Introduction to Computer Science Week 10

Recursion

- How to think about it
- Example: Quicksort
- Stepwise refinement:
  using specification before method is implemented

Programming by contract

① That can't work can it?
② That looks scary
③ Ah! Now I see.

Steve Vickers 2014

A method is recursive if it calls itself
- directly (a calls a)
- or indirectly (a calls b calls c calls---- a)

Implementation

Recursion ⇒
return address, parameters, local variables of method have to be kept safe at different levels for different depths of recursion
⇒ use a stack frame for each call
- e.g. as in JVM.

Why bother?

Is recursion really necessary?

e.g. factorial

```
public static int fact (int n){
  if (n == 0){
    return 1;
  } else {
    return n*fact(n-1);
  }
}
```

Just as effective to use repetition —

Devil's advocate

```
public static int fact (int n){
  int a = 1;
  for (int i = 1; i <= n; i++){
    a = a*i;
  }
  return a;
}
```

e.g. Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

each is sum of previous two

recursive:

```
public static int fibRec(int n){
    if (n <= 0){
        return 0;
    } else if (n == 1){
        return 1;
    } else {
        return fibRec(n-2) + fibRec(n-1);
    }
}
```

Fibonacci – repetition

```
public static int fibRep(int n){
  if (n <= 0) {
    return 0;
  } else {
    int i = 1;
    int a = 0;
    int b = 1;
    /* loop invariant
     * 1 <= i <= n, a = fib(i-1), b = fib(i)
     */
    while (i < n){
      int c = a+b; // fib(i+1)
      a = b;
      b = c;
      i++;
    }
    return b;
  }
}
```

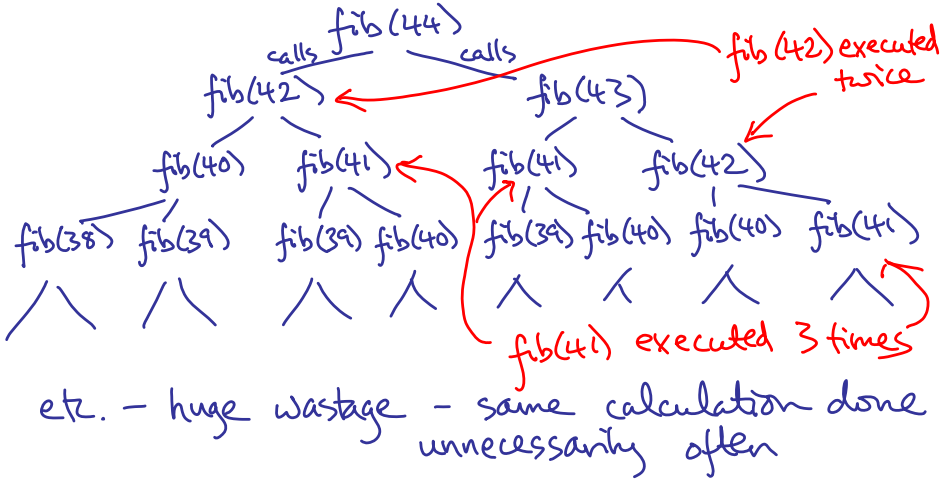
establishes invariant

replaces a,b by b, a+b – reestablishes invariant for new i

– invariant & i ≥ n imply b is correct answer

For Fibonacci, repetition is much better!

For recursion: what calls what? it's quicker



etc. – huge wastage – same calculation done unnecessarily often

Could you ban recursion?

How would Java do it?

Suppose a is a public method in class C,
x is a variable of interface type
a calls x.b

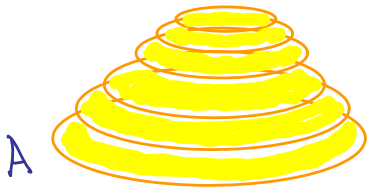
When compiling class C, compiler doesn't know how b will be implemented.
– it might call a, so a becomes recursive

