Föreläsning 2
Datastrukturer (DAT037)

Fredrik Lindblad[1]

1 november 2017

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

Innehåll

- Ordo-notation del 2
- Kostnadsmodeller
- Programanalys del 2
- Amorterad tidskomplexitet
- Generics

Vilka påståenden är korrekta?

- A. $n^2 = O(n)$
- B. $n^2 = \Omega(n)$
- C. $n = O(n^2)$
- D. $n = \Omega(n^2)$
- E. $10n + 7 = \Theta(3n + 1)$
- F. $10n + 7 = \Theta(13n + 12)$

Vilka påståenden är korrekta?

- A. $n^2 = O(n)$
- B. $n^2 = \Omega(n)$
- C. $n = O(n^2)$
- D. $n = \Omega(n^2)$
- E. $10n + 7 = \Theta(3n + 1)$
- F. $10n + 7 = \Theta(13n + 12)$

Svar: B, C, E och F

Ordonotation: regler

Notera:

$$1 = O(n)$$
$$1 = O(n^2)$$
$$n = O(n \log n)$$
$$n = O(n^2)$$

Ordonotation: regler

Om $T(n)$ är ett polynom av grad k:

$$T(n) = O(n^k).$$

- $7n^2 + 3n + 2 = O(n^2)$.
- $0.1n^3 + 1000 = O(n^3)$.

Ordonotation: regler

Om $k > 0$ är en *konstant*:

$$(\log_2 n)^k = O(n),$$
$$\log_2(n^k) = k \log_2 n = O(\log n).$$

För konstant $a > 1$:

$$\log_a n = \log_2 n / \log_2 a = O(\log_2 n) = O(\log n).$$

- $(\log_2 n)^{10000} = O(n)$.
- $\log_2(n^{10000}) = O(\log n)$.

Ordonotation: regler

Om $T(n) = O(f(n))$ och $U(n) = O(g(n))$:

$$\begin{aligned} T(n) + U(n) &= O(f(n) + g(n)) \\ &= O(\max(f(n), g(n))), \\ T(n)U(n) &= O(f(n)g(n)). \end{aligned}$$

- $7n^2 + 3n^2 = O(n^2 + n^2) = O(n^2)$.
- $2n^3 + (\log_2 n)^{1000} = O(n^3 + n) = O(n^3)$.
- $2n^3 \log_7 5n^2 = O(n^3 \log n)$.

Komplexitetsklasser

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset$$
$$\subset O(n^2) \subset O(n^k) \subset O(l^n)$$

där $k > 2$ och $l > 1$.

Tidskomplexitet

Hur analyserar man tidskomplexitet?

- *Mäta*.
  Nackdelar: Kan vara tidskrävande,
  kanske inget bra stöd för design.
- *Räkna instruktioner*.
  Nackdelar: Komplicerat.
- *Förenklad modell*.
  Nackdelar: Inte tillämpligt i alla lägen.

Uniform kostnadsmodell

- Enkel dator.
- Varje instruktion tar en tidsenhet.
  Bara enkla instruktioner,
  men godtyckligt stora tal.
- Oändligt minne.

Uniform kostnadsmodell

- Enkel dator.
- Varje instruktion tar en tidsenhet.
  Bara enkla instruktioner,
  men godtyckligt stora tal.
- Oändligt minne.

- Inte realistisk.
- Fungerar ganska bra om man är försiktig.
- Används ofta i kursen.

Logaritmisk kostnadsmodell

- Enkel dator.
- Tid beräknas i termer av indatas storlek, räknat i antal bitar.
- Oändligt minne.
- Lite mer realistisk, lite krångligare.

Vad är den asymptotiska tidskomplexiteten?

```
public static void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

Vad är den asymptotiska tidskomplexiteten?

