Lecture 12: Reasoning About Recursive Algorithms

CS2504/CS4092– Algorithms and Linear Data Structures

Dr Kieran T. Herley

Department of Computer Science
University College Cork

2013/14

Summary

Review of simple recursive functions. Recursion tree concept. Using recursion tree and induction to reason about recursive algorithms. Fibonacci numbers. Towers of Hanoi problem. Recursive binary search.

Anatomy of A Recursive Method

Algorithm consists of

```
Algorithm sum(n)
  if n ≤ 1 then
      return n                    // base
   else
      return sum(n−1) + n // rec
```

Anatomy of A Recursive Method

```
Algorithm sum(n)
  if n ≤ 1 then
      return n                    // base
   else
      return sum(n−1) + n // rec
```

Algorithm consists of

Base Case(s)

- "Simple" case; solves problem directly; no recursion

Anatomy of A Recursive Method

```
Algorithm sum(n)
  if n ≤ 1 then
      return n                    // base
   else
      return sum(n−1) + n // rec
```

Algorithm consists of

Base Case(s)

- "Simple" case; solves problem directly; no recursion

Recursive Case(s)

- sum(n) "delegates" / "subcontracts" work to sum(n−1)

Anatomy cont'd

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
   else
      return sum(n−1) + n
```

Observation

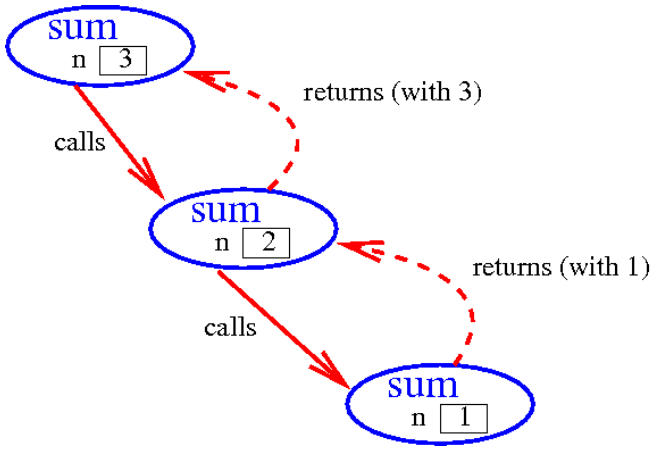
Each recursive call "makes progress" towards completing task: subcomputation sum(n−1) "simpler" (i.e. "closer" to base case) than sum(n)

Intermission

View recursion trace slides [recsumtrace.pdf] here.

Recursion Tree

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
  else
      return sum(n−1) + n
```



Recursion Tree

- Tree-like representation of entire computation
- Each call represented by a node
- Parent-child relationship models caller-callee pattern

Proving That sum Works

Proposition

For all $n \geq 1$,

sum(n) returns the sum of the first n natural numbers.

- Articulate *precise* statement of what method is intended to achieve
- Formulate argument that method conforms to statement
- (Chasing discrepencies between staement and observed behaviour is also a useful debugging tool.)

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
  else
      return sum(n−1) + n
```

Proving That sum Works

Proposition

For all $n \geq 1$, sum(n) returns the sum of the first n natural numbers.

- Will prove by *induction on n*.
- Base case ($n = 1$):
  - sum(1) clearly returns 1
  - This is sum of first 1 natural numbers
  - *i.e.* Prop. true for $n = 1$

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
  else
      return sum(n−1) + n
```

Proof cont'd

Inductive case

- Suppose Prop. true for all $n' < n$ (inductive hypothesis) and consider sum(n) .
- Method executes else clause
- By inductive hypothesis, sum(n-1) returns $1 + 2 + \cdots + n - 1$
- So

$$\text{sum}(n-1) + n = \underbrace{1 + 2 + \cdots + n - 1}_{\text{sum}(n-1)} + n$$

  i.e. sum of first n natural numbers
- Thus Proposition holds for n also.

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
   else
      return sum(n−1) + n;
```

Proof cont'd

Inductive case

- Suppose Prop. true for all $n' < n$ (inductive hypothesis) and consider sum(n) .
- Method executes else clause
- By inductive hypothesis, sum(n-1) returns $1 + 2 + \cdots + n - 1$
- So

$$\text{sum}(n-1) + n = \underbrace{1 + 2 + \cdots + n - 1}_{\text{sum}(n-1)} + n$$

  i.e. sum of first n natural numbers
- Thus Proposition holds for n also.

