Lecture 25, 12 November 2026

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming and Data Structures with Python

Lecture 25, 12 Nov 2026

Height balanced trees

- `self.height()` — number of nodes on longest path from root to leaf
  - 0 for empty tree
  - 1 for tree with only a root node
  - 1 + max of heights of left and right subtrees, in general
- Height balance
  - `self.left.height()` and `self.right.height()` differ by at most 1
  - AVL trees — Adelson-Velskii, Landis
- Does height balance guarantee $O(\log n)$ height?

Balanced Search Tree

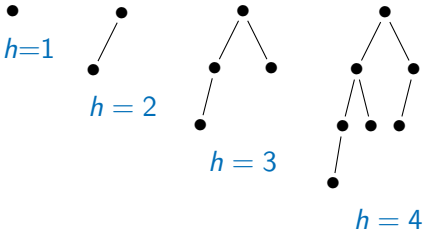
$size \simeq 2^h$

$h \simeq \log(size)$

Size balance

Height balanced trees

- Minimum size height-balanced trees



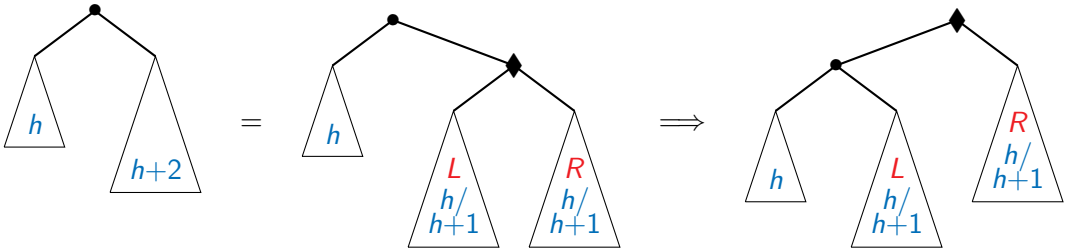
- General strategy to build a small balanced tree of height h
  - Smallest balanced tree of height $h-1$ as left subtree
  - Smallest balanced tree of height $h-2$ as right subtree

- $S(h)$, size of smallest height-balanced tree of height h
- Recurrence
  - $S(0) = 0$, $S(1) = 1$
  - $S(h) = 1 + S(h-1) + S(h-2)$
- Compare to Fibonacci sequence
  - $F(0) = 0$, $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$
- $S(h)$ grows exponentially with h
- For size n, h is $O(\log n)$

Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

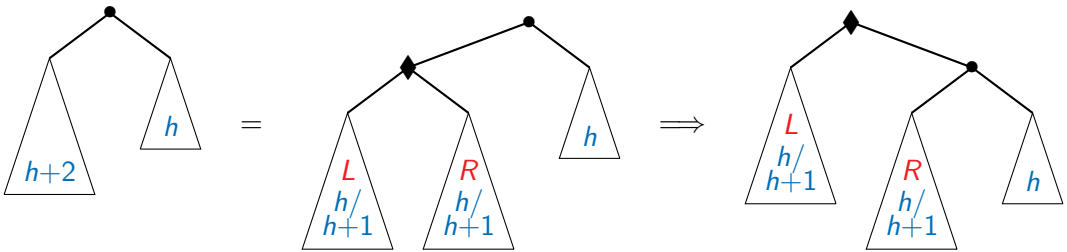
Left rotation — converts slope -2 to $\{0, 1, 2\}$



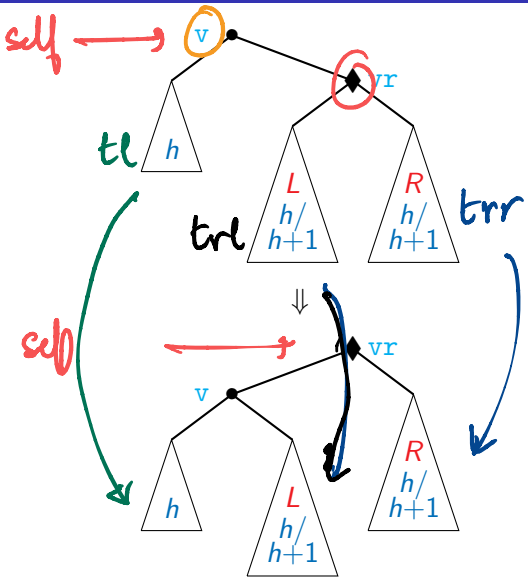
Correcting imbalance

- Slope of a node : `self.left.height() - self.right.height()`
- Balanced tree — slope is $\{-1, 0, 1\}$
- `t.insert(v)`, `t.delete(v)` can alter slope to -2 or $+2$

Right rotation — converts slope $+2$ to $\{-2, -1, 0\}$



Implementing rotations




```
class Tree:
  ...

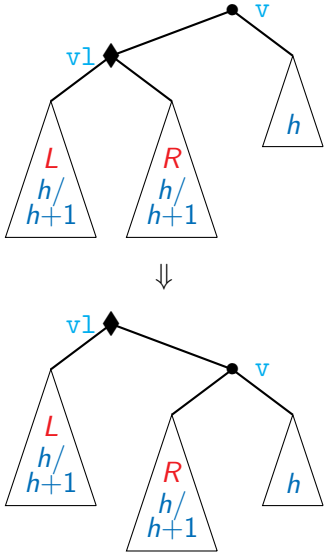
  def leftrotate(self):
    v = self.value
    vr = self.right.value
    tl = self.left
    trl = self.right.left
    trr = self.right.right

    newleft = Tree(v)
    newleft.left = tl
    newleft.right = trl

    self.value = vr
    self.left = newleft
    self.right = trr

    return
```

Implementing rotations




```
class Tree:
  ...

  def rightrotate(self):
    v = self.value
    vl = self.left.value
    tll = self.left.left
    tlr = self.left.right
    tr = self.right

    newright = Tree(v)
    newright.left = tlr
    newright.right = tr

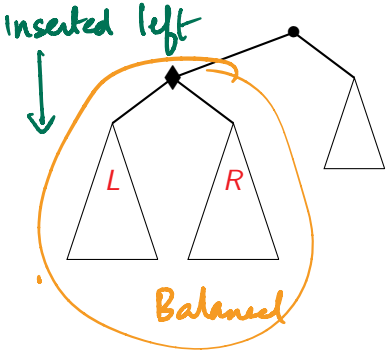
    self.value = vl
    self.left = tll
    self.right = newright

    return
```

