אלגוריתמים 1

מחברת סיכומים מההרצאות של פרופ' לויט ואדים ומתרגולים

אלגוריתמים 1 - מחברת הרצאות

נערך ונכתב על-ידי דור עזריה

2021

ספר זה לא נבדק על ידי מרצה, יתכן שימצאו טעויות.

לסיכומים נוספים שלי במדעי המחשב ומתמטיקה:
https://dorazaria.github.io/

מוזמנים לעקוב אחרי 😊:
LinkedIn: https://www.linkedin.com/in/dor-azaria/
GitHub: https://github.com/DorAzaria

תוכן עניינים

טבלת סיבוכיות

בעיה	סיבוכיות	בעיה	סיבוכיות
בעיית החניה	$O(n^2)$	מציאת מעגל ברשימה מקושרת חד-כיוונית בלי זרוע	$O(n)$
מציאת מעגל ברשימה מקושרת חד-כיוונית עם זרוע	מציאת איבר במעגל - $O(n+m)$ מציאת נקודת תחילת המעגל $O(m)$ מציאת אורך הזרוע $O(m)$ (או $O(1)$במידה וספרנו יחד עם חישוב תחילת המעגל) מציאת אורך המעגל $O(n)$	מציאת מינימום במערך	$O(n)$
מציאת מקסימום במערך	$O(n)$	מציאת מינימום ומקסימום במערך	אפשרות א' - $2n-2$ השוואות אפשרות ב' - $2n-2$ השוואות. אפשרות ג' - $2n-3$ השוואות. אפשרות ד' - $3\frac{n-2}{2}+1+2$ השוואות.
אינדוקציה מול רקורסיה	חישוב עצרת - רקורסיבי - $O(2n)=O(n)$ אינדוקטיבי - $O(n)$ חישוב סדרת פיבונאצ'י - רקורסיבי - $O(2^n)$ אינדוקטיבי - $O(n)+O(2)=O(n)$	מציאת מקסימום ומקסימום במערך	אפשרות א' - $2n-3$ השוואות. אפשרות ב' - $2n-3$ השוואות אפשרות ג' - נעבור על המערך בזוגות - $3\frac{n-2}{2}+1+2=\frac{3n}{2}-3+3=\frac{3n}{2}$ השוואות. אפשרות ד' -אלגוריתם אופטימלי - $n-1+log(n)-1=n+log(n)-2$ השוואות.
משחק המספרים	אפשרות א' - אלגוריתם חמדני אפשרות ב' - זוגי או אי-זוגי אפשרות ג' - אדפטיבית אפשרות ד' - חיפוש שלם	LCS-תת-המחרוזת המשותפת הארוכה ביותר	אפשרות א' - אלגוריתם חמדני - $O(n\cdot m)$ אפשרות ב' - אלגוריתם חמדני משופר - $O(n+m)+O(min(n,m))$ אפשרות ג' - חיפוש שלם - $O(2^{n+m}\cdot min(n,m))$ אפשרות ד' עבור LCS - תכנות דינאמי - $O(n\cdot m)$
תת-הסדרה העולה - LIS הארוכה ביותר	אפשרות א' - אלגוריתם חמדני - $O(n)$ אפשרות ב' - אלגוריתם חמדני משופר - $O(n^2)$ אפשרות ג' - אלגוריתם באמצעות LCS $O(n^2+nlogn)$ אפשרות ד' - אלגוריתם באמצעות חיפוש שלם - $O(2^n\cdot n)$	בעיית המטוס	מילוי המטריצה + מספר מסלולים מינימלים לכל קודקוד- $O(n\cdot m)$ החזרת המסלול הזול ביותר $O(n+m)$
רצף אחדות במערך	אפשרות א' - אלגוריתם חמדני $O(n)$ אפשרות ב' - חמדני משופר $O(n)$ אפשרות ג' - תכנות דינאמי $O(n)$	מטריצת אחדות	אפשרות א': $O(4\cdot n^2)$ אפשרות ב': $O(n^2)$
כדור הזכוכית	למקרה הכללי: $O(b\cdot n^2)$	פיבונאצ'י עם מטריצות	רקורסיבי - $O(log\ n)$

משחק הניחושים (אליס ובוב)

תיאור הבעיה

- אליס ובוב משחקים משחק, כל אחד מטיל מטבע בחדר נפרד ומנחש מה הייתה הטלת חברו.
- הם חברי צוות ולכן או שהם ינצחו יחד או שהם יכשלו יחד.
- לפני שהמשחק מתחיל, הם יכולים לדבר אחד עם השני ולהסכים על אסטרגיה.
- אם לפחות אחד מהניחושים נכון- אליס ובוב ניצחו כצוות אחרת- נכשל.

יש 4 אסטרגיות שיובילו את אליס ובוס לניצחון:

אפשרות ראשונה - 50% הצלחה - ½

כל אחד אומר את מה שיצא לו **או** כל אחד אומר את ההפך ממה שיצא לו.

בוב	אליס	מה שיצא	ההפך ממה שיצא
0	1	✘	✔
0	0	✔	✘
1	0	✘	✔
1	1	✔	✘

אפשרות שניה - 75% הצלחה - ¾

אליס תמיד אומרת 0 **וגם** בוב תמיד אומר 1.

בוב	אליס	בוב 1	אליס 0	סה"כ
0	1	✘	✘	✘
0	0	✘	✔	✔
1	0	✔	✔	✔
1	1	✔	✘	✔

(זה לא משנה אם אחד טעה ואחד צדק, העיקר שאחד מהם צדק במשהו וסה"כ הם מהווים הצלחה משותפת).

אפשרות שלישית - 75% הצלחה - ¾

אליס תמיד אומרת 1 וגם בוב תמיד אומר 1 (כנ"ל לגבי 0)

בוב	אליס	בוב 1	אליס 1	סה"כ
0	1	✘	✔	✔
0	0	✘	✘	✘
1	0	✔	✘	✔
1	1	✔	✔	✔

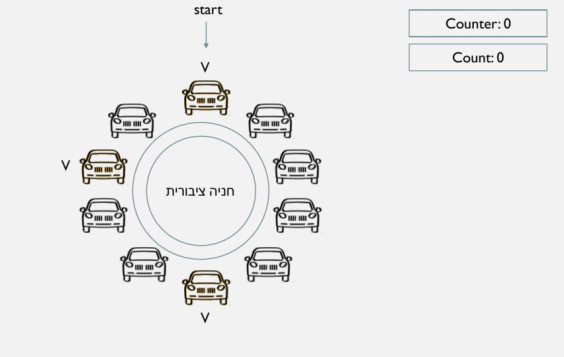
אפשרות רביעית - 100% הצלחה

אליס תמיד אומרת את מה שהיא קיבלה ובוב תמיד משקר ואומר את מה שהוא לא קיבל.

אליס	בוב	אליס	בוב(משקר)	סה"כ
0	1	✘	✔(אמר 0)	✔
0	0	✔	✘(אמר 1)	✔
1	0	✘	✔(אמר 1)	✔
1	1	✔	✘(אמר 0)	✔

ולכן הפתרון הוא: $X \cup \overline{Y}$

בעיית החניה



תיאור הבעיה

החוקר צריך לספור כמה מכוניות יש בחנייה מעגלית.

☚ אורך המעגל אינו ידוע למחשב.
☚ החנייה גדולה והמחשב רואה רק את המכונית שנמצאת לידו ואת המכונית הבאה.
☚ המחשב יכול לסמן את המכונית בסימן כלשהו, אך הסימן יכול להופיע כבר על מספר מכוניות.
☚ המחשב יכול למחוק את הסימן הקודם ולכתוב סימן חדש.

אלגוריתם

מבנה נתונים - רשימה מקושרת דו- כיוונית או מערך מעגלי (בעזרת מודולו).

1. נסמן ב-V את הרכב הראשון.
2. נתקדם במעגל ונספור את המכוניות עד שנראה מכונית עם הסימן V.
3. נמחק את הסימן V ונרשום במקומו W.
4. נחזור אחורה לנקודת ההתחלה לפי מספר צעדים שספרנו.
5. אם נראה W - סגרנו מעגל, ומספר המכוניות שספרנו הינו מספר המכוניות במעגל. אחרת, אם נראה V - נחזור לסעיף 2.

סיבוכיות

- **במקרה הטוב**

אין לנו אף מכונית שמסומנת ב-V (חוץ מהמכונית הראשונה שסימנו בהתחלה) ולכן נעבור על מעגל המכוניות פעמיים - (פעם אחת עד שנראה V נוסף ונהפוך ל-W ופעם נוספת כשנחזור אחורה). לכן סך הכל קיבלנו סיבוכיות $O(2n) = O(n)$

- **במקרה הגרוע**

מכיוון שהסיבוכיות נמדדת לפי המקרה הגרוע אז סיבוכיות בעיית החניה היא $O(n^2)$.
על כל מכונית במעגל מסומן לנו V ולכן זמן הריצה יהיה סכום של סדרה חשבונית -

$$1 + 1 + 2 + 2 + \dots + n + n = 2(1 + 2 + \dots + n) = 2 \cdot \frac{n \cdot (n+1)}{2} = n \cdot (n+1) = O(n^2)$$

מימוש בגיטהאב

מימוש פתרון הבעיה בעזרת רשימה מקושרת דו-כיוונית

מחלקת Node.java

```
public class Node {

   String signed;
   Node next,prev;
   static int id_counter = 0;
   int id;
   public Node() {
       this.signed = "null";
       this.next = null;
       this.prev = null;
       this.id = id_counter++;
   }

   @Override
   public String toString() {
       return "{id=" + id +
               ", next= " + this.next.id +
               ", prev= " + this.prev.id +
               ", signed= " + signed + "}->\n";
   }
}
```

מחלקת CircularList.java (מבנה הנתונים)

```
public class CircularList {
   Node head, tail;
   int size;

   public CircularList(){
       this.head = null;
       this.tail = null;
       this.size = 0;
   }

   public void add(Node newNode) {
       if(head == null) {
           head = newNode;
           tail = newNode;
           newNode.next = head;
           newNode.prev = tail;
       }else {
           Node current = tail;
           current.next = newNode;
```

```
            newNode.prev = current;
            newNode.next = head;
            tail = newNode;
        }
        size++;
    }

    public int size() {
        return this.size;
    }

    public Node getNode(int id) {
        Node current = head;
        while (current != tail) {
            if(current.id == id)
                return current;
            current = current.next;
        }
        if(tail.id == id)
            return tail;
        return null;
    }

    public void print(){
        Node current = head;
        while(current != tail) {
            System.out.print(current.id+"->");
            current = current.next;
        }
        System.out.print(tail.id+"->");
        System.out.println();
    }
}
```

המחלקה הראשית ParkingProblem.java

```
public class ParkingProblem {

    public static int solution(CircularList list) {
        if(list.head == null)
            return 0;

        Node current = list.head;
        current.signed = "v";
        int current_counter = 1;
        int main_counter = 0;
```

```
        boolean flag = false;

        while(!flag) {

            while (!current.next.signed.equals("v")) {
                current_counter++;
                current = current.next;
            }
            current.next.signed = "w";
            current_counter++;
            main_counter = current_counter;
            while (current_counter != 0) {
                current_counter--;
                current = current.prev;
            }
            if(current.signed.equals("w")) {
                flag = true;
            }
        }

        return main_counter;
    }


    public static void main(String[] args) {
        CircularList list = new CircularList();

        for(int i = 0 ; i < 5 ; i++)
            list.add(new Node());

        list.getNode(1).signed = "v";
        list.getNode(3).signed = "v";
        list.print();
        System.out.println(solution(list));
    }
}
```

מימוש פתרון הבעיה בעזרת מערך מעגלי (מודולו)

כדי לממש פתרון כזה צריך בכלל להבין מה זה מודולו?
אינטואיציה הקלאסית: חשבון מודולרי הוא מה שכולנו עושים כשאנחנו מנסים לדעת מה תהיה השעה עוד כך וכך שעות. אם עכשיו השעה היא 19:00 ואנחנו שואלים "מה תהיה השעה עוד 10 שעות?" אנחנו מוסיפים 10 ל-19, מקבלים 29, ואז מחלקים ב-24 (מספר השעות ביממה), לוקחים את השארית - 5, וזו התשובה.
כאשר השעה כעת היא 20:00, ואנו רוצים לדעת מה תהיה השעה 9 שעות מאוחר יותר, הפעולה שאנו עושים היא $20 + 9 \equiv 5 \; mod \; (24)$.

מימוש בשיטת מודולו במערכים בדרך כלל נכתבת בצורה הבאה:

```
int[] a = {1,2,3,4,5};
int start = 3;
for (int i = 0; i < a.length; i++) {
   System.out.print(a[(start + i) % a.length] + ",");
}
```

כאשר start הוא מיקום ההתחלתי עבור המערך (חייב להיות בגבולות המערך).
נקבל את ההדפסה הבאה: "4,5,1,2,3".
כי עבור צעד ראשון $a[3] = a[3 \; \% \; 5] = a[(3 + 0) \; \% \; 5]$ כלומר $3 \equiv 3 \; mod(5)$.
עבור צעד שני $a[4] = 5 = a[4 \; \% \; 5] = a[(3 + 1)\%5]$ כלומר $4 \equiv 4 \; mod(5)$.
וכך הלאה...

אם נרצה לצעוד לאחור נחליף במקום " start + i " ל- " start - i + a.length ".

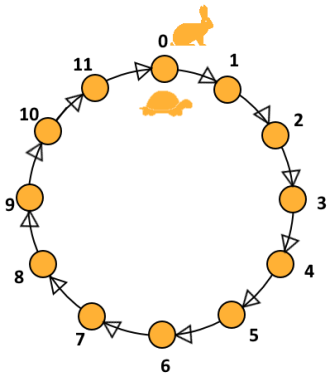
```
int[] a = {1,2,3,4,5};
int start = 3;
for (int i = 0; i < a.length; i++) {
   System.out.print(a[(start - i + a.length) % a.length] + ",");
}
```

נקבל את ההדפסה הבאה: "4,3,2,1,5".

המימוש דיי טיפשי כי ידוע לנו אורך המערך ואנו נעזרים בו כדי לפתור באמצעות מודולו.
אבל נניח כי האורך לא נתון לקריאת המשתמש אלא ערך פרטי.
מחלקת פתרון ParkingProblemModulo.java

```
public class ParkingProblemModulo {
   /**
    * 1 means someone marked the car as visited.
    * 2 means that I marked the car as visited.
    * @param arr a numeric array.
    * @return the number of cars.
    */
   public static int solution(int[] arr) {

       int temp_counter = 0;
       int main_counter = 0;
       int start = 1;
       boolean flag = false;
       while(!flag) {

           // go forward
           while(arr[(start + temp_counter) % arr.length] != 1) {
               temp_counter++;
           }

           arr[(start + temp_counter ) % arr.length] = 2;
           temp_counter++;
           main_counter = temp_counter;

           // go backward
           while( temp_counter != 0 ) {
               temp_counter--;
           }
           if(arr[temp_counter] == 2) {
               flag = true;
           }
       }
       return main_counter;
   }

   public static void main(String[] args) {
       int[] a = {1,1,1,1,1};
       System.out.println(solution(a));
       a = new int[]{1,1,1,1,1,1,1,1,1,1};
       System.out.println(solution(a));
   }
}
```

מציאת מעגל ברשימה מקושרת חד-כיוונית בלי זרוע (הארנב והצב)



תיאור הבעיה

☚ צריך להראות כי המסלול הוא מעגלי.

☚ נתון מסלול וקיימים 2 רובוטים - אחד מהיר (הארנב)
והשני איטי (הצב), מהירות הארנב גדולה פי 2 ממהירות הצב.

☚ שניהם נעים באותו כיוון במעגל ומתחילים באותו נקודת התחלה.

☚ כדי להוכיח כי מדובר במסלול מעגלי, נבדוק אם הארנב והצב ייפגשו בשלב כלשהו, אם הם יפגשו - המסלול אכן מעגלי, אחרת המסלול לא מעגלי.

הוכחה

נוכיח כי הארנב והצב אכן נפגשים (כלומר שקיים מסלול מעגלי).

נסמן:

n - מספר איברי הרשימה ,

k - מספר האיברים מנקודת ההתחלה ועד לנקודת המפגש,

p - מספר סיבובי הצב,

q - מספר סיבובי הארנב,

i - מספר הצעדים שעשה הצב,

2i- מספר הצעדים שעשה הארנב.

נציג את המשוואות עבור מספר הצעדים של כל אחד מהם:

מספר הצעדים שעשה הצב הוא $i = n \cdot p + k$,

מספר הצעדים שעשה הארנב הוא $2i = n \cdot q + k$.

נכפיל את משוואת הצב ב-2 ונשווה: $2n \cdot p + 2k = n \cdot q + k$

קיבלנו שמספר האיברים מנקודת ההתחלה ועד לנקודת המפגש הוא $k = n(q - 2p)$.

מכאן נובע ש-k היא כפולה של n, כלומר הצב והארנב נפגשים בנקודת ההתחלה.

סיבוכיות

מכיוון שאנו עוברים מספר פעמים על כל האיברים במעגל ובודקים האם קיים מעגל או לא,

הסיבוכיות של מציאת מעגל ברשימה מקושרת חד-כיוונית בלי זרוע היא $O(n)$.

- זה לא משנה אם אנחנו עוברים פעם אחת על המעגל או למשל 5 פעמים בסופו של דבר $O(5n) = O(n)$.

מימוש בגיטהאב

מימוש פתרון הבעיה בעזרת רשימה מעגלית

מחלקת Node.java

```
public class Node {

   Node next;
   int id;
   static int unique_id = 0;

   public Node() {
       this.next = null;
       this.id = unique_id++;
   }

}
```

מחלקת CircularList.java (מבנה הנתונים)

```
public class CircularList {

   Node head;
   private int size = 0;

   public CircularList(){
       this.head = null;
   }

   public void add(Node newNode) {
       if(head == null) {
           head = newNode;
           newNode.next = head;
       }
       else {
           Node current = head;
           for(int i = 0 ; i < size-1 ; i++) {
               current = current.next;
           }
           current.next = newNode;
           newNode.next = head;
       }
       size++;
   }

   public int size(){
       return this.size;
```

```
    }

    public Node getNode(int id) {
        Node current = head;
        for(int i = 0 ; i < size ; i++) {
            if (current.id == id)
                return current;
            current = current.next;
        }
        return null;
    }
}
```

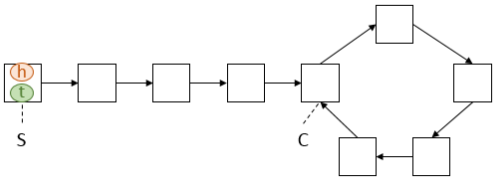
מחלקת פתרון

```
public class HareAndTortoiseProblem {

    public static boolean isCycled(CircularList list) {
        if(list.head == null)
            return false;
        Node turtle = list.head;
        Node hare = list.head;

        while(hare.next.next != null) {
            hare = hare.next.next;
            turtle = turtle.next;
            if(hare == turtle)
                return true;
        }
        return false;
    }

    public static void main(String[] args) {
        CircularList list = new CircularList();

        for(int i = 0 ; i < 8 ; i++)
            list.add(new Node());

        System.out.println(isCycled(list)); // prints true
        list.getNode(7).next = null;
        System.out.println(isCycled(list)); // prints false

    }
}
```

מציאת מעגל ברשימה מקושרת חד-כיוונית עם זרוע

(הארנב והצב / Floyd's Cycle Detection Algorithm)

תיאור הבעיה



☚ נתון מסלול מעגלי שמחובר אליו מסלול נוסף - זרוע.

☚ קיימים 2 רובוטים - אחד מהיר (הארנב) והשני איטי (הצב).

☚ מהירות הארנב גדולה פי 2 ממהירות הצב.

☚ נקודת ההתחלה של הצב והארנב היא על הזרוע.

☚ הצב והארנב אינם יודעים מתי התחילו את הספירה, אך הם יודעים את נקודת ההתחלה.

הוכחה

נוכיח כי הארנב והצב אכן נפגשים.

נסמן:

n - אורך המעגל ,

m - אורך הזרוע,

k - מספר האיברים מנקודת ההתחלה של המעגל ועד לנקודת המפגש,

p - מספר סיבובי הצב,

q - מספר סיבובי הארנב,

i - מספר הצעדים שעשה הצב,

2i - מספר הצעדים שעשה הארנב.

נציג את המשוואות עבור מספר הצעדים של כל אחד מהם:

מספר הצעדים שעשה הצב הוא $i = m + n \cdot p + k$,

מספר הצעדים שעשה הארנב הוא $2i = m + n \cdot q + k$.

נכפיל את משוואת הצב ב-2 ונשווה: $2m + 2np + 2k = m + nq + k$

קיבלנו שמספר האיברים מנקודת ההתחלה ועד לנקודת המפגש הוא $k = n(q - 2p) - m$.

מכאן נובע ש- $k = n - m$ (לא משנה כמה סיבובים הם עשו),

כלומר הצב והארנב נפגשים במרחק n-m מתחילת המעגל. (אורך המעגל פחות אורך הזרוע).

∎

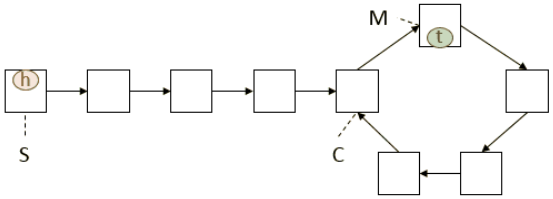
- נשים לב - יש לנו כאן מקרה קצה - עבור המקרה בו $n < m$ ניקח $m = m\%n$.

שאלות למחשבה

נראה פתרון קוד עבור שאלות אלה בדף הבא.

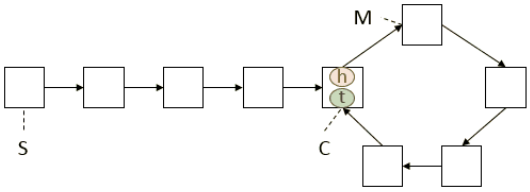
1. החזירו איבר שבוודאות נמצא במעגל
   תשובה: נחזיר את האיבר במיקום ה-k (נקודת המפגש).

2. החזירו את נקודת תחילת המעגל
   תשובה: לאחר ששני הרובוטים נפגשים, שמים את הארנב בתחילת הרשימה והצב נשאר בנקודת המפגש. שניהם מתחילים לזוז במהירות של הצב. לשני הרובוטים יש לעשות m צעדים עד נקודת ההתחלה של המעגל, כלומר נקודת המפגש שלהם היא נקודת ההתחלה של המעגל.



3. החזירו את אורך הזרוע
   תשובה: אורך הזרוע הוא מספר הצעדים שהם עשו בשאלה 2.
   פשוט נפעיל counter שיספור לנו את מספר הצעדים עד לנקודת המפגש
   ואז נבצע שוב את התשובה של שאלה 2.

4. החזירו את אורך המעגל
   תשובה: הארנב נשאר בנקודת ההתחלה של המעגל והצב הולך עד שהוא יפגוש את הארנב - מספר הצעדים שהצב עשה - הוא אורך המעגל.



סיבוכיות

- מציאת איבר שבוודאות נמצא במעגל - $O(n + m)$
- מציאת נקודת תחילת המעגל $O(m)$
- מציאת אורך הזרוע $O(m)$
- מציאת אורך המעגל $O(n)$

הערה- אם נפעיל את מציאת נקודת תחילת המעגל, אפשר על הדרך לספור את מספר הצעדים מתחילת הרשימה ועד לתחילת המעגל, כלומר אפשר לשלוף את אורך הזרוע בסיבוכיות $O(1)$.

מימוש בגיטהאב

מימוש פתרון הבעיה בעזרת רשימה מקושרת

מחלקת Node.java

```
public class Node {

   int id;
   static int id_counter = 0;
   Node next;

   public Node() {
       this.next = null;
       this.id = id_counter++;
   }
}
```

מחלקת LinkedList.java (מבנה הנתונים)

```
public class LinkedList {

   Node head, tail;
   int size;

   public LinkedList(){
       this.head = null;
       this.tail = null;
       this.size = 0;
   }

   public void add(Node newNode){
       if(head == null) {
           head = newNode;
           tail = newNode;
       }
       else{
           tail.next = newNode;
           tail = tail.next;
       }
       size++;
   }


   public int size(){
       return size;
   }

   public Node getNode(int id) {
       Node current = head;
```

```
        if(tail.id == id)
            return tail;

        while (current != tail) {
            if(current.id == id)
                return current;
            current = current.next;
        }
        return null;
    }
}
```

מחלקת הפתרון

```
public class HareAndTortoise {

    static Node theKiss = null;
    static int length_of_arm = 0;

    public static boolean isCycled(LinkedList list) {
        if(list.head == null)
            return false;
        Node hare = list.head;
        Node turtle = list.head;

        while (hare.next.next != null) {
            hare = hare.next.next;
            turtle = turtle.next;
            if(hare == turtle) {
                theKiss = hare;
                return true;
            }
        }

        return false;
    }

    public static Node firstNodeOfCircle(LinkedList list) {
        if(list.head == null)
            return null;

        length_of_arm = 0;
        Node turtle = theKiss;
        Node hare = list.head;
```

```
        while (hare.next != null) {
            if (turtle != null) {
                turtle = turtle.next;
            }
            length_of_arm++;
            hare = hare.next;
            if(hare == turtle) {
                return hare;
            }
        }
        return null;
    }

    public static int lengthOfCircle(Node firstNodeOfCircle) {
        int length = 0;
        Node tortoise = firstNodeOfCircle;
        length++;
        tortoise = tortoise.next;
        while (tortoise != firstNodeOfCircle) {
            length++;
            tortoise = tortoise.next;
        }
        return length;
    }
    public static void main(String[] args) {
        LinkedList list = new LinkedList();

        for(int i = 0 ; i < 6 ; i++)
            list.add(new Node());

        System.out.println("isCycled?: " + isCycled(list));
        list.tail.next = list.getNode(2);
        System.out.println("isCycled?: " + isCycled(list));

        // Question 1 - return a node in the circle.
        System.out.println(theKiss.id);
        // Question 2 - return the first node of the circle.
        Node firstNode = firstNodeOfCircle(list);
        System.out.println(firstNode.id);
        // Question 3 - return the length of the arm.
        System.out.println(length_of_arm);
        // Question 4 - return the length of the circle.
        System.out.println(lengthOfCircle(firstNode));
    }
}
```

מציאת מינימום במערך

תיאור הבעיה

נתון מערך A בעל n איברים - יש למצוא את הערך המינימלי במערך.

5	7	8	4	9	14	3	52	16	2

min

האלגוריתם

```
public static int min(int[] A) {
  int min = A[0];
  for(int i = 1 ; i < A.length ; i++)
    if(A[i] < min)
      min = A[i];
  return min;
}
```

טענה

האלגוריתם מתקיים עבור כל n.

הוכחה

נוכיח באינדוקציה לפי מספר איברי המערך.

- בסיס האינדוקציה: עבור n=1 מתקיים ש- $min = A[0]$ ולכן האלגוריתם עובד.
- הנחת האינדוקציה: נניח שהאלגוריתם עובד עבור n כלשהו.
- שלב האינדוקציה: צריך להוכיח את הטענה עבור n+1.

נסמן $min(N) := min(A[0], A[1], ..., A[n])$ ומתקיים:

$$min(A[0], A[1],..., A[n+1]) = min(min(A[0],..., A[n]), A[n+1]) = min(min(N), A[n+1])$$

לפי הנחת האינדוקציה $min(A[0], A[1],....., A[n])$מחושב נכון ולכן יש לנו 2 מקרים:

1. אם $min(N) \leq A[n+1]$ אז $min(min(N), A[n+1]) = min(N)$
   ולכן $min(N)$קטן מכל איברי המערך.
2. אם $min(N) > A[n+1]$ אז $min(min(N), A[n+1]) = A[n+1]$
   ולכן לכל איבר i במערך מתקיים ש- $A[n+1] < min(N) < A[i]$.

סיבוכיות

יש לנו כאן n-1 השוואות ולכן הסיבוכיות היא $O(n)$.

מציאת מקסימום במערך

תיאור הבעיה

נתון מערך A בעל n איברים - יש למצוא את הערך המקסימלי במערך.

