

Combinatorial Search Example

Principles of Functional Programming

Sets

Sets are another basic abstraction in the Scala collections.

A set is written analogously to a sequence:

```
val fruit = Set("apple", "banana", "pear")  
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets:

```
s.map(_ + 2)  
fruit.filter(_.startsWith("app"))  
s.nonEmpty
```

(see Scaladoc for `scala.Set` for a list of all supported operations)

Sets vs Sequences

The principal differences between sets and sequences are:

1. Sets are unordered; the elements of a set do not have a predefined order in which they appear in the set
2. sets do not have duplicate elements:

```
s.map(_ / 2)           // Set(2, 0, 3, 1)
```

3. The fundamental operation on sets is contains:

```
s.contains(5)          // true
```

Example: N-Queens

The eight queens problem is to place eight queens on a chessboard so that no queen is threatened by another.

- ▶ In other words, there can't be two queens in the same row, column, or diagonal.

We now develop a solution for a chessboard of any size, not just 8.

One way to solve the problem is to place a queen on each row.

Once we have placed $k - 1$ queens, one must place the k th queen in a column where it's not "in check" with any other queen on the board.

Algorithm

We can solve this problem with a recursive algorithm:

- ▶ Suppose that we have already generated all the solutions consisting of placing $k-1$ queens on a board of size n .
- ▶ Each solution is represented by a list (of length $k-1$) containing the numbers of columns (between 0 and $n-1$).
- ▶ The column number of the queen in the $k-1$ th row comes first in the list, followed by the column number of the queen in row $k-2$, etc.
- ▶ The solution set is thus represented as a set of lists, with one element for each solution.
- ▶ Now, to place the k th queen, we generate all possible extensions of each solution preceded by a new queen:

Implementation

```
def queens(n: Int) =  
  def placeQueens(k: Int): Set[List[Int]] =  
    if k == 0 then Set(List())  
    else  
      for  
        queens <- placeQueens(k - 1)  
        col <- 0 until n  
        if isSafe(col, queens)  
      yield col :: queens  
  placeQueens(n)
```

Exercise

Write a function

```
def isSafe(col: Int, queens: List[Int]): Boolean
```

which tests if a queen placed in an indicated column `col` is secure amongst the other placed queens.

It is assumed that the new queen is placed in the next available row after the other placed queens (in other words: in row `queens.length`).

Exercise

```
def isSafe(col: Int, queens: List[Int]): Boolean =  
  !checks(col, 1, queens)
```

where the checks predicate takes in an additional second parameter delta the distance in rows between the first row of queens and the row where the current queen is placed. checks is defined as follows:

```
def checks(col: Int, delta: Int, queens: List[Int]): Boolean = queens match  
  case qcol :: others =>  
    qcol == col                // vertical check  
    || (qcol - col).abs == delta // diagonal check  
    || checks(col, delta + 1, others)  
  case Nil =>  
    false
```