Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ]
  - isort([x1,x2,..,xn]) = insert(x1,isort([x2,...,xn]))

… Recursive programs

```
def factorial(n):
  if n <= 0:
    return(1)
  else:
    return(n*factorial(n-1))
```

Sub problems

- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-3), …, factorial(0)
- isort([x2,…,xn]) is a subproblem of isort([x1,x2,…,xn])
  - So is isort([xi,…,xj]) for any $1 \le i \le j \le n$
- Solution of f(y) can be derived by combining solutions to subproblems

Evaluating subproblems

Fibonacci numbers

- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

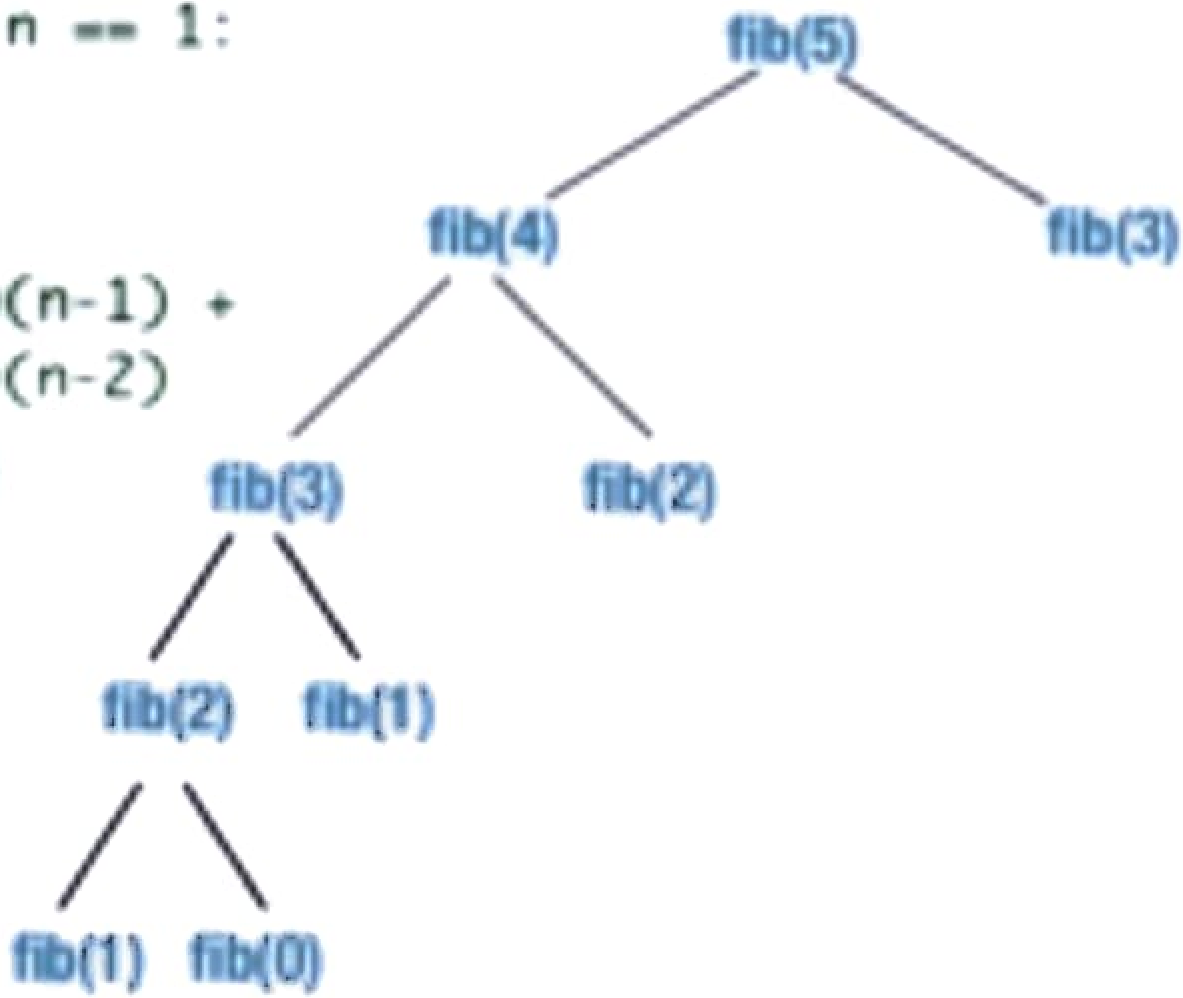
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

fib(5)

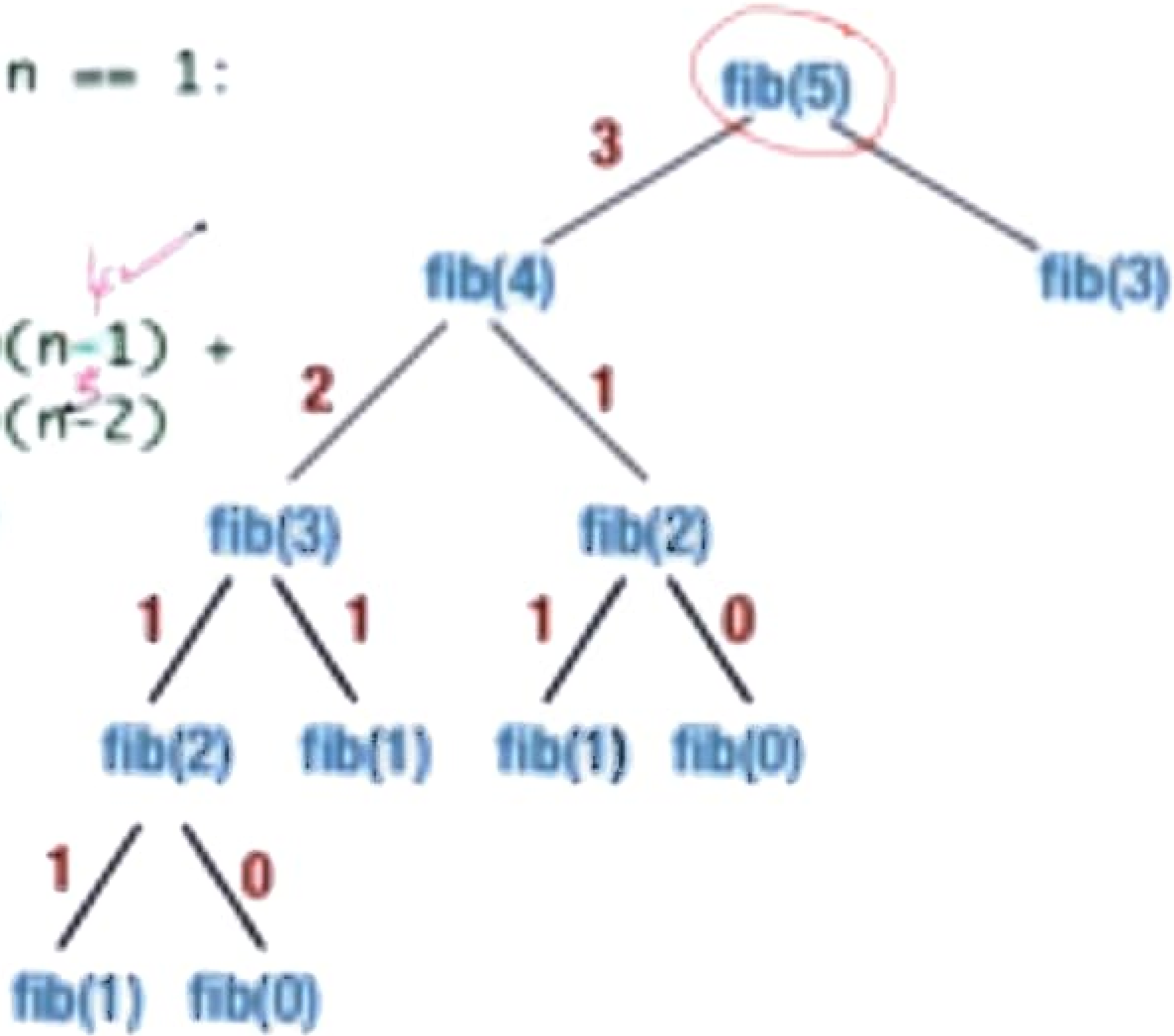
Computing fib(5)

```
def fib(n):
   if n == 0 or n == 1:
     value = n
   else:
     value = fib(n-1) +
             fib(n-2)
   return(value)
```



