Sökning och sortering

Programmering för språkteknologer 2

Sara Stymne

2013-09-16

Idag

- Sökning
- Analys av algoritmer – komplexitet
- Sortering

Vad är sökning?

- Sökning innebär att hitta ett värde i en samling av värden
- Vi kommer att fokusera på att hitta ett värde i en array
- Returvärden
  - Om värdet finns: index för den plats där det finns
  - Om värdet inte finns: -1

Vad är sökning?

- Sökning innebär att hitta ett värde i en samling av värden
- Vi kommer att fokusera på att hitta ett värde i en array
- Returvärden
  - Om värdet finns: index för den plats där det finns
  - Om värdet inte finns: -1 (omöjligt som index)

Vad är sökning?

- Sökning innebär att hitta ett värde i en samling av värden
- Vi kommer att fokusera på att hitta ett värde i en array
- Returvärden
  - Om värdet finns: index för den plats där det finns
  - Om värdet inte finns: -1 (omöjligt som index)
- Tillvägagångssätt beror på arrayen
  - Osorterad
  - Sorterad

Vad är sökning?

- Sökning innebär att hitta ett värde i en samling av värden
- Vi kommer att fokusera på att hitta ett värde i en array
- Returvärden
  - Om värdet finns: index för den plats där det finns
  - Om värdet inte finns: -1 (omöjligt som index)
- Tillvägagångssätt beror på arrayen
  - Osorterad – Linjär sökning
  - Sorterad – Binär sökning

Linjär sökning

- Hitta ett värde i en osorterad array
- Lösning: Loopa igenom arrayen tills vi hittar värdet, eller tills vi letat igenom hela arrayen om det inte finns

Linjär sökning

- Hitta ett värde i en osorterad array
- Lösning: Loopa igenom arrayen tills vi hittar värdet, eller tills vi letat igenom hela arrayen om det inte finns

```
public static int linSearch(int[] arr, int target) {
  for (int i=0; i < arr.length; i++) {
     if (arr[i]==target) {
        return i;    // first hit index
     }
  }
  return -1
}
```

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
  - I värsta fall?
  - I genomsnitt?

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
  - I genomsnitt?

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
    - n – värdet finns ej, alla positioner måste letas igenom
  - I genomsnitt?

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
    - n – värdet finns ej, alla positioner måste letas igenom
  - I genomsnitt?
    - $n/2$ – om vi bara letar efter värden som finns

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
    - n – värdet finns ej, alla positioner måste letas igenom
  - I genomsnitt?
    - $n/2$ – om vi bara letar efter värden som finns
    - $n/2 < t < n$ – om vi även letar efter värden som inte finns

Hur lång tid tar linjär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayens storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
    - n – värdet finns ej, alla positioner måste letas igenom
  - I genomsnitt?
    - $n/2$ – om vi bara letar efter värden som finns
    - $n/2 < t < n$ – om vi även letar efter värden som inte finns
    - Beror även på fördelningen av värden i arrayen

Binär sökning

- Om arrayen är sorterad kan vi utnyttja det för snabbare sökning

Binär sökning

- Om arrayen är sorterad kan vi utnyttja det för snabbare sökning
- Leta i mitten först
  - Om det är rätt värde, returnera
  - Om värdet är mindre än mittenvärdet, leta i den undre delen
  - Om värdet är mindre än mittenvärdet, leta i den övre delen
- Upprepa!

