Software Development (cs2500)

Lecture 28: Recursion

M. R. C. van Dongen

November 27, 2013

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 )`.

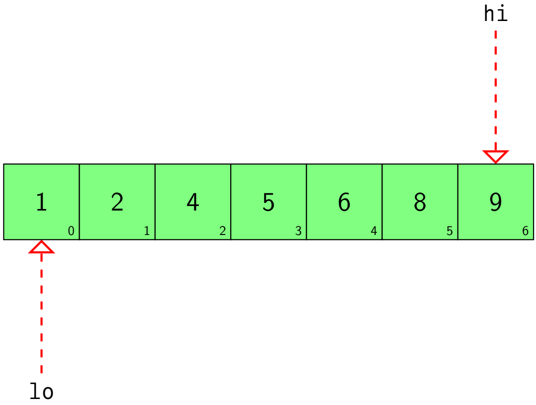
Implementation in Java

Java

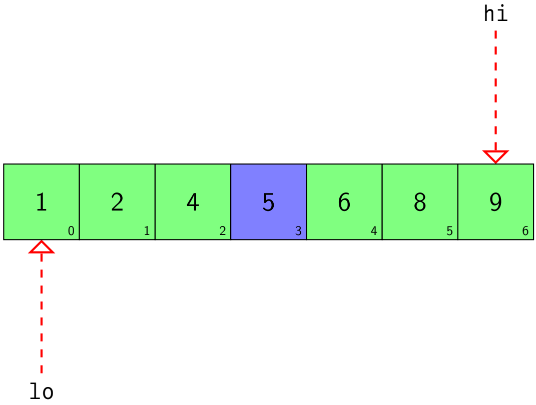
```java
public static int binSearch( int item, int[] items ) {
    return binSearch( item, items, 0, items.length - 1 );
}

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

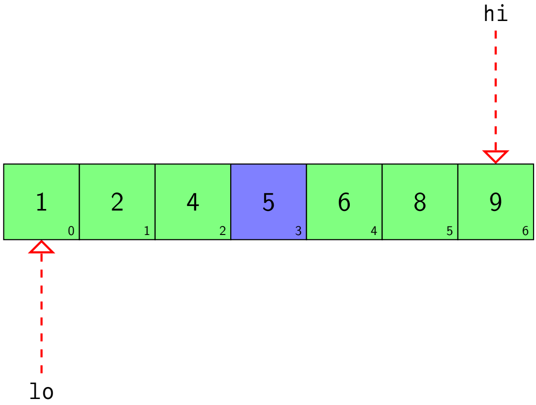


```
binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



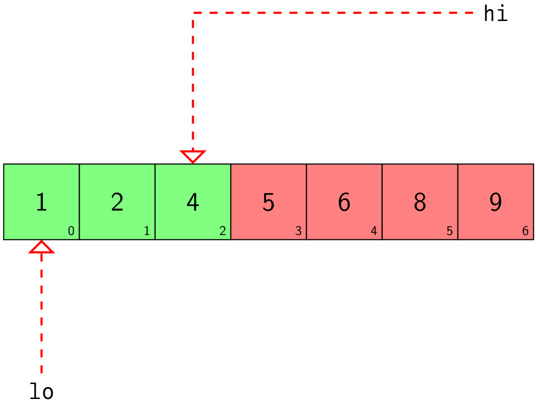
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

item < item[mid]



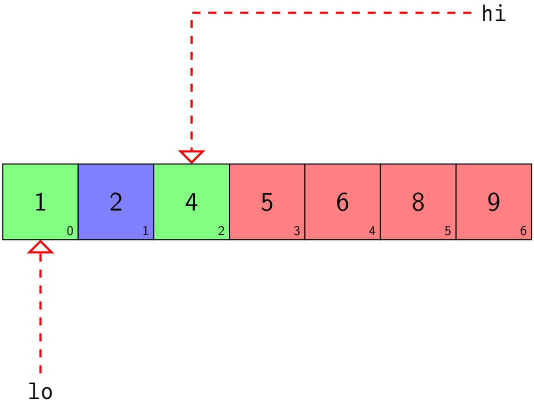
`binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )`

