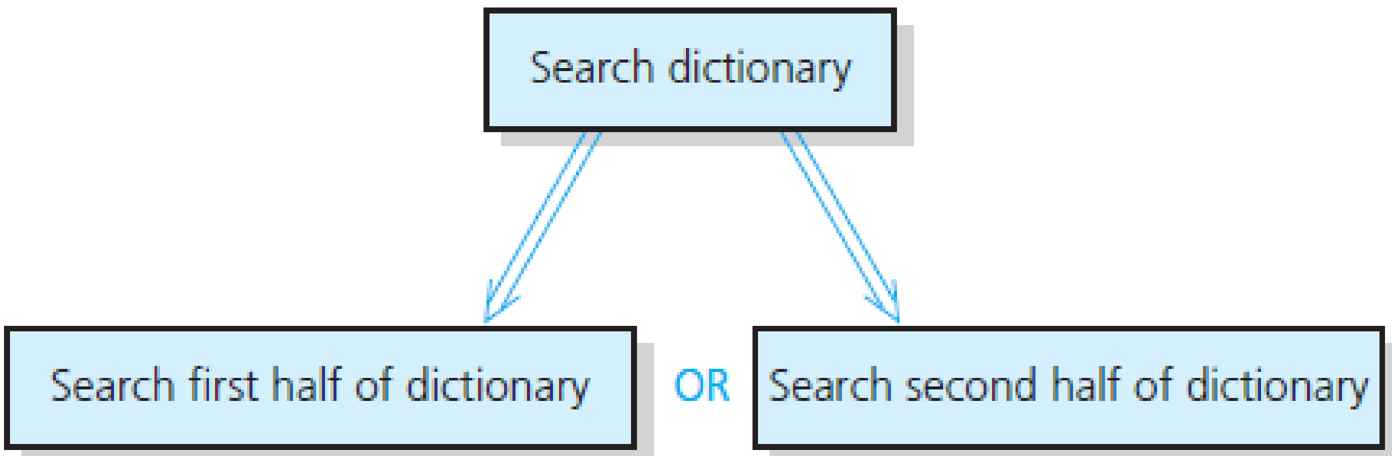
EECE 2560:
Fundamentals of Engineering Algorithms

Recursion

Recursive Solutions (1 of 3)

- Recursion breaks problem into smaller identical problems
  - An alternative to iteration
- **Example** A recursive search:



Recursive Solutions (2 of 3)

- A recursive function calls itself
- Each recursive call solves an identical, but smaller, problem
- Test for **base case** enables recursive calls to stop
- Eventually, one of smaller problems must be the base case

Recursive Solutions (3 of 3)

Questions for constructing recursive solutions

1. How to define the problem in terms of a smaller problem of same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of problem can serve as base case?
4. As problem size diminishes, will you reach base case?

The Factorial of n (1 of 3)

- An iterative solution:

$$\textbf{factorial}(n) = n \times (n-1) \times (n-2) \times \ldots \times 1 \text{ for an integer } n > 0$$

$$\textbf{factorial}\,(0)=1$$

- A recursive solution:

$$\textbf{factorial}(n) = \begin{cases} 1 & \textbf{if} \quad n = 0 \\ n \times \textbf{factorial}(n-1) & \textbf{if} \quad n > 0 \end{cases}$$

Note: Do not use recursion if a problem has a simple, efficient iterative solution

