Recursion

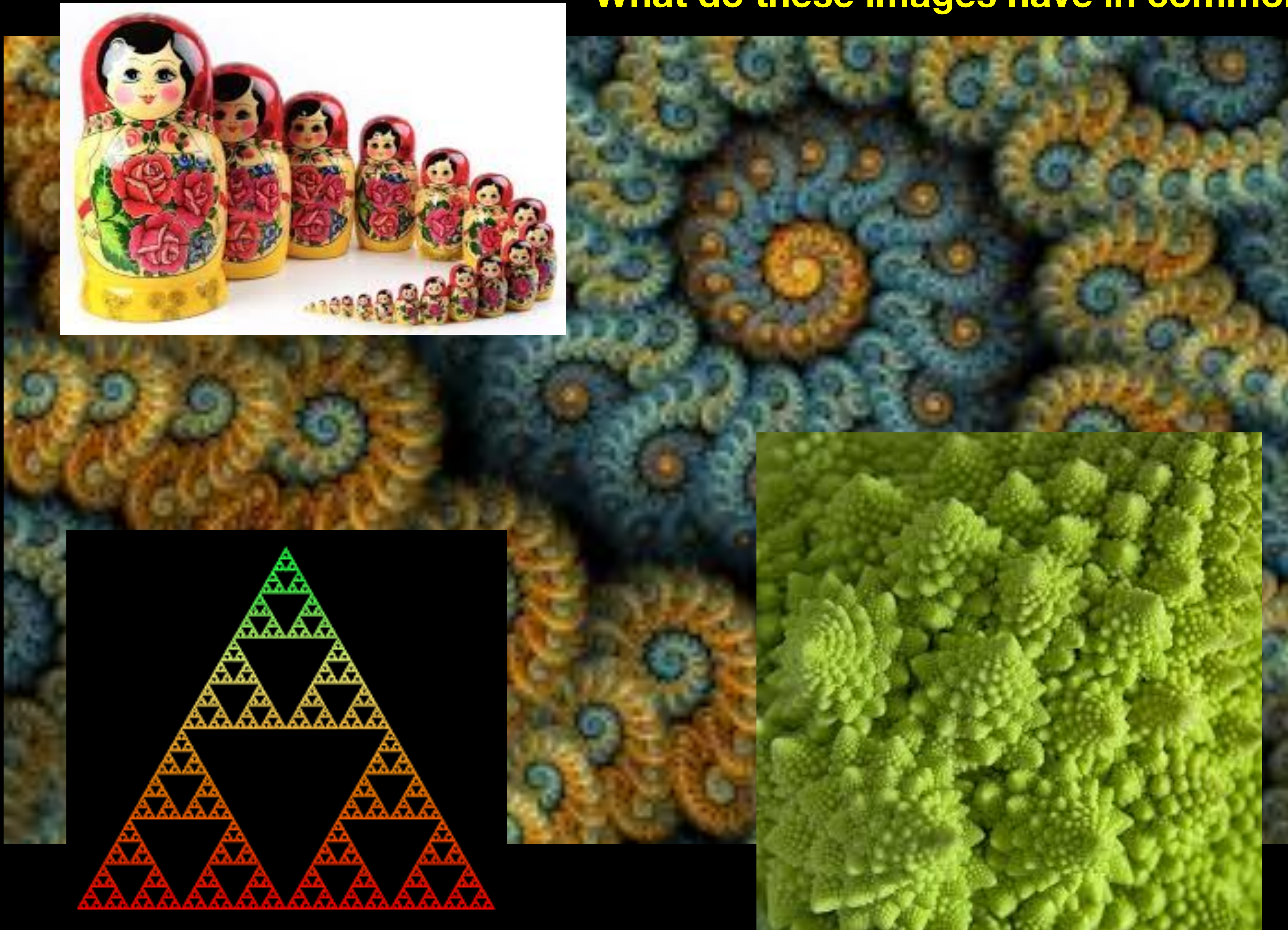
Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

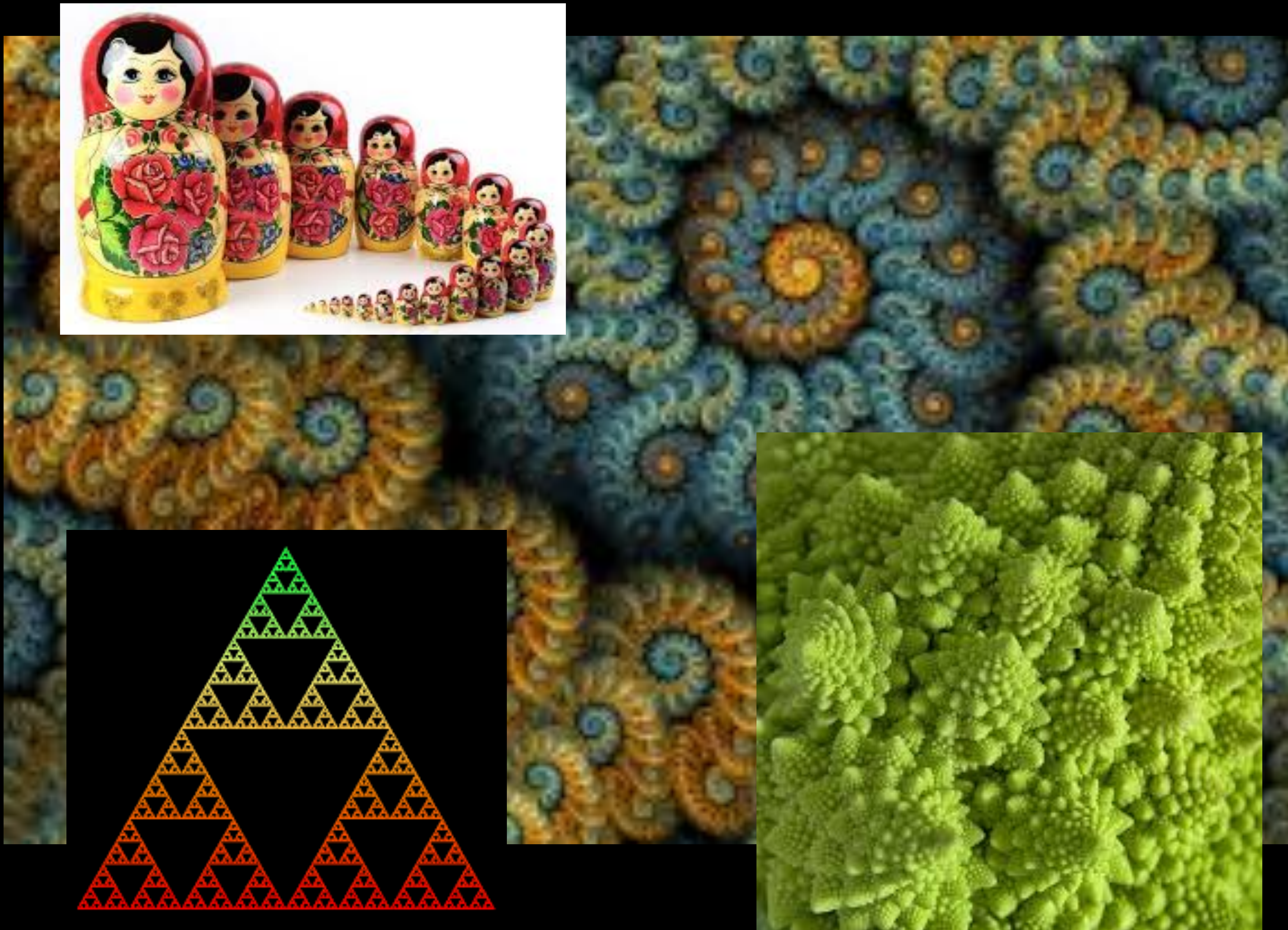
Announcements

Recursion

What do these images have in common



They contain a SMALLER copy of THEMSELVES



Print String Backwards

"Hello"

Print String Backwards

"Hello"

Procedure:

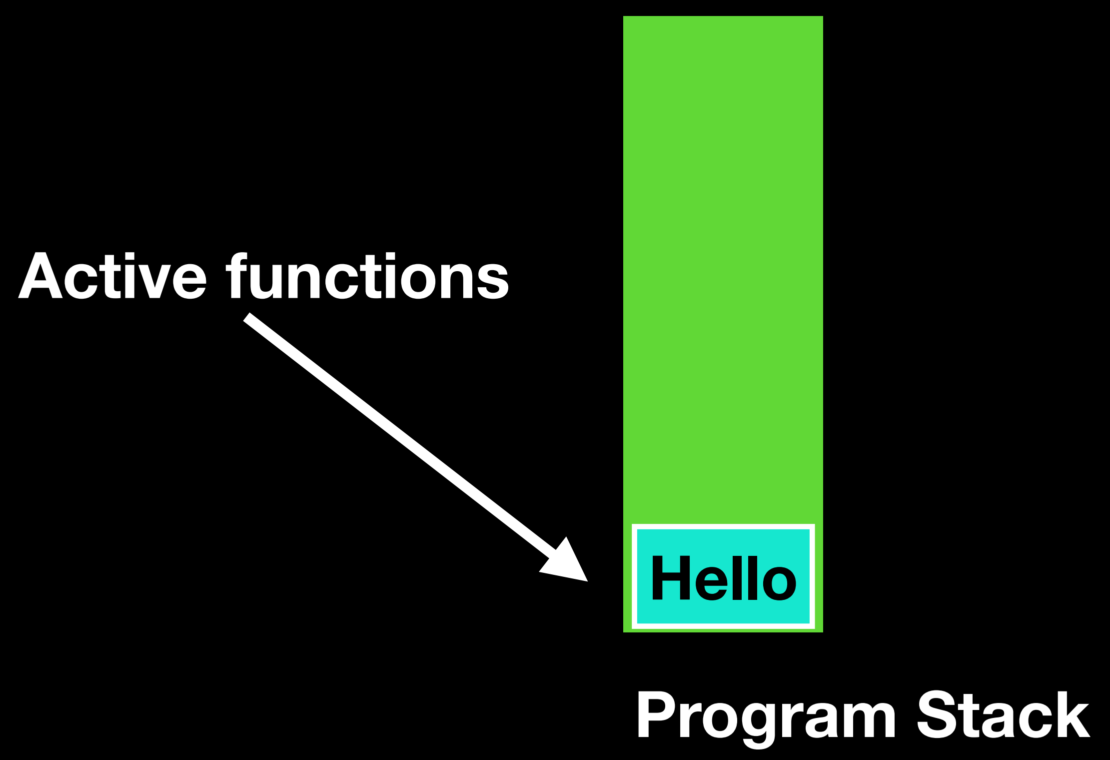
If there are characters to print
Print the last character and reverse the rest

