

Dagens föreläsning: Rekursiva algoritmer, referenser och minne

{

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 är en algoritm?

- I grund och botten är de kod som **försöker lösa ett specifikt problem**. Ofta är de abstrakta koncept med mer än en implementation
- En algoritm beskriver en högre nivå av abstraktion än hårdvaran som den körs på. Den är **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
- ~~“En algoritm är som ett recept”~~ är därför egt. en dålig analogi

Algoritmer är mjukvara (eller?)



Weaving software into core memory by hand

157K views · 13 years ago



oisiaa

The software of the Apollo guidance computer was hand woven into rope core memory.

- Hårdvaruutvecklare **bygger arkitektur**, programmerare **bygger program och algoritmer**. Gränsen kan dock vara flytande
- Under månlandningen sydde man t.ex in program i minnet och skapade hårdkodade algoritmer för att beräkna landningskoordinater

Varför lär vi oss algoritmer och datastrukturer?

- Om man vill programmera behöver man **en djupare förståelse** för vad som händer under lagren av abstraktion
- **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 programmeringsspråk?

- Ett programmeringsspråk är ett **lager av abstraktion** som vi använder för att ge instruktioner till en dator
- **Kontrollflöden** definierar programmeringsspråk. 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 |

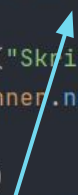
- **Rekursion** är ytterligare en typ av kontrollflöde

Vad är rekursion?

- Rekursion uppstår när någonting **definieras i termer av sig självt**
- Självrefererande kod (ex: metod som anropar sig själv), dvs **en instans som levererar en instans av sig själv**
- Inte bara ett koncept inom programmering
- Från latinets *recurre*:
"spring tillbaka"

```
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;
}
```

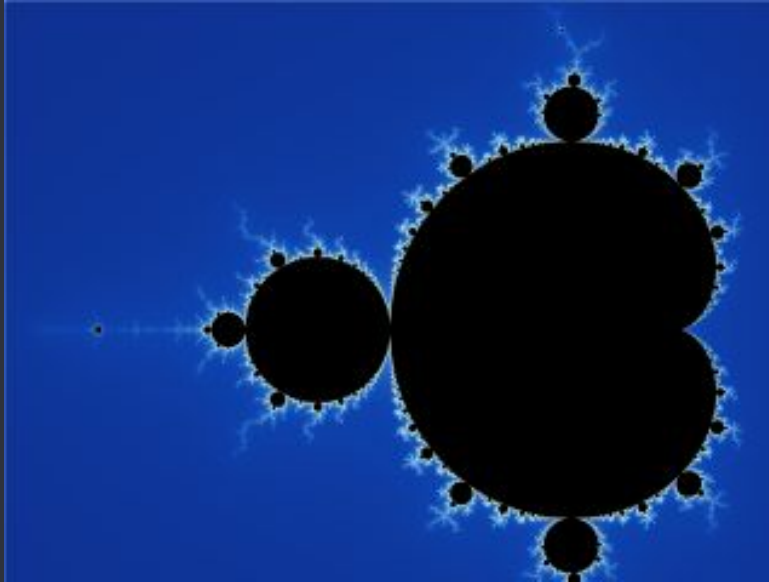


- Ett av de **mest fundamentala koncepten** för hur världen och vi själva är uppbyggda