∴ difficult to ban recursion in Java

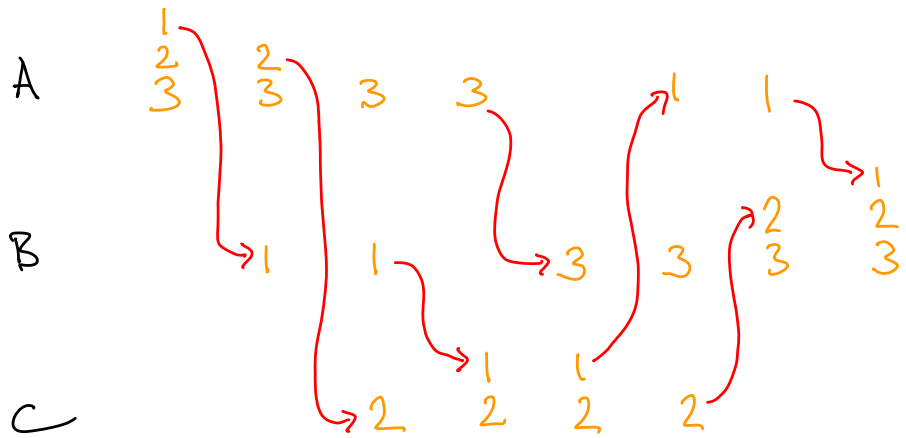
Sometimes recursion is just what you want!

– sometimes repetition difficult

e.g. Towers of Hanoi
- move pile of golden discs from A to B
- only allowed 3 piles, A, B, C
- only move one at a time, top of one pile to top of another
- never have a larger disc on top of a smaller

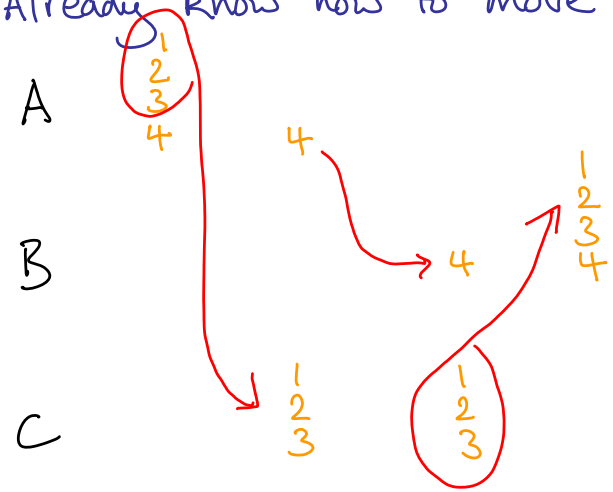


e.g. with 3 discs



How to move 4 discs?

Already know how to move 3 discs.



Interface for Towers of Hanoi

```
public interface Towers {
  /** getter for total number of discs
   * @return number of discs
   */
  public int getTotal();

  /** Move top of one pile to another.
   *
   * @param from pile to move from
   * @param to     pile to move to
   * @throws IllegalArgumentException if -
   *   from, to not in range 0,1,2, or
   *   "from" pile is empty, or
   *   top of "from" pile is bigger that top of "to".
   */
  public void move(int from, int to);
}
```

```
public static void solve(Towers t){
    solveN(t, t.getTotal(), 0, 1);
}

private static void solveN(
    Towers t, int n, int from, int to
){
    if (n == 0 || from == to){
        return;
    }
    int other = 3 - from - to;
    solveN(t, n-1, from, other);
    t.move(from, to);
    solveN(t, n-1, other, to);
}
```

moves all discs from pile 0 (A) to 1 (B)

other pile - not from or to

Why do we believe it works?

First try — what does it execute?

e.g. for 4 discs

solveN(t, 4, 0, 2)

- calls solveN(t, 3, 0, 1)
  - calls solveN(t, 2, 0, 2)
    - calls solveN(t, 1, 0, 1)
      - calls solveN(t, 0, 0, 2) ✓ trivial
      - t.move(0, 1)
      - calls solveN(t, 0, 2, 1) ✓ trivial
    - t.move(0, 2)
    - calls solveN(t, 1, 1, 2) ------

Quickly get lost

brain hurts

Why do we believe it works?

2nd try: think of recursive calls as subcontractors

Trust them to do what they are contracted to do.

Central question

What is that?