Rebalancing, root has slope +2

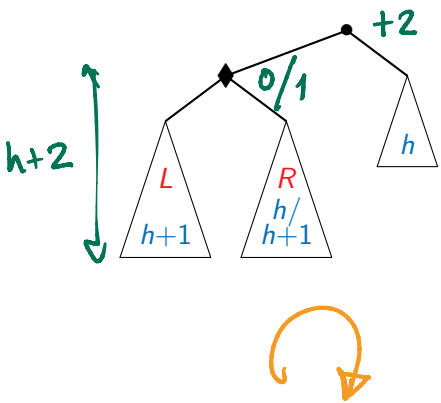
- Rebalance bottom-up, assume subtrees are balanced





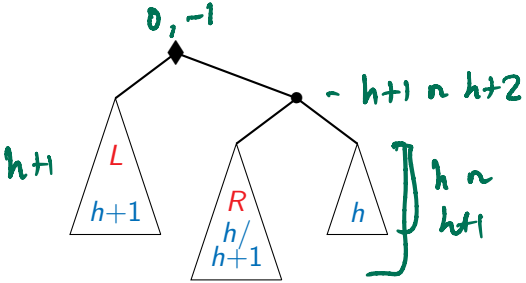
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$



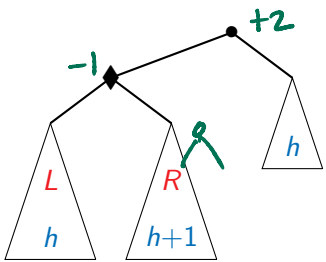
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced



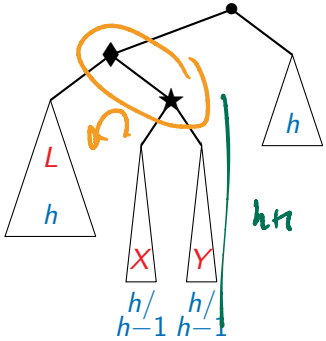
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is -1



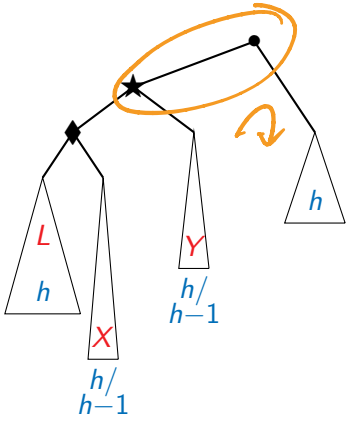
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is -1
  - Expand R



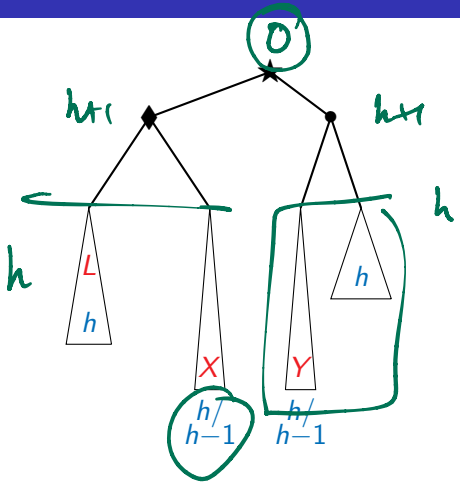
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is -1
  - Expand R
  - Rotate left at ♦



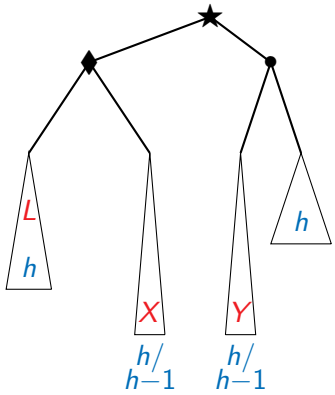
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is -1
  - Expand R
  - Rotate left at ♦
  - Rotate left at •



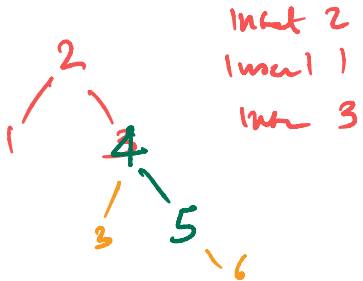
Rebalancing, root has slope +2

- Rebalance bottom-up, assume subtrees are balanced
- Case 1: Slope at ♦ is in $\{0, 1\}$
  - Rotate right at •
  - All nodes are balanced
- Case 2: Slope at ♦ is -1
  - Expand R
  - Rotate left at ♦
  - Rotate left at •
- Rebalance with root slope -2 is symmetric



Update `insert()` and `delete()`

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up




```
class Tree:
    ...
    def insert(self,v):
        if self.isempty():
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v:
            return

        if v < self.value:
            self.left.insert(v)
            self.left.rebalance()
            return

        if v > self.value:
            self.right.insert(v)
            self.right.rebalance()
            return
```



Update `insert()` and `delete()`

- Use the rebalancing strategy to define a function `rebalance()`
- Rebalance each time the tree is modified
- Automatically rebalances bottom up

```
class Tree:
    ...
    def delete(self,v):
        ...
        if v < self.value:
            self.left.delete(v)
            self.[redacted].rebalance()
            return
        if v > self.value:
            self.right.delete(v)
            self.[redacted].rebalance()
            return
        if v == self.value:
            if self.isleaf():
                self.makeempty()
            elif self.left.isempty():
                self.copyright()
            elif self.right.isempty():
                self.copyleft()
            else:
                self.value = self.left.maxval()
                self.left.delete(self.left.maxval())
            return
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$

```
class Tree:
    ...
    def height(self):
        if self.isempty():
            return(0)
        else:
            return(1 +
                   max(self.left.height(),
                       self.right.height())
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`

```
class Tree:
    ...
    def height(self):
        if self.isempty():
            return(0)
        else:
            return(1 +
                   max(self.left.height(),
                       self.right.height())
```

Computing slope

- To compute the slope we need heights of subtrees
- But, computing height is $O(n)$
- Instead, maintain a field `self.height`
- After each modification, update `self.height` based on `self.left.height`, `self.right.height`

```
class Tree:
    ...
    def insert(self,v):
        ...
        if v < self.value:
            self.left.insert(v)
            self.[redacted].rebalance()
            self.height = 1 +
                          max(self.left.height,
                              self.right.height)
            return

        if v > self.value:
            self.right.insert(v)
            self.[redacted].rebalance()
            self.height = 1 +
                          max(self.left.height,
                              self.right.height)
            return
```

Summary

- Using rotations, we can maintain height balance
- Height balanced trees have height $O(\log n)$
- `find()`, `insert()` and `delete()` all walk down a single path, take time $O(\log n)$



Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$

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

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$

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

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

Inductive definitions, recursive programs, subproblems

- Factorial
  - $fact(0) = 1$
  - $fact(n) = n \times fact(n-1)$

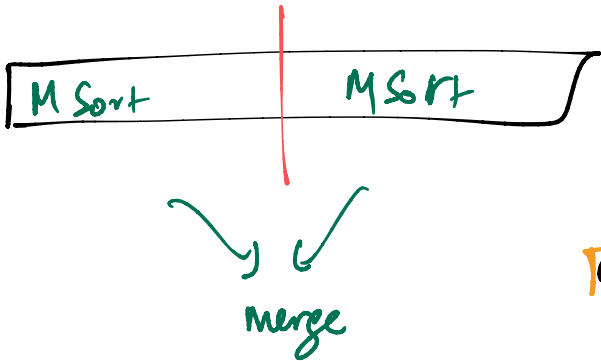
```
def fact(n):
  if n <= 0:
    return(1)
  else:
    return(n * fact(n-1))
```

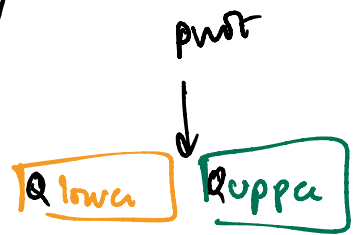
- Insertion sort
  - $isort([\,]) = [\,]$
  - $isort([x_0, x_1, \ldots, x_n]) = insert(isort([x_0, x_1, \ldots, x_{n-1}]), x_n)$

- $fact(n-1)$ 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 $isort([x_0, \ldots, x_j])$ for any $0 < j < n$
- Solution to original problem can be derived by combining solutions to subproblems

Merge sort ~ Quick sort

Disjoint subproblems

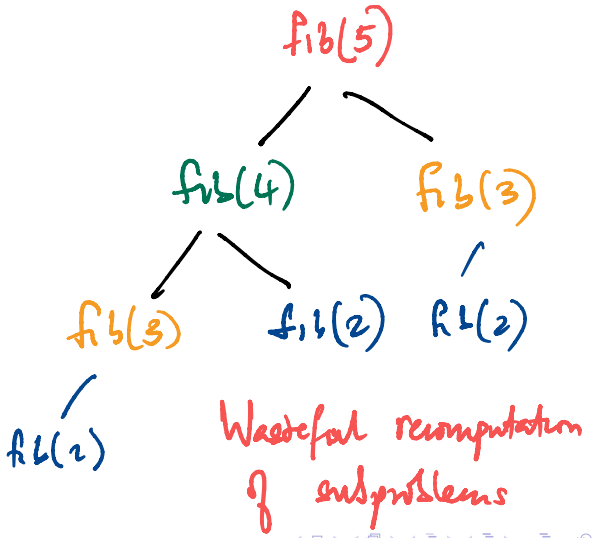




Evaluating subproblems

- 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)
```



Evaluating subproblems

- Build a table of values already computed
  - Memory table

Evaluating subproblems

- Build a table of values already computed
  - Memory table
- Memoization
  - Check if the value to be computed was already seen before

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

Memoizing recursive implmentations

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

Memoizing recursive implmentations