Search to Left of `mid`



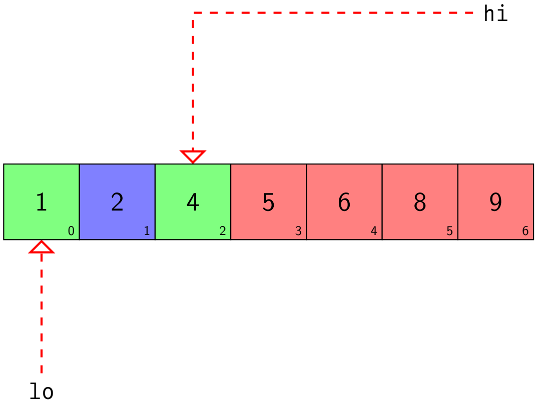
`binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )`

`mid = (lo + hi) / 2`



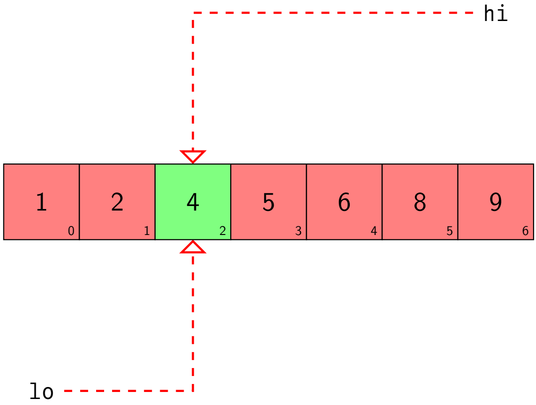
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

item > item[mid]

