UDESC

Complexidade de Algoritmos

Prof. Rafael Alceste Berri

rafaelberri@gmail.com

Prof. Diego Buchinger

diego.buchinger@outlook.com

Prof. Cristiano Damiani Vasconcellos

cristiano.vasconcellos@udesc.br

Análise de Complexidade de Tempo de Algoritmos Recursivos

Algoritmos Recursivos

Geralmente a análise de complexidade de **algoritmos recursivos** é um pouco **mais complexa** pois é preciso entender bem a recursividade do algoritmo.

- Quantas vezes um algoritmo chama a si mesmo?
- Existe um pior caso? Quantas chamadas recursivas são feitas?
- Devemos considerar o tempo gasto na chamada de função? E o espaço utilizado na memória?

Algoritmos Recursivos - Execução

- Quando uma chamada de função é feita, é criado **um registro** de ativação na **pilha** de execução do programa;
- O **registro de ativação** guarda:
  - Os parâmetros e variáveis locais da função;
  - O ponto de retorno da função.
- Quando a **função termina**, o registro de ativação é **desempilhado** e a execução **volta ao subprograma** que chamou a função.

Algoritmos Recursivos

Para se obter a complexidade de um algoritmo recursivo, precisa-se primeiro utilizar a estrutura do algoritmo e identificar:

- Especifica-se T(n) como uma **função dos termos anteriores** (relação de **recorrência**);
- Especifica-se a **condição de parada** (ex: T(1)).

Exemplo (Fatorial)

Fatorial
- 0! = 1
- n! = n(n-1)!

```
int fatorial( int n ){
   if (n == 0)
        return 1;
   return n * fatorial( n-1 );
}
```

- **Condição de parada?**
- **Qual relação de recorrência T(n)?**

Exemplo (Fatorial)

```
int fatorial( int n ){
   if (n == 0)
       return 1;
   return n * fatorial( n-1 );
}
```

Relação de Recorrência:

$T(n) = T(n\text{-}1) + O(1)$

$T(0) = O(1)$ (Parada)

Fatorial (não recursivo)

```
int fatorial(int n)
{
  int f = 1;
  while(n > 0)
  {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Fatorial (não recursivo)

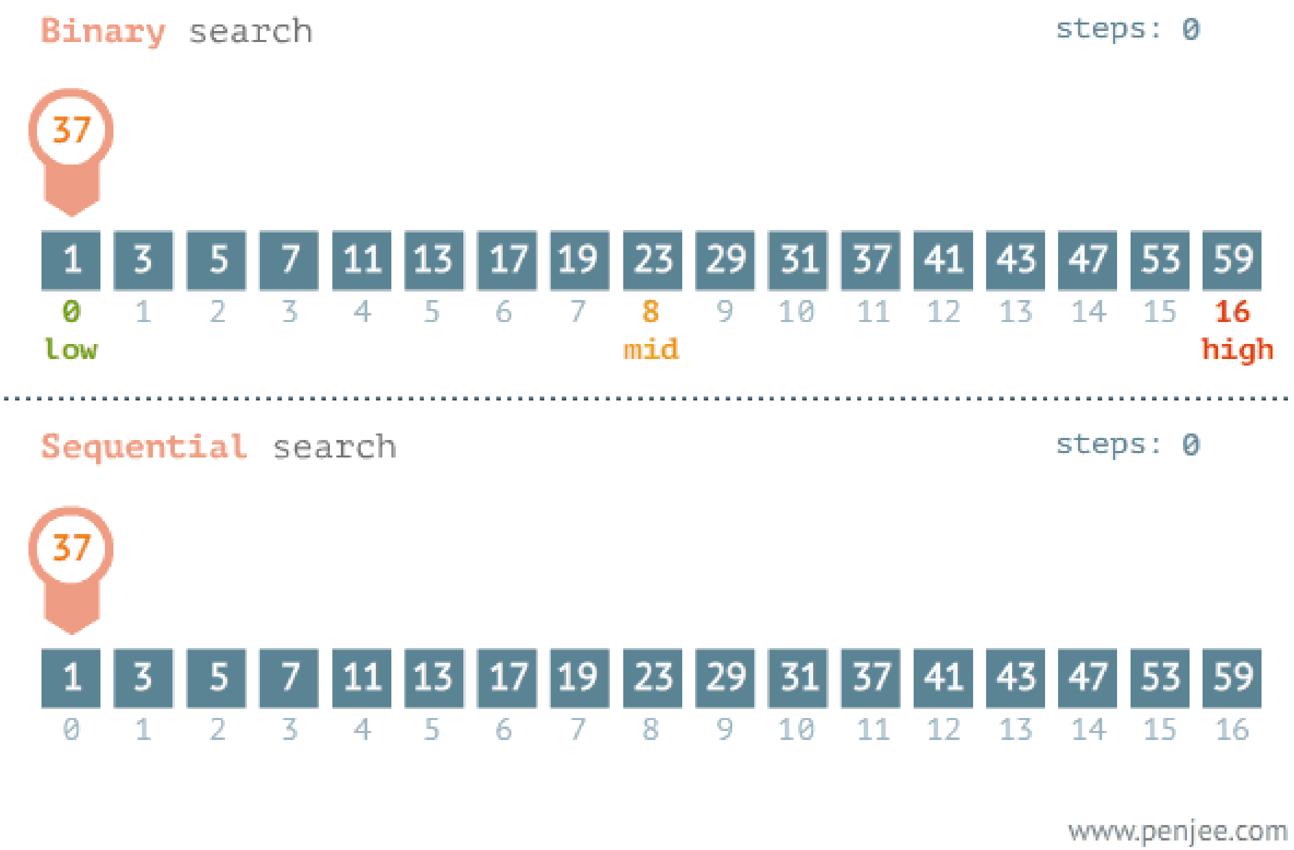
- Complexidade de Tempo: O(n)
- Complexidade de Espaço: O(1)

