**ממ"ן 15**

**שאלה 1**

**סעיף א'**

המקרה הטוב של השגרה QuickSort על קלט בגודל 3 מתבצע כאשר הקלט האחרון הינו החציון.

|  |  |  |
| --- | --- | --- |
| C | B | A |

לצורך העניין נתבונן בקלט:

|  |  |  |
| --- | --- | --- |
| B | C | A |

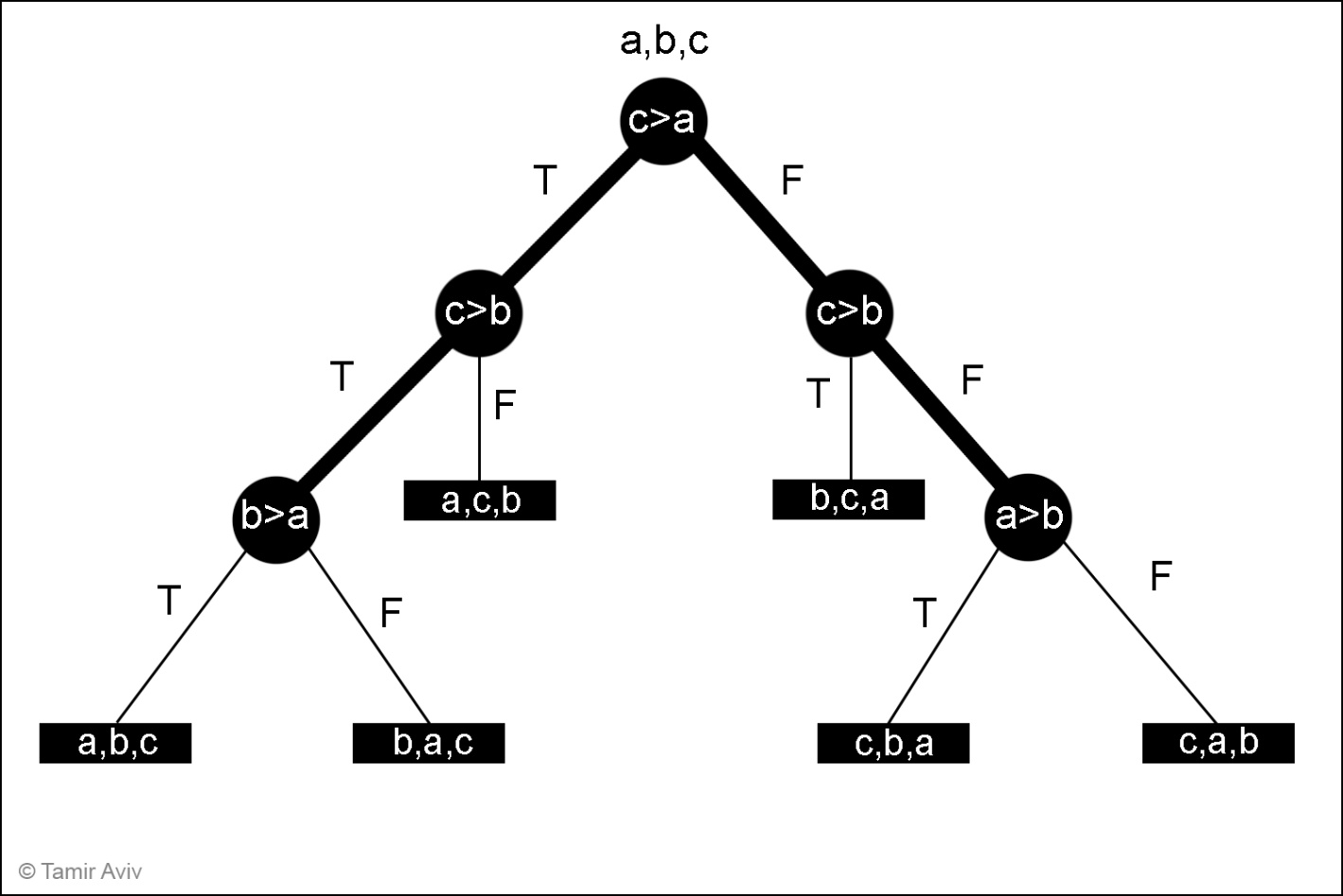
במידה ו C הוא אכן החציון, אז יערכו סה"כ 2 השוואות עד אשר C יונח במקומו:

לאחר מכן השגרה תחלק את הבעיה ל 2 תתי בעיות, בכל תת בעיה יהיה רק ערך אחד ולכן לא יתבצעו עוד השוואות והקלט יוחזר ממויין.

