CX1107
Data Structures and Algorithms

Backtracking and Dynamic Programming

The Eight Queens Problem and 0/1 Knapsack Problem

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

College of Engineering

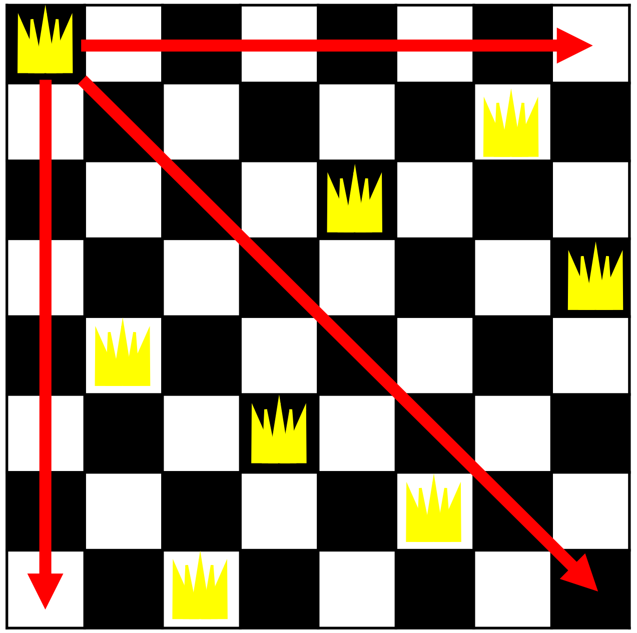
School of Computer Science and Engineering

Example of Depth First Search

The Eight Queens Problem

by Max Bezzel in 1848

The Eight Queens Problem



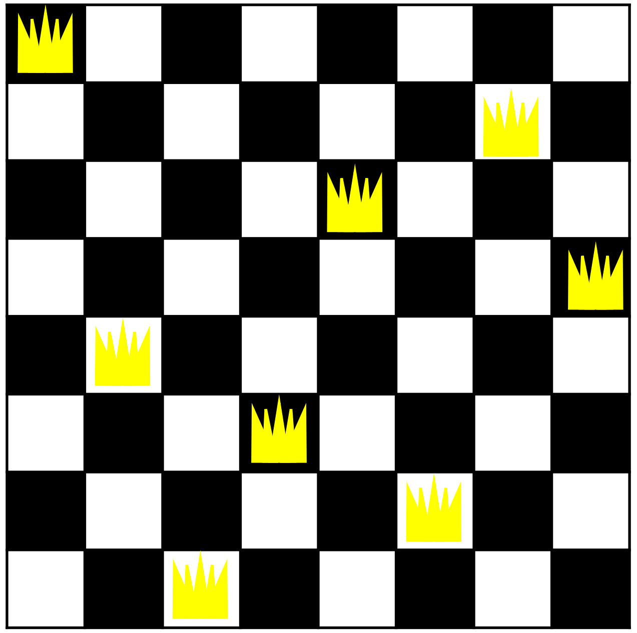
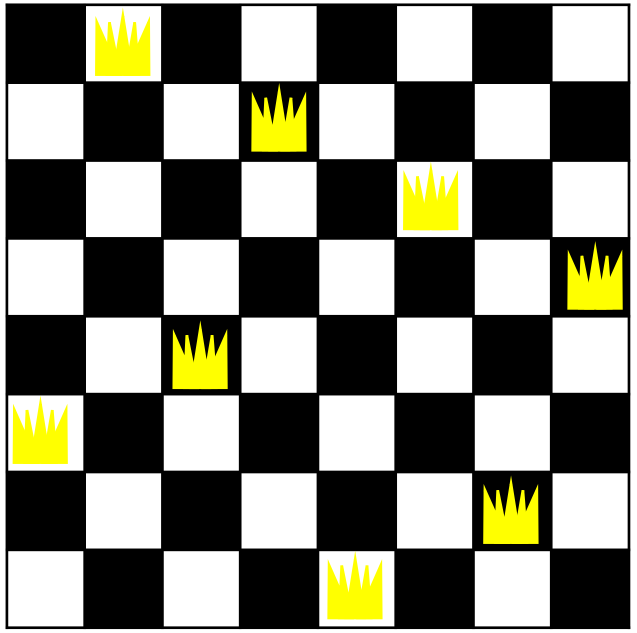
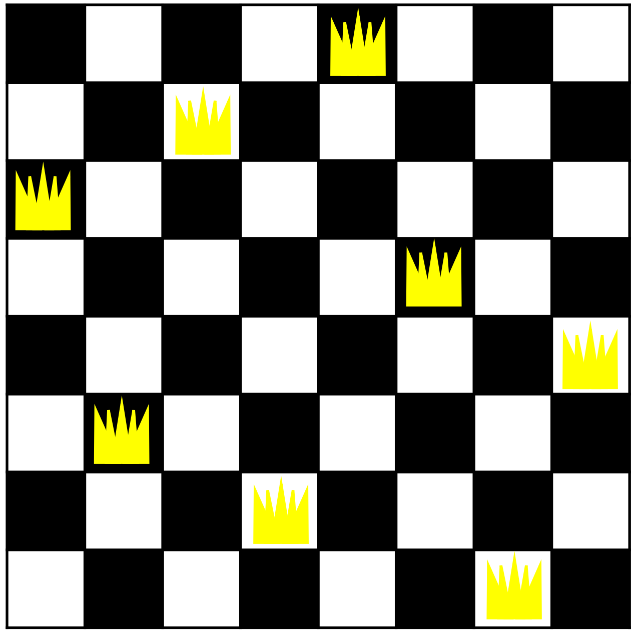
- A chessboard has 8 rows and 8 columns
- A queen can move within its row or its column or along its diagonal
- Place 8 queens on the board
  - No queen can attack any other queen in a move

