Algorithms

授課老師：張景堯

Backtracking

Chapter.9-Combinatorial Search and Heuristic Methods

Sudoku

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Solving Sudoku

- Solving Sudoku puzzles involves a form of exhaustive search of possible configurations.
- However, exploiting constraints to rule out certain possibilities for certain positions enables us to *prune* the search to the point people can solve Sudoku by hand.
- Backtracking is the key to implementing **exhaustive search** programs correctly and efficiently.

(回溯法就是一種有效率且結果正確的**窮舉法**)

Backtracking(回溯)

- Backtracking is a systematic method to iterate through all possible configurations of a search space. It is a general algorithm which must be customized for each application.
- We model our solution as a vector $a = (a_1, a_2, \ldots, a_n)$, where each element a_i is selected from a finite ordered set S_i.
- Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S.

(課程的回溯法框架，是將解建成有順序的串列a，每一個元素a_i從有限解空間S取值，最後組成完整可能解，就像做排列組合題目，從欲排列的資料依序取值組合成所有解)

The Idea of Backtracking

- At each step in the backtracking algorithm, we start from a given partial solution, say, $a = (a_1, a_2, \dots, a_k)$, and try to extend it by adding another element at the end.
- After extending it, we test whether what we have so far is a complete solution.
- If not, the critical issue is whether the current partial solution a is potentially extendible to a solution.
  - If so, recur and continue.
  - If not, delete the last element from a and try another possibility for that position if one exists.

(經由修剪掉違背規定的部分解，進行搜尋，走不通時則回溯到上一次符合規定的部分解，繼續往另一可能解找下去)

Recursive Backtracking(遞迴回溯法)

```
Backtrack(a, input):
    if a is a solution:
        print(a)
    else:
        compute S  /* the set of candidates */
        for c in S :
            a.append(c)
            Backtrack(a, input)
            a.pop( )  /* After backtrack remove the last one */
```

Backtracking and DFS

- Backtracking is really just **depth-first search** on an implicit graph of all possible configurations.
  - Backtracking can easily be used to iterate through **all subsets or permutations** of a set.
  - Backtracking ensures **correctness** by enumerating all possibilities.
  - For backtracking to be **efficient**, we must prune dead or redundant branches of the search space whenever possible.

Backtracking Implementation

```python
def do_backtrack(a:list, inputs:list):
    c = []
    if (is_a_solution(a, inputs)):
        process_solution(a, inputs)
    else:
        construct_candidate(a, inputs, c)
        for i in c:
            a.append(i)
            do_backtrack(a, inputs)
            a.pop()

def is_a_solution(a:list, inputs:list)->bool: return NotImplemented
def construct_candidate(a:list, inputs:list, c:list): return NotImplemented
def process_solution(a:list, inputs:list): return NotImplemented
```

Is_a_solution(a, inputs)

- This Boolean function tests whether the current first $k = len(a)$ elements of vector a are a complete solution for the given problem.
- The last argument, inputs, allows us to pass general information into the routine to evaluate whether a is a solution.

(判斷 a 串列是不是題目的解答)

construct_candidates(a, inputs, c)

- This routine fills a list c with the complete set of **possible candidates** for the $\boldsymbol{next}$ position of a.
- The number of candidates returned is $len(c)$.

(從當前部分解 a 串列，找出下一個可能當解的元素集合)

process_solution(a, inputs)

- This routine prints, counts, or somehow processes a complete solution once it is constructed.
- Backtracking ensures correctness by enumerating all possibilities. It ensures efficiency by never visiting a state more than once.
- Because a new candidates c is allocated with each recursive procedure call, the subsets of not-yet-considered extension candidates at each position will not interfere with each other.

(對解答進行處理，如顯示、計數或把a與inputs整併等等)

Constructing all Subsets

- To construct all 2^n subsets, set up an array/vector of n cells, where the value of a_i is either **True** or **False**, signifying whether the ith item is or is not in the subset.

- To use the notation of the general backtrack algorithm, $S = (True, False)$, and v is a solution whenever $len(a) \geq len(inputs)$.

Subset Generation Tree / Order

- What order will this generate the subsets of {1, 2, 3}?