```
int fatorial(int n)
{
  int f = 1;
  while(n > 0)
  {
    f = f * n;
    n = n - 1;
  }
  return f;
}
```

Recursividade é a melhor solução?

- Recursividade nem sempre é a melhor solução, mesmo quando a definição matemática do problema é feita em termos recursivos;
  - Normalmente, maior complexidade de espaço que a versão não recursiva.

Exemplo (Pesquisa Binária)



Exemplo
(Pesquisa Binária)

```
int pesqbin(int *v, int p, int r, int e){
    int q;
    if ( r < p )
        return -1;
    q = (p + r)/2;
    if (e == v[q])
        return q;
    if (e < v[q])
        return pesqbin(v, p, q-1, e);
    return pesqbin(v, q+1, r, e);
}
```

** Lembrete: o vetor deve estar ordenado!!*

Exemplo (Pesquisa Binária)

```c
int pesqbin(int *v, int p, int r, int e){
   int q;
   if ( r < p )
       return -1;
   q = (p + r)/2;
   if (e == v[q])
       return q;
   if (e < v[q])
       return pesqbin(v, p, q-1, e);
   return pesqbin(v, q+1, r, e);
}
```

Relação de Recorrência:

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

$T(1) = O(1)$

A relação abaixo descreve de forma mais precisa a execução, mas a simplificação acima não altera a complexidade:

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

$T(1) = O(1)$

Relação de Recorrência

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
T(n/2) &= T(n/2^2) + 1 \\
T(n/2^2) &= T(n/2^3) + 1 \\
\ldots & \qquad \ldots \\
T(n/2^{h-1}) &= T(n/2^h) + 1
\end{aligned}
$$

Relação de Recorrência:

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

$T(1) = O(1)$

Relação de Recorrência

$T(n) \quad = \cancel{T(n/2)} + 1$

$\cancel{T(n/2)} \quad = \cancel{T(n/2^2)} + 1$

$\cancel{T(n/2^2)} \quad = \cancel{T(n/2^3)} + 1$

...

$\cancel{T(n/2^{h-1})} \quad = T(\, n/2^h\,) + 1$

Relação de Recorrência:

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

$T(1) = O(1)$

$$T(n) = \underbrace{1 + 1 + \ldots + 1}_{} + T(1)$$

h vezes => ops... Mas quanto vale h??

Relação de Recorrência

$T(n) = \cancel{T(n/2)} + 1$

$\cancel{T(n/2)} = \cancel{T(n/2^2)} + 1$

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

...

$\cancel{T(n/2^{h-1})} = T(n/2^h) + 1$

$$\boxed{\frac{n}{2^h} = 1} \qquad \boxed{\begin{array}{l} n = 2^h \\ h = \log_2 n \end{array}}$$

$$T(n) = \underbrace{1 + 1 + \ldots + 1}_{} + T(1)$$

h vezes, ou seja, $log_2\, n$ *vezes*

O(log n)

Mudança de base:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Exemplo
(Pesquisa Binária)

