Dynamic Programming

Inductive definitions ... Recursive programs

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$
- Insertion sort
  - $isort([]) = []$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, x_1, \ldots, x_{n-1}]), x_n)$

```
def fact(n):
  if n <= 0:
    return 1

  return n * fact(n - 1)
```

```
def isort(l):
  if l == []:
    return l

  return insert(isort(l[:-1]), l[-1])
```

Optimal substructure property

- Solution to original problem can be derived by combining solutions to subproblems
- $fact(n-1)$ is a subproblem of $fact(n)$
  - So are $fact(n-2), fact(n-3), \ldots, fact(0)$
- $isort([x_0, x_1, \ldots, x_{n-1}])$ is a subproblem of $isort([x_0, x_1, \ldots, x_n])$
  - So is $isort([x_i, \ldots, x_j])$ for any $0 \leq i < j \leq n$

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slot for instructor i starts at $s(i)$ and finishes at $f(i)$
- Slots may overlap, so not all bookings can be honoured

- Choose a subset of bookings to maximize the number of teachers who get to use the room

Subproblems

- Each subset of bookings is a subproblem

Generic greedy strategy

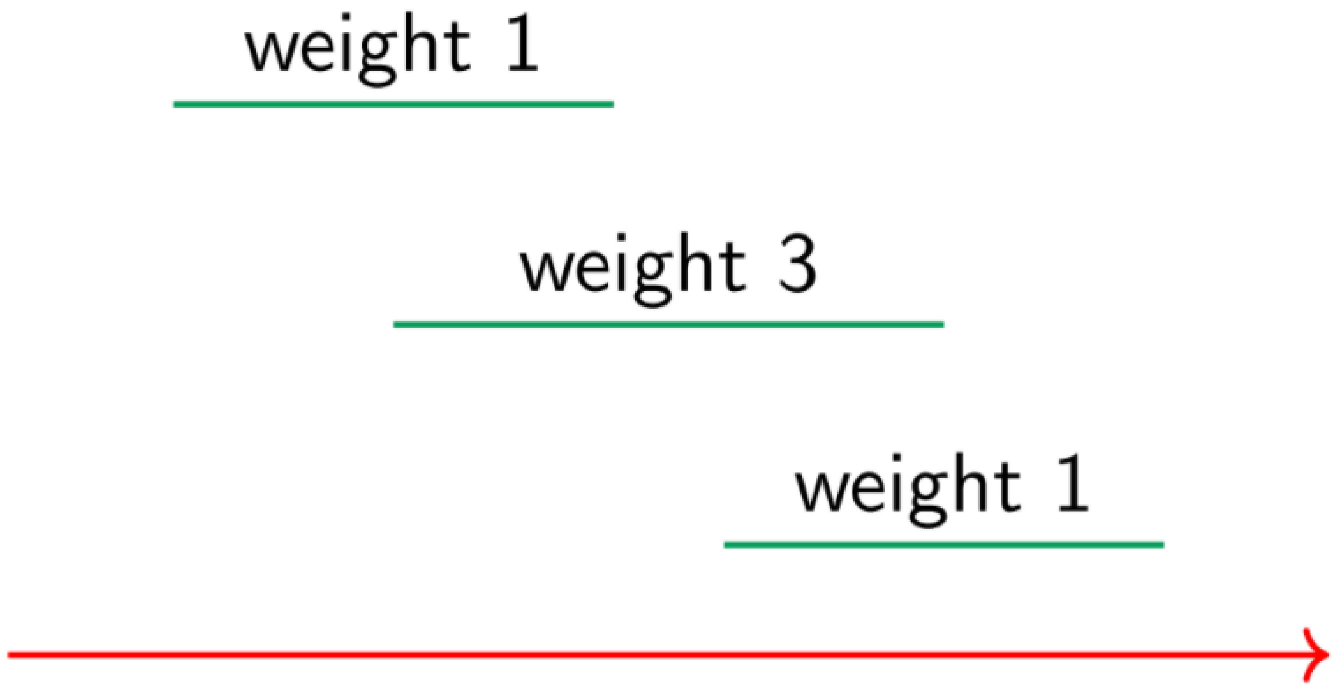
- Pick one request from those in contention
- Eliminate bookings in conflict with this choice
- Solve the resulting subproblem

Subproblems

- Each subset of bookings is a subproblem
- Given N bookings, 2^N subproblems
- Greedy strategy looks at only a small fraction of subproblems
    - Each choice rules out a large number of subproblems
    - Greedy strategy needs a proof of optimality

Weighted Interval Scheduling Problem

- Same scenario as before but each request comes with a weight
    - For instance, the room rent for using the resource
- Revised goal: maximizethe total weight of the bookings that are selected
    - Not the same as maximizing the number of bookings selected
- Greedy strategy for unweighted case
    - Select request with earliest finish time
- Not a valid strategy any more



- Search for another greedy strategy that works ...
- ... or look for an inclusive solution that is "obviously" correct

Weighted Interval Scheduling

- Order by bookings by starting time, $b_1, b_2, \ldots, b_n$
- Begin with b_1
    - Either b_1 is part of the optimal solution, or it is not
    - Include $b_1 \implies$ eliminate conflicting requests in $b_2, \ldots, b_n$ and solve resulting subproblem
    - Exclude $b_1 \implies$ solve subproblem $b_2, \ldots, b_n$
    - Evaluate both options, choose the maximum
- Inductive solution considers all options
    - For each b_j, optimal solution either has b_j or does not
    - For b_1, we have explicitly checked both cases
    - If b_2 is not in conflict with b_1, it will be considered in both subproblems with and without b_1
    - If b_2 is in conflict with b_1, it will be considered in the subproblem where b_1 is excluded

The challenge

- b_1 is in conflict with b_2, but both are compatible with $b_3, b_4, \ldots, b_n$
    - Choose $b_1 \implies$ subproblem $b_3, \ldots, b_n$

    - Exclude $b_1 \implies$ subproblem $b_2, \ldots b_n$
    - Next stage
        - Choose/exclude b_2
        - Both choices result in $b_3, \ldots, b_n$, same subproblem
- Inductive solution generates same subproblem at different stages
- Naive recursion implementation evaluates each instance of subproblem from scratch
- Can we avoid this wasteful recomputation
- Memoization and Dynamic Programming?

Memoization

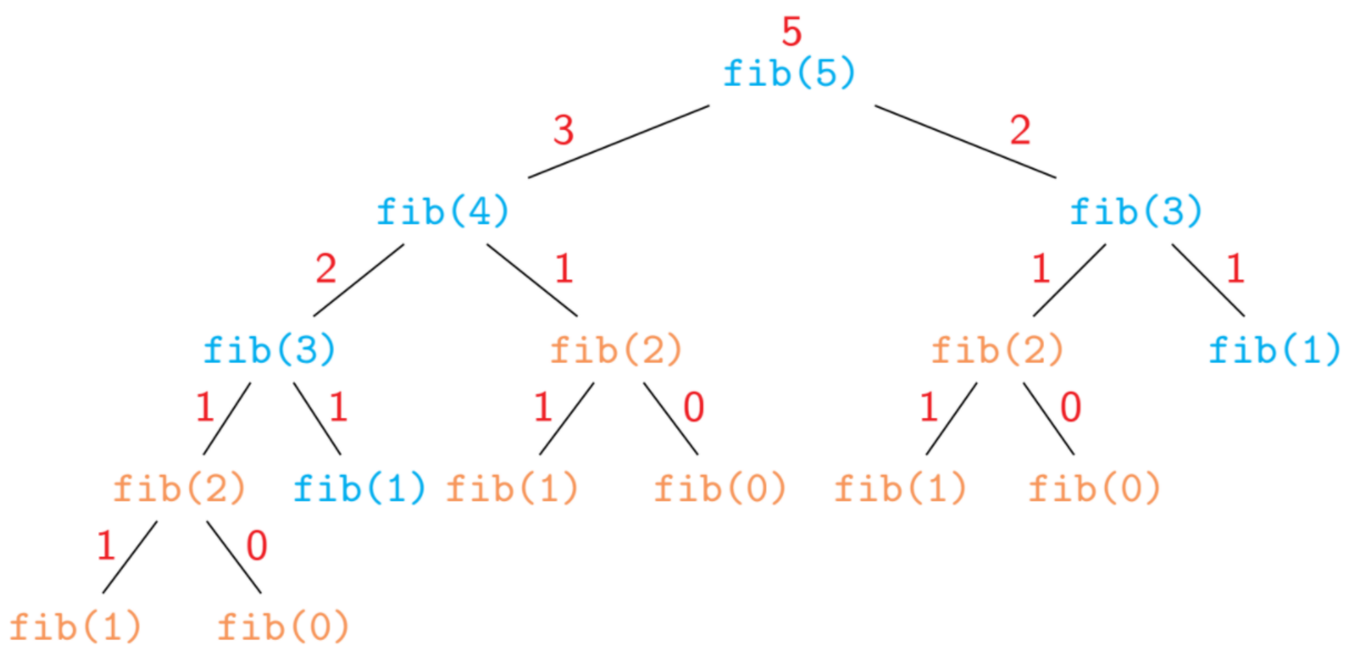
Evaluating Subproblems

- Fibonacci numbers
  - $fib(0) = 0$
  - $fib(1) = 1$
  - $fib(n) = fib(n-1) + fib(n-2)$

```
def fib(n):
  if n <= 1:
    value = n
  else:
    value = fib(n - 1) + fib(n - 2)

  return value
```

