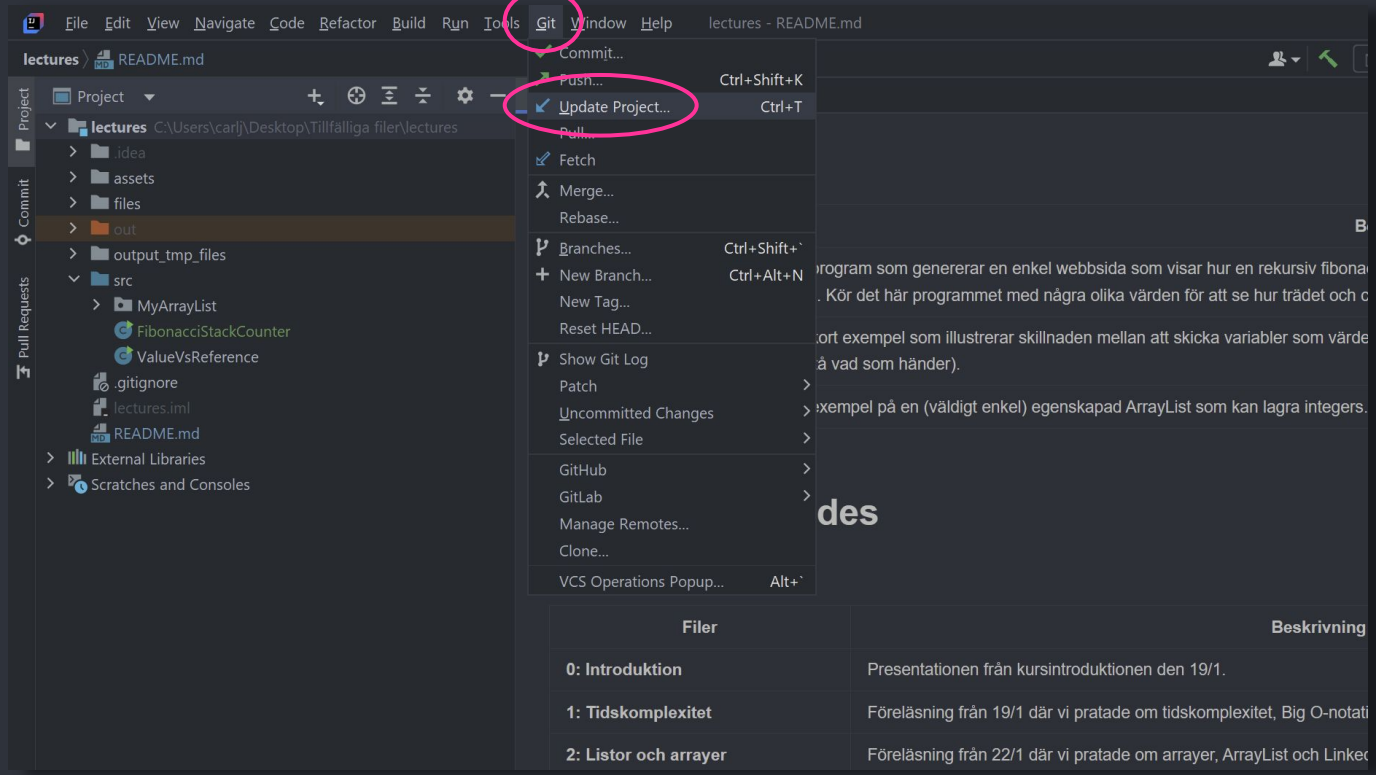
Algoritmer och Datastrukturer

Föreläsning 6

Comparator och
Garbage Collection

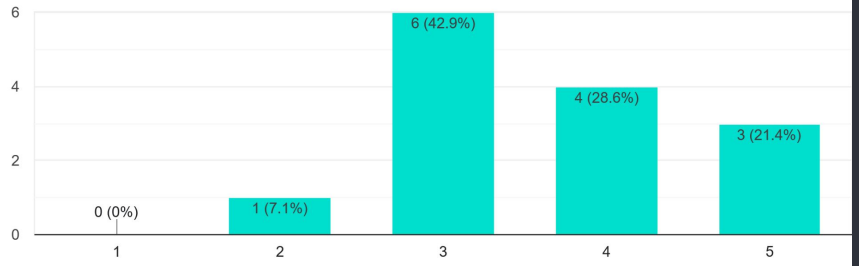
https://github.com/carljohanj/algo



Halvtidsvärdering: resultat

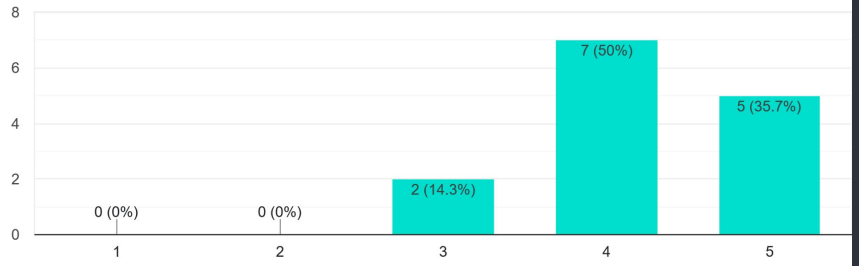
Hur nöjd är du med din egen insats på kursen så här långt?

14 responses



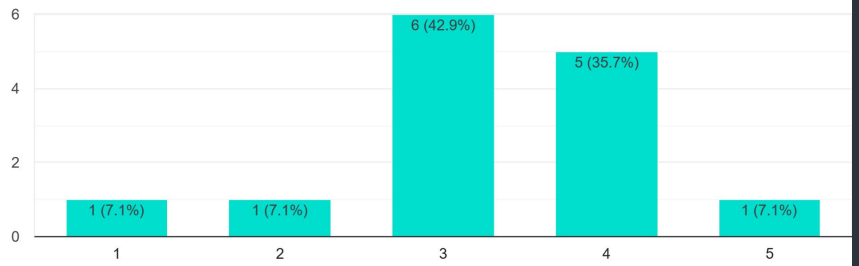
Vad tycker du om undervisningen så här långt?

14 responses



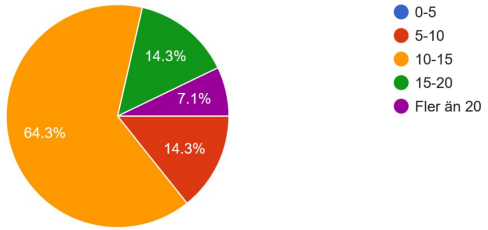
Hur har kursboken varit som komplement till föreläsningarna för att förstå ämnet?

14 responses



Hur många timmar har du (uppskattningsvis) lagt på kursen per vecka?

14 responses



Halvtidsvärdering: resultat

“Det är väldigt bra med kod exempel under föreläsningarna gör att man själv kan testa och se hur det fungerar.”

“Jag HATAR B27”

