Backtracking II: Examples

For a first example we give a backtracking algorithm that finds all permutations of the numbers 0, 1, 2, ..., n-1.   We will add numbers to the permuation one at a time. We a very simple feasibility test: we can extend a partial solution *sofar* with the number *curr* if *curr* is not already in *sofar.* A partial solution is a complete solution if it has length n. To find all permuations we use our pattern for all backtracking solutions:

```
(define allsols (lambda (n)
        (letrec ([backtrack (lambda (curr sofar)
                ; backtrack returns all solutions that extend sofar with value curr or  higher
                  (cond
                        [<sofar is a complete solution> (list sofar) ]
                        [<curr is out of the range of possible values for this step> null]
                        [(feasible  curr sofar)
                           (let ([res (backtrack n <first value for next step> (cons curr sofar))]
                                 [res2 (backtrack n  <value after curr> sofar)])
                                      (append res1 res2)
                        [else (backtrack n <value after curr> sofar)])])])
                (backtrack <first value> null)))
```

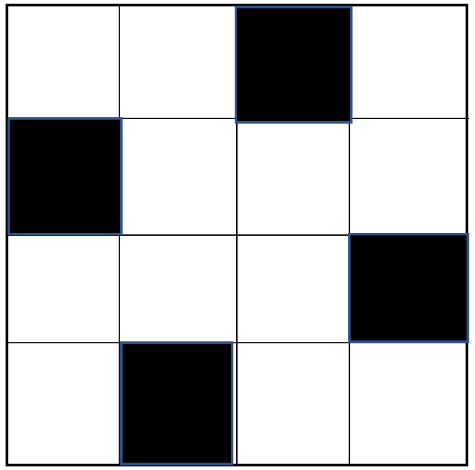
This results in the following procedure:

```
(define allperms (lambda (n)
      (letrec ([back (lambda (curr sofar)
                  ; back returns a list of all permuations with prefix
                  ; sofar and next element curr or higher
             (cond
                  [(= n (length sofar)) (list sofar)]
                  [(= n curr) null]
                  [(not (member curr sofar))
                          (let ([t1 (back 0 (cons curr sofar))]
                             [t2 (back (+ 1 curr) sofar)])
                                (append t1 t2))]
                  [else (back (+ 1 curr) sofar)]))])
            (back 0 null))))
```

For example, (allperms 3) returns

```
'((2 1 0) (1 2 0) (2 0 1) (0 2 1) (1 0 2) (0 1 2))
```

For a next example we solve the n-Queens problem. Remember that this asks us to find n squares on an nxn grid so that no two are in the same row, column or diagonal. For example, for a 4x4 grid one solution is



We will represent solutions as lists of the (row col) selected squares. For example, the solution above is '((2 3) (0 2) (3 1) (1 0))

We need feasibility tests to determine if we can extend a partial solution with the new square (*row col*). To ensure that *x* is a feasible row, we need to determine if *x* is the first element of any square already in the partial solution:

```
(define rows (lambda (x sofar)
            ; returns #t if x is a row in sofar
            (member x  (map car sofar))))
```

We are going to find a square for the columns one at a time, so there will be no possibility of putting two squares in the same column. Note that if we have partial solution *sofar*, the next square we choose is for column (length sofar).

It is easier than you might think to check the diagonals. Consider the following grid with the location of each square indicated:

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Take any of the diagonals that rise as we read it from left to right. For example, one such diagonal has entries (2,0), (1,1), and (0,2). Each (row, col) entry in such a diagonal sums to the same value.

Once you have seen that, it is easy to spot the invariant on the downward sloping diagonals:

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Instead of the sum, it is the *difference* between the row and column that is the same on each element of these diagonals. For example, one downward diagonal has entries (0,1), (1,2), and (2,3). The row-col difference for each of these entries is -1.

This leads to functions for checking whether *x* is a feasible row for the next step of our solution:

```
(define updiags (lambda (x sofar)
    ;returns #t if (x is on the upward diagonal of a pair in sofar
              (let ([row x]
                     [col (length sofar)])
                (member (+ row col) (map (lambda (p) (apply + p)) sofar)))))
```

Of course, there is a similar function for the downward diagonals.

