UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO

Yudi Asano Ramos – Nº USP: 12873553
Diogo Barboza de Souza – Nº USP: 12745657

Trabalho 1 – Estrutura de Dados 2

São Carlos/SP
2022

1. Introdução

A elaboração do projeto a seguir, da matéria de Estrutura de Dados II, foi feita com o uso dos assuntos dados em aula, ordenação e análise de complexidade de algoritmos. Além disso, para que o projeto se relacionasse com o mundo afora e não somente à computação, foi elaborada a aplicação a bioinformática, com a análise do sequenciamento de DNA.

Neste trabalho, foi necessário saber como fazer esta análise de complexidade, ao transformar os algoritmos dados em forma de texto, para sua forma em programação.

Dividiremos o relatório na introdução, a descrição dos problemas propostos e atividades extras propostas.

2. Descrição dos problemas propostos

2.1. Implementação da ordenação

2.1.1. Arquivos utilizados para o código

Dada as estruturas do algoritmo, para a conversão para o código, no PDF do trabalho, utilizamos o arquivo “proj_digitos.c”, que compôs todos os nossos algoritmos (OrdenaNumeros, OrdenaDigitos, etc), o arquivo “main.c” que, como o nome já propõe, será o arquivo principal, onde estará as ações que o código final fará e por último, temos o arquivo “func.h”, que conectará todas os arquivos “.c” para o “main.c”.

Além dos arquivos “.c” e “.h”, ainda temos os arquivos “.txt”, saída.txt, que de acordo com o trabalho será o output final com as respostas, os fragmentos e genomas dados que são as entradas. Finalmente, na pasta também haverá o arquivo MAKEFILE, que compilará e rodará toda a pasta com os códigos (este arquivo um executáveis “.exe”).

2.1.2. Realização dos critérios de avaliação

2. Para a primeira análise da primeira função OrdenaDigitos, a análise foi:

```
Entrada: A[n][2], vetor com pares de numeros a serem ordenados.
Entrada: n, numero de elementos em A.
Saída: A ordenado pelo primeiro elemento de cada par.

1 maior ← maior inteiro armazenado em A[i][0];
2 posicao ← 1;
3 enquanto (maior/posicao) > 0 fa¸ca
4 OrdenaDigitos (A,n,posicao);
5 posicao ← posicao * 10;
6 fim
```

Para o melhor caso foi o obtido a notação: O(n), visto que temos apenas um laço de for e um while, que não estão conectados, não haverá nenhum aumento exponencial.

Para o pior caso foi obtido a notação: O(n) novamente, pois neste caso, implementando os ifs, não haverá outro laço duplo, rodando apenas os valores do for inicial.

Para o caso médio foi obtido a notação: Devido a análise do melhor caso e do pior caso, e os mesmos serem de mesmo valor, podemos dizer que o caso médio também será este valor, O(n).

3. Implementando o código:

```
void OrdenaDigitos( int **A, int n, int posicao){


   long B[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
   int digito;


   for (int i = 0; i <= n-1;i++ ){
      digito = A[i][0]/posicao;
      digito = digito % 10;
      B[digito] = B[digito] + 1;


   }
   for (int i = 1; i<=9; i++){
      B[i] = B[i] + B[i-1];
```

```
    }


    //inicialização e alocação de memoria para C
    int **C;
    C = malloc (n * sizeof (int*));


    for (int i=0; i < n; i++){
        C[i] = malloc (2 * sizeof (int)) ;

    }
    int u = 0;


    //numero significativo
    for (int i=n-1; i >= 0; i--){
        digito = A[i][0]/posicao;
        digito = digito % 10;
        B[digito] = B[digito] - 1;
        C[B[digito]][0] = A[i][0];
        C[B[digito]][1] = A[i][1];

    }


    //ordenação apropriada de A
    for (int i = 0; i<= n-1; i++){
        A[i][0] = C[i][0];
        A[i][1] = C[i][1];

    }


    //liberação de memória
    for (int i=0; i < n; i++){
        free (C[i]) ;


    }
    free (C);


}
```

4. O código OrdenaNumeros, fará, primeiramente, uma chamada de ordenação de dígitos para cada posição dos dígitos presentes nos

números. Isso significa que se, por exemplo, encontrarmos o número 2859, ordenaremos da seguinte forma: inicialmente 2859 será tratado como 9, o digito mais à direita, e após a análise partiremos para o próximo número mais à direita, 5, e segue assim até o fim do número, desta maneira sendo feito para os demais números exigidos e por fim ordenando-os em ordem crescente.