“Kursboken är betydligt tyngre än den vi hade under Java kursen, men detta är förväntat. Carl-Johans presentationer är som vanligt topp och han är väldigt pedagogisk.”

“Överlag känns kursen bra så här långt. Boken kan vara lite matte tung ibland men den är ett bra kompliment till föreläsningarna som förklara begreppen och algoritmerna på ett ganska enkelt sätt vilket gör att man kan använda boken för att få en djupare förståelse.”

“D24 är en väldigt bra sal för denna kurs, den är trevlig och det finns många uttag att använda”

“D24 funkat bra, b27 är lite lökig sal”

“Har lite svårt och förstå det mycket å göra och jag tror att kursen är lite tungt och svårare än Java kursen”

“Undervisningen är bra men innehållet är lite svårt att förstå.”

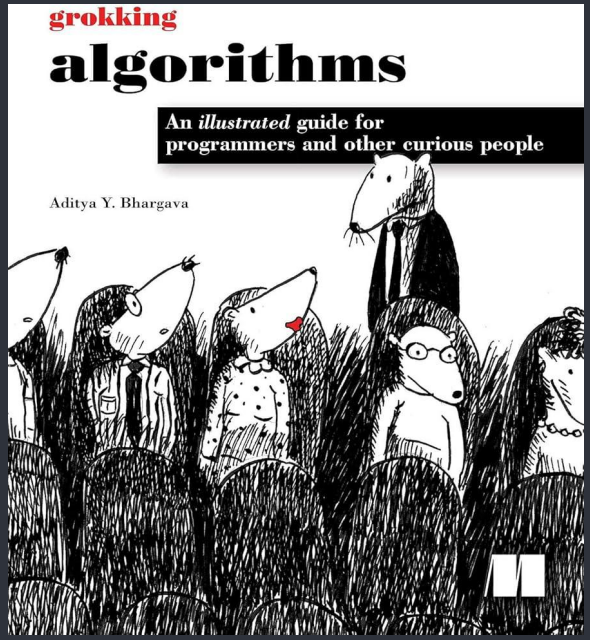
“Skjutit upp inlämmingar för mycket men annars helt ok”