```
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)
```

Memoizing recursive implmentations

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

0 1 1 2 3 5 8 – –

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

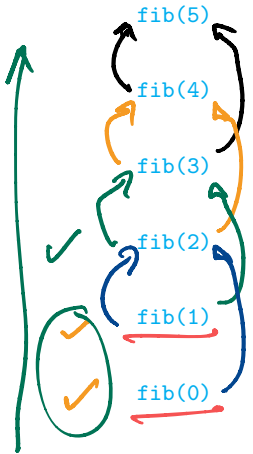


Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

Need to understand the inductive dependency to do this

Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available
  - Fill table iteratively
  - Never need to make a recursive call

Evaluating `fib(5)`

`fib(5)`

`fib(4)`

`fib(3)`

`fib(2)`

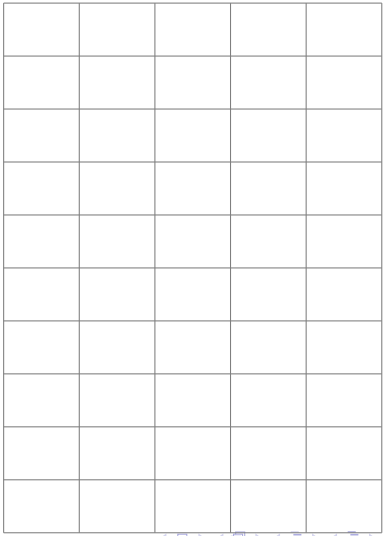
`fib(1)`

`fib(0)`

Grid paths

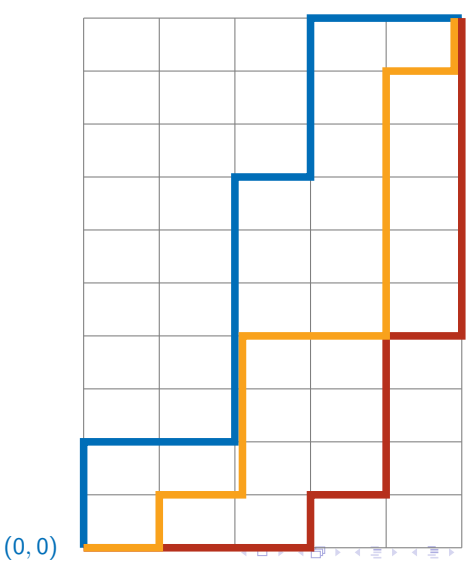
- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from $(0, 0)$ to (m, n)?





Combinatorial solution

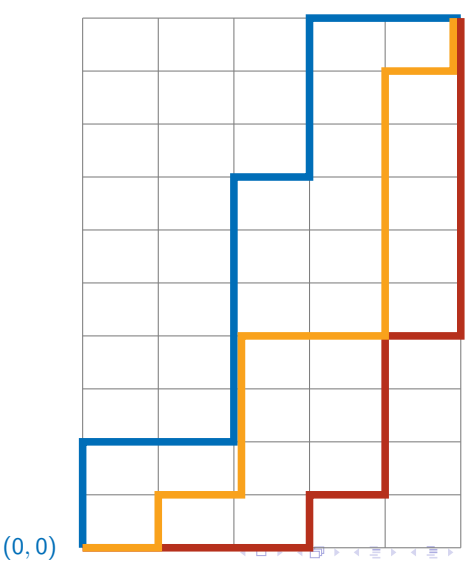
- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - Out of 15, exactly 5 are right moves, 10 are up moves
  - Fix the positions of the 5 right moves among the 15 positions overall



Combinatorial solution

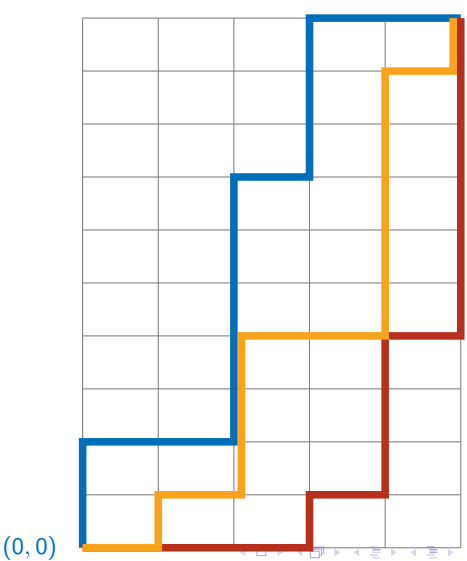
- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - Out of 15, exactly 5 are right moves, 10 are up moves
  - Fix the positions of the 5 right moves among the 15 positions overall
  - $\binom{15}{5} = \frac{15!}{10! \cdot 5!} = 3003$
  - Same as $\binom{15}{10}$ — fix the 10 up moves

