TDA 548: Grundläggande Programvaruutveckling

Föreläsning 1, vecka 7:
Rekursion

Magnus Myréen

Chalmers, läsperiod 1, 2016-2017

Nytt: Extra labbtillfälle för **Grupp B**

(för att grupp Bs labbtider har på senaste tiden kolliderat med GruDat)

E studion 13-15 onsdag 12 oktober 2016
E studion 15-17 torsdag 13 oktober 2016

Feedback

Hur fungerar labbarna?

Hur fungerar övningarna?

Fungerar övning före föreläsning?

Idag

Rekursion

- fakulteten
- exponentiering
- binary search
- Fibonacci
- 91 funktionen
- quick sort

men också:

- jämförelse med iteration
- att mäta tid i Java
- roliga problem
- live programmering

Nästa gång: mera objektorientering, i detta fall: arv, subtyper, API …

Att tänka rekursivt…

Hur långt är tåget?

Rekursivt svar:

om det inte finns tågvagnar: då är dess längd 0

i övriga fall: längden är 1 + längden av resten av tåget

samma fråga som ursprungligen, men med mindre indata (input)


metro
Woman spotted yesterday
reading today's paper

Fakulteten

Iterativt

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ 1 \times 2 \times 3 \times \ldots \times n & \text{if } 1 \leq n \end{cases}$$

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

Induktivt eller rekursivt

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ n \times (n-1)! & \text{if } 1 \leq n \end{cases}$$

Induktiva definitioner

använder självreferens, t.ex.

En kö är antingen tom
eller en person följt av en kö.

n! är 1, om n är 0 eller 1,
eller n × (n-1)! om n är större.

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ n \times (n-1)! & \text{if } 1 \leq n \end{cases}$$

Induktiva definitioner består av

1. Ett basfall
2. Ett fall där definitionen (vanligen) minskar problemets storlek och refererar till sig själv.

Induktionsbevis och rekusion

Induction poof:

to prove that a statement P(n) holds for all integers n

Base case: Prove P(0) (or P(small number))

Assumption:

assume that P(n-1) is true (or P(n), P(n/2)... is)

Induction step:

prove that P(n-1) implies P(n) for all $n \geq 0$

Recursive computations:

Base case: Solve the problem on inputs of size 0 (or 1 or some small instance)

Assumption: assume you can solve the problem on input size less than n (n-1, $n/2$, $n/4$...)

Induction step: Show how you can solve it also on inputs of size n for all $n \geq 1$.

Fakulteten (forts)

Iterativt

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ 1 \times 2 \times 3 \times \ldots \times n & \text{if } 1 \leq n \end{cases}$$

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

Induktivt eller rekursivt

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ n \times (n-1)! & \text{if } 1 \leq n \end{cases}$$

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
if (1 < 1) {
  return 1;
} else {
  return 1 * fac(0);
}
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
if (1 < 1) {
  return 1;
} else {
  return 1 * fac(0);
}
```

```
1
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
if (1 < 1) {
  return 1;
} else {
  return 1 * fac(0);
}
```

1

rekursionen går ner

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
if (1 < 1) {
  return 1;
} else {
  return 1 * fac(0);
}
```

```
1
```

rekursionen kommer upp

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 * fac(1);
}
```

```
if (1 < 1) {
  return 1;
} else {
  return 1 *  1;
}
```

rekursionen kommer upp

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 * fac(2);
}
```

```
if (2 < 1) {
  return 1;
} else {
  return 2 *  1;
}
```

rekursionen kommer upp

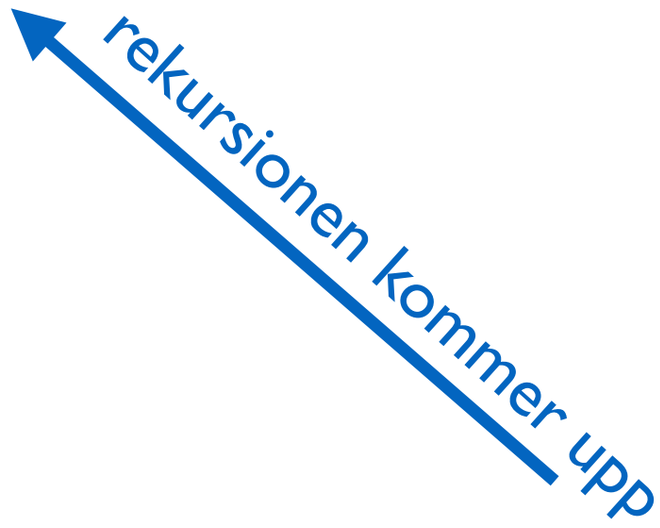
```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 * fac(3);
}
```

```
if (3 < 1) {
  return 1;
} else {
  return 3 *  2;
}
```