5	7	8	4	9	14	3	52	16	2

max

האלגוריתם

```
public static int max(int[] A) {
  int max= A[0];
  for(int i = 1 ; i < A.length ; i++)
    if(A[i] > max)
      max= A[i];
  return max;
}
```

טענה

האלגוריתם מתקיים עבור כל n.

הוכחה

נוכיח באינדוקציה לפי מספר איברי המערך.

- בסיס האינדוקציה: עבור n=1 מתקיים ש- $max = A[0]$ כנדרש.
- הנחת האינדוקציה: נניח שהאלגוריתם מתקיים עבור n כלשהו.
- צעד האינדוקציה: נוכיח טענה זו עבור n+1.

נסמן $max(N) := max(A[0],\ A[1],\ \ldots.\ ,\ A[n])$ ומתקיים:

$$max(A[0],\ A[1],\ \ldots.\ A[n+1]) = max(max(A[0],\ldots,A[n]), A[n+1]) = max(N, A[n+1])$$

לפי ההנחה מתקים עבור כל n, ולכן עבור האיבר n+1 נחלק ל-2 מקרים:

1. אם $max(N) < A[n+1]$ אז $max(max(N), A[n+1]) = A[n+1]$
   ולכן $A[n+1]$ גדול מכל האיברים ב- $max(N)$.
2. אם $max(N) \geq A[n+1]$ אז $max(max(N), A[n+1]) = max(N)$.

∎

סיבוכיות

יש לנו כאן n-1 השוואות ולכן הסיבוכיות היא $O(n)$.

מציאת מינימום ומקסימום במערך

תיאור הבעיה

נתון מערך A בעל n איברים - יש למצוא את הערך המינימלי והמקסימלי במערך זה.

5	7	8	4	9	14	3	52	16	2
							max		min

פתרון הבעיה

מימוש בגיטהאב

עבור בעיה זו ישנם 4 אפשרויות פתרון, כל אפשרות מציגה סיבוכיות השוואה שונה,
המטרה היא להשתמש בכמות השוואות קטנה ככל הניתן כדי לייעל את סיבוכיות האלגוריתם.

- אפשרות א':
  - נפעיל את האלגוריתם min שהצגנו למציאת מינימום.
  - נפעיל את האלגוריתם max שהצגנו למציאת מקסימום.
  - נחזיר את התוצאות.

```java
public static int getMax(int[] arr) {
   int ans = arr[0];
   for(int i = 1 ; i < arr.length ; i++) {
       if(ans < arr[i])
           ans  = arr[i];
   }
   return ans;
}

public static int getMin(int[] arr) {
   int ans = arr[0];
   for(int i = 1 ; i < arr.length ; i++) {
       if( ans > arr[i])
           ans = arr[i];
   }
   return ans;
}

public static void main(String[] args) {
   int[] arr = {84,31,3,1,567,4,2,93202,32,3};
   System.out.println(getMax(arr));
   System.out.println(getMin(arr));
}
```

סיבוכיות השוואה: סה"כ נקבל $2n - 2$ השוואות.

❖ אפשרות ב':

⇐ נמזג את 2 האלגוריתמים $min(A),\ max(A)$לאלגוריתם אחד.

```
public static int[] minMax(int[] arr) {
   int max = arr[0];
   int min = arr[0];

   for(int i = 1 ; i < arr.length; i++) {
       if(max < arr[i])
           max = arr[i];
       if(min > arr[i])
           min = arr[i];
   }

   return new int[] { max, min };
}

public static void main(String[] args) {
   int[] arr = {84,31,3,1,567,4,2,93202,32,3};
   System.out.println(Arrays.toString(minMax(arr)));
}
```

⇐ **סיבוכיות השוואה:** סה"כ נקבל $2n\ -\ 2$ השוואות.

❖ אפשרות ג':

⇐ נצמצם עוד השוואה בעזרת קביעת min,max בהתחלה.

```
public static int[] improvedMinMax(int[] arr) {
   int max = 0;
   int min = 0;

   if(arr[0] > arr[1]) {
       max = arr[0];
       min = arr[1];
   } else{
       max = arr[1];
       min = arr[0];
   }

   for( int i = 2 ; i < arr.length ; i++ ) {
       if(max < arr[i]) {
           max = arr[i];
       }
       if(min > arr[i]){
           min = arr[i];
       }
   }
```

```
    return new int[] {max, min};
}

public static void main(String[] args) {
    int[] arr = {84,31,3,1,567,4,2,93202,32,3};
    System.out.println(Arrays.toString(improvedMinMax(arr)));
}
```

⇜ **סיבוכיות השוואה:** סה"כ נקבל $2(n-2)+1 = 2n-4+1 = 2n-3$.

❖ אפשרות ד':

⇚ ננסה לצמצם עוד את מספר ההשוואות -נעבור על המערך בזוגות.

```
public static int[] finalSolution(int[] arr){
    int max = 0;
    int min = 0;

    if(arr[0] > arr[1]) { // 1 comparison
        max = arr[0];
        min = arr[1];
    } else {
        max = arr[1];
        min = arr[0];
    }

    // n-2 values, 2*jumps which is n-2/2, in each iteration we make 3 comparisons
    // in conclusion, n-2/2 + n-2/2 + n-2/2 = 3(n-2)/2
    for(int i = 2 ; i < arr.length - 1 ; i+=2) {

        if(arr[i] < arr[i+1]) { // 3 comparisons

            if(min > arr[i]) {
                min = arr[i];
            }
            if(arr[i+1] > max){
                max = arr[i+1];
            }

        } else { // 3 comparisons
            // if arr[i] >= arr[i+1]
            if(min > arr[i+1]) {
                min = arr[i+1];
            }
            if(max < arr[i]) {
                max = arr[i];
            }
```

```
        }

    }

    if(arr.length % 2 != 0) { // 2 comparisons
        if(arr[arr.length-1] > max) {
            max = arr[arr.length-1];
        }
        else if(arr[arr.length-1] < min) {
            min = arr[arr.length-1];
        }
    }
    // in conclusion 1 + 3n/2 + 2 = 3n/2 + 4 = O(3n/2)
    return new int[] { max, min};
}

public static void main(String[] args) {
    int[] arr = {84,31,3,1,567,4,2,93202,32,3};
    System.out.println(Arrays.toString(finalSolution(arr)));
}
```

↞ **סיבוכיות השוואה**: סה"כ נקבל $\frac{3n}{2} - 3 + 3 = \frac{3n-6}{2} + 3 = \frac{n-2}{2} + 1 + 2$ השוואות.

מציאת מקסימום ומקסימום במערך

תיאור הבעיה

נתון מערך A בעל n איברים, יש למצוא את 2 הערכים הכי מקסימלים במערך.

לצורך הנחה כללית לנושא זה: $max1 > max2$.

5	7	8	4	9	14	3	52	2	16
							max1		**max2**

פתרון הבעיה

מימוש בגיטהאב

עבור אלגוריתם זה ישנם 4 אפשרויות, כל אפשרות מציגה סיבוכיות שונה,

המטרה היא להשתמש **בכמות השוואות קטנה** ככל הניתן כדי לייעל את סיבוכיות האלגוריתם.

- ❖ אפשרות א':
  - ⇚ נפעיל את האלגוריתם max המוכר שהצגנו למציאת מקסימום.
  - ⇚ נמחק את הערך שיצא מהמערך.
  - ⇚ נפעיל שוב את האלגוריתם max למציאת מקסימום נוסף.
  - ⇚ נחזיר את התוצאות.

```java
/**
* Comparisons : 2n - 3
* @param arr a numeric array.
* @return the two maximum numbers in the array.
*/
public static int[] maxMax(int[] arr){
   int maxIndex = maximum(arr); // n-1 comparisons
   int max1 = arr[maxIndex];
   arr[maxIndex] = Integer.MIN_VALUE;
   maxIndex = maximum(arr); // n-2 comparisons
   int max2 = arr[maxIndex];
   return new int[] {max1, max2};
}

/**
* a simple method to return the maximum value of the given array.
* @param arr a numeric array.
* @return the maximum value in this array.
*/
```

```java
public static int maximum(int[] arr) {
   int max =  arr[0];
   int index = 0;
   for(int i = 1 ; i < arr.length ; i++) {
       if(max < arr[i]) {
           max = arr[i];
           index = i;
       }
   }
   return index;
}

public static void main(String[] args) {
   int[] arr = {84,31,3,1,567,4,2,93202,32,3};
   System.out.println(Arrays.toString(maxMax(arr)));
}
```

⇜ **סיבוכיות השוואה:** סה"כ נקבל $2n - 3$ השוואות.

❖ אפשרות ב':

⇚ נצמצם עוד השוואה בעזרת קביעת max1,max2 בהתחלה.

```java
/**
* Not so different from the first solution...
* Comparisons : 2n - 3
* @param arr a numeric array.
* @return the two maximum numbers in the array.
*/
public static int[] maxMax(int[] arr) {
   int max1 = arr[0];
   int max2 = arr[1];

   if(max2 > max1) { // 1 comparison
       max1 = arr[1];
       max2 = arr[0];
   }

   for(int i = 2; i < arr.length; i++) {

       if(max1 < arr[i]) { // n-2 comparisons
           max2 = max1;
           max1 = arr[i];
       }
       else if(max2 < arr[i]) { // n-2 comparisons
               max2 = arr[i];
       }
```

```
    } // end for

    return new int[] { max1, max2 };
}

public static void main(String[] args) {
    int[] arr = {84,31,3,1,567,4,2,93202,32,3};
    System.out.println(Arrays.toString(maxMax(arr)));
}
```

⇜ **סיבוכיות השוואה:** סה"כ נקבל $2n - 3$ השוואות.

❖ אפשרות ג' - נעבור על המערך בזוגות:

⇚ נבדוק מי יותר גדול $a[i], a[i+1]$.

⇚ נבדוק מה הערך החדש של max1 (או שהוא לא משתנה).

⇚ יש לנו 2 מועמדים ל-max2: שהם 2 אלו שיותר קטנים מ-max1 החדש.

⇚ נמצא את max2.

⇚ נמשיך הלאה לזוג הבא..

```
public static int[] maxMax(int[] arr) {
    int max1 = arr[0];
    int max2 = arr[1];

    if(max2 > max1) {
        max1 = arr[1];
        max2 = arr[0];
    }
    for(int i = 2 ; i < arr.length -1 ; i += 2) {

        if(arr[i] > arr[i+1]) {

            if(arr[i] > max1) {
                if(arr[i+1] > max1) {
                    max2 = arr[i+1];
                } else {
                    max2 = max1;
                }

                max1 = arr[i];
            }
            else if(arr[i] > max2) {
                max2 = arr[i];
            }
        }
        else { // arr[i] < arr[i+1]
```

```
            if(arr[i+1] > max1) {
                if(arr[i] > max1) {
                    max2 = arr[i];
                } else {
                    max2 = max1;
                }

                max1 = arr[i+1];
            }
            else if(arr[i+1] > max2) {
                max2 = arr[i+1];
            }
        }
    }

    if(arr.length%2 != 0) {
        if(arr[arr.length-1] > max1) {
            max2 = max1;
            max1 = arr[arr.length-1];
        } else if(arr[arr.length-1] > max2) {
            max2 = arr[arr.length-1];
        }
    }
    return new int[] {max1, max2};
}
public static void main(String[] args) {
    int[] arr = {84, 31, 3, 1, 567, 4, 2, 93202, 32, 3};
    System.out.println(Arrays.toString(maxMax(arr)));
}
```

⇜ **סיבוכיות השוואה:** סה"כ נקבל $3\frac{n-2}{2} + 1 + 2 = \frac{3n}{2} - 3 + 3 = \frac{3n}{2}$ השוואות.

❖ אפשרות ד' -אלגוריתם אופטימלי:

⇚ מעבר על המערך בזוגות והצמדת מחסנית לכל איבר.

ניצור מחלקה פנימית בשם Node.

```
static class Node {
    int number;
    Stack<Integer> stack;
    public Node(int num) {
        number = num;
        stack = new Stack<>();
    }
}
```

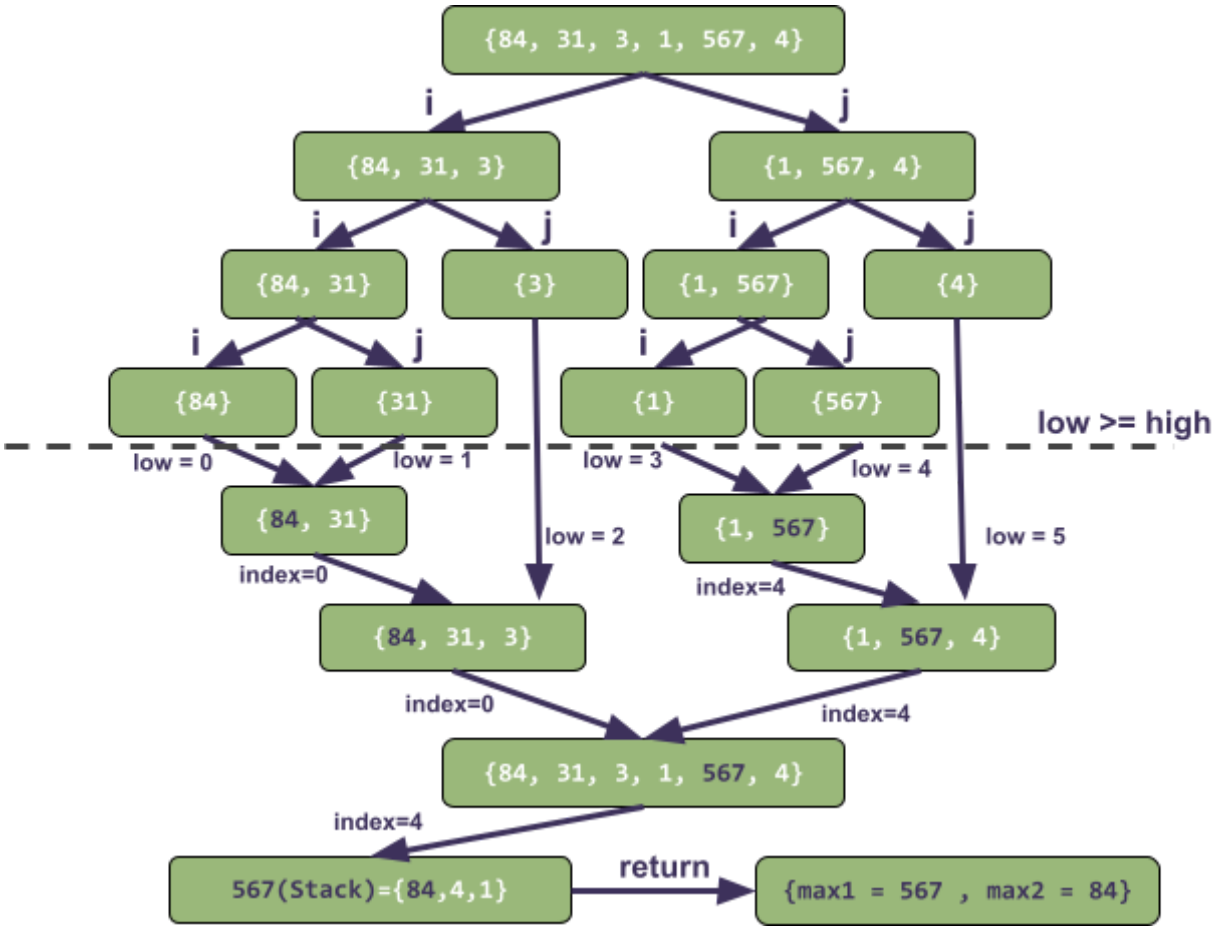
ולמחלקת הפתרון הראשית נממש את הפונקציות הבאות (בשיטה רקורסיבית):

```
public static int[] maxMax(int[] arr) {

   Node[] nodes = new Node[arr.length];

   // init - O(n)
   for(int i = 0 ; i < nodes.length ; i++) {
       nodes[i] = new Node(arr[i]);
   }

   int index = maxMaxRec(nodes, 0 , nodes.length - 1);

   Node biggest = nodes[index];
   int max1 = biggest.number;
   int max2 = biggest.stack.pop();

   while(!biggest.stack.isEmpty()) {
       int temp_max2 = biggest.stack.pop();
       if(temp_max2 > max2)
           max2 = temp_max2;
   }

   return new int[] {max1, max2};
}


private static int maxMaxRec(Node[] nodes, int low, int high) {

   if(low < high) {

       int middle = (high + low)/2;
       int i = maxMaxRec(nodes,low, middle);
       int j = maxMaxRec(nodes, middle + 1 , high);

       int index;
       if(nodes[i].number > nodes[j].number) {
           nodes[i].stack.push(nodes[j].number);
           index = i;
       }
       else { // nodes[i].number <= nodes[j].number

           nodes[j].stack.push(nodes[i].number);
           index = j;
       }

       return index;
   }
```

```
    else {
        return  low;
    }
}

public static void main(String[] args) {
    int[] arr = {84, 31, 3, 1, 567, 4};
    System.out.println(Arrays.toString(maxMax(arr)));
}
```

↞ **סיבוכיות השוואה:** סה"כ נקבל $n - 1 + log(n) - 1 = n + log(n) - 2$ השוואות.

מצורף טסט השוואות זמנים עבור 4 שיטות הפתרון לבעיה בגיטהאב שלי.



אינדוקציה מול רקורסיה

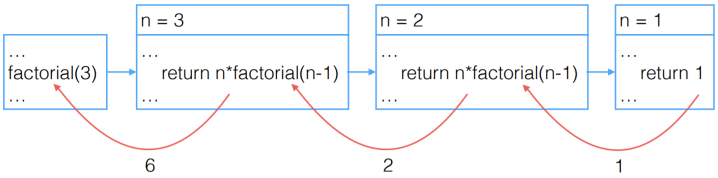
בנושא זה נציג באיזה שיטה עדיף להשתמש כדי לממש את האלגוריתם נכון יותר - באמצעות אינדוקציה או רקורסיה.
גם אם הסיבוכיות של שתי השיטות שווה, ישנם שיקולים נוספים שחשוב לשים עליהם דגש.

אז מתי עדיף להשתמש באינדוקציה ומתי רקורסיה?

מבחינת ניהול זיכרון של רקורסיה, בכל קריאה לפונקציה נוצר **עותק חדש שלה.**
ובעותק יש:

- ⇚ הקצאה חדשה של כל המשתנים הפנימיים, כולל הפרמטרים המופיעים בחתימה שלה.
- ⇚ השמה התחלתית של ערכים לפרמטרים לפי מה שהועבר בקריאה אליה.
- ⇚ אתחול של הפקודה הבאה לבצע בקוד - בעותק החדש זו הפקודה הראשונה בפונקציה.

התהליך נראה כך:



נראה דוגמה להשוואה של רקורסיה מול אינדוקציה עבור חישוב עצרת ונראה את ההשלכות ביניהם:

מימוש בגיטהאב

חישוב עצרת

קלט - מספר אי-שלילי שלם, פלט - !n.

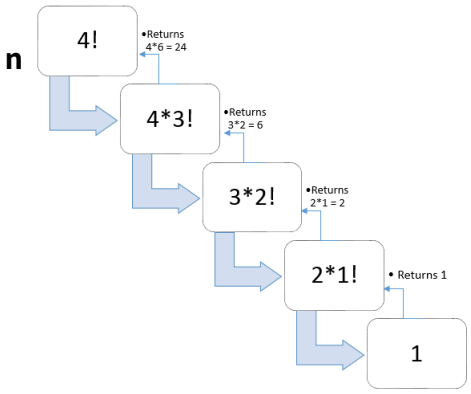
רקורסיבי	אינדוקטיבי

`public static int factorial(int n) {`
`  if(n == 0)`
`  return 1;`
 	
`  return n*factorial(n-1);`
`}`
	`public static int factorial(int n) {`
	`  int f = 1;`
	
	`  for(int i = 1 ; i <= n ; i++){`
	`  f *= i;`
	`  }`
	`  return f;`
	`}`

סיבוכיות נעבור 2n פעמים - נכנסים n פעמים כדי לחשב את הצעדים ואת גודל המחסנית ויוצאים n פעמים - סה"כ $O(n) + O(n) = O(2n) = O(n)$.	**סיבוכיות** נעבור פעם אחת על הלולאה - סה"כ $O(n)$.

קוד הרקורסיבי תואם להגדרה באינדוקציה ויפה יותר אבל דורש **יותר** זיכרון במהלך הביצוע - בתחתית הרקורסיה מוחזקים בו-זמנית **עותקים של factorial.**

למרות שהסיבוכיות שלהם שווה, עדיף לנו להשתמש באינדוקציה, למה? - כי אפשר לראות לפי האלגוריתם בשיטת האינדוקציה כי מתבצע **n פחות** פעולות מאשר שיטת הרקורסיה המבצע 2n פעולות.



הדמייה רקורסיבית

חישוב סדרת פיבונאצ'י

הסדרה מוגדרת באינדוקציה:

$$F_1 = F_2 = 1$$

$$F_n = F_{n-2} + F_{n-1}$$

האיברים הראשונים בסדרה הם:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

קלט - מספר אי-שלילי

פלט - ערך באינדקס לפי סדר הסדרה.

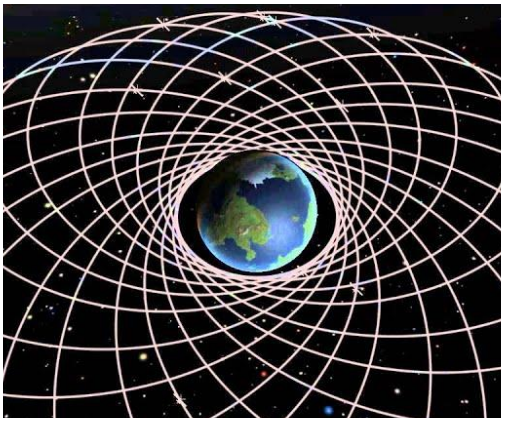
רקורסיבי	אינדוקטיבי
אלגוריתם	אלגוריתם

`public static int fiboRec(int n) {`
`  if( n == 0 || n == 1) {`
`  return n;`
`  }`
`  return fiboRec(n-1) + fiboRec(n-2);`
`  // O(2^N)`
`}`
	`public static int fiboInductive(int n) {`
	`  int[] arr = new int[n+1];`
	`  arr[0] = 0;`
	`  arr[1] = 1;`
	
	`  for(int i = 2 ; i <= n ; i++)`
	`  arr[i] = arr[i-1] + arr[i-2];`
	
	`  return arr[n];`
	`}`

סיבוכיות כל קריאה לפונקציה גוררת שתי קריאות לפונקציה - לכן,הסיבוכיות היא $O(2^n)$	**סיבוכיות** עבור ההשמה של איבר 0 ו-1 זה $O(2)$. עבור הלולאה אנו רצים $n-1$ פעמים פחות 2 ההשמה שבוצעו כבר ולכן התחלנו לרוץ החל מאינדקס 2 כלומר הלולאה מבצעת $O(n-3) = O(n)$. סה"כ - $O(n) + O(2) = O(n)$.

קל לראות במקרה זה שהסיבוכיות בין שתי השיטות בהחלט שונה,

מן הסתם עדיף להשתמש בשיטת האינדוקטיבית כי $O(n) < O(2^n)$ לכל $n \geq 0$.



הערכת ביצועים עבור רקורסיה

⇚ נספור כמה קריאות מתבצעות ל-fibonacci.

⇚ נסמן ב-$T(n)$ את מספר הקריאות הנדרשות כדי לחשב את $fibonacci(n)$.

$$T(1) = T(2) = 1$$

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

⇚ הפתרון מקיים $T(n) = 2F_n - 1$.

מבחינת צריכת זכרון

⇚ בשלב מסויים בריצה פתוחים בבת אחת **n עותקים** של הפונקציה.

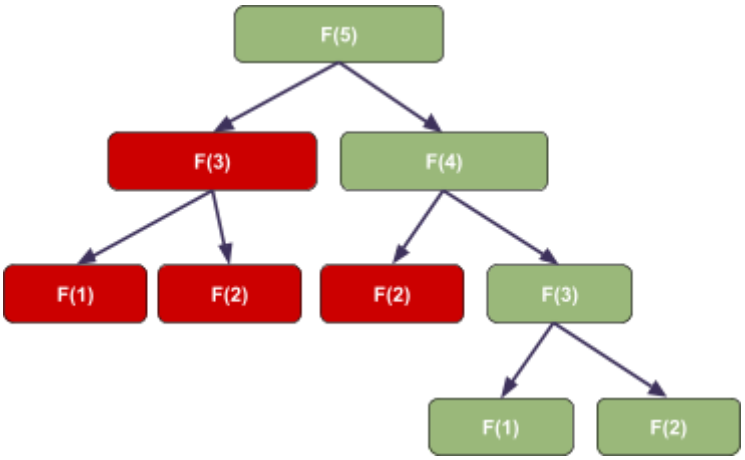
מה הבעיה?

אנחנו מחשבים מחדש אותם ערכים פעם אחר פעם.

⇚ למשל, עבור n == 10, הקריאה $fibo(n-2)$ מחשבת את F_8 וגם הקריאה $fibo(n-1)$ מחשבת את F_8 באחת הקריאות הרקורסיביות שם.

⇚ הערכים F_1-וF_2 מוחזרים בסה"כ F_n פעמים.

⇚ אם נזכור את שני הערכים האחרונים, לא נצטרך לחשב מחדש ברקורסיה את כל הערכים הקודמים.



בתמונה אפשר לראות את מספר הפעולות החוזרות (מסומן באדום).

בהמשך נציג את ***יחס הזהב*** ונראה שימוש לפתרון פיבונאצ'י.

סדרת פיבונאצ'י

הקדמה

סדרת פיבונאצ'י היא סדרה רקורסיבית (סדרה שבה האיבר ה-n נקבע על סמך איברים קודמים),
המוגדרת על-ידי כלל הנסיגה הבא:

$$F_1 = F_2 = 1$$
$$F_n = F_{n-2} + F_{n-1}$$

כאשר F_n הוא איבר במקום ה-n בסדרת פיבונאצ'י.

לאיברי הסדרה, הנקראים "מספרי פיבונאצ'י", יש כמה תכונות מעניינות. אחת מהן קשורה ליחס בין איברים עוקבים של הסדרה, כלומר למנה $\frac{F_n}{F_{n-1}}$.

אם מסתכלים על הגבול של המנה הזו - $\lim_{n\to\infty} \frac{F_n}{F_{n-1}}$ (כלומר, לאיזה מספר המנה מתקרבת כש-n שואף לאינסוף), מקבלים מספר אי-רציונלי: $\Phi = \frac{1+\sqrt{5}}{2} = 1.618033...$.

המספר הזה מכונה "**יחס הזהב**" או "**מספר הזהב**" ונהוג לסמנו באות Φ.

פתרון

מימוש בגיטהאב

דרך פשוטה למציאת איבר n בסדרת פיבונאצ'י הוא בעזרת יחס הזהב,
חשוב - שיטה זו עובדת החל מאינדקס החמישי והשיטה יודעת לחשב עד לאיבר באינדקס ה-34 ולא יותר מזה!

שיטה **אינדוקטיבית**:

```
static double PHI = (1 + Math.sqrt(5)) / 2;
static int[] fib = { 0, 1, 1, 2, 3, 5 };

public static int goldenInduction(int n) {
   if(n <= 5) {
       return fib[n];
   }
   int ans = fib[5];
   for (int i = 5; i < n ; i++) {
       ans = (int) Math.round(ans * PHI);
   }

   return ans;
}
```

שיטה **רקורסיבית**:

```
static double PHI = (1 + Math.sqrt(5)) / 2;
static int[] fib = { 0, 1, 1, 2, 3, 5 };

public static int goldenRecursive(int n) {
   if(n <= 5) {
       return fib[n];
   }
   if(n == 6)
       return (int) Math.round(5 * PHI);

   return (int) Math.round( goldenRecursive(n-1) * PHI);
}
```

פתרון בעזרת נוסחת binet בסיבוכיות זמן **קבוע** $O(1)$, אך גם כאן יש הגבלה עד איבר באינדקס 34 ולא יותר מזה! הנוסחה: $S_n = \Phi^n - \frac{(-\Phi^{-n})}{\sqrt{5}}$.

```
static double PHI = (1 + Math.sqrt(5)) / 2;

public static int binetFormula(int n) {
   return (int) ((Math.pow(PHI, n) - Math.pow(-PHI, -n))/Math.sqrt(5));
}
```

עבור איברי ב**אינדקס שלילי** של סדרת פיבונאצ'י נראה את הפתרון **הרקורסיבי** הבא,
כאשר הפתרון לא נפתר עם יחס הזהב אלא כפתרון פשוט שעובד **לכל איבר n** ולא מוגבל כמו הפתרונות הקודמים שהצגנו עבור יחס הזהב.

```
public static int fiboRecursive(int n) {
   if( n == 0 || n == 1) {
       return n;
   }
   return fiboRecursive(n-1) + fiboRecursive(n-2);
}


public static int negativeFiboRecursive(int n) {
   if(n >= 0) { // if its not a negative index then use the positive method.
       return fiboRecursive(n);
   }
   return negativeFiboRecursive(n+2) - negativeFiboRecursive(n+1);
}
```

סדרת פיבונאצ'י ומטריצות

שיטה נוספת להחזיר איבר באינדקס n בסדרת פיבונאצ'י היא בעזרת חישוב נוסחת הסדרה על-ידי מטריצות.

