Föreläsning 1
Datastrukturer (DAT037)

Fredrik Lindblad[1]

30 oktober 2017

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Innehåll

- Introduktion
- Dynamiska arrayer
- Asymptotisk komplexitet
- Programanalys del 1
- Ordo-notation del 1

Datastrukturer

- Organisering av data
- Inte bara korrekt, utan effektivt (framförallt hastighet men även minneskrav)
- Abstrakt datatyp (ADT) – vissa vanliga tillämpningar av datasamlingar (lista, stack, kö, prioritetskö, mängd, avbildning, graf)
- Vanliga datastrukturer som implementerar dessa ADTer – dynamisk array, länkad lista, heap, sökträd, m.m.
- Plus några klassiska algoritmer – sortering, grafalgoritmer

Exempel

```
public class Bits {

    public static void main(String[] args) {

        int n = Integer.parseInt(args[0]);

        String bits = new String();

        for (int i = 0; i < n; i++) {
            bits += i & 1;
        }

        System.out.print(bits);

    }
}
```

Hur lång tid tar det att köra ”java Bits 100000”, jämfört med ”java Bits 50000”?

- Lika snabbt, 2 eller 4 ggr långsammare?

Hur lång tid tar det att köra ”java Bits 100000”, jämfört med ”java Bits 50000”?

- Lika snabbt, 2 eller 4 ggr långsammare?

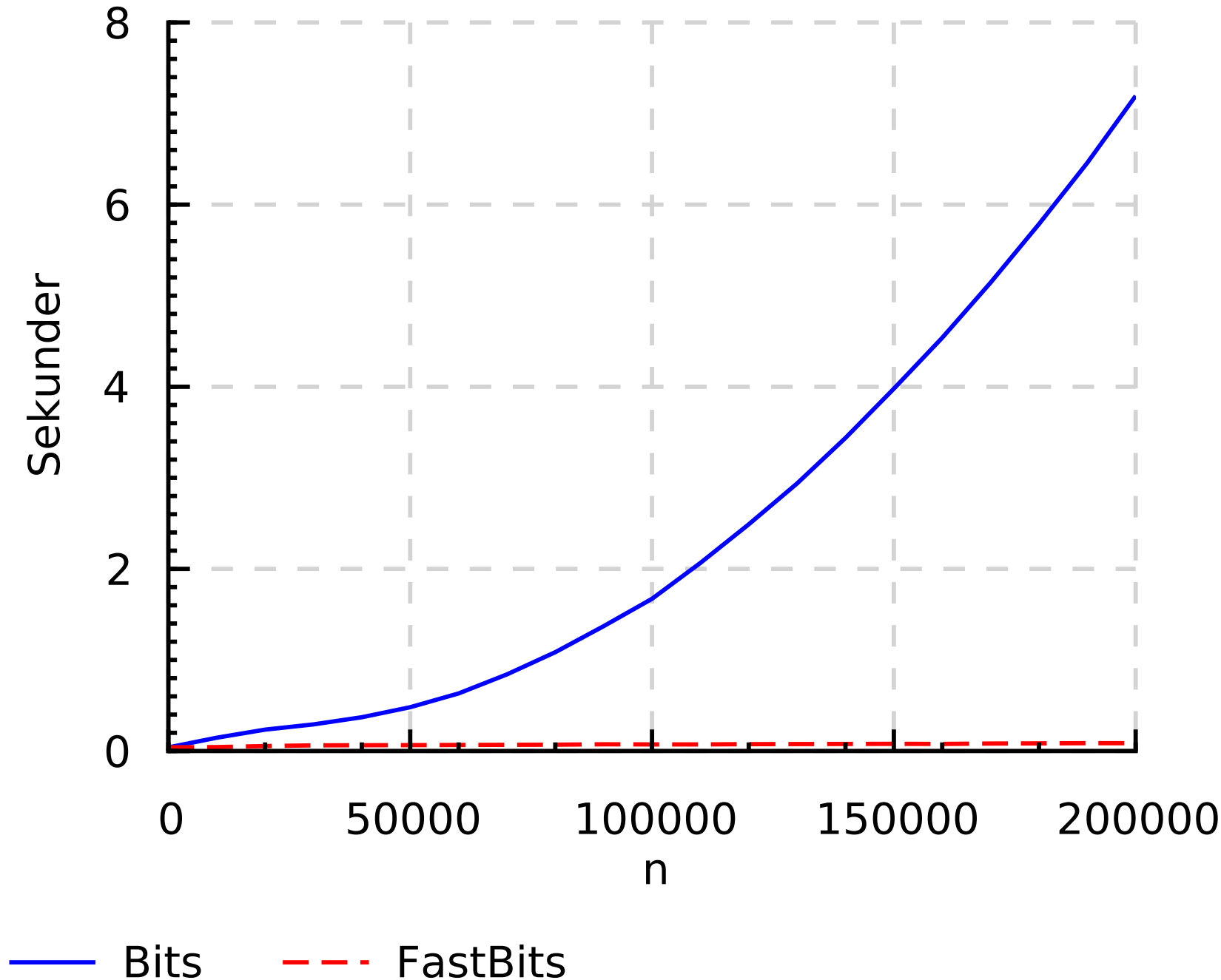
Svar : Ungefär 4 ggr långsammare?

