Lista de Exercícios de Algoritmos e Estruturas de Dados

Primeira Lista

Universidade Federal do Rio de Janeiro – UFRJ
Professor Heraldo L. S. de Almeida, D.Sc.
Monitor Carlos Eduardo Marciano
06/05/2016

Elaborada por Carlos Eduardo Marciano

UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

Observações Iniciais

Esta disciplina é lecionada em linguagem C. Para todos os exemplos de código, assuma que os devidos *headers* já estão inclusos. A lista segue a ordem dos conteúdos dados em sala de aula.

Índice

1. Linguagem C

1.1) [Ponteiros e Arrays] Examine o seguinte código:

```
int arr[5] = { 30, 20, 50, 70, 10 };
int *parr = &arr[4];
int inx = 0;

inx = *parr++;
```

a) O código compila?

b) Após executar o código, qual será o valor de *inx*?

c) Após executar o código, para onde *parr* estará apontando?

1.2) [Aritmética de Ponteiros] Considere o seguinte código:

```
char* nome1 = "Luis";
char* nome2 = "Fernando";
char* nome3 = "Vitoria";
char* nome4 = "Leticia";

char** nomes[4] = {nome1, nome2, nome3, nome 4};

void exibir (char** arr, int tamanho);
```

Escreva o conteúdo da função *exibir*, sabendo que ela deve percorrer o *array* de nomes e printar um a um. Não há restrições para a formatação de exibição. Faça isso:

a) Utilizando o operador ++ para incrementar ponteiros.

b) Utilizando um offset int i para somar aos ponteiros.

1.3) [Nomes de Arrays] Cite um contexto em que o nome de um array não é usado como equivalente ao endereço deste array.

1.4) [Typedefs] Escreva os typedefs para os seguintes tipos:

a) ***BOOL_t***: tipo *int*.

b) ***OBJECT_t***: uma estrutura cujos dois primeiros membros, chamados *flink* e *blink*, são do tipo "pointero para *OBJECT_t*", e que o terceiro membro, chamado *object_id*, é do tipo (*char**).

c) ***OBJECT_p_t***: tipo ponteiro para *OBJECT_t*.

d) **PROC_t**: uma função que retorna *BOOL_t* e recebe um único argumento que é do tipo pointero para *OBJECT_t*.

e) **PROC_p_t**: tipo ponteiro para *PROC_t*.

1.5) [Malloc] Crie a função *malloc_list* que receba um *int x* e aloque espaço na memória para um array de *x* elementos do tipo *char**. Use um loop *for* para inicializar todos os elementos como *NULL*. A função deve retornar um ponteiro para o espaço alocado (equivalente a um ponteiro para o primeiro elemento).

1.6) [Ponteiros] O que aparecerá quando executarmos o programa abaixo?

```
int  count = 10, *temp, sum = 0;

temp = &count;
*temp = 20;
temp = &sum;
*temp = count;
printf("count = %d, *temp = %d, sum = %d\n", count, *temp, sum);
```

2. Análise Assintótica de Funções

2.1) [Equivalências] Responda às questões, justificando seu raciocínio:

a) É verdade que $2^{n+1} = O(2^n)$?

b) É verdade que $2^{2n} = O(2^n)$?

2.2) [Propriedades] Determine se cada afirmação abaixo é sempre verdadeira, nunca verdadeira ou se depende da situação. Considere as funções *f* e *g* assintoticamente não-negativas. No caso de sempre verdadeira ou nunca verdadeira, demonstre o motivo. Se sua resposta for depende da situação, dê um exemplo em que a afirmativa é verdadeira e outro exemplo em que ela é falsa.

a) $f(n) = O(f(n)^2)$

b) $f(n) + g(n) = \theta(max(f(n), g(n)))$

c) $f(n) + O(f(n)) = \theta(f(n))$

d) $f(n) = \Omega(g(n))$ *e* $f(n) = o(g(n))$

e) $f(n) \neq O(g(n))$ *e* $g(n) \neq O(f(n))$

2.3) [Limite Assintótico Superior] Suponha que cada expressão abaixo represente o tempo *T(n)* consumido por um algoritmo para resolver um problema de tamanho *n*. Escreva os termo(s) dominante(s) para valores muito grandes de *n* e especifique o menor limite assintótico superior *O(n)* possível para cada algoritmo.