- Exhausting search will take $\binom{64}{8} = 4.43$ billion ways
- Each row can contain exactly one queen: $8! = 40{,}320$
- Better algorithm?

Backtracking Algorithm

1. Starts by placing a queen in the top left corner of the chess board.
2. Places a queen in the second column and moves her until a place where she cannot be hit by the queen in the first column.
3. Places a queen in the third column and moves her until she cannot be hit by either of the first two queens and so on.
4. If there is no place for the i^{th} queen, the program backtracks to move the $(i-1)^{\text{th}}$ queen.
5. If the $(i-1)^{\text{th}}$ queen is at the end of the column, the program removes the queen and backtracks to the $(i-2)$ column and so on.
6. If the current column is the last column and a safe place has been found for the last queen, then a solution to the puzzle has been found.

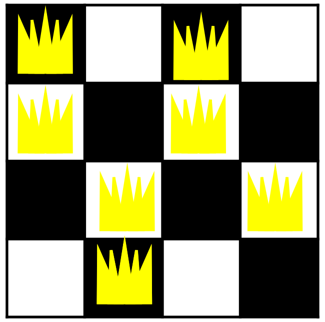
Backtracking Algorithm

- If the current column is the first column and its queen is being moved off the board then all possible configurations have been examined, all solutions have been found, and the algorithm terminates.
- This puzzle has **92** solutions.

The Eight Queens Problem’s Algorithm

```
function NQUEENS($Board[N][N]$, $Column$)
    if Column >= N then return true                          ▷ Solution is found
    else
        for $i \leftarrow 1, N$ do
            if Board[i][Column] is safe to place then
                Place a queen in the square
                if NQueens($Board[N][N]$, $Column + 1$) then return true    ▷ Solution is found
                end if
                Delete the queen
            end if
        end for
    end if
return false                                                 ▷ no solution is found
end function
```



Dynamic Programming

by Richard Ernest Bellman in 1953

What is Dynamic Programming?