“tydligt och lättförståeligt”

“mycke info per föreläsning :< “

Grokking Algorithms

- Vi kommer att byta till den här som kurslitteratur nästa år
- Jag vågade inte ha en kursbok med kodexempel i ett annat programmeringsspråk än det vi använder, men Python är ju å andra sidan i praktiken redan pseudokod
- Ni kan läsa den gratis om ni loggar in med era studentkonton på O'Reilly: https://learning.oreilly.com/library/view/grokking-algorithms-second/9781633438538/
- Tycker ni att kursboken är tung så titta på den här och se om den kan hjälpa er: själva texten är mycket pedagogisk och lättläst



Lite repetition!

Platskomplexitet

- Vi nämnde **platskomplexitet (“space complexity”** på engelska) senast: det här är ett mått på hur minneseffektiv en datastruktur eller algoritm är. Precis som tidskomplexitet kan vi uttrycka den i Big O-notation

- **Tidskomplexitet är ett mått på hur körtiden för en algoritm ökar i förhållande till datamängden**

- **Platskomplexitet är ett mått på hur minnesanvändningen ökar i förhållande till datamängden**

- Platskomplexitet är främst någonting man talar om i system där RAM-minnet är begränsat (t.ex. Internet of Things-saker), eller då datamängderna är så pass stora att den extra overhead som en mellanliggande datastruktur resulterar i blir helt oacceptabel

- Låt oss titta på ett par exempel:

Platskomplexitet: O(n)

- Vi skapar en ny array (newArray) inuti metoden med samma längd som originalarrayen: det här innebär att alla platser i newArray initialiseras till 0 och vi får O(n) i tidskomplexitet för den här operationen

- Men även platskomplexiteten är O(n) eftersom vi skapar en helt ny array! Om storleken på originalarrayen dubblas så dubblas även storleken på newArray, alltså O(n)

```
                                                    Tidskomplexitet      Platskomplexitet

public int[] copyArray(int[] originalArray)
{
   int[] newArray = new int[originalArray.length];     // O(n)              // O(n)

   for (int i = 0; i < originalArray.length; i++)      // O(n)
   {
      newArray[i] = originalArray[i];                  // O(1)
   }

   return newArray;                                    // O(1)
}
```

Platskomplexitet: O(1)

```
public void copyArray(int[] array1, int[] array2)
{
   for (int i = 0; i < array1.length; i++)
   {
      array2[i] = array1[i];
   }
}
```

- **Vi tar bara in två referenser till arrayer som redan existerar:** oavsett hur mycket data dessa arrayer innehåller är referenserna bara 2 x 8 byte
- **Metoden allokerar inget extra minne:** allt den gör är att kopiera värdena i den ena arrayen till den andra
- **O(1) i platskomplexitet eftersom minnesanvändningen aldrig ökar** oavsett hur stora dessa arrayer är
- O(n) i tidskomplexitet eftersom vi loopar genom alla platser i arrayerna

Repetition: tidskomplexitet

– Vad blir tidskomplexiteten?

```
public void doSomething(List<Integer> numbers)
{
   int sum = 0;                  // O(1) +

   for (int i : list)            // O(n) *
   {
      sum += i;                  // O(1) +
   }

   for (int i : list)            // O(n) *
   {
      sum += i;                  // O(1)
   }
}
```

Svar: O(n)

Repetition: tidskomplexitet

– Vad blir tidskomplexiteten?

```
public void doSomething(List<Integer> numbers)
{
   int sum = 0;                    // O(1) +

   for (int i : numbers)           // O(n) *
   {
      for (int j : numbers)        // O(n) *
      {
           sum += j;               // O(1)
      }
   }
}
```