```
int pesqbin2 (int *v, int n, int e) {
    int p, q, r;

    p = 0; r = n-1;
    do {
        q = (p + r) / 2;
        if (e == v[q])
            return q;
        if (e < v[q])
            r = q -1;
        else
            p = q + 1;
    } while (p<= r);
    return -1;
}
```

Recursividade de cauda

Uma função apresenta recursividade de cauda se **nenhuma operação é executada após o retorno da chamada** recursiva, exceto retornar seu valor.

Em geral, **compiladores**, que executam **otimizações** de código, substituem as funções que apresentam recursividade de cauda por uma versão **não recursiva** dessa função.

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

* Por que implementação ruim? O que é ineficiente?

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

Relação de Recorrência:

$T(n) = T(n\text{-}1) + T(n\text{-}2) + O(1)$

$T(1) = O(1)$

$T(0) = O(1)$

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

Relação de Recorrência simplificada:

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

$T(0) = O(1)$

UDESC

Relação de Recorrência

$T(n) = 2T(n\text{-}1) + O(1)$

$2T(n\text{-}1) = 2^2T(n\text{-}2) + 2O(1)$

$2^2T(n\text{-}2) = 2^3T(n\text{-}3) + 2^2O(1)$

... ...

$2^{n-1}T(1) = 2^nT(n\text{-}n) + 2^{n-1}O(1)$

$2^nT(0) = 2^n\,O(1)$

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

$$\boldsymbol{O(\ 2^n\ )}$$

Relação de Recorrência:

$T(n) = 2T(n\text{-}1) + O(1)$

$T(0) = O(1)$

Exemplo (Fibonacci)

Ao usar a relação de recorrência correta chegaríamos em uma resposta parecida:

Relação de Recorrência:

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

T(1) = O(1)

T(0) = O(1)

$$\boldsymbol{O(\ \varphi^{n}\ )}$$

$$\varphi = \frac{1+\sqrt{5}}{2}$$

$$\varphi \cong 1{,}6180$$

Exemplo (Fibonacci)

- Ineficiente
  - Termos T(n-1) e T(n-2) são computados independentemente e repetidas vezes;
  - Número de chamadas recursivas é igual ao número de fibonacci sendo calculado;
  - Custo para o cálculo de T(n)
    - $\boldsymbol{O(\varphi^n)}$
    - φ = (1 + 51/2)/2 = 1,61803 é a proporção áurea
    - **Complexidade exponencial**

Quando é útil usar recursividade?

- Problemas cuja implementação iterativa é **complexa** e requer uso explícito de uma **pilha**
  - Algoritmos tipo **dividir para conquistar** (quicksort);
  - Caminhamento em **árvores**;
  - Busca exaustiva.

Atividade 3

Escreva duas versões do algoritmo de fibonacci: a versão “ruim” apresentada e uma versão “boa” usando vetores. Faça uma comparação de tempo de execução com valores entre 25 e 60.