נזכר בנוסחה הרקורסיבית לסדרת פיבונאצ'י $F_n = F_{n-1} + F_{n-2}$.

אם כך, אפשר לייצג באותו אופן את הנוסחה בצורה הבאה $F_{n+1} = F_n + F_{n-1}$.

לפי אלגברה לינארית, נציג צירוף לינארי עבור F_n ו-F_{n-1}:

$$F_n = \mathbf{1} \cdot F_{n-1} + \mathbf{1} \cdot F_{n-2}$$

$$F_{n-1} = \mathbf{1} \cdot F_{n-1} + \mathbf{0} \cdot F_{n-2}$$

כלומר

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = (F_{n-1} + F_{n-2}) + (F_{n-1}) = F_n + F_{n-1} = F_{n+1}$$

ולכן ניתן לייצג כל איבר הבא n בסדרת פיבונאצ'י בכפל מטריצות באופן הבאה:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

כאשר n הוא האיבר באינדקס n של סדרת פיבונאצ'י.

אלגוריתם

⇚ כדי לחשב את סדרת פיבונאצ'י נשתמש בכפל מטריצות.

⇚ נגדיר מטריצה ראשונית להיות ערכי ההתחלה של הסדרה ונכפיל אותה n-1 פעמים.

הוכחה

נוכיח באינדוקציה את נכונות האלגוריתם.

☚ הנחת האינדוקציה - $A^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$.

☚ צעד האינדוקציה - צריך להוכיח כי $A^{n+1} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$:

$$A^{n+1} = A^n \cdot A = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$$

כדי לקבל פתרון נכפול את $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ n פעמים, ונחזיר את הפתרון שהתקבל במיקום $mat[0][0]$ כלומר את F_{n+1}

זאת מאחר וסדרת פיבונאצ'י מתחילה באיבר 0 ועד עכשיו ספרנו אינדקסים החל מאיבר 1.

מימוש הפתרון בשיטה **אינדוקטיבית** בזמן ריצה $O(n)$

```
public static int fibMatrix(int n) {
   if (n <= 0) {
       return 0;
   }

   int[][] fiboMatrix = { {1,1}, {1,0} };

   matrixPower(fiboMatrix,n);

   return fiboMatrix[0][0];
}

private static void matrixPower(int[][] matrix, int n) {

   int[][] ans = { {1,1}, {1,0} };

   for (int i = 1; i < n-1; i++) {
       matrixMultiply(matrix,ans);
   }

}

private static void matrixMultiply(int[][] mat1, int[][] mat2) {

   int x =  mat1[0][0] * mat2[0][0] + mat1[0][1] * mat2[1][0];
   int y =  mat1[0][0] * mat2[0][1] + mat1[0][1] * mat2[1][1];
   int z =  mat1[1][0] * mat2[0][0] + mat1[1][1] * mat2[1][0];
   int w =  mat1[1][0] * mat2[0][1] + mat1[1][1] * mat2[1][1];

   mat1[0][0] = x;
   mat1[0][1] = y;
   mat1[1][0] = z;
   mat1[1][1] = w;
}

public static void main(String[] args) {
   System.out.println(fibMatrix(6));
}
```

בשיטה **הרקורסיבית**, הנוסחה: $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n = \begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix}$, תספק לנו פתרון בסיבוכיות זמן ריצה של רק $O(log\ n)$

בדומה לפתרון חישוב חזקה ברקורסיה, נצל את אותה השיטה לטובת הפתרון שלנו.

נכפול באופן רקורסיבי את המטריצה, המימוש מאוד דומה לקודם עם הבדל בפונקציה power.

```
 public static int fibMatrixLog(int n) {
    if (n <= 0) {
        return 0;
    }

    int[][] fiboMatrix = { {1,1}, {1,0} };

    matrixPowerLog(fiboMatrix,n-1);

    return fiboMatrix[0][0];
}



private static void matrixPowerLog(int[][] matrix, int n) {

    if(n == 0 || n == 1)
        return;

    int[][] ans = { {1,1}, {1,0} };

    matrixPowerLog(matrix, n/2);
    matrixMultiplyLog(matrix,matrix);

    if(n%2 != 0) {
        matrixMultiplyLog(matrix,ans);
    }

}


private static void matrixMultiplyLog(int[][] mat1, int[][] mat2) {

    int x =  mat1[0][0] * mat2[0][0] + mat1[0][1] * mat2[1][0];
    int y =  mat1[0][0] * mat2[0][1] + mat1[0][1] * mat2[1][1];
    int z =  mat1[1][0] * mat2[0][0] + mat1[1][1] * mat2[1][0];
    int w =  mat1[1][0] * mat2[0][1] + mat1[1][1] * mat2[1][1];

    mat1[0][0] = x;
    mat1[0][1] = y;
    mat1[1][0] = z;
    mat1[1][1] = w;
}

public static void main(String[] args) {
    System.out.println(fibMatrixLog(6));
}
```

חישוב חזקה

תיאור הבעיה

נרצה לחשב חזקה בצורה היעילה ביותר.

⇐ נרצה לחשב את a^n.

בראייה ראשונית, אפשר לראות כי סיבוכיות זמן הריצה עבור $a^n = a \cdot a \cdot \ldots \cdot a$ הוא $O(n)$.

בדומה לנושא של אינדוקציה מול רקורסיה נראה את 2 הפתרונות עבורם ונחליט איזה שיטה יעילה יותר.

מימוש בגיטהאב

חישוב חזקה בשיטה **רקורסיבית**

```
public static int powerRecursion(int a, int n) {
   return n == 0 ? 1 : a * powerRecursion(a,n-1);
}
public static void main(String[] args) {
   System.out.println(powerRecursion(2,5)); // 32
   System.out.println(powerRecursion(3,3)); // 27
}
```

בנושא אינדוקציה מול רקורסיה הראינו כי רקורסיה מקצה זיכרון נוסף עבור כל קריאה לפונקציה כולל עבור כל הפרמטרים הנמצאים בתוך הפונקציה כולל המשתנים בארגומנט השליחה.

סה"כ סיבוכיות זמן הריצה היא $O(n) = O(2n)$.

חישוב חזקה בשיטה **אינדוקטיבית**

```
public static int powerInduction(int a, int n) {
   int solution = 1;
   while (n > 0) {
       solution *= a;
       n--;
   }
   return solution;
}
public static void main(String[] args) {
   System.out.println(powerInduction(2,5)); // 32
   System.out.println(powerInduction(3,3)); // 27
}
```

גם כאן קל לראות כי סיבוכיות זמן הריצה היא $O(n)$ אך בגלל שרקורסיה עושה פעולות נוספות של הקצאות זיכרון עבור כל קריאה, השיטה האינדוקטיבית יותר יעילה במקרה זה.

עד עכשיו למדנו חישוב חזקה בסיבוכיות זמן ריצה של $O(n)$, לדוגמה, עבור x^8 נחשב ידנית באופן הבא:

$$x^8 = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$$

האם אפשר לספק פתרון נוסף ביעילות טובה יותר מ-$O(n)$?

אפשר לראות כי עבור x^8 ביצענו 7 פעולות כפל עד שהגענו לפתרון, אנו יכולים לייעל את השיטה כך שנבצע פחות פעולות הכפלה ואז סיבוכיות זמן הריצה שלנו תרד משמעותית.

אם

$$x^4 = x^2 \cdot x^2$$

אז באופן דומה, אפשר לחשב את x^8 בצורה הבאה:

$$x^8 = x^2 \cdot x^2 \cdot x^2 \cdot x^2$$

ככה בעצם קיבלנו 3 פעולות הכפלה במקום 7 פעולות, שזה בעצם $\log n \Rightarrow \log_2 8 = 3$.

לכן הסיבוכיות עשויה להיות $O(log\ n)$.

ידוע שכל מספר עשרוני ניתן לכתוב כסכום של מספרים בחזקות של 2.

נסתכל על x^{11}, ננסה לפתור זאת באמצעות הנוסחה הבאה: $x^{11} = x^8 \cdot x^2 \cdot x^1$.

אנו רואים שכאן השקענו סוג של $O(log\ n)$ פעולות, יש באפשרותינו כעת להעזר בחזקות הבאות:

$$x^1,\ x^2,\ x^4,\ x^8$$

מאחר ו-$11 = 8 + 2 + 1$ נרשום את הייצוג הבינארי של המספר הזה:

$$11 = 8 + 2 + 1 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

כלומר $1011_2 = 11_{10}$.

דרך נוספת וקלה יותר לקראת הפתרון, הוא לחלק את מספר החזקה בכל צעד בחצי, אם השארית היא 1 אז נחבר את החישוב לתוצאה הסופית, אחרת לא נחשב כלום ונמשיך לצעד הבא, כלומר:

מספר	שלם	שארית
11|2	5	1
5|2	2	1
2|2	1	0
1|2	0	1

נקרא את המספר שקיבלנו בשארית מלמטה למעלה וזה באמת 1011.

חישוב חזקה בעזרת מספרים בינאריים בשיטה **רקורסיבית**

```
public static int recursion(int a, int n) {
   return rec(a, n , 1);
}

public static int rec(int a, int n, int answer) {

   if(n == 0)
       return answer;

   if(n % 2 == 1)
       answer = answer * a;

   return rec(a * a,n/2,answer);
}

public static void main(String[] args) {
   System.out.println(recursion(2,5)); // 32
   System.out.println(recursion(3,3)); // 27
}
```

אמנם אין לנו 2 קריאות לפונקציה בכל שלב בפונקציה, אך זה לא אומר שלא מדובר בסיבוכיות $O(log\ n)$ כי מאחר ותנאי העצירה שלנו הוא באחריות פרמטר n וכי בכל צעד אנו קוראים לצעד הבאה כאשר $n/2$ אז באמת מתקיים כאן סיבוכיות זמן ריצה של $O(log\ n)$.

חישוב חזקה בעזרת מספרים בינאריים בשיטה **אינדוקטיבית**

```
public static int induction(int a, int n) {
   int answer = 1;

   while (n > 0) {

       if(n % 2 == 1)
           answer = answer * a;

       n = n / 2;
       a = a * a;
   }

   return answer;
}
```

גם כאן סיבוכיות זמן הריצה היא $O(log\ n)$.

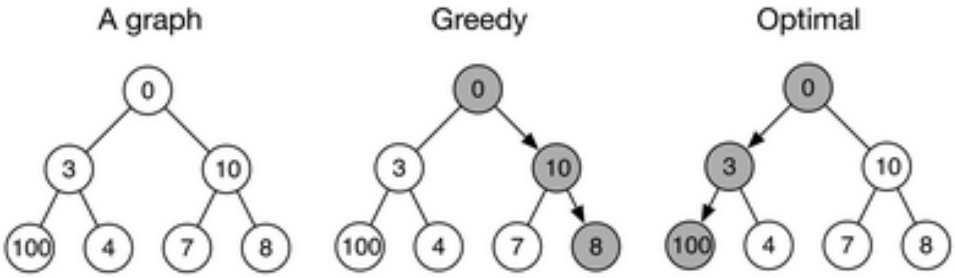
מה זה אלגוריתם חמדני?

הגדרה

אלגוריתם יקרא חמדן (Greedy) אם בכל שלב הוא בוחר באפשרות המשתלמת ביותר באותו הרגע מבלי לקחת בחשבון השלכות לטווח רחוק.

הסבר

במדעי המחשב אלגוריתם חמדן הוא אלגוריתם לפיו בוחרים את האפשרות הטובה ביותר הנראית לעין בשלב הנוכחי, מבלי לקחת בחשבון את ההשפעה של צעד זה על המשך הפתרון.
אלגוריתמים חמדנים נפוצים בפתרון בעיות מיטוב, בהן מנסים למצוא את הפתרון הטוב ביותר. לעיתים, כאשר לא ניתן למצוא את הפתרון האופטימלי בזמן סביר, שימוש באלגוריתם חמדן עשוי לתת קירוב טוב לפתרון המיטבי בזמן קצר.
במקרים מסוימים האלגוריתם החמדן הוא גם האלגוריתם האופטימלי.
כלומר, סדרת בחירות חמדניות נותנת לפעמים את הפתרון האופטימלי הכללי לבעיה.
אמנם אלגוריתם זה מספק פתרון מהיר ביעילות, אך החיסרון שלו שהפתרון לעיתים לא המיטבי.



A greedy algorithm fails to maximise the sum of nodes along a path from the top to the bottom because it lacks the foresight to choose suboptimal solutions in the current iteration that will allow for better solutions later

מה זה אלגוריתם חיפוש שלם (Brute Force Search)

אלגוריתם זה עובר על כל המצבים האפשריים הקיימים ולכן הוא מחזיר פתרון נכון יותר לעומת אלגוריתמים חמדניים אבל היעילות שלו מבחינת סיבוכיות זמן הריצה פחות טובה מהחמדני.
זה בעצם כמו ליצור עץ של כל המצבים ואז לעבור על כל העץ ולחפש את הפתרון הטוב ביותר עבורנו.

משחק המספרים

מצורפים טסטים עבור הבעיה הזאת **בגיטהאב**.

תיאור הבעיה

נתון מערך A בעל n איברים **(n זוגי)** - במשחק זה משתתפים שני שחקנים.
כל אחד מהשחקנים **בתורו** יכול לבחור מספר אחד מהקצה השמאלי או מהקצה הימני של המערך.
המנצח הוא השחקן בעל **סכום המספרים הגדול ביותר.**

0	1	2	3	4	5	6	7	8	9
5	7	8	4	9	14	3	52	16	2

Player 1 **VS** Player 2

נניח כי השחקן הראשון הוא אנחנו והוא תמיד המתחיל בכל משחק.
בנוסף, תמיד נחשוב שאנחנו משחקים מול השחקן הטוב ביותר בעולם, אין פה מקום לטעויות.

פתרון הבעיה

מימוש בגיטהאב

עבור בעיה זו נציג 4 אלגוריתמים שונים, כל אלגוריתם מציג שיטת ניצחון אחרת.
השאיפה היא לא רק תמיד לנצח את השחקן השני, אלא גם **לנצח ברווח המקסימלי ביותר** של סכום המספרים שלנו בכל משחק (ניצחון).

- ❖ אפשרות א' - אלגוריתם חמדני
  - ⇚ אנו צריכים לבחור מספר אחד מהקצה השמאלי או מהקצה הימני של המערך.
  - ⇚ בכל שלב כזה אנו נבחר את האפשרות הטובה ביותר הנראית לעין.
  - ⇚ אנחנו נבחר מספר מבלי לקחת בחשבון את המשך השלבים.

```
public static int[] greedy(int[] arr) {
   int player1 = 0, player2 = 0, games = 0;
   int left_corner = 0, right_corner = arr.length-1;

   while (games != arr.length) {

       if(games % 2 == 0) {
           if(arr[left_corner] > arr[right_corner]) {
               player1 += arr[left_corner];
               left_corner++;
```

```
            }else {
                player1 += arr[right_corner];
                right_corner--;
            }
        }
        else {
            if(arr[left_corner] > arr[right_corner]) {
                player2 += arr[left_corner];
                left_corner++;
            }else {
                player2 += arr[right_corner];
                right_corner--;
            }
        }
        games++;
    }
    System.out.println("difference is " + (player1 - player2));
    return new int[] {player1,player2};
}

public static void main(String[] args) {
    int[] arr = {5, 7, 8, 4, 9, 14, 3, 52, 16, 2};
    System.out.println(Arrays.toString(greedy(arr)));
}
```

⇜ זה אסטרטגיה לא יעילה **שלא** מספקת לנו פתרון, נקבל את ההדפסה הבאה:

```
difference is -38
[79 ,41]
```

❖ אפשרות ב' - זוגי או אי-זוגי

⇚ לפני תחילת המשחק אנחנו נחשב את סכום האיברים במקומות הזוגיים במערך וגם את סכום האיברים במקומות האי-זוגיים במערך ונבחר את הסכום הגדול מביניהם.

⇚ לאחר שבחרנו את הסכום, נתחיל את המשחק כמובן כשיש לנו זכות להתחיל ראשונים.

⇚ בכל שלב, אנחנו תמיד נבחר במערך את המספר במקום הזוגי או האי-זוגי וזה תלוי באיזה סוג סכום בחרנו לפני תחילת המשחק, למשל אם סכום הזוגיים היה גדול יותר אז לאורך כל שלבי המשחק תמיד נבחר את המקומות הזוגיים.

⇚ אנו צריכים לבחור מספר אחד מהקצה השמאלי או מהקצה הימני של המערך כל עוד זה מספר במיקום זוגי.

הוכחה

נוכיח אסטרטגיה זו באינדוקציה:

יהי סדרת מספרים $a_1, a_2, ..., a_n$ שחקן א' מחשב את 2 הסכומים (הזוגיים והאי-זוגיים):

$S_1 = a_1 + a_3 + .. + a_{n-1}$ ו- $S_2 = a_2 + a_4 + ... + a_n$.

במקרה שבו $S_1 \geq S_2$ שחקן א' בוחר ב-a_1, ממשיך לבחור במספרים במקומות האי-זוגיים ומנצח.

נבחן את המקרה שבו $S_1 \geq S_2$ (המקרה ההפוך סימטרי לחלוטין):

- בסיס האינדוקציה: כאשר שחקן א' בוחר ב-a_1 לשחקן ב' יש 2 אפשרויות בחירה:

a_2 ו-a_n ששניהם נמצאים במקומות הזוגיים. במידה והוא בוחר ב-a_2 אז שחקן א' יכול לבחור ב-a_3 ובמידה והוא בוחר ב-a_n אז שחקן א' יכול לבחור ב-a_{n-1}.

כלומר לשחקן א' יש שוב אפשרות בחירה באיבר במקום אי זוגי הנותן סכום גדול יותר. יחד עם זאת שחקן ב' תמיד יוכל לבחור רק באיברים הנמצאים במקומות זוגיים הנותנים סכום קטן יותר.

- הנחת האינדוקציה: נניח כי בשלב בו שחקן א' בוחר מספר, בסדר נותרו המספרים $a_i, a_{i+1}, ..., a_{j-1}, a_j$ כאשר i זוגי ו-j אי זוגי (המקרה ההפוך סימטרי לחלוטין).
- שלב האינדוקציה: ברור ששחקן א' יבחר ב-a_j כי הוא אי-זוגי ולשחקן ב' נותרו שוב 2 אפשריות במיקום הזוגי a_i ו-a_{j-1} הנותנות לו סכום קטן יותר. בכל בחירה של שחקן ב' לשחקן א' נפתחת בחירה של מספר אי זוגי (a_{i+1} או a_{j-2}) שנמצא במקום אי זוגי ונותן סכום גדול יותר.

I

```
public static String chooseEvenOrOdd(int[] arr) {
   int odd_sum = 0 , even_sum = 0;

   for(int i = 0 ; i < arr.length; i++) {
       if(i%2==0)
           even_sum += arr[i];
       else
           odd_sum += arr[i];
   }

   if(even_sum > odd_sum)
       return "even";
   else
       return "odd";
}

public static int[] game(int[] arr) {
   int player1 = 0, player2 = 0, games = 0;
```

```
int left_corner = 0, right_corner = arr.length-1;
String strategy = chooseEvenOrOdd(arr);

while (games != arr.length) {

    if(strategy.equals("even")) {

        if(games % 2 == 0) { // Player1 turn

            if (left_corner % 2 == 0) {
                player1 += arr[left_corner];
                left_corner++;
            } else {
                player1 += arr[right_corner];
                right_corner--;
            }

        }
        else { // Player2 turn

            if(arr[left_corner] > arr[right_corner]) {
                player2 += arr[left_corner];
                left_corner++;
            } else {
                player2 += arr[right_corner];
                right_corner--;
            }
        }

    } else { // odd

        if(games % 2 == 0) { // Player1 turn

            if(left_corner % 2 == 1) {
                player1 += arr[left_corner];
                left_corner++;
            } else {
                player1 += arr[right_corner];
                right_corner--;
            }
        } else { // Player2 turn

            if(arr[left_corner] > arr[right_corner]) {
                player2 += arr[left_corner];
                left_corner++;
            } else {
                player2 += arr[right_corner];
```

```
                    right_corner--;
                }
            }
        }
        games++;
    }

    return new int[] {player1,player2};
}

public static void main(String[] args) {
    int[] arr = {5, 7, 8, 4, 9, 14, 3, 52, 16, 2};
    System.out.println(Arrays.toString(game(arr)));
}
```

⇜ קוד לא כל כך חכם, וגם האסטרטגיה לא בדיוק טובה לנו.

נתבונן במערך הבא:

Player 1 **VS** **Player 2**

אם נפעל באותה השיטה הקודמת נקבל שסכום האי-זוגיים וסכום הזוגיים שווים בסכומם.

כלומר **1+6+3 = 10** וגם **3 + 1 + 6 = 10.**

במקרה זה לא הצלחנו לספק את הפתרון הדרוש לבעיה, ברור שיש פתרון נוסף שנותן מענה למקרה זה.

❖ אפשרות ג' - אדפטיבית:

⇚ בכל שלב נחשב את הסכום הזוגי/אי-זוגי הקיימים ונבחר במספר המשתלם לנו.

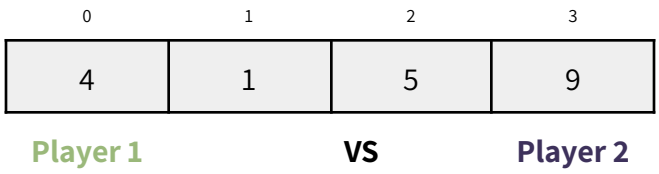
⇚ מקרה זה יותר טוב מאפשרות ב' כי הוא מאפשר לנו לנצח גם במקרה של סכום שווה כמו המקרה שראינו בעמוד הקודם.

```
public static String chooseEvenOrOdd(int[] arr, int left, int right) {
    int odd_sum = 0 , even_sum = 0;

    for(int i = left ; i < right+1; i++) {
        if(i%2==0)
            even_sum += arr[i];
        else
            odd_sum += arr[i];
    }
```

```
    if(even_sum > odd_sum)
        return "even";
    else
        return "odd";
}

public static int[] game(int[] arr) {
    int player1 = 0, player2 = 0, games = 0;
    int left_corner = 0, right_corner = arr.length-1;

    while (games != arr.length) {

        String strategy = chooseEvenOrOdd(arr, left_corner, right_corner);

        if(strategy.equals("even")) {

            if(games % 2 == 0) { // Player1 turn

                if (left_corner % 2 == 0) {
                    player1 += arr[left_corner];
                    left_corner++;
                } else {
                    player1 += arr[right_corner];
                    right_corner--;
                }

            }
            else { // Player2 turn

                if(arr[left_corner] > arr[right_corner]) {
                    player2 += arr[left_corner];
                    left_corner++;
                } else {
                    player2 += arr[right_corner];
                    right_corner--;
                }
            }

        } else { // odd

            if(games % 2 == 0) { // Player1 turn

                if(left_corner % 2 == 1) {
                    player1 += arr[left_corner];
```

```
                left_corner++;
            }

        else {
                player1 += arr[right_corner];
                right_corner--;
            }
        } else { // Player2 turn

            if(arr[left_corner] > arr[right_corner]) {
                player2 += arr[left_corner];
                left_corner++;
            } else {
                player2 += arr[right_corner];
                right_corner--;
            }
        }
    }
    games++;
  }

  System.out.println("difference is " + (player1 - player2));
  return new int[] {player1,player2};
}

public static void main(String[] args) {
  int[] arr = {5, 7, 8, 4, 9, 14, 3, 52, 16, 2};
  int[] arr2 = {1,3,6,1,3,6};
  System.out.println(Arrays.toString(game(arr)));
  System.out.println(Arrays.toString(game(arr2)));
}
```

האם האסטרטגיה שהצגנו תמיד נותן לנו את הרווח המקסימלי ביותר?
במערך הבא, השיטה האדפטיבית לא מספקת רווח מקסימלי עבורנו:



נציג את האפשרות הבאה שמספקת את הרווח המקסימלי ביותר:

❖ אפשרות ד' - תכנות דינאמי:

⇚ נעבור על כל אפשרויות המשחקים שלנו.

⇚ נבנה מטריצת עזר ובאלכסון שלה נציב את איברי המערך החד-מימדי (המשחק הנתון).

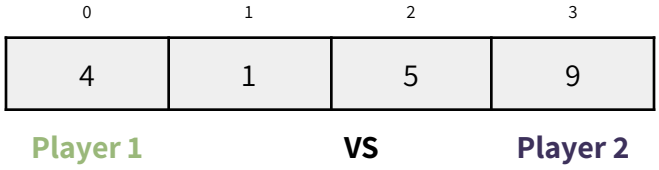
⇚ כל תא במטריצה הוא אפשרות משחק.

⇚ נרוץ מהפינה הימנית התחתונה עד להתחלה של המטריצה, בכל תא יהיה את מקסימום הרווח שניתן לקבל.

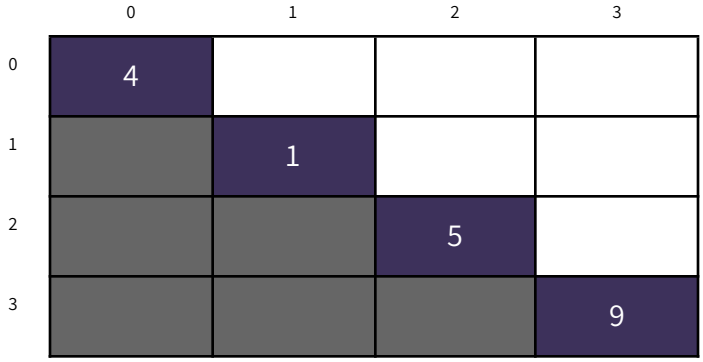
⇚ לאחר הפעלת כל אפשרויות המשחקים על המטריצה , נשלח לפונקציה שתבצע חישוב לאחור כדי לחשב את הסכום שצבר שחקן 1 וגם עבור שחקן 2.

⇚ לבסוף נציג את ההפרש המקסימלי שקיבלנו ונציג את הסכומים שאסף כל שחקן.

