TUTORIAL 6
BACKTRACKING AND DYNAMIC PROGRAMMING

Dr Liu Siyuan

Email: syliu@ntu.edu.sg

Office hour: Monday & Wednesday 4-5pm

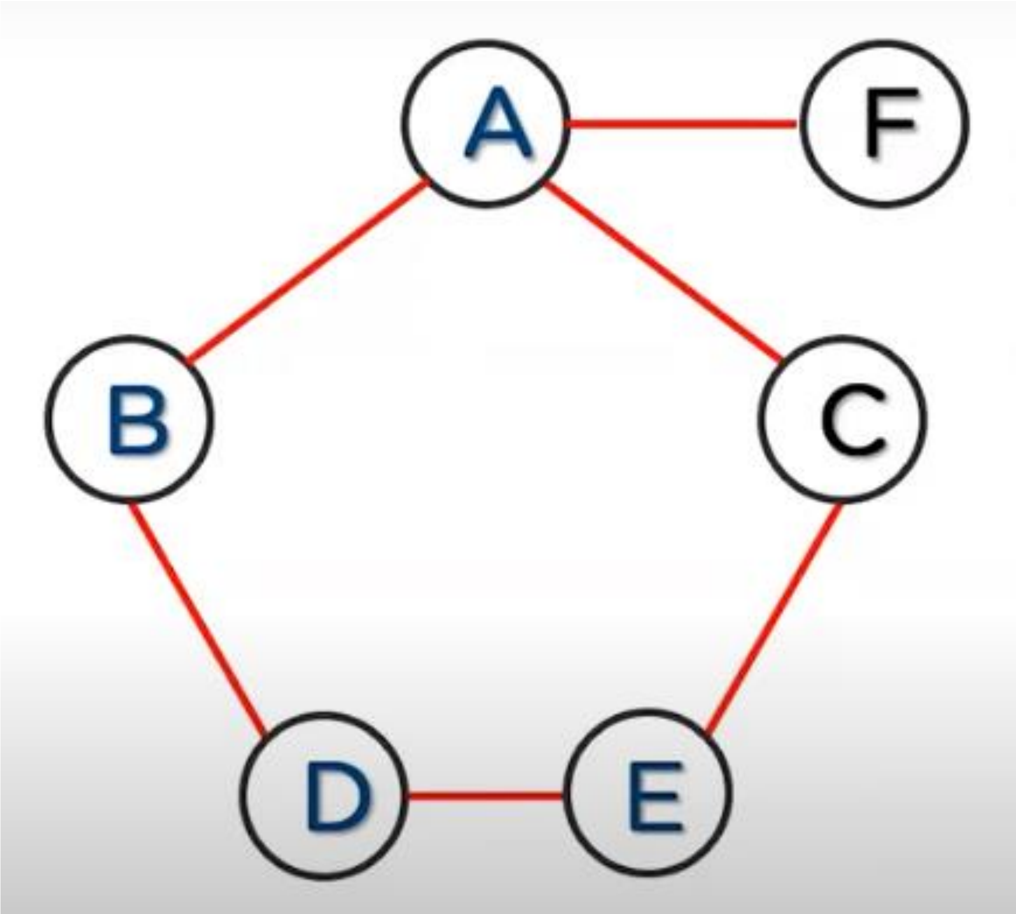
Office: N4-2C-117a

Q1

- Give a pseudocode of finding a simple path connecting two given vertices in an undirected graph by Depth-First-Search.
- Simple path: A path is simple if all of its vertices are distinct.

