Introduction to Computer Science:

algorithm paradigm

May 2020

Honguk Woo

Algorithm paradigms

- General approaches to construction of efficient solutions to problems
  - Can be used as a reference to design a new algorithm to problems

  - Yet, we learned *"Divide and Conquer"*
    - Divide a problem instance into smaller sub-instances of the same problem, solve these recursively, and then put solutions together to a solution of the given instance
    - *e.g, Binary Search, Mergesort, Qsort …*
    - Relevant to many algorithms that can be implemented by using recursion

  - *There are many other algorithm paradigms*
    - *Bruce forth*
    - *Backtracking*
    - *Dynamic Programming*
    - *Greedy*

Binary Search Algorithm with Recursion

- Divide into two *BinarySearch* with halves

```
BinarySearch (first, last, item)
IF (first > last)
        RETURN FALSE
ELSE
        Set middle to (first + last)/ 2
        IF (item equals data[middle])
                        RETURN TRUE
        ELSE
                IF (item < data[middle])
                        BinarySearch (first, middle – 1, item)
                ELSE
                        BinarySearch (middle + 1, last, item)
```

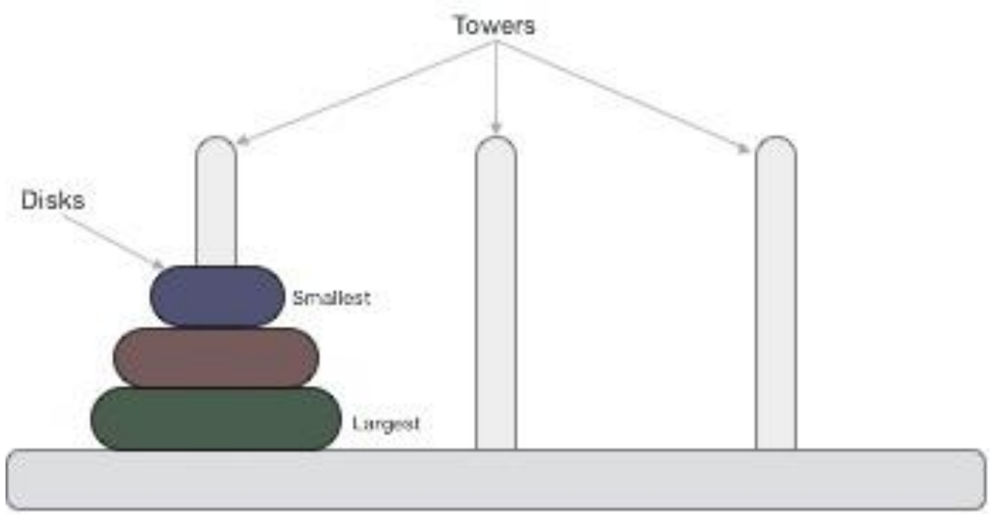
Quicksort sort Algorithm with Recursion

```
Quicksort(first, last)
IF (first < last)          // There is more than one item
   Select pivot
   pivotPoint = partition (pivot, first, last)
   Quicksort (first, pivotPoint - 1)
   Quicksort (pivotPoint + 1, last)
```

```
partition(pivot, first, last)
Set left to first + 1
Set right to last
WHILE (left <= right)
      Increment left until data[left] > pivot OR left > right
      Decrement right until data[right] < pivot OR left > right
      IF(left < right)
          Swap data[left] and data[right]
Set pivotPoint to right
Swap data[first] and data[pivotPoint]
Return pivotPoint
```

Discussion : Tower of Hanoi

- The mission is to move all the disks to some another tower without violating the sequence of arrangement.
  - Only one disk can be moved among the towers at any given time
  - Only the "top" disk can be removed
  - No large disk can sit over a small disk



Tower of Hanoi : Hint

- *Let's assume three towers (source, aux, destination)*
- First, we move the smaller (top) disk to aux
- Then, we move the larger (bottom) disk to destination
- And finally, we move the smaller disk from aux to destination

- In general
- **Step 1** − Move n-1 disks from **source** to **aux**
- **Step 2** − Move n^{th} disk from **source** to **dest**
- **Step 3** − Move n-1 disks from **aux** to **dest**