במקרה שבו C אינו החציון אחרי 2 השוואות נקבל שוב חלוקה ל 2 תתי בעיות, אך הפעם בתת בעיה אחת יהיה 0 ערכים ובתת בעיה השניה יהיה 2 ערכים. ועל כן נבצע עוד השוואה אחת (על התת בעיה בגודל 2 ) על מנת למיין את הקלט, כלומר:

סה"כ השוואות במקרה הטוב - 2 השוואות.

סה"כ השוואות במקרה הממוצע והמקרה הגרוע - 3 השוואות.בממוצע מתבצעות 2.666 השוואות.

**סעיף ב'**

גם כאן, ע"פ מבנה העץ, נוכל לראות כי המקרה הטוב מתבצע כאשר C הוא החציון, ובמקרה זה כעבור 2 השוואות הקלט יוחזר ממויין.

בכל שאר המקרים, אנו נאלץ לבצע 3 השוואות על מנת להחזיר את הקלט ממויין, ולכן עץ ההחלטה מתיישב בצורה טובה עם תשובתנו לסעיף הקודם.

**סעיף ג'**

בניית ערימה: בניית ערימה מתבצעת בזמן ליניארי, ועל מערך בגודל 4 תבצע 3-4 פעולות ( לדוגמא: 3 פעולות כאשר המערך ממויין, ו 4 פעולות כאשר המערך ממויין בסדר הפוך ).

החזרת האיברים: החזרת האיברים בסדר ממויין משתמשת בשגרה Max-Heapify על מנת להשאיר את הערמה כערמת מקסימום, אשר עובדת בסיבוכיות של log n. לכן על מערך בגודל 4 השגרה תבצע עוד 3 השוואות ( 2 בהוצאת האיבר הראשון, ו 1 בהוצאת האיבר השני ).

סה"כ השוואות במקרה הטוב - 6 השוואות.

סה"כ השוואות במקרה הגרוע - 7 השוואות.

**סעיף ד'**

נתבונן במיון הכנסה על מערך בגודל 4.

במקרה הגרוע, נעבור על התאים 2,3,4 , ועל כל אחד מהם נבצע השוואה עם כל התאים הקודמים להם. כלומר:

בדרך אחרת - ידוע שמיון הכנסה מבצע לכל היותר:

אם כך, הגענו לכך שבמקרה הגרוע מיון הכנסה מבצע על מערך בגודל 4 לכל היותר 6 השוואות, בניגוד למיון ערימה שמבצע 7 השוואות במקרה הגרוע, ועל כן, מיון ערימה אינו אופטימאלי עבור קלטים בגודל 4.

ובכל זאת, אין כאן סתירה לאופטימאליות מבחינת סיבוכיות זמן של אלגוריתם מיון ערימה, מכיוון שלמיון ערימה ישנו קצב גידול נמוך במספר ההשוואות עבור קלטים גדולים יותר, ואינה מתייחסת ליעילות עבור קלט ספציפי.

(עבור קלטים גדולים מיון ערימה יעבוד בסיבוכיות של במקרה הגרוע בעוד מיון הכנסה יעבוד בסיבוכיות של במקרה הגרוע).

**שאלה 2**

*כל מספר בטווח יכול להיות מוצג כמספר ליניארי בעל n ספרות*

*( שכן המספר הגדול ביותר בטווח הזה: בייצוג ליניארי )*.

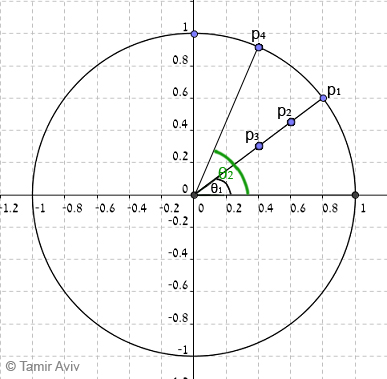
ולכן, נתבונן במיון בסיס בבסיס n. 4-

ע"פ למה 8.4 בעמוד 143, בהינתן n מספרים בני n ספרות, ושלם חיובי , מיון בסיס ממיין את מספרים אלו תוך זמן: .

