AULA 10

- Última aula antes da P1.
- Prova será entregue no dia 11/8 com prazo para devolução 17/8 (inadiável)
-

MAIS ALGUNS EXEMPLOS DE DIVIDIR E CONQUISTAR

— Algoritmo de Strassen (1969): multiplicação matricial

Queremos calcular $C = A \cdot B$, em que A e B (e consequentemente também C) são matrizes $n \times n$. Assuma, por simplicidade, que n é potência de 2.

- Algoritmo "ingênuo":

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Tempo: $O(n^3)$ somas e multiplicações

- Dividir e conquistar "ingênuo"

Multiplicação em blocos:

$$\underbrace{\begin{pmatrix} r & s \\ t & u \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{B} \underbrace{\begin{pmatrix} e & f \\ g & h \end{pmatrix}}_{C}$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\} \text{8 multiplicações de matrizes } n/2 \times n/2, \text{ calculadas recursivamente}$$

Tempo:
$$\begin{cases} T(1) = O(1) \\ T(n) = 8\,T\left(\frac{n}{2}\right) + O(n^2) \end{cases}$$

Portanto, $T(n) = \Theta(n^3)$

Pelo Teorema Mestre
$a = 8$
$b = 2$
$\log_b a = 3$

- Para melhorar a complexidade precisamos de um D&C mais "esperto", reduzindo o número de multiplicações matriciais recursivas.

- Algoritmo de Strassen:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

Calcule:

Somente 7 multiplicações de matrizes $\frac{n}{2} \times \frac{n}{2}$, calculadas recursivamente

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a+b) \cdot h \\ P_3 &= (c+d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a+d) \cdot (e+h) \\ P_6 &= (b-d) \cdot (g+h) \\ P_7 &= (a-c) \cdot (e+f) \end{aligned}$$

E depois, calcule:

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

Isso funciona? Surpreendentemente, SIM!

Veja, por exemplo:

$$r = \overbrace{(a+d)(e+h)}^{P_5} + \overbrace{d(g-e)}^{P_4} - \overbrace{(a+b)h}^{P_2} + \overbrace{(b-d)(g+h)}^{P_6}$$

$$= \cancel{ae} + \cancel{ah} + \cancel{de} + \cancel{dh} + \cancel{dg} - \cancel{de} - \cancel{ah} - \cancel{bh} + bg + \cancel{bh} - \cancel{dg} - \cancel{dh}$$

$$= ae + bg$$

Mesmo para s, t e u.

Agora o tempo é $\begin{cases} T(1) = O(1) \\ T(n) = 7T\left(\frac{n}{2}\right) + O(n^2) \end{cases}$

$$T(n) = \Theta(n^{\log_2 7})$$

$$T(n) = O(n^{2,81})$$

Comportamento assintótico. Vale a pena para matrizes grandes! Recursão não precisa descer até os escalares!

Em 1990, Coppersmith-Vinograd: $O(n^{2.376})$

Em 2014, François Le Gall: $O(n^{2.373})$ arXiv:1401.7714

⋮

— Transformada de Fourier Rápida (FFT)

Seja $\vec{x} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}$ um vetor em $\mathbb{C}^N$, assuma $N = 2^n$

Dizemos que o vetor $\vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$ dado por

$$y_k = \sum_{j=0}^{n-1} x_j e^{\frac{2\pi i}{N} kj}$$

é a Transformada de Fourier Discreta de $\vec{x}$.

- Note que $\vec{y} = F\vec{x}$, em que $F_{jk} = e^{\frac{2\pi i}{N} jk}$

- Então, temos um produto de matriz $N \times N$ por um vetor $N \times 1$. Pelo método "ingênuo", a complexidade é $O(N^2)$.

- Podemos melhorar, usando o algoritmo FFT de Cooley & Tukey (1965), um dos mais importantes da história!

- Fundamenta-se em um resultado anterior de Danielson & Lanczos (1942), que por sua vez inspira-se em resultados anteriores de Runge (1903, 1905). Até Gauss trabalhou em problema similar, ainda que em casos particulares.