Computing fib(5)

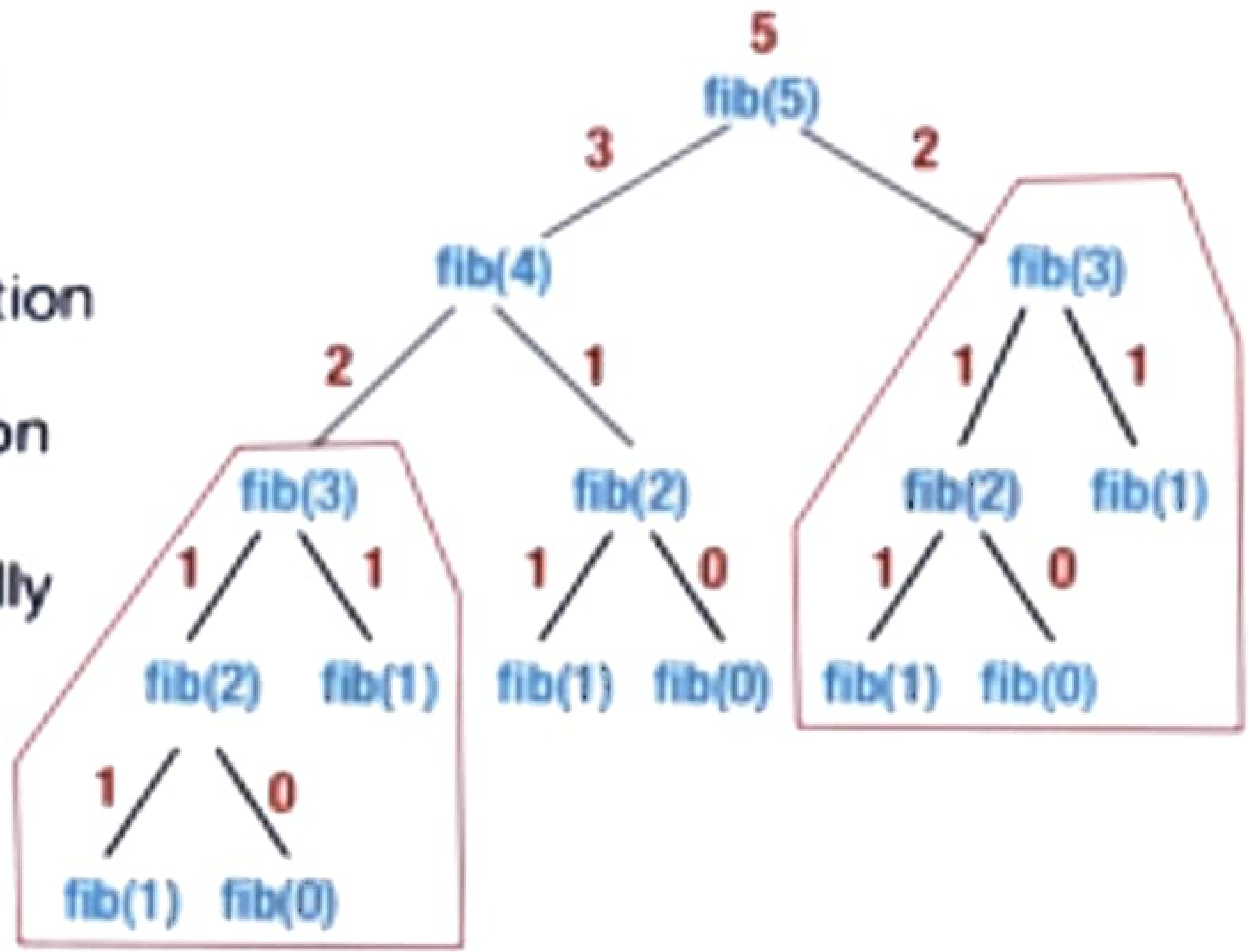
```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Computing fib(5)

Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially



Never re-evaluate a subproblem

- Build a table of values already computed
  - Memory table
- Memoization
  - Remind yourself that this value has already been seen before

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

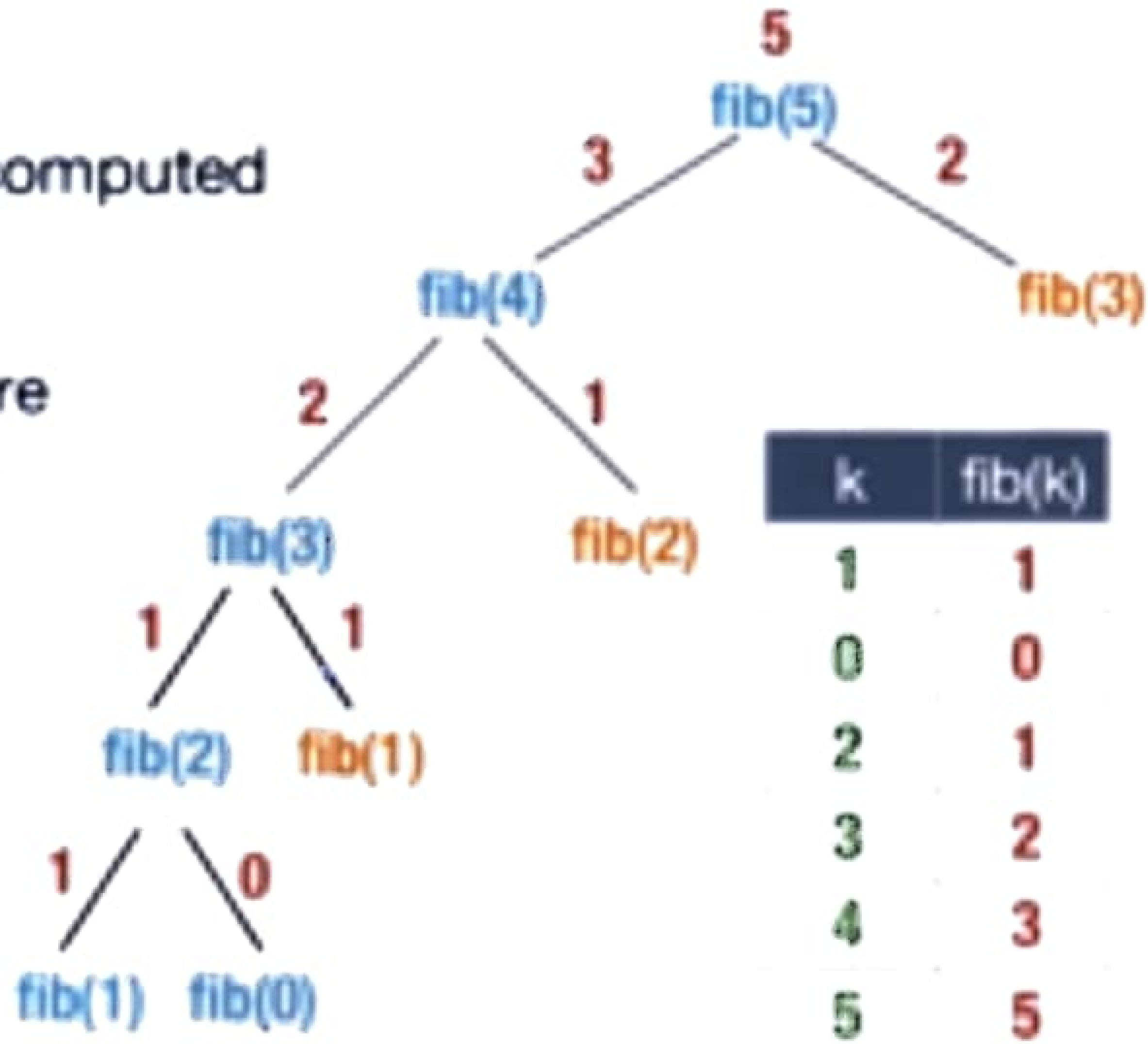
fib(5)

k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear



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

Memoized fibonacci

