IIT Madras
BSc Degree

Home Week-8 Week-10

PDSA - Week 9

Dynamic programming

- Solution to original problem can be derived by combining solutions to subproblems

  Examples: Factorial, Insertion sort, Fibonacci series
- Anticipate the structure of subproblems
- Derive from inductive definition
- Solve subproblems in topological order

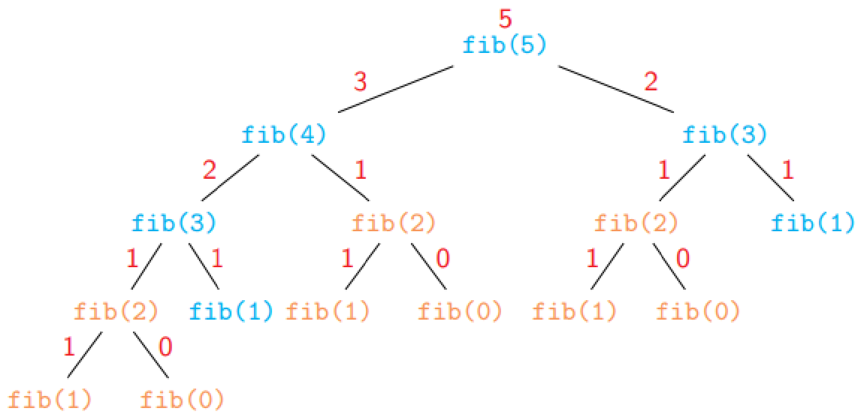
Memoization

- Inductive solution generates same subproblem at different stages
- Naïve recursive implementation evaluates each instance of subproblem from scratch
- Build a table of values already computed – Memory table
- Store each newly computed value in a table
- Look up the table before making a recursive call

Example of n^{th} number in Fibonacci series:-

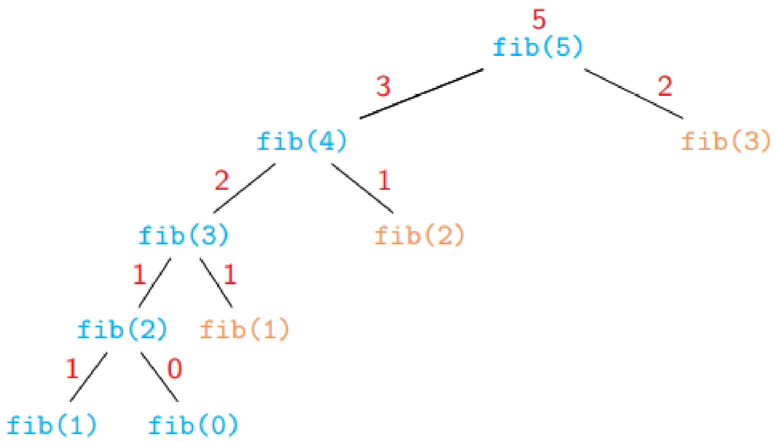
Simple recursive

```
def fibrec(n):
    if n <= 1:
        return n
    return fibrec(n - 1) + fibrec(n - 2)
```



Memoization

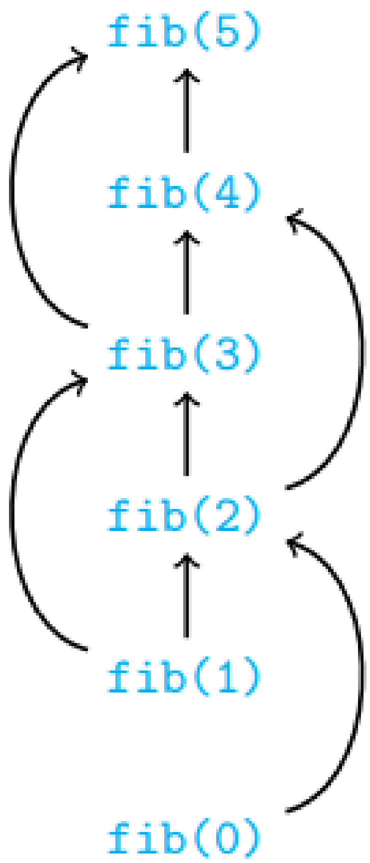
```
memo ={}
def fib(n):
    if n <= 1:
        memo[n] = n
    if n not in memo:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```



k	1	0	2	3	4	5
fib(k)	1	0	1	2	3	5

Dynamic programming

