Föreläsning 4: Dekomposition

Dekomposition

Dekomposition är en generell metod för att lösa problem. Metoden bygger på att man delar upp ett problem i delproblem **av samma typ** som ursprungsproblemet. Uppdelningen fortsätter sedan tills problemet är nerbrutet i många triviala delproblem. Resultaten från delproblemen sätts sedan ihop till en lösning av ursprungsproblemet. Komplexitetens beroende av problemets storlek kan oftast beräknas genom analys av ett rekursionssamband.

Sortering

Två algoritmer för sortering i tid $O(n \log n)$: Mergesort, Ouicksort. Båda bygger på dekomposition.

MergeSort

```
MergeSort(v[i..j])
(1)     if i < j
(2)         m ← ⌊(i+j)/2⌋
(3)         MergeSort(v[i..m])
(4)         MergeSort(v[m+1..j])
(5)         v[i..j] = Merge(v[i..m], v[m+1..j])
```

Låt $T(N)$ vara tiden för att sortera N tal. Då gäller

$$T(N) = \begin{cases} O(1) & N = 1 \\ T\left(\lfloor \frac{N}{2} \rfloor\right) + T\left(\lceil \frac{N}{2} \rceil\right) + \Theta(N) & \text{annars} \end{cases}$$

eftersom Merge tar tid $\Theta(N)$ när indata är vektorer av längd N.
Denna typ av rekursionssamband är vanlig vid analys av algoritmer.

Master Theorem

Sats. *För $a \geq 1$, $b > 1$ och $d > 0$ har rekursionsekvationen*

$$T(1) = d$$
$$T(n) = aT(n/b) + f(n)$$

lösningen

- $T(n) = \Theta(n^{\log_b a})$ *om* $f(n) = O(n^{\log_b a - \epsilon})$ *för något* $\epsilon > 0$
- $T(n) = \Theta(n^{\log_b a} \log n)$ *om* $f(n) = \Theta(n^{\log_b a})$
- $T(n) = O(f(n))$ *om* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *för något* $\epsilon > 0$ *och* $af(n/b) \leq cf(n)$ *för något* $c < 1$ *för alla tillräckligt stora* n.

Tillämpad på analysen av MergeSort fås att tidskomplexiteten är $\Theta(N \log N)$.

Mer om dekomposition

Vi vill bestämma $x \cdot y$ där x och y binärt ges av

$$x = \underbrace{x_{n-1} \cdots x_{n/2}}_{a} \underbrace{x_{n/2-1} \cdots x_1 x_0}_{b} = 2^{n/2}a + b$$

$$y = \underbrace{y_{n-1} \cdots y_{n/2}}_{c} \underbrace{y_{n/2-1} \cdots y_1 y_0}_{d} = 2^{n/2}c + d$$

För $n = 2^k$ kan vi använda dekomposition:

```
MULT(x, y)
(1)     if length(x) = 1
(2)         return x · y
(3)     else
(4)         [a, b] ← x
(5)         [c, d] ← y
(6)         prod ← 2^n Mult(a, c) + Mult(b, d)
                   +2^{n/2}(Mult(a, d) + Mult(b, c))
(7)         return prod
```

Tidskomplexitet: $T(n) = 4T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$ så $T(n) = \Theta(n^2)$.

Karatsubas multiplikationsalgoritm

Utnyttja $(a+b)(c+d) = ac + bd + (ad + bc)$.
En av de fyra multiplikationerna kan tas bort:

```
MULT(x, y)
(1)   if length(x) = 1
(2)       return x · y
(3)   else
(4)       [a, b] ← x
(5)       [c, d] ← y
(6)       ac ← Mult(a, c)
(7)       bd ← Mult(b, d)
(8)       abcd ← Mult(a + b, c + d)
(9)       return 2^n · ac + bd +
          2^{n/2}(abcd − ac − bd)
```

Nu fås $T(n) = 3T(n/2) + \Theta(n)$, $T(1) = \Theta(1)$ med lösningen $T(n) = \Theta(n^{\log_2 3}) \in O(n^{1.59})$.
Ännu bättre prestanda kan fås om man delar talen i tre eller fler delar — r delar ger $O(n^{\log_r(2r-1)})$.
I praktiken är det inte säkert att detta lönar sig om inte n är väldigt stort. Dekomposition leder till en del overhead som kan ta en stor del av tiden för små instanser.

Matrismultiplikation

Vid multiplikation av $n \times n$-matriser kan man använda blockmatrisform:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Genom sambanden

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

får man 8 multiplikationer och således

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 8T(n/2) + \Theta(n^2) & n > 1 \end{cases}$$

och därmed $T(n) = \Theta(n^3)$.