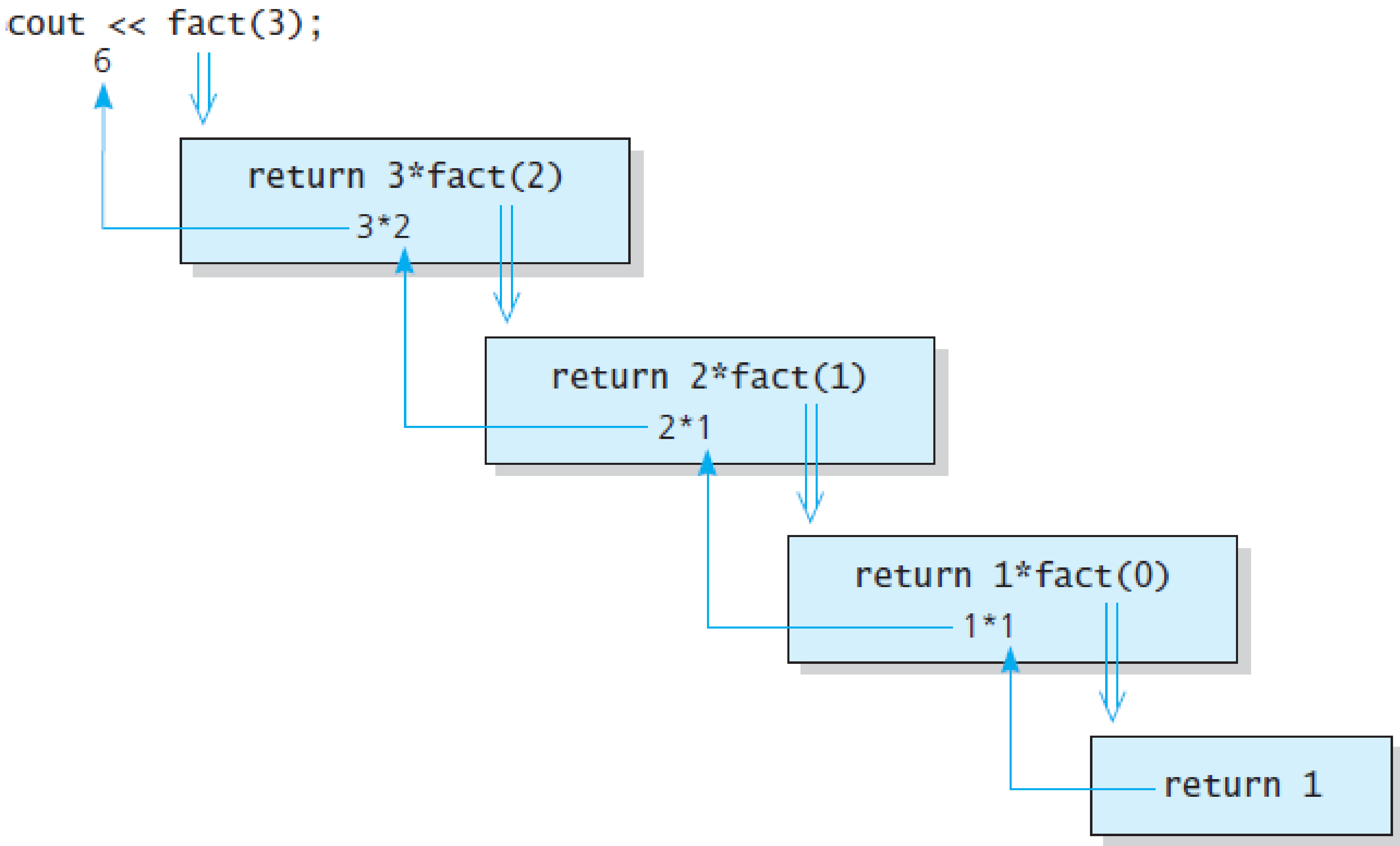
The Factorial of n (2 of 3)

```
int factItr(int n)
{
  int fc=1;
  for(int i = 1; i <=n; ++i) {
        fc *= i;
    }

  return fc;
}  // end factItr
```

```
int fact(int n)
{
   if (n == 0)
      return 1;
   else
      return n * fact(n - 1); // n * (n-1)! is n!
}  // end fact
```

The Factorial of n (3 of 3)



The Binary Search (1 of 4)

<u>Problem</u>: Implement function **binarySearch** to search for **target** in **anArray**.

Consider the following details before implementing the algorithm:

1. How to pass half of **anArray** to recursive calls of **binarySearch** ?
2. How to determine which half of array contains **target**?
3. What should base case(s) be?
4. How will **binarySearch** indicate result of search?

