# Backtracking

Notes 2026

## Contents



# 1 Backtracking Notes

## 1.1 Introduction

- These notes are sketches on Backtracking for a sequence of notes on exhaustive searching
- The notes draw on:
- M269 25J Book Chp 22 *Backtracking*
- Bob Moore Backtracking tutorial slides and Python code
- Bird and Wadler (1988, page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1) *Implicit search and the n-queens problem*

  Both of the *Bird* references have some discussion of efficiency and improving performance — you should be aware the the former constructs partial solutions column (File) by column while the later constructs partial solution row (Rank) by row

ToC

## 1.2 Exhaustive Search and Backtracking

- **Exhaustive** look at every option which fits the basic criteria
- **Greedy** take locally optimum choice at every stage

  Hardly ever works — requires a proof that this approach works for this particular case

  Proofs not often included in a course — unfortunate

  When greedy *does* work, it is more efficient that other methods

- **Backtracking** check if a partial solution can be extended towards a full solution

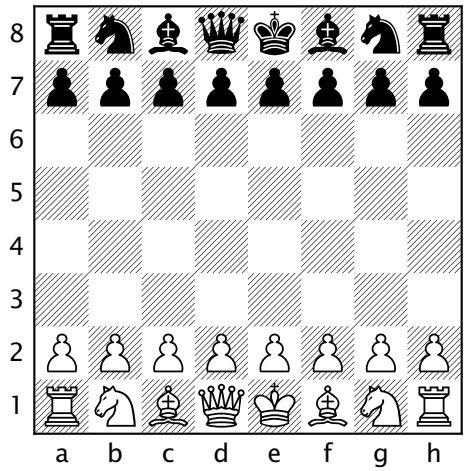
  If not, go back and make a different choice

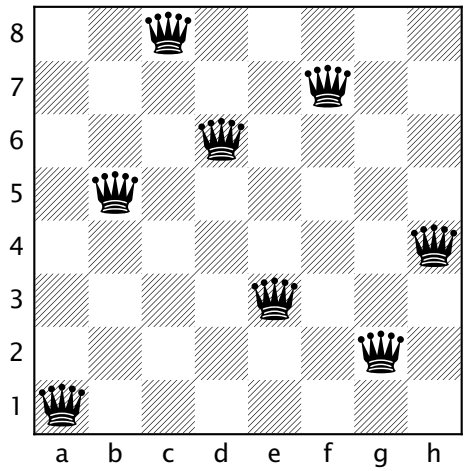
  The *backtracking* may be done implicitly via recursion
- **Problem** choice of representation and initial setup — define *partial solutions*
- Define global constraints that must be satisfied by any solution (but perhaps not by partial solutions)
- Define local constraints that must be satisfied by partial solutions and determines if a partial solution can be extended
- Note that backtracking relies on local contraints — if there are no local constraints then you are on an exhaustive search
- **extend** — checks for solution
- **satisfiesGlobal** checks if solution satisfies global constraints
- **canExtend** checks if can extend partial solution

ToC

## 1.3 8-Queens Problem

- The problem of placing eight queens on an 8x8 chessboard so that no two queens threaten each other.
- Problem posed by Max Bezzel, chess composer, in 1848
- First solution by Franz Nauck in 1850
- Carl Friedrich Gauss also worked on the 8-queens problem and the generalised *n*-queens problem
- In 1874 S Günther proposed a solution method using determinants. J W L Glaisher refined the approach
- Edsger Dijkstra 1972 used 8 queens to illustrate structured programming and backtracking
- Each column (or row) must contain exactly one queen. So a strategy for finding a solution is as follows.
- Place a queen in in the first column in any position.
- Then place a queen in the second column in any position not in check from the first queen.
- Continue until all eight queens hav been placed.
- If at any point this is not possible (because all positions are in check) then *backtrack* and reapply the method to find a different position for the queens in the first *m* columns and try again




