**partition(A, low, high)**

pivot = low

low++

while (low <= high)

if (A[low] <= A[pivot])

low++

else if (A[high] > A[pivot])

high--

else

A[low] A[high] //swap

end-if

end-while

A[low] A[high]

return high //high is the new pivot

end-partition

**סיבוכיות - Quick Sort**

בכל קריאה לפונקציה quickSort אנו מחלקים את המערך לשניים, על כן עלינו לחשב כמה פעמים יש לחלק את גודל המערך לשניים עד שנגיע לאיבר אחד.

אמנם חלוקה זו נכונה רק כאשר בכל חלוקה מחלקים את המערך לשני חלקים שווים. אם מחלקים אותו לשליש ושני שליש, או בכל חלוקה אחרת, מובן שהיעילות תרד. אי אפשר לדעת מראש כיצד תתבצע החלוקה, מכיוון שהיא נקבעת על פי ערכו של איבר הציר. למרבה הפלא, דווקא מיון של מערך שהוא כבר ממוין יהיה המקרה הגרוע ביותר, כי בכל פעם שנחלק את המערך יהיה האיבר הראשון ובחלקו השני יהיו שאר האיברים. כך יהיו לנו n-1 חלוקות. לכן מספר הקריאות ל- quickSort יהיה במקרה הטוב , ובמקרה הגרוע *.*

בנוסף עבור כל קריאה ל-quickSort אנו קוראים גם לפונקציה partition, שעוברת באופן סדרתי על כל איברי המערך שהיא מקבלת ובודקת אם גדולים או קטנים מאיבר הציר. לכן הסיבוכיות שלה היא O(n).

לסיכום, הסיבוכיות של אלגוריתם Quick Sort היא  **במקרה הטוב, אך במקרה הגרוע הסיבוכיות היא** . בפועל, אלגוריתם מיון מהיר נחשב לאלגוריתם המיון ההשוואתי **היעיל ביותר הידוע**, זאת מאחר שהסיכוי למקרה הגרוע הוא מאוד נמוך.

## מיון ערימה - Heap Sort

מיון ערימה הוא אלגוריתם מיון השוואתי המבוסס על הפונקציות המרכזיות במבנה נתונים ערימה בינארית (הסברנו אותם בקובץ של מבנה נתונים - Binary Heap). מיון ערימה הוא סוג של מיון בחירה, המשתמש בכמות קטנה וקבועה של שטח אחסון. כדי למיין את המערך מהמספר הקטן לגדול נשתמש בערימת מקסימום (Max Heap), וכדי למיין את המערך מהגדול לקטן נשתמש בערימת מינימום (Min Heap). מכיוון שכבר הסברנו על ערימת מינימום גם כאן נמשיך במיון מהגדול לקטן, אמנם אותם עקרונות בדיוק תופסים גם במיון מהקטן לגדול.

נפעיל על המערך את פונקציית buildMinHeap המסדרת את המערך לפי תכונת ערימה בינארית, שהיא ממוינת בקירוב אך לא לגמרי, משום שבערימה בינארית אין חשיבות לסדר הבנים של כל איבר. אמנם התכונה המרכזית שחשובה לנו כאן בערימה מינימום בינארית היא שהשורש הוא בודאי האיבר הכי קטן בעץ. כעת נבצע את הפעולות הבאות בלולאת for הרצה מ- ועד בקפיצות של -1:

1. נחליף בין האיבר הראשון לאיבר האחרון במערך *. נקבל שהאיבר הכי קטן בוודאי נמצא בסוף המערך.*
2. *נפעיל פונקציית* minHeapify *על האיבר הראשון כך שהוא ישורשר למקום המתאים לו במערך. וכעת, מלבד האיבר בסוף המערך, האיבר הכי קטן נמצא שוב בתחילת המערך .*

*נחזור על שני פעולות אלו בלולאה, כך שבכל איטרציה אנו מכניסים את האיבר הכי קטן בסוף המערך ומקטינים את המערך ב-1. לאחר סיום הלולאה נקבל מערך ממוין מהמספר הגדול לקטן.*

**Heap Sort - pseudo-code**

**heapSort(A)**

buildMinHeap(A)

n = A.length

for i=n-1 to i=0

A[0] A[i] //swap

minHeapify(A, 0, i)

