Divisão e Conquista

Pedro Henrique Del Bianco Hokama
14 de Agosto de 2019

Referências:

- Notas de aulas fortemente baseadas no curso: Stanford Algorithms by Tim Roughgarden
  - `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  - Vídeos: 3.1 até 3.5
- CLRS: Cap 4
- DASGUPTA, PAPADIMITRIOU, VAZIRANI: Cap 2

1 Introdução

O paradigma de divisão e conquista tem três passos:

1. **Dividir** o problema em subproblemas menores.
2. **Conquistar** os subproblemas recursivamente.
3. **Combinar** a solução dos subproblema para encontrar uma solução para o problema original.

2 Contando Inversões

Iremos estender o MergeSort para resolver o problema de contar o número de inversões de um arranjo. Formalmente:

Definição 2.1: Problema do Número de Inversões:

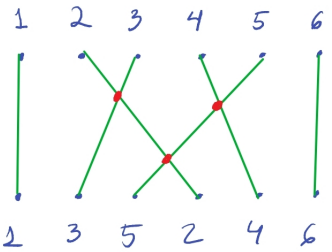
Entrada: Um arranjo A contendo n inteiros em uma ordem arbitrária.

Saída: O número de inversões, ou seja, o número de pares (i, j) de índices $1 < i, j < n$ tais que $i < j$ e $A[i] > A[j]$.

Exemplo 2.1: (1, 3, 5, 2, 4, 6)

Inversões:

- os elementos 3 e 2, então os índices $(2, 4)$ formam uma inversão
- os elementos 5 e 2, então os índices $(3, 4)$ formam uma inversão
- os elementos 5 e 4, então os índices $(3, 5)$ formam uma inversão



Motivação: Uma medida de similaridade numérica entre duas listas ranqueadas. Por exemplo: Filtragem colaborativa.

Exercício 2.1: Qual o número máximo de inversões em um arranjo?

Exercício 2.2: Como seria uma solução força-bruta para esse problema?

2.1 Uma algoritmo de divisão e conquista

É possível criar um algoritmo recursivo, baseado no paradigma de divisão e conquista para resolver o Problema do Número de Inversões. Primeiramente iremos categorizar as inversões em 3 tipos:

- **Esquerda:** se $i, j \leq n/2$
- **Direita:** se $i, j > n/2$
- **Split:** se $i \leq n/2 < j$

Exercício 2.3: Quantas inversões de cada tipo tem o arranjo do exemplo 2.1?

Note que qualquer inversão obrigatoriamente é de um desses 3 tipos. A ideia então é contar recursivamente o número de inversões na primeira metade do arranjo, depois na segunda metade (a definição do subproblema fica levemente diferente) e por fim contar o número de inversões split.

Algoritmo 1: Count

Entrada: Um arranjo A, o comprimento do arranjo n
Saída: O número de inversões
início
  se $n \leq 1$ **então** devolva 0;
  senão
    x = Count(Primeira metade de A, $n/2$);
    y = Count(Segunda metade de A, $n/2$);
    z = ContaSplit(A,n);
  fim
  devolva $x + y + n$;
fim

Note que se fizermos o ContaSplit em tempo linear, assim como o Merge do MergeSort é linear, o algoritmo terá o tempo de execução assintoticamente igual ao MergeSort $O(n \log n)$. Note também que o número total de inversões split pode ser quadrático.

- Fortalecer o Count para contar e ordenar, a ideia é aproveitar o MergeSort.
- Na função Merge será fácil calcular o número de inversões split.

Algoritmo 2: SortCount

Entrada: Um arranjo A, o comprimento do arranjo n
Saída: Um arranjo com os mesmos elementos de A porém ordenados, O número de inversões
início
  se $n \leq 1$ **então** devolva 0;
  senão
    (B, x) = SortCount(Primeira metade de A, $n/2$);
    (C, y) = SortCount(Segunda metade de A, $n/2$);
    (D, z) = MergeCountSplit(B, C, n);
  fim
  devolva $(D, x + y + n)$;
fim

- Objetivo: implementar o MergeCountSplit em tempo linear
- Lembrando o Pseudo Código do Merge:

Algoritmo 3: Merge

Entrada: B e C arranjos ordenados com $n/2$

Saída: Arranjo D de tamanho n com os mesmos elementos de B e C mas ordenados

```
início
    i = 1;
    j = 1;
    para k de 1 até n faça
        se B[i] < C[j] então
            D[k] = B[i];
            i++;
        senão
            D[k] = C[i];
            j++;
        fim
    fim
fim
//verificar finalizações (se B ou C acabarem etc).
```

- No Merge os vetores B e C já são recebidos ordenados, então os menores elementos de cada arranjo estarão em $B[i]$ e $C[j]$.
- Podemos modificar o Merge para contar inversões splits?
- Qual a relação dos itens de B e C se **não houver** inversões do tipo split? Nesse caso certamente os elementos de B são todos menores que os elementos de C. Note que (pela **contra positiva** se houver um elemento de B maior que um elemento de C haveria uma inversão split.
- Nesse caso B seria copiado inteiramente antes de C. A ideia então é que quando eu copio um elemento de C (antes de B) acabar, existe uma relação com o número de inversões split.