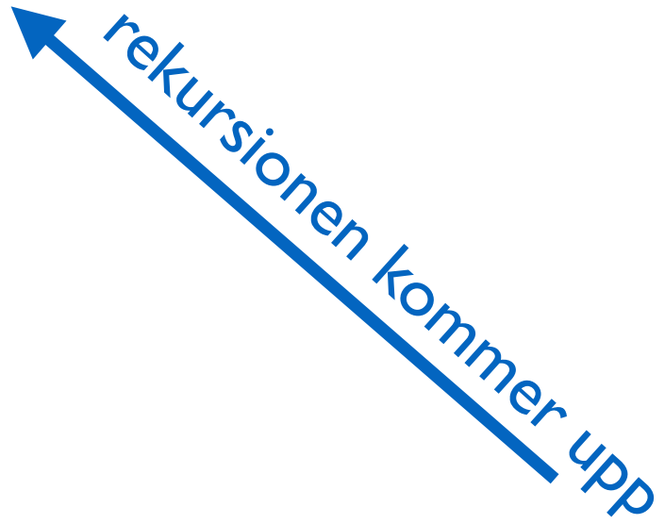

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

Evaluering

```
System.out.println(fac(4));
```

```
if (4 < 1) {
  return 1;
} else {
  return 4 *  6;
}
```

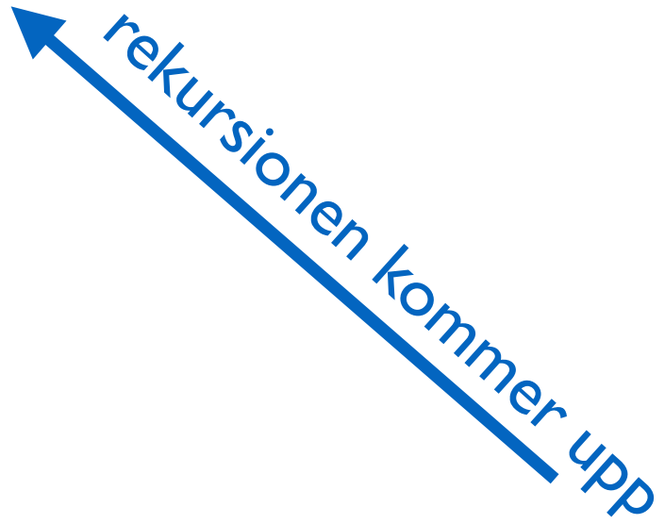
```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```



Evaluering

```
System.out.println( 24 );
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```



Sammanfattning av beräkningen

```
System.out.println(fac(4));
```

```
fac(4) =
4 * fac(3) =
4 * 3 * fac(2) =
4 * 3 * 2 * fac(1) =
4 * 3 * 2 * 1 * fac(0) =
4 * 3 * 2 * 1 * 1 =
4 * 3 * 2 * 1 =
4 * 3 * 2 =
4 * 6 =
24
```

```
public static int fac(int n) {
  if (n < 1) {
    return 1;
  } else {
    return n * fac(n-1);
  }
}
```

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ n \times (n-1)! & \text{if } 1 \leq n \end{cases}$$

Ett test program

```
public class Fac {

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

  public static void main(String[] args) {
    System.out.println("fac(1)  = " + fac(1));
    System.out.println("fac(4)  = " + fac(4));
    System.out.println("fac(20) = " + fac(20));
  }

}
```

Utskrift:

```
fac(1)  = 1
fac(4)  = 24
fac(20) = -2102132736
```

umm… Vad hände här?

svar: värdet blev för stort.

Obs. overflow har inget med rekursionen att göra.