- **Board representation** list of ranks (rows)
- The board above is `[1,5,8,6,3,7,2,4]`
- **Health warning** some implementations represent a board as list of files (columns) and may count from top to bottom
- These notes are based on:
- Bird (1998, sec 6.5.1 page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1 page 369) uses ranks (rows) not files (columns)
- Bob Moore's M269 Backtracking tutorial notes 2026
- M269 notes chp 22
- Knuth (2023, sec 7.2.2 page 30) *The Art of Computer Programming: The Combinatorial Aldorithms Volume 4B*
- We first declare some type aliases for position and chessboard

```
6 type Rank  = int    # Rows
7 type File  = int    # Columns
8 type Board = [Rank] # Example
```

- The function `isSafe(brd,n)` tests whether the partial solution `brd` can be extended by one column with a queen in Rank *n*

- `isSafe(brd,n)` uses the function `inCheck(posn1,posn2)` to test if two queens at `posn1` and `posn2` hold each other in check

```
def inCheck(posn1: (File,Rank), posn2: (File,Rank)) -> bool :
  """ Return True if Queen in position posn1 == (i,j)
  can be threatened by Queen in posn2 == (m.n)
  Precondition: not (i == m)
  since we are placing queens File (column) by File
  """
  (i,j) = (posn1[0],posn1[1])
  (m,n) = (posn2[0],posn2[1])
  return ((j == n)
          or (i + j == m + n)
          or (i - j == m - n))

def isSafe(brd: Board, n: Rank) -> bool :
  nxtFl = len(brd) + 1
  lstOfCks = [not (inCheck((i,j), (nxtFl,n)))
              for (i,j) in list(zip(range(1,len(brd)+1), brd))
              ]
  return all(lstOfCks)
```

- The function `queens` takes a board size `brdSize` and recursively finds all partial solutions of size `m`
- `queeens8` and `queens4` specialise `queens` to boards of size 8 and 4

```
def queens(brdSize: Rank,m: File) -> [Board] :
  if m == 0 :
    return [[]]
  else :
    lstBrd = [brd + [n]
              for brd in queens(brdSize, m-1)
              for n in range(1,brdSize + 1)
              if isSafe(brd,n)]
    return lstBrd


def queens8(m: File) -> [Board] :
  return queens(8, m)

def queens4(m: File) -> [Board] :
  return queens(4, m)
```

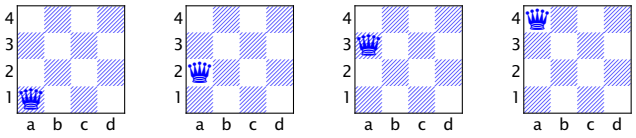
ToC

## 1.4 4-Queens Example

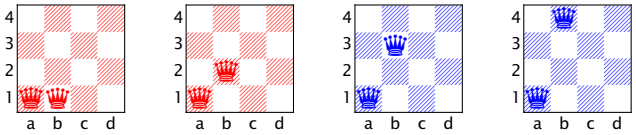
- Knuth (2023, sec 7.2.2 page 31) *Backtrack Programming* suggests that one of the best ways to learn about backtracking is to execute the algorithm by hand in the special case of `queens(4,4)`
- This is done with the chessboard illustrated and colour coded
  - Blue for partial solutions
  - Red for not feasible
  - Green for solutions
- The initial blank board is not shown
- Following the boards is a different visualisation of the the *backtracking tree* with some further results for boards of different sizes
- Working column by column, fill the next column where the position is safe

```
Python3>>> queens4(0)
[[]]                                  by line 33
Python3>>> queens4(1)
[[1], [2], [3], [4]]                  by line 35
Python3>>> queens4(2)                  by line 35
[[1, 3], [1, 4], [2, 4], [3, 1], [4, 1], [4, 2]]
Python3>>> queens4(3)                  by line 35
[[1, 4, 2], [2, 4, 1], [3, 1, 4], [4, 1, 3]]
Python3>>> queens4(4)
[[2, 4, 1, 3], [3, 1, 4, 2]]          by line 35
```