- Wasteful recomputation
- Computation tree grows exponentially

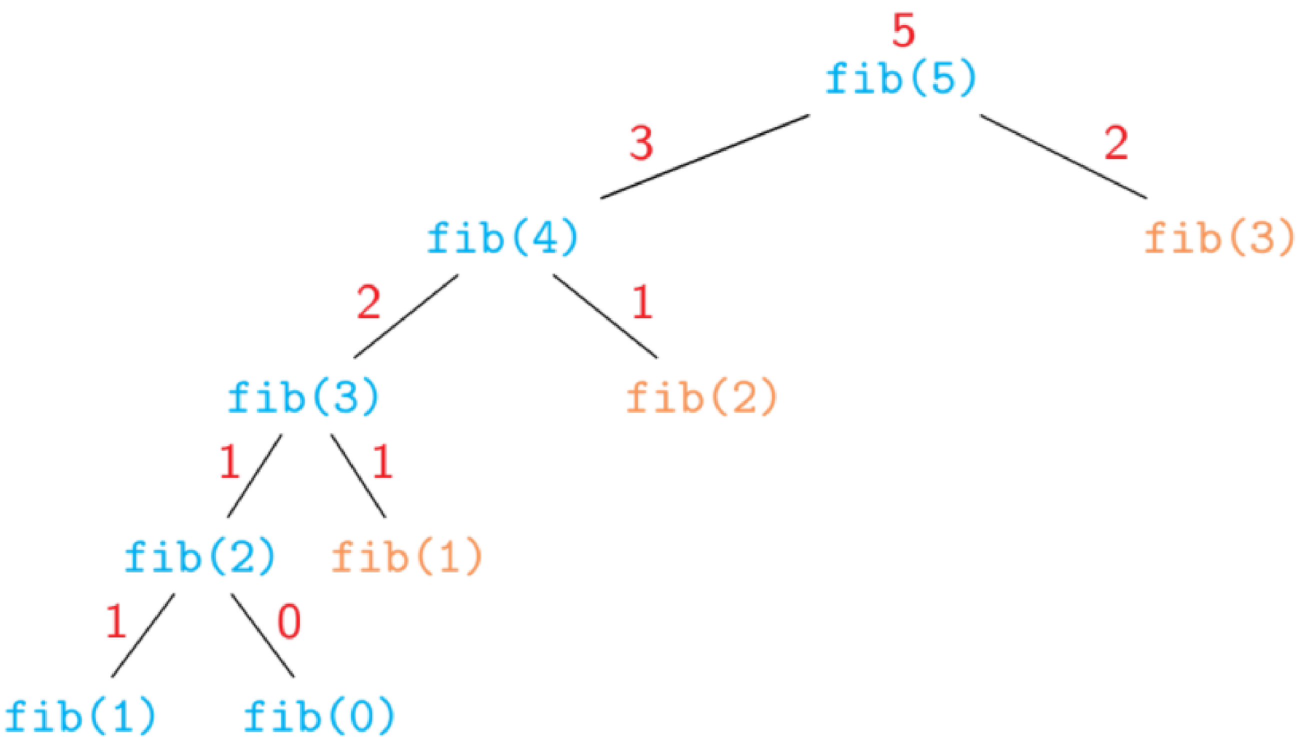
Evaluating `fib(5)`



Evaluating Subproblems

- Build a table of values already computed

- Memory table
- Memoization
    - Check if the value to be computed was already seen before
- Store each newly computed value in a table
- Look up the table before making a recursive call
- Computation tree becomes linear



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

Memoizing recursive implementations

```
def fib(n):
  if n in fibtable.keys():
    return fibtable[n]

  if n <= 1:
    value = n
  else:
    value = fib(n - 1) + fib(n - 2)
```

```
  fibtable[n] = value
  return value
```

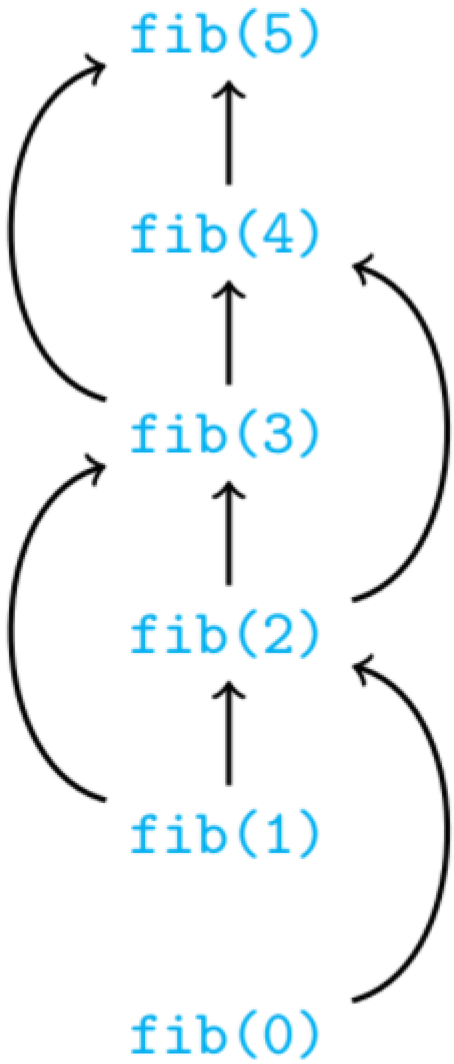
In general

```
def f(x, y, z):
  if (x, y, z) in ftable.keys():
    return ftable[(x, y, z)]

  recursively compute values
    from subproblems

  ftable[(x, y, z)] = value
  return value
```

Dynamic programming

- Anticipate the structure of subproblems
    - Derive from inductive definition
    - Dependencies form a DAG
- Solve subproblems in topological order
    - Never need to make a recursive call

Evaluating `fib(5)`



Summary

Memoization

- Store subproblem values in a table
- Look up the table before making a recursive call