Tower of Hanoi : Python code

```
def towerofHanoi(n, source, dest, aux):
    if (n == 1):
        print("move disk 1 from " + source + " to " + dest)
    else:
        towerofHanoi(n-1, source, aux, dest)
        print("move disk " + str(n) + " from " + source + " to " + dest)
        towerofHanoi(n-1, aux, dest, source)


towerofHanoi(4, "A", "B", "C")
```

Tower of Hanoi : Python code

```
towerofHanoi(2, "A", "B", "C")
```

move disk 1 from A to C

move disk 2 from A to B

move disk 1 from C to B

Tower of Hanoi : Python code

```
towerofHanoi(4, "A", "B", "C")
```

move disk 1 from A to C

move disk 2 from A to B

move disk 1 from C to B

move disk 3 from A to C

move disk 1 from B to A

move disk 2 from B to C

move disk 1 from A to C

move disk 4 from A to B

move disk 1 from C to B

move disk 2 from C to A

move disk 1 from B to A

move disk 3 from C to B

move disk 1 from A to C

move disk 2 from A to B

move disk 1 from C to B

Backtracking

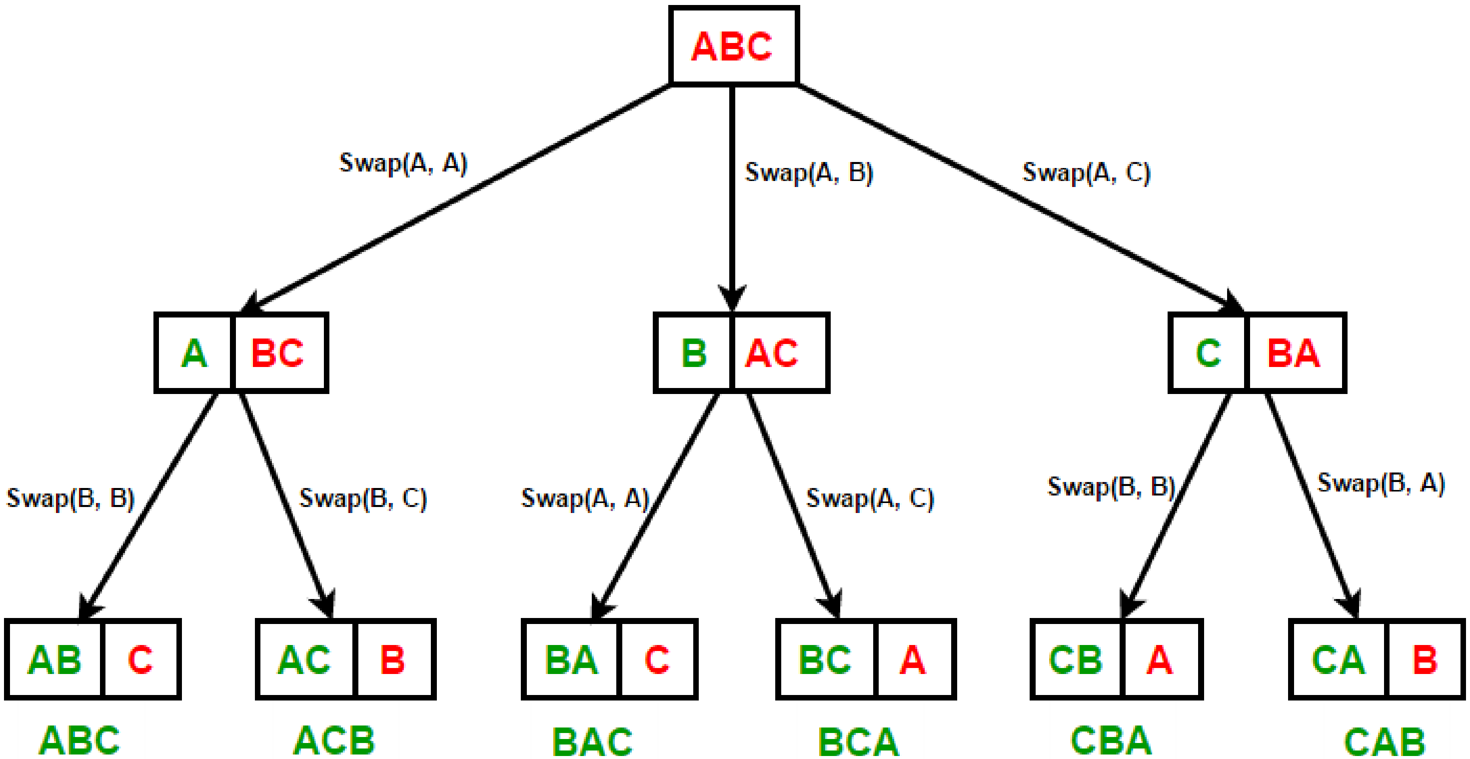
- Brute Force (Exhaustive Search)
  - a general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement
    - e.g., sequential search
  - Recursion can be used to implement Brute force algorithms
    - e.g., “ABC” string permutation

