Introduction to Java (cs2514)

Lecture 14: Recursion

M. R. C. van Dongen

March 3, 2017

Outline

- We study recursion:
  - Methods that call themselves;
  - Definitions that are defined in terms of themselves.
- We start with some easy/recreative applications:
  - We study a recursive method for computing factorials.
  - We study the recursive breeding habits of rabits.
  - We study the famous towers of hanoi.
- We end with a practical real-world application:
  - Binary search.

Recursion: Merriam Says

Function: noun

Etymology: Late Latin recursion-, recursio, from recurrere

Date: 1616

1. return
2. the determination of a succession of elements (as numbers or functions) by operation on one or more preceding elements according to a rule or formula involving a finite number of steps
3. a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first

Recursion

- Many concepts in computer science and mathematics are defined or computed *recursively*, i.e. using *recursion*.
- The idea is to define a complicated concept in terms of itself.

Recursion: Base Case

Base Case: Simple computation.

- We don't have to call the method itself.

Recursive Computation: Complicated computation involving:

- Simple computations.
- Lower order computation(s).

Recursion: Recursive Computation

Base Case: Simple computation.

- We don't have to call the method itself.

Recursive Computation: Complicated computation involving:

- Simple computations.
- Lower order computation(s).

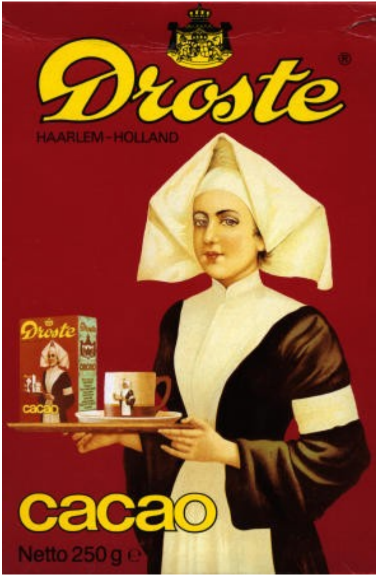
Recursive Algorithm: Dictionary Search

Dictionary Contains all Possible Words: One Word per Page

To *search* for the word given n pages do the following:

- If there's only one page ($n = 1$): We've found the word.
- Otherwise ($n > 1$):
    - Find the page in the "middle."
    - Read the word on the middle page.
    - If that word is our word: We've found the word.
    - If our word is smaller: *search* to the left.
    - Otherwise: *search* to the right.

Recursive Picture



Recursive Picture

Pitfalls

- Recursive computations involve themselves.
- If we're not careful we may get an infinite chain of computations.
- For example, we may be
  - Computing what's on Box 1 with Box 2 on it, which involves
  - Computing what's on Box 2 with Box 3 on it, which involves
  - Computing what's on Box 3 with Box 4 on it, which involves
  - ....
- Each recursive computation should eventually terminate.
- This only happens if they all reach some *base case* condition.
  - (The base conditions may be different.)

Controlling the Size

Guaranteeing Termination

- Each computation should have a *"size:"*
  - A non-negative integer should do.
- The size should depend on one or several method parameters.
- Base-case computations have small fixed sizes.
- Recursive sub-computations should get smaller and smaller.
- Using induction we can prove termination.

How does this Work?

Dictionary Search Revisited

- Let's call the top computation C_0.
- Let C_1 be the recursive computation of C_0,
- Let C_2 be the recursive computation of C_1, and so on.
- Finally, let S_i be the size of C_i.
  - By nature of the algorithm we have $S_i > S_{i+1}$.
- Let's assume an infinite chain of computations

$$C_0 \, , C_1 \, , C_2 \, , \ldots \; .$$

- Then we have an infinite chain of *integers*

$$S_0 > S_1 > S_2 > \cdots \; .$$

- But this is impossible since $S_i \geq 0$, for all i.

Computing Factorials

- Let n be a positive integer.
- The *factorial* of n, denoted $n!$, is defined as follows:

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n\,.$$

- Using the product notation we may write this as follows:

$$n! = \prod_{i=1}^{n} i\,.$$

Computing Factorials: An Iterative Solution

Java

```java
public static int factorial( int n ) {
    int product = 1;
    for (int i = 1; i <= n; i ++) {
        product = product * i;
    }
    return product;
}
```

Computing Factorials: A Recursive Solution

Base Case: Clearly $1! = 1$.

Recursion: The recursion may be found by noticing that

$$\prod_{i=1}^{n} i = n \times \prod_{i=1}^{n-1} i\,.$$

This gives us

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

Computing Factorials: A Recursive Solution

Base Case: Clearly $1! = 1$.

Recursion: The recursion may be found by noticing that

$$\prod_{i=1}^{n} i = n \times \prod_{i=1}^{n-1} i \,.$$

This gives us

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

Combining the Base Case and Recursive Case

$$n! = \begin{cases} 1 & \text{if } n = 1\,; \\ (n-1)! \times n & \text{if } n > 1\,. \end{cases}$$

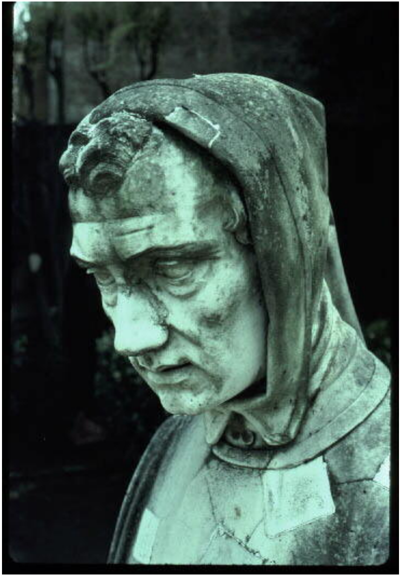
Java

```java
public static int factorial( int n ) {
    final int result;

    if (n == 1) {
        result = 1;                            // Base Case
    } else {
        result = factorial( n - 1 ) * n; // Recursion
    }

    return result;
}
```

Simulating a Computation

```
factorial( 4 ) = ( factorial( 3 ) * 4 ) )
               = ( ( factorial( 2 ) * 3 ) * 4 )
               = ( ( ( factorial( 1 ) * 2 ) * 3 ) * 4 )
               = ( ( ( 1 * 2 ) * 3 ) * 4 )
               = ( ( 2 * 3 ) * 4 )
               = ( 6 * 4 )
               = 24.
```

Yer Man



Facts about Fibonacci

- Born: about 1175 AD.
- Died: 1250 AD.
- Famous mathematician.
- Introduced the Decimal System into Europe.
- Well known for many of his problems.