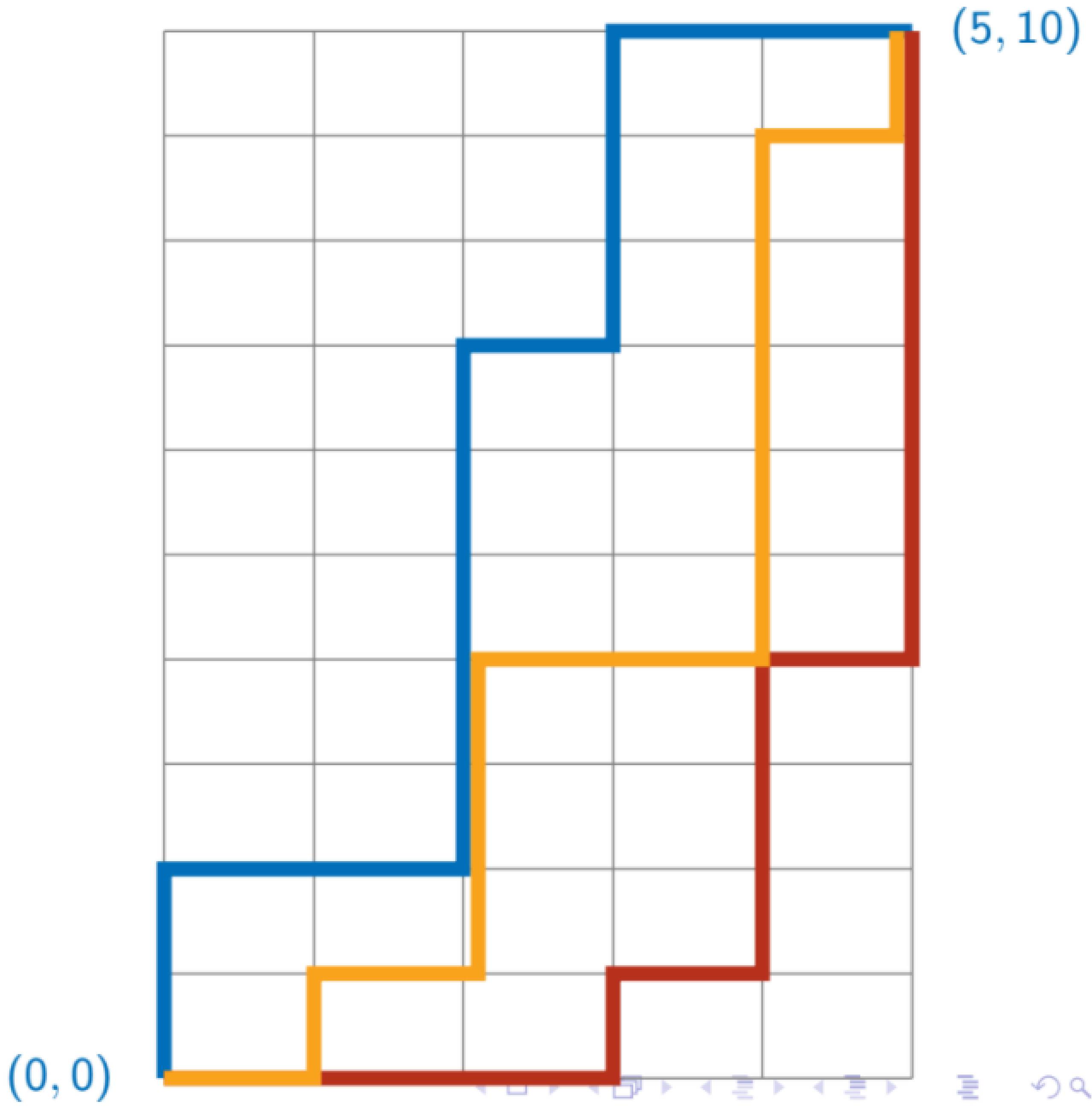
Dynamic programming

- Solve subproblems in a topological order of dependency
    - Dependencies must form a DAG
- Iterative evaluation of subproblems, no recursion

Grid Paths

Grid Paths

- Rectangular paths of one-way roads
- Can only go up and right
- How many paths form $(0, 0)$ to (m, n)?



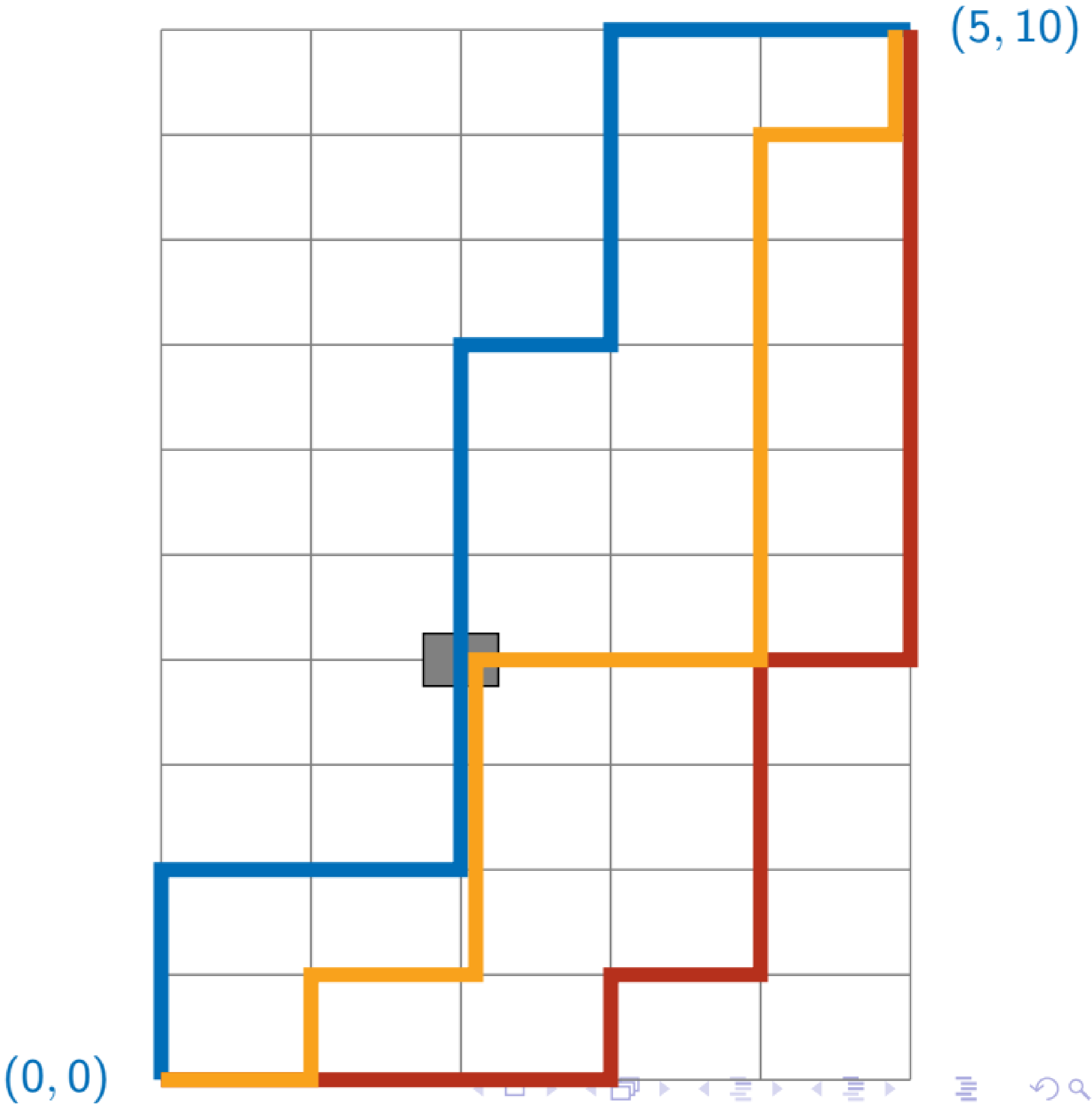
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
    - In general, $m + n$ segments from $(0, 0)$ to (m, n)

- Out of 15, exactly 5 are right moves, 10 are up moves
- Fix the position of the 5 right moves among the 15 positions overall
    - $C_5^{15} = \frac{15!}{10!.5!} = 3003$
    - Same as C_{10}^{15} -- fix the 10 up moves

Holes

- What if an intersection is blocked?
    - For instance, $(2, 4)$
- Need to discard paths passing through $(2, 4)$
    - Two of our earlier examples are invalid paths

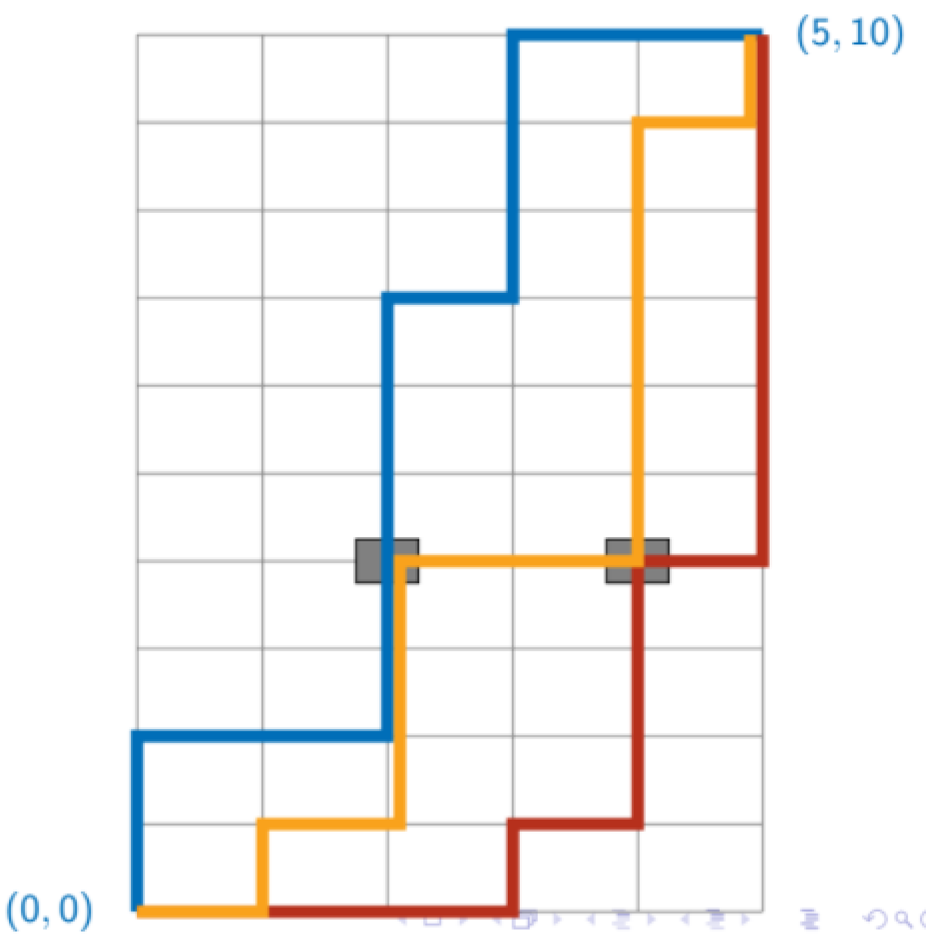


Combinatorial solution for holes

- Discard paths passing through $(2,4)$
- Every path via $(2,4)$ combines a path from $(0,0)$ to $(2,4)$ with a path from $(2,4)$ to $(5,10)$
  - Count these separately
  - $C_2^{2+4} = 15$ paths from $(0,0)$ to $(2,4)$
  - $C_3^{3+6} = 84$ paths from $(2,4)$ to $(5,10)$
- $15 \times 84 = 1,260$ paths via $(2,4)$
- $3,003 - 1,260 = 1,743$ valid paths avoiding $(2,4)$