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

Stephen Hawking

Fraktaler

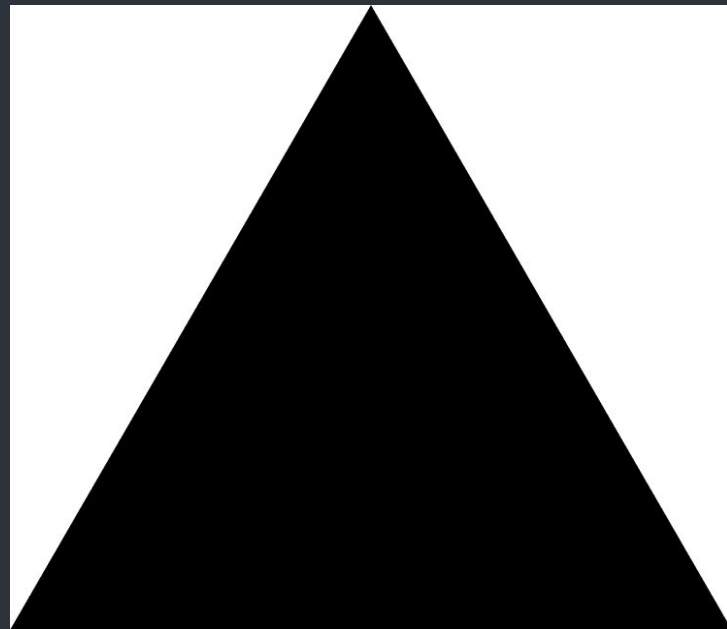


Mandelbrotfraktalen

- Benoit B. Mandelbrot
- Arbetade på IBM på 70- och 80-talet
- Upprepar sig i all oändlighet

Sierpinski triangeln

- Oändlig fraktal som ritar upp en liksidig triangel som sedan delas in i tre mindre trianglar, vilka i sin tur delas in i tre ännu mindre, osv
- 1999 upptäckte man att **fraktal-antenn**, antenner med självrefererande design som påminde om Sierpinski, var mer stabila än traditionella
- Har haft stor påverkan för wifi-baserad kommunikation



Fibonacci-sekvensen

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

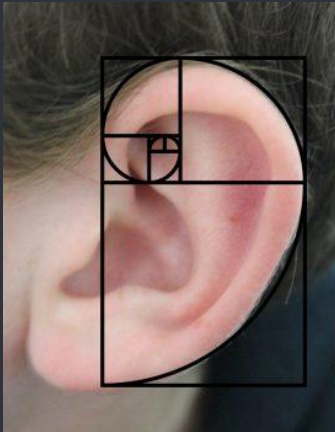
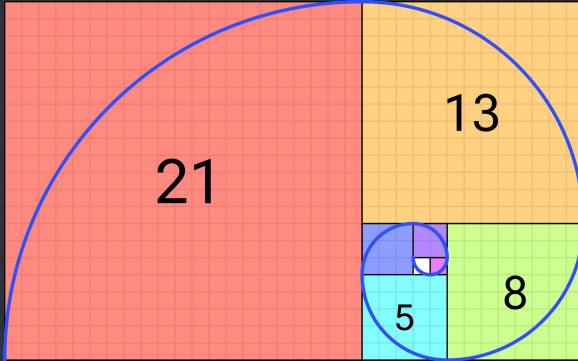
$$F(n) = F(n-1) + F(n-2)$$

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

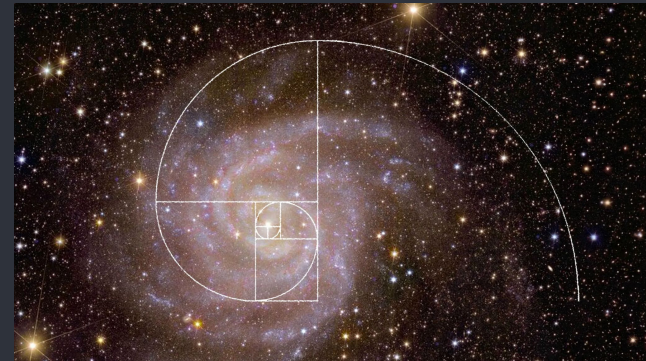
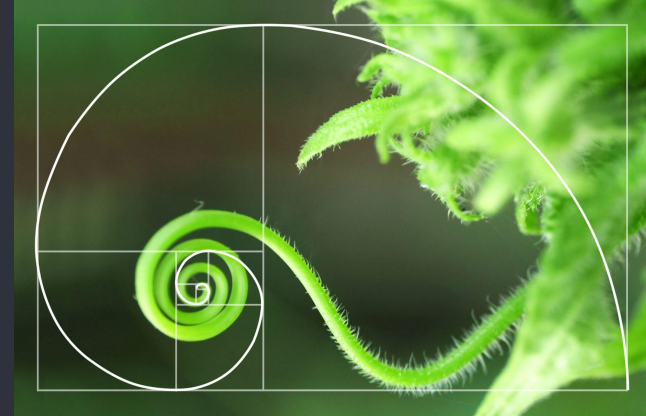
– Varje nytt tal i sekvensen är produkten av de två föregående

