- review recursion exercises
- more recursion examples
- tail recursion

Next week and homework
- next week: queues
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, sections 15.5, 15.6, 16.3 on queues and priority queues

Lab
- first graded lab assigned in 'Stack applications'. See Canvas for due date

Review last week

- introduced recursion last week:

```
~~~ foo(~~~~)
{
    ~~~;
    foo(~~~~);
    ~~~;
}
```

- saw and did lots of recursion examples

- saw the two Rules for recursion:

  1. there must be a terminating case
  2. each recursive call should be simpler than previously

- recursion exercises assigned last week, for you to write your first recursion methods

Introduction to this week

- finish recursion by

  – reviewing recursion exercises

  – look at a few more examples

- also, look at tail recursion

  – a special case, where we would use iteration, not recursion

- recursion becomes most useful shortly, when processing recursive data structures

  – e.g. linked lists; trees

Review recursion exercises

Objective: review the three recursive methods you wrote as exercises. The more recursion you do, the more you understand

- remember:
  - first write recursive definition of problem
  - then translate definition into a Java method that you can test in BlueJ

1. integer division by recursive subtraction

- e.g. 26 / 8 is 3
  - terminology here : numerator / denominator
  - the algorithm is to recursively subtract denominator from the numerator until the denominator is bigger than the numerator e.g.

```
26 / 8 = 1 + 18 / 8
             1 + 10 / 8
                   1 + 2 / 8
                         0      <-- terminating case
```

  - NOTE: integer division, so 2 / 8 is 0, which is the terminating case. Recursive ascent begins from here, with partial results ascending upwards

- so, recursive definition of problem:

  terminating case: n / d is 0 if (n < d) otherwise
  recursive step: n / d is 1 + (n – d) / d

- then write method directly from recursive definition:

```
int divide(int n, int d)
{
   if (n < d)
         return 0;
*  return 1 + divide(n – d, d);
}
```

  - using `*` here to mark the return address

- will work through an example method call, showing return addresses pushed to the return address stack

  – in BlueJ, say effective method call is:

```
@  divide(26, 8)
```

  – then return address (RA) and local variables pushed to the stack at each method call will be:

call	RA	n	d	returned value
4	*	10	8	?
3	*	18	8	?
2	*	26	8	?
1	@	returned val is ?		

Figure 1 the return address stack at the end of recursive descent

  – this shows the return address stack at the end of recursive descent

- now update the returned value then pop the stack at each method return. This is the recursive ascent

call	RA	n	d	returned value updated
~~4~~	~~*~~	~~10~~	~~8~~	~~0~~
~~3~~	~~*~~	~~18~~	~~8~~	~~1~~
~~2~~	~~*~~	~~26~~	~~8~~	~~2~~
~~1~~	~~@~~	~~returned val is 3~~		

Figure 2 illustrating the recursive ascent

- (remember, strikethrough ~~like this~~ is used here to indicate that a line is actually popped from the stack with each method return)

- notice that the method return value is updated each time, so that partial results ascend upwards as we pop the stack

2. exponentiation x^n by recursive multiplication

- e.g. 3^4 is 3 * 3 * 3 * 3 = 81

  - assume integer only, valid ranges

  - recursive definition is given for this one:

    $x^n = 1$ if $n = 0$
    $x^n = x * x^{n-1}$ if $n > 0$

- write method directly from recursive definition:

```
int exp(int x, int n)
{
   if (n == 0)
         return 1;
   return x * exp(x, n-1);
}
```

3. palindromes by recursion

- e.g. "racecar", "xxxx" are palindromes. Read the same forwards and backwards

  - assume clean input

  - return true if phrase is a palindrome

- algorithm:

```
r   a   c   e   c   a   r
^                       ^
i                       j
```

  - start with index i, j at beginning and end of word, as above

  - if characters are the same, call `palindrome()` recursively with indexes moved e.g.

```
r   a   c   e   c   a   r
    ^               ^
    i               j
```

– and so on…

- hint: what is the terminating condition? When do we stop the recursion?

  – when the indexes are equal, or cross over

  – e.g. indexes meet, for an odd number of chars:

```
r   a   c   e   c   a   r
            ^
            i
            j
```

  – e.g. indexes cross over, for an even number of chars:

```
x   x   x   x
    ^   ^
    j   i
```

- hint: use a string for the characters e.g.

