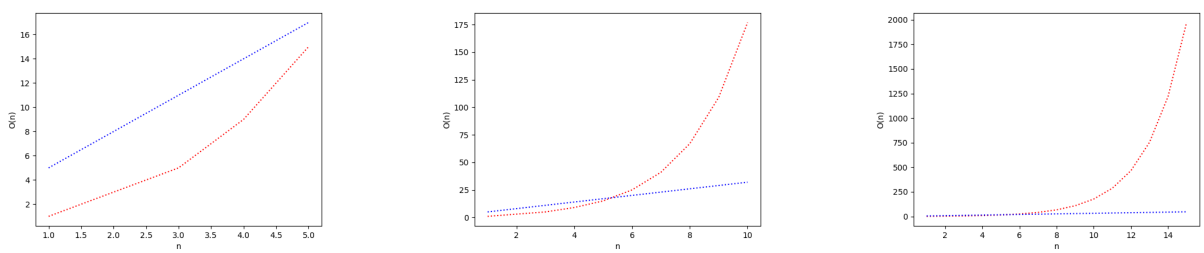
Complexidade de Algoritmos
Lista de Exercícios 1

Vinícius Takeo Friedrich Kuwaki

7 de Dezembro de 2020

1 Fibonacci

Para a realização da questão 1, eu implementei um algoritmo em C que gerava dois arquivos contendo os parâmetros para a plotagem dos gráficos utilizando a biblioteca matplotlib do Python. Na Figura 1 estão os casos gerados para entrada de tamanho n igual a 5, 10 e 15.



n = 5 | n = 10 | n = 15

Figura 1: Comparativo entre o Fibonacci Recursivo e Iterativo

O caso de n = 15 já era suficiente para, mas é interessante notar observar como o gráfico é gerado pela biblioteca quando o conjunto de dados é similar.

O algoritmo linear é apresentado a seguir:

```
unsigned int lin(unsigned int n)
{
    unsigned int fib1 = 0;
    unsigned int fib2 = 1;

    cont += 2;

    while (n != 0)
    {
        fib2 = fib1 + fib2;
        fib1 = fib2;
        n--;
        cont += 3;
    }

    return fib2;
}
```

Como o Moodle não permitiu o envio de arquivos .py, o script para a geração do código não foi enviado em conjunto.

2 Potência

Para calcularmos a relação de recorrência é necessário analisar o algoritmo:

```
unsigned int potencia (unsigned int b, unsigned int e)
{
    unsigned int r; // O(1)
    if (e == 0) // O(1)
        return 1; // O(1)
    r = potencia(b, e/2);
    if (e % 2 == 0) // O(1)
        return r*r; // O(1)
    else
        return r*r*b; // O(1)
}
```

É possível notar que, a complexidade das chamadas não recursivas, no melhor caso, isto é, quando $\mathbf{e = 0}$ é 1. No pior do casos ocorrerão chamadas recursivas recorrentes a $\mathbf{T(n/2) + 5}$.

Para resolver essa relação de recorrência, basta expandir a série:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 5 \\ T(\frac{n}{2}) &= T(\frac{n}{2^2}) + 5 \\ T(\frac{n}{2^2}) &= T(\frac{n}{2^3}) + 5 \\ &... \\ T(\frac{n}{2^{l-1}}) &= T(\frac{n}{2^l}) + 5 \end{aligned}$$

Como $T(\frac{n}{2^l})$ é a base da recursão, pode assumir que é 1, pois no melhor caso a complexidade é 1. Logo, é possível determinar que:

$$\begin{aligned} \frac{n}{2^l} &= 1 \\ l &= \log_2 n \end{aligned}$$

Como o somatório tem tamanho l, ou seja, a recorrência é em **l**, multiplica-se **l** pelo caso base:

$$\begin{aligned} T(n) &= (T(1) + 5) * \log_2 n \\ T(n) &= 6 * \log_2 n \end{aligned}$$

Logo, a recorrência é expressa por $T(n) = 6 * \log_2^n$, gerando uma complexidade de tempo de $O(log^n)$. Já a complexidade de espaço pode ser calculada, analisando a pilha de recursão e os recursos alocados nela: no pior caso, a pilha vai ter tamanho $\log_2 n$, suficiente para afirmar que para esse algoritmo, a complexidade de espaço é $\mathbf{S(n) = O(}log_n\mathbf{)}$, coincidentemente igual a complexidade de tempo.