```
def fib(n):
    T = [0] * (n + 1)
    T[1] = 1
    for i in range(2, n + 1):
        T[i] = T[i - 1] + T[i - 2]
    return T[n]
```



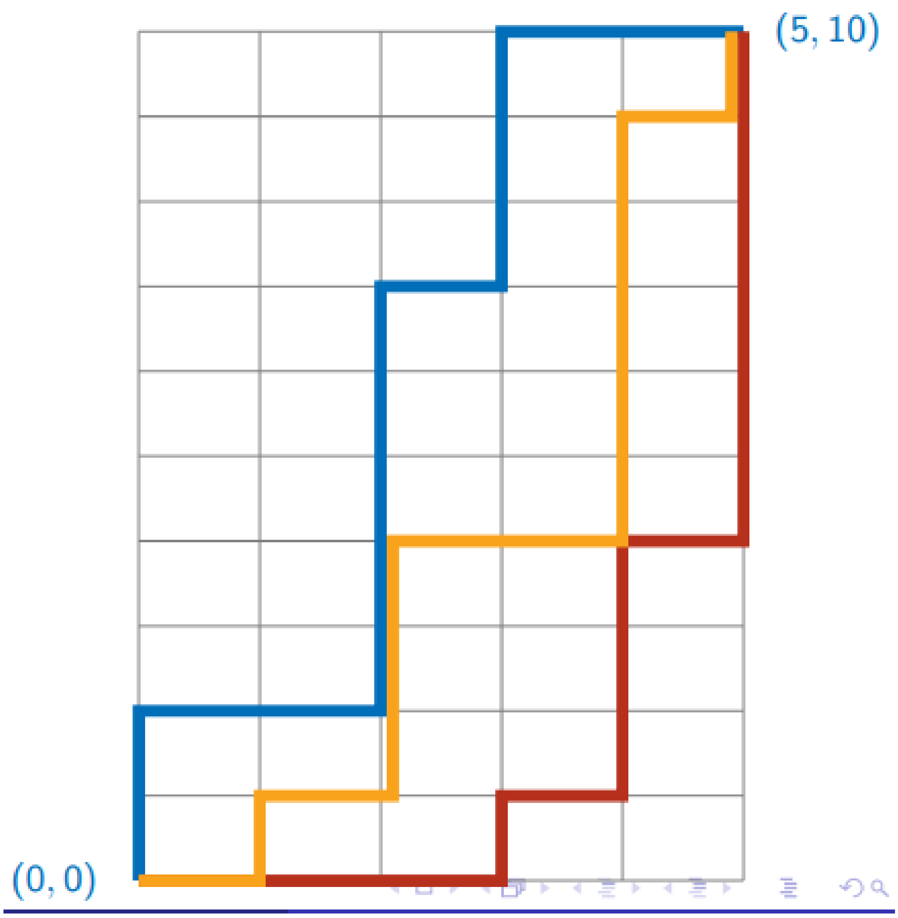
Comparison

```
#simple recursive
def fibrec(n):
    if n <= 1:
        return n
    return fibrec(n - 1) + fibrec(n - 2)

# memoization topdown
memo ={}
def fibmem(n):
    if n <= 1:
        memo[n] = n
    if n not in memo:
        memo[n] = fibmem(n-1) + fibmem(n-2)
    return memo[n]

# DP tabulation bottom up
def fibtab(n):
    T = [0] * (n + 1)
    T[1] = 1
    for i in range(2, n + 1):
        T[i] = T[i - 1] + T[i - 2]
    return T[n]

```

