# Rekursiva algoritmer, referenser och minneshantering

{

Vi ska:

- Kika på rekursiva mönster
- Göra en binär sökfunktion
- Skriva en rekursiv fibonaccialgoritm
- Undersöka dynamiska alternativ
- Lära oss hur callstacken fungerar
- Förstå minne och referenser i Java
- Prata om tidskomplexiteter och variablers livslängd
- Förstå basfall

}

Mail:
carl-johan.johansson@im.uu.se

# Vad en algoritm är

- Finns förvånansvärt få definitioner av en algoritm som alla är överens om, men i grund och botten är de metoder som försöker lösa ett specifikt problem
- En algoritm beskriver en högre nivå av abstraktion än hårdvaran som den körs på. Dvs inte fysiska operationer för en maskin av typen “lyft en spak”, “sänk en nål”, osv
- En algoritm beräknar någonting och kan producera nya resultat beroende på sin indata
- Algoritmer kan ofta anpassa sitt beteende baserat på inputen
- ~~“En algoritm är som ett recept”~~ är därför egt. en dålig analogi

# Varför lär vi oss algoritmer?

- Skillnad mellan att vara bekant med ett programspråk och att förstå det
- Om man vill programmera behöver man en djupare förståelse för vad som händer under lagren av abstraktion
- Lärdomarna är till stor del generella. Church-Turing-hypotesen gör gällande att alla datorer i grund och botten är likadana
- Det här innebär att datastrukturer och algoritmer ofta gör samma saker: de är något vi använder för att manipulera minne, och minne ser likadant ut överallt
- Arrayer, listor, sorteringar, osv finns i alla språk

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”

Linus Thorvalds

# Vad kännetecknar ett programmerings-språk?

- Ett programmeringsspråk är ett lager av abstraktion som vi använder för att ge instruktioner till en dator

- Kontrollflöden definierar programmeringgspråk. De är strukturer som manipulerar programflödet på något vis, och påverkar därmed även tidskomplexiteten. Exempel på kontrollflöden inkluderar:

| | |
|---|---|
| Selektion | If-satser |
| Iteration | for-loopar, forEach-loopar, while-loopar |
| Felhantering | Try-catch, undantagsfel |
| Metodanrop | Utomstående kodblock som körs |

- Rekursion är ytterligare en typ av kontrollflöde. Men vad är det?

“To understand recursion, one must first understand recursion.”

Stephen Hawking

# Vad är rekursion?

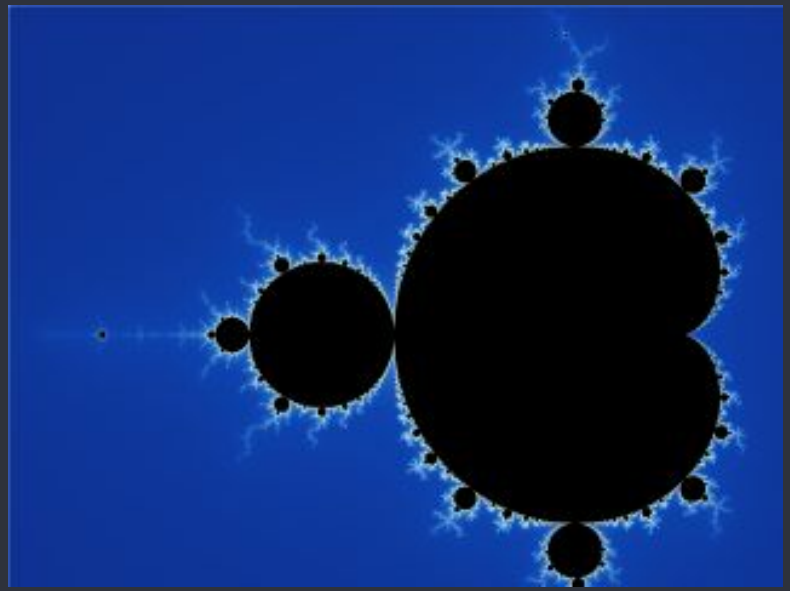
- Metod som anropar sig själv
- Självrefererande kod
- Inte bara ett koncept inom programmering
- Rekursion uppstår när någonting definieras i termer av sig självt

```java
public static String inputName(Scanner scanner)
{
    System.out.println("Skriv in ditt namn: ");
    String input = scanner.nextLine();

    if(input.isEmpty())
        return inputName(scanner);
    else
        return input;
}
```

Exempel: inputName() anropar sig själv ovan

# Naturlig rekursion



## Mandelbrotfraktalen

- Benoit B. Mandelbrot
- Arbetade på IBM på 80-talet
- Upprepar sig i all oändlighet

# Fibonaccisekvensen

n: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ....