What is the contract?

What do the subcontractors require, what do they deliver?

We're trying to move top n discs "from" pile → "to" pile

within Towers of Hanoi rules

```
private static void solveN(
    Towers t, int n, int from, int to
){
    if (n == 0 || from == to){
        return;
    }
    int other = 3 - from - to;
    solveN(t, n-1, from, other);
    t.move(from, to);
    solveN(t, n-1, other, to);
}
```

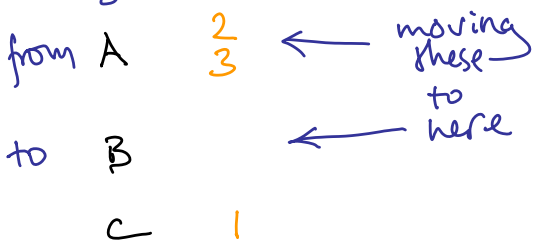
subcontractor moves top n-1 to "other" pile

we move n'th disc

subcontractor moves n-1 discs to on top of n'th

How do we know rules are obeyed?

e.g. moveN(t, 2, 0, 1) in situation



Algorithm says move 2 on top of 1 - illegal!

∴ algorithm requires some preconditions – otherwise it doesn't work.

These are part of the contract.

Preconditions for moveN

The n discs we are moving must be the smallest n discs.

Then there is no problem putting them on top of any of the others.

```
/**
 * Moves a number of discs from top of one pile to top
 *   of another, within the Tower of Hanoi rules.
 * requires: the discs to be moved are the n smallest
 *   discs of all.
 * @param t      Towers system to use
 * @param n      number of discs to move
 * @param from   pile to move from
 * @param to     pile to move to
 */
private static void solveN(Towers t, int n, int from, int to) {
    if (n == 0 || from == to){
        return;
    }
    int other = 3 - from - to;
    solveN(t, n-1, from, other);
    t.move(from, to);
    solveN(t, n-1, other, to);
}
```

precondition

NB The n discs start in the correct order, smallest on top.

Hence

The n-1 discs here are the n-1 smallest. Precondition is established for recursive call

Moving nth smallest. All smaller ones on "other", ∴ OK to move to "to"

Precondition established as before

Recursive programming: general principles

Think: If I could get someone else to solve the same problem in a simpler situation, could I use their help?

Draw up a contract
- What precisely is "the same problem"?
- What preconditions are needed? "requires"

Assume recursive calls meet their contract
- Remember to establish their preconditions before you call them

Use their results to meet your own contract

Termination

There's a risk recursion might never stop

You'll get StackOverflowError at run-time – more and more stack frames created, till no more room

e.g.
```
void infinity(){
    infinity();
}
```

cf. loops

To avoid this

① Recursive call must "make progress" – parameters must be simpler by some measure
e.g. moveN – fewer discs to move

② Must eventually "bottom out" at special base case with no recursion e.g. moving 0 discs – do nothing

③ The measure of progress is a recursion variant. It must be smaller on each recursive call. e.g. $n \mapsto n-1$

Simpler example: factorial

The recursion variant here is n

```
/** Calculate factorial
 * requires: n >=0
 * @param n a number
 * @return n!
 */
public static int fact(int n){
  if (n == 0){
    return 1;
  } else {
    return n*fact(n-1);
  }
}
```

n > 0

hence n-1 ≥ 0 – precondition established

progress made: n-1 < n
recursion variant has gone from n to n-1

QUICKSORT

Sorting algorithm by Tony Hoare

- essentially recursive
  - no easy equivalent with repetition
- a subproblem
  in-place partition = Dijkstra's "Dutch National Flag" algorithm

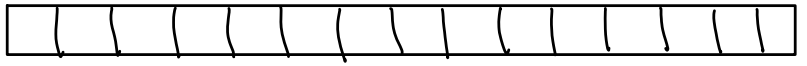
is good illustration of loop invariant

See "Reasoned Programming"

Array of integers to sort (ascending order)

Basic idea – but we'll modify it slightly

Starts off unsorted



① "Partition" the array: rearrange so small elements all at one end, large at other

② Do the same recursively to sort each of the two parts.