DFS Algorithm

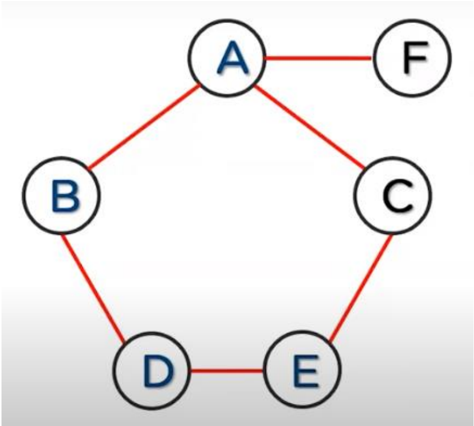
function DFS(Graph G, Vertex v)
  create a Stack, S
  push v into S
  mark v as visited
  while S is not empty **do**
    peek the stack and denote the vertex as x
    if no unvisited vertices are adjacent to x **then**
      pop a vertex from S
    else
      push an unvisited vertex u adjacent to x
      mark u as visited
    end if
  end while
end function

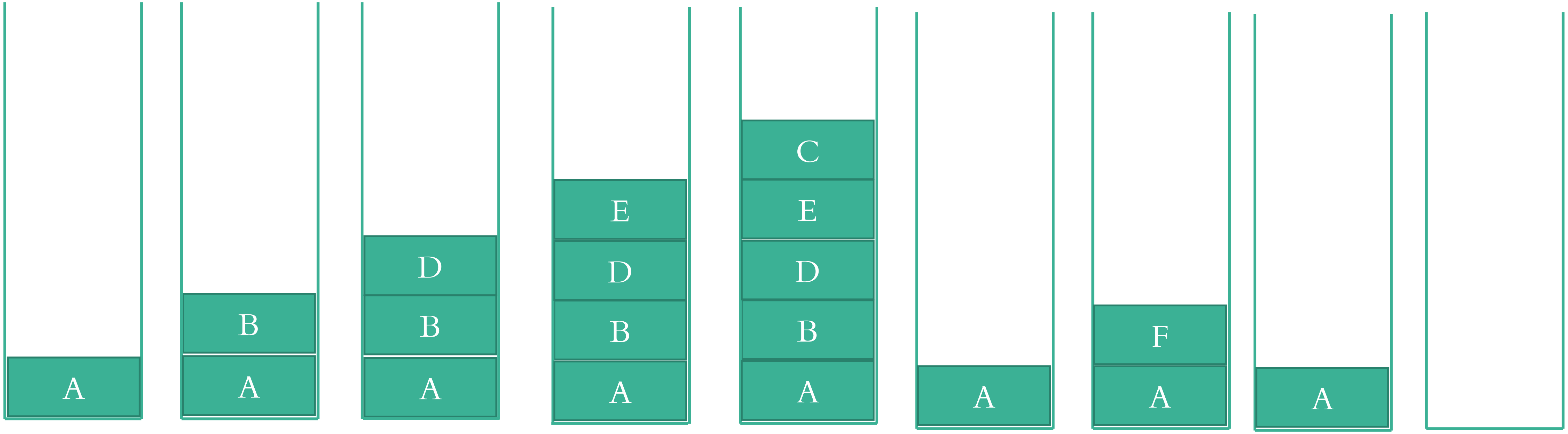


- Given A as starting vertex
- By using DFS, output can be:
  - ABDECF (alphabetical order)
  - AFCEDB (reverse alphabetical order)

Visited: A B D E C F

In the stack, each node is connected to the bottom node through a simple path, which includes all other nodes between the node and the bottom node





DFS Algorithm

```
function DFS(Graph G, Vertex v)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as x
        if no unvisited vertices are adjacent to x then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to x
            mark u as visited
        end if
    end while
end function
```

Check the top node of the stack (x) to see if it is the end vertex (w). If it is, add all the nodes between w and the bottom node (v) to the simple path.

Algorithm 1 Depth First Search (DFS)

```
function SimplePath(Graph G, Vertex v,Vertex w)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as x
        if x == w then
            while S is not empty do
                pop a vertex from S
                peek the stack
                print the link
            end while
            return Found
        end if
        if no unvisited vertices are adjacent to x then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to x
            mark u as visited
        end if
    end while
  return Not Found
  end function
```

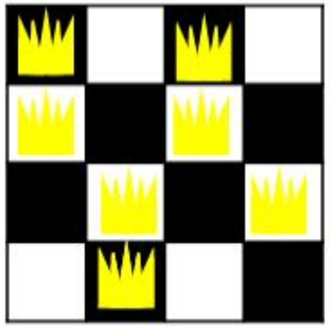
Q2

Design a backtracking algorithm to print out all possible permutation of a given sequence. For example, input is given as "1234". The 24 output permutations are printed out from "1234" to " 4321".