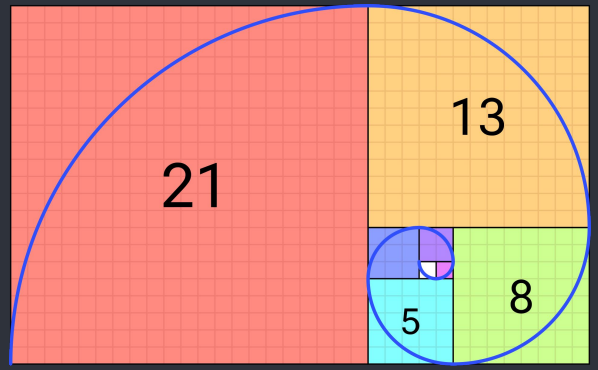
$$F(n) = F(n-1) + F(n-2)$$

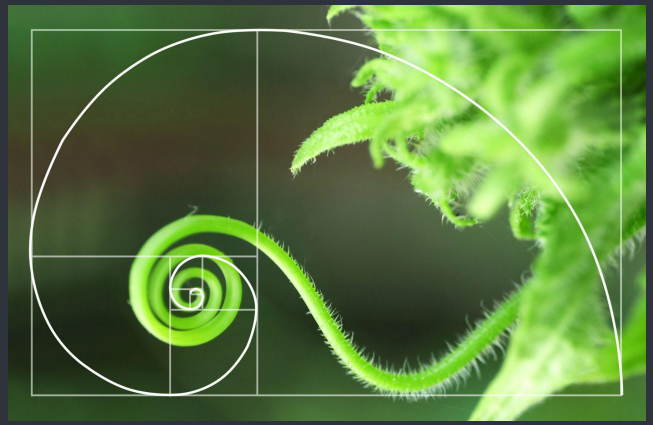
$$F(9) = F(8) + F(7) = 13 + 8 = 21$$

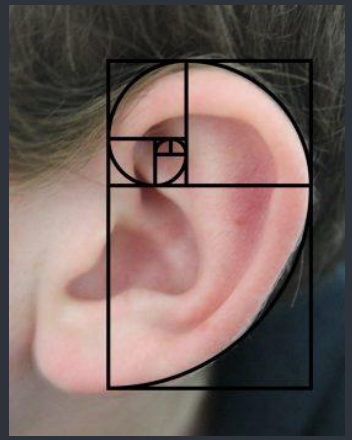
– Varje nytt tal i sekvensen är produkten av de två föregående

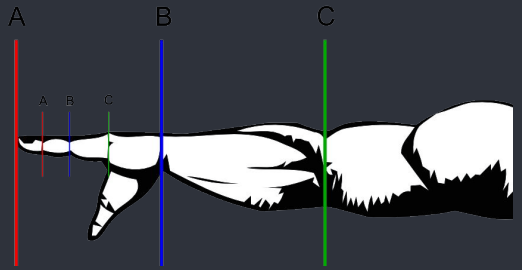
Fraktaler.java

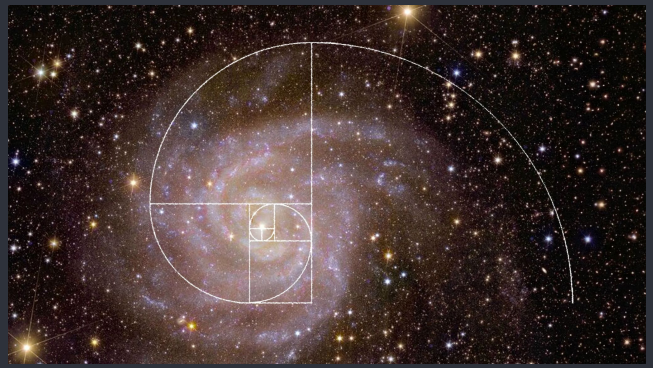
Fibonacci.java







Fibonacci.java

Celler.java

# Biologisk rekursion

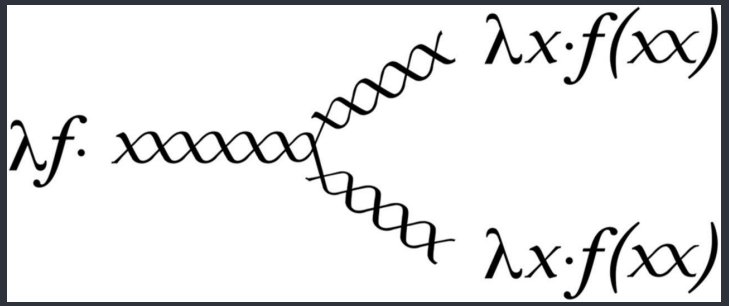
Celldelning:

- Sker naturligt i kroppen
- Varje cell duplicerar sig själv och upprepar sedan samma mönster

Plantor:

- Phyllotaxis: mönster i hur blad och frön organiseras
- Fibonaccisekvensen optimerar packning och minimerar överlapp

"Replication in biological systems is intuitively similar to recursion in computational systems."