Qual a complexidade da versão “boa” do algoritmo?

Envie para rafaelberri@gmail.com Assunto: “TC-CAL03”
Anexar: resultados/respostas + códigos fontes

Propriedades dos Somatórios

$$\sum_{i=0}^{n} ca_i = c\sum_{i=0}^{n} a_i \quad \text{(Distributiva)}$$

$$\sum_{i=0}^{n} (a_i + b_i) = \sum_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i \quad \text{(Associativa)}$$

$$\sum_{i=n}^{0} a_i = \sum_{i=0}^{n} a_i \quad \text{(Comutativa)}$$

Propriedades dos Somatórios

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

$$\sum_{i=1}^{\log n} i = n \log n$$

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

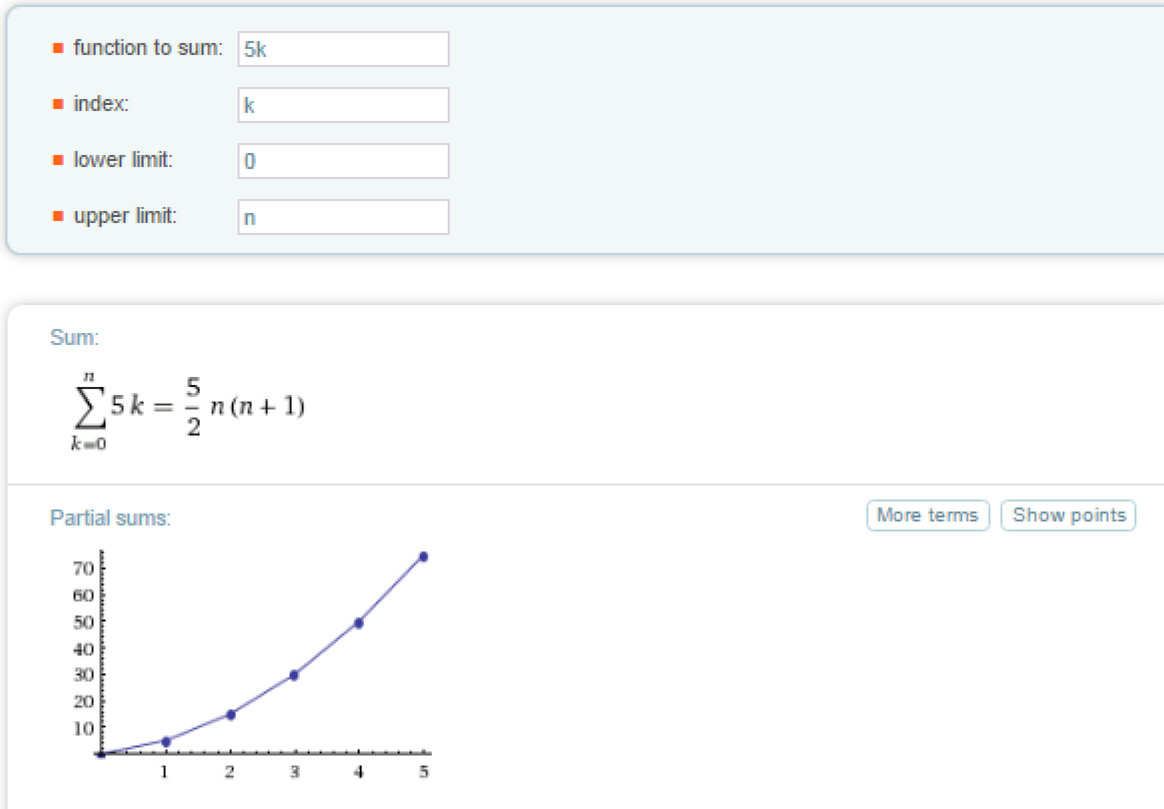
$$\sum_{i=0}^{\log n} 2^i = 2n - 1$$

$$\sum_{i=1}^{n} \log i = \log(n!)$$

$$\boxed{\frac{1}{2} n \log n \leq \log(n!) \leq n \log n \quad \therefore \quad \log(n!) = \Theta(n\ log\ n)}$$

Somatórios

Existem algumas ferramentas que calculam as fórmulas dos somatórios e até mesmo geram gráficos.



Ex: https://www.wolframalpha.com/input/?i=sum

Propriedades das Potências

$$a^m a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}, a \neq 0$$

$$(ab)^n = a^n b^n$$

$$\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}, b \neq 0$$

$$(a^m)^n = a^{nm}$$

Propriedades dos Logaritmos

$$a^b = c \Leftrightarrow \log_a c = b$$

$$a^{\log_a b} = b$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^c = c\log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Atividades de sala

Resolva as seguintes recorrência:

(Utilize os somatórios conhecidos para chegar a um resultado final ou utilize a ferramenta apresentada)

a. $T(n) = T(n-1) + \Theta(\log n)$

$T(1) = \Theta(\log 1)$

b. $T(n) = 3T(n-1) + \Theta(1)$

$T(1) = \Theta(1)$

c. $T(n) = T(\sqrt{n}) + \Theta(1)$

$T(2) = \Theta(1)$

Atividades de sala

Resolva os seguintes somatórios:

$$a)\ \sum_{i=0}^{n}(2i+1)$$

$$c)\ \sum_{i=0}^{\log_2 n} 2^i$$

$$b)\ \sum_{i=0}^{n}(ni-5i)$$

$$d)\ \sum_{i=0}^{\sqrt{n}} i$$

Atividade 4

Elabore um algoritmo recursivo para “gerar as posições de marcação de uma régua” com comprimento informado por parâmetro (posição esq/dir). A função recursiva receberá a altura inicial efetuando a marcação central com esta altura. Divida a régua em 4 partes e efetue uma nova marcação central com altura – 1 em cada parte. E assim sucessivamente... Faça enquanto altura da marca é maior que zero. Imprima na tela a posição da marca e sua respectiva altura para testar. Após implementar o algoritmo, escreva a relação de recorrência e a respectiva complexidade do algoritmo para o pior caso.