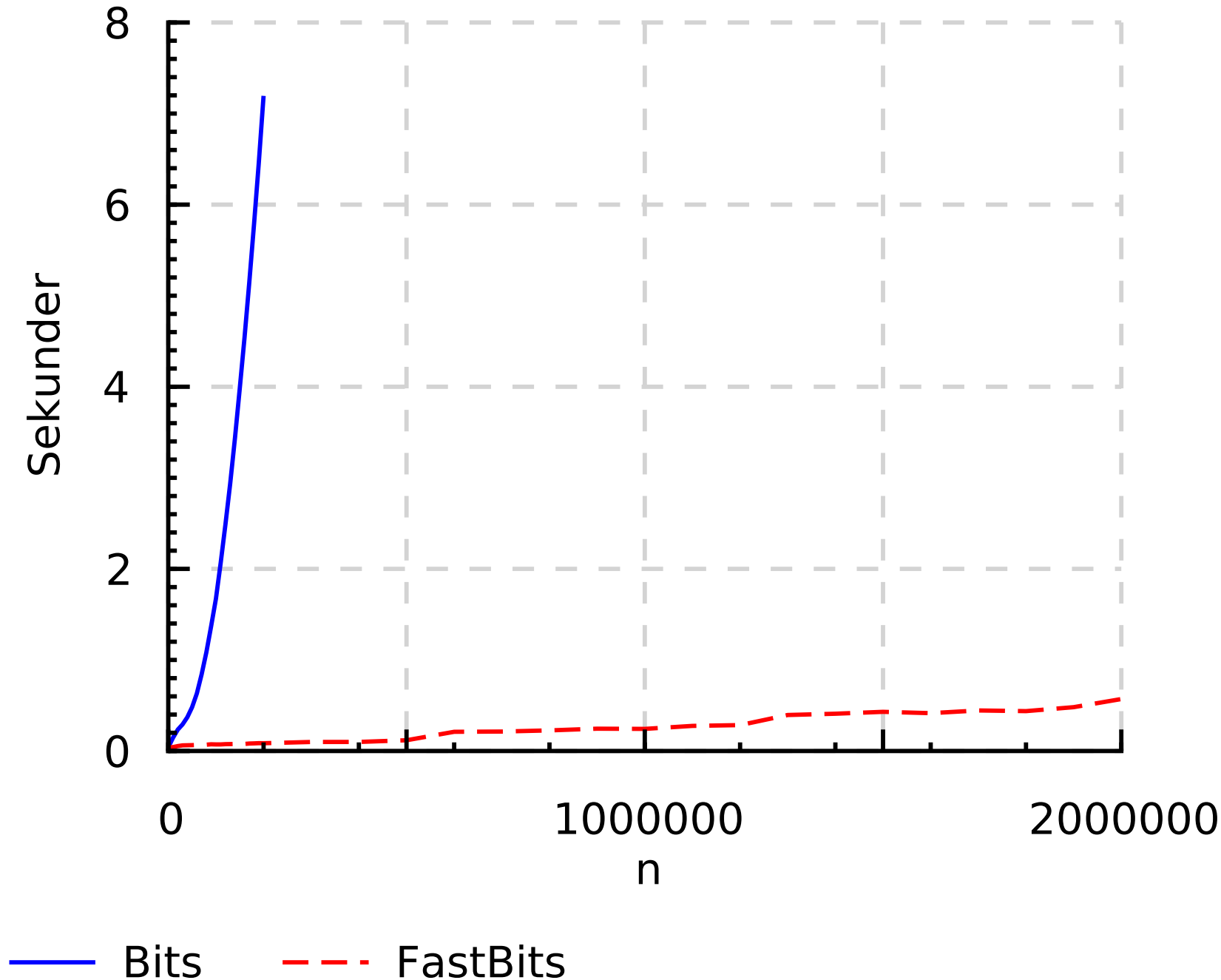
Dåligt val av datastruktur

```
public class Bits {

    public static void main(String[] args) {

        int n = Integer.parseInt(args[0]);

        String bits = new String();

        for (int i = 0; i < n; i++) {
            bits += i & 1;
        }

        System.out.print(bits);

    }
}
```

