

Dagens föreläsning: Rekursiva algoritmer, callstacken 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

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

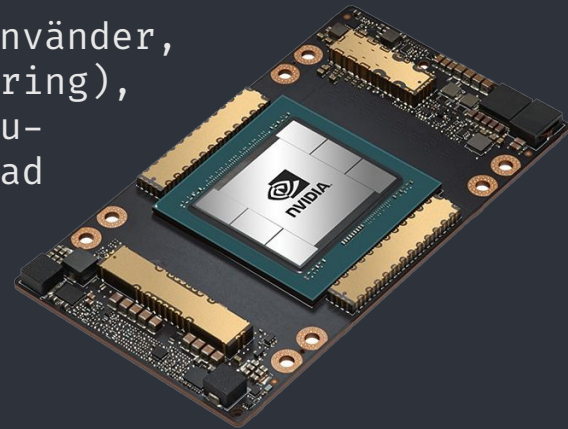
Linus Thorvalds

Vad är en algoritm?

- I grund och botten är algoritmer kod som **försöker lösa ett specifikt problem**
- En algoritm beskriver en högre nivå av abstraktion än rena maskininstruktioner av typen “lyft en spak”, “sänk en nål”, osv
- En algoritm **beräknar någonting** och **kan producera nya resultat** beroende på sin indata; den kan ha olika beteende beroende på datamängden, osv
- Att säga att ~~“en algoritm är som ett recept”~~ är därför egt. en dålig analogi

Algoritmer är mjukvara (eller?)

- Hårdvaruutvecklare **bygger arkitektur**, programmerare **bygger program och algoritmer**. Stämmer dock inte alltid:
- **Core Rope Memory**: Under Apollouppdragen sydde man in program i minnet på mänlandaren och skapade inbyggda algoritmer för att beräkna landningskoordinater
- **GPU:er** implementerar hårdvarualgoritmer för bland annat parallellprocessering, raytracing, videoencoding och liknande
- **AES-kryptering**: Mycket av den kryptering vi använder, som HTTPS (webbläsare), BitLocker (diskkryptering), VPN (nätverk) och-så-vidare utförs av hårdvarualgoritmer som är inbyggda i datorn - så kallad **hårdvaruaccelerering**
- Hårdvarualgoritmer är **permanenta** när de väl har implementerats och går inte att ändra på efter tillverkning



Varför lär vi oss algoritmer och datastrukturer i Java?

- 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: en funktion som kan beräknas av en dator kan i teorin beräknas av alla datorer så länge de är Turing-kompletta
- Det här innebär att algoritmer är universella: **syntax och implementation** kan skilja sig, men alla programmeringsspråk kan utföra samma underliggande beräkningar och **logiken för algoritmen är densamma**
- Även datastrukturer är universella: Arrayer, listor, osv finns i alla språk. Det du kan göra i ett språk kan du göra i ett annat

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, osv
Metodanrop	Subrutiner; utomstående kodblock som körs

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

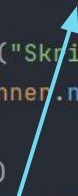
Rekursion

- Rekursion uppstår när någonting **definieras i termer av sig själv**
- Självrefererande kod (exempelvis en metod som anropar sig själv)
- Går att uttrycka så här i programmering: **en instans som levererar en instans av sig själv**
- Inte bara någonting som finns inom programmering: ett av de **mest fundamentala koncepten** för hur världen och vi själva är uppbyggda