**end-heapSort**

**סיבוכיות - Heap Sort**

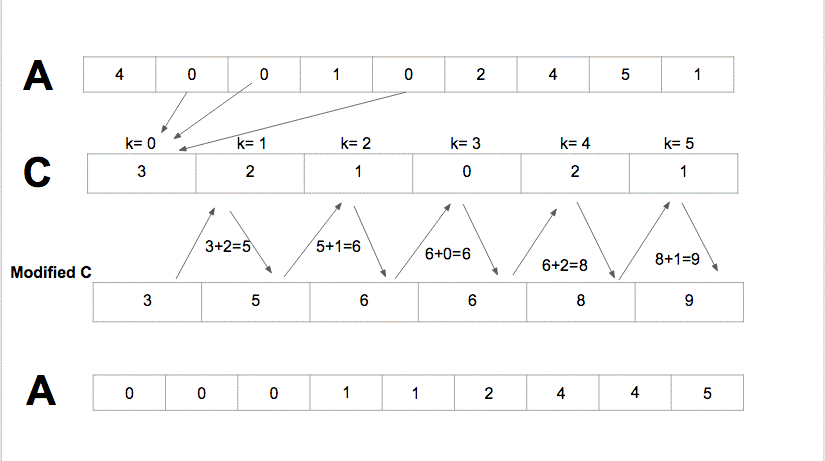
הסיבוכיות של buildMinHeap היא . הסיבוכיות של minHeapify היא , ומכיוון שאנו מפעילים את minHeapify סך הכל n פעמים החישוב הכולל הוא:

נשים לב שחישוב זה מתאים גם למקרה הכי גרוע. הסיבוכיות של המיון Heap Sort הוא *. בפועל* Heap Sort *הוא איטי במקצת מ-*Quick Sort*, אמנם הסיבוכיות שלו קבועה גם במקרה הכי גרוע.*

# מיונים לא-השוואתיים

## *מיון מנייה* Counting Sort

מיון מניה או מיון ספירה הוא אלגוריתם מיון **לא השוואתי** עבור מספרים שלמים, המתבסס על העובדה שהמספרים נמצאים בטווח חסום כדי לבצע את המיון בזמן מהיר, יותר מזה שמסוגלים לו אלגוריתמי המיון הכלליים.

האלגוריתם קודם מוצא את המספר המקסימלי והמינימלי במערך ומחשב את הטווח ביניהם range. לאחר מכן בונה מערך עזר בגודל זה, בכל תא i במערך העזר נכניס את מספר הפעמים שהאיבר i מופיע במערך הראשון. משנים את מערך העזר כך שכל תא יסכום גם את התא הקודם לו. לאחר מכן, נעביר את כל האיברים בצורה מסודרת ממערך העזר למערך הראשון.

**Counting Sort – pseudo-code**

**Counting-Sort(A)**

max = min = A[0]

for i=1 to length(A)

if(A[i]>max) than

max = A[i]

else if (A[i]<min) than

min = A[i]

end-for

range = max-min+1

count[range]

for i=0 to A.length

index = A[i]- min

count[index]++

end-for

for i=1 to range

count[i] = count[i]+count[i-1]

end for

i=0

for j=0 to range

while (i<count[j])

A[i] = j +min

i++

end-for

end-counting-sort

**סיבוכיות - Counting Sort:**

באלגוריתם זה בשני הלולאה הראשונות אנו עוברים בכל לולאה n פעמים על המערך. בלולאה השלישית range פעמים. ובלולאה האחרונה עוברים על לולאה ה-for range פעמים, ועל לולאת ה-while בכל לולאות ה-for סך הכל n פעמים. כך שהסיבוכיות היא:

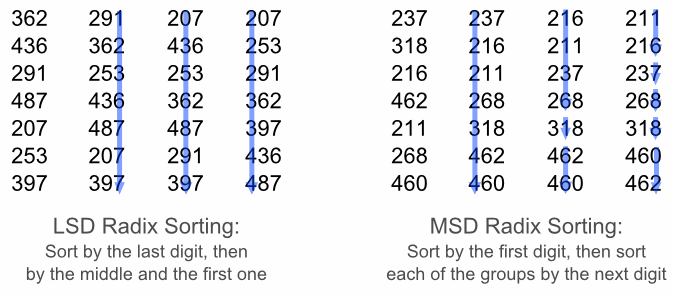
המקרה הטוב ביותר הוא כאשר הטווח בסדר גודל של הקלט, כלומר , שאז הסיבוכיות היא *. לכן, כאשר טווח המספרים הוא בגודל הקלט או פחות מכך נעדיף להשתמש באלגוריתם זה על פני המיונים ההשוואתיים, שבהם הסיבוכיות הטובה ביותר היא .*

*אפשר ליישם את האלגוריתם בסיבוכיות גם כאשר טווח הערכים הוא , כלומר לא ליניארי לגודל הקלט. למשל, עבור קלט בגודל* n *וטווח של , ממירים את המספרים לבסיס* n*, כך שמספר הספרות המקסימלי הוא 2, ומספר התאים במערך* count *הוא בסך הכל* n, *ולכן* range *אינו אלא* n. *ואז מיישמים מיון בסיס.*

