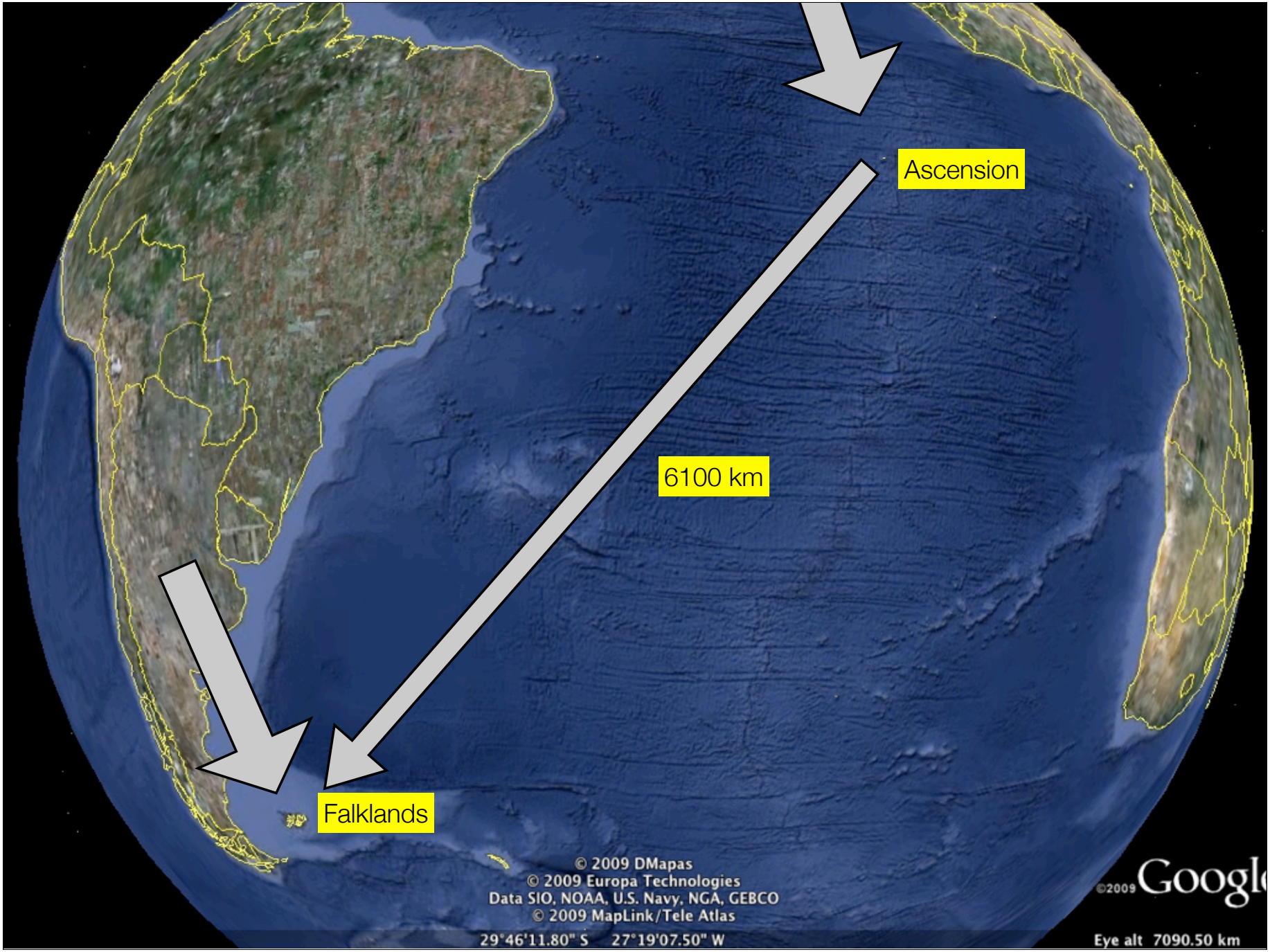
Lecture 2IP65
Week 6

Recursion

Kees Huizing
October 2010


Ascension
6100 km
Falklands
© 2009 DMapas
© 2009 Europa Technologies
Data SIO, NOAA, U.S. Navy, NGA, GEBCO
© 2009 MapLink/Tele Atlas
29°46'11.80" S 27°19'07.50" W
©2009 Google
Eye alt 7090.50 km