Rabbits

A pair of [baby rabbits] are put in a field and, if rabbits take a month to become mature and then produce a new pair every month after that, how many pairs will there be in twelve months time?

Rabbits do not Escape and Don't Die

A pair of [baby rabbits] are put in a field and, if rabbits take a month to become mature and then produce a new pair every month after that, how many pairs will there be in twelve months time?

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1

Introduction to Java

M. R. C. van Dongen

Outline

Recursion

Factorial Computation

Fibonacci Numbers

A Fibonacci Problem

Fibonacci's Solution

The Fibonacci Sequence

Tracing the Calls

Towers of Hanoi

Binary Search

Question Time

For Monday

About this Document

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1			

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0		

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2			

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1		

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3			

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1		

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4			

Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2		

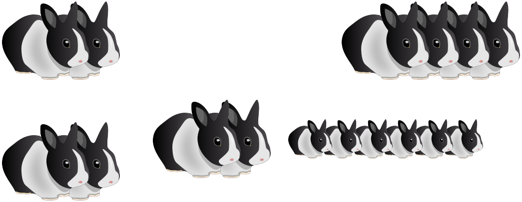
Fibonacci's Solution

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	

Fibonacci's Solution

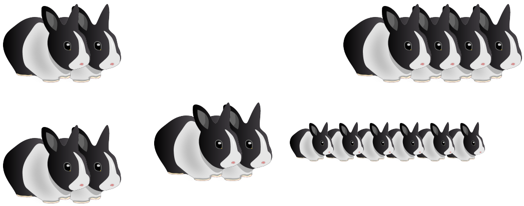
Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5

Fibonacci's Solution



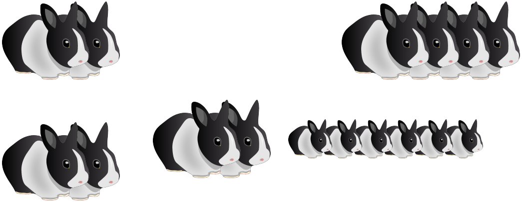
Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5			

Fibonacci's Solution



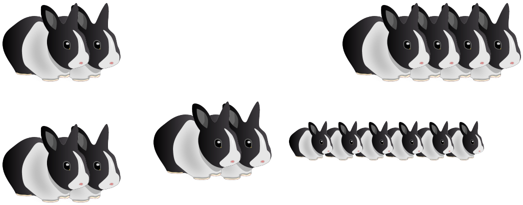
Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3		

Fibonacci's Solution



Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	

Fibonacci's Solution



Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8

The Fibonacci Sequence

- Fibonacci's solution involves the series of numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots .$$

- Given the first two we can compute the remaining numbers:

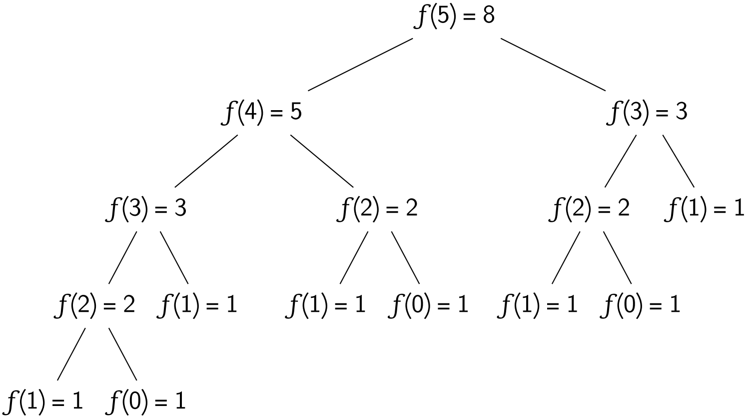
$$F_n = \begin{cases} 1 & \text{if } n = 0\,; \\ 1 & \text{if } n = 1\,; \\ F_{n-1} + F_{n-2} & \text{if } n > 1\,. \end{cases}$$

Back to Java

Java

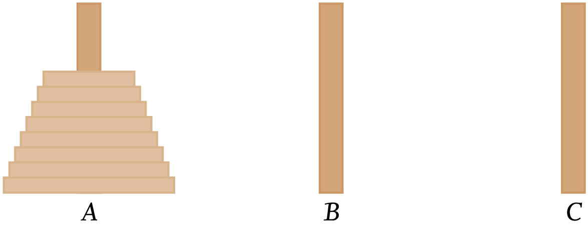
```
public static int fibonacci( int n ) {
    final int result;

    if (n <= 1) { /* Base Case */
        result = 1;
    } else {      /* Recursion */
        result = fibonacci( n - 1 ) + fibonacci( n - 2 );
    }

    return result;
}
```

Tracing the Calls

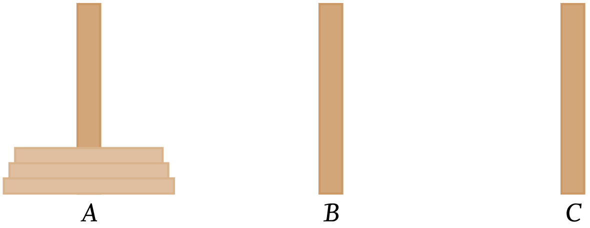


The Towers of Hanoi

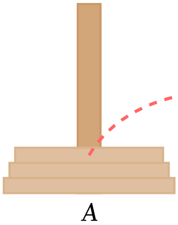
- We're given a tower of 8 disks and three pegs: A, B, and C.
- Each disk has a hole in the centre.
- Initially, the disks are stacked in decreasing size on Peg A.
- The objective is to transfer the stack to a different peg, but
  - We're only allowed to stack disks on pegs,
  - We're only allowed to move one disk at a time, and
  - We can only stack a smaller disk on top of a larger disk.