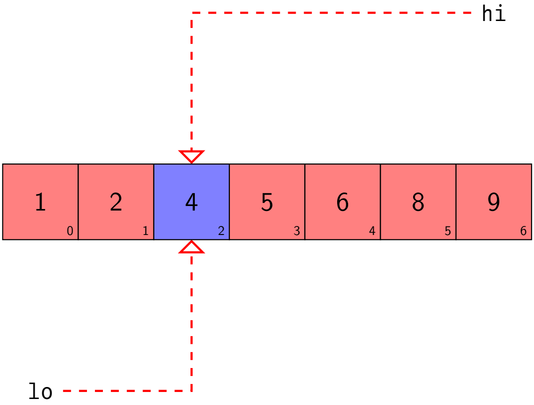


`binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )`

Search to Right of `mid`

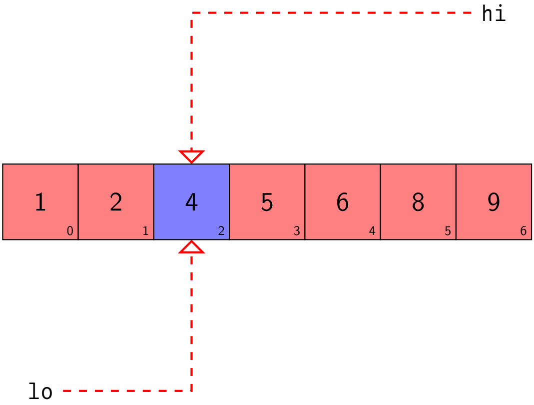


```
binSearch( 4, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



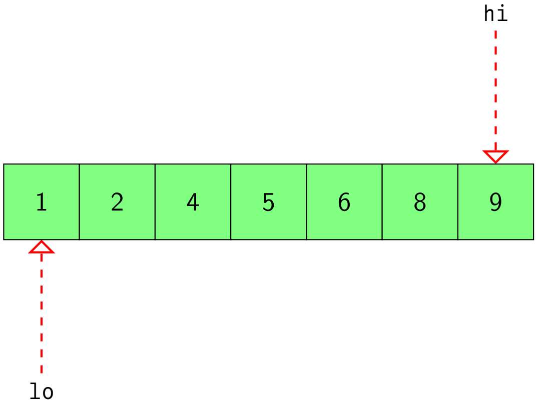
binSearch(4, {1,2,4,5,6,8,9}, 0, 6)

Celebration



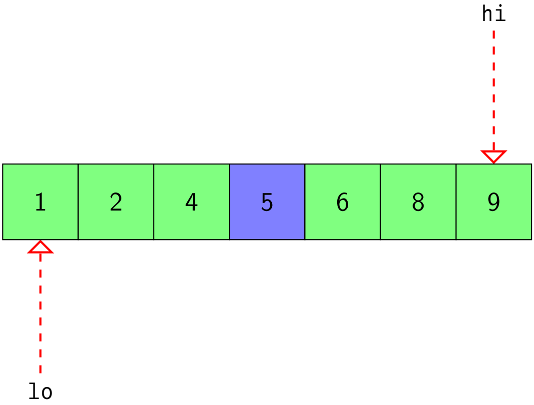
`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

Intial Situation



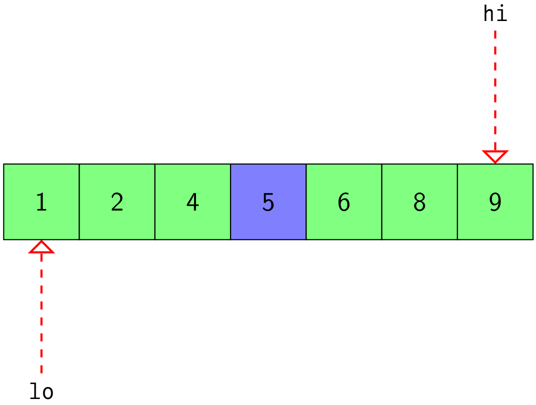
`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

`mid = (lo + hi) / 2`



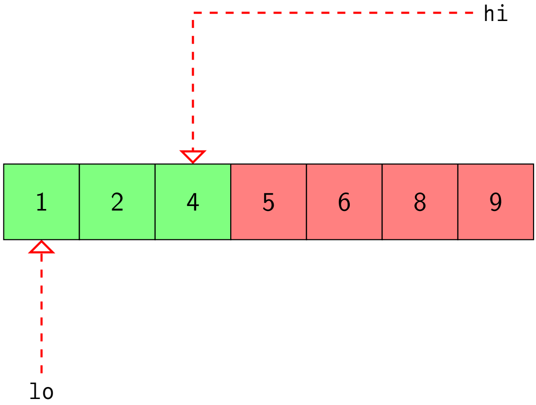
```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
```

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

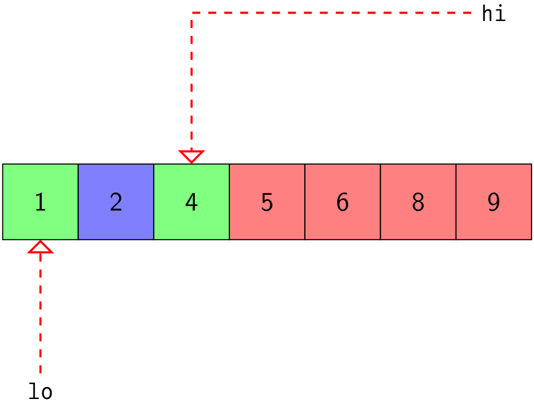


`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

Search to Left of `mid`

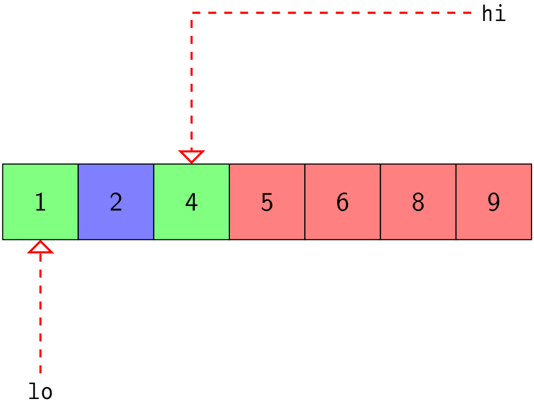


