# Упражнения: Рекурсия и обратно връщане

Тествайте решението в Judge: <https://judge.softuni.bg/Contests/2726/Recursive-Algorithms-and-Backtracking>.

# Част I – Генериране на вектори от нули и единици

Генерирайте всички **n**-битови вектори от нули и единици в **лексикографски** ред.

### Примери

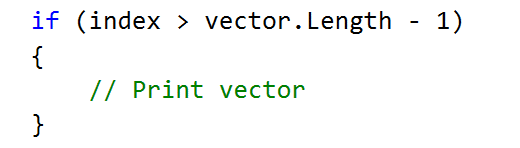
|  |  |
| --- | --- |
| **Вход** | **Изход** |
| 3 | 000  001  010  011  100  101  110  111 |
| 5 | 00000  00001  00010  …  11110  11111 |

### Насоки

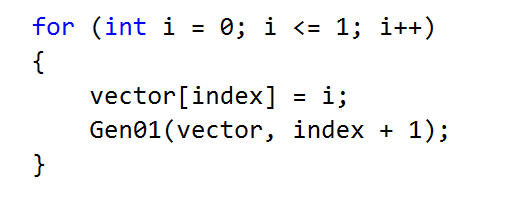
1. Методът трябва да има **масив** и **индекс** като параметри. Масивът ще e **вектор**.



1. **Дъното** на рекурсията, е когато индексът е по-голям от дължината на вектора



1. За да генерираме всички комбинации, създаваме for-цикъл с **рекурсивно извикване**:



# Част II – Пъзелът 8 кралици

В упражнението ще трябва да имплементирате рекурсивния алгоритъм за решаване на **пъзелът 8 кралици**. Нашата цел е да напишем програма, която намира **всички възможни начини** как да поставим **8 кралици** на шахматна дъска, така че две кралици да не могат да се атакуват (на диагонал, ред и колона).

## Примери

|  |  |
| --- | --- |
| **Вход** | **Изход** |
| *(няма вход)* | \* - - - - - - -  - - - - \* - - -  - - - - - - - \*  - - - - - \* - -  - - \* - - - - -  - - - - - - \* -  - \* - - - - - -  - - - \* - - - -  \* - - - - - - -  - - - - - \* - -  - - - - - - - \*  - - \* - - - - -  - - - - - - \* -  - - - \* - - - -  - \* - - - - - -  - - - - \* - - -  …  *(+ още 90 решения)* |

## За пъзелът "8 кралици"

Можете да научите за пъзелът "8 кралици" от Wikipedia: <http://en.wikipedia.org/wiki/Eight_queens_puzzle>.

## Дефиниране на структурата, която да помни шахматната дъска

Нека да направим структура от данни, която съдържа **шахматната дъска.** Тя се състои от **8 \* 8** клетки, като всяка една може да бъде **празна** или **запълнена с кралица**. Първо трябва да дефинираме **размера на шахматната дъска** като константа:



## Дефиниране на структурата, която да помни атакуващите позиции

Трябва да запазваме позициите на кралиците в структура от данни. Във всеки момент по време на изпълнение на програмата, трябва да знаем дали дадена позиция **{ред, колона}** е под атаката на кралица или не.

Има много начини на съхраним **атакуващите позиции**:

* Като съхраним мястото на всички кралици, които стоят до момента, и да проверим дали може да поставим на нова.
* Като използваме матрица **int[,]**, за да запазим всички **атакуващи позиции** и да проверим дали може да поставим нова. Това е ще бъде сложно за поддържане, защото матрицата трябва да **променя** много места, когато трябва да се **постави**/**премахне** кралица.
* Като използваме сет от **атакуващи редове, колони и диагонали**. Нека да проверим тази идея:



Горните определения имат следните изисквания:

* Имаме **8 реда**, които са номерирани от **0** до **7**.
* Имаме **8 колони**, които са номерирани от **0** до **7**.
* Имаме **15 леви** **диагонала**, които са номерирани от **-7** до **7**. Може да използваме следната формула, за да пресметнем левия диагонал по ред и колона: **ляв диагонал = колона – ред**.
* Имаме **15 десни диагонала**, които са номерирани от **0** до **14**, чрез формулата: **десен диагонал = колона + ред**.

