Föreläsning 6

Innehåll

- Algoritmer och effektivitet
  - Att bedöma och jämföra effektivitet för algoritmer
  - Begreppet tidskomplexitet

Val av algoritm och datastruktur

- Det räcker inte med att en algoritm är korrekt – den måste även producera resultat inom rimlig tid.
- Både val av algoritm och datastruktur påverkar tidsåtgången/effektiviteten.
- Vi måste kunna beräkna effektiviteten för att kunna jämföra olika val av algoritm och/eller datastruktur.

Tidsåtgång och problemets storlek

- Tiden det tar att lösa ett problem är oftast en strängt växande funktion av problemets storlek.
  - Det tar längre tid att summera 100 000 tal än att summera 100 tal.
- Vissa enkla problem kan dock lösas på konstant tid oavsett storleken.
  - Om vi t.ex. har en vektor med n tal kan vi alltid ta reda på det i:e talet i vektorn på konstant tid oavsett hur stort n är.
  - Exempel:
    `int nbr = a[i];`

Mäta exekveringstiden

- Vi kan mäta exekveringstiden:

```
long startTime = System.nanoTime();
...
long endTime = System.nanoTime();
long execTime = endTime - startTime;
```

- Men även annat än indatans storlek påverkar tidsåtgången:
  - uppstart av exekveringen
  - optimeringar
  - kompilator
  - dator
  - ...
- Dessutom måste algoritmen implementeras om den ska exekveras.

Tidskomplexitet

- Tidskomplexitet T(n) är ett mått på hur tidsåtgången växer med problemets storlek n.
  - Tidskomplexiteten uttrycks som en funktion av problemets storlek.
- Beräkna tidskomplexiteten genom att räkna hur många den mest frekventa basala operationen som utförs.
  - Med basal operation menas en operation som tar konstant tid på en dator, t.ex. en aritmetisk operation eller en jämförelseoperation.
- Exempel: Antag att vektorn a innehåller n tal som ska summeras.

```
int sum = 0;
for (int i = 0; i < a.length; i++) {
    sum += a[i];
}
```

Satsen `sum += a[i];` utförs n gånger och algoritmen får då $T(n) = n$.

Tidskomplexitet

- Med tidskomplexitet kan vi jämföra olika algoritmers effektivitet.
  - Tidskomplexiteten uttrycks så att vi är oberoende av vilken dator som exekverar algoritmen.
  - Vi behöver inte exekvera algoritmerna för att kunna jämföra deras effektivitet.
- låg tidskomplexitet $\iff$ effektiv $\iff$ billig, låg kostnad
  hög tidskomplexitet $\iff$ dålig effektivitet $\iff$ dyr, hög kostnad

Beräkna tidskomplexitet

Exempel

Beräkna tidskomplexiteten för att summera talen i en kvadratisk matris `m`:

```
for (int i = 0; i < m.length; i++) {
    for (int k = 0; k < m[i].length; k++) {
        sum += m[i][k];
    }
}
```

- Antag att matrisen innehåller $n \times n$ element.
- De satser som utförs flest gånger är de som finns i den innersta for-loopen.
- Satsen `sum += m[i][k];` utförs $n \cdot n$ gånger och algoritmen får då $T(n) = n^2$ (kvadratisk tidskomplexitet).

Diskutera

- Att summera n tal har tidskompexiteten n. Vad händer med exekveringstiden om n fördubblas?
- Summeringen av matriselementen i en nxn matris har tidskompexiteten n^2. Vad händer med exekveringstiden om n fördubblas?

Tidskomplexitet och verklig tidsåtgång

- Antag att vi räknat ut att tidskomplexiteten $T(n)$ för en viss algoritm.
- **Verklig tidsåtgång** $t(n)$ blir då $c \cdot T(n)$, där c är en konstant som bl.a. beror på hur snabb datorn är.
  - Ju snabbare dator, desto mindre är c
- Exempel: Summering av n tal:
  - $T(n) = n$
  - Verklig tidsåtgång $t(n) = c \cdot n$
  - c är tidsåtgången för additionerna och tilldelningarna på den dator där summeringen exekveras.
  - Exekveringstiden växer linjärt med antal element som ska summeras. Exempel: På en viss dator tar 17 ms att summera 1000 000 tal. Det bör då ta ungefär 34 ms att summera 2000 000 tal på samma dator.

Diskutera

Metoden nedan undersöker om alla element i vektorn a är unika. Exempel:
Om a = {1, 4, 2, 9, 7} ska resultatet bli `true`.
Om a = {1, 4, 2, 9, 2} ska resultatet bli `false`.

```
public static boolean allUnique(int[] a) {
    boolean unique = true;
    for (int i = 0; i < a.length - 1; i++) {
        for (int k = i + 1; k < a.length; k++) {
            if (a[i] == a[k]) {
                unique = false;
            }
        }
    }
    return unique;
}
```

Räkna antal jämförelser. Tidskomplexitet?