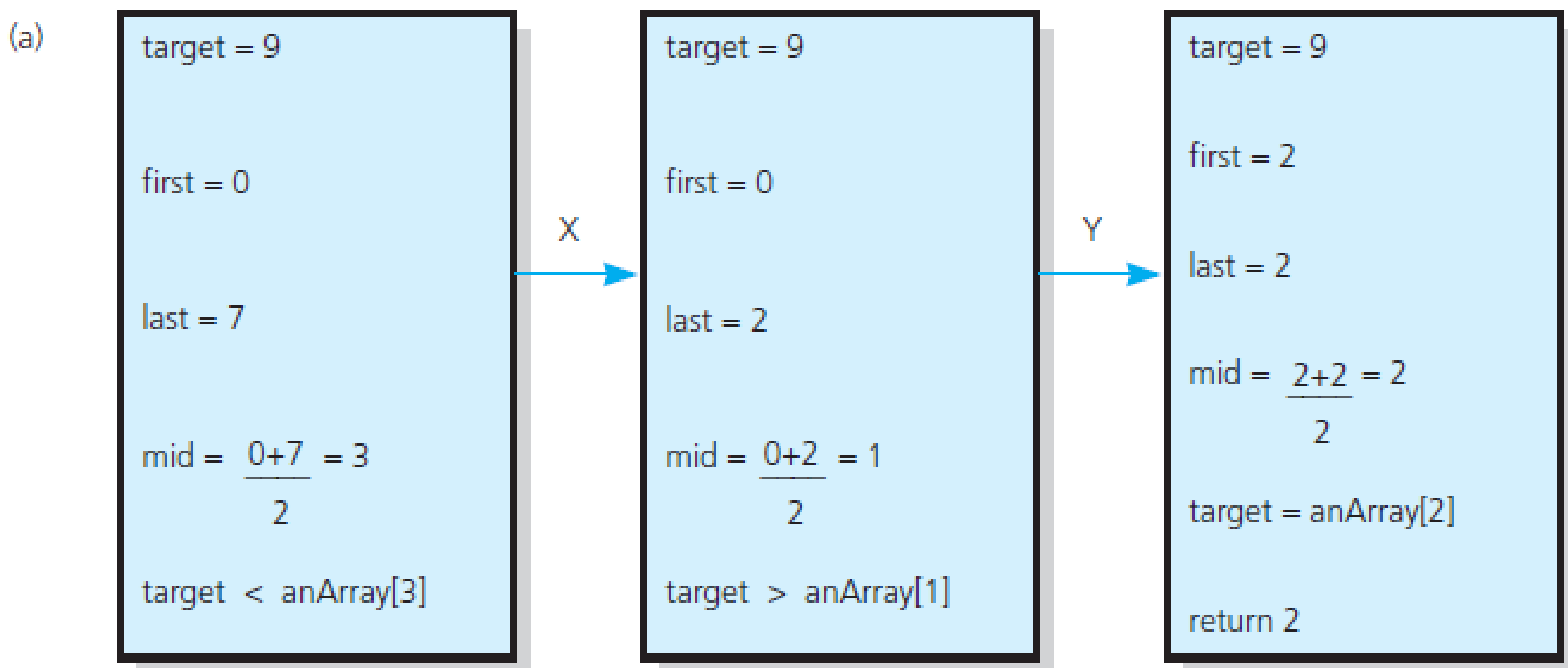
The Binary Search (2 of 4)

```
int binarySearch(const int anArray[], int first, int last, int target)
{
   int index;
   if (first > last)
      index = -1; // target not in original array
   else
   {
      // If target is in anArray, then
      // anArray[first] <= target <= anArray[last]
      int mid = first + (last - first) / 2;
      if (target == anArray[mid])
         index = mid; // target found at anArray[mid]
      else if (target < anArray[mid])
         index = binarySearch(anArray, first, mid - 1, target); //X
      else
         index = binarySearch(anArray, mid + 1, last, target); //Y
   } // end if

   return index;
} // end binarySearch
```