Expressão	Termo(s) Dominante(s)	*O*(...)
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n\log_{10}(n)$		
$0.3n + 5n^{1.5} + 2.5n^{1.75}$		
$n^2\log_2(n) + n(\log_2(n))^2$		
$n\log_3(n) + n\log_2(n)$		
$3\log_8(n) + \log_2(\log_2(\log_2(n)))$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n\log_2(n) + n(\log_2(n))^2$		
$100n\log3(n) + n^3 + 100n$		
$0.003\log4(n) + \log_2(\log_2(n))$		

3. Análise Assintótica de Algoritmos Iterativos

3.1) [Tempo logarítmico] Um método de ordenação de complexidade $O(nlogn)$ gasta exatamente 1 milissegundo para ordenar 1000 elementos. Supondo que o tempo $T(n)$ para ordenar *n* desses elementos é diretamente proporcional a $nlogn$, ou seja, $T(n) = c.nlogn$:

a) Estime a constante c usando uma base conveniente para o logaritmo.

b) Estime o tempo consumido por esse algoritmo, em segundos, para ordenar 1,000,000 elementos.

c) É notório que, dependendo da base que escolhemos para o logaritmo, encontramos uma constante *c* diferente. Encontre uma fórmula para T(n) que independa da base escolhida para o log, sabendo que nosso algoritmo leva T(N) milissegundos para ordenar N números. *Dica*: uma divisão de logaritmos de mesma base torna essa divisão independente da base. Além disso, se a constante varia com essa base, é razoável assumir que ela não irá figurar na expressão.

3.2) [Análise de Algoritmo] Analise o algoritmo abaixo, escrito em C, que recebe dois *arrays, a* e *b,* de tamanhos iguais *n*. Determine:

```c
float f(float* a, float* b, int n) {
    int i, j;
    float s = 0.0;
    for (i=1; i<n; i++) {
        if (a[i]>600) {
            for (j=n-1; j>=0; j--) {
                s += a[i]*b[j];
            }
        } else if (a[i]<300) {
            for (j=n; j<n*n; j+=5) {
                s += a[i]*b[j];
            }
        } else {
            for (j=1; j<n; j=3*j) {
                s += a[i]*b[j];
            }
        }
    }
return s;
```

a) O maior limite assintótico inferior para o melhor caso em função do parâmetro *n*.

b) O menor limite assintótico superior para o pior caso em função do parâmetro *n*.

c) As condições que o *array a* deve satisfazer para caracterizar o melhor caso (a título de informação, a prova de 2015.1 pedia o pior caso).

3.3) [Análise de Algoritmo] Encontre o menor limite assintótico superior para o algoritmo abaixo, escrito em C:

```
int f(int n) {
    int i, j, k, sum = 0;
    for ( i=1; i < n; i *= 2 ) {
        for ( j = n; j > 0; j /= 2 ) {
            for ( k = j; k < n; k += 2 ) {
                sum += (i + j * k );
            }
        }
    }
}
```

3.4) [Análise de Algoritmo] Suponha que o *array a* contenha *n* valores. Suponha também que a função *randomValue* necessite de um número constante de processamentos para retornar cada valor, e que a função *goodSort* leve um número de etapas computacionais proporcional a $n \log n$ para ordenar o array. Determine o maior limite assintótico inferior possível para o seguinte fragmento de código, escrito em C:

```
for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
        a[ j ] = randomValue( i );
    }
    goodSort( a );
}
```

3.5) [Comparação de Algoritmos] Suponha que ofereçam a você dois pacotes de software, **A** e **B**, para processamento dos dados de sua empresa, que contêm 10^9 registros. O tempo de processamento médio do pacote **A** é $T_A(n) = 0.001n$ milissegundos, e o tempo médio de **B** é $T_B(n) = 500\sqrt{n}$ milissegundos.

a) Qual desses pacotes é o mais indicado para processar os dados da empresa?

b) A partir de quantos registros um dos pacotes passa a ser melhor que o outro?