Bättre val av datastruktur

```
public class FastBits {

    public static void main(String[] args) {

        int n = Integer.parseInt(args[0]);

        StringBuffer bits = new StringBuffer();

        for (int i = 0; i < n; i++) {
            bits.append(i & 1);
        }

        System.out.print(bits);

    }
}
```


8
6
4
2
0
Sekunder
0
50000
100000
150000
200000
n
Bits
FastBits


8
6
4
2
0
Sekunder
0
1000000
2000000
n
Bits
FastBits

Tillbaka till exemplet

+= motsvarar för strängar append

```
private char[] string;

public void append(char c) {
  char[] temp = new char[string.length + 1];

  for (int i = 0; i < string.length; i++) {
      temp[i] = string[i];
  }
  temp[string.length] = c;

  string = temp;
}
```

Antag att varje primitiv operation tar en viss konstant tid.

Tillbaka till exemplet

Analysera koden och räkna hur många sådana tidsenheter som går åt.

```
private char[] string;

public void append(char c) {
  char[] temp = new char[string.length + 1];  // ~ n

  for (int i = 0; i < string.length; i++) {   // ~ 3n
      temp[i] = string[i];
  }
  temp[string.length] = c;                    // 1

  string = temp;                              // 1
}
```

Totalt (för $n = \texttt{string.length}$): $\sim 4n + 2$.

Tillbaka till exemplet

```
private char[] string;

public void append(char c) {
  char[] temp = new char[string.length + 1];  // ~ n

  for (int i = 0; i < string.length; i++) {   // ~ 3n
      temp[i] = string[i];
  }
  temp[string.length] = c;                    // 1

  string = temp;                              // 1
}
```

Vi kommer använda ett mått som låter oss säga totalt (för $n = \texttt{string.length}$): $\sim n$.

Detta mått bortser från konstanta termer och koefficienter, vilket gör att vi abstraherar bort vad som sker för små indata/låga tal (det är de stora körningarna som tar tid och är intressanta) och den faktiska exekveringstiden (vilken beror på hårdvara, kompilator, etc.).

Tillbaka till exemplet

```
String bits = new String();          // 1

for (int i = 0; i < n; i++) {        // ???
    bits += i & 1;                   // ~ i
}

System.out.print(bits);              // ~ n
```

???

$$\sum_{i=0}^{n-1} i = 0 + 1 + 2 + ... + (n-1)$$

$$= n\frac{n-1}{2}$$

$$\sim n^2$$

Tillbaka till exemplet

```
String bits = new String();          // 1

for (int i = 0; i < n; i++) {        // ~ n^2
    bits += i & 1;                   // ~ i
}

System.out.print(bits);              // ~ n
```

Totalt: $\sim n^2$.

Tillbaka till exemplet

```
String bits = new String();          // 1

for (int i = 0; i < n; i++) {        // ~ n^2
    bits += i & 1;                   // ~ i
}

System.out.print(bits);              // ~ n
```

Totalt: $\sim n^2$.