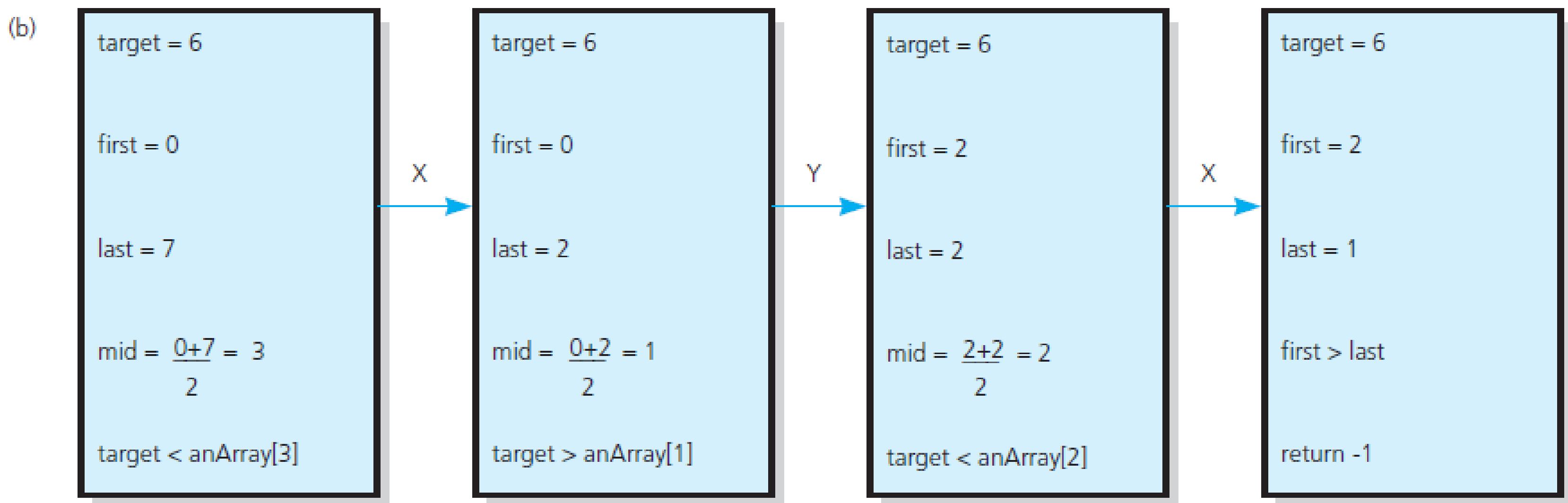
The Binary Search (3 of 4)

Box traces of **binarySearch** with **anArray** = <1, 5, 9, 12, 15, 21, 29, 31>:
(a) a successful search for 9;



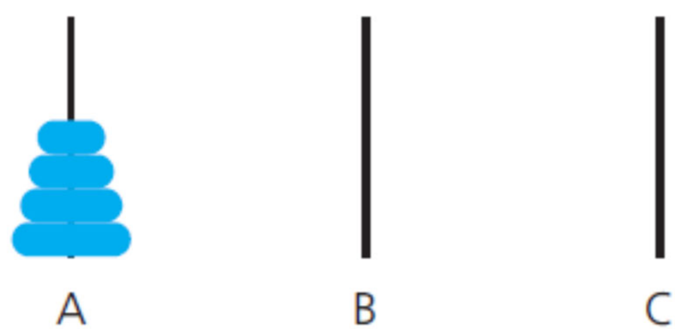
The Binary Search (4 of 4)

Box traces of **binarySearch** with **anArray** = <1, 5, 9, 12, 15, 21, 29, 31>:
(b) an unsuccessful search for 6



The Towers of Hanoi

- Given *n* disks and three poles: *A*, *B*, and *C*.
- The disks were of different sizes. Any disk can be placed only on **top of disks larger** than it.
- Initially, all the disks are on pole *A*.
- The puzzle is to move the disks, **one by one**, from pole *A* to pole *B*. You can use pole *C* in the course of the transfer.