What do "small" and "big" mean?

Choose some "pivot" value x so

small means $< x$

big means $\geqslant x$

See how to choose x later.

But note – we don't try to get equal numbers of big and small

The partition could be very unequal.

Base case

When a region has 0 or 1 elements we don't need a recursive call

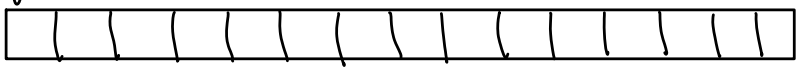
– it's already sorted.

Making progress

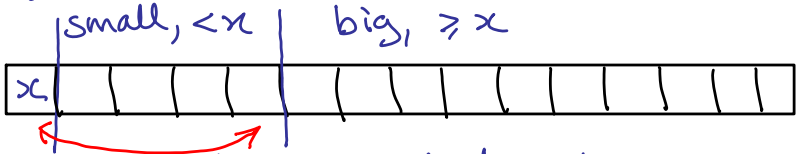
Want partition regions to be shorter than original

– no progress if all small or all big

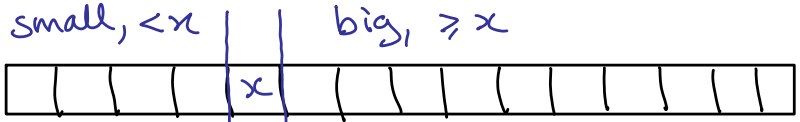
Modified idea starts off unsorted



(1a) Use first element as pivot x, partition the rest



(1b) Swap x with last small element



(2) Use recursion to sort small & big regions

Advantages

(1) Quick & easy way to choose a pivot x

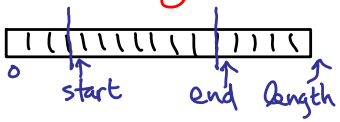
(2) Makes progress – even in worst case (all elements big or all small) partition regions are shorter than the original because they miss out the x

```
/**
 * Sort a region of an array in ascending order.
 * Elements outside the given region are unchanged.
 * requires: 0 <= start <= end <= arr.length
 * @param arr   array to sort
 * @param start start of region (inclusive)
 * @param end   end of region (exclusive)
 */
private static void qs(int[] arr, int start, int end){
  if (end <= start+1){    //region of length 0 or 1
    return;
  }
  partition
  swap
  two recursive calls of qs
}
```

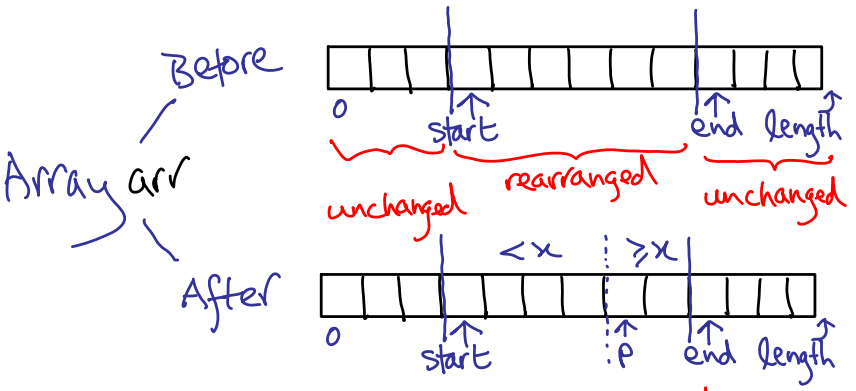
base case

Now know start<end-1

extra parameters to show region

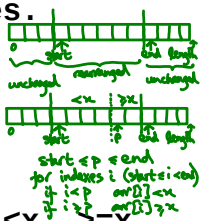


Specifying partition



Specifying partition

```
/**
 * Partition a region of an array.
 * Rearranges elements in region so that small ones
 *   all have smaller indexes than the big ones.
 * Elements outside the region are unchanged.
 * requires: 0 <= start <= end <= arr.length
 * @param arr   array to partition
 * @param start start of region (inclusive)
 * @param end   end of region (exclusive)
 * @param x     pivot - "small" and "big" are <x, >=x.
 * @return      start index (inclusive) of big elements
 *              in region after partition.
 */
private static int partition(int[] arr, int start, int
end, int x){
  Haven't written the code yet!
}
```