3 Recorrência

a) A Figura 2 descreve a obtenção da relação de recorrência da letra a. Como ao final, obtém-se 1 + n vezes um somatório que nunca será maior que 2, temos que n domina assintoticamente o somatório, trazendo consigo que a complexidade é $\mathbf{T(n) = O(n)}$.

b) Já a Figura 3 descreve a obtenção da relação de recorrência da letra b. Como a expansão da série nos dá um temo do qual 2 elevado a n+1 domina assintoticamente os demais, temos que a complexidade da recorrência é $\mathbf{T(n) = O(}2^n\mathbf{)}$.

$$T(n) = T(n/2) + n$$
$$T(1) = 1$$

$$T(n) = T(n/2^1) + n/2^0$$
$$T(n/2^1) = T(n/2^2) + n/2^1$$
$$T(n/2^3) = T(n/2^3) + n/2^2$$
....
$$T(n) = T(\cancel{n/2^l}) + n/2^{l-1}$$
$$T(n) = T(1) + n/2^{l-1}$$

$$T(n) = T(1) + n/2^{l-1}$$
$$T(n) = 1 + \sum_{i=0}^{log_2 n} \frac{n}{2^i}$$
$$T(n) = 1 + n * \sum_{i=0}^{log_2 n} \frac{1}{2^i}$$
$$T(n) = 1 + n * \sum_{i=0}^{log_2 n} (\frac{1}{2})^i$$

Figura 2: Questão 3a.

c) Na Figura 4 é possível ver o calculo da relação de recorrência da letra c. A expansão da série nos dá $2n^2$ - n, logo, a relação de recorrência é $\mathbf{T(n)} = O(n^2)$.

d) A Figura 5 descreve a obtenção da relação de recorrência da letra d. Como ao final obtém se uma expressão composta por um somatório de uma constante com um logaritmo, temos que a relação de recorrência é, $\mathbf{T(n)} = log(n)$

4 Indução

As Figuras 6, 7 e 8 apresentam o processo da prova.

5 Listas

As funções de inserção, tanto no início quando no final de uma lista simplesmente estão apresentadas a seguir, cada qual contendo seu algoritmo e complexidade.

5.1 Inserção no Início

A função de inserção no início é dada pelo algoritmo a seguir, este de complexidade constante, visto que base acessar o primeiro elemento da lista e sobrescrever o valor da estrutura no campo **elem**.

```
void insereInicio(struct Lista *lista, int elemento)
{
    lista->elem = elemento;
}
```

5.2 Inserção ao Final

Já a função de inserção ao final, descrita pelo algoritmo a seguir possui complexidade O(n), onde n é o número de elementos da lista:

$$T(n) = 2T(n-1) + n$$
$$T(1) = 1$$

$$T(n) = 2T(n-1) + n$$
$$2T(n-1) = 2^2T(n-2) + 2(n-1)$$
$$2^2T(n-2) = 2^3T(n-3) + 2^2(n-2)$$
...

$$T(n) = 2^lT(n-l) + 2^{l-1}T(n-l-1) + ... + 2(n-1) + n$$
$$n - l = 1 => l = n - 1$$

$$T(n) = \sum_{i=0} 2^i(n-1)$$
$$T(n) = \sum_{i=0} 2^i n - \sum_{i=0} i * 2^i$$
$$T(n) = n * \sum_{i=0} 2^i - \sum_{i=0} i * 2^i$$

$$\sum_{i=0} 2^i = 2^0 + 2^1 + \ldots + 2^n + 2^{n+1}$$
$$\sum_{i=0} 2^i + 2^{n+1} = 2^0 + \sum_{i=0}^{n} 2^{i+1}$$
$$\sum_{i=0} 2^i + 2^{n+1} = 2^0 + 2\sum_{i=0}^{n} 2^i$$
$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$T(n) = n * \sum_{i=0} 2^i - \sum_{i=0} i * 2^i$$
$$T(n) = n * (2^{n+1} - 1) - \sum_{i=0} i * 2^i$$

$$\sum_{i=0} i * 2^i = 0 * 2^0 + 1 * 2^1 + 2 * 2^2 + \ldots + n * 2^n + (n+1) * 2^{n+1}$$
$$\sum_{i=0} i * 2^i + (n+1) * 2^{n+1} = 0 * 2^0 + \sum_{i=0} (i+1) * 2^{i+1}$$
$$\sum_{i=0} i * 2^i + n * 2^{n+1} + 2^{n+1} = 2\sum_{i=0} i * 2^i + 2\sum_{i=0} 2^i$$
$$\sum_{i=0} i * 2^i + n * 2^{n+1} + 2^{n+1} = 2\sum_{i=0} i * 2^i + 2(2^{n+1} - 1)$$
$$n * 2^{n+1} + 2^{n+1} = \sum_{i=0} i * 2^i + 2 * 2^{n+1} - 2$$
$$\sum_{i=0} i * 2^i = n * 2^{n+1} - 2^{n+1} + 2$$

$$T(n) = n * (2^{n+1} - 1) - (n * 2^{n+1} - 2^{n+1} + 2)$$
$$T(n) = n * 2^{n+1} - n - n * 2^{n+1} + 2^{n+1} - 2$$
$$T(n) = 2^{n+1} - n - 2$$