Binär sökning – rekursiv

```
public static int binSearch(int[] arr, int target) {
   return binSearch(arr,target,0,arr.length-1);
}

public static int binSearch(int[] arr, int target,
                            int low, int high) {
   if (high < low) {
     return -1;
   } else {
      int mid = (low+high)/2;
      if (arr[mid] == target) {
         return mid;
      } else if (arr[mid] > target) {
         return binSearch(arr,target, low, mid-1);
      } else {  // (arr[mid] < target)
         return binSearch(arr,target, mid+1, high);
      }
   }
}
```

Rekursion

- Rekursiv funktion – funktion som anropar sig själv
- En rekursiv funktion har två typer av fall:
  - Basfall – fall där lösningen är trivial
  - Rekursiva fall – fall där metoden anropar sig själv

Rekursion

- Rekursiv funktion – funktion som anropar sig själv
- En rekursiv funktion har två typer av fall:
  - Basfall – fall där lösningen är trivial
    - Binär sökning: -1 om svaret inte finns, samt korrekt index om vi hittat rätt värde
  - Rekursiva fall – fall där metoden anropar sig själv
    - Binär sökning: sök vidare i ena halvan genom att anropa sig själv

Binär sökning – iterativ

```
public static int binSearch(int[] arr, int target) {
   return binSearch(arr,target,0,arr.length-1);
}

public static int binSearchLin(int[] arr, int target,
                               int low, int high) {
   while (high >= low) {
      int mid = (low+high)/2;
      if (arr[mid] > target) {
         high = mid - 1;
      } else if (arr[mid] < target) {
         low = mid + 1;
      } else {    //arr[low] == target
         return mid;
      }
   }
   return -1;
}
```

Hur lång tid tar binär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayen storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
  - I värsta fall?
  - I genomsnitt?

Hur lång tid tar binär sökning?

- Räkna tiden i antal jämförelser (==) som görs
- Baserat på arrayen storlek: n
- Hur lång tid tar sökningen:
  - I bästa fall?
    - 1 – värdet finns först i arrayen
  - I värsta fall?
    - $\log n$ – värdet finns ej
  - I genomsnitt?
    - $1 < t \leq \log n$

Sökning – jämförelse

	Linjär	**Binär**
Kan användas	alltid	för sorterad array
För länkad lista	ja	nej
Tid (värsta)	n	log n
Implementering	extremt lätt	något krångligare

Sökning – alternativ

- Linjär och binär sökning fungerar bra för arrayer
- Man kan dock använda alternativa datastrukturer för att underlätta sökning:
  - Binära sökträd
    - Alternativ datastruktur till array
  - Hashtabeller
    - Lookup ur en hashtabell är i regel snabbt

Algoritmer och datastrukturer

- Algoritm
  - ”en systematisk procedur som i ett andligt antal steg anger hur man utför en beräkning eller löser ett givet problem” (NE)
  - En beskrivning av hur man löser ett problem
- Datastruktur
  - En struktur som används för att lagra data, till exempel länkad lista, array, hashtabell

Algoritmer och datastrukturer

- Algoritm
  - ”en systematisk procedur som i ett andligt antal steg anger hur man utför en beräkning eller löser ett givet problem” (NE)
  - En beskrivning av hur man löser ett problem
- Datastruktur
  - En struktur som används för att lagra data, till exempel länkad lista, array, hashtabell
- Vilka algoritmer som är tillämpbara för ett problem beror delvis på vilken datastruktur som används

Analys av algoritmer

- Viktiga frågor att besvara om ett program/algoritm
  - Ger den ett korrekt resultat?
  - Ger den alltid lösning? (fastnar ej i oändlig loop, tex)
  - Hur snabb är den?
  - Hur mycket minne kräver den?

Analys av algoritmer

- Viktiga frågor att besvara om ett program/algoritm
  - Ger den ett korrekt resultat?
  - Ger den alltid lösning? (fastnar ej i oändlig loop, tex)
  - **Hur snabb är den? – tidskomplexitet**
  - Hur mycket minne kräver den?

Tidskomplexitet

- Hur kan man avgöra vilken algoritm som är den snabbaste för att lösa ett ett problem?