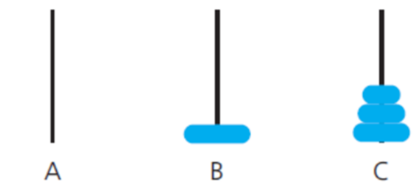


The Towers of Hanoi (2 of 5)

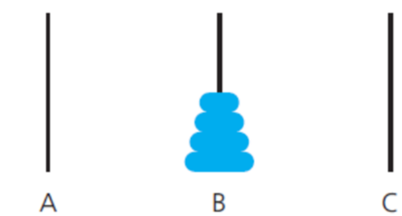
(a) The initial state:

(b) After moving *n* - 1 disks from *A* to *C*:

(c) After moving 1 disk from *A* to *B*:



(d) After moving *n* - 1 disks from *C* to *B*:



The Towers of Hanoi (3 of 5)

– Problem: beginning with *n* disks on pole A and zero disks on poles B and C, **solveTowers(n, A, B, C).**

- Solution

1. With all disks on A, solve **solveTowers(n – 1, A, C, B)**
2. With the largest disk on pole A and all others on pole C, solve **solveTowers(1, A, B, C)**
3. With the largest disk on pole B and all the other disks on pole C, solve **solveTowers(n – 1, C, B, A)**

The Towers of Hanoi (4 of 5)

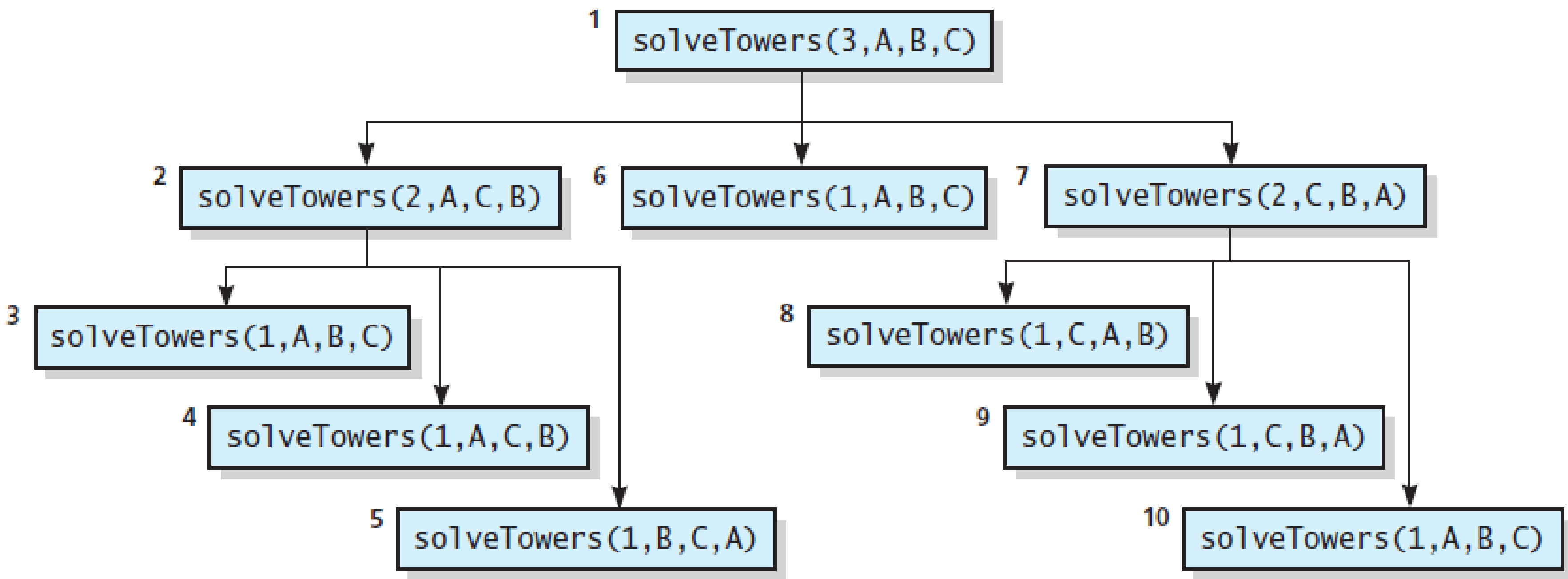
```
void solveTowers(int count,
                 char source, char destination, char spare) {

  if (count == 1)
   {
      std::cout << "Move top disk from pole " << source
           << " to pole " << destination << std::endl;
   }
   else
   {
      solveTowers(count - 1, source, spare, destination); // X
      solveTowers(1, source, destination, spare);         // Y
      solveTowers(count - 1, spare, destination, source); // Z
   }  // end if
}  // end solveTowers
```