Tidskomplexitet

Undersök om alla element är unika

i-värde för yttre for-loopen	*Antal ggr satserna i inre for-loopen utförs*
0	$n-1$
1	$n-2$
...	...
$n-4$	3
$n-3$	2
$n-2$	1

Totalt antal gånger satserna i inre for-satsen utförs är:

$$1+2+3+\ldots+n-1 = n(n-1)/2$$

D.v.s. $T(n) = n^2/2 - n/2$ vilket är $\approx n^2/2$ för stora n-värden

Aritmetisk summa

Repetition från matematiken

Antag att skillnaden mellan två på varandra följande tal i en följd är konstant. Då kan vi beräkna summan av talen genom formeln:

$$a_1 + a_2 + a_3 + \ldots + a_n = n(a_1 + a_n)/2$$

Då blir

$$1+2+3+\ldots+n = n(1+n)/2$$

och

$$1+2+3+\ldots+n-1 = (n-1)(1+n-1)/2 = (n-1)n/2$$

Förbättrad algoritm

Undersök om alla element är unika

Vi bör avbryta algoritmen så fort vi hittat en dubblett:

```
public static boolean allUnique(int[] a) {
    for (int i = 0; i < a.length - 1; i++) {
        for (int k = i + 1; k < a.length; k++) {
            if (a[i] == a[k]) {
                return false;
            }
        }
    }
    return true;
}
```

Nu blir det svårare att beräkna exakt hur många gånger jämförelsen `(a[i] == a[k]` utförs. Det beror på indata. Enklast är att räkna på värsta fallet som här inträffar då inga dubbletter finns. Vi får då samma tidskomplexitet som för den oförbättrade algoritmen.

Tidskomplexitet

Undersök om alla element är unika

För (den oförbättrade) algoritmen fick vi fram uttrycket $T(n) = n(n-1)/2 = n^2/2 - n/2$ för tidskomplexiteten.

För stora värden på n dominerar den kvadratiska termen. $T(n) \approx n^2/2$ för stora värden på n.

n	$T(n) = n(n-1)/2$	$n^2/2$	$T(n)/(n^2/2)$
10	45	50	0.9
100	4 950	5 000	0.99
1 000	499 500	500 000	0.999
10 000	49 995 000	50 000 000	0.9999

Uttrycka tidskomplexitet med hjälp av Ordo

- För att uttrycka hur en algoritms tidskomplexitet uppför sig vid stora värden på n kan vi använda Ordo-notation.
  - Beräkna tidskomplexiteten genom att räkna hur många gånger de dominerande operationerna utförs.
    Exempel: $T(n) = n^2/2 - n/2$
  - Stryk sedan allt utom den dominerande termen. Stryk även ev. konstanter på den dominerande termen.
    Exempel: $T(n) = O(n^2)$
- Detta ger en övre gräns för tidskomplexiteten. $T(n) = O(n^2)$ betyder att för stora värden på n understiger den verkliga exekveringstiden cn^2.

Ordobegreppet

- **Ordo**-begeppet från matematiken ger oss möjlighet att begränsa en funktion "uppåt".
- En funktion $T(n)$ är $O(f(n))$ (uttalas "T är ordo f") om det finns konstanter n_0 och c så att

$$|T(n)| \leq cf(n) \textbf{ för alla } n > n_0$$

- För alla polynom T(n) av grad k:

$$T(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \ldots$$

gäller att de är $O(n^k)$.

Ordobegreppet

I komplexitetsberäkningar förutsätts att man inte ger en större övre gräns än nödvändigt:

- Om $T(n) = 2n^2 + n$ är $T(n) = O(n^2)$
- men det är även sant att $T(n) = O(n^3)$ eller $O(n^4)$ eller ...

Asymptotisk tidskomplexitet

- När man i stället för att ange ett exakt uttryck på tidskomplexiteten använder ordonotation brukar man tala om algoritmens **asymptotiska tidskomplexitet**.
- Man anger ju hur den asymptotiskt uppför sig för stora värden på n (problemets storlek).

Diskutera – tidskomplexitet och verklig tidsåtgång

För stora värden på n gäller att verklig tidsåtgång $t(n) = c \cdot T(n)$

Problem: En algoritm har tidskomplexiteten $T(n) = O(n^2)$. Antag att då man undersöker om elementen i en vektor med 10000 element är unika får en exekveringstid på 20 ms. Ungefär hur lång tid tar det att exekvera programmet för n = 1000000?