Tidskomplexitet

- Hur kan man avgöra vilken algoritm som är den snabbaste för att lösa ett ett problem?
- Att mäta faktisk tid är opraktiskt – olika datorer är olika effektiva till exempel

Tidskomplexitet

- Hur kan man avgöra vilken algoritm som är den snabbaste för att lösa ett ett problem?
- Att mäta faktisk tid är opraktiskt – olika datorer är olika effektiva till exempel
- Använd **asymptotisk analys** – tendensen på lång sikt

Asymptotisk analys

- Analys av hur tiden för en algoritm växer när storleken på indata växer
- Storlek?
  - Array: antal element
  - Sträng: antal tecken
  - Allmänna fallet: ofta antal byte
- Vi kan diskutera olika fall
  - Värsta
  - Genomsnittliga
  - Bästa

Asymptotisk analys

- Analys av hur tiden för en algoritm växer när storleken på indata växer
- Storlek?
  - Array: antal element
  - Sträng: antal tecken
  - Allmänna fallet: ofta antal byte
- Vi kan diskutera olika fall
  - Värsta – Vanlig analys
  - Genomsnittliga – Görs ofta, men är ofta svårare
  - Bästa – Ofta inte så meningsfull

Asymptotisk analys

- Grundkrav: En algoritm ska fungera för indata av godtycklig storlek (n)
- Beräkna körtiden som en funktion av storleken på indata $T(n)$
- Ignorera konstanta faktorer
- Fokusera på dominerande faktorer vid stora indata

Asymptotisk analys

- Grundkrav: En algoritm ska fungera för indata av godtycklig storlek (n)
- Beräkna körtiden som en funktion av storleken på indata $T(n)$
- Ignorera konstanta faktorer
- Fokusera på dominerande faktorer vid stora indata
- Analysen ska vara maskinoberoende
- Kraftfulla maskiner ökar hastigheten med en konstant

Primitiva operationer

- En primitiv operation antas ta konstant tid:
  - Tilldelning, ex. x = y;
  - Aritmetiska operationer, ex. x+5;
  - Jämförelser, ex. x < 5;
  - Arrayindexeringar, ex. myArray[5];
  - Metodretur, ex. return 5;
  - ...
- Låt T(n) vara antalet primitiva operationer som en funktion av ”storleken på indata”

Exempel – multiplicera talen i en array

```
public static int multiply(int[] arr) {
  int res = 1;
  int i=0;
  for (; i < arr.length; i++) {
     res = res * arr[i];
  }
  return res;
}
```

Exempel – multiplicera talen i en array

```
public static int multiply(int[] arr) {
  int res = 1;                          1
  int i=0;                              1
  for (; i < arr.length; i++) {         1+1 (*n)
     res = res * arr[i];                1+1+1 (*n)
  }
  return res;                           1
}
```

$T(n) = 5 * n + 3$

Ordo

- O – Ordo (big-O) är en övre gräns (upper bound) för hur tiden för en algoritm växer
- Definition:

  $T(n)$ är en icke-negativ funktion
  $T(n) \in O(f(n))$ (dvs. $T(n)$ tillhör mängden $O(f(n))$)
  om det finns positiva konstanter c och n_0 sådana att
  $T(n) \leq c * f(n)$ för $n \geq n_0$

Exempel – multiplicera talen i en array

```
public static int multiply(int[] arr) {
  int res = 1;                          1
  int i=0;                              1
  for (; i < arr.length; i++) {         1+1 (*n)
     res = res * arr[i];                1+1+1 (*n)
  }
  return res;                           1
}
```

$T(n) = 5 * n + 3$

Exempel – multiplicera talen i en array

- $T(n) = 5 * n + 3$
- Antagande: $f(n) = n$
  - $5 * n + 3 \leq cn$
  - $(c - 5) * n \geq 3$
  - $n \geq \frac{3}{c-5}$
  - Låt $c = 6$, $n_0 = 3$
    $\Rightarrow 5 * n + 3 \leq 6 * n$ för $n \geq 3$
  - $\Rightarrow T(n) \in O(n)$

Tidskomplexitet – fall

- Tidskomplexiteten gäller för ett visst fall
  - Värsta
  - Bästa
  - Medel
- För multiplikation av värden i en array spelar det ingen roll, men i andra fall kan det göra det.
- Viktigt att vara tydlig med vad man menar!