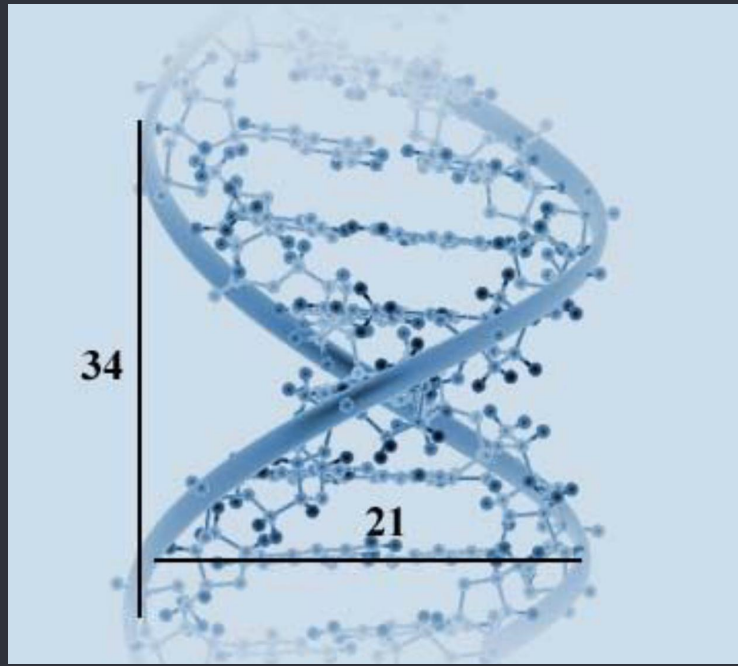

Biologisk Replikationsgaffel: DNA-delning i två nya sekvenser

Celler.java

GoldenRatio.java

# Det gyllene snittet (golden ratio)

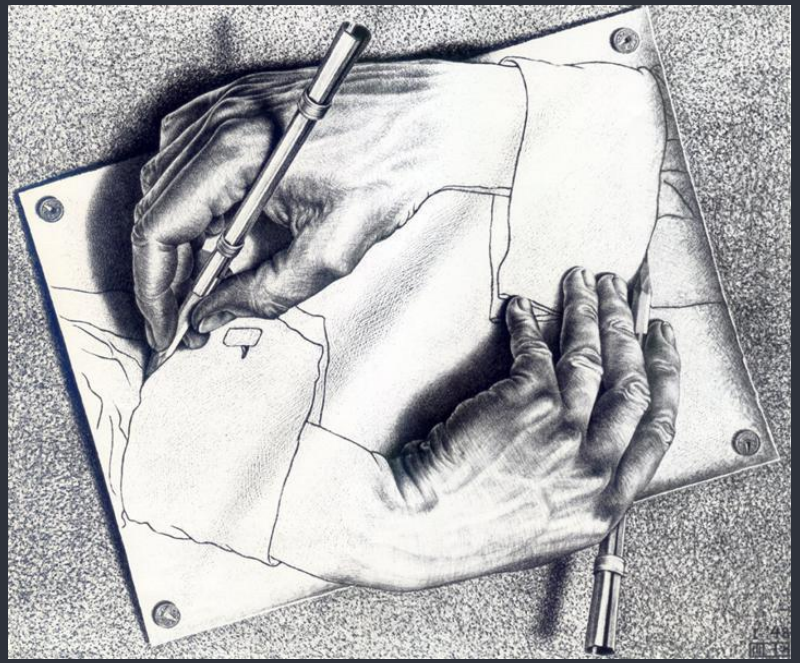
- Längden på en kurva i en DNA-spiral är 34 Ångström. Bredden på själva spiralen är 21 Ångström
- Båda är fibonaccital!
- Förhållandet mellan fibonaccitalen närmar sig ~1.618 ju större numren är
- Kallas det gyllene snittet
- Berömd ratio som finns överallt: i naturen, i konsten, osv