The Towers of Hanoi (5 of 5)

The order of recursive calls that results from:

solveTowers(3, A, B, C)



Divide-and-Conquer

- Algorithms are ***recursive*** in structure typically follow the ***divide-and-conquer*** approach.
- Divide-and-Conquer steps:
  1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
  2. **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
  3. **Combine** the solutions to the sub-problems into the solution for the original problem.

Merge Sort Algorithm

MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$.
3. "***Merge***" the 2 sorted lists.

- The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.
  - **Divide:** Divide the *n*-element sequence to be sorted into two sub-sequences of *n*/2 elements each.
  - **Conquer:** Sort the two sub-sequences recursively using merge sort.
  - **Combine:** Merge the two sorted sub-sequences to produce the sorted answer.

Recursion and Efficiency

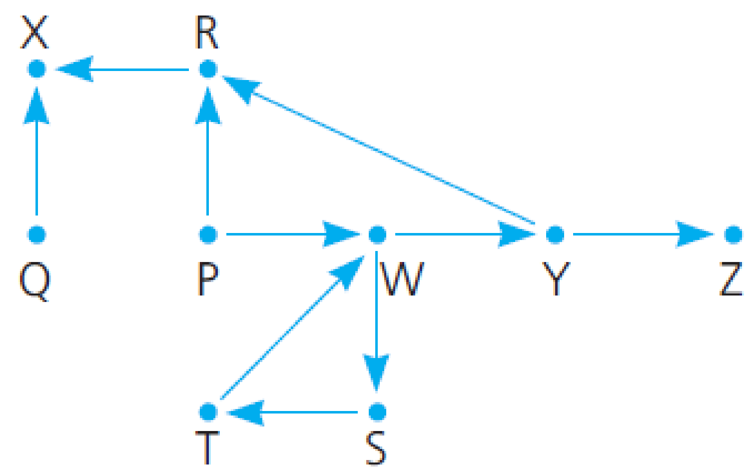
- Factors that contribute to inefficiency
  - Overhead associated with function calls
  - Some recursive algorithms inherently inefficient
- Keep in mind
  - Recursion can clarify complex solutions … but …
  - Clear, efficient iterative solution may be better

Backtracking

- Strategy for guessing at a solution and …
  - Backing up when a dead end is reached
  - Retracing steps in reverse order
  - Trying a new sequence of steps
- Combine recursion and backtracking to solve problems

Searching for an Airline Route (1 of 6)

- Must find a path from some point of origin to some destination point
- Program to process customer requests to fly
  - From some origin city
  - To some destination city
- Inputs to the problem:
  - names of cities served
  - Pairs of city names representing flight origins and destinations
  - Pair of city names for the requested origin, destination.