Tidskomplexitet linjär sökning

- Värsta fallet: $O(n)$
- Bästa fallet: $O(1)$ – 1 används om det tar konstant tid

Tidskomplexitet linjär sökning

- Värsta fallet: $O(n)$
- Bästa fallet: $O(1)$ – 1 används om det tar konstant tid
- Medelfallet: $O(n)$
  - Antag att medelfallet tar tiden $T(n/2)$
  - $n/2 = n * 1/2$

    $1/2$ är en konstant vi kan bortse ifrån

Tidskomplexitet binär sökning

- Värsta fallet: $O(\log n)$
- Bästa fallet: $O(1)$
- Medelfallet: $O(\log n)$

Tidskomplexitet – beräkning

- När man anger komplexitet bortser man från konstanter och termer av lägre ordning
- Exempel:
  - $T(5 + n) \in O(n)$
  - $T(5 + 10n) \in O(n)$
  - $T(500000 + 100000n) \in O(n)$

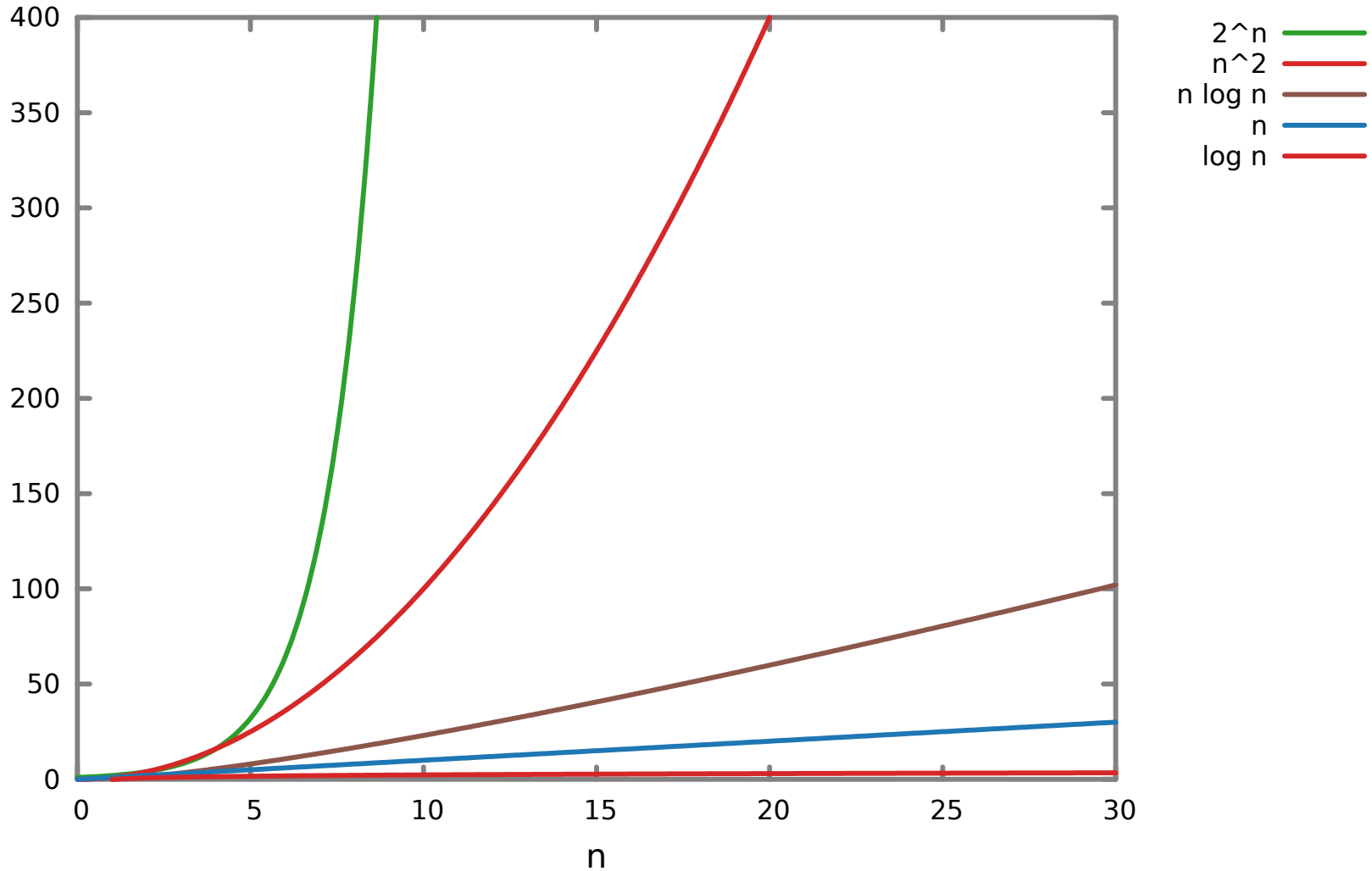
Tidskomplexitet – beräkning

- När man anger komplexitet bortser man från konstanter och termer av lägre ordning
- Exempel:
  - $T(5 + n) \in O(n)$
  - $T(5 + 10n) \in O(n)$
  - $T(500000 + 100000n) \in O(n)$
  - $T(n^2 + n) \in O(n^2)$
  - $T(15n^2 + 5/8 * n) \in O(n^2)$
  - $T(n + \log\ n) \in O(n)$
  - $T(25n\ \log\ n + 800) \in O(n\ \log\ n)$