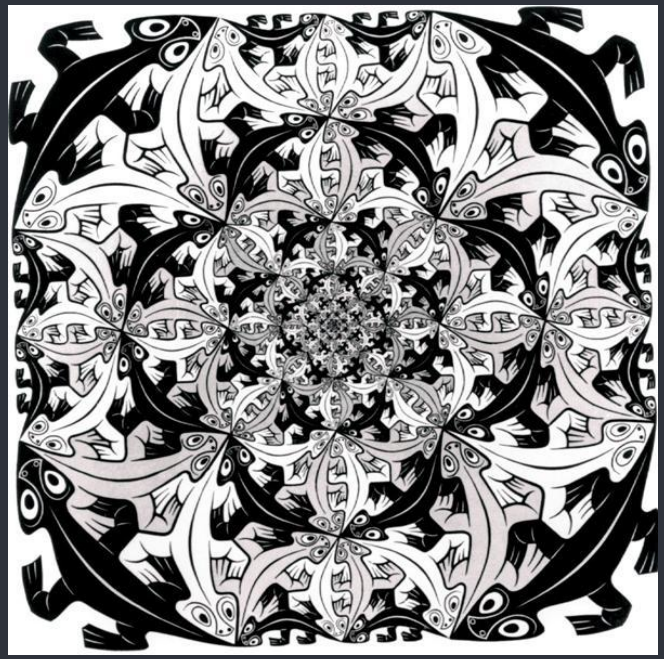


GoldenRatio.java

MCEscher.java

# Rekursiv konst



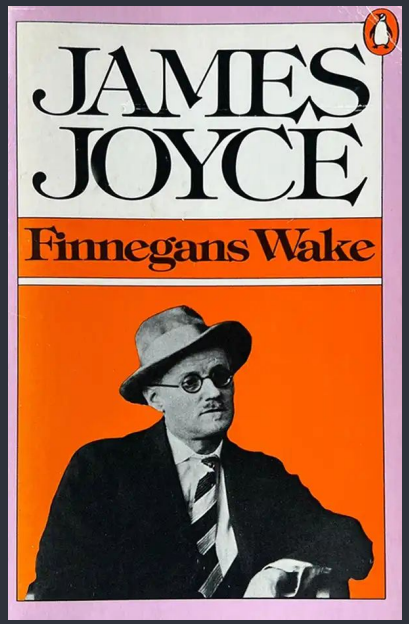


Filmtips: MC Escher - Journey to Infinity https://www.imdb.com/title/tt8297550

MCEscher.java

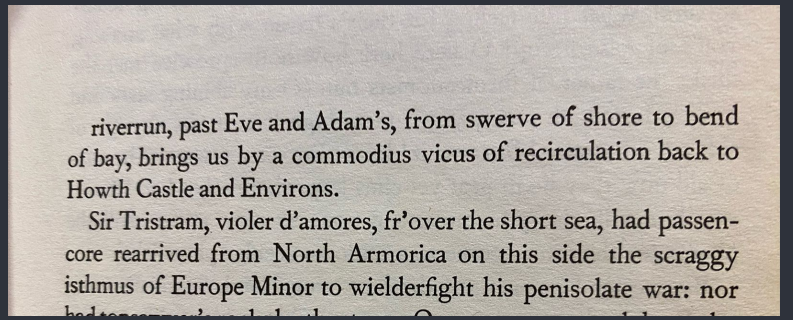
FinnegansWake.java

# Rekursiv litteratur




– Börjar och slutar i samma mening

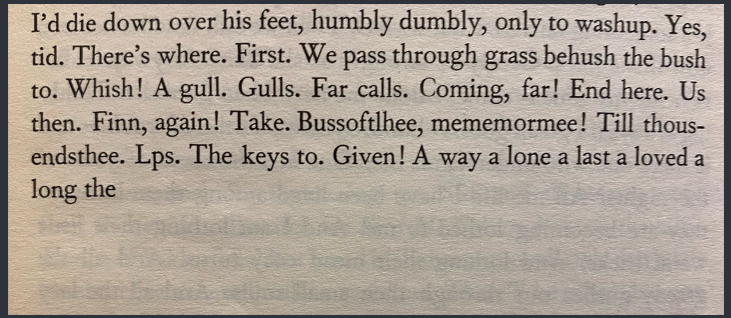
"If it took me seventeen years to write it then a reader should take seventeen years to read it"



riverrun, past Eve and Adam's, from swerve of shore to bend of bay, brings us by a commodius vicus of recirculation back to Howth Castle and Environs.

Sir Tristram, violer d'amores, fr'over the short sea, had passencore rearrived from North Armorica on this side the scraggy isthmus of Europe Minor to wielderfight his penisolate war: nor

Första sidan



I'd die down over his feet, humbly dumbly, only to washup. Yes, tid. There's where. First. We pass through grass behush the bush to. Whish! A gull. Gulls. Far calls. Coming, far! End here. Us then. Finn, again! Take. Bussoftlhee, mememormee! Till thousendsthee. Lps. The keys to. Given! A way a lone a last a loved a long the

Sista sidan

FinnegansWake.java

BinarySearch.java

# Rekursion i programmering

- Skillnad på rekursion och iteration: en while-loop är inte rekursiv även om den upprepar sig själv
- Rekursiva metoder anropar sig själva för att dela upp ett problem i mindre och mindre delar
- Undviker kodduplicering
- Kan lösa komplexa problem på simpla vis
- Vissa rekursiva datastrukturer har unika fördelar (vi ska prata mer om dessa i morgon)

FinnegansWake.java

BinarySearch.java

# Rekursiv algoritm: Binär sökning

- ”Binär” eftersom den utför antingen/eller, inte för att den opererar på binära nummer
- Så kallad Divide-and-Conquer-algoritm: den tar ett problem och halverar det varje gång den körs
- Behöver ett basfall så att den någon gång slutar köras
- Oerhört effektiv: kan hitta rätt värde i en lista på en miljard värden med enbart 30 sökningar
- Fungerar bara på sorterade samlingar