*ולכן, אם נבחר , נקבל כי:*

מ.ש.ל

**שאלה 3**



נפתח בהסבר קצר:

במעגל היחידה שמולנו P1,P2,P3,P4 נקודות ברביע הראשון.

נמיין את כלל קבוצת הנקודות Pi לפי גודל הזווית.

אך נשים לב שקיימות נקודות עם אותה זווית (לדוגמא P1,P2,P3 ) אשר כולן נמצאות על אותו ישר, במקרה זה לא חשוב מה יהיה סדר הנקודות.

מכיוון שאנו מניחים שהתפלגותן של הנקודות אחידה, נשתמש במיון דלי עם שינויים קלים על מנת לקבל זמן ריצה של .

כאשר נחלק את הרבע מעגל ל n שטחים שווים נקבל שבכל קטע הזווית היא

A - מערך של נקודות.

B - מערך עזר של רשימות מקושרות(דליים).

**BucketPointSort(A)**

**1.** n<-- length[A]

**2.** for i<--1 to n do

**3.** insert A[i] into B[ WhereToInB(A[i]) ]

**4.** for i<--0 to n-1 do

**5.** sort list B[i] with insertion sort (according to the angle()

**6.** concatenate the list B[0],B[1],...,B[n-1] together in order

**WhereToInB(P)**

**1.**

**2.**  return

**שאלה 4**

כדי לממש ד-תור, נשתמש ב "מערך מעגלי" כך שהאיבר אחרי האיבר האחרון במערך הוא בעצם האיבר הראשון.

נשתמש באינדקסים tail[D] ו- head[D] אשר מצביעים על האיבר הראשון והאיבר האחרון בתור.

ונשתמש במשתנה currentSize[D] אשר סופר כמה איברים נמצאים כרגע בתור.

**Push-Right(D,x)**

if currentSize[D] ≠ length[D] then // check that the array is not full

D(tail[D]) <-- x

if tail[D] = length[D] then // if it's the last cell - jump to the first cell

tail[D] <-- 1

else

tail[D] <-- tail[D]+1

currentSize[D]= currentSize[D]+1

else

Explain the user that the queue is full

**Pop-Left(D)**

if currentSize[D]≠ 0 then // check that the array is not empty

x<-- D[head[D]]

if head[D]=length[D] then // if it's the last cell - jump to the first cell

head[D]<-- 1

else

head[D]<-- head[D]+1

currentSize[D]= currentSize[D]-1

return x

else

Explain the user that the queue is empty

**Push-Left(D,x)**

if currentSize[D] ≠ length[D] then // check that the array is not full

if head[D] = 1 then // if it's the first cell - jump to the last cell

head[D]<-- length[D]

D(head[D]) <-- x

else

head[D]<--head[D]-1

D(head[D]) <-- x

currentSize[D]= currentSize[D]+1

else

Explain the user that the queue is full

**Pop-Right(D)**

if currentSize[D] ≠ 0 then // check that the array is not empty

if tail[D]=1 then // if it's the first cell - jump to the last cell

tail[D]<-- length[D]

else

tail[D]<-- tail[D]-1

currentSize[D]= currentSize[D]-1

return tail[D]

else

Explain the user that the queue is empty

נשים לב שכלל השיטות מבצעות מספר קבוע של השוואות, ולכן היעילות של כל שיטה היא O(1).

**שאלה 5**

**סעיף א'**

**Build-s(P)**

**1.** for i<--1 to length[P] do

**2.** conter<--0

**3.**  for j<--i to 1 do

**4.** if(P[j]≤P[i])

**5.** counter<--counter+1

**6.** else

**7.** break the loop

**8.**  S[i] <-- conter

נכונות השגרה

הלולאה הראשונה (1) רצה על כל איברי המערך P. הלולאה השניה (3) בעצם רצה על כל האיברים מאיבר i ומטה עד לאיבר האחרון אשר מקיימת את תנאי (4).