```
def fib(n):
  if fibtable[n]:
    return(fibtable[n])
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

Memoized fibonacci

```
def fib(n):
  if fibtable[n]:
    return(fibtable[n])
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

In general

```
function f(x,y,z):
  if ftable[x][y][z]:
    return(ftable[x][y][z])
  value = expression in terms of
                  subproblems
  ftable[x][y][z] = value
  return(value)
```

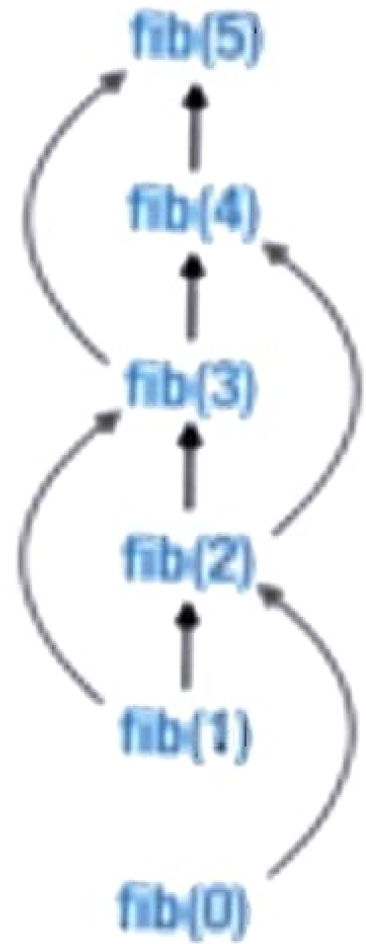
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic,

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic.

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



Dynamic programming fibonacci

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


  return(fibtable[n])
```

Summary

Memoization

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

Dynamic programming:

- Solve subproblems in order of dependency
  - Dependencies must be acyclic
- Iterative evaluation

Grid Paths

- Roads arranged in a rectangular grid
- Can only go up or right
- How many different routes from (0,0) to (m,n)?

Combinatorial solution

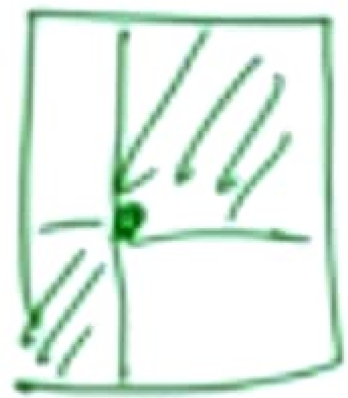
- Every path from (0,0) to (5,10) has 15 segments
  - In general m+n se [illegible] om (0,0) to (m,n)
- Of these exactly 5 a [illegible] ves, 10 are up moves
- Fix the positions of the 5 right moves among the overall 15 positions
  - 15 choose 5 = (15!)/(10!)(5!) = 3003
  - Same as 15 choose 10: fix the 10 up moves

Holes

- What if an intersection is blocked?
  - (2,4), for example
- Paths through (2,4) need to be discarded
  - Two of our earlier examples are invalid paths

Combinatorial solution

- Every path through (2,4) goes from (0,0) to (2,4) and then from (2,4) to (5,10)
  - Count these separately:
    - (4+2) choose 2 = 15
    - (6+3) choose 3 = 84



  - Multiply to get all paths through (2,4): 1260
  - Subtract from 15 choose 5 = 3003 to get valid paths that avoid (2,4): 1743

Combinatorial solution

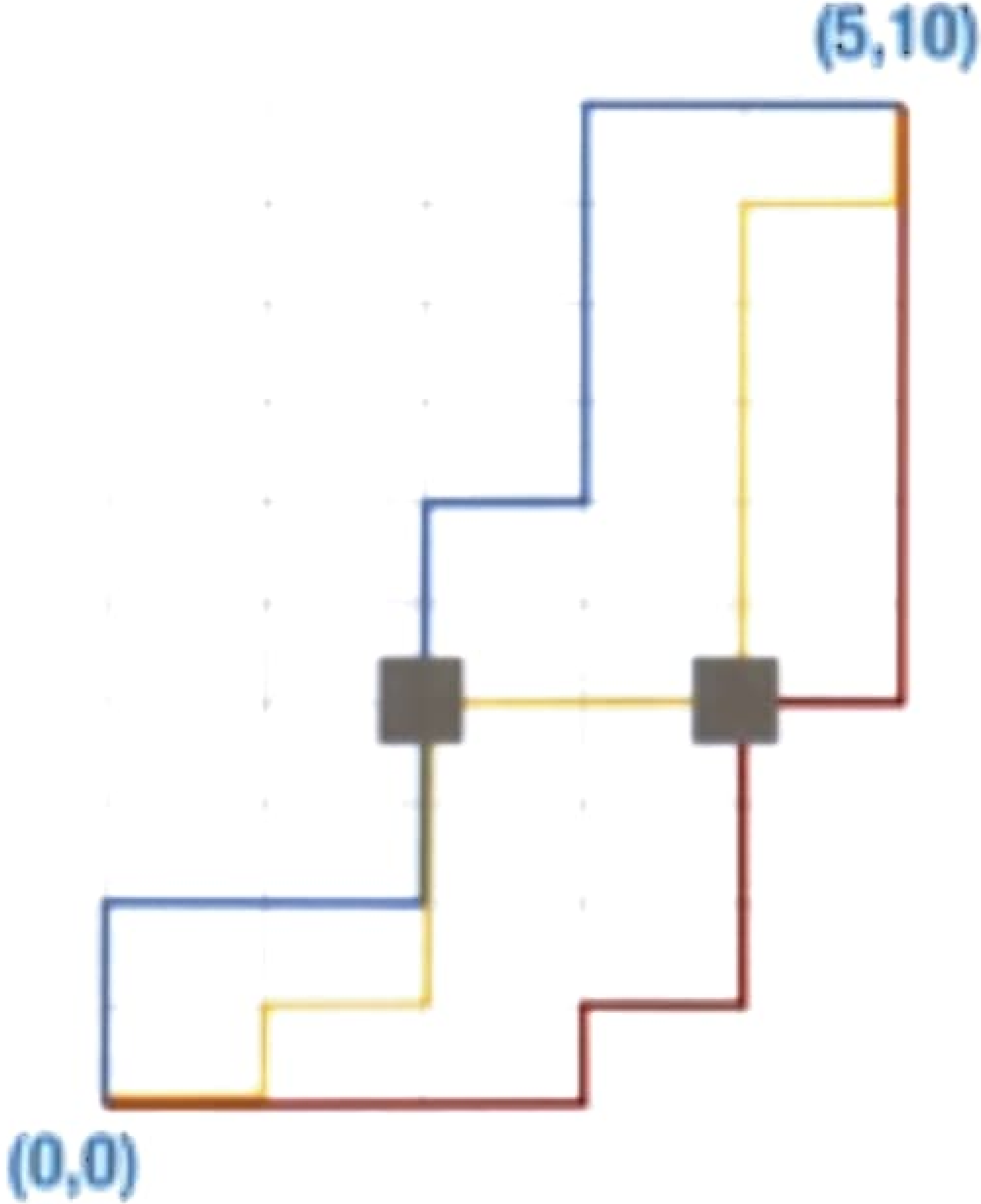
- Every path from (0,0) to (5,10) has 15 segments
  - In general m+n segments from (0,0) to (m,n)
- Of these exactly 5 are right moves, 10 are up moves
- Fix the positions of the 5 right moves among the overall 15 positions
  - 15 choose 5 = (15!)/(10!)(5!) = 3003
  - Same as 15 choose 10: fix the 10 up moves