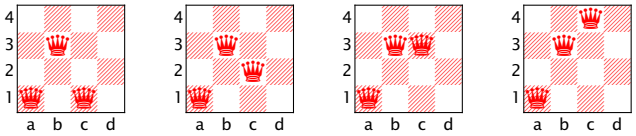
- Level 1



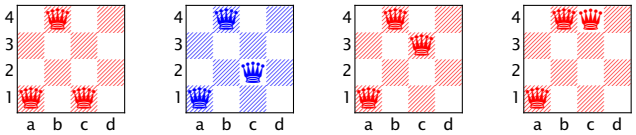
- Level 2 Block 1



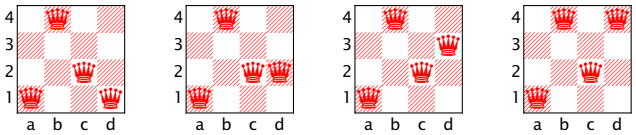
- Level 3 Block 1



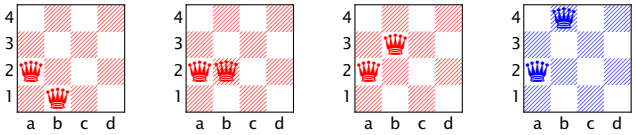
- Level 3 Block 2



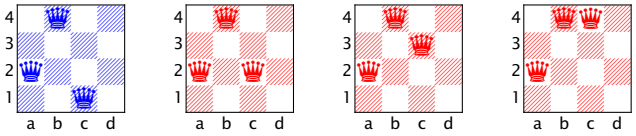
- Level 4 Block 1



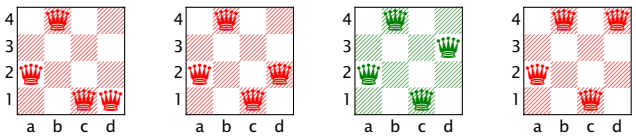
- Level 2 Block 2



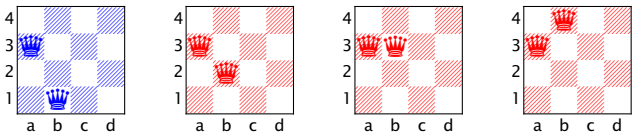
- Level 3 Block 3



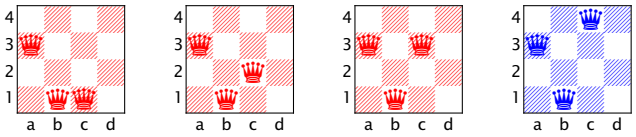
- Level 4 Block 2



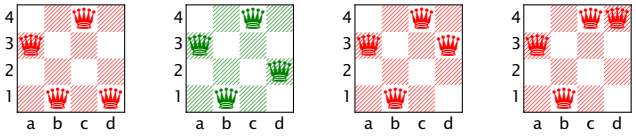
- Level 2 Block 3



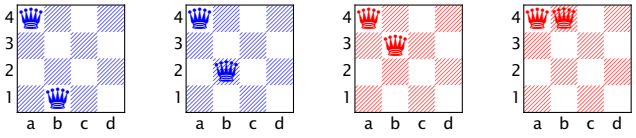
- Level 3 Block 4



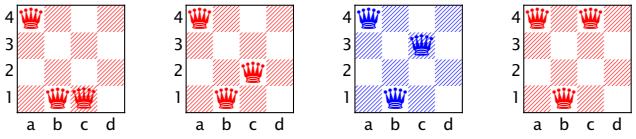
- Level 4 Block 3



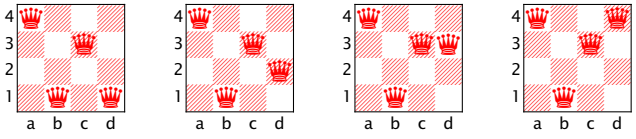
- Level 2 Block 4



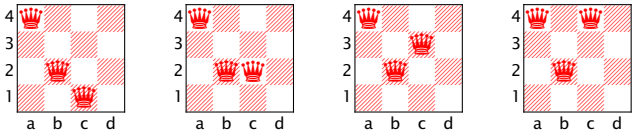
- Level 3 Block 5



- Level 4 Block 4



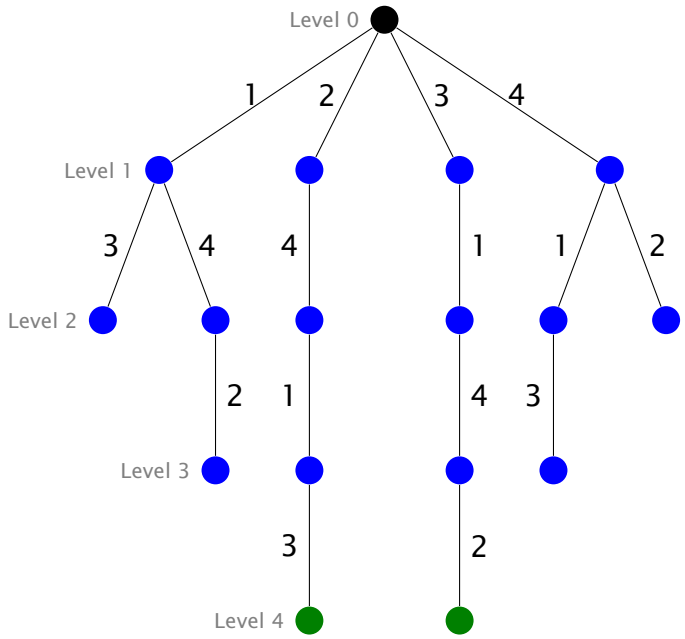
- Level 3 Block 6



- The above chessboards *can* be used to form a tree of queen placement on the board
- This can also be represented as a *Backtrack tree* — see below

ToC

## 1.5 Backtrack Trees and Profiles




- The *profile* $(p_0, p_1, \ldots, p_n)$ of a backtrack tree is the number of nodes at each level

```