```

n=int(input())
import time
t1 = time.perf_counter()
res1 = fibrec(n)
ft1 = time.perf_counter() - t1

t1 = time.perf_counter()
res2 = fibmem(n)
ft2 = time.perf_counter() - t1

t1 = time.perf_counter()
res3 = fibtab(n)
ft3 = time.perf_counter() - t1

print(res1,ft1)
print(res2,ft2)
print(res3,ft3)
```

Dynamic programming problems

Grid paths



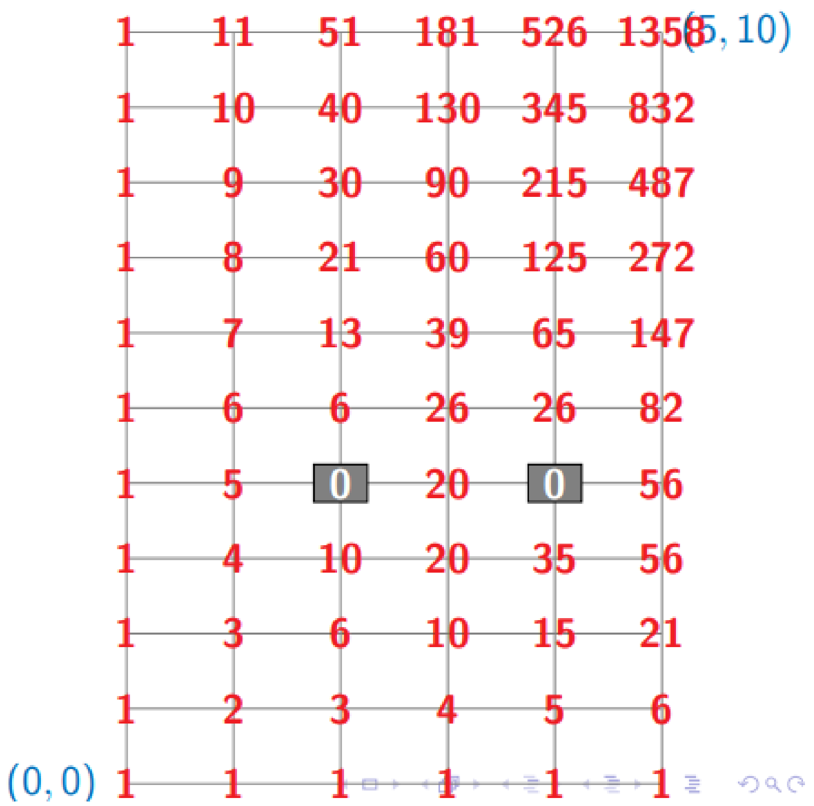
- Rectangular grid of one-way roads
- Can only go up and right

- How many paths from (0, 0) to (m, n)?
- Every path has $(m+n)$ segments
- What if an intersection is blocked?
- Need to discard paths passing through blocked intersection
- Inductive structure

  - How can a path reach (i, j)
    - Move up from $(i, j-1)$
    - Move right from $(i-1, j)$
  - Each path to these neighbours extends to a unique path to (i, j)
  - Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
    - $P(i, j) = P(i-1, j) + P(i, j-1)$
    - $P(0, 0) = 1$ — base case
    - $P(i, 0) = P(i-1, 0)$ — bottom row
    - $P(0, j) = P(0, j-1)$ — left column