Combinatorial solution

- Every path through (2,4) goes from (0,0) to (2,4) and then from (2,4) to (5,10)
  - Count these separately:
    - (4+2) choose 2 = 15
    - (6+3) choose 3 = 84
  - Multiply to get all paths through (2,4): 1260
  - Subtract from 15 choose 5 = 3003 to get valid paths that avoid (2,4): 1743

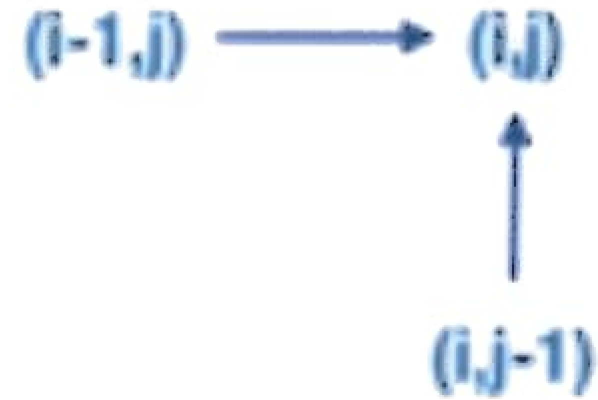
Holes

- What if two intersections are blocked?
- Subtract paths through (2,4), (4,4)
  - Some paths are counted twice!
- Add back paths through both holes
- Inclusion-exclusion: messy



Inductive formulation

- How can a path reach (i,j)
  - Move up from (i,j-1)
  - Move right from (i-1,j)
- Every path to these neighbours extends in a unique way to (i,j)



Inductive formulation

- Paths(i,j) : Number of paths from (0,0) to (i,j)
- Paths(i,j) = Paths(i-1,j) + Paths(i,j-1)
- Boundary cases
  - Paths(i,0) = Paths(i-1,0) # Bottom row
  - Paths(0,j) = Paths(0,j-1) # Left column
  - Paths(0,0) = 1 # Base case

Computing Paths(i,j)

- Naive recursion will recompute multiple times
  - Paths(5,10) requires Paths(4,10) and Paths(5,9)
  - Both Paths(4,10) and Paths(5,9) require Paths(4,9)
- Use memoization …
- … or compute the subproblems directly in a suitable way

Dealing with holes

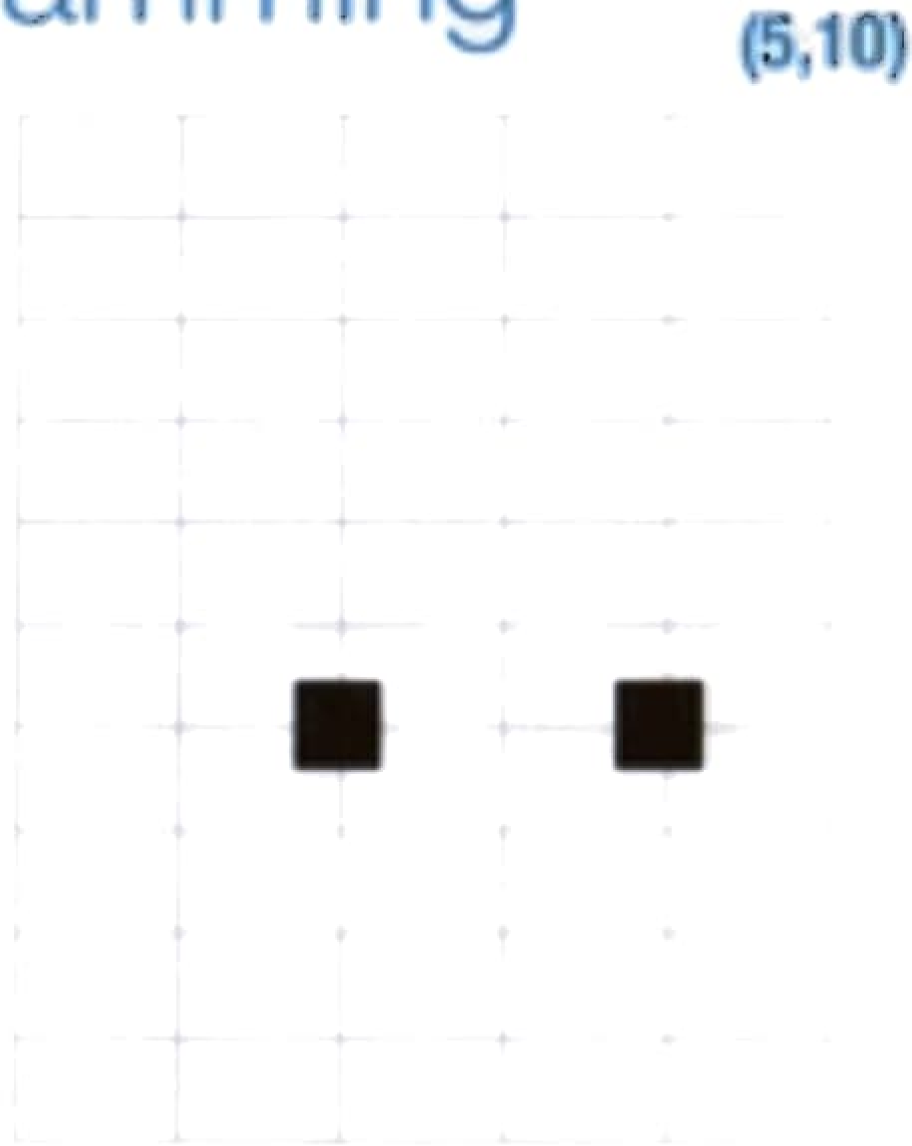
- Paths(i,j) = 0, if there is a hole at (i,j)
- Paths(i,j) = Paths(i- … (i,j-1), otherwise
- Boundary cases
  - Paths(i,0) = Paths(i-1,0) # Bottom row
  - Paths(0,j) = Paths(0,j-1) # Left column
  - Paths(0,0) = 1 # Base case