Recursive Call
Notice it's the last thing it does

Print String Backwards

Hello

o

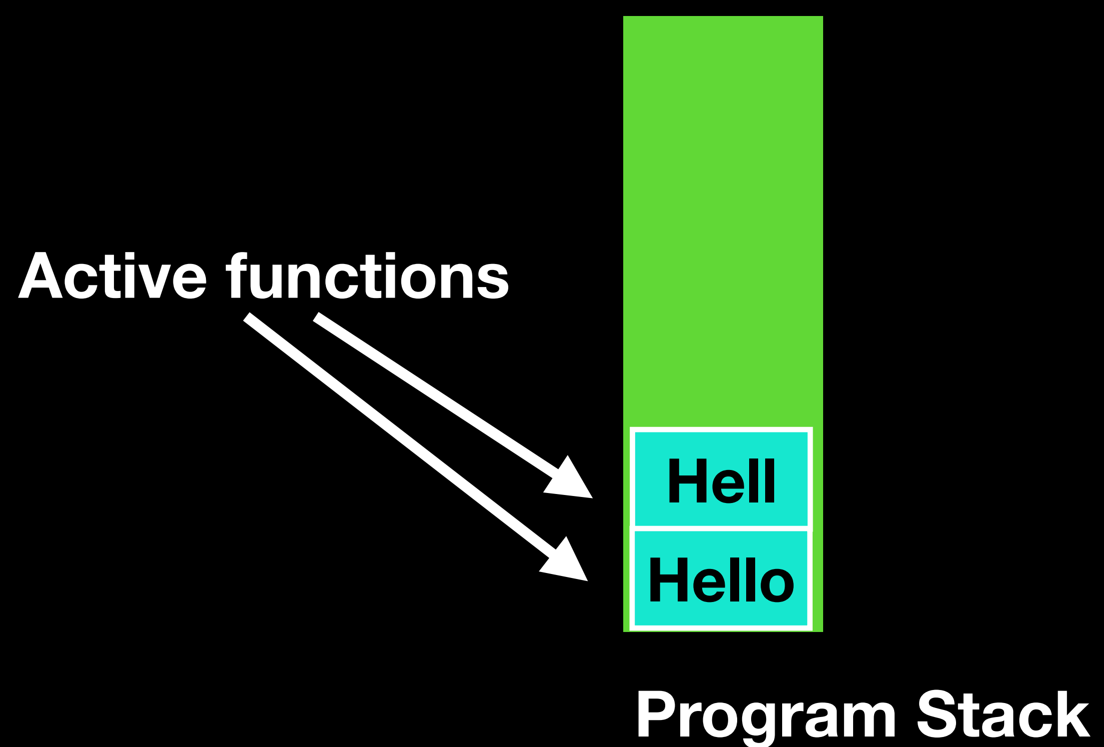


Print String Backwards

Hello

o

Hell



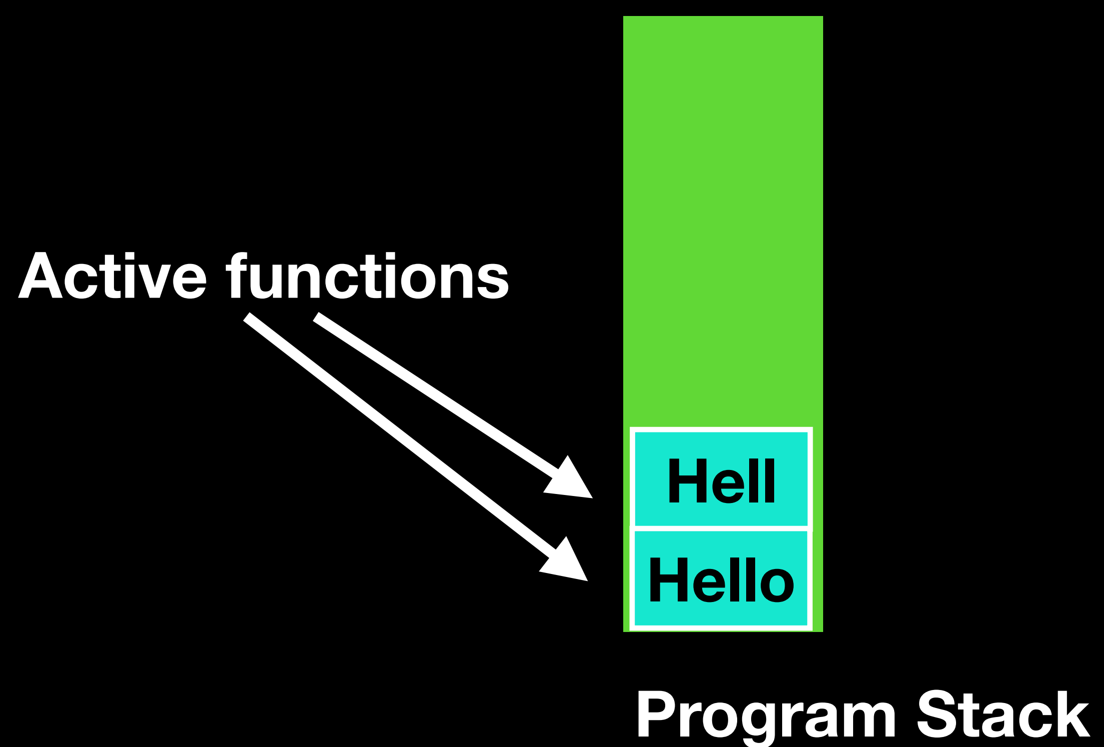
Print String Backwards

Hello

o

Hell

o l



Print String Backwards

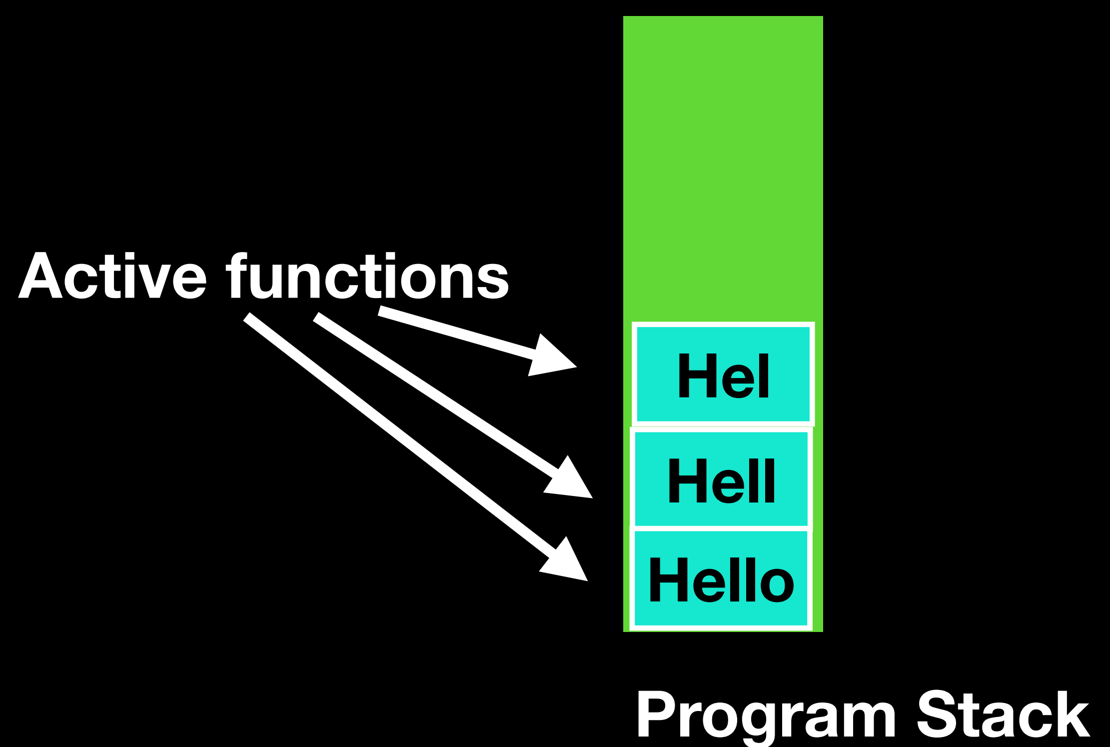
Hello

o

→ Hell

o l

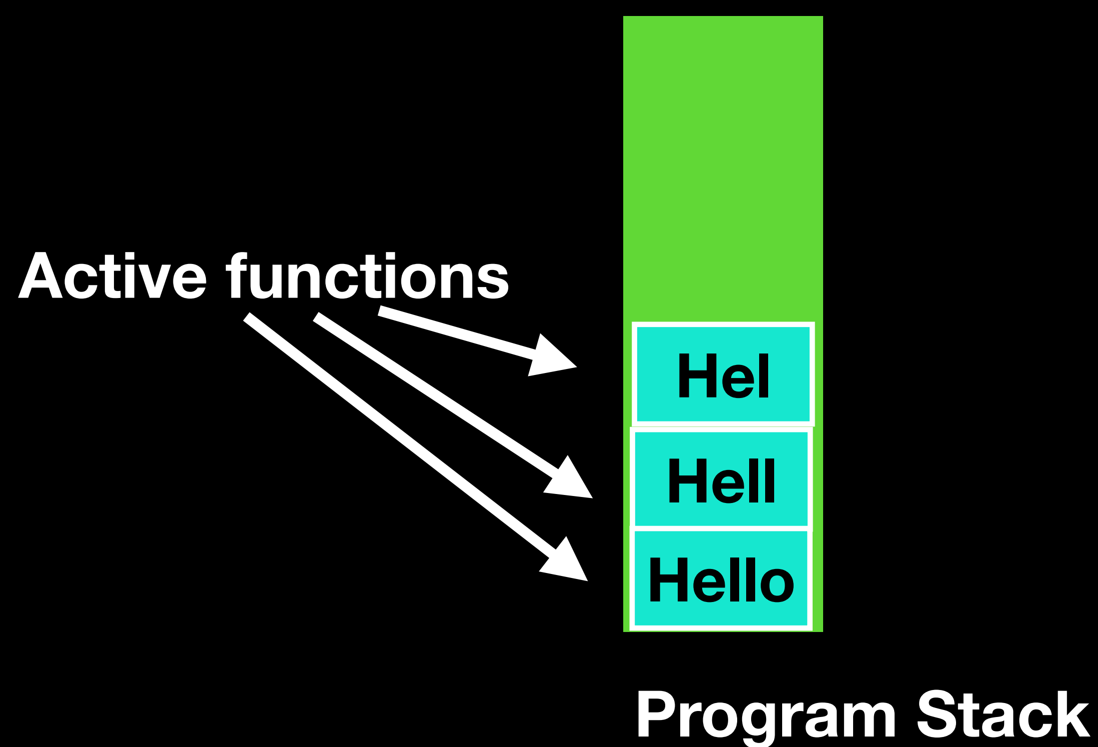
→ Hel



Print String Backwards

Hello
o
→ Hell
o l

→ Hel
o l l



Print String Backwards

Hello

o

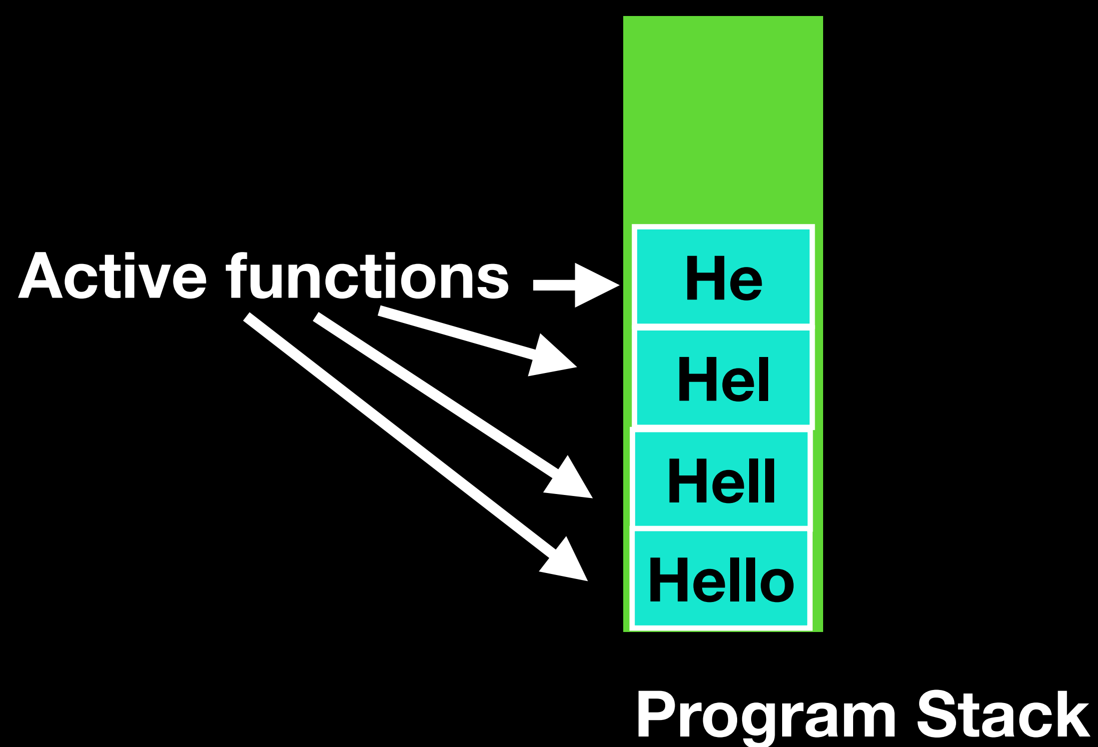
→ Hell

o l

→ Hel

o l l

→ He



Print String Backwards

Hello

o

→ Hell

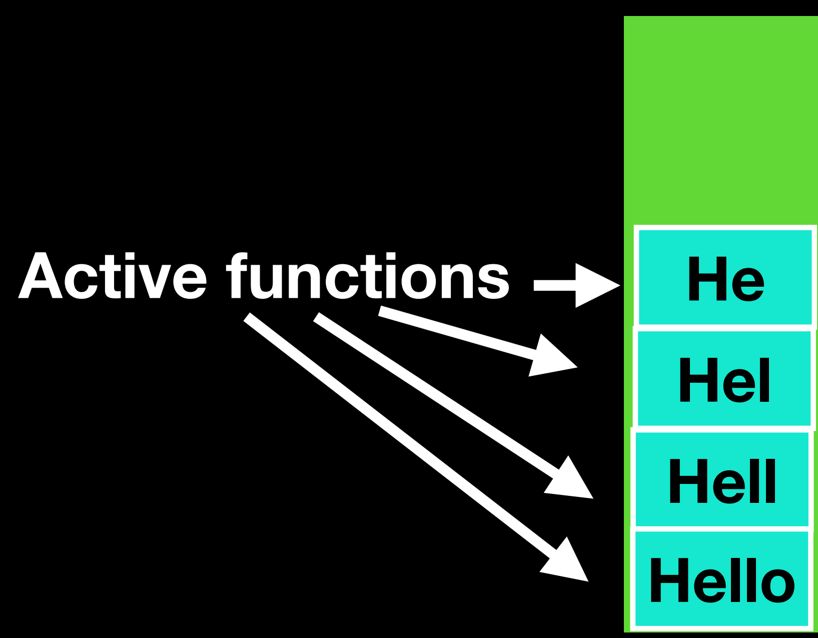
o l

→ Hel

o l l

→ He

o l l e



Program Stack

Print String Backwards

Hello

o

→ Hell

o l

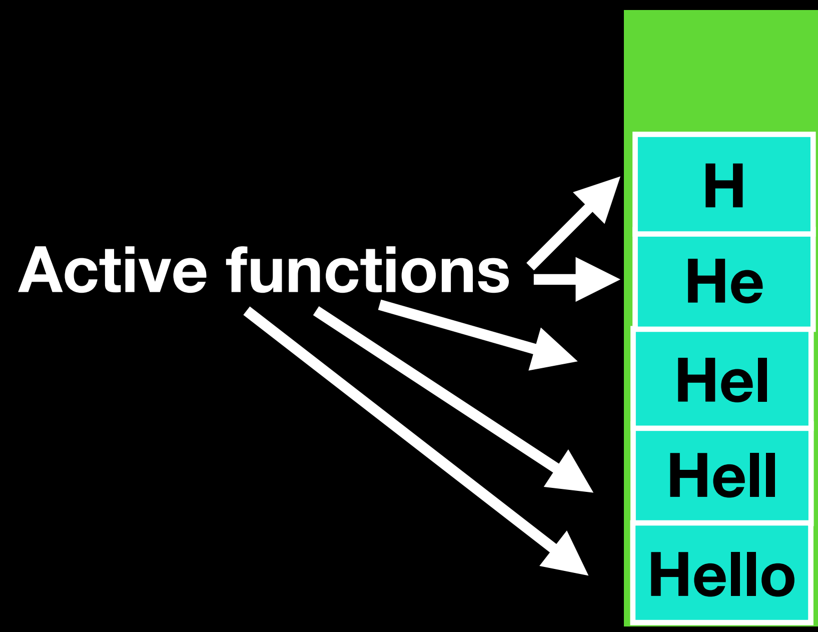
→ Hel

o l l

→ He

o l l e

→ H



Program Stack

Print String Backwards

Hello

o

Hell

o l

Hel

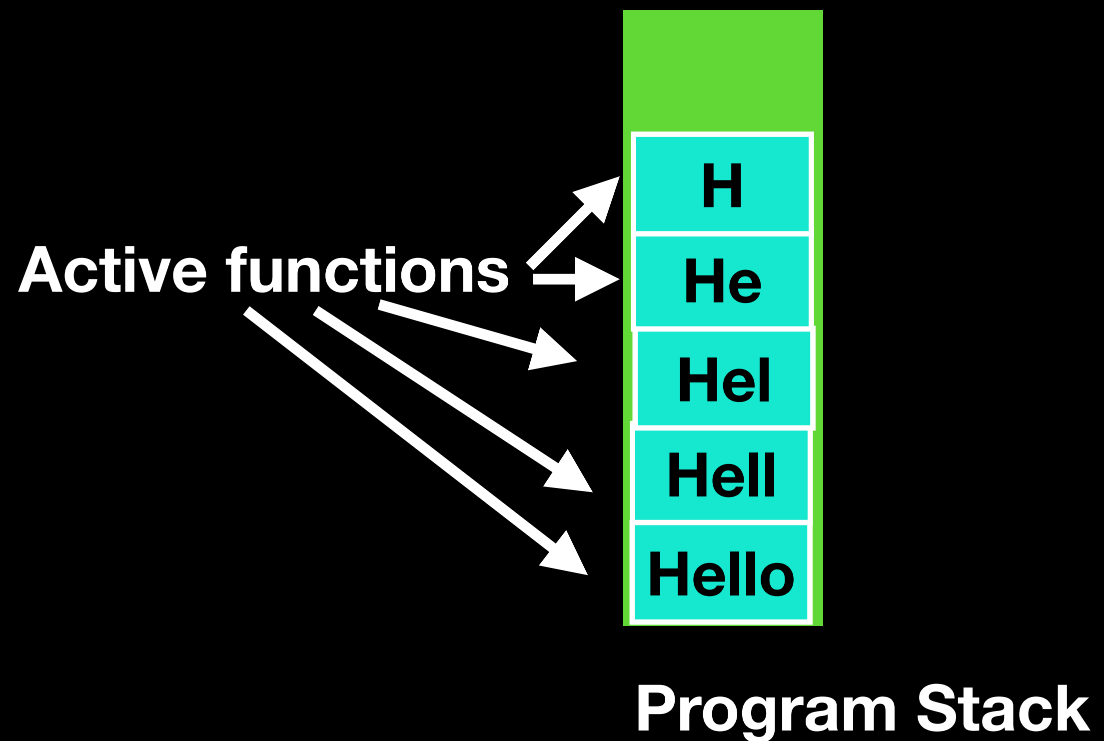
o l l

He

o l l e

H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

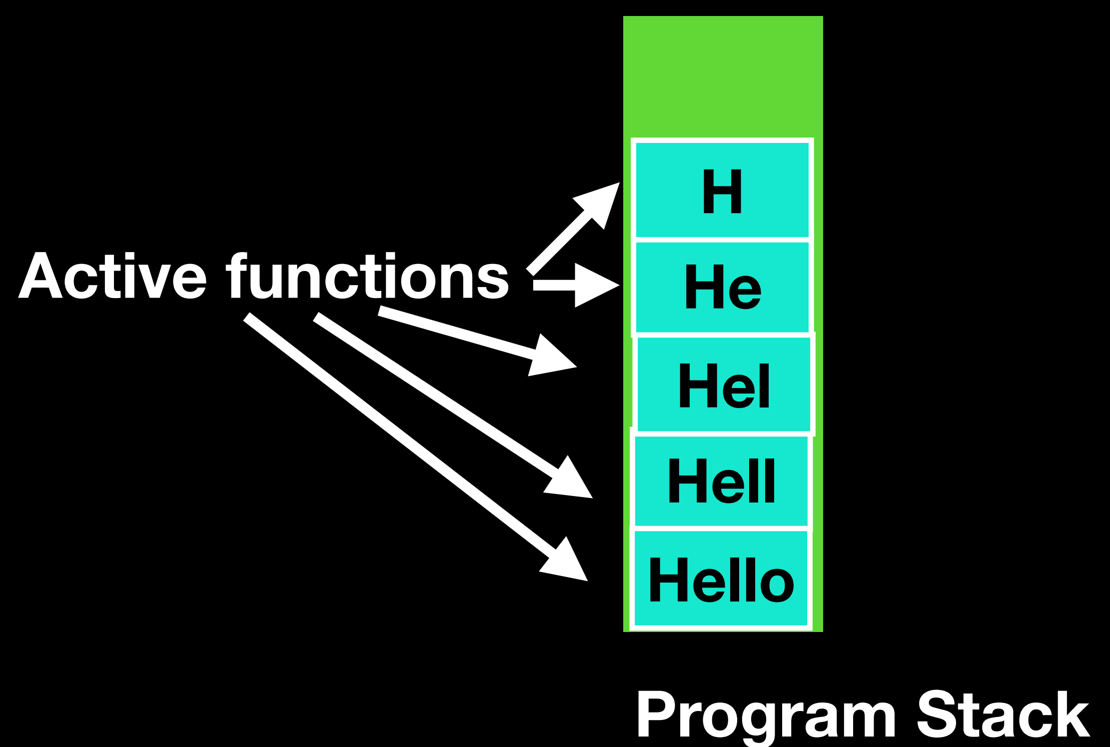
o l l

→ He

o l l e

→ H

o l l e H



BASE CASE

Print String Backwards

Hello

o

→ Hell

o l

→ Hel

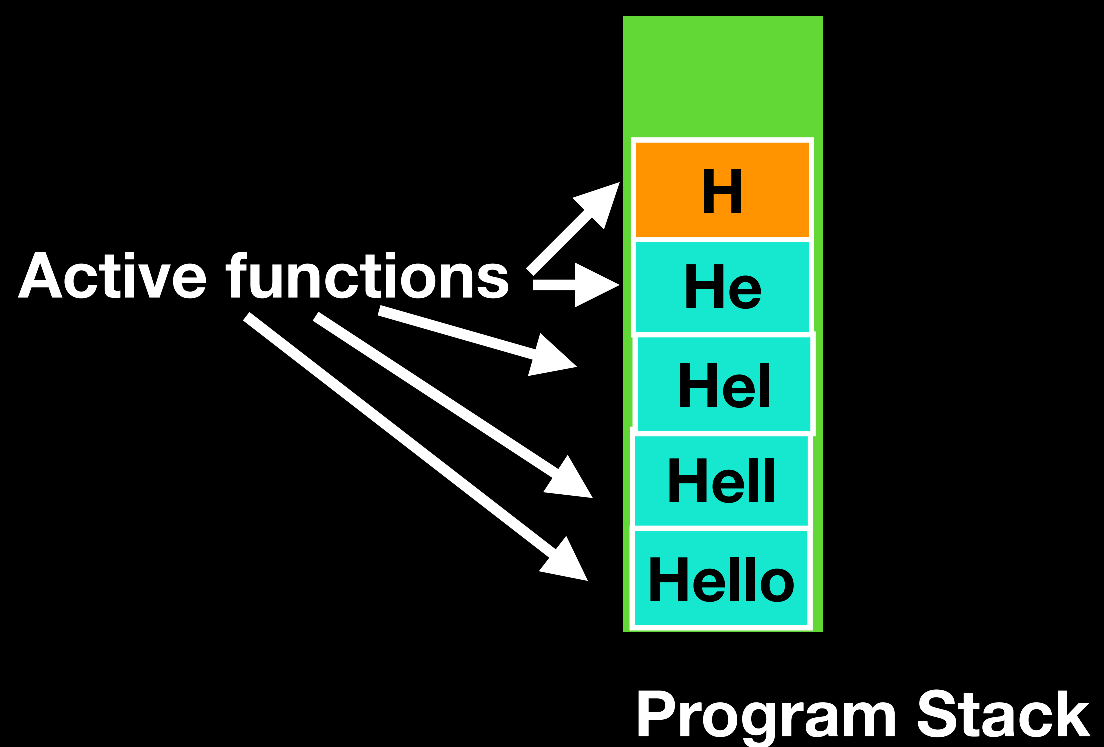
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

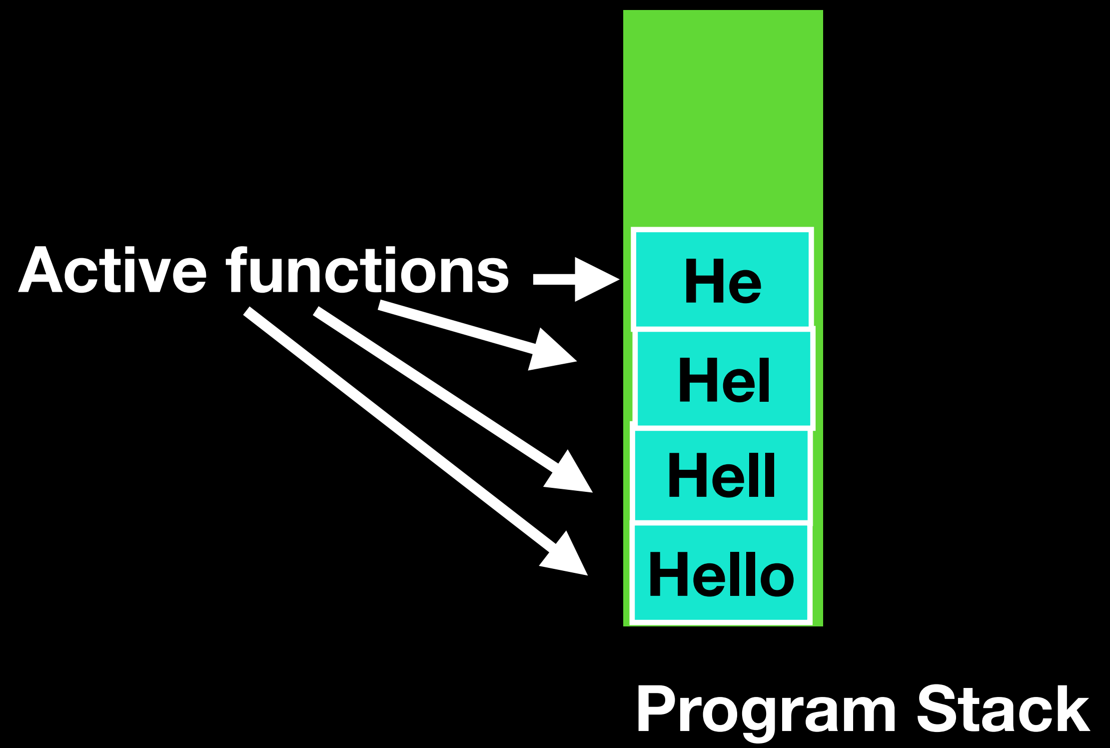
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

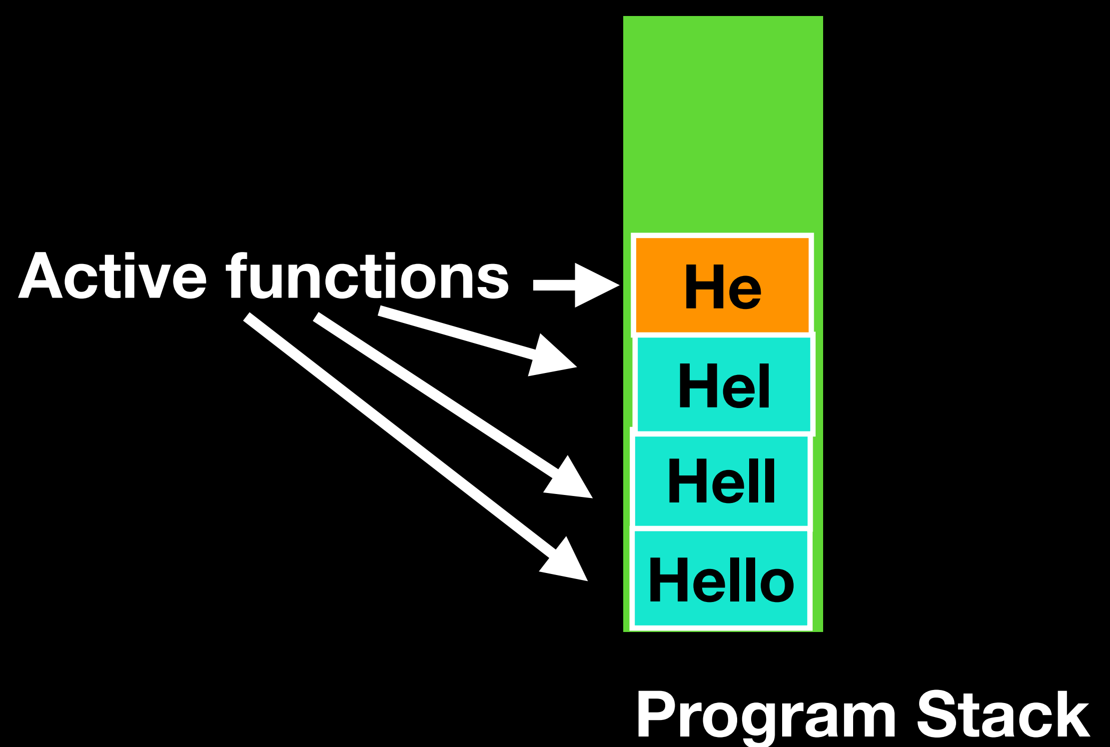
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

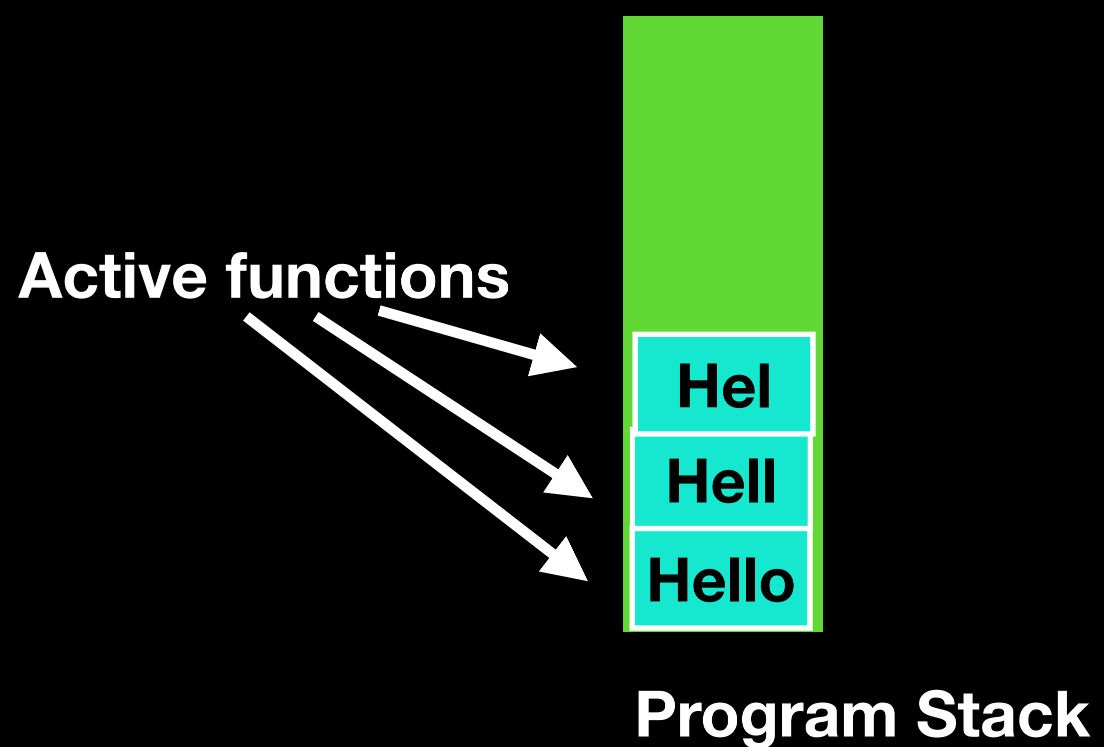
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

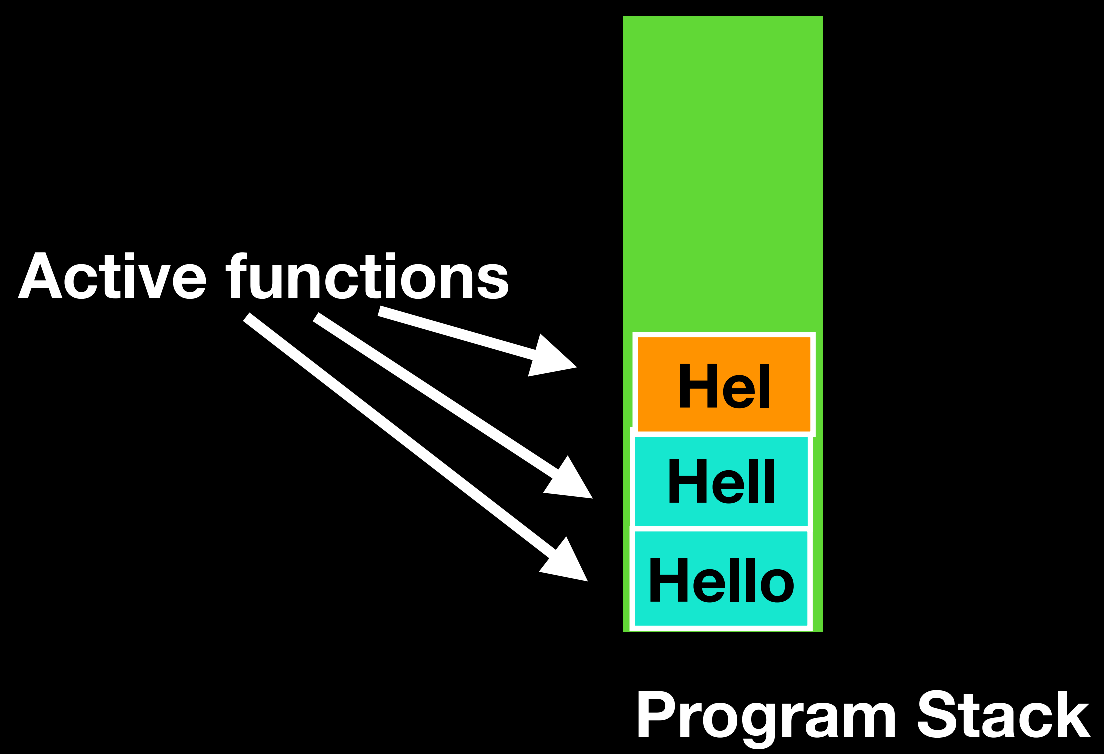
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