Vi kan använda **BigInteger**

```
import java.math.BigInteger;

public class BigFac {

  public static BigInteger fac(BigInteger n) {
    if (n.equals(BigInteger.ZERO)) {
      return BigInteger.ONE;
    } else {
      return n.multiply(fac(n.subtract(BigInteger.ONE)));
    }
  }

  public static void main(String[] args) {
    System.out.println("fac(1)  = " + fac(new BigInteger("1")));
    System.out.println("fac(4)  = " + fac(new BigInteger("4")));
    System.out.println("fac(20) = " + fac(new BigInteger("20")));
    System.out.println("fac(50) = " + fac(new BigInteger("50")));
  }

}
```

Utskrift:

```
fac(1)  = 1
fac(4)  = 24
fac(20) = 2432902008176640000
fac(50) = 30414093201713378043612608166064768844377641568960512000000000000
```

Uppgift

Skriv ett program som beräknar hur många nollor det finns i ett decimal tal.

Idé: I ett negativ tal finns det like många nollor som i motsvarande positiva tal.

Om talet är 0, finns det en nolla i talet.

Om talet är positivt och < 10, finns det ingen nolla.

I övriga fall, dvs. positivt och > 10

om talet mod 10 är 0, då finns det en nolla + nollorna i talet / 10

annars: nollorna i talet / 10

Lösning

```
public class Z {

    public static int k(int n) {
        if (n < 0) {
            return k(0 - n);
        } else if (n == 0) {
            return 1;
        } else if (n < 10) {
            return 0;
        } else if (n % 10 == 0) {
            return 1 + k(n / 10);
        } else {
            return k(n / 10);
        }
    }

    public static void main(String[] args) {
        System.out.println("nollor(-1050) = " + k(-1050));
    }

}
```

Uppgift

Skriv en *rekursiv* metod som vänder om en sträng,
dvs rev(”Hello”) bör bli "olleH".

Uppgift

Skriv en *rekursiv* metod som vänder om en sträng,
dvs rev(”Hello”) bör bli "olleH".

```
public static String rev(String str) {
    if (str.length() == 0) {
        return str;
    } else {
        return rev(str.substring(1)) + str.substring(0,1);
    }
}
```

Ofta går det att skriva om rekursionen som en iterativa metod … *men inte alltid.*

Några animationer

http://impedagogy.com/wp/wp-content/uploads/2014/08/Recursionelevator.gif

http://i.imgur.com/P1Chi6u.gif

http://media.giphy.com/media/VLNG6LOKeURfG/giphy.gif

Roligt med rekursion…

Vad räknar McCarthy's funktion m?

$$m(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ m(m(n + 11)) & \text{if } n \leq 100 \end{cases}$$

Samma i Java:

```
public static int m(int n) {
  if (n > 100) {
    return n - 10;
  } else {
    return m(m(n + 11));
  }
}
```

McCarthy's funktion

```
public class M {

  public static int m(int n) {
    if (n > 100) {
      return n - 10;
    } else {
      return m(m(n + 11));
    }
  }

  public static void main(String[] args) {
    for (int j=0; j<100; j++) {
      System.out.println("m(" + j + ") = " + m(j));
    }
  }

}
```

McCarthy's funktion

```
public class M {

  public static int m(int n) {
    if (n > 100) {
      return n - 10;
    } else {
      return m(m(n + 11));
    }
  }

  public static void main(String[] args) {
    for (int j=0; j<100; j++) {
      System.out.println("m(" + j + ") = " + m(j));
    }
  }

}
```

Utskrift:

```
m(0) = 91
m(1) = 91
m(2) = 91
m(3) = 91
m(4) = 91
m(5) = 91
m(6) = 91
m(7) = 91
m(8) = 91
m(9) = 91
m(10) = 91
m(11) = 91
m(12) = 91
m(13) = 91
m(14) = 91
m(15) = 91
m(16) = 91
m(17) = 91
m(18) = 91
…
m(99) = 91
```

Detta är McCarthy's 91 funktion.

http://en.wikipedia.org/wiki/McCarthy_91_function

Varför blir det 91?

Slutar beräkningen?

Ibland blir input större inför rekursionen.

