Tidskompleksitet

Christian Stefansen

cstef@diku.dk

2002-05-23

"People who analyze algorithms have double happiness."
– Donald E. Knuth

Datalogi 0GB

Tidsforbrug og tidskompleksitet

- Hvorfor tidskompleksitet?
- Intuitiv forklaring af $\mathcal{O}$-notation
- Matematisk definition af $\mathcal{O}$-notation
- Θ (store theta) og Ω (store omega)
- Fra program til tidsforbrug til tidskompleksitet
  - Et lineært eksempel ($\mathcal{O}(n)$)
  - Et kvadratisk eksempel ($\mathcal{O}(n^2)$)
  - Et logaritmisk eksempel ($\mathcal{O}(\lg n)$)
  - Et eksponentielt eksempel ($\mathcal{O}(2^n)$)
- Om tidsforbrug og tidskompleksitet i praksis
- Tidskompleksitet af flere variable
- Tidskompleksitet som funktion af forskellige mål af inddata

- Tabel over datastrukturer og deres tidskompleksitet
- Sorteret liste over ofte sete tidskompleksiteter

Hvorfor tidskompleksitet?

- Fra konkrete målinger til generelle udtalelser om programmets effektivitet
- Hvordan vokser tids- eller lagerforbrug som funktion af inddata?
- Konstanter faktoriseres væk af hurtigere maskiner

Intuitiv forklaring af $\mathcal{O}$-notation

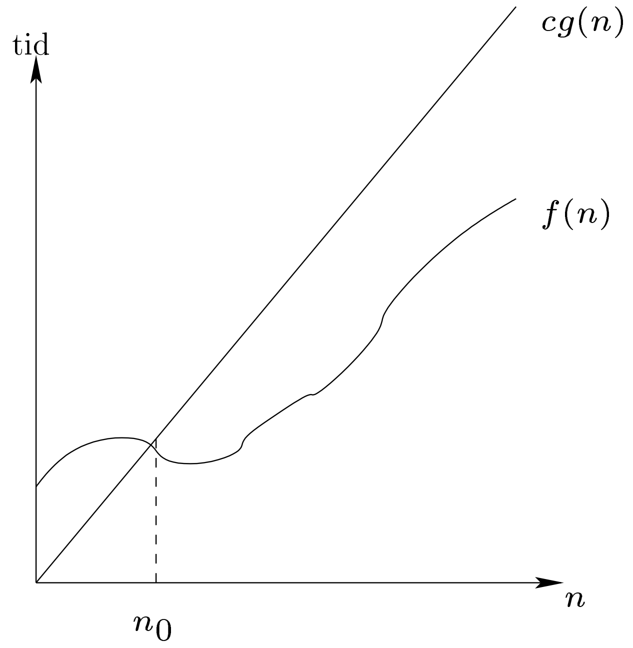
- Forhold mellem inddatas størrelse og ressourceforbrug (oftest tidsforbrug eller lagerforbrug)
- Givet inddatas størrelse kan vi sætte en øvre grænse for ressourceforbruget
- $\mathcal{O}$-notation kan gemme store konstanter. For små inddata er en dårlig funktion måske den bedste!

Matematisk definition af $\mathcal{O}$-notation

f er $\mathcal{O}(g(n))$ kun hvis

$$\exists c \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$$

$\mathcal{O}(g(n))$ er en mængde af funktioner, men ofte skriver man alligevel $f(n) = \mathcal{O}(g(n))$, når man mener $f(n) \in \mathcal{O}(g(n))$. Det tillades også at sjuske[1], så man kan have udtryk som $1 + n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$.



Figur 1: Lad $g(n)$ være lineær, så er $f(n) = \mathcal{O}(n)$

[1]Man skal altid huske at være opmærksom på, hvad man gør. Notationen er en lettelse fra en lidt tungere formalisme, men kan snyde. Vær varsom!