We can put all of the feasibility checks into one procedure:

```
(define ok (lambda (x sofar)
        (not (or (rows x sofar) (updiags x sofar) (downdiags x sofar)))))
```

After all of this, the backtrack solution to the n-Queens problem is a simple application of the backtracking pattern. Remember the pattern:

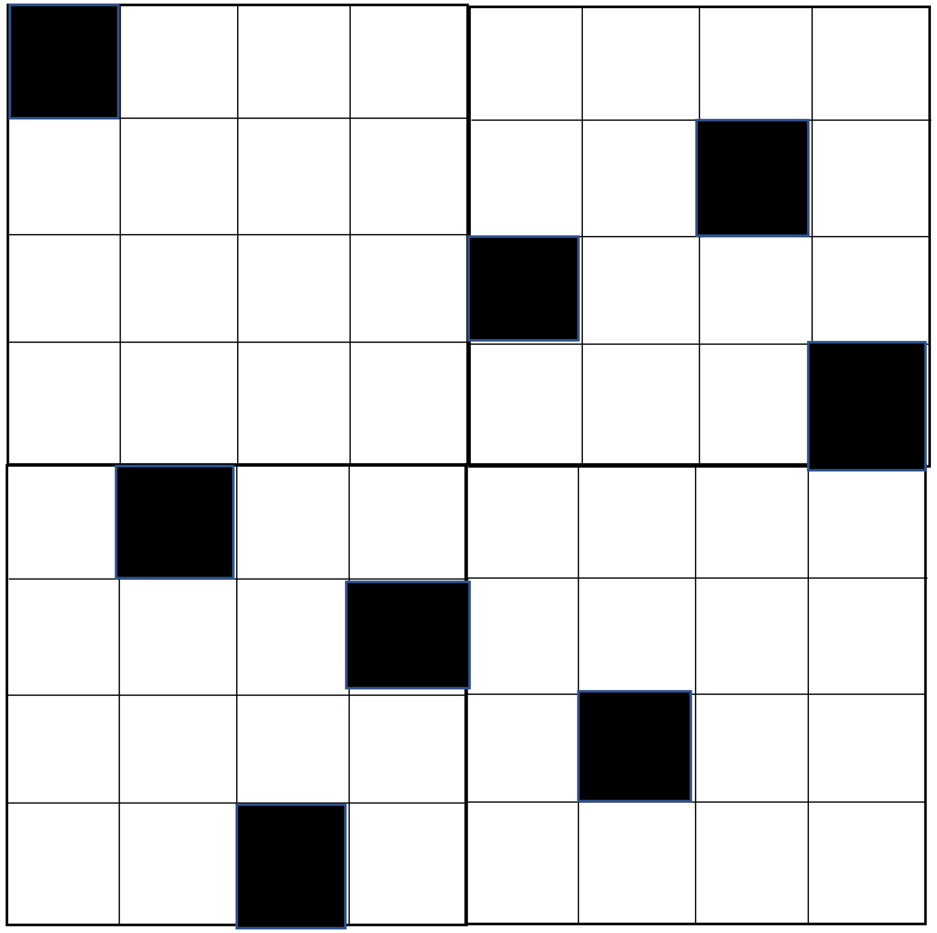
```
(define backtrack (lambda (n curr sofar)
        ; returns the first extension of sofar into a solution with
        ; curr or higher as the value for the current step
        (cond
                [<sofar is a complete solution> sofar]
                [<curr is out of the range of possible values for this step> null]
                [(feasible  curr sofar)
                        (let ([res (backtrack n <first value for next step> (cons curr sofar))])
                                (if (null? res)
                                        (backtrack n  <value after curr> sofar)
                                        res))
                [else (backtrack n <value after curr> sofar)])))
```

This becomes:

```
(define queens (lambda (n curr sofar)
              ; returns the first solution it finds with sofar as
              ; its tail, curr or higher as  the next entry.
              (cond
                [(= n (length sofar)) sofar]
                [(= n curr) null]
                [(ok curr sofar)
                          (let ([res (queens n 0 (cons (list curr (length sofar)) sofar))])
                                       (if (null? res)
                                                 (queens n (+ 1 curr) sofar)
                                                 res))]
               [else (queens n (+ 1 curr) sofar)])))
```

For example, (queens 8 0 null) returns

'((3 7) (1 6) (6 5) (2 4) (5 3) (7 2) (4 1) (0 0))



We can just as easily find all solutions:

```
(define allqueens (lambda (n)
                (letrec ([back (lambda (curr sofar)
              ; returns all solutions it finds with sofar as
              ; its tail, curr or higher as  the next entry.
              (cond
                [(= n (length sofar)) (list sofar)]
                [(= n curr) null]
                [(ok curr sofar)
                   (let ([t1 (back 0 (cons (list curr (length sofar)) sofar))]
                         [t2 (back (+ 1 curr) sofar)])
                             (append t1 t2))]
                 [else (back (+ 1 curr) sofar)]))])
                 (back 0 null))))
```

(allqueens 8) finds 92 solutions, though this includes many redundancies such as rotations and reflections of other solutions.