Fibonacci.java

1
2
3
4
5
6
7
8
9
10
11
12
13
14



Fibonacci.java



Romanescobroccoli



Biologisk rekursion

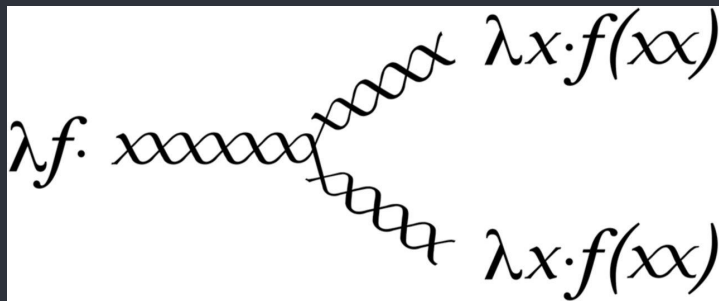
Celldelning:

- 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

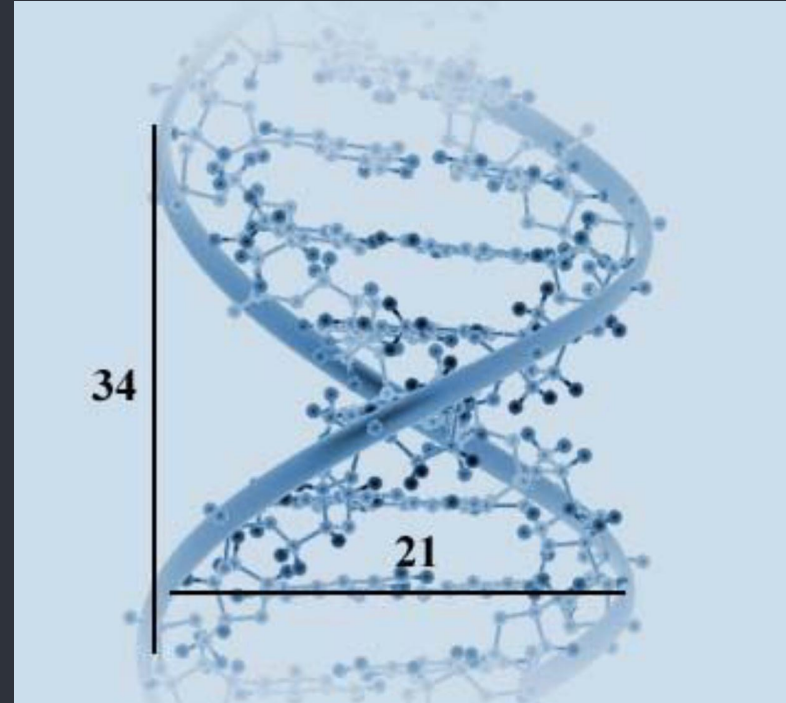
“Replication in biological systems is intuitively similar to recursion in computational systems.”