Hell

o l

Hel

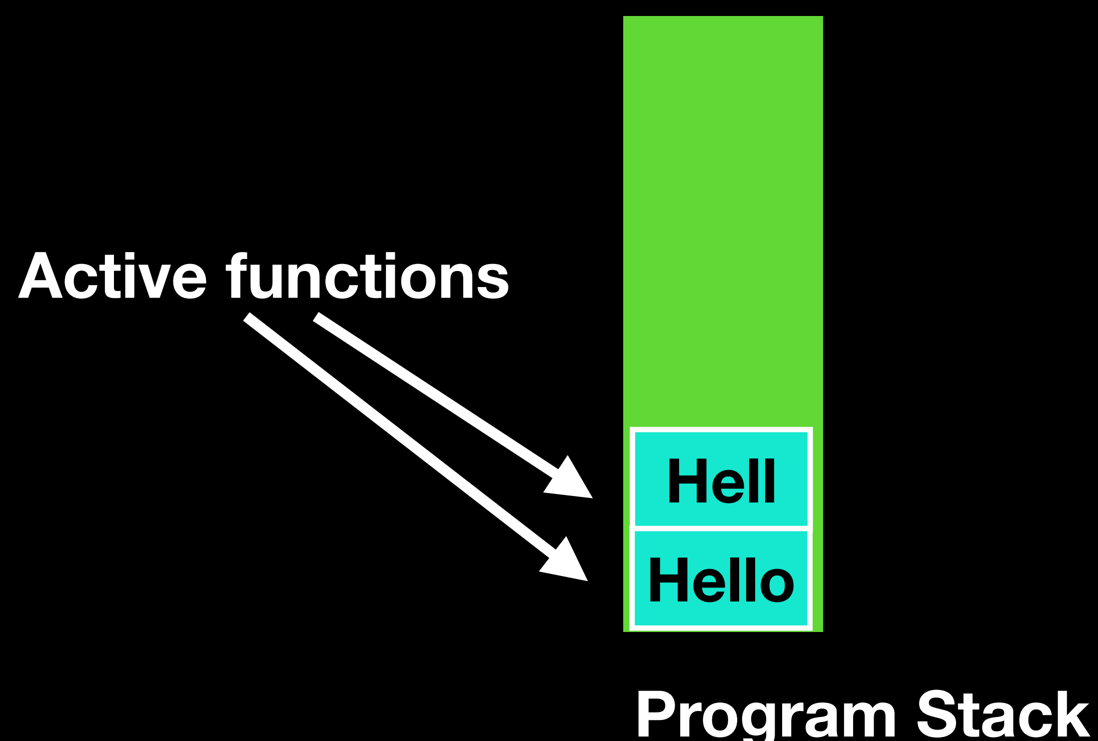
o l l

He

o l l e

H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

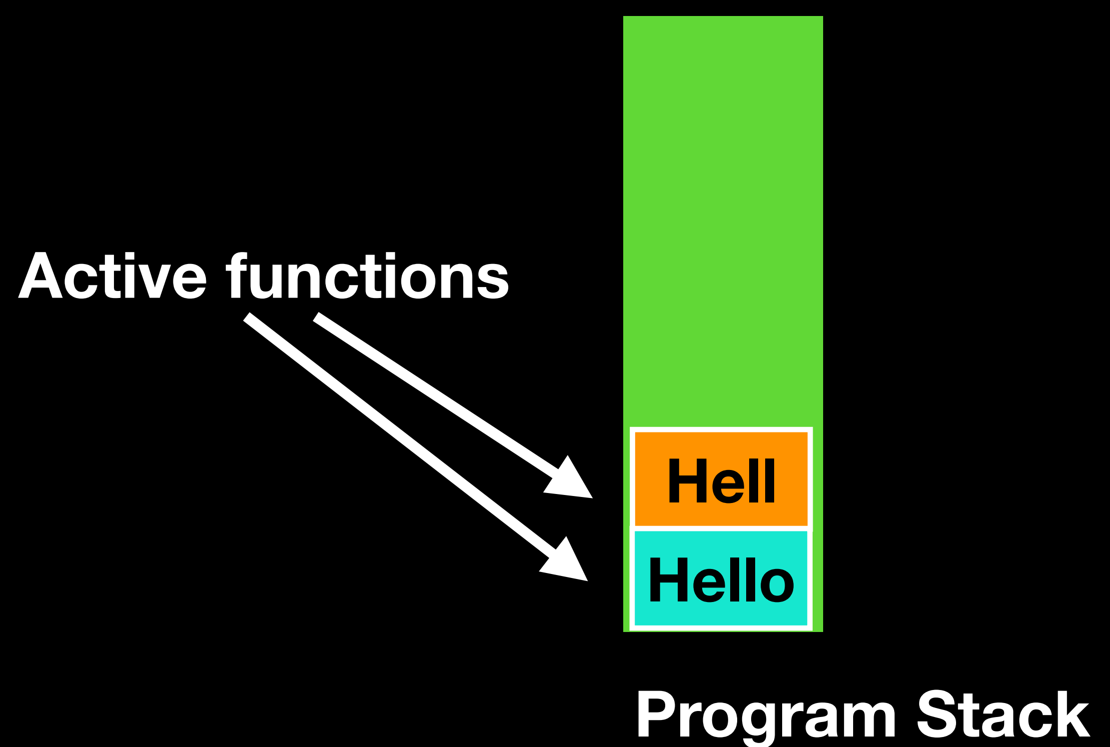
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

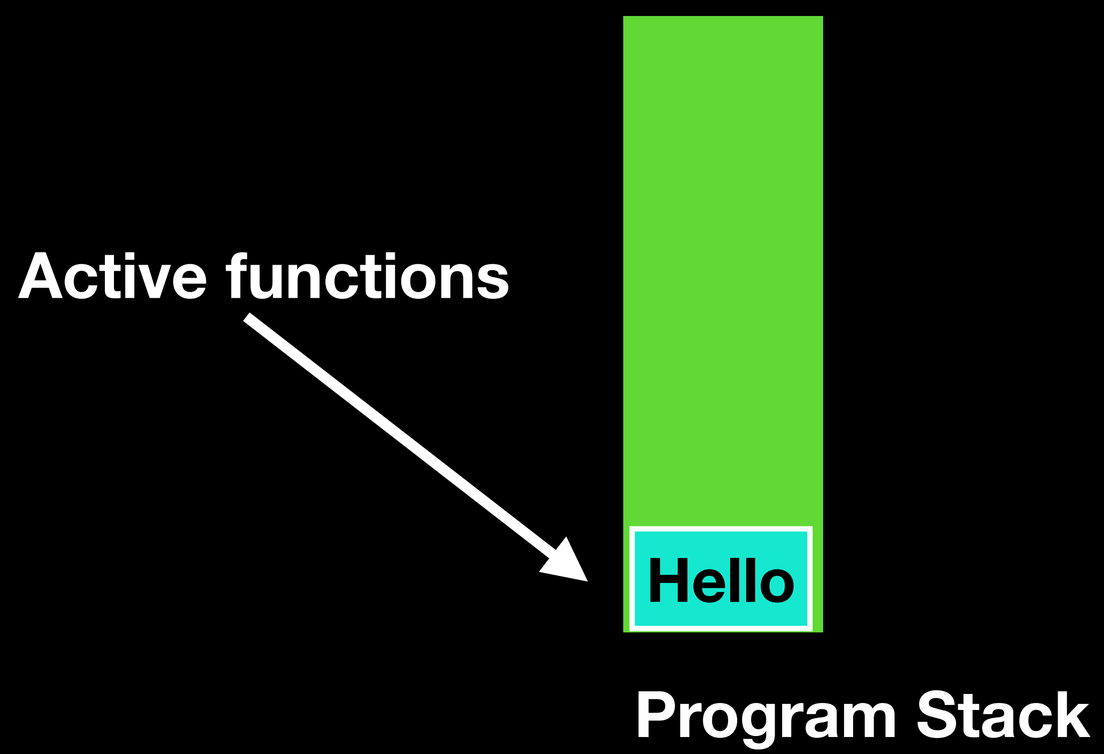
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

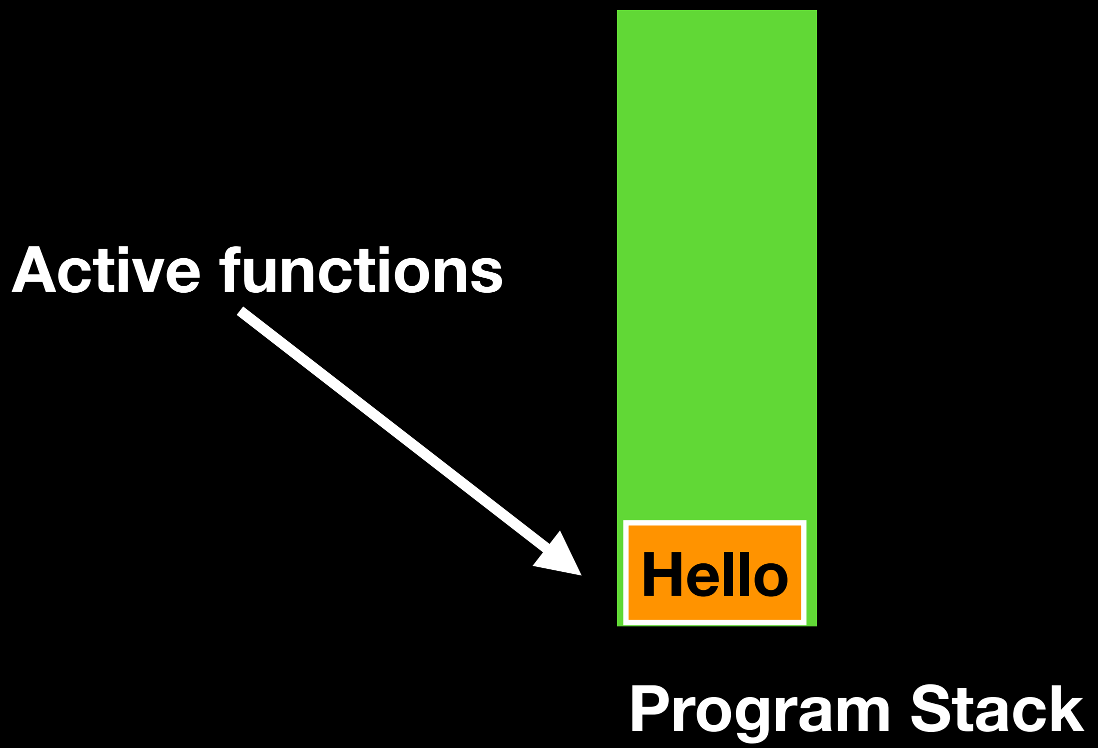
o l l

→ He

o l l e

→ H

o l l e H



Print String Backwards

Hello

o

→ Hell

o l

→ Hel

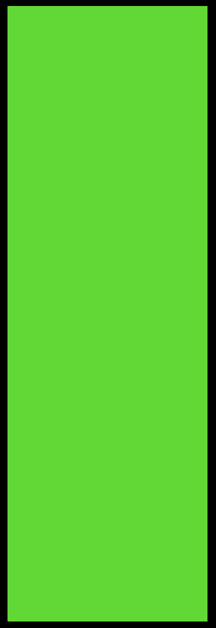
o l l

→ He

o l l e

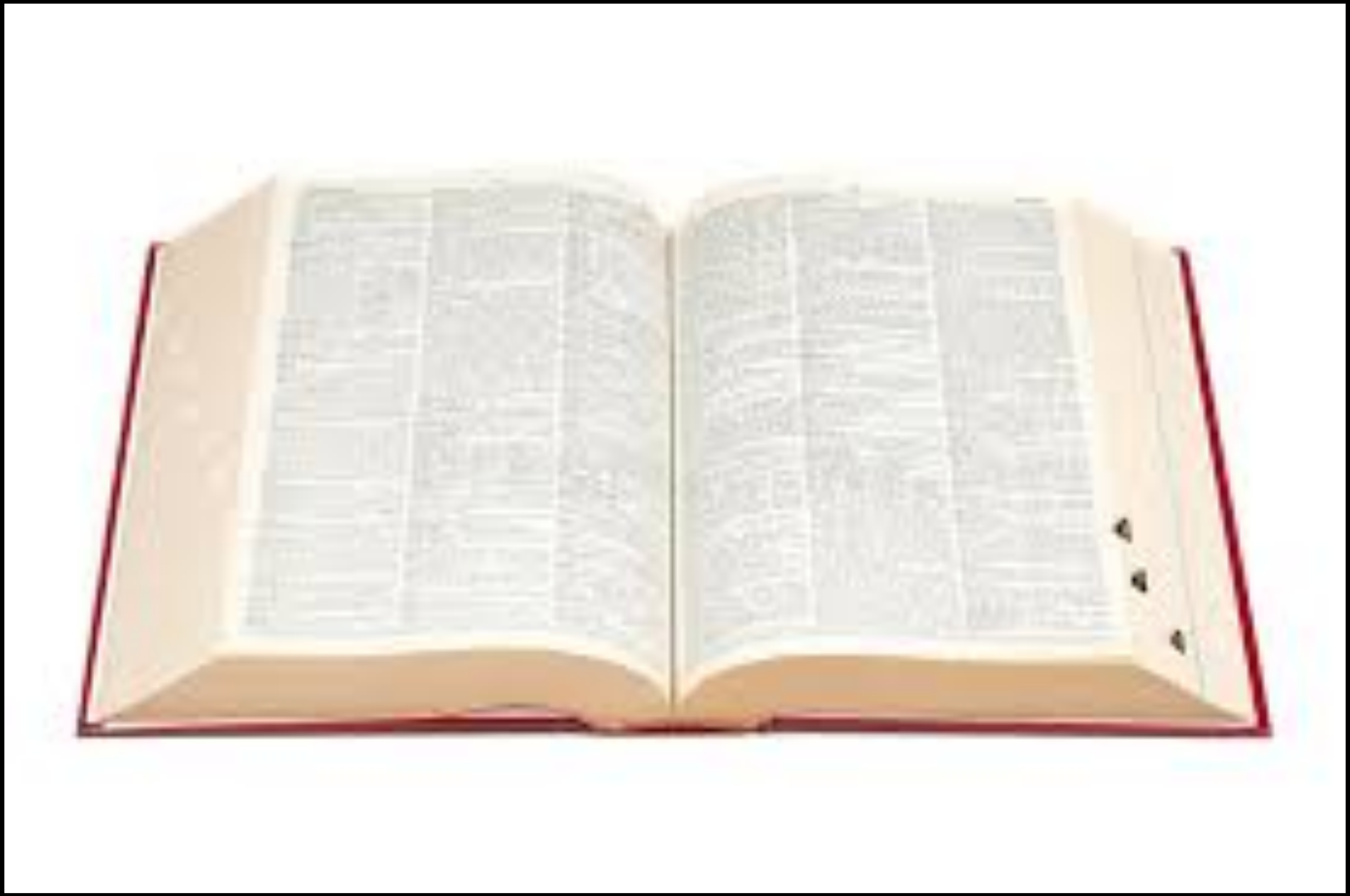
→ H

o l l e H



Program Stack

If I hand you a **printed** dictionary (an actual book) and ask you to find the word "Kalimba", what do you do?



Look in ?

LOOK FOR WORD "Kalimba" IN DICTIONARY

- Open dictionary at random page

_ If "Kalimba" is on page FOUND!!!

- Else if "Kalimba" is lexicographically < first word on page

LOOK FOR WORD "Kalimba" IN ***LOWER*** *HALF* ← **Recursive Call**

- Else if "Kalimba" is lexicographically > last word on page

LOOK FOR WORD "Kalimba" IN ***UPPER*** *HALF* ← **Recursive Call**

How is this different from recursive solution to print backwards?

How is this different from recursive solution to print backwards?

- Two recursive calls

- Execute either one or the other

- Cuts problem in 1/2

Different Flavors of Recursion

Reverse String: write first character, reverse the remaining single smaller string

Dictionary: either inspect upper-half or lower-half

Solve a problem by breaking it up into one or more smaller "*similar*" problems

Recursive Problem-Solving

```
if(problem is sufficiently simple){

     directly solve the problem
     i.e. do something and/or return the solution

} else{

     split problem up into one or more smaller
     problems with the same structure as the original

     solve some or all of those smaller problems

     do something or combine results to return
          solution if necessary
}
```

Recursive Problem-Solving

```
if(problem is sufficiently simple){

    directly solve the problem
    i.e. do something and/or return the solution

} else{

    split problem up into one or more smaller
    problems with the same structure as the original

    solve some or all of those smaller problems

    do something or combine results to return
         solution if necessary
}
```

BASE CASE

Why Recursion

An alternative to iteration

Not always practical (some compilers optimize tail-recursive algorithms)

Elegant and intuitive solution for some problems

Factorial

1 x 2 x 3 x … x n

$$n! = \prod_{k=1}^{n} k$$

For example:

0!=1, **1!**=1, **2!**=2, **3!**=6, **4!**=24, **5!**=120

The empty product

But what if we start from `n`?

n!=

But what if we start from `n`?

n! = n x (n-1) x (n-2) x (n - 3) x … …. …. … 2 x 1

What is this?

But what if we start from `n`?

$$\mathbf{n!} = n \times \underbrace{(n-1) \times (n-2) \times (n-3) \times \ldots \quad \ldots. \quad \ldots. \quad \ldots 2 \times 1}_{\mathbf{(n-1)!}}$$

But what if we start from `n`?

n! = n x (n-1) x (n-2) x (n - 3) x … …. …. … 2 x 1

(n-1)!

(n-1)! = (n-1) x (n-2) x (n - 3) x … …. …. … 2 x 1

What is this?

But what if we start from `n`?

$$n! = n \times \underbrace{(n-1) \times (n-2) \times (n-3) \times \ldots \quad \ldots. \quad \ldots. \quad \ldots 2 \times 1}_{(n-1)!}$$

$$(n-1)! = (n-1) \times \underbrace{(n-2) \times (n-3) \times \ldots \quad \ldots. \quad \ldots. \quad \ldots 2 \times 1}_{(n-2)!}$$

Recursion that Returns a Value

$$n! = n \times (n-1)!$$

Same function being called within solution

Recursion that Returns a Value

n! = n x (n-1)!

```
/** Computes the factorial of the nonnegative integer n.
 @pre:  n must be greater than or equal to 0.
 @post:  None.
 @return:  The factorial of n; n is unchanged. */
int factorial(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * factorial(n - 1); // n * (n-1)! is n!
}  // end fact
```

Recursion that Returns a Value

n! = n x (n-1)!

```
/** Computes the factorial of the nonnegative integer n.
 @pre:  n must be greater than or equal to 0.
 @post:  None.
 @return:  The factorial of n; n is unchanged. */
int factorial(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * factorial(n - 1); // n * (n-1)! is n!
}  // end fact
```

BASE CASE

Recursion that Returns a Value

n! = n x (n-1)!

```
/** Computes the factorial of the nonnegative integer n.
 @pre:  n must be greater than or equal to 0.
 @post:  None.
 @return:  The factorial of n; n is unchanged. */
int factorial(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * factorial(n - 1); // n * (n-1)! is n!
}  // end fact
```

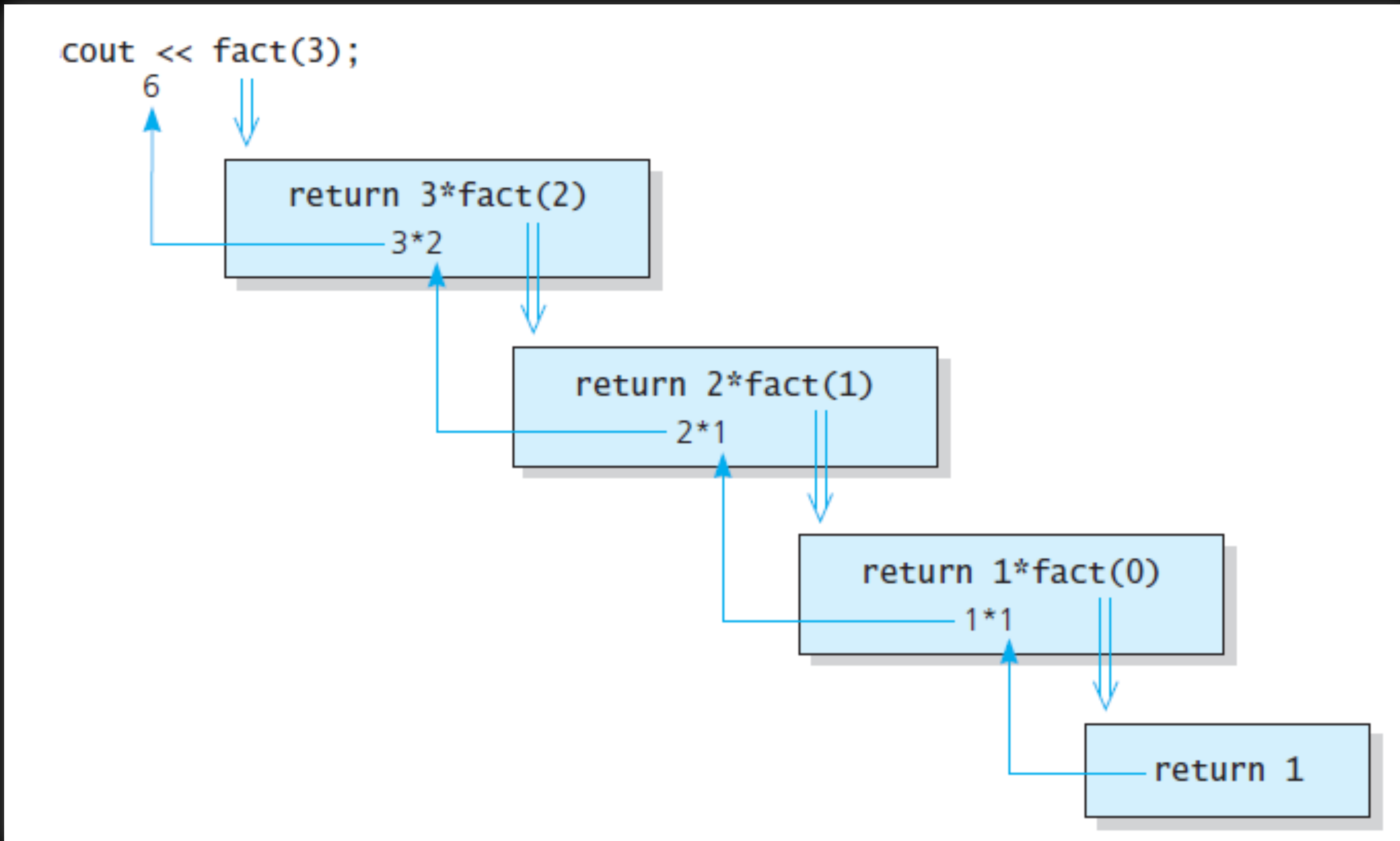
Recursion that Returns a Value

n! = n x (n-1)!

```
/** Computes the factorial of the nonnegative integer n.
 @pre:  n must be greater than or equal to 0.
 @post:  None.
 @return:  The factorial of n; n is unchanged. */
int factorial(int n)
{
   if (n == 0)
      return 1;
   else // n > 0, so n-1 >= 0. Thus, fact(n-1) returns (n-1)!
      return n * factorial(n - 1); // n * (n-1)! is n!
}  // end fact
```

