java.util Methods for Arrays:

fill( A, x)=تستخدم لمىء المصفوفة

copyOf(A, n)تستخدم لنسخ الى مصفوفة جديدة بحجم= A

copyOfRange(A, s, t=:تستخدم بين المؤشرين لنسخ جزء من المصفوفة الى مصفوفة جديدة

toString(A)تستخدم لتحويل المصفوفة الي سلسلة نصية تحتوي على تمثيل نصي لعناصر المصفوفة =

sort(A)تستخدم لفرز المصفوفات وترتيب عناصرها بترتيب تصاعدي =

binarySearch(A, x)تستخدم للبحث في مصفوفات مرتبة باستخدام البحث الثنائي =

1-السؤال الأول :

الخطوات العامة لتوليد الأعداد العشوائية باستخدام طريقة مولد التدرج الخطي (LCG)

1. التهيئة: قم بتعيين القيمة الأولية (البذور) cur إلى القيمة المحددة (92 في هذه الحالة).
2. التوليد: استخدم الصيغة next = (a \* cur + b) % n لحساب الرقم العشوائي التالي.
3. التحديث: قم بتعيين cur إلى next للدورة التالية.
4. التكرار: كرر الخطوات 2 و 3 لتوليد العدد المطلوب من الأرقام العشوائية.

الحسابات لـ a = 12، b = 5، n = 100، و seed = 92:

1. العدد الأول:
   * next = (12 \* 92 + 5) % 100
   * next = 1109 % 100
   * next = 9
2. العدد الثاني:
   * next = (12 \* 9 + 5) % 100
   * next = 113 % 100
   * next = 13
3. العدد الثالث:
   * next = (12 \* 13 + 5) % 100
   * next = 161 % 100
   * next = 61
4. العدد الرابع:
   * next = (12 \* 61 + 5) % 100
   * next = 737 % 100
   * next = 37
5. العدد الخامس:
   * next = (12 \* 37 + 5) % 100
   * next = 449 % 100
   * next = 49

وبالتالي، فإن الخمسة الأرقام العشوائية التالية التي تم إنشاؤها باستخدام المعلمات المحددة هي:

9، 13، 61، 37، 49

تذكر: هذه الأرقام ليست عشوائية حقًا، ولكنها تظهر بعض خصائص الأرقام العشوائية وهي غالبًا كافية للعديد من التطبيقات. تعتمد جودة الأرقام العشوائية التي يتم إنشاؤها على اختيار a و b و n.

2-السؤال الثاني:

import java.util.Random;

public static void removeRandomEntries(Object[] array) {

Random random = new Random();

while (array.length > 0) {

int randomIndex = random.nextInt(array.length);

Object removedElement = array[randomIndex];

array[randomIndex] = array[array.length - 1]; // Swap with the last element

array = Arrays.copyOf(array, array.length - 1); // Remove the last element (which is now the duplicate)

System.out.println("Removed element: " + removedElement);

}

}

3-السؤال الثالث :

شرح التغييرات التي يجب إجراؤها على برنامج Code Fragment 3.8 بحيث يمكنه تنفيذ الشفرة السيزارية لرسائل مكتوبة بلغة أبجدية أخرى غير الإنجليزية، مثل اليونانية أو الروسية أو العبرية:

لجعل برنامج Code Fragment 3.8 قادرًا على تنفيذ الشفرة السيزارية لرسائل مكتوبة بلغات أبجدية أخرى غير الإنجليزية، مثل اليونانية أو الروسية أو العبرية، يجب إجراء بعض التغييرات الرئيسية:

1. تغيير المصفوفة alphabet: يجب استبدال مصفوفة alphabet المضمنة في البرنامج والتي تحتوي على الحروف الأبجدية الإنجليزية بأخرى تحتوي على الحروف الأبجدية للغة الهدف. على سبيل المثال، لتشفير اللغة اليونانية، يمكن استخدام المصفوفة التالية:

alphabet = "αβγδεζηθικλμνξοπρστυφχψω";