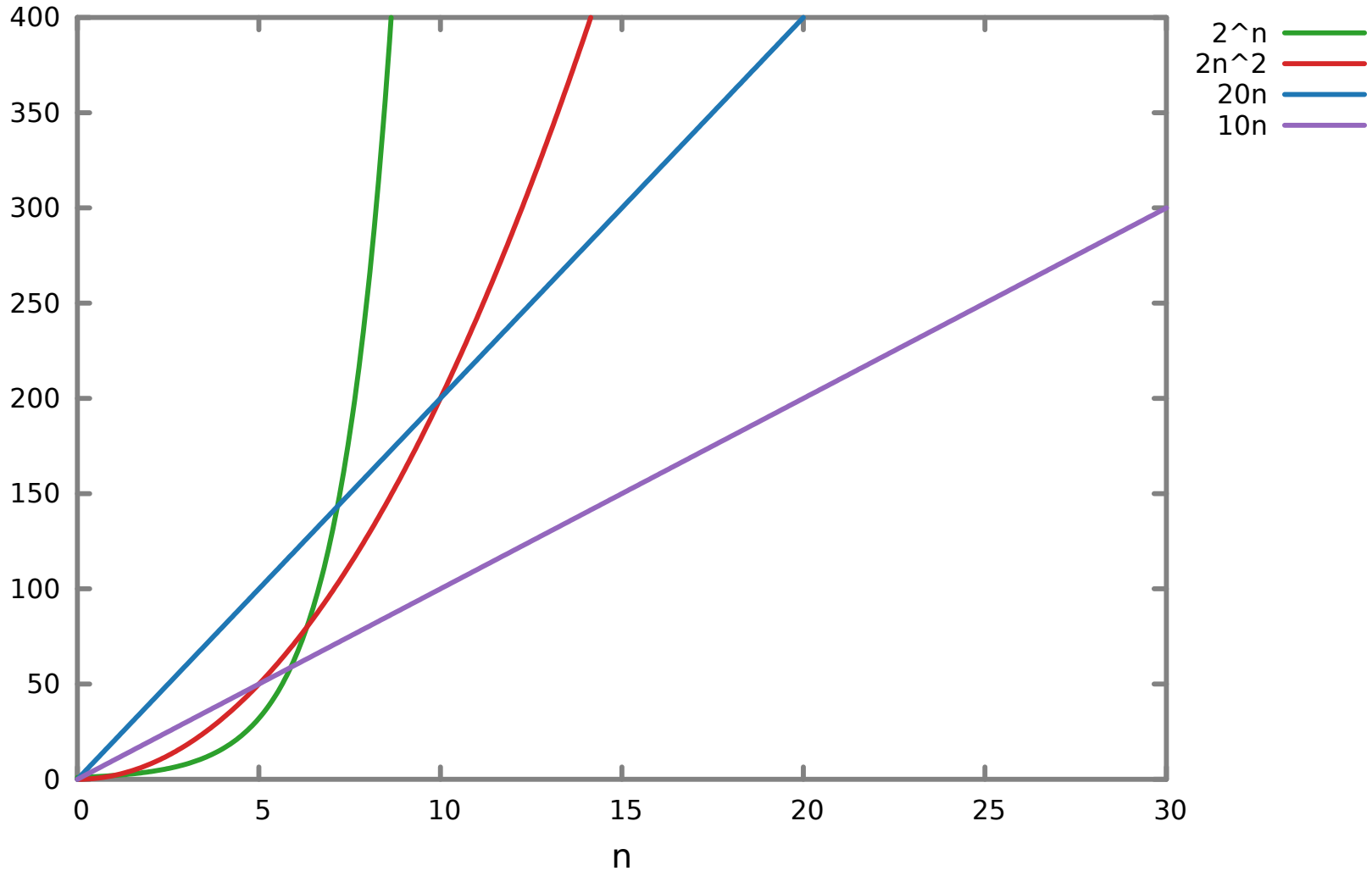
Ordoklasser

Ordo	Benämning
$O(1)$	Konstant
$O(\log\ n)$	Logaritmisk
$O(n)$	Linjär
$O(n\ \log\ n)$	
$O(n^2)$	Kvadratisk
$O(n^3)$	Kubisk
$O(n^x)$	Polynomisk (för $x > 1$)
$O(2^n)$	Exponentiell
$O(n!)$	Faktoriell

Asymptotisk storlek spelar roll!



Asymptotisk storlek spelar roll trots konstanter!



Omega och Theta

- O – övre gräns (upper bound)
- Även andra varianter finns:
  - Ω – undre gräns (lower bound)
  - Θ –
    $T(n) \in \Theta(n)$ iff $T \in O(n)$ och $T \in \Omega(n)$

Asymptotisk analys – diskussion

- Asymptotisk analys är ett bra verktyg för att diskutera algoritmer
- För stor indata är alltid en algoritm med högre komplexitet långsammare
- Men vad ”stor” innebär kan variera
- För små indata kan en algoritm vara bättre som har högre komplexitet
  - Exempel: För riktigt små arrayer kan linjär sökning vara snabbare än binär sökning eftersom vi inte behöver beräkna medelvärden

Asymptotisk analys – diskussion

- Asymptotisk analys är ett bra verktyg för att diskutera algoritmer
- För stor indata är alltid en algoritm med högre komplexitet långsammare
- Men vad ”stor” innebär kan variera
- För små indata kan en algoritm vara bättre som har högre komplexitet