Teorema (de Danielson & Lanczos): Sejam $\vec{x}$ e $\vec{y}$ vetores em $\mathbb{C}^N$, t.q. $\vec{y} = F\vec{x}$. Então

$$y_k = y_k^{PAR} + e^{\frac{2\pi i}{N} k}\, y^{ÍMPAR},$$

em que

$$y_k^{PAR} = \sum_{j=0}^{\frac{n}{2}-1} x_{2j}\, e^{\frac{2\pi i}{N} kj}$$

$$y_k^{ÍMPAR} = \sum_{j=0}^{\frac{n}{2}-1} x_{2j+1}\, e^{\frac{2\pi i}{N} kj}$$

Essa é a base para construir um algoritmo D&C para Transf. de Fourier:

- Dividir vetor $\vec{x}$ entre índices pares e ímpares, $O(n)$
- Resolver cada metade recursivamente
- Combinar resultados parciais usando fórmula de Danielson & Lanczos

A complexidade então fica

$$\begin{cases} T(1) = O(1) \\ T(n) = 2T\left(\frac{n}{2}\right) + O(n) \end{cases}$$

Logo,

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

- Muitas aplicações!
Uma aplicação muito interessante é multiplicação de polinômios, que pode ser feita de modo mais eficiente por meio da FFT! (Fica como "exercício" para os curiosos.)

MAIS UM EXEMPLO DE PROGRAMAÇÃO DINÂMICA

— Qual a ordem ótima para avaliar o produto de n matrizes?

$$M = M_1 \times M_2 \times \dots \times M_j \times \dots \times M_n$$

dimensões: $[r_0 \times r_1]$ $[r_1 \times r_2]$ $[r_{j-1} \times r_j]$ $[r_{n-1} \times r_n]$

Exemplo: $M = M_1 \times M_2 \times M_3 \times M_4$

$[10 \times 20]$ $[20 \times 50]$ $[50 \times 1]$ $[1 \times 100]$

Se fizermos $M_1 \times (M_2 \times (M_3 \times M_4))$ temos 125.000 operações
Se fizermos $(M_1 \times (M_2 \times M_3)) \times M_4$ temos 2.200 operações

*Assumindo, por simplicidade, que o produto de uma matriz $n \times p$ por uma matriz $p \times r$ gasta npr operações

- Algoritmo "ingênuo" de força bruta: testar todas as possibilidades de parênteses: tempo $O(2^n)$

Argumento: $P(n)$: número de formas que podemos pôr os parênteses em n matrizes

$$P(n) = \begin{cases} 1, & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{se } n \geq 2 \end{cases}$$

- Vejamos a estrutura da solução ótima:

Seja m_{ij} o custo mínimo para computar $M_i \times \dots \times M_j$ para $1 \leq i \leq j \leq n$.

Então,

$$m_{ij} = \begin{cases} 0, & \text{se } i = j \\ \min\limits_{i \leq k < j} \{m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j\}, & \text{se } j > i \end{cases}$$

Argumento:

$$\underbrace{(M_i * M_{i+1} * \dots * M_k)}_{\substack{\text{custo mínimo: } m_{ik} \\ \text{matriz resultante: } r_{i-1} \times r_k}} \times \underbrace{(M_{k+1} * \dots * M_j)}_{\substack{\text{custo mínimo: } m_{k+1,j} \\ \text{matriz resultante: } r_k \times r_j}}$$

- Método de programação dinâmica calcula os m_{ij} em ordem crescente da diferença dos subíndices
  - Primeiro m_{ii} para todo i
  - Depois $m_{i,i+1}$
  - Depois $m_{i,i+2}$

  e assim por diante

Exemplo: $r = 10, 20, 50, 1, 100$

Primeiro, $m_{11}=0 \quad m_{22}=0 \quad m_{33}=0 \quad m_{44}=0$

Depois, $m_{12}=10\,000 \quad m_{23}=1000 \quad m_{34}=5000$

Depois, $m_{13}=1200 \quad m_{24}=3000$

Finalmente, $m_{14}=2200$ é o custo mínimo para multiplicar todas as matrizes.

Algoritmo (como em Cormen et al.)

dados: vetor $r[1..n]$

para $i := 1, \ldots, n$ efetuar

$m[1,1] := 0$

para $l := 2, \ldots, n$ efetuar → comprimento da cadeia

para $i := 1, \ldots, n-l+1$ efetuar → quando $l=2 \Rightarrow i = 1..n-1$, $l=n \Rightarrow i=1..1$

$j := i+l-1$ → quando $i=1 \Rightarrow j=2..n$, $i=n-1 \Rightarrow j=n$

$m[i,j] := \infty$

(i,j) índice qual entrada da matriz será preenchida

para $k := 1, \ldots, j-1$ efetuar

$q := m[i,k] + m[k+1, j] + r[i-1] \cdot r[k] \cdot r[j]$

se $q < m[i,j]$ então

$m[i,j] := q$

$s[i,j] := k$

retornar m, s

Exemplo:

$n = 4$

$r = (10, 20, 50, 1, 100)$

i \ j	1	2	3	4	m_i
1	0	10.000 (1)	1.200 (1)	2.200	3
2		0	1.000 (2)	3.000	3
3			0	5000	3
4				0	

$s = 5$

$$((M_1) \times (M_2 \times M_3)) \times (M_4)$$