- Backtracking (Recursion with Pruning)
  - a general algorithmic technique that considers searching every possible combination
  - the search process can be pruned to avoid considering cases that don't look promising
  - *Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, that incrementally builds candidates to the solutions, and abandons each partial candidate (“backtracks”) as soon as it determines that the candidate cannot possibly be completed to a valid solution. (definition from wikipedia)*

Example of Brute force : Permutations of string

- Given a string **str**, the task is to print all the permutations of **str**.
- str = “`ABC`”, how many permutations ?
  - `ABC`
  - `ACB`
  - `BAC`
  - `BCA`
  - :

Permutations of “ABC” string



Recursion Tree for string "ABC"

http://www.techiedelight.com/generate-permutations-string-java-recursive-iterative/

“ABC” permutation : C code

```
def permute(s, l, r):
    if (l == r):
        print(s)
    else:
        for i in range(l, r+1):
            s[l], s[i] = s[i], s[l]
            permute(s, l+1, r)
            s[l], s[i] = s[i], s[l]



s = ["A", "B", "C"]
permute(s, 0, len(s)-1)
```

output

```
['A', 'B', 'C']
['A', 'C', 'B']
['B', 'A', 'C']
['B', 'C', 'A']
['C', 'B', 'A']
['C', 'A', 'B']
```

Question : swap function

- Define swap function

```
def permute(s, l, r):
    if (l == r):
        print(s)
    else:
        for i in range(l, r+1):
            swap(            )
            permute(s, l+1, r)
            swap(            )



s = ["A", "B", "C"]
permute(s, 0, len(s)-1)
```

```
def swap(                   ):
```

Question : another permute

```
def permute_2(pre_str, rem_str, n):
    if (len(pre_str) == n):
        print(pre_str)
    else:





s = ["A", "B", "C"]
permute_2("", s, len(s))
```

N-Queens : problem definition

- Given a chess board having n×n cells, we need to place n queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally (meaning **that no two queens can be in the same row, column, diagonal)**
- n=4 case :
  - list all case systematically
  - test each case if it is a solution
  - ${}_{16}C_4$ cases – n^2Cn cases
  - better way?

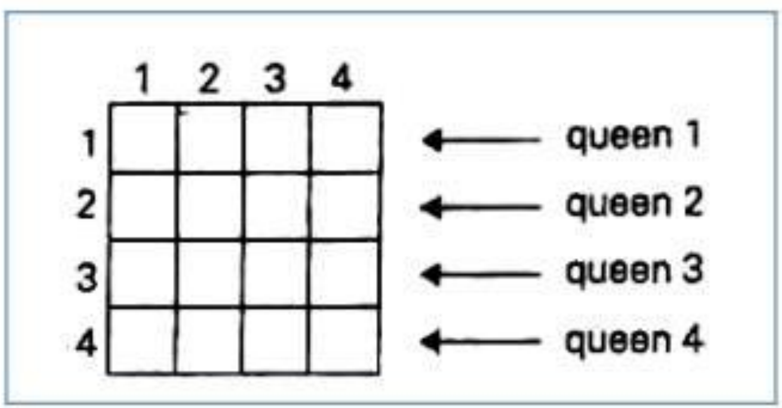
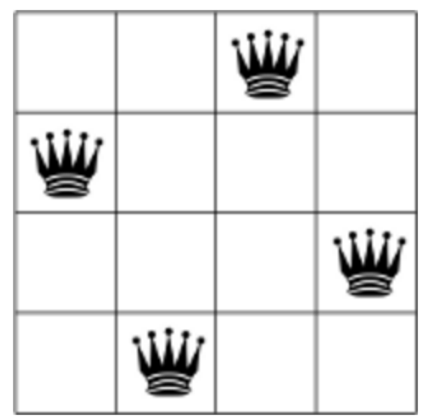


