Föreläsning 1
Datastrukturer (DAT037)

Fredrik Lindblad

2016-10-31

Tidsanalys/tidskomplexitet

`IntMultiSet` som representerar en multimängd av heltal, ex. $\{2, 7, 4, 5, 2, 9\}$. Hur lång tid tar det att exekvera `hasDuplicate`?

```
class IntMultiSet {
 int[] a;
 ...
 boolean hasDuplicate() {
  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;
 }
}
```

Vi antar först att multimängden inte innehåller dubletter.
Mät tiden empirisk genom att köra koden på en faktisk dator.

Exekveringstiden beror i allmänhet på datan som hanteras. Ju mer data desto mer tid. För stora mängder är exekveringstiden ett problem. Antag att heltalen hela tiden är sorterade i listan. Då kan `hasDuplicate` implementeras annorlunda.

```
class IntMultiSet {
 int[] a;  // invariant: sorted
 ...
 boolean hasDuplicate() {
  for (int i = 0; i < a.length - 1; i++) {
   if (a[i] == a[i+1]) return true;
  }
  return false;
 }
}
```

Exekveringstiden är summan av tiden det tar att utföra varje liten del av programmet.
Varje liten del kan antas ta en viss tid som inte beror på datan. Men vi vet inte hur mycket.
Dessa tider beror på kompilatorn, VM, processor.
För att kunna bedöma och jämföra algoritmers effektivitet utan hänsyn till dessa faktorer behöver abstrahera bort de faktiska tiderna.

Vi beräknar tiden att exekvera andra varianten av `hasDuplicate`. Fortfarande antar vi att dubletter saknas.

```
boolean hasDuplicate() {
 for (int i = 0; i < a.length - 1; i++) {
                                          // a, b, c
  if (a[i] == a[i+1]) return true;        // d
 }                                        // e
 return false;                            // f
}
```

$$T = a + b + \Sigma_{i=0}^{\texttt{a.length}-2}(d + e + c + b) + f$$

Låt $n = \texttt{a.length}$.

$$\begin{gathered} T = a + b + \Sigma_{i=0}^{n-2}(d + e + c + b) + f = \\ = a + b + f + (d + e + c + b)(n - 1) = \\ = (a + f - c - d - e) + (d + e + c + b)n \end{gathered}$$

Låt T vara en funktion av n.

$$T(n) = g + hn$$

för något g och h som vi inte känner till och som är beroende av processor etc.

Vi vill kunna uttrycka exekveringstiden på något sätt så vi kan undvika okända konstanter såsom g och h.
Lösning: använd asymptotisk tillväxttakt, ordo-notation.

Ordo-notation

$T(n) = O(f(n))$
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 $n \geq n_0$.

$T(n)$ är funktion, $\mathbb{N} \to \mathbb{R}$. $O(f(n))$ är egentligen en mängd av funktioner, så med $T(n) = O(f(n))$ menar man $T(n) \in O(f(n))$. Om $T_1(n) = O(f(n))$ och $T_2(n) = O(f(n))$ betyder inte det att $T_1(n) = T_2(n)$.

Denna sätt att mäta resursförbrukning (oftast tid, men även arbetsminne) kallas asymptotisk (tids-/minnes-)komplexitet.
Om $T(n) = O(f(n))$ så är tillväxttakten för $T(n)$ asymptotiskt begränsad av tillväxttakten för $f(n)$.

Låt oss applicera detta på exekveringstiden för den andra `hasDuplicate`.
Påstående: $T(n) = g + hn = O(n)$

Låt oss applicera detta på exekveringstiden för den andra `hasDuplicate`.
Påstående: $T(n) = g + hn = O(n)$

$$T(n) = g(1 - n) + (g + h)n$$

Då $n \geq 1$ så är $T(n) \leq (h + g)n$.
Villkoret för ordo är alltså uppfyllt med (t.ex.)
$c = g + h$ och $n_0 = 1$.

