

T3. Análise do Tempo de Execução de Algoritmos Recursivos

[Copiar link](#)

Algoritmos com uma chamada recursiva

Exemplo: contagem de ocorrências num array

Relembremos a função `conta` de [+T1. Tempo de Execução de Algoritmos Iterativos](#):

```

1  int conta (int k, int v[], int N) {
2      int i = 0, r = 0;                                c1      1
3      while (i<N) {                                    c2      N+1
4          if (v[i] == k) r++;                            c3      N
5          i++;                                           c4      N
6      }
7      return r;                                         c5      1
8  }
```

É imediato escrever uma versão recursiva desta função:

```

1  int conta (int k, int v[], int N) {
2      int r;
3      if (N == 0) r = 0;
4      else {
5          r = conta(k, v+1, N-1);
6          if (v[0] == k) r++;
7      }
8      return r;
9  }
```

Note-se que a invocação `conta(k, v+1, N-1)` chama recursivamente a função, passando-lhe o array de comprimento $N-1$ com início na posição 1 do array v (i.e., a “cauda” de v).

Ao analisarmos o tempo de execução desta função deparamo-nos com uma dificuldade: a própria função que caracteriza o tempo de execução terá que ter uma definição recursiva, uma vez que $T(N)$ dependerá necessariamente de $T(N - 1)$.

A análise de algoritmos recursivos requer pois a utilização de um instrumento que permita exprimir o tempo de execução sobre um input de tamanho N em função do tempo de execução sobre inputs de tamanhos inferiores. Esse instrumento são as *equações de recorrência*, estudadas pela primeira vez por Fibonacci, no início do século XIII.

A recorrência que caracteriza o tempo de execução do algoritmo acima é a seguinte:

$$\begin{aligned} T(N) &= \Theta(1), \text{ se } N = 0 \\ T(N) &= T(N - 1) + \Theta(1), \text{ se } N > 0 \end{aligned}$$

A primeira cláusula corresponde ao caso de paragem da função, quando $N=0$, em que apenas são executadas operações de tempo constante. No caso recursivo, é feita uma invocação da função sobre um input de tamanho $N-1$, e ainda operações de tempo constante, em particular `if (v[i] == k) r++`.

Como resolver a recorrência acima? Podemos simplesmente expandir a definição:

$$\begin{aligned} T(N) &= T(N - 1) + \Theta(1) \\ &= T(N - 2) + \Theta(1) + \Theta(1) \\ &= \dots \\ &= T(0) + \Theta(1) + \dots + \Theta(1) \\ &= (N + 1) * \Theta(1) \\ &= \Theta(N) \end{aligned}$$

Como seria de esperar, o tempo de execução da versão recursiva do algoritmo de contagem de ocorrências é igual ao da versão iterativa.

Nota

Poderíamos igualmente escrever uma recorrência para o cálculo preciso (i.e. não assintótico) do número de execuções de uma determinada operação. Por exemplo a recorrência para o número de operações `v[0] == k` executadas seria:

$$T(N) = 0, \text{ se } N = 0$$

$$T(N) = T(N - 1) + 1, \text{ se } N > 0$$

com solução $T(N) = N$.

EXERCÍCIO: Apresente uma versão alternativa da função `conta`, ainda recursiva, mas utilizando um *acumulador*. Será que o tempo de execução da função que escreveu pode ser descrito pela mesma recorrência da anterior?

EXERCÍCIO: Relembre a função de identificação de duplicados num array de [+T1. Tempo de Execução de Algoritmos Iterativos](#). Escreva uma versão recursiva desta função e uma recorrência que caracterize o seu tempo de execução. Resolva essa recorrência, expandindo-a como no exemplo acima.

```

1 void dup_rec (int v[], int a, int b) {
2     if (a>=b) return;
3     for (t=a+1; t<=b ; t++)
4         if (v[a]==v[t]) printf("%d igual a %d\n", v, t);
5     dup_rec (v, a+1, b);
6 }
7
8 T(N) = 0, se N<=1
9 T(N) = N-1 + T(N-1), se N>1
10
11 T(N) = N-1 + N-2 + N-3 + .... 1 + T(1)
12      = 1 + 2 + 3 + ... + N-2 + N-1

