DCC206 – Algoritmos 1

Aula 13 – Dividir e Conquistar – Parte 2

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- Na aula anterior, estudamos a forma geral dos algoritmos que seguem o paradigma de “dividir-e-conquistar”
- Vimos que a lógica central é dividir um problema maior em subproblemas (mais fáceis de resolver) e combinar essas soluções parciais para obter a solução final
- Esse comportamento nos levou a modelar o tempo de execução dos algoritmos como uma relação de recorrência
- Então, vimos três métodos para resolver relações de recorrência e deduzir a complexidade de tempo desses algoritmos
- Hoje, veremos como o paradigma é usado na construção de algoritmos para resolver alguns problemas práticos e usaremos as técnicas vistas para deduzir a complexidade de tempo

Contagem de inversões

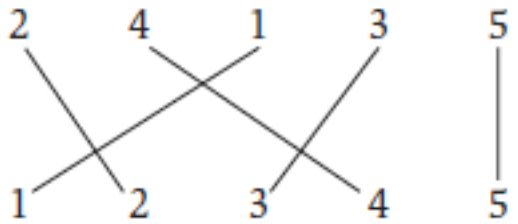
- Vamos iniciar revisitando uma das primeiras soluções baseadas em dividir-e-conquistar que estudamos: o algoritmo de mergesort
- Antes, contudo, vamos contextualizar nossa aplicação
- É muito comum hoje em dia recebermos recomendações dos diferentes serviços que usamos
- Recebemos recomendações de canais/pessoas a seguir em redes sociais, artistas/músicas para ouvir em plataformas de áudio, filmes ...
- Um dos pilares desses sistemas de recomendação é a identificação de usuários similares a nós

Contagem de inversões

- Existem diferentes formas de se avaliar a similaridade de usuários
- Como estamos interessados em fazer recomendações, existem algumas que levam em consideração as preferências dos usuários para calcular a similaridade
- Se tivermos uma lista de n filmes ranqueados pela preferência do usuário A e quisermos calcular a similaridade dele com o usuário B, podemos calcular a quantidade de inversões no ranqueamento de A para o de B

Contagem de inversões

- Por exemplo, se o usuário A ranqueou os filmes a, b, c, d, e na ordem b, a, c, e, d, e o usuário B ranqueou na ordem a, e, b, c, d, queremos saber em quantas vezes houve troca nas preferências entre A e B
- De forma genérica, se assumirmos que a ordem natural dos filmes é b=1, a=2, c=3, e=4, d=5, queremos calcular quantas vezes B quebrou essa ordem



Contagem de inversões

- Quanto menos inversões houver, mais similares são os ranqueamentos dos usuários, ou seja, mais similares eles são.
- Nosso objetivo é criar um algoritmo que calcule esse número
- Se assumirmos que o ranqueamento de B é dado por $b_1, b_2, \ldots, b_n$, queremos saber quantas vezes $b_i > b_j$ para $i < j$
- A forma ingênua de resolver o problema é verificar para todo para (b_i, b_j) se $b_i > b_j$
  - Como existem $\binom{n}{2}$ pares, temos um algoritmo quadrático para resolver o problema

Contagem de inversões

- Note que, em um caso extremo, B apresentaria um ranking oposto ao de A (ordem decrescente), o que resultaria em um número quadrático de inversões
- Assim, será que podemos reduzir o custo do algoritmo ingênuo?
- A resposta é sim!
- Veremos uma solução baseada em dividir-e-conquistar com custo O(n log n)
  - Essa solução não pode olhar todos os pares para calcular o número de inversões, já que temos um número quadrático no pior caso!

Contagem de inversões

- O problema se assemelha muito ao de ordenação, já que queremos contar o número de trocas necessárias para que a lista de B seja ordenada como a de A
- Logo, podemos tentar reaproveitar o mergesort (solução dividir-e-conquistar) para resolver esse problema
- Para entender a lógica, suponha que dividimos a lista (de B) em duas partes de tamanho n/2
- Se quisermos calcular o total de inversões, precisamos calcular o número de inversões em cada metade e, depois, contar o número de inversões entre a primeira e segunda metade