```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



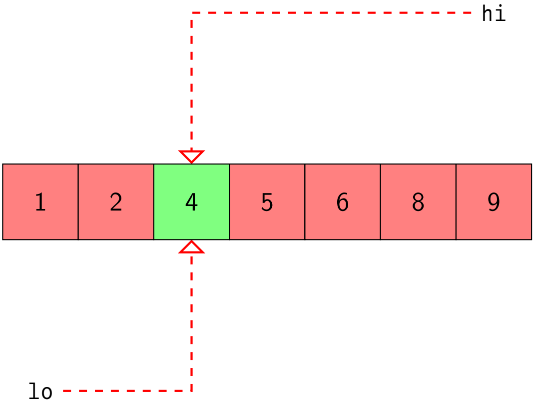
```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
```

`item > item[ mid ]`

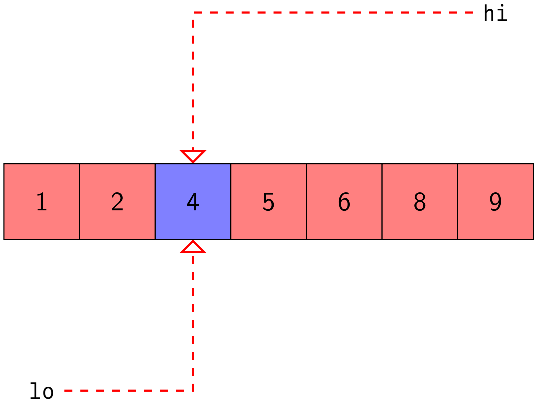


binSearch(3, {1,2,4,5,6,8,9}, 0, 6)

Search to Right of mid

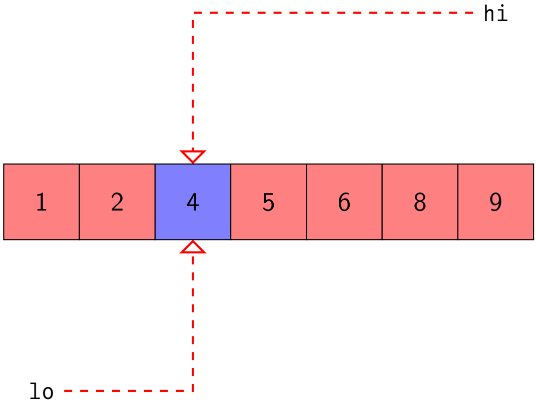


```
binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )
mid = (lo + hi) / 2
```



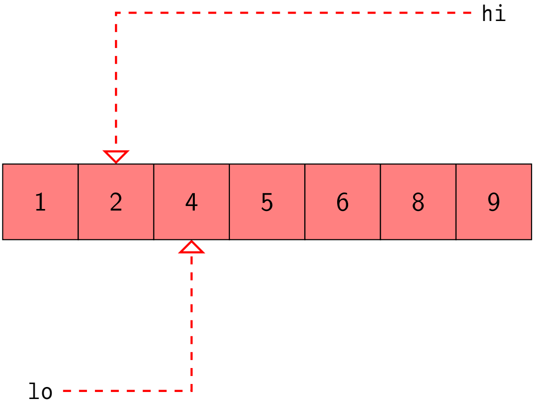
`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