$(1) \rightarrow (1,2) \rightarrow (1,2,3)* \rightarrow$

$(1,2,-)* \rightarrow (1,-) \rightarrow (1,-,3)* \rightarrow$

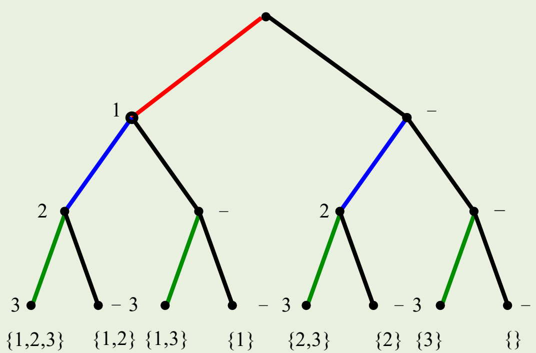
$(1,-,-)* \rightarrow (1,-) \rightarrow (1) \rightarrow$

$(-) \rightarrow (-,2) \rightarrow (-,2,3)* \rightarrow$

$(-,2,-)* \rightarrow (-,-) \rightarrow (-,-,3)* \rightarrow$

$(-,-,-)* \rightarrow (-,-) \rightarrow (-) \rightarrow ()$

```
[1]->[1, 2]->[1, 2, 3]->*
[1, 2]->[1, 2, '-']->*
[1, 2]->[1]->[1, '-']->[1, '-', 3]->*
[1, '-']->[1, '-', '-']->*
[1, '-']->[1]->[]->['-']->['-', 2]->['-', 2, 3]->*
['-', 2]->['-', 2, '-']->*
['-', 2]->['-']->['-', '-']->['-', '-', 3]->*
['-', '-']->['-', '-', '-']->*
['-', '-']->['-']->[]->end
```



Using Backtrack to Construct Subsets

- We can construct all subsets of n items by iterating through all 2^n length-n vectors of true or false, letting the ith element denote whether item i is (or is not) in the subset.
- Thus the candidate set $S = (True, False)$ for all positions, and a is a solution when $len(a) \geq len(inputs)$.

```python
def is_a_solution(a:list, inputs:list)->bool:
    return len(a) == len(inputs)


def construct_candidate(a:list, inputs:list, c:list):
    c.append(True)
    c.append(False)
```

Process the Subsets

- Here we **print the elements in each subset**, but you can do whatever you want – like test whether it is a vertex cover solution. . .

```
def process_solution(a:list, inputs:list):
    # print([inputs[i] for i, x in enumerate(a) if x])
    print([inputs[i] if x else '-' for i, x in enumerate(a)])
```

Main Routine: Subsets

- Finally, we must instantiate the call to backtrack with the right arguments.

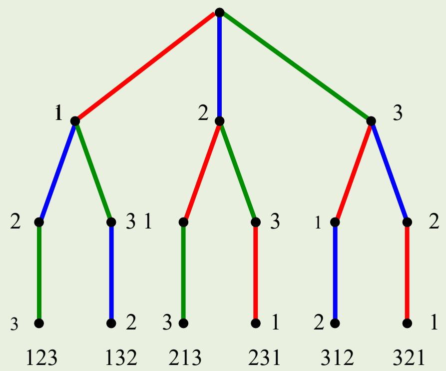
```
do_backtrack([], ['1', '2', '3'])
```

Constructing all Permutations

- How many permutations are there of an n-element set?
- To construct all $n!$ permutations, set up an array/vector of n cells, where the value of a_i is an integer from 1 to n which has not appeared thus far in the vector, corresponding to the ith element of the permutation.
- To use the notation of the general backtrack algorithm, $S = (1, \ldots, n) - a$, and a is **one of solutions** whenever $len(a) \geq len(input)$.

Permutation Generation Tree / Order

$$
\begin{array}{rcl}
(1) & \rightarrow & (1,2) \rightarrow (1,2,3)* \rightarrow (1,2) \rightarrow (1) \rightarrow (1,3) \rightarrow \\
(1,3,2)* & \rightarrow & (1,3) \rightarrow (1) \rightarrow () \rightarrow (2) \rightarrow (2,1) \rightarrow \\
(2,1,3)* & \rightarrow & (2,1) \rightarrow (2) \rightarrow (2,3) \rightarrow (2,3,1)* \rightarrow (2,3) \rightarrow () \\
(2) & \rightarrow & () \rightarrow (3) \rightarrow (3,1) \rightarrow (3,1,2)* \rightarrow (3,1) \rightarrow (3) \rightarrow \\
(3,2) & \rightarrow & (3,2,1)* \rightarrow (3,2) \rightarrow (3) \rightarrow ()
\end{array}
$$