Contagem de inversões

- Contar o número de inversões entre a primeira e segunda metade significa avaliar os pares (b_i,b_j) tais que b_i está na primeira metade e b_j na segunda
- Então, se cada metade estiver ordenada, podemos inferir o número de inversões de forma indireta
- Podemos construir uma lista ordenada a partir das duas metades ordenadas e contar o número de inversões
- Usamos dois ponteiros como no mergesort, um para a primeira metade, a_i, e outro para segunda metade, b_j
- Se $a_i \leq b_j$, a_i é inserido na lista ordenada e não há inversões nesse caso (a_i é menor que todos os elementos restantes da segunda metade), a_i é movido para a próxima posição
- Se $a_i > b_j$, b_j deve ser inserido na lista ordenada, assim, sabemos que b_j está invertido em relação a todos os elementos da primeira metade maiores ou iguais a a_i, acrescentamos esse número na soma e atualizamos o ponteiro

Contagem de inversões

```
Sort-and-Count(L)
  If the list has one element then
      there are no inversions
  Else
      Divide the list into two halves:
         A contains the first ⌈n/2⌉ elements
         B contains the remaining ⌊n/2⌋ elements
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Endif
   Return r = r_A + r_B + r, and the sorted list L
```

```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If b_j is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile

  Once one list is empty, append the remainder of the other list
      to the output
  Return Count and the merged list
```

Contagem de inversões

- Assim como no mergesort, o algoritmo principal divide as listas em duas
- O custo de contar inversões numa lista com um único elemento é constante
- O custo de combinar o resultado das duas contagens é linear
  - Cada elemento das duas metades é processado uma única vez
- Logo, o custo do algoritmo é T(n) = 2T(n/2) + bn
- f(n) = bn, a=2, b=2, $\mathrm{n}^{\log_2 2} = n$
- f(n) = O(n), então, pelo teorema mestre, o custo total é Θ(n log n)

Multiplicação de grandes inteiros

- Uma outra aplicação do paradigma de dividir-e-conquistar é na construção de algoritmos eficientes para multiplicação de grandes inteiros
- Esses algoritmos são essenciais para construção de algoritmos de criptografia moderna
- Esse problema é frequentemente subestimado
  - Sempre consideramos que o custo de operações aritméticas é constante
- O que acontece se quisermos multiplicar dois números com n dígitos (bits)?

Multiplicação de grandes inteiros

- O algoritmo de multiplicação de números que aprendemos na escola tem custo quadrático
  - Multiplicamos um número por cada dígito do outro e depois somamos os resultados
- Vamos tentar desenvolver um algoritmo baseado em dividir-e-conquistar para resolver esse problema
- Suponha que queiramos multiplicar dois número de dois dígitos (o caso binário é similar): 23 x 14
- Podemos dividi-los em duas metades e tentar efetuar as multiplicações
  - $23 \times 14 = (2 \times 10^1 + 3 \times 10^0) \times (1 \times 10^1 + 4 \times 10^0) = (2 \times 1 \times 10^2) + (2 \times 4 \times 10^1) + (3 \times 1 \times 10^1) + (3 \times 4 \times 10^0)$

Multiplicação de grandes inteiros

- Nesse caso, temos que realizar 4 multiplicações e 4 somas
- As multiplicações por 10 são shifts à esquerda na representação e são facilmente resolvidas
- Podemos estender o raciocínio para n dígitos e, assim, temos que o custo do algoritmo é
  - $T(n) = 4T(n/2) + bn$
  - Pelo teorema mestre, $T(n) = \Theta(n^2)$
- Então, a abordagem dividir-e-conquistar não é melhor que aquela aprendida na escola

Algoritmo de Karatsuba

- Em 1960, A. Karatsuba, um matemático russo, provou pela primeira vez que era possível multiplicar dois números com n dígitos com custo menor que quadrático
- Vamos voltar à multiplicação novamente
  - $23 \times 14 = (2 \times 1 \times 10^2) + (2 \times 4 \times 10^1) + (3 \times 1 \times 10^1) + (3 \times 4 \times 10^0) = (2 \times 1)10^2 + (2 \times 4 + 3 \times 1)10^1 + (3 \times 4)10^0$
- Karatsuba observou que o termo central poderia ser computado de outra forma
  - $2 \times 4 + 3 \times 1 = (2+3) \times (1+4) - 2 \times 1 - 3 \times 4$
- Ou seja, $23 \times 14 = (2 \times 1)10^2 + ((2+3) \times (1+4) - 2 \times 1 - 3 \times 4)10^1 + (3 \times 4)10^0$.
  - Como 2×1 e 3×4 devem ser computados de qualquer forma, reduzirmos o número de multiplicações de 2 para 1

