## Combination Sum (cont).

```
    backtrack(remaining - candidates[i], ...)
    combination.pop()

backtrack(target, [], 0)
return result
```

## Coin change 2 (nbr. of ways)

```
def change(amount, coins):
    dp = [0]*(amount+1)
    dp[0] = 1
    for coin in coins:
        for i in range(coin, amount+1):
            dp[i] += dp[i-coin]
    return dp[amount]
```

## Combination sum IV (coin change 2 but order is important)

```
def combinationSum4(nums, target):
    dp = [0]*(target+1)
    dp[0] = 1
    for i in range(1, target+1):
        for num in nums:
            if i-num >= 0:
                dp[i] += dp[i-num]
    return dp[target]
```

## House Robber

```
def rob(nums):
    if not nums: return 0
    if len(nums)==1: return nums[0]
    prev1, prev2 = 0,0
    for num in nums:
        temp = prev1
        prev1 = max(prev2+num, prev1)
        prev2 = temp
    return prev1
```

## Decode ways (decode string of nbrs)

```
def numDecodings(s):
    if not s or s[0] == '0':
        return 0
    dp = [0]*(len(s)+1)
    dp[0], dp[1] = 1, 1
    for i in range(2, len(s)+1):
        if s[i-1] != '0':
            dp[i] += dp[i-1]
        two_digit = int(s[i-2:i])
        if 10 <= two_digit <= 26:
            dp[i] += dp[i-2]
    return dp[len(s)]
```

## Jump Game

```
def canJump(nums):
    cur, goal = 0,0
    n = len(nums)
    for i in range(n):
        if i > goal:
            return False
        goal = max(goal, nums[i]+i)
    if goal >= n-1:
        return True
    return False
```

## Burst balloons (burst $i^{th}$ balloon get nums[i-1]*nums[i]*nums[i+1] coins)

```
def maxCoins(nums):
    nums = [1] + nums + [1]
    n = len(nums)
    dp = [[0]*n for _ in range(n)]
    for length in range(2, n):
        for left in range(0, n-length):
            right = left + length
            for k in range(left+1, right):
                dp[left][right] = max(dp[
                    left][right], nums[left]*
                    nums[k]*nums[right] +
                    dp[left][k] + dp[k][right])
    return dp[0][n-1]
```

## Matrix chain multiplication

```
def chainMultip(p):
    n = len(p) - 1  # nbr. of matrices
    dp = [[0 for _ in range(n)] for _ in
          range(n)]
    for length in range(2, n+1):
        for i in range(n-length+1):
            j = i + length - 1
            dp[i][j] = float('inf')
            for k in range(i, j):
                cost = dp[i][k] + dp[k+1][j] +
                       p[i] * p[k+1] * p[j+1]
                dp[i][j] = min(dp[i][j], cost)
    return dp[0][n-1]
```

## Graph (GRAPH)

### clone graph

```
def clone_graph(node):
    if not node:
        return None
    visited = {}
    def dfs(node):
        if node in visited:
            return visited[node]
        clone = Node(node.val)
        visited[node] = clone
        for neigh in node.neighbors:
            clone.neighbors.append(dfs(n-
                eighbor))
        return clone
    return dfs(node)
```

### course schedule (can finish all courses?)

```
def canFinish(numCourses, prereqs):
    graph = defaultdict(list)
    in_degree = [0] * numCourses
    for course, prereq in prereqs:
        graph[prereq].append(course)
        in_degree[course] += 1
    queue = deque([i for i in range(
        numCourses) if in_degree[i]==0])
    count=0
    while queue:
        current = queue.popleft()
        count += 1
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)
    return count == numCourses
```

### Pacific Atlantic Waterflow

```
def pacificAtlantic(heights):
    if not heights:
        return []
    m, n = len(heights), len(heights[0])
    pacific_reach = [[False]*n for _ in range(m)]
    atlantic_reach=[[False]* n for _ in range(m)]
    def dfs(row, col, reachable):
        reachable[row][col] = True
        dir = [(0,1), (0,-1), (1,0), (-1,0)]
        for dr, dc in dir:
            nr, nc = row+dr, col+dc
            if (0 <= nr < m and 0 <= nc < n
                and not reachable[nr][nc] and
                heights[nr][nc] >= heights[r][c]):
                dfs(nr, nc, reachable)
    for i in range(m):
        dfs(i, 0, pacific_reach)  # left edge (pacific)
        dfs(i, n-1, atlantic_reach)  # right (atlantic)
    for j in range(n):
        dfs(0, j, pacific_reach)
        dfs(m-1, j, atlantic_reach)
    result = []
    for i in range(m):
        for j in range(n):
            if pacific_reach[i][j] & atlantic_
                reach[i][j]:
                result.append([i,j])
    return result
```

### Nbr. of Islands (grids of 1's)

```
def numIslands(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    num_islands = 0
    def dfs(row, col):
        if row<0 or row>=m or col<0 or
            col>=n or grid[row][col]=='0':
            return
        grid[row][col] = '0'
        dfs(row+1, col)
        dfs(row-1, col)
        dfs(row, col+1)
        dfs(row, col-1)
    for i in range(m):
        for j in range(n):
            if grid[i][j] == '1':
                num_islands += 1
                dfs(i,j)
    return num_islands
```