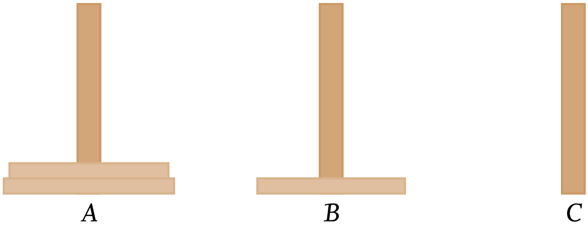
Simulation



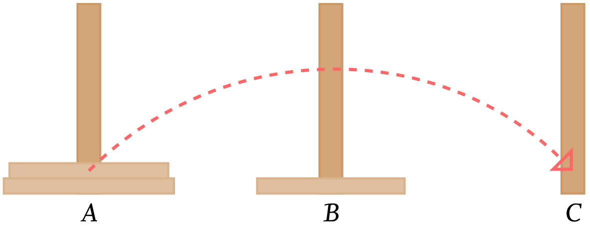
Simulation



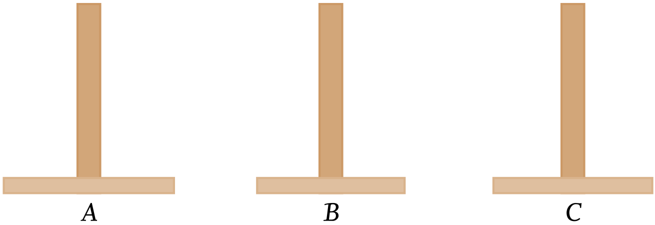
Simulation



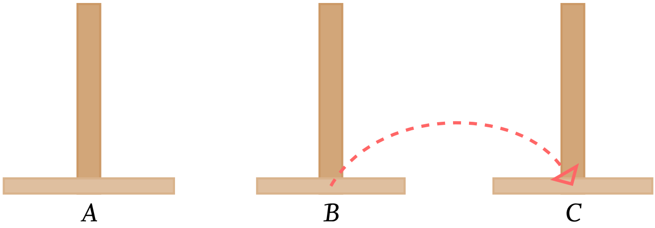
Simulation



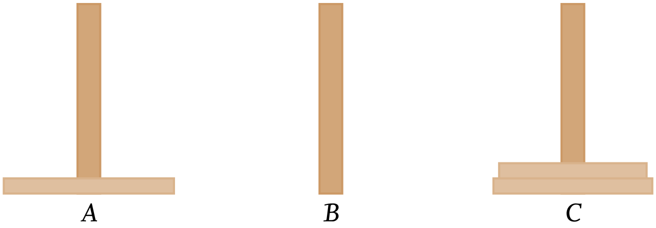
Simulation



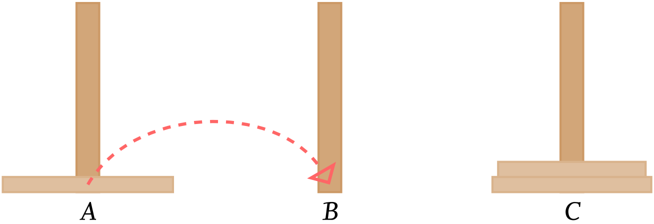
Simulation



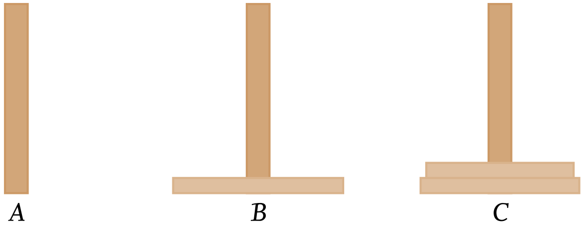
Simulation



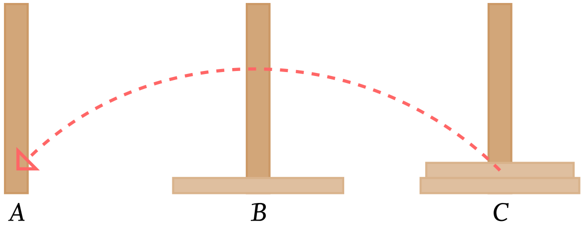
Simulation



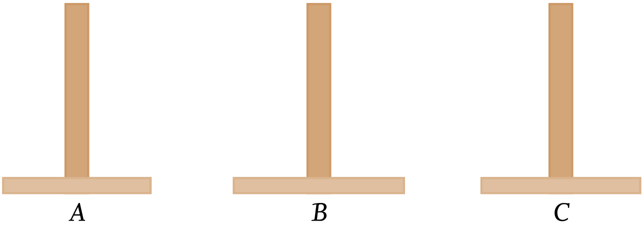
Simulation



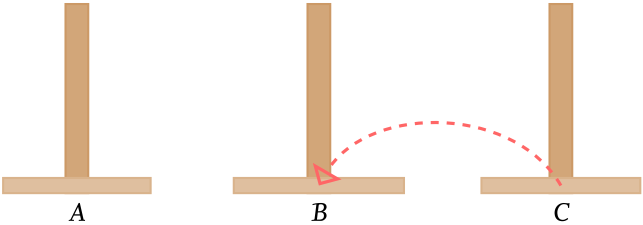
Simulation



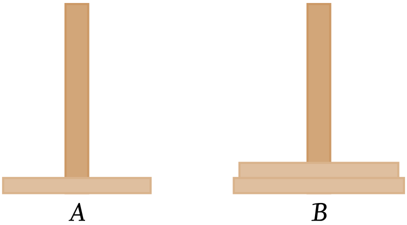
Simulation



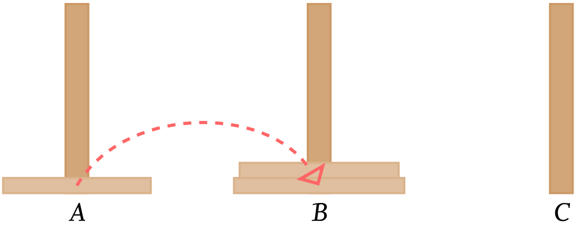
Simulation



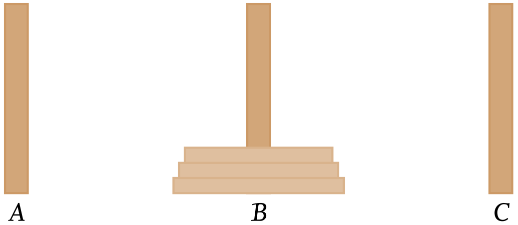
Simulation



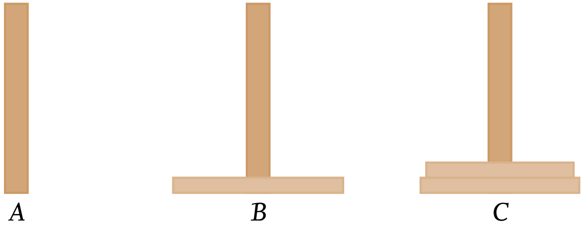
Simulation



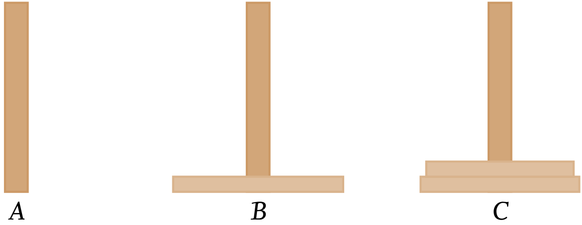
Simulation



Intermediate State of the 3-Disk Version

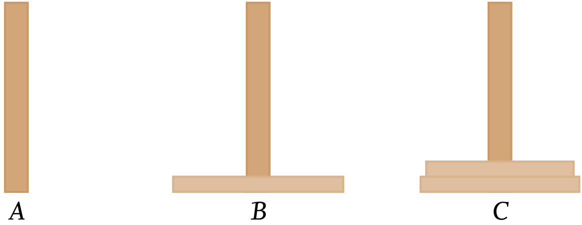


Intermediate State of the 3-Disk Version



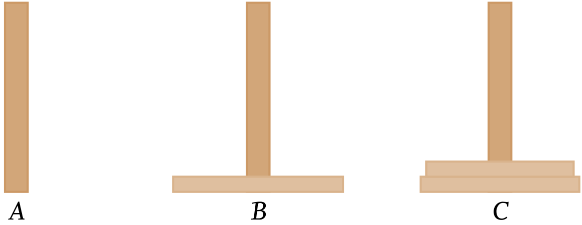
- Here we *recursively* moved disks from C to B and were done!

Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

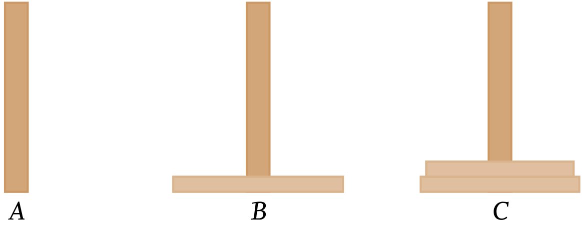
Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