Exempel: `inputName()` anropar sig själv

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

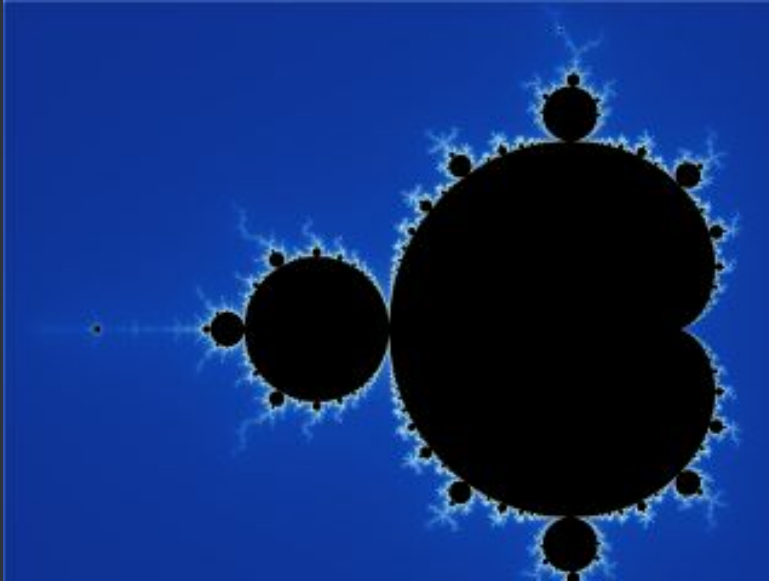
    if(input.isEmpty())
        return inputName(scanner);
    else
        return input;
}
```



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

Stephen Hawking

Fraktaler

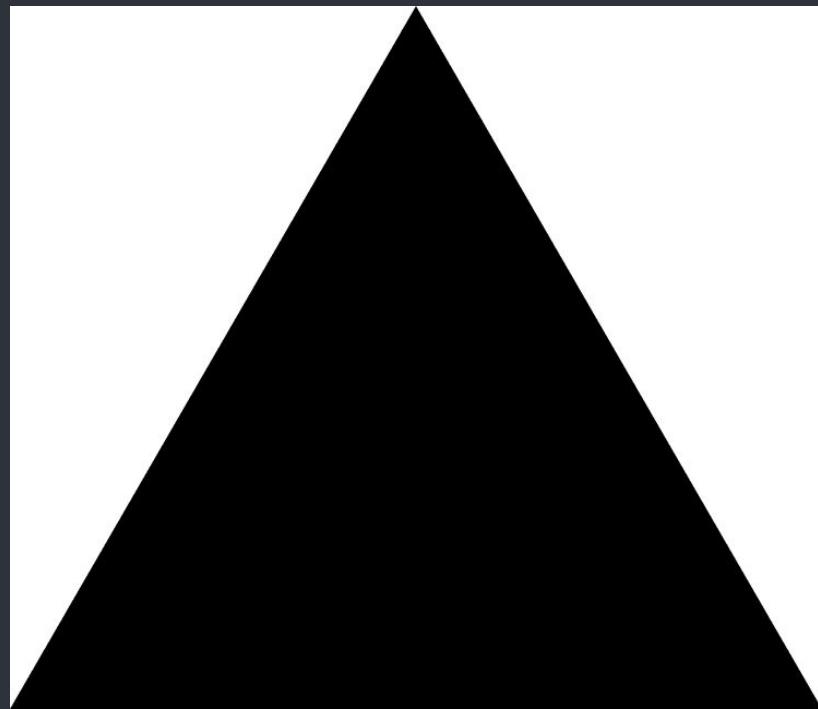


Mandelbrotfraktalen

- **Benoit B. Mandelbrot**
- Arbetade på IBM på 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 **fraktalantenn**, antenner med självrefererande design som påminde om dessa mönster, gav bättre mottagning än traditionella antenner
- Har haft stor påverkan på bland annat **wifi-baserad kommunikation**



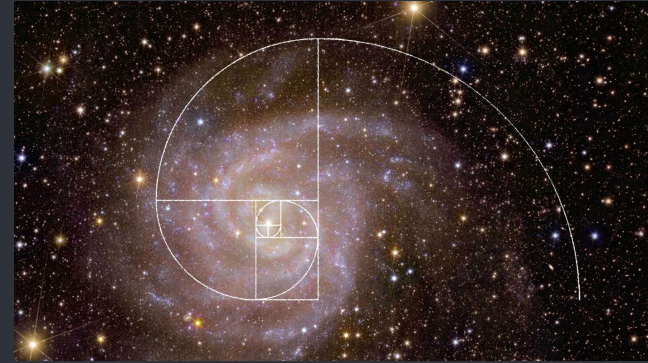
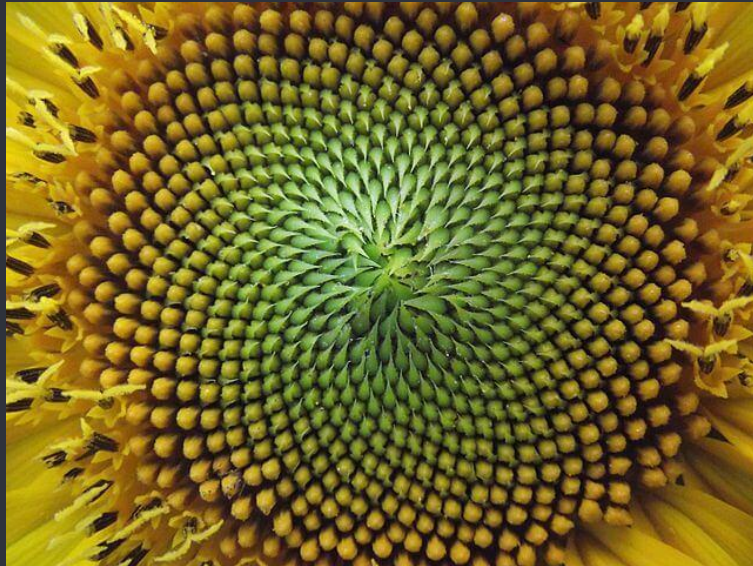
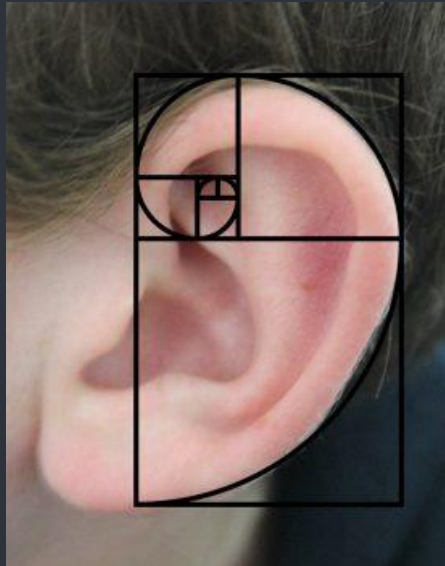
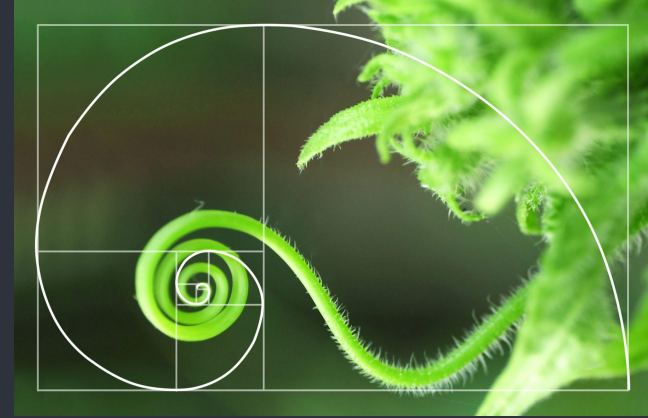
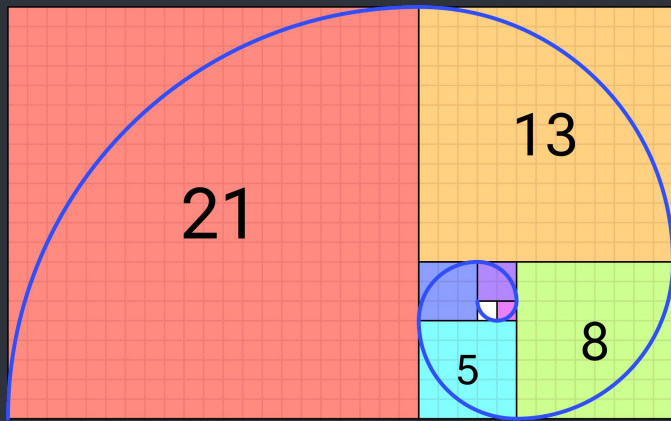
Fibonaccisekvensen

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

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

$$F(8) = F(7) + F(6) = 13 + 8 = 21$$

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



Fraktalbroccoli med fibonacci-spiraler



Biologisk rekursion

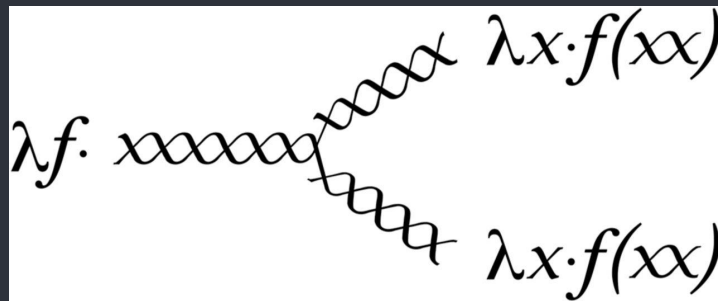
Cellldelning

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

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.”

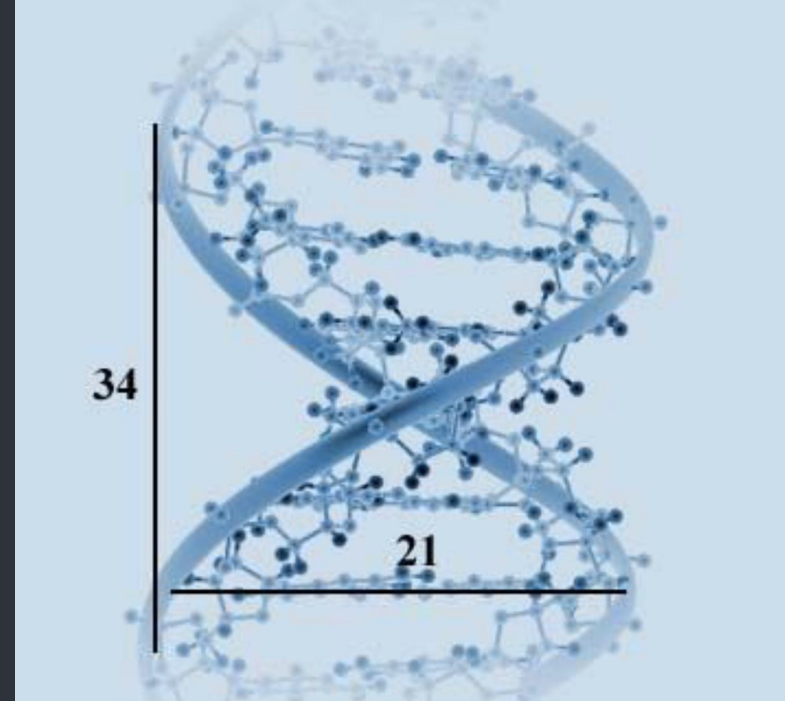


Biologisk Replikationsgaffel: DNA-delning i två nya sekvenser

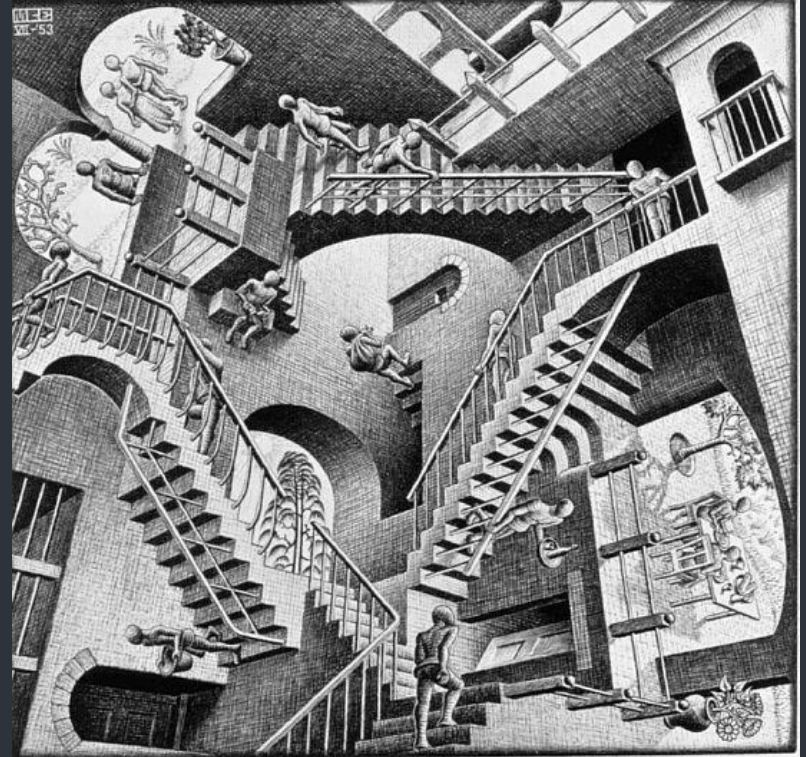
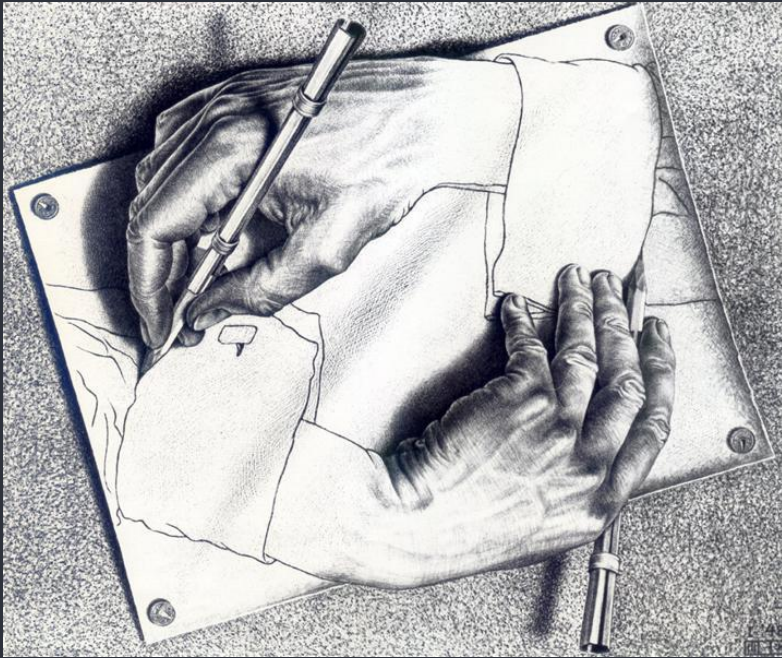
Det gyllene snittet

- Det **gyllene snittet** är förhållandet mellan fibonaccitalen
- Det här värdet närmar sig **~1.618** ju större talen är
- Finns överallt i naturen: bredden på en DNA-spiral är exempelvis 21 Ångström och höjden på en kurva i en DNA-spiral är 34 Ångström; båda är fibonaccital

$$\frac{34}{21} \approx 1.6191$$



Rekursiv bildkunst

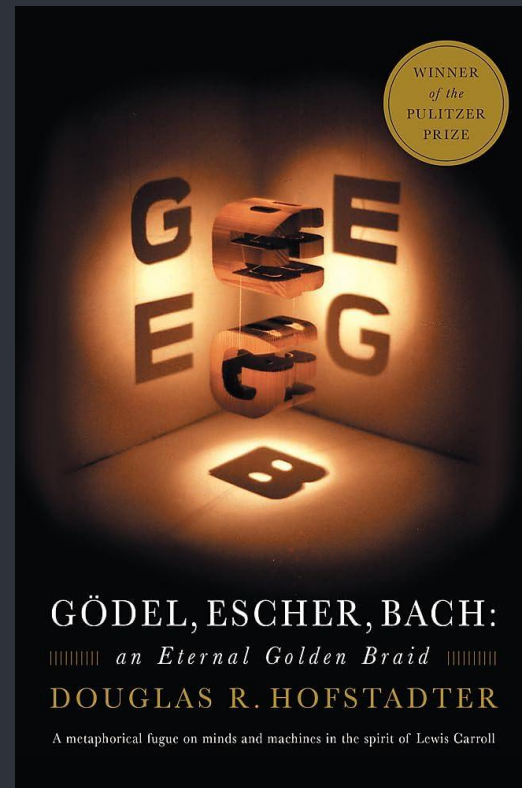


Filmtips: MC Escher – Journey to Infinity <https://www.imdb.com/title/tt8297550>

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 matematikern **Douglas Hofstadter** som fick en Pulitzer för boken