Dealing with holes

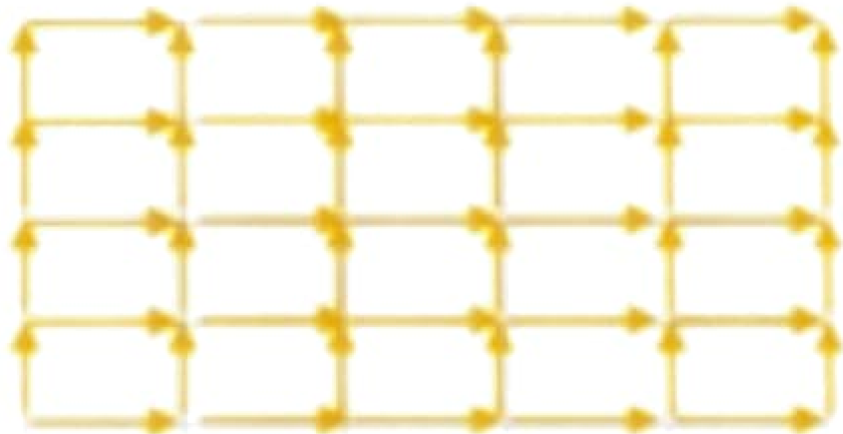
- Paths(i,j) = 0, if there is a hole at (i,j)
- Paths(i,j) = Paths(i-1,j) + Paths(i,j-1), otherwise
- Boundary cases
  - Paths(i,0) = Paths(i-1,0) # Bottom row
  - Paths(0,j) = Paths(0,j-1) # Left column
  - Paths(0,0) = 1 # Base case

Dynamic programming

- Identify dependency structure
- Paths(0,0) has no dependencies
- Start at (0,0)



Dynamic programming



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

1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

Dynamic programming

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

1	11	51	181	526	1363
1	10	40	130	345	837
1	9	30	90	215	492
1	8	21	60	125	272
1	7	13	39	65	147
1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

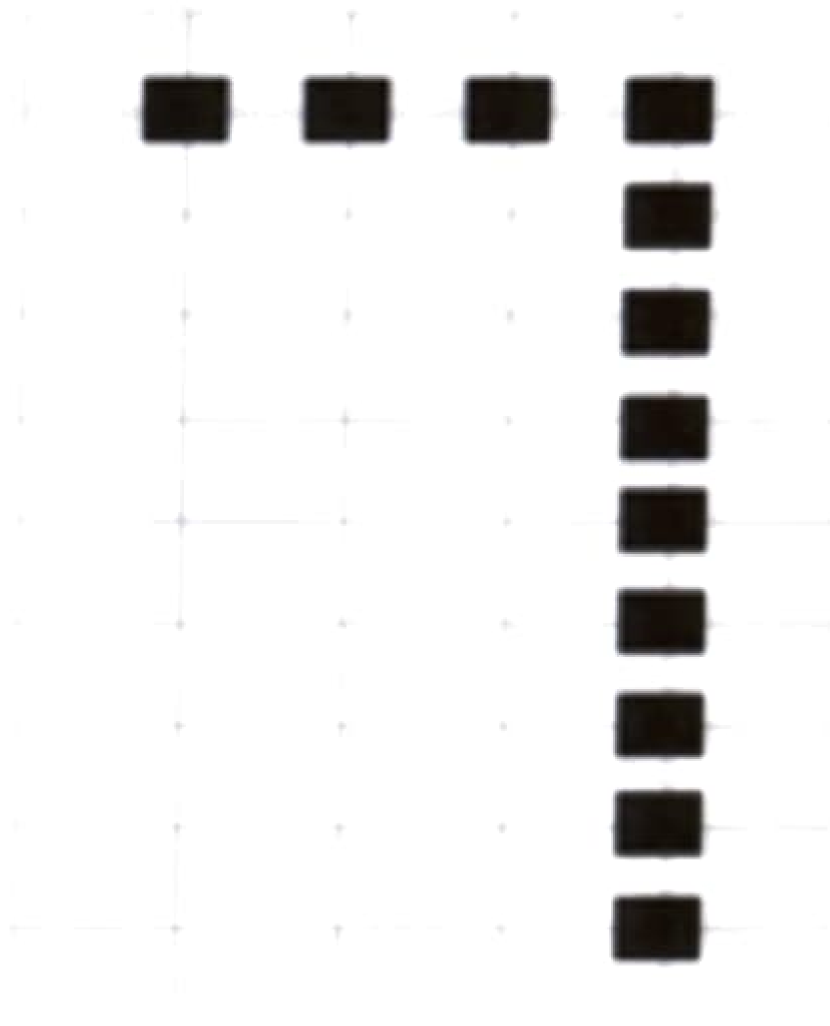
Dynamic programming

- Start at (0,0)
- Fill by column

1	11	51	181	526	1363
1	10	40	130	345	837
1	9	30	90	215	492
1	8	21	60	125	272
1	7	13	39	65	147
1	6	6	26	26	82
1	5	0	20	0	56
1	4	10	20	35	56
1	3	6	10	15	21
1	2	3	4	5	6
1	1	1	1	1	1

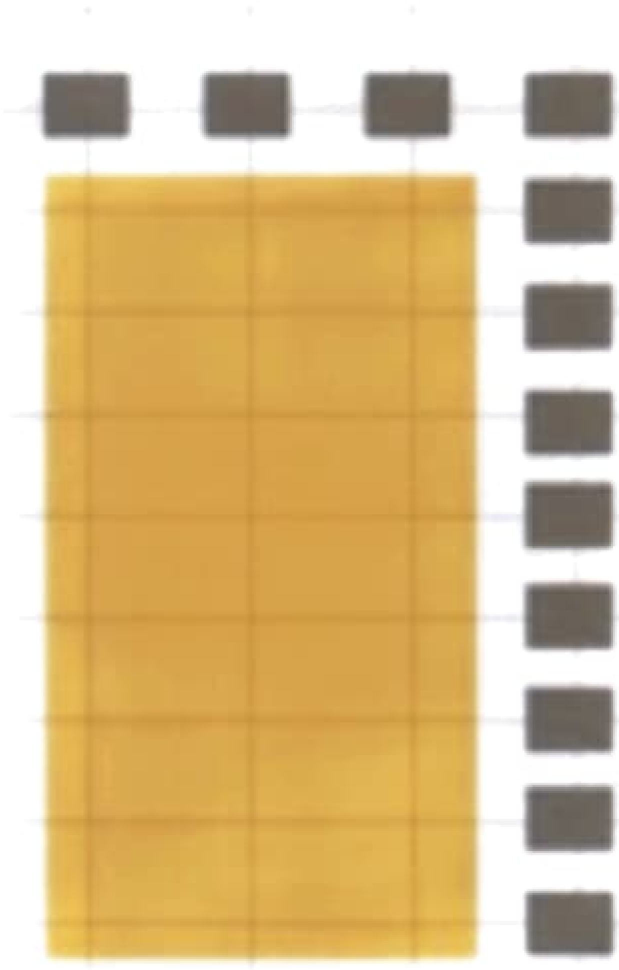
Memoization vs dynamic programming

- Holes just inside the border
- Memoization never explores the shaded region



Memoization vs dynamic programming

- Memo table has O(m+n) entries
- Dynamic programming blindly fills all O(mn) entries
- Iteration vs recursion — "wasteful" dynamic programming is still better, in general



Longest common subword

- Given two strings, find the (length of the) longest common subword
  - "secret", "secretary" — "secret", length 6
  - "bisect", "trisect" — "sect", length 4
  - "bisect", "secret" — "sec", length 3
  - "director", "secretary" — "ec", "re", length 2

More formally …

- Two strings $u = a_0a_1 \ldots a_{m-1}$, $v = b_0b_1 \ldots b_{n-1}$
- If $a_ia_{i+1} \ldots a_{i+k-1} = b_jb_{j+1} \ldots b_{j+k-1}$ for some i and j, u and v have a common subword of length k
- Aim: Find the length of the longest common subword of u and v

Longest common subword

- Given two strings, find the (length of the) longest common subword
  - “secret”, “secretary” — “secret”, length 6
  - “bisect”, “trisect” — “sect”, length 4
  - “bisect”, “secret” — “sec”, length 3
  - “director”, “secretary” — “ec”, “re”, length 2