BASE CASE

WILL LEAD TO BASE CASE


cout << fact(3);
6
return 3*fact(2)
3*2
return 2*fact(1)
2*1
return 1*fact(0)
1*1
return 1

Types of Recursion

Reverse String:

- single recursive call
- Base case: stop => no return value

Dictionary:

- split problem into halves but solve only 1
- Base case: stop => no return value

Factorial:

- single recursive call
- Base case: return a value for computation in each recursive call

Why/When use recursion

Usually less efficient than iterative counterparts (we will see example later in the course)

Inherent overhead associated with function calls

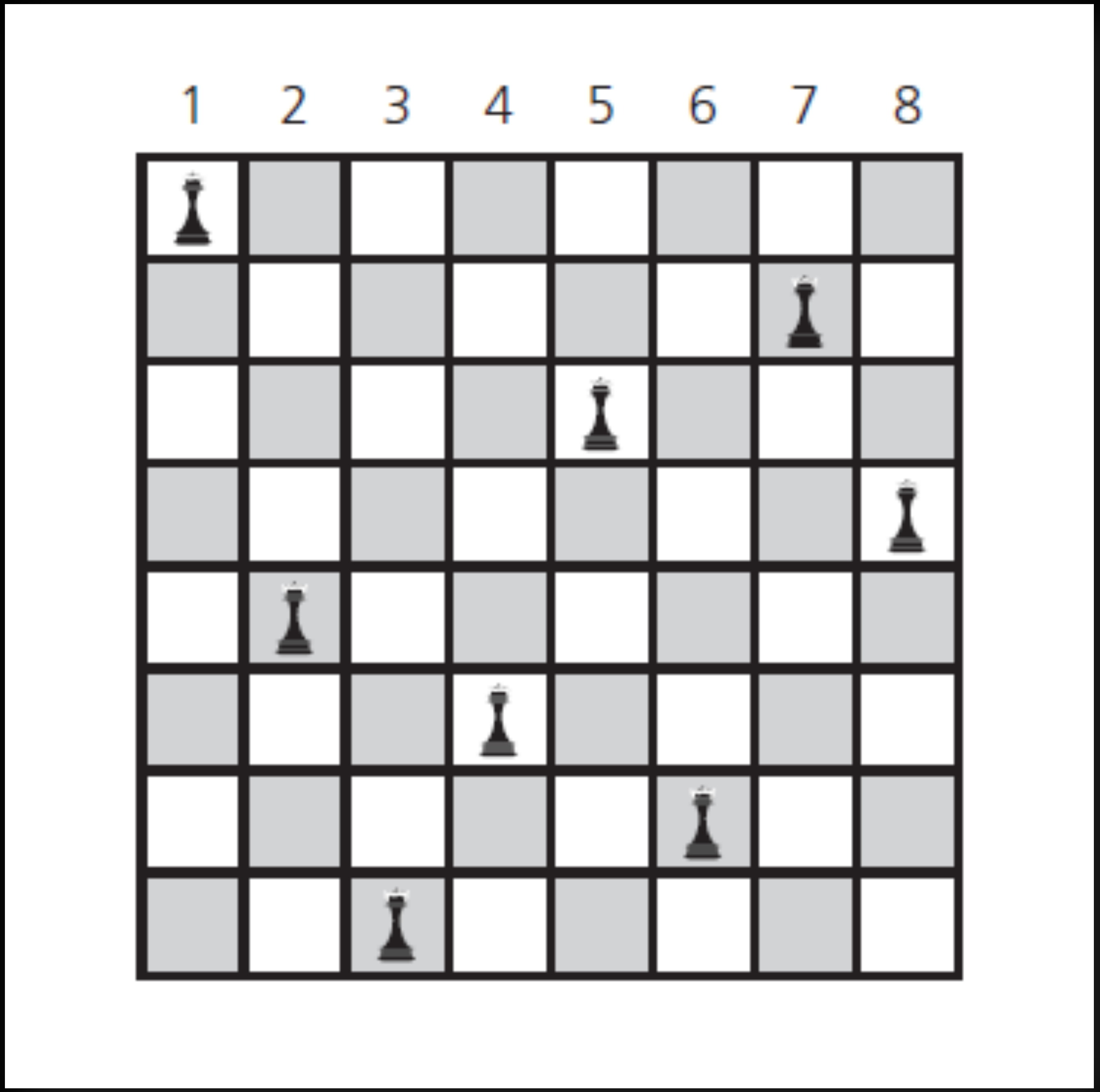
Repeated recursive calls with same parameters

Compilers can optimize tail-recursive (recursive call is the last statement in the function) functions to be iterative

Sometimes logic of iterative solution can be very complex in comparison to recursive solution

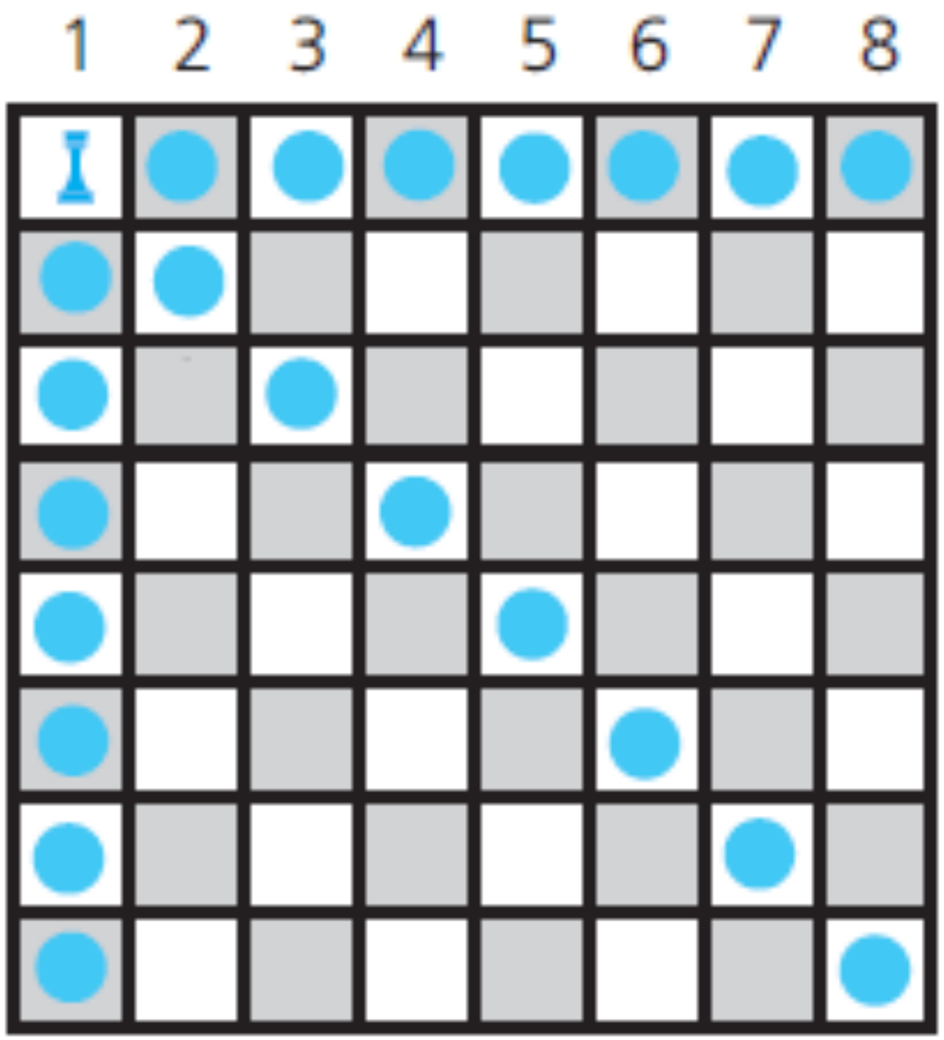
Recursive Backtracking

The Eight Queens Problem



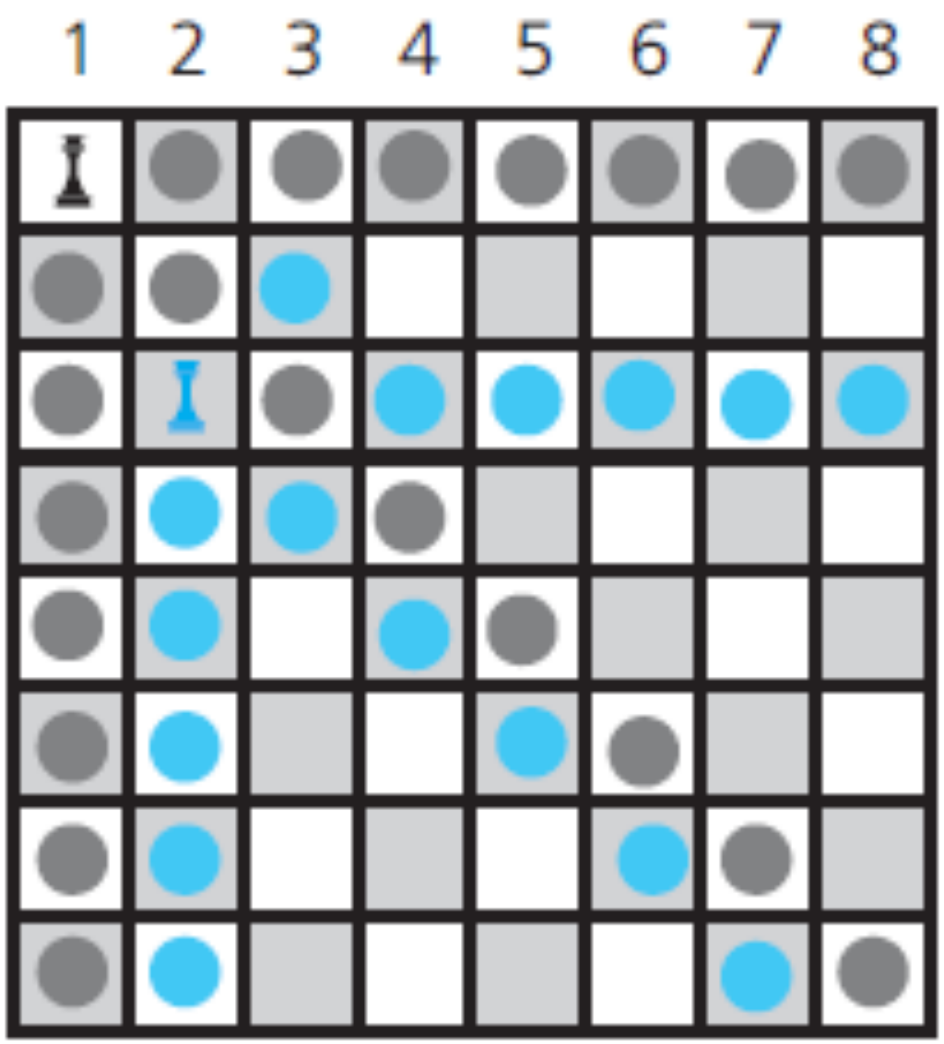
Place 8 Queens on the board s.t. no queen is on the same row, column or diagonal