Svar: $O(n^2)$

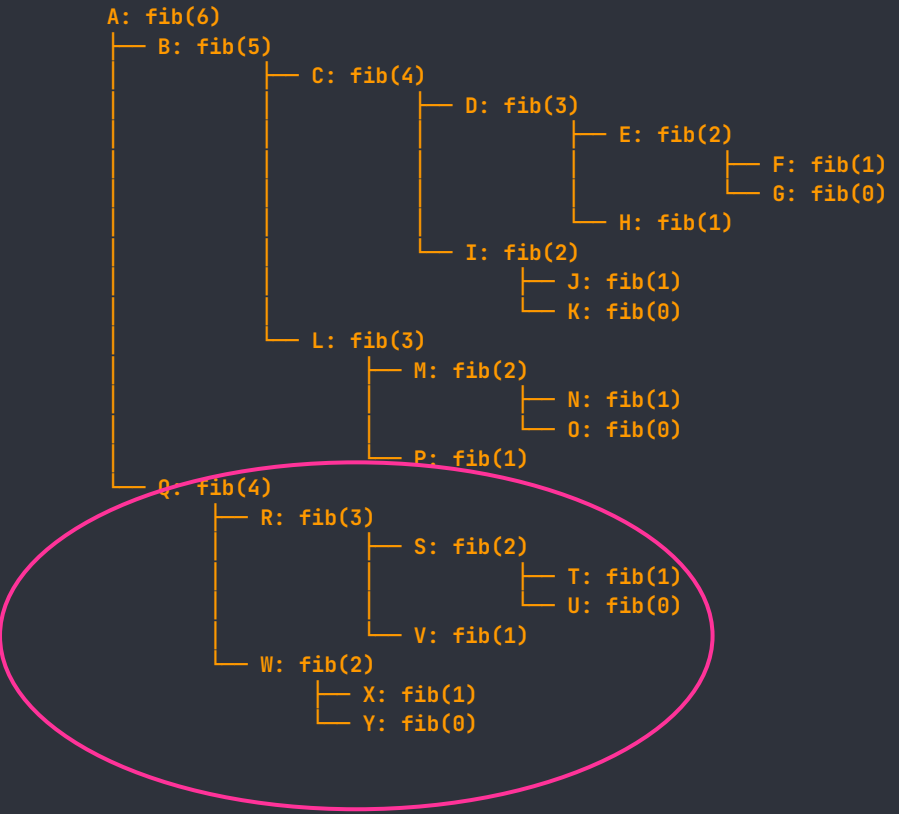
Repetition: Vad blir tidskomplexiteten?

```
public class LoopExample
{
   int sum = 0;                          // O(1) +

   public int calculate(int n)
   {
      for (int i = 0; i < n; i++)        // O(n) *
      {
         for (int j = 0; j < n; j++)     // O(n) *
        {
            increment(n);                // O(n) *
        }
   }

   public void increment(int n)
   {
      if (n == 0) { return n; }          // O(1) +
      sum++;                             // O(1) +
      return increment(n-1);             // O(n)
   }

}
```

Svar: $O(n^3)$

– Här är det viktigt att förstå att även om den inre loopen bara gör ett anrop till increment() så har den metoden också O(n), och vi måste inkludera det i beräkningen

Repetion: fibonacci



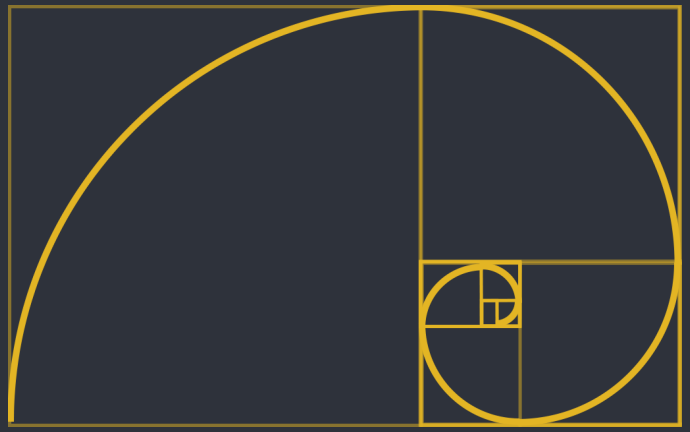
- Om vi kör `FibonacciStackCounter` några gånger med olika värden kan vi notera hur den andra anropskedjan, som används för att beräkna `fib(n-2)`, alltid blir kortare än den första (utom vid de allra minsta värdena för n). Varför är det så?