- Handlar till stor del om **rekursiva mönster** i musik, konst, matematik, datorer och mänskligt medvetande



Ämnet för veckan: rekursion i programmering

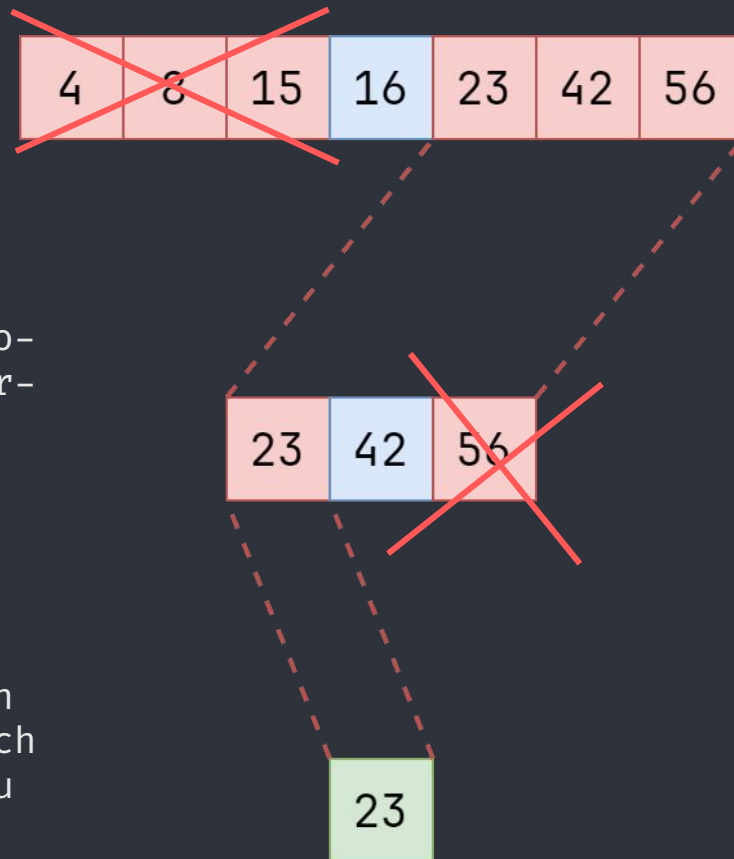
- Det är skillnad på **rekursion** och **iteration**: en while-loop är inte rekursiv även om den upprepas många gånger
- Rekursiva metoder **anropar sig själva** för att dela upp ett problem i mindre och mindre delar
- Kräver ett **basfall** så att de någon gång slutar anropa sig själva
- Undviker kodduplicering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

Exempel på rekursiv algoritm: Binär Sökning

- **"Binär"** eftersom den alltid väljer en av två vägar att gå (den beräknar alltså inte binära nummer)
- En 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**

Binär Sökning

- Om mittvärdet är 23, returnera mittvärdet
- (Mittvärdet är 16) kolla om mittvärdet är större eller mindre än 23
- (Mittvärdet är mindre än 23) Vi kan nu ignorera vänstra halvan eftersom vi vet att värdet vi letar efter inte finns där
- Vi har nu en samling som är hälften så stor som den ursprungliga och kollar mittvärdet igen
- (Mittvärdet är 42) Eftersom 23 är mindre än 42 måste värdet finnas i vänstra halvan, och vi kan kasta den högra halvan. Det finns nu bara ett tal kvar



Binär Sökning vs linjär sökning



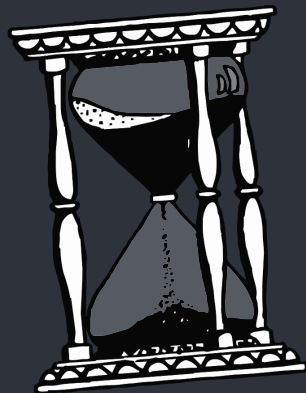
[Kodexempel]

Koden finns i filen `BinarySearch.java`

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 **$O(n^3)$**

Tumregel: Om din kod kräver **$O(n^3)$** tidskomplexitet har du gjort ett misstag någonstans och bör designa om den



“If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.”

Linus Thorvalds

Tidskomplexiteten för Divide-and-Conquer-algoritmer

- Binär sökning får tidskomplexiteten $O(\log n)$
- Datamängden **halveras** varje gång den här algoritmen gör ett nytt metodanrop!
- Viktigt att kunna 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 ,
gänger en logfaktor $\log n$

Exempel på hur de skiljer sig åt

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

Fotnot: $O(n \log n)$ är fortfarande mycket bra: **MergeSort**, som är en av de snabbaste sorteringsalgoritmerna, har denna tidskomplexitet