  - Exempel: För riktigt små arrayer kan linjär sökning vara snabbare än binär sökning eftersom vi inte behöver beräkna medelvärden
- Man kan diskutera minneskomplexitet på samma sätt som tidskomplexitet

Algoritmanalys – vad ska ni kunna?

- Föra grundläggande resonemang kring tidskomplexiteten för ett program/algoritm för en övre gräns i olika fall (Ordo)
- Speciellt med avseende på:
  - De sök- och sorteringsalgoritmer som tas upp under kursen
  - Vanliga operationer för de datastrukturer som tas upp under kursen
- Utifrån enklare kod komma resonera er fram till tidskomplexiteten (som exemplet med multiplikation av värdena i en array)

Vad är sortering

- Placera objekten i en samling i en viss ordning
- Vi fokuserar på sortering i en array
- De viktigaste operationerna som kan användas då är
  - compare, eller $==$, $<$, $>$ – jämför två värden
  - swap – byt plats på två värden

Sorteringsalgoritmer

- Det finns en mängd olika sorteringsalgoritmer
- Enkla algoritmer, mindre effektiva
  - Selection sort
  - Bubble sort
  - Insertion sort
- Mer komplexa och effektiva algoritmer
  - Merge sort
  - Quick sort
  - Shell sort

Selection sort

- Urvalssortering
- DEMO!!