Нека използваме за **пример** следната шахматна дъска с 8 царици на следните позиции:

* {0, 0}; {1, 6}; {2, 4}; {3, 7}; {4, 1}; {5, 3}; {6, 5}; {7, 2}



Следвайки нашия пример може да видим, че кралица {4, 1} окупира **ред 4**, **колона 1**, **ляв диагонал -1** и **десен диагонал 5**.

## Написване на алгоритъма с обратно връщане

Сега трябва да напишем нашия **алгоритъм с обратно връщане** за поставянето на 8 кралици.

Алгоритъмът започва от ред 0 и се опитва да постави кралица на някаква колона на ред 0. След това се опитва да постави кралица на ред 1, а следващата на кралица на ред 2 и т.н. до последния ред. Кодът за генериране на позицията на следващата кралица трябва да изглежда така:



Програмата първоначално извиква метода от ред 0:



## Проверяване на свободна позиция

Нека да напишем **код за проверка дали има свободно място на определена позиция**. Една позиция е свободна, когато не е под атака от другите кралица. Това означава, че ако има някои редове, колони или диагонали, които са окупирани от друга кралица, позицията не е свободна. Кодът трябва да изглежда така:



Припомнете си, че за да пресметнем левия диагонал, използваме **колона – ред** , а за десния диагонал използваме **ред + колона**.

## Маркиране/Освобождаване на атакуваща позиция

След като кралицата е **поставена**, трябва да **маркираме** **редовете**, **колоните** и **диагоналите**, които могат да бъдат **атакувани**:



Когато премахнем **кралица**, трябва да направим метод за **освобождаване на всички редове**, **колони** или **диагонали**, които са били **атакувани**. Напишете го сами:



## Отпечатване на решенията

Когато намерим решение, трябва да го отпечатаме на конзолата. Първо трябва да намерим броя на верните решения:



След това отпечатваме всички редове и колони и отпечатваме **шахматните полета:**