- Den andra anropskedjan `fib(n-2)` kommer alltid att bli lite kortare **eftersom vi anropar den med ett lägre nummer än `fib(n-1)`.** Det kommer inte behövas lika många anrop för att räkna ut det numret.

Repetion: fibonacci

- **Innebär det här att tidskomplexiteten verkligen är $O(2^n)$ för algoritmen, eller är den egentligen något annat?** Varför tror ni att vi trots det betecknar den som $O(2^n)$ i så fall?

- **Tidskomplexiteten för en rekursiv fibonacci-algoritm är mycket riktigt något lägre än $O(2^n)$ i praktiken.** Detta beror på att inte varje nytt metodanrop genererar två nya anrop: de anrop som når `0` eller `1` först kommer att stanna och sedan returnera sina värden.



- Det uppochnedvända trädet kommer alltid att bli ojämnt i kronan eftersom den högra anropskedjan startar med ett lägre värde och därför aldrig går lika djupt som den vänstra. Varje nytt fibonaccianrop som görs genererar i genomsnitt ~1.6 nya anrop.

- Faktum är att **förhållandet mellan anropen exakt speglar förhållandet mellan fibonaccinumren själva:** 1.618, också känt som det gyllene snittet, som betecknas **φ**

Repetion: fibonacci

- Anledningen till att vi ändå uttrycker tidskomplexiteten som $O(2^n)$ i stället för t.ex. $O(\varphi^n)$ eller $O(1.618^n)$ är för att det **dels är lättare att förstå, men även för att ”Big O”-notationen är intresserad av den övre gränsen,** som algoritmen aldrig kommer att överskrida

- **Takeaway:** Tidskomplexitet är inte menad att vara ett exakt värde utan att beskiva en tendens. Tidskomplexiteten är fortfarande exponentiell för en rekursiv fibonaccialgoritm, och för tydlighetens skull betecknar vi den likadant för alla exponentiella algoritmer

- **Vill vi vara matematiskt korrekta kan vi inkludera decimalerna,** men det finns inte många sammanhang där det är intressant: 1.6^n och 2^n är lika oanvändbara för $n > 60$

Dagens föreläsning: att sortera objekt

Att sortera objekt

- Vi kan lätt sortera primitiva datatyper såsom integers eftersom de går att jämföra direkt (“Är 3 större än 2?”), men **hur sorterar vi egentligen objekt?** De är ju inga primitiva typer!

- För att göra det här behöver vi ta hjälp av så kallade komparatorer.

- Vi kan sortera objekt på två vis i Java: antingen med hjälp av Något som heter Comparator, eller också genom att implementera ett generiskt interface Som heter Comparable

- **Jag rekommenderar att använda Comparator** eftersom det **är enklare, mer modernt och ett bättre verktyg** att använda sig av i majoriteten av de situationer som uppstår

- Comparable är dock bra att ha koll på, och har fördelar om man t.ex. designar återanvända API:n för externa användare

- Vi ska titta lite kort på båda i dag

Comparator

```
@FunctionalInterface
public interface Comparator<T> {
```

> Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

- Definierat utanför klassen vi använder det för (vi behöver inte skriva en egen implementation för hur något ska sorteras)
- Kan användas för att sortera på flera olika sätt
- Stöds av alla samlingsdatatyper som har sortering: listor, arrayer, TreeSet, TreeMap, Streams, osv

[Kodexempel]

Koden finns i paketet ComparatorExample

Metodreferensoperatorn

- Någonting händer på den här raden i vår kod:

```
list.sort(Comparator.comparingInt(Person::getAge));
```

- Dessa dubbelkolon **kallas för metodreferensoperatorn**

- Vi använder den här operatorn när vi vill berätta för koden att den ska använda en viss metod - i det här fallet metoden getAge() i klassen Person - för att göra något. Vi kan inte skriva så här:

```
list.sort(Comparator.comparingInt(Person.getAge()));
```

- ...eftersom det i så fall **skulle anropa metoden getAge() direkt på den här raden.** Vi vill bara att comparingInt() ska använda den när den gör sina inre jämförelser! Vi **refererar därför till metoden** vi vill att den ska anropa när det är dags att göra så