$$\binom{n}{r} = \binom{n}{n-r}$$



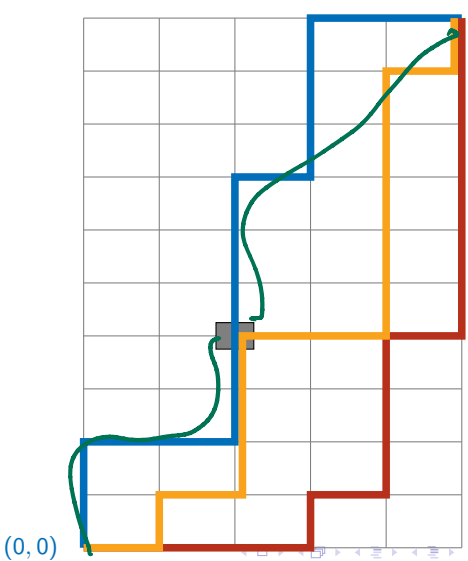
Combinatorial solution

- Every path from $(0, 0)$ to $(5, 10)$ has 15 segments
  - Out of 15, exactly 5 are right moves, 10 are up moves
  - Fix the positions of the 5 right moves among the 15 positions overall
  - $\binom{15}{5} = \frac{15!}{10! \cdot 5!} = 3003$
  - Same as $\binom{15}{10}$ — fix the 10 up moves
- In general $m+n$ segments from $(0, 0)$ to (m, n)



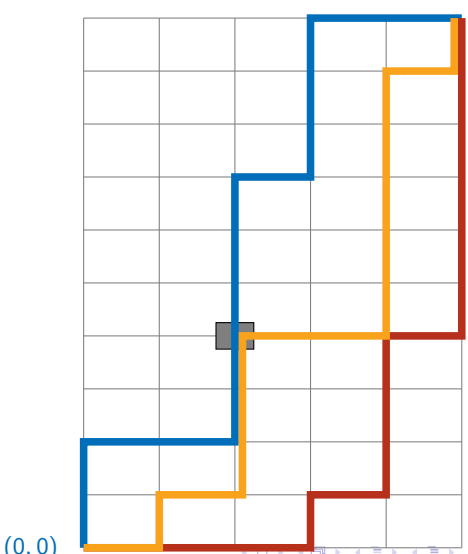
Holes

- What if an intersection is blocked?
  - For instance, $(2, 4)$



Combinatorial solution for holes

- Discard paths passing through $(2, 4)$

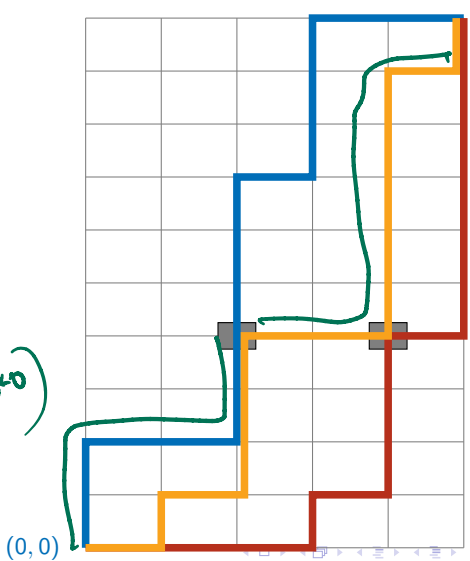


More holes

- 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-exclusion — counting is messy

Bad $(0,0 \to 2,4) \times (2,4 \to 5,10)$

Total − Bad



Inductive formulation

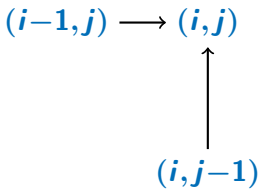
- How can a path reach (i, j)

Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$

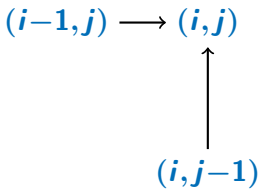
Inductive formulation

- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$



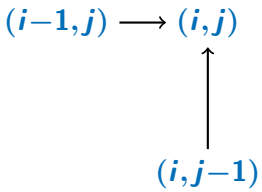
Inductive formulation

- 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)



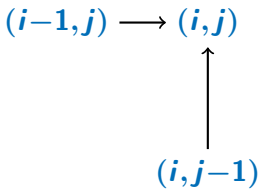
Inductive formulation

- 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)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$



