Name : Arnav Manish Borse
Register number : 22BCE1505

PPS 5

1) Problem is: A person is climbing a staircase, it takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many ways you can climb to the top.

pseudocode:

```
function countways(n):
    if n==0 or n==1:
        return 1

    dp = [0]*(n+1)
    dp[0] = 1          // Base case stay on the ground do nothing
    dp[1] = 1          // to reach first step take one single step

    for i from 2 to n:
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

dp[i] represents number of ways to reach top i th step

3.

pseudocode

```
function memfib (n)
    if (n <= 0)
        return 0
    if (n == 1)
        return 0
    if (n == 2)
        return 1
    if (n == 3)
        return 2
    if memo[n] != NULL
        return memo[n]

// we initialize memo array with size n+1

return fib(n)
```

4) Java pseudocode (Brute force)

```
function rodcut (lengths, prices, n) {
    if n == 0
        return 0
    maxrevenue = INTMIN  <-- minimum possible value
    for i from 1 to n:
        revenue = prices[i] + rodcut (length, price, n-i)
        maxrevenue = max (maxrevenue, revenue)
    return maxrevenue
```

$T = O(2^n)$

5) top-down approach i.e by using dynamic programming approach

```
function rodcut (lengths, prices, n)
    memo = array of size n+1 which is initialized with -1

    function helper (lengths, prices, n):
        if n == 0
            return 0
        if memo[n] != -1:
            return memo[n]
        max revenue = INT_MIN
        for i from 1 to n:
            revenue = prices[i] + helper(lengths, prices, n-i)
            maxrevenue = max(max revenue, revenue)
        memo[n] = max revenue  // memo update
        return maxrevenue

    return helper(lengths, prices, n)
```

$O(n^2)$

6) botton up approach to solve ~~the so the~~ rodcutting problem

```
function rodcut (length, prices, n)
    revenue = array of size n+1 which is initialized with 0

    for i from 1 to n:
        max revenue = INT_MIN
        for j from 1 to i:
            maxrevenue = max(maxrevenue, prices[j] + revenue[i-j])
```

revenue [i] = max revenue

return revenue [n]

7)

pseudocode

function rodcut (lengths, prices, n)
memo_revenue = array of size n+1 initialized with -1
memo_cuts = array of size n+1 initialized with -1

function helper (length):
if length == 0:
return 0
if memo_revenue [length] != -1
return memo_revenue [length]
max_revenue = INT_MIN → (minimum no)
best_cut = -1
for i from 1 to length
revenue = prices[i] + helper (length - i)
if revenue > max_revenue:
max_revenue = revenue
best_cut = i
memo_revenue [length] = max_revenue
memo_cuts [length] = best_cut
return max_revenue

also return or print the memo_cut [length]
which has all small pieces.

lengths = [ ]

8)

```
function modified rodcutting (n, prices)
    memo = array of size n+1 initialized with -1
    memo[0] = 0

    for i from 1 to n
        max revenue = -1
        for cut edge in [3,5]
            if i - cut length >= 0
                revenue = prices[cut edge] + memo[i - cut length]
                if revenue > max revenue
                    max revenue = revenue
        if max revenue == -1 ; // if no valid cut is possible
            memo[i] = -1          // we marked it as impossible for cut
        else
            memo[i] = max revenue

    return memo[n]
```

9)

```
function max revenue k pieces (n, price, k)
    memo = matrix of size (n+1) x (k+1) initialize with -1

    function helper (length, pieces):
        if pieces == 1
            return memo[length][pieces] = max(price[length], memo[length][pieces])

        if memo[length][pieces] != -1:
// if already calculated     return memo[length][pieces]
```

```
maxrevenue = -1
for i from 1 to length
    revenue = max(prices[i], helper(length - i, pieces - 1));
    maxrevenue = max(maxrevenue, revenue)
memo[length][pieces] = maxrevenue
return maxrevenue

// in the main function
for i from 1 to n :
    // base case selling 1 piece of length n
    memo[i][1] = prices[i]

for j from 2 to k :
    helper(n, j).

return memo[n][k]
```

10) Example fibonacci series

divide and conquer algorithm