Exercício 2.4: Simule o Merge de $B = (1, 3, 5)$ e $C = (2, 4, 6)$.

Teorema 2.1: O número de inversões split envolvendo um elemento y do segundo arranjo C é o número de elementos restantes no primeiro arranjo B quando y é copiado para o arranjo saída D.

Prova: Seja x um elemento do primeiro arranjo B.

1. Se x foi copiado antes de y, então $x < y$, e nenhuma inversão existe.
2. Se y foi copiado antes de x, então $y < x$ mas a posição de y é maior que a posição de x então existe uma inversão split entre x e y.

■

Exercício 2.5: Modificar o Merge para contar as inversões.

Exercício 2.6: Calcular a complexidade total de SortCount.

3 Multiplicação de Matrizes

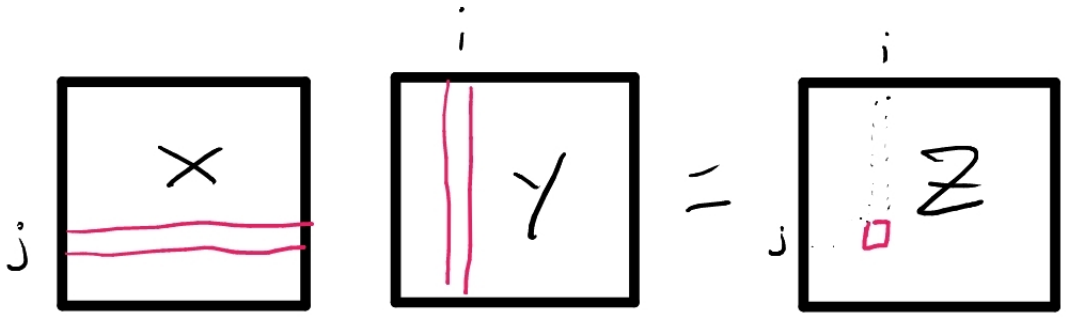
Iremos ver o algoritmo de Strassen para multiplicação de matrizes. O estudo desse problema é motivado pela grande aplicação em diversas áreas:

- Computação Gráfica
- Machine Learning
- Biologia Computacional
- Algoritmos Matemáticos e Físicos.
- e muitas outras.

- Veremos o algoritmo aplicado a **matrizes quadradas**, porém pode ser estendido para outros casos.

Sejam X e Y duas matrizes quadrada $n \times n$, desejamos obter a matriz $Z = X.Y$ também $n \times n$, tal que

$$Z_{ij} = \sum_{k=1}^{n} X_{ik}.Y_{kj}$$



Note que o tamanho da entrada é de fato n^2, mas iremos trabalhar usando o n. Exemplo de uma matriz $n = 2$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Exercício 3.1: Qual o tempo de execução de um algoritmo direto? Resp: $\Theta(n^3)$

3.1 Aplicando o paradigma da divisão e conquista

- Identificar os passos da divisão e conquista: **dividir**, **conquistar** e **combinar**
- Dividir cada matriz em quadrantes:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \text{ e } Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

De fato, a multiplicação de matrizes nesse caso se comporta como se multiplicássemos elementos isolados.

$$Z = X.Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Podemos então criar um algoritmo recursivo que computa esses 8 subproblemas e cada uma com dimensão $n/2$, e depois combinar com as somas, que podem ser feitas em tempo $O(n^2)$. Porém como veremos futuramente com o Teorema Mestre, esse algoritmo é $O(n^3)$.

- Lembre-se que no algoritmo de Karatsuba usamos a ideia de Gauss para fazer 3 chamadas recursivas e não 4. A ideia aqui é tentar diminuir o numero de chamadas recursivas.

3.2 Algoritmo de Strassen (1969)

A ideia é que façamos 7 chamadas recursivas ao invés de 8 escolhendo sabiamente. E depois fazer uma série de somas e subtrações para obter $X.Y$ mas ainda em tempo $O(n^2)$. Esse algoritmo vai ter um tempo de execução menor que $O(n^3)$.

Os 7 produtos a serem computados são:

1. $P_1 = A(F - H)$,
2. $P_2 = (A + B)H$,
3. $P_3 = (C + D)E$,
4. $P_4 = D(G - E)$,
5. $P_5 = (A + D)(E + H)$,
6. $P_6 = (B - D)(G + H)$ e
7. $P_7 = (A - C)(E + F)$.

$$X.Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}$$

Como veremos futuramente o tempo de execução desse algoritmo é subcubico!