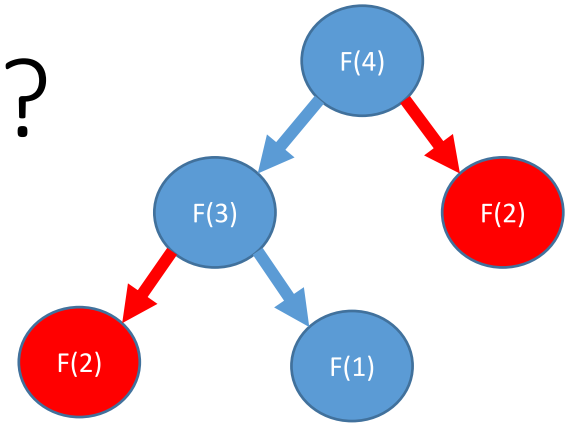
- It is not a programming language like C
    - The term “Programming” refers to a tabular method (filling tables)
        - It is applied to optimization problems
            - Other “programming” methods in mathematical optimization are
                - Linear Programming
                - Integer Programming
                - Convex Programming
                - Semidefinite Programming
            - not related to coding
- Applied from system control to economics

What is Dynamic Programming (DP)?

- It is similar to divide-and-conquer strategy
    - Breaking the big problem into sub-problems
    - Solve the sub-problems recursively
    - Combining the solutions to the sub-problems

- What is the difference between them?
    - DP applied when the sub-problems are not independent
        - Every sub-problem is solved once and is saved in a table
    - The problem usually can have multiple optimal solutions
        - DP may just return one of them

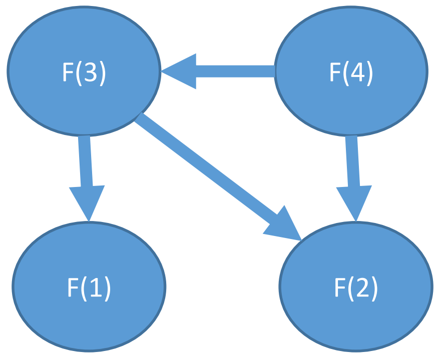
What is Dynamic Programming (DP)?



- Optimal substructure
  - Combination of optimal solutions to its sub-problems

- Overlapping sub-problems
  - Having the same sub-problems

$\Theta(2^n) \Rightarrow \Theta(n^p)$



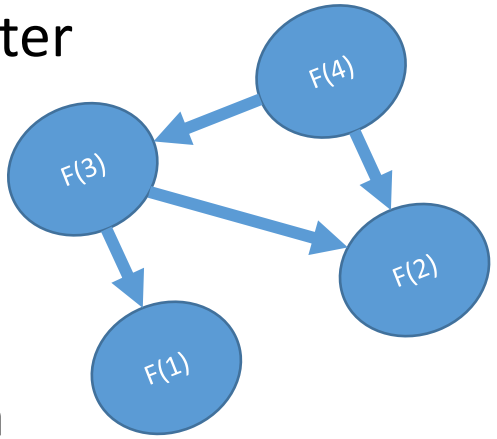
➢ Fibonacci Series: $F_i = F_{i-1} + F_{i-2}$

- Recursion: problem can be solved recursively
- Memoization: Store optimal solutions to sub-problems in table (or memory or cache)
  => If the sub-problems are independent, DP is not useful!

Dynamic Programming = Recursion + Memoization

Dynamic Programming Approaches

- Top-down approach
  - Recursively using the solution to its sub-problems
  - Memoize the solutions to the sub-problems and reuse them later
- Bottom-up approach
  - Figure out the order of calculation
  - Solve the sub-problems to build up solutions to larger problem

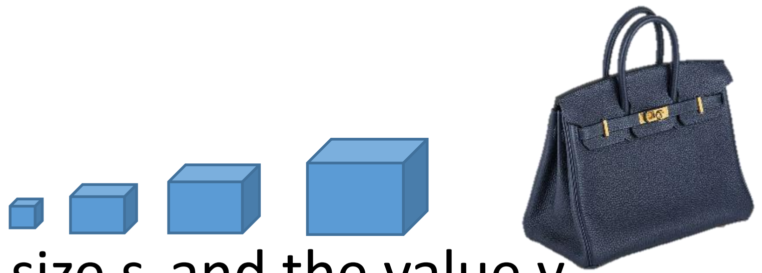


Examples of DP

- String algorithms like longest common subsequence, longest increasing subsequence, longest common substring etc.
- Graph algorithms like Bellman-Ford algorithm, Floyd’s algorithm
- Chain matrix multiplication