```
kaffepaus(15);
```

Rekursiv fibonaccialgoritm

- Vi pratade om **fibonaccisekvensen** tidigare, där varje tal är produkten av de två föregående. Den här talsekvensen är rekursiv och går därför också att beräkna med hjälp av en rekursiv algoritm
- Den **binära sökfunktionen** anropade sig själv **en gång** varje gång metoden kördes
- En **rekursiv fibonacci** anropar sig själv **två gånger** i varje metodanrop
- Behöver precis som binär sökning ett stoppvillkor (kallat ett **basfall**)
- Världens mest eleganta algoritm?

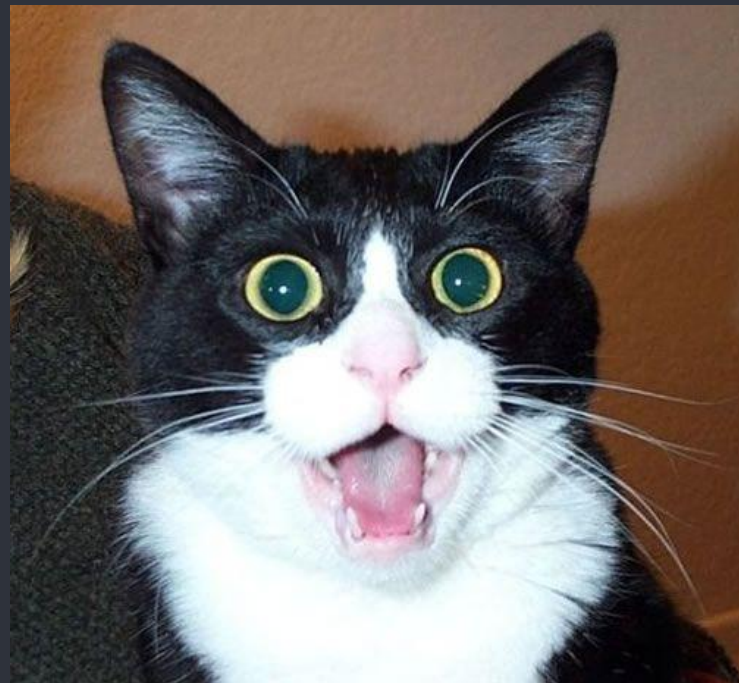
Rekursiv fibonaccialgorithm