Θ (store theta) og Ω (store omega)

- $\mathcal{O}$ sætter en øvre grænse.
- Ω sætter en nedre grænse.
- Θ sætter både en øvre og en nedre grænse.

f er $\mathcal{O}(g)$ kun hvis

$$\exists c \exists n_0 \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$$

f er $\Omega(g)$ kun hvis

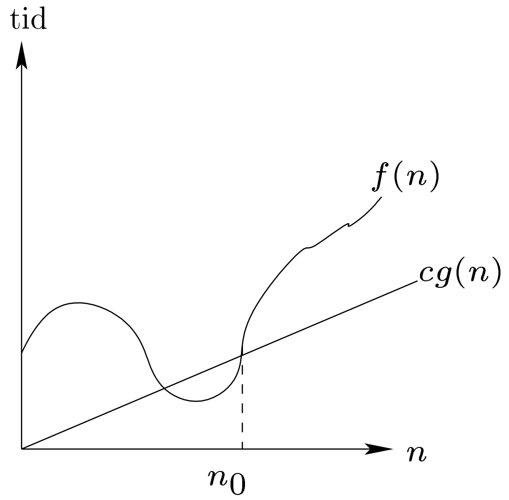
$$\exists c \exists n_0 \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)$$

f er $\Theta(g)$ kun hvis

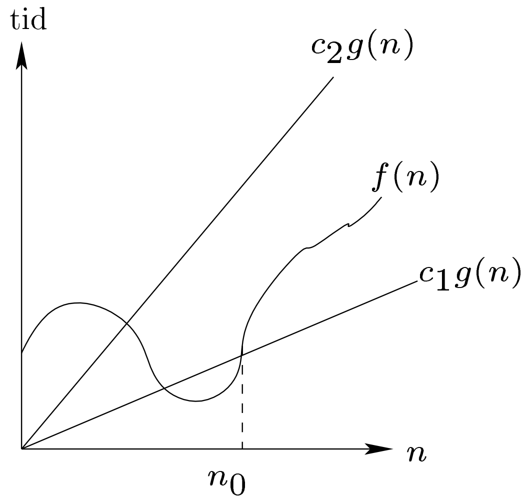
$$\exists c_1 \exists c_2 \exists n_0 \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Videre findes ω (lille omega) og o (lille o), som gennemgås i algoritmikkurset, Dat2A.

Illustration af Ω og Θ



Figur 2: $f(n) = \Omega(g(n))$



Figur 3: $f(n) = \Theta(g(n))$

Fra program til tidsforbrug til tidskompleksitet

Lad c_1, c_2, c_3 og c_4 være den tid det tager, at udføre den linje de står ud for.

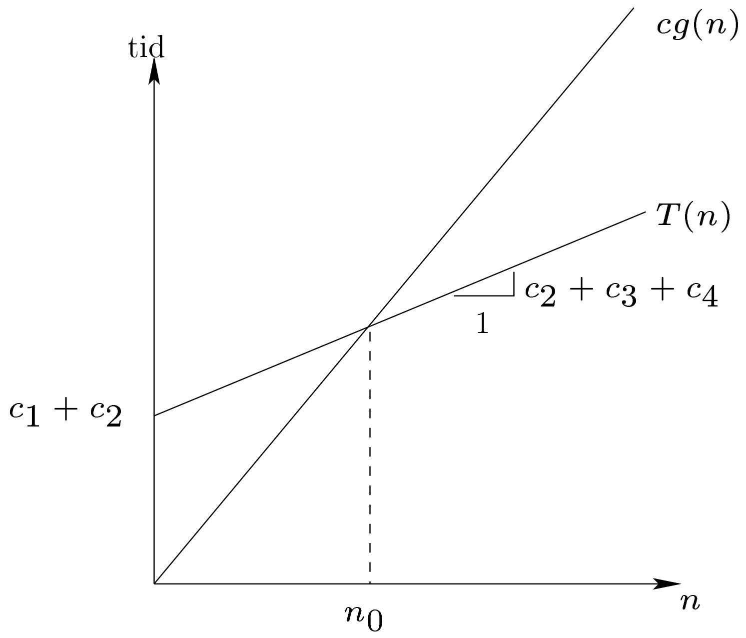
```
for (int i = 0;                          (c1) (1)
     i < n;                              (c2) (n+1)
     i++)                                (c3) (n)
    sum++;                               (c4) (n)
```