4. Análise Assintótica de Algoritmos Recursivos

4.1) [Análise de Algoritmo] Utilize uma das técnicas conhecidas de análise de algoritmos recursivos e forneça um limite assintótico restrito θ() para cada algoritmo abaixo, escrito em C:

a)
```c
int easyQuestion(int* A, int n) {
    int x, i;
    if (n < 20){
        return (A[0]);
    }
    x = easyQuestion(A, 3*n/4);
    for (i=n/2; i<(n/2)+8; i++) {
        x += A[i];
    }
    return x;
}
```

b)
```c
int thisOneIsTricky(int* A, int n) {
    if (n < 12){
        return (A[0]);
    }
    int y, i, j, k;
    for (i=0; i<n/2; i++) {
        for (j=0; j<n/3; j++) {
            for (k=0; k<n; k++) {
                A[k] = A[k] - A[j] + A[i];
            }
        }
    }
    y = thisOneIsTricky(A, n-5);
    return y;
}
```

c)

```
int youWontGuessThisOne(int* A, int n){
    if (n < 50) {
        return (A[n]);
    }
    int x, j;
    x = youWontGuessThisOne(A, n/4);
    for (j=0; j<n/3; j++) {
        A[j] = A[n-j] - A[j];
    }
    x += youWontGuessThisOne(A, n/4);
    return x;
}
```

d)

```
int okLastOneIPromise(int* A, int n){
    if (n < 15) {
        return (A[n]);
    }
    int x=0, i, j, k;
    for (i = 0; j<4; j++){
        for (j=0; j<n-i; j++){
            for (k=0; k<n/2; k++){
                A[j] = A[k] - A[n-j];
            }
        }
        x += okLastOneIPromise(A, n/2);
    }
    return x;
}
```

A. Respostas

A.1 Linguagem C

1.1) **a)** Sim.

b) 10.

c) O ponteiro *parr* estará apontando para fora do array.

1.2) **a)**

```
void exibir (char** arr, int tamanho){
    int i;
    for (i=0; i<tamanho; i++){
        printf("%s\n", *arr++);
    }
}
```

b)

```
void exibir (char** arr, int tamanho){
    int i;
    for (i=0; i<tamanho; i++){
        printf("%s\n", *(arr+i));
    }
}
```

1.3) Uma das seguintes possibilidades:

- quando utilizado pelo operador *sizeof()*;

- quando utilizado como lvalue de uma atribuição (erro de compilação, o que não aconteceria com um ponteiro comum).

1.4) **a)**

```
typedef int BOOL_t;
```

b)

```
typedef struct Node OBJECT_t;
struct Node
{
    OBJECT_t* flink;
    OBJECT_t* blink;
    char* object_id;
};
```