```
Algorithm sum(n)
  if n ≤ 1 then
      return n
   else
      return sum(n−1) + n;
```

Base and Inductive cases imply Proposition holds for all n.

Fibonacci Numbers

- The Fibonacci sequence is defined as follows:
  - Each number is the sum of the two preceeding ones
  - The first two numbers are 0 and 1
- The sequence begins:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$$

- Can re-state definition s following equation:

$$f_n \triangleq \begin{cases} n & \text{if } n \leq 1 \\ f_{n-2} + f_{n-1} & \text{if } n > 1. \end{cases}$$

1

[1] Symbol $\triangleq$ denotes "is defined to be".

A Recursive Fibonacci Algorithm

Equation

$$f_n \triangleq \begin{cases} n & \text{if } n \leq 1 \\ f_{n-2} + f_{n-1} & \text{if } n > 1. \end{cases}$$

Algorithm

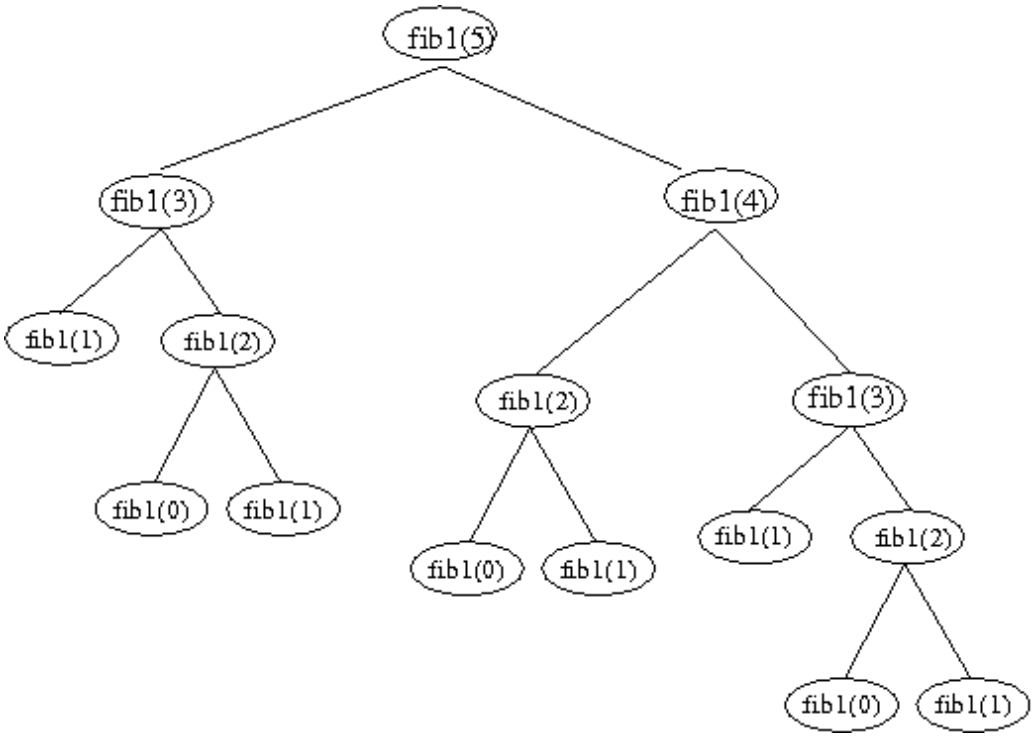
```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) + fib(n−1)
```

[2]

[2] Not the best way to compute Leo's numbers

Recursion Tree for fib(5) Computation

Algorithm fib(n):
    if n $\leq$ 1 **then**
        return n
    else
        return fib(n−2) +
                fib(n−1)



Note multiple invocations of some fib(k) quantities–wasteful

Proving That fib Works

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

Proposition

For all $n \geq 1$, fib(n) returns f_n (the n^{th} Fibonacci number).

- Will prove by induction on n.

Proving That fib Works

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

Proposition

For all $n \geq 1$, fib(n) returns f_n (the n^{th} Fibonacci number).

- Will prove by induction on n.
- Base case ($n = 0, 1$):
  - fib(0) returns $0 = f_0$ (zeroth Fibonacci no.)
  - fib(1) returns $1 = f_1$ ("oneth" Fibonacci no.)
- *i.e.* Prop. true for $n = 0, 1$

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–returns f_{n-2} (by IH)

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–returns f_{n-2} (by IH)
    - fib(n-1)–

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–returns f_{n-2} (by IH)
    - fib(n-1)–returns f_{n-1} (again by IH)

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

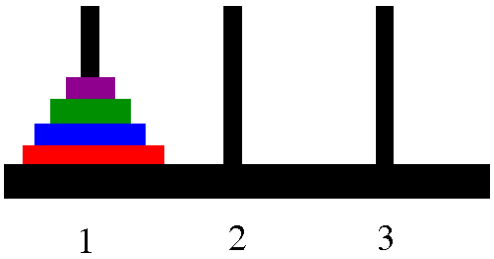
- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–returns f_{n-2} (by IH)
    - fib(n-1)–returns f_{n-1} (again by IH)
  - The value returned, fib(n-2) plus fib(n-1), is thus the sum of f_{n-2} and f_{n-1}

Proof cont'd