1. تعديل دالة encrypt: يجب تعديل دالة encrypt لأخذ بعين الاعتبار ترتيب الحروف في اللغة الهدف. بدلاً من استخدام معامل shift ثابت، يمكن استخدام دالة تحسب الإزاحة بناءً على ترتيب الحروف في اللغة الهدف. على سبيل المثال، للدلالة على أن الحرف a يتقدم بمقدار 3 أماكن في اللغة اليونانية، يمكن استخدام الدالة التالية:

int shiftGreek(char ch) {

if (ch >= 'α' && ch <= 'ω') {

return ch - 'α' + 3;

} else {

return ch;

}

}

1. معالجة الرموز الخاصة: بعض اللغات، مثل اللغة الروسية، تستخدم رموزًا خاصة مثل الألفبائية السيريلية. يجب تعديل البرنامج لمعالجة هذه الرموز بشكل صحيح، إما عن طريق ترميزها بشكل مختلف أو باستبعادها من عملية التشفير.
2. التعامل مع الحروف الكبيرة والصغيرة: بعض اللغات، مثل اللغة العبرية، تتميز باختلاف بين الحروف الكبيرة والصغيرة. يجب تحديد ما إذا كان سيتم الحفاظ على حالة الحروف أثناء التشفير أو عدم ذلك.
3. معالجة الحروف المتحركة والثابتة: بعض اللغات، مثل اللغة اليونانية، تقسم الحروف إلى حروف متحركة (تتحرك) وحروف ثابتة (لا تتحرك). قد يحتاج البرنامج إلى تعديل لمعالجة هذه الميزة بشكل صحيح.

4-السؤال الرابع:

public class TicTacToe {

private char[][] board;

private boolean gameOver;

public TicTacToe() {

board = new char[3][3];

resetBoard();

}

public void putMark(int row, int col, char mark) {

if (gameOver) {

throw new IllegalStateException("Game is already over");

}

if (board[row][col] != ' ') {

throw new IllegalArgumentException("Cell is already occupied");

}

board[row][col] = mark;

checkWinner();

}

private void checkWinner() {

for (int i = 0; i < 3; i++) {

if ((board[i][0] == board[i][1] && board[i][1] == board[i][2]) && board[i][0] != ' ') {

gameOver = true;

return;

}

if ((board[0][i] == board[1][i] && board[1][i] == board[2][i]) && board[0][i] != ' ') {

gameOver = true;

return;

}

}

if ((board[0][0] == board[1][1] && board[1][1] == board[2][2]) && board[0][0] != ' ') {

gameOver = true;

return;

}

if ((board[0][2] == board[1][1] && board[1][1] == board[2][0]) && board[0][2] != ' ') {

gameOver = true;

return;

}

if (!isBoardFull()) {

gameOver = false;

return;

}

gameOver = true;

}

public boolean isGameOver() {

return gameOver;

}

public boolean isWinner(char mark) {

return (board[0][0] == mark && board[0][1] == mark && board[0][2] == mark)

|| (board[1][0] == mark && board[1][1] == mark && board[1][2] == mark)

|| (board[2][0] == mark && board[2][1] == mark && board[2][2] == mark)

|| (board[0][0] == mark && board[1][0] == mark && board[2][0] == mark)

|| (board[0][1] == mark && board[1][1] == mark && board[2][1] == mark)

|| (board[0][2] == mark && board[1][2] == mark && board[2][2] == mark)

|| (board[0][0] == mark && board[1][1] == mark && board[2][2] == mark)

|| (board[0][2] == mark && board[1][1] == mark && board[2][0] == mark);

}

public char getMark(int row, int col) {

return board[row][col];

}

public boolean isBoardFull() {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == ' ') {

return false;

}

}

}

return true;