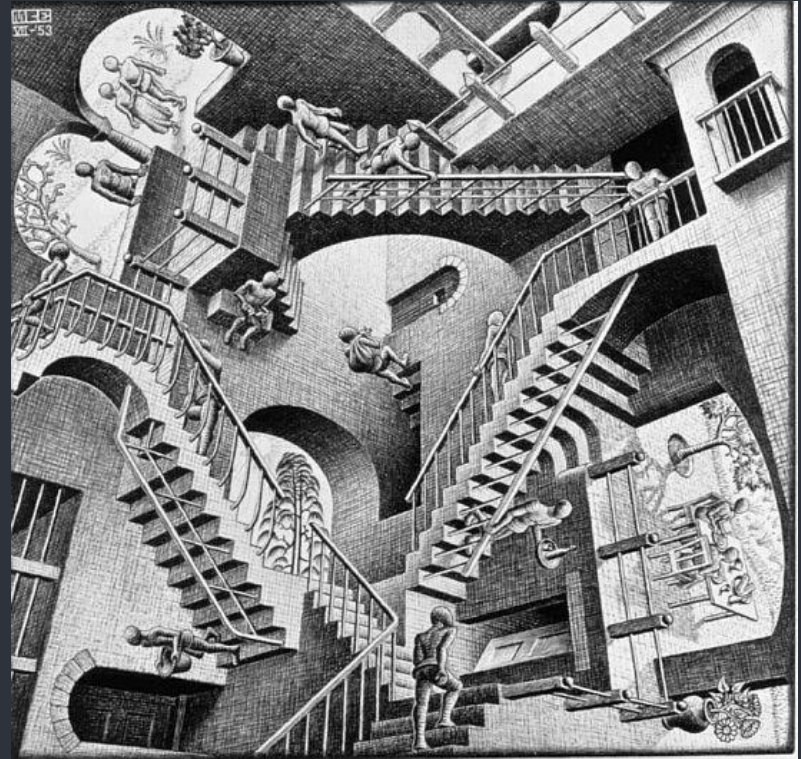
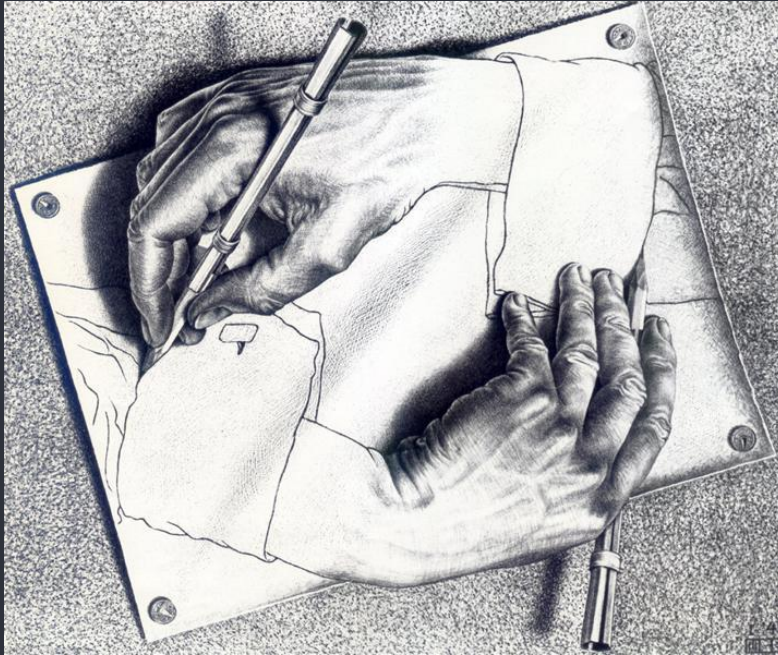
Biologisk Replikationsgaffel: DNA-delning i två nya sekvenser

Det gyllene snittet (golden ratio)

- Längden på en kurva i en DNA-spiral är 34 Ångström. Bredden på spiralen är 21 Ångström.
- Förhållandet mellan fibonaccitalen närmar sig ~ 1.618 och kallas det gyllene snittet
- Finns i naturen, i konsten, osv