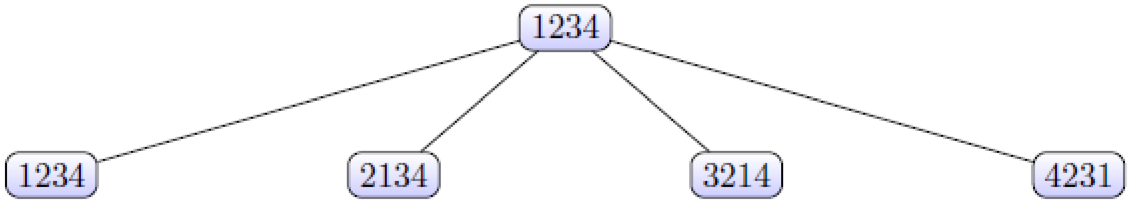
```
Backtracking(n)
        Base case: return true

        for 1 to n
                do something/move forward
                if (Backtracking(n-1)) return true
                reverse whatever you have done earlier (backtracking)
        return false;
```

The Eight Queens Problem's Algorithm

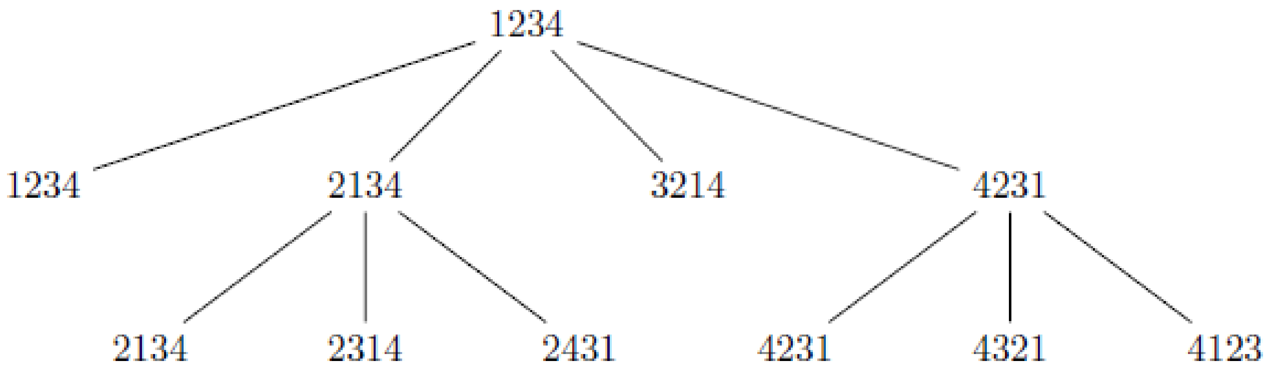
```
function NQueens(Board[N][N], Column)
    if Column >= N then return true                          ▷ Solution is found
    else
        for i ← 1, N do
            if Board[i][Column] is safe to place then
                Place a queen in the square
                if NQueens(Board[N][N], Column + 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
```