}

public void resetBoard() {

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

board[i][j] = ' ';

}

}

gameOver = false;

}

}

5-السؤال الخامس :

فيما يلي شرح الفرق بين اختبارات المساواة السطحية والعميقة للمصفوفات أحادية وثنائي الأبعاد في جافا:

المصفوفات أحادية البعد (int[]):

* الاختبار السطحي للمساواة (==): يقارن فقط ما إذا كانت المصفوفتان تشيران إلى نفس الموقع في الذاكرة. أي أنهما نفس الكائن الحرفي. لن يتحقق مما إذا كانت تحتوي على نفس القيم أم لا.
* الاختبار العميق للمساواة (Arrays.deepEquals()): يتحقق من المساواة من خلال مقارنة كل عنصر فردي في المصفوفتين، حتى لو كانت المصفوفتان كائنين منفصلين. لذلك، سيحدد بشكل صحيح ما إذا كانت المصفوفتان تحتويان على نفس القيم بالترتيب نفسه.

المصفوفات ثنائية الأبعاد (int[][])::

* الاختبار السطحي للمساواة (==): كما في الحالة السابقة، يتحقق فقط من أن المصفوفتين تشيران إلى نفس الموقع في الذاكرة. لن يتحقق من محتويات المصفوفات.
* الاختبار العميق للمساواة (Arrays.deepEquals()): في حالة المصفوفات ثنائية الأبعاد، يقوم بمقارنة العناصر الموجودة في الصفوف والأعمدة المقابلة في كلتا المصفوفتين. بمعنى آخر، سيقارن كل عنصر فردي داخل كل صف وعمود لتحديد المساواة الحقيقية.

6-السؤال السادس:

فيما يلي شرح خوارزمية للعثور على الرقم المكرر في مصفوفة A تحتوي على أرقام من 1 إلى n - 1 inclusive، منها واحد مكرر:

الخطوة 1: إنشاء مصفوفة visited:

أولاً، نحتاج إلى إنشاء مصفوفة visited من حجم n. في البداية، نقوم بتعيين جميع عناصر visited إلى 0. ستستخدم هذه المصفوفة لتتبع الأرقام التي تم رؤيتها بالفعل.

الخطوة 2: استكشاف المصفوفة:

بعد ذلك، نقوم باستكشاف المصفوفة A. بالنسبة لكل عنصر x في المصفوفة A، نقوم بما يلي:

* التحقق مما إذا كان العنصر قد تم رؤيته بالفعل: إذا كانت visited[x - 1] == 1، فإن العنصر x هو العنصر المكرر. نقوم بإرجاع x.
* تحديد العنصر على أنه مرئي: نقوم بتعيين visited[x - 1] = 1 للإشارة إلى أن العنصر x قد تم رؤيته.

التحليل:

تتمتع الخوارزمية بزمن تشغيل معقد من O(n)، حيث n هو حجم المصفوفة. وذلك لأن الخوارزمية تستكشف المصفوفة مرة واحدة وتتحقق من كل عنصر بحد أقصى مرة واحدة.

تتمتع الخوارزمية أيضًا بمساحة معقدة من O(n)، حيث n هو حجم المصفوفة. وذلك لأن الخوارزمية تستخدم مصفوفة مساعدة visited من حجم n.

مثال:

لنفترض أن المصفوفة A هي التالية:

A = [1, 2, 3, 2]

ستؤدي الخوارزمية إلى استخراج العنصر 2، وهو العنصر المكرر.