48 def backtrackProfile(brdSize: Rank) -> list[int] :
49   return [len(queens(brdSize,m)) for m in range(brdSize+1)]
```

```
Python3>>> backtrackProfile(4)
[1, 4, 6, 4, 2]
Python3>>> sum(backtrackProfile(4))
17
Python3>>> backtrackProfile(8)
[1, 8, 42, 140, 344, 568, 550, 312, 92]
Python3>>> sum(backtrackProfile(8))
2057
```

- **queens4** Solutions 2
  - Backtrack nodes 17
  - Number of possible sequences $4^4 = 256$
  - Arbitrary placings $\binom{16}{4} = 1820$
- **queens8** Solutions 92
  - Backtrack nodes 2057
  - Number of possible sequences $8^8 = 16{,}777{,}216$
  - Arbitrary placings $\binom{64}{8} = 4{,}426{,}165{,}368$

ToC

## 1.6 *n*-Queens Solutions

| #-Qns | # Solns | #-Qns | # Solns |
|---|---|---|---|
| 0 | 1 | 14 | 365,596 |
| 1 | 1 | 15 | 2,279,184 |
| 2 | 0 | 16 | 14,772,512 |
| 3 | 0 | 17 | 95,815,104 |
| 4 | 2 | 18 | 666,090,624 |
| 5 | 10 | 19 | 4,968,057,848 |
| 6 | 4 | 20 | 39,029,188,884 |
| 7 | 40 | 21 | 314,666,222,712 |
| 8 | 92 | 22 | 2,691,008,701,644 |
| 9 | 352 | 23 | 24,233,937,684,440 |
| 10 | 724 | 24 | 227,514,171,973,736 |
| 11 | 2680 | 25 | 2,207,893,435,808,352 |
| 12 | 14,200 | 26 | 22,317,699,616,364,044 |
| 13 | 73,712 | 27 | 234,907,967,154,122,528 |