```
Algorithm fib(n):
    if n ≤ 1 then
        return n
    else
        return fib(n−2) +
               fib(n−1)
```

- Inductive case ($n > 1$):
  - By Inductive Hypothesis (IH) fib(n') returns $f_{n'}$ for all $n' < n$
  - Consider fib (n): algorithm computes
    - fib(n-2)–returns f_{n-2} (by IH)
    - fib(n-1)–returns f_{n-1} (again by IH)
  - The value returned, fib(n-2) plus fib(n-1), is thus the sum of f_{n-2} and f_{n-1} *i.e.* f_n, the n-th Fib. no.

The Towers Of Hanoi



Setting

- three poles
- n disc of varying size
- discs stacked in order of size on pole 1

Goal Transfer all discs to pole 3

Rules

- Can only move top disc on pole
- Can move only one disc at at time
- Cannot place any disc on top of a smaller one

Some Easy Cases

n = 1

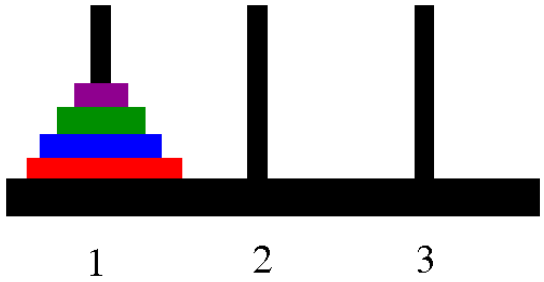
1 –> 3

n = 2

1 –> 2 1 –> 3 2 –> 3

Can we devise a general strategy?

A Useful Observation



Consider time when largest disc moves from 1 to 3

A Before this, the other $n-1$ must be shifted "out of the way" to pole 2

B (Largest disc moved from 1 to 3)

C After this, the other $n-1$ must be shifted from pole 2 to 3

Observation

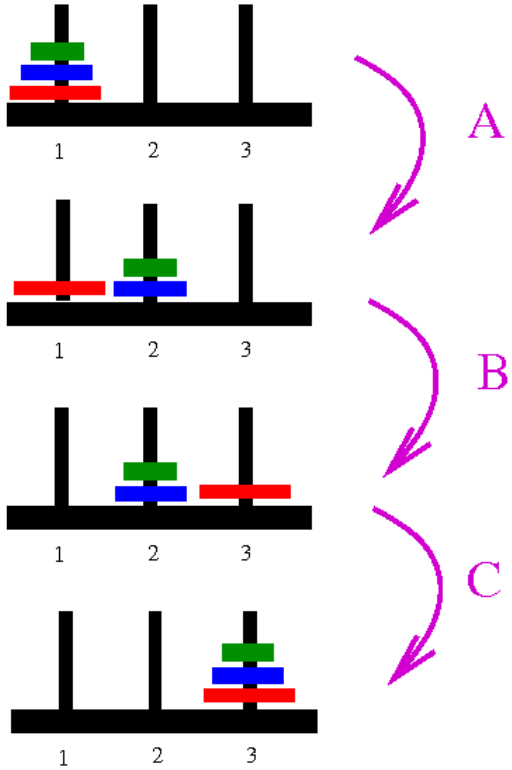
Task of shifting n discs between two poles can be completed using two subtasks (A and C) each shifting $n-1$ discs between poles

Illustration

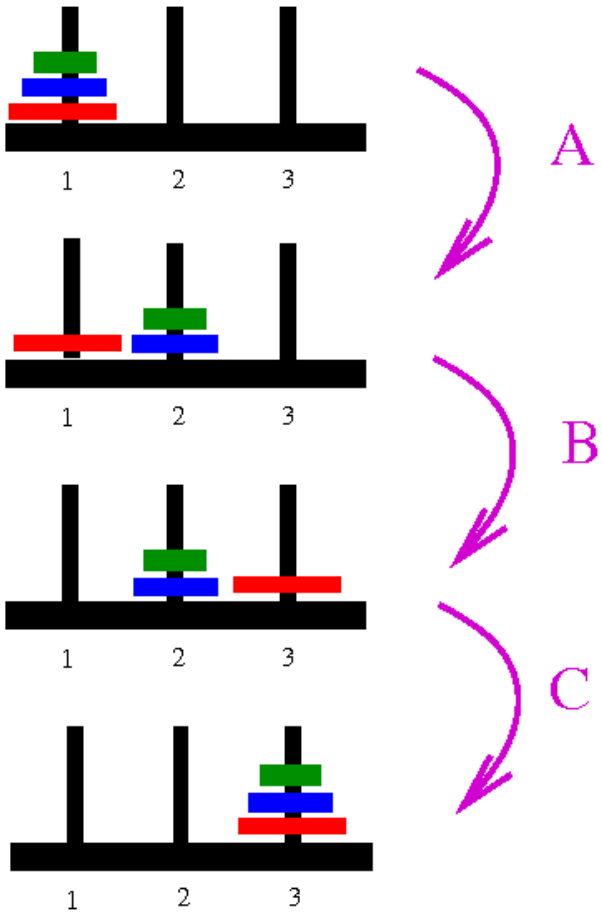
```
move top disc from 1 to 3 )
move top disc from 1 to 2 ) A
move top disc from 3 to 2 )

move top disc from 1 to 3 ) B

move top disc from 2 to 1 )
move top disc from 2 to 3 ) C
move top disc from 1 to 3 )
```



Our Strategy



To move k discs from f to t (using pole s as "spare")

$k = 0$ do nothing!

$k > 0$

A move top $k-1$ discs from f to s

B move top disc from f to t

C move top $k-1$ discs from s to t

Algorithm

Strategy

To move k discs from f to t (pole s is "spare")

$k = 0$ do nothing!

$k > 0$

A move top $k-1$ discs from f to s

B move top disc from f to t

C move top $k-1$ discs from s to t

Pseudocode

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
     printf ("move top disc from %d to %d\n", from, to)
    move(k−1, spare, to, from)


. . .
move(n, 1, 3, 2)
```

Illustration ($k = 0$)

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
     printf ("move top disc from %d to %d\n",
                 from, to)
    move(k−1, spare, to, from)

