Subject: Dynamic Programming

Sat Sun Mon Tue Wed Thu Fri

Date : / /

Fibonacchi Top-down (Memoized)

```
Algorithm Fib(n):
    for i=0 to n:
        F[i] = -∞
    Fib_rec(n)

Algorithm Fib_rec(n):
    if F[n] ≥ 0 then
        return F[n]
    if n ≤ 1 then
        F[n] = n
    else
        F[n] = Fib_rec(n-1) + Fib_rec(n-2)
    return F[n]
```

Time complexity $O(n)$

Non-memoized Top-down (Brute force)

```
Algorithm Fibo(n):
    if n ≤ 1 then
        return n
    else
        return Fibo(n-1) + Fibo(n-2)
```

Subject

Sat Sun Mon Tue Wed Thu Fri

Date : / /

Fibonacchi Bottom-up Algorithm

```
Algorithm fib_bu(n):
    for i=0 to n:
        if i<= 1 then
            F[i] = i
        else
            F[i] = F[i-1] + F[i-2]
    return F[n]
```

Time complexity O(n)

C_r^n Combination Top-down non memoized Algorithm

```
Algorithm comb(n, r):
    if (r == 1)   // base case 1
        return n
    else if r=n or r=0.  // base case 2
        return 1
    else
        return comb(n-1, r-1) + comb(n-1, r)
```

Date : / /

C_r^n combination Top-down Memoized Algorithm (2DDP)

~~for i=0 to~~

```
Algorithm comb (n,r):
    for i=0 to n
        for j=0 to min (i, r)
            c[i, j] = -∞
    return rec (n, r)
```

~~Algorithm rec (n,r):~~

```
Algorithm rec (n, r):
    if c[n, r] != -∞ then
        return c[n,r]
    if r = 1 then  // base case 1
        c[n, r] = n
    else if r=n or r=0 then // base case 2
        c[n, r] = 1
    else
        c[n, r] = rec(n-1, r-1) + rec(n-1, r)
    return c[n, r]
```

Time complexity : $O(nr)$

Subject

Sat Sun Mon Tue Wed Thu Fri

Date : / /

Bottom-up Algorithm (DP) C_r^n: combination

Algorithm combination (n, r):

```
for i=0 to n
    for j=0 to min(i, r)
        if j = 1 then
            c[i, j] = i
        else if j = i or j = 0 then
            c[i, j] = 1
        else
            c[i, j] = c[i-1, j-1] + c[i-1, j]
    return c[n, r]
```

Time complexity $o(n\,r)$

Weighted Interval Scheduling (Naive (slow) Implementation)

Implementing the recurrence directly:

```
Weighted Int-Sched(j):
    If j=0:
        Return 0
    Else
        return max(w[j] + weighted Int sched(p[j]),
                   weighted Int-Sched(j-1))
```

Time complexity: exponential time.

Subject

Sat Sun Mon Tue Wed Thu Fri

Date : / /

Weighted Interval Scheduling Top-down (Memoized)

```
MemoizedIntSched(j):
    If j=0 : Return 0
    Else If M[j] is not empty:
        Return M[j]
    Else
        M[j] = max(W[j]+MemoizedIntSched(P[j]),
                   MemoizedIntSched(j-1))
    Return M[j]
```

Time complexity O(n)

Weighted Interval Scheduling Bottom-up

```
ForwardIntSched(j):
    M[0] = 0
    for j=1, ....., n:
        M[j] = max(W[j]+M[P(j)], M[j-1])
```

Subject _____ Date : / /

Greedy Algorithms

Activity Scheduling (Greedy Activity)

```
func GreedyActivity(S[1, 2, ....., n])
    n = length(S)
    A = {a_1}        // since S is sorted by finish time, first element of S would have the smallest finish time.
    k = 1            // Index of current activity
    for m in 2 to n:
        if S[m] >= f[k]:
            A = A U {a_m}
            k = m
    return length(A)
```

Subject

Date : / /

Optimal Merge Pseudo code (Min Heap)

```
func OptimalMerge (S[1, ..... ,n])
    H ← MinHeap
    H ← insert (S)
    total_cost = 0
    while len(H) > 1:
        f ← extract (H)
        s ← extract (H)
        merge = f + s
        total_cost += merge
        insert (H, merge)
    Return total_cost
```

$$\text{Total work} = \underset{\text{size}}{\text{depth} \times \text{node}} + \cdots + \underset{\text{size}}{\text{depth} \times \text{node}}$$

$$= \sum_{i \in S} d_i * x_i$$

depth size