```
boolean palindrome(String phrase, int left, int right)
```

  – remember, use `charAt()` to get a `char` from a `String`

  – when running your method, BlueJ will want to see a `String` value as the first parameter. So double quotes `“”` are required e.g. you would enter `“racecar”`, and so on

- so, recursive definition of problem:

terminating case:	stop recursion when indexes 'are equal or cross over'
recursive step:	palindrome "if outer two characters match" and "remaining string is a palindrome"

- write method directly from recursive definition:

```
boolean palindrome(String phrase, int left, int right)
{
```

```
        if (left >= right)
            return true;
        return phrase.charAt(left) == phrase.charAt(right) &&
            palindrome(phrase, left + 1, right - 1);
    }
```

<u>Summary</u>

- notice how the two Rules of recursion must always be obeyed

- notice how recursive methods drop out of recursive definitions

- from Canvas, ‘Recursion II’ module, Example programs, download, read, run and understand my example programs for these three recursion exercises

More recursion examples
Objective: let's practice some more. The more we do, the better we get

Fibonacci series by recursion

- Fibonacci series is where each element is sum of the 2 preceding elements e.g.

  1 1 2 3 5 8 13 21 34 …

  1^{st} 2^{nd} 3^{rd} 4^{th} … n^{th}

  - the 1^{st} term is the special case starting point

- here's the recursive definition to calculate the n^{th} term in the Fibonacci series

  terminating case: fib(n) is 1 if n <= 2 otherwise
  recursive step: fib(n) is fib(n – 2) + fib(n – 1)

- then the implementation:

```
public int fib(int n)
{
   if (n <= 2)
         return 1;
   return fib(n – 2) + fib(n – 1);
}
```

Recursively sum elements in an array

- recursively sum elements in an integer array named `a[]` containing `n` elements

  - hint using recursion: we are given `n`, so use this to start at RHS of array and add leftwards

  - e.g.

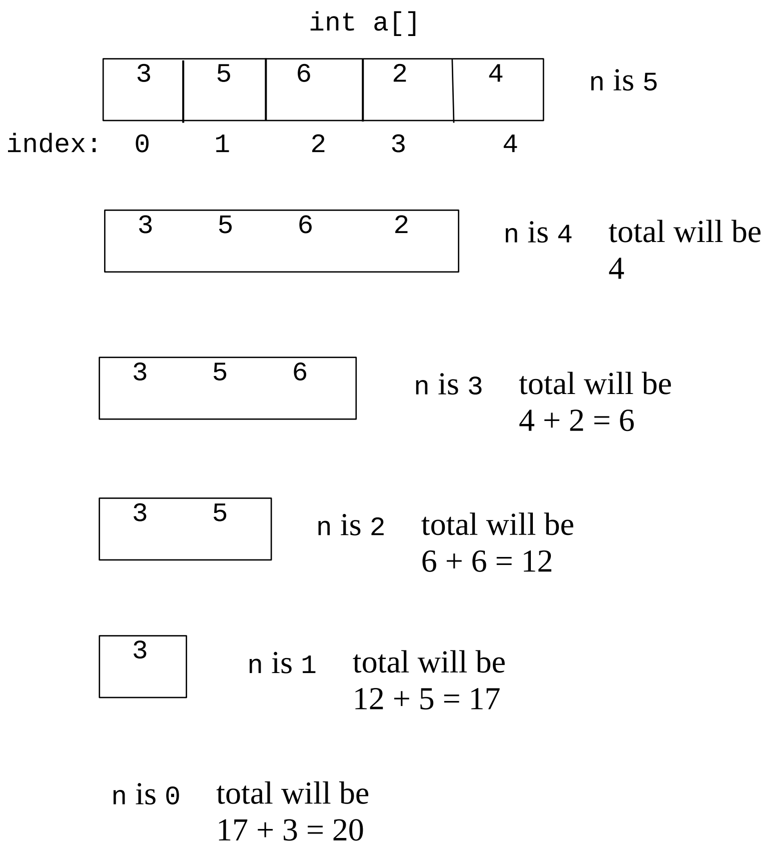

Figure 3 end of recursive descent

– recursive descent ends at the terminating step, when `n` is `0`

- so recursive definition:

terminating case: sum(a[], n) is 0 if n is 0, otherwise
recursive step: is a[n – 1] + sum(a[], n – 1)

- now translate into Java:

```
public int sum(int a[], int n)
{
    if (n == 0)
        return 0;
    return a[n - 1] + sum(a, n - 1);
}
```

– is simpler, clearer with just a regular `for` loop :-)…

– but this is just for practice!

Summary

- with good recursive definition, can write recursive methods quite directly
- must always follow two Rules of recursion
- recursion is more usefully applied to upcoming recursive data structures
- recursion is generally more expensive than an iterative solution
- from Canvas, 'Recursion II' module, Example programs, download, read, run and understand my `Fibonacci` and `Sum array` example programs To run these program:
  - first create an object on the BlueJ workbench by running the default constructor
  - then run the method you want
  - hint: for `Sum array` you need to use correct Java syntax where you enter the value of the first parameter `int[]` in BlueJ. So for example: {3, 5, 6, 2, 4}

Tail recursion

Objective: see a special case of recursion. Is particularly inefficient, so rewrite using loops to avoid it!

<u>Binary search example</u>

- binary search is an efficient way to search an <u>ordered</u> list
  - e.g. find a random target value from an ordered list of 100 numbers:

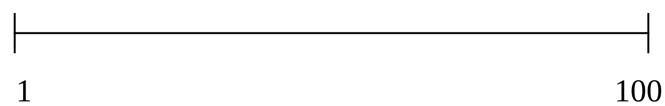

Figure 4 illustrating binary search

  - keep guessing the midpoint, removes half the search space with each question

<u>Non-recursive implementation first</u>

- implement as an array, using 2 indexes `left` and `right`

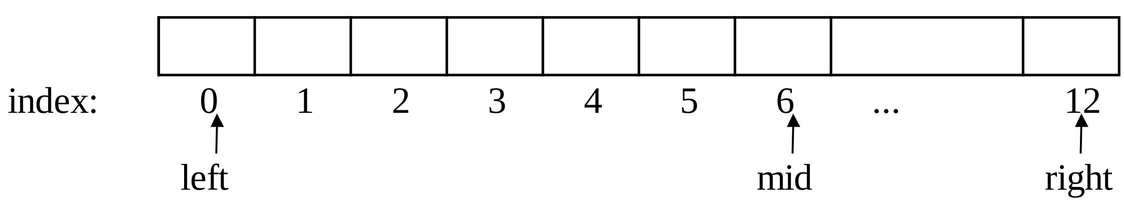

Figure 5 non-recursive implementation

  - calculate `mid` as halfway between `left` and `right`
  - 3 cases:

```
if (list[mid] > target)
   right = mid – 1;
else if (list[mid] < target)
   left = mid + 1;
else
   //found target
```

  - terminate when found or when `left` and `right` cross

- method directly returns index if `target` is found, <u>-1 if not found</u>

- in the method header below, parameter `list` is the list to be searched
- `left` is index of the first item
- `right` is index of last item
- `target` is the item to find

```
public int search(int list[], int left, int right, int target)
{
    int mid;
    while (left <= right) {        //NOTE: <= is necessary
        mid = (left + right) / 2;
        if (list[mid] > target)
            right = mid - 1;
        else if (list[mid] < target)
            left = mid + 1;
        else     //found target
            return mid;
    }
    return -1;
}
```

<u>Now recursive implementation</u>

- now rewrite recursively:

  terminating case: return –1 if left and right crossed over, otherwise
  recursive step: recursively search ½ of the list