The Eight Queens Problem



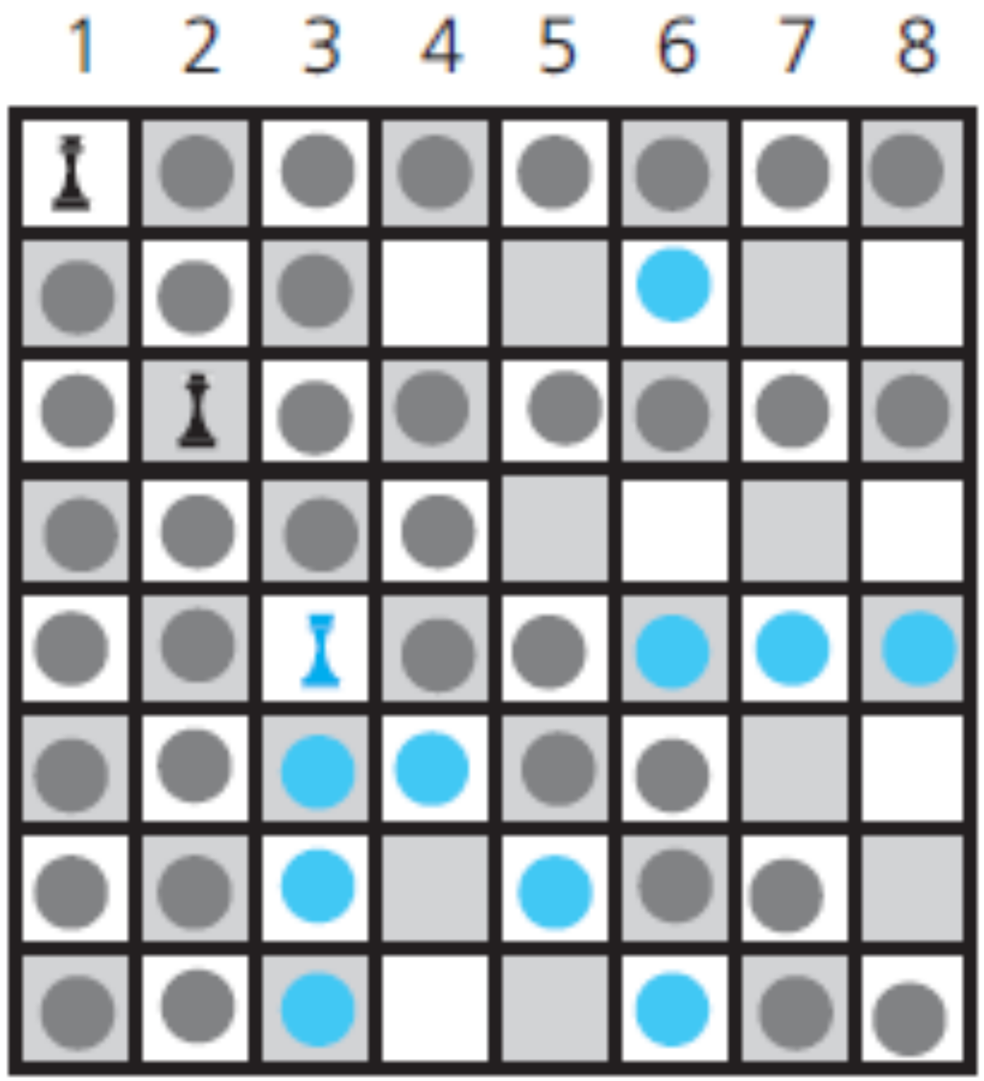
(a) The first queen in column 1

The Eight Queens Problem



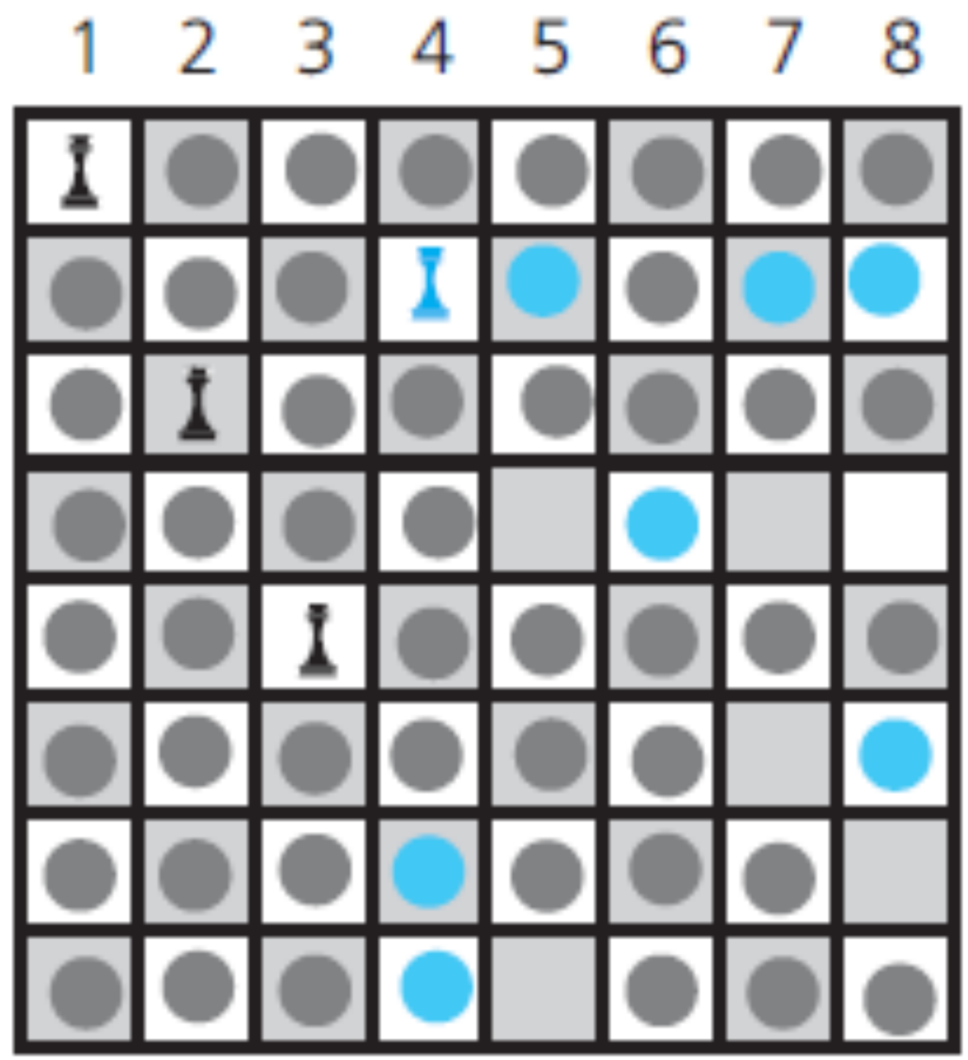
(b) The second queen in column 2

The Eight Queens Problem



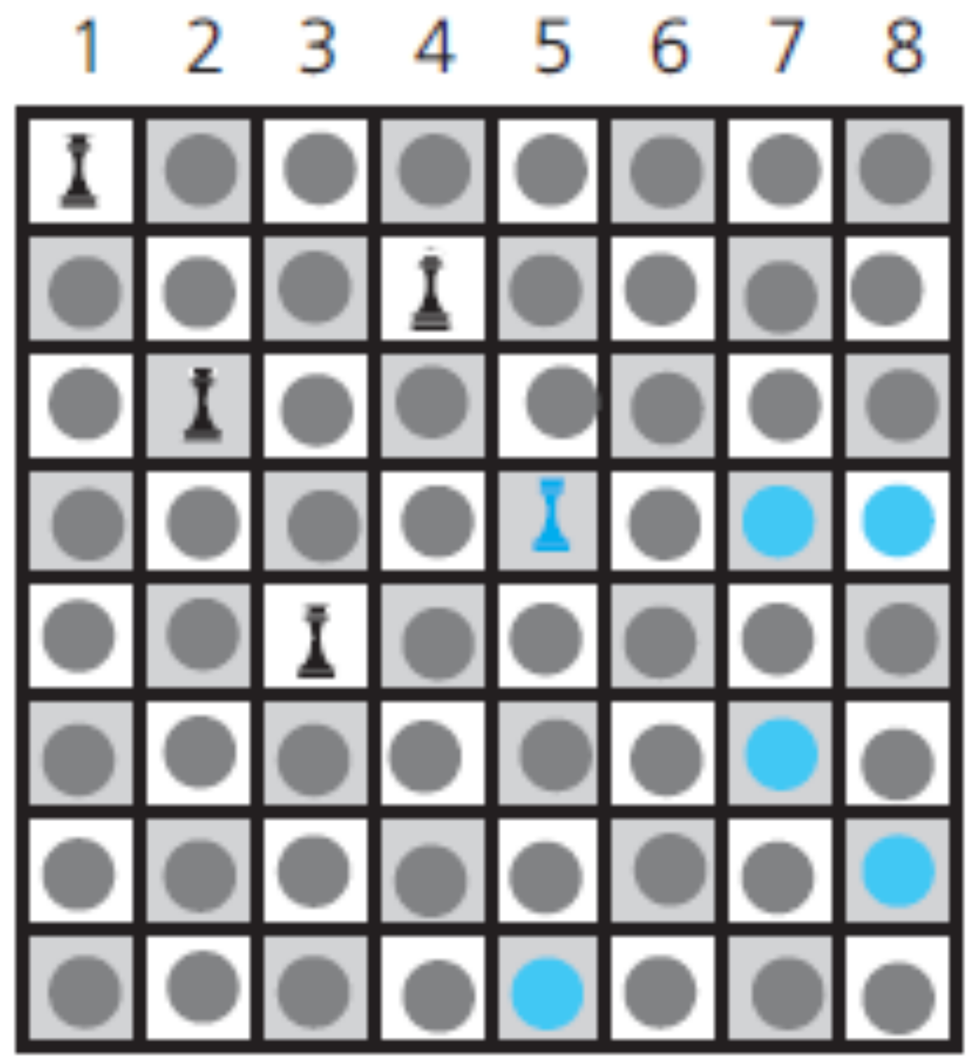
(c) The third queen in column 3

The Eight Queens Problem



(d) The fourth queen in column 4

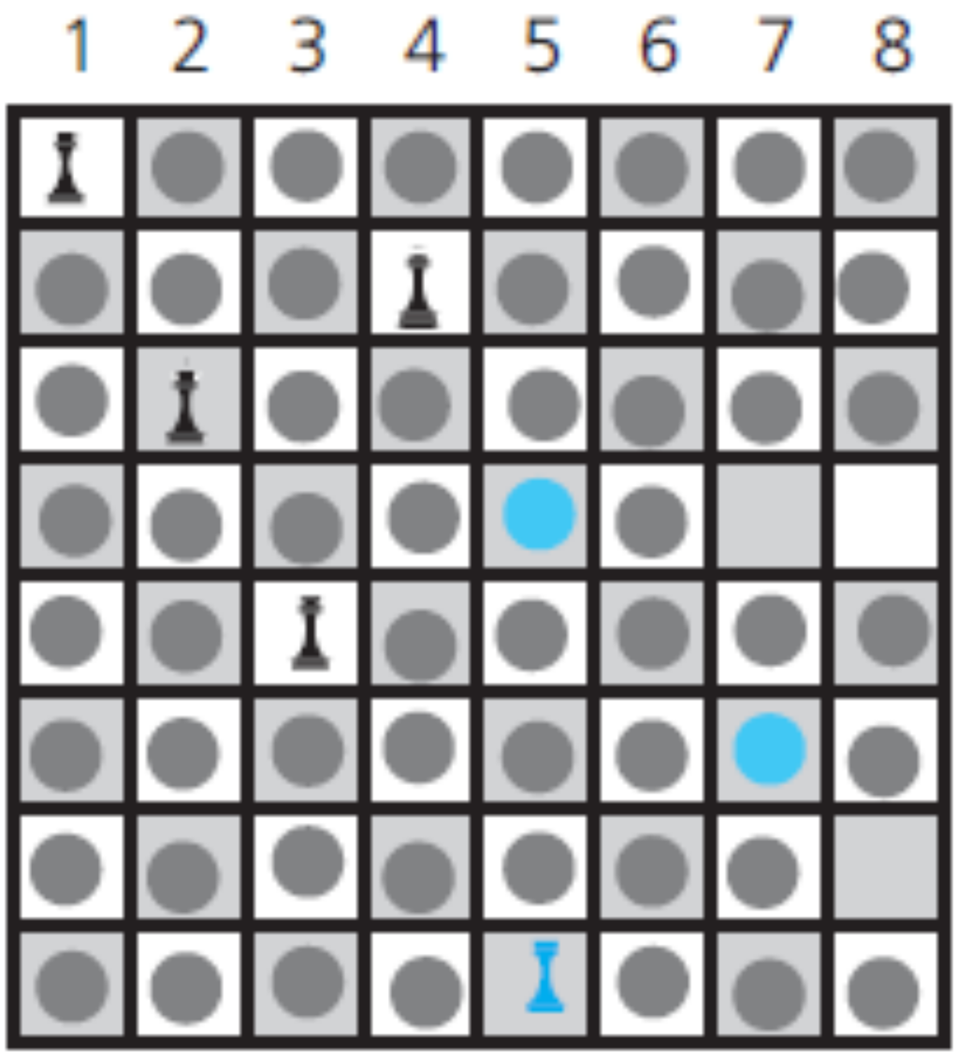
The Eight Queens Problem



(e) The five queens can attack all of column 6

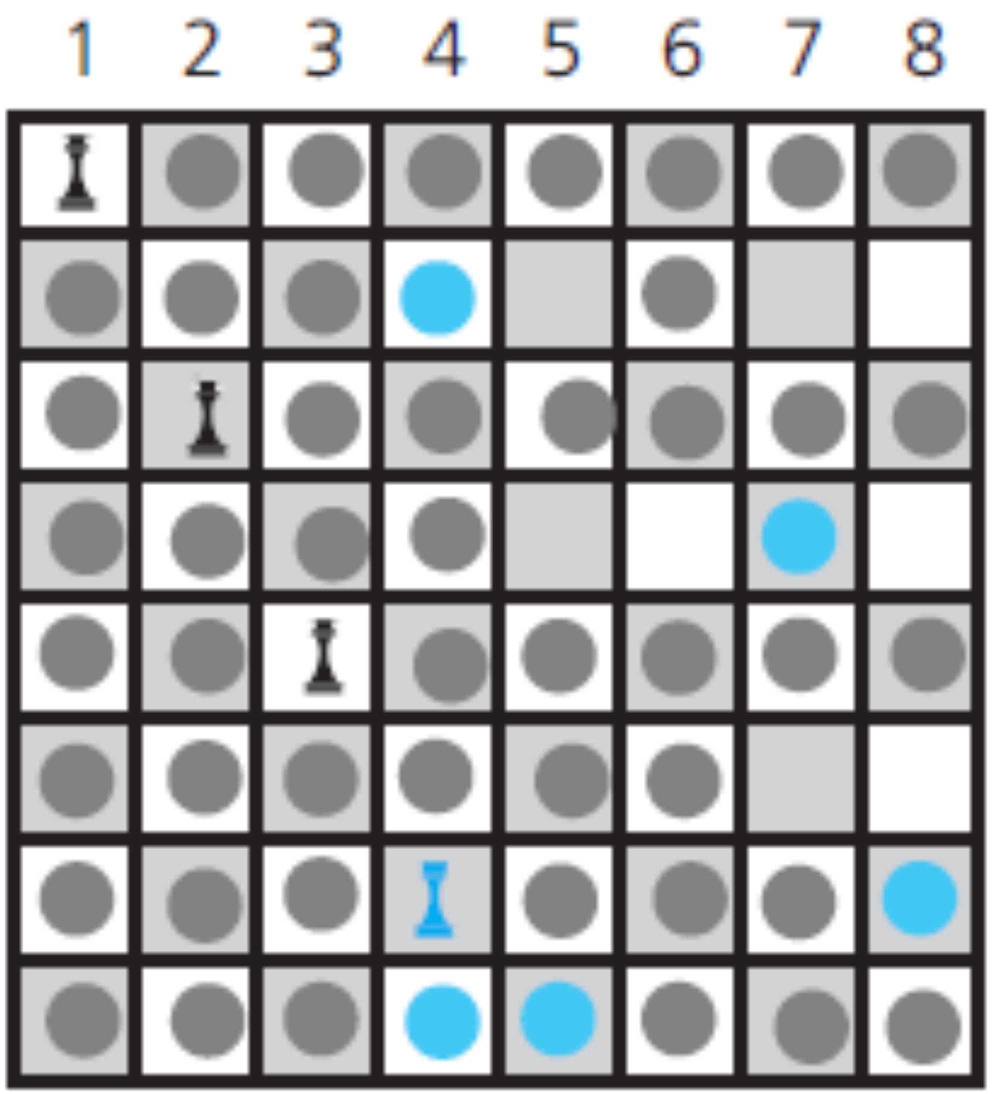
The Eight Queens Problem

Backtracking!



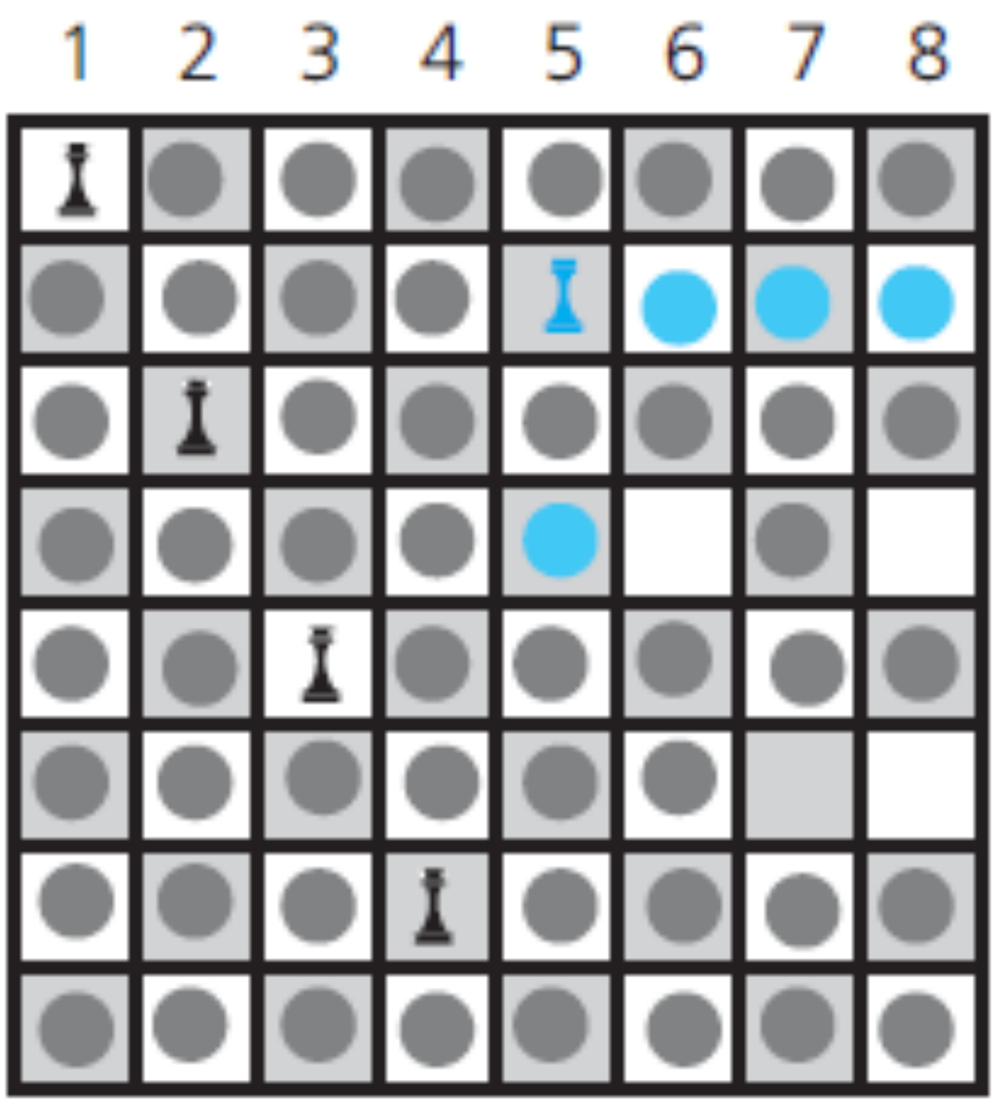
(f) Backtracking to column 5 to try another square for the queen

The Eight Queens Problem



(g) Backtracking to column 4 to try another square for the queen

The Eight Queens Problem



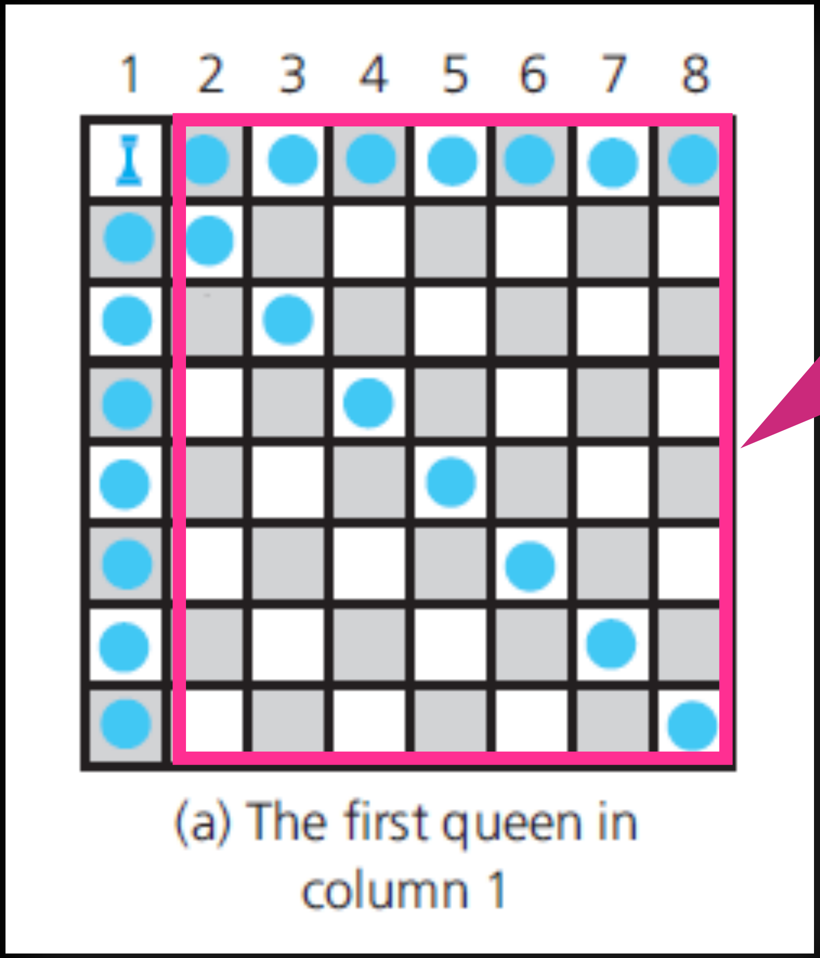
(h) Considering column 5 again

The Eight Queens Problem

How can we express this problem recursively?

The Eight Queens Problem

How can we express this problem recursively?



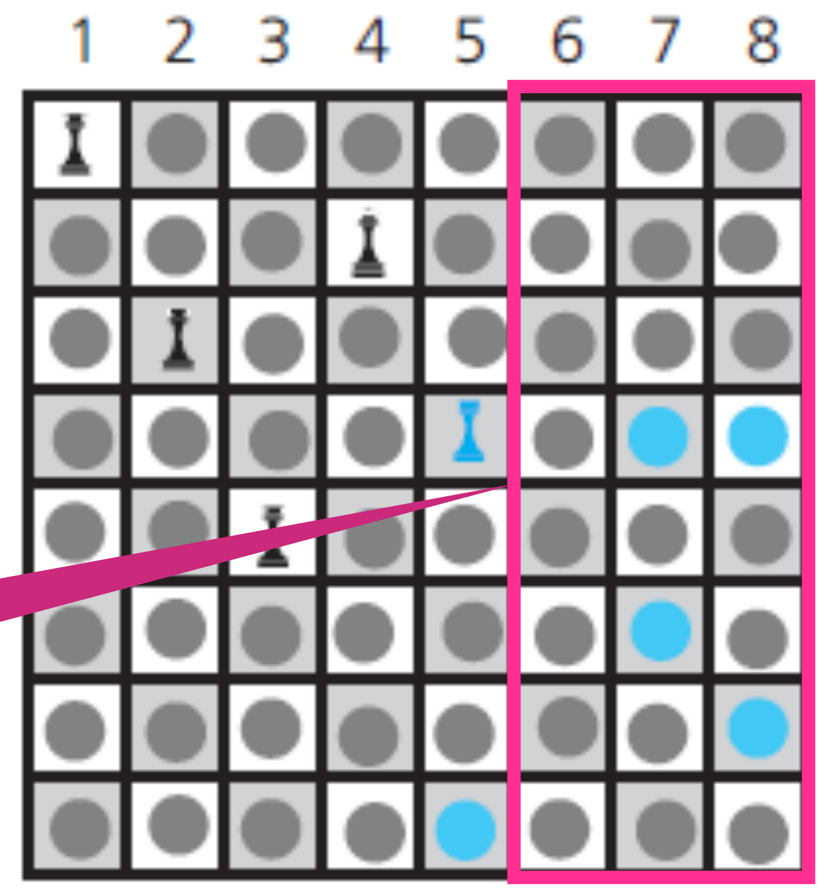
Place queen on column i
Recursively solve on columns (i+1) to 8

The Eight Queens Problem

How do we backtrack?

The Eight Queens Problem

How do we backtrack?

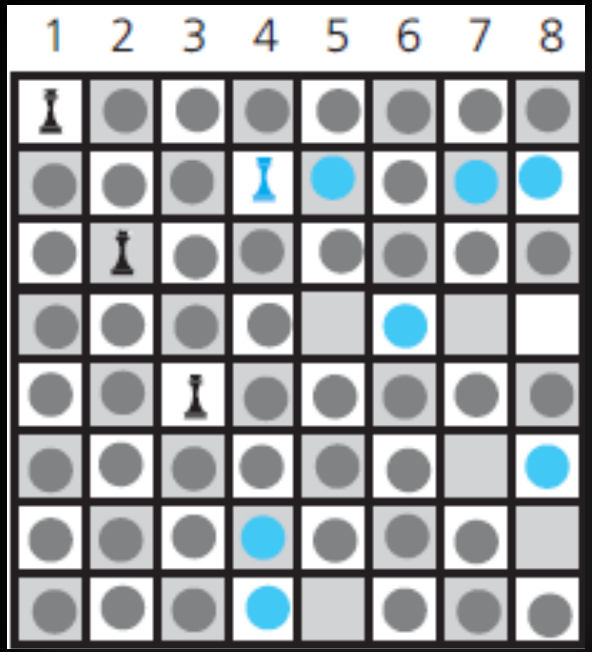


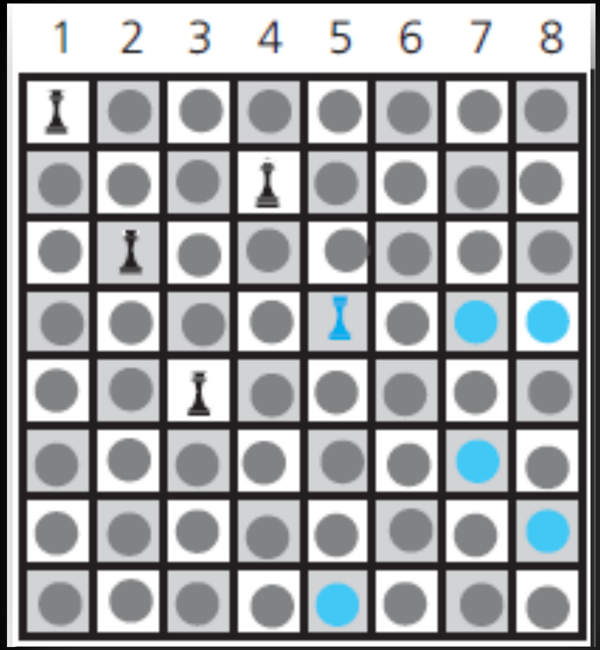
(e) The five queens can attack all of column 6

Communicate to calling function that there are no options left, it should try something else!

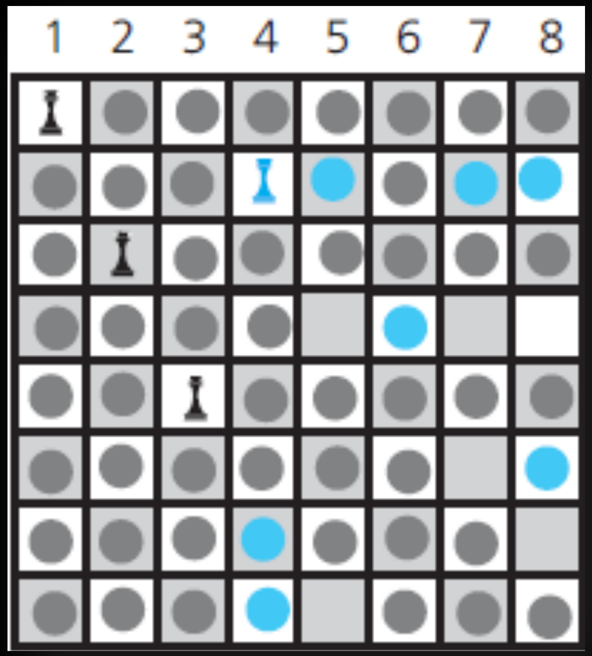
The Eight Queens Problem

```
bool placeQueens(board, column)
{
    if(column > BOARD_SIZE)
        return true; //Problem is solved!
    else
    {
        while(there are safe squares in this column)
        {
            place queen in next safe  square;
            if(placeQueens(board, column+1)) //recursively look forward
                return true; //queen safely placed
        }
        return false; //recursive backtracking
    }
}
```

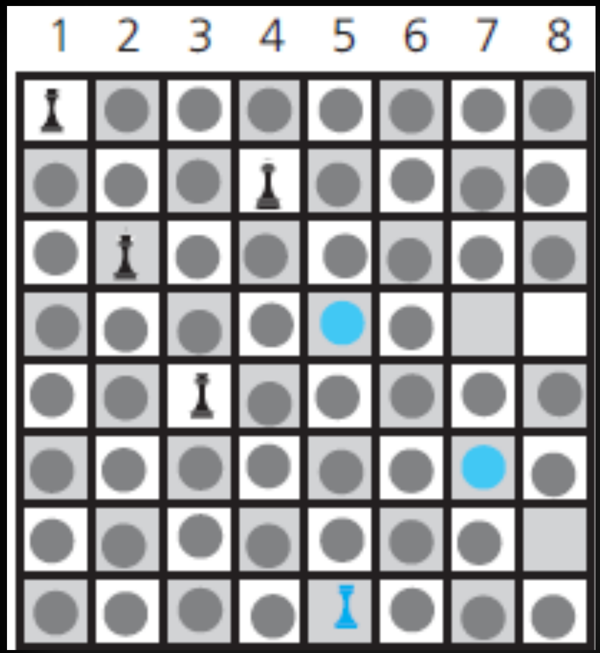




The Eight Queens Problem




```
bool placeQueens(board, column)
{
    if(column > BOARD_SIZE)
        return true; //Problem is solved!
    else
    {
        while(there are safe squares in this column)
        {
            place queen in next safe  square;
            if(placeQueens(board, column+1)) //recursively look forward
                return true; //queen safely placed
        }
        return false; //recursive backtracking
    }
}
```