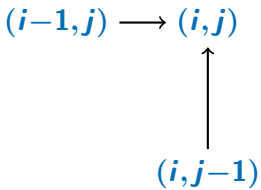
Inductive formulation

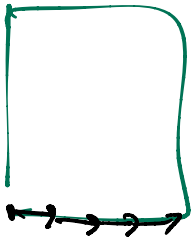
- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case



Inductive formulation

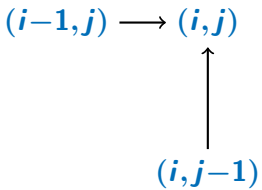
- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case
  - $P(i,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

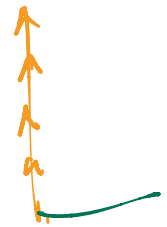




Inductive formulation

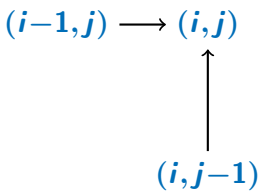
- How can a path reach (i, j)
  - Move up from $(i, j-1)$
  - Move right from $(i-1, j)$
- Each path to these neighbours extends to a unique path to (i, j)
- Recurrence for $P(i, j)$, number of paths from $(0, 0)$ to (i, j)
  - $P(i, j) = P(i-1, j) + P(i, j-1)$
  - $P(0, 0) = 1$ — base case
  - $P(i,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





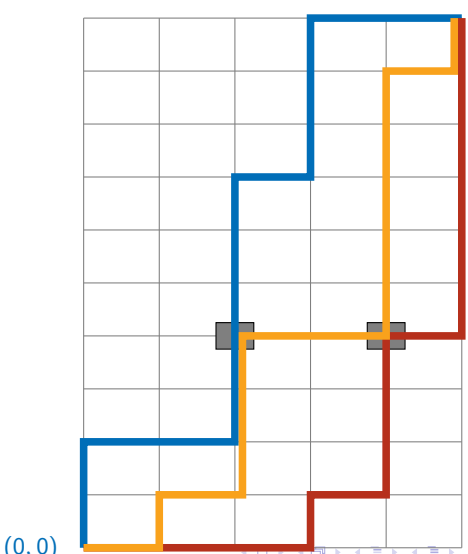
Inductive formulation

- 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)



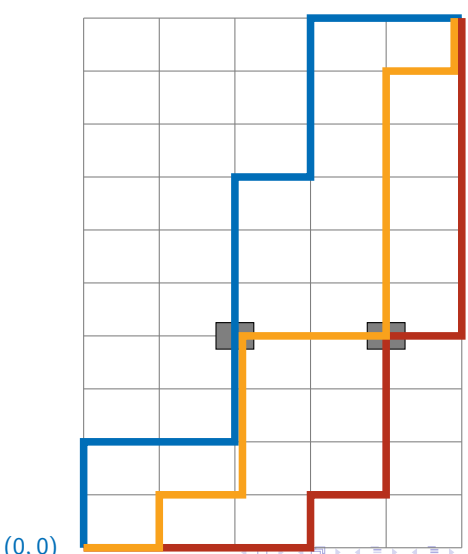
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly



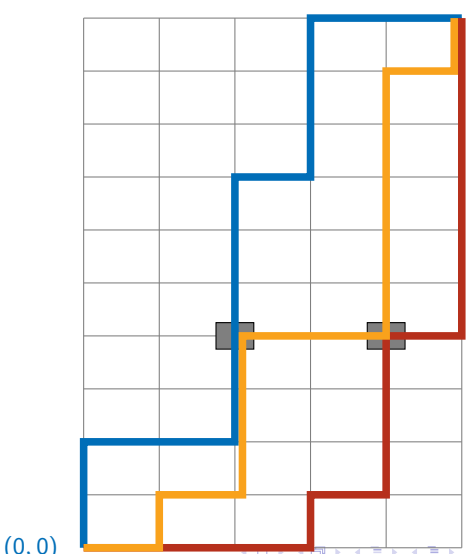
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$



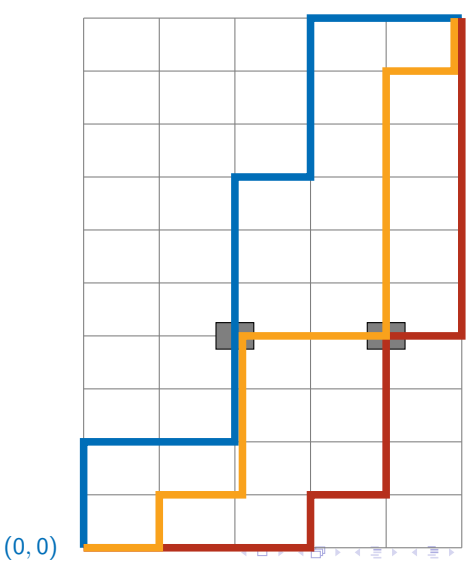
Computing $P(i,j)$

- Naive recursion recomputes same subproblem repeatedly
  - $P(5,10)$ requires $P(4,10)$, $P(5,9)$
  - Both $P(4,10)$, $P(5,9)$ require $P(4,9)$