## Тестване на кода

Пъзелът “8 кралици” има **92 решения**. Проверете дали вашият код генерира и отпечатва всички правилно. Използвайте solutionsFound, за да проверите дали имате толкова решение. По-долу са решенията:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| http://www.datagenetics.com/blog/august42012/s01.png | http://www.datagenetics.com/blog/august42012/s02.png | http://www.datagenetics.com/blog/august42012/s03.png | http://www.datagenetics.com/blog/august42012/s04.png | http://www.datagenetics.com/blog/august42012/s05.png | http://www.datagenetics.com/blog/august42012/s06.png | http://www.datagenetics.com/blog/august42012/s07.png | http://www.datagenetics.com/blog/august42012/s08.png |
| http://www.datagenetics.com/blog/august42012/s09.png | http://www.datagenetics.com/blog/august42012/s10.png | http://www.datagenetics.com/blog/august42012/s11.png | http://www.datagenetics.com/blog/august42012/s12.png | http://www.datagenetics.com/blog/august42012/s13.png | http://www.datagenetics.com/blog/august42012/s14.png | http://www.datagenetics.com/blog/august42012/s15.png | http://www.datagenetics.com/blog/august42012/s16.png |
| http://www.datagenetics.com/blog/august42012/s17.png | http://www.datagenetics.com/blog/august42012/s18.png | http://www.datagenetics.com/blog/august42012/s19.png | http://www.datagenetics.com/blog/august42012/s20.png | http://www.datagenetics.com/blog/august42012/s21.png | http://www.datagenetics.com/blog/august42012/s22.png | http://www.datagenetics.com/blog/august42012/s23.png | http://www.datagenetics.com/blog/august42012/s24.png |
| http://www.datagenetics.com/blog/august42012/s25.png | http://www.datagenetics.com/blog/august42012/s26.png | http://www.datagenetics.com/blog/august42012/s27.png | http://www.datagenetics.com/blog/august42012/s28.png | http://www.datagenetics.com/blog/august42012/s29.png | http://www.datagenetics.com/blog/august42012/s30.png | http://www.datagenetics.com/blog/august42012/s31.png | http://www.datagenetics.com/blog/august42012/s32.png |
| http://www.datagenetics.com/blog/august42012/s33.png | http://www.datagenetics.com/blog/august42012/s34.png | http://www.datagenetics.com/blog/august42012/s35.png | http://www.datagenetics.com/blog/august42012/s36.png | http://www.datagenetics.com/blog/august42012/s37.png | http://www.datagenetics.com/blog/august42012/s38.png | http://www.datagenetics.com/blog/august42012/s39.png | http://www.datagenetics.com/blog/august42012/s40.png |
| http://www.datagenetics.com/blog/august42012/s41.png | http://www.datagenetics.com/blog/august42012/s42.png | http://www.datagenetics.com/blog/august42012/s43.png | http://www.datagenetics.com/blog/august42012/s44.png | http://www.datagenetics.com/blog/august42012/s45.png | http://www.datagenetics.com/blog/august42012/s46.png | http://www.datagenetics.com/blog/august42012/s47.png | http://www.datagenetics.com/blog/august42012/s48.png |
| http://www.datagenetics.com/blog/august42012/s49.png | http://www.datagenetics.com/blog/august42012/s50.png | http://www.datagenetics.com/blog/august42012/s51.png | http://www.datagenetics.com/blog/august42012/s52.png | http://www.datagenetics.com/blog/august42012/s53.png | http://www.datagenetics.com/blog/august42012/s54.png | http://www.datagenetics.com/blog/august42012/s55.png | http://www.datagenetics.com/blog/august42012/s56.png |
| http://www.datagenetics.com/blog/august42012/s57.png | http://www.datagenetics.com/blog/august42012/s58.png | http://www.datagenetics.com/blog/august42012/s59.png | http://www.datagenetics.com/blog/august42012/s60.png | http://www.datagenetics.com/blog/august42012/s61.png | http://www.datagenetics.com/blog/august42012/s62.png | http://www.datagenetics.com/blog/august42012/s63.png | http://www.datagenetics.com/blog/august42012/s64.png |
| http://www.datagenetics.com/blog/august42012/s65.png | http://www.datagenetics.com/blog/august42012/s66.png | http://www.datagenetics.com/blog/august42012/s67.png | http://www.datagenetics.com/blog/august42012/s68.png | http://www.datagenetics.com/blog/august42012/s69.png | http://www.datagenetics.com/blog/august42012/s70.png | http://www.datagenetics.com/blog/august42012/s71.png | http://www.datagenetics.com/blog/august42012/s72.png |
| http://www.datagenetics.com/blog/august42012/s73.png | http://www.datagenetics.com/blog/august42012/s74.png | http://www.datagenetics.com/blog/august42012/s75.png | http://www.datagenetics.com/blog/august42012/s76.png | http://www.datagenetics.com/blog/august42012/s77.png | http://www.datagenetics.com/blog/august42012/s78.png | http://www.datagenetics.com/blog/august42012/s79.png | http://www.datagenetics.com/blog/august42012/s80.png |
| http://www.datagenetics.com/blog/august42012/s81.png | http://www.datagenetics.com/blog/august42012/s82.png | http://www.datagenetics.com/blog/august42012/s83.png | http://www.datagenetics.com/blog/august42012/s84.png | http://www.datagenetics.com/blog/august42012/s85.png | http://www.datagenetics.com/blog/august42012/s86.png | http://www.datagenetics.com/blog/august42012/s87.png | http://www.datagenetics.com/blog/august42012/s88.png |
| http://www.datagenetics.com/blog/august42012/s89.png | http://www.datagenetics.com/blog/august42012/s90.png | http://www.datagenetics.com/blog/august42012/s91.png | http://www.datagenetics.com/blog/august42012/s92.png |  |  |  |  |

Изпратете кода си в Judge, като отпечатате всичките 92 решения, разделени с един ред.

## Оптимизиране на решението

Сега трябва да оптимизираме нашият код:

* Премахваме сета attackedRows. Не е нужен, защото всички царици ще бъдат поставени на всеки ред.
* Използваме масив bool[] за attackedColumns, attackedLeftDiagonals и attackedRightDiagonals вместо сетове. Забележете, че масивите са индексирани от 0 до техния размер. Не могат да бъдат отрицателни числа.