التنفيذ:

فيما يلي تنفيذ الخوارزمية بلغة جافا:

Java

public class FindRepeatingInteger {

public static int findRepeatingInteger(int[] array) {

int[] visited = new int[array.length];

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

if (visited[array[i] - 1] == 1) {

return array[i];

}

visited[array[i] - 1] = 1;

}

return -1;

}

public static void main(String[] args) {

int[] array = {1, 2, 3, 2};

int repeatingInteger = findRepeatingInteger(array);

System.out.println("The repeating integer is " + repeatingInteger);

}

}

سيؤدي هذا الكود إلى طباعة ما يلي:

The repeating integer is 2

7- السؤال السابع:

إليك شرح خوارزمية للعثور على العدد المكرر في مصفوفة A بحجم n ≥ 2 تحتوي على أعداد صحيحة من 1 إلى n−1 inclusive:

الخطوات:

1. إنشاء مصفوفة علامات: نبدأ بإنشاء مصفوفة علامات visited بحجم n ونضبط جميع عناصرها على 0. سنستخدم هذه المصفوفة لتتبع الأعداد التي رأيناها بالفعل.
2. تصفح المصفوفة A: نقوم بتصفح كل عنصر x في المصفوفة A وتنفيذ التالي:
   * التحقق من تكرار العنصر: إذا كانت قيمة visited[x - 1] تساوي 1، فهذا يعني أننا رأينا العدد x من قبل، وبالتالي فهو العدد المكرر. نقوم بإرجاع x فوراً.
   * وضع علامة على العنصر كـ "مرئي": في حال لم يكن العدد x مكرراً، نقوم بوضع علامة عليه عن طريق ضبط visited[x - 1] على 1.

التعقيد الزمني: تتمتع هذه الخوارزمية بتعقيد زمني O(n)، حيث n هو حجم المصفوفة. ويعود ذلك إلى أننا نقوم بتصفح المصفوفة مرة واحدة فقط، ونتحقق من كل عنصر مرة واحدة على الأكثر.

التعقيد المكاني: تتمتع الخوارزمية أيضاً بتعقيد مكاني O(n)، وذلك بسبب استخدامنا لمصفوفة العلامات visited بحجم n.

مثال:

لنفرض أن لدينا المصفوفة A التالية:

A = [1, 2, 3, 2]

ستؤدي الخوارزمية إلى إيجاد العدد 2 باعتباره العدد المكرر.

8-السؤال الثامن:

فيما يلي خوارزمية للعثور على الأعداد الخمسة المكررة في مصفوفة B بحجم n ≥ 6 تحتوي على أعداد صحيحة من 1 إلى n−5 inclusive:

الخطوات:

1. إنشاء مصفوفة علامات: ننشئ مصفوفة علامات visited بحجم n−5 ونضبط جميع عناصرها على 0.
2. تصفح المصفوفة B: لكل عنصر x في المصفوفة B، نقوم بما يلي:
   * التحقق من تكرار العنصر: إذا كانت قيمة visited[x - 1] تساوي 1، إذن x هو أحد الأرقام المكررة. نضيف x إلى قائمة الأرقام المكررة.
   * وضع علامة على العنصر كـ "مرئي": نضبط visited[x - 1] على 1 لتجنب تكرار إضافته إلى قائمة الأرقام المكررة.
3. إرجاع قائمة الأرقام المكررة: بعد تصفح المصفوفة B بالكامل، نعيد قائمة الأرقام المكررة التي تم جمعها.

التعقيد الزمني: تتمتع الخوارزمية بتعقيد زمني O(n)، حيث n هو حجم المصفوفة. إذ نقوم بتصفح المصفوفة مرة واحدة فقط، ونجري عمليات ثابتة لكل عنصر.

التعقيد المكاني: تتمتع الخوارزمية بتعقيد مكاني O(n−5)، بسبب استخدام مصفوفة العلامات visited بحجم n−5 والقائمة المستخدمة لتخزين الأرقام المكررة (بحجم أقصى 5).

مثال:

لنفرض أن لدينا المصفوفة B التالية:

B = [1, 2, 3, 4, 2, 5, 4, 6, 3]

ستؤدي الخوارزمية إلى إيجاد الأرقام المكررة {2, 3, 4, 2, 4}.