Figura 3: Questão 3b.

```
0 void insereFim(struct Lista *lista, int elemento)
1 {
2     struct Lista *aux = lista->ptr; // O(1)
3     while (aux->ptr != NULL) // O(1)
4     {
5         aux = aux->ptr; // O(1) repetido n vezes
6     }
7     aux->elem = elemento; // O(1)
8 }
```

Note que a função necessita de um laço de repetição, para buscar o último elemento, este indicado por **NULL**. Ou seja, o ponteiro auxiliar caminha na lista até que o próximo elemento seja vazio (linhas 3-6), encontrando assim o último elemento. Quando o último elemento é encontrado, sua posição é armazenada no ponteiro **aux**. Então, tal como na função de inserção no início, basta sobreescrever o valor do campo **elem**. Como a função depende do tamanho da lista, sua complexidade é $\mathbf{O(n)}$.

$T(n) = 4T(n/2) + n$
$T(1) = 1$

$T(n) = 4T(n/2) + n$
$4T(n/2^1) = 4^2T(n/2^1) + 4(n/2^1)$
$4^2T(n/2^2) = 4^3T(n/2^2) + 4^2(n/2^2)$
....
$T(n) = n + 4(n/2^1) + 4^2(n/2^2) + ... + 4^lT(n/2^l)$

$$T(n) = \sum_{i=0}^{n-1} 4^i(\frac{n}{2^i})$$

$$T(n) = \sum_{i=0}^{n-1} 2^i n$$

$$T(n) = n * \sum_{i=0}^{n-1} 2^i$$

$T(n) = n(2^{l+1}-1)$
$T(n) = n(2*2^l-1)$
$T(n) = n(2*n-1)$
$T(n) = 2n^2-n$

Figura 4: Questão 3c.

$T(n) = T(n/2) + log_2n$
$T(1) = 1$

$T(n) = T(n/2^1) + log_2n$
$T(n/2^1) = T(n/2^2) + log_2(n/2^1)$
$T(n/2^3) = T(n/2^3) + log_2(n/2^2)$
....
$T(n) = T(\cancel{n/2}^l) + log_2(n/2^{l-1})$
$T(n) = T(1) + log_2(n/2^{l-1})$
$l = n/2^l => l = log_2n$

$$T(n) = \sum_{i=0}^{log_2(n)-1} log_2\frac{n}{2^i}$$

$$T(n) = \sum_{i=0}^{log_2(n)-1} \frac{log_2n}{log_2 2^i}$$

$$T(n) = \sum_{i=0}^{log_2(n)-1} \frac{log_2n}{i * log_2 2}$$

$$T(n) = \sum_{i=0}^{log_2(n)-1} \frac{log_2n}{i}$$

$$T(n) = log_2n \sum_{i=0}^{log_2(n)-1} \frac{1}{i}$$

Figura 5: Questão 3d.

6 Árvores

A transformação da função de imprimir a árvore binária recursivamente em uma iterativa, é tal como descrito na função **imprimirNr()** a seguir.

```
struct Arvore pilha[TAM];
int topo = 0;

```

Hipótese
$T(n) = 2n^2 - n$ $T(2^k) = 2*(2^k)^2 - 2^k$ $T(2^k) = 2*2^{2*k} - 2^k$

Figura 6: Hipótese apresentada no enunciado.

Base: k = 1	Rel. Recor T(2)
$T(2^1) = 2*2^{2*1} - 2^1$ $T(2) = 2*2^2 - 2$ $T(2) = 2*4 - 2$ $T(2) = 8 - 2$ $T(2) = 6$	$T(2) = 4T(1) + 2$ $T(2) = 4*1 + 2$ $T(2) = 6$
Base foi provada!	

Figura 7: Base da indução, k = 1.