```

$$T(N) = \sum_{k=1}^{N-1} k = \frac{(N-1)N}{2} = \theta(N^2)$$

```

1 void aux (int v[], int i, int j, int x) {

```

```
2     for (t=i; t<=j ; t++)
3         if (v[x]==v[t]) printf("%d igual a %d\n", x, t);
4     }
5
6     Taux (N) = N
7
8     void dup_rec_2 (int v[], int a, int b) {
9         if (a<b) {
10             aux(v, a+1, b, a);
11             dup_rec_2 (v, a+1, b);
12         }
13     }
14
15     T(N) = 0, se N<=1
16     T(N) = Taux(N-1) + T(N-1), se N>1
17     = N-1 + T(N-1), se N>1
```

```
1     void aux_rec (int v[], int i, int j, int x) {
2         if (i>j) return;
3         if (v[x]==v[i]) printf("%d igual a %d\n", i, t);
4         aux_rec(v, i+1, j, x);
5     }
6
7     Taux_rec(N) = 0, se N=0
8     Taux_rec(N) = 1 + Taux_rec(N-1), se N>0
9
10    Taux_rec(N) = 1+1+...+1 = N
11
12    void dup_rec_3 (int v[], int a, int b) {
```

```
13     if (a<b) {
14         aux_rec(v, a+1, b, a);
15         dup_rec_3 (v, a+1, b);
16     }
17 }
18
19 T(N) = 0, se N<=1
20 T(N) = Taux_rec(N-1) + T(N-1), se N>1
21     = N-1 + T(N-1), se N>1
```

O exemplo anterior caracteriza-se por ter o mesmo comportamento no melhor e no pior caso. Vejamos agora como analisar *algoritmos recursivos com diferentes casos* no que diz respeito ao tempo de execução.

Exemplo: procura linear num array

Relembremos o algoritmo de procura linear estudado em [+T2. Análise de Pior Caso, Melhor Caso, e Caso Médio](#):

```
1 int procura(int v[], int a, int b, int k) {
2     int i = a;
3     while ((i<=b) && (v[i]!=k))
4         i++;
5     if (i>b)
6         return -1;
7     else return i;
8 }
```

Uma versão recursiva pode ser escrita como se segue, fazendo avançar o índice inferior `a` ao longo do array.

```
1 int procura(int v[], int a, int b, int k) {  
2     if (a > b) return -1;  
3     if (v[a]==k) return a;  
4     return procura(v, a+1, b, k);  
5 }
```

Antes de mais note-se que este algoritmo recursivo tem dois casos de paragem:

- O caso $(a > b)$, que corresponde ao array vazio, e que será executado caso k não seja encontrado entre os índices a e b ;
- O caso $(v[a]==k)$, que será executado quando k é encontrado na posição a do array.

Não é possível definir uma recorrência que entre em linha de conta com este segundo caso de paragem, uma vez que ele não depende do valor de N , mas sim do conteúdo do array. É no entanto possível escrever *recorrências específicas para a análise do tempo de execução no pior e no melhor caso*.

O **pior caso** de execução acontece quando k não ocorre no array, e é caracterizado pela seguinte recorrência (o caso de paragem é o primeiro que identificámos acima):

$$T_p(N) = \Theta(1), \text{ se } N = 0$$

$$T_p(N) = T_p(N - 1) + \Theta(1), \text{ se } N > 0$$

Em que $N = b - a + 1$. Note-se que no caso recursivo é feita uma invocação da função sobre um input de tamanho $N-1$, sendo também executadas operações de tempo constante. Esta recorrência, que é igual à do primeiro exemplo que vimos, tem solução

$$T_p(N) = \Theta(N).$$

Quanto ao **melhor caso**, ele ocorre quando k se encontra na posição a do array, e neste caso a recorrência “degenera” na seguinte definição não-recorrente, uma vez que o caso de paragem é imediatamente executado:

$$T_m(N) = \Theta(1)$$

A análise da versão recursiva revela que o algoritmo executa em tempo constante no melhor caso e linear no pior, tal como a versão iterativa.

Algoritmos de Divisão e Conquista

Os exemplos que vimos anteriormente são algoritmos muito simples que admitem versões iterativas e recursivas. No entanto, a recursividade é um instrumento fundamental na definição de algoritmos baseados numa estratégia algorítmica específica: os algoritmos de **divisão e conquista**. Trata-se aqui de algoritmos cuja definição sem a utilização de recursividade não é de todo trivial.