Strassens algoritm

Genom sambanden

$$M_1 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_2 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_3 = (A_{11} - A_{21})(B_{11} + B_{12})$$
$$M_4 = (A_{11} + A_{12})B_{22}$$
$$M_5 = A_{11}(B_{12} - B_{22})$$
$$M_6 = A_{22}(B_{21} - B_{11})$$
$$M_7 = (A_{21} + A_{22})B_{11}$$

$$C_{11} = M_1 + M_2 - M_4 + M_6$$
$$C_{12} = M_4 + M_5$$
$$C_{21} = M_6 + M_7$$
$$C_{22} = M_2 - M_3 + M_5 - M_7$$

kan antalet multiplikationer minskas från 8 till 7 så $T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$.

Snabb Fouriertransform (FFT)

Givet två polynom $A(x) = \sum_{j=0}^{n-1} a_j x^j$ och $B(x) = \sum_{j=0}^{n-1} b_j x^j$, hur bestämmer man effektivt produkten $C(x) = \sum_{j=0}^{2(n-1)} c_j x^j$?
Hur fort det går beror på vilken representation som används.
Två vanliga representationer:

- **Koefficientform**
  Ett polynom representeras som de n koefficienterna.
- **Punkt-värde-form**
  Polynomet $A(x)$ representeras som n par (x_i, y_i) där $y_i = A(x_i)$. De n talen x_i är fixa.

Koefficientform:
Att evaluera ett polynom $A(x) = \sum_{j=0}^{n-1} a_j x^j$ på koefficientform kräver n multiplikationer med Horners regel:

$$A(x) = a_0 + x(a_1 + x(a_2 + x(a_3 \cdots)))$$

Att addera $A(x) = \sum_{j=0}^{n-1} a_j x^j$ och $B(x) = \sum_{j=0}^{n-1} b_j x^j$ tar $O(n)$ tid medan det tar $\Theta(n^2)$ tid att multiplicera dem.
Går det att multiplicera polynom snabbare?
Punkt-värde-form:
Att multiplicera två polynom på punkt-värde-form går snabbt ($O(n)$ multiplikationer) om de är evaluerade i samma $2n$ punkter:
Produkten av $\{(x_i, y_i)\}$ och $\{(x_i, z_i)\}$ är $\{(x_i, y_i z_i)\}$ och summan är $\{(x_i, y_i + z_i)\}$.
Å andra sidan är det knepigare att evaluera ett polynom på punkt-värde-form i en godtycklig punkt.
Att gå från punkt-värde-form till koefficientform (d.v.s. interpolera) kan göras på $\Theta(n^2)$ tid med Lagranges interpolationsformel:

$$A(x) = \sum_{j=0}^{n-1} y_j \frac{\prod_{k \neq j}(x - x_k)}{\prod_{k \neq j}(x_j - x_k)}$$

Schematisk multiplikationsalgoritm:
Koefficientform lämpar sig bra för evaluering i godtyckliga punkter medan punkt-värde-form lämpar sig bra för multiplikation. Kombinera styrkorna hos de båda representationerna!
Multiplikation av $A(x)$ och $B(x)$:

1. Evaluera polynomen i punkterna x_i.
2. Multiplicera polynomen på punkt-värde-form.
3. Interpolera tillbaka till koefficientform.

Problem: Hur ska punkterna x_i väljas så att evalueringen och interpolationen går snabbt?

Diskreta fouriertransformen

Evaluera polynomen i de komplexa enhetsrötterna $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ där $\omega_n = e^{2\pi i/n}$.
Transformen av $A(x) = \sum_{j=0}^{n-1} a_j x^j$ skrivs

$$DFT_n(\langle a_0, \ldots, a_{n-1}\rangle) = \langle y_0, \ldots, y_{n-1}\rangle$$

där

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j e^{2\pi ijk/n}.$$

De n koefficienterna ger n "frekvenser". Jämför

$$\hat{f}(t) = \int_{-\infty}^{\infty} f(x)e^{-itx}dx$$

som är den kontinuerliga transformen.