בעצם אנו סופרים מהו התת מערך הגדול ביותר מ k מסויים עד i (כולל!) שבכל הקטע הזה מתקיים התנאי שאנו רוצים, וברגע שעברנו על כל המספרים שמקיימים את התנאי וספרנו, אפשר להכניס את גודלו למיקום המתאים ב S.

השגרה כמובן תמיד תעצור, מכיוון שמדובר בלולאת FOR, ולכל איבר i נוכל למצוא את אורך התת מערך הנדרש, וכמובן תעבור על כל איברי המערך אם צריך ותבצע השוואות ולכן תמיד יוכנס הערך הנכון ל S.

יעילות האלגוריתם

כפי שתיארנו בהסבר, על כל איבר במערך P, אנחנו עושים השוואה עם כל האיברים שלפניו שמקיימים את התנאי.

במקרה הגרוע, כל המערך בסדר עולה, כלומר נקבל מספר פעולות של:

וכידוע, זמן ריצה זה שקול ל:

**סעיף ב'**

A - מחסנית עזר.

**Build-s(P)**

**1.** for i<--1 to length[P] do

**2.** endWhile<-- FALSE

**3.** while endWhile=FALSE do

**4.**  if Stack-Empty(A) then

**5.**  S[i] <-- i

**6.** else

**7.** x<-- Pop(A)

**8.** if P[i]<P[x] then

**9.** Push(A,x)

**10.**  S[i] = i-x

**11.** endWhile=true

**12.** Push(A,i)

נכונות השגרה

הלולאה הראשונה (1) רצה על כל איברי המערך P. בעזרת מחסנית עזר A, אנו בונים מחסנים כך שבראש המחסנית תמיד יהיה האיבר הצמוד ל i (i-1) שנקרא לו X.

* אם מתקיים P[i]<P[x], נחזיר את X למחסנית ונעדכן את הערך של S[i] ל i-x שזה הוא בדיוק ה k המבוקש.
* אם לא מתקיים התנאי הנ"ל, אז אין טעם להחזיר את X מכיוון שלכל איבר בהמשך שגדול מ P[i] בוודאי גדול גם מ P[x] ולכן נשמור רק את i.
* אם המחסנית ריקה - סימן ש P[i] גדול מכל האיברים הקדומים לו במערך, ולכן S[i] יקבל את הערך של המיקום הנ"ל - i.

הלולאת While תעצר בוודאות או עקב כך שהמחסנית ריקה או בגלל שמצאנו איבר המקיים את תנאי (8), ותמיד יוכנס S[i] הערך הנכון, ולכן השגרה תבצע את הבניה בצורה נכונה.

יעילות האלגוריתם

לולאת ה For תעבור על N המשתנים של מערך P.

נבדוק מה קורה עם לולאת ה While. לולאת ה While עוצרת ב 2 מקרים:

1. כאשר הוצאנו איברים עד שהמחסנית הייתה ריקה או עד שמתקיים תנאי (8)

( O(k) - K מספר האיברים ).

1. כאשר מצאנו איבר המקיים את תנאי (8), ואז בסוף הלולאת FOR בעצם הוספנו עוד איבר למחסנית (אחד יותר ממה שהיה קודם), ( O(1) ).

בעצם, המקרה הגרוע מתרחש כאשר המחסנית מלאה ב N-1 איברים, והאיבר האחרון הוא האיבר הגדול ביותר, ובמקרה כזה נצטרך לעשות Pop למחסנית N-1 פעמים.

אבל כדי שמצב זה יקרה, צריך שלכל N-2 האיברים עד עכשיו יתקיים התנאי 8. כלומר ל N-2 איברים, לולאת ה While עבדה ב O(1). ולכן נבצע את הפעולה הזאת של הוצאת N-1 איברים מהמחסנית רק פעם אחת בכל לולאת ה FOR.

למעשה נוכל להסיק שלולאת ה While לא תשפיע לנו על הסיבוכיות זמן ריצה, כי לרוב תתבצע במספר מועט של פעולות, ואילו תתבצע במספר רב של פעולות אז כל האיברים עד אותו איבר התבצעו בוודאות ב O(1) פעולות.

ולכן, נוכל להסיק שזמן הריצה הינו:

תמיר אביב,

305652000