עבור המערך הבא:



נצהיר על מטריצה חדשה ונציב באלכסון הראשי שלה את מערך המשחק:



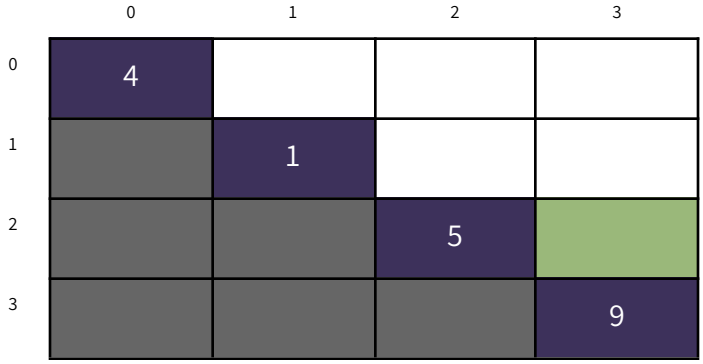
מימוש צעד זה:

```
int[][] matrix = new int[arr.length][arr.length];

// fill the main diagonal
for(int i = 0; i < arr.length; i++) {
   matrix[i][i] = arr[i];
}
```

אחרי הצבת האלכסון נתחיל להפעיל את המשחקים ולהציב בכל תא מתאים את הרווח המקסימלי עבור על משחק. לאורך חישוב המטריצה נפעיל את הנוסחה הבאה עבור כל תא:

$$matrix[i][j] = Math.max(\, matrix[i][i] - matrix[i+1][j] \,,\, matrix[j][j] - matrix[i][j-1] \,)$$

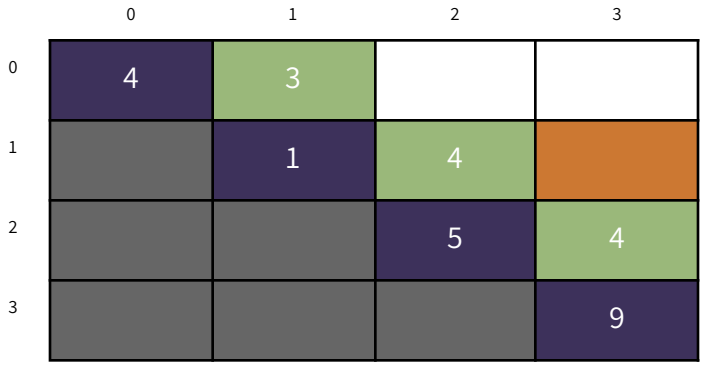
עבור חישוב האלכסון הראשון נסתכל על התא **שצבוע בירוק** ונשאל עבור איזה משחק מבין 2 האפשרויות נקבל את הרווח המקסימלי? - כלומר:



$$matrix[2][3] = Math.max(\, matrix[2][2] - matrix[2+1][3] \,,\, matrix[3][3] - matrix[2][3-1] \,)$$
$$matrix[2][3] = Math.max(\, 5 - 9 \,,\, 9 - 5 \,) = Math.max(\, -4 \,,\, 4 \,) = 4$$

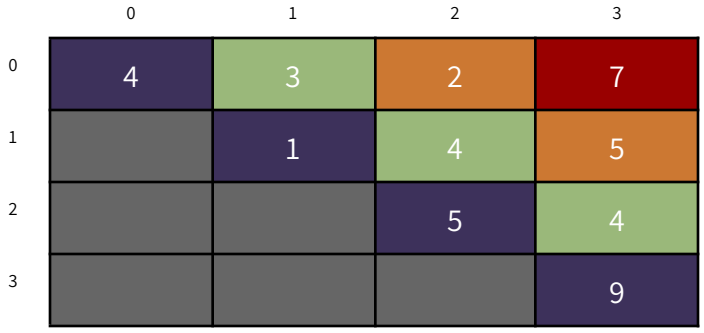
וכך נפעל עבור כל שאר האלכסון הירוק באיור הבא.

עבור המשחק בתא **הכתום** נחשב:



$$matrix[1][3] = Math.max(\, matrix[1][1] - matrix[1+1][3] \,,\, matrix[3][3] - matrix[1][3-1] \,)$$
$$matrix[1][3] = Math.max(-3 \,,\, 5) = 5$$

וכך גם את המשך האלכסון ובאותו שיטה למשך המטריצה...



כעת אפשר לראות כי המשחק הגדול ביותר (המערך ששלחנו בהתחלה) הוא התא הצבוע **באדום** כאשר המספר **7** הוא הרווח המקסימלי ביותר שהשחקן שלנו יכול לקבל עבור המשחק (המערך).
המימוש של בניית המטריצה (התהליך שעשינו עד עכשיו):

```
for(int i = arr.length - 2 ; i >= 0 ; i--) {
   for(int j = i + 1 ; j < arr.length; j++) {
       mat[i][j] = Math.max(matrix[i][i] - mat[i+1][j] , mat[j][j] - mat[i][j-1]);
   }
}
```

כעת נרצה לחשב את הסכום שצבר השחקן שלנו לאורך המשחק המרכזי, לכן נחזור אל הנוסחה שהצגנו ונתחיל מהתא של המשחק המרכזי ונשאל איזה תא סיפק לי את הרווח שבו אני עומד?

$$matrix[i][j] = Math.max(matrix[i][i] - matrix[i+1][j], matrix[j][j] - matrix[i][j-1])$$

כלומר עבור המקרה שלנו, אם אנחנו מסתכלים על **7** , נחזור לנוסחה ונבחן מי הגורם לרווח זה?

$$7 = Math.max(4 - 5, 9 - 2)$$

ברור ש- 7 = 2 - 9 ולכן נבחר את **9** להיות חלק מהסכום שלנו, וכך הלאה עד שנסיים את חישוב המשחק.
מימוש תכנות דינאמי:

```
public static int[] dynamic(int[] arr) {

   int[][] mat = new int[arr.length][arr.length];

   for(int i = 0; i < arr.length; i++) {
       mat[i][i] = arr[i];
   }
   for(int i = arr.length - 2 ; i >= 0 ; i--) {
       for(int j = i + 1 ; j < arr.length; j++) {
           mat[i][j] = Math.max(mat[i][i] - mat[i+1][j] , mat[j][j] - mat[i][j-1]);
       }
   }
   return game(mat);
```

```
}
public static int[] game(int[][] matrix) {
   int player1 = 0, player2 = 0, games = 0;
   int left = 0, right = matrix.length-1;
   String p1_path = "" , p2_path = "";
   while (games != matrix.length) {
       if(left == right) {
           player2 += matrix[left][left];
           break;
       }
       if (matrix[left][right] == matrix[left][left] - matrix[left + 1][right]) {
           if(games % 2 == 0) { // Player1 turn
               player1 += matrix[left][left];
               p1_path += matrix[left][left] + " ";
           }
           else  { // Player2 turn
               player2 += matrix[left][left];
               p2_path += matrix[left][left] + " ";
           }
           left++;
       }
       else {
           if(games % 2 == 0) { // Player1 turn
               player1 += matrix[right][right];
               p1_path += matrix[right][right] + " ";
           }
           else {// Player2 turn
               player2 += matrix[right][right];
               p2_path += matrix[right][right] + " ";

           }
           right--;
       }
       games++;
   }
   return new int[] {player1,player2};
}

public static void main(String[] args) {
   System.out.println(Arrays.toString(dynamic(new int[] {1,3,6,1,3,6})));
   System.out.println(Arrays.toString(dynamic(new int[] {5,4,1,5,6,4})));
   System.out.println(Arrays.toString(dynamic(new int[] {4,1,5,9})));
}
```

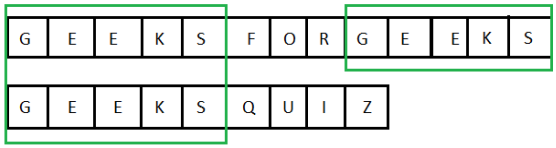
← **סיבוכיות זמן ריצה**: מפעילים תכנות דינאמי nאח לחישוב הרווחים ולכן $O(n^2)$.

LCS - תת המחרוזת המשותפת הארוכה ביותר

תיאור הבעיה

נתונות 2 מחרוזות X,Y.

נרצה למצוא את תת-המחרוזת המשותפת הארוכה ביותר.



OUTPUT: 5
As longest Common String is "Geeks"

דוגמא

עבור המחרוזות "$X = "abcade"$, $Y = "aebdc"$ תוחזר המחרוזת "abc".

פתרון הבעיה

עבור בעיה זו נציג 3 אלגוריתמים שונים.

מימוש בגיטהאב

- אפשרות א' - אלגוריתם חמדני
  - ⇐ נעבור על המחרוזת X, נתחיל לחפש את המופע של האות הראשונה במחרוזת Y.
  - ⇐ אם מצאנו, נמשיך לאות הבאה ב-X ונחפש אותה החל מאותו מקום בו עצרנו ב-Y עד שנגיע לסוף של Y.

```
public static String greedy(String X, String Y) {
   String ans = "";
   int i = 0, j = 0;

   while( i < X.length() && j < Y.length() ) {
     if(X.charAt(i) == Y.charAt(j)) {
         ans += X.charAt(i);
         i++;
     }
     j++;
   }
   return ans;
}
public static void main(String[] args) {
   String X = "ababcb", Y = "cbab";
   System.out.println(greedy(X,Y)); // prints “ab”
   System.out.println(greedy(Y,X)); // prints “cb”
}
```

- **נכונות השיטה:** השיטה לא מחזירה את התשובה הנכונה תמיד כי לא תמיד האיבר הראשון שנמצא הוא חלק מהסדרה. בנוסף, לפעמים כדאי לוותר על מספר איברים כדי לקחת אחרים טובים יותר.
  לדוגמא, אם היינו מפעילים את האלגוריתם על המחרוזות:
  $X = "axxxb"$, $Y = "bxxxa"$ אז היינו מקבלים תת-מחרוזת באורך 1.

- **סיבוכיות השיטה:** $O(n \cdot m)$- רק במקרה הגרוע - כי אם אין התאמה כלל בין X ל-Y אז עבור כל אות ב-X נצטרך לעבור על כל Y.

❖ אפשרות ב' - אלגוריתם חמדני משופר

⇚ נבנה מערך עזר בגודל 26 (מספר האותיות באנגלית) על המחרוזת הקצרה ביותר (נניח X) אשר ייתן אינדיקציה איזה אותיות קיימות במחרוזת.

⇚ נפעיל את הקוד של החמדני כאשר נעבור על מחרוזת Y.

⇚ אם האות קיימת, אז נבדוק את מיקומו ונוסיף ל-ans (מחרוזת הפתרון) ואחר כך נוריד במיקום של האות במערך --1 .

⇚ הרעיון הוא לא לחפש את האותיות של Y של כבר לא מופיעות ב-X.

לדוגמה עבור המחרוזות $X = "abca"$ו-$Y = "adcbc"$, נבנה מערך עזר עבור X מאחר והוא המינימלי ביותר. הבנייה מסתמכת לפי ASCII כאשר בטבלה הערך של 'a' הוא 97, ולכן כך נתאים את האותיות והסדר שלהם לאינדקסים במערך בגודל 26, לכן עבור המקרה שלנו:

'a' - 97 = **0**	'b' - 97 = **1**	'c' - 97 = **2**	'd' - 97 = **3**
2	1	1	0

מימוש אלגוריתם חמדני משופר

```
public static String improvedGreedy(String x,String y) {
   int[] occurrences = new int[26];

   for (int i = 0; i < y.length(); i++)
       occurrences[y.charAt(i)-'a']++;

   String answer = "";
   int limit = 0;

   for(int i = 0; i < x.length(); i++) {
       int place = (x.charAt(i)-'a');
       if(occurrences[place]>0) {
           int index = y.indexOf(x.charAt(i),limit);
           if(index != -1) {// if the char does occur
```

```
                answer += x.charAt(i);
                limit = index + 1; // reduce the limit
                occurrences[place]--;
            }
        }
    }
    return answer;
}
public static void main(String[] args) {
    String X = "cbab", Y = "ababcb";
    System.out.println(improvedGreedy(X,Y));
}
```

- **נכונות השיטה**: כמו באלגוריתם הקודם, אם היינו מפעילים את האלגוריתם על המחרוזות: $X = "axxxb",\ Y = "bxxxa"$ אז היינו מקבלים תת-מחרוזת באורך 1.

- **סיבוכיות**: אנחנו רצים על כל מחרוזת פעם אחת בלבד ולכן $O(m + n)$ אבל אנחנו צריכים לעבור על אחת המחרוזות פעם נוספת כדי למלא את המערך ולכן נוסיף $O(min(m, n))$ אנו מבינים שעדיף לנו למלא את המערך במחרוזת הקצרה יותר.
  לכן סה"כ נקבל: $O(n + m) + O(min(n, m))$.

❖ אפשרות ג' - חיפוש שלם

⇚ הרעיון הוא לקחת כל תתי-מחרוזות של X ו- Y ולחפש מחרוזת משותפת ארוכה ביותר.

⇚ אלגוריתם לבניית כל תתי-מחרוזות של מחרוזת נתונה. מספר תתי-מחרוזות של מחרוזת בגודל n שווה ל- $2^n - 1$ (לא כולל המחרוזת הריקה).

נסתכל על המחרוזת כביטים, לדוגמה עבור $X = abc$, n=3, מספר כל תתי המחרוזות הוא $2^3 - 1 = 7$.

	a	b	c	תת המחרוזת
1	0	0	1	c
2	0	1	0	b
3	0	1	1	bc
4	1	0	0	a
5	1	0	1	ac
6	1	1	0	ab
7	1	1	1	abc

עבור כל מילה ניצור מערך של מחרוזות subsets בגודל $2^n - 1$ ונמלא אותה בכל תתי המחרוזות האפשריים עבור אותה המחרוזת. נטייל בלולאה בגודל המערך ועל כל איטרציה אנחנו נפעל בדיוק כמו שתיארנו בדוגמה של הטבלה מלמעלה.

כלומר, אנחנו ניצור מערך מספרי מאופס binary בגודל של המחרוזת שאיתה אנחנו מתעסקים ועליה נבצע אותם פעולות כמו שהראינו בטבלה.

על כל איטרציה בלולאה שלנו נחבר נוסיף 1 (**plusOne**) למספר הבינארי שלנו (המערך המספרי) בדיוק כמו בטבלה וזה ייצג את תת המחרוזת שאותה נייצר וכך הלאה...

מימוש חיפוש שלם:

```
public static void plusOne(int[] binary) {
   int size = binary.length - 1;
   while(size >= 0 && binary[size] == 1) {
       binary[size--] = 0;
   }
   if(size >= 0)
       binary[size] = 1;
}

public static String[] subsets(String str) {
   int array_size = ((int) (Math.pow(2, str.length())))-1;
   String[] subsets = new String[array_size];
   int[] binary = new int[str.length()];

   for(int i = 0; i < subsets.length; i++) {
       plusOne(binary);
       String subset = "";

       for (int j = 0; j < binary.length; j++) {
           if(binary[j] == 1)
               subset += str.charAt(j);
       }
       subsets[i] = subset;
   }
   return subsets;
}

public static String bruteForce(String X, String Y) {
   String shortest = X, longest= Y, ans = "";
   if(X.length() > Y.length()){
       shortest = Y;
       longest= X;
   }
```

```
   String[] shortestSet = subsets(shortest);
   String[] longestSet = subsets(longest);

   for(int i = 0; i < shortestSet.length; i++) {
       for(int j = 0; j < longestSet.length; j++) {

           if(shortestSet[i].equals(longestSet[j])) {
               if(shortestSet[i].length() > ans.length())
                   ans = shortestSet[i];
           }
       }
   }
   return ans;
}

public static void main(String[] args) {
   String X = "axxxb", Y = "bxxxa";
   System.out.println(bruteForce(X,Y));
}
```

- **סיבוכיות**: נחשב את תתי המחרוזות של X כלומר $O(2^m)$.
  נעבור בלולאה כפולה עם מספר תתי המחרוזות של X כלומר 2^mכפול מספר תתי המחרוזות של Y כלומר וביחד $2^n \cdot 2^m = 2^{n+m}$.
  וגם מספר ההשוואות הוא $min(n,m)$ כי בודקים שוויון עד האורך של המחרוזת הקצרה ביותר. סה"כ נקבל: $O(2^{n+m} \cdot min(n,m) + 2^m) = O(2^{n+m} \cdot min(n,m))$.
- **נכונות השיטה**: בודקים את כל האפשרויות ולכן בהכרח נגיע גם לתשובה הנכונה.

נציג פתרון דומה עבור חיפוש שלם והפעם בשיטת פעולות מתמטיות להמרה בינארית.
בדומה למימושים ולשיטות שהצגנו עבור הנושא של חישוב חזקות בעזרת מספרים בינאריים,
ניצור מחלקה שרצה על כל ה-$2^n - 1$איברים במערך ועל כל אינדקס בריצה נבצע המרה למספר בינארי באמצעות חילוק האינדקס ב-2 ובקבלת שארית 1 נצרף את נשרשר את התו במיקום הזה עד קבלת תת המערך עבור מקרה זה.

```
public static String[] subset(String str) {
   String[] subsets = new String[(int)Math.pow(2,str.length())];

   for(int decimal = 0 ; decimal < subsets.length; decimal++) {
       String subset = "";
       int binary = decimal, i = 0;
       while(binary != 0 ) {
```

```
            if(binary % 2 == 1) {
                subset += str.charAt(i);
            }
            binary /= 2;
            i++;
        }
        subsets[decimal] = subset;
    }

    return subsets;
}


public static String bruteForce(String X, String Y) {
    String ans = "", shortest = X, longest = Y;
    if(X.length() > Y.length()) {
        shortest = Y;
        longest = X;
    }

    String[] shortestSubset = subset(shortest);
    String[] longestSubset = subset(longest);

    for(int i = 0 ; i < shortestSubset.length ; i++) {
        for(int j = 0; j < longestSubset.length; j++) {

            if(shortestSubset[i].equals(longestSubset[j])) {
                if(shortestSubset[i].length() > ans.length()) {
                    ans = shortestSubset[i];
                }
            }
        }
    }
    return ans;
}

public static void main(String[] args) {
    String X = "abcbdab", Y = "bdcaba";
    System.out.println(bruteForce(X,Y)); // bcba
}
```

הסיבוכיות שעד עכשיו היה לנו הם:

$O(n \cdot m)$ עבור חיפוש חמדני.

$O(n + m)$ עבור חיפוש חמדני משופר.

$O(2^{n+m} \cdot min(n,m))$ עבור חיפוש עץ שלם.

ננסה לפתח פתרון יותר יעיל שמשלב את היעילות של החיפוש החמדן ואת הפתרון האופטימלי שהחיפוש השלם מספק לבעיה - וכך נקבל את הפתרון הטוב ביותר לבעיה שלנו.

שיטה חדשה 1

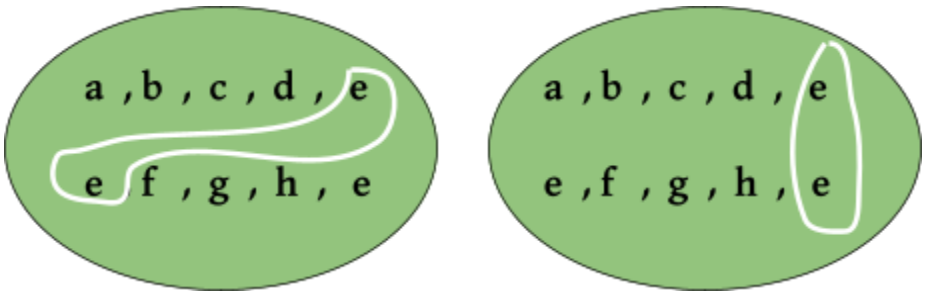
$$\overline{X} = \{x_1, x_2, \ldots, x_n\} \ , \ \overline{Y} = \{y_1, y_2, \ldots, y_m\}$$
$$LCS(\overline{X}, \overline{Y})$$

יהא $LCS(\overline{X} \cdot \alpha \ , \overline{Y} \cdot \beta)$ כך ש- $\alpha = \beta$.

במקרה זה, באופן הכי פשוט, $LCS(\alpha, \beta) = 1$ לכן מתקיים $LCS(\overline{X} \cdot \alpha \ , \overline{Y} \cdot \beta) \geq LCS(\overline{X}, \overline{Y}) + 1$.

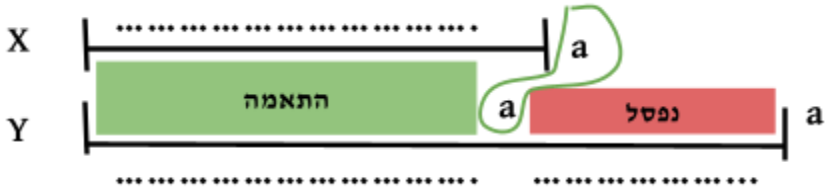
כלומר הוספנו לכל מחרוזת תו זהה ולכן יש לנו לפחות התאמה אחת, אין מצב שהיו פחות התאמות.

למשל באיור הבא אפשר לראות שני אפשרויות להתאמה:



כלומר האורך של כל התאמה תהיה לפחות אחת.

אבל מה קורה שיש התאמה בין αחיצוני ל-βפנימית?



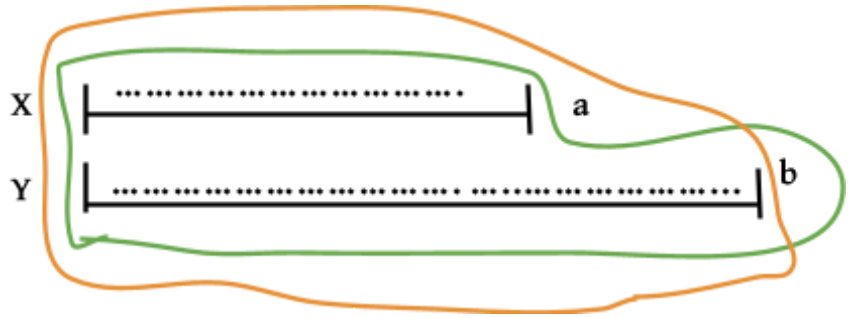
אפשר לראות שקיבלנו התאמה של אותו תו αעוד לפני שנפגשנו ב-$\alpha = \beta$המתוכנן שלנו.

פסלנו את כל שאר התווים החל אותו תו שבו מצאנו התאמה.

שיטה חדשה 2

יהא $LCS(\overline{X} \cdot \alpha\ ,\ \overline{Y} \cdot \beta)$ כך ש- $\alpha \neq \beta$.

במקרה זה נוכל לתת פתרון לבעיה בשיטה הקודמת, נתבונן באיור הבא:



אם $\alpha \neq \beta$ אז אין התאמה המתבססת על α מול β אבל, מה שכן יכול לקרות לטובתנו:

או שזה $LCS(\overline{X},\ \overline{Y} \cdot \beta)$ או שזה $LCS(\overline{X} \cdot \alpha,\ \overline{Y})$ כלומר או ש-α שותפה ו-β לא, או להפך...

ומאחר ומדובר בתת המחרוזת המקסימלית ביותר נבחר:

$$MAX\ (\ LCS(\overline{X} \cdot \alpha,\ \overline{Y})\ ,\ LCS(\overline{X},\ \overline{Y} \cdot \beta)\ )$$

כלומר עבור 2 השיטות שהצגנו:

$$LCS(\overline{X} \cdot \alpha, \overline{Y} \cdot \beta) = \begin{cases} LCS(\overline{X}, \overline{Y}) + 1 & \alpha = \beta \\ max(\ LCS(\overline{X} \cdot \alpha, \overline{Y})\ ,\ LCS(\overline{X}, \overline{Y} \cdot \beta)\ ) & \alpha \neq \beta \end{cases}$$

לסיכום, נציג בפסאודו-קוד את המקרים הבאים:

```
if α == β
  then LCS(X̄ · α , Ȳ · β) = LCS(X̄, Ȳ) + 1
else if α! = β
  then LCS(X̄ · α , Ȳ · β) = max ( LCS(X̄ · α, Ȳ) , LCS(X̄, Ȳ · β) )
```

❖ אפשרות ד' עבור LCS - תכנות דינאמי

⇐ בעזרת 2 השיטות שהצגנו מקודם נבנה שיטת פתרון חדשה.

⇐ בחיפוש השלם, בדקנו תתי מחרוזות מיותרות כי תת המחרוזת המשותפת הארוכה ביותר החל מאיבר כלשהו היא לפחות אותה תת מחרוזת אם נחשב החל מתחילת המחרוזות.

⇐ כלומר ראינו שעבור 3 האפשרויות עד כה לא באמת קיבלנו פתרון אופטימלי.

⇐ נרצה לפתור את הבעיה באופן יעיל ביותר (מהיר) כמו בשיטת החמדן אך כמובן שגם נרצה לפתור באופן אופטימלי (נכון) כמו בשיטת החיפוש השלם.

⇐ התכנות הדינאמי מספק לנו את הפתרון היעיל והאופטימלי ביותר עבור LCS.

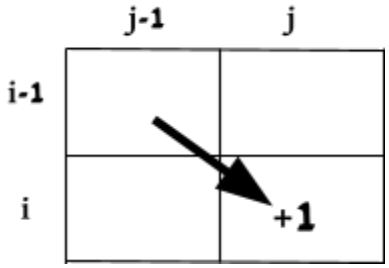
נראה שפתרון זה יעיל בסיבוכיות זמן ריצה של $O(n \cdot m)$.

תכנון הפתרון

נממש בצורה אינדוקטיבית עם מבנה נתונים של מערך דו-מימדי.

עבור שיטה 1 כאשר $\alpha = \beta$ נראינו ש- $LCS(\overline{X} \cdot \alpha, \overline{Y} \cdot \beta) = LCS(\overline{X}, \overline{Y}) + 1$.

לכן כאשר $\overline{X}[i] = \overline{Y}[j]$ נפעל כך: $arr[i][j] = arr[i-1][j-1] + 1$, כמו בציור:

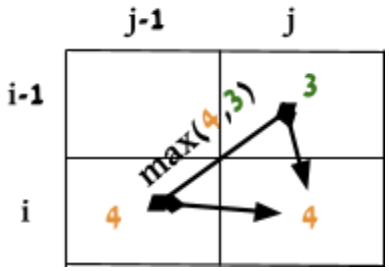


ועבור שיטה 2 כאשר $\alpha \neq \beta$ נראינו ש- $LCS(\overline{X} \cdot \alpha, \overline{Y} \cdot \beta) = max(LCS(\overline{X} \cdot \alpha, \overline{Y}), LCS(\overline{X}, \overline{Y} \cdot \beta))$

לכן כאשר $\overline{X}[i] \neq \overline{Y}[j]$ נבחר את המקסימום מבין ה-2:

$arr[i][j] = max(arr[i][j-1], arr[i-1][j])$

כמו בציור:



ובסוף התהליך, נקבל את המטריצה הבאה:

			0	1	2	3	4	5	6
		$\overline{X}$	a	b	c	b	d	a	b
	$\overline{Y}$		0	0	0	0	0	0	0
0	b	0	0	1	1	1	1	1	1
1	d	0	0	1	1	1	2	2	2
2	c	0	0	1	2	2	2	2	2
3	a	0	1	1	2	2	2	3	3
4	b	0	1	2	2	3	3	3	4
5	a	0	1	2	2	3	3	4	4

ניתן לראות כי קיימות קיימות כאן 3 אפשרויות של LCS כך שכל אחת מהן באורך 4.

מימוש **אינדוקטיבי**

```
public static int[][] generateMatrix(String X, String Y) {

   int[][] matrix = new int[X.length() + 1][Y.length() + 1];
   for(int i = 0 ; i < matrix.length; i++)
       matrix[i][0] = 0;
   for(int i = 0 ; i < matrix[0].length; i++)
       matrix[0][i] = 0;

   for(int i = 1 ; i < matrix.length ; i++) {
       for(int j = 1 ; j < matrix[0].length; j++) {

           if(X.charAt(i-1) == Y.charAt(j-1)) {
               matrix[i][j] = matrix[i-1][j-1] + 1;
           } else {
               matrix[i][j] = Math.max(matrix[i][j-1],matrix[i-1][j]);
           }
           System.out.print(matrix[i][j] + " ");
       }
       System.out.println();
   }
```

```
    return matrix;
}

public static String dynamic(String X, String Y) {
    String ans = "";
    int[][] matrix = generateMatrix(X,Y);
    int length = matrix[X.length()][Y.length()];
    int i = X.length(), j = Y.length();
    while(length > 0) {
        if(X.charAt(i-1) == Y.charAt(j-1)) {
            ans = X.charAt(i-1) + ans;
            i--;
            j--;
            length--;
        } else if(matrix[i][j-1] > matrix[i-1][j]) {
            j--;
        } else {
            i--;
        }
    }
    return ans;
}

public static void main(String[] args) {
    String X = "abcbdab", Y = "bdcaba";
    System.out.println(dynamic(X,Y)); // bcba
}
```

- **סיבוכיות**: עבור האתחול של עמודת אפסים לכל מילה (כדי שנדע מתי לעצור כשנחזיר את הפתרון) הוא $O(n)+O(m)$ ועבור הלולאה הקנונית נקבל $O(n\cdot m)$.
  ולסיכום $O(n)+O(m)+O(n\cdot m)=O(n\cdot m)$.