9-السؤال التاسع:

إليك كود Java لتنفيذ طرق add(e) و remove(i) لفئة Scoreboard بدون الحفاظ على ترتيب إدخالات اللعبة، مع عدد خطوات لا يعتمد على n:

Java

public class Scoreboard {

private int numEntries = 0; // عدد الإدخالات الحالية

private GameEntry[] board; // مصفوفة لإدخالات اللعبة

public Scoreboard(int capacity) {

board = new GameEntry[capacity];

}

// إضافة إدخال لعبة جديد بدون الحفاظ على الترتيب

public void add(GameEntry e) {

if (numEntries < board.length) {

// استبدال الإدخال الأخير بالإدخال الجديد

board[numEntries] = e;

numEntries++;

}

}

// إزالة إدخال لعبة في الفهرس i بدون الحفاظ على الترتيب

public GameEntry remove(int i) throws IndexOutOfBoundsException {

if (i < 0 || i >= numEntries) {

throw new IndexOutOfBoundsException("Invalid index: " + i);

}

GameEntry removedEntry = board[i];

// استبدال الإدخال المراد إزالته بالإدخال الأخير

board[i] = board[numEntries - 1];

board[numEntries - 1] = null; // تعيين الإدخال الأخير إلى null

numEntries--;

return removedEntry;

}

// ... طرق أخرى من فئة Scoreboard

}

شرح:

* add(e):
  + يتحقق من وجود مساحة كافية في المصفوفة.
  + إذا كانت هناك مساحة، فإنه يضع الإدخال الجديد مباشرة في آخر موقع فارغ في المصفوفة ويزيد عدد الإدخالات. لا حاجة لاستخدام حلقة لتبديل الإدخالات.
* remove(i):
  + يتحقق من صحة الفهرس i.
  + يحفظ الإدخال المراد إزالته في متغير مؤقت.
  + ينسخ الإدخال الأخير في المصفوفة إلى الفهرس i، مما يؤدي إلى استبدال الإدخال المراد إزالته.
  + يعين الإدخال الأخير في المصفوفة إلى null لتجنب الاحتفاظ بمراجع غير ضرورية.
  + ينقص عدد الإدخالات. لا حاجة لاستخدام حلقة لتبديل الإدخالات.

لا تعتمد هذه الطرق على حلقات، لذلك لا يعتمد عدد الخطوات على عدد الإدخالات n.

10-السؤال العاشر:

فيما يلي بعض الأمثلة لقيم a و b في المولد شبه العشوائي الموصوف في الصفحة 113 من هذا الفصل والتي تؤدي إلى نتائج ليست عشوائية للغاية عند n = 1000:

المجموعة 1:

* a = 1
* b = 0

يؤدي هذا إلى تسلسل متكرر من نفس الرقم (1) لجميع قيم n، وهو بالتأكيد ليس عشوائياً.

المجموعة 2:

* a = 2
* b = 0

يؤدي هذا إلى تسلسل يزداد بمقدار 2 في كل خطوة، مما ينتج عنه نمط واضح لا يعتبر عشوائياً.

المجموعة 3:

* a = 4
* b = 0

يؤدي هذا إلى تسلسل يزداد بمقدار 4 في كل خطوة، مما ينتج عنه نمط أكثر وضوحاً من التكرار.

المجموعة 4:

* a = 10
* b = 0

ينتج عن هذا تسلسل يزداد بمقدار 10 في كل خطوة، مما يبرز نمط التكرار بشكل أكثر حدة.

في هذه الأمثلة، تكون القيم الناتجة بعيدة عن العشوائية بسبب النماذج والتكرار الواضحة التي تظهر فيها.

من المهم اختيار قيم a و b بعناية في المولدات العشوائية لضمان توزيع جيد للأرقام العشوائية وإنتاج تسلسلات أكثر عشوائية وغير متوقعة.

11-السؤال الحادي عشر:

إليك الشرح بالعربي:

الرقم الذي من المرجح أن يساويه x هو 0.

