Week 9 - Revision

Dynamic programming

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

  Examples: Factorial, Insertion sort, Fibonacci series

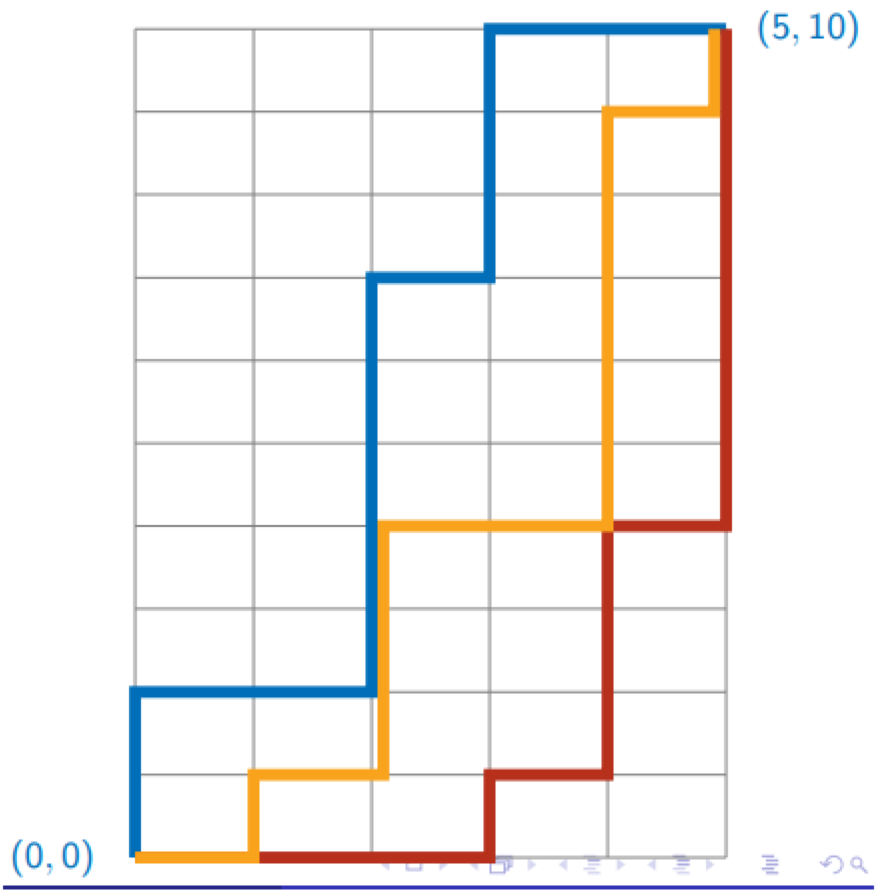
- Anticipate the structure of subproblems
- Derive from inductive definition
- Solve subproblems in topological order

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

Dynamic programming Example

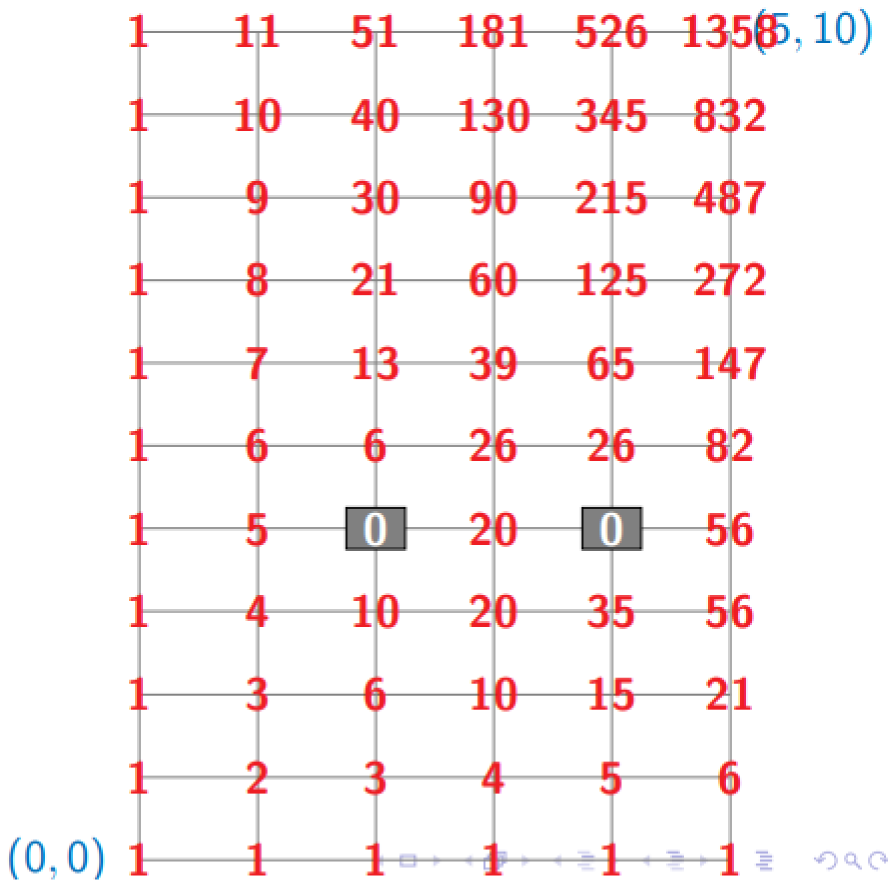
Grid paths



- 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

longest common sub word

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

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

- 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

- Complexity: $O(mn)$

Implementation

```
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 = input()
s2 = input()
print(LCW(s1,s2))
```