- לכן הבאנו פתרון משולב מהצד החמדני לצד עץ החיפוש השלם כלומר:

$$O(n+m) < O(n\cdot m) < O(2^{n+m}\cdot min(n,m))$$

מימוש **רקורסיבי**

```
public static int[][] generateMatrix(String X, String Y) {
       int[][] matrix = new int[X.length()+1][Y.length()+1];
       for(int i = 0 ; i < matrix.length; i++)
           matrix[i][0] = 0;
       for(int i = 0 ; i < matrix[0].length; i++)
           matrix[0][i] = 0;
       generateRec(matrix, X, Y, 1,1);
       return matrix;
   }

public static void generateRec(int[][] matrix,String X,String Y, int i ,int j) {
       if(i == matrix.length)
           return;
       if(j == matrix[0].length) {
           generateRec(matrix, X, Y, i + 1, 1);
       }
       else {
           if (X.charAt(i - 1) == Y.charAt(j - 1))
               matrix[i][j] = matrix[i - 1][j - 1] + 1;
           else
               matrix[i][j] = Math.max(matrix[i][j - 1], matrix[i - 1][j]);

           generateRec(matrix, X, Y, i, j + 1);
       }
   }

public static String dynamicRec(int[][] matrix,String X,String Y, int i,int j, int
length) {
       if(length == 0)
           return "";
       if(X.charAt(i-1) == Y.charAt(j-1))
           return dynamicRec(matrix,X,Y,i-1,j-1,length-1) + X.charAt(i-1);
       if(matrix[i][j-1] > matrix[i-1][j])
           return dynamicRec(matrix,X,Y,i,j-1,length);

       return dynamicRec(matrix,X,Y,i-1,j,length);
   }
public static String dynamic(String X, String Y) {
       int[][] matrix = generateMatrix(X,Y);
       int n = X.length(), m = Y.length();
       return dynamicRec(matrix, X, Y, n, m, matrix[n][m]);
   }
public static void main(String[] args) {
       String X = "abcbdab", Y = "bdcaba";
```

```
        System.out.println(dynamic(X,Y)); // bcba
    }
```

אם נרצה להחזיר את **כל** ה-LCS שלנו (יכול להיות שקיימות כמה מחרוזות LCS שונות באותו האורך).
נצרף את הפונקציות הבאות למימוש הרקורסיבי שהצגנו בדף הקודם.

```
public static void getAllRec(int[][] matrix, String X, String Y, int i , int j,
HashSet<String> set) {
    if(i == 0 || j == 0)
        return;

    int n = X.length(), m = Y.length();
    if(matrix[i][j] == matrix[n][m]) {
        set.add(dynamicRec(matrix,X,Y,i,j,matrix[n][m]));

        getAllRec(matrix,X,Y,i-1,j,set);
        getAllRec(matrix,X,Y,i,j-1,set);
        getAllRec(matrix,X,Y,i-1,j-1,set);
    }
}

public static HashSet<String> getAllLCS(String X, String Y) {
    int[][] matrix = generateMatrix(X,Y);
    int n = X.length(), m = Y.length();
    HashSet<String> set = new HashSet<>();
    getAllRec(matrix, X, Y, n, m, set);
    return set;
}

public static void main(String[] args) {
    String X = "abcbdab", Y = "bdcaba";
    HashSet<String> set = getAllLCS(X,Y);
    for (String str : set) {
        System.out.print(str + " , ");
    }
}
```

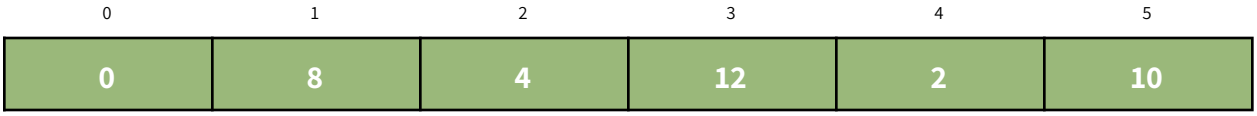
LIS - תת-הסדרה העולה הארוכה ביותר

תיאור הבעיה

נתון מערך של מספרים. יש למצוא את אורך תת הסדרה העולה הארוכה ביותר מתוך המערך.

דוגמה

עבור הסדרה במערך:



נקבל את הפתרונות הבאים:

$$0, 8, 12 \quad OR \quad 0, 4, 10 \quad OR \quad 0, 4, 12 \quad OR \quad 0, 8, 10 \quad OR \quad 0, 2, 10$$

פתרון הבעיה

מימוש בגיטהאב

- ❖ אפשרות א' - אלגוריתם חמדני
  - ⇚ נקבע שהאיבר הראשון יהיה תחילת הסדרה.
  - ⇚ נעבור על המערך וניקח בכל שלב את האיבר הבא בגודלו עד שנגיע לסוף המערך.
  - ⇚ לדוגמא, עבור הסדרה הבאה: 1, 3, 4, 2אנחנו ניקח את 1, אחר כך ניקח את 3 שגדול מהמספר הקודם ולבסוף ניקח את 4 כי אחריו אין איברים שגדולים ממנו ולכן נקבל את הפתרון הבא: 1, 3, 4

```
public static Stack<Integer> greedy(int[] arr) {
   Stack<Integer> stack = new Stack<>();
   stack.push(arr[0]);
   for(int i = 1; i < arr.length; i++) {
       if(stack.peek() < arr[i])
           stack.push(arr[i]);
   }
   return stack;
}

public static void main(String[] args) {
   int[] arr = {1,3,4,2};
   System.out.println(greedy(arr)); // [1, 3, 4]
}
```

⇚ **סיבוכיות**: אנחנו עוברים פעם אחת על כל המערך ולכן $O(n)$.

- **נכונות השיטה**: אלגוריתם זה יחזיר את הסדרה העולה הראשונה שימצא ולא בהכרח הארוכה ביותר בסדרה ולכן זה לא אלגוריתם טוב.

❖ אפשרות ב' - אלגוריתם חמדני משופר

⇚ נרוץ על המערך ועבור על איבר נפעיל את האלגוריתם החמדני שהצגנו באפשרות א'.

⇚ בסוף כל הפעלה נבצע השוואה ונמצא את תת הסדרה העולה הארוכה ביותר מבין כל האיטרציות שביצענו.

⇚ לדוגמה, נתבונן בסדרה $1,\ 100,\ 101,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7$.

נקבל תחילה את הסדרה $1, 100, 101$ כך ש-1 הוא האיבר הראשון.

אחר כך נמשיך ונקבע את 100 להיות האיבר הראשון ואז נקבל סדרה $100\ ,\ 101$ ככה נמשיך עד שנקבל את ה-LIS שלנו שהוא במקרה זה $1, 2,\ 3,\ 4,\ 5,\ 6,\ 7$.

```java
public static Stack<Integer> greedy(int[] arr, int start) {
   Stack<Integer> stack = new Stack<>();
   stack.push(arr[0]);
   for(int i = start+1; i < arr.length; i++) {
       if(stack.peek() < arr[i])
           stack.push(arr[i]);
   }
   return stack;
}

public static Stack<Integer> improved(int[] arr) {
   Stack<Integer> stack = new Stack<>();

   for(int i = 0; i < arr.length ; i++) {
       Stack<Integer> temp_stack = greedy(arr,i);
       if(temp_stack.size() > stack.size())
           stack = temp_stack;
   }

   return stack;
}

public static void main(String[] args) {
   int[] arr = {1, 100, 101, 2, 3, 4, 5, 6, 7}; // [1, 2, 3, 4, 5, 6, 7]
   System.out.println(improved(arr));
}
```

- **סיבוכיות**: על כל איבר חוזרים ובודקים את המשך המערך החל ממנו ולכן $O(n^2)$.
- **נכונות השיטה**: האלגוריתם לא החזיר את התשובה הנכונה כי לא פתרנו את הבעיה שלפעמים כדאי לוותר על מספר איברים כדי לקחת אחרים טובים יותר.

❖ אפשרות ג' - אלגוריתם באמצעות LCS

⇐ נשתמש באלגוריתם של LCS (מציאת תת-המחרוזת המשותפת הארוכה ביותר בין 2 מחרוזות).

⇐ נשכתב את האלגוריתם שיתאים ל-2 מערכים של מספרים ונפעיל אותו על המערך הנתון ועל המערך הנתון לאחר מיון כלומר $LCS(arr,\ Sort(arr))$.

⇐ האלגוריתם עובד בשיטת תכנות דינאמי של LCS.

```
public static int[] LCS(int[] X) {// O(n^2)).

   int[] Y = new int[X.length];

   for(int i = 0; i < X.length; i++)
       Y[i] = X[i];

   Arrays.sort(Y);// sort s_arr O(n*log(n)
   int[][] matrix = new int[X.length+1][Y.length+1];
   generateMatrix(matrix,X,Y,1,1);
   int i = matrix.length - 1;
   int j = matrix.length - 1;
   int end = matrix[i][j];
   int start = 0;
   int[] solution = new int[end];

   while(start < end) {
       if(X[i-1] == Y[j-1]) {
           solution[end-start-1] = X[i-1];
           i--;
           j--;
           start++;
       }
       else if(matrix[i-1][j] >= matrix[i][j-1]) {
           i--;
       }
       else {
           j--;
       }
   }

   return solution;
}

public static void generateMatrix(int[][] matrix, int[] X, int[] Y, int i, int j) {
   if(i == matrix.length)
       return;
   if(j == matrix.length) {
       System.out.println();
       generateMatrix(matrix,X,Y,i+1,1);
```

```
    } else {
        if(X[i-1] == Y[j-1]) {
            matrix[i][j] = matrix[i-1][j-1] + 1;
        }else {
            matrix[i][j] = Math.max(matrix[i-1][j], matrix[i][j-1]);
        }
        System.out.print(matrix[i][j] + " ");
        generateMatrix(matrix,X,Y,i,j+1);
    }
}

public static void main(String[] args) {
    int[] arr = {1, 100, 101, 2, 3, 4, 5, 6, 7};
    System.out.println(Arrays.toString(LCS(arr)));
    // prints [1, 2, 3, 4, 5, 6, 7]
}
```

- **סיבוכיות**: סיבוכיות ההעתקה$O(n)$, סיבוכיות המיון $O(nlogn)$וסיבוכיות בניית המטריצה היא $O(n^2)$ ולכן סה"כ נקבל $O(n^2 + nlogn)$.

- **נכונות השיטה**: מכיוון שהמטרה היא למצוא תת סדרה עולה ארוכה ביותר אז בהכרח התשובה היא תת סדרה של המערך הממוין כך ששומרים על סדר האיברים במערך ולכן תת הסדרה המשותפת בין 2 המערכים היא בדיוק התשובה.

- **דוגמא**:

בהינתן סדרת מספרים (מערך) : $arr = \{1, 100, 101, 2, 3, 4, 5, 6, 7\}$

נמיין את המערך $s\_arr = \{1, 2, 3, 4, 5, 6, 7, 100, 101\}$

נשלח ליצירת מטריצה ונקבל: (השורה זה המערך המקורי והעמודה זה הממוין):

1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 2 3
1 2 2 2 2 2 2 2 3
1 2 3 3 3 3 3 3 3
1 2 3 4 4 4 4 4 4
1 2 3 4 5 5 5 5 5
1 2 3 4 5 6 6 6 6
1 2 3 4 5 6 7 7 7

לאחר מכן נשלוף את הפתרון הארוך ביותר כלומר ניצור מערך פתרון באורך 7 ובכל פעם

שיש לנו התאמה בין המערך הממוין למערך המקורי נצרף למערך הפתרון את ההתאמה ההתאמה הזאת.
הפתרון יתקבל באמת בסדר עולה מכיוון שאנו מכניסים את ההתאמות החל מהמקום nאח במטריצה ונרוץ "באלכסון" עד ההתאמה האחרונה.
לבסוף נקבל את הפתרון הבא: [1, 2, 3, 4, 5, 6, 7].

❖ אפשרות ד' - אלגוריתם באמצעות חיפוש שלם

⇚ נייצר את כל תתי המערכים האפשריים.

⇚ נבדוק עבור על תת מערך אם הוא תת סדרה עולה (מתחילתו ועד סופו).

⇚ אם כן, נבדוק האם הוא הכי ארוך שמצאנו עד עכשיו ונשמור אותו.

⇚ הפונקציה תחזיר בסוף מערך של תת הסדרה העולה הגדולה ביותר.

```
public static Vector<int[]> getSubsets(int[] arr) {
   Vector<int[]> subsets = new Vector<>();
   int size = (int)Math.pow(2,arr.length) - 1;

   for(int decimal = 0; decimal < size; decimal++) {

       int binary = decimal;
       int i = 0;
       Vector<Integer> vec_set = new Vector<>();
       while(binary > 0) {
           if(binary % 2 == 1) {
               vec_set.add(arr[i]);
           }
           binary /= 2;
           i++;
       }
       int[] arr_set = new int[vec_set.size()];
       for(int s = 0 ; s < vec_set.size(); s++) {
           arr_set[s] = vec_set.get(s);
       }
       subsets.add(arr_set);
   }

   return subsets;
}

public static int[] bruteForce(int[] arr) {

   Vector<int[]> subsets = getSubsets(arr);
   int[] solution = new int[0];
   int max = 0;
```

```
    for(int i = 0 ; i < subsets.size(); i++) {
        int[] subset = subsets.get(i);
        boolean isIncreasing = true;
        for(int j = 1 ; j < subset.length; j++) {
            if(subset[j-1] > subset[j]) {
                isIncreasing = false;
                break;
            }
        }
        if(isIncreasing && subset.length > max) {
            solution = subset;
            max = solution.length;
        }
    }

    return solution;
}


public static void main(String[] args) {
    int[] arr = {1, 100, 101, 2, 3, 4, 5, 6, 7};
    System.out.println(Arrays.toString(bruteForce(arr)));
    //prints [1, 2, 3, 4, 5, 6, 7]
}
```

- **סיבוכיות**: מספר תתי הקבוצות הוא 2^n וגם על כל תת-מערך עוברים ובודקים האם הוא ממוין בסדר עולה כך שלכל היותר תת מערך כזה הוא בגודל n ולכן סה"כ $O(2^n \cdot n)$.
- **נכונות השיטה**: בודקים את כל האפשרויות ולכן בהכרח נגיע גם לתשובה הנכונה.

שאלה

האם אנחנו רוצים למצוא את אורך המחרוזת או רק דוגמא למחרוזת המקיימת LIS?

נפצל את השאלה ל-2 אפשרויות:

❖ אפשרות ה' - מציאת אורך המחרוזת.

⇐ נדרוס איברים תוך כדי שמירה על האורך הנכון.

דוגמה

למשל עבור 10, 12, 20, 21, 16, 17, 18נבחר את 10, 12, 20, 21.

מאחר והאיבר הבא אחרי 21 הוא 16 והוא נוגד את ה-LIS במידה ונצרף אותו לסדרה הנבחרת, אז "נדרוס" את האיברים כלומר נדרוס את 20 ונצרף במקומו את 16.

כלומר נקבל את הסדרה הבאה: 10, 12, 16, 21.

בכל פעם שאנו מוסיפים איבר אנו בודקים שהוא יותר גדול מכל האיברים הנמצאים משמאלו. אנו חייבים לשמור על אורך הסדרה הנבחרת בכל ביצוע של דריסה אלא אם צירפנו איבר שהוא "לטובתנו" כלומר שומר על סדר עולה ותקין ואז נרחיב את גודל הסדרה בהתאם.
לבסוף נקבל את הסדרה הבאה 10, 12, 16, 17, 18.

⇐ עבור דוגמה זו האורך נכון והסדרה שהתקבלה חוקית, אבל **לא תמיד נקבל סדרה חוקית.**

דוגמה

נציג סדרה: 5, 9, 4, 20, 6, 3, 7, 8, 11.
כאן הסדרה שתתקבל היא לא חוקית כי קיבלנו 3, 6, 7, 8, 11ובסדרה המקורית 6, 3, 7, 8, 11
אך עדיין, קיימת תת סדרה שהאורך שלה הוא באורך הזה: 4, 6, 7, 8, 11 או 5, 6, 7, 8, 11.
כלומר, אנחנו לא תמיד נקבל את תת הסדרה הארוכה ביותר (מבחינת ערכים חוקיים) אבל זה לא משנה, כי הדרישה היא להחזיר את האורך הגדול ביותר וזה כן יהיה בידיים שלנו.

```
public static int binarySearch(int[] sequence, int left, int right, int value) {

   while(right - left > 1) { // while we can compare min two values

       int middle = (right+left)/2;

       if(value <= sequence[middle]) {
           right = middle;
       }
       else if(value > sequence[middle]) {
           left = middle;
       }
   }
   return right;
}

public static int length(int[] arr) {
   int[] sequence = new int[arr.length];
   sequence[0] = arr[0];

   int length = 0;
   for(int i = 1 ; i < sequence.length; i++) {

       if(arr[i] < sequence[0]) {
           sequence[0] = arr[i];
       }
       else if(arr[i] > sequence[length]) {
           length++;
           sequence[length] = arr[i];
```

```
        } else {
            int index = binarySearch(sequence,0,length,arr[i]);
            sequence[index] = arr[i];
        }
    }
    return length+1;
}

public static void main(String[] args) {
    int[] arr = {5,9,4,20,6,3,7,8,11};
    System.out.println(length(arr));
}
```

↞ **סיבוכיות**: על כל איבר בסדרה נחפש איפה לשים אותו בתת הסדרה שאנו יוצרים.
נעשה זאת בחיפוש בינארי כדי לחסוך, סה"כ $O(n \cdot logn)$.

❖ אפשרות ו' - החזרת תת הסדרה בתכנות דינאמי.

⇚ ניצור מטריצה חדשה nxn (גודל הסדרה x גודל הסדרה).

⇚ נאתחל את ראש המטריצה $matrix[0][0] = arr[0]$ונתחיל למלא את המטריצה בלולאה.

⇚ אם הערך $arr[i]$לא גדול מהאיבר האחרון שנוסף למטריצה ולא קטן מהאיבר הראשון במטריצה, אז נחפש את הערך המתאים בתוך המטריצה בעזרת חיפוש בינארי שיחזיר לנו את האינדקס הדרוש להצבת הערך החדש $arr[i]$.

⇚ כל שורה במטריצה תגדיר לנו את הטווח החדש שנוסף והאיברים החוקיים והמתאימים ביותר לפתרון הנחוץ.

⇚ בסוף נעתיק את השורה האחרונה לפי מספר הטווח למערך חדש בגודל הטווח שקיבלנו ונחזיר אותו.

דוגמה: עבור המערך 5, 9, 4, 20, 6, 3:

	0	1	2	...5
0	3	0	0	0
1	4	6	0	0
2	5	9	20	0
...5	0	0	0	0

ונחזיר את תת הסדרה **5,9,20**.

מימוש **אינדוקטיבי**:

```
public static int binarySearch(int[][] matrix, int left, int right, int value) {
   while(left <= right) {
       if(left == right) {
           return left;
       }
       int middle = (right+left)/2;
       if(value == matrix[middle][middle]) {
           return middle;
       }
       if(value < matrix[middle][middle]) {
           right = middle;
       }
       else {
           left = middle+1;
       }
   }
   return -1;
}

public static int[] dynamic(int[] arr) {
   int[][] matrix = new int[arr.length][arr.length];
   int length = 1;
   matrix[0][0] = arr[0];

   for(int i = 1; i < arr.length; i++) {
       int index;
       if(arr[i] > matrix[length-1][length-1]) {
           index = length;
       }
       else if(arr[i] < matrix[0][0]) {
           index = 0;
       } else {
           index = binarySearch(matrix,0,length,arr[i]);
       }
       matrix[index][index] = arr[i];

       if(index == length){
           length++;
       }

       for(int j = 0 ; j < index; j++) {
           matrix[index][j] = matrix[index-1][j];
       }// end j for

   }// end i for
```

```
    int[] solution = new int[length];
    for(int i = 0; i < length; i++) {
        solution[i] = matrix[length-1][i];
    }

    return solution;
}
public static void main(String[] args) {
    int[] arr = {5,9,4,20,6,3};
    System.out.println(Arrays.toString(dynamic(arr)));
}
```

מימוש **רקורסיבי**:

```
public static int[] dynamic(int[] arr) {
    int[][] matrix = new int[arr.length][arr.length];
    matrix[0][0] = arr[0];

    int length = recursive(matrix,arr,1,1);

    int[] solution = new int[length];
    fillSolution(matrix,solution,0);

    return solution;
}

public static int recursive(int[][] matrix, int[] arr, int length,
int i) {
    if(i == arr.length){
        return length;
    }

    int index;
    if(arr[i] > matrix[length-1][length-1]) {
        index = length;
    }
    else if(arr[i] < matrix[0][0]) {
        index = 0;
    } else {
        index = binarySearch(matrix,0,length,arr[i]);
    }
```

```
    matrix[index][index] = arr[i];
    newRow(matrix,index,0);

    if(index == length) {
        return recursive(matrix,arr,length+1,i+1);
    } else {
        return recursive(matrix,arr,length,i+1);
    }
}

public static int binarySearch(int[][] matrix, int left, int right,
int value) {

    if(left > right) {
        return -1;
    }
    if(left == right){
        return left;
    }

    int middle = (right+left)/2;

    if(value < matrix[middle][middle]) {
        return binarySearch(matrix,left,middle,value);
    }
    else if(value > matrix[middle][middle]){
        return binarySearch(matrix,middle+1,right,value);
    }
    else { // value == matrix[middle][middle]
        return middle;
    }
}

public static void newRow(int[][] matrix, int index, int i) {
    if(i == index) {
        return;
    }
    matrix[index][i] = matrix[index-1][i];
```

```
    newRow(matrix,index,i+1);
}

public static void fillSolution(int[][] matrix, int[] solution, int
i) {
    if(i == solution.length) {
        return;
    }
    solution[i] = matrix[solution.length-1][i];
    fillSolution(matrix,solution,i+1);
}
public static void main(String[] args) {
    int[] arr = {5,9,4,20,6,3};
    System.out.println(Arrays.toString(dynamic(arr)));
    // prints [5, 9, 20]
}
```

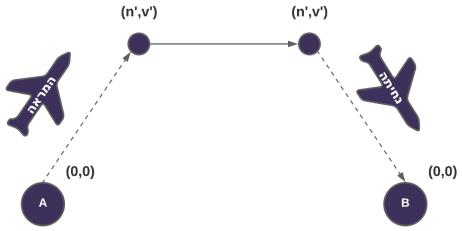
סיבוכיות זמן הריצה:

בכל שלב בלולאה i נבצע: חיפוש בינארי ($O(log(n))$ונעתיק במקרה הגרוע את כל השורה שמעליו $O(n)$

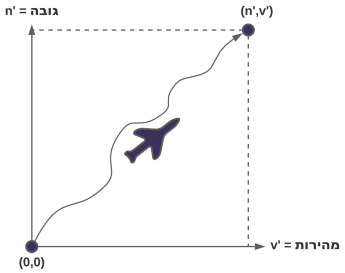
מאחר והלולאה באורך n אז הסיבוכיות היא ($O(n \cdot log(n) + n^2) = O(n \cdot (log(n) + n))$.

בעיית המטוס

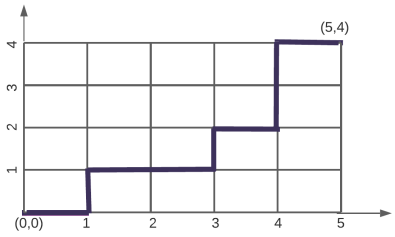
תיאור הבעיה



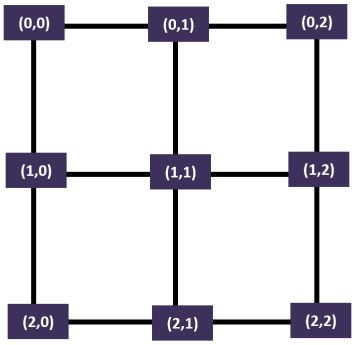
איך להמריא ולנחות בצורה הכי מהירה וזולה.
הטייס רוצה בשלב ההמראה לחסוך כמה שיותר דלק, אנחנו צריכים להגיד לטייס באיזה קצב לטוס בצורה שהכי תחסוך את הדלק.



הפרמטרים שלנו במקרה זה הם גובה ומהירות. אנחנו צריכים למצוא פונקציה שמשתנה בהתאם לשינויים של הפרמטרים בצורה הכי חסכונית שיש.אפשרות לפתרון הוא לפרק את הבעיה לתתי-תחומים לפי השינויים. אנחנו מתרגמים את הבעיה לצורה יותר "טכנית" וכך נוכל לפתור את זה בצורה פשוטה יותר.



נתונה מטריצה המייצגת לוח משבצות $(n\ x\ m)$כך שעבור כל מעבר בין 2 קודקודים סמוכים יש משקל כלשהו.
יש למצוא את המסלול עם העלות הנמוכה ביותר החל מקודקוד (0,0) עד קודקוד (n, m) כאשר מותר לצעוד למעלה או ימינה בלבד.המטריצה מיוצגת על-ידי מערך דו-מימדי של Nodes כאשר בכל Node יש x,y.



המטרה

למצוא את המסלול הקצר ביותר (המסלול בעל העלות המינימלית) מנקודה (0,0) לנקודה (M,N).כדי להיכנס לראש של מתכנת נהפוך את המטריצה שלנו בצורה הבאה כדי לפשט את הבעיה עוד יותר (האיור משמאל):

פתרון הבעיה

מימוש בגיטהאב

עבור כל אפשרות נשתמש באובייקט הבא *Node.java*:

- goRight, goDown - המחיר לגשת למטה או ימינה מהקודקוד הזה.
- entry - המחיר הטוב ביותר מנקודה (0,0) עד לקודקוד זה.
- numOfPaths - מספר המסלולים הקצרים ביותר עד לקודקוד הזה.
- entryFromTheEnd - כמו entry, רק שזה מהסוף עד לקודקוד.

```
public class Node {
   int goRight, goDown, entry, numOfPaths, entryFromTheEnd;
   public Node(int x, int y) {
       this.goRight = x;
       this.goDown = y;
       entry = 0;
       entryFromTheEnd = 0;
       numOfPaths = 1;
   }
}
```

- אפשרות א' - אלגוריתם חמדני.
  - נבחר בכל שלב את המעבר עם העלות הנמוכה יותר.

נתבונן במטריצה $matrix$ $Node[][]$ באיור מלמטה:

נתחיל מנקודת ההתחלה שלנו (0,0) ונרצה למצוא את המסלול הזול ביותר ל-(1,1).

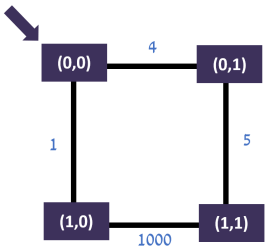
באופן חמדני נבחר "down" כי 4>1.

כעת אנו עומדים בנקודה (1,0), נותר לנו רק לבחור "right" ,

כלומר במשקל 1000 כדי להגיע ליעד.

קל לראות כמה החמדני לא יעיל לנו כי הגענו ליעד עם מסלול באורך 1001 ,

כלומר"$entry == 1001$. $marix[1][1]$"כאשר היינו יכולים למצוא מסלול זול יותר באורך 9 ולכן החמדני בכלל לא רלוונטי עבור פתרון הבעיה.



- **סיבוכיות**: $O(n + m)$- מספר המעברים במסלול החל מקודקוד $(0, 0)$עד קודקוד (n, m)כאשר מותר לצעוד למטה או ימינה בלבד.
- **נכונות השיטה**: השיטה לא מחזירה את התשובה הנכונה כי ייתכן ואחרי מעבר זול יגיע מעבר יקר ולהיפך ולכן לפעמים יהיה כדאי לוותר על מעבר זול כדי להרוויח בהמשך.

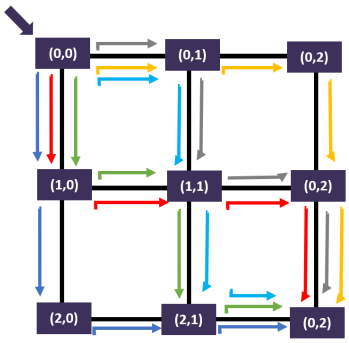
❖ אפשרות ב' - חיפוש שלם.

⇚ נייצר את כל המסלולים מהנקודה 1,1 ל m,n ונסכום את כל המשקלים של כל מסלול.

⇚ ניקח את המסלול בעל העלות הנמוכה ביותר.