c)

```
typedef OBJECT_t* OBJECT_p_t;
```

d)

```
typedef BOOL_t PROC_t(OBJECT_t*);
```

e)

```
typedef PROC_t* PROC_p_t;
```

1.5)

```c
char** malloc_list (int x){

    char** pArray;
    pArray = malloc(sizeof(*pArray)*x);

    //      Observação: note como não há tipos-padrão (int, char, etc)
    // na linha do malloc. A função malloc retorna um ponteiro para null,
    // que é automaticamente promovido para qualquer outro tipo (no caso,
    // para char**). Ao declararmos pArray com antecedência, abrimos mão
    // do typecast, visto que o compilador já fica sabendo qual seu tipo.
    //      Também suprimimos o tipo-padrão dentro do operador sizeof(),
    // substuindo-o pelo nome *pArray, equivalente a char* no contexto.
    //      Isso contribui para uma manutenção mais eficiente do código.
    //      Se preferir o método mais convencional, pode-se utilizar:
    //          pArray = (char**) malloc(sizeof(char*));
    //      Leitura sugerida: http://stackoverflow.com/questions/605845/do-i-c
    // ast-the-result-of-malloc

    int i;
    for (i=0; i<x; i++){
        pArray[i] = NULL;
    }

    return pArray;
}

//      Uma função como essa, que aloca memória para um número variável de
// elementos, pode ser perigosa. No cotidiano, você precisa gravar o tamanho
// do array em algum lugar para que seja possívell usar o free() depois.
```

1.6)

```
count = 20, *temp = 20, sum = 20

Process returned 33 (0x21)   execution time : 0.028 s
Press any key to continue.
```

A.2 Análise Assintótica de Funções

2.1) a) É verdade. Para que 2^{n+1} pertença a $O(2^n)$, é preciso achar uma constante *c* tal que, para algum valor de *m*, a seguinte desigualdade seja verdadeira:

$$2^{n+1} \leq c.2^n \quad , \quad \text{para } n \geq m$$

Notamos que a potência 2^{n+1} é equivalente a 2.2^n, fornecendo c=2

b) Falso. Para que 2^{2n} pertença a $O(2^n)$, é preciso achar uma constante *c* tal que, para algum valor de *m*, a seguinte desigualdade seja verdadeira:

$$2^{2n} \leq c.2^n \quad , \quad \text{para } n \geq m$$

Uma vez que 2^{2n} equivale a $2^n.2^n$, sempre haverá um valor de *n* maior do que qualquer constante *c* que possamos escolher. Assim, 2^n não é um limite assintótico superior para 2^{2n}.

2.2) a) Sempre verdadeira.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f(n)}{f(n)^2} = \lim_{n\to\infty}\frac{1}{f(n)} < \infty \text{ (se } f(n) \text{ não tender a zero)}$$

b) Sempre verdadeira. É uma das propriedades dos limites assintóticos.

c) Depende da situação. Apenas é verdadeiro quando *O(f(n))* for da menor ordem possível.

- Exemplo verdadeiro:

  $f(n) = 2n + 3$ e $O(f(n)) = n$
  Então $3n+3 = \theta(n) = \theta(f(n))$

- Exemplo falso:

  $f(n) = 2n + 3$ e $O(f(n)) = n^2$
  Então $n^2+2n+3 = \theta(n^2) \neq \theta(f(n))$

d) Sempre falsa.

A primeira condição exige que $\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0$; a segunda, que $\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$

e) Sempre falsa.

A condição suficiente para ser *O(f(n))* é: $\lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty$

Analogamente, para <u>não</u> ser: $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \infty$

Para que ambos fossem verdadeiras, seria preciso também que $\lim_{n\to\infty}\frac{g(n)}{f(n)} = \infty$

2.3)

Expressão	Termo(s) Dominante(s)	*O(...)*
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10}(n)$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2\log_2(n) + n(\log_2(n))^2$	$n^2\log_2(n)$	$O(n^2 logn)$
$n\log_3(n) + n\log_2(n)$	$n\log_3(n)$; $n\log_2(n)$	$O(nlogn)$
$3\log_8(n) + \log_2(\log_2(\log_2(n)))$	$3\log_8(n)$	$O(logn)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n\log_2(n) + n(\log_2(n))^2$	$n(\log_2(n))^2$	$O(n(logn)^2)$
$100n\log_3(n) + n^3 + 100n$	n^3	$O(n^3)$
$0.003\log4(n) + \log_2(\log_2(n))$	$0.003\log4(n)$	$O(logn)$