```
public static void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

Varje rad är en primitiv operation, $O(1)$.
$O(1 + 1 + 1) = O(1)$

Vad är den asymptotiska tidskomplexiteten?

```
public static void scalarProd(double[] v, double f) {
    for (int i = 0; i < v.length; i++) {
        v[i] *= f;
    }
}
```

Vad är den asymptotiska tidskomplexiteten?

```
public static void scalarProd(double[] v, double f) {
    for (int i = 0; i < v.length; i++) {
        v[i] *= f;
    }
}
```

Låt $n =$ `v.length`. Initiering av loop: O(1), villkorskoll och inkremenering: O(1), loopens kropp: O(1)

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

Vad är tidskomplexiteten?

```
public static int countDuplPair(int[] a) {
    int n = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] == a[j]) n++;
        }
    }
    return n;
}
```

Vad är tidskomplexiteten?

```
public static int countDuplPair(int[] a) {
    int n = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (a[i] == a[j]) n++;
        }
    }
    return n;
}
```

Låt $n =$ `a.length`. Alla atomiska satser: O(1)

$$O(1 + \sum_{i=0}^{n-1}(1 + \sum_{j=i+1}^{n-1}(1+1))) =$$

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

$$O(n(n-1)-n(n-1)/2) = O(n(n-1)/2) = O(n^2)$$

Vad är tidskomplexiteten?

```
public static double f(double[] a) {
    int i = a.length - 1;
    if (i == -1) return 0.0;
    double x = 0;
    for (;;) {
        x += a[i];
        if (i == 0) break;
        i /= 2;
    }
    return x;
}
```

Vad är tidskomplexiteten?

Alla enkla satser i koden på föregående slide tar konstant tid ($O(1)$). Det väsentliga är hur många gånger slingan upprepas. Låt $n =$ `a.length`

n	antel iterationer
1	1
2	2
$3 \ldots 4$	3
$5 \ldots 8$	4
$(2^k + 1) \ldots 2^{k+1}$	$k + 2$

Vad är tidskomplexiteten?

För $n \in [(2^k + 1) \ldots 2^{k+1}]$ är antalet iterationer $k + 2$.
Alltså är antalet iterationer
$\leq \log_2(n - 1) + 2 = O(\log n)$.
Komplexiteten för metoden är $O(\log n)$.

Värsta-, bästafallskomplexitet

Vad är tidskomplexiteten?

```
boolean hasDuplicate(int[] a) {
 for (int i = 0; i < a.length; i++) {
  for (int j = i + 1; j < a.length; j++) {
   if (a[i] == a[j]) return true;
  }
 }
 return false;
}
```

Exekveringstiden beror inte alltid enbart på indatans storlek, utan kan bero av dess beskaffenhet i övrigt. Därför talar man om värsta- och bästa fallskomplexitet. För de två exemplen räknade vi med att det inte fanns dubletter så att slingorna skulle exekveras fullt ut. Det var alltså värstafallskomplexiteten. I bästa fall hittas en dublett med en gång, d.v.s. bästa fallskomplexiteten är $O(1)$.
Vanligast är att prata om värstafallskomplexiteten eftersom man använder $O()$ och är intresserad av hur lång tid det tar som längst.

Värsta-, bästafallskomplexitet

```
boolean hasDuplicate(int[] a) {
 for (int i = 0; i < a.length; i++) {
  for (int j = i + 1; j < a.length; j++) {
   if (a[i] == a[j]) return true;
  }
 }
 return false;
}
```

(Värstafalls)komplexiteten är $O(n^2)$.
Bästafallskomplexiteten är $O(1)$.

Amorterad tidskomplexitet

- Lägga till ett element till en dynamisk array: $O(\ell)$, där ℓ är antalet element i arrayen.
- I avsaknad av annan information:
  Lägga till n element till en tom dynamisk array: $O(n^2)$.
- Vår analys från förra föreläsningen: $\Theta(n)$.
- $O(\ell)$ ger inte tillräckligt mycket information.

Amorterad tidskomplexitet

- Lägga till ett element till en dynamisk array: $O(\ell)$, där ℓ är antalet element i arrayen.
- I avsaknad av annan information:
  Lägga till n element till en tom dynamisk array: $O(n^2)$.
- Vår analys från förra föreläsningen: $\Theta(n)$.
- $O(\ell)$ ger inte tillräckligt mycket information.
- Vi kommer att använda

  amorterad tidskomplexitet

  för att hantera den här situationen på ett smidigt sätt.

Bokföringsmetoden

- Man kan låta tidiga, billiga operationer "betala" för dyra, sena.
- Efter dubblering:
  n snabba insättningar,
  1 dubblering.
- Insättning: Lägg ett mynt på cellen, och ett på en "gammal" cell.
- Kopiering: Har ett mynt på varje cell, kopieringen betald.