## *מיון בסיס* Radix Sort

הוא אלגוריתם מיון **יציב**, לא השוואתי, עבור מספרים שלמים. האלגוריתם עובר בכל איטרציה רק על ספרה אחת של כל המספרים במערך, ומחלק אותם לקבוצות לפי גודל ספרה זו בלבד. לאחר מכן הוא עובר על הספרה הבאה ומחלק את המספרים לקבוצות באותה הצורה, וכן הלאה עד הספרה האחרונה. העיקרון החשוב באלגוריתם זה הוא שהמיון הוא יציב, כלומר מספרים בעלי אותו ערך בספרה כלשהי יישארו באותו סדר גם לאחר המיון, מה שאומר שכל המיונים בכל הספרות הקודמות נשמר. לאחר כל מיוני ספרות אלו נקבל מערך ממוין.

יש שני דרכים ליישם אלגוריתם זה:

1. מיון המספרים מהספרה הכי נמוכה אל הספרה הכי גבוהה - LSD Radix Sort (Least Significant Digit).
2. מיון המספרים מהספרה הכי גבוהה אל הספרה הכי נמוכה - MSD Radix Sort (Most Significant Digit).

האלגוריתם שנכתוב מתייחס למספרים בבסיס 10, אמנם בקלות ניתן להתאימו גם למספרים בבסיסים אחרים על ידי שינוי המשתנה base. האלגוריתם מוצא את המספר הגדול ביותר במערך שהוא עם הכי הרבה ספרות, ואז רצים בלולאת while כמספר הספרות שיש במספר המקסימלי. לאחר מכן האלגוריתם בונה מערך עזר temp בגודל n, שבו בכל איטרציה של לולאת ה-while אנו נמיין את המספרים לפי הספרה שאנו בוחנים, בסוף נעשה העתקה ממערך זה אל המערך המרכזי.

בתוך לולאת ה-while בונים מערך עזר נוסף counter בגודל base, בו בכל אינדקס אנו נספור כמה מספרים יש במערך שהספרה שאנו בוחנים שווה לאינדקס זה. לאחר מכן נשנה את מערך העזר counter כך שכל תא יסכום גם את התא הקודם לו.

כעת אנו עוברים על סיפרה מסוימת בכל המספרים, ומכניסים את המספר לפי מיקום הספרה שלו אל מערך העזר temp עם הקפדה על מיון יציב. לאחר מיון זה מעתיקים את כל הערכים ממערך העזר אל המערך המרכזי. ולבסוף מכפילים את exp ב-base, כך שבפעם הבאה שנחלק את האיבר המקסימלי ב-exp נעבור לספרה הבאה. לולאת ה-while תעבור בדרך זו על כל הספרות, ובסיום האלגוריתם נקבל מערך ממוין.

האלגוריתם הבא מתייחס למקרה של מיון בסיס מהספרה הנמוכה.

**Radix Sort – pseudo-code**

**radixSort(A)**

n = A.length

base = 10, exp = 1

temp[n] //helping array

max = A[0]

for i = 1 to n-1 ***//O(n)***

if (A[i] > max)

max = A[i]

end-for

while ((max / exp) > 0*)* ***//***

counter[base] //another helping array

for i = 0 to n-1 ***//O(n)***

j = (A[i] / exp) % base

counter[j]++

end-for

for i = 1 to base ***//O(base)***

counter[i] = counter[i] + counter[i-1]

end-for

for i = n – 1 to 0 step -1 ***//O(n)***

j = (A[i] / exp) % base

counter [j]--

index = counter[j]

temp[index] = A[i]

end-for

for i = 0 to n-1 ***//O(n)***

A[i] = temp[i]

end-for

exp = exp \* base

end-while

end-radixSort

**סיבוכיות - Radix Sort:**

יהי n מספר האיברים במערך, k מספר הספרות באיבר הכי גדול במערך (max), ויהי base הבסיס של המספרים במערך. נשים לב כי בלולאה for הראשונה רצים n פעמים למציאת איבר מקסימלי. על לולאת while רצים כמספר הפעמים שצריך לחלק את max ב-base עד שנקבל ביטוי הקטן מ-1, לכן נחשב כמה פעמים צריך לחלק את max ב-base עד שנקבל 1. ביטוי זה שווה למספר הספרות ב-max שהגדרנו אותו k.