. . .
move(n, 1, 3, 2);
```

Effect none

Comment correctly moves zero discs!

Recursion Tree

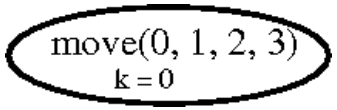


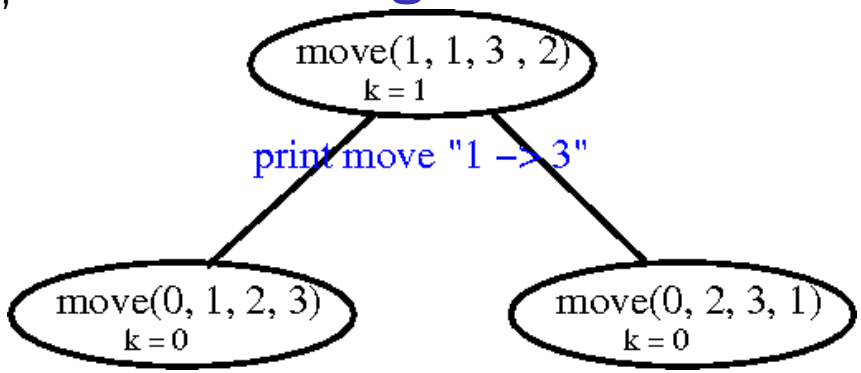
Illustration ($k = 1$)

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
     printf ("move top disc from %d to %d\n"
                 from, to)
    move(k−1, spare, to, from)
```

Effect Moves disc between "from" an "to" poles

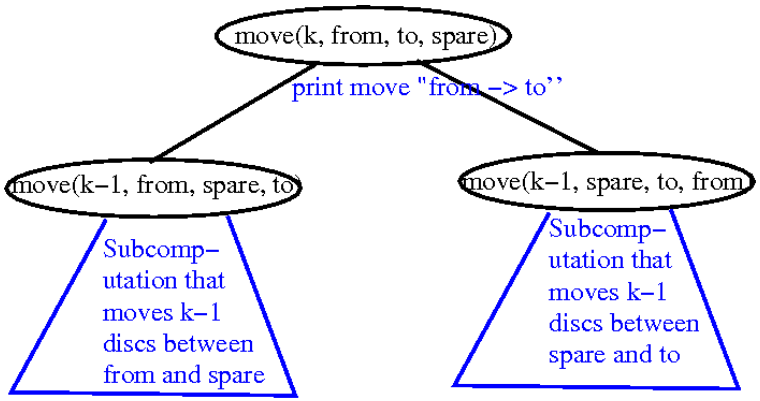
Comment Correctly moves one disc

Recursion Diagram



General Picture

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
     printf ("move disc from %d to %d\n",
                 from, to)
    move(k−1, spare, to, from)
```



Proving That Algorithm Works

Proposition For all k, if discs 1 to k rest on pole `f` and poles `f`, `t`, `s` are distinct, then `move(k, f, t, s)` will move these k discs from pole `f` to pole `t`.

Base Case $k = 0$

- If-test ($k > 0$) is false, so no action taken
- Correctly moves (zero) discs as specified!

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
    printf ("move disc from %d to %d\n",
            from, to)
    move(k−1, spare, to, from)
```

Proof cont'd

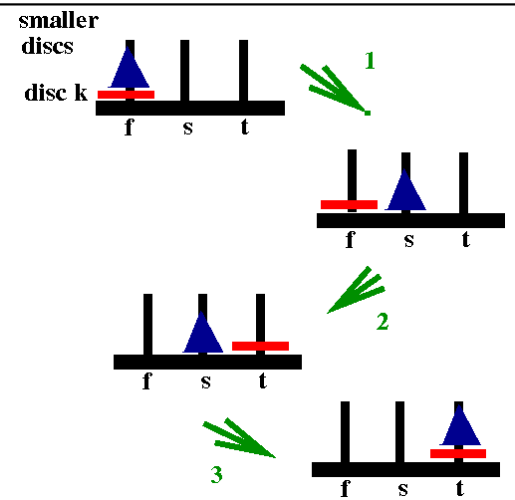
Proposition For all k, if discs 1 to k rest on pole `f` and poles `f`, `t`, `s` are distinct, then `move(k, f, t, s)` will move these k discs from pole `f` to pole `t`.