$T(n)$ er tiden programmet bruger for inddata af størrelsen n.

$$\begin{aligned} T(n) &= c_1 + (n+1)c_2 + nc_3 + nc_4 \\ &= (c_2 + c_3 + c_4)n + c_1 + c_2 \end{aligned}$$

Fra program til tidsforbrug til tidskompleksitet

$$\begin{aligned} T(n) &= c_1 + (n+1)c_2 + nc_3 + nc_4 \\ &= (c_2 + c_3 + c_4)n + c_1 + c_2 \end{aligned}$$



Påstand: $T(n) = \mathcal{O}(n)$

Hvorfor: Lad $\alpha = c_2 + c_3 + c_4$ og $\beta = c_1 + c_2$. Vi kan sætte $c = 2 \cdot \alpha$. Da er $cg(n)$ stejlere. Skæringspunkt findes, hvor $\alpha n + \beta = cn$. Ved udregning fås $n_0 = \frac{\beta}{\alpha}$.

Et lineært eksempel

```
for (int i = 0;                           (c1) (1)
     i < n;                               (c2) (n+1)
     i++)                                 (c3) (n)
    sum++;                                (c4) (n)
for (int i = 0;                           (c1) (1)
     i < n;                               (c2) (n+1)
     i++)                                 (c3) (n)
    sum++;                                (c4) (n)
```

$$
\begin{aligned}
T(n) &= c_1 + (n+1)c_2 + nc_3 + nc_4 + \\
&\quad\; c_1 + (n+1)c_2 + nc_3 + nc_4 \\
&= 2(c_2 + c_3 + c_4)n + 2c_1 + 2c_2
\end{aligned}
$$

Tiden er altså en affin funktion af inddata og funktionen er $\mathcal{O}(n)$. Faktisk er den ikke blot $\mathcal{O}(n)$, men $\Theta(n)$.

(Endda er funktioner, der er $\mathcal{O}(n)$, også $\mathcal{O}(n^2)$, så T er også $\mathcal{O}(n^2)$, men dette er en mindre præcis øvre grænse.)

Et kvadratisk eksempel

```
for (int i = 1; i <= n; i++)            (c1) (n)
    for (int j = 1; j <= i; j++)        (c2) (n*n)
        sum++;                          (c3) (n*n)
```

Programstumpen c_3 gennemløbes 1 gang for $i = 1$, 2 gange for $i = 2$, 3 gange for $i = 3$, . . . , n gange for $i = n$. Altså:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

(Kan vises ved matematisk induktion!)

$$\begin{aligned} T(n) &= c_1 n + c_2(\frac{1}{2}n^2 + \frac{1}{2}n) + c_3(\frac{1}{2}n^2 + \frac{1}{2}n) \\ &\leq cn^2 \end{aligned}$$

Man behøver ikke beviset, da man blot kan opvurdere i til n i c_2. Så bliver tidsforbruget:

$$T(n) = c_1 n + (c_2 + c_3)n^2 \leq cn^2$$

Tidskompleksiteten er altså $\mathcal{O}(n^2)$.

Et logaritmisk eksempel

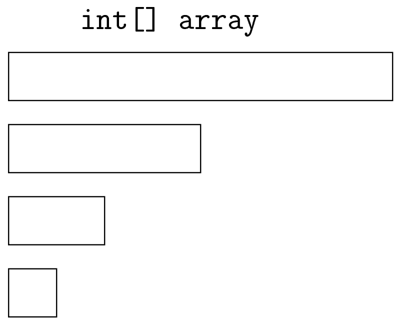
```
int search (int[] A, int key,
            int start, int stop) {
  if (start > stop)                      (c1)
    return -1;

  int trythis = (start+stop)/2;          (c2)

  if (A[trythis] == key)                 (c3)
    return trythis;
  if (A[trythis] < key)                  (c4)
    return search (A, key, trythis+1, stop);
  else
    return search (A, key, start, trythis-1);
}
```

Lad n betegne længden af arrayet `int[] A`.

$$\begin{aligned} T(n) &\leq c_1 + c_2 + c_3 + c_4 + T(\lfloor n/2 \rfloor) \\ &\leq c\lceil \lg n \rceil \end{aligned}$$

Idet vi kan vælge c, så $c \geq c_1 + c_2 + c_3 + c_4$.