# Exempel: Vi söker efter värdet 23

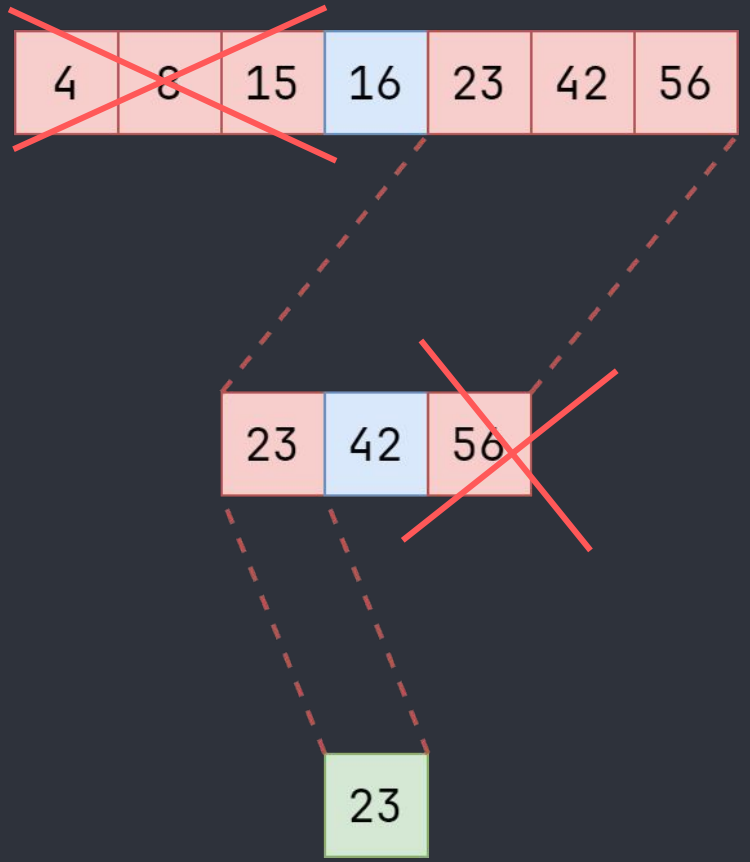
Hur algoritmen funkar:

Om mittvärdet är 23, returnera mittvärdet.

Annars: kolla om mittvärdet värdet är större eller mindre än 23.

Gör sedan ett rekursivt anrop och uteslut vänstra halvan av listan om värdet är större, eller den högra halvan om värdet är mindre. Kolla mittvärdet igen.

Upprepa tills värdet hittats, eller det inte finns några värden kvar.



# [Kodexempel]

Koden finns i klassen BinarySearch.java i zipfilen

# Snabb repetition: Hur vi bestämmer tidskomplexitet

- Vi letar efter den snabbast växande faktorn: de andra spelar ingen roll
- Multiplikation, subtraktion, jämförelseoperatorer och liknande behandlas som konstanter - $O(1)$
- En loop som körs $n$ gånger får komplexiteten $O(n)$
- Nästlade loopar får $O(n^2)$
- Tre nästlade loopar får komplexiteten $O(n^3)$

# Tidskomplexiteten för Divide-and-Conquer-algoritmer

- Binär sökning får tidskomplexiteten O(logn)

- Ni är bekanta med tidskomplexiteter som ökar, men här halveras den varje gång algoritmen körs!

- Viktigt att skilja på O(logn) och O(n * logn):

  O(logn) : Tiden ökar logaritmiskt med antalet n
  O(n * logn) : Tiden ökar linjärt med antalet n, PLUS en logfaktor log n.

DivideAndConquer.java

Table.java

# Exempel på hur de skiljer sig åt

(Kom ihåg att ordo är värsta fallet)

| Antal invärden<br>n | Antal anrop<br>O(logn) | Antal anrop<br>O(n * logn) |
|---|---|---|
| 10 | ~ 3 | 30 |
| 100 | ~ 6 | 600 |
| 1,000 | 10 | 10,000 |
| 10,000 | ~ 13 | 130,000 |
| 1,000,000 | 20 | 20,000,000 |
| 1,000,000,000 | 30 | 30,000,000,000 |

# Rekursiv Fibonaccialgoritm

- En binär sökfunktion anropar sig själv en gång varje gång metoden körs
- En rekursiv fibonacci anropar sig själv två gånger i varje metodanrop
- Behöver precis som binär sökning ett stoppvillkor (basfall)
- Elegant och kort kod
- Oerhört beräkningstung (målet med i dag är att vi ska förstå varför)

# [Kodexempel]

Koden finns i klassen RecursiveFibonacci.java i zipfilen

RekursivFibonacci.java

Cat.java

# Vad är det som händer?!

- Hur kan det krävas så många rekursiva anrop när det bara handlar om att plussa ihop 50 nummer??
- Det här hade gått att göra på papper utan 21 miljarder beräkningar!

Chockad utvecklare →

# [Kodexempel på tavlan]

Det finns en bättre bild av det rekursiva anropsträdet i Rekursivt_fibonacciträd.pdf

# Tidskomplexiteten för rekursiv fib

- Exempel på tidskomplexiteter ni stött på tidigare

  Varje gång antalet n dubblas:

  | | | | |
  |---|---|---|---|
  | $O(n)$ | linjär komplexitet | tiden dubblas | :) |
  | $O(n^2)$ | kvadratisk komplexitet | tiden fyrdubblas | :\| |
  | $O(n^3)$ | kubisk komplexitet | tiden tiodubblas | :( |

- Komplexiteten för rekursiv fibonacci är $O(2^n)$! o_O
  Benämns som exponentiell tidskomplexitet

- Det här innebär att varje gång n ökar med 1 så dubblas tiden som krävs för att exekvera algoritmen

# Minneshantering i Java

- För att förstå vad som händer med en rekursiv fibonacci behöver vi förstå hur minne fungerar i Java
- Två typer av minne: stackminne och heapminne
- Viktigt att ha koll på hur de fungerar, även om vi inte allokerar minne på egen hand i Java
- Kan vara hjälpsamt att tänka på stacken som en prydlig stapel medan heapen är mer som en stor ostrukturerad hög
- Stacken är liten, heapen är enorm

# Pass by reference vs Pass by value

- Java sköter det här automatiskt (i språk som C och C++ måste man däremot specificera vad man skickar in i en metod)