`item < item[ mid ]`



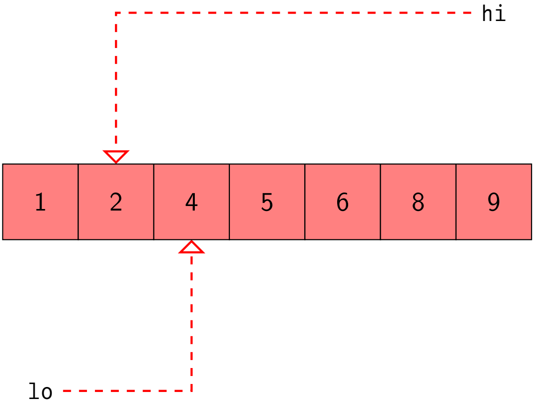
`binSearch( 3, {1,2,4,5,6,8,9}, 0, 6 )`

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

    return result;
}
```

The `Quicksort` Algorithm

- Sorting algorithms are a very important class of algorithms.
  - Sorting efficiently is crucial to many applications.
- `Quicksort` is a simple but efficient sorting algorithm.
- Given n *random* items its requires $O(n \log n)$ comparisons (on average).
- But, it requires $O(n^2)$ comparisons in the worst case.
- If the input is given as an array, we can sort the array in-situ.
- The algorithm was invented by C. A. R. Hoare in 1962.
- For simplicity we shall study the version for sorting `int` arrays.
- `Arrays` defines several `quicksort`-based sorting methods.

Main Idea

Base case: If $n \leq 1$ then the input is sorted.

Recursion: If $n > 1$:

1. Select any item from the input.
2. Partition remaining items into classes L and G.
   - L are the items less than or equal to the pivot.
   - G are the remaining items.
3. Members of L should end up before those of G.
4. Put the pivot between L and G.
5. Recursively sort L and G.

The Wrapper

Java

```java
public static void qsort( int[] items ) {
    qsort( items, 0, items.length - 1 );
}
```

Main Algorithm

Java

```java
// Sorts items[ lo .. hi ] in non-descending order.
private static void qsort( int[] items, int lo, int hi ) {
   if (hi - lo >= 1) {
      int pivotPosition = partition( items, lo, hi );
      qsort( items, lo, pivotPosition - 1 );
      qsort( items, pivotPosition + 1, hi );
   }
}
```

Main Algorithm: Any Sorting to Do?

Java

```
// Sorts items[ lo .. hi ] in non-descending order.
private static void qsort( int[] items, int lo, int hi ) {
   if (hi - lo >= 1) {
      int pivotPosition = partition( items, lo, hi );
      qsort( items, lo, pivotPosition - 1 );
      qsort( items, pivotPosition + 1, hi );
   }
}
```

Main Algorithm: Partition

Divide

Java

```java
// Sorts items[ lo .. hi ] in non-descending order.
private static void qsort( int[] items, int lo, int hi ) {
   if (hi - lo >= 1) {
      int pivotPosition = partition( items, lo, hi );
      qsort( items, lo, pivotPosition - 1 );
      qsort( items, pivotPosition + 1, hi );
   }
}
```

Main Algorithm: Sort Items to Left of Pivot

Divide and Conquer

Java

```java
// Sorts items[ lo .. hi ] in non-descending order.
private static void qsort( int[] items, int lo, int hi ) {
    if (hi - lo >= 1) {
        int pivotPosition = partition( items, lo, hi );
        qsort( items, lo, pivotPosition - 1 );
        qsort( items, pivotPosition + 1, hi );
    }
}
```

Main Algorithm: Sort Items to Right of Pivot

Divide and Conquer

Java

```
// Sorts items[ lo .. hi ] in non-descending order.
private static void qsort( int[] items, int lo, int hi ) {
    if (hi - lo >= 1) {
        int pivotPosition = partition( items, lo, hi );
        qsort( items, lo, pivotPosition - 1 );
        qsort( items, pivotPosition + 1, hi );
    }
}
```

Partition

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Pivot Selection and Exchange

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Partitioning of Lower Elements

Java

```java
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Move Item to Left?

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Move Item to Left? Exchange

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Move Item to Left? Adjust Destination

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Loop Invariant

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Consequence of Loop Invariant

Java

```java
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Move Pivot to Destination

Java

```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Final Invariant

Java

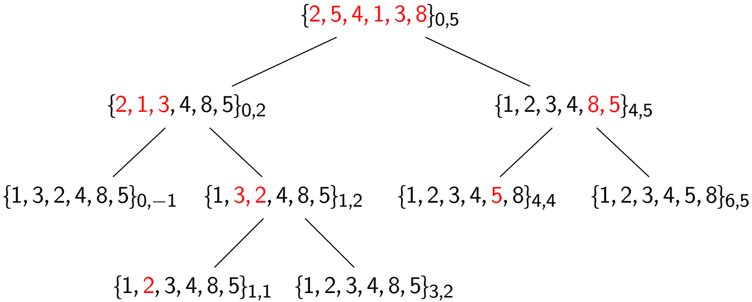
```
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Partition: Return Pivot Position

Java

```java
private static int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

Call Trace of `qsort( {2,5,4,1,3,8}, 0, 5 )`



Recursion and Tail Recursion

- Recursion:
  - Advantage:
    - Elegant, and easy to write.
    - Easy correctness/termination proofs.
  - Disadvantage:
    - Method call overhead.
- To overcome method call overhead many programmers:
  - First implement a recursive algorithm; and
  - Then transform it to an equivalent iterative algorithm.
- If a method has at most one recursive call it is called *tail recursive.*
- They can be transformed to equivalent iterative algorithms.