Et logaritmisk eksempel



Figur 4: Binær søgning

$$
\begin{aligned}
T(n) &\leq c_1 + c_2 + c_3 + c_4 + T(\lfloor n/2 \rfloor) \\
&\leq c\lceil \lg n \rceil
\end{aligned}
$$

Sidste skridt følger af:

intuitivt For hver rekursion foretages en halvering af inddata. Antal gange et tal n kan halveres før det bliver mindre eller lig 1 er netop $\lceil \lg n \rceil$.

matematisk Kan vises ved induktion

Tidskompleksiteten er altså $\mathcal{O}(\lg n)$.

Et eksponentielt eksempel

```
int fibonacci (int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci (n-1) +
               fibonacci (n-2);
}
```

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_1 \\ &\leq c2^n \end{aligned}$$

Her er c_1 det samlede konstante tidsforbrug i `fibonacci` for én rekursion.

Hver rekursion betyder en fordobling af tidsforbruget, derfor fås $\mathcal{O}(2^n)$.

Om tidsforbrug og tidskompleksitet i praksis

- Tidsforbrug
  - Præcis, men ofte omstændig at udrede
  - Kan bruges som hjælp, til at finde ud af, hvor mange gange de enkelte dele af programmet gennemløbes
  - Sjældent besværet værd, når først man er blevet fortrolig med tidskompleksitet
- Tidskompleksitet
  - Praktisk og langtidsholdbar
  - Kan gemme betydelige konstanter
  - Til praktiske formål, bør en evt. maksimale størrelse af inddata tages i betragtning. For små inddata er den asymtotisk bedste algoritme ikke nødvendigvis den hurtigste.

Sommetider er man interesseret i andre ressourcer, som f.eks. lager eller IO-operationer. Hertil kan man så analysere lagerkompleksitet eller IO-kompleksitet.

Tidskompleksitet af flere variable

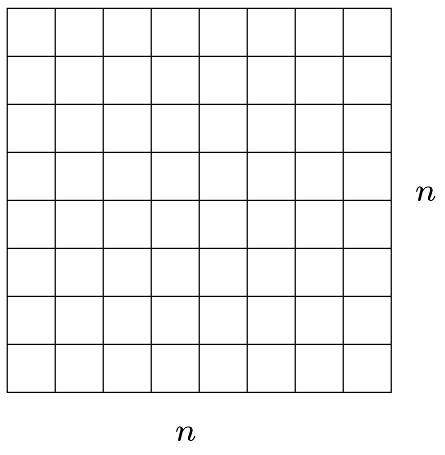
Her er n, s og k inddata:

```
for (int i = 0; i < n; i++)
    sum++;                          (c1) (n)
for (int i = 0; i < s * s; i++)
    sum++;                          (c2) (s*s)
for (int i = 0; i < k / 2; i++)
    sum++;                          (c3) (k/2)
```

$$\begin{aligned} T(n) &= c_1 n + c_2 s^2 + c_3 \frac{k}{2} \\ &\leq c(n + s^2 + k) \end{aligned}$$

Tidskompleksiteten bliver altså $\mathcal{O}(n + s^2 + k)$.

Tidskompleksitet som funktion af forskellige mål af inddata



Figur 5: Et kvadrat med $n * n = m$ felter

Betragt en algoritme P som besøger alle felter i kvadratet, hvor hvert besøg koster $\mathcal{O}(1)$.

Da er tidskompleksiteten af P $\mathcal{O}(n^2)$ eller $\mathcal{O}(m)$. Tidskompleksiten afhænger altså af, hvilket mål for inddatas størrelse, den er funktion af.

Vi siger, at P er lineær i antal felter eller kvadratisk i sidernes længde.

En simpel kodebryder

Mange krypteringsalgoritmer bl.a. RSA er baseret på, at det er vanskeligt at faktorisere store heltal.