تفسير ذلك:

* احتمال أن يكون أي عنصر من عناصر A هو 0 هو 1/10.
* نظرًا لوجود 100 عنصر في A، فإن احتمال أن يكون أحدها على الأقل 0 هو 1 - (9/10)^100، وهو ما يزيد عن 0.9999.
* عندما يكون أحد العناصر على الأقل هو 0، فإن حاصل ضرب جميع العناصر سيكون 0، بغض النظر عن قيم العناصر الأخرى.

صيغة احتمال أن يساوي x العدد 0:

P(x = 0) = 1 - (9/10)^100

يشير هذا إلى أن احتمال أن يكون x مساويًا لـ 0 هو أكثر من 0.99.

12-السؤال الثاني عشر :

فيما يلي طريقة shuffle(A) باللغة العربية لإعادة ترتيب عناصر المصفوفة A بشكل عشوائي:

Java

import java.util.Random;

public class ShuffleArray {

public static void shuffle(Object[] A) {

Random random = new Random();

for (int i = A.length - 1; i > 0; i--) {

// اختر عشوائيًا فهرسًا بين 0 و i (شامل)

int j = random.nextInt(i + 1);

// تبادل العناصر في الفهرسين i و j

Object temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

public static void main(String[] args) {

Integer[] A = {1, 2, 3, 4, 5};

shuffle(A);

System.out.println(Arrays.toString(A)); // Output will be a random permutation of the original list

}

}

شرح طريقة shuffle(A):

1. إنشاء كائن Random: ننشئ كائنًا من فئة Random لتوليد أرقام عشوائية.
2. التكرار عبر المصفوفة: نقوم بالتكرار من نهاية المصفوفة إلى العنصر الأول (باستثناءه).
3. اختيار فهرس عشوائي: في كل تكرار، نستخدم nextInt(i + 1) لاختيار فهرس عشوائي j بين 0 و i (شامل).
4. تبديل العناصر: نقوم بتبديل العناصر الموجودة في الفهرسين i و j، مما يؤدي إلى خلط العناصر بشكل عشوائي.

التحليل:

* تعقيد زمني: O(n)، حيث n هو حجم المصفوفة.
* تعقيد مكاني: O(1)، حيث لا نستخدم أي مساحة إضافية بخلاف المتغيرات المؤقتة.
* العشوائية: تضمن هذه الطريقة أن كل ترتيب ممكن للعناصر في المصفوفة A متساوٍ في الاحتمال، مما ينتج عنه خلط عشوائي حقيقي.

13-السؤال الثالث عشر:

إستراتيجية لتعقب لقاءات اللاعبين وتحديد الفائز في لعبة متعددة اللاعبين بأكثر من 1000 لاعب:

المتطلبات:

* n ≥ 1000 لاعبًا مرقمين من 1 إلى n.
* تتحقق الفوز عند لقاء كل لاعب آخر على الأقل مرة واحدة.
* يسمح بالتعادلات.
* تتوفر طريقة meet(i, j) التي تعمل عند اجتماع اللاعبين i و j.

الحل المقترح:

1. استخدام مصفوفة اللقاءات:
   * أنشئ مصفوفة ثنائية الأبعاد بحجم n × n، حيث يتم تمثيل لقاء بين اللاعبين i و j بالعضو (i, j) من المصفوفة.
   * قيمة العضو (i, j) تكون صحيحة (1) إذا تلاقي اللاعبان i و j، وإلا تكون خاطئة (0).
2. تحديث لقاءات اللاعبين:
   * عند استدعاء meet(i, j):
     + قم بتعيين قيمة العضو (i, j) و (j, i) في مصفوفة اللقاءات إلى 1.
3. تعقب تقدم اللاعبين:
   * لكل لاعب i، احسب عدد اللقاءات المسجلة له في مصفوفة اللقاءات (مجموع قيم صف i أو عمود i).
   * استخدم قائمة منفصلة لتخزين اللاعبين الذين بلغوا عدد اللقاءات المطلوب للفوز (n - 1).
4. تحديد الفائز:
   * إذا كانت القائمة منفصلة تحتوي على أكثر من n - 1 لاعبًا، تكون هناك حالة تعادل.
   * خلاف ذلك، فإن اللاعب الوحيد الموجود في القائمة منفصلة هو الفائز.

