ESCOLA POLITÉCNICA PUCRS

Disciplina: Projeto e Otimização de Algoritmos

Professor: Rafael Scopel

Trabalho T1

- Este trabalho consiste no desenho, análise e implementação de algoritmos greedy e de divisão e conquista.
- O trabalho pode ser realizado grupos de até 5 alunos.
- O trabalho deve ser implementado em Java.

Objetivos do trabalho:

Problema 1

Alguns de seus amigos entraram no crescente campo da mineração de dados de séries temporais, onde procuramos padrões em sequências de eventos que ocorrem ao longo do tempo. As ordens de compra nas bolsas de valores são uma fonte de dados com uma ordenação natural no tempo. Dada uma longa sequência S de tais eventos, seus amigos querem uma maneira eficiente de detectar certos "padrões" neles – por exemplo, eles podem querer saber se os quatro eventos

S = **{"buy Google", "buy Apple", "buy Google", "buy NVIDIA"}**

ocorrem nesta sequência S, em ordem, mas não necessariamente consecutivamente.

Eles começam com uma coleção de eventos possíveis (por exemplo, as transações possíveis) e uma sequência S de n desses eventos. Um determinado evento pode ocorrer múltiplas vezes em S (por exemplo, as ações do Google podem ser compradas muitas vezes em uma única sequência S). Diremos que uma sequência S' é uma subsequência de S se houver uma maneira de deletar certos eventos de S de modo que os eventos restantes, em ordem, sejam iguais à sequência S'. Assim, por exemplo, a sequência de quatro eventos acima é uma subsequência da sequência

S = **{"buy Amazon", "buy Google", "buy Apple", "buy Google", "buy Google", "buy NVIDIA"}**

S' = **{"buy Google", "buy Apple", "buy Google", "buy NVIDIA"}**

O objetivo deles é ser capaz de imaginar sequências curtas e detectar rapidamente se são subsequências de S. Portanto, forneça um algoritmo que receba duas sequências de eventos S' de comprimento m e S de comprimento n, cada uma possivelmente contendo um evento mais de uma vez, e decide se S' é uma subsequência de S.

Estruture a resposta no formato de um relatório com as seguintes sessões:

1. O Problema;
2. O Algoritmo;
3. Análise do Algoritmo;
4. Implementação e Tempo de Execução. O método implementado retorna verdadeiro se S_line (S') é uma subsequência de S e deve respeitar a seguinte assinatura:

boolean hasTrend(String[] S, String[] S_line)

Sugestão de Resposta:

Deixe a sequência S consistir em $s_1, \ldots, s_n$ e a sequência S'consistir em $s'_1, \ldots, s'_m$. Fornecemos um algoritmo ganancioso que encontra o primeiro evento em S que é igual a s'_1, armazena a posição k_j na sequência S onde esses dois eventos são iguais (essa é a decisão Greedy!!!) e, em seguida, encontra o primeiro evento depois deste que é igual a s'_1, e assim por diante. Usaremos $k_1, k_2, \ldots$ para denotar a correspondência que encontramos até agora, i para denotar a posição atual em S e j a posição atual em S'.

hasTrend(S, S')

$inicialize\ i = j = 1$

While $(i \leq n\ \ AND\ j \leq m)$
  If (s_i é igual a s'_j)
    $k_j = i$
    $j = j+1$
  $i = i + 1$

If $(j == m + 1)$
  Return true
else
  Retur false

O tempo de execução é $O(n)$: uma iteração através do loop $while$ leva $O(1)$, e cada iteração incrementa i, então pode haver no máximo n iterações.

Também está claro que o algoritmo encontra uma correspondência correta se alguma existir (lembre-se nem sempre S' será uma subsequência de S). É mais difícil mostrar que se o algoritmo não conseguir encontrar uma correspondência, então não existe correspondência. Por isso, utilizaremos a abordagem abaixo:

Suponha que S' seja igual à subsequência $s_{l_1}, \ldots, s_{l_m}$ de S. Ou seja, queremos provar que se existe esta subsequência o nosso algoritmo irá encontrá-la. Assim não precisamos provar o caso em que ele não encontra a subsqüência, pois este é um resultado indireto da nossa prova de que o algoritmo sempre encontra a subsequência se ela existir!

Provamos por indução que o algoritmo terá sucesso em encontrar uma correspondência e terá $k_j \leq l_j$ para todo $j = 1, \ldots, m$. Isso é análogo à prova em que o algoritmo ganancioso encontra a solução ótima para o problema de escalonamento intervalar: provamos que o algoritmo ganancioso está sempre à frente.

Para cada $j = 1, \ldots, m$ o algoritmo encontra uma correspondência k_j e tem $k_j \leq l_j$.

Prova. A prova é por indução em j. Lembre-se que percorremos a sequência original S procurando por correspondências em S'. Primeiro considere $j = 1$. O algoritmo permite que k_1 seja o primeiro evento igual a s'_1, então devemos ter que $k_1 \leq l_1$ (lembre-se que k_1 e l_1 são índices da sequência de eventos). Ou seja, o algoritmo greedy escolheu um primeiro evento s'_1 anterior ou igual ao primeiro evento da subsequência s_{l_1}. O greedy se mantém a frente.