Storleksordning för funktioner

När en algoritm har tidskomplexiteten $T(n) = O(1)$, så säger man att tidskomplexiteten är konstant. På samma sätt används namn på vanligt förekommande storleksordningar för $T(n) = O(f(n))$, enligt tabellen:

f(n)	***namn***
1	konstant
$\log n$	logaritmisk
n	linjär
$n \log n$	log-linjär
n^2	kvadratisk
n^3	kubisk
2^n	exponentiell

Tidskomplexitet för några algoritmer

- Sätta in ett element först i en lista med n element: $O(1)$
- Söka med binärsökning bland n sorterade element: $O(\log n)$
- Söka bland n osorterade element: $O(n)$
- Sortera med bättre metoder som Mergesort, Heapsort och Quicksort (som kommer senare i kursen): $O(n \log n)$
- Sortera n tal med urvalssortering, bubbelsortering eller insättningssortering: $O(n^2)$
- Multiplicera två $n \times n$-matriser med den vanliga metoden: $O(n^3)$
- Knäcka lösenord med med n siffror genom att testa alla kombinationer: $O(10^n)$

Tidskomplexitet – hur mycket hjälper en snabbare dator?

En algoritm behandlar 1000 element på 1 sek på vår dator idag.

Inför framtiden hoppas vi på en 10 gånger så snabb dator.
Hur många element kan vi behandla på samma tid då?

Svaret beror på algoritmens tidskomplexitet.

$T(n)$	idag	framtiden
n	1 000	10 000
$n \log n$	1 000	7 717
n^2	1 000	3 162
n^3	1 000	2 154
2^n	1 000	**1 003**

Tidskomplexitet kan bero på indata

```
/** Undersök om talet x finns i heltalsvektorn a. */
public static boolean contains(int[] a, int x) {
   for (int i = 0; i < a.length; i++)  {
      if (a[i] == x) {
         return true;
      }
   }
   return false; // kommer vi hit fanns inte x i vektorn
}
```

- Det som görs inuti for-satsen dominerar.
- Komplikation: Går inte att säga exakt hur många gånger den utförs. Det beror på indata.

Tidskomplexitet – värsta fall och medelfall

- För algoritmer vars tidskomplexitet beror på indata kan man ställa två frågor:
  - Vad är den i värsta fall för instanser av storlek n?
  - Vad är den i medeltal för alla instanser av storlek n?

- Tidskomplexitet i **värsta fall** för instanser av storlek n brukar betecknas $W(n)$. W står för Worst case.
- Tidskomplexitet i **medelfall** för instanser av storlek n brukar betecknas $A(n)$. A står för Average.

Värsta fall och medelfall – formella definitioner

Om det finns k olika instanser av storlek n och tidskomplexiteten för den i:e instansen är $T_i(n)$ och sannolikheten för denna instans är P_i gäller:

$$W(n) = \max T_i(n)$$

$$A(n) = \sum P_i \cdot T_i(n)$$

- $W(n)$ är ofta lättare att beräkna än medelfallstid.
- $A(n)$ ofta svårare att berökna. För en del algoritmer okänt på grund av svårigheten att ge värden åt sannolikheter för de olika instanserna av storlek n.

Värsta fall och medelfall för linjärsökning

- Problemets storlek = vektorns längd. Kalla denna n, d.v.s. n = a.length.
- I värsta fall blir jämförelsen alltid false och for-satsen utförs då för alla i-värden från 0 till och med $n-1$.
- Satserna inuti for-satsen utförs då n gånger.
- Alltså: $W(n) = n = O(n)$.

- Medelfallstiden är svår att beräkna eftersom det finns ett oändligt antal instanser av problemet för storlek n.
- Men under vissa förutsättningar kan man räkna fram medefallstiden $A(n) = (n+1)/2 = O(n)$.

Tidskomplexitet för algoritmer som anropar metoder

De flesta algoritmer uttrycker sig inte enbart med basala $O(1)$-operationer som $+$, $-$, . . . utan unyttjar mera kraftfulla operationer.

Exempel: Sortera en vektor och undersök sedan om alla element är unika.

```
public static boolean allUnique(int[] a) {
    Arrays.sort(a);
    for (int i = 0; i < a.length - 1; i++) {
        if (a[i] == a[i + 1]) {
            return false;
        }
    }
    return true;
}
```

Tidskomplexitet? Bättre eller sämre än vår tidigare algoritm?

Tidskomplexitet för algoritmer som anropar metoder

Antag att vektorn innehåller n element.