```
public int search(int list[], int left, int right, int target)
{
    if (left > right)
        return -1;

    int mid = (left + right) / 2;
    if (list[mid] > target)
        return search(list, left, mid - 1, target);
    else if (list[mid] < target)
        return search(list, mid + 1, right, target);
    else    //found id
        return mid;
}
```

  - (note that Java required the extra returns here, so that every path is guaranteed to return an int)

- works fine. But the problem here is – where do the recursive calls return to?

- answer is to the statement after the call – which is a `return`, effectively the end of the method

<u>Tail recursion is where ”no work is done on recursive ascent”</u>

- this is <u>tail recursion</u>, where no action is taken on return from a recursive call
  - so on recursive ascent, `search()` doesn't do anything with returned value!
  - eventually returned index pops out at first call to method
  - …so we have all the recursive method call overhead, but don't do any work with the intermediate results on the stack!!!
  - expensive!
- e.g. a drawing of standard recursion

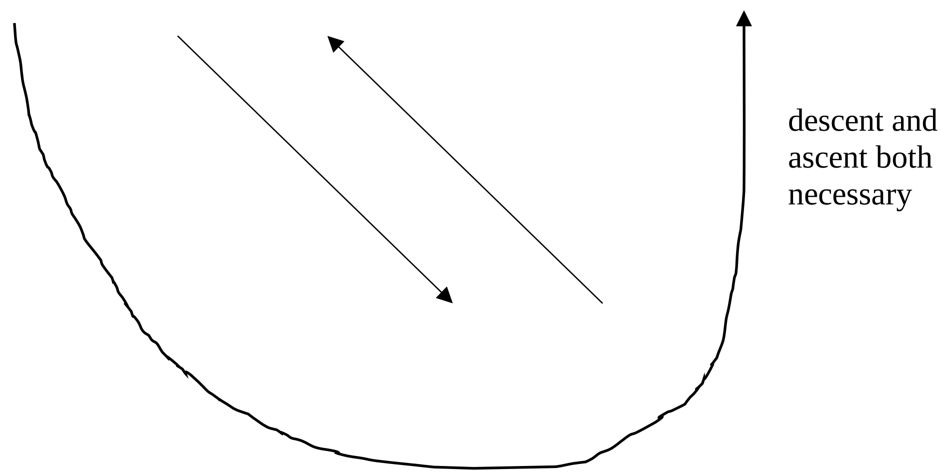

Figure 6 standard recursion, work is done on both the recursive descent and ascent

  - with standard recursion, work is done on both the recursive descent and ascent
- a drawing of tail recursion

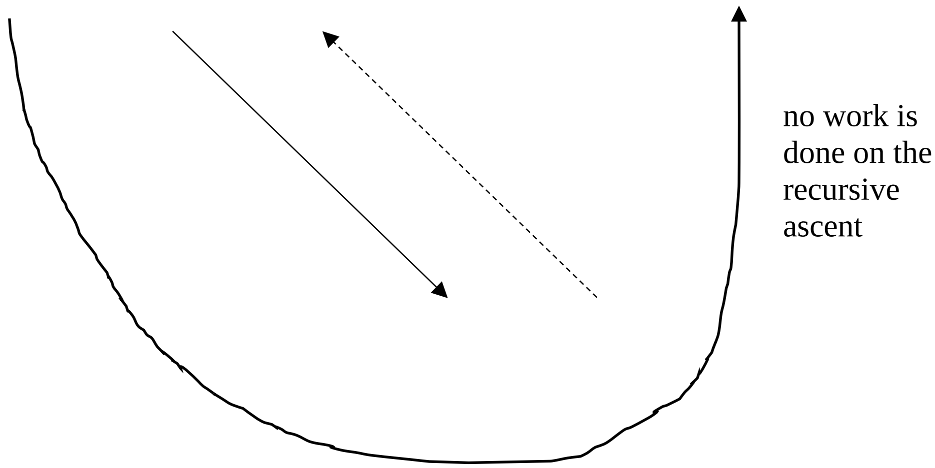

Figure 7 tail recursion, no work is done on the recursive ascent

– with tail recursion, no work is done on the recursive ascent

– expensive, so should re-write iteratively

<u>Summary</u>

- we should rewrite tail recursion as a while loop – can do so, and iteration is more efficient than recursion here

- from Canvas, 'Recursion II' module, Example programs, download, read, run and understand my `Binary search – while loop` and `– recursive` examples

  – (notice how I just hardcoded several tests in `main()`. Is ugly. But effective)

Next week and homework

- next week: queues

- homework by next week:

  - download, run and understand this week's example programs

  - read Horstmann, sections 15.5, 15.6, 16.3 on queues and priority queues

Lab

- first graded lab assigned in 'Stack applications'. See Canvas for due date