- Primitiva datatyper (int, double, osv) skickas som värde som parameterargument: det vill säga, de kopieras

- Objekt skickas som referenser: man vill inte ha kopior av stora arrayer, klassinstanser, osv

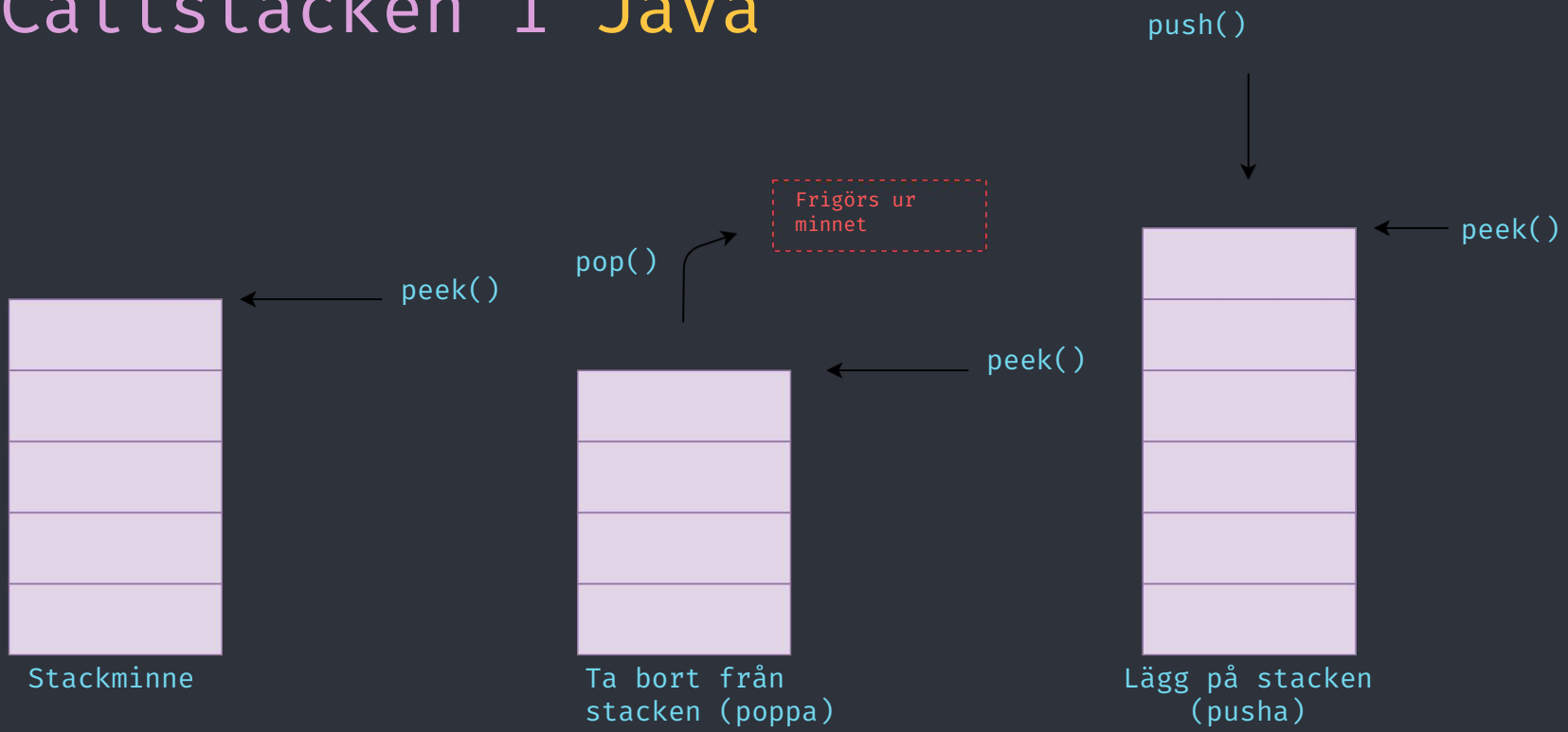
# [Kodexempel]

Koden finns i klassen ValuesVsReferences.java i zipfilen

ValuesAndReferences.java

Stack.java

# Callstacken i Java



Stack.java

Storage.java

# Vad lagras på stacken?

- Metodanrop (activation frames)
- Primitiva variabler (int, double, osv)
- Objektreferenser (själva objekten lagras däremot i heapminnet)
- Stackminnet är litet för att det ska gå fort att accessa

Defaultstorleken i 64-bitarsversionen av JVM är ca 1 MB

# Stacken är varför programmeringsspråk fungerar som de gör

- Anledningen till att variabler har en livslängd
- Anledningen till att metoder har parametrar
- Vi behöver kunna skicka runt referenser mellan stacklagren
- När en objektsreferens försvinner från stacken tar garbage collectorn bort objektet från heap-minnet

# Rekursion blev möjlig när stacken evolverade

- Även assemblerspråk hade en stackpekare (minns ni LMC från Programvaruteknisk Baskurs?). Stacken var dock bara några byte stor och användes främst för att lagra returadresser för subrutiner
- Lisp introducerades 1960: Första högnivåspråket tillsammans med Fortran. Introducerade funktionell programmering
- Lisp är designat kring rekursion. Rekursivitet är dess dominerande kontrollflöde
- Första språket där funktioner hade returvärden och kunde behandlas som uttryck