Comparable

- Ännu ett (lite äldre) sätt att implementera sortering på i Java
- **Comparable är ett generiskt interface som vi måste förlänga en klass med** (vi måste alltså för det första kunna modifiera typen som vi vill sortera!), och vi behöver då definiera en metod som heter `compareTo()`
- **Det är sedan compareTo() som Collections.sort() och Arrays.sort() anropar internt** för att skilja objekt åt när vi ber dem sortera en samling åt oss

```
public interface Comparable<T> {

    Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer
    as this object is less than, equal to, or greater than the specified object.
```

- Definieras inuti klassen självt (vi måste skriva implementationen)
- Kan bara finnas en compareTo() i en klass (men vi kan ändå overrida den med Comparator)

Comparable

```
public class Person implements Comparable<Person>
{
      private String name;
      private int age;

      public Person(String name, int age)
      {
         this.name = name;
         this.age = age;
      }

      public int compareTo(Person other)
      {
         return Integer.compare(other.age, this.age);
      }
}
```

Här anger vi typen som vi ska jämföra med (vi vill jämföra Personer med Personer)

Den här metoden måste finnas med om en klass implementerar Comparable

Ett bättre och säkrare sätt att jämföra än att göra tre if-checkar (för större än, mindre än eller lika med)

Comparator vs comparable

- Om vi använder Comparator så anropar vi `sort()` direkt på listan om samlingen är en lista med objekt, eller `Arrays.sort()` om det är en array med objekt

- Den **stora fördelen med Comparator**, förutom att det blir mindre kod, är att **vi kan sortera ALLT så länge vi har ett urvalskriterium:** vi behöver inte ha tillgång till källkoden för klassen (bra om det inte är vi som skrivit den!)

- Vi kan även sortera objekt på flera olika kriterium: **vi kan göra en samling där vi sorterar på namn och en där vi sorterar på ålder, eller också först namn och sen ålder**

- Om vi skriver en implementation för **Comparable säger vi: "Det finns bara ETT sätt att sortera det här objektet på"**. Vi kan inte definiera flera olika `compareTo()`-metoder

- **Kan vara bra att använda Comparable** om det alltid bara finns en enda naturlig ordning, om ordningen är en del av typens identitet (t.ex. LocalDate), eller om du inte har kontroll över hur användare sorterar och vill implementera ett default-case

Vilken algoritm använder Arrays och List för att sortera objekten?

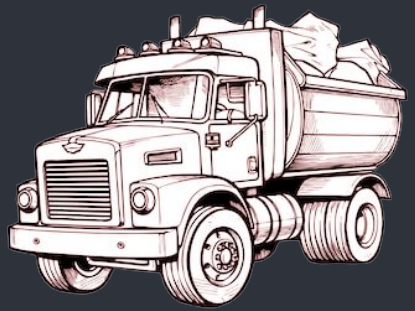
- Både Arrays.sort() och List.sort() **använder TimSort internt**
- **TimSort är en hybridalgoritm:** den är snabb och stabil (men inte in-place)
- “Hybrid” eftersom den använder InsertionSort för små samlingar och MergeSort för större datamängder
- TimSort är optimerad för att identifiera “runs” (redan sorterade sekvenser i en samling)
- **O(n*logn) i både värsta, bästa och genomsnittliga fallet** - MEN tids-komplexiteten närmar sig O(n) för samlingar som redan är mestadels sorterade
- **Vi säger fortfarande att den är O(n*logn)**, men detta innebär att TimSort har bättre prestanda än en normal MergeSort

```
kaffepaus(15);
```

Garbage Collection: hur Java frigör minne

- I ett språk som **C++** har man inte bara en konstruktor - man har även en **destruktor** där man kan specificera vad som ska raderas ur minnet när ett objekt förstörs

- Tar man inte bort allokerat minne efter sig kommer det att **ligga kvar** på heapen efter att själva objektreferensen är borta från stacken: **man får då en minnesläcka** och blir arg, och går och skapar Java och programmerar i det i stället



- I Java **sköts minnesrensningen automatiskt** av en bakgrundsprocess som kallas för **garbage collection,** som funkar ungefär som en sopgubbe: när vi samlat ihop skräp kommer sopbilen och hämtar det så att soptunnan (dvs heapen) blir tom igen

Två problem som garbage collection löste

- Modern C++ har bättre rutiner och mönster för minneshantering än man hade på 90-talet: **i dag har vi smart pointers, RAII-rutiner och liknande** som är till för att göra språket säkrare

- **Java löste två enorma problem** med sin automatiska minneshantering när det kom:

  * **Minnesläckor:** när en programmerare glömmer att deallokera minne som inte längre används

  * **Hängande referenser:** när en pekare fortfarande existerar som pekar mot minne som har tagits bort manuellt

- Java erbjöd en lösning som gjorde att **utvecklare kunde fokusera på att bygga applikationer** i stället för att jaga minnesläckor som konsumerade allt ledigt RAM-minne

Garbage collection

- När ett Javaprogram körs skapar vi **objekt i minnet på en plats vi kallar för heapen.** När dessa objekt sedan inte längre används blir de “skräp” som upptar värdefullt minne utan att längre ha något syfte

- **Garbage collectorn är ett bakgrundsprogram** som regelbundet scannar av minnet och ser vilka objekt som fortfarande har levande referenser: dessa stannar kvar i minnet

- Objekt som saknar en referens, däremot, blir märkta för borttagning

Här hamnar dina gamla arrayer! ⟶

Eden Space

- Alla nya objekt i Java allokeras i en region av minnet som JVM kallar för **“Eden”, eller “Eden space”**
- Små nyfödda strängbebisar springer runt här och dansar i ring
- **Om en region överlever en GC-cykel så flyttas den till “Survivor Space”**, och om dessa överlever fler cykler flyttas de så småningomtill en “Old region” för långlivade objekt
- **De flesta objekt överlever inte Eden** utan stannar här tills de dör
- Ett bättre namn för Eden med det här i åtanke vore kanske ett bårhus!

[Kodexempel]

Koden finns i GarbageCollection.java

Garbage-loggen

– Loggen visar att garbage collectorn körs två gånger: en efter vi skapat byte-arrayen, och ännu en gång efter att vi plockat bort referensen till den.

```
Program started
Creating big object:

[0.064s][info][gc,heap] GC(0) Eden regions: 4→0(53)
[0.064s][info][gc,heap] GC(0) Survivor regions: 0→1(6)
[0.064s][info][gc,heap] GC(0) Old regions: 2→2
[0.064s][info][gc,heap] GC(0) Humongous regions: 0→0

Big object reference removed
Requesting garbage collection:

[2.098s][info][gc,heap] GC(2) Eden regions: 1→0(53)
[2.098s][info][gc,heap] GC(2) Survivor regions: 1→0(6)
[2.098s][info][gc,heap] GC(2) Old regions: 2→3
[2.098s][info][gc,heap] GC(2) Humongous regions: 101→0

Program finished
```

Generation 1

Generation 2

Garbage collection, generation 1

- Innan vårt javaprogram någonsin kör en enda rad användarskriven kod händer en mängd grejer så fort vi trycker på **Run**:
  * JVM kommer att ladda in kärnklasser, såsom `java.lang.Object`, `java.lang.String`, `java.lang.System` och `java.lang.Thread`
  * JVM initialiserar huvudtråden för programmet, skapar interna buffrar för data, osv
  * Genererar metadata för kompilatorn, skapar stringliteraler, allokerar hjälpobjekt, etc
- **De flesta av de här objekten lever i Eden Space** eftersom de inte är långvariga, och det är därför vanligt att GC:n aktiveras direkt i början av ett program för att rensa bort dem
- Det är detta som vi ser hända på första loggraden:

  ```
  [0.064s][info][gc,heap] GC(0) Eden regions: 4→0(53)
  ```

- Innan GC kördes användes 4 Eden-regioner, nu används 0 (av tot. 53)

Garbage collection, generation 1

- Nästa rad i loggen ser ut så här:

  ```
  [0.064s][info][gc,heap] GC(0) Survivor regions: 0→1(6)
  ```

- Det här berättar för oss att en region överlevde från Eden Space och flyttades till “överlevarregionen” där långlivade objekt håller till

- **Den mest sannolika kandidaten för den här regionen är temporära buffrar som skapades för våra strängar som säger “Program started” och “Creating big object”**

- Rad 3 och 4 i loggen säger:

  ```
  [0.064s][info][gc,heap] GC(0) Old regions: 2→2
  [0.064s][info][gc,heap] GC(0) Humongous regions: 0→0
  ```

- Den första av de här två raderna indikerar att det fanns två “gamla regioner” innan GC:n kördes, och det finns fortfarande två kvar efteråt. Den andra av raderna säger att inga enorma regioner har skapats i minnet än så länge