التحليل:

* تعقيد زمني:
  + استدعاء meet(i, j): O(1) (تحديث مصفوفة اللقاءات).
  + حساب تقدم اللاعب: O(n) (مجموع قيم صف أو عمود).
  + تحديد الفائز: O(1) (فحص القائمة منفصلة).
* تعقيد مكاني: O(n^2) (لصيانة مصفوفة اللقاءات).

مميزات:

* بسيطة وسهلة الفهم.
* فعالة من حيث الوقت للعمليات الأساسية.
* تعمل بشكل جيد مع عدد كبير من اللاعبين (n ≥ 1000).

عيوب:

* تتطلب مساحة كبيرة لتخزين مصفوفة اللقاءات (عند زيادة عدد اللاعبين).
* يتطلب حسابًا إضافيًا لتتبع تقدم كل لاعب.

حلول بديلة:

* استخدام قوائم مجاورة للتمثيل العلاقات بين اللاعبين، مما يقلل المساحة المطلوبة ولكن يزيد تعقيد العمليات الأخرى.
* استخدام خوارزميات أكثر تطورًا لتحليل العلاقات ومراقبة لقاءات اللاعبين، لكن قد تكون أكثر تعقيدًا في التنفيذ.

اختر الحل المناسب بناءً على متطلبات مساحة التخزين وأداء التشغيل المطلوبة في لعبتك.

14-السؤال الرابع عشر:

فيما يلي طريقة Java باللغة العربية تقوم بجمع مصفوفتين ثلاثية الأبعاد مكونًا تلو الآخر:

Java

public class ArrayAddition3D {

public static int[][][] addArrays3D(int[][][] arr1, int[][][] arr2) {

int rows = arr1.length;

int cols = arr1[0].length;

int depth = arr1[0][0].length;

int[][][] result = new int[rows][cols][depth];

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

for (int k = 0; k < depth; k++) {

result[i][j][k] = arr1[i][j][k] + arr2[i][j][k];

}

}

}

return result;

}

public static void main(String[] args) {

int[][][] arr1 = {

{ {1, 2, 3}, {4, 5, 6} },

{ {7, 8, 9}, {10, 11, 12} }

};

int[][][] arr2 = {

{ {13, 14, 15}, {16, 17, 18} },

{ {19, 20, 21}, {22, 23, 24} }

};

int[][][] sum = addArrays3D(arr1, arr2);

System.out.println("Sum of arrays:");

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

for (int j = 0; j < sum[0].length; j++) {

for (int k = 0; k < sum[0][0].length; k++) {

System.out.print(sum[i][j][k] + " ");

}

System.out.println();

}

System.out.println();

}

}

}

شرح الطريقة:

1. التحقق من الأبعاد: نتأكد من أن كلا المصفوفتين لهما نفس الأبعاد حتى يمكن جمعهما.
2. إنشاء مصفوفة النتيجة: ننشئ مصفوفة ثلاثية الأبعاد جديدة بنفس الأبعاد لحفظ نتيجة الجمع.
3. الجمع المكون تلو الآخر: نقوم بالتنقل عبر جميع العناصر في المصفوفتين باستخدام ثلاث حلقات متداخلة، ونجمع العناصر المقابلة في كل موضع ونحفظ النتيجة في مصفوفة النتيجة.
4. إرجاع النتيجة: نعيد مصفوفة النتيجة التي تحتوي على مجموع المصفوفتين الأصليتين.