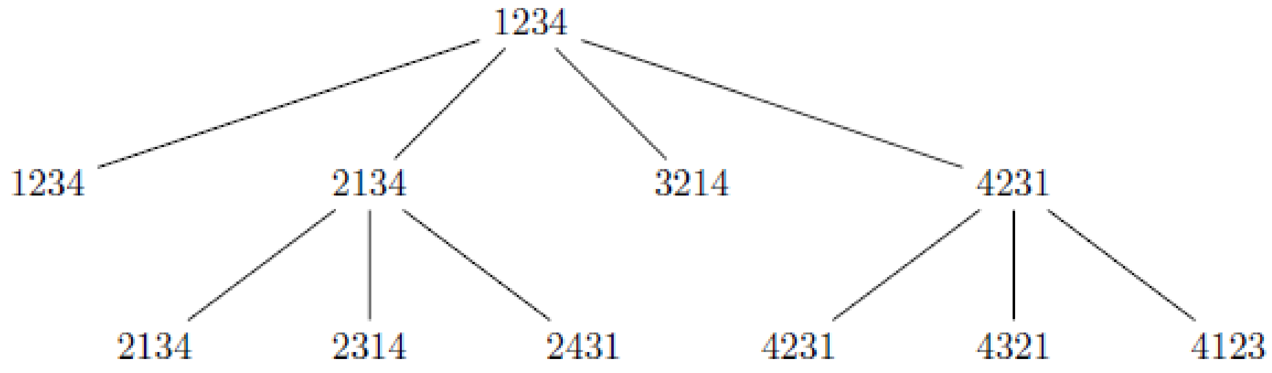


To systematically print out all the permutations, we first swap the first element with each other element

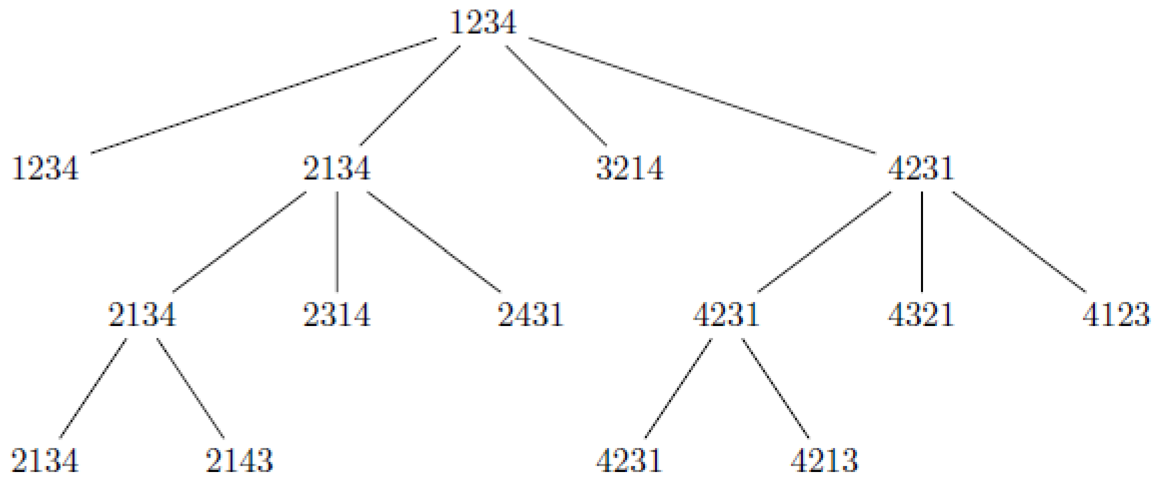


Next we need to swap the second element with each other element (except the element in the first position). Here we only case the second and the fourth cases.