Algoritmo de Karatsuba

- De uma forma geral, se quisermos multiplicar dois números a e b com n dígitos podemos dividi-los em duas metades
  - $a = a_1 10^{n/2} + a_0$
  - $b = b_1 10^{n/2} + b_0$
- A multiplicação de a por b é definida por
  - $c = a \times b = c_2 10^{n} + c_1 10^{n/2} + c_0$
  - $c_2 = (a_1 \times b_1)$
  - $c_0 = (a_0 \times b_0)$
  - $c_1 = (a_1+a_0) \times (b_1+b_0) - (c_2+c_0)$
- Nesse caso, temos que realizar 4 somas/subtrações de n dígitos, mas apenas 3 multiplicações de n/2 dígitos
- Logo, $T(n) = 3T(n/2) + bn$
  - $n^{\log_2 3} \approx n^{1.585}$
  - Pelo teorema mestre, $T(n) = \Theta(n^{1.585})$
- Do ponto de vista prático, podemos aplicar o algoritmo até que o número de dígitos/bits seja pequeno o suficiente para ser computado pelo algoritmo tradicional
  - Python usa limiar 70 dígitos (2025): https://github.com/python/cpython/blob/main/Objects/longobject.c

Multiplicação de matrizes

- Outro problema de grande interesse para desenvolvimento de algoritmos eficientes é a multiplicação de matrizes quadradas
- O avanço dos métodos de aprendizado profundo (redes neurais) requer a computação eficiente de multiplicações de matrizes
  - O funcionamento desses métodos de IA é fundamentado em álgebra linear (multiplicação de matrizes)
- O algoritmo tradicional para multiplicação de matrizes tem custo $O(n^3)$
  - Z = XY, em que X e Y são matrizes quadradas de ordem n
  - $Z[i,j] = \sum_{k=0}^{n-1} X[i,k] * Y[k,j]$

Multiplicação de matrizes

- Podemos tentar desenvolver uma metodologia baseada em dividir-e-conquistar, dividindo a matriz em quatro partes (blocos)
  - $\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$
  - I = AE + BG
  - J = AF + BH
  - K = CE + DG
  - L = CF + DH
- Nosso algoritmo então precisa computar 8 multiplicações de matrizes n/2 e 4 somas (com custo n^2)
- Ou seja, o custo do algoritmo é $T(n) = 8T(n/2) + bn^2$
  - Pelo teorema mestre, $T(n) = O(n^3)$

Algoritmo de Strassen

- Claramente, a abordagem dividir-e-conquistar ingênua não é melhor que o algoritmo tradicional
- No entanto, em 1969, o matemático alemão V. Strassen estudou propriedades aritméticas/algébricas para reduzir o número de multiplicações de submatrizes para computar I, J, K e L usando apenas 7 multiplicações
  - S1 = A(F-H)
  - S2 = (A+B)H
  - S3 = (C+D)E
  - S4 = D(G-E)
  - S5 = (A+D)(E+H)
  - S6 = (B-D)(G+H)
  - S7 = (A-C)(E+F)

Algoritmo de Strassen

- Usando as 7 submatrizes anteriores, ele definiu as submatrizes I, J, K e L
- I = S5 + S6 + S4 – S2
  - I = (A+D)(E+H) + (B-D)(G+H) + D(G-E) - (A+B)H = AE + BG
- J = S1+S2
  - J = A(F-H) + (A+B)H = AF + BH
- K = S3 + S4
  - K = (C+D)E + D(G-E) = CE + DG
- L = S1 – S7 – S3 + S5
  - L = A(F-H) - (A-C)(E+F) - (C+D)E + (A+D)(E+H) = CF + DH

Algoritmo de Strassen

- Assim, o custo do algoritmo será $T(n) = 7T(n/2) + bn^2$
- Pelo teorema mestre, como $n^{2.8} < n^{\log_2 7} < n^{2.81}$, $T(n) = \Theta(n^{\log_2 7})$
  - Ou seja, $T(n) = O(n^{2.81})$
- Existem outros algoritmos ainda mais sofisticados e eficientes para computar a multiplicação de matrizes
- Recentemente, a equipe do DeepMind da Google usou o algoritmo AlphaZero para encontrar um algoritmo mais eficiente para multiplicação de matrizes
  - Mostraram o uso de IA para encontrar algoritmos ainda mais eficientes

Leitura

- Seção 5.3 Kleinberg e Tardos
- Seção 11.2 e 11.3 Goodrich e Tamassia