Falklands

- One tank load gives 3000 km (3 Mm)
- Tanker flight: 1000 km one way ⇒ deliver 1000 km worth ⇒1000 km return
- f(s): number of tanker flights needed to fly *s Mm* ending with tanks full

```
// pre: ns >=0
// post: returns number of tanker flights to support
//       one ariplane with full tanks after s Mm (1000 km)
int f(int s) {
    if (s == 0) {
        return 0;
    } else {
        return f(s-1) // support for yourself
             + 1          // tanker plane
             + f(s-1)   // support for tanker one way
             + f(s-1);  // support for tanker return
    }
}
```

(Method) Recursion

- the phenomenon that a method calls itself (possibly indirectly)

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

- note: long fac(int n) is better here, because a long can hold bigger numbers than an int; for the rest they are the same

- typical situation: when a *recursive definition* or *recurrence relation* is known
  - in this case: $n! = \begin{cases} 1 & \text{if } n = 0 \\ n((n-1)!) & \text{if } n > 0 \end{cases} \quad \forall n \in \mathbb{N}.$ (Wikipedia)

Recursion: general idea

- Divide problem into:
  - simple base case(s)
    - n=0 ⇒ return 1
  - reduction to same problem of smaller size
    - n>0 ⇒ n * fac(n-1)
- Often reduction to *several instances* of same problem
- Sometimes no direct self call, but via other method: f calls g, g calls f.
  Mutual recursion

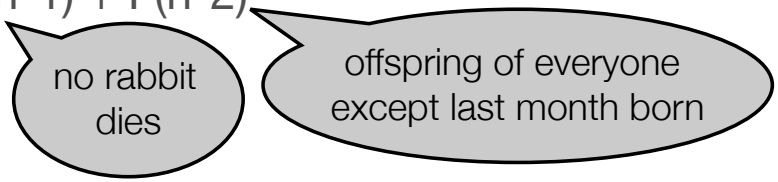
Efficiency

- The son of mr. Bonaccio gets a young pair of rabbits
- After one month, they are fertile and discover the flowers and the bees
- After one more month, mother gives birth to a healthy young pair of rabbits
  - after one month, they are fertile and discover the flower and the bees
  - after one more month, daughter gives birth to a healthy young pair of rabbits
    - after one month, they are fertile and discover the flower and the bees
    - after one more month, daughter gives birth to a healthy young pair of rabbits
- After one more month, mother gives birth to a healthy young pair of rabbits
  - ...

Fibonacci

- Figlio di Bonnacio (Leonardo of Pisa, 1170-1250),

- F(0) = 0
  F(1) = 1
  F(n) = F(n-1) + F(n-2)



- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- a lot of interesting properties

  - F(n)/F(n-1) approaches $(1+\sqrt{5})/2$, which is the *golden ratio* (gulden snede)

Recursive solution

```
//@pre n >= 0
//@post \returns nth Fibonacci number
int fibrec(int n) {
    if (n==0) {
        return 0;
    } else if (n==1) {
        return 1;
    } else { assert n > 1;
        return fibrec(n-1) + fibrec(n-2);
    }
}
```

- leads to many recursive calls
- e.g., fibrec(6) calls fibrec(2) 5 times, fibrec(2) 8 times, fibrec(0) 13 times !
- store intermediate results in global array (or HashMap) and take value instead of call ==> *Dynamic Programming*
- or find a dedicated *iterative* solution

Iterative solution

```
//@pre n >= 0
//@post \returns nth Fibonacci number
int fibit(int n) {
    int a = 0;
    int b = 1;
    int i = 0;

    while(i < n) {
        b = a + b;
        a = b - a;
        i++;
    }
    return a;
}
```