<table>
<tr><td colspan="2">Passo: k + 1</td></tr>
<tr><td colspan="2">$T(2^{k+1}) = 2*(2^{k+1})^2 - 2^{k+1}$
$T(2^{k+1}) = 2*2^{2k+2} - 2^{k+1}$

Qual a relação entre k+1 e k?

$2^{k+1} / 2^k$
Pela regra da potência: 2^{k+1-1}
Logo a relação é 2
Significa que $T(2^{k+1}) = 2T(2^k)$
Logo, para provar o passo, basta provar que a relação</td></tr>
<tr><td colspan="2">$T(2^{k+1}) = 2T(2^k)$</td></tr>
<tr><td>$T(n) = 2n^2 - n$
$T(2^{k+1}) = 2(2^{k+1})^2 - 2^{k+1}$
$\mathbf{T(2^{k+1}) = 2*2^{2k+2} - 2^{k+1}}$</td><td>$T(2^{k+1}) = 2T(2^k)$
$T(2^{k+1}) = 2*\text{hipotese}$
$T(2^{k+1}) = 2[2*2^{2k} - 2^k]$
$T(2^{k+1}) = 2^{2k+2} - 2*2^k$
$\mathbf{T(2^{k+1}) = 2^{2k+2} - 2^{k+1}}$</td></tr>
<tr><td colspan="2">Passo provado!</td></tr>
</table>

Figura 8: Passo da indução: k+1.

```
int vazia()
{
    return topo;
}

void empilha(struct Arvore *r)
{
    pilha[topo] = *r;
    topo++;
}

struct Arvore *desempilha() // retorna o topo
{
    topo--;
    return &pilha[topo];
}

```

```c
struct Arvore *aloca(int elemento)
{
    struct Arvore *novo = malloc(sizeof(struct Arvore));
    novo->elem = elemento;
    return novo;
}

void conecta(struct Arvore *pai, struct Arvore *esq, struct Arvore *dir)
{
    pai->esq = esq;
    pai->dir = dir;
}

void imprimirNr(struct Arvore *r)
{
    struct Arvore *aux = r;
    while (topo != 0 || aux != NULL)
    {
        while (aux != NULL)
        {
            empilha(aux);
            aux = aux->esq;
        }
        aux = desempilha();
        printf("%d ", aux->elem);
        aux = aux->dir;
    }
}

void main()
{

    struct Arvore *no1 = aloca(1);
    struct Arvore *no2 = aloca(2);
    struct Arvore *no3 = aloca(3);
    conecta(no1, no2, no3);

    struct Arvore *no4 = aloca(4);
    struct Arvore *no5 = aloca(5);
    conecta(no2, no4, no5);

    struct Arvore *no6 = aloca(6);
    struct Arvore *no7 = aloca(7);
    conecta(no3, no6, no7);

    imprimirNr(no1);
}
```

As funções referentes a pilha utilizam um vetor estático de tamanho MAX, contendo apenas inserções no topo e remoções também no topo, apenas decrementando a variável que controla essa posição no vetor. Logo, as funções de empilha e desempilha possuem $\mathbf{O(1)}$, assim como a de pilha de vazia, que retorna o tamanho do topo, se este for diferente de 0, significa que a pilha não está vazia.

6.1 Relação de Recorrência e Complexidade de Tempo

Ao realizar um análise do algoritmo, observa-se que o melhor caso, onde o nó é NULL, tem se O(1), sendo esse T(1). Como a recursão é chamada duas vezes dentro do algoritmo contando mais um passo O(1) da impressão, tem se a relação de recorrência $\mathbf{T(n) = T(n/2) + 1}$.

Na Figura 9 é possível ver a resolução da relação de recorrência, que nos dá uma complexidade de **O(log n)**.

6.2 Complexidade de espaço

A complexidade de espaço por fim, será **log n** visto que, essa será a altura máxima da pilha durante o pior caso, ou seja, quando a árvore tiver um ramo inteiro empilhado, desde a raiz até um folha. Como a altura de uma árvore balanceada é dada por log de n na base 2. Temos que a complexidade de espaço no pior caso é **O(log n).**

$T(n) = 2T(n/2) + 1$
$T(1) = 1$

$T(n) = 2T(n/2) + 1$
$2T(n/2) = 2^2T(n/2^2) + 1$
$2^2T(n/2^2) = 2^3T(n\text{-}3) + 1$
...
$T(n) = 2^lT(n\text{-}2^l) + l$
$l = \log_2 n$

$T(n) = l$
$T(n) = \log_2 n$

Figura 9: Relação de Recorrência do Percurso Recursivo em uma Árvore ABB.