- **Source** OEIS: A000170
- Knuth (2023, page34) gives some anecdotes about the *n*-Queens calculations
- *Q*(11) and *Q*(12) were the last to be calculated by hand, the later in 1910
- Q(13) was calculated using a computer in 1960
- The calculation of *Q*(14) was calculated in 1963 taking 120 hours on an IBM 1620
- *Q*(15) took two hours in 1974 on an IBM 360-75
- *Q*(27) was calculated in 2016 taking 383 days using a cluster of FPGA devices — it will probably be some time before this is exceeded
- Here are the timings and memory usage for my home computer — an Apple M1 Max from 2022 with 32 GB memory
- The code is a Haskell translation of Python listed earlier

```
GHCi> :set +s
GHCi> backtrackProfileSum(11)
(166926,[1,11,90,536,2468,8492,21362,37248,44148,34774,15116,2680])
(23.28 secs, 27,516,106,792 bytes)
GHCi> backtrackProfileSum(12)
(856189,[1,12,110,756,4080,16852,52856,120104
        ,195270,222720,160964,68264,14200])
(148.38 secs, 174,226,075,008 bytes)
```

```
36  backtrackProfile :: Rank -> [Int]
37  backtrackProfile brdSize
38    = [length (queens brdSize m) |
39       m <- [0..brdSize]]

41  backtrackProfileSum :: Rank -> (Int,[Int])
42  backtrackProfileSum brdSize
43    = (profileSum,profileLst)
44      where
45        profileLst = backtrackProfile brdSize
46        profileSum = sum profileLst
```

- Here are further timings and memory usage for my home computer
- This is for sizes 13 and 14

- Notice we are heading towards some serious waiting even with not very large inputs

```
GHCi> :set +s
GHCi> backtrackProfileSum(13)
(4674890,[1,13,132,1030,6404,31100,117694
         ,335010,707698,1086568,1151778,813448,350302,73712])
(998.87 secs, 1,152,159,900,192 bytes)
GHCi> backtrackProfileSum(14)
(27358553,[1,14,156,1364,9632,54068,241484,835056
          ,2211868,4391988,6323032,6471872,4511922,1940500,365596])
(7317.33 secs, 8,046,531,055,416 bytes)
```

ToC

### 1.7 References

- Bird and Wadler (1988, sec 6.5.1 page 161) *Introduction to Functional Programming*
- Bird and Gibbons (2020, sec 15.1 page 369) *Algorithm Design with Haskell*
- Knuth (2023, sec 7.2.2 page 31) *Backtrack Programming*
- Bell and Stevens (2009) *A survey of known results and research areas for n-queens*
- Rosetta Code: N-queens problem
- Chess.com: Eight Queens Puzzle
- Medium: Eight Queens Problem (members only)
- OEIS: All solutions to the problem of eight queens
- OEIS: n Queens
- John D Cook: Special solutions to the eight queens problem
- Using Symmetry to Optimize an N-Queens Counting Algorithm
- How 8 Queens Works
- Solving 8-queens with determinants comment by Mike Spivey
- Backtracking Algorithms
- N Queen Problem

ToC

## 2 Web Sites & References

## References

Bell, Jordan and Brett Stevens (2009). A survey of known results and research areas for *n*-queens. *Discrete Mathematics*, 309(1):1–31. ISSN 0012-365X. doi: https://doi.org/10.1016/j.disc.2007.12.043. URL https://www.sciencedirect.com/science/article/pii/S0012365X07010394. 9

Bird, Richard (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition. ISBN 0134843460. 3

Bird, Richard and Jeremy Gibbons (2020). *Algorithm Design with Haskell*. Cambridge University Press. ISBN 9781108869041. URL https://www.cs.ox.ac.uk/publications/books/adwh/. 1, 3, 9

Bird, Richard and Phil Wadler (1988). *Introduction to Functional Programming*. Prentice Hall, first edition. ISBN 0134841972. 1, 9

Knuth, Donald (2023). *Art of Computer Programming, The: Combinatorial Algorithms, Volume 4B*. Addison-Wesley Professional, Boston Munich. ISBN 0201038064. 3, 4, 8, 9

ToC