1. Start at initial state.

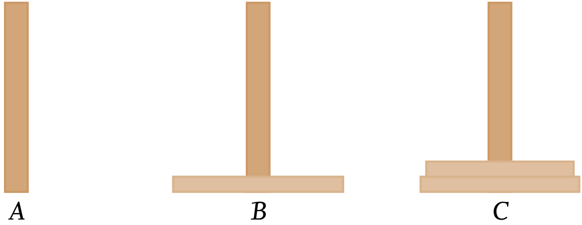
Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

1. Start at initial state.
2. Solve the sub-problem to arrive at the intermediate state.

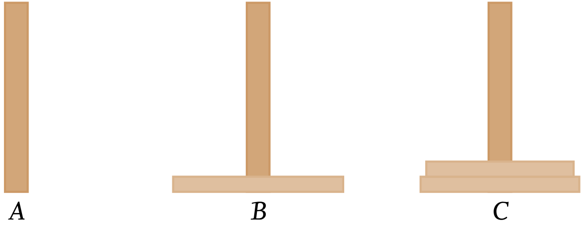
Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

1. Start at initial state.
2. Solve the sub-problem to arrive at the intermediate state.
3. Use recursion to go from the intermediate to the target state.

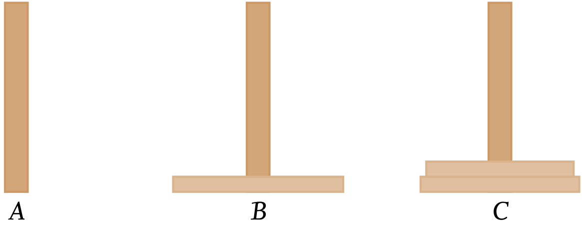
Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

  1. Start at initial state.
  2. Solve the sub-problem to arrive at the intermediate state.
  3. Use recursion to go from the intermediate to the target state.
- So, how did we get at the intermediate state?

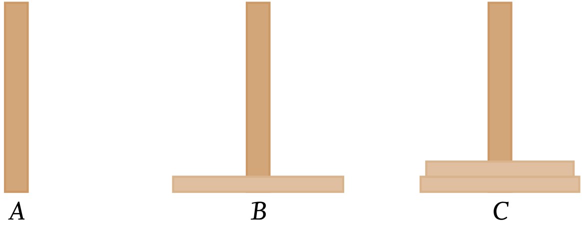
Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

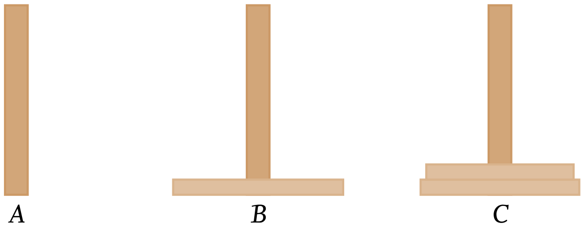
  1. Start at initial state.
  2. Solve the sub-problem to arrive at the intermediate state.
  3. Use recursion to go from the intermediate to the target state.
- So, how did we get at the intermediate state?
  1. We started with all disks stacked on Peg A.
  2. We moved all disks except for the largest one from A to C.

Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:
  1. Start at initial state.
  2. Solve the sub-problem to arrive at the intermediate state.
  3. Use recursion to go from the intermediate to the target state.
- So, how did we get at the intermediate state?
  1. We started with all disks stacked on Peg A.
  2. We moved all disks except for the largest one from A to C.
  3. We moved the largest disk to Peg B.

Intermediate State of the 3-Disk Version



- Here we *recursively* moved disks from C to B and were done!
- So, how did we arrive at the intermediate state?
- If we can solve this subproblem, we can solve the whole problem:

  1. Start at initial state.
  2. Solve the sub-problem to arrive at the intermediate state.
  3. Use recursion to go from the intermediate to the target state.
- So, how did we get at the intermediate state?
  1. We started with all disks stacked on Peg A.
  2. We moved all disks except for the largest one from A to C.
     - But this is just the 2-disk version: move 2 disks from A to C.
  3. We moved the largest disk to Peg B.

Designing the Algorithm

Base case: If $n = 1$:

1. Move disk n to target peg.

Recursion: If $n > 1$:

1. Move disks $1, \ldots, n-1$ to intermediate peg.
2. Move disk n to target peg.
3. Move disks $1, \ldots, n-1$ to target peg.

Designing the Algorithm

Base case: If $n = 1$:

1. Move disk n to target peg.

Recursion: If $n > 1$:

1. Move disks $1, \ldots, n-1$ to intermediate peg.
2. Move disk n to target peg.
3. Move disks $1, \ldots, n-1$ to target peg.

Alternative solution

- If $n \geq 1$ then
  1. Move disks 1, ..., $n - 1$ from source to intermediate peg.
  2. Move disk n to target peg.
  3. Move disks 1, ..., $n - 1$ from intermediate to target peg.

Back to Java

Java

```
/**
 * @param n Number of disks.
 * @param source The source peg: should be 0, 1, or 2.
 * @param target The target peg: should be 0, 1, or 2.
 * <PAR> {@code source} and {@code target} should be different.</PAR>
 */
private static
void hanoi( final int n, final int source, final int target ) {
    if (n >= 1) {
        // Compute the number of the intermediate peg:
        final int intermediate = 3 - source - target;
        hanoi( n - 1, source, intermediate );
        moveDisk( n, source, target );
        hanoi( n - 1, intermediate, target );
    }
}

public static
void final hanoi( int n ) {
    // move n disks from Peg 0 to Peg 1.
    hanoi( n, 0, 1 );
}
```

Back to Java

Java

```
/**
 * @param n Number of disks.
 * @param source The source peg: should be 0, 1, or 2.
 * @param target The target peg: should be 0, 1, or 2.
 * <PAR> {@code source} and {@code target} should be different.</PAR>
 */
private static
void hanoi( final int n, final int source, final int target ) {
    if (n >= 1) {
        // Compute the number of the intermediate peg:
        final int intermediate = 3 - source - target;
        hanoi( n - 1, source, intermediate );
        moveDisk( n, source, target );
        hanoi( n - 1, intermediate, target );
    }
}

public static
void final hanoi( int n ) {
    // move n disks from Peg 0 to Peg 1.
    hanoi( n, 0, 1 );
}
```

Back to Java

Java

```
/**
 * @param n Number of disks.
 * @param source The source peg: should be 0, 1, or 2.
 * @param target The target peg: should be 0, 1, or 2.
 * <PAR> {@code source} and {@code target} should be different.</PAR>
 */
private static
void hanoi( final int n, final int source, final int target ) {
    if (n >= 1) {
        // Compute the number of the intermediate peg:
        final int intermediate = 3 - source - target;
        hanoi( n - 1, source, intermediate );
        moveDisk( n, source, target );
        hanoi( n - 1, intermediate, target );
    }
}

public static
void final hanoi( int n ) {
    // move n disks from Peg 0 to Peg 1.
    hanoi( n, 0, 1 );
}
```

Back to Java

Java

```
private static void moveDisk( final int disk, final int source, final int target ) {
    final String pegNames[] = { "A", "B", "C" };
    System.out.println( "Move disk " + disk
                        + " from " + pegNames[ source ]
                        + " to " + pegNames[ target ] );
}
```

Binary Search

- *Binary search* is an algorithm that:
  - Determines whether a given item is in a sorted list, and
  - If it is, returns the position of that element in the list.
- It works like the "dictionary search" algorithm.
- It repeatedly halves the number of elements.
  - It is a typical case of a *divide and conquer* algorithm.
  - Because of the halving it is sometimes called *dichotomic*.
- Requires (worst-case) time that is logarithmic in size of the input.

The Basic Idea

- Before studying the algorithm let's define its main task.

  Input: The input of the algorithm consists of:
  - An item; and
  - A list of items sorted in non-decreasing order.
  - For simplicity the items in list are unique.

  Output: The output of the algorithm is an `int`.

  The output depends on one of the following cases.

  Item is in list: The index of item in the list.

  Item is not in list: A negative number.
- For simplicity we'll assume that all items are `int`s.
- Furthermore, we'll assume that the list is presented as an array.

The Algorithm

`binSearch( item, items, lo, hi )`

`lo > hi`: Return `-1`.

`lo <= hi`:

1. Determine "the" middle index.
   - We implement this as `mid = (lo + hi) / 2`.

2. Compare `item` and `items[ mid ]`.
   - **`item == items[ mid ]`:**
     - Return `mid`.
   - **`item < items[ mid ]`:**
     - Return `binSearch( item, items, lo, mid - 1 )`.
   - **`item > items[ mid ]`:**
     - Return `binSearch( item, items, mid + 1, hi )`.

The Algorithm

`binSearch( item, items, lo, hi )`

`lo > hi`: Return `-1`.