Iterative solution with annotation

```
//@pre n ≥ 0
//@post \returns Fib(n)
int fibit2(int n) {
    int a = 0;
    int b = 1;
    int i = 0;
    //@inv a == Fib(i) && b == Fib(i+1)
    while(i < n) {
        //assert a == A && b == B
        b = a + b;
        //assert a == A && b == A + B
        a = b - a;
        //assert a == B && b == A + B
        //assert a == Fib(i+1) && b == Fib(i+2)
        i++;
    }
    return a;
}
```

More recursive calls

- Binomium: how many ways to choose *k* people from a group of *n*?
  - k = n? choose everyone, only 1 possibility
  - k = 0? leave everybody out, i.e., 1 possibility
  1. choose Isaac and then choose *k-1* people from the rest, a group of *n-1*
  2. don't choose Isaac, so choose *k* people from the rest, a group of *n-1*

- $$\binom{n}{n} = 1$$
  $$\binom{n}{0} = 1$$
  $$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{if } n > k \geq 1$$

```
int binom(int n, int k) {
    if (n==k) {
        return 1;
    } else if (k==0) {
        return 1;
    } else {
        return binom(n-1, k-1) + binom(n-1, k);
    }
}
```

Recursion vs. iteration

- “opposite” of recursive: iterative

```
int fac_iterative(int n) {
  int f = 1;
  while (n>0) {
    f = n * f;
    n = n-1;
  }
  return f;
}
```

recursive solution:

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

- iterative implementations can often be made more efficient

- recursive implementations are often shorter and easier to write (if the problem type is right) and, in the end, easier to understand

Recursion: dynamic storage needs

- Task: print a file of lines backwards, last line first, etc.

  `aap noot mies` becomes `mies noot aap`

- Crab Canon (from: *Gödel, Escher, Bach* by Douglas R. Hofstadter, 1979)

- Problem: storage in an array is not easy, since number of lines is not known on beforehand.
  Other ideas?

Dynamic storage

- Use recursion!

```
void invert() {
    if ( ! scanner.hasNext()) {
        // do nothing for empty file
    } else {
        String line;
        line = scanner.nextLine();
        invert();
        System.out.println( line );
    }
}
```

- every call of *invert* gives a new variable *line*

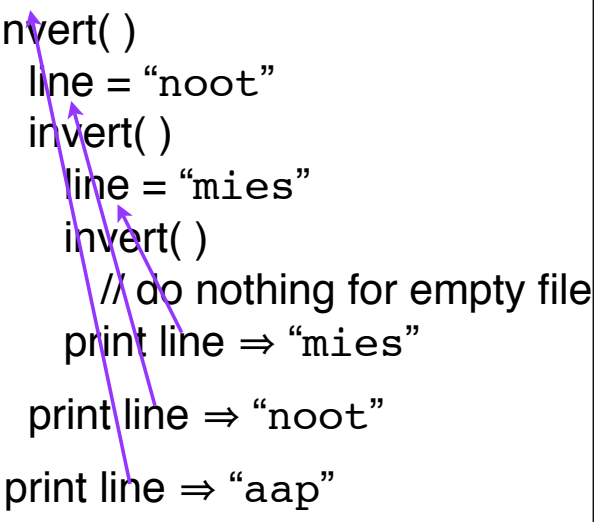
- there are more possibilities than recursion to use dynamic storage
  more about this later

Example

```
aap
noot
mies
```

```
mies
noot
aap
```

```
invert( )
  line = "aap"
  invert( )
    line = "noot"
    invert( )
      line = "mies"
      invert( )
        // do nothing for empty file
      print line ⇒ "mies"
    print line ⇒ "noot"
  print line ⇒ "aap"
```



Recursive design

- Typical errors:
  - no or incomplete base cases
  - not always reduction in size

  } risk of no termination

  - too many steps in size reduction part: keep it as simple as possible!

- Formulate carefully what the recursive method solves then use this in the code that calls the method. Pre/post helps