Constructing All Permutations

- To avoid **repeating** permutation elements, $S = \{1, \ldots, n\} - a$, and a is a solution whenever $len(a) == len(inputs)$:

```python
def construct_candidate(a:list, inputs:list, c:list):
    for i in inputs:
        if i not in a:
            c.append(i)
```

Auxilliary Routines

- Completing the job of generating permutations requires specifying *process_solution* and *is_a_solution*, as well as setting the appropriate arguments to backtrack. All are essentially the same as for subsets:

```python
def is_a_solution(a:list, inputs:list)->bool:
    return len(a) == len(inputs)


def process_solution(a:list, inputs:list):
    print(a)
```

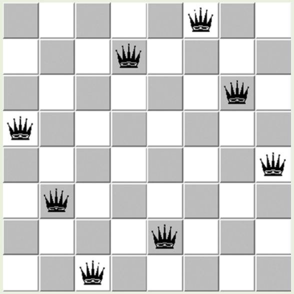
Main Program : Permutations

```
do_backtrack([], [1, 2, 3])
```

N-Queens Problem

Chapter.9-Combinatorial Search and Heuristic Methods

The Eight-Queens Problem



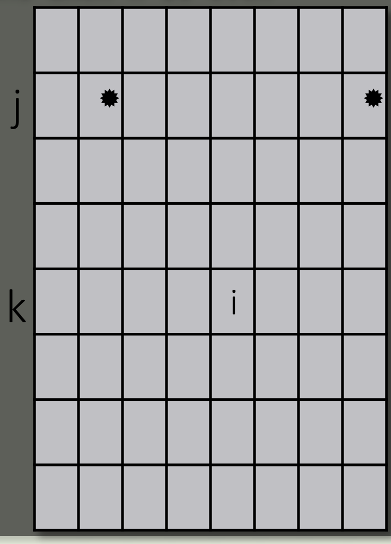
- The eight queens problem is a classical puzzle of positioning eight queens on an 8×8 chessboard such that no two queens threaten each other.
- 4-Queens: https://www.youtube.com/watch?v=R8bM6pxlrLY

Eight Queens: Representation

- What is concise, efficient representation for an n-queens solution, and how big must it be?
- Since no two queens can occupy the same column, we know that the n columns of a complete solution must form a permutation of n. By avoiding repetitive elements, we reduce our search space to just $8! = 40{,}320$ – clearly short work for any reasonably fast machine.
- The critical routine is the candidate constructor. We repeatedly check whether the kth square on the given row is threatened by any previously positioned queen. If so, we move on, but if not we include it as a possible candidate:

Candidate Constructor: Eight Queens

```python
def construct_candidate(a:list, inputs:list, c:list):
    k = len(a)
    n = len(inputs)
    # if k == 0 and n & 1 == 0:
    #     n //= 2 #Only get left part of symmetric solutions when n is even
    for i in range(n):
        legal_move = True
        for j in range(k):
            if abs(k-j) == abs(i-a[j]): #Diagonal threat
                legal_move = False
            if i == a[j]: #Column threat
                legal_move = False
        if legal_move:
            c.append(i)
```



Auxilliary Routines

- The remaining routines are simple, particularly since we are only interested in counting the solutions, not displaying them:

```
def is_a_solution(a:list, inputs:list)->bool:
    return len(a) == len(inputs)


def process_solution(a:list, inputs:list):
    global solution_count
    solution_count += 1
    # if len(inputs) & 1 == 0:
    #     solution_count += 1
```

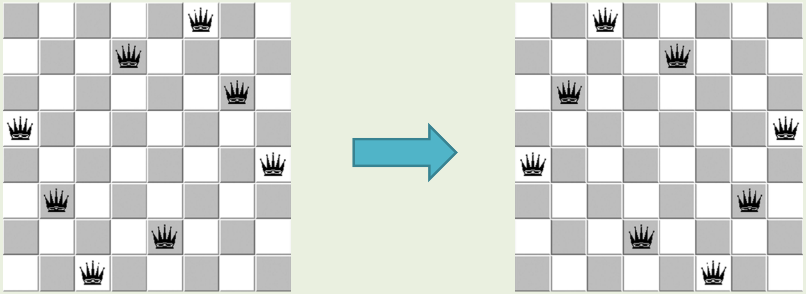
Finding the Queens: Main Program

```
solution_count = 0
n = 14
do_backtrack([],[0]*n)
print(solution_count)
```

- This program can find the 365,596 solutions for $n = 14$ in minutes.