Agora considere um caso em que $j > 1$. Suponha que $j - 1 < m$ e assuma pela hipótese de indução que o algoritmo encontrou a correspondência k_{j-1} e tem $k_{j-1} \leq l_{j-1}$. O algoritmo permite que k_j seja o primeiro evento após k_{j-1} que é igual a s'_j se tal evento existir. Sabemos que s_{l_j} é um tal evento (pois assumimos isto

acima) e $l_j > l_{j-1} > k_{j-1}$. Então $s_{l_j} = s'_j$ e $l_j > k_{j-1}$. O algoritmo encontra o primeiro índice desse tipo, então obtemos que $k_j \leq l_j$.

```java
public static boolean hasTrend(String S[], String S_line[]) {

    int n = S.length;
    int m = S_line.length;
    int i = 0, j = 0;
    int[] k = new int[m];

    while (i < n && j < m ) {

        if(S[i].equals(S_line[j])) {

            k[j] = i;
            j++;
        }
        i++;
    }

    if(j==m) {

        System.out.println("\nSequence indexes: " + Arrays.toString(k) + "\n");
        return true;
    }

    return false;
}
```

Problema 2

Até 1969 matemáticos acreditavam sem impossível resolver um problema de multiplicação de matrizes em tempo menor que $\Theta(n^3)$. Procure pelos trabalhos de V. Strassen e responda se isso ainda é verdade.

Estruture a resposta no formato de um relatório com as seguintes sessões:

1. O Problema;
2. O Algoritmo;
3. Análise do Algoritmo;
5. Implementação e Tempo de Execução. O método implementado retorna a matriz resultante das matrizes A e B passadas por parâmetro e deve respeitar a seguinte assinatura:

int[][] multiply(int[][] A, int[][] B)

Entregáveis do Trabalho:

1. Relatório em Word com:
    a. Capa com título e nome dos integrantes;
    b. Resolução do problema 1;
    c. Resolução do problema 2;

2. Código fonte comentado.

Critérios de Avaliação:

- Código fonte compilável sem erros;
- Qualidade e profundidade das análises do relatório;
- Qualidade e documentação do código fonte;
- **Respeite as assinaturas solicitadas no trabalho. Implementações diferentes da assinatura especificada não serão avaliadas, recebendo nota zero.**

Sugestão de Resposta:

Capítulos 4.1 e 4.2 -> CORMEN, T. H. Algoritmos: teoria e prática. 3a ed., Rio de Janeiro: Elsevier-Campus, 2012.

```
MATRIX-MULTIPLY(A, B, C, n)
1  for i = 1 to n                  // compute entries in each of n rows
2     for j = 1 to n               // compute n entries in row i
3        for k = 1 to n
4           c_ij = c_ij + a_ik · b_kj   // add in another term of equation (4.1)
```

Algoritmo original é $O(n^3)$.

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1
 2  // Base case.
 3      c11 = c11 + a11 · b11
 4      return
 5  // Divide.
 6  partition A, B, and C into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22;
        and C11, C12, C21, C22; respectively
 7  // Conquer.
 8  MATRIX-MULTIPLY-RECURSIVE(A11, B11, C11, n/2)
 9  MATRIX-MULTIPLY-RECURSIVE(A11, B12, C12, n/2)
10  MATRIX-MULTIPLY-RECURSIVE(A21, B11, C21, n/2)
11  MATRIX-MULTIPLY-RECURSIVE(A21, B12, C22, n/2)
12  MATRIX-MULTIPLY-RECURSIVE(A12, B21, C11, n/2)
13  MATRIX-MULTIPLY-RECURSIVE(A12, B22, C12, n/2)
14  MATRIX-MULTIPLY-RECURSIVE(A22, B21, C21, n/2)
15  MATRIX-MULTIPLY-RECURSIVE(A22, B22, C22, n/2)
```

Algoritmo recursivo $T(n) = 8T(n/2) + \Theta(1)$ é $O(n^3)$.

Strassen's strategy for reducing the number of matrix multiplications at the expense of more matrix additions is not at all obvious—perhaps the biggest understatement in this book! As with MATRIX-MULTIPLY-RECURSIVE, Strassen's algorithm uses the divide-and-conquer method to compute $C = C + A \cdot B$, where A, B, and C are all $n \times n$ matrices and n is an exact power of 2. Strassen's algorithm computes the four submatrices C_{11}, C_{12}, C_{21}, and C_{22} of C from equations (4.5)–(4.8) on page 82 in four steps. We'll analyze costs as we go along to develop a recurrence $T(n)$ for the overall running time. Let's see how it works:

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 3 of MATRIX-MULTIPLY-RECURSIVE, taking $\Theta(1)$ time, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2). This step takes $\Theta(1)$ time by index calculation, just as in MATRIX-MULTIPLY-RECURSIVE.
2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.
3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.
4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices, which takes $\Theta(n^2)$ time.

Algoritmo de Strassen $T(n) = 7T(n/2) + \Theta(n^2)$ é $O(n^{2.81})$.