בתוך לולאת ה-while ישנם ארבעה לולאת for שרצים שלוש פעמים על n ופעם אחת על base. לכן חישוב מספר הפעולות הוא:

מכיוון שבסיס המספרים הוא כמעט תמיד שווה ל-10, ובמקרים חריגים 16, ניתן להתעלם מערך זה. לכן נקבל שהסיבוכיות של מיון בסיס הינה: *.*

*סיבוכיות זו אינה תלויה רק בגודל הקלט, ולכן היעילות שלה לעומת אלגוריתמי השוואה אחרים נעשה בדרך הבאה: כאשר מספר הספרות במספר המקסימלי* k *הוא בסדר גודל של גודל הקלט* n*, נקבל שהסיבוכיות הינה שזוהי סיבוכיות גרועה מאוד (כמו של מיון בועות), ולכן לא נרצה להשתמש במיון זה. אמנם כאשר* k *זניח ביחס ל-*n*, נקבל שהסיבוכיות הינה שזוהי סיבוכיות מהירה ביותר, ולכן נעדיף להשתמש במקרה זה במיון בסיס.*

**הוכחת נכונות Radix Sort:**

באינדוקציה על מספר הספרות באיבר המקסימלי במערך, נסמן ערך זה בתור k.

בסיס האינדוקציה: כאשר k=1 אזי כל מספר מורכב מספרה אחת בלבד. בוודאי שמיון על פי הספרה היחידה ממיין את המערך, לכן הטענה מתקיימת.

הנחת האינדוקציה: נניח כי עבור מספר ספרות כלשהו k=n, לאחר הפעלת מיון בסיס נקבל מערך ממוין.

*שלב האינדוקציה: נוכיח שהאלגוריתם ממיין גם מערך שבו* k=n+1. *לפי הנחת האינדוקציה המערך ממוין עבור* n *ספרות מימין, עכשיו ממיינים לפי ספרה שמאלית. נתבונן בשני מספרים* a, b*:*

*כאשר ספרה שמאלית של מספר* a *קטנה מספרה שמאלית של מספר* b*, אזי* b *יופיע לפני* a*, בלא להתחשב ב-*n *ספרות קודמות. וכן ההפך, כאשר ספרה שמאלית של מספר* a *גדולה מספרה שמאלית של* b*, אזי* a *יופיע לפני* b*. ואילו כאשר ספרה שמאלית של* a *שווה לספרה שמאלית של* b*, אז המיון נשמר לפי* n *ספרות קודמות (מיון יציב), לפי הנחת האינדוקציה. לכן כל שני מספרים* a, b *בכל המצבים אכן נמצאים בסדר עולה מקטן לגדול. מ.ש.ל.*

# חיפוש איבר

## חיפוש סדרתי Linear Search

הוא אלגוריתם חיפוש שבו אנו עוברים על כל איברי המערך באופן סדרתי, מאיבר הראשון עד לאחרון. אם האיבר נמצא מחזירים את אינדקס האיבר במערך, אחרת מחזירים -1, כלומר אינדקס שלא קיים המציין שהאיבר כלל לא נמצא במערך.

**Linear Search - pseudo-code**

**linearSearch(A, value)**

n = A.length

for i=0 to n

if(A[i]==value)

return i

end-for

return -1 //value is not in the array

end-linearSearch

**סיבוכיות - Linear Search**

במקרה הגרוע ביותר בו האיבר שאנו מחפשים נמצא באינדקס האחרון במערך, או שהוא כלל אינו נמצא במערך, נעבור על כל האיברים בלולאה. לכן הסיבוכיות של Linear Search היא: .

## חיפוש בינארי Binary Search

במדעי המחשב, "הפרד ומשול" היא פרדיגמת תכנון אלגוריתמים חשובה. היא מבוססת על עקרון הרקורסיה, כלומר לפרק את הבעיה לשתיים או יותר תת בעיות מאותה הצורה (או צורה דומה לה) שוב ושוב, עד שהבעיות הופכות לפשוטות דיין כדי שניתן יהיה לפתור אותן ישירות. לאחר מכן הפתרונות לתת הבעיות משולבים יחד כדי לתת פתרון לבעיה המקורית. שיטה זו היא הבסיס לאלגוריתמים יעילים רבים, בסעיף זה אנו נעסוק באלגוריתם "חיפוש בינארי" שגם הוא מתבסס על עקרון זה:

"חיפוש בינארי" הוא אלגוריתם למציאת אינדקס של איבר מבוקש במערך **ממוין** (בדרך כלל מהקטן לגדול). האלגוריתם קודם מחשב את האיבר האמצעי שבמערך. כעת יש שלוש אופציות: אם האיבר האמצעי הוא האיבר המבוקש, הרי מצאנו את שחיפשנו ונסיים כאן. אם לא, נבדוק אם האיבר המבוקש קטן יותר מהאיבר האמצעי. אם אכן קטן ממנו נבחר את חציו השמאלי של המערך (שבו כל האיברים הקטנים מהאיבר האמצעי). אם האיבר המבוקש גדול יותר, נבחר את חציו הימני של המערך (שבו כל האיברים הגדולים מהאיבר האמצעי). כעת נחזור על כל מה שעשינו עם תת-המערך שבחרנו.