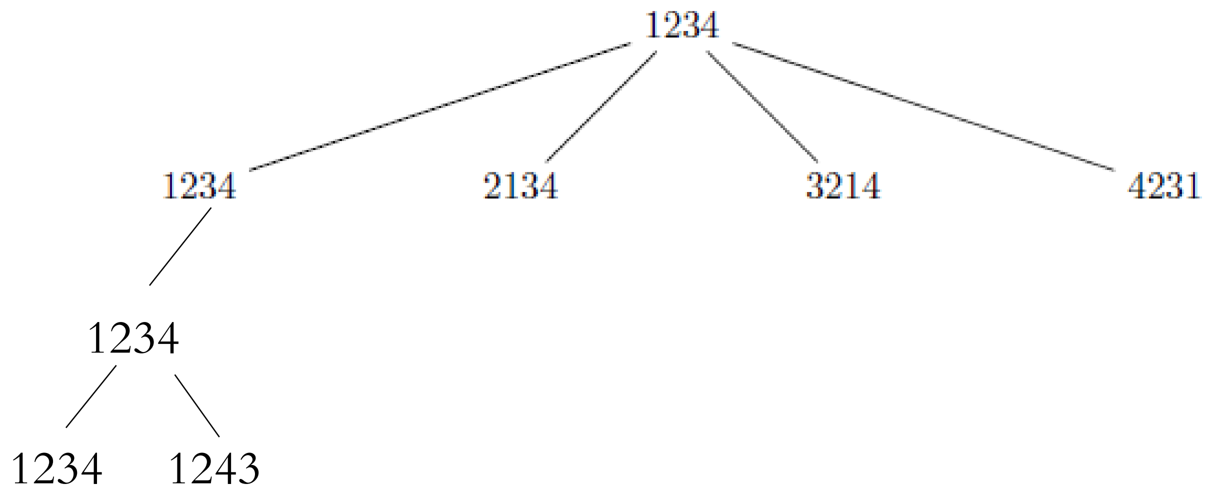
Next we need to swap the third element with the following element (in the example, we only leave the last element to swap).



To print out all the permutation we need to iteratively swap one element with each other element and recursively do so on its smaller sequence (reduce by one element) until we reach the last element.

Algorithm 2 Backtracking algorithm for Permutation

function PERMUTATION($char[]seq$, $sInx$, $eIdx$)
    if $sInx == eIdx$ **then**
        print seq
    else
        for $i \leftarrow$ sInx to eInx **do**
            swap the sInxth character and the i^{th} character in seq
            Permutation(seq,sInx+1,eIdx)
            swap the sInxth character and the i^{th} character in seq
        end for
    end if
end function



Q3

Find length of longest substring of a given string of digits, such that sum of digits in the first half and second half of the substring is same. For example, if the input string is “142124”, the whole string is the answer. The sum of the first 3 digits = the sum of the last 3 digits (1+4+2 = 1+2+4). Thus, the length is 6. If the input is “12345678”, then the output is 0. If the input is “9430723”, then the output is 4 (4307).

A brute force approach

Example: 9430723

1st round: i=0, j=1, lSum=9, rSum=4, maxLen = 0

2nd round: i=0, j=2, lSum=9, rSum=7, maxLen = 0

……

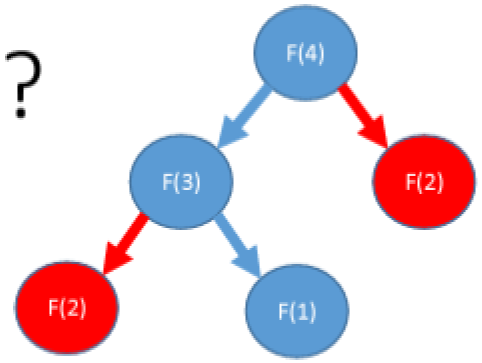
Algorithm 3 The Brute Force Solution

```
function MaxSubString(char[] seq)
    maxLen ← 0
    for i ← 0 to length of seq do
        for j ← i+1 to length of seq step 2 do
            len ← length of the substring between indices i and j
            if maxLen >= len then                    ▷ maxLen > length of substring, do nothing
                continue
            end if
            for k ← 0 to len/2 do
                lSum to sum of digits in the first half
                rSum to sum of digits in the second half
            end for
            if lSum == rSum then
                maxLen ← len
            end if
        end for
    end for
    return maxLen
end function
```

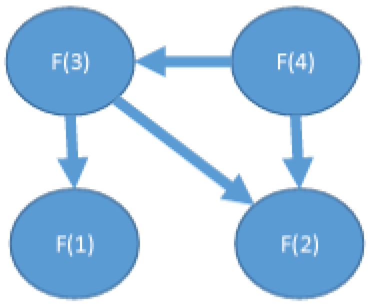
What is the time complexity?

$$\sum_{i=0}^{n} \frac{(n-i)^2}{2} \times 2 = O(n^3)$$

What is Dynamic Programming (DP)?