```
public long fibonacci(int n)
{
    if(n ≤ 1) { return n; }
    return fibonacci(n-1) + fibonacci(n-2);
}
```



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

Det rekursiva anropsträdet

Exemplet finns i filen `Recursive_fibonacci_tree.pdf`

Tidskomplexiteten för rekursiv fibonacci

- 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 åttadubblas	😞

- 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**
- Det här är mer ris än vi producerat i världen sen människan började odla ris
- Staplat på hög på en schackruta skulle stapeln 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 $A(4,3)$: 10^{19728}
- Skulle även **ta längre tid att beräkna än åldern för universum** självt

Lunch.sleep(60);

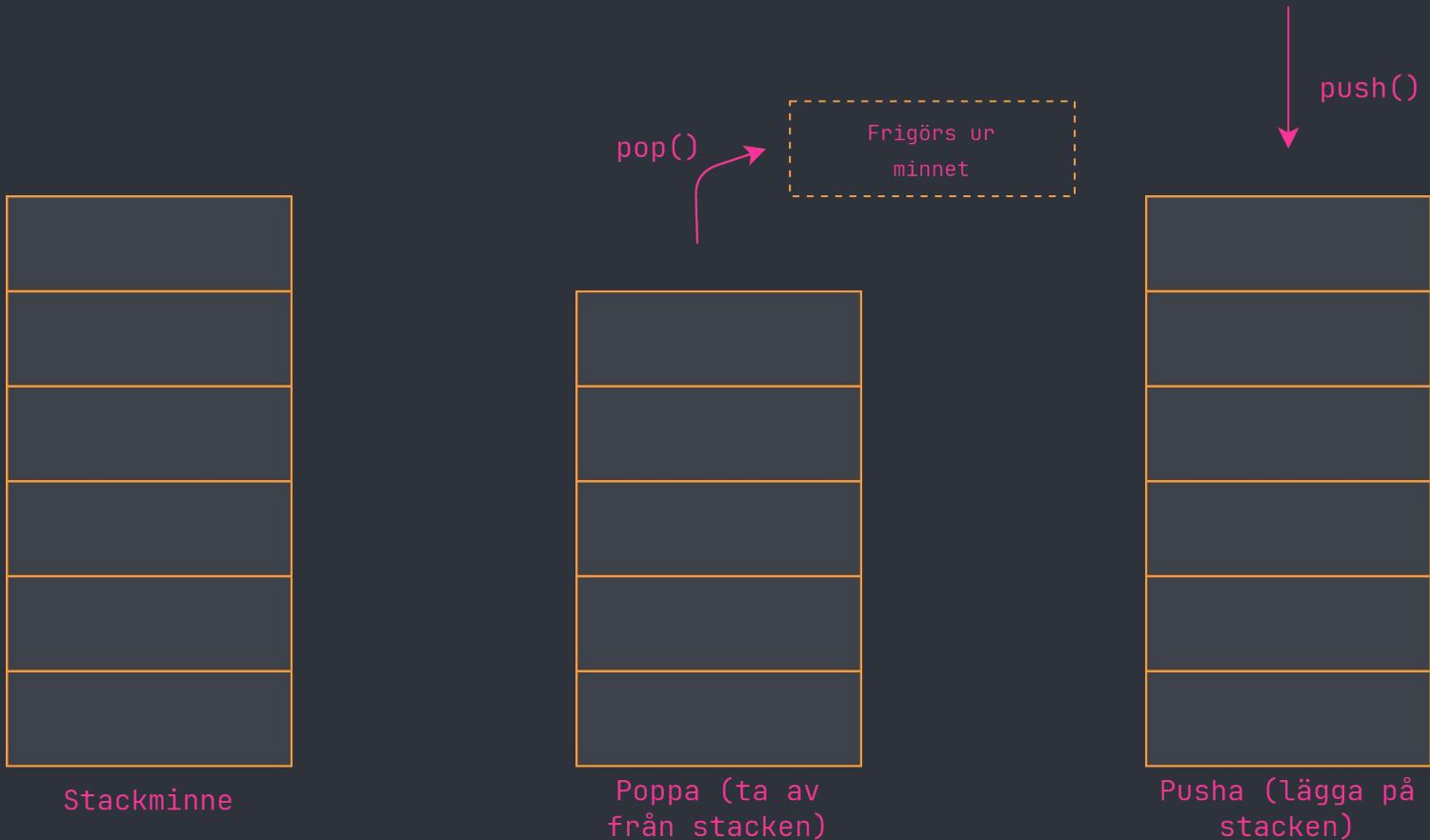
Vi samlas igen 13.00

(ingen akademisk kvart efter lunch – kom i tid)

Minneshantering i Java

- För att förstå vad som händer med en rekursiv fibonacci behöver vi förstå hur minne fungerar i Java
- Det finns 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**
- Callstacken är liten, heapen är enorm
- När vi säger **“stackminne”** är det callstacken vi syftar till

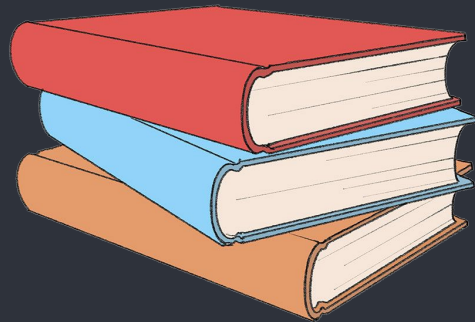
Callstacken ser ut såhär:



Vad lagras på callstacken?

- **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 få åtkomst till

Defaultstorleken i 64-bitarsversionen av JVM är 1 MB (1024 KB)



Activation frames (stack frames)

- Varje gång ett metodanrop görs skapas en **activation frame**, också kallad för **stack frame**, På callstacken
- Typisk storlek mellan **20-100** bytes

Metoden till höger har:

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 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;
}
```