  - $P(i, j) = 0$ if there is a hole at (i, j)

$$\begin{tikzcd} (i-1,j) \arrow[r] & (i,j) \\ & (i,j-1) \arrow[u] \end{tikzcd}$$

- Fill the grid by row, column or diagonal

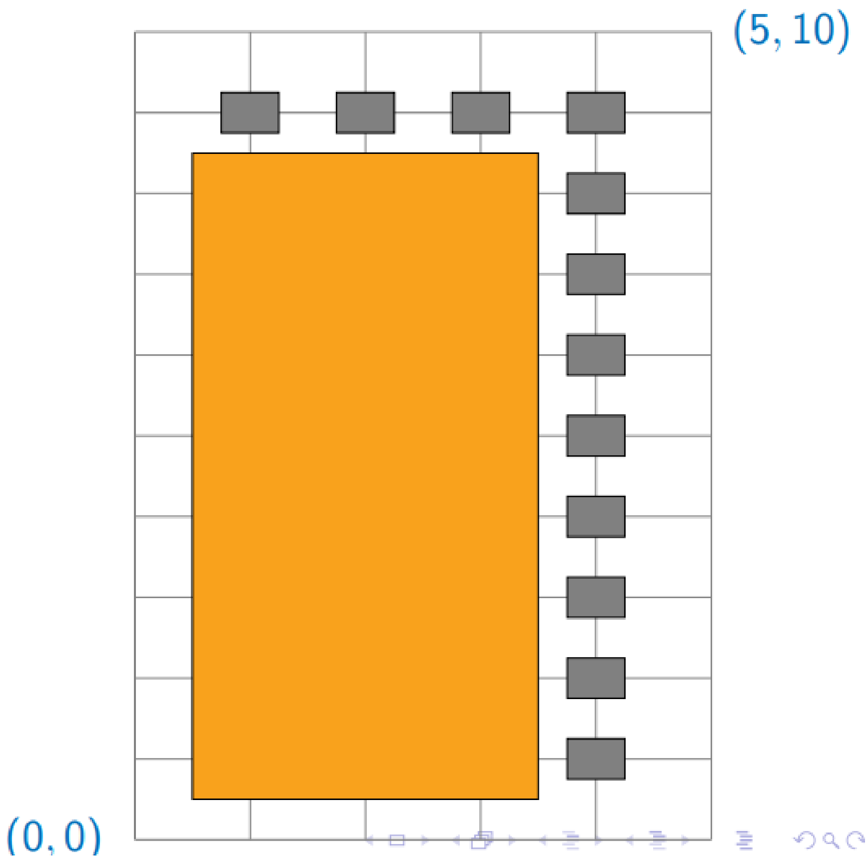


- Complexity is $O(mn)$ using dynamic programming, $O(m+n)$ using memorization

Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region
- Memo table has O(m + n) entries

- Dynamic programming blindly fills all mn cells of the table
- Tradeoff between recursion and iteration
    - “Wasteful” dynamic programming still better in general



Longest Common Sub Word (LCW)

- Given two strings, find the (length of the) longest common sub word
- Subproblems are LCW(i, j), for $0 \le i \le m$, $0 \le j \le n$
- Table of $m + 1$ $n + 1$ values
- Inductive structure

$$LCW[i,j] = \begin{cases} 1 + LCW[i+1, j+1], & if \;\; a_i = b_j \\ 0, & if \;\; a_i \neq b_j \end{cases}$$

- Start at bottom right and fill row by row or column by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Implementation