Brute force

- $u = a_0a_1\ldots a_{m-1}$ and $v = b_0b_1\ldots b_{n-1}$
- 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 the length of the longest match
- Assuming $m > n$, this is $O(mn^2)$
    - mn pairs of positions
    - From each starting point, scan can be $O(n)$

Brute force

- $u = a_0a_1...a_{m-1}$ and $v = b_0b_1...b_{n-1}$
- Try every pair of starting positions i in u, j in v
    - Match (a_i, b_j), (a_{i+1}, b_{j+1}),... as far as possible
    - Keep track of the length of the longest match
- Assuming $m > n$, this is $O(mn^2)$
    - mn pairs of positions
    - From each starting point, scan can be $O(n)$

Inductive structure

- $a_i a_{i+1} \ldots a_{i+k-1} = b_j b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k at (i,j) iff
  - $a_i = b_j$ and
  - $a_{i+1} \ldots a_{i+k-1} = b_{j+1} \ldots b_{j+k-1}$ is a common subword of length $k-1$ at $(i+1,j+1)$
- $LCW(i,j)$: length of the longest common subword starting at a_i and b_j
  - If $a_i \neq b_j$, $LCW(i,j)$ is 0, otherwise $1+LCW(i+1,j+1)$
  - Boundary condition: when we have reached the end of one of the words

Inductive structure

- Consider positions 0 to m in u, 0 to n in v
  - m, n means we have reached the end of the word
- $LCW(m+1,j) = 0$ for all j
- $LCW(i,n+1) = 0$ for all i
- $LCW(i,j) = 0$, if $a_i \neq b_j$,

  $1 + LCW(i+1,j+1)$, if $a_i = b_j$

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or 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	•							

Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- 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

LCW(u,v), DP

```
def LCW(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCW[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCW[len(u)+1][c] = 0 # c for col
  maxLCW = 0
  for c in range(len(v)+1,-1,-1):
    for r in range(len(u)+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] > maxLCW:
        maxLCW = LCW[r][c]
  return(maxLCW)
```

LCW(u,v), DP

```
def LCW(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCW[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCW[len(u)+1][c] = 0 # c for col
  maxLCW = 0
  for c in range(len(v)+1,-1,-1):
    for r in range(len(u)+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] > maxLCW:
        maxLCW = LCW[r][c]
  return(maxLCW)
```

Complexity

- Recall that the brute force approach was $O(mn^2)$
- The inductive solution is $O(mn)$ if we use dynamic programming (or memoization)
  - Need to fill an $O(mn)$ size table
  - Each table entry takes constant time to compute

Longest common subsequence

- Subsequence: can drop some 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" — "sect", length 4
  - "director", "secretary" — "ectr", "retr", length 4

LCS

- LCS is longest path we can find between non-zero LCW entries, moving right and 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 closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

Inductive structure

u	a_0	a_1	a_2	...	...	a_{m-1}	a_m
v	b_0	b_1	b_2	...	b_{n-1}	b_n	

- If $a_0 = b_0$,

  $LCS(a_0 \ldots a_{m-1}, b_0 \ldots b_{n-1}) = 1 + LCS(a_1 a_2 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$

  - Can force (a_0, b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(a_1 a_2 \ldots a_{m-1}, b_0 b_1 \ldots b_{n-1})$ and $LCS(a_0 a_1 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$ and take the maximum

Inductive structure

u	a_0	a_1	a_2	...	...	a_{m-1}
v	b_0	b_1	b_2	...	b_{n-1}	

- If $a_0 = b_0$,

  $LCS(a_0 \ldots a_{m-1}, b_0 \ldots b_{n-1}) = 1 + LCS(a_1 a_2 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$

  - Can force (a_0, b_0) to be part of LCS

- If not, a_0 and b_0 cannot both be part of LCS

  - Not sure which one to drop

  - Solve both subproblems $LCS(a_1 a_2 \ldots a_{m-1}, b_0 b_1 \ldots b_{n-1})$ and $LCS(a_0 a_1 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$ and take the maximum

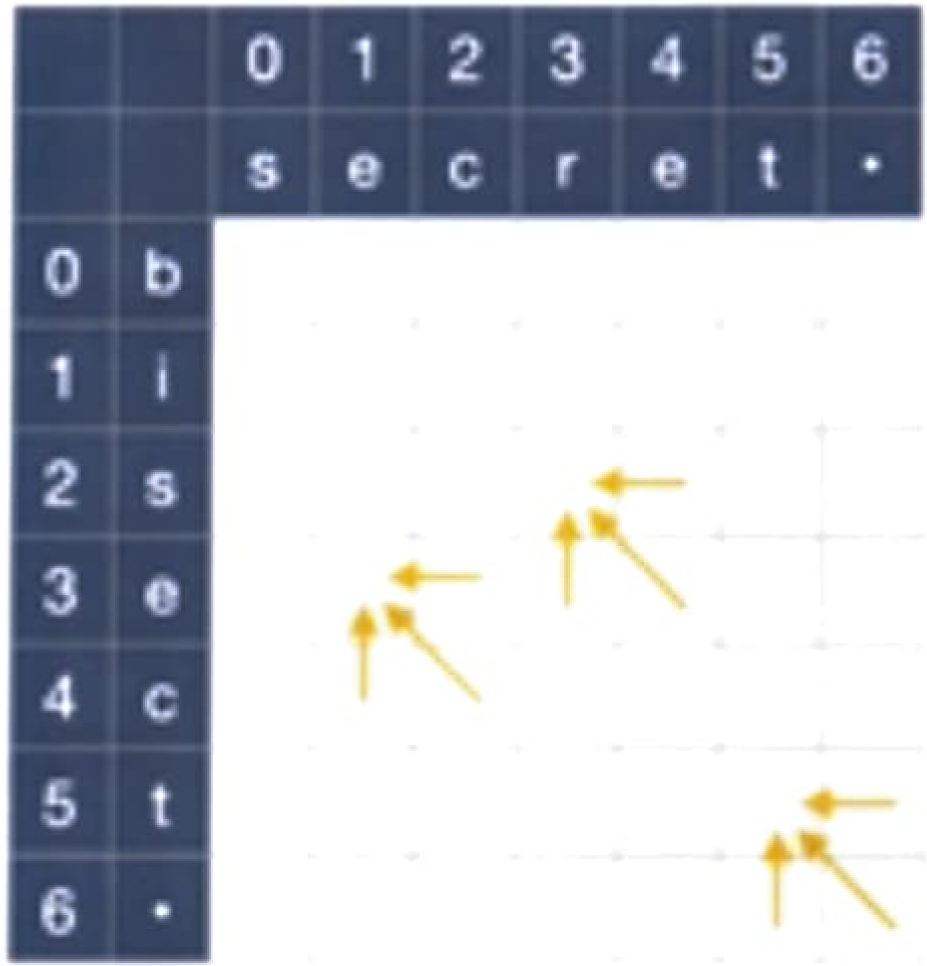
Inductive structure

u	a_i	a_{i+1}	a_{i+2}	...	...	a_{m-1}	a_m
v	b_j	b_{j+1}	b_{j+2}	...	b_{n-1}	b_n	

- LCS(i,j) stands for LCS($a_i a_{i+1} \ldots a_m$, $b_j b_{j+1} \ldots b_n$)
- 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+1,j), LCS(i,j+1))
- As with LCW, extend positions to m+1, n+1
  - LCS(m+1,j) = 0 for all j
  - LCS(i,n+1) = 0 for all i