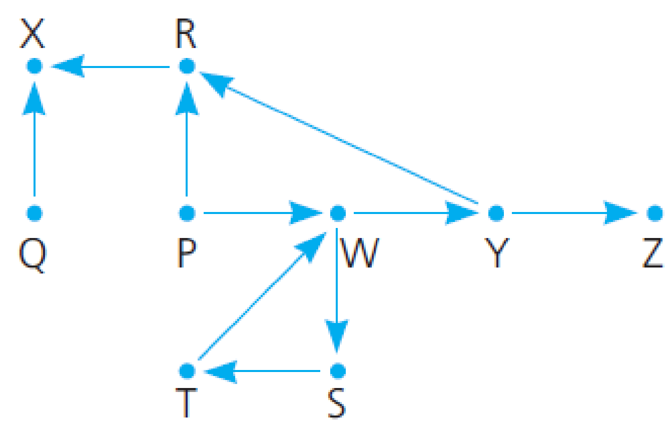


Searching for an Airline Route (2 of 6)

A recursive search strategy

To fly from the origin to the destination:

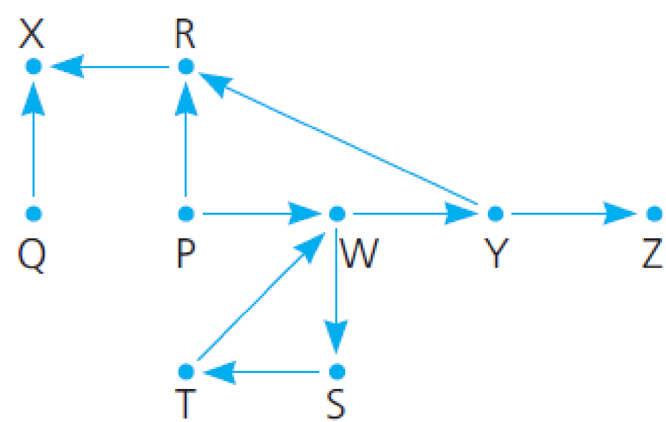
> *Select a city* `C` *adjacent to the origin*
> *Fly from the origin to city* `C`
> **`if`** (`C` *is the destination city*)
>   *Terminate— the destination is reached*
> **`else`**
>   *Fly from city* `C` *to the destination*



Searching for an Airline Route (3 of 6)

- Possible outcomes of exhaustive search strategy
  1. Reach the destination city
  2. Reach a city from which no departing flights
  3. You go around in circles

- Use backtracking to recover from a wrong choice (2 or 3)



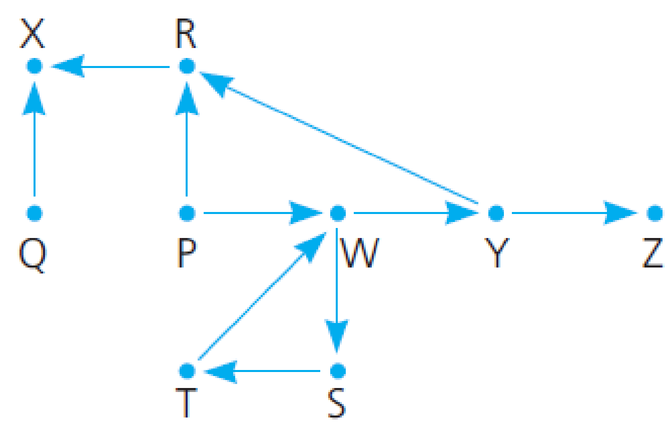
Searching for an Airline Route (4 of 6)

Refinement of the recursive search algorithm:

```
// Discovers whether a sequence of flights from originCity to destinationCity exists.
searchR(originCity: City, destinationCity: City): boolean

   Mark originCity as visited
   if (originCity is destinationCity)
      Terminate—the destination is reached
   else
      for (each unvisited city C adjacent to originCity)
         searchR(C, destinationCity)
```

Note: backtracking occurs by combining iteration with recursive calls.