- **36 byte** lagras med andra ord på callstacken när den här metoden anropas
- Arrayen (values) på heapen däremot **kommer innehålla ~80 000 byte**

Stack Overflow: ett problem för rekursiva algoritmer

- Eftersom callstacken är så liten **fylls den upp snabbt**
- Rekursiva algoritmer fortsätter lägga på activation frames på callstacken tills de nått ett basfall
- Om vi försöker beräkna väldigt stora nummer kommer vi att lägga på så många activation frames att stackminnet tar slut, och vi **får då ett Stack Overflow** (“stacköverfyllnad” på svenska)
- Folk påstår ofta att det här är varför rekursiv fibonacci är problematisk, men **det är inte sant**: problemet är att den producerar så många beräkningar (det finns adrig fler activation frames på callstacken samtidigt än storleken på numret vi försöker beräkna)

Referenser

- En **referens** är en variabel **på stacken** som lagrar (“pekar mot”) en annan variabels minnesplats som finns **på heapen**
- Liknar pekare i andra språk. När någon nämner **“referens”** i Java är det egentligen en sorts pekare som man syftar till
- Pekare finns i alla moderna språk, men alla språk låter oss inte hantera dem direkt (detta görs främst i **C** och **C++** där programmeraren har mer kontroll över minne)
- En av de **stora idéerna** 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 fungerar i Java

- I Java är som sagt alla objektvariabler egentligen **referenser** som pekar mot data som finns på **heapen**
- När vi t.ex. skapar en **ArrayList** så lagras den på heapen, medan **referensen** till den hålls på stacken
- Vanliga **integers** som deklareraras sparas däremot på stacken

Programkod:

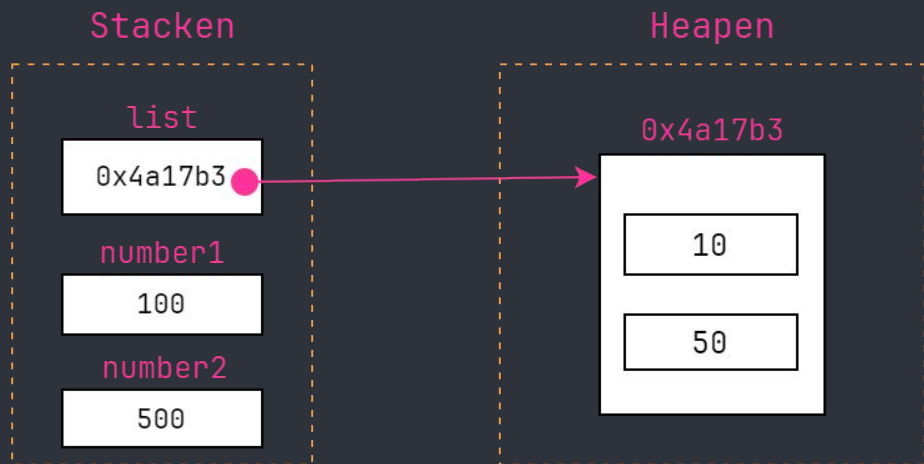
```
ArrayList<Integer> list = new ArrayList<>();
```

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

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

```
int number1 = 100;
```

```
int number2 = 500;
```



Varför NullPointerException är en mardröm

- Null är **defaulttillstånd** för de flesta objekt som inte har initialiserats i Java: vi får då en referens som inte pekar mot någonting på heapen

(Det här är varför ni **alltid** ska initialisera era variabler **via konstruktorn** så att ni är säkra på att ett skapat objekt är i ett giltigt tillstånd)

- **NullPointerExceptions** upptäcks bara under körtid och är svåra att hitta ursprunget för (mardröm i program med tusentals kodrader)
- De är ett så stort problem i Java att man uppfann ett helt nytt språk, **Kotlin**, för att komma runt det
- Tony Hoare som introducerade nullreferenser 1965 har själv kallat dem för **“the billion dollar mistake”**

Exempel 1: kompileringsfel

```
public class Main
{
    public static void main(String[] args)
    {
        String string;

        System.out.println(string.length());
    }
}
```

Variable 'string' might not have been initialized

Initialize variable 'string' Alt+Shift+Enter

More actions... Alt+Enter

String string

NullPointerExample

- Här försöker vi anropa en metod på en oinitialiserad sträng varpå IntelliJ ger oss ett felmeddelande och vägrar låta oss kompilera

Exempel 2: kompilering trots nullpekare

```
public class Main
{
    public static void main(String[] args)
    {
        String string = getStringValue();

        System.out.println(string.length());
    }

    public static String getStringValue()
    {
        return null;
    }
}
```