Fibonacci Numbers

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

We may compute the *n*th member of the sequence as follows:

$$f_n = \begin{cases} 1 & \text{if } n \leq 1 ; \\ f_{n-1} + f_{n-2} & \text{otherwise} . \end{cases}$$

Fibonacci Numbers

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

We may compute the *n*th member of the sequence as follows:

$$f_n = \begin{cases} 1 & \text{if } n \leq 1\,; \\ f_{n-1} + f_{n-2} & \text{otherwise}\,. \end{cases}$$

Java

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

    if (n <= 1) {
        result = 1;
    } else {
        result = f( n - 1 ) + f( n - 2 );
    }

    return result;
}
```

Fibonacci Numbers

Not Tail Recursive

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

We may compute the *n*th member of the sequence as follows:

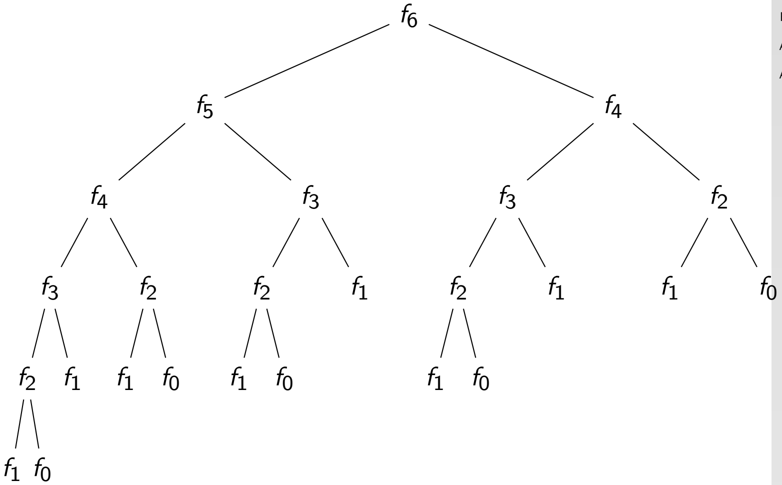
$$f_n = \begin{cases} 1 & \text{if } n \leq 1; \\ f_{n-1} + f_{n-2} & \text{otherwise}. \end{cases}$$

Java

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

    if (n <= 1) {
        result = 1;
    } else {
        result = f( n - 1 ) + f( n - 2 );
    }

    return result;
}
```

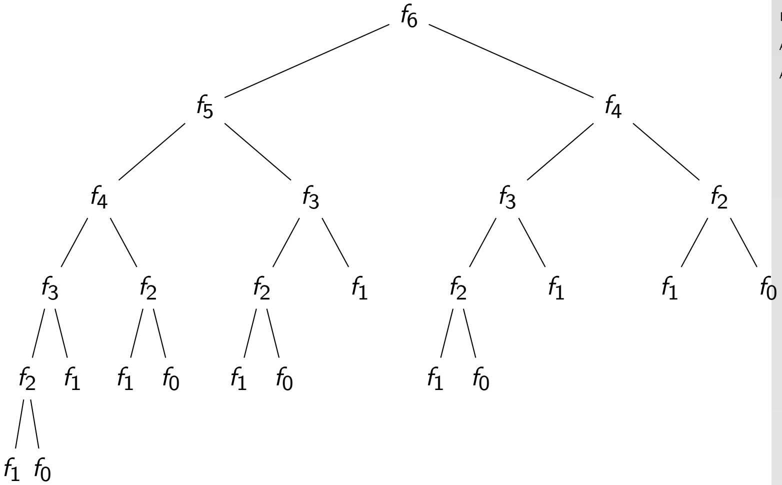
Time Complexity of Naive Computation of f_n

Previous Method for Computing f_n is Hopelessly Inefficient



Time Complexity of Naive Computation of f_n

Need almost Twice as Many Calls to f_{n-2} as to f_{n-1}



More Intelligent Computation of f_n

- Trick: Compute

$$\underbrace{\langle\, f_0, f_1 \,\rangle\, , \langle\, f_1, f_2 \,\rangle\, , \langle\, f_2, f_3 \,\rangle\, , \ldots , \langle\, f_{n-1}, f_n \,\rangle}_{\text{length } n},$$

and return $f_{\max(1,n)}$ (the second member of the last pair).