```
function fibo(n)
    if n == 0
        return 0
    if n == 1
        return 1
    return fibo(n-1) + fibo(n-2)
```

dynamic programming approach

```
function fibo(n)
    if (n <= 1)
        return n
    arr = array of size
    n+1, initialized with
    -1
```

dynamic programming

```
function fibon(n)
    function fib(arr, n):
        if(n <= 1):
            return n
        if arr[n] == -1
            arr[n] = fib(arr, n-1) + fib(arr, n-2)
    return arr[n]
```

PPS 6

(1)

```
function activity selection last start(activities)
    activities.sort(reverse = True) (in decreasing order)
    selected_activities = []

    for activity in activities
        if activity.start <= activity.end
            selected_activities.append(activity)
            prev_activity_end = activity.end

    return selected_activities
```

2)

Act
function Activity selection (S)

sort activities by finish time in ascending order
initialize empty set A

for activity a in S:
    if a.start time >= last finish time

        Add a to set A
        last finish time = a.finish time

return A
    this will give the required output

3)

ExtActSel (S, L)

Sort activities in S based on their finish time in increasing order
S.sort (x: x.finish time)
H=[] array of size m to store lecture hall

Availability = [0] // initialized to 0
for i, activity in S:
    for j (hall) in L
        if Availability[j] <= activity.finish time
            H[i] = j
            Availability[j] = activity.finish time

break

return H // it returns the array H containing lecture hall assignments for each activity.

4] Activity selection dynamic programming approach:

```
func: RecursiveActivitySelection(activities, i):
    if i == 0:
        return 1        # // Base case : only one activity can be selected
    maxActivities = 1   # // initializing with current activity itself
    # consider all activities from 0 to i-1
    for j 0 to i-1
        if (activities[j] do not overlap with activities[0])
            numActivities = 1 + RecursiveActivitySelection(activities, j)
            maxActivities = max(maxActivities, numActivities)

    return maxActivities

// we call the function
print max non overlapping activities → maxActivities
```

5)

```
def greedy_knap(items, W)
    calculate value-to-weight ratio
        or profit
    sort(items, reverse=True)  # decreasing order sort

    current_weight = 0
    current_value = 0

    for i in range(len(items)):
        if current_weight + items[i].weight <= W:
            current_weight += items[i].weight
            current_value += items[i].value

    return current_value
```

DA questions given in class

1) Question was asked:

(Question is) You are given an array of n elements you need to find the maximum sum of non adjacent elements. The challenge is that the time complexity of your algorithm should be O(n). The numbers can be any ~~rational~~ integer.

example input arr = [3, 2, 7, 10]
output 13 (sum of 3 and 10)

3)

Pseudocode:

```
class point

    function a(self, x, y)
        self.x = x
        self.y = y

function orientation(p, q, r):
    val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)
    if val == 0:
        return 0
    return 1
    if val > 0
        return 1
    else
        return 2

function intersect(p1, q1, p2, q2):
    o1 = orientation(p1, q1, p2)
    o2 = orientation(p1, q1, q2)
    o3 = orientation(p2, q2, p1)
    o4 = orientation(p2, q2, q1)

    if o1 != o2 && o3 != o4:
        return true
    if o1 == 0 && onsegment(p1, p2, q1):
        return true
    if o2 == 0 && onsegment(p1, q2, q1):
        return true
    if
```

if o3 == 0 && onsegment(p2, p1, q2):
    return true
if o4 == 0 && onsegment(p2, q1, q2):
    return true

return false

function count intersecting segments(segments) {
    endpoints = [] // array

    for segment in segments:
        endpoints.append(segment[0]) // left endpoint
        endpoints.append(segment[1]) // right endpoint

    endpoints.sort(x)
    intersections = 0
    // active_set is defined

    for point in endpoints
        if point in active set:
            active_set.remove(point)
        else
            ~~for i = 0; i < (active_set); i++) {~~
            for (active segment) in active set
                if intersect(point, point, ~~active~~ i, active_segment[0], active_segment[1])
                    intersections += 1
            active_set.add(point)
    return intersections