More holes

- What if two intersections are blocked?
- Discard paths via $(2,4)$, $(4,4)$
  - Some paths are counted twice
- Add back the paths that pass through both holes
- **Inclusion-exculsion** – counting is messy



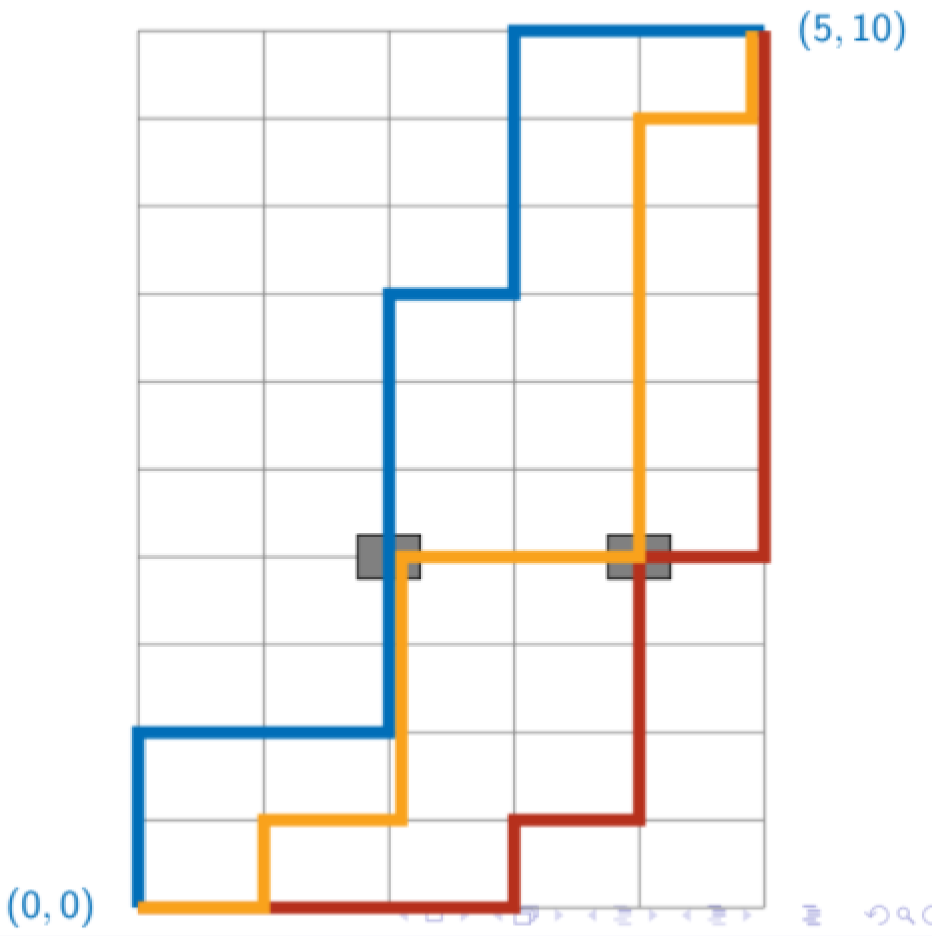
Inductive formulation

- 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)

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

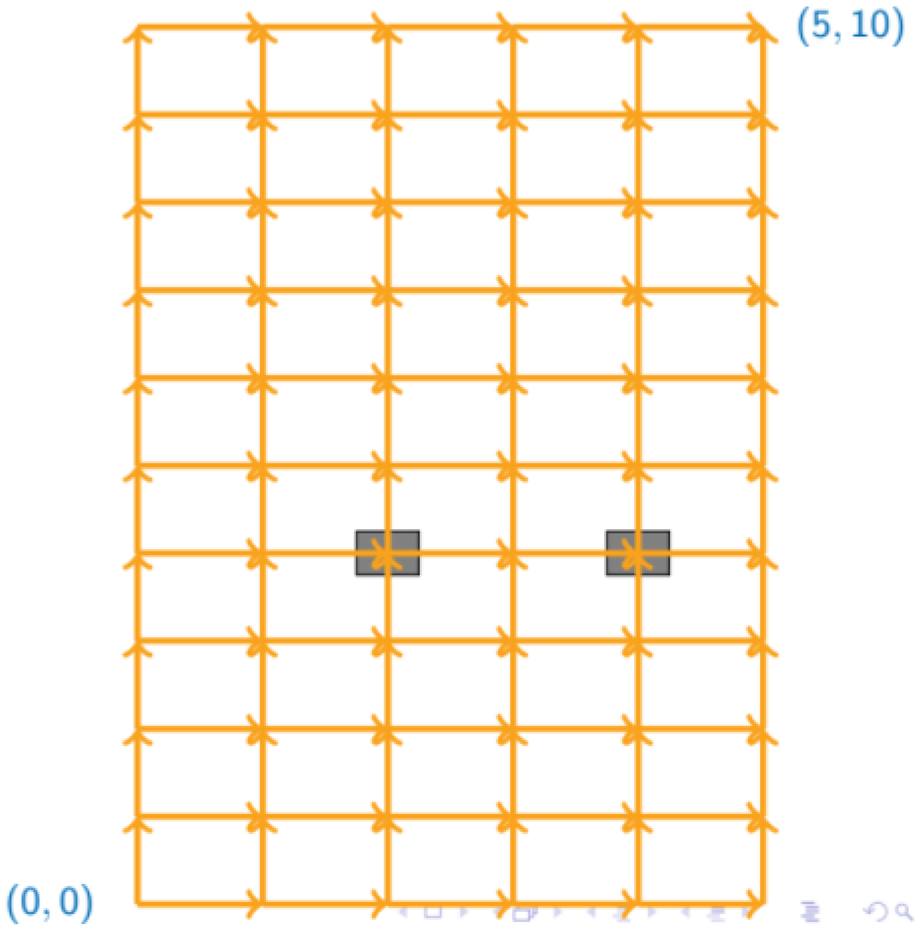
Computing $P(i,j)

- Naive recursion recomputes same subproblem repeatedly
    - $P(5,10)$ requires $P(4,10)$, $P(5,9)$
    - Both $P(4,10)$, $P(5,9)$ require $P(4,9)$
- Use memoization ...
- ... or find a suitable order to compute the subproblems