Inddata: Et meget stort heltal n
Opgave: Det vides, at $n = p_1 \cdot p_2$, hvor p_1 og p_2 er meget store primtal.
Uddata: p_1 og p_2

Antag at vi kan checke, om et tal q går op i n i konstant tid.

Tidskompleksiteten er altså $\mathcal{O}(n)$, da man blot kan prøve alle muligheder efter.

Dog er man ofte interesseret i tidskompleksiteten som funktion af størrelsen af inddata snarere end størrelsen af n.

Længden af n er $m = \lg n$ og herved bliver tidskompleksiteten $\mathcal{O}(2^m)$. Hvis altså længden af tallet n øges med blot 1 bit, vil algoritmen tage dobbelt så lang tid.

Hvis det binære tal n er blot 600 bit langt, vil det være en beregningsmæssigt uoverkommelig opgave at bryde koden.

Tabel over datastrukturer og deres tidskompleksitet

	indsæt	tilføj	opdater	hent
`LinkedList`	$\mathcal{O}(n)/\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)/\mathcal{O}(1)$	$\mathcal{O}(n)/\mathcal{O}(1)$
`ArrayList`[2]	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`TreeMap`	$\mathcal{O}(\lg n)$		$\mathcal{O}(\lg n)$	$\mathcal{O}(\lg n)$
`HashMap`[3]	$\mathcal{O}(1)$		$\mathcal{O}(1)$	$\mathcal{O}(1)$

indsæt indsættelse på en bestemt plads

tilføj tilføjelse efter sidste plads

opdater ændring af indholdet for en given plads/nøgle

hent hent indhold fra en given plads/nøgle

Indsæt, opdater og hent tager $\mathcal{O}(n)$ for `LinkedList`, hvis man starter fra begyndelsen af listen. Ofte befinder man sig dog allerede på den plads, der ønskes berørt, og da kan operationerne udføres i konstant tid.

I Java gøres dette ved at anvende en iterator.

[2] Indsættelse og tilføjelse kører amortiseret konstant tid.

[3] Alle operationer kører forventet konstant tid.

Sorteret liste over ofte sete tidskompleksiteter

Tidskompleksitet	Navn
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\lg \lg n)$	
$\mathcal{O}(\lg n)$	logaritmisk
$\mathcal{O}(\sqrt{n})$	
$\mathcal{O}(n)$	lineær
$\mathcal{O}(n \lg \lg n)$	
$\mathcal{O}(n \lg n)$	
$\mathcal{O}(n^2)$	kvadratisk
$\mathcal{O}(n^3)$	kubisk
$\mathcal{O}(n^k)$	polynomiel (for fast k)
$\mathcal{O}(2^n)$	eksponentiel

Prims algoritme for mindste udspændende træ

```
static Vertex[] doPrim (Vertex[] V, Vertex[][] adj,
                        int[][] weight, Vertex root) {

  // Dan kø. Alle nøgler sættes "uendeligt store".
  PriorityQueue Q = new PriorityQueue(V);            O(V)

  Q.decreaseKey(root, 0);

  Vertex[] parent = new Vertex[V.length];
  // Knuders "navn" er blot et unikt heltal
  parent[root.name] = null;

  while (!Q.isEmpty()) {                             O(V)
    Vertex u = Q.dequeueMin();                       O(VlgV)
      for (int i = 0; adj[u.name][i] != null; i++) { O(E)
        Vertex v = adj[u.name][i];
        if (Q.contains(v) &&
            weight[u.name][v.name] < Q.getKey(v)) {
          parent[v.name] = u;
          Q.decreaseKey(v, weight[u.name][v.name]);  O(ElgV)
        }
      }
  }
  return parent;
}
```

Prims algoritme for mindste udspændende træ

Tidskompleksitet af Prims algoritme:

$$2\mathcal{O}(V) + \mathcal{O}(V \lg V) + \mathcal{O}(E) + \mathcal{O}(E \lg V) = \mathcal{O}(E \lg V)$$

Litteratur

CLR Cormen, Leiserson, Rivest: *Introduction to Algorithms*, The MIT Press, 1997

MAW Mark Allen Weiss: *Data Structures & Problem Solving using Java*, Addison-Wesley, 1998