- Subset Sum
- 0/1 Knapsack
- Travelling salesman problem

0/1 Knapsack



- Given *n* items, where the ith item has the size s_i and the value v_i
- Put these items into a knapsack of capacity *C*

- *Optimization problem: Find the largest total value of the items that fits in the knapsack*

$$\max_x \sum_{i=1}^{n} v_i x_i$$

Subject to

$$\sum_{i=1}^{n} s_i x_i \le C$$

$$x_i \in \{0,1\} \qquad\qquad i = 1, 2, \dots, n$$

0/1 Knapsack

$$\max_{x} \sum_{i=1}^{n} v_i x_i$$

Subject to

$$\sum_{i=1}^{n} s_i x_i \leq C$$

$$x_i \in \{0,1\} \qquad i = 1, 2, \ldots, n$$

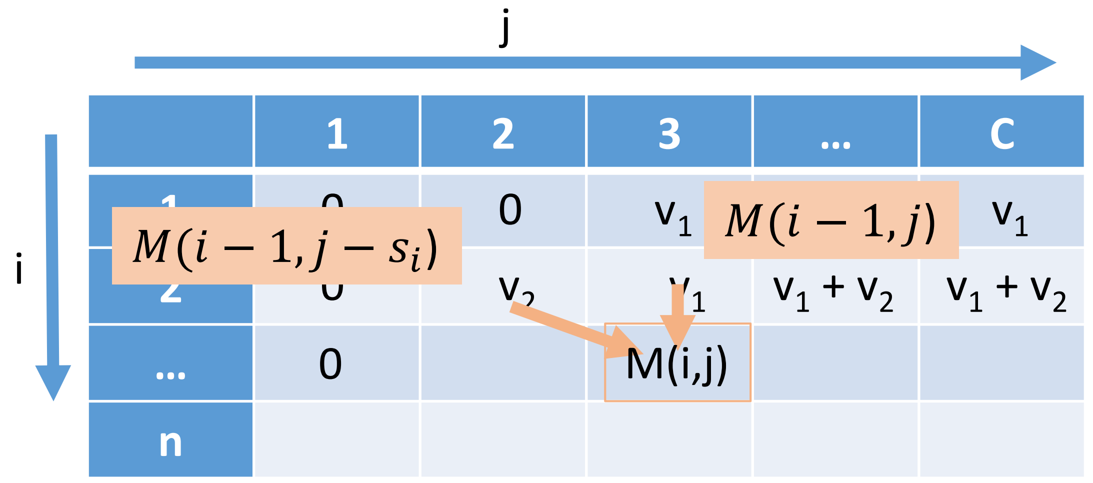
- Brute-force algorithm or Exhaustive Search
- The ith item is either included (1) or excluded (0)
- The time complexity of the algorithm is $\Theta(2^n)$

Item 1	Item 2	Item 3	Value
0	0	0	0
0	0	1	V3
0	1	0	V2
0	1	1	V2+V3
1	0	0	V1
1	0	1	V1+V3
1	1	0	V1+V2
1	1	1	V1+V2+V3

Can you see that some sub-problems are overlapping?

Using DP to solve 0/1 KnapSack

- The recursive formula
  - $M(i,j) = \max\{\underbrace{M(i-1,j)}_{\text{ith item is unused}},\ \underbrace{M(i-1,j-s_i)+v_i}_{\text{ith item is used}}\}$
    - i = 1, … n
    - j = 1, … C
- Create a n-by-C matrix, M
- All the possible sizes from 1 to C
- Bottom up approach
- Time Complexity is $\Theta(nC)$



Summary

- Backtracking
- Dynamic Programming