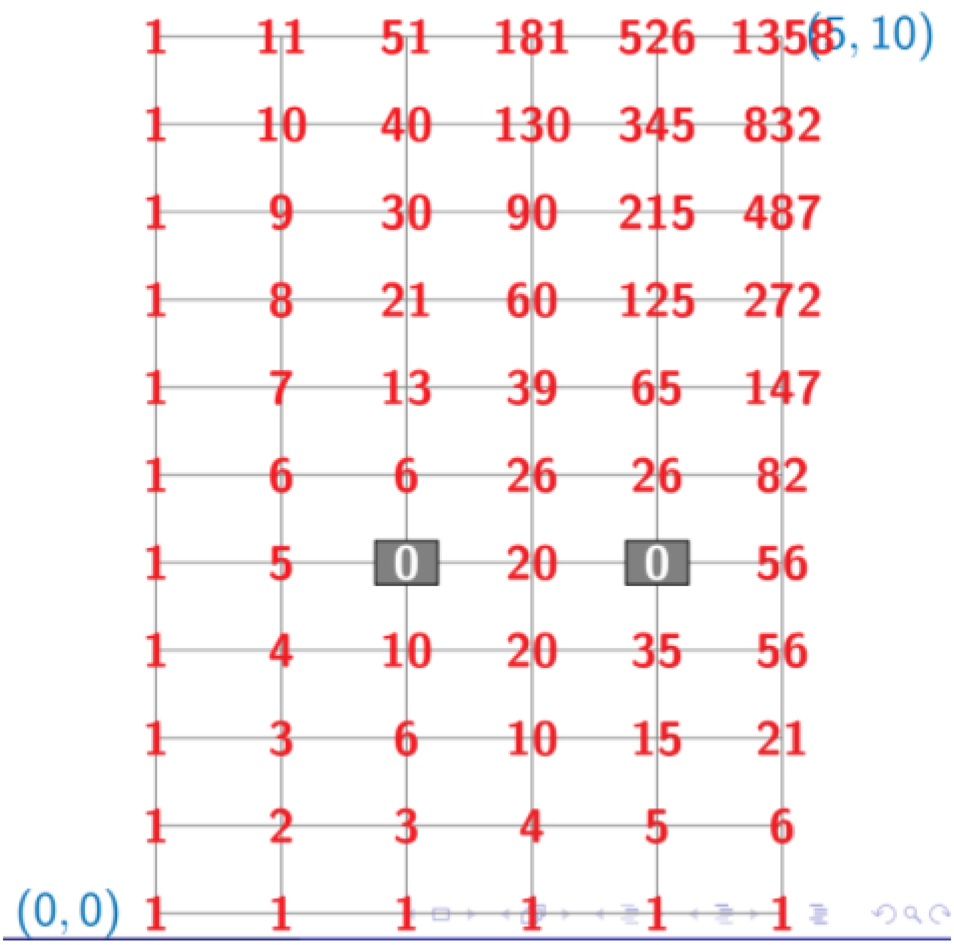


Dynamic Programming

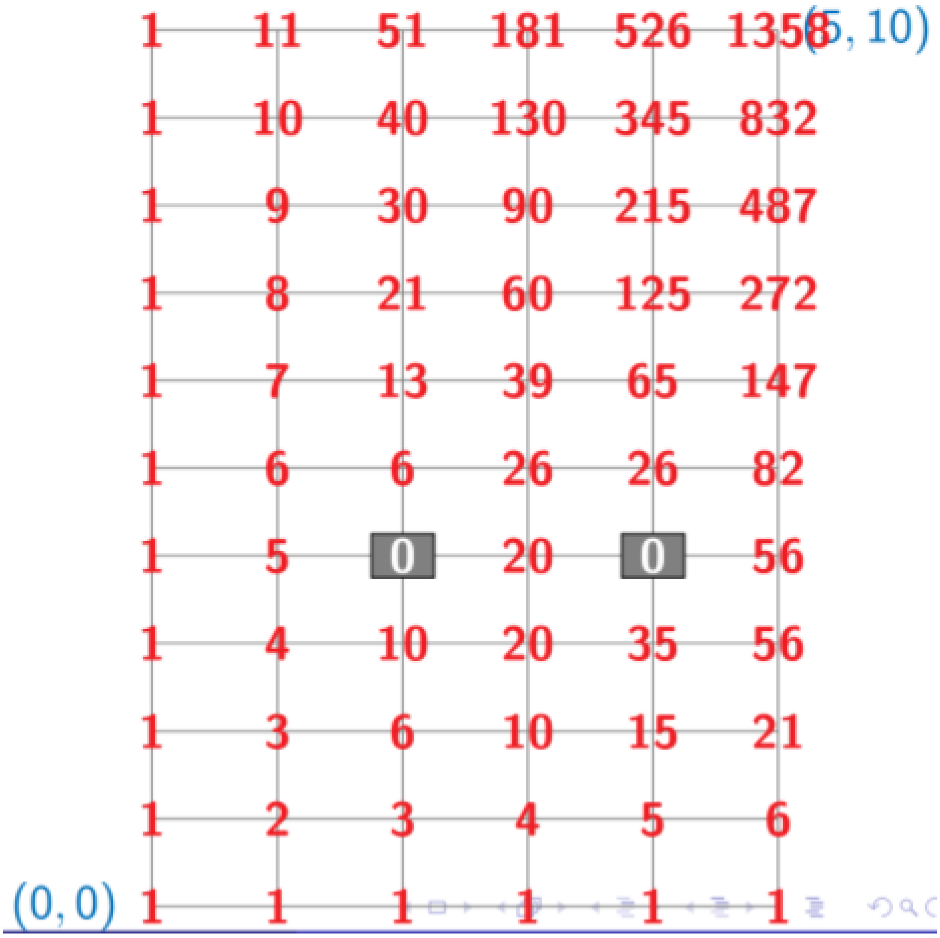
- Identify the DAG structure
- $P(0, 0)$ has no dependencies