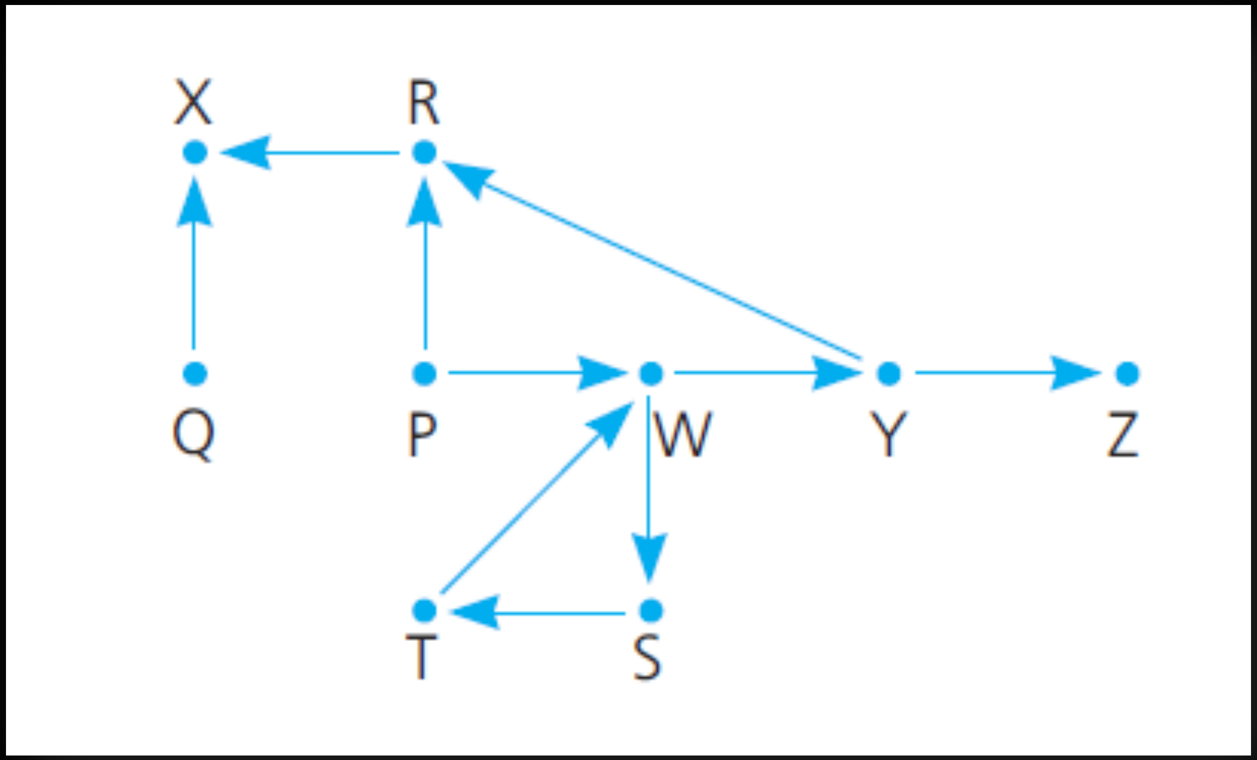


Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

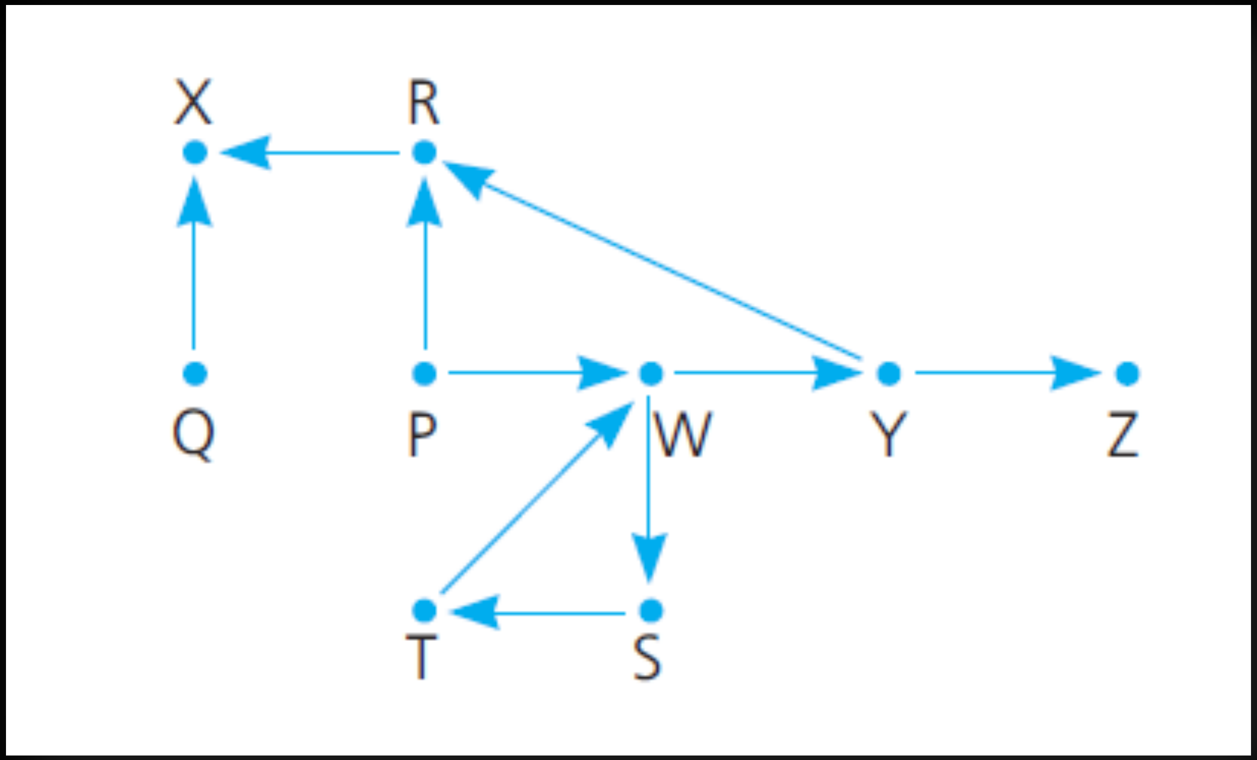
Origin = P , **Destination = Z**



Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.
`bool findPath(map, origin, destination)`

Origin = P , Destination = Z

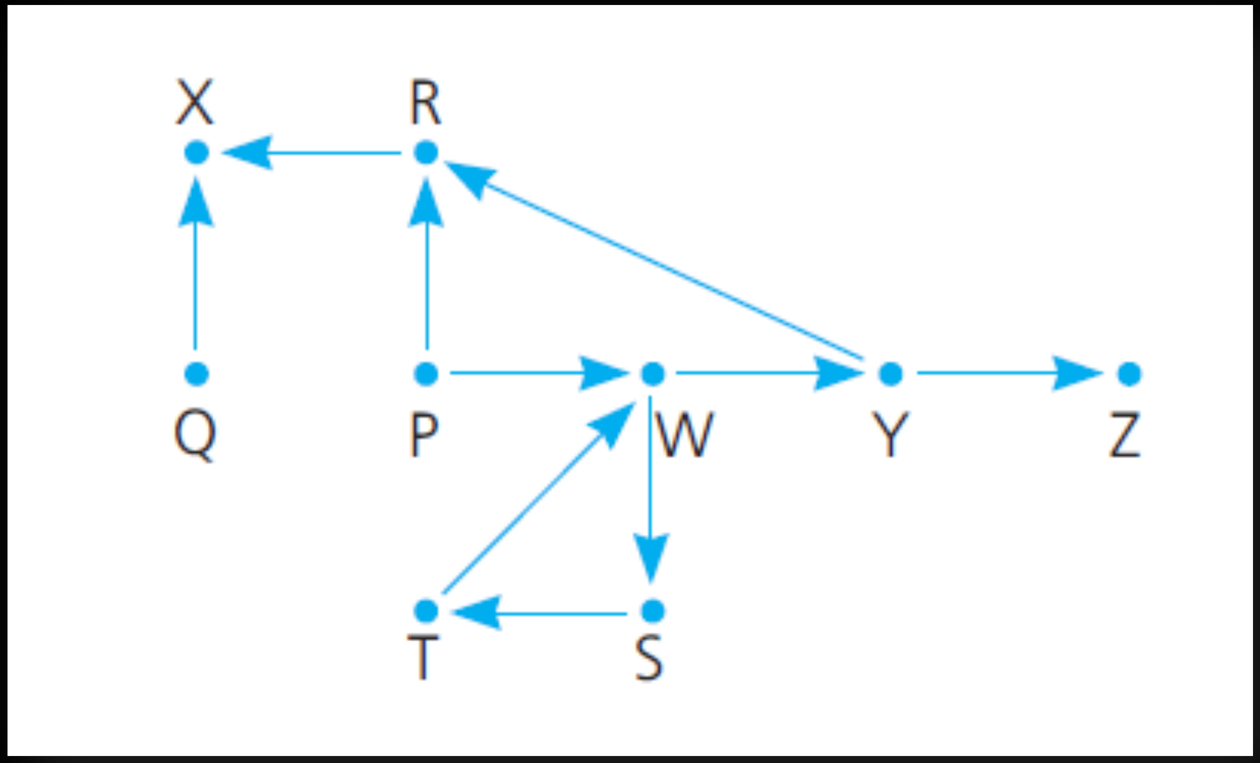


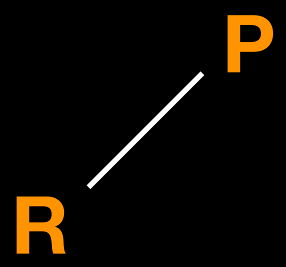
P

Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.
`bool findPath(map, origin, destination)`

Origin = P , Destination = Z



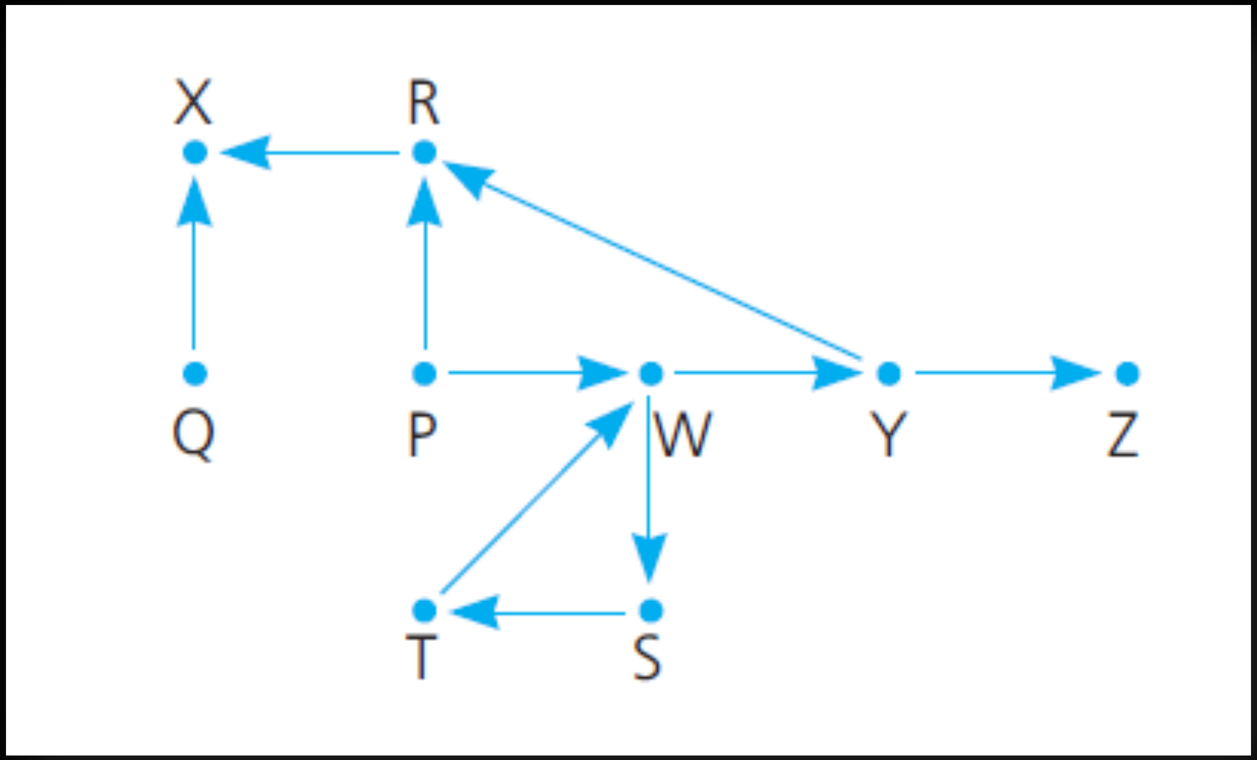


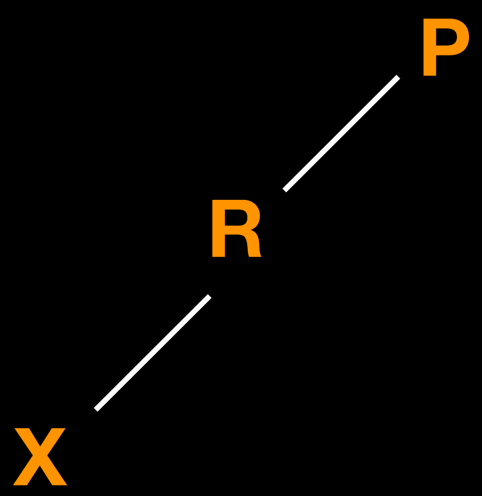
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



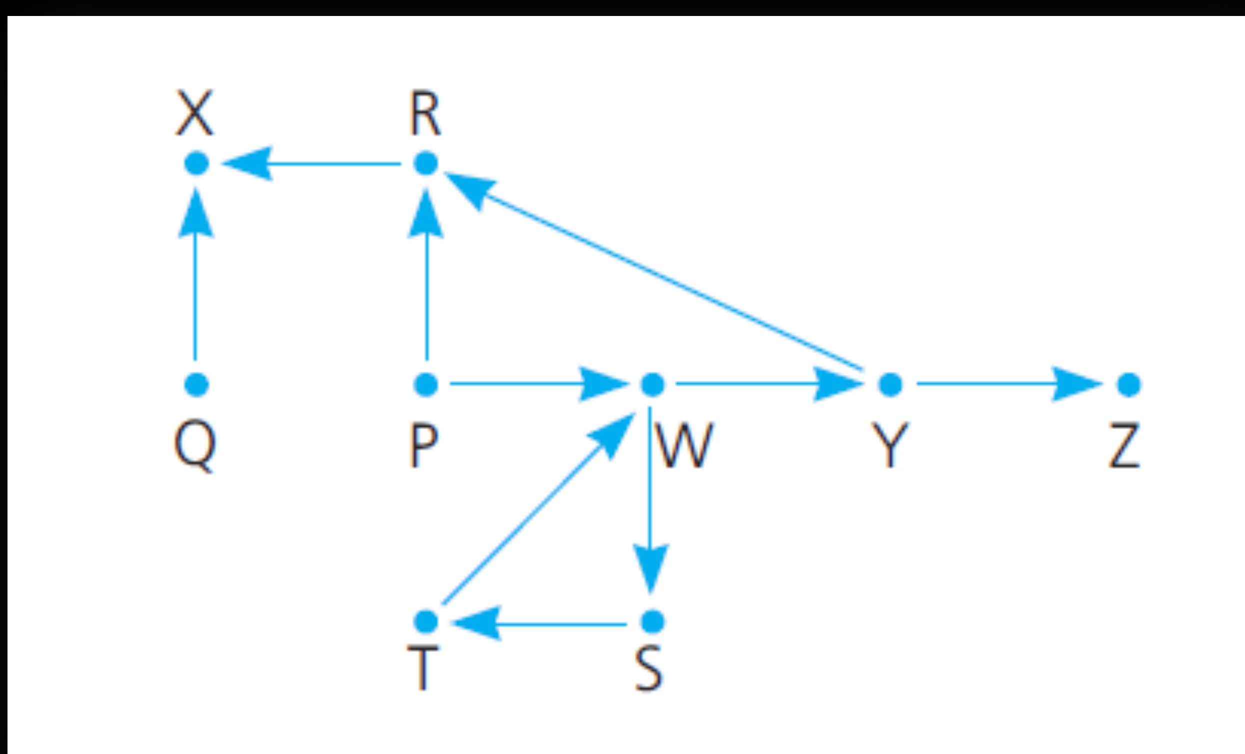


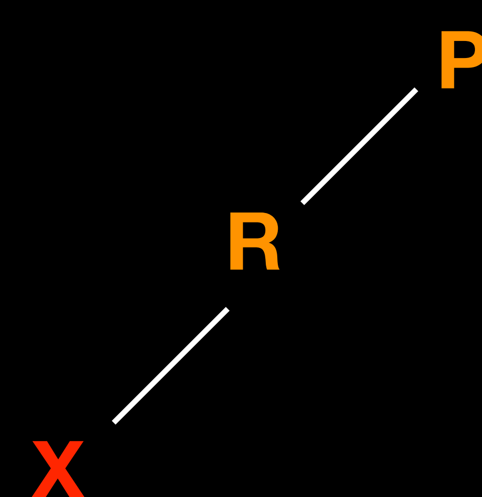
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



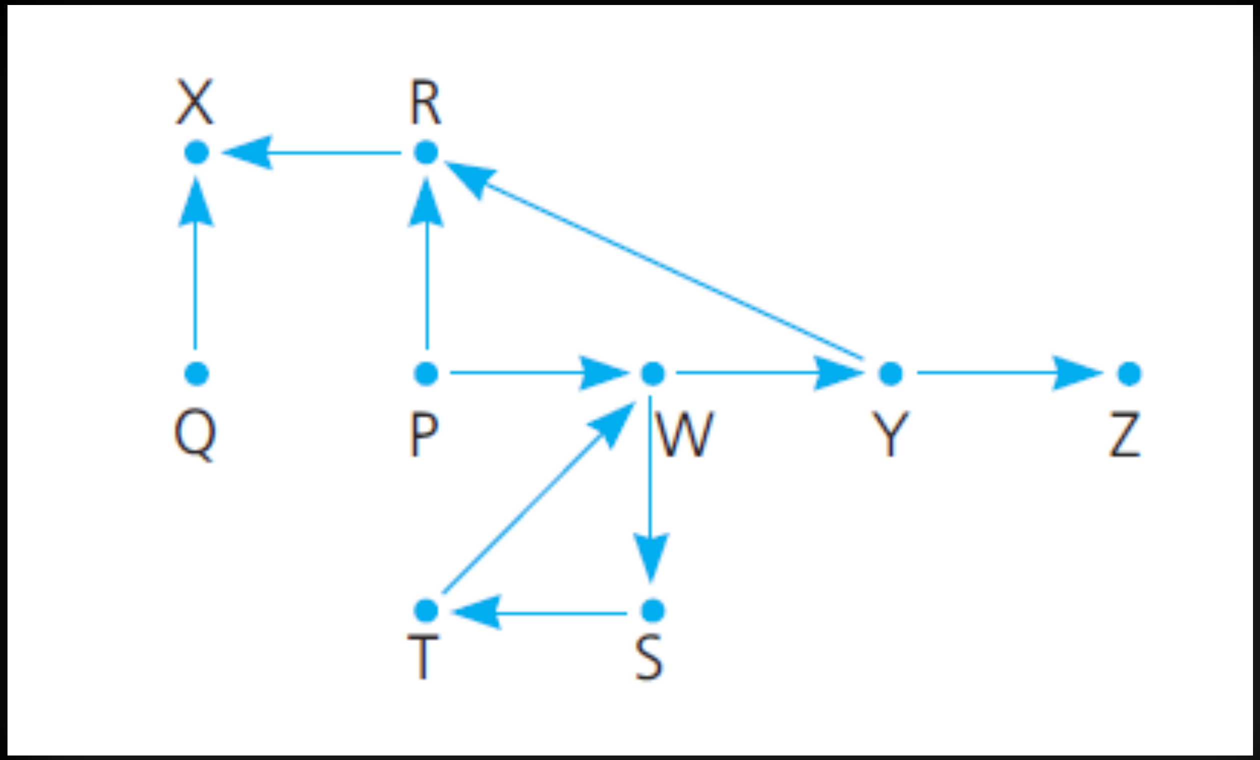


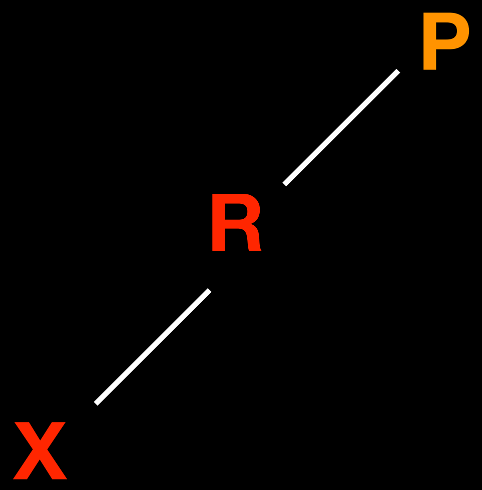
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