- Algoritmen gör ett anrop av metoden sort som *inte* är $O(1)$, sedan görs i värsta fall $n-1$ jämförelser som är $O(1)$.
  - Metoden `sort` innehåller en effektiv sorteringsalgoritm med $T(n) = O(n \log n)$.
  - De $n-1$ jämförelserna kostar sammanlagt $O(n)$.
- Sammanlagd kostnad för algoritmen blir $O(n \log n) + O(n)$. $T(n) = O(n \log n)$

D.v.s. sorteringen dominerar i detta fall och bestämmer tidskomplexiteten.

Tidskomplexitet för algoritmer som anropar metoder

Exempel: Antag att en vektor a innehåller n tal. I följande algoritm läggs alla unika tal från a in i en lista.

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < a.length; i++) {
    if (! list.contains(a[i])) {
        list.add(a[i]);
    }
}
```

Vad får algoritmen för tidskomplexitet?

Tidskomplexitet för algoritmer som anropar metoder

```
for (int i = 0; i < a.length; i++) {
    if (! list.contains(a[i])) {
        list.add(a[i]);
    }
}
```

1. Vilka metoder anropas? – `contains` och `add`
2. Hur många gånger anropas var och en av dem i värsta fall? – n gånger där n är antal element i vektorn
3. Vad har de för tidskomplexitet i värsta fall?

- För att besvara fråga 3 behöver vi veta antal element i listan. Problemet är att det varierar. Listan är tom från början och att antalet element ökar efter hand. Men vi räknar med det maximala antalet. Listan kan som mest innehålla $n = a.length$ element (om inga dubbletter finns).

Tidskomplexitet för algoritmer som anropar metoder

```
for (int i = 0; i < a.length; i++) {
    if (! list.contains(a[i])) {
        list.add(a[i]);
    }
}
```

- Antag att $n = a.length$. (I värsta fall innehåller listan $a.length$ element.)
- Inuti `contains` söks i värsta fall alla element igenom. Denna sökning har tidskomplexiteten $O(n)$. De n anropen av `contains` kostar då $O(n^2)$.
- I `add` läggs elementet till sist. Denna operation har tidskomplexiteten $O(1)$. I värsta fall anropas `add` n gånger vilket kostar $O(n)$.
- Sammanlagd kostnad för algoritmen blir $O(n^2) + O(n)$ vilket innebär att $T(n) = O(n^2)$.

Tidskomplexitet för algoritmer som anropar metoder

Allmänna fallet

Antag att en algoritm

- anropar metoderna $o_1, o_2, ..., o_k$.
- Metoderna o_i anropas m_i gånger.
- Metoderna o_i har värstafalltidskomplexiteten $W_i(n)$.

Då blir kostnaden för algoritmen i värsta fall:

$$m_1 \cdot W_1(n) + m_2 \cdot W_2(n) + ... + m_k \cdot W_k(n)$$

Deltermer som *inte* dominerar detta uttryck kan sedan strykas om man bara vill ha en ordo-uppskattning.

Bättre datastruktur för att lagra en samling unika element

En mängd (eng. Set) passar bättre än en lista om man ska lagra unika element.

```
Set<Integer> list = new HashSet<Integer>();
for (int i = 0; i < a.length; i++) {
   list.add(a[i]);
}
```

- Dubblettkontroll sker inuti `add`. Metoden `add` i klassen `HashSet` har i medelfall (och i praktiken) tidskomplexiteten $O(1)$.
  (Mängder och klasserna `HashSet` behandlas senare under kursen.)
- Nackdelen med den här lösningen är att den ordningen mellan elementen som fanns i vektorn förstörs.

Diskutera

Vad får följande algoritm för tidskomplexitet?

```
List<Integer> list = new LinkedList<Integer>();
...
int s = 0;
for (int i = 0; i < list.size(); i++) {
    s += list.get(i);
}
System.out.println(s);
```

Ovanstående sätt att iterera genom en länkad lista är ineffektivt. Istället bör man använda en iterator. Vilken tidskomplexitet får algoritmen då?

```
for (int nbr : list) {
    s += nbr;
}
```


Exempel på vad du ska kunna

- Förklara begreppet tidskomplexitet.
- Beräkna tidskomplexiteten $T(n)$ för enkla algoritmer.
- Uttrycka tidskomplexitet med O (ordo).
- Förstå vad olika storleksordningar på tidskomplexiteten innebär för algoritmers användbarhet (konstant, logaritmisk, linjär, kvadratisk, exponentiell,...).
- Förstå och kunna använda sambandet mellan verklig tidsåtgång och teoretiskt beräknad tidskomplexitet, $t(n) \approx c \cdot T(n)$.
- Avgöra när en algoritms tidskomplexitet beror på indata och i sådana fall kunna beräkna tidskomplexitet för värsta fall, $W(n)$.
- Beräkna tidskomplexitet för algoritmer som använder färdiga operationer.