- Koderna går att kompilera trots att vi gör exakt samma sak, men IntelliJ misstänker bus

Exempel 3: kompilatorn är bortfintad

```
public static void main(String[] args)
{
    ArrayList<String> list = getStringList();

    System.out.println(list.get(0));
}

public static ArrayList<String> getStringList()
{
    ArrayList<String> list = new ArrayList<>();
    list.add(null);

    return list;
}
```

- Här har IntelliJ inte en susning om att något skumt försiggår längre *trots* att vi hårdkodar in null i en lista

Metoder: Pass by reference vs Pass by value

- Data skickas in i metoder på två vis i Java: antingen kopieras värdet eller också skickas en referens till det
- 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`, `long`, 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 ValuesVsReferences.java i repositoryt

Callstacken möjliggjorde dynamisk allokering av minne

- Utan uppdelningen mellan stacken och heapen hade vi inte kunnat allokera minne dynamiskt
- **Lisp** var det första högnivåspråket tillsammans med Fortran, och även det första språket som hade en callstackliknande struktur
- Subrutiner kunde plötsligt använda parametrar för att ta emot data och hade returvärden: **vi fick våra första metoder**
- Lisp är **naturligt rekursivt**: alla datastrukturer använder rekursion som kontrollflöde

Innan dynamisk minnesallokering var program begränsade

Assemblerprogram (statisk minnesallokering):

```
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
RFS            ; 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 använder den samma variabler

Callstacken är varför moderna programmeringsspråk fungerar som de gör

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