Fig: Board for the Four-queens problem



N-Queens : check if the configuration is safe

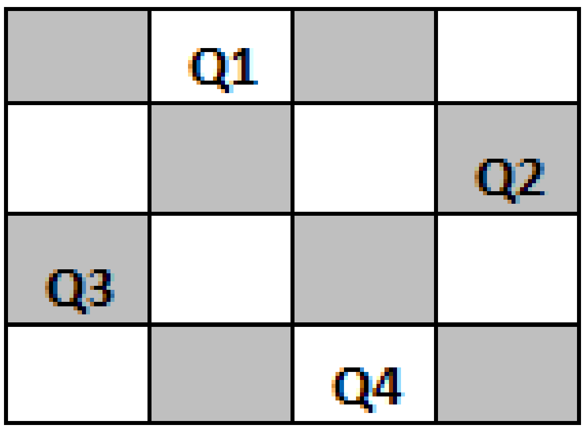
```
def isSafeConfig(s):
    for i in range(len(s)):
        for j in range(len(s)):
            if (i != j):
                if s[i] == s[j]:
                    return False
                elif abs(i-j) == abs(s[i]-s[j]):
                    return False
    return True

def permute(s, l, r):
    if (l >= r):
        global sol
        sol = sol + 1
        if(isSafeConfig(s)):
            print(str(sol) + ":" + str(s))
    else:
        for i in range(1, r+1):
            s[l] = i
            permute(s, l+1, r)

sol = 0
s = [1, 2, 3, 4]
permute(s, 0, len(s))
```

output

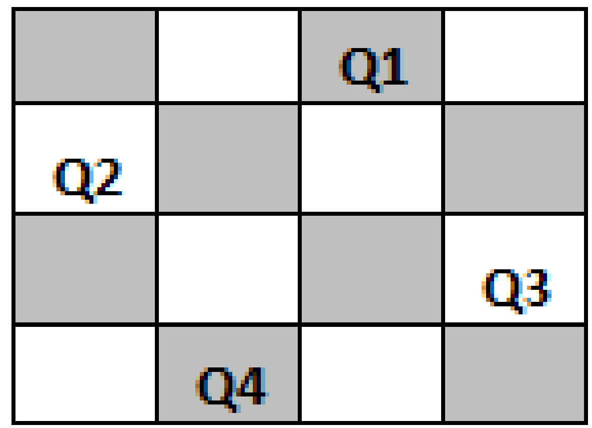
```
115: [2, 4, 1, 3]
142: [3, 1, 4, 2]
```

N-Queens : safe configurations

- [2, 4, 1, 3]
- [3, 1, 4, 2]