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ t(3 \times n + 1) & \text{if } n \text{ is odd} \\ t(n/2) & \text{if } n \text{ is even} \end{cases}$$

Samma i Java:

```
public class T {
  public static int t(int n) {
    System.out.print(n + " ");
    if (n < 1) { System.exit(1); }
    if (n == 1) {
      return 1;
    } else if (n % 2 == 1) {
      return t(3 * n + 1);
    } else {
      return t(n / 2);
    }
  }
  public static void main (String[] args) {
    t(Integer.parseInt(args[0]));
  }
}
```

Utskrift:

```
$ javac T.java
$ java T 125
125 376 188 94 47 142 71 214 107
322 161 484 242 121 364 182 91
274 137 412 206 103 310 155 466
                              395
                               58
                              566
283 850 425 1276 638 319 958 479
1438 719 2158 1079 3238 1619 4858
2429 7288 3644 1822 911 2734 1367
4102 2051 6154 3077 9232 4616
2308 1154 577 1732 866 433 1300
650 325 976 488 244 122 61 184 92
46 23 70 35 106 53 160 80 40 20
10 5 16 8 4 2 1
```

Koden var i upp vid **9232**!

Slutar beräkningen?

Ibland blir input större inför rekursionen.

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ t(3 \times n + 1) & \text{if } n \text{ is odd} \\ t(n/2) & \text{if } n \text{ is even} \end{cases}$$

Slutar beräkningen för ***alla*** utgångsvärden av n?

Svar: ingen vet! Det är en *open research question*…

Vi kan testa litegrann… men hur vet vi att räkningen slutar för ***alla*** input?
(Bör göras med BigInteger.)

Flera animationer

https://i.chzbgr.com/maxW500/7732713984/h2E032A91/

http://d5lx5634mkgoi.cloudfront.net/wp-content/uploads/2010/02/Slide190.gif

http://33.media.tumblr.com/142251408fc0a2446612ef96e7cffbe7/tumblr_moltlz30w01q

Lite mera seriöst igen…

Beräkna exp

Att beräkna x^n:

Iterativt, idé $x^n = x \times x \times \ldots \times x$

```
public static double power1(double x, int n) {
    double tmp = 1.0;
    for (int i=1; i<=n; i++) {
        tmp = tmp * x;
    }
    return tmp;
}
```

multiplicerar **x** in i **tmp**, **n** gånger.

Rekursivt, idé $x^n = x \times x^{(n-1)}$

```
public static double power2(double x, int n) {
    if (n == 0) {
        return 1.0;
    } else {
        return x * power2(x,n-1);
    }
}
```

… gör samma sak.

Samma algoritm!

Beräkna exp (forts)

Bättre rekursion?

Obs: $x^n = x^{n/2} \times x^{n/2}$ ***eller ännu bättre:*** $x^n = (x \times x)^{n/2}$

Mera precis:

$$x^{2\times n} = (x \times x)^n$$
$$x^{2\times n+1} = (x \times x)^n \times x$$

I Java:

```
public static double power3(double x, int n) {
    if (n == 0) {
        return 1.0;
    } else {
        if (n % 2 == 0) {
            return power3(x * x, n / 2);
        } else {
            return power3(x * x, n / 2) * x;
        }
    }
}
```

heltalsdivision,
dvs blir heltal

Jämförelse

```
power2(9,3.0) =
3.0 * power2(8,3.0) =
3.0 * 3.0 * power2(7,3.0) =
3.0 * 3.0 * 3.0 * power2(6,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * power2(5,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * power2(4,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * power2(3,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * power2(2,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * power2(1,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * power2(0,3.0) =
3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 3.0 * 1.0 =
19683.0
```

```
power3(9,3.0) =
power3(4,9.0) * 3.0 =
power3(2,81.0) * 3.0 =
power3(1,6561.0) * 3.0 =
power3(0,6561.0) * 6561.0 * 3.0 =
1.0 * 6561.0 * 3.0 =
19683.0
```

bättre algoritm!

… men vilken tar längre tid att beräkna?

Att mäta tid i Java

I `java.lang.System` finns metoder:

`static long currentTimeMillis()`

Returns the current time in milliseconds.

`static long nanoTime()`

Returns the current value of the *most precise available* system timer, in nanoseconds.

Exempel användning:

```
long start = System.currentTimeMillis();
gör arbete
long duration = System.currentTimeMillis() - start;
System.out.println("Duration: " + duration + " ms");
```

Att mäta tid på power funktionerna

```
public static void main(String[] args) {

    int reps = 10000;
    long start, duration;

    // test power1
    start = System.currentTimeMillis();
    for (int i = 0; i < reps; i++) {
        power1(2.0, 5000);
    }
    duration = System.currentTimeMillis() - start;
    System.out.println("Duration1: " + duration + " ms");

    // test power3
    start = System.currentTimeMillis();
    for (int i = 0; i < reps; i++) {
        power3(2.0, 5000);
    }
    duration = System.currentTimeMillis() - start;
    System.out.println("Duration3: " + duration + " ms");

}
```

```
$ java Pow
Duration1: 93 ms
Duration3: 3 ms
```

Hitta element i array

Hur kollar man om ett element finns i en array?