longest common sub sequence

- 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

- Complexity: $O(mn)$

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):
```

```python
        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 = input()
s2 = input()
print(LCS(s1,s2))
```

Edit distance

- Minimum number of editing operations needed to transform one document to the other
- Subproblems are ED(i, j), for $0 \le i \le m$, $0 \le j \le n$
- 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

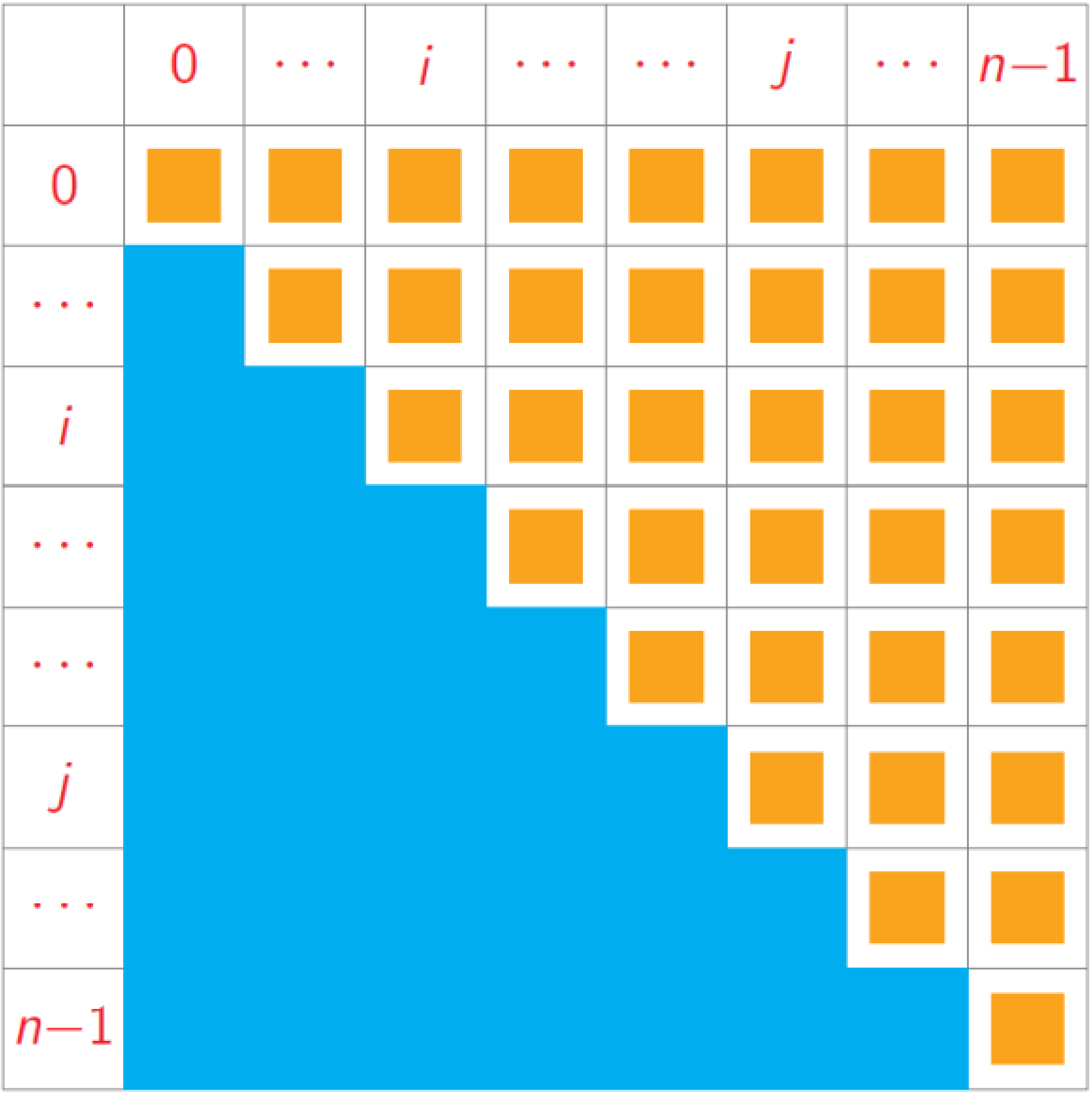
- Complexity: $O(mn)$

Implementation

```python
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])
```

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), $0 \leq i,\ j < n$, only for $i \leq j$
- C (i, j), depends on C (i, k − 1) , C(i, k) for every $i < k \leq j$
- Diagonal entries are base case, fill matrix from main diagonal



- Complexity: $O(n^3)$

Implementation

```
def C(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]
            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])
    return(C[0,n-1])
import numpy as np
a = np.array([[2,3],[3,4],[4,5]])
print(C(a))
```