A.3 Análise Assintótica de Algoritmos Iterativos

3.1) a) Dada a natureza dos números no enunciado, torna-se interessante usar logaritmos na base 10. Substituindo os valores na expressão $T(n) = c.nlogn$:

$$1 = c.1000.\log_{10}1000$$
$$1 = 3000.c$$
$$c = 1/3000$$

b) Substituindo os valores do item anterior na expressão $T(n) = c.nlogn$:

$$T(n) = \frac{1}{3000}.10^6.\log_{10}10^6$$

T(n) = 2000 milissegundos = 2 segundos

c) Para N qualquer, temos a expressão $T(N) = c.NlogN$. Ao isolarmos a constante *c*, encontramos $c = \frac{T(N)}{N \log N}$. Agora, basta substituirmos este resultado na expressão geral $T(n) = c.nlogn$, o que nos dá a resposta $T(n) = T(N)\frac{n \log n}{N \log N}$, que não depende da constante, mas sim do resultado da busca de N elementos.

Para exercitar, tente recalcular a questão anterior usando essa nova expressão.

3.2) Acompanhe a análise assintótica de cada laço do algoritmo abaixo:

```
float f(float* a, float* b, int n) {
    int i, j;
    float s = 0.0;
    for (i=1; i<n; i++) {
        if (a[i]>600) {
            for (j=n-1; j>=0; j--) {
                s += a[i]*b[j];                 O(n)
            }
        } else if (a[i]<300) {
            for (j=n; j<n*n; j+=5) {
                s += a[i]*b[j];                 O(n²)           O(n)
            }
        } else {
            for (j=1; j<n; j=3*j) {
                s += a[i]*b[j];                 O(logn)
            }
        }
    }
return s;
```

Algumas considerações: o loop exterior é *O(n)* devido ao fato de precisamente *n* iterações ao longo de sua execução. Sobre os loops interiores:

- <u>Caso a[i]>600:</u> neste loop, há precisamente n iterações, como pode ser visto pelos valores que a variável *j* assume.
- <u>Caso a[i]<300:</u> aqui, há iterações proporcionais a $n*n = n^2$. Não importa que *j* seja incrementado de 5 em 5. No máximo, isso fará com quem existam $n^2/5$ iterações, que ainda é $O(n^2)$.
- <u>Caso remanescente:</u> neste loop, a variável *j* é incrementada em uma progressão geométrica de razão 3 (i.e.: 1, 3, 9, 27, 81, 243...). Pegando um exemplo desta progressão, para $n=243$, temos $\log_3 243 = 5$ iterações. Portanto, concluímos que este laço possui limite assintótico *O(logn)*.

a) O melhor caso ocorre quando todas as iterações caem na última condição. Este loop é θ*(logn)*, ou seja, possui limites assintóticos superiores e inferiores iguais. O loop exterior é θ*(n)*. Logo, pela regra do produto, o maior limite assintótico inferior para o melhor caso é ω*(nlogn)*.

b) O pior caso ocorre quando todas as iterações caem na segunda condição. Usando um raciocínio análogo ao utilizado no item anterior, concluímos, pela regra do produto, que o menor limite assintótico superior para o pior caso é *O(n^3).*

c) Para que o melhor caso ocorra, todos os elementos a_n do array a devem satisfazer a condição $300 \leq a_n \leq 600$.

3.3) O tempo de execução dos loops exterior, intermediário e interior é proporcional a *logn* (veja como *i* cresce numa PG de razão 2), *logn* (veja como j atende a uma PG de razão ½) e *n* (a variável k cresce em PA), respectivamente. Assim, o menor limite assintótico superior que podemos encontrar, pela regra do produto, é *O(n(logn)²)*.