## \* Решение, базирано на пермутация

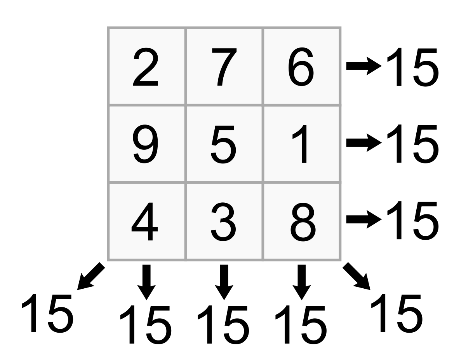
Изпробвайте да имплементирате **по-ефективно** решение, **базирано на пермутация**, за пъзелът "8 кралици". Погледнете този код, за да разберете идеята:

**TODO: LOOK FOR C# LINK**

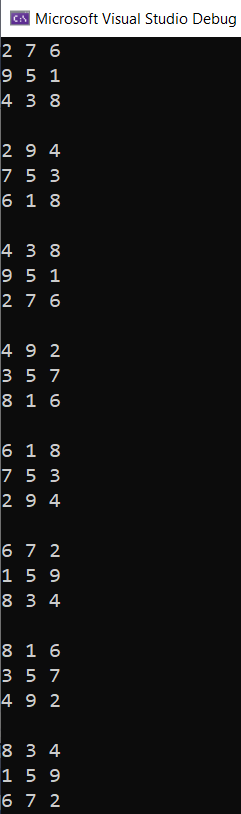
<http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html>.

# Част III – \* Магически квадрат

Вашата последна задача е да използвате рекурсия, за да отпечатате всички **3X3 магически квадрати** с числа от **1** до **9** включително (**без повторения**). **Магическият квадрат** е специален квадрат, който има **еднаква сума** от числата на всеки **ред**, **колона** и **основния диагонал**. За повече яснота, погледнете отдолу:



Трябва на вашата конзола да се отпечата:



Ако искате, можете да намерите **магически квадрат** с числата от **0** до **9** включително. Трябва да имате общо **16**.

## Насоки

За по-голяма простота можете да запазите квадратите като масив от цели числа. Масивът трябва да има размер 9 за всички клетки в квадрата 3X3:



Имайте предвид, че трябва да следите и числата, които вече са били използвани в текущия квадрат, тъй като те не трябва да се повтарят. Може да създадете масив, където числата имат стойност 0 (където числото не е използвано) или 1 (където числото е използвано). Всеки път, когато „отидете на“ рекурсивно повикване и „се върнете“ от рекурсивно повикване, трябва да актуализирате масива. Масивът трябва да има размер 9, за да пази всички числа от 1 до 9:



След това създайте метод за пермутация на числата. Можете да използвате инструкции от сайта, за да напишете рекурсивния алгоритъм: <https://erwnerve.tripod.com/prog/recursion/magic.htm> .

Напишете метод TestMagic(), за да проверите дали сегашният квадрат е магически. Добра идея е да генерирате магически квадрат от квадратен масив, така че лесно да можете проверите дали сумите на диагоналите, редовете и колоните са еднакви.

Ако матрицата е магически квадрат, отпечатайте неговите числа като 3X3 квадрат. Числата трябва да бъдат разделени с интервал и магическия квадрат трябва да бъде разделен с ред.

За да проверите дали матрица е магически квадрат или не, може да използвате следният метод:

|  |
| --- |
| static bool isMagicSquare(int[,] matrix)  {  int N = 3;  // пресмятане на сумата  // от оснивния диагонал  int sum = 0, sum2 = 0;  for (int i = 0; i < N; i++)  sum = sum + matrix[i, i];  // втория диагонал  for (int i = 0; i < N; i++)  sum2 = sum2 + matrix[i, N - 1 - i];  if (sum != sum2)  return false;  // за сумата на редовете  for (int i = 0; i < N; i++)  {  int rowSum = 0;  for (int j = 0; j < N; j++)  rowSum += matrix[i, j];  // проверяваме дали сумата на всеки ред  // е еднаква с основния диагонал  if (rowSum != sum)  return false;  }  // за сумата на колоните  for (int i = 0; i < N; i++)  {  int colSum = 0;  for (int j = 0; j < N; j++)  colSum += matrix[j, i];  // проверяваме дали сумата на всяка колона  // е еднаква с основния диагонал  if (sum != colSum)  return false;  }  return true;  } |