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal



Recovering the sequence

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

		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

LCS(u,v), DP

```
def LCS(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCS[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCS[len(u)+1][c] = 0 # c for col
  for c in range(len(v),-1,-1):
    for r in range(len(u),-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)
  - Need to fill an O(mn) size table
  - Each table entry takes constant time to compute

Multiplying matrices

- To multiply matrices A and B, need compatible dimensions
  - A of dimension m x n, B of dimension n x p
  - AB has dimension mp
- Each entry in AB take O(n) steps to compute
  - AB[i,j] is A[i,1]B[1,j] + A[i,2]B[2,j] +...+ A[i,n]B[n,j]
- Overall, computing AB is O(mnp)

Multiplying matrices

- Matrix multiplication is associative
  - ABC = (AB)C = A(BC)
  - Bracketing does not change the answer …
  - … but can affect the complexity of computing it!

Multiplying matrices

- Matrix multiplication is associative
  - $ABC = (AB)C = A(BC)$
  - Bracketing does not change the answer …
  - … but can affect the complexity of computing it!

Multiplying matrices

- Suppose dimensions are A[1,100], B[100,1], C[1,100]
  - Computing A(BC)
    - BC is [100,100], 100 x 1 x 100 = 10000 steps
    - A(BC) is [1,100], 1 x 100 x 100 = 10000 steps
  - Computing (AB)C
    - AB is [1,1], 1 x 100 x 1 = 100 steps
    - (AB)C is [1,100], 1 x 1 x 100 = 100 steps
- A(BC) takes 20000 steps, (AB)C takes 200 steps!

Multiplying matrices

- Suppose dimensions are A[1,100], B[100,1], C[1,100]
  - Computing A(BC)
    - BC is [100,100], 100 x 1 x 100 = 10000 steps
    - A(BC) is [1,100], 1 x 100 x 100 = 10000 steps
  - Computing (AB)C
    - AB is [1,1], 1 x 100 x 1 = 100 steps
    - (AB)C is [1,100], 1 x 1 x 100 = 100 steps
- A(BC) takes 20000 steps, (AB)C takes 200 steps!

Multiplying matrices

- Given matrices $M_1, M_2, \ldots, M_n$ of dimensions $[r_1,c_1]$, $[r_2,c_2]$, ..., $[r_n,c_n]$
  - Dimensions match, so $M_1 \times M_2 \times \ldots \times M_n$ can be computed
  - $c_i = r_{i+1}$ for $1 \leq i < n$
- Find an optimal order to compute the product
  - That is, bracket the expression optimally

Inductive structure

- Product to be computed: $M_1 \times M_2 \times \ldots \times M_n$
- Final step would have combined two subproducts
  - $(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_n)$, for some $1 \leq k < n$
  - First factor has dimension (r_1, c_k), second (r_{k+1}, c_n)
  - Final multiplication step costs $O(r_1 c_k c_n)$
  - Add cost of computing the two factors

Subproblems

- Final step is
  $(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times M_{k+2} \times \ldots \times M_n)$
- Subproblems are $(M_1 \times M_2 \times \ldots \times M_k)$ and
  $(M_{k+1} \times M_{k+2} \times \ldots \times M_n)$
- Total cost is $\mathrm{Cost}(M_1 \times M_2 \times \ldots \times M_k)$ +
  $\mathrm{Cost}(M_{k+1} \times M_{k+2} \times \ldots \times M_n) + r_1 c_k c_n$
- Which k should we choose?
- No idea! Try them all and choose the minimum!

Inductive formulation

- Cost($M_1 \times M_2 \times \ldots \times M_n$) =

  minimum value, for $1 \leq k < n$, of

  Cost($M_1 \times M_2 \times \ldots \times M_k$) +

  Cost($M_{k+1} \times M_{k+2} \times \ldots \times M_n$) +

  $r_1 c_k c_n$

- When we compute Cost($M_1 \times M_2 \times \ldots \times M_k$) we will get subproblems of the form $M_j \times M_{j+1} \times \ldots \times M_k$

In general ...

- $\text{Cost}(M_i \times M_{i+1} \times \ldots \times M_j) =$

  minimum value, for $i \le k < j$, of

  $\text{Cost}(M_i \times M_{i+1} \times \ldots \times M_k) +$

  $\text{Cost}(M_{k+1} \times M_{k+2} \times \ldots \times M_j) +$

  $r_i c_k c_j$

- Write Cost(i,j) to denote $\text{Cost}(M_i \times M_{i+1} \times \ldots \times M_j)$

Final equation

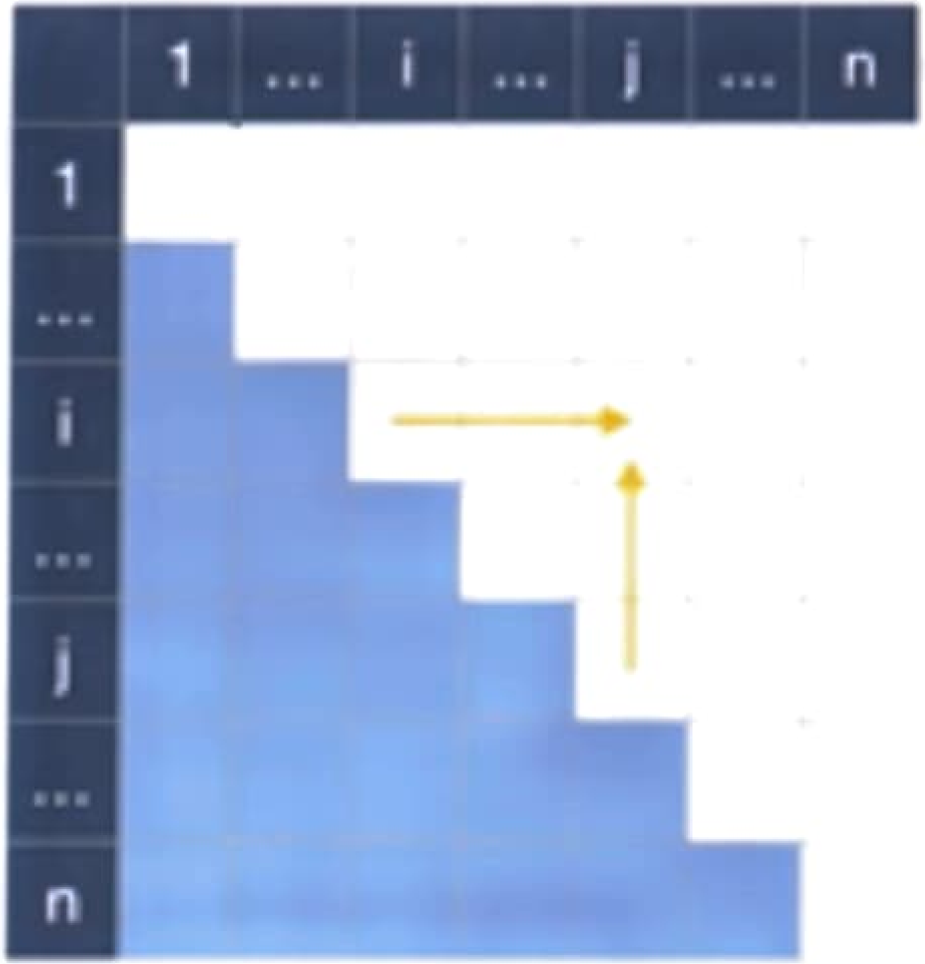
- $Cost(i,i) = 0$ — No multiplication to be done
- $Cost(i,j) = \min$ over $i \le k < j$

  $$[\, Cost(i,k) + Cost(k+1,j) + r_i c_k c_j \,]$$

- Note that we only require $Cost(i,j)$ when $i \le j$

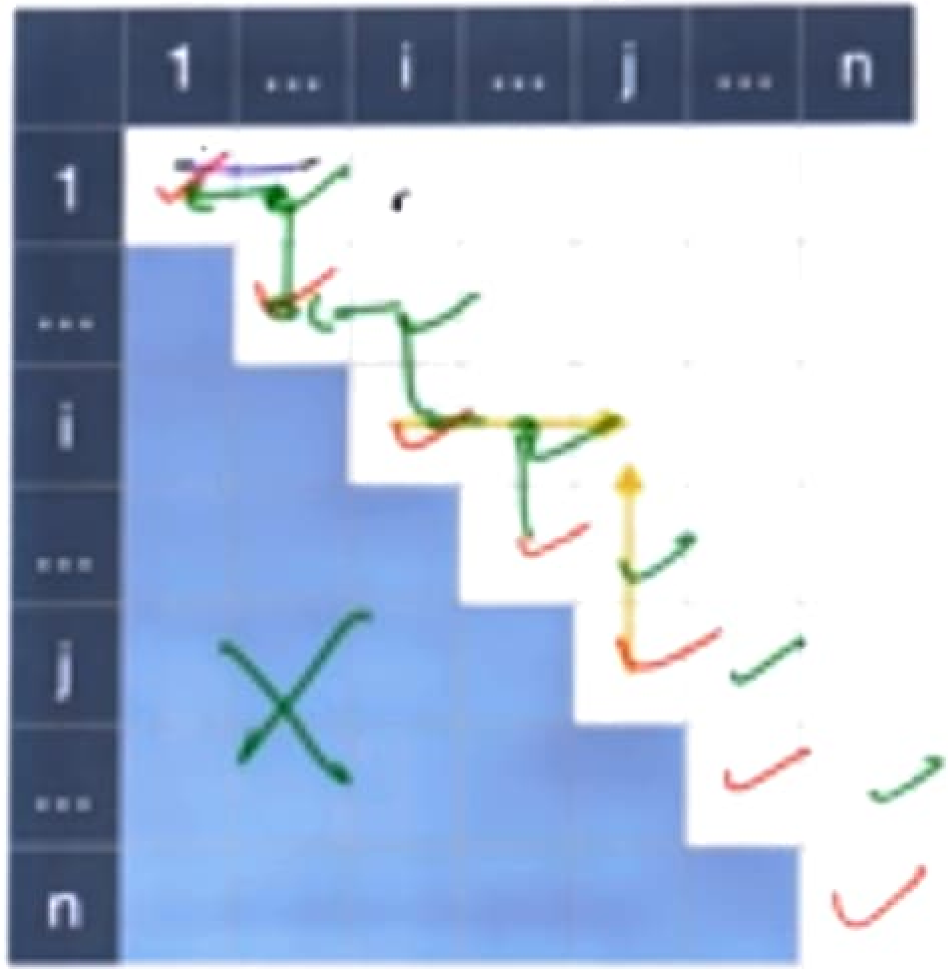
Subproblem dependency

- Cost(i,j) depends on Cost(i,k), Cost(k+1,j) for all i ≤ k < j
- Can have O(n) dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right



Subproblem dependency

- Cost(i,j) depends on Cost(i,k), Cost(k+1,j) for all $i \le k < j$
- Can have O(n) dependent values, unlike LCS, LCW
- Start with main diagonal and fill matrix by columns, bottom to top, left to right



MMCost(M1,...,Mn), DP