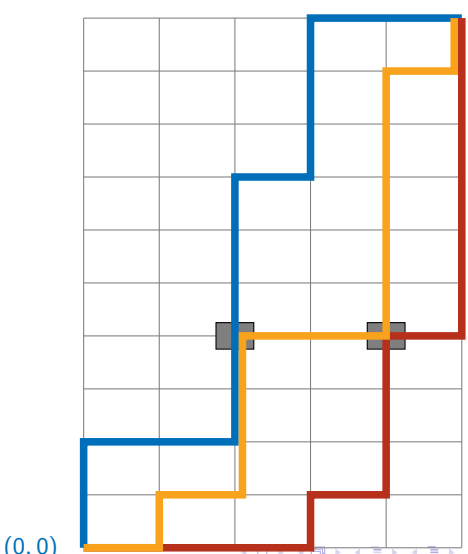
Computing $P(i,j)$

- 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 ...



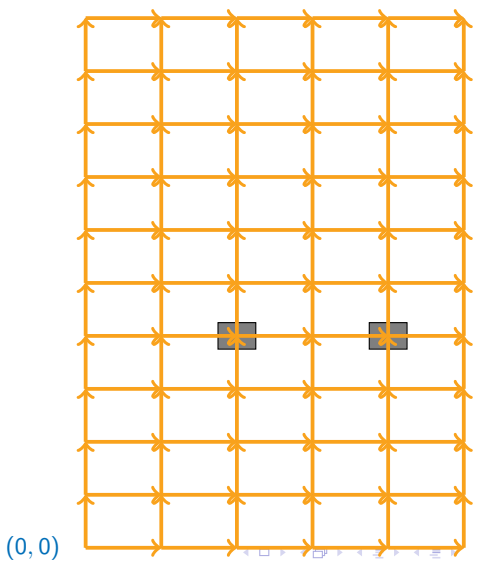
Computing $P(i,j)$

- 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



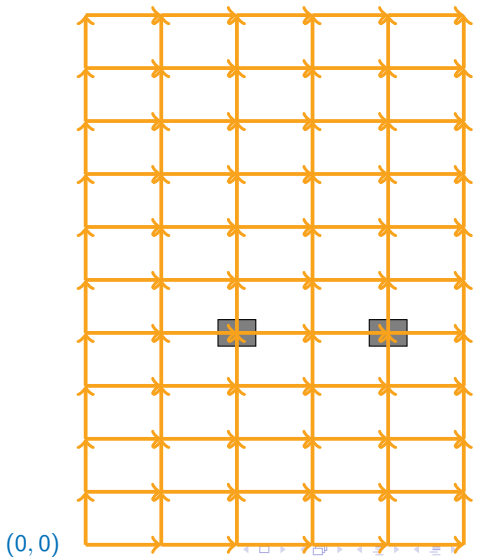
Dynamic programming

- Identify subproblem structure



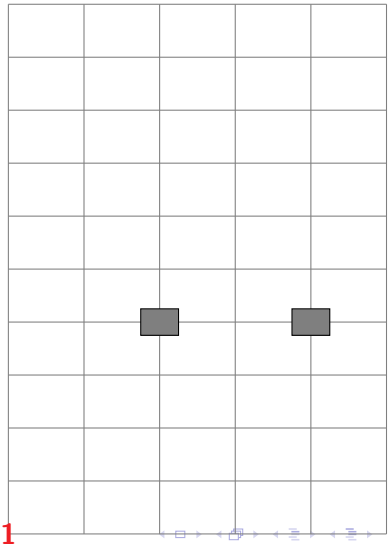
Dynamic programming

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



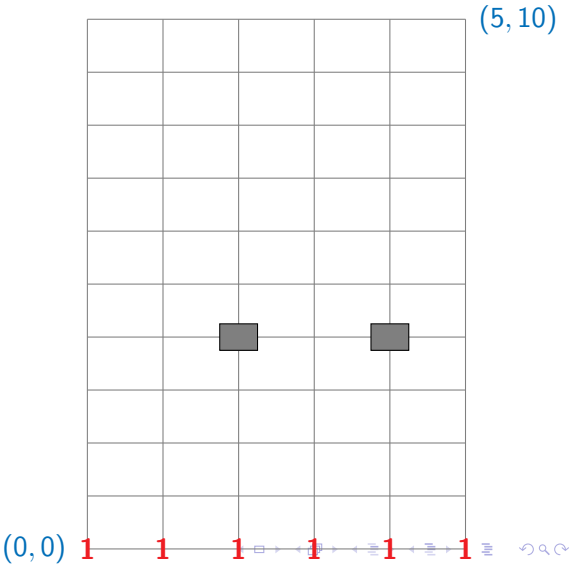
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$



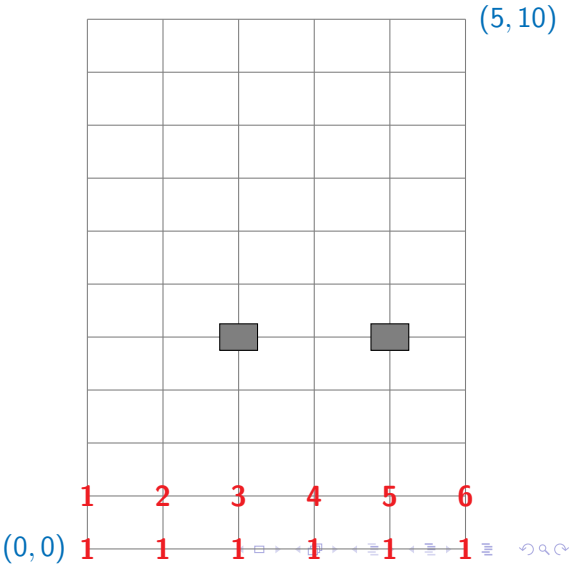
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



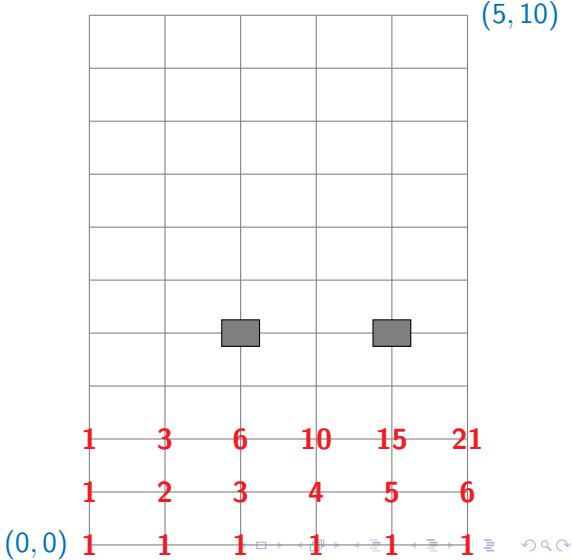
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



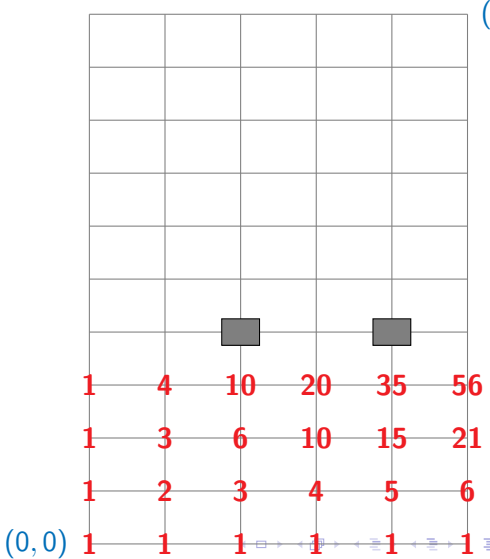
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



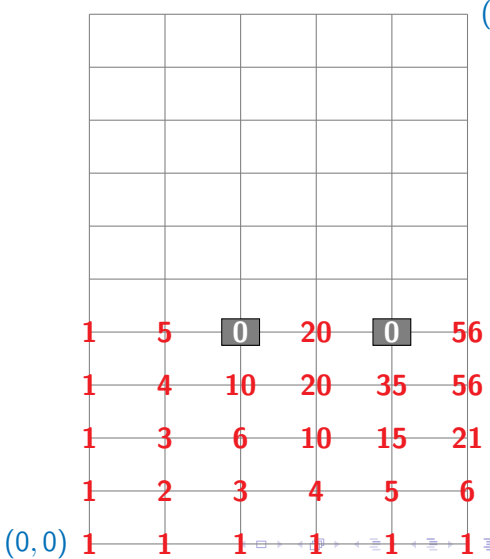
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



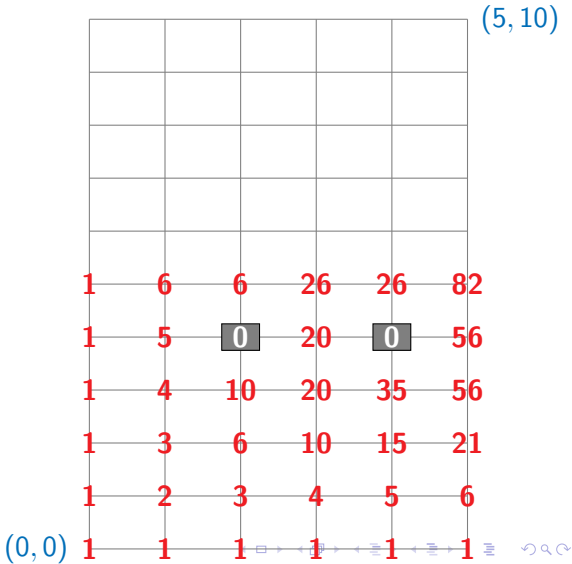
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



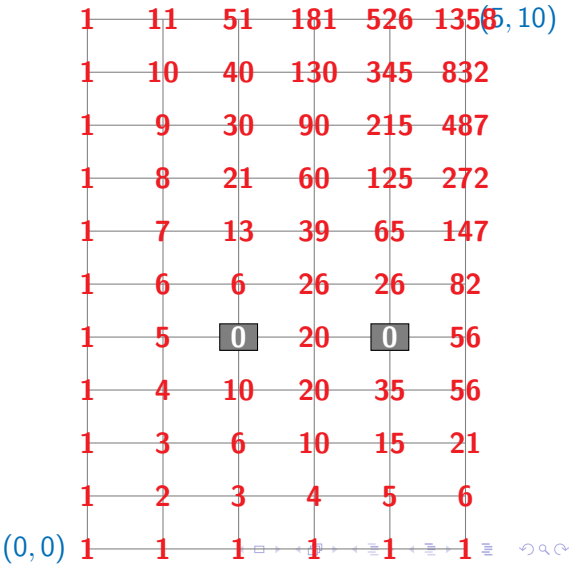
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



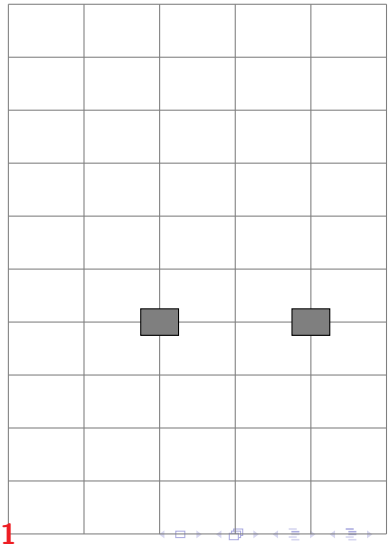
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row



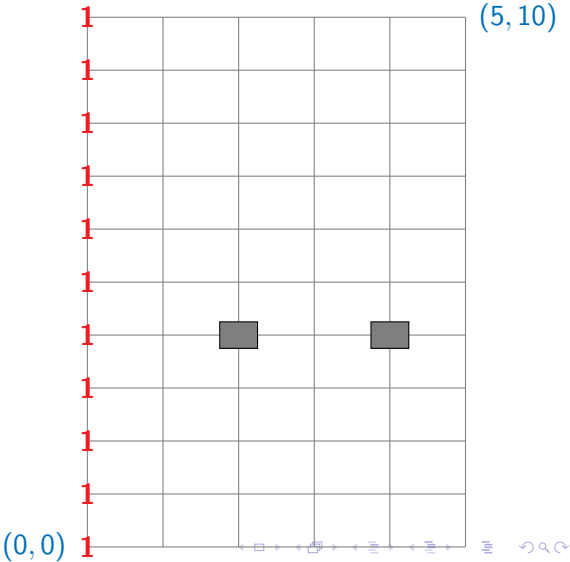
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