Lisp.java

ActivationFrames.java

# Activation frames (metodanrop)

- Varje gång ett metodanrop görs i Java skapas en activation frame, också kallad stack frame, På callstacken

- Typisk storlek mellan 20-100 bytes

- Exempel för metod med:
  2 int-parametrar (4 byte var = 8 byte)
  1 arrayreferens (8 byte i 64-bitars-system)
  1 lokal int (4 byte)
  1 lokal double (8 byte)
  1 returadress (8 byte i 64-bitsystem)

  Summa: 8 + 8 + 4 + 8 + 8 = 36 byte

```java
public static double[] calculate(int a, int b)
{
    double[] values = new double[10000];
    int localInt = a + b;
    double localDouble = localInt * 2.5;
    Arrays.fill(values, localDouble);

    return values;
}
```

# Varför rekursiv fibonacci är ökänd

- Storlek på en activation frame för rekursiv fibonacci:

  1 int-parameter (4 byte)
  1 lokal int (4 byte)
  1 returadress (8 byte)

  4 + 4 + 8 = 16 byte

- Minne som teoretiskt krävs för att beräkna fib(50):
  16 byte x 40730022147 metodanrop = 617,981 MB (618 GB!!)

- I verkligheten utför Java optimisering bakom kulisserna, annars skulle programmet krascha på två sekunder

StackOverflow.java

Exponential.java

# Saker som hela tiden fördubblas är inte bra!

- Exponentiell tillväxt är oanvändbar i algoritmer förutom vid väldigt små värden

- Liknar “Riskornen på schackbrädet”: man lägger ett riskorn på 1:a rutan och två på den 2:a, fyra på den 3:e, åtta på den 4:e, osv. Riset på den 64:e (sista) rutan kommer att väga ca 500 miljarder ton

- Utan optimering skulle det på liknande vis ta ta tusentals år att beräkna det 64:e fibonaccinumret med rekursion

# Skräckexempel: Ackermann

- Ackermannfunktionen var den första algoritmen man upptäckte som inte är primitivt rekursiv
- Rekursiv fibonacci har en exponentiell tidsutveckling. Ackermann har superexponentiell tidsutveckling.
- Även för små inputvärden är outputen astronomisk
- Antalet atomer i den observerbara delen av universum: $10^{80}$
- Outputen för Ackermann som anropas med A(4,2): $10^{19728}$
- Skulle även ta längre tid att beräkna än åldern för universum självt

Fördjupning om man är intresserad: https://www.youtube.com/watch?v=i7sm9dzFtEI

Ackermann.java

RecursiveTraversion.java

# Rekursion fungerar enligt principen “Djupet först”

- Rekursiva anrop fungerar alltid enligt principen djupet först
- I exemplet med fib(6) som vi ritade upp på tavlan utförs alltid det vänstra av de två anropen först, dvs fib(n-1)
- När ett av dessa så småningom når ett basfall (det vill säga anropas med 1 eller 0) och returnerar ett värde till föregående metod så kommer den direkt att anropa fib(n-2)
- Blir lättare att förstå om vi tilldelar varje anrop en bokstav

RecursiveTraversion.java

Opposites.java

# Summering: två motsatta algoritmer

- En binär sökning halverar problemet i varje nytt rekursivt anrop
- En rekursiv fibonaccialgoritm dubblerar problemet: varje nytt rekursivt anrop leder till två nya anrop ända tills basfallet är nått
- Kan också sammanfattas så här:
  * Binär sökning har en logaritmisk tillväxt i effektivitet
  * Rekursiv fibonacci har en exponentiell tillväxt i ineffektivitet

# Naiva algoritmer

- Den rekursiva versionen av fibonacci som vi just skapade kallas för en naiv algoritm

- En naiv algoritm är den simplaste lösningen för ett problem, men inte den mest effektiva

  Exempel: Det enklaste sättet att hitta ett värde i en lista är att bara iterera genom den, men som vi såg tidigare är en binär sökning mycket mer effektiv

- Naiva algoritmer är ofta antimönster (anti-patterns). Anti-mönster är dåliga designlösningar som får negativa konsekvenser i längden

NaiveRecursion.java

NonNaiveFibonacci.java

# Ett annat sätt att beräkna fibonacci

- Motsatsen till en naiv algoritm är en effektiv algoritm

- En effektiv algoritm bör inte bara minimera tidskomplexitet utan även använda minne effektivt (platskomplexitet)

- En lag inom beräkningsteori är att all primitiv rekursion även kan uttryckas med loopar

- Kan vi göra fibonaccialgoritmen mer effektiv genom att använda en for-loop i stället?

# [Kodexempel]

Koden finns i klassen IterativFibonacci.java i zipfilen

NonNaiveFibonacci.java

IterativeComplexity.java

# Vad får vi för tidskomplexitet?

- Iterativ fibonacci har tidskomplexiteten O(n)
  Rekursiv fibonacci hade tidskomplexiteten $O(2^n)$

- Kom ihåg att tidskomplexitet inte säger något om hur fort en algoritm exekveras, utan hur tiden påverkas av datamängden.