Envie para rafaelberri@gmail.com Assunto: “TC-CAL04”
Anexar: resultados/respostas + códigos fontes

Método da Substituição

- Para Mergesort: $T(n) = 2(T(n/2)) + O(n)$
  Parada: $T(1) = O(1)$
- Basta "chutar" uma resposta (limite assintótico), e em seguida, provar por indução que funciona!
- Por exemplo, suponha a recursão da forma:

  $T(n) = 2\ T(n/2) + n$
- Supondo que a solução seja $T(n) = O(\ n\log(n))$
- Agora, como provamos que isto é verdade?
- Basta provar que $T(n) \leq c * n * \log(n)$ para algum $c > 0$.

Método da Substituição

- Iniciamos assumindo que a fórmula é verdade para n/2:

  $T(n/2) \leq c * n/2 * \log (n/2)$
- Então verificamos se a recorrência pode ser provada!!
- Agora substituímos esta fórmula na recorrência:

$$
\begin{aligned}
T(n) &\leq 2(c * n/2 * \log(n/2)) + n \\
&\leq c * n * \log(n/2) + n \\
&= c * n * \log n - c * n * \log 2 + n \\
&= c * n * \log n - c * n + n \\
&\leq c * n * \log n,
\end{aligned}
$$

para qualquer $c \geq 1$

Método da Substituição

- Agora basta provar a Base da Indução: pela definição, T(1)=1.
- Calculando pelo nosso “chute”, $T(1) \leq c *1* \log 1$....

.....mas log 1 = 0, então nenhuma constante c satisfaz.......

- Mas se não funciona para a base, não se pode aceitar a indução.......
- ...e agora, como resolver?

Método da substituição

- Devemos lembrar que a análise assintótica apenas exige que $T(n) = c*n*lg(n)$ a partir de um $n \geq n0$, onde n0 é uma constante escolhida adequadamente!
- Se, ao invés de 1, selecionarmos a base da nossa indução como sendo n=2 e n=3 (pois só n=2 ainda dependeria de n=1....), temos:
  - $T(2) = 2\ T(1) + 2 = 4$, e
  - $T(3) = 2\ T(3/2) + 3 = 2*1+3 = 5$
- Será que o nosso "chute" consegue satisfazer estes casos base?

Método da Substituição

$T(2) \le c * 2 * \log(2)$

$T(3) \le c * 3 * \log(3)$

- Como log(2) = 1, e log(3) = 1.57, fazendo a constante $c \ge 2$, a base está verificada.
- Desta forma, conseguimos verificar (por indução) que nosso "chute" realmente satisfaz a recorrência, portanto provamos que $T(n) = O(n*\log(n))$ para esta recursão!
- Problema desta técnica: **como "acertar" o chute?**

Atividades de sala

Resolva as seguintes recorrência por substituição:

a. $T(n) = T(n-1) + \Theta(\log n)$

$T(1) = \Theta(\log 1)$