p from diagram

Implementing qs

(▬ is unchanged)

partition

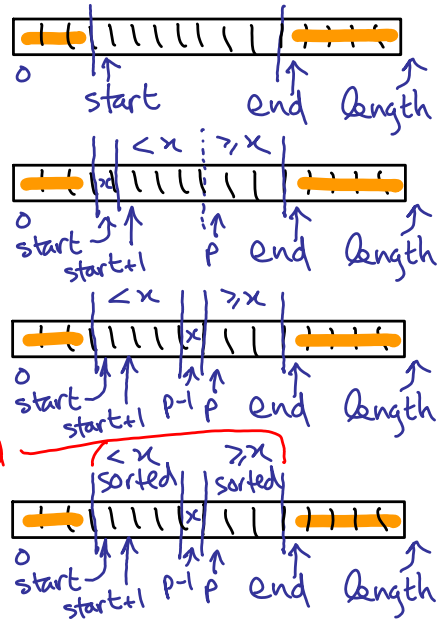
```
p = partition
    (arr, start+1, end, x);
```

swap - arr[start] and arr[p-1]

2 x qs

```
qs(arr, start, p-1)
qs(arr, p, end)
```

all sorted



Preconditions

```
/**
 * Sort a region of an array in ascending order.
 * Elements outside the given region are unchanged.
 * requires: 0 <= start <= end <= arr.length
 * @param arr   array to sort
 * @param start start of region (inclusive)
 * @param end   end of region (exclusive)
 */
private static void qs(int[] arr, int start, int end){
    if (end <= start+1){    //region of length 0 or 1
        return;
    }
    int x = arr[start];
    int p = partition(arr, start+1, end, x);
        //now swap arr[start] with arr[p-1]
    arr[start] = arr[p-1];
    arr[p-1] = x;
    qs(arr, start, p-1);
    qs(arr, p, end);
}
```

Now know start+1<end
∴ precondition for partition ok

Next, start+1 ≤ p ≤ end
∴ start ≤ p-1
preconditions established for qs

At this point ...

defined the method for

We have implemented qs.
Will it work correctly?
Obviously it can't work at all:
we haven't implemented partition.
BUT if we make partition work the way we specified then qs should work.

"Stepwise refinement"

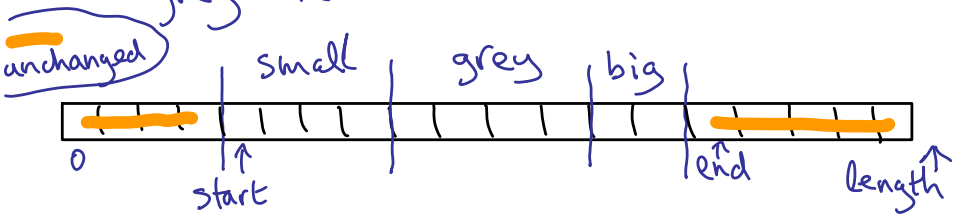
Top-down programming
↑ specify ↑ implement

Can confidently use methods by their specification, even before they're implemented.
Similar to using a library —
use API, not method definitions
Also underlies reasoning for recursion:
recursive call (subcontractor) is specified (contract)

Implementing partition

Homework!

Good example of loop invariant.
Part way through:
some small elements put to one end
some big elements put to other end
"grey" area — unchecked — in middle



Loop invariant

Describe how variables will be used

start ≤ l ≤ r ≤ end

In region for indexes i with start ≤ i < end:

- elements same as originally but rearranged
- if i < l then arr[i] < x
- if i ≥ r then arr[i] ≥ x