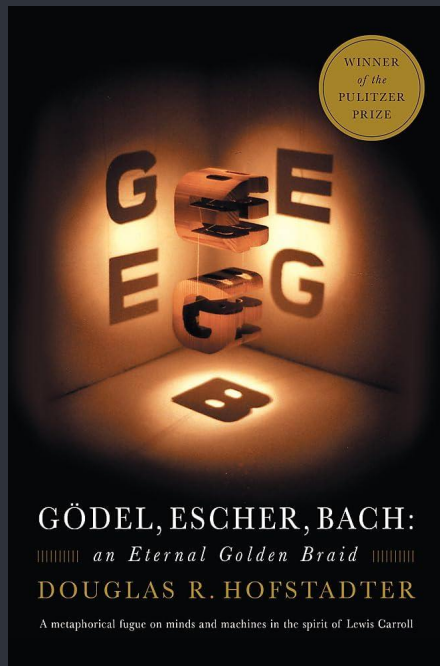
Rekursiv konst



Boktips: Gödel, Escher, Bach

"In the end, we are self-perceiving, self-inventing, locked-in mirages that are little miracles of self-reference"

- Level up från Charles Petzolds bok Code
- Skriven av datorvetaren och kognitionsforskaren Douglas Hofstadter som fick en Pulitzer
- Handlar till stor del om rekursiva mönster i musik, konst, matematik, datorer och mänskligt medvetande



Rekursion i programmering

- Skillnad på rekursion och iteration: en while-loop är inte rekursiv även om den upprepas
- Rekursiva metoder **anropar sig själva** för att dela upp ett problem i mindre och mindre delar
- Kräver ett **basfall** så att de någon gång slutar anropa sig själva
- Undviker kodduplicering och kan lösa komplexa problem på enkla vis

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
- Oerhört effektiv: kan hitta rätt värde i en lista på en miljard värden med enbart 30 sökningar
- Fungerar bara på sorterade samlingar

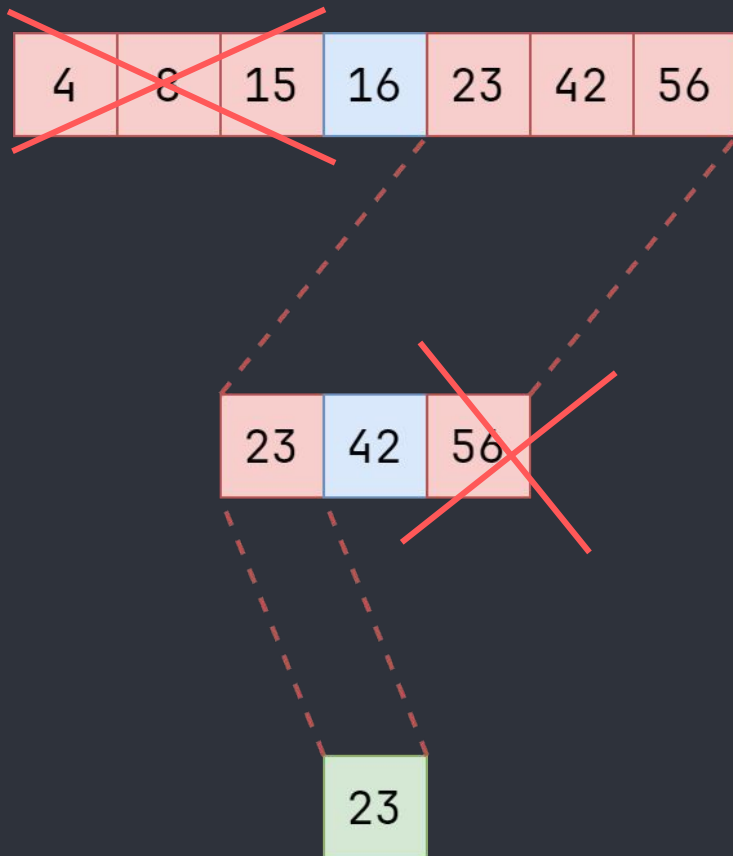
Exempel: Vi söker efter värdet 23

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 $O(\log n)$
- Ni är bekanta med tidskomplexiteter som ökar, men här halveras den varje gång algoritmen körs!
- Viktigt att skilja på $O(\log n)$ och $O(n * \log n)$:
 - $O(\log n)$: Tiden ökar **logaritmiskt** med antalet n
 - $O(n * \log n)$: Tiden ökar **linjärt** med antalet n ,
PLUS en logfaktor $\log n$.