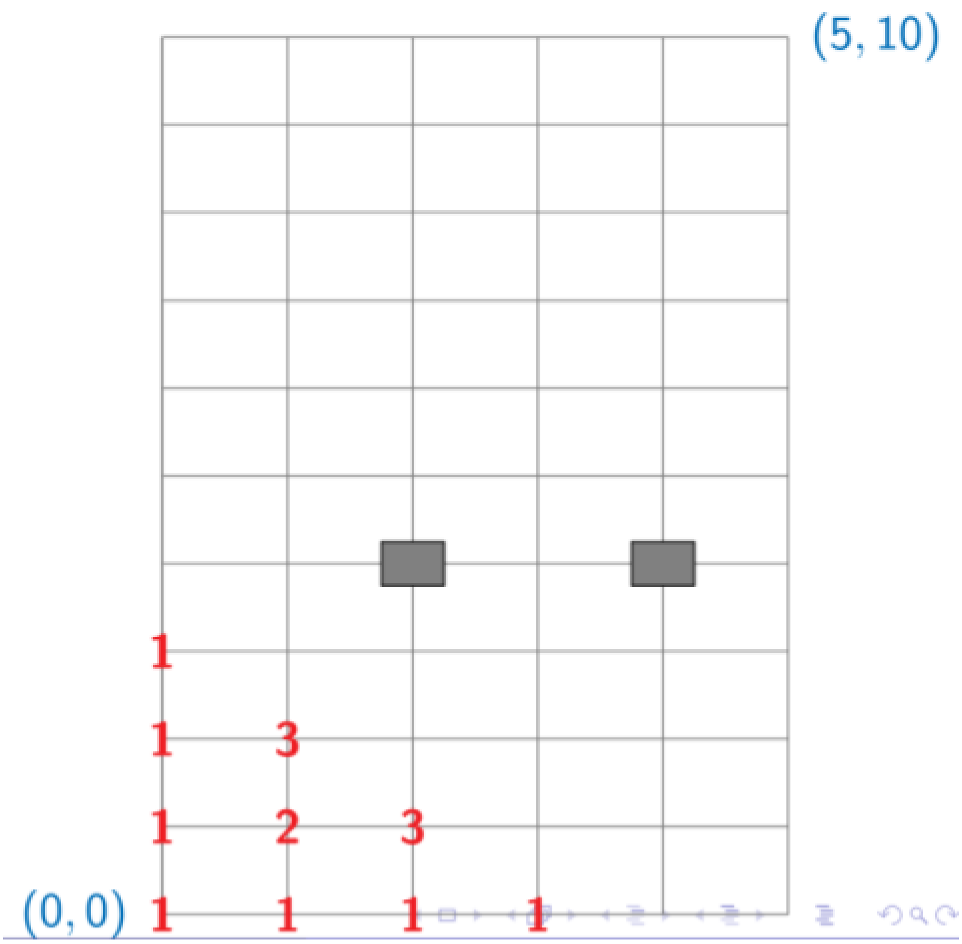
- Start at $(0, 0)$
- Fill row by row



- Fill column by column

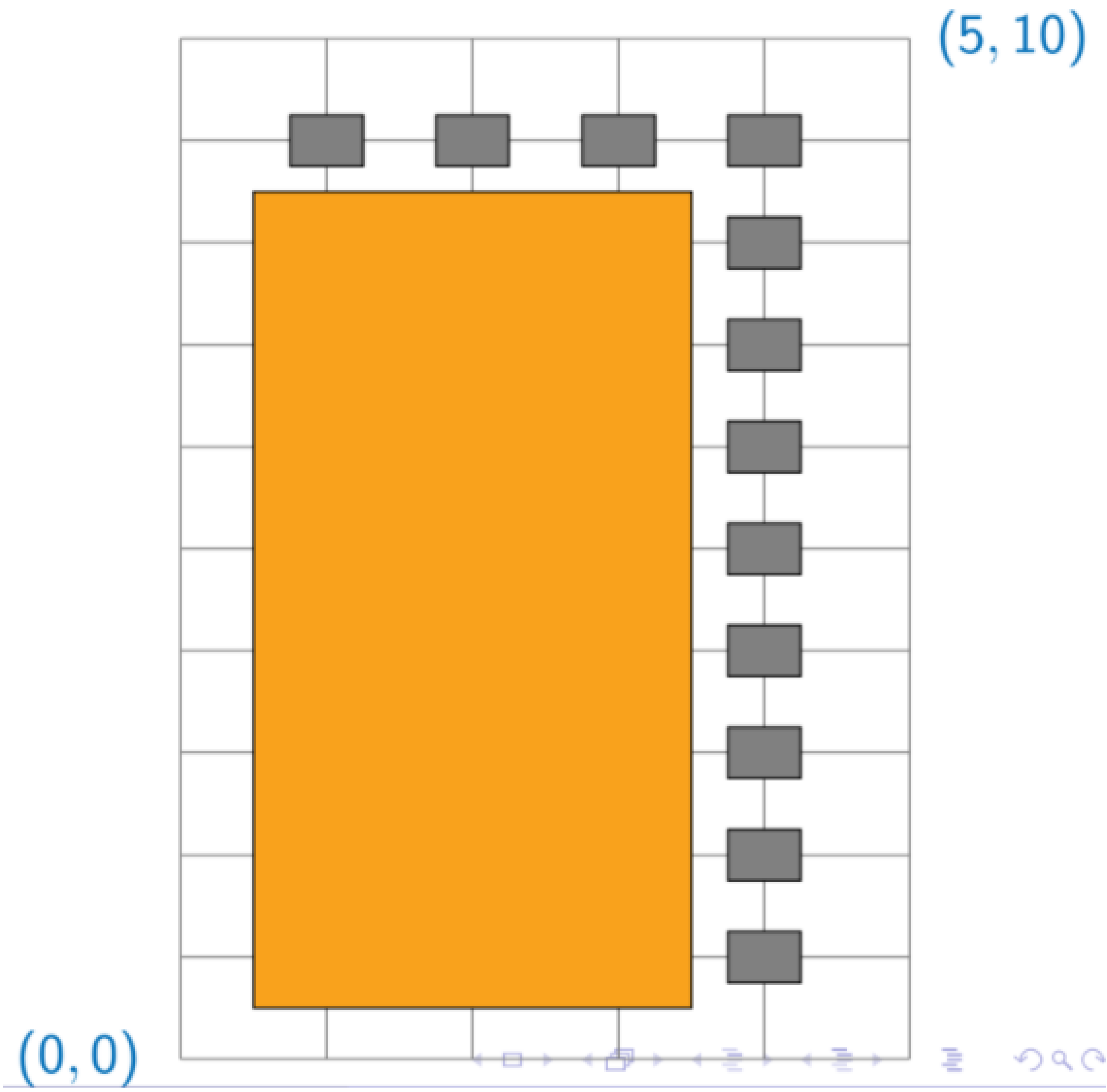


- Fill diagonal by diagonal



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



Common Subwords and Sequences

Longest common subword

- Given 2 strings, find the (length of the) longest common subword
  - `"secret", "secretary"` -- `"secret"`, length 6
  - `"bisect", "trisect"` -- `"isect"`, length 5
  - `"bisect", "secret"` -- `"sec"`, length 3
  - `"director", "secretary"` -- `"ee", "re"`, length 2
- Formally
  - $u = a_0 a_1 \ldots a_{m-1}$
  - $v = b_0 b_1 \ldots b_{n-1}$
  - Common subword of length k - for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
  - Find the largest such k - length of the longest common subword

Brute force

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u, $j$in v
  - Match $(a_i, b_j), (a_{i+1}, b_{j+1}) \ldots$ as far as possible
  - Keep track of longest match
- Assuming $m > n$, this is $O(mn^2)$
  - mn pairs of starting positions
  - From each starting position, scan could be $O(n)$

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i, j)$ - length of the longest common subword in $a_i a_{i+1} \ldots a_{m-1}, b_j b_{j+1} \ldots b_{n-1}$
  - If $a_i \neq b_j, LCW(i, j) = 0$
  - If $a_i = b_j, LCW(i, j) = 1 + LCW(i + 1, j + 1)$
  - Base case: $LCW(m, n) = 0$

- In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$
- In general, $LCW(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1).(n+1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at the 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							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

		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

Reading off the solution

- Find entry (i, j) with the largest LCW value
- Read off the actual subword diagonally

		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

```
def LCW(u, v):
  import numpy as np
  (m, n) = (len(u), len(v))
  lcw = np.zeros((m + 1, n + 1))
  max_lcw = 0

  for c in range(n - 1, -1, -1):
    for r in range(m - 1, -1, -1):
      if u[r] == v[c]:
        lcw[r, c] = 1 + lcw[r + 1, c + 1]
      else:
        lcw[r, c] = 0

      if lcw[r, c] > max_lcw:
        max_lcw = lcw[r, c]
```

```
  return max_lcw
```

Complexity

- Recall that the brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization
  - Fill a table of size $O(mn)$
  - Each table entry takes constant time to compute

Longest Common Subsequence

- **Subsequence** - can drop letters in between
- Given two strings, find the (length of the) longest common subsequence
  - `"secret", "secretary"` - `"secret"`, length 6
  - `"bisect", "trisect"` -- `"isect"`, length 5
  - `"bisect", "secret"` -- `"sec"`, length 3
  - `"director", "secretary"` -- `"ee", "re"`, length 2
- LCS is the longest path connecting non-zero LCW entries, moving right/down

		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

Applications

- Analyzing genes

- DNA is a long string over `A, T, G, C`
- Two species are similar if their DNA has long common subsequences

- `diff` command in Unix/Linux
  - Compares text files
  - Find the longest matching subsequence of lines
  - Each line of text is a "character"

Inductive structure

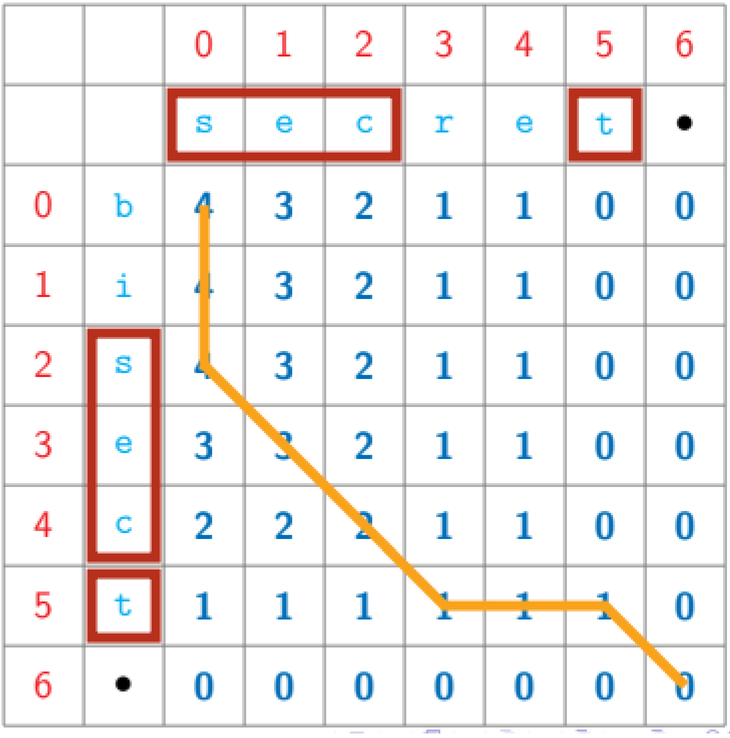
- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i,j)$ - length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}, b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$, $LCS(i,j) = 1 + LCS(i+1, j+1)$
  - Can assume (a_j, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of LCS
  - Which one should we drop?
  - Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the minimum
- Base cases as with LCW
  - $LCS(i,n) = 0$ for all $0 \leq i \leq m$
  - $LCS(m,j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1).(n+1)$ values
- $LCS(i,j)$ depends on $LCS(i+1, j+1), LCS(i, j+1), LCS(i+1, j)$

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

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



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

- No dependency for $LCS(m, n)$ - start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled
- Each diagonal step is an element of LCS

```
def LCS(u, v):
  import numpy as np
  (m, n) = (len(u), len(v))
  lcs = np.zeros((m + 1, n + 1))

  for c in range(n - 1, -1, -1):
    for r in range(m - 1, -1, -1):
      if u[r] == v[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]
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization
  - Fill a table of size $O(mn)$
  - Each table entry takes constant time to compute

Edit Distance

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
    - Insert a character
    - Delete a character
    - Substitute one character by another
- "The lecture taught the ~~students were able~~ to appreciate how the concept of optimal substructures ~~property~~ can~~d~~ ~~it~~bs~~e~~ used in designing algorithms"
- insert, ~~delete~~, ~~subst~~itute

Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 ~~subst~~ituted
- Edit distance is at most 44

Edit distance

- Minimum number of editing operations needed to transform one document to the other
    - Insert a character
    - Delete a character
    - Substitute one character by another
- Also called Levenshtein distance
    - Vladimir Levenshtein, 1965
- Applications
    - Suggestions for spelling correction
    - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
    - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v

- `bisect, secret` - LCS is `sect`
  - Delete `b, i` in `bisect` and `r, e` in `secret`
  - Delete `b, i` and then insert `r, e` in `bisect`
- LCS equivalent to edit distance without substitution

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Recall LCS
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i + 1, j + 1)$
- if $a_i \neq b_j$, $LCS(i, j)$ = max(LCS(i,j+1), LCS(i+1,j))
- Edit distance - aim is to transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i, j)$ - edit distance for $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $ED(i, j) = ED(i + 1, j + 1)$
- If $a_i \neq b_j$, $ED(i, j) = 1 + min(ED(i + 1, j + 1), ED(i + 1, j), ED(i, j + 1))$
- Base case
  - $ED(m, n) = 0$
  - $ED(i, n) = m - i$ for all $0 \leq i \leq m$, Delete $a_i a_{i+1} \dots a_{m-1}$ fro u
  - $ED(m, j) = n - j$ for all $0 \leq j \leq n$, Insert $b_j b_{j+1} \dots b_{n-1}$ into u

Subproblem dependency

- Subproblems are $ED(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1).(n + 1)$ values

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

- Like LCS, $ED(i,j)$ depends on $ED(i+1,j+1), ED(i,j+1), ED(i+1,j)$

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

		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

- No dependency for $ED(m,n)$ - start at bottom right and fill by row, column or diagonal

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`, Delete `i`, Insert `r`, Insert `e`