Asymptotisk komplexitet bryr sig inte om konstanta termer, $O(m + f(n)) = O(f(n))$.
Inte heller konstanta koefficienter,
$O(cf(n)) = O(f(n))$.
Detta är precis vad vi vill ha.
När man uttrycker komplexitet, gör det så enkelt som möjligt. T.ex. $O(n)$ istället för $O(5 + 3n)$.
Konstant komplexitet, d.v.s., den som ej beror på datans storlek, skriver man $O(1)$.

Vanliga tillväxttakter för tidskomplexitet

$$O(1) \subset O(\log(n)) \subset O(n) \subset O(n \cdot \log(n)) \subset$$
$$\subset O(n^2) \subset O(n^3) \subset O(n^p) \subset O(k^n)$$

($p > 3$, $k > 1$)
konstant, logaritmisk, linjär, n-log n, kvadratisk, kubisk, polynomiell, exponentiell

Bara den snabbast växande termen är relevant,
t. ex. $O(n^2 + n \cdot \log(n)) = O(n + n^2) = O(n^2)$.
Logaritmens bas är irrelevant,
$O(_c\log(n)) = O(_d\log(n))$.

$O()$ är en övre gräns. $T(n) = O(f(n))$ betyder att $T(n)$ inte växer snabbare än $f(n)$.
Motsatsen är $\Omega()$.

$T(n) = \Omega(f(n))$
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 $n \geq n_0$.

$\Theta()$ uttrycker att två funktioner växer lika snabbt.
$T(n) = \Theta(f(n))$ omm $T(n) = O(f(n))$ och
$T(n) = \Omega(f(n))$
$o()$ uttrycker att en funktion växer snabbare än en annan.
$T(n) = o(f(n))$ omm $T(n) = O(f(n))$ men inte
$T(n) = \Omega(f(n))$
Man brukar ofta använda $O()$ i betydelsen $\Theta()$, d.v.s. om man skriver $T(n) = O(f(n))$ så menar man $T(n) = \Theta(f(n))$. Annars kunde man skriva $T(n) = O(2^n)$ som svar på de flesta frågor om algoritmers komplexitet.

Tidskomplexitet för första versionen av `hasDuplicate`

```
boolean hasDuplicate() {
 for (int i = 0; i < a.length; i++) {          // a, b, c
  for (int j = i + 1; j < a.length; j++) {     // d, e, f
   if (a[i] == a[j]) return true;              // g
  }                                            // h
 }                                             // k
 return false;                                 // l
}
```

Tidskomplexitet för första versionen av `hasDuplicate`

```
boolean hasDuplicate() {
 for (int i = 0; i < a.length; i++) {          // a, b, c
  for (int j = i + 1; j < a.length; j++) {     // d, e, f
   if (a[i] == a[j]) return true;              // g
  }                                            // h
 }                                             // k
 return false;                                 // l
}
```

$$T(n) = O(a + b + \sum_{i=0}^{n-1}(d + e + \sum_{j=i+1}^{n-1}(g + h + f + e) + \\ + k + c + b) + l) = O(1 + \Sigma_{i=0}^{n-1}(1 + \Sigma_{j=i+1}^{n-1} 1)) = \\ = O(n + \sum_{i=0}^{n-1}(n - 1 - i)) = O(n + \sum_{i=0}^{n-1} i) = \\ = O(n + \tfrac{n(n-1)}{2}) = O(n^2)$$

Vi använde

$$\sum_{i=0}^{n-1} 1 = n \quad \text{och} \quad \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Den första är rätt självklar. Hur visar man att den andra gäller?

Vi använde

$$\sum_{i=0}^{n-1} 1 = n \quad \text{och} \quad \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

Den första är rätt självklar. Hur visar man att den andra gäller? Svar: Induktion
Basfall: $n = 0$, $\sum_{i=0}^{0-1} i = 0 = \frac{0(0-1)}{2}$
Induktionssteg: Antag att likheten gäller för $n-1$.
Ska visa att den gäller för n.

$$\sum_{i=0}^{n-1} i = n - 1 + \sum_{i=0}^{n-2} i = n - 1 + \frac{(n-1)(n-2)}{2} =$$
$$= \frac{2n-2+n^2-3n+2}{2} = \frac{n^2-n}{2} = \frac{n(n-1)}{2}$$

Slutsats: Likheten gäller för alla $n \geq 0$.