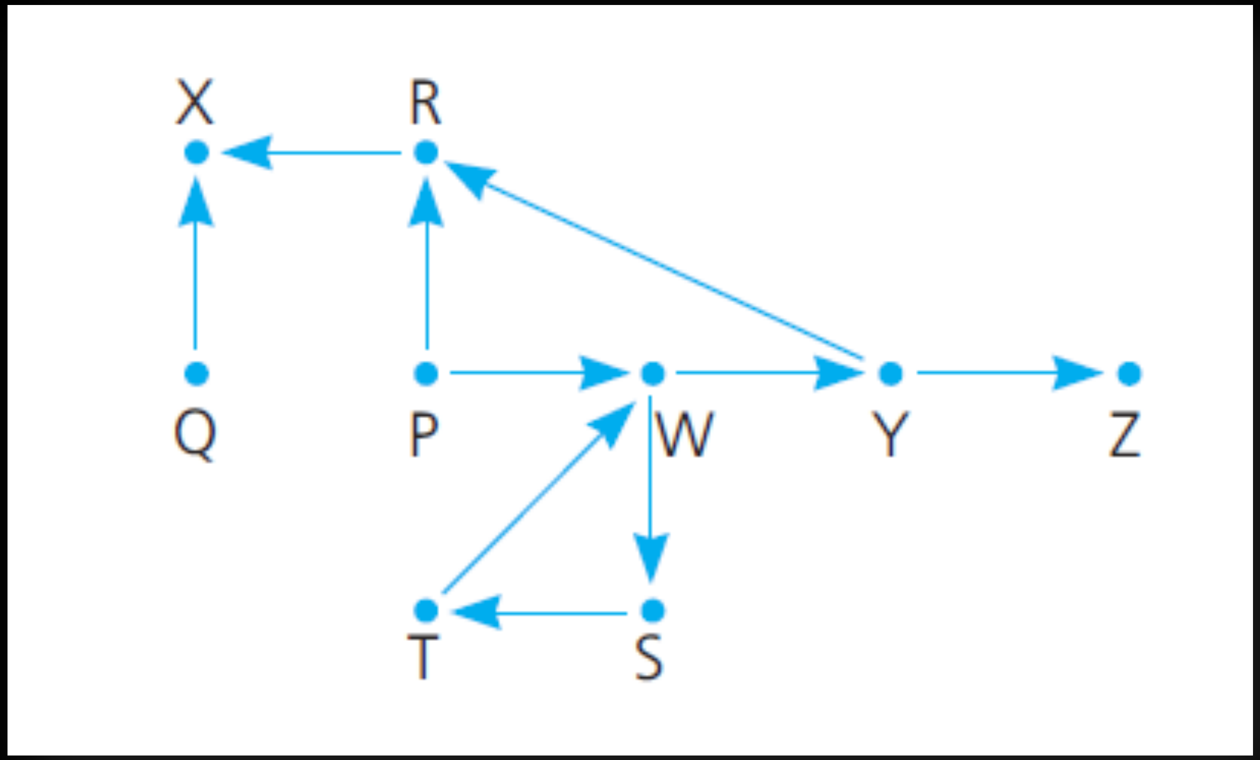


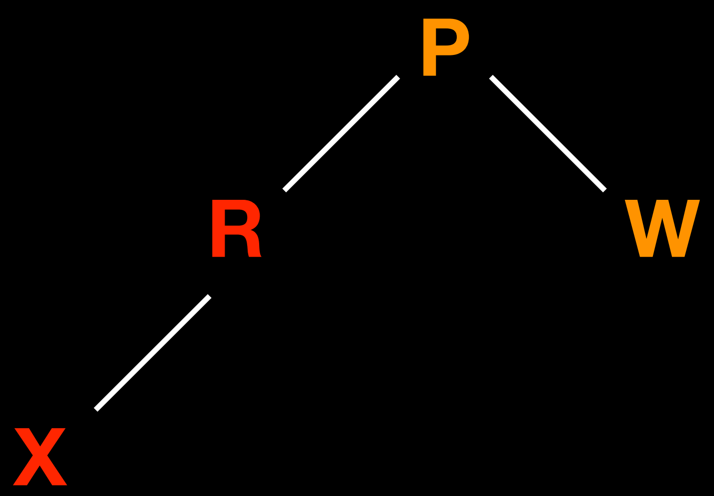
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



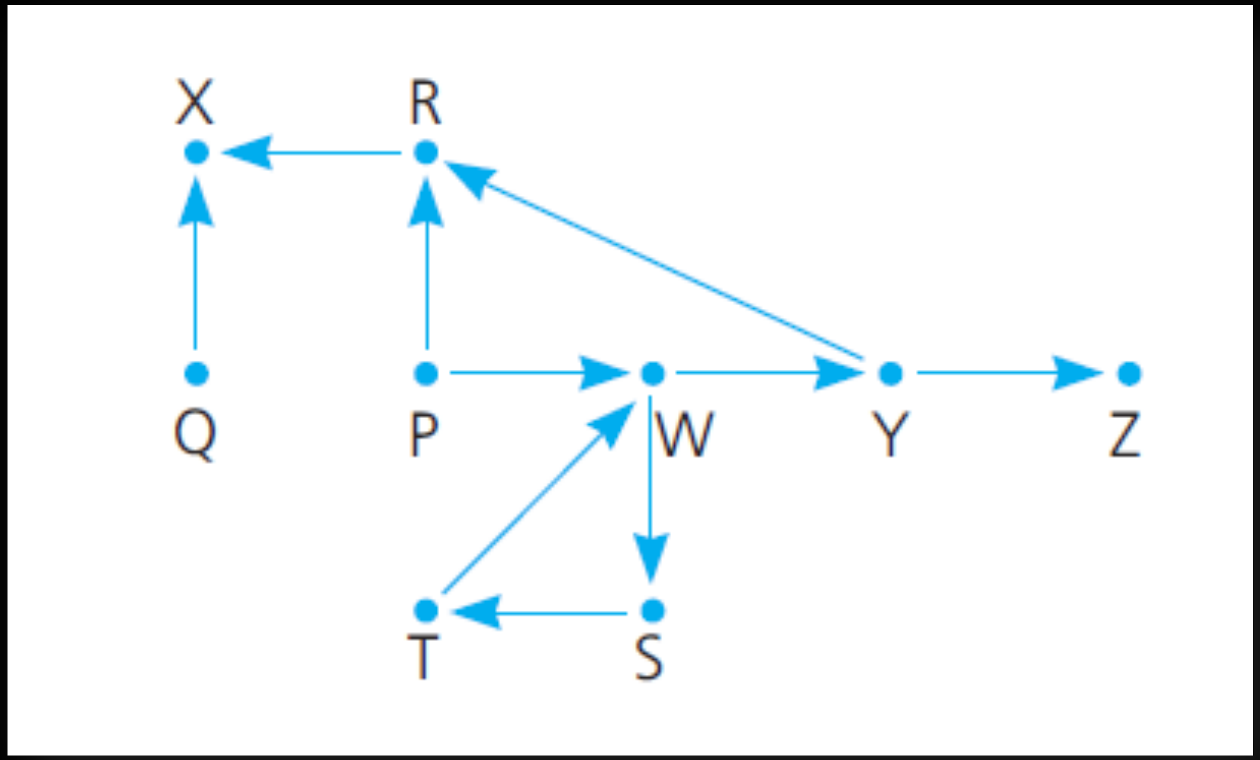


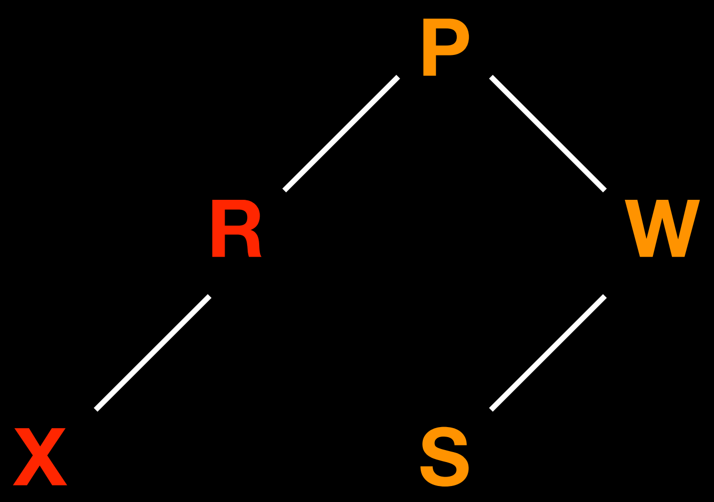
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



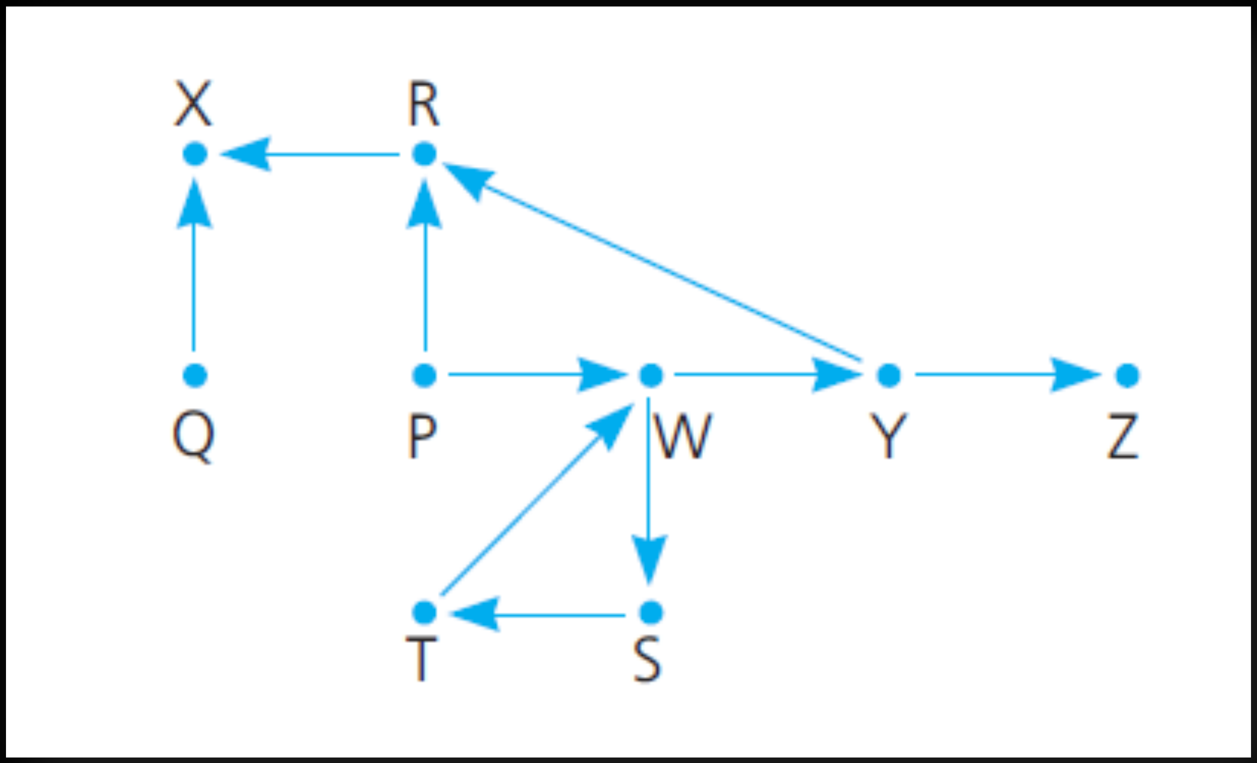


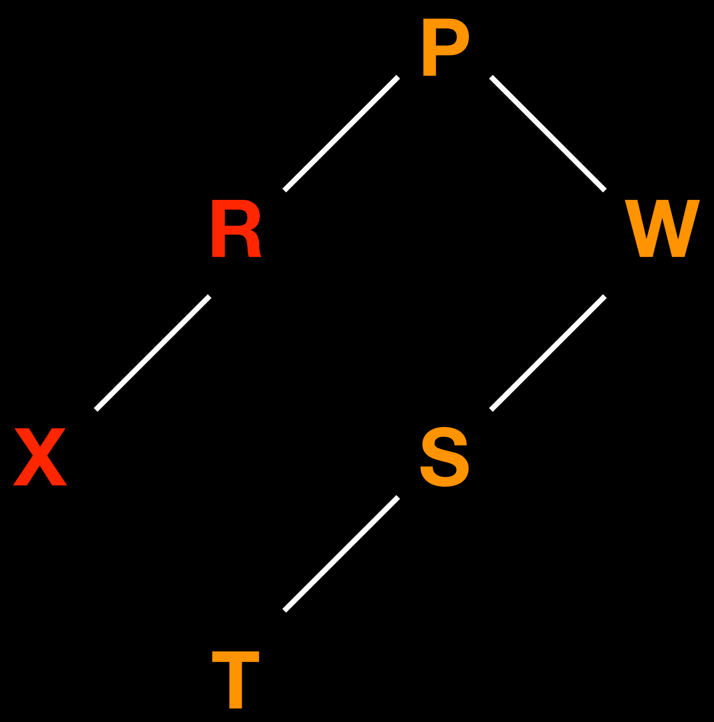
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



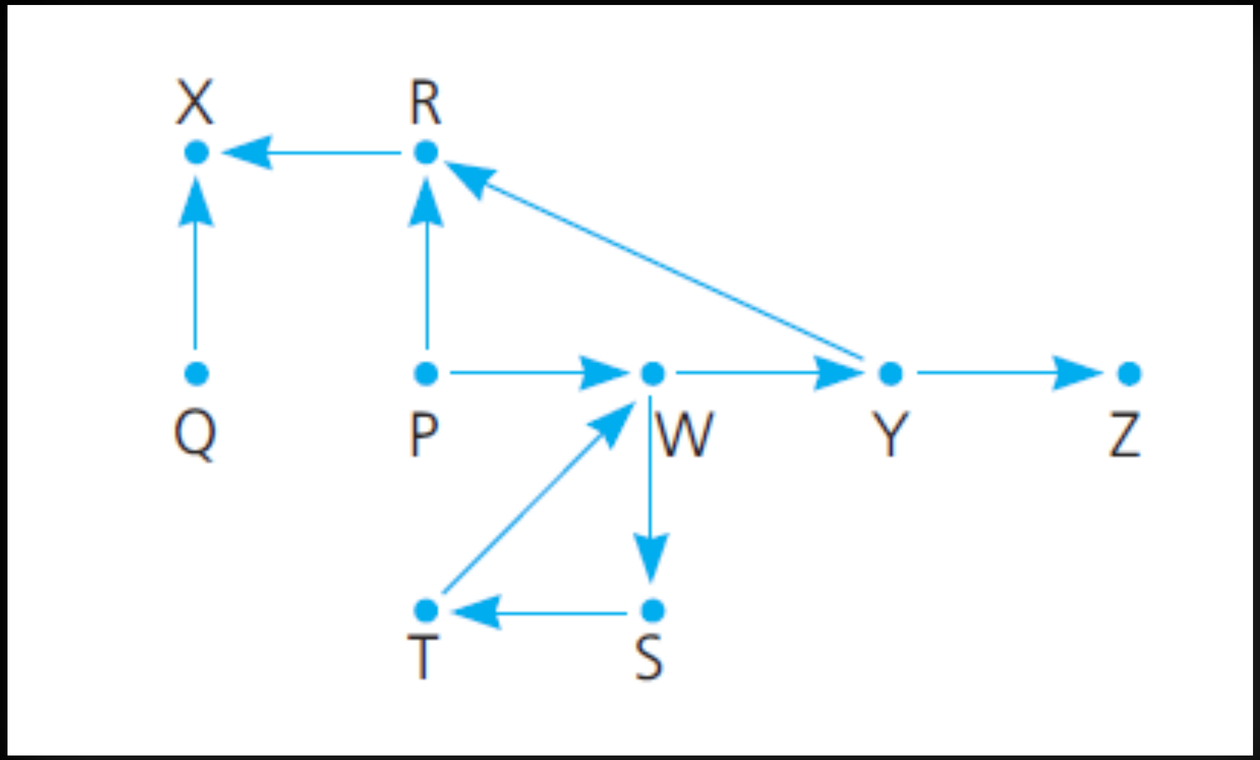


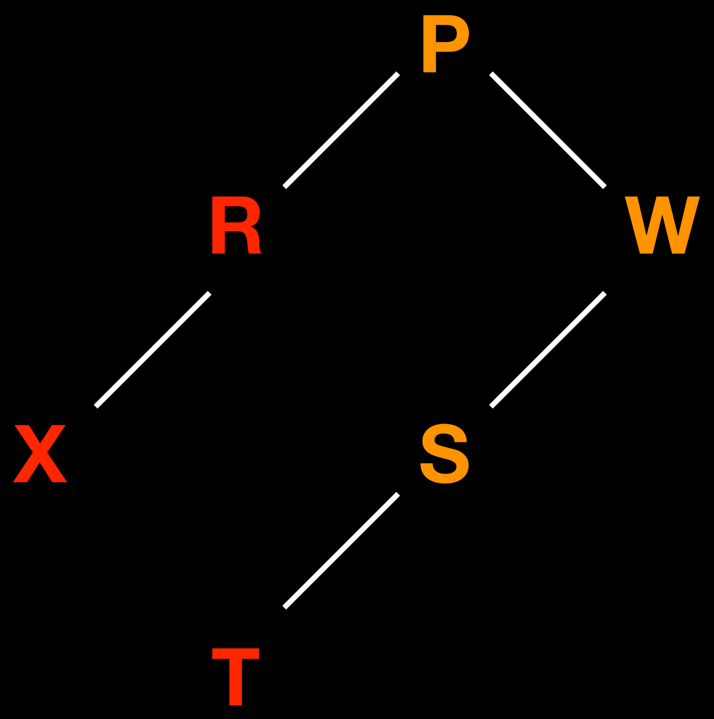
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



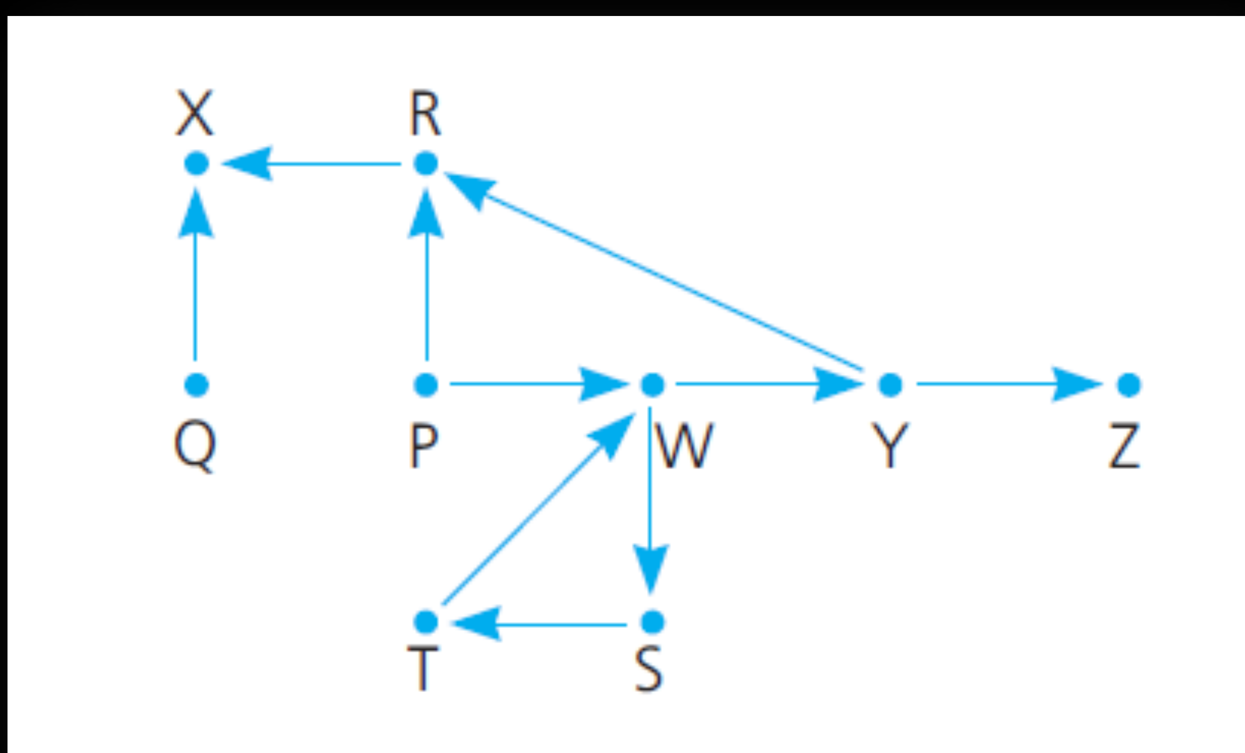


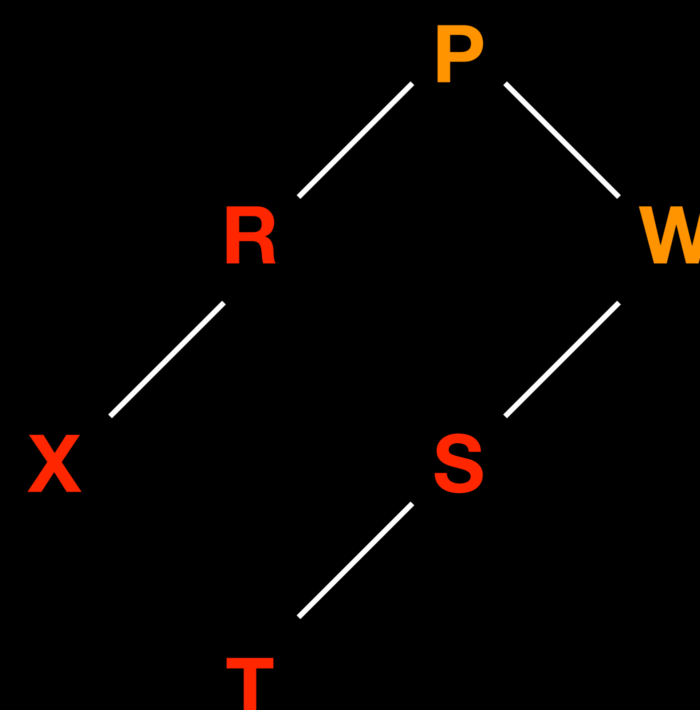
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



