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

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.

```
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:

0	1	2	3	4	5
1	3	6	1	3	6

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:

0	1	2	3
4	1	5	9

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

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:

0	1	2	3
4	1	5	9

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

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

	0	1	2	3
0	4			
1		1		
2			5	
3				9

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:

	0	1	2	3
0	4			
1		1		
2			5	
3				9

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

וכך נפעל עבור כל שאר האלכסון הירוק באיור הבא.
עבור המשחק בתא **הכתום** נחשב:

	0	1	2	3
0	4	3		
1		1	4	
2			5	4
3				9

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

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

	0	1	2	3
0	4	3	2	7
1		1	4	5
2			5	4
3				9

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 nxn לחישוב הרווחים ולכן $O(n^2)$.