Inductive Case $k > 0$

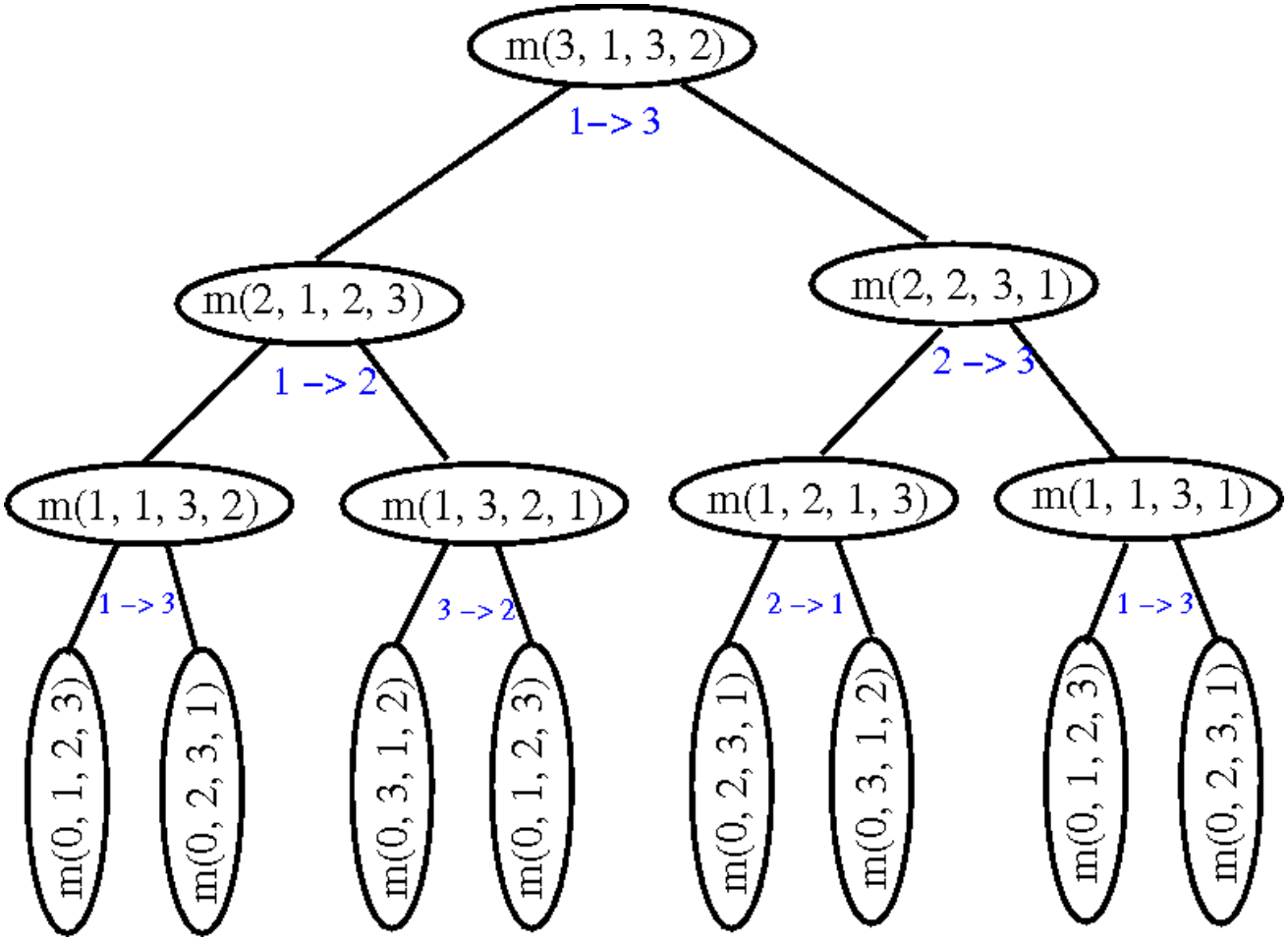
1. `move(k-1, from, spare, to)` moves all but largest (disc k) onto pole `spare` (ind. hyp.)
2. `print` moves largest (disc k) directly to pole `to`
3. `move(k-1, spare, to, from)` moves all but largest onto pole `to` (ind. hyp.)

Taken together, shows algorithm moves k discs to pole `t`

```
Algorithm move(k, from, to, spare)
  if k >0 then
    move(k−1, from, spare, to)
    printf ("move disc from %d to %d\n",
            from, to)
    move(k−1, spare, to, from)
```



Three-Disc Illustration



`move` abbreviated to `m`; $x \rightarrow y$ abbreviates "move disc from x to y"; inorder traversal of rec. tree yields sequence of moves

Illustration cont'd

$$(1 \to 3)^*, (1 \to 2), (3 \to 2)^*, (1 \to 3), (2 \to 1)^*, (2 \to 3), (1 \to 3)^*$$