Hitta element i array

Hur kollar man om ett element finns i en array?

Enkel kod:

```
public static boolean lookup(int k,int[] arr) {
    for (int i=0; i<arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
        return false;
}
```

… om arrayn *sorterad* går det att göra *mycket snabbare*!

Binärsökning

Exempel: Kolla om värdet 5 finns i arrayn?

Idé: Inspektera elementet i mitten av arrayn:

Är mitten elementet det vi söker?

Om mitten elementet är mindre kan vi "kasta bort" den högra sidan.

vi vet ju att arrayn är sorterad.

… och så kan vi upprepa sägen ovan tills arrayn är tom.

Binärsökning (forts)

Som rekursiv Java kod:

```
public static boolean bsearch1(int k,int[] arr,int b,int e) {
    // System.out.println("b,e = " + b + "," + e);
    if (e <= b) { return false; }
    int i = (b + e) / 2; // mitten
    if (arr[i] == k) {
        return true;
    } else if (arr[i] < k) {
        return bsearch1(k,arr,i+1,e);
    } else /* k < arr[i] */ {
        return bsearch1(k,arr,b,i);
    }
}

public static boolean bsearch(int k,int[] arr) {
    return bsearch1(k,arr,0,arr.length);
}
```

Binärsökning (forts)

Samma som iterativa kod:

```
public static boolean bsearchIt(int k,int[] arr) {
    int b = 0;
    int e = arr.length;
    while (b < e) {
        int i = (b + e) / 2; // mitten
        if (arr[i] == k) {
            return true;
        } else if (arr[i] < k) {
            b = i+1;
        } else /* k < arr[i] */ {
            e = i;
        }
    }
    return false;
}
```

Fibonacci

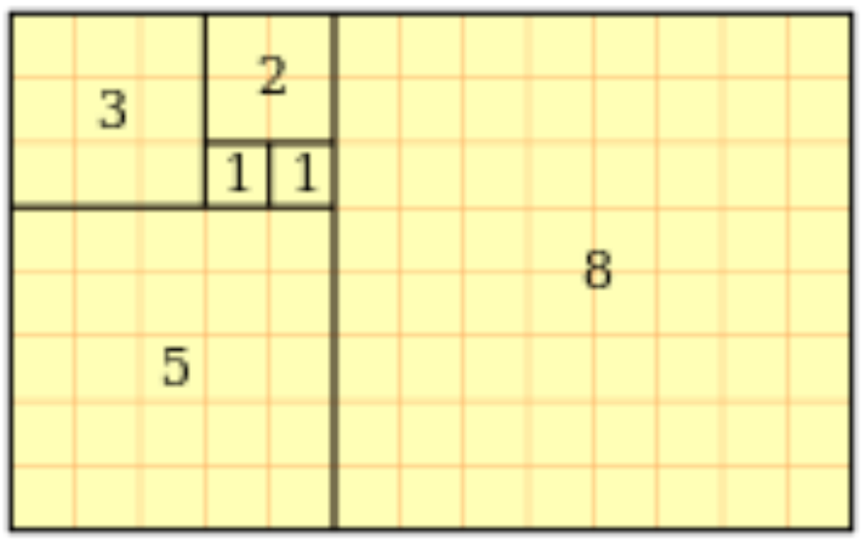
En känd rekursiv funktion:

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } 2 \leq n \end{cases}$$

Värden:

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
```

Som bild:



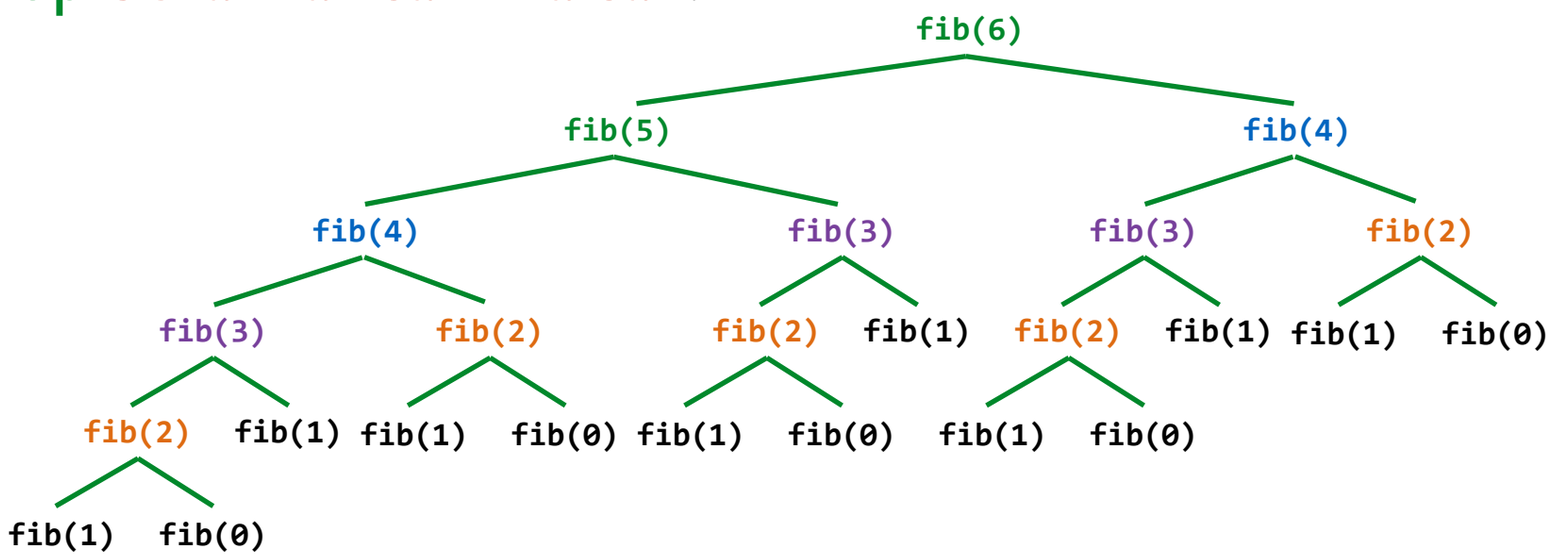
Fibonacci

Rekursion leder inte alltid till bra kod!

I Java:

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

Flera rekursiva anrop beräknar samma sak:



Iterativ version av Fibonacci?

Rekursion leder inte alltid till bra kod!

I Java:

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

Uppgift: skriv en iterativ version av Fibonacci.

Rekursion i typerna

En annan form av rekursion:
Drawable3D är rekursiv genom typerna

```
public interface Drawable3D {

    public Drawable3D rotate(double xy_angle, double yz_angle);
    public void draw(Graphics g, int width, int height);
    public Drawable3D translate(double x, double y, double z);

}
```

Man kan skriva:

```
public class Many3D implements Drawable3D {

    private Many3D p1;
    private Many3D p2;
...
```

Oändligt? Nej, det kan ju vara ***null***.

Iterativ eller rekursiv?

Iterativ kod är snabbare, *speciellt i Java*

… men vissa algoritmer går inte att skriva som iteration.

… nja, det går nog, men det kan vara mycket svårt!

Uppgift: implementera QuickSort

https://www.youtube.com/watch?v=ywWBy6J5gz8

Uppgift: implementera QuickSort

byter plats på två element

```
public static void swap(int[] array, int i, int j) {
    int a_i = array[i];
    int a_j = array[j];
    array[i] = a_j;
    array[j] = a_i;
}
```

flyttar alla element < pivot till vänster, resten till höger

```
public static int partition(int[] array, int pivot, int b, int e) {
    while (b < e) {
        if (array[b] < pivot) {
            b = b+1;
        } else {
            swap(array,b,e-1);
            e = e-1;
        }
    }
    return b;
}
```

returnerar gränsen mellan < och >= element

Uppgift: implementera QuickSort

```
public static void qsort(int[] array, int b, int e) {
    int nElements = e - b;
    if (nElements < 2) { return; }
    int pivot = array[b];
    int split = partition(array,pivot,b+1,e);
    swap(array,b,split-1);
    qsort(array,b,split-1);
    qsort(array,split,e);
}
```

slutar genast ifall det finns färre än två element, för då är vi färdiga

väljer en pivot

arrangerar om elementen

flyttar pivot elementet till mitten

sorterar rekursivt båda partitionerna