Rest of array is unchanged

```
/**
 * Partition a region of an array.
 * Rearranges elements in region so that small ones
 *   all have smaller indexes than the big ones.
 * Elements outside the region are unchanged.
 * requires: 0 <= start <= end <= arr.length
 * @param arr   array to partition
 * @param start start of region (inclusive)
 * @param end   end of region (exclusive)
 * @param x     pivot - "small" and "big" are <x, >=x.
 * @return      start index (inclusive) of big elements
 *              in region after partition.
 */
private static int partition(int[] arr, int start, int end, int x){
  int l = ...;
  int r = ...;
  /* invariant
   * start <= l <= r <= end
   * In region for indexes i with start <= i < end:
   *   elements same as originally but rearranged
   *   if i < l then arr[i] < x
   *   if i >= r then arr[i] >= x
   * Rest of array is unchanged.
   */
  while (...){          ...     }
  ...
}
```

specification (Javadoc)

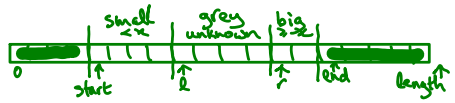
initialize

LOOP INVARIANT

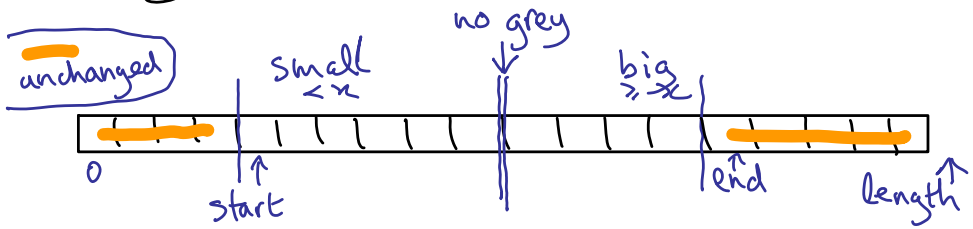
test

loop body

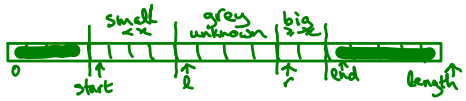
finalize



Finally Stop when ... no grey left



Then partition is complete, ...

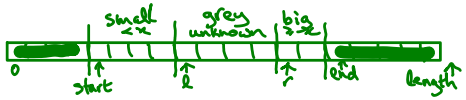


Shows how to implement both test and finalization.

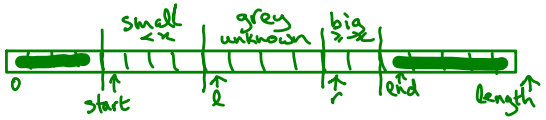
Initially

Initialize

```
int l = ? ;
int r = ? ;
```



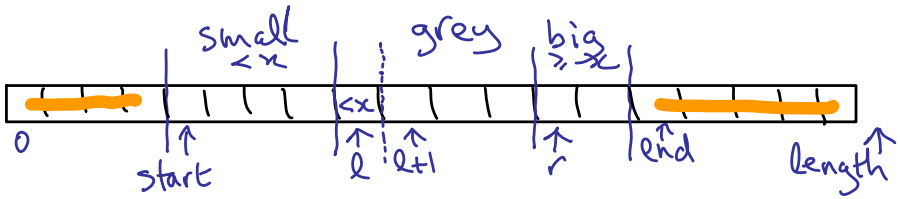
Loop body



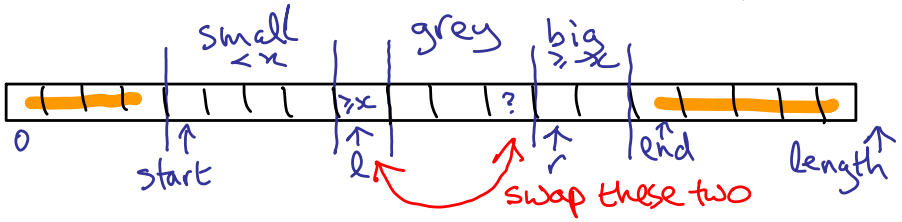
Look at arr [l]

Act according to whether it is small or big

If arr [l] small



If arr [l] big - it is at wrong end of region



Summary

Recursion
- treat recursive calls as subcontractors
- trust them to do what is specified
- use them to help you meet your own contract (specification)
- make sure the contract is clear!
- Remember base case & making progress

Stepwise refinement
- rely on contract of method
- can do this even before method implemented