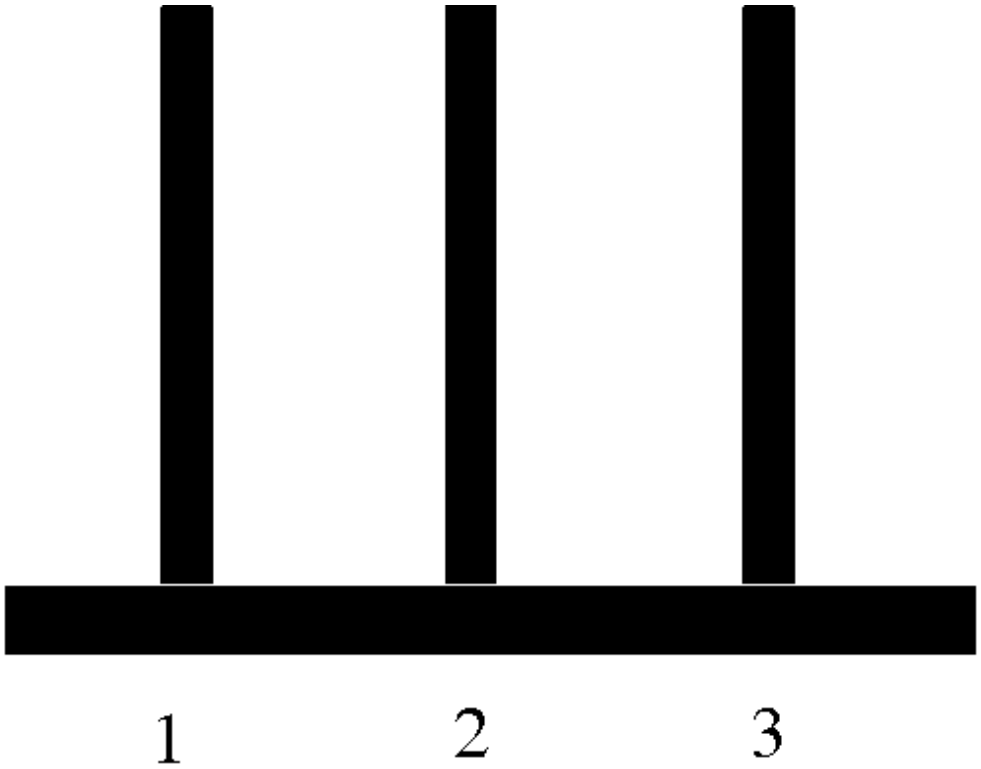
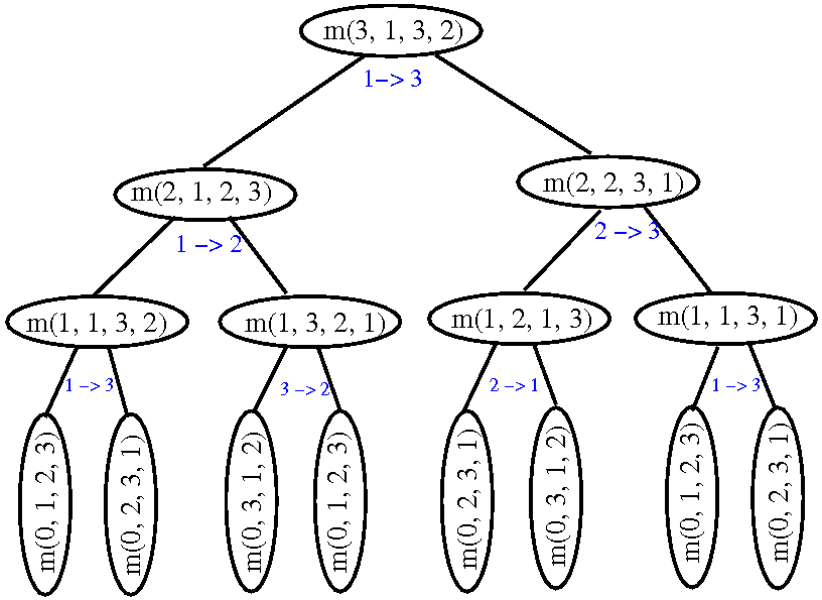
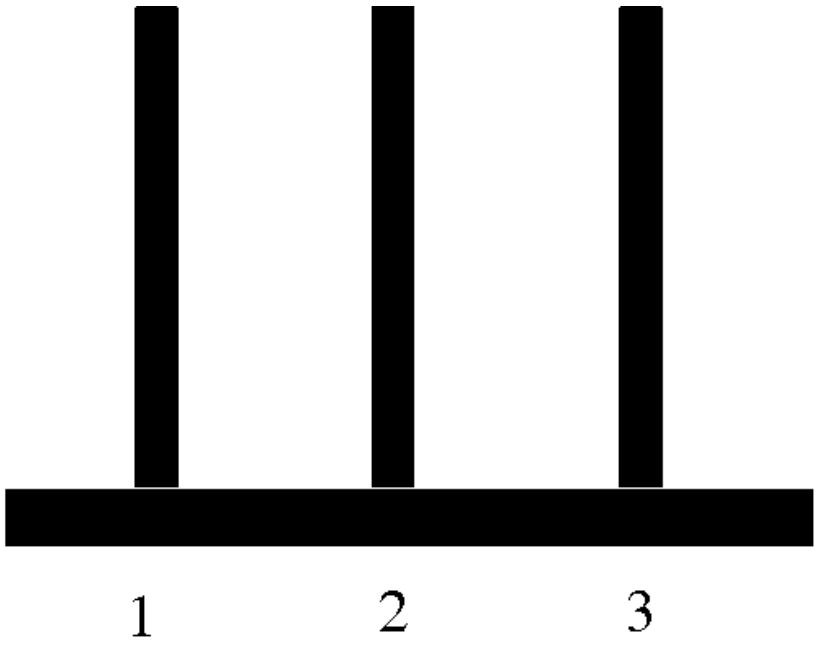


Illustration cont'd

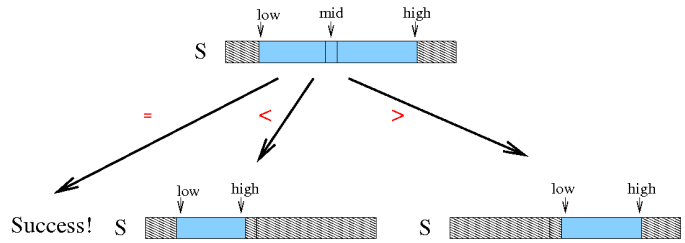


$$(1 \to 3)^*, (1 \to 2), (3 \to 2)^*,$$
$$(1 \to 3), (2 \to 1)^*, (2 \to 3),$$
$$(1 \to 3)^*$$



Binary Search (Recursive)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return
                BinarySearch(S,k, low, mid−1)
        else
            return
                BinarySearch(S,k,  mid+1, high)
```

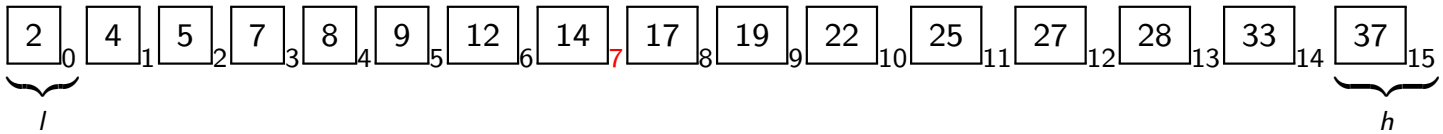


BS in Action ($k = 22$)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S, k,  low, mid−1)
        else
            return BinarySearch(S,k,  mid+1, high)
```