```python
def LCW(s1,s2):
    import numpy as np
    (m,n) = (len(s1),len(s2))
    lcw = np.zeros((m+1,n+1))
    maxw = 0
    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if s1[r] == s2[c]:
                lcw[r,c] = 1 + lcw[r+1,c+1]
            else:
                lcw[r,c] = 0
            if lcw[r,c] > maxw:
                maxw = lcw[r,c]
    return maxw
s1 = 'bisect'
s2 = 'secret'
print(LCW(s1,s2))
```

Output

```
3.0
```

Complexity

$O(mn)$

Longest Common Sub Sequence (LCS)

- Subsequence – can drop some letters in between
- Subproblems are LCS(i, j), for $0 \le i \le m$, $0 \le j \le n$
- Table of $m + 1$ $n + 1$ values
- Inductive structure

$$LCS[i,j] = \begin{cases} 1 + LCS[i+1,j+1], & if\ \ a_i = b_j \\ max(LCS[i+1,j], lcs[i,j+1]), & if\ \ a_i \neq b_j \end{cases}$$

- Start at bottom right and fill row by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Implementation

```
def LCS(s1,s2):
    import numpy as np
    (m,n) = (len(s1),len(s2))
    lcs = np.zeros((m+1,n+1))
    for c in range(n-1,-1,-1):
        for r in range(m-1,-1,-1):
            if s1[r] == s2[c]:
                lcs[r,c] = 1 + lcs[r+1,c+1]
            else:
                lcs[r,c] = max(lcs[r+1,c], lcs[r,c+1])
    return lcs[0,0]
s1 = 'secret'
s2 = 'bisect'
print(LCS(s1,s2))
```

Output

```
4.0
```

Complexity

$O(mn)$

Edit distance

- Minimum number of editing operations needed to transform one document to the other
- Subproblems are ED(i, j), for 0 ≤ i ≤ m, 0 ≤ j ≤ n
- Table of m + 1 n + 1 values ▪
- Inductive structure

$$ED[i,j] = \begin{cases} ED[i+1,j+1], & if \;\; a_i = b_j \\ 1 + min(ED[i+1,j+1], ED[i+1,j], ED[i,j+1]), & if \;\; a_i \neq b_j \end{cases}$$

- Start at bottom right and fill row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	●
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	●	6	5	4	3	2	1	0

Implementation

```
def ED(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    ed = np.zeros((m+1,n+1))
    for i in range(m-1,-1,-1):
```

```
        ed[i,n] = m-i
    for j in range(n-1,-1,-1):
        ed[m,j] = n-j
    for j in range(n-1,-1,-1):
        for i in range(m-1,-1,-1):
            if u[i] == v[j]:
                ed[i,j] = ed[i+1,j+1]
            else:
                ed[i,j] = 1 + min(ed[i+1,j+1], ed[i,j+1], ed[i+1,j])
    return(ed[0,0])
print(ED('bisect','secret'))
```

Output

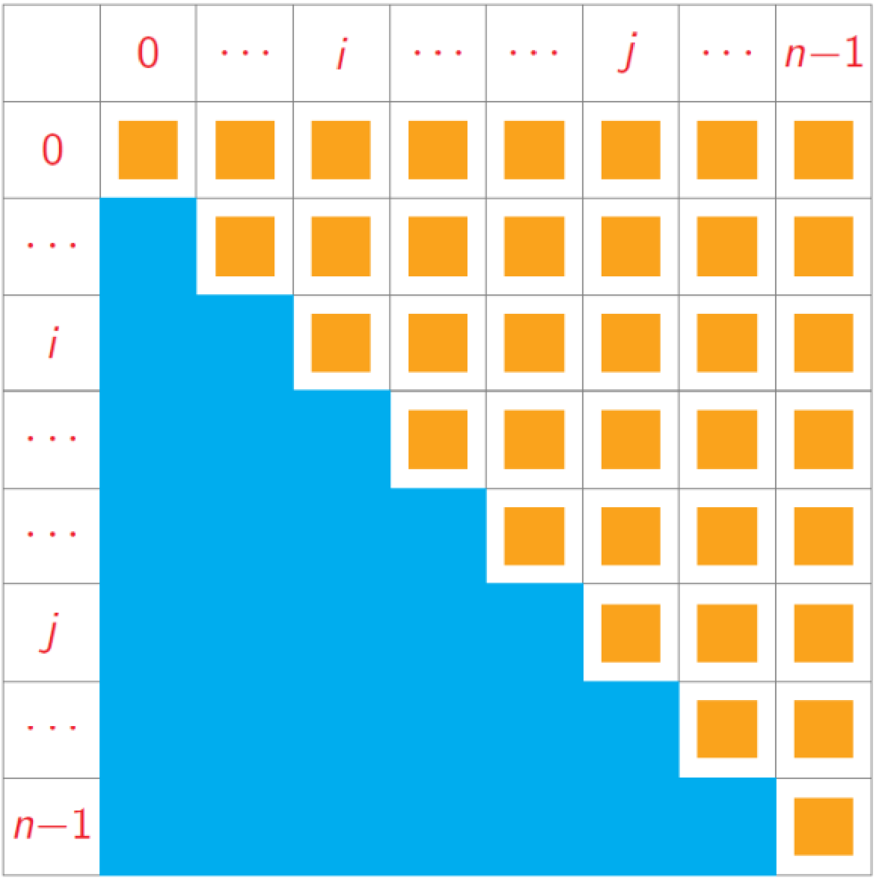
```
4.0
```

Complexity

$O(mn)$

Matrix multiplication

- Matrix multiplication is associative
- Bracketing does not change answer but can affect the complexity
- Find an optimal order to compute the product
- Compute C (i, j), $0 \le i,\ j < n$, only for $i \le j$
- C (i, j), depends on C (i, k – 1) , C(k, j) for every $i < k \le j$
- Diagonal entries are base case, fill matrix from main diagonal



Implementation