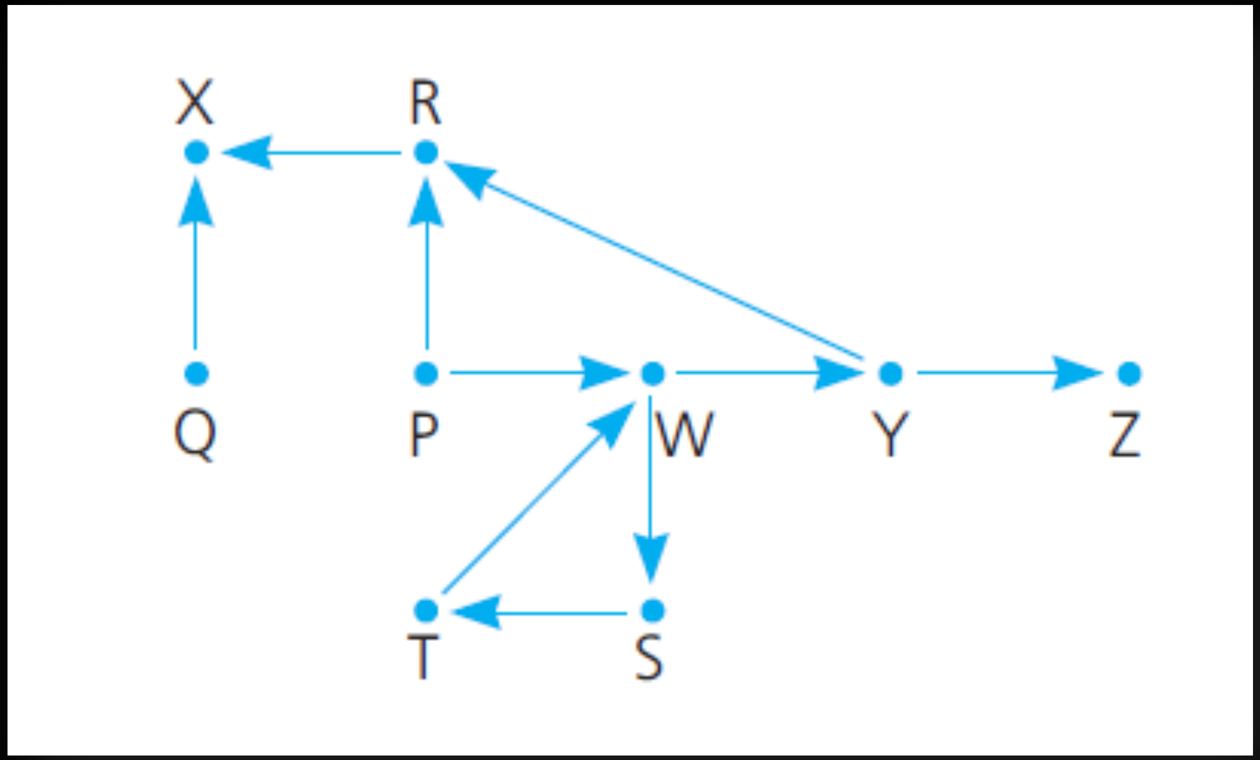


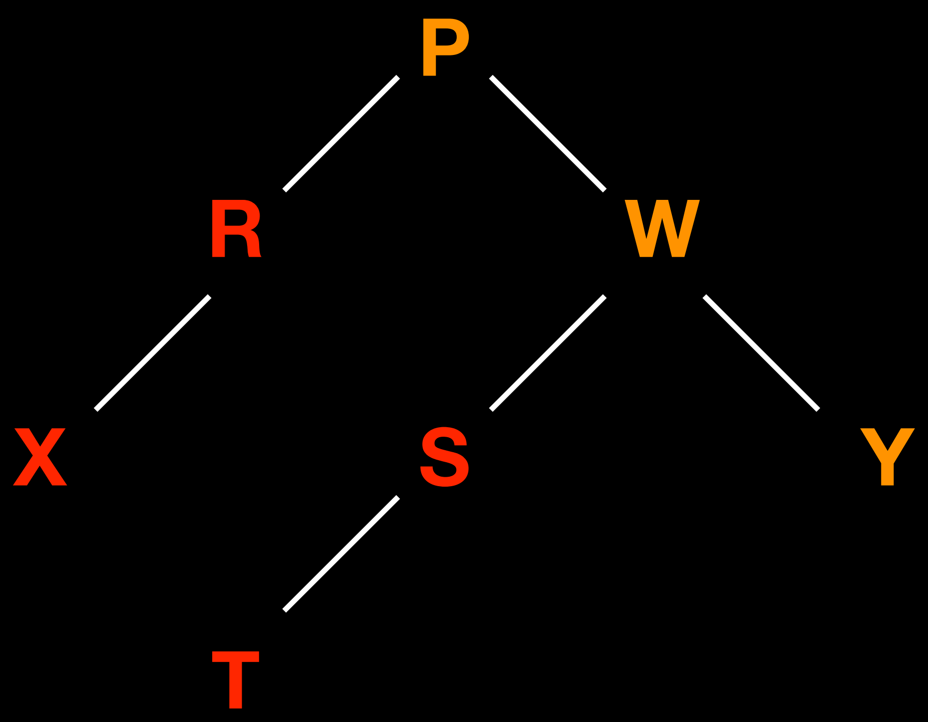
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z



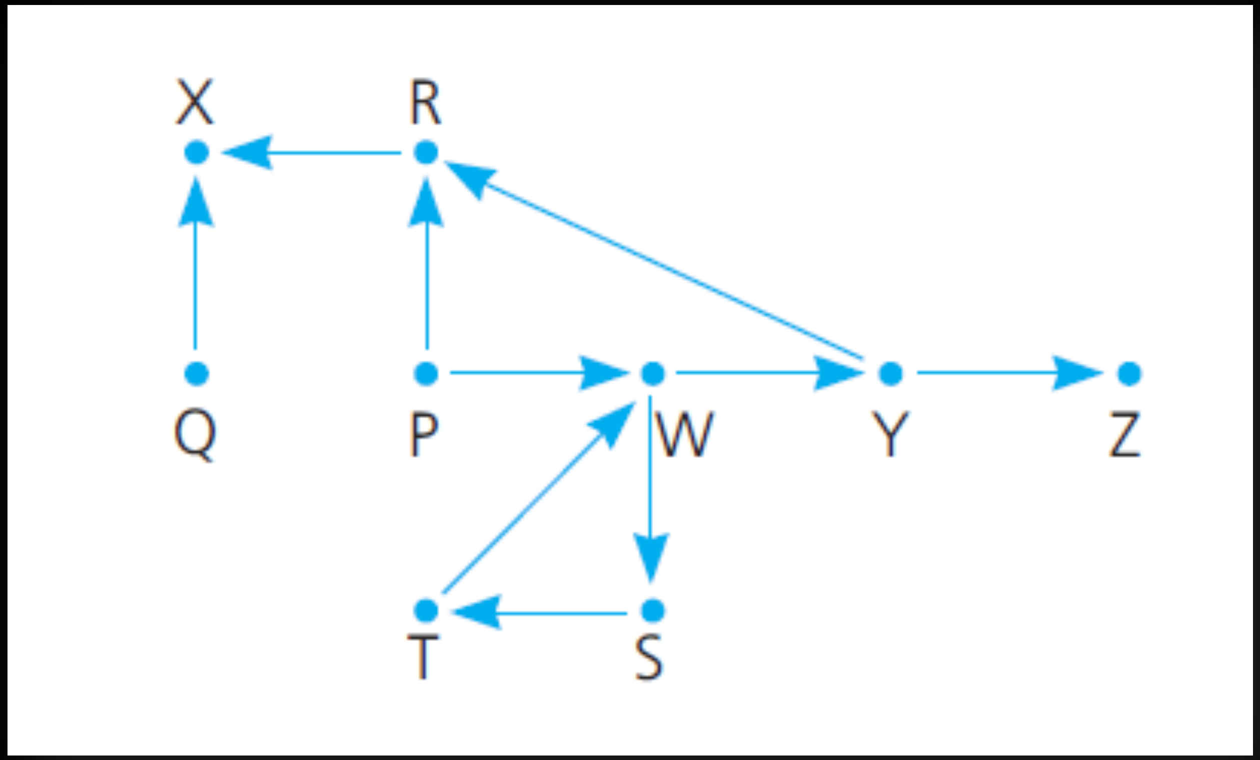


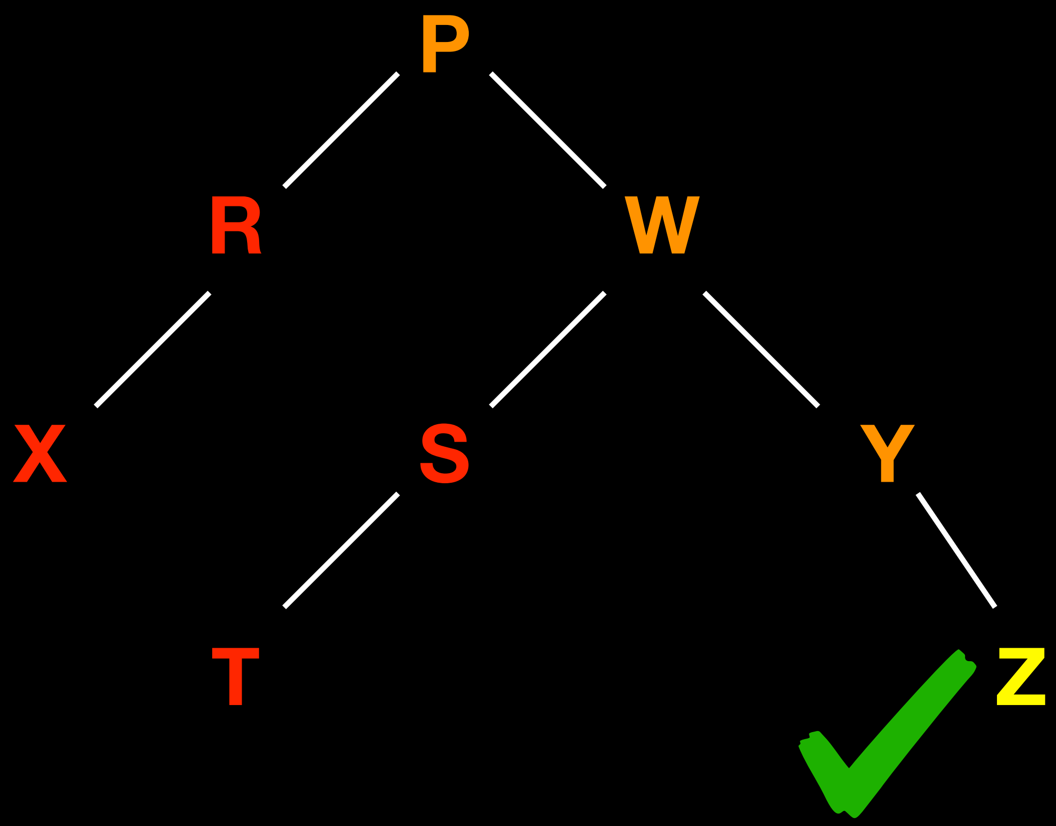
Path Finding

Recursive Backtracking that finds a path from origin to destination.
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Origin = P , Destination = Z





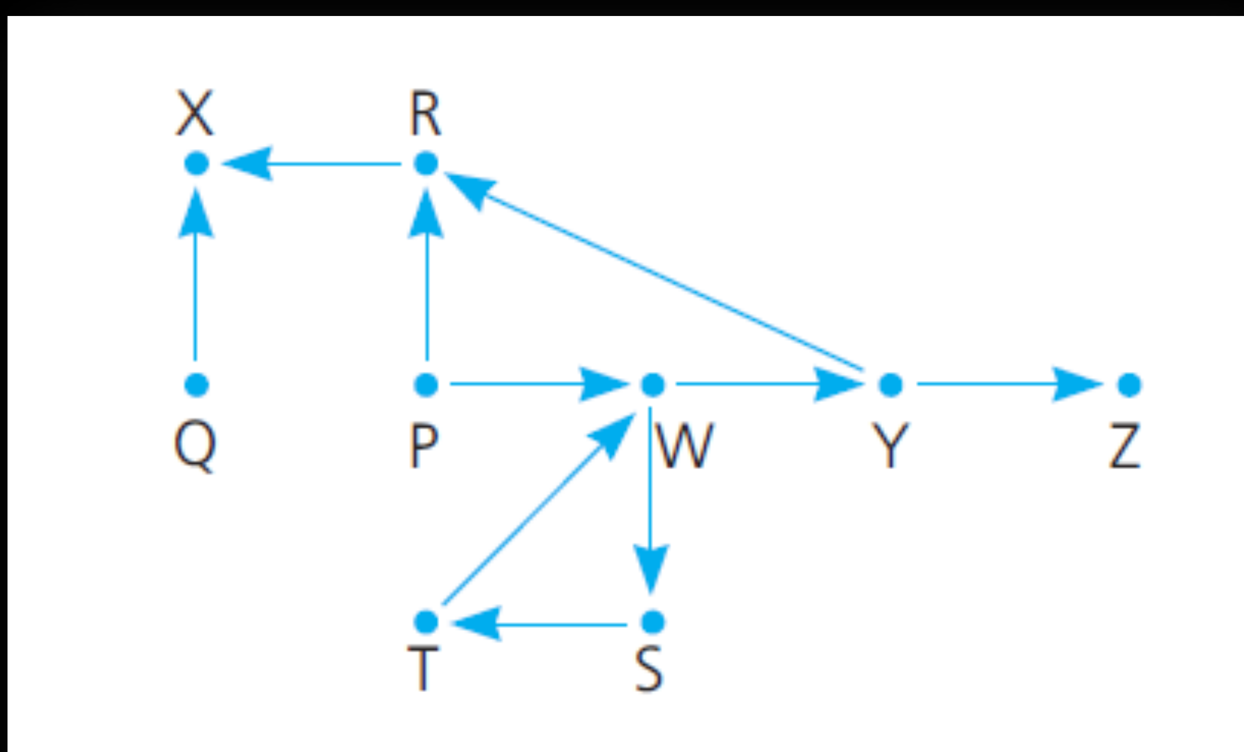
Path Finding

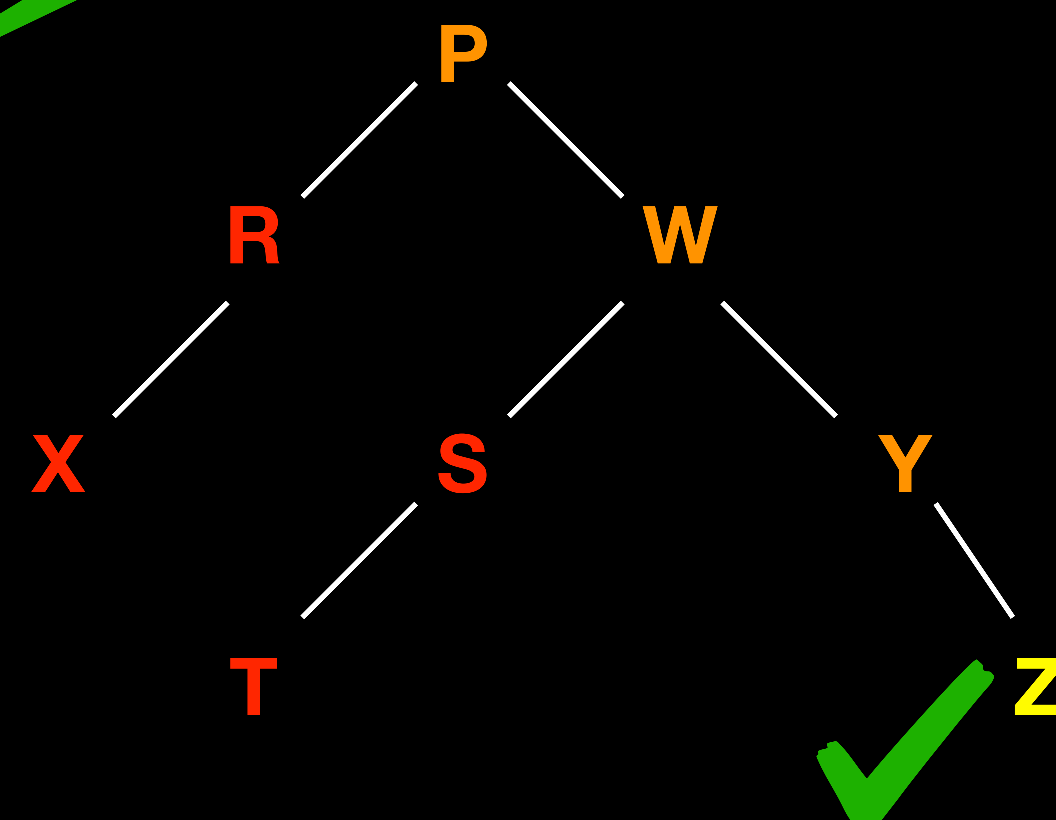
Recursive Backtracking that finds a path from origin to dest
Assume cities are visited in alphabetical order.

```
bool findPath(map, origin, destination)
```

Don't get bogged down by what a map is.
In design phase you know it's available and you can look up where you can go next from

Origin = P , Destination = Z

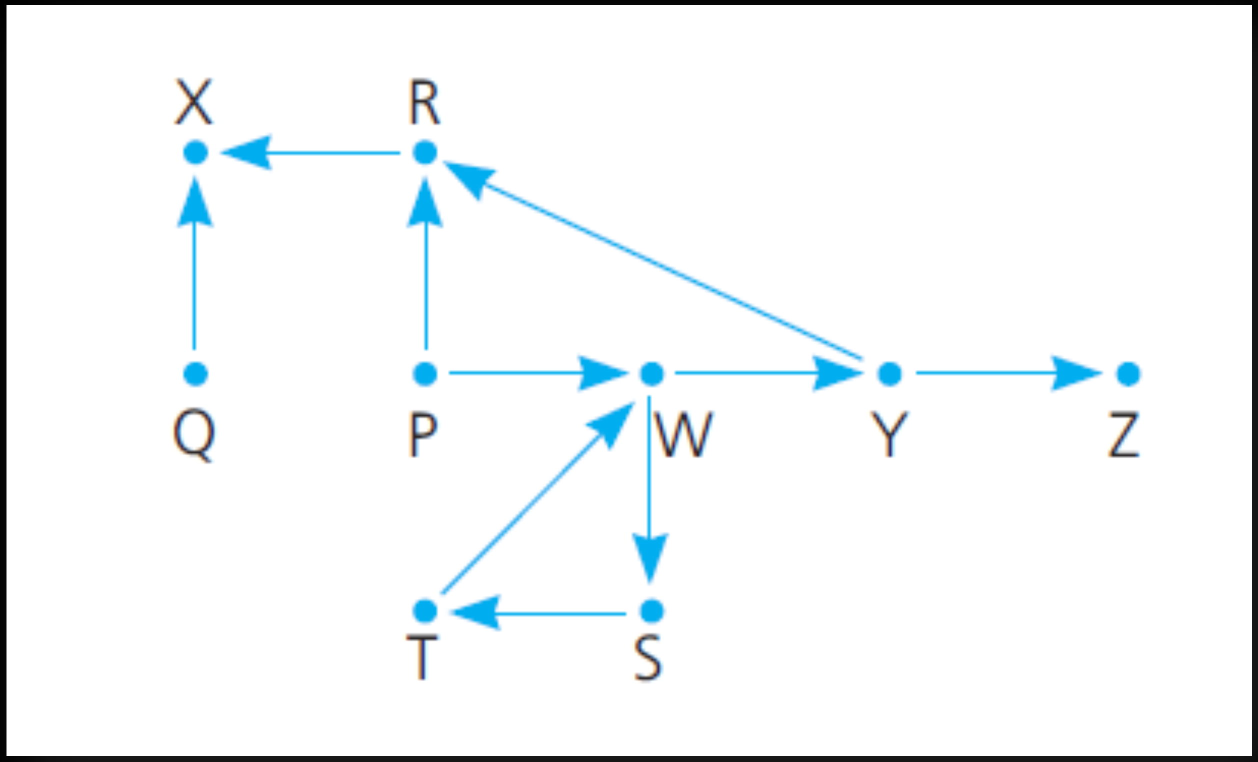


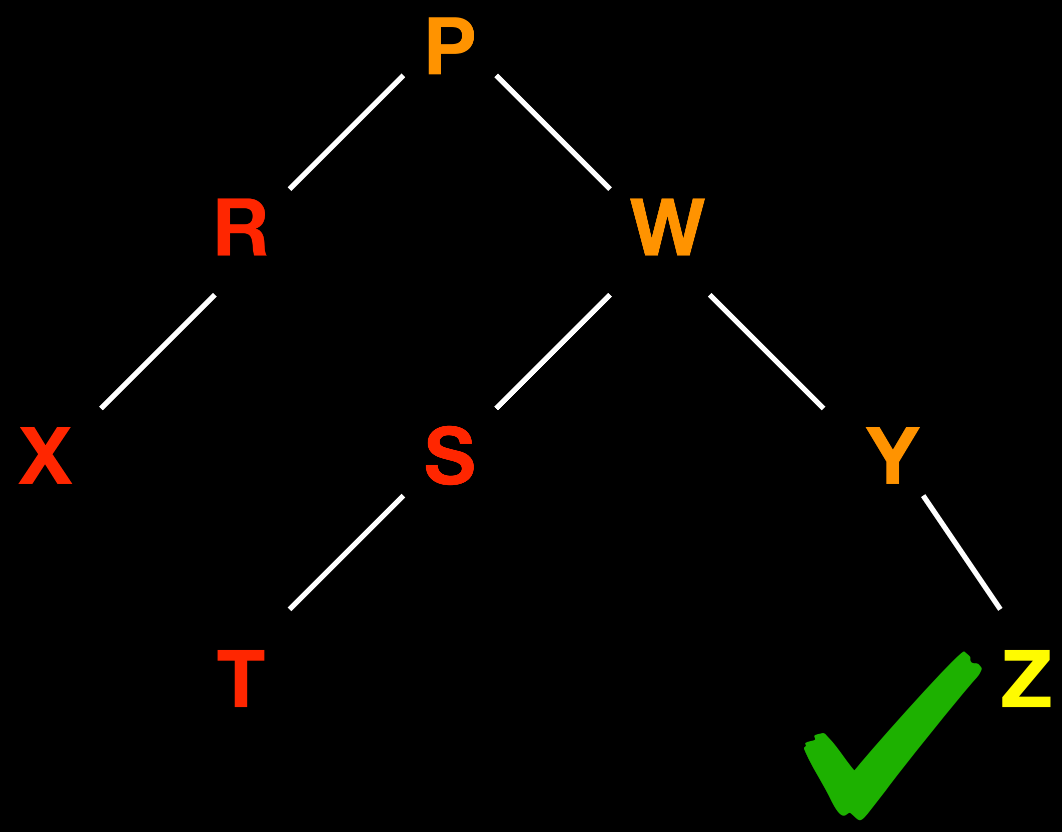


Path Finding

```
bool findPath(map, origin, destination)
{
    mark origin as visited in map
    if origin == destination
        return true
    else
        for each unvisited city C reachable from origin
            if findPath(map, C, destination)   <-- Recursive call
                return true
        return false //recursive backtracking
}
```

Origin = P , Destination = Z





Recursion and Induction

Principle of Mathematical Induction:

Suppose you want to prove that a statement `P(n)` about an integer `n` is true for every positive integer `n`.

To prove that `P(n)` is true for all $n \geq 1$, do the following two steps:

- **Base Step:** Prove that `P(1)` is true.
- **Inductive Step:** Let `k ≥ 1`. Assume `P(k)` is true, and prove that `P(k + 1)` is also true.

Recursion and Induction

```
//a: nonzero real number, n: nonnegative integer
power(a, n)
{
  if (n = 0)
    return 1
  else
    return a * power(a, n - 1)
}
```

Prove by mathematical induction on n that the algorithm above is correct. We will show $P(n)$ is true for all $n \geq 0$, where
$P(n)$: For all nonzero real numbers a, power(a, n) correctly computes a^n.

Recursion and Induction

Base step: If $n = 0$, the first step of the algorithm tells us that `power(a,0)=1`. This is correct because $a^0 = 1$ for every nonzero real number a, so P(0) is true.

Inductive step:

Let $k \geq 0$.

Inductive hypothesis: `power(a, k)` $= a^k$, for all a != 0.

We must show next that `power(a, k+1)` $= a^{k+1}$.

Since $k + 1 > 0$ the algorithm sets

`power(a, k + 1) = a * power(a, k)`

By inductive hypotheses `power(a, k) =` a^k

so `power(a, k + 1) = a* power(a, k)` $= a * a^k = \mathbf{a^{k+1}}$