Power of Symmetry (棋盤對稱性)

- Consider putting in a mirror to get the reflecting solution…



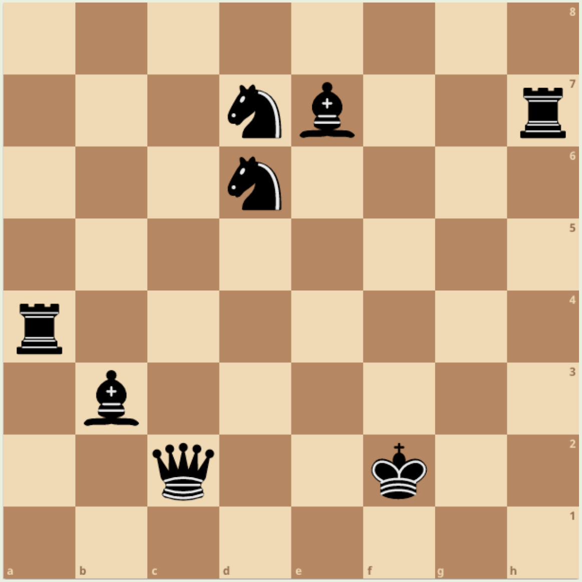
- We can reduce the execution time to half.

Combinatorial Search

Chapter.9-Combinatorial Search and Heuristic Methods

Can Eight Pieces Cover a Chess Board?

- Consider the 8 main pieces in chess (king, queen, two rooks, two bishops, two knights). Can they be positioned on a chessboard so every square is threatened?



Combinatorial Search(組合搜尋)

- Only 63 squares are threatened in this configuration. Since 1849, no one had been able to find an arrangement with bishops on different colors to cover all squares.
- We can resolve this question by **searching through all possible board configurations** *if* we spend enough time.
- We will use it as an example of how to attack a combinatorial search problem.
- With clever use of backtracking and pruning techniques, surprisingly large problems can be solved by exhaustive search.

(進行組合搜尋要能有智慧地透過回溯與修剪縮小搜尋空間提高效率，而不是暴力窮舉搜尋)

How Many Chess Configurations Must be Tested?

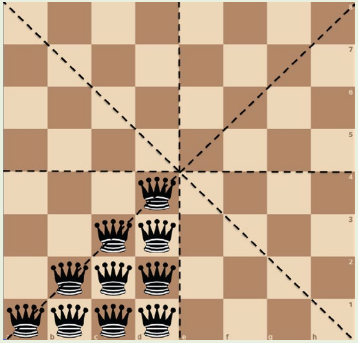
- Picking a square for each piece gives us the bound:

$$\frac{64!}{(64-8)!} = 178{,}462{,}987{,}637{,}760 \approx 10^{15}$$

- Anything much larger than 10^8 is unreasonable to search on a modest computer in a modest amount of time.

Exploiting Symmetry(利用對稱性)

- However, we can exploit symmetry to save work. With reflections along **horizontal**, **vertical**, and **diagonal axis**, the queen can go in only 10 non-equivalent positions.



- Even better, we can restrict the one bishop to 32 spots and another bishop to 31, while being certain that we get all distinct configurations.

$$10 \times 32 \times 31 \times \frac{61 \times 60}{2} \times \frac{59 \times 58}{2} \times 57 = 1{,}770{,}466{,}147{,}200 \approx 1.8 \times 10^{12}$$

Q B B 2R 2N K

Covering the Chess Board

- In covering the chess board, we prune whenever we find there is a square which we *cannot* cover given the initial configuration!
- Specifically, each piece can threaten a certain maximum number of squares (queen 27, king 8, rook 14, etc.) We *prune* whenever the number of un-threated squares exceeds the sum of the maximum remaining coverage.(皇后最多吃27格、國王8、城堡14等，如果剩下空格超出剩餘棋子最多能吃的格子總和，就可以不用找下去了，直接回溯)
- As implemented by a graduate student project, this backtrack search eliminates 95% of the search space, when the pieces are ordered by decreasing mobility.
- With precomputing the list of possible moves, this program could search 1,000 positions per second.