`lo <= hi`:

1. Determine "the" middle index.
   - We implement this as `mid = (lo + hi) / 2`.
   - Unfortunately, this is not correct due to overflow.
   - You can fix this by implementing it as
     - '`mid = lo + (hi - lo) / 2`' or as
     - '`mid = (hi + lo) >>> 1`'.
2. Compare `item` and `items[ mid ]`.
   - **`item == items[ mid ]`**:
     - Return `mid`.
   - **`item < items[ mid ]`**:
     - Return `binSearch( item, items, lo, mid - 1 )`.
   - **`item > items[ mid ]`**:
     - Return `binSearch( item, items, mid + 1, hi )`.

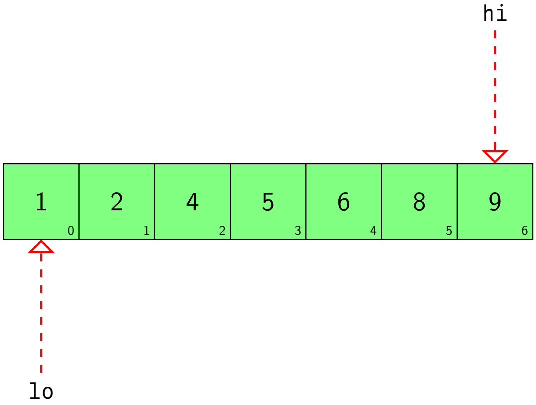
Implementation in Java

Java

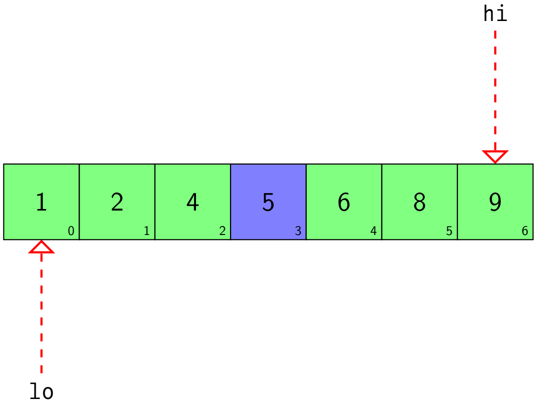
```
public static int binSearch( int item, int[] items ) {
    return binSearch( item, items, 0, items.length - 1 );
}

public static int binSearch( int item, int[] items, int lo, int hi ) {
    final int result;

    if (lo > hi) {
        result = - 1;
    } else {
       int mid = (lo + hi) / 2;
       if (item == items[ mid ]) {
           result = mid;
       } else if (item < items[ mid ]) {
           result = binSearch( item, items, lo, mid - 1 );
       } else {
           result = binSearch( item, items, mid + 1, hi );
       }
    }

    return result;
}
```

binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

Intial Situation

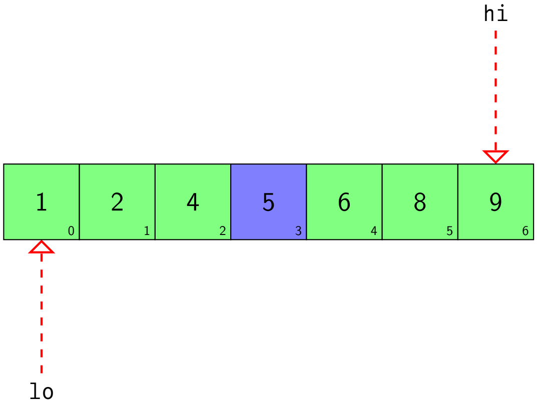


```
binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



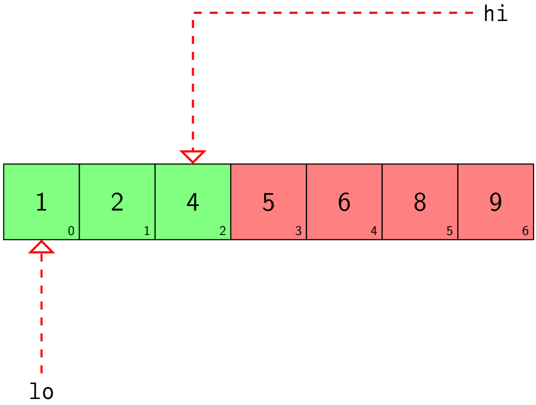
`binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )`

`item < item[ mid ]`



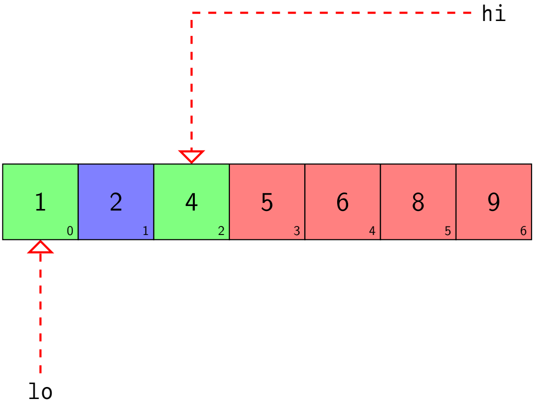
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

Search to Left of mid

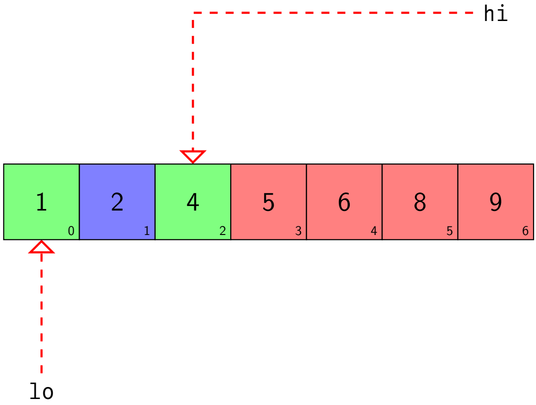


binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

`mid = (lo + hi) / 2`

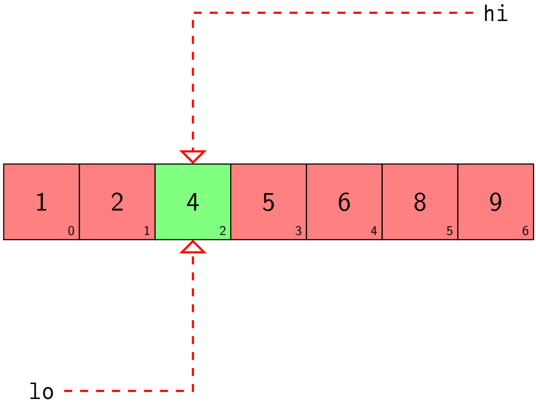