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



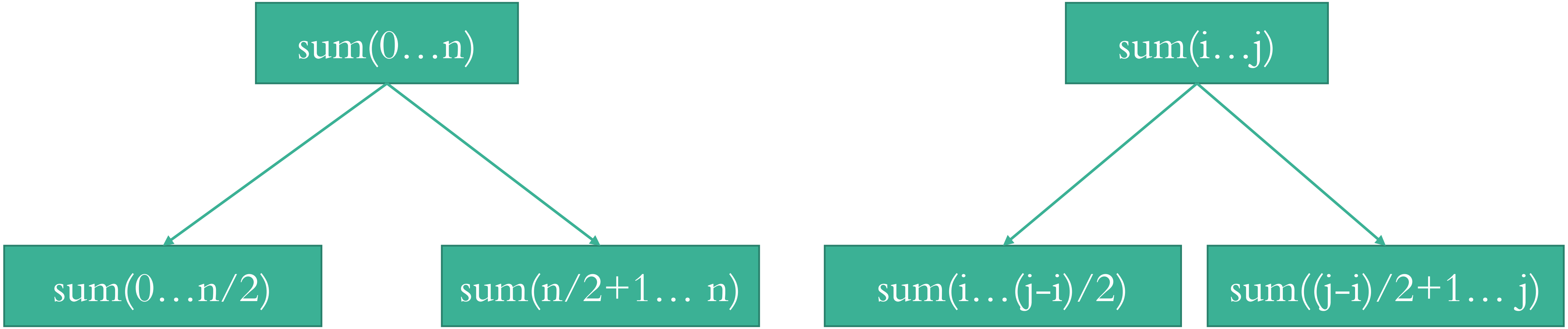
- Optimal substructure
  - Combination of optimal solutions to its sub-problems

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

➢ 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


sum(0…n)
sum(0…n/2)
sum(n/2+1… n)
sum(i…j)
sum(i…(j-i)/2)
sum((j-i)/2+1… j)

- Dynamic Programming Approach: Let sum[i][j] be the sum of digits from i to j and assume that the matrix has been initialized and the lower triangular of the matrix will not be used (when i > j)

$$sum[i][j] = sum[i][j-k]+sum[j-k+1][j], \text{ where } k = \text{floor}((j-i+1)/2)$$

- For 9430723

j/k

i		0	1	2	3	4	5	6
	0	9						
	1		4					
	2			3				
	3				0			
	4					7		
	5						2	
	6							3

j/k

i		0	1	2	3	4	5	6
	0	9						
	1		4					
	2			3				
	3				0			
	4					7		
	5						2	
	6							3

- Dynamic Programming Approach: Let sum[i][j] be the sum of digits from i to j and assume that the matrix has been initialized and the lower triangular of the matrix will not be used (when i > j)

$$sum[i][j] = sum[i][j-k]+sum[j-k+1][j], \text{ where } k = floor((j-i+1)/2)$$

- For 9430723

j/k

i

	0	1	2	3	4	5	6
0	9						
1		4					
2			3				
3				0			
4					7		
5						2	
6							3

j/k

i

	0	1	2	3	4	5	6
0	9	13	16	16	23	25	28
1		4	7	7	14	16	19
2			3	3	10	12	15
3				0	7	9	12
4					7	9	12
5						2	5
6							3

Algorithm 4 The DP Solution

```
function MaxSubStringDP(char[] seq)
    maxLen ← 0
    for len = 2 to n do
        for i = 0 to n − len + 1 do          ▷ pick i and j to make the length of substring be len
            j ← i + len − 1
            k ← ⌊len/2⌋
                                              ▷ calculate sum[i][j] from table
            if len mod 2 == 0 and sum[i][j − k] == sum[j − k + 1][j] and len > maxLen then
                maxLen ← len                  ▷ Update maxLen
            end if
        end for
    end for
    return maxLen
end function
```

What is the time complexity?

$$\sum_{len=2}^{n} n - len + 1 = O(n^2)$$

Additional space required $O(n^2)$