3.4) O loop interior demanda um número de processamentos proporcional a *n*, mas a função chamada logo a seguir possui complexidade maior, proporcional a *nlogn*. Pela regra da soma, o tempo de execução da função é dominante sobre o loop. Dado que o loop exterior, que engloba ambos os itens analisados, tem complexidade n, chegamos à conclusão que o maior limite assintótico inferior possível é ω*(n^2logn)*.

3.5) a) Substituindo 10^9 em $T_A(n)$, obtemos $T_A(10^9)$ = 1,000,000 milissegundos = 1,000 segundos. Quando substituímos 10^9 em $T_B(n)$, encontramos $T_B(10^9) \approx$ 15,811,388 milissegundos ≈ 15,811 segundos. Logo, concluímos que o pacote **A** é mais adequado ao número de registros com o qual estamos trabalhando.

b) Encontrar quando um pacote apresenta desempenho superior ao outro é o mesmo que calcular qual o valor de *n* em que os tempos de cada um se igualam. O valor que satisfaz $T_A(n) = T_B(n)$ é $n=250.10^9$. Logo, nossos dados precisariam ser 250 vezes maiores para compensar o uso do pacote **B**.

A.4 Análise Assintótica de Algoritmos Recursivos

4.1) a) É possível perceber que, para cada recursão, o algoritmo executa um laço de complexidade constante (o loop *for* executa aproximadamente 8 iterações em qualquer circunstância) e realiza sua chamada recursiva de tamanho *3n/4*. Assim, chegamos na seguinte relação de recursividade:

$$T(n) = T(3n/4) + c$$

Sinta-se livre para utilizar um dos seguintes métodos:

- <u>Método Mestre:</u> é possível perceber que os valores de a (constante que multiplica o tempo T, ou seja, número de subproblemas), b (constante pela qual o tamanho do problema é dividido a cada chamada) e c (expoente da complexidade polinomial $f(n)$ fora das chamadas recursivas) que procuramos são a=1, b=4/3 e c=0. Isso nos leva ao caso em que $c=\log_b a$:

$$T(n) \in \Theta(n^{\log_b a} \log_b n) =$$
$$\Theta(n^0 \log_{4/3} n) =$$
$$\boldsymbol{\Theta(\log n)}$$

- <u>Método de Análise da Recursão:</u> outra forma de resolver esse problema é analisando como ocorre cada chamada recursiva:

$$c + T(3n/4) =$$
$$c + c + T((3/4)^2 n) =$$
$$c + c + c + T((3/4)^3 n) =$$
$$c + c + \ldots + c + T(1) = \quad \} \log_{4/3} n \text{ termos}$$
$$c + c + \ldots + c + c = \quad \} \log_{4/3} n \text{ termos}$$
$$\mathbf{c.\log_{4/3} n} \in \boldsymbol{\Theta(\log n)}$$

b) Ao analisarmos o algoritmo, percebemos que, para cada recursão, há 3 loops *for* de complexidade $\Theta(n)$ cada (totalizando $\Theta(n^3)$) e uma chamada recursiva, fora dos loops, de tamanho *n-5*. Assim, chegamos à seguinte relação de recursividade:

$$T(n) = T(n-5) + cn^3$$

- <u>Método de Análise da Recursão:</u> vamos agora analisar o que ocorre a cada chamada recursiva. Vamos fazer isso em duas etapas: na primeira, estaremos buscando um limite assintótico superior. Na segunda, estaremos interessados em achar um limite assintótico inferior. Ao provarmos que esses limites são iguais, poderemos afirmar que o limite assintótico restrito θ( ) é de igual complexidade.