```
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
```

Complexity

- Again, $O(mn)$, using dynamic programming or memoization
  - Fill a table of size $O(mn)$
  - Each table entry takes constant time to compute

Matrix Multiplication

Multiplying matrices

- Multiple matrices A, B
  - $AB[i,j] = \Sigma_{k=0}^{n-1} A[i,k]B[k,j]$
- Dimensions must be compatible
  - $A : m \times n, B : n \times p$
  - $AB : m \times p$
- Computing each entry in AB is $O(n)$
- Overall, computing AB is $O(mnp)$
- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Brackting does not change answer
  - ... but can affect the complexity
- Let $A : 1 \times 100, B : 100 \times 1, C : 1 \times 100$
- Computing $A(BC)$
  - $BC : 100 \times 100$, takes $100.1.100 = 10000$ steps to compute
  - $A(BC) : 1 \times 100$, takes $1.100.100 = 10000$ steps to compute
- Computing $(AB)C$
  - $AB : 1 \times 1$, takes $1.100.1 = 100$ steps to compute
  - $(AB)C : 1 \times 100$, takes $1.1.100 = 100$ steps to compute
- 20000 steps vs 200 steps

- Given n matrices $M_0 : r_0 \times c_0, M_1 : r_1 \times c_1, \ldots, M_{n-1} : r_{n-1} \times c_{n-1}$
  - Dimensions match: $r_j = c_{j-1}, 0 < j < n$
  - Product $M_0 . M_1 \ldots M_{n-1}$ can be computed
- Find the optimal order to compute the product
  - Multiple 2 matrices at a time
  - Bracket the expression optimally

Inductive Structure

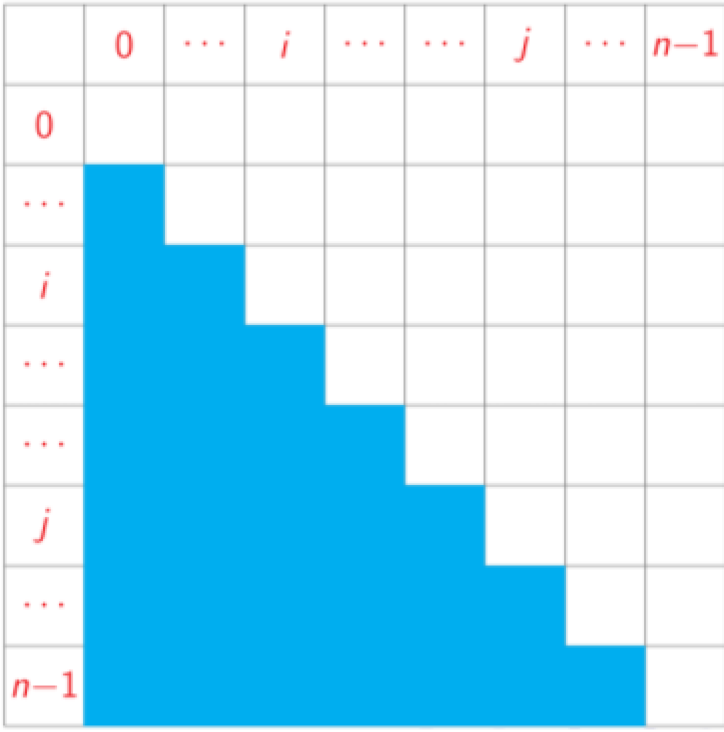
- Final step combines two subproducts $(M_0 . M_1 \ldots M_{k-1}) . (M_k . M_{k+1} \ldots M_{n-1})$ for some $0 < k < n$
- First factor is $r_0 \times c_{k-1}$, second is $r_k \times c_{n-1}$, where $r_k = c_{k-1}$

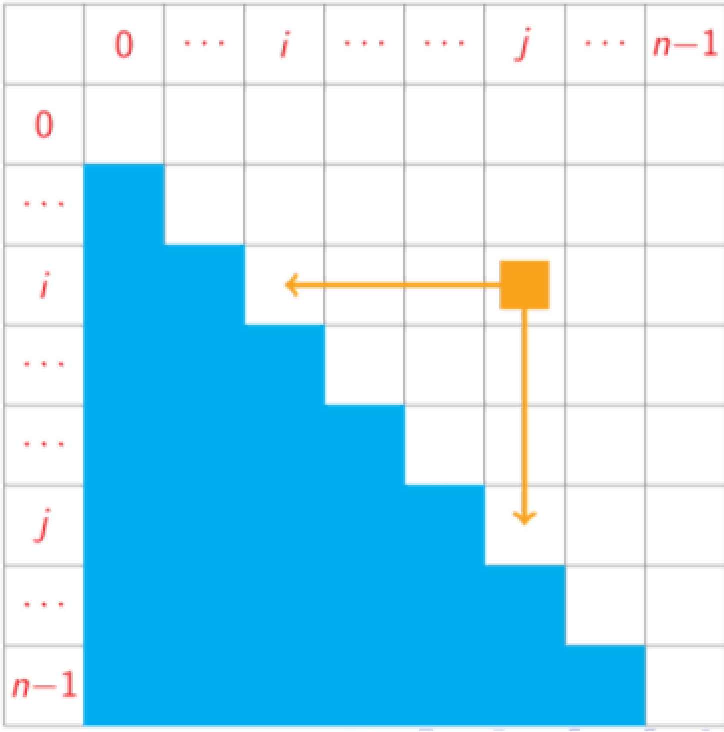
- Let $C(0, n-1)$ denote the overall cost
- Final multiplication is $O(r_0 r_k c_{n-1})$
- Inductively, costs of factors are $C(0, k-1)$ and $C(k, n-1)$
- $C(0, n-1) = C(0, k-1) + C(k, n-1) + r_0 r_k c_{n-1}$