End Game

- But this is still too slow!

$$\frac{1.8 \times 10^{12}}{10^3} = 1.8 \times 10^9 \text{ seconds} > 60\ years$$

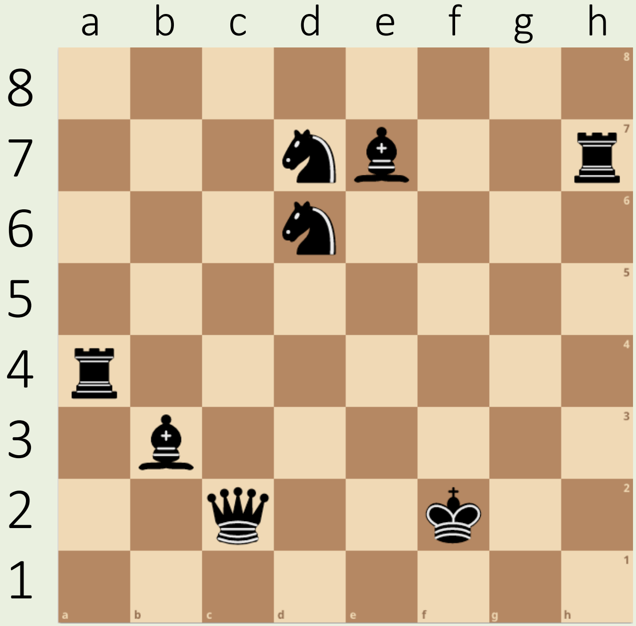
- Although we might further speed the program by an order of magnitude, we need to prune more nodes!
- By using a more clever algorithm, we eventually were able to prove no solution existed, in less than one day’s worth of computing.
- You too can fight the combinatorial explosion!

(使用組合搜尋要盡一切技巧進行修剪避免碰到組合爆炸)
組合爆炸動畫：https://www.youtube.com/watch?v=Q4gTV4r0zRs

Exercises

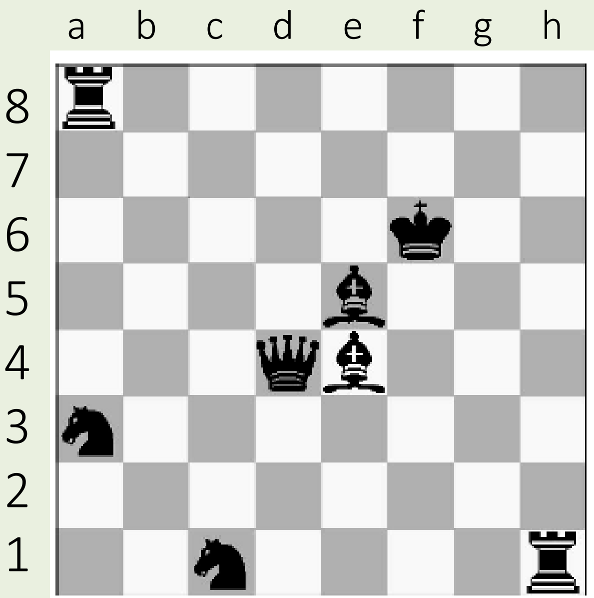
Problem of the Day

- Q1: There are 63 squares threatened by 8 main pieces in chess (king, queen, two rooks, two bishops, two knights) in this case. Can you find which square is unthreatened in this case?



Problem of the Day (Cont.)

- Q2: There are 63 squares threatened by 8 main pieces in chess (king, queen, two rooks, two bishops, two knights) in this case. Can you find which square is unthreatened in this case?



Problem of the Day (Cont.)

- Q3: A derangement is a permutation p of $\{0, \ldots, n\}$ such that no item is in its proper position, i.e. $p_i \neq i$ for all $0 \leq i \leq n$.
- How many legal derangements of [0,1,2]?

Program Exercises (Moodle CodeRunner)

- Exercise 08 (close at 5/6 23:59)
  - N-Queens Problem:
    - *Positioning N queens on an N×N chessboard such that no two queens threaten each other, thus, a solution requires that no two queens share the same row, column, or diagonal.*
  - Implement a Sudoku Solver:
    - *Sudoku is a popular logic-based combinatorial number placement puzzle. The objective is to fill a 9X9 grid with digits subject to the constraint that each column, each row, and each of the nine 3x3 sub-grids that compose the grid contains unique integers in [1, 9].*