Selection sort

- Urvalssortering
- DEMO!!
- Intuition:
  1. Dela upp arrayen i en sorterad del (tom från början) och en osorterad del (hela arrayen från början)
  2. leta upp det minsta värdet i den osorterade delen
  3. lägg det sist i den sorterade delen
  4. Upprepa steg 2+3 tills hela arrayen är sorterad

Selection sort – kod

```
public void sort(int[] lista)
   for(int i = 0; i < lista.length - 1 ; ++i) {
      int minIndex = i;
      for(int j = i+1; j < lista.length; ++j) {
         if (lista[j] < lista[minIndex]) {
            minIndex = j;
         }
      }
      swap(lista, i, minIndex);
   }
}

public void swap(int[] lista, int i, int j) {
   int temp = lista[i];
   lista[i] = lista[j];
   lista[j] = temp;
}
```

Selection sort – komplexitet

```
public void sort(int[] l)
   for(int i = 0; i < l.length - 1 ; ++i) {  // n-1
      int minIndex = i;
      for(int j = i+1; j < l.length; ++j) { // n-1+n-2+...+2+1
         if (lista[j] < l[minIndex]) {      // konstant
            minIndex = j;                   // konstant
         }
      }
      swap(l, i, minIndex);                 // konstant
   }
}
```

Selection sort – komplexitet

```
public void sort(int[] l)
   for(int i = 0; i < l.length - 1 ; ++i) {  // n-1
      int minIndex = i;
      for(int j = i+1; j < l.length; ++j) { // n-1+n-2+...+2+1
         if (lista[j] < l[minIndex]) {      // konstant
            minIndex = j;                   // konstant
         }
      }
      swap(l, i, minIndex);                 // konstant
   }
}
```

- $n + n - 1 + n - 2 + ... + 2 + 1 = \frac{(n^2+n)}{2} \in O(n^2)$
- så kallade triangeltal

Bubble sort och insertion sort

DEMO!!

Bubble sort

- Bubbelsortering
- Intuition:
  1. Dela upp arrayen i en sorterad del (tom från början) och en osorterad del (hela arrayen från början)
  2. För varje par av värden från höger till vänster i den osorterade delen
     - Byt plats om det högra värdet $<$ det vänstra värdet
  3. Detta innebär att det minsta värdet "bubblas upp" till den första platsen i den osorterade delen, och kan flyttas över till en sorterade delen
  4. Upprepa tills hela arrayen är sorterad

Insertion sort

- Insättningssortering
- Intuition:
  1. Dela upp arrayen i en sorterad del (tom från början) och en osorterad del (hela arrayen från början)
  2. Flytta det första värdet i den osorterade delen till rätt plats i den sorterade delen
  3. Upprepa tills hela arrayen är sorterad

Komplexitet bubble sort och insertion sort

- $O(n^2)$
- Använd samma resonemang som för selection sort

Sorteringsalgoritmer – egenskaper

- Tidskomplexitet – hur snabbt går den?
- Minneskomplexitet – hur mycket minne tar den?
- Stabilitet – ligger lika värden i samma ordning som innan sorteringen

Sorteringsalgoritmer – egenskaper

- Tidskomplexitet – hur snabbt går den?
- Minneskomplexitet – hur mycket minne tar den?
- Stabilitet – ligger lika värden i samma ordning som innan sorteringen
- För de algoritmer vi gått igenom hittills:
  - Tid: $O(n^2)$
  - Minne: $O(n)$ för själva arrayen, $O(1)$ extra minne
  - Stabila