Observation

Exactly the same pattern of comparisons as for non-recursive version seen earlier– same principles, different packaging

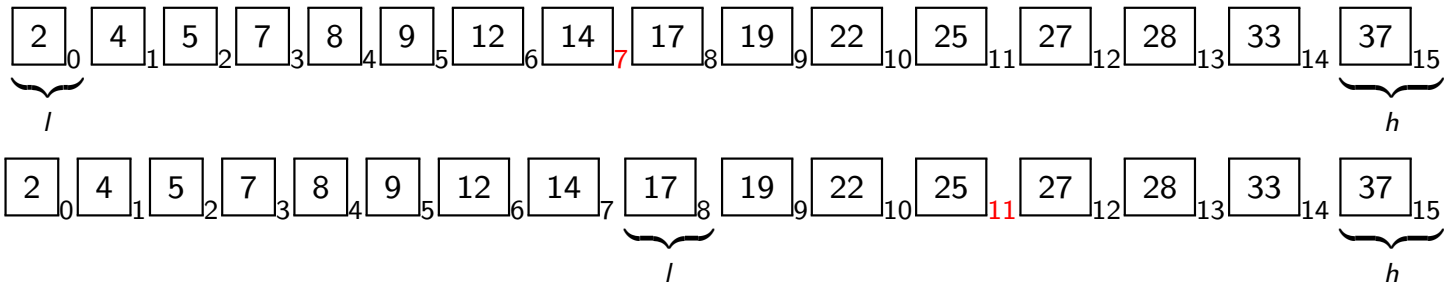


BS in Action ($k = 22$)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S, k,   low, mid−1)
        else
            return BinarySearch(S,k,   mid+1, high)
```

Observation

Exactly the same pattern of comparisons as for non-recursive version seen earlier– same principles, different packaging

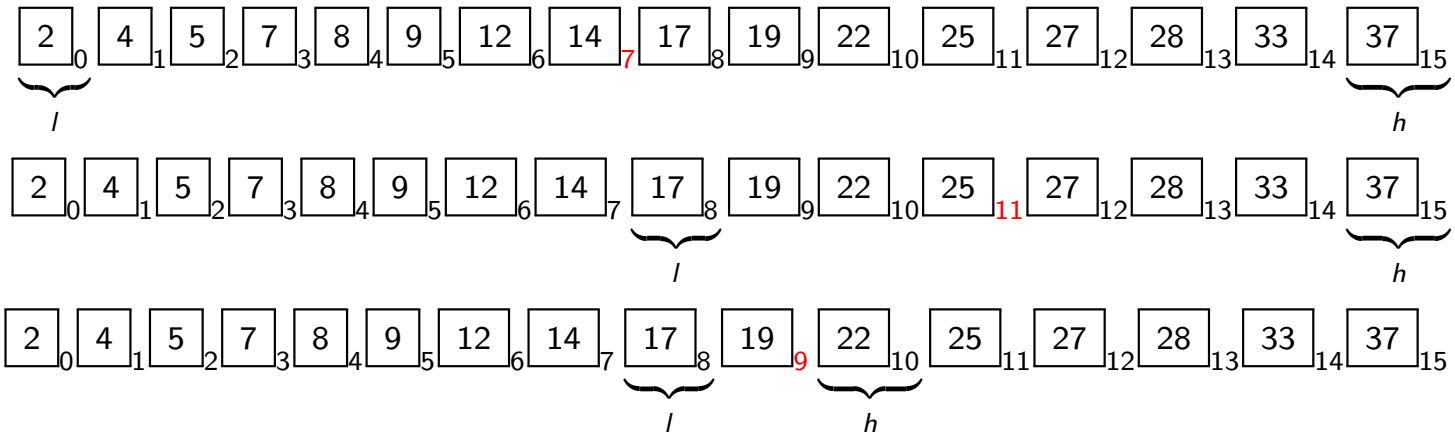


BS in Action ($k = 22$)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S, k,   low, mid−1)
        else
            return BinarySearch(S,k,   mid+1, high)
```

Observation

Exactly the same pattern of comparisons as for non-recursive version seen earlier– same principles, different packaging