- I det iterativa fallet dubblas tiden när antalet n dubblas:
  20 loopvarv om n = 20, 40 loopvarv om n = 40
  Den kommer dock att exekvera hundratusentals gånger snabbare än den rekursiva versionen

- I båda fallen kommer dock våra algoritmer bara att kunna beräkna ganska låga n-värden (n < 160). Varför?

# Primitiva datatyper är begränsade

- När talkedjor växer så snabbt som fibonacci (exponentiellt) är det lätt att överskrida gränsen för hur stora nummer som går att spara i en `int`
- Använd alltid `long` som data- och returtyp i stället för `int` när stora beräkningar sker
- Finns dock tillfällen då inte en `long` heller räcker till, som nu
- Ingenting ni generellt behöver tänka på (mest relevant inom kryptografi, maskininlärning, osv)

# Dynamisk programmering

- Den iterativa versionen av fibonacci är ett exempel på så kallad dynamisk programmering, en optimeringsteknik

- Dumt namn; har egt. inget att göra med datorprogrammering (begreppet är matematiskt och myntades redan på 1950-talet)

- Finns två typer av dynamisk programmering:

| Memoisering (Memoization) | Tabulering (Tabulation) |
| --- | --- |
| Top-down-approach | Bottom-up-approach |
| Rekursiv approach | Iterativ approach |

- Idén är att spara resultaten från delproblem så att man slipper göra samma kalkyleringar flera gånger

# Dynamisk rekursiv fibonacci

- Vi använde tabulering för att göra en iterativ lösning, men vi hade kunnat använda memoisering också och skapa en bättre rekursiv version

- En sådan algoritm behöver någon form av datastruktur för att spara tidigare beräkningar i så att vi slipper göra om dem

- Vanligast är att använda en array eller lista

- Precis som den iterativa versionen kommer den här algoritmen att få tidskomplexiteten O(n)

# [Kodexempel]

Koden finns i klassen RecursiveFibonacciDynamic.java i zipfilen

# Vad särskiljer rekursivitet?

> “Om rekursion är en metod som anropar sig själv, vad är egentligen skillnaden mellan den och en for-loop?”

- Båda är en sorts upprepning. Skillnaden ligger i hur kontrollflöden, tillståndshantering och minnesallokering fungerar
- Rekursion sparar tillståndet (state) för variabler på stacken, till skillnad från loopar som inte kan utnyttja mer minnesresurser
- Använder parametrar för att skicka uppdaterad data
- Rekursion arbetar med stacken, iteration arbetar med heapen

Distinguised.java

UniqueRecursion.java

# Det finns datastrukturer som är rekursiva till naturen

- Divide-and-Conquer-algoritmer är naturligt rekursiva

  Exempel: Binär sökning, MergeSort, QuickSort

- Träd och Grafer är datastrukturer som använder rekursion för att kunna traversera alla noder (vi kommer prata mer om dem nästa tillfälle)

- Rekursion kan vara väldigt kraftfull, men också oerhört resurskrävande om den används felaktigt (ex: fibonacci)

UniqueRecursion.java

Summary.java

# Sammanfattning av dagen, pt. 1

{

- **Rekursiva mönster**
  - Finns i naturen
  - Metoder som anropar sig själva
  - Behöver ett basfall
- **Binär sökning**
  - Divide-and-conquer
  - O(logn)
- **Rekursiv fibonacci**
  - Tidskomplexiteten $O(2^n)$
  - Varje gång n ökar med 1 dubblas exekveringstiden
- **Stacken**
  - LIFO
  - Anledningen till att variabler har en livslängd
  - Vid varje metodanrop skapas en activation frame

}

Summary.java

Summarypt2.java

# Sammanfattning av dagen, pt. 2

{

- **Naiva algoritmer**
  - Enklaste lösningen på ett problem
  - Sällan den bästa (antimönster)
- **Primitiva Datatyper**
  - Har begränsat med minne
  - Använd long för stora beräkningar
- **Kontroll-flöden**
  - Definierar programmeringsspråk
  - Påverkar ofta tidskomplexiteten
- **Dynamisk programmering**
  - Optimeringsteknik
  - Memoisering (rekursiv) eller tabulering (iterativ)

}

# Inför morgondagen

- Presentationen från dagens föreläsning finns i modulen för vecka 5

- Koppla av ett par timmar när ni kommer hem. Titta sen genom dagens presentation en gång till någon gång under kvällen (behöver bara göras översiktligt, men saker fastnar bättre då!)

- Ni kommer få en länk till Github i morgon där allt material, inklusive kodexempel från i dag, finns upplagt

```
return 0;
```

# Nästa föreläsning: Stackar, köer, binära träd, grafer

Vi ska bland annat:

- Titta närmare på stacken som datastruktur
- Lära oss om binära träd och balanserade trädstrukturer
- Lära oss litegrann om köer och deras användningsområden
- Prata om abstrakta datatyper
- Titta översiktligt på grafer och olika sorters graftraversering (bredden först vs. djupet först)

```
}
```

Mail:
carl-johan.johansson@im.uu.se