⇚ נפתור ברקורסיה כאשר נתחיל מקודקוד (0,0) ונלך פעם אחת ימינה ופעם אחת למטה. נכנס לרקורסיה שוב ושוב עד שנגיע לקודקוד (N,M) וכך נעבור על כל המסלולים האפשריים.

כפי שאפשר לראות באיור מלמטה אמנם הגענו בהכרח לפתרון אבל עבור המטריצה הקטנה הזו ביצענו כמות גדולה של פעולות. עבור מטריצות גדולות יותר נגיע כבר למצב של פיצוץ קומבינטורי.



⇚ **סיבוכיות**:($O($ $\binom{n+m}{n} \cdot (n+m)$) כלומר מספר המסלולים כפול מעבר על כל מסלול וחישוב העלות של המסלול.

❖ אפשרות ג' - תכנות דינאמי.

⇚ בחיפוש השלם, עשינו בדיקות מיותרות, כי אם 2 מסלולים מגיעים לאותה נקודה ואחד מהם בעלות נמוכה יותר, ברור שניקח אותו ואין טעם להמשיך עם המסלול הארוך יותר.

⇚ נייצר מטריצה שבה בכל תא $[i][j]$ נשמור את אורך המסלול הקצר ביותר מהנקודה (0,0) עד הנקודה (i,j)ונמלא את המטריצה באופן הבא:

העלות להגיע לנקודה (i,j) שווה למינימום:

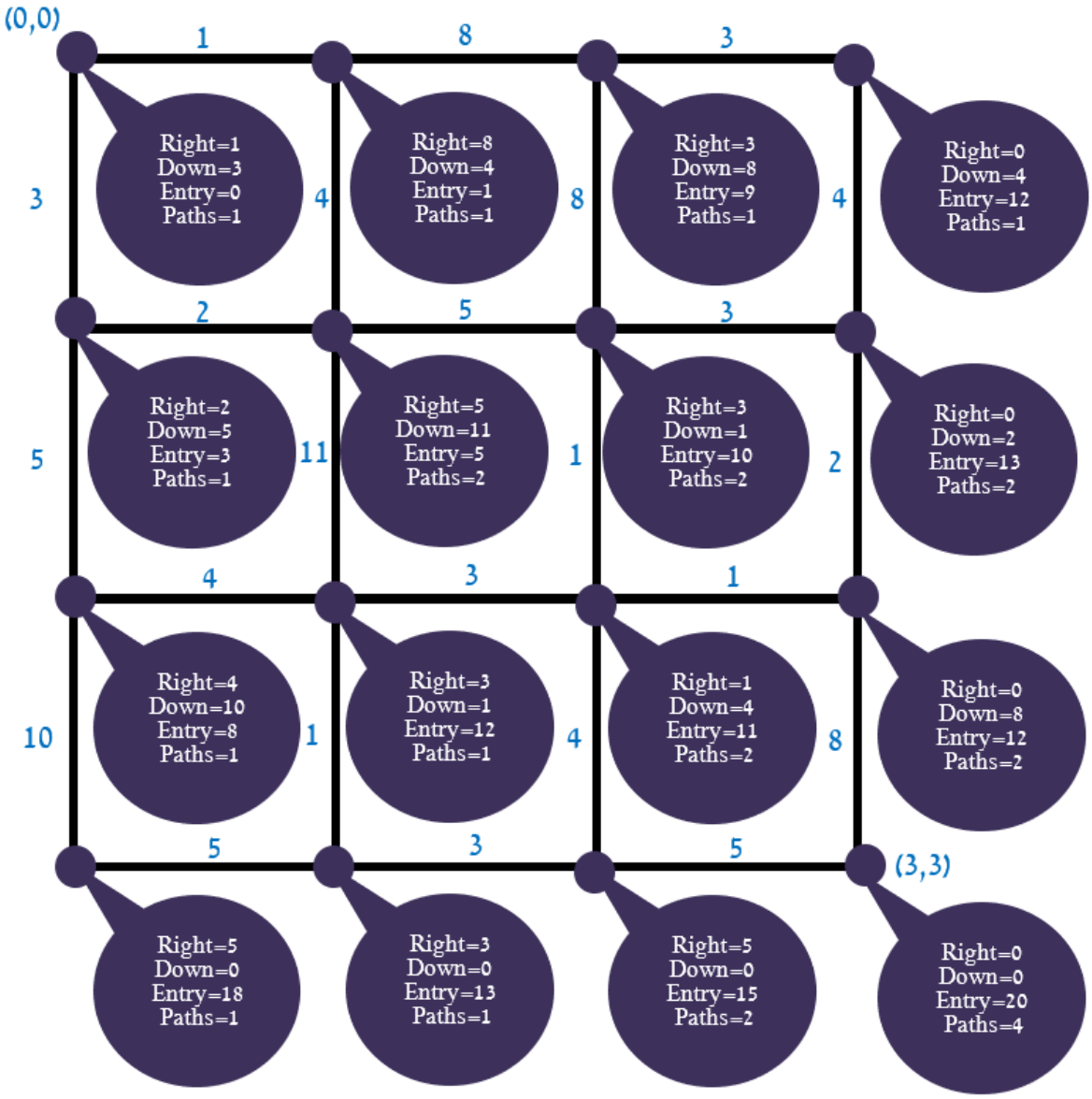
מבין העלות להגיע ל$(i,j-1)$ ואז ללכת ימינה, לבין העלות להגיע ל $(i-1,j)$ ואז ללכת למטה, כלומר :

```
int downPath = matrix[i - 1][j].entry + matrix[i - 1][j].goDown
int rightPath = matrix[i][j - 1].entry + matrix[i][j - 1].goRight;
matrix[i][j].entry = Math.min(rightPath, downPath);
```

נמלא תחילה עמודה ראשונה ושורה ראשונה (בדומה לבסיס אינדוקציה),

מן הסתם מספר המסלולים עבור כל קודקוד בפעולה זאת הוא מסלול 1 בלבד מכיוון שאני מוגבל להמשיך רק ימינה או למטה מכל קודקוד.

נתבונן באיור לדוגמא של matrix בגודל 3x3 כאשר המספרים על הצלעות הם המשקל ועבור כל ערך Entry נקבל את המחיר הזול ביותר מנקודה (0,0) עד לנקודה נוכחית שלה, בנוסף paths זה מספר המסלולים הזולים ביותר מנקודה (0,0) ועד לקודקוד זה:



יצירת המטריצה (תואם לציור מלמעלה) וחישוב ערכי Entry עבור כל צומת במטריצה.
לטובת נוחות המימוש, נדמה את הנתונים לשימושים שמוכרים לנו כבר, למשל עבור ההשמה הבאה:

```
mat[0][0] = new Node(3,1);
```

הערך 3 יהיה הערך לרדת למטה goDown במטריצה (כמו הערך i בלולאה כפולה על מערך דו מימדי).
הערך 1 יהיה הערך לימין goRight במטריצה (כמו הערך j בלולאה כפולה על מערך דו מימדי).
חשוב לציין כי עבור מקרים דומים ל- `mat[1][3] = new Node(2,0)` ימינה הוא 0 מכיוון שהגענו לקצה מימין ואין עוד לאן להמשיך משם.

```
public static void generateMatrix(Node[][] matrix) {

   for(int i = 1 ; i < matrix.length; i++) { // O(N)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   }
   for(int i = 1 ; i < matrix[0].length; i++) { // O(M)
       matrix[0][i].entry = matrix[0][i-1].entry + matrix[0][i-1].goRight;
   }
   for(int i = 1; i < matrix.length; i++) { // O(N*M)
       for(int j = 1; j < matrix[0].length; j++) {
           int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
           int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
           matrix[i][j].entry = Math.min(fromAbove,fromLeft);
       }
   }
}
public static void main(String[] args) {
   Node[][] mat = new Node[4][4];
   mat[0][0] = new Node(3,1);
   mat[0][1] = new Node(4,8);
   mat[0][2] = new Node(8,3);
   mat[0][3] = new Node(4,0);
   mat[1][0] = new Node(5,2);
   mat[1][1] = new Node(11,5);
   mat[1][2] = new Node(1,3);
   mat[1][3] = new Node(2,0);
   mat[2][0] = new Node(10,4);
   mat[2][1] = new Node(1,3);
   mat[2][2] = new Node(4,1);
   mat[2][3] = new Node(8,0);
   mat[3][0] = new Node(0,5);
   mat[3][1] = new Node(0,3);
   mat[3][2] = new Node(0,5);
   mat[3][3] = new Node(0,0);
   generateMatrix(mat);
}
```

⇜ **סיבוכיות השיטה**: $O(n \cdot m)$ ממלאים את המטריצה לפי החוקיות.

יתרון נוסף של הפונקציה **generateMatrix** הוא האפשרות לחשב את מספר המסלולים הזולים ביותר עבור כל צומת (במקביל לבניית המטריצה) בתוך הלולאה של הפונקציה הזאת.

```
public static void generateMatrix(Node[][] matrix) {

   for(int i = 1 ; i < matrix.length; i++) { // O(N)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   }
   for(int i = 1 ; i < matrix[0].length; i++) { // O(M)
       matrix[0][i].entry = matrix[0][i-1].entry + matrix[0][i-1].goRight;
   }

   for(int i = 1; i < matrix.length; i++) { // O(N*M)
       for(int j = 1; j < matrix[0].length; j++) {
           int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
           int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
           matrix[i][j].entry = Math.min(fromAbove,fromLeft);

// addition:
           int leftPaths = matrix[i][j-1].numOfPaths;
           int abovePaths = matrix[i-1][j].numOfPaths;
           if(fromAbove > fromLeft) {
               matrix[i][j].numOfPaths = leftPaths;
           }
           else if(fromAbove < fromLeft) {
               matrix[i][j].numOfPaths = abovePaths;
           }
           else { // if they are equals.
               matrix[i][j].numOfPaths = abovePaths + leftPaths;
           }
       }
   }
}
```

↞ **סיבוכיות השיטה**: $O(n \cdot m)$ ממלאים את המטריצה לפי החוקיות ובנוסף מציבים את מספר המסלולים הזולים ביותר לכל צומת.

מימוש **רקורסיבי** לבניית המטריצה

```
public static void generateRec(Node[][] matrix, int i, int j) {
   if(i == matrix.length)
       return;
   if(j == matrix.length) {
       generateRec(matrix,i+1,1);
   }
   else {
       int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
       int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
       matrix[i][j].entry = Math.min(fromAbove,fromLeft);

       int abovePaths = matrix[i-1][j].numOfPaths;
       int leftPaths = matrix[i][j-1].numOfPaths;
       if(fromAbove > fromLeft){
           matrix[i][j].numOfPaths = leftPaths;
       } else if(fromAbove < fromLeft) {
           matrix[i][j].numOfPaths = abovePaths;
       } else {
           matrix[i][j].numOfPaths = leftPaths + abovePaths;
       }
       generateRec(matrix,i,j+1);
   }
}

public static void generateMatrix(Node[][] matrix) {
   for(int i = 1 ; i < matrix.length; i++)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   for(int j = 1 ; j < matrix[0].length; j++)
       matrix[0][j].entry = matrix[0][j-1].entry + matrix[0][j-1].goRight;

   generateRec(matrix,1,1);
}
```

↞ **סיבוכיות השיטה**: $O(n \cdot m)$ ממלאים את המטריצה לפי החוקיות ובנוסף מציבים את מספר המסלולים הזולים ביותר לכל צומת.

נמשיך לעבוד עם שיטת התכנות הדינאמי, ונציג עבורה פונקציות שימושיות עבור מסלולים:

- **getCornerPath** - מחזיר את המסלול הזול ביותר בין נקודת ההתחלה (0,0) לבין הנקודה (n,m).
- **getPath** - מחזיר את המסלול הזול ביותר עבור צומת i,j שנשלח לפונקציה.
- **getAllPaths** - מחזיר את המסלול הזול ביותר לכל הצמתים.
- נוסיף שדה חדש לכל Node מסוג `myShortestPath` **HashSet<Vector<Node>>** השומר את כל המסלולים הזולים שלו ב-Vector המכיל את כל ה-Node במסלול.

מימוש **אינדוקטיבי**:

```
public static void getCornerPath(Node[][] matrix) {
   int i = matrix.length-1, j = matrix[0].length-1;
   getPathOfNode(matrix,i,j);
}
public static void getPathOfNode(Node[][] matrix, int i, int j) {
   if(i < matrix.length && j < matrix[0].length) {
       Vector<Node> vector = new Vector<>();
       Node source = matrix[i][j];
       int tempi = i, tempj = j;

       while(i != 0 || j != 0) {
           vector.add(matrix[i][j]);
           if(i == 0) { j--; }
           else if(j == 0) { i--;}
           else {
               int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
               int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;

               if(fromAbove > fromLeft) { j--;}
               else if(fromAbove < fromLeft) { i--; }
                   else {
                   i--;
                   j--;
               }
           }
       }
       vector.add(matrix[i][j]);
       source.myShortestPath.add(vector);
   }
}
public static void getAllPaths(Node[][] matrix) {
   for(int i = 1 ; i < matrix.length; i++) {
       for(int j = 1; j < matrix[0].length ; j++) {
           getPathOfNode(matrix,i,j);
       }
   }
}
```

במימוש **רקורסיבי** זה קיימת גם פונקציה **getAllPaths** שמחזירה לא רק את המסלול הזול ביותר (יחיד) לכל הקודקודים בגרף אלא גם מחזירה את **כל** המסלולים הזולים ביותר עבור **כל** צומת (במידה ויש יותר ממסלול אחד כזה):

```
public static void getCornerPath(Node[][] matrix) {
   int i = matrix.length-1, j = matrix[0].length-1;
   System.out.println("\n===== getCornerPath ("+i+","+j+") recursive =========");
   getPath(matrix, matrix[i][j],new Vector<>(),i,j);
   printPaths(matrix[i][j],i,j);
}

public static void getPathOfNode(Node[][] matrix, int i, int j) {
   System.out.println("\n===== getPathOfNode ("+i+","+j+") recursive =========");
   if(i < matrix.length && j < matrix[0].length) {
       getPath(matrix, matrix[i][j],new Vector<>(),i,j);
       printPaths(matrix[i][j],i,j);
   }
}

public static void getPath(Node[][] matrix, Node source,Vector<Node> path, int i,
int j) {
   path.add(matrix[i][j]);
   if(i == 0 && j == 0) {
       source.myShortestPath.add(path);
   } else if(i == 0) {
       getPath(matrix,source,path,i,j-1);
   } else if(j == 0) {
       getPath(matrix,source,path,i-1,j);
   } else {
       int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
       int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;

       if(fromAbove > fromLeft) {
           getPath(matrix,source,path,i,j-1);
       } else if(fromAbove < fromLeft) {
           getPath(matrix,source,path,i-1,j);
       } else {
           Vector<Node> pathLeft = new Vector<>(path);
           getPath(matrix,source,pathLeft,i,j-1);

           Vector<Node> pathAbove = new Vector<>(path);
           getPath(matrix,source,pathAbove,i-1,j);
       }
   }
}
```

```
public static void getAllPaths(Node[][] matrix) {
   System.out.println("\n===== getAllPaths recursive =========");
   allPathsRec(matrix,1,1);
}

public static void allPathsRec(Node[][] matrix, int i, int j) {
   if(i == matrix.length) {
       return;
   }
   if(j == matrix[0].length) {
       allPathsRec(matrix,i+1,1);
   } else {
       getPath(matrix,matrix[i][j],new Vector<>(),i,j);
       printPaths(matrix[i][j],i,j);
       allPathsRec(matrix,i,j+1);
   }
}

public static void printPaths(Node node,int i, int j) {
   HashSet<Vector<Node>> set = node.myShortestPath;
   System.out.println("Node: ("+i+","+j+") {");
   int counter = 1;
   for(Vector<Node> vec : set) {
       System.out.print(counter+"): " +vec);
       System.out.println();
       counter++;
   }
   System.out.println("}");
}
```

סיבוכיות השיטה:

p = מספר המסלולים הקצרים ביותר עבור הקודקוד (רק עבור המימוש של הקודים האלה כי הוא מחזיר יותר ממסלול זול ביותר אחד במידה ויש).

באופן כללי אם התבקש ממנו להחזיר רק מסלול זול אחד אז הסיבוכיות היא $O(n + m)$.

- **getCornerPath** - סיבוכיות $O(p \cdot (n + m))$
- **getPath** - סיבוכיות $O(p \cdot (n + m))$
- **getAllPaths** - סיבוכיות $O(n \cdot m \cdot (n + m) \cdot p)$

רצף אחדות במערך

תיאור הבעיה

בהינתן מערך המורכב מאפסים ואחדות בלבד יש למצוא את תת-הקטע הארוך ביותר המורכב מאחדות בלבד.

0	1	2	3	4	5	6	7	8	9
1	1	0	1	0	0	**1**	**1**	**1**	**1**

פתרון הבעיה

מימוש בגיטהאב

❖ אפשרות א' - אלגוריתם חמדני

⇐ נתחיל לטייל במערך ונספור את האחדות שנתקלנו בהם עד שנגיע ל-0 כלשהו ונעצור.

```
public static int greedy(int[] arr){
   int ans = 0;
   for(int i = 0 ; i < arr.length ; i++) {
       if(arr[i] == 0) {
           break;
       }
       ans++;
   }
   return ans;
}
public static int greedyRec(int[] arr, int i){
   if(i == arr.length)
       return 0;
   if(arr[i] == 0)
       return 0;
   return greedyRec(arr,i+1) + 1;
}
public static void main(String[] args) {
   int[] arr = {1,1,0,1,0,0,1,1,1,1};
   System.out.println(greedy(arr)); // prints 2
   System.out.println(greedyRec(arr,0)); // prints 2
}
```

⇜ **סיבוכיות זמן הריצה**: $O(n)$.

⇜ **נכונות השיטה**: קל לראות כי בדוגמת המימוש לא החזרנו את רצף האחדות הגדול ביותר.

❖ אפשרות ב' - אלגוריתם חמדני משופר

⇚ נתחיל לטייל במערך ונספור את האחדות שנתקלנו בהם.

⇚ נשמור בערך max את רצף האחדות הארוך ביותר עד כה.

⇚ נחזיר את max.

```
public static int improved(int[] arr) {
   int max = 0;
   int current_max = 0;

   for(int i = 0; i < arr.length; i++) {
       if(arr[i] == 1) {
           current_max++;
           if(current_max > max){
               max = current_max;
           }
       }
       else {
           current_max = 0;
       }
   }

   return max;
}
public static void main(String[] args) {
   int[] arr = {1,1,0,1,0,0,1,1,1,1};
   System.out.println(improved(arr)); // prints 4
}
```

⇜ **סיבוכיות זמן הריצה**: $O(n)$.

⇜ **נכונות השיטה**: אמנם האלגוריתם חמדני אבל הוא בהכרח נותן לנו את התשובה הנכונה.

❖ אפשרות ג' - תכנות דינאמי

⇚ נגדיר מערך עזר באורך המערך המקורי.

⇚ כל תא במערך העזר מייצג את מספר רצף האחדות שצברנו עד כה.

⇚ במהלך הריצה נשמור במשתנה max את הערך הגדול ביותר עד כאן ובסוף נחזיר אותו.

⇚ נשים לב שחמדן ותכנות דינמי שווים בסיבוכיות זמן הריצה.

⇚ היתרון הוא שברגע שיש מערך עזר שמכיל עוד מידע נוכל להחזיר למשל באיזה תאים נמצא הרצף, או האם יש 2 רצפים בעלי אורך זהה מקסימלי וכו'..

דוגמה

	0	1	2	3	4	5	6	7	8	9
arr	1	1	0	1	0	0	**1**	**1**	**1**	**1**

	0	1	2	3	4	5	6	7	8	9
count	1	2	0	1	0	0	**1**	**2**	**3**	**4**

מימוש שיטת תכנות דינמי

```
public static int dynamic(int[] arr){
   int[] count = new int[arr.length];

   if(arr[0] == 1)
       count[0] = 1;

   int max = 0;
   for(int i = 1; i < arr.length; i++) {
       if(arr[i] == 1) {
           count[i] = count[i-1] + 1;
           if(count[i] > max)
               max = count[i];
       }
   }
   return max;
}

public static void main(String[] args) {
   int[] arr = {1,1,0,1,0,0,1,1,1,1};
   System.out.println(dynamic(arr)); // prints 4
}
```

⇜ **סיבוכיות זמן הריצה**: $O(n)$.

מטריצת אחדות

תיאור הבעיה

בהינתן מערך דו-מימדי **A** (מטריצה) המורכב מאפסים ואחדות בלבד, יש למצוא את תת-המטריצה **הריבועית** הגדולה ביותר המורכבת מאחדות בלבד.

בדוגמה הבאה, נסתכל על המטריצה A, מטריצת האחדות הגדולה ביותר היא בגודל 3.

	0	1	2	3	4
0	1	1	0	0	0
1	1	1	1	1	1
2	0	1	1	1	0
3	1	1	1	1	1
4	0	1	0	0	0

❖ אפשרות א':

⇚ נעזר ברעיון לפתרון הדינאמי עבור בעיית רצף האחדות הגדול ביותר במערך.

תזכורת

	0	1	2	3	4	5	6	7	8	9
arr	1	1	0	1	0	0	**1**	**1**	**1**	**1**

	0	1	2	3	4	5	6	7	8	9
count	1	2	0	1	0	0	**1**	**2**	**3**	**4**

⇐ נבנה 3 מטריצות עזר:

1. מטריצה **B** המייצגת את רצף האחדות לפי שורות.
2. מטריצה **C** המייצגת את רצף האחדות לפי עמודות.
3. מטריצה **D** המייצגת את גודל ריבוע האחדות בכל תא.

מטריצה D מייצגת את גודל ריבוע האחדות בכל תא. לכן נצטרך לבדוק מה קורה בתא זה במטריצות האחרות שבנינו כדי לדעת האם אנחנו ממשיכים ריבוע קיים או לא.

יש בידינו 4 מטריצות.

B שורות

	0	1	2	3	4
0	1	2	0	0	0
1	1	2	3	4	5
2	0	1	2	3	0
3	1	2	3	4	5
4	0	1	0	0	0

A מקורית

	0	1	2	3	4
0	1	1	0	0	0
1	1	1	1	1	1
2	0	1	1	1	0
3	1	1	1	1	1
4	0	1	0	0	0

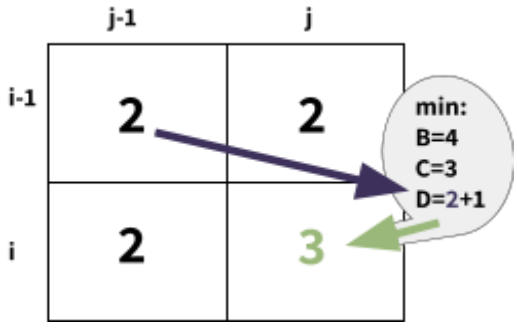
C עמודות

	0	1	2	3	4
0	1	1	0	0	0
1	2	2	1	1	1
2	0	3	2	2	0
3	1	4	3	3	1
4	0	5	0	0	0

איך נמלא את מטריצת הפתרון D?
עבור תא ספציפי: אם במטריצה המקורית יש בו 0 נרשום 0 אך אם יש בו 1 נצטרך לבדוק האם הוא משלים לנו ריבוע אחדות מלמעלה, משמאל ומהאלכסון.
נעתיק את השורה הראשונה של A ל-D וגם את העמודה הראשונה של A ל-D ונתחיל לחשב:

$$D[i][j] = min(B[i][j]\,,\ C[i][j],\ D[i-1][j-1]\ +1)$$

במטריצה D נבצע לדוגמה:



מימוש בגיטהאב

מימוש שיטה א':

```
public static int solution(int[][] A) {
   printMatrix(A);
   int[][] D = generateD(A);
   printMatrix(D);
   return getBiggest(D);
}


public static void printMatrix(int[][] mat) {

   for(int i = 0; i < mat.length; i++) {
       for(int j = 0; j < mat[0].length; j++) {
           System.out.print(mat[i][j] + " ");
       }
       System.out.println();
   }
   System.out.println();
}

//Rows
public static int[][] generateB(int[][] A) {
   int[][] B = new int[A.length][A[0].length];

   for(int i = 0 ; i < A.length; i++) {
       B[i][0] = A[i][0];
   }
   for(int i=0; i < A.length; i++) {
       for(int j=1; j < A[0].length; j++) {
           if(A[i][j] == 1) {
               B[i][j] = B[i][j-1] + 1;
           }
       }
   }

   return B;
}

//Columns
public static int[][] generateC(int[][] A) {
   int[][] C = new int[A.length][A[0].length];

   for(int i = 0 ; i < A[0].length; i++) {
       C[0][i] = A[0][i];
   }
```

```
    for(int i=1; i < A.length; i++) {
        for(int j=0; j < A[0].length; j++) {
            if(A[i][j] == 1) {
                C[i][j] = C[i-1][j] + 1;
            }
        }
    }
    return C;
}

// solution matrix
public static int[][] generateD(int[][] A) {
    int[][] B = generateB(A); // rows
    int[][] C = generateC(A); // columns
    int[][] D = new int[A.length][A[0].length];

    for(int i = 0 ; i < A.length; i++) {
        D[i][0] = A[i][0];
    }
    for(int i = 0 ; i < A[0].length; i++) {
        D[0][i] = A[0][i];
    }
    for(int i=1; i < A.length; i++) {
        for(int j=1; j < A[0].length; j++) {
            if(A[i][j] == 1) {
                D[i][j] = min(B[i][j],C[i][j],D[i-1][j-1] +1) ;
            }
        }
    }
    return D;
}
public static int min(int b, int c, int d) {
    int min = b;
    if(min > c) min = c;
    if(min > d) min = d;
    return min;
}

public static int getBiggest(int[][] D) {
    int max = 0;
    for(int i = 0 ; i < D.length; i++) {
        for(int j = 0 ; j < D[0].length; j++) {
            if(D[i][j] > max)
                max = D[i][j];
        }
    }
```

```
    return max;
}

public static void main(String[] args) {

    int[][] A = {{1,1,0,0,0},
                 {1,1,1,1,1},
                 {0,1,1,1,0},
                 {1,1,1,1,1},
                 {0,1,0,0,0}};
    System.out.println("the biggest square is: " +solution(A)); // prints 3
}
```

- **סיבוכיות זמן ריצה:** בניית 3 מטריצות בגודל nxn, $O(3 \cdot n^2)$ וגם החזרת הערך הגדול ביותר $O(n^2)$- סה"כ $O(4 \cdot n^2)$.
- **נכונות הפתרון:** מבטיח בהכרח פתרון, אבל נאלץ להשתמש ב-3 מטריצות, האם אפשר לייעל את זה? -

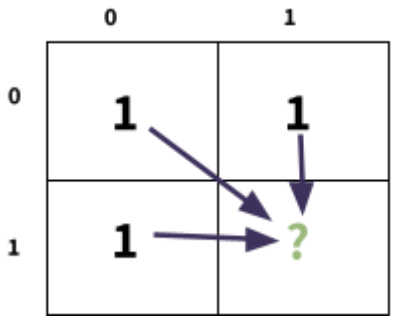
❖ אפשרות ב':

⇐ נרצה לקבל מיד את מטריצת התשובות ללא שימוש במטריצות עזר.

⇐ המטריצות היחידות שבידינו הם מטריצה A ומטריצת הפתרון D.

איך נעשה זאת? -

1. נעתיק את השורות העמודות הראשונות של A ל-D.
2. נתבונן בציור, אפשר לראות כי עבור '?' נציב במטריצה D את הערך 2, לכן אין לנו באמת צורך יותר במטריצות העזר שהוצגו באפשרות הראשונה.



3. אם במטריצה A יופיע בתא את 0 אז נציב 0 גם אצל D.
4. אם במטריצה A יופיע בתא את 1 אז נפעל לפי הצעד הבא:

$$D[i][j] = min(D[i-1][j]\ ,\ D[i][j-1],\ D[i-1][j-1]\ ) + 1$$

נשים לב כי הפעם חיברנו 1+ עבור התא.