יש שני דרכים לכתוב את הפסאודו-קוד של חיפוש בינארי: בצורה איטרטיבית ובצורה רקורסיבית:

**Binary Search Recursive - pseudo-code**

**binarySearchRecursive(A, value)**

n = A.length-1

If(value<A[0] OR value>A[n]

return -1 //value is not in the array

else

return binarySearchRecursive(A, value, 0, n)

end-binarySearchRecursive

**binarySearchRecursive(A, value, low, high)**

if(low <= high)

mid = (low + high)/2

if(value == A[mid])

return mid //value was found

else if(value < A[mid])

return binarySearchRecursive(A, value, low, mid-1)

else

Return binarySearchRecursive(A, value, mid+1, high)

else

return -1 //value is not in the array

end-binarySearchRecursive

**Binary Search Inductive - pseudo-code**

**binarySearchInducion(A, value)**

n = A.length-1

if (value<A[0] OR value>A[n])

return -1 //value is not in the array

low = 0

high = n

while (low <= high)

mid = (low + high)/2

if (value == A[mid]) //value was found

return mid

else if (value < A[mid])

high = mid-1

else

low = mid+1

end-while

**end-binarySearchInducion**

**סיבוכיות - binarySearch:**

במקרה הגרוע ביותר בכל הפעלה של הפונקציה ברקורסיה או איטרציה של הלולאה, אנו מחלקים את המערך הנבדק לשניים, עד שגודל המערך הנבדק הוא 1. נחשב את מספר החלוקות של n עד שנגיע ל-1 למציאת הסיבוכיות.

לכן הסיבוכיות של binarySearch היא: *.*

## *חיפוש בינארי בין לבין* Binary Search Between

הוא אלגוריתם המקבל מערך **ממוין** A בגודל n, וערך כלשהו value. האלגוריתם יחזיר את האינדקס של המערך המקיים: , כלומר את האינדקס של הערך הגדול הכי קרוב ל-value שנמצא במערך. אם value קטן מהאיבר הראשון יוחזר 0, ואם גדול מהאיבר האחרון יוחזר n.

לדוגמה: עבור המערך אם נכניס את הערך 5 נקבל את הערך 2, עבור הערך 9 נקבל 3, ועבור הערך 20 האלגוריתם יחזיר 5.

במידה והערך שהכנסנו בגבולות המערך, האלגוריתם מחשב את האיבר האמצעי שבמערך. כעת יש שלוש אופציות: אם האיבר האמצעי הוא האיבר המבוקש, הרי מצאנו את שחיפשנו ונחזיר את האינדקס שלו. אם לא, נבדוק אם האיבר המבוקש קטן יותר מהאיבר האמצעי. אם אכן קטן ממנו נבחר את חציו השמאלי של המערך כולל האיבר האמצעי. אם האיבר המבוקש גדול יותר, נבחר את חציו הימני של המערך לא כולל האיבר האמצעי. כעת נחזור על כל מה שעשינו עם תת-המערך שבחרנו.

כאשר תת המערך בגדול של 1, כלומר אינדקס האיבר הגדול בתת המערך שווה לאינדקס האיבר הקטן, נחזיר את אינדקס זה.

**Binary Search Between - pseudo-code**

**binarySearchBetween(A, value)** //A is sorted array

n = A.length

if (value<A[0]) return 0

if (value>A[n-1]) return n

low = 0

high = n-1

while (low <= high)

middle = (low + high)/2

if (low == high) //stop search

return high

else

if (value == A[middle]) //value was found

return middle

else if (value < A[middle])

high = middle

else

low = middle+1

end-if

end-if

end-while

return -1 //value was not found

end-binarySearchBetween

**סיבוכיות - binarySearchBetween**

גם כאן במקרה הגרוע ביותר אנו נחלק את המערך הנבדק שוב ושוב לשני חלקים, ונתמקד רק בחצי הגדול מ-value, עד שגודל המערך הוא אחד. לכן גם כאן נחשב את מספר החלוקות של n עד שנגיע ל-1 למציאת הסיבוכיות.

לכן הסיבוכיות של binarySearchBetween היא: *.*