Sorteringsalgoritmer – egenskaper

- Tidskomplexitet – hur snabbt går den?
- Minneskomplexitet – hur mycket minne tar den?
- Stabilitet – ligger lika värden i samma ordning som innan sorteringen
- För de algoritmer vi gått igenom hittills:
  - Tid: $O(n^2)$
  - Minne: $O(n)$ för själva arrayen, $O(1)$ extra minne
  - Stabila
- Det finns mer avancerade algoritmer med tidskomplexitet $O(n \log n)$
- Det är gränsen för hur snabba sorteringsalgoritmer som baseras på jämförelser kan vara
- Dessa algoritmer är dock inte alltid stabila, och de kan ta extra minne

Avancerade sorteringsalgoritmer

- Merge sort
  - Värsta fallet: $O(n \log n)$
  - Stabil
  - Behöver extra minne (lite olika beroende på hur den implementeras)

Avancerade sorteringsalgoritmer

- Merge sort
  - Värsta fallet: $O(n \log n)$
  - Stabil
  - Behöver extra minne (lite olika beroende på hur den implementeras)
- Quick sort
  - Medelfallet: $O(n \log n)$
  - Värsta fallet: $O(n)$
  - Ej stabil
  - Behöver lite extra minne, beror på implementationen

Avancerade sorteringsalgoritmer

- Merge sort
  - Värsta fallet: $O(n \log n)$
  - Stabil
  - Behöver extra minne (lite olika beroende på hur den implementeras)
- Quick sort
  - Medelfallet: $O(n \log n)$
  - Värsta fallet: $O(n)$
  - Ej stabil
  - Behöver lite extra minne, beror på implementationen
- Shell sort
  - Komplexitet svåranalyserad
  - Beroende på implementation värsta fallet runt $O(n^{3/2})$
  - Ej stabil
  - Inget extra minne

Ordnade klasser

- Java har ett interface `Comparable`
- Innehåller metoden `int compareTo(T t)` som returnerar:
  - 0 if the object is equal to t
  - < 0 if the object is smaller than t
  - > 0 if the object is larger than t
- Detta interface krävs när Java behöver kunna jämföra instanser av objekt, till exempel vid sortering

Lab 2, del 1

- Sortering av heltal i ett grafiskt labskal
  - Implementera två sorteringslagoritmer själva (insertion och shell sort)
  - Jämför fem sorteringsalgoritmer under olika förutsättningar – koppla till teorin

Lab 2, del 1

- Sortering av heltal i ett grafiskt labskal
  - Implementera två sorteringslagoritmer själva (insertion och shell sort)
  - Jämför fem sorteringsalgoritmer under olika förutsättningar – koppla till teorin
- Labskalet begränsat för att kunna synkronisera algoritmerna
- Tillhandahåller följande metoder:
  - cmp(i,j) – jämför värdena på plats i och j i arrayen (jfr compareTo)
  - swap(i,j) – byter plats på värdena på plats j och i
  - elementCount() – hur många element finns det i arrayen (jfr length/size())

Mer om sorteringslagoritmer

- Vi har bara gått igenom kod för en sorteringsalgoritm idag
- Ni ska kunna beskriva alla de algoritmer som vi gått igenom
  - Kunna visa hur de fungerar genom ett exempel för en array
  - Kunna skriva (pseudo)kod för dem
- I labben får ni skriva egen kod för två algoritmer
- Labben innehåller även given kod för andra algoritmer – studera den
- Läs i kursböckerna om sorteringsalgoritmer
- Även (engelska) Wikipedia har bra material om sortering

I veckan

- Tisdag
  - Deadline för lab 1!
- Onsdag
  - Föreläsning
    - Generiska algoritmer och klasser (behövs för lab 2, del 2)
    - Länkade listor mm (behövs för lab 3)
    - Mer om komplexitet
  - Lab
- Jobba själv innan onsdag
  - Sätt igång med del 1 i lab 2
  - Läs på om sökning, sortering, komplexitet
    - Komplexitet: grundläggande Eck 8.5, mer avancerat Shaffer 3.1–3.7