מימוש אפשרות ב'

```
public static int solution(int[][] A) {
   int max = 0;
   int[][] D = new int[A.length][A[0].length];

   for(int i = 0 ; i < A.length; i++) {
       D[i][0] = A[i][0];
       D[0][i] = A[0][i];
   }

   for(int i = 1 ; i < A.length; i++) {
       for(int j = 1 ; j < A[0].length; j++) {
           if(A[i][j] == 1) {
               D[i][j] = min(D[i-1][j], D[i][j-1], D[i-1][j-1]) + 1;
               if(D[i][j] > max)
                   max = D[i][j];
           }
       }
   }

   return max;
}

public static int min(int a, int b, int c) {
   int min = a;
   if(min > b) min = b;
   if(min > c) min = c;
   return min;
}
public static void main(String[] args) {

   int[][] A = {{1,1,0,0,0},
           {1,1,1,1,1},
           {0,1,1,1,0},
           {1,1,1,1,1},
           {0,1,0,0,0}};
   System.out.println("the biggest square is: " +solution(A));
}
```

- **סיבוכיות זמן ריצה:** בניית מטריצה D ובמקביל קבלת max ולכן $O(n^2)$.
- **נכונות הפתרון:** מבטיח בהכרח פתרון עבור כל מטריצה (nxn).

בעיית הפיצה

1) נתבונן באיור הבא: אלי ובני הזמינו פיצה. אלי מחלק את הפיצה ל6 חלקים שווים. אלי אוכל 2 משולשים בזמן שבני אוכל משולש 1.

המטרה: שאלי יאכל כמה שיותר משולשים.

2) כל אחד לוקח משולש, אלי כבר סיים את המשולש שלו ובני רק בחצי של המשולש הראשון.

3) אלי לוקח משולש נוסף בזמן שבני עדיין עם הראשון, ושניהם מסיימים כל אחד משולש ביחד.

4) כעת קל לראות למבט הלאה (וממבט ראשון) כי אלי כמובן יאכל פי 2 משולשים ממה שבני אכל.

לכן עבור מקרה כללי:

תיאור הבעיה

- ⇚ מהירות האכילה של אלי גדולה **פי-X** ממהירות האכילה של בני.
- ⇚ ניתן לחלק את הפיצה ל-N משולשים **שווים**.
- ⇚ במהלך הארוחה כל אחד לוקח משולש נוסף לאחר שסיים את הקודם.
- ⇚ אסור ששניהם יגיעו אל המשולש האחרון בו זמנית!.
- ⇚ יש למצוא את החלוקה האופטימלית כך שאלי יאכל כמה שיותר משולשי פיצה.

נשים לב שיש מקרים בהם אלי לא ינצח כמו למשל אם הוא יחלק את הפיצה ל- ½ .

וגם אם נחלק למשל את הפיצה ל-¼ , נגיע למצב שרבים על המשולש האחרון וזה אסור לפי תיאור הבעיה.
אם נמשיך לחלק את הפיצה לכל מיני חלקים אנחנו נגלה כי מסתתרת פה חוקיות מסוימת.
בעצם, ביצענו כאן מעין מחקר קטן שבעזרתו גילינו מה החלוקה הטובה ביותר עבור אלי.

בהסתמך על הדוגמאות שראינו, אז אם מספר המשולשים הוא $N = 3$ואלי אוכל פי $X = 2$,
קל וברור שמספר המשולשים N צריך להיות גדול או שווה ל-X+1 .

נוכיח כי $N \geq X + 1$.

הוכחה

☚ נניח שהחלוקה האופטימלית היא X=N (כלומר, כמות משולשי הפיצה שווה למהירות האכילה של אלי).
במקרה זה, אלי יאכל $\frac{N-1}{N}$משולשים, ובני יאכל $\frac{1}{N}$משולשים, ואז הם לא יסיימו בו זמנית.
(למשל X=2 אז הפיצה מחולקת ל-½ , אלי אכל את המשולש שלו בזמן שבני אכל רק חצי מהמשולש שלו).

☚ אם נחלק ל - $N = X + 1$ חלקים, אז אלי יאכל $\frac{N}{N+1}$ובני יאכל $\frac{1}{N+1}$ואז הם סיימו בו זמנית.

צריך להוכיח ש: $\frac{N-1}{N} < \frac{N}{N+1}$,

$$\frac{N-1}{N} < \frac{N}{N+1} \setminus \cdot N \cdot (N+1)$$
$$(N-1) \cdot (N+1) < N^2$$
$$N^2 - 1 < N^2$$

אי השיוויון מתקיים ולכן עדיף לחלק ל- X+1 חלקים.

∎

אך אנחנו רוצים יותר מזה, ונרצה להימנע ממקרה שבו שניהם מגיעים למשולש האחרון בו זמנית.
כאשר X+1 הוא מספר המשולשים שהם מסיימים לאכול בו זמנית, ו-P היא הכפולה (כמות הסיבובים),
אז מספר החלקים N צריך להיות $N \neq (X+1) \cdot P + 1$כדי להימנע מהמצב שבו שניהם מגיעים למשולש האחרון בו זמנית. (הם אוכלים יחד $(X+1) \cdot P$ואז ישאר המשולש האחרון שזה ה- (+1) שנשאר הם ילחמו על המשולש הזה וזה אסור לפי תיאור הבעיה).

הוכחה

נניח כי $N = (X+1) \cdot P + r$כאשר r הוא השארית ו- $2 \leq r \leq X$.
במקרה זה, אלי יאכל $\frac{X \cdot P + r - 1}{(X+1) \cdot P + r}$מהפיצה ובני יאכל $\frac{P+1}{(X+1) \cdot P + r}$מהפיצה. (אלי אוכל את השארית r כי הוא יגיע אליה לפני שבני יספיק לסיים).

נוכיח שאלי צריך לחלק את הפיצה ל-X+1 משולשים, כלומר צ"ל כי - $\frac{X \cdot P + r - 1}{(X+1) \cdot P + r} < \frac{X}{X+1}$.

$$\frac{X \cdot P + r - 1}{(X+1) \cdot P + r} < \frac{X}{X+1} \setminus \cdot [(X+1) \cdot P + r] \cdot [X+1]$$
$$(X \cdot P + r - 1) \cdot (X+1) < X \cdot ((X+1) \cdot P + r)$$
$$X^2 \cdot P + XP + Xr + r - X - 1 < X^2 \cdot P + XP + Xr$$
$$r < X + 1$$

השארית קטנה מהחלוקה ולכן מתקיים והם לא יגיעו בו זמנית למשולש האחרון.
הפונקציה היא $f(X) = X + 1$ וזוהי החלוקה האופטימלית.

∎

מימוש בגיטהאב

מימוש הבעיה, מחזיר true אם הפרמטרים מאפשרים מצב אופטימלי בו אלי יאכל יותר משולשים מבני ולא יקרה מצב שנשאר משולש אחד ואחרון לשניהם.

```java
public static boolean pizza(double X , int N) {
   int F = (int)X + 1 ;
   int P =  N / (F + 1);
   int r = N % (F+1);

   if(2 <= r && r <= (int)X) {
       double t = (X*P + r - 1) / ((X+1)*P + r);
       if(t < X/(X+1) ) {
           return true;
       }
   }
   return false;
}

public static void main(String[] args) {
   System.out.println(pizza(2,6)); // true
   System.out.println(pizza(2,4)); // false
}
```

בעיית המזכירה

תיאור הבעיה

- ⇚ משרד מסוים נותן שירות ל-n לקוחות, מטרת מזכירת המשרד היא להקטין ככל האפשר את זמן הממוצע שהלקוחות נמצאים במשרד.
- ⇚ הזמן שהלקוח נמצא במשרד מורכב מזמן ההמתנה שלו עד תורו יחד עם זמן הטיפול שלו.
- ⇚ ספויילר: המצב הטוב ביותר הוא כאשר זמני הטיפול של הלקוחות נמצאים בסדר עולה.

ניתוח

נסמן i כלקוח , לכן $1 \leq i \leq n$ כך ש- ti - הוא זמן הטיפול של אחד מ-n הלקוחות.

Ti - הזמן שהלקוח נמצא במשרד. (זמן ההמתנה + זמן הטיפול).

לכן,

$$T_1 = t_1$$

(כי הלקוח הראשון לא צריך להמתין בתור, אז נציין רק את זמן הטיפול שלו).

$$T_2 = t_1 + t_2$$

(כי עבור הלקוח השני, זמן ההמתנה הוא הזמן של הטיפול ללקוח הראשון + זמן הטיפול שלו)

....

$$T_n = t_1 + t_2 + ... + t_n$$

על המזכירה למצוא הממוצע של זמן ההמתנה המינימלי ביותר: $Average = \frac{T_1 + T_2 + ...+ T_n}{n}$.

מספיק לחשב $min(T_1 + T_2 + ... + T_n)$, כי ה-n במכנה הוא גם ככה מספר קבוע.

דוגמה עבור: $t_1 = 10$, $t_2 = 1$, $t_3 = 8$.

תור הלקוחות	זמן המתנה ממוצע
(t_1, t_2, t_3)	$(10) + (1 + 10) + (1 + 10 + 8) = 40$
(t_1, t_3, t_2)	$(10) + (10 + 8) + (10 + 8 + 1) = 47$
(t_2, t_1, t_3)	$(1) + (1 + 10) + (1 + 10 + 8) = 31$
(t_2, t_3, t_1)	$(1) + (1 + 8) + (1 + 8 + 10) = 29$
(t_3, t_1, t_2)	$(8) + (8 + 10) + (8 + 10 + 1) = 45$
(t_3, t_2, t_1)	$(8) + (8 + 1) + (8 + 1 + 10) = 37$

פתרון

⇐ עבור חיפוש שלם צריך לעבור על כל האפשרויות לכן $n! > 2^n$ וזה ממש לא יעיל.

⇐ נשים לב כי בדוגמה שהצגנו וסימנו בכחול, התשובה הטובה ביותר מתקבלת כאשר מערך של זמני הטיפול ממוין מקטן לגדול.

זמן ההמתנה הכולל הוא:

$$sum = T_1 + \ldots + T_i + T_{i+1} + \ldots + T_n = t_1 + \ldots + (t_1 + \ldots + t_i) + (t_1 + \ldots + t_i + t_{i+1}) + \ldots + (t_1 + \ldots + t_i + t_{i+1} + \ldots + t_n)$$

נוכיח את הטענה בדרך השלילה:

ננית שזמן ההמתנה ti (איבר כלשהו במערך לא ממויין) גדול יותר מזמן ההמתנה של הבא בתור, $t_i > t_{i+1}$,

אם נחליף ביניהם $t_i \leftrightarrow t_{i+1}$ נקבל :

$$sum' = T_1 + \ldots + T_i' + T_{i+1} + \ldots + T_n = t_1 + \ldots + (t_1 + \ldots + t_{i+1}) + (t_1 + \ldots + t_{i+1} + t_i) + \ldots + (t_1 + \ldots + t_i + t_{i+1} + \ldots + t_n)$$

ונקבל:

$$sum - sum' = t_i - t_{i+1} > 0$$

ומכאן נקבל שהזמן הממוצע קטן יותר כאשר האיברים (זמני הטיפול) הסמוכים נמצאים בסדר עולה.

המסקנה היא שעל המזכירה למיין את מערך זמני הטיפול מהקצר לארוך והתאמה לקבוע את התור.

מימוש בגיטהאב

מימוש:

```
public static double getAverageTime(int[] times) {
   Arrays.sort(times);
   double avg = 0;

   for(int i = 0 ; i < times.length ; i++){
       double temp_avg = 0;

       for(int j = 0 ; j <= i ; j++) {
           temp_avg += times[j];
       }
       avg += temp_avg;
   }
   return avg/times.length;
}
public static void main(String[] args) {
   System.out.println(getAverageTime(new int[]{10,1,8}));
}
```

בעיית החציון

תיאור הבעיה

- ⇚ בהינתן מערך לא ממוין של מספרים אקראיים, יש למצוא את איבר שהוא גדול מהחציון של המערך.
- ⇚ כלומר, מספר שחצי מאיברי המערך גדולים ממנו וחצי מאיברי המערך קטנים ממנו.

הגדרה

חציון (**median**) הוא מספר (לא בהכרח איבר המערך) שמחצית מאיברי המערך גדולים ממנו ומחצית מאיברי המערך קטנים ממנו.
כאשר מספר איברי המערך אי-זוגי – החציון הוא איבר שנמצא באמצע המערך (**ממוין** !)
כאשר מספר איברי המערך זוגי – החציון שווה לממוצע של שני איברים הנמצאים באמצע המערך (**ממוין** !).

$$\mathrm{Med}(X) = \begin{cases} \frac{X[\frac{n}{2}]+X[\frac{n+1}{2}]}{2} & \text{if n is even} \\ X[\frac{n+1}{2}] & \text{if n is odd} \end{cases}$$

דוגמה

עבור אורך מערך אי-זוגי: arr[]={1,3,6,**8**,12,23,77}, median=8
עבור אורך מערך זוגי: arr[]={1,3,**6**,**12**,23,77}, median=(6+12)/2=9

פתרון הבעיה

מימוש בגיטהאב

- ❖ אפשרות א' - פתרון נאיבי
  - ⇚ נמיין את המערך בסיבוכיות $O(n \cdot (log\ n))$.
  - ⇚ נשלוף את האיבר האחרון במערך במיקום $arr[length - 1]$ כי הוא בהכרח גדול מהחציון של כל איברי המערך - לקיחת האיבר - $O(1)$.

```
public static int naive(int[] arr) {
   Arrays.sort(arr); // O(n*log n)
   return arr[arr.length-1]; // O(1)
}
```

⇜ **סיבוכיות זמן הריצה**: $O(n \cdot (log\ n) + 1)$.

❖ אפשרות ב' - אופטימלי

⇚ אם ניקח את האיבר הראשון במערך, ההסתברות שהוא גדול מהחציון הוא 50%.

⇚ אם ניקח את שני האיברים הראשונים במערך :

$a[0]$	$a[1]$	**(0 = מחצית שמאלית, 1= מחצית ימנית)**
0	0	שניהם נמצאים במחצית השמאלית של המערך
0	1	$a[0]$ בשמאלית, $a[1]$בימנית ולכן $a[0] < a[1]$
1	0	$a[0]$ בימנית , $a[1]$בשמאלית ולכן $a[0] > a[1]$
1	1	שניהם נמצאים במחצית הימנית של המערך

לכן ההסתברות שאחד האיברים $a[0]$או $a[1]$יהיו במחצית הימנית (כלומר גדולים מהחציון) היא $\frac{3}{4}$(75%).

לכן נבחר את המקסימום כדי להבטיח זאת $max(\ a[0],\ a[1]\ )$.

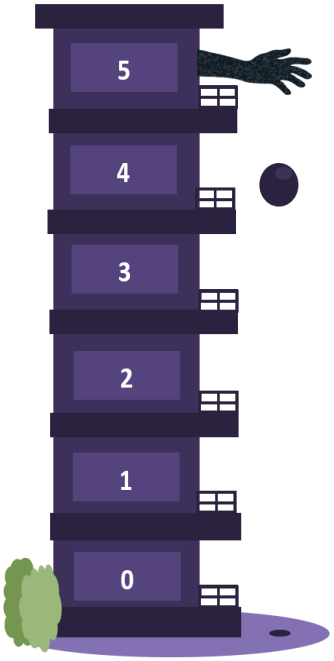
⇚ אם ניקח את שלושת האיברים הראשונים של המערך, ההסתברות שהמקסימום מבניהם יהיה גדול מהחציון היא הסתברות $\frac{7}{8}$.

⇚ אם ניקח את 64 האיברים הראשונים של המערך, ההסתברות שהגדול מביניהם נמצא במחצית הימנית של המערך היא (מספר כל תתי הקבוצות של קבוצה בת 64 איברים כלומר 2^{64}).
כלומר $1 - \frac{1}{64} = \frac{63}{64}$, לכן מספיק לחשב מקסימום של 64 איברים ראשונים של המערך ואנו נקבל איבר שגדול מהחציון. הסיבוכיות היא $O(1)$ מכיוון שאין חשיבות למספר 64, אלא המספר צריך להיות גדול מספיק על מנת שישאף ל-0.

```
public static int optimal(int[] arr) {
   int max = arr[0];

   for(int i = 1; i < arr.length-1 && i < 64 -1; i+=2) {

       if(arr[i] > arr[i+1] && max < arr[i]) {
           max = arr[i];
       }
       else if(max < arr[i+1]) {
           max = arr[i+1];
       }
   }

   return max;
}
```

בעיית כדור הזכוכית

תיאור הבעיה



- ⇚ לפניכם בניין בן **n** קומות ולרשותכם **b** כדורי זכוכית.
- ⇚ עליכם לגלות, באמצעות זריקת הכדורים מהקומות, מהי הקומה הנמוכה ביותר שאם תזרקו ממנה כדור היא תישבר.
- ⇚ מובן שאם הכדור שזרקתם נשברה, לא תוכלו להשתמש בה שוב להמשך הבדיקה.

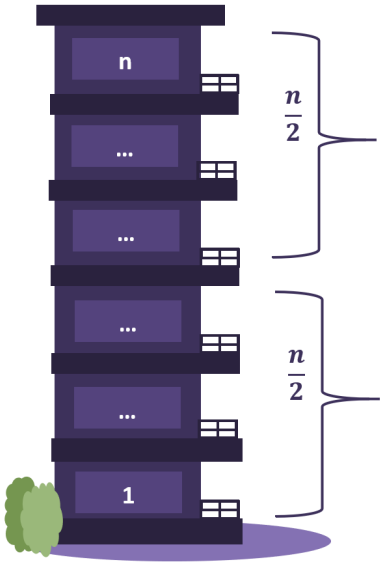
הערות -

1. אם הכדור נשבר מקומה כלשהי, הוא גם יישבר מכל הקומות הגבוהות יותר.
2. אם הכדור לא נשבר לאחר זריקה, נשתמש בו שוב.
3. אם הכדור נשבר מקומה i אז היא תשבר גם בכל קומה i+1.
4. אם הכדור לא נשבר מקומה i אז היא לא תישבר גם בכל קומה i-1.
5. אם אני למשל עם 100 כדורים וקומה אחת בלבד, אז מספיק לי כדור אחד בשביל הבעיה.

פתרון הבעיה

❖ **בידינו b=1 כדורים בלבד, ו-n קומות.**

- ⇚ מתחילים את זריקות הכדור מקומה אחת – אם הוא יישבר, מצאנו את הקומה, במקרה שהכדור לא ישבר זורקים אותו מקומה 2 .וכך עולים קומה-קומה וזורקים את הכדור עד שהוא ישבר.
  - ↜ **סיבוכיות זמן ריצה**: $O(n)$- במקרה הגרוע.



❖ **בידינו b=2 כדורים בלבד, ו-n קומות.**

- ⇚ נחלק את הבניין ל**שני חלקים שווים**, זורקים כדור ראשון מקומה $\frac{n}{2}$:

אם הכדור הראשון נשבר, נתחיל מהקומה הראשונה ונעבור בכל קומה עד שישבר.

אם הכדור הראשון לא נשבר, נבדוק מקומה$\frac{3n}{4}$ וכן הלאה..

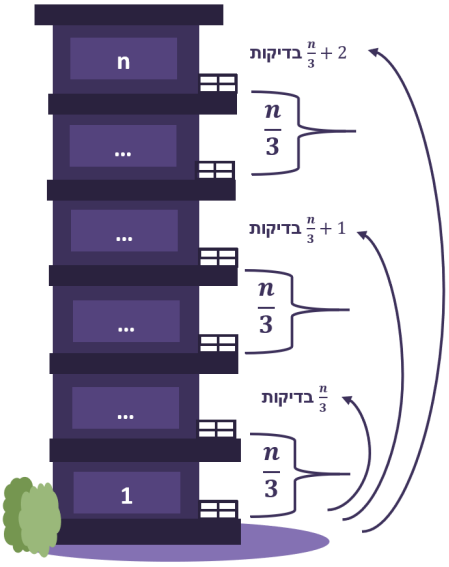
- ↜ **במקרה הטוב** - $log_2 n$ניסיונות. שזה בעצם מקרה שיש לנו $log_2 n$ כדורים או יותר.

⇜ **במקרה הגרוע** - יש לנו $\frac{n}{2} + 1$ ניסיונות (כי אם הכדור הראשון נשבר, נשאר לי עוד $\frac{n}{2}$ ניסיונות).

⇚ נחלק את הבניין ל**שלושה חלקים שווים,** זורקים כדור ראשון מקומה $\frac{n}{3}$:

אם הכדור הראשון נשבר, נתחיל מהקומה הראשונה ונעבור בכל קומה עד שישבר.

אם הכדור הראשון לא נשבר, נבדוק מקומה $\frac{2n}{3}$ וכן הלאה...



סך הכל: $\frac{n}{3} + 2$ ניסיונות (כפי שאפשר לראות בציור ניסינו בקומות הראשונות $\frac{n}{3}$ ולא נשבר, קפצנו 1+ , ניסינו $\frac{n}{3} + 1$ לא נשבר, קפצנו עוד 1+ , ולכן במקרה הגרוע ניסינו $\frac{n}{3} + 2$).

ראינו שאם נחלק ל**שני חלקים שווים**, במקרה הגרוע - $\frac{n}{2} + 1$ ניסיונות.

ראינו שאם נחלק ל**שלושה חלקים שווים,** במקרה הגרוע - $\frac{n}{3} + 2$ ניסיונות.

לכן, אם נחלק את הבניין ל-**k חלקים שווים**, במקרה הגרוע יש $\frac{n}{k} + (k - 1)$ ניסיונות.

ואם נחלק את הבניין ל-**n חלקים שווים**, במקרה הגרוע יש $\frac{n}{n} + (n - 1)$ ניסיונות.

נרצה לדעת מהי החלוקה האופטימלית ביותר

תזכורת - כדי לחשב מינימום של קטע, נגזור את הקטע, נשווה לאפס ואז נמצא את הערך המינימלי שלו.

לכן, מהו הערך של k שנותן מינימום לפונקציה $f(n,k) = \frac{n}{k} + (k - 1)$, כאשר n הוא קבוע.

נחשב $min\, f(x) = (\frac{n}{x} + x)$, לכן נגזור:

$$f'(x) = -\frac{n}{x^2} + 1 = 0$$

$$-\frac{n}{x^2} + 1 = 0 \setminus + \frac{n}{x^2}$$

$$1 = \frac{n}{x^2} \setminus \cdot x^2$$

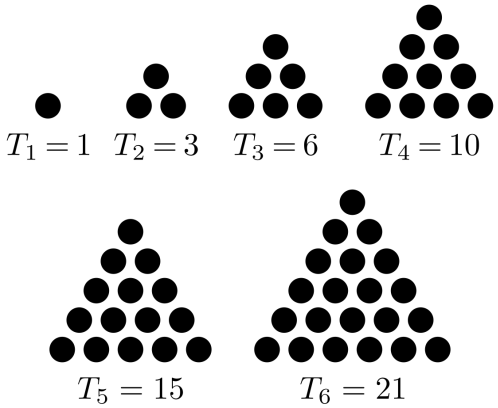
$$x^2 = n \setminus \cdot \sqrt{\ }$$

$$x = \sqrt{n}$$

לכן, קיבלנו שהחלוקה האופטימלית היא $\sqrt{n}$חלקים שווים.
ובמקרה הגרוע כאשר $f(\sqrt{n}) = \frac{n}{\sqrt{n}} + (\sqrt{n} - 1) \approx 2\sqrt{n}$.

אם n הוא מספר ריבועי, אז $k = \sqrt{n}$.
אם n הוא לא מספר ריבועי, אז ניקח מספר ריבועי מינימלי m הגדול מ-n כך שמתקיים:
$(m-1)^2 \leq n \leq m^2$.

- **בידינו עדיין b=2 כדורים בלבד, ו-n קומות**, אבל נציג חלוקה אחרת - מספרים משולשים.
המספר המשולשי ה-n-י מסומן T_n, והוא שווה לסכום כל המספרים הטבעיים מ-1 עד n.
ישנם אינסוף מספרים משולשיים וניתן להציג כל מספר משולשי בצורת משולש שווה-צלעות.



נניח ש-n הוא מספר משולשי: $n = 1 + 2 + \ldots + k = \frac{k \cdot (k+1)}{2}$ **(נוסחת גאוס)**,
נחלק את הבניין ל-k חלקים: (1,2,3,4…ועד k):
כאשר זורקים את הכדור הראשון מקומה k **והוא נשבר**, נשארים לנו עוד k-1 ניסיונות.
כלומר, סך הכל - k ניסיונות.
כאשר זורקים את הכדור הראשון מקומה k **והוא לא נשבר**, זורקים אותו מקומה $k + (k-1)$,
אם הוא נשבר, נשאר עם k-2 ניסיונות.
כלומר, סך הכל שוב - $k - 2 + 2 = k$ניסיונות.

- תמיד יהיה לנו k ניסיונות.

כאשר n מספר משולשי ו- $n = \frac{k \cdot (k+1)}{2}$אז:

$$2n = k(k+1)$$

$$2n = k^2 + k$$
$$2n - k = k^2$$

ומכיוון ש- $2n - k < 2n$ נקבל: $k^2 < 2n$, נוציא שורש ונקבל ש- $k < \sqrt{2n}$.

מסקנה - $\sqrt{2n} \leq 2\sqrt{n}$, ולכן חלוקה לפי מספר משולשים טובה יותר מחלוקה למספרים שווים - $\sqrt{n}$. כי הראנו שמקרה הגרוע עבור חלוקה למספרים שווים נקבל $2\sqrt{n}$, ואם מקרה זה יכול להיות גדול יותר מהחלוקה המינימלית שמצאנו עבור מספרים משולשים - $\sqrt{2n}$ אז ברור שעדיף לנו להשתמש בשיטת המספרים המשולשיים.

❖ **בידינו b>2 כדורים , ו-n קומות**.

⇚ בהינתן b כדורים ובניין בעל n קומות, נגדיר את הפונקציה $f(n, b)$ - מספר מינימלי של ניסיונות במקרה הגרוע:

$$f(n, 2) = \min_{1 \leq i \leq n} max(\, f(n - i, 2)\,,\, f(i - 1, 1)\,) + 1$$

⇚ כאן משתמשים בתכנות דינמי מורכב יותר: בכל שלב משתמשים בכל התוצאות של השלבים הקודמים.

כאשר זורקים כדור מקומה i יש שתי אפשרויות:

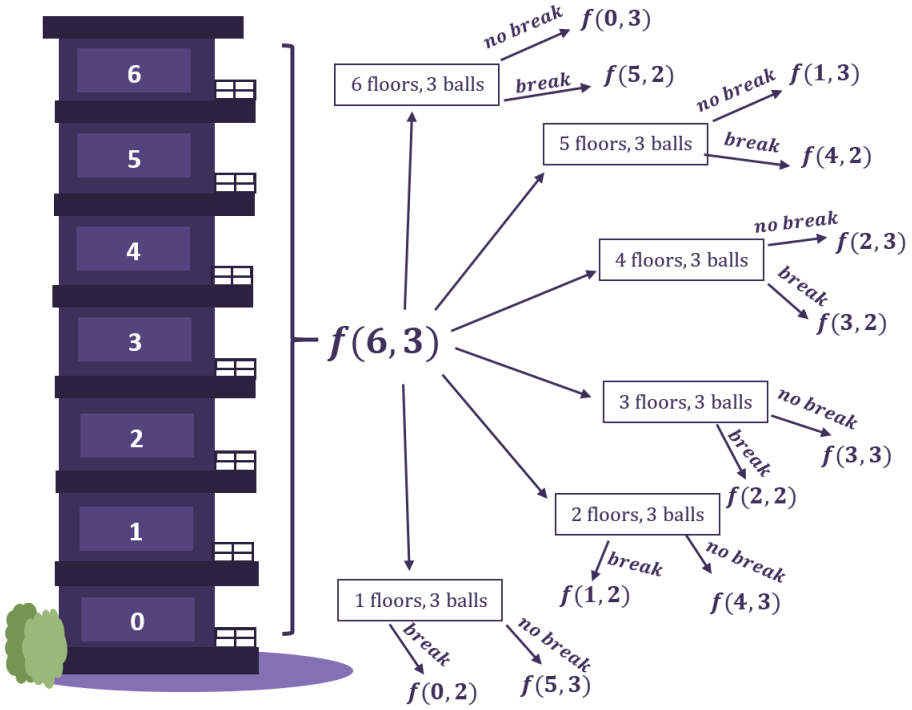
1. הכדור לא נשבר, נשארו 2 כדורים ו- $n - i$ קומות.
2. הכדור נשבר, נשאר כדור אחד ו- $i - 1$ קומות. הולכים על המקרה הגרוע ומחשבים את המספר המינימלי עבור כל הקומות.