- Which k should we choose?
    - Try all and choose the minimum!
- Subproblems?
    - $M_0 \cdot M_1 \dots M_{k-1}$ would decompose as $(M_0 \dots M_{j-1}) \cdot (M_j \dots M_{k-1})$
    - Generic subproblem is $M_j \cdot M_{j+1} \dots M_k$
- $C(j, k) = min_{j<l\leq k}[C(j, l-1) + C(l, k) + r_j r_l c_k]$
- Base case: $C(j, j) = 0$ for $0 \leq j < n$

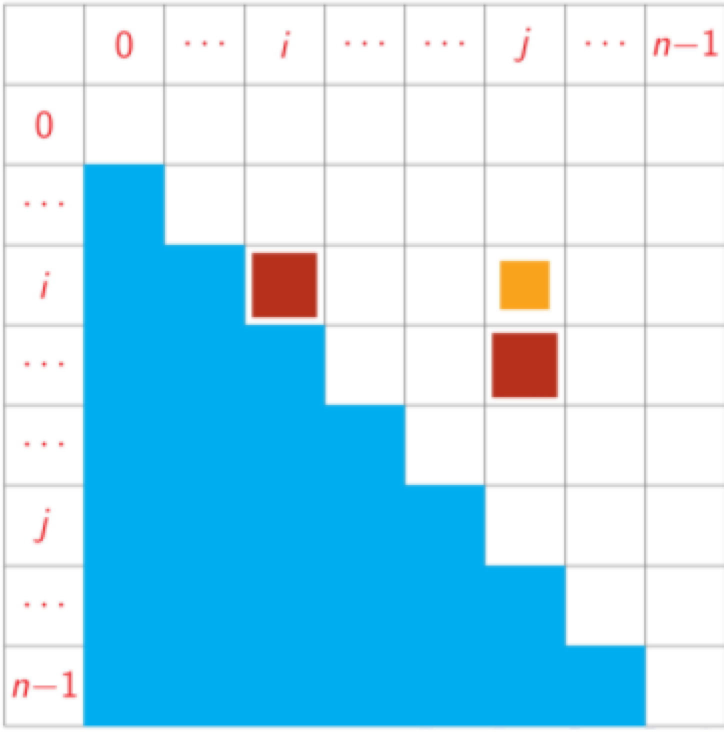
Subproblem dependency

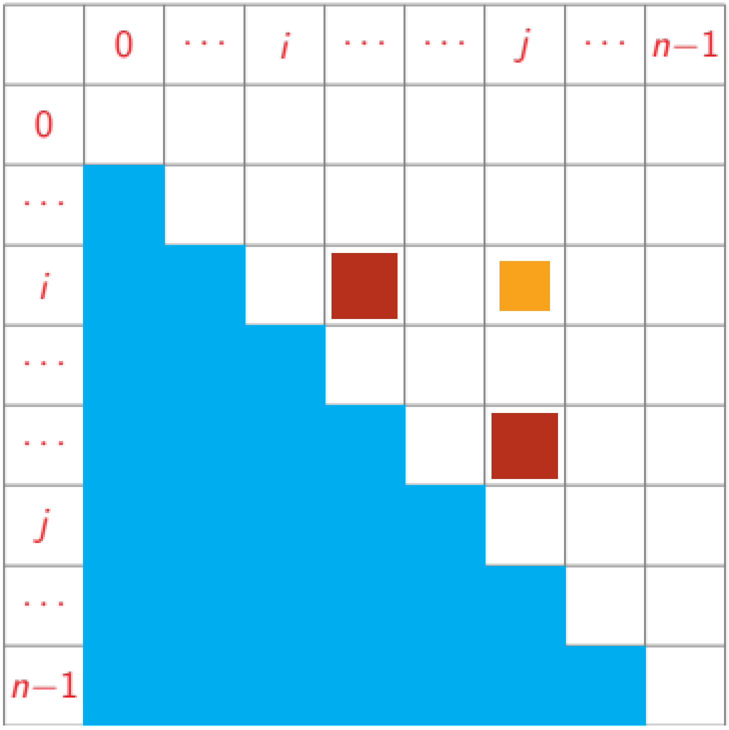
- Compute $C(i, j), 0 \leq i, j < n$
    - Only for $i \leq j$
    - Entries above main diagonal

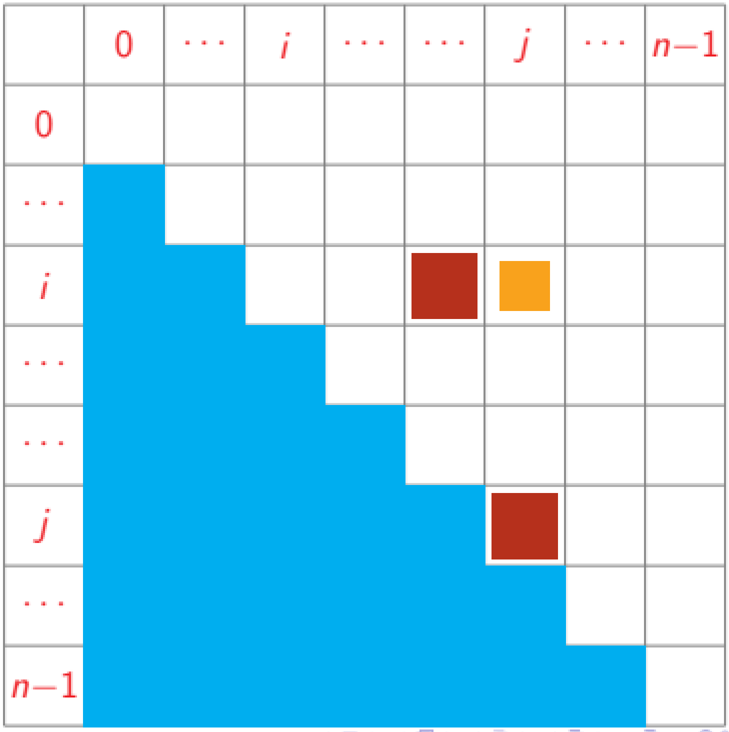
	0	$\cdots$	i	$\cdots$	$\cdots$	j	$\cdots$	$n-1$
0								
$\cdots$								
i								
$\cdots$								
$\cdots$								
j								
$\cdots$								
$n-1$								

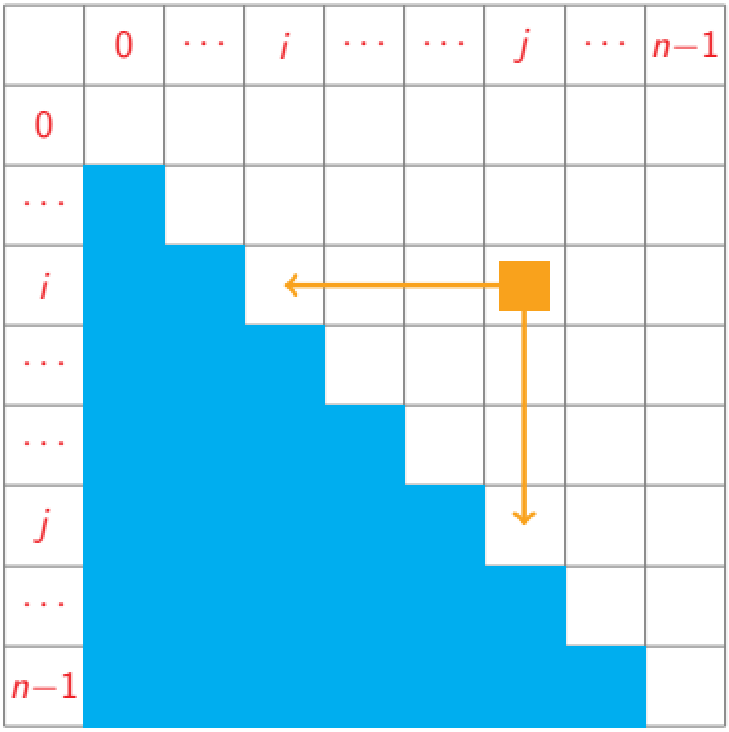

0
· · ·
i
· · ·
· · ·
j
· · ·
n−1
0
· · ·
i
· · ·
· · ·
j
· · ·
n−1

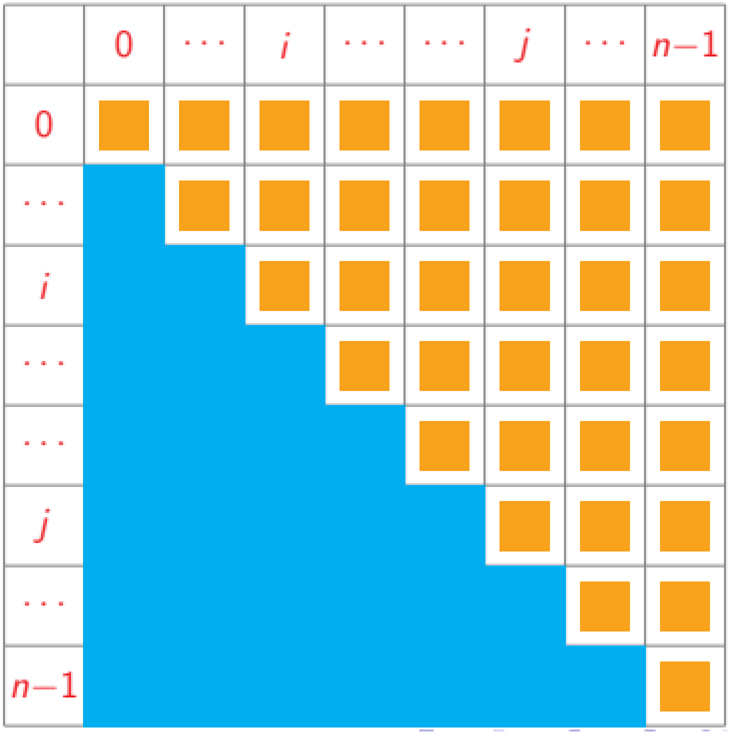

0
· · ·
i
· · ·
· · ·
j
· · ·
n−1
0
· · ·
i
· · ·
· · ·
j
· · ·
n−1


0
⋯
i
⋯
⋯
j
⋯
n−1
0
⋯
i
⋯
⋯
j
⋯
n−1


0
⋯
i
⋯
⋯
j
⋯
n−1
0
⋯
i
⋯
⋯
j
⋯
n−1


0 ··· i ··· ··· j ··· n−1
0 ··· i ··· ··· j ··· n−1


0 ··· i ··· ··· j ··· n−1
0 ··· i ··· ··· j ··· n−1



- $C(i, j)$ depends on $C(i, k-1), C(k, j)$ for every $i < k \leq j$
  - $O(n)$ dependencies per entry, unlike LCW, LCS and ED
- Diagonal entries are base case
- Fill matrix by diagonal, from main diagonal

Implementation

```
def C(dim):
  # dim: dimension matrix,
  #      entries are pairs (r_i, c_i)
  import numpy as np
  n = dim.shape[0]
  C = np.zeros((n, n))

  for i in range(n):
    C[i, i] = 0

  for diff in range(1, n):
    j = i + diff
    C[i, j] = C[i, i] + C[i + 1, j] + dim[i][0] * dim[i + 1][0]

    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]
```

```
return C[0, n - 1]
```

Complexity

- We have to fill a table of size $O(n^2)$
- Filling each entry takes $O(n)$
- Overall, $O(n^3)$