I) Para o limite assintótico superior:

$$cn^3 + T(n-5) =$$
$$cn^3 + c(n-5)^3 + T(n-10) =$$
$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + T(1)}_{n/5 \text{ termos}} =$$
$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + c}_{n/5 \text{ termos}}$$

A última expressão acima é notavelmente menor que:

$$\underbrace{cn^3 + cn^3 + cn^3 + cn^3 + \ldots + cn^3}_{n/5 \text{ termos}}$$

Logo, ao construirmos uma expressão necessariamente maior que o tempo do algoritmo com o qual estamos trabalhando, podemos dizer que essa expressão define um limite assintótico superior para o tempo de execução do algoritmo. Assim, como há *n/5* termos, fazemos:

$$c.n^3.(n/5) = c'.n^4 \in \mathbf{O(n^4)}$$

II) Para o limite assintótico inferior:

$$cn^3 + T(n-5) =$$
$$cn^3 + c(n-5)^3 + T(n-10)$$

Vamos continuar expandindo as chamadas recursivas até chegarmos em uma chamada cujo tamanho é a metade de n:

$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + T(n/2)}_{n/10 \text{ termos}}$$

A chamada T(n/2) resultará em $c(n/2)^3 + T((n/2)-5)$. Visto que um termo somado a algo positivo é necessariamente maior do que ele mesmo, temos uma desigualdade bastante ingênua:

$$c(n/2)^3 + T((n/2)-5) \geq c(n/2)^3$$

Apesar de óbvia, esta desigualdade nos permitirá gerar uma nova desigualdade, dessa vez com relação à expressão do tempo de execução:

$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + T(n/2)}_{n/10 \text{ termos}} \geq$$
$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + c(n/2)^3}_{n/10 \text{ termos}}$$

Visto que todos os termos dependentes de *n* que multiplicam a constante *c* (por exemplo: n-5) são maiores que *n/2*, a desigualdade abaixo é válida:

$$\underbrace{cn^3 + c(n-5)^3 + c(n-10)^3 + c(n-15)^3 + \ldots + c(n/2)^3}_{n/10 \text{ termos}} \geq$$
$$\underbrace{c(n/2)^3 + c(n/2)^3 + c(n/2)^3 + c(n/2)^3 + \ldots + c(n/2)^3}_{n/10 \text{ termos}}$$

Finalmente, temos uma expressão necessariamente menor do que o tempo de execução do algoritmo e que pode facilmente nos fornecer um limite assintótico inferior deste tempo. Multiplicando o termo pelo número de vezes que ele aparece:

$$c.(n^3/8).(n/10) = c'.n^4 \in \boldsymbol{\omega(n^4)}$$

III) Definindo o limite assintótico restrito:

Tendo em mãos um limite assintótico superior e um limite assintótico inferior, concluímos nosso raciocínio com o limite assintótico restrito:

$$T(n) \in O(n^4)$$
$$T(n) \in \omega(n^4)$$
$$\text{Logo, } \mathbf{T(n) \in \theta(n^4)}$$

c) O algoritmo executa, para qualquer caso diferente do caso base, duas chamadas recursivas (uma antes do loop e outra depois) e um *loop for* de complexidade $\theta(n)$. Isto nos dá a seguinte equação de recorrência:

$$T(n) = 2*T(n/4) + cn$$

Sinta-se livre para utilizar um dos seguintes métodos:

- Método Mestre: é possível perceber que os valores de a (constante que multiplica o tempo T, ou seja, número de subproblemas), b (constante pela qual o tamanho do problema é dividido a cada chamada) e c (expoente da complexidade polinomial $f(n)$ fora das chamadas recursivas) que procuramos são a=2, b=4 e c=1. Isso nos leva ao caso em que $c > \log_b a$ (pois 1 > 0,5):

$$T(n) \in \theta(f(n)) =$$
$$\boldsymbol{\theta(n)}$$

- Método de Análise da Recursão: outra forma de resolver esse problema é analisando como ocorre cada chamada recursiva:

$$cn + 2\,T(n/4) =$$
$$cn + 2\,(\,c.(n/4) + 2\,T(n/4^2)\,) \;\Rightarrow\; cn + cn(2/4) + 2^2.T(n/4^2) =$$
$$cn + cn(2/4) + cn(2/4)^2 + cn(2/4)^3 + \dots + 2^k T(n/4^k) = \quad \text{em que } k = \log_4(n)$$
$$cn + cn(2/4) + cn(2/4)^2 + cn(2/4)^3 + \dots + 2^{\log_4(n)}c =$$
$$cn + cn(2/4) + cn(2/4)^2 + cn(2/4)^3 + \dots + (2/4)^{\log_4(n)}cn = \quad \text{já que } n/4^{\log_4(n)} = 1$$
$$cn(\,1 + (1/2) + (1/2)2 + \dots + (1/2)^{\log4(n)}\,) \approx \quad \text{PG de razão ½ e } a_0 = 1$$
$$cn(2) = 2cn = \quad \text{soma da PG}$$
$$\mathbf{c'.n \in \boldsymbol{\theta(n)}}$$

d) A recursividade encontra-se dentro de um *loop for* de 4 iterações e, portanto, é chamada quatro vezes por execução, cada uma dividindo o problema pela metade. O trabalho desenvolvido fora da recursão encontra-se principalmente dentro dos dois *loops for* mais internos, de complexidade $\theta(n)$ cada. Como esses dois loops estão entrelaçados, temos uma complexidade total de $\theta(n^2)$. Isto nos leva à seguinte relação de recursividade:

$$T(n) = 4\ T(n/2) + cn^2$$

Sinta-se livre para utilizar um dos seguintes métodos:

- <u>Método Mestre:</u> é possível perceber que os valores de *a* (constante que multiplica o tempo T, ou seja, número de subproblemas), *b* (constante pela qual o tamanho do problema é dividido a cada chamada) e *c* (expoente da complexidade polinomial *f(n)* fora das chamadas recursivas) que procuramos são a=4, b=2 e c=2. Isso nos leva ao caso em que $c = \log_b a$:

$$T(n) \in \theta(n^{\log_b a} \log_b n) =$$
$$\theta(n^2 \log_2 n) =$$
$$\boldsymbol{\theta(n^2 \log n)}$$

- <u>Método de Análise da Recursão:</u> outra forma de resolver esse problema é analisando como ocorre cada chamada recursiva:

$$cn^2 + 4\ T(n/2) =$$
$$cn^2 + 4(\ c.(n/2)^2 + 4\ T(n/2^2)\ ) \quad => \quad cn^2 + cn^2 + 4^2.T(n/2^2)) =$$
$$cn^2 + cn^2 + 4^2(c.(n/2^2)^2 + 4\ T(n/2^3)) \quad => \quad cn^2 + cn^2 + cn^2 + 4^3.T(n/2^3) =$$
$$cn^2 + cn^2 + cn^2 + \ldots + cn^2 + 4^{\log_2(n)}.T(1) = \quad \} \log_2 n \text{ termos}$$
$$cn^2 + cn^2 + cn^2 + \ldots + cn^2 + cn^2 \qquad \text{já que } 4^{\log_2(n)} = 2^{2\log_2(n)} = n^2$$

Multiplicando o termo comum pelo número de vezes que ele aparece:

$$\boldsymbol{c.n^2.\log_2 n \in \theta(n^2 \log n)}$$

B. Bibliografia

- CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: Teoria e Prática. Tradução de: Introduction to Algorithms, 3rd ed. Rio de Janeiro: Elsevier, 2012.
- Data Structures and Algorithms, The Ohio State University. Disponível em: <*http://web.cse.ohio-state.edu/~lrademac/Fa14_2331/RecursiveAnalysis.pdf*>. Acessado em: 06/05/2016.
- Algorithms and Data Structures, The University of Auckland. Disponível em: <*https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/220exercises1.pdf*>. Acessado em: 06/05/2016.
- Lista de Exercícios do ex-monitor Felipe. Disponível em: <*http://www.del.ufrj.br/~heraldo/eel470_lista1.pdf*>. Acessado em: 06/05/2016.