```
binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )
item > item[ mid ]
```



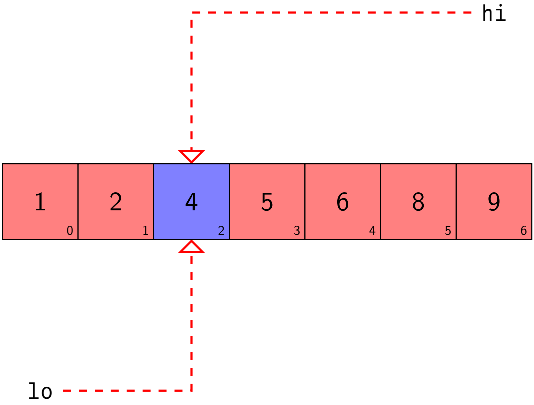
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

Search to Right of mid



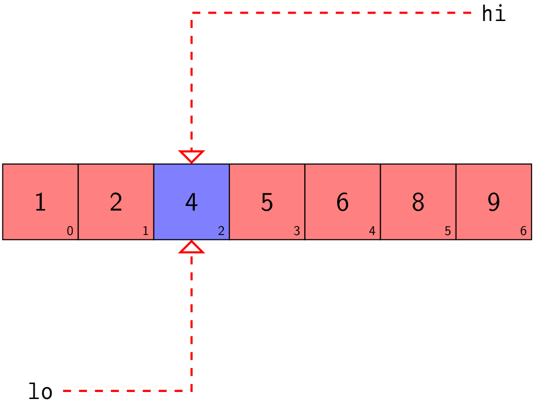
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

`mid = (lo + hi) / 2`



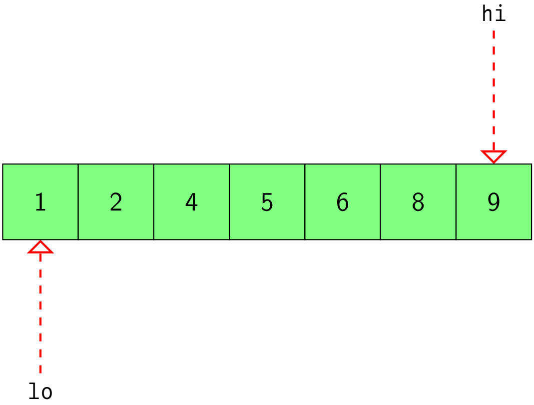
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

Celebration



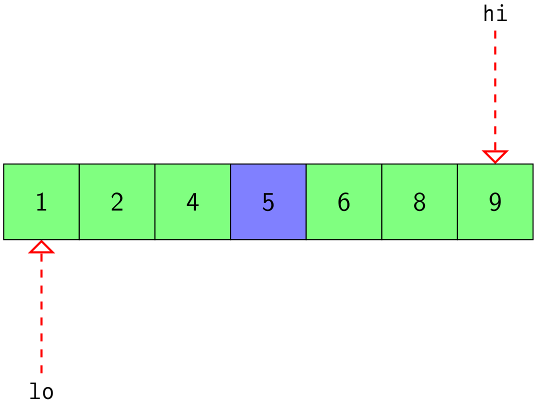
binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Intial Situation



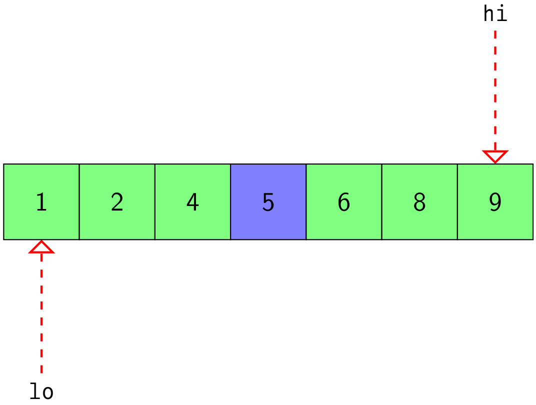
binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

`mid = (lo + hi) / 2`



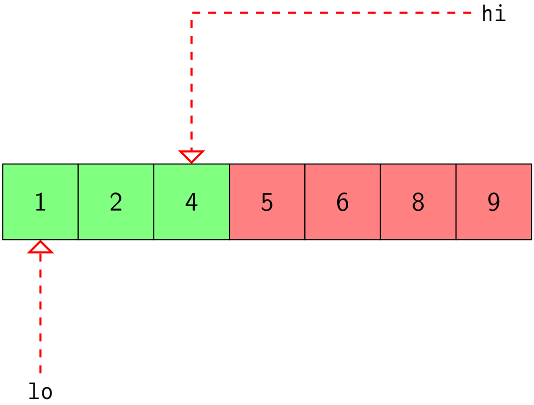
```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
```

```
item < item[ mid ]
```

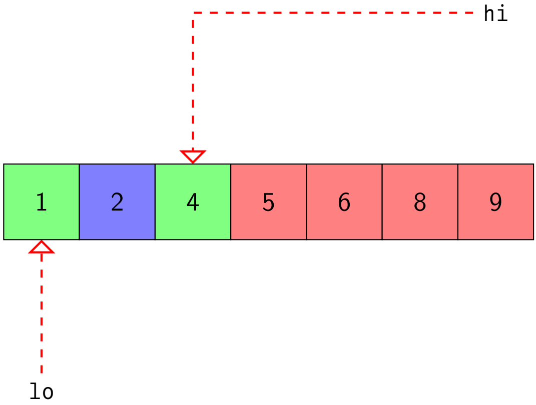


binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Search to Left of mid

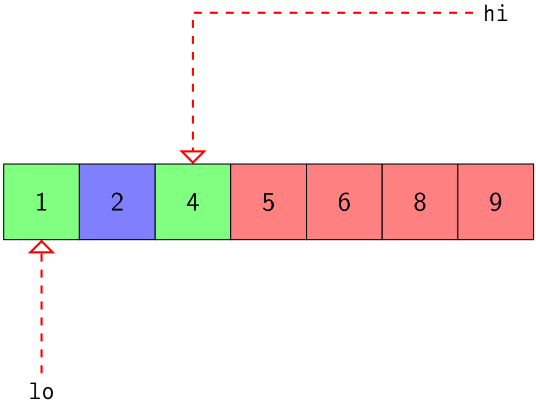