5. Sua complexidade de acordo com o código abaixo:

```
  void OrdenaNumeros(int **A, int n){

 int posicao = 1;
 int maior = A[0][0];
 for (int i = 0; i <= n-1; i++){
     if (maior < A[i][0]){
         maior = A[i][0];
     }
 }

 while (maior/posicao > 0){
     OrdenaDigitos(A, n, posicao);
     posicao = posicao*10;
 }

}
```

Complexidade é O(n).

```
 int posicao = 1;
 int maior = A[0][0];
 for (int i = 0; i <= n-1; i++){
     if (maior < A[i][0]){
         maior = A[i][0];
     }
 }

 while (maior/posicao > 0){
     OrdenaDigitos(A, n, posicao);
     posicao = posicao*10;
 }
}
```

Complexidade é O(n).

6. A análise ContagemInterseçoes implementado em código:

```
void ContagemIntersecoes(char arq_A[], char arq_B[], int nA, int nB, char
arq_saida[]){
    FILE *arqinA;
    FILE *arqinB;
    FILE *arqoutcont;
```

```
arqinA=fopen(arq_A,"r");
arqinB=fopen(arq_B,"rt");

int **A, **B, contagens[nA];


// aloca um vetor de n* ponteiros para linhas
A = malloc (nA * sizeof (int*));
B = malloc (nB * sizeof (int*));
// aloca cada uma das linhas (vetores de 2 inteiros)
for (int i=0; i < nA; i++){
    A[i] = malloc (2 * sizeof (int)) ;

}


for (int i=0; i < nB; i++){

    B[i] = malloc (2 * sizeof (int)) ;
}


// vetor contagens com zeros
for (int i =0; i<=nA-1; i++){
    contagens[i] = 0;
}

//leitura dos arquivos e anexamento nas matrizes
for (int i = 0; i<=nA-1; i++){
    fscanf(arqinA,"%d,%d", &A[i][0], &A[i][1]);
}
for (int i = 0; i<=nB-1; i++){
    fscanf(arqinB,"%d,%d", &B[i][0], &B[i][1]);
}

//ordenação das matrizes A e B
OrdenaNumeros(A, nA);
OrdenaNumeros(B, nB);



int primeiro_iB = 0;

//verificação de onde há cada intersessão de cada intervalo
for ( int iA = 0; iA <= nA-1; iA++){
    for (int iB = primeiro_iB; iB <= nB-1; iB++){
        if(A[iA][1] < B[iB][0] || A[iA][0] > B[iB][1]){
            if(contagens[iA] == 0){
                primeiro_iB = iB;
            }
        }
        else {
            contagens[iA] = contagens[iA]+1;
        }
    }
```

```
    }


    //escrita no arquivo saida
    arqoutcont=fopen(arq_saida,"wt");
    for (int i =0; i<=nA-1; i++){
        fprintf(arqoutcont, "%d\n", contagens[i]);
    }

    //fecha arquivos usados
    fclose(arqinA);
    fclose(arqinB);
    fclose(arqoutcont);
}
```