FFT: Effektiv beräkning av DFT

Vi har $y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j e^{2\pi ijk/n}$. Separera udda och jämna gradtal i A:

$$\begin{aligned} A(x) &= \sum_{j=0}^{n/2-1} a_{2j}(x^2)^j + x\sum_{j=0}^{n/2-1} a_{2j+1}(x^2)^j \\ &= A^{[0]}(x^2) + xA^{[1]}(x^2) \end{aligned}$$

För $k < n/2$ är

$$\begin{aligned} A^{[0]}(\omega_n^{2k}) &= \sum_{j=0}^{n/2-1} a_{2j} e^{4\pi ijk/n} \\ &= \sum_{j=0}^{n/2-1} a_{2j}\omega_{n/2}^{jk} \\ &= DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)_k \end{aligned}$$

där $DFT_n(\langle a_0, \ldots, a_{n-1}\rangle)_k$ är det k:te elementet av transformen.
På samma sätt fås, för $k < n/2$,

$$A^{[1]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)_k$$

För $k \geq n/2$ kan man lätt visa att

$$A^{[0]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)_{k-n/2}$$
$$A^{[1]}(\omega_n^{2k}) = DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)_{k-n/2}$$
$$\omega_n^k = -\omega_n^{k-n/2}$$

För att bestämma $DFT_n(\langle a_0, \ldots, a_{n-1}\rangle)$ utgår vi alltså från $DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)$ och $DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)$ och kombinerar ihop lämpliga värden.
FFT är ett exempel på dekomposition — basfallet är $DFT_1(\langle a_0\rangle) = \langle a_0\rangle$.

Algoritm för FFT

Vi antar att n är en tvåpotens.

$DFT_n(\langle a_0, a_1, \ldots a_{n-1}\rangle)$
(1) **if** $n = 1$
(2) **return** $\langle a_0\rangle$
(3) $\omega_n \leftarrow e^{2\pi i/n}$
(4) $\omega \leftarrow 1$
(5) $y^{[0]} \leftarrow DFT_{n/2}(\langle a_0, a_2, \ldots, a_{n-2}\rangle)$
(6) $y^{[1]} \leftarrow DFT_{n/2}(\langle a_1, a_3, \ldots, a_{n-1}\rangle)$
(7) **for** $k = 0$ **to** $n/2 - 1$
(8) $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
(9) $y_{k+n/2} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
(10) $\omega \leftarrow \omega \cdot \omega_n$
(11) **return** $\langle y_0, y_1, \ldots, y_{n-1}\rangle$

Tidskomplexiteten $T(n)$ uppfyller

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

med lösningen $T(n) = \Theta(n \log n)$.

Invers till DFT

För att gå tillbaka från punkt-värde-form till koefficientform måste DFT inverteras.
Relationen $y = DFT_n(a)$ kan på matrisform skrivas

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Inversen $a = DFT_n^{-1}(y)$ svarar mot att matrisen ovan inverteras. Man kan visa att

$$DFT_n^{-1}(\langle y_0, y_1, \ldots, y_{n-1}\rangle) = \langle a_0, a_1, \ldots, a_{n-1}\rangle$$
$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-jk}$$

så FFT-algoritmen kan användas också för
bestämning av DFT^{-1}.

Polynommultiplikation med FFT

Vi vill bestämma $C(x) = \sum_{j=0}^{2n-2} c_i x^i = A(x)B(x)$ där $A(x)$ och $B(x)$ har grad $n-1$. Eftersom $C(x)$ har $2n-1$ koefficienter duger det inte att arbeta med DFT_n — vi måste betrakta $A(x)$ och $B(x)$ som polynom av grad $2n-1$.
Algoritm:
$\langle y_0, \ldots y_{2n-1}\rangle \leftarrow DFT_{2n}(\langle a_0, \ldots, a_{n-1}, 0, \ldots, 0\rangle)$
$\langle z_0, \ldots z_{2n-1}\rangle \leftarrow DFT_{2n}(\langle b_0, \ldots, b_{n-1}, 0, \ldots, 0\rangle)$
$\langle c_0, \ldots c_{2n-1}\rangle \leftarrow DFT_{2n}^{-1}(\langle y_0 z_0, \ldots, y_{2n-1} z_{2n-1}\rangle)$
(Detta förutsätter att n är en tvåpotens.)
Tre DFT på vektorer av längd $2n$ och $2n$ multiplikationer i transformplanet $\Rightarrow$ $\Theta(n \log n)$ tid.

Tillämpningar av FFT

Inom bl.a. följande områden använder man DFT eller besläktade transformer.

- Signalteori
  DFT ger den diskreta motsvarigheten till frekvensspektrum.
- Bildbehandling
  DFT och DCT (diskreta cosinustransformen) används för analys och komprimering.
- Aritmetik på stora tal
  De effektivaste algoritmerna för multiplikation (tid $O(n \log n \log\log n)$) använder FFT.