```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



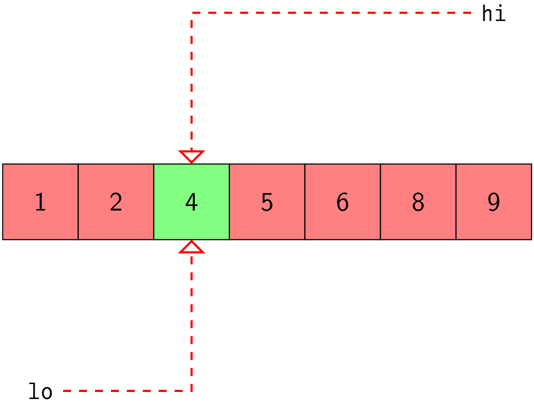
`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

`item > item[ mid ]`

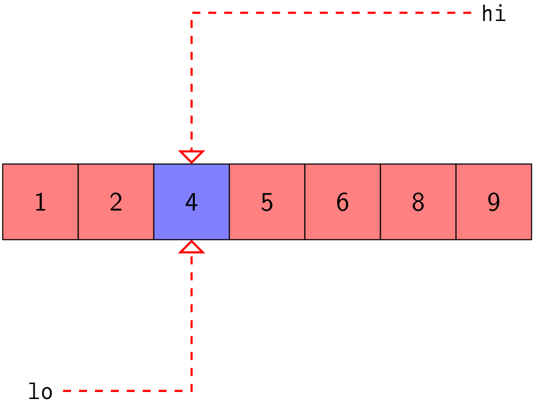


binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Search to Right of mid

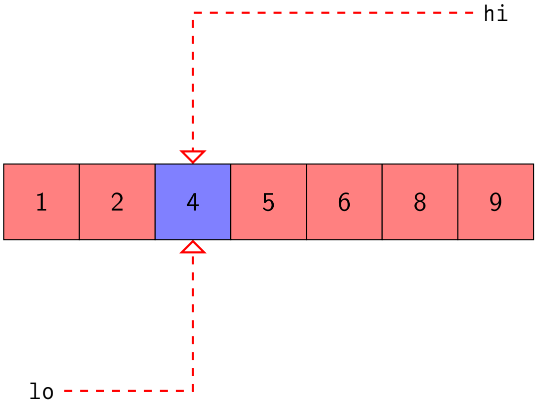


```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

`item < item[ mid ]`



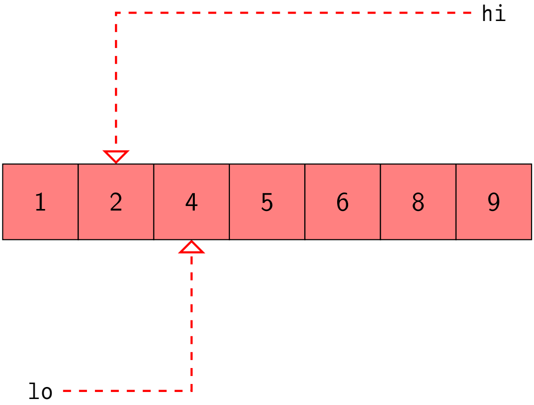
Introduction to Java

M. R. C. van Dongen

Outline
Recursion
Factorial Computation
Fibonacci Numbers
Towers of Hanoi
Binary Search
The Basic Idea
The Algorithm
Implementation in Java
Simulation
Comparable Interface
Question Time
For Monday
About this Document

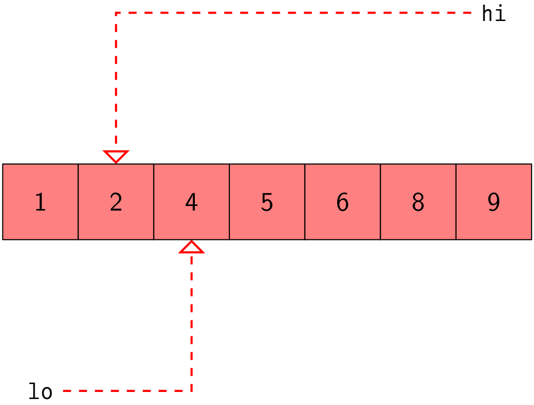
binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Search to Left of mid



binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Bummer



The `Comparable` Interface

- We've seen how to use binary search for `int`s.
- We should be able to generalise it for other *comparable* things.
- *Implementing* an *interface* is almost the same as extending a class.
  - If class B implements interface A, B behaves as A.
- A class *implements* the `Comparable` *interface* if it overrides

```
int compareTo( Object that )
```

- Many classes implement the `Comparable` interface:
  - `Integer`,
  - `Double`,
  - `String`,
  - ....

A Comparable-Compatible Version

Java

```
public static int binSearch( Comparable item, Comparable[] items, int lo, int hi ) {
    final int result;

    if (lo > hi) {
        result = - 1;
    } else {
       int mid = (lo + hi) / 2;
       int outcomeOfComparison = item.compareTo( items[ mid ] );
       if (outcomeOfComparison == 0) {
           result = mid;
       } else if (outcomeOfComparison < 0) {
           result = binSearch( item, items, lo, mid - 1 );
       } else {
           result = binSearch( item, items, mid + 1, hi );
       }
    }

    return result;
}
```

Question Time

Questions Anybody?

For Monday

- Study the presentation, and
- Implement the Towers of Hanoi from scratch.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.