Garbage collection, generation 2

- Vi sätter nu objektreferensen (byte-arrayen) till null och säger åt systemet att det borde komma och hämta soporna

- Garbage collectorn kör en andra gång, och vi får följande inledande loggrad:

  ```
  [2.098s][info][gc,heap] GC(2) Eden regions: 1→0(53)
  ```

- Det här indikerar att den andra GC:n kördes 2.098 sekunder efter att programmet startades , och att den rensade bort ett kortlivat objekt från Eden

- Nästa två rader:

  ```
  [2.098s][info][gc,heap] GC(2) Survivor regions: 1→0(6)
  [2.098s][info][gc,heap] GC(2) Old regions: 2→3
  ```

- ... säger att ett överlevarobjekt från förra generationen inte längre behövdes och rensades ur minnet, samt att det skapades ännu en “Old Region” där långlivade objekt placeras.

Garbage collection, generation 2

- Vi kommer nu till fjärde och sista raden i loggen för den andra hämtningen, och här kan vi se att något stort har tagits bort:

```
[2.098s][info][gc,heap] GC(2) Humongous regions: 101→0
```

- **Det här är nyckelraden i loggen.** Här ser vi att 101 “humongous” regioner har tagits bort! En sådan här “enorm region” motsvarar ungefär en megabyte, vilket stämmer bra överens med den förväntade storleken på vår byte[]-array!

- Efter GC:n städat upp finns det 0 “humongous regions” kvar i bruk, och vi har därmed ett kvitto på att garbage collectorn gjort sitt jobb

Varför garbage collection är viktig att förstå

- Vi **pratar egentligen bara om algoritmisk optimering** på den här kursen, och garbage collection påverkar inte Big O-notationen
- **Men i system där frames per second (spel och bildrendering), latens (servrar) och jitter (realtidssystem) spelar GC-avbrott mycket större roll än ren hastighet**
- En långsammare algoritm som inte gör några stora allokeringar av minne kan vara vida överlägsen en snabbare som inte är in-place
- Man vill även undvika allokeringar i loopar och liknande så långt det är möjligt
- Det kan vara bra att ha det här i åtanke om man någon gång jobbar med **prestandakritiska system eller** kanske gör **egna projekt där grafikprogrammering är viktigt**

Att designa runt GC

- **Java är ett snabbt programmeringsspråk nuförtiden,** men om man t.ex. jobbar i OpenGL eller liknande behöver man tänka på att GC:n körs ibland i bakgrunden

- Uppdaterar man skärmen 60 gånger i sekunden spelar även små Eden-allokeringar stor roll eftersom det introducerar en massa små avbrott: 60-100 ms räcker för att det ska se ut som lagg på skärmen

- Man kan designa runt det här med buffertar och cachar för objekt

- **LWJGL är ett bibliotek med OpenGL/Vulcan-bindings** för Java och Kotlin som är jättebra (https://www.lwjgl.org/)

- Det är designat för att allokera vissa minnesresurser utanför heapen med ByteBuffertar som garbage collectorn inte kan röra (farligt att använda om man inte vet vad man gör!)

Project Valhalla

- **Det som kostar i programmering är heap-allokeringar:** de är långsammare (vi tvingas jaga referenser på heapen) och innebär att Garbage Collection alltid kommer köras när saker deallokeras
- **Även små objekt som strängar innebär GC-tryck i Java**
- Projekt Valhalla är en JDK-utveckling i Java som försöker optimera Språket så att vi får små objekt som inte behöver heap-allokeras
- Det gör många fler saker: man kan läsa vidare på nätet om man är Intresserad, t.ex. här
- **Kommer vara en revolution för Java när det är färdigt** (det finns lite preview-features i Java 26 som släpps nu i Mars)

I eftermiddag

- Vi har samma rum (D22) efter lunch
- **Labbpass mellan 13.15 - 15.** Kom och redovisa laborationer, jobba med uppgifterna på kurssidan, ställ frågor om sånt ni undrar över
- Jag tar med mig Sara så att redovisningarna går lite fortare
- **Kom ihåg att göra Dugga 1 innan den stänger** (sista datum 15 februari)
- Jag låser upp Labb 5 i eftermiddag om någon vill börja kika på den, men den ska inte lämnas in på flera veckor så känn ingen panik: **om ni vill vara i fas med kursen bör ni försöka göra färdigt Labb 3 den här veckan**