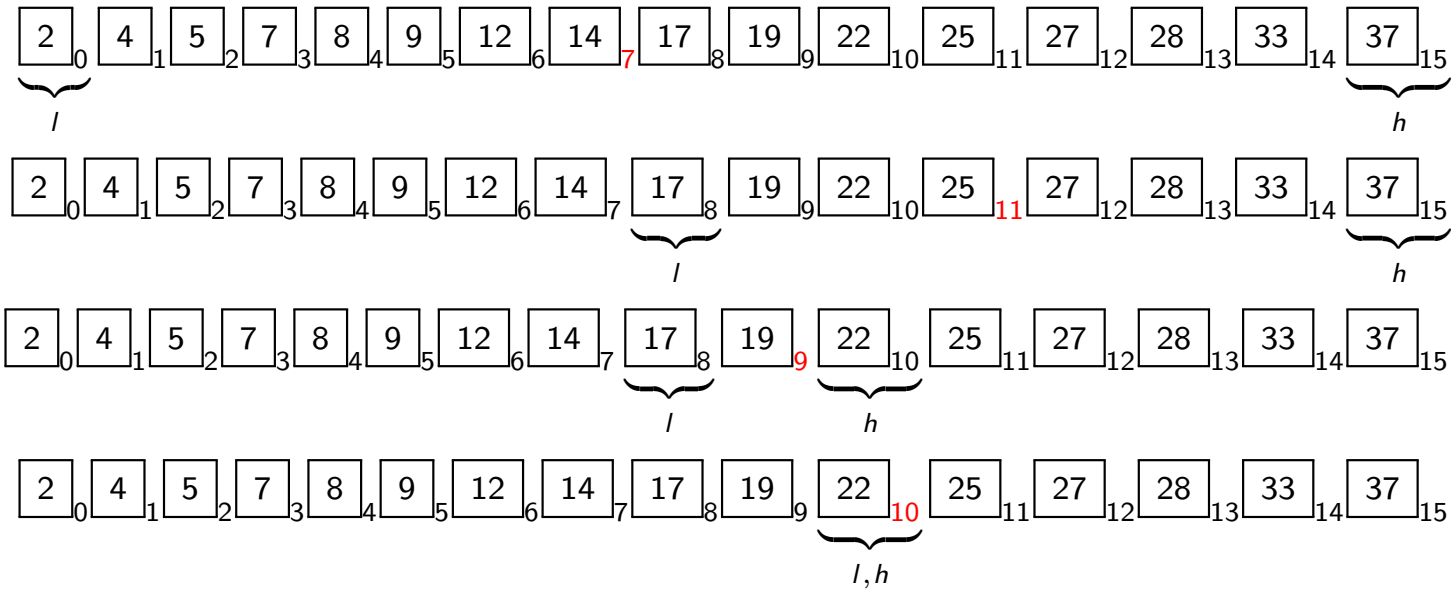


BS in Action ($k = 22$)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S, k,  low, mid−1)
        else
            return BinarySearch(S,k,  mid+1, high)
```

Observation

Exactly the same pattern of comparisons as for non-recursive version seen earlier– same principles, different packaging



Binary Search (Recursive)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S,k,  low, mid−1)
        else
            return BinarySearch(S,k,  mid+1, high)
```

Proposition

For all n, $BinarySearch(S, k, \ell, h)$ returns index within $S[\ell..h]$ housing k (or -1 when k is not present).

Note: The length of $S[\ell..h]$ is $h - \ell + 1$ *i.e.* the number of entries it contains; interpret $h < \ell$ as empty interval (length zero)

Proof That Recursive Algorithm Works

Proposition

For all n, BinarySearch(S, k, ℓ, h) returns index within $S[\ell..h]$ housing k (or -1 when k is not present).

- Proceed by induction on search interval length
- Base Case: length $= 0$ (*i.e.* $h < \ell$):
  - empty interval, k not present
  - alg. returns -1, as required

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(S,k, low, mid−1)
        else
            return BinarySearch(S, k, mid+1, high)
```

Proof cont'd

Proposition

For all n, BinarySearch(S, k, ℓ, h) returns index within $S[\ell..h]$ housing k (or -1 when k is not present).

- Inductive Case: non-empty interval (*i.e.* $l \leq h$):
- Assume proposition holds for smaller intervals (IH)

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(
                S,k, low, mid−1)
        else
            return BinarySearch(
                S,k, mid+1, high)
```

Proof cont'd

Proposition

For all n, BinarySearch(S, k, ℓ, h) returns index within $S[\ell..h]$ housing k (or -1 when k is not present).

```
Algorithm BinarySearch(S, k, low, high)
    if low > high then
        return −1
    else
        mid ←(low + high)/2
        midKey ←key(mid)
        if k = midKey then
            return mid
        else
        if k < midKey then
            return BinarySearch(
                S,k, low, mid−1)
        else
            return BinarySearch(
                S,k, mid+1, high)
```

- Inductive Case: non-empty interval (*i.e.* $l \leq h$):
- Assume proposition holds for smaller intervals (IH)
- Consider case $k =$ key(mid)
  - alg. returns index of k, *i.e.* mid
- Consider case $k <$ key(mid)
  - if k in $S[\ell..h]$, it must be in $S[\ell..\text{mid} - 1]$
  - BinarySearch$(S, k, \ell, \text{mid} - 1)$ returns
    - index of k in $S[\ell..\text{mid} - 1]$, if present
    - -1, if not present
- Consider case $k <$ key(mid)
  - similar to above

Alternative Implementation for findEntry in ArrayBasedMap2

```
public ValueType get(KeyType k)
{   int indexWithKey = findEntry(k);
    if (indexWithKey != NO_SUCH_KEY)
        return ( entries [indexWithKey]).getValue();
    else
        return null;
}
private int findEntry(KeyType key)
{   return findEntry(key, 0, size()-1);
}
```

ArrayBasedMap2 cont'd

```
private int findEntry(KeyType key, int low, int high)
{   if (low > high)
    {   return NO_SUCH_KEY;
    }
    else
    {   int mid = (low + high)/2;
        Entry<KeyType, ValueType> e = entries[mid];
        int compResult =
          comp.compare(key, e.getKey());
        if (compResult == 0)
        {   return mid;
        }
        else
        if (compResult < 0)
        {   return findEntry(key, low, mid-1);
        }
        else
        {   return findEntry(key, mid+1, high);
        }
    }
}
```