```
def MM(dim):
    n = dim.shape[0]
    C = np.zeros((n,n))
    for i in range(n):
        C[i,i] = 0
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            C[i,j] = C[i,i] + C[i+1,j] + dim[i][0] * dim[i+1][0] * dim[j][1]
            print(C)
            for k in range(i+1, j+1):
                C[i,j] = min(C[i,j],C[i,k-1] + C[k,j] + dim[i][0] * dim[k][0] * dim[j][1])
            print(C)
    return(C[0,n-1])
import numpy as np
a = np.array([[2,3],[3,4],[4,5]])
print(MM(a))
```

Output

```
64
```

Complexity

$O(n^3)$

Other implementation

Inductive structure

$$C[i,j] = \begin{cases} 0, & if \ \ i = j \\ min[(C[i][k] + C[k+1][j] + dim[i][0] * dim[k][1] * dim[j][1]) for \ i \leq k < j], & if \ \ i < j \end{cases}$$

```python
def MM(dim):
    n = len(dim)
    C = []
    for i in range(n):
        L = []
        L=[0]*n
        C.append(L.copy())
    for diff in range(1,n):
        for i in range(0,n-diff):
            j = i + diff
            KL = []
            for k in range(i, j):
                KL.append(C[i][k] + C[k+1][j] + dim[i][0] * dim[k][1] * dim[j][1])
            C[i][j] = min(KL)
    return(C[0][n-1])
a = [[4,10],[10,3],[3,12],[12,20],[20,7]]
print(MM(a))
```

Output

```
1344
```

Complexity

$O(n^3)$

Example

For example, we have matrices {M0, M1, M2, M3, M4} and the dimensions list of the given matrices is `[[4,10],[10,3],[3,12],[12,20],[20,7]]`.

Matrix C : -

0	120	264	1080	1344
0	0	360	1320	1350
0	0	0	720	1140
0	0	0	0	1680
0	0	0	0	0

Here 1344 value is representing minimum number of multiplication steps.

We can identify the order of multiplication of matrix by storing the `k` value(value of `k` for which we get minimum steps) in matrix with steps.

Matrix C : -

0	120/0	264/0	1080/0	1344/1
0	0	360/1	1320/1	1350/1
0	0	0	720/2	1140/3
0	0	0	0	1680/2
0	0	0	0	0

So initially we have matrices {M0, M1, M2, M3, M4} and at a time 2 matrices we can multiply.

We will check the k value for `C[0][4]` which is 1, so we can parenthesize the order like `{(M0 M1)(M2 M3 M4)}` now we have to check the order in the second bracket matrix M2, M3, M4, so we will check the value `C[2][4]` which is 3 then we can parenthesize the order like `((M2 M3) M4)` So, the final order will be `(M0 M1)((M2 M3) M4)`.