Bokföringsmetoden

Amorterad tidskomplexitet =

faktisk tidskomplexitet

$+$ värdet av nya mynt

$-$ värdet av sparade mynt som används.

Bokföringsmetoden

- Amorterad tidskomplexitet för insättning utan kopiering:

$$O(1) + 2m = O(1).$$

  Här är m värdet av ett mynt (en konstant > 0).
- Amorterad tidskomplexitet för kopiering:

$$O(n) - nm.$$

- Om m är tillräckligt stor:

$$O(n) - nm = O(1).$$

Bokföringsmetoden

- Amorterad tidskomplexitet för att lägga till ett element till en dynamisk array: $O(1)$.
- I avsaknad av andra operationer.
- En operation kan ges olika amorterad tidskomplexitet i olika analyser, beroende på hur mycket vi betalar, och hur många sparade mynt vi använder.

Amorterad tidskomplexitet

- Vissa operationer långsamma:
  kan vara problematiskt i realtidssammanhang.
- (Värstafalls)komplexiteten för insättning i dynamisk array är $O(n)$ för varje enskilt anrop.
- Notera att om man utför n operationer med komplexiteten $O(f(n))$ amorterat, så är komplexiteten för denna sekvens $O(nf(n))$ (ej amorterat).

Generics

Att återanvända kod är en princip som genomsyrar programmerandet och design av programspråk. Vi har sub-rutiner och klass-arv etc. som stöder detta.
En sak som också främjar detta är generics/polymorfism.
När data inte behöver vara en av specifik typ kan vi i java säga att den kan vara vad som helst.

Här är ett exempel på en generisk metod:

```
static <E> void reverse(E[] arr) {
 E tmp;
 for (int i = 0; i < arr.length / 2; i++) {
  tmp = arr[i];
  arr[i] = arr[arr.length-1-i];
  arr[arr.length-1-i] = tmp;
 }
}
```

Förutom generiska metoder kan man även definiera generiska interface och klasser.
Generics kommer vi använda mycket eftersom datastrukturer handlar om att organisera många data-värden. Man vill förstås implementera datastrukturer för alla möjliga typer av data på en gång.
Ofta kan inte de generiska typerna vara helt godtyckliga utan måste ha någon egenskap som används av metoden/klassen.

Här är ett exempel där jämförelse måste vara definierad för den okända typen:

```
static <E extends Comparable<? super E>>
       E findMin(E[] arr) {
 if (arr.length == 0) return null;
 E min = arr[0];
 for (int i = 1; i < arr.length; i++) {
  if (arr[i].compareTo(min) < 0) min = arr[i];
 }
 return min;
}
```

Abstrakt datatyp/datastruktur

Abstrakt datatyp (matematisk abstraktion): lista.

Datastrukturer (implementation):

- Array.
- Enkellänkad lista.
- ...

ADT-er

- Kan använda en datastruktur för en viss ADT för att implementera en annan ADT.

Samlingar (Collections) i Java

- Java Collections Framework.
- ADT $\sim$ gränssnitt, datastruktur $\sim$ klass.
- `Collection` utökar `Iterable`, d.v.s. alla samlingar har metoden `iterator` som kan användas för att genomlöpa alla element.
- En `Collection` har också några andra metoder gemensamt, t.ex. `add`, `remove`, `contains`, `size`.

Iterator

En iterator genomlöper som regel själva underliggande datastrukturen, inte en kopia av den. Problem kan uppstå om man ändrar datastrukturen under tiden.
Gränssnittet har tre metoder:

- `boolean hasNext()`
- `E next()`
- `void remove()` (denna är frivilligt att implementera)

Mängd (Set)

Den abstrakta datatypen mängd ska implementeras i labb 1.

ADT	Operationer (exkl konstruerare)
Mängd	add, remove, contains

En mängd innehåller aldrig två instanser som är lika (enligt `equals`). En mängd är en samling.
Vi återkommer senare i kursen till mängder och de gängse implementationerna av dem.

Exempel, mängd och iterator

```
Set<String> s = new HashSet<>();
s.add("A");
s.add("B");
s.add("A");
s.remove("A");
System.out.println(s.contains("A"));  // false
System.out.println(s.contains("B"));  // true
s.add("C");
Iterator<String> iter = s.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}  // prints B and C (maybe not in that order)
for (String e : s) {
    System.out.println(e);
}  // this loop has same effect as iterator
   // creation and while loop.
```