7. A análise CtrlF implementado em código:

```
void CtrlF(char arquivo_texto[], char arquivo_trecho[], char
arquivo_saida[]){

    //preparação dos arquivos
    FILE *arq_txt;
    FILE *arq_tre;
    FILE *aux;
    FILE *arq_sai;


    arq_txt = fopen(arquivo_texto, "rt");
    arq_tre = fopen(arquivo_trecho, "rt");
    aux = fopen(arquivo_trecho, "rt");
    arq_sai = fopen(arquivo_saida, "wt");



    // descobrir o numeros de caracteres para alocar o texto
    char *texto;
    int h = 0;
    while (!feof(arq_txt)){
        char p;
        fscanf(arq_txt, "%c", &p);
        h++;
    }

    //alocação de memoria para texto
    texto = (char *)malloc((h+1) * sizeof(char));


    rewind (arq_txt);


    //leitura do arquivo e anexação dos caracteres em vetor texto
    fscanf(arq_txt, "%s", texto);
```

```
//alocação de memoria inicial para trecho
char *trecho;
trecho = (char *)malloc((100) * sizeof(char));


rewind (arq_txt);
int t = 0;
while (!feof(arq_tre)){


    // descobrir o numeros de caracteres de cada trecho
    int h = 0;
    while (!feof(aux)){
        char p;

        fscanf(aux, "%c", &p);
        if (p == '\n'){
            if( h == 0){
                t = 1;
                continue;
            }
            break;
        }
        h++;


    }


    //usar o numero descoberto para realocar memoria se necessario
    if (h>100){

        trecho = realloc(trecho, h+1);

    }


    //flag caso tenha uma linha vazia do arquivo
    if (t == 1){
        char p;
        fscanf(arq_tre, "%c", &p);
        fscanf(arq_tre, "%s", trecho);
    }


    //anexação dos caracteres no vetor, para linha normal
    else{
        fscanf(arq_tre, "%s", trecho);
    }

    //encontrar aonde está o inicio e o fim dos trechos no texto
    int i = 0;
    while(texto[i] != '\0'){
        int j = 0;
```

```
            while((trecho[j] != '\0') && (texto[i+j] == trecho[j])){
                j++;

            }
            if(trecho[j] == '\0'){
                fprintf(arq_sai,"%d,%d\n", i, (i+j-1) );


                //break;
            }
            i++;

        }
    }


    //fechamento de arquivos
    fclose(arq_txt);
    fclose(arq_tre);
    fclose(aux);
    fclose(arq_sai);


    //liberação de memoria usada nos vetores
    free (trecho);
    free (texto);
}
```

8. O programa feito para executar o procedimento de ContagemLeituras foi:

```
void ContagemLeituras (char arquivo_genoma[], char arquivo_pos_genes[],
char arquivo_fragmentos[], char arquivos_pos_fragmentos[], int n_genes,
int n_fragmentos, char arquivo_saida[]){


    //chamada das funções
    CtrlF(arquivo_genoma, arquivo_fragmentos, arquivos_pos_fragmentos);

    ContagemIntersecoes(arquivo_pos_genes, arquivos_pos_fragmentos,
      n_genes, n_fragmentos, arquivo_saida);
}
```

9. Com o auxílio do arquivo saída.txt, foi encontrado a seguinte output:

O programa contido no arquivo .zip responde à pergunta 9.

10. Os resultados encontrados na análise empírica de complexidade foram:

no (1) caso, onde estão fixados os arquivos genoma_grande.txt e pos_genes_grande.txt:

n	tempo(s)
300	2,409
3000	50,627
30000	464,738

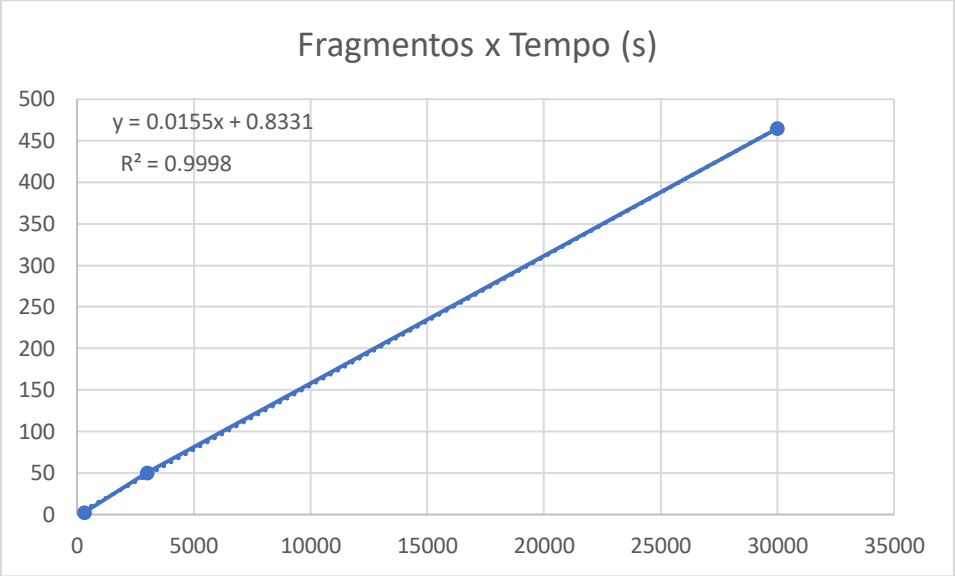
No (2) caso, onde estão fixados genoma_grande.txt e fragmentos_grande.txt:

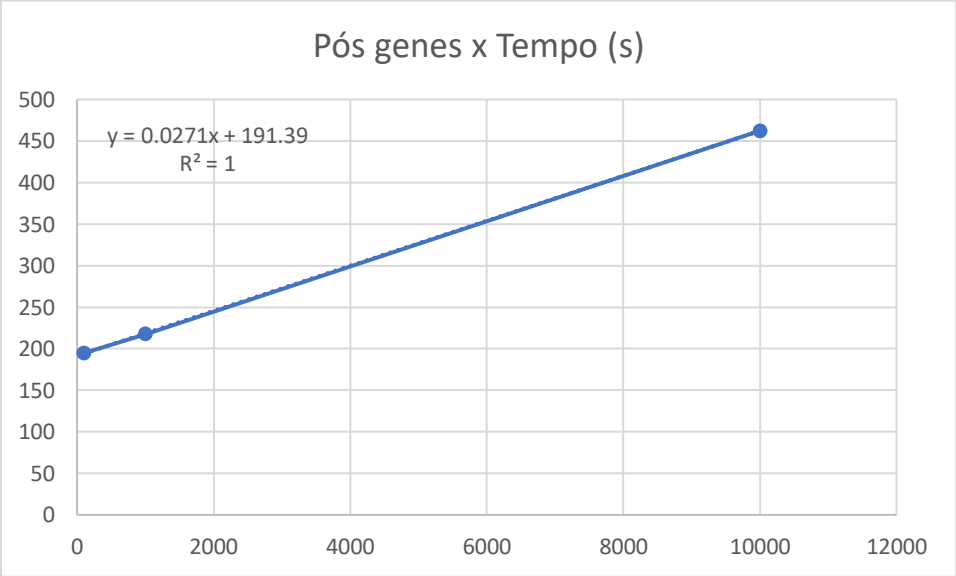
n	tempo(s)
100	194,828
1000	217,683
10000	462,373

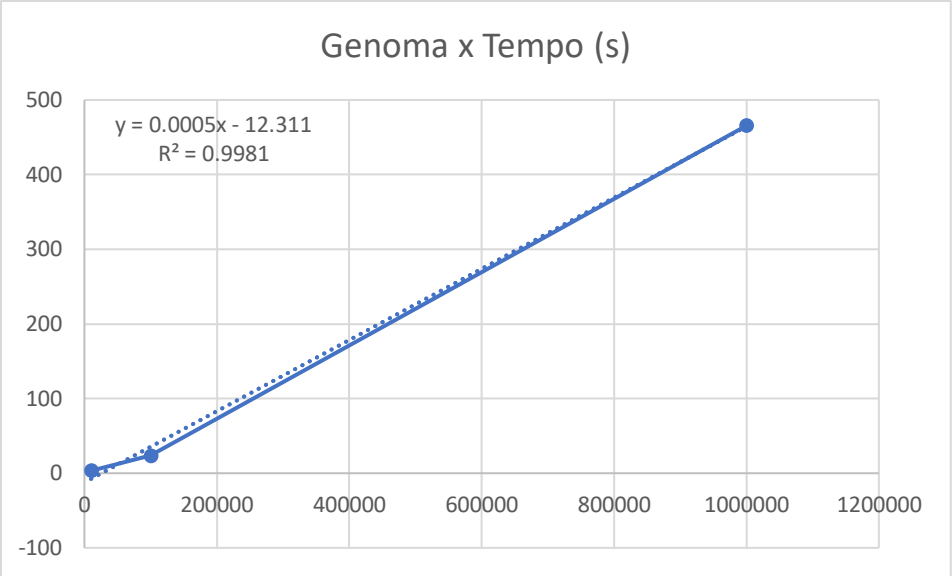
No (3) caso, onde estão fixados pos_genes_grande.csv e fragmentos_grande.txt:

n	tempo(s)
10000	3,284
100000	23,458
1000000	465,552

11. Os gráficos obtidos para o critério da questão 10 foram, respectivamente:


Fragmentos x Tempo (s)
y = 0.0155x + 0.8331
R² = 0.9998
500
450
400
350
300
250
200
150
100
50
0
0
5000
10000
15000
20000
25000
30000
35000


Pós genes x Tempo (s)
y = 0.0271x + 191.39
R² = 1
500
450
400
350
300
250
200
150
100
50
0
0
2000
4000
6000
8000
10000
12000



12. –Visto como as funções utilizadas funcionaram e os tempos calculados nas mesmas, é viável e importante considerar que essas poderiam ser desenvolvidas com um tempo de execução menor, visto que neste projeto temos já definido a quantidade dos valores dos arquivos para cada caso dado.

13. –No desenvolvimento das funções foram encontradas dificuldades ao realizar a manipulação de arquivos assim como a alocação de memória em determinados vetores. Estes problemas foram encontrados devido a dificuldade na interpretação dos códigos originais dados pelo professor, isso porque nestes não estão especificados o uso da alocação da memória, o que nos confundiu a ponto de não sabermos onde implementa-los.

14. Para a análise da contagemLeituras: (resposta: $O(n^2)$)

Como esse algoritmo conta com o uso de dois algoritmos anteriores podemos tirar em conclusão que o resultado será analisando o maior dos valores de Big O.

15. Em ContagemIntersecoes, como temos dois laços juntos, já é possível identificar de cara que sua notação é $O(n^2)$, pois temos dois laços for variáveis de acordo com n, em sua composição.

16. A função ContagemInterseçoes possui complexidade de $O(n^2)$ devido a laços duplos, assim, afim de conseguir uma complexidade de O(nA + nB), temos duas possíveis opções para a diminuição da sua notação. A primeira opção é juntar o laço duplo e tornar em apenas um laço completo, ou retirar um dos laços que integram o laço duplo e colocar de forma que não rodem ao mesmo tempo, ou seja, um while rodará antes do outro.

17. Analisando o algoritmo ctrlF, percebemos o uso dos laços while. Iniciando com o primeiro while(feof(arquivo)), vemos que sua complexidade é de 1, visto que não há condição dependende de uma variável. Entrando para o segundo while, já é possível ver que a sua complexidade será de n, pois nele possui um vetor que rodará a cada i vezes (i posições), do primeiro laço. E assim também ocorre no segundo laço, em que dado um vetor rodará j vezes (j posições), tendo uma variável n. Por conclusão, como estes estão laçando um com outros temos que a complexidade é $O(n*n*1) = O(n^2)$.

18. Finalmente, pegando a maior das notações temos que é igual a $O(n^2)$.

Fazendo a análise do melhor caso, assim propondo uma alternativa de menor complexidade, temos que: em ctrlF, o while só entrará até o primeiro while(texto[i] != (fim da string)), não entrando no segundo while, visto que o resultado desse caso, temos que o texto[i] será igual a fim da string (condição do segundo while). Então, o resultado da complexidade do ctrlF será de 1.

Para o segundo algoritmo, ContagemIntersecoes, em melhor caso, entrando só no primeiro laço, temos que sua complexidade é de n.

Analisando o algoritmo final, temos que no melhor caso a complexidade é de n.

3. Conclusão

Ao concluir o trabalho 1 de estrutura de dados, foi possível concluir que de acordo com a quantidade de números, ou no caso do conteúdo do trabalho, genes, fragmentos e genomas, aumentam, maior será o tempo de execução do código.

Com isto, percebemos como é importante, não só para a computação, mas como também para as áreas biológicas, matemáticas e até históricas em geral, o algoritmo adequado para o estudo, para que sua complexidade demonstre sua eficiência em tempo real, e não somente em teoria, onde todo código é valido.