Searching for an Airline Route (5 of 6)

ADT flight map operations:

```
// Reads flight information into the flight map.
 +readFlightMap(cityFileName: string, flightFileName: string): void
// Displays flight information.
 +displayFlightMap(): void
// Displays the names of all cities that HPAir serves.
 +displayAllCities(): void
// Displays all cities that are adjacent to a given city.
 +displayAdjacentCities(aCity: City): void
// Marks a city as visited.
 +markVisited(aCity: City): void
// Clears marks on all cities.
 +unvisitAll(): void
// Sees whether a city was visited.
 +isVisited(aCity: City): boolean
// Inserts a city adjacent to another city in a flight map.
 +insertAdjacent(aCity: City, adjCity: City): void
// Returns the next unvisited city, if any, that is adjacent to a given city.
// Returns a sentinel value, NO_CITY, if no unvisited adjacent city was found.
 +getNextCity(fromCity: City): City
// Tests whether a sequence of flights exists between two cities.
 +isPath(originCity: City, destinationCity: City): boolean
```

Searching for an Airline Route (6 of 6)

C++ implementation of **isPath**

```
bool Map::isPath(City originCity, City destinationCity) {
 bool result, done;
 markVisited(originCity); // Mark the current city as visited
 if (originCity == destinationCity) // Base case: the destination is reached
    result = true;
 else // Try a flight to each unvisited city
 {
  done = false;
  City nextCity = getNextCity(originCity); // returns next unvisited city
  while (!done && (nextCity != NO_CITY)) {
    done = isPath(nextCity, destinationCity);
    if (!done)
       nextCity = getNextCity(originCity);
   } // end while
  result = done;
 } // end if
 return result;
} // end isPath
```

But, where is the path?!

Recursion and Mathematical Induction

- Recursion solves a problem by
  - Specifying a solution to one or more base cases
  - Then demonstrating how to derive solution to problem of arbitrary size from solutions to smaller problems of same type.
- We can use induction to prove that recursive algorithm either
  - is correct or
  - performs certain amount of work

Correctness of Recursive Factorial (1 of 2)

Pseudocode describes recursive function that computes factorial:

```
fact(n: integer): integer

   if (n is 0)
      return 1
   else
      return n * fact(n - 1)
```

- **Basis.** *Show that the property is true for n = 0 .*
- **Inductive hypothesis.** *Assume that the property is true for n = k.* That is, assume that:

$$fact(k) = k! = k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

- **Inductive conclusion.** *Show that the property is true for n = k + 1.* That is, you must show that

$$fact(k+1) = (k+1) \times k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

Correctness of Recursive Factorial (2 of 2)

- By definition of the function `fact` , *`fact(k+1)`* returns the value

  $$(k+1) \times fact(k)$$

- But by the inductive hypothesis, *`fact(k)`* returns the value

  $$k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

- Thus, *`fact(k+1)`* returns the value

  $$(k+1) \times k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

which is what you needed to show to establish that:

property is true for an arbitrary k → property is true for k+1

- The inductive proof is thus complete.

The Cost of Towers of Hanoi (1 of 3)

- Pseudocode to solution to the Towers of Hanoi problem

```
solveTowers(count, source, destination, spare)
{
   if (count is 1)
      Move a disk directly from source to destination
   else
   {
      solveTowers(count - 1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count - 1, spare, destination, source)
   }
}
```

- Consider: given *N* disks …
  – How many moves does **solveTowers** make?

The Cost of Towers of Hanoi (2 of 3)

- Claim:

$$moves(N) = 2^N - 1 \quad \text{for all } N \geq 1$$

- Basis:
  - Show that the property is true for $N = 1$
- Inductive hypothesis:
  - Assume that property is true for $N = k$
- Inductive conclusion
  - Show that property is true for $N = k + 1$

The Cost of Towers of Hanoi (3 of 3)

```
solveTowers(count, source, destination, spare)
{
   if (count is 1)
      Move a disk directly from source to destination
   else
   {
      solveTowers(count - 1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count - 1, spare, destination, source)
   }
}
```

- moves(1) = 1
- moves(k+1) = 2×moves(k) + 1 = $2 \times 2^k - 2 + 1 = 2^{k+1} - 1$
- The inductive proof is thus complete.