Anta att vi kan utföra $\sim 10^6$ “instruktioner”/s.
$n = 10^5 \Rightarrow \sim 3$ timmar
$n = 10^6 \Rightarrow \sim 10$ dagar
$n = 10^7 \Rightarrow \sim 3$ år

Analys

- Verkar onödigt att kopiera arrayen varje gång.
- När vi kopierar kan vi lägga till några extra “tomma” element i slutet av arrayen, så att vi inte behöver kopiera lika ofta.
- Låt oss göra arrayen dubbelt så stor.

Dubbelt så stor

```
private char[] string;
private int    length;

public void append(char c) {
  if (length == string.length) {
    char[] temp = new char[2 * string.length];

    for (int i = 0; i < string.length; i++) {
        temp[i] = string[i];
    }

    string = temp;
  }

  string[length++] = c;
}
```

Dubbelt så stor

```
StringBuffer bits = new StringBuffer();  // 1

for (int i = 0; i < n; i++) {            // n + ???
    bits.append(i & 1);
}

System.out.print(bits);                  // ~ n
```

???

Anta att n är en jämn tvåpotens, $n = 2^k$ $(k \in \mathbb{N})$:

$$\begin{aligned}\sum_{i=0}^{k} 2^i &= 1 + 2 + 4 + ... + 2^k \\ &= 2^{1+k} - 1 \\ &= 2n - 1 \\ &\sim n\end{aligned}$$

Dubbelt så stor

```
StringBuffer bits = new StringBuffer();  // 1

for (int i = 0; i < n; i++) {            // ~ n
    bits.append(i & 1);
}

System.out.print(bits);                  // ~ n
```

Totalt: $\sim n$.

Anta att vi kan utföra 10^6 “instruktioner”/s.
$n = 10^6 \Rightarrow \sim 1$ sekund
$n = 10^9 \Rightarrow \sim 20$ minuter
$n = 10^{12} \Rightarrow \sim 10$ dagar

Datastrukturer

- `String`: `append` kopierar varje gång.
  Kan ej ändras: “immutable”.
- `StringBuffer`: `append` kopierar sällan.
  Dynamisk array.

Blir programmet effektivt om vi lägger till 10 element istället för att göra arrayen dubbelt så stor?

Blir programmet effektivt om vi lägger till 10 element istället för att göra arrayen dubbelt så stor?

Svar: Knappt 10 gånger snabbare, men fortfarande är tidsåtgången $\sim n^2$.

Dynamiska arrayer med `remove`

Har `add` och `remove` amorterade tidskomplexiteten $O(1)$ om man halverar arrayen när den blir...

- ...halvfull?
- ...kvartsfull?

Varför?

Dynamiska arrayer med `remove`

Har `add` och `remove` amorterade tidskomplexiteten $O(1)$ om man halverar arrayen när den blir...

- ...halvfull?
- ...kvartsfull?

Varför?

Svar: Inte för halvfull, men för kvartsfull. Om storleken halveras vid halvfull och dubbleras vid full så tar varje operation $\sim n$ om man gör varannan `append` och varannan `remove` och storleken är sådan att arrayen krymper och växer varje gång.

Asymptotisk komplexitet

Kursen fokuserar på *asymptotisk* komplexitet:
vad händer när storleken går mot oändligheten?

$$\begin{aligned} 8n + 3 \quad &\Rightarrow \quad O(n) \\ 4n^2 + 3n + 6 \quad &\Rightarrow \quad O(n^2) \end{aligned}$$

Ordo-notation (en variant)

$T(n) = O(f(n))$

om och endast om

det finns ett naturligt tal n_0

och ett reellt tal $c > 0$

så att $T(n) \leq cf(n)$ för alla $n \geq n_0$.

Ω

$T(n) = \Omega(f(n))$

om och endast om

det finns ett naturligt tal n_0

och ett reellt tal $c > 0$

så att $T(n) \geq cf(n)$ för alla $n \geq n_0$.

$T(n) = \Theta(f(n))$

om och endast om

$T(n) = O(f(n))$ och $T(n) = \Omega(f(n))$

Exempel

Kan använda Θ för att uttrycka resultatet av tidigare analyser:

- `Bits`: $\Theta(n^2)$, kvadratisk asymptotisk tidskomplexitet.
- `FastBits`: $\Theta(n)$, linjär asymptotisk tidskomplexitet.