Solution 1

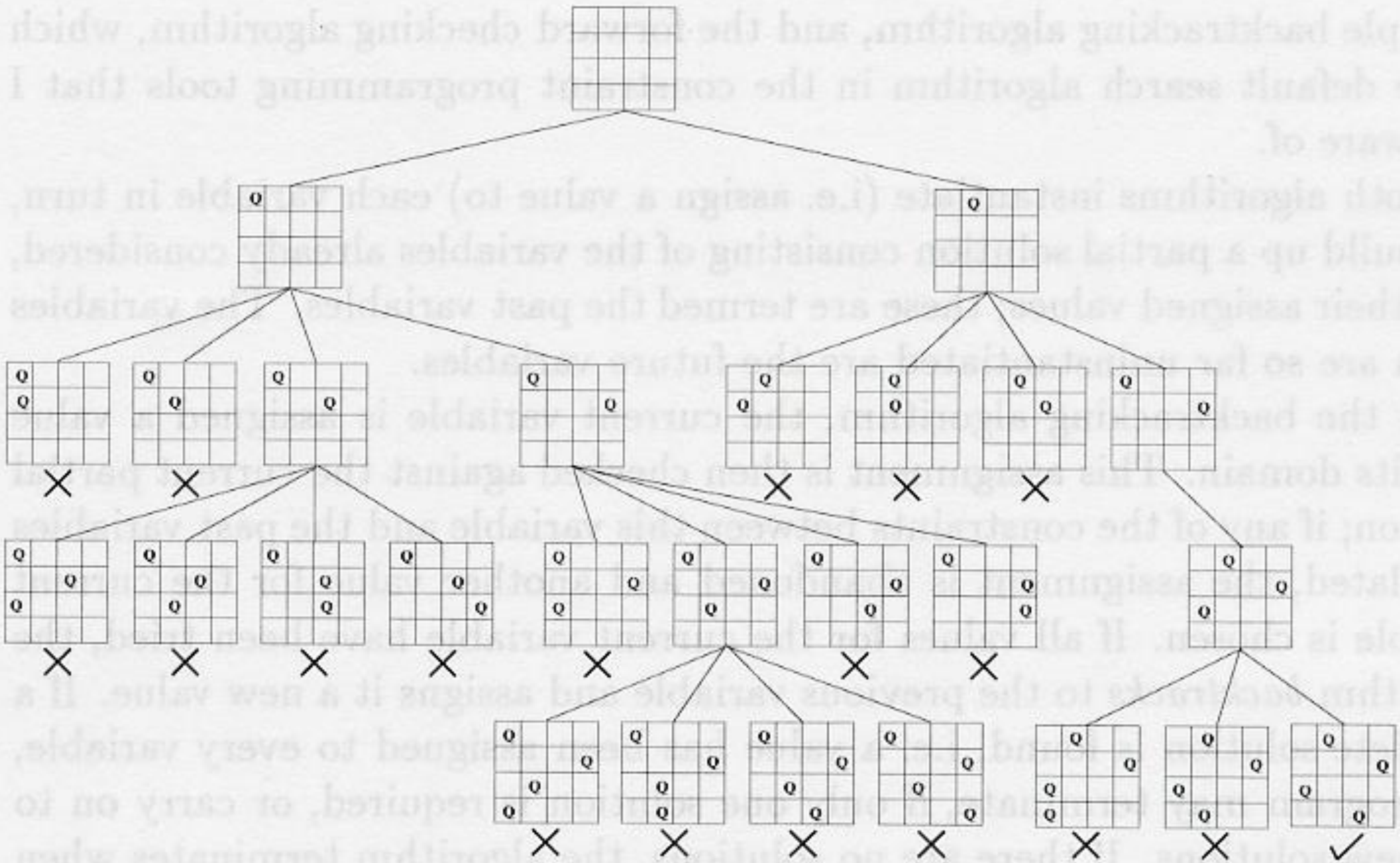


Solution 2

N-Queens : complexity of Brute force

- In case of 8-Queens
  - If we think that each queen can be at any place
    - the 1st queen = 64 cases
    - the 2nd queen = 64 cases
    - :
    - the 8th queen = 64 cases
    - TOTAL $64^8 = 2.81 * 10^{14}$ cases
  - If we think that each queen can be at any place but at a different row
    - the 1st queen = 8 cases
    - the 2nd queen = 8 cases
    - :
    - the 8th queen = 8 cases
    - TOTAL $8^8 = 1.67 * 10^7$ cases

N-Queens : backtracking



N-Queens : backtracking : Python code

```
def isSafeConfigYet(s, last):
    for i in range(last+1):
        for j in range(last+1):
            if (i != j):
                if s[i] == s[j]:
                    return False
                elif abs(i-j) == abs(s[i]-s[j]):
                    return False
    return True

def permute(s, l, r):
    if (l >= r):
        global sol
        sol = sol + 1
        print(str(sol) + ": " + str(s))
    else:
        for i in range(1, r+1):
            s[l] = i
            if(isSafeConfigYet(s, l) == True):
                permute(s, l+1, r)

sol = 0
s = [1, 2, 3, 4]
permute(s, 0, len(s))
```

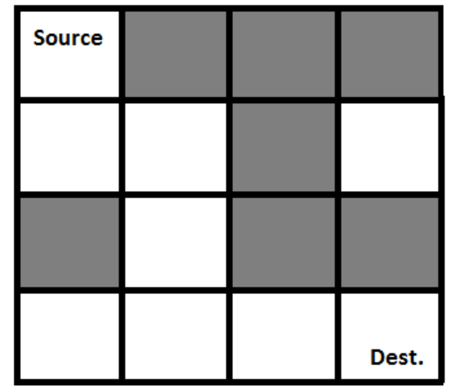
Execution time test

- 8-queens problem

- Without backtracking : 40 sec
- Backtracking : 0.17 sec

Quiz : N*N Maze

- A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]
- A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.




```
# Initialising the maze
  maze = [ [1, 0, 0, 0],
           [1, 1, 0, 1],
           [0, 1, 0, 0],
           [1, 1, 1, 1] ]

  solveMaze(maze)
```

output

```
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1
```