Exempel på hur de skiljer sig åt

(Kom ihåg att ordo är värsta fallet)

Antal invärden

n

10

100

1,000

10,000

1,000,000

1,000,000,000

Antal anrop

$O(\log n)$

~ 3

~ 6

10

~ 13

20

30

Antal anrop

$O(n * \log n)$

30

600

10,000

130,000

20,000,000

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

[Kodexempel]

Koden finns i klassen RecursiveFibonacci.java i zipfilen

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 programmerare



[Kodexempel på tavlan]

Det finns en bättre bild
i `Recursive_fibonacci_tree.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_0
Benämns som exponentiell tidskomplexitet

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

Saker som hela tiden fördubblas är inte bra!

- Exponentiell tillväxt är **oanvändbar** 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**
- Staplat på hög skulle det vara ca 9 ggr **längre än hela solsystemet**



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
- 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

Minnestyper och -processer 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

Memory.java

Stack.java

Callstacken

1
2
3
4
5
6
7
8
9
10
11
12
13
14

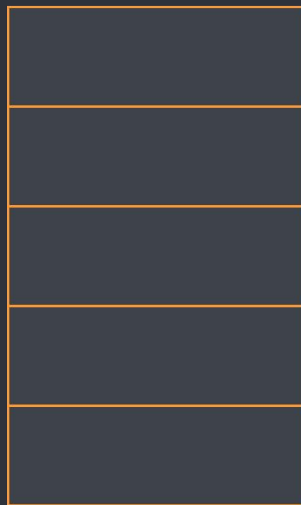


Stackminne

pop()

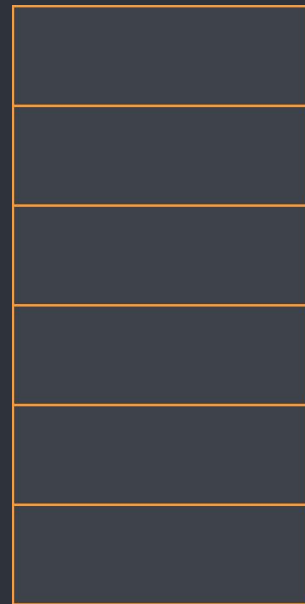


Frigörs ur
minnet



Poppa (ta av
från stacken)

push()

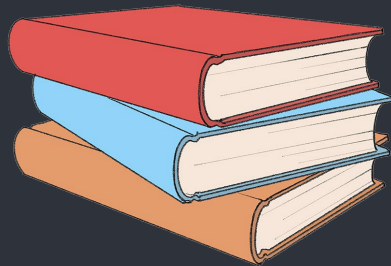


Pusha (lägga på
stacken)

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 1 MB (1024 KB)



Activation frames (metodanrop)