```
def MMC(R,C):
  # R[0..n-1],C[0..n-1] have row/column sizes

  for r in range(len(R)):
    MMC[r][r] = 0

  for c in range(1,len(R)):     # c = 1,2,…n-1
    for r in range(c-1,-1,-1):# r = c,c-1,…,0
      MMC[r][c] = infinity  # Something large
      for k in range(r,c)   # k = r,r+1,…,c-1
        subprob = MMC[r][k] + MMC[k][c] +
                                      R[r]C[k]C[c]
        if subprob < MMC[r][c]:
          MMC[r][c] = subprob
```

Complexity

- As with LCS, ED, we to fill an $O(n^2)$ size table
- However, filling MMC[i][j] could require examining $O(n)$ intermediate values
- Hence, overall complexity is $O(n^3)$

Complexity

- As with LCS, ED, we to fill an $O(n^2)$ size table
- However, filling MMC[i][j] could require examining $O(n)$ intermediate values
- Hence, overall complexity is $O(n^3)$

Python vs other languages

- Python is a good programming language to start with because
  - No declaration of names in advance
  - Indentation avoids punctuation – { }, ( ), ;
  - No explicit memory management
- Are there any down sides to this?

Debugging

- Declaring names helps debug code
  - "Simple" typos are caught by compiler
  - Mistyped name will be "undeclared"
- Static typing — assigning types to names
  - Again catch "simple" typos by type mismatch

Classes and objects

- Can only associate a type with a name by creating an object
- Empty tree, with name and type declarations
  - Declare `t` to be of type `Tree`
  - Empty tree — `t` has value `None`
- Instead, cumbersome convention with empty nodes to denote frontier etc

Classes and objects

- We want public interface, private implementation
- For a `Point p`, `p.x` and `p.y` should not be available directly outside the class
  - Stack implemented as a list has public methods `push()` and `pop()` but `s.append()` not ruled out
- Need to declare parts of implementation private
  - Only methods inside the class can access private names

Classes and objects

- Ideally, all internal names are private
- Special functions to access and update values
  - `p.getx()` gets x-coordinate
  - `p.setx(v)` sets x-coordinate
- x-coordinate is an "abstract" attribute
- Works even if internal representation is (r,θ)

Classes and objects

- Handle integrity of compound values
- Date is a tuple (day,[illegible])
    - Range for day is 1[illegible]th is 1—12
    - Valid combinations depend on all three fields
        - 29 - 02 is valid only in a leap year
- `d.setdate(d,m,y)` vs separate `d.setd(d)`, `d.setm(m)`, `d.sety(y)`

Classes and objects

- Handle integrity of compound values
- Date is a tuple (day,month,year)
  - Range for day is $1-31$, month is $1-12$
  - Valid combinations depend on all three fields
    - 29 - 02 is valid only in a leap year
- `d.setdate(d,m,y)` vs separate `d.setd(d)`, `d.setm(m)`, `d.sety(y)`

Storage allocation

- Python needs to allocate space dynamically
  - Each assignment to a name could a new type
- Name declarations allow some static allocation
  - Still need dynamic allocation for lists, trees etc that grow at run time

Storage allocation

- Python needs to allocate space dynamically
  - Each assignment to a name could a new type
- Name declarations allow some static allocation
  - Still need dynamic allocation for lists, trees etc that grow at run time
  - Static arrays can optimize access time: base address plus offset

Dynamic storage

- What happens when we execute `del(x)`?
- Or when we delete a list node by bypassing it?
- Do these "dead" values continue to use memory?

Garbage collection

- Python, Java and other languages reclaim space using automatic "garbage collection"
  - Periodically mark all memory reachable from names in use in the program
  - Collect all unmarked memory locations as free space
  - Run time overhead to schedule garbage collector
- In C, need to explicitly ask for and return dynamic memory

Functional programming

- Declarative vs imperative
- "What to compute" vs "how to compute it"
- Directly specify functions inductively


```
factorial :: Int -> Int # Type

factorial 0 = 1
factorial n = n * factorial (n-1)
```

Functional programming

- List processing

```
sumlist :: [Int]

sumlist [] = 0
sumlist l = (head l) + sumlist (tail l)
```

Functional programming

- List processing

```
sumlist :: [Int] -> Int

sumlist [] = 0
sumlist l = (head l) + sumlist (tail l)
```

Summary

- No programming language is "universally" the best
  - Otherwise why are there so many?
- Python's simplicity makes it attractive to learn
  - But also results in some limitations
- Use the language that suits your task best
- Learn programming, not programming languages!