More Intelligent Computation of f_n

- **Trick**: Compute

$$\underbrace{\langle f_0, f_1 \rangle, \langle f_1, f_2 \rangle, \langle f_2, f_3 \rangle, \ldots, \langle f_{n-1}, f_n \rangle}_{\text{length } n},$$

and return $f_{\max(1,n)}$ (the second member of the last pair).

- The following shows how to do this recursively:

$$f(n) = F(\langle 1, 1 \rangle, 1, \max(1, n)),$$

where

$$F(\langle f_{i-1}, f_i \rangle, i, n) = \begin{cases} f_i & \text{if } i = n, \\ F(\langle f_i, f_{i-1} + f_i \rangle, i + 1, n) & \text{otherwise}. \end{cases}$$

Possible Implementation

Java

```java
public class Pair<S,T> {
    private S first;
    private T second;

    public Pair( final S first, final T second ) {
        this.first = first;
        this.second = second;
    }

    public S getFirst( ) {
        return first;
    }

    public void setFirst( final S first ) {
        this.first = first;
    }

    ...
}
```

Implementation (Continued)

Java

```
private static int fibonacci( final int order ) {
    final Pair<Integer,Integer> pair
        = new Pair<Integer,Integer>( 1, 1 );
    return fibonacci( pair, 1, order );
}
```

Implementation (Continued)

Java

```
private static int fibonacci( final Pair<Integer,Integer> numbers,
                              final int currentOrder,
                              final int order ) {
    final int second = numbers.getSecond( );
    final int result;
    if (currentOrder == order) {
        result = second;
    } else {
        final int first = numbers.getFirst( );
        final int sum = first + second;
        final Pair<Integer,Integer> pair
            = new Pair<Integer,Integer>( second, sum );
        result = fibonacci( pair, currentOrder + 1, order );
    }
    return result;
}
```

An Interesting Observation

No need to explicitly construct pairs $\langle f_i, f_{i+2} \rangle$

$$f_n = \begin{cases} 1 & \text{if } n = 0\,; \\ F(1, 1, 1, n) & \text{otherwise}\,, \end{cases}$$

where $F(f_{i-1}, f_i, i, n)$ is given by:

$$F(f_{i-1}, f_i, i, n) = \begin{cases} f_i & \text{if } i = n\,; \\ F(f_i, f_i + f_{i-1}, i + 1, n) & \text{otherwise}\,. \end{cases}$$

Recursive Implementation

C Program

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    final int result;

    if (curr == n) {
        result = fibCurr;
    } else {
        result = F( fibCurr, fibPrev + fibCurr, curr + 1, n );
    }

    return result;
}
```

Recursive Implementation: Tail Recursive!

C Program

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    final int result;

    if (curr == n) {
        result = fibCurr;
    } else {
        result = F( fibCurr, fibPrev + fibCurr, curr + 1, n );
    }

    return result;
}
```

Iterative Implementation

Java

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr

Iterative Implementation

Java

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1

Iterative Implementation

Java

```java
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1
1	2	2

Iterative Implementation

Java

```java
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1
1	2	2
2	3	3

Iterative Implementation

Java

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1
1	2	2
2	3	3
3	5	4

Iterative Implementation

Java

```java
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1
1	2	2
2	3	3
3	5	4
5	8	5

Iterative Implementation

Java

```
public static int int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

fibPrev	fibCurr	curr
1	1	1
1	2	2
2	3	3
3	5	4
5	8	5
8	13	6

For Wednesday

- Study [Horstmann 2013, Sections 12.1–12.2].

Acknowledgements

- This lecture corresponds to [Horstmann 2013, Sections 12.1–12.2].

About this Document

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