– Varje gång ett metodanrop görs 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

```
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 omsätts för att beräkna **fib(50)**:

16 byte x 2075316483 metoodanrop = **332 Gigabyte !!**

- I verkligheten skrivs dock inte varje stack frame till RAM utan hålls i en temporär cache; annars skulle det här ta timmar!

Myter om rekursiv fibonacci

- En vanlig missuppfattning är att rekursiv fibonacci orsakar stacköverflyllnad (stack overflow), men antalet activation frames på callstacken samtidigt är aldrig större än fibonacci-talet man vill beräkna (om $n = 50$ så finns det max 50 frames samtidigt på stacken)
- En annan missuppfattning är att vi skriver all den där datan till minnet, men den existerar bara i temporära CPU-register: Vi skriver inte över samma 16 byte i minnet 20 miljarder gånger!
- Anledningen till att den här algoritmen suger är helt enkelt att den är så löjligt ineffektiv i sin tidsutveckling

Vad som händer på stacken vid rekursiva anrop

- 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:

[Kodexempel på tavlan]

Det finns en bättre bild
i `Recursive_fibonacci_tree.pdf`

Om referenser (pekare)

- En **pekare** är en variabel som lagrar (“pekar mot”) en annan variabels minnesplats
- När någon nämner “**referens**” i Java är det egentligen en sorts pekartyp som man syftar till
- Pekare finns i alla moderna språk, men alla språk låter inte användaren hantera dem direkt
- En av de stora idèerna bakom Java var att programmerare inte längre skulle behöva bry sig om såna här saker
- Resultatet är 30 år av det här:



Det uppstod problem



java.lang.NullPointerException

Ett oväntat undantag uppstod.

Ett oväntat undantag uppstod.

Ett oväntat undantag uppstod.

Ok

Detaljer >>

Hur referenser funkar i Java

- NullPointerException är ett så stort problem i Java att man uppfann ett helt språk, **Kotlin**, för att komma runt det
- I Java är alla objektvariabler egentligen **referenser** som pekar mot data som finns på **heapen**

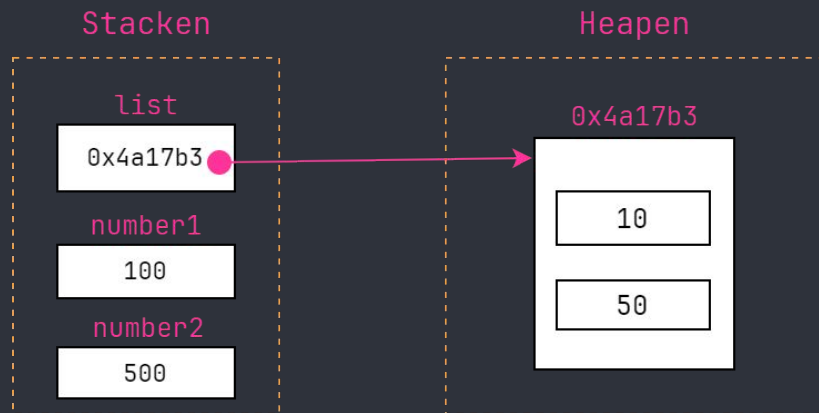
```
ArrayList<Integer> list = new ArrayList<>();
```

```
list.add(10);
```

```
list.add(50);
```

```
int number1 = 100;
```

```
int number2 = 500;
```



Ex 1: Kompileringsfel

```
1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         String string;
6
7         System.out.println(string.length());
8     }
9 }
```

Variable 'string' might not have been initialized

Initialize variable 'string' Alt+Shift+Enter

More actions... Alt+Enter

String string

NullPointerExample

Ex 2: Kompilering trots nullpekare

```
1  public class Main
2
3  {
4
5      public static void main(String[] args)
6      {
7          String string = getStringValue();
8
9          System.out.println(string.length());
10     }
11
12     public static String getStringValue()
13     {
14         return null;
15     }
16 }
```

Ex 3: Kompilatorn är bortfintad

```
1  public static void main(String[] args)
2
3  {
4      ArrayList<String> list = getStringList();
5
6      System.out.println(list.get(0));
7  }
8
9  public static ArrayList<String> getStringList()
10 {
11     ArrayList<String> list = new ArrayList<>();
12     list.add(null);
13
14     return list;
15 }
```

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

Varför referenser är viktiga att förstå

- De är intimt bundna till hur callstacken fungerar
- Utan uppdelningen mellan stacken och heapen hade vi inte kunnat skapa dynamiska program

Innan dynamisk minnesallokering

Assemblerprogram:

```
LDA #5          ; Ladda 5 i ackumulatorn
STA NUM1        ; Spara värdet i variabeln NUM1
LDA #3          ; Ladda 3 i ackumulatorn
STA NUM2        ; Spara värdet i variabeln NUM2
JSR ADD_NUMS    ; Hoppa till subrutin ADD_NUMS
HLT             ; Programmet stoppar
```

ADD_NUMS:

```
LDA NUM1        ; Ladda NUM1 i ackumulatorn
ADD NUM2        ; Addera NUM2
STA RESULT      ; Lagra resultat i RESULT
RTS             ; Återvänd från subrutin
```

```
NUM1:  .BYTE 0 ; Minnesallokering för NUM1
NUM2:  .BYTE 0 ; Minnesallokering för NUM2
RESULT: .BYTE 0 ; Minnesallokering för RESULT
```

Problem utan callstack:

Statisk minnesallokering kan bara jobba med fördefinierade resurser (NUM1, NUM2, RESULT)

ADD_NUMS kan inte skapa någonting, t.ex. en array

Kan inte ta emot indata

Anropas ADD_NUMS på nytt ändrar den samma variabler

Callstacken möjliggjorde dynamisk allokering av minne

- Utan callstacken hade vi inte haft dynamisk minnesallokering
- **Lisp** var det första högnivåspråket tillsammans med Fortran
- Första språket med en callstackliknande struktur
- Subrutiner kunde plötsligt använda parametrar för att ta emot data och hade returvärden: **vi fick våra första funktioner**
- Lisp är **naturligt rekursivt**: alla datastrukturer använder rekursion som kontrollflöde

Callstacken är varför programmerings- språk fungerar som de gör

- Anledningen till att variabler har en livslängd: när vi poppar en stack frame försvinner referenserna
- Anledningen till att metoder har parametrar: Vi behöver kunna skicka runt referenser mellan stacklagren när stacken byggs på
- När en objektsreferens försvinner från stacken tar **garbage collector**n bort objektet från heapminnet

Kaffepaus

Recap: två motsatta algoritmer

- En binär sökning **halverar** problemet i varje nytt rekursivt anrop
- En rekursiv fibonaccialgoritm **dubblar** 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 skapade i förmiddags är en **naiv algoritm**
- En naiv algoritm är den simplaste lösningen för ett problem, men nästan aldrig 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

Ett bättre 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 `IterativeFibonacci.java` i zipfilen

Vad får vi för tidskomplexitet?

- **Iterativ fibonacci** har tidskomplexiteten $O(n)$
Rekursiv fibonacci har 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 **miljontals 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
- Dock 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 en rekursiv algoritm?

- Både **rekursion** och **iteration** 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**

Vanliga rekursiva algoritmer

- **Divide-and-Conquer-algoritmer** är naturligt rekursiva
Exempel: Binär sökning, MergeSort, QuickSort
- **Backtrackingalgoritmer** och **traverseringsalgoritmer** brukar också använda sig av rekursivitet (vi ska prata mer om dem under vecka 7)
- Rekursion kan vara väldigt kraftfull, men också **oerhört resurskrävande** om den används felaktigt (ex: naiv fibonacci)

Sammanfattning av dagen, pt. 1

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

Sammanfattning av dagen, pt. 2

| | | | |
|----|---|---------------|---|
| 1 | { | | |
| 2 | | | |
| 3 | | Naiva | - Enklaste lösningen på ett problem |
| 4 | | algoritmer | - Sällan den bästa (antimönster) |
| 5 | | | |
| 6 | | Primitiva | - Har begränsat med minne |
| 7 | | Datatyper | - Använd long för stora beräkningar |
| 8 | | | |
| 9 | | Kontroll- | - Definierar programmeringsspråk |
| 10 | | flöden | - Påverkar ofta tidskomplexiteten |
| 11 | | | |
| 12 | | Dynamisk | - Optimeringsteknik |
| 13 | | programmering | - Memoisering (rekursiv) eller tabulering (iterativ) |
| 14 | } | | |

Inför morgondagen

- Jag kommer skicka ut ett anslag med presentationen från dagens föreläsning
- Koppla av ett par timmar när ni kommer hem. Ögna sen genom presentationen en gång till under kvällen (saker fastnar bättre då)
- Ni kommer få en länk till [Github](#) i morgon där allt material, inklusive presentation + kodexempel från i dag, finns upplagt