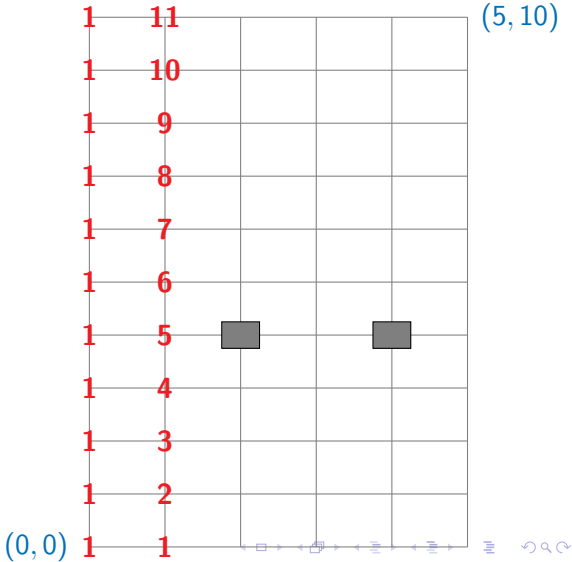
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



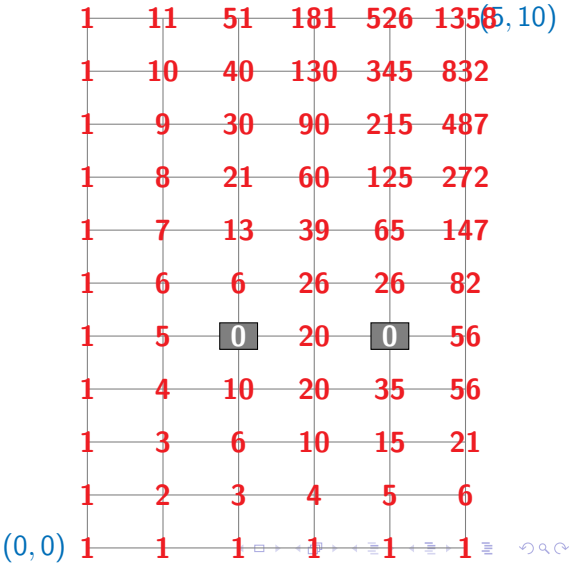
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



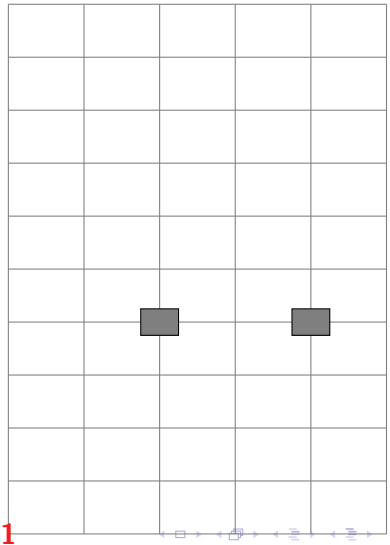
Dynamic programming

- Identify suproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column



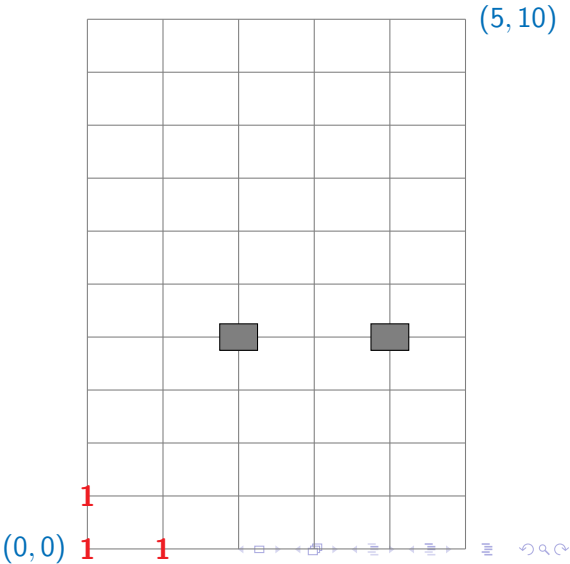
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



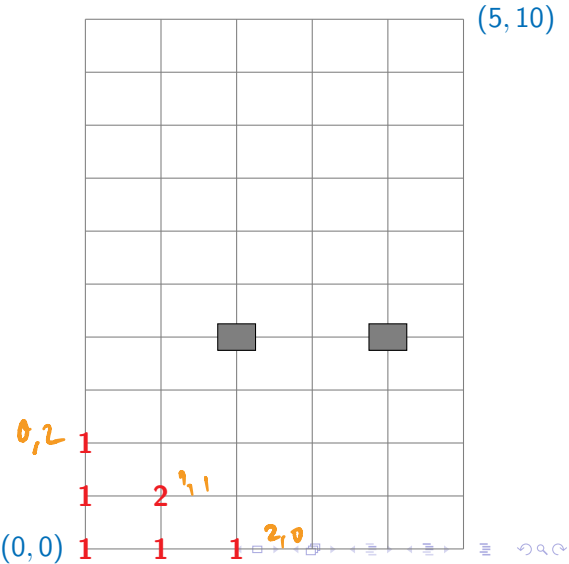
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



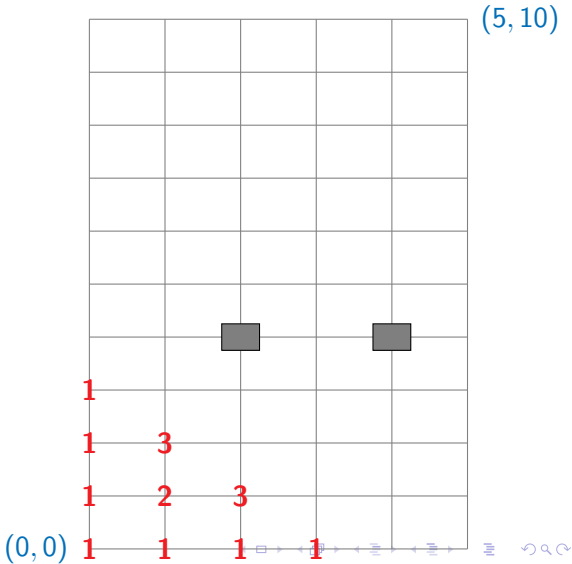
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



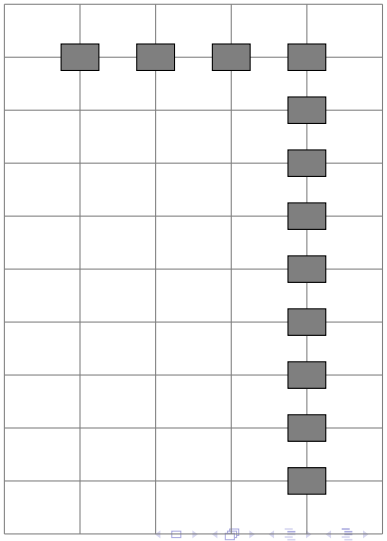
Dynamic programming

- Identify subproblem structure
- $P(0,0)$ has no dependencies
- Start at $(0,0)$
- Fill row by row
- Fill column by column
- Fill diagonal by diagonal



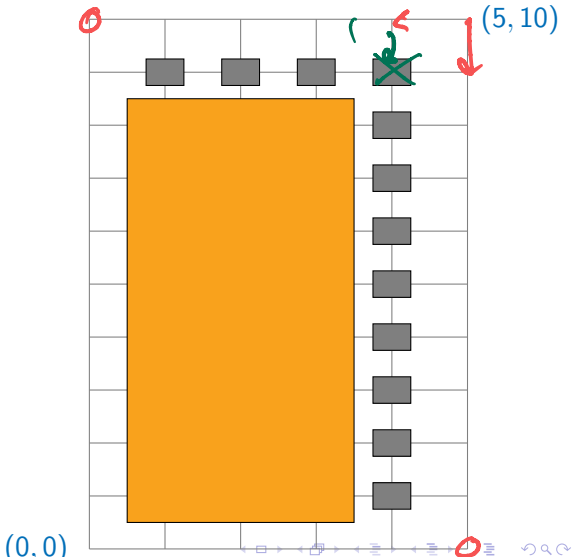
Memoization vs dynamic programming

- Barrier of holes just inside the border



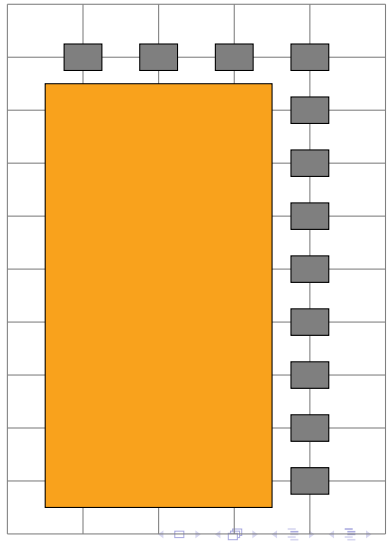
Memoization vs dynamic programming

- Barrier of holes just inside the border
- Memoization never explores the shaded region



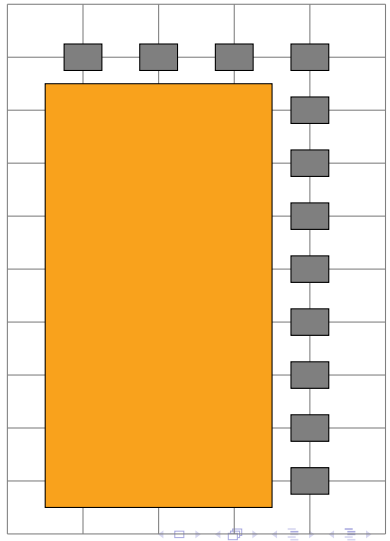
Memoization vs dynamic programming

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



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



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