```
kaffepaus(15);
```

Recap: två motsatta algoritmer

- Vi har spenderat det mesta av dagen med att titta på två algoritmer som på sätt och vis är varandras exakta motsatser
- En binär sökning **halverar** problemet i varje nytt rekursivt anrop medan en rekursiv fibonaccialgoritm **dubblar** problemet

Vi skulle även kunna sammanfatta dem så här:

*** Binär sökning** har en logaritmisk tillväxt i **effektivitet**:
Den blir mer effektiv ju större datamängden är

*** Rekursiv fibonacci** har en exponentiell tillväxt i **ineffektivitet**:
Den blir (snabbt!) mer ineffektiv ju större datamängden är

- Vi nämnde att en binär sökfunktion var ett exempel på en så kallad **Divide-and-conquer**-algoritm, men vad kallar man fibonacci?

Naiva algoritmer

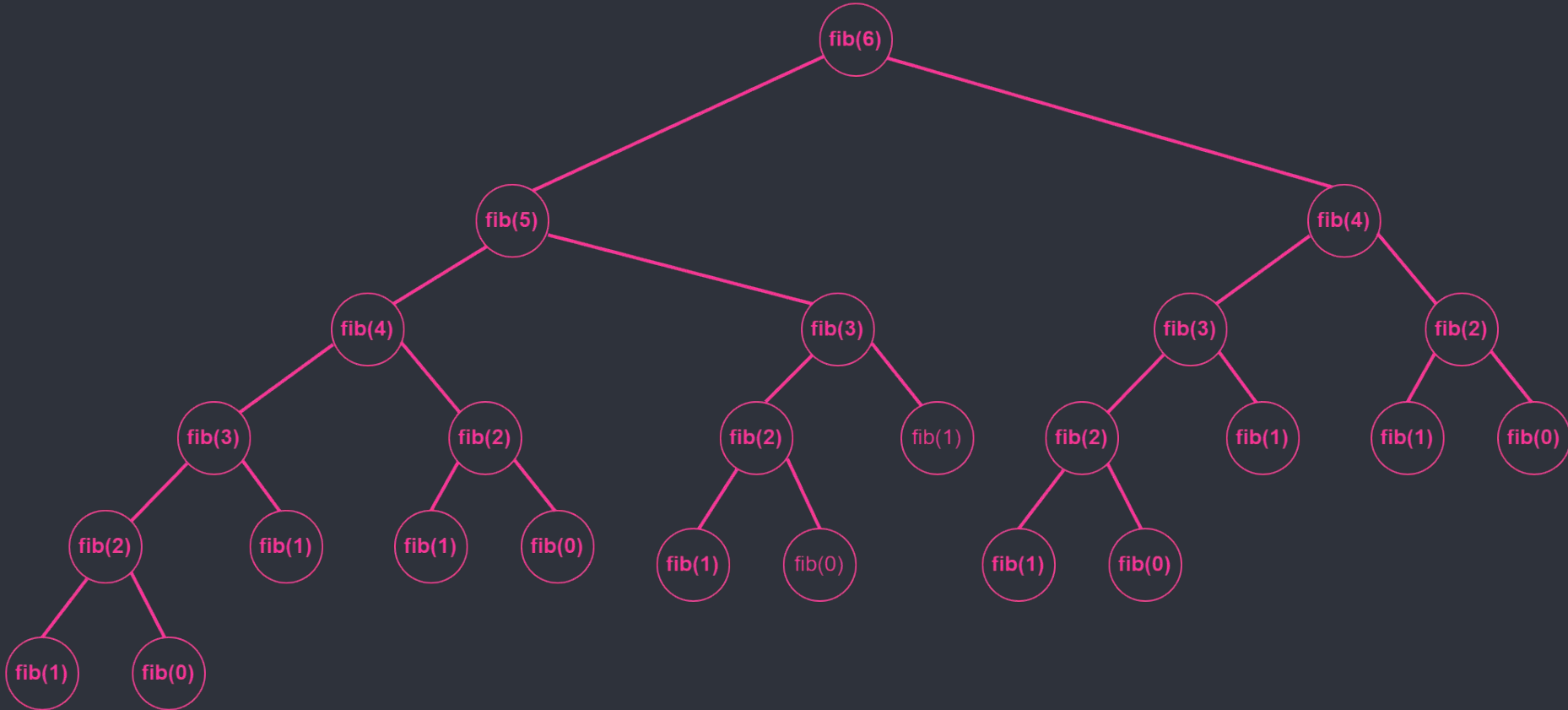
- En **naiv eller dum algoritm** är den simplaste lösningen för ett problem, men sällan den mest effektiva (det finns dock undantag!)
- Den rekursiva versionen av fibonacci som vi skapade i förmiddags är en **naiv algoritm**

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

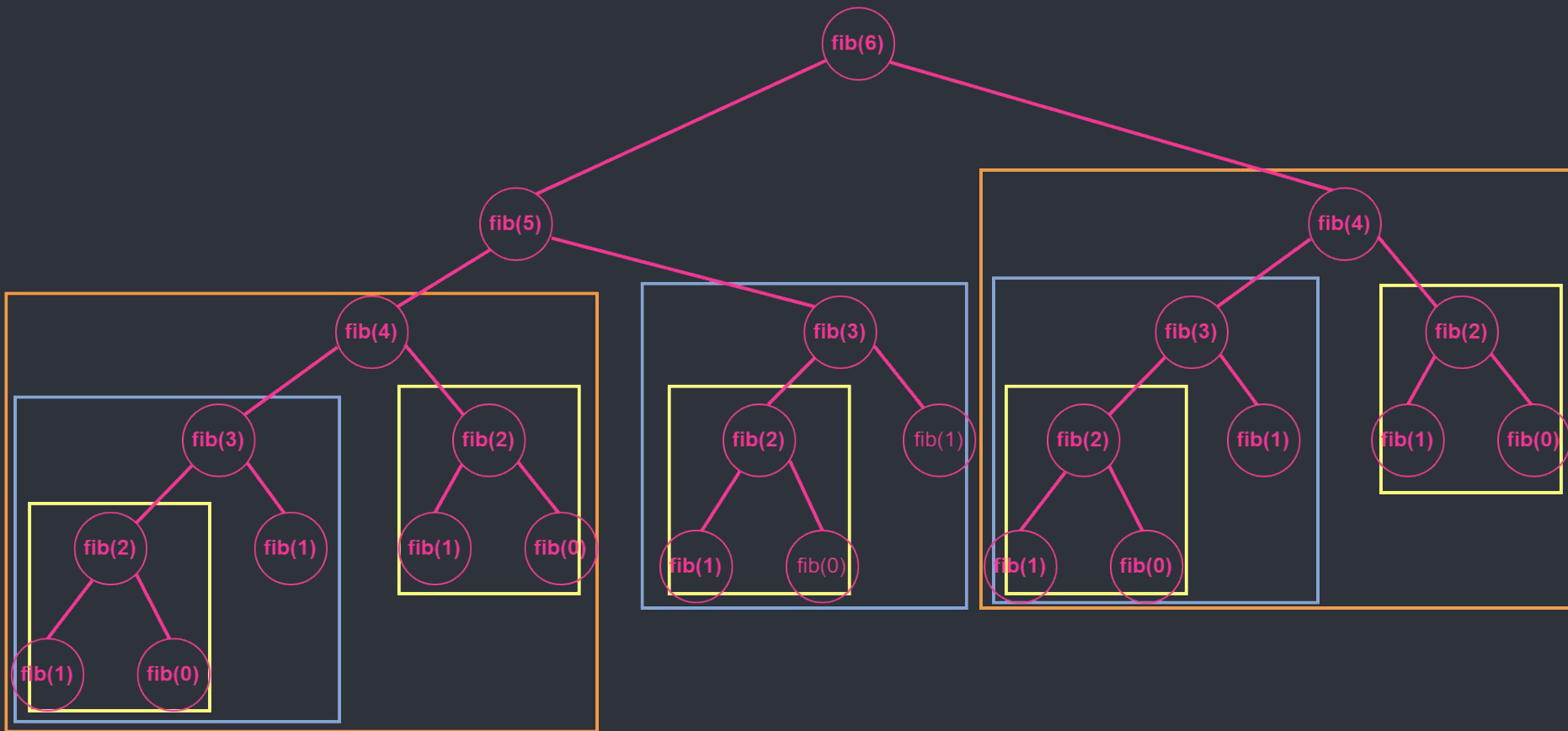


- Naiva algoritmer är ofta **antimönster** (eng. anti-patterns). Antimönster är dåliga designlösningar som får negativa konsekvenser i längden
- Kan vi göra en smart fibonacci i stället?

Q: Vad är problemet här?



A: Vi beräknar varje nummer flera gånger



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 (**så kallad 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?

Dynamisk programmering

- Det finns en optimeringsteknik som kallas för **dynamisk programmering** som kan hjälpa oss med vårt problem
- Dumt namn, har ingenting att göra med datorprogrammering: begreppet är matematiskt och myntades redan på 1940-talet
- Det finns två typer av dynamisk programmering:

Memoisering (Memoization)

Top-down-approach

Rekursiv approach

Tabulering (Tabulation)

Bottom-up-approach

Iterativ approach

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

[Kodexempel]

Koden finns i klassen `FibonacciIterative.java` i repot

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 exekverar, utan **hur tiden påverkas av datamängden**
- I det iterativa fallet dubblas tiden när antalet n dubblas: 20 loopvarv om $n = 20$, 40 loopvarv om $n = 40$, osv. 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 ($n < 100$). **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**, och vi får ett **overflow**
- Använd alltid **long** som data- och returtyp i stället för **int** när stora beräkningar eller ackumuleringar behövs
- 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 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 `FibonacciDynamicRecursive.java` i repot

Summaring: Vad skiljer rekursion från iteration?

- 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**
- Rekursion är kraftfull eftersom den **naturligt speglar** hur strukturerna ser ut för många verkliga problem och fenomen

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 när vi pratar om **träd** och **grafer**)
- **Matematiska algoritmer** använder ofta rekursion för komplexa beräkningar.
Exempel: **Största gemensamma nämnare** (Greatest Common Divisor), **Exponentiering via kvadrering** (Exponentiation by Squaring), osv

Sammanfattning av dagen, del 1

Rekursiva mönster

- Finns i naturen
- Metoder som anropar sig själva
- Behöver ett basfall

Binär sökfunktion

- Divide-and-conquer
- $O(\log n)$

Rekursiv fibonacci

- Exponentiell tidskomplexitet, $O(2^n)$
- Varje gång n ökar med 1 dubblas tiden

Stacken

- LIFO (Last in, First Out)
- Lagrar primitiva typer och referenser till objekt

Heapen

- Lagrar själva objekten

Sammanfattning av dagen, del 2

Naiva algoritmer

- Enklaste lösningen på ett problem
- Sällan den bästa

Primitiva datatyper

- Har begränsat med minne
- Använd long för stora beräkningar

Kontrollflöden

- Definierar programmeringsspråk
- Påverkar ofta tidskomplexiteten

Dynamisk programmering

- Optimeringsteknik
- Memoisering eller tabulering