נציין כי $f(2, 2) = 2$ וגם $f(1, 2) = 1$.

במקרה הכללי:

$$f(n, b) = \min_{1 \leq i \leq n} max(\, f(n - i, b)\,,\, f(i - 1, b - 1)\,) + 1$$

הפונקציה $f(n, b)$ תהיה שווה ל- $log_2 n + 1$ ($log_2 n$ בערך תחתון) כאשר $b \geq log_2 n$ (בערך עליון).



נציג מטריצת פתרון תכנות דינאמי עבור מקרה זה:

כמות ניסיונות		כדורים			
		0	**1**	**2**	**3**
קומות	**0**	0	0	0	0
	1	0	1	1	1
	2	0	2	2	2
	3	0	3	2	2
	4	0	4	3	3
	5	0	5	3	3
	6	0	6	3	3

הסבר למילוי המטריצה

עבור כל תא במטריצה נחשב את הנוסחה הבאה:

$$f(n,b) = \min_{1 \le i \le n} [\, 1 + max(\, f(n-i,b)\,,\, f(i-1,b-1)\,)\,]$$

נזכר שכאשר זורקים כדור מקומה i יש שתי אפשרויות:

1. הכדור לא נשבר, נשארו b כדורים ו- $n - i$ קומות.
2. הכדור נשבר, נשאר b-1 כדורים ו- $i - 1$ קומות.

הולכים על המקרה הגרוע ומחשבים את המספר המינימלי עבור כל הקומות.

למשל, עבור תא $(3, 2)$ במטריצה שלנו, כלומר **3 קומות ו-2 כדורים**, נחשב בצורה הבאה:

אם אני זורק את הכדור הראשון מהקומה ה-i:	אם הכדור לא נשבר $f(n - i, b)$	אם הכדור נשבר $f(i - 1, b - 1)$	1+ max	$\min_{1 \le i \le n}$
i = 1	$f(3 - 1, 2) = f(2, 2) =$	$f(1 - 1, 2 - 1) = f(0, 1$	$1 + max(2, 0) =$	min[3,2,3] = **2**
i = 2	$f(3 - 2, 2) = f(1, 2)$	$f(2 - 1, 2 - 1) = f(1, 1$	$1 + max(1, 1) =$	
i = 3	$f(3 - 3, 2) = f(0, 2)$	$f(3 - 1, 2 - 1) = f(2, 1$	$1 + max(0, 2) =$	

לכן בתא (3,2) נציב $f(3, 2) = 2$.

למשל, עבור תא $(4, 3)$ במטריצה שלנו, כלומר **4 קומות ו-3 כדורים**, נחשב בצורה הבאה:

אם אני זורק את הכדור הראשון מהקומה ה-i:	אם הכדור לא נשבר $f(n - i, b)$	אם הכדור נשבר $f(i - 1, b - 1)$	1+ max	$\min_{1 \le i \le n}$
i = 1	$f(4 - 1, 3) = f(3, 3) =$	$f(1 - 1, 3 - 1) = f(0, 2$	$1 + max(2, 0) =$	min[3,3,3,3] = **3**
i = 2	$f(4 - 2, 3) = f(2, 3)$	$f(2 - 1, 3 - 1) = f(1, 2$	$1 + max(2, 1) =$	
i = 3	$f(4 - 3, 3) = f(1, 3)$	$f(3 - 1, 3 - 1) = f(2, 2$	$1 + max(1, 2) =$	
i = 4	$f(4 - 4, 3) = f(0, 3) =$	$f(4 - 1, 3 - 1) = f(3, 2)$	$1 + max(0, 2) =$	

לכן בתא (4,3) נציב $f(4, 3) = 3$.

נמשיך להציב עד סוף המטריצה ונקבל כי עבור $f(6, 3) = 3$.

מימוש **אינדוקטיבי**

מימוש בגיטהאב

```
public static int minimalAttempts(int floors, int balls) {
   int[][] attempts = new int[floors+1][balls+1];

   for(int i = 0; i < attempts.length; i++)
       attempts[i][1] = i;
   for(int j = 1 ; j < attempts[0].length; j++)
       attempts[1][j] = 1;


   for(int b = 2; b < attempts[0].length; b++) { // balls
       for(int n = 2; n < attempts.length ; n++) { // floors
           int min = Integer.MAX_VALUE;
           for(int i = 1; i <= n; i++) {
               int max = Math.max(attempts[i-1][b-1], attempts[n-i][b]) + 1;
               if(min > max) {
                   min = max;
               }
           }
           attempts[n][b]= min;
       }
   }
   return attempts[floors][balls];
}
public static void main(String[] args) {
   System.out.println("minimal: " + minimalAttempts(105,2));
}
```

מימוש **רקורסיבי**

```
public static int minimalAttempts(int floors, int balls) {
   int[][] attempts = new int[floors+1][balls+1];
   for(int i = 0; i < attempts.length; i++)
       attempts[i][1] = i;
   for(int j = 1 ; j < attempts[0].length; j++)
       attempts[1][j] = 1;

   generateRec(attempts,2,2,1, Integer.MAX_VALUE);
   return attempts[floors][balls];
}

private static void generateRec(int[][] attempts, int b, int n, int i, int min) {
   if(b == attempts[0].length) {
       return;
   } else if(n == attempts.length) {
       generateRec(attempts,b+1,2,1,min);
```

```
    } else if(i == n) {
        attempts[n][b] = min;
        generateRec(attempts,b,n+1,1,Integer.MAX_VALUE);
    } else {
        int max = Math.max(attempts[i-1][b-1], attempts[n-i][b]) + 1;
        if(min > max) {
            min = max;
        }
        generateRec(attempts,b,n,i+1,min);
    }
}

public static void main(String[] args) {
    System.out.println("minimal: " + minimalAttempts(105,2));
}
```

סיבוכיות זמן ריצה (לא כולל הדפסה) : $O(n^2 \cdot b)$. n=מס' הקומות , b = מספר הכדורים.

מימוש עבור **2** כדורים בלבד ו-**n** קומות

```
public static int twoBalls(int n) {
    int[] f = new int[n+1];
    f[0] = 0;
    f[1] = 1;

    for (int i = 3; i <= n; i++) {
        int min = n;
        for (int j = 1; j < i-1; j++) {
            int x = Math.max(j, f[i-j]+1);
            if (x < min)
                min = x;
        }
        f[i] = min;
    }
    return f[n];
}
```

מימוש עבור **3** כדורים בלבד ו-**n** קומות

```
public static int threeBalls(int n) {
    int[] f3 =new int[n+1];
    if(n==1) {
        f3[n] =1;
    }
    else if(n==2) {
```

```
        f3[n] = 2;
    }
    else { // if n>=3
        int[] f2 = new int[n+1];
        for(int i = 1; i < n ; i++) {
            f2[i] = twoBalls(i);
        }
        f3[0] = 0;
        f3[1] = 1;
        f3[2] = 2;
        f3[3] = 2;
        for(int i = 4; i <= n ; i++) {
            int min = n;
            for(int j = 1; j < i ; j++) {
                int x = Math.max(f2[j-1]+1, f3[i-j]+1);

                if(x < min) {
                    min = x;
                }
            }
            f3[i] = min;
        }
    }
    return f3[n];
}
```

LDS - תת-הסדרה היורדת הארוכה ביותר (לא נלמד)

תיאור הבעיה

נתון מערך של מספרים. יש למצוא את אורך תת הסדרה היורדת הארוכה ביותר מתוך המערך.

דוגמה

עבור הסדרה במערך:

נקבל את הפתרונות הבאים:

$$12, 8, 0 \quad OR \quad 10, 4, 0 \quad OR \quad 12, 4, 0 \quad OR \quad 10, 8, 0 \quad OR \quad 10, 2, 0$$

מימוש בגיטהאב

פתרון הבעיה

- ❖ אפשרות א' - אלגוריתם חמדני
  - ⇚ נקבע שהאיבר הראשון יהיה תחילת הסדרה.
  - ⇚ נעבור על המערך וניקח בכל שלב את האיבר הבא שקטן מקודמו עד שנגיע לסוף המערך.
  - ⇚ לדוגמא, עבור הסדרה הבאה: 5 ,2 ,3 ,4אנחנו ניקח את 4, אחר כך את 3 , 2 ואז נתקל בסוף ב-5 ושעולה מ-2 ולך לא יכנס לתת הסדרה של הפתרון.

```
public static Stack<Integer> greedy(int[] arr) {
   Stack<Integer> sequence = new Stack<>();
   sequence.add(arr[0]);
   for(int i = 1 ; i < arr.length; i++) {
       if(sequence.peek() > arr[i])
           sequence.add(arr[i]);
   }
   return sequence;
}

public static void main(String[] args) {
   int[] arr = {4,3,2,5};
   System.out.println(greedy(arr)); // [4, 3, 2]
}
```

- ⇜ **סיבוכיות**: אנחנו עוברים פעם אחת על כל המערך ולכן $O(n)$.
- ⇜ **נכונות השיטה**: אלגוריתם זה יחזיר את הסדרה היורדת הראשונה שימצא ולא בהכרח הארוכה ביותר בסדרה ולכן זה לא אלגוריתם טוב.

❖ אפשרות ב' - אלגוריתם חמדני משופר

⇚ נרוץ על המערך ועבור על איבר נפעיל את האלגוריתם החמדני שהצגנו באפשרות א'.

⇚ בסוף כל הפעלה נבצע השוואה ונמצא את תת הסדרה היורדת הארוכה ביותר מבין כל האיטרציות שביצענו.

⇚ לדוגמה, נתבונן בסדרה $\{7,\ 6,\ 5,\ 4,\ 3,\ 2,\ 101,\ 100,\ 1\}$.
נקבל תחילה את הסדרה 7,6,5,4,3,2 כאשר 7 הוא ראש הסדרה, אחר כך נמשיך החל מראש הסדרה 6 ונקבל 6,5,4,3,2 וכו'...

```java
public static Stack<Integer> greedy(int[] arr, int start) {
   Stack<Integer> sequence = new Stack<>();
   sequence.add(arr[start]);

   for(int i = start+1; i < arr.length; i++) {
       if(sequence.peek() > arr[i])
           sequence.add(arr[i]);
   }

   return sequence;
}

public static Stack<Integer> improved(int[] arr) {
   Stack<Integer> sequence = new Stack<>();

   for(int i = 0 ; i < arr.length; i++) {
       Stack<Integer> temp_seq = greedy(arr,i);
       if(temp_seq.size() > sequence.size())
           sequence = temp_seq;
   }
   return sequence;
}

public static void main(String[] args) {
   int[] arr = {7, 6, 5, 4, 3, 2, 101, 100, 1};
   System.out.println(improved(arr)); // prints [7, 6, 5, 4, 3, 2, 1]
}
```

⇜ **סיבוכיות**: על כל איבר חוזרים ובודקים את המשך המערך החל ממנו ולכן $O(n^2)$.

⇜ **נכונות השיטה**: האלגוריתם לא החזיר את התשובה הנכונה כי לא פתרנו את הבעיה שלפעמים כדאי לוותר על מספר איברים כדי לקחת אחרים טובים יותר.

❖ אפשרות ג' - אלגוריתם באמצעות LCS

⇐ נשתמש באלגוריתם של LCS (מציאת תת-המחרוזת המשותפת הארוכה ביותר בין 2 מחרוזות).

⇐ נשכתב את האלגוריתם שיתאים ל-2 מערכים של מספרים ונפעיל אותו על המערך הנתון ועל המערך הנתון לאחר מיון כלומר $LCS(arr,\ Sort(arr))$.

⇐ בשונה מ-LIS, הפעם נמיין את המערך **בסדר יורד** ולא בסדר עולה.

⇐ האלגוריתם עובד בשיטת תכנות דינאמי של LCS.

```
public static int[] LCS(int[] X) {
   int[] temp_Y = new int[X.length];

   for(int i = 0; i < X.length; i++)
       temp_Y[i] = X[i];

   Arrays.sort(temp_Y);

   int[] Y = new int[X.length];

   for(int i = X.length-1; i >= 0 ; i--) {
       Y[X.length-1-i] = temp_Y[i];
   }

   int[][] matrix = new int[X.length+1][Y.length+1];
   generateMatrix(matrix,X,Y,1,1);

   int i = matrix.length - 1;
   int j = matrix.length - 1;
   int end = matrix[i][j];
   int start = 0;
   int[] solution = new int[end];

   while(start < end) {
       if(X[i-1] == Y[j-1]) {
           solution[end-start-1] = X[i-1];
           i--;
           j--;
           start++;
       }
       else if(matrix[i-1][j] >= matrix[i][j-1]) {
           i--;
       }
       else {
           j--;
       }
   }
   return solution;
}
```

```java
public static void generateMatrix(int[][] matrix, int[] X, int[] Y, int i, int j) {
   if(i == matrix.length)
       return;
   if(j == matrix.length) {
       System.out.println();
       generateMatrix(matrix,X,Y,i+1,1);
   } else {
       if(X[i-1] == Y[j-1]) {
           matrix[i][j] = matrix[i-1][j-1] + 1;
       }else {
           matrix[i][j] = Math.max(matrix[i-1][j], matrix[i][j-1]);
       }
       System.out.print(matrix[i][j] + " ");
       generateMatrix(matrix,X,Y,i,j+1);
   }
}
public static void main(String[] args) {
   int[] arr = {7, 6, 5, 4, 3, 2, 101, 100, 1};
   System.out.println(Arrays.toString(LCS(arr)));
   // prints [7, 6, 5, 4, 3, 2, 1]
}
```

- **סיבוכיות**: סיבוכיות ההעתקה$O(n)$, סיבוכיות המיון $O(nlogn \ + \ n)$וסיבוכיות בניית המטריצה היא $O(n^2)$ ולכן סה"כ נקבל $O(n^2 \ + \ nlogn \ + \ 2n)$.

- **נכונות השיטה**: מכיוון שהמטרה היא למצוא תת סדרה היורדת הארוכה ביותר אז בהכרח התשובה היא תת סדרה של המערך הממוין כך ששומרים על סדר האיברים במערך ולכן תת הסדרה המשותפת בין 2 המערכים היא בדיוק התשובה.

❖ אפשרות ד' - אלגוריתם באמצעות חיפוש שלם

⇚ נייצר את כל תתי המערכים האפשריים.

⇚ נבדוק עבור על תת מערך אם הוא תת סדרה יורדת (מתחילתו ועד סופו).

⇚ אם כן, נבדוק האם הוא הכי ארוך שמצאנו עד עכשיו ונשמור אותו.

⇚ הפונקציה תחזיר בסוף מערך של תת הסדרה היורדת הגדולה ביותר.

```java
public static Vector<int[]> subsets(int[] arr) {
   Vector<int[]> subsets = new Vector<>();
   int length = (int)Math.pow(2,arr.length)-1;

   for(int decimal = 0; decimal < length; decimal++) {
       int binary = decimal;
       int i = 0;
```

```
        Vector<Integer> subset = new Vector<>();
        while (binary != 0) {
            if(binary % 2 == 1) {
                subset.add(arr[i]);
            }
            i++;
            binary /= 2;
        }
        int[] ss = new int[subset.size()];

        for(int j = 0 ; j < ss.length; j++) {
            ss[j] = subset.get(j);
        }
        subsets.add(ss);
    }
    return subsets;
}

public static int[] bruteForce(int[] arr) {

    Vector<int[]> vector = subsets(arr);
    int[] bestSubset = new int[0];
    for(int i = 0 ; i < vector.size(); i++) {
        boolean isDecreasing = true;
        int[] subset = vector.get(i);
        for(int j = 1; j < subset.length; j++) {
            if(subset[j-1] < subset[j]){
                isDecreasing = false;
            }
        }
        if(isDecreasing && bestSubset.length < subset.length) {
            bestSubset = subset;
        }
    }
    return bestSubset;
}

public static void main(String[] args) {
    int[] arr = {7, 6, 5, 4, 3, 2, 101, 100, 1};
    System.out.println(Arrays.toString(bruteForce(arr)));
    // print [7, 6, 5, 4, 3, 2, 1]
}
```

- **סיבוכיות**: מספר תתי הקבוצות הוא $2^n - 1$ וגם על כל תת-מערך עוברים ובודקים האם הוא ממוין בסדר יורד כך שלכל היותר תת מערך כזה הוא בגודל n ולכן סה"כ $O(2^n \cdot n)$.
- **נכונות השיטה**: בודקים את כל האפשרויות ולכן בהכרח נגיע גם לתשובה הנכונה.

שאלה

האם אנחנו רוצים למצוא את אורך המחרוזת או רק דוגמא למחרוזת המקיימת LDS?

❖ אפשרות ה' - מציאת אורך המחרוזת.

⇐ נדרוס איברים תוך כדי שמירה על האורך הנכון.

```
public static int binarySearch(int[] sequence, int left, int right, int value) {

   while(right - left > 1) { // while we can compare min two values

       int middle = (right+left)/2;

       if(value >= sequence[middle]) {
           right = middle;
       }
       else if(value < sequence[middle]) {
           left = middle;
       }
   }
   return right;
}
public static int length(int[] arr) {
   int[] sequence = new int[arr.length];
   sequence[0] = arr[0];
   int length = 0;
   for(int i = 1 ; i < sequence.length; i++) {
       if(arr[i] > sequence[0]) {
           sequence[0] = arr[i];
       }
       else if(arr[i] < sequence[length]) {
           length++;
           sequence[length] = arr[i];
       } else {
           int index = binarySearch(sequence,0,length,arr[i]);
           sequence[index] = arr[i];
       }
   }
   System.out.println(Arrays.toString(sequence));
   return length+1;
}
public static void main(String[] args) {
   int[] arr = {11,8,7,3,6,20,4,9,5};
   System.out.println(length(arr)); // prints 5
}
```

⇜ **סיבוכיות זמן הריצה**: על כל איבר בסדרה נחפש איפה לשים אותו בתת הסדרה שאנו יוצרים.

נעשה זאת בחיפוש בינארי כדי לחסוך, סה"כ $O(n \cdot logn)$.

ארדש-סקרש (Erdos-Szekeres)

(רלוונטי להוכחות, חשיבה תיאורטית ויכול לתת מענה לפתרון שאלה יצירתית במבחן).

במתמטיקה דיסקרטית, **משפט ארדש-סקרש** הוא משפט הקובע, שלכל s ,r טבעיים, בכל סדרה באורך sr+1 של מספרים ממשיים שונים יש תת-סדרה עולה באורך s+1 או תת-סדרה יורדת באורך r+1.

משפט

יהא $A = (a_1,...,a_n)$ סדרה של n מספרים ממשיים שונים.

אם $n \geq sr + 1$ אז ל-A יש תת-סדרה עולה של s+1 או תת סדרה יורדת באורך r+1 (או גם וגם).

הוכחה

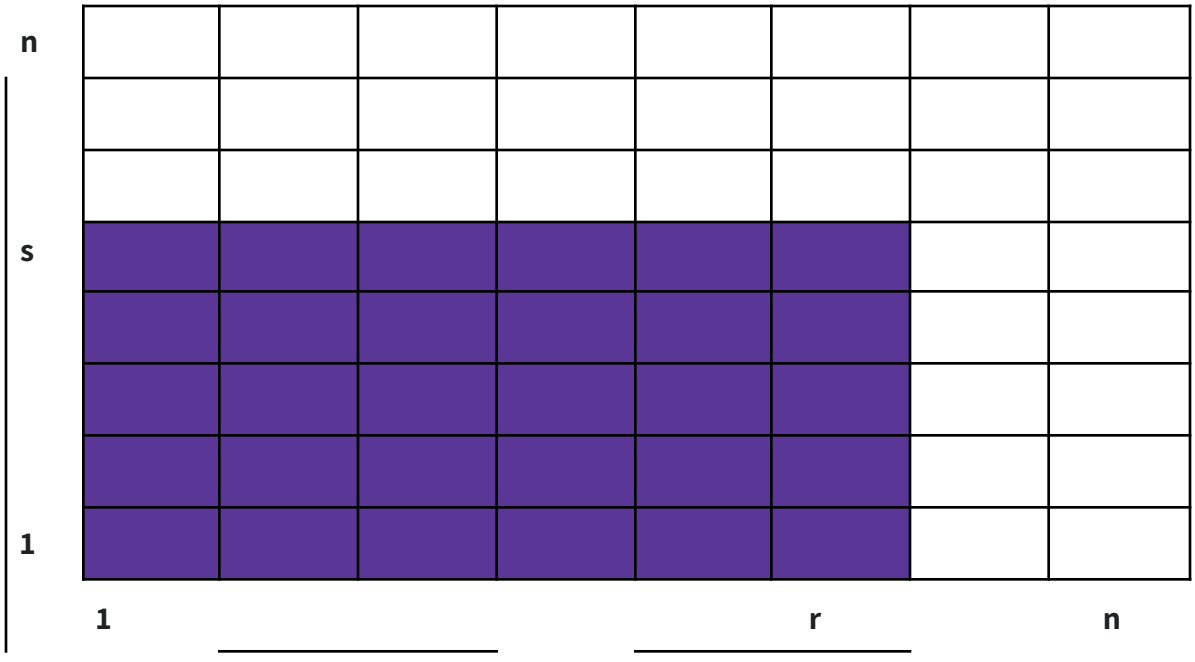
נקשר לכל איבר a_i של A , זוג מהצורה (x_i, y_i) כאשר x_i הוא מספר האיברים בסדרה העולה הארוכה ביותר המסתיימת ב-a_i ו- y_i הוא מספר האיברים בסדרה היורדת הארוכה ביותר המתחילה ב-a_i.

וגם $(x_i, y_i) \neq (x_j, y_j)$ תמיד כאשר $i \neq j$.

אכן, אם יש לנו מקרה של: ..... a_i ..... a_j ...,

- אז $a_i < a_j$ ומספר האיברים בסדרה העולה הארוכה ביותר המסתיימת ב-a_i יכולה להתרחב על ידי a_j (כך ש: $x_i < x_j$).
- או $a_i > a_j$ ומספר האיברים בסדרה היורדת הארוכה ביותר המתחילה ב-a_j ויכול להמשיך על ידי a_i, (כך ש: $y_i > y_j$).

כעת ניצור לוח משבצות של n^2 שובכי יונים:



נציב כל איבר a_i בשובך עם קואורדינטות (x_i, y_i).

ניתן להציב את הצמד הזה בחלק מהשובכים, מאחר ו-$1 \le x_i, y_i \le n$ לכל $1 \le i \le n$.

יותר מזה, לא ייתכן שיהיה יותר מצד אחד בכל שובך וזאת מאחר ש-$(x_i, y_i) \ne (x_j, y_j)$ תמיד כאשר $i \ne j$.

מאחר ו-$|A| = n \ge sr + 1$, יש לנו יותר איברים ממספר השובכים (המסומנים בשטח הסגול בתמונה מלמעלה).

לכן קיים איבר a_i מחוץ לסימון השטח הסגול.

אך זה אומר ש- $x_i \ge s + 1$ או ש-$y_i \ge r + 1$ (או שניהם), וזה בדיוק מה שרצינו להוכיח.

∎

מיון מיזוג, חיפוש בינארי ואסימפטוטיקה

אסימפטוטיקה

$$1 < log(n) < nlogn < n^2 < n^3 <\ldots< 2^n < 3^n < n^n$$

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Merge Sort

סיבוכיות $O(n \cdot log(n))$ $\Theta(n \cdot log(n))$ $\Omega(n \cdot log(n))$

מיון זה הוא רקורסיבי. הוא מחלק את המערך לשתי קבוצות, כל קבוצה הוא שוב מחלק לשניים וכן האלה (באופן רקורסיבי) עד תנאי העצירה – קבוצות בנות איבר אחד.

בשלב השני הפונקציה ממזגת כל תת-קבוצה עם תת-קבוצה אחרת, באמצעות מיזוג של שתי קבוצות ממוינות, וכן הלאה עד להיווצרות מערך ממוין.

הסבר

בכל קריאה לפונקציה **Merge-Sort** מחלקים את המערך לשניים, לכן עלינו לחשב כמה פעמים יש לחלק מספר בשניים עד שנגיע ל-0. במילים אחרות, כמה פעמים נצטרך להכפיל את 2 בעצמו (חזקה) על מנת להגיע למספר המבוקש:

$$2^x = N \Rightarrow x = log_2 N \Rightarrow O(log_2 N)$$

לאחר כל חלוקה קוראים לפונקציה ***Merge***. הסיבוכיות שלה היא $O(n)$ משום שהיא עוברת באופן סדרתי על שני חלקי של המערך ומשוואה איבר לאיבר. בנוסף לזה הפונקציה עוברת עוד 3 פעמים על המערך:

$$3 \cdot O(n) + O(n) = O(n)$$

הסיבוכיות של הפונקציה **Merge-Sort** היא:

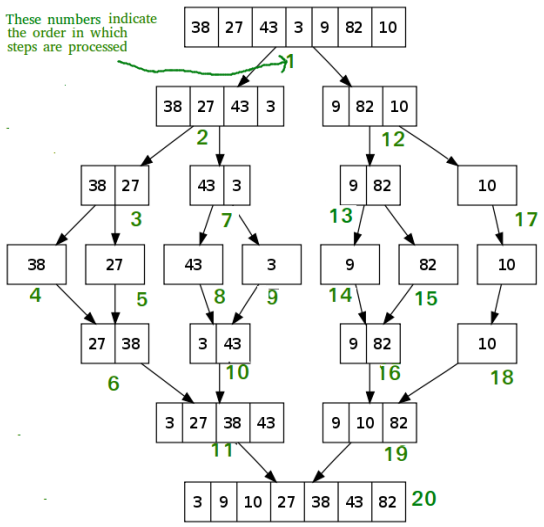
$$O(log_2 n) \cdot O(n) = O(n \cdot log_2 n)$$

[סרטון אלגוריתם]

מימוש

```
private static void mergeSort(int[] arr) {
   mergeSort(arr,0,arr.length-1);
}

private static void mergeSort(int[] arr, int left, int right) {
   if(left < right) {
       int middle = (left+right)/2;

       mergeSort(arr,left,middle); // left
```

```
        mergeSort(arr,middle+1,right); // right
        Merge(arr,left,middle,right);
    }
}

private static void Merge(int[] arr, int left, int middle, int right) {

    int[] temp = new int[right - left + 1];

    int i = left; // left half
    int j = middle + 1; // right half
    int k = 0; // The Running Pointer

    while( i <= middle && j <= right) {

        if(arr[i] < arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while(i <= middle)
        temp[k++] = arr[i++];

    while(j <= right)
        temp[k++] = arr[j++];


    for(i = left, k = 0 ; k < temp.length && i <= right; k++, i++)
        arr[i] = temp[k];
}

public static void main(String[] args) {
    int[] arr = {48,3,7,9,43,1,2,4,6,8};
    mergeSort(arr);
    System.out.println(Arrays.toString(arr));
}
```



Binary Search

סיבוכיות $O(log(n))$ $\Theta(log(n))$ $\Omega(log(n))$

הסבר

בכל קריאה לפונקציה ***binarySearch*** אנו מחלקים את המערך לשניים, על כן עלינו לחשב את מספר הפעמים שבהן נצטרך לחלק מספר בשניים עד שנגיע ל-0. במילים אחרות, כמה פעמים נצטרך להכפיל את 2 בעצמו (חזקה) על מנת להגיע למספר המבוקש:

$$2^x = N \Rightarrow x = log_2 N \Rightarrow O(log_2 N)$$

[סרטון אלגוריתם]

מימוש

```java
public static int binarySearch(int arr[], int left, int right, int x) {

   if(right >= 1) {

       int middle = left + (right - left)/2;

       if(arr[middle] == x)
           return middle;

       if(arr[middle] > x)
           return binarySearch(arr, left, middle - 1 , x);

       else
           return binarySearch(arr, middle + 1 , right, x);
   }
   return -1;
}

public static void main(String[] args) {
   int[] arr = {48,3,7,9,43,1,2,4,6,8};
   System.out.println(binarySearch(arr,0,arr.length-1,43)); // 4
}
```

חלאס.