Um algoritmo recursivo de divisão e conquista tem a seguinte estrutura típica:

1. **Divisão** do problema em n sub-problemas
2. Resolução recursiva dos n sub-problemas (passo de **Conquista**)
3. **Combinação** das soluções dos sub-problemas para obter a solução do problema inicial

O caso de paragem ocorre normalmente no caso de inputs muito pequenos.

Em geral, o tempo de execução de um algoritmo de divisão e conquista será caracterizado por uma recorrência com a seguinte forma:

$$T(N) = \Theta(1), \text{ se } N \leq k$$

$$T(N) = D(N) + aT(N/a) + C(N), \text{ se } N > k$$

Em que cada *divisão* gera a sub-problemas, sendo o tamanho de cada sub-problema uma fracção $1/a$ do original (ou próximo disso); k é o tamanho dos problemas com solução trivial; e D e C são funções que caracterizam o tempo das operações de *divisão* e *combinação*, respectivamente.

Exemplo: algoritmo Merge Sort

Neste algoritmo de ordenação bem conhecido a estrutura de divisão e conquista é instanciada da seguinte forma:

1. **Divisão** do array em duas partes de tamanho igual (a menos de uma unidade, no caso de o comprimento ser ímpar)
2. **Conquista**: ordenação recursiva dos dois vectores
3. **Combinação**: fusão dos dois vectores ordenados.

Este último passo será implementado por uma função auxiliar `merge`. A definição seguinte assume que são declarados globalmente dois arrays auxiliares L e R com tamanho suficiente para armazenar temporariamente os elementos das duas partes ordenadas de A que se pretende fundir. Será colocada uma *sentinela* (o valor do maior inteiro representável) no final dos arrays L e R, o que simplificará o algoritmo.

A primeira parte ordenada de A está contida entre os índices p e q; a segunda está contida entre os índices q+1 e r.

```
1 void merge(int A[], int p, int q, int r) {
2     int n1 = q-p+1, n2 = r-q;
3     for (i=0 ; i<n1 ; i++) L[i] = A[p+i];      // L[], R[] globais
4     for (j=0 ; j<n2 ; j++) R[j] = A[q+j+1];
5     L[n1] = INT_MAX; R[n2] = INT_MAX;
6
7     i = 0; j = 0;
8     for (k=p ; k<=r ; k++)
9         if (L[i] <= R[j]) {
10             A[k] = L[i]; i++;
11         } else {
12             A[k] = R[j]; j++;
13         }
14 }
```

O passo básico do algoritmo compara dois valores contidos nas primeiras posições de ambos os arrays, colocando o menor dos valores no array resultado. Este passo é executado em *tempo constante*, e são sempre executados N passos básicos, com $N = r - p + 1$. O algoritmo executa então em tempo $C(N) = \Theta(N)$.

Depois de definida a função de fusão ordenada, o algoritmo de ordenação é facilmente implementado através de uma função recursiva como a seguinte, que ordena o array entre os índices p e r .

```
1 void merge_sort(int A[], int p, int r) {  
2     if (p < r) {  
3         q = (p+r)/2;  
4         merge_sort(A, p, q);  
5         merge_sort(A, q+1, r);  
6         merge(A, p, q, r);  
7     }  
8 }
```

Sendo a invocação inicial, para ordenar um array de comprimento N ,

`merge_sort(A,0,N-1)`.

O tempo de execução pode ser caracterizado pela seguinte recorrência. Note-se a utilização dos operadores de arredondamento para captar o efeito da divisão inteira

`q = (p+r)/2`.

$$T(N) = \Theta(1), \text{ se } N = 1$$

$$T(N) = T\lceil N/2 \rceil + T\lfloor N/2 \rfloor + \Theta(N), \text{ se } N > 1$$

em que o termo $\Theta(N)$ corresponde ao tempo de execução da função de fusão ordenada.

Mais uma vez resolveremos a recorrência começando por expandi-la. Mas vamos para isso considerar uma simplificação que permitirá simplificar a sua análise: admitiremos que N é uma potência de 2, o que significa que todos os valores tomados por esta variável na expansão da recorrência serão pares, e por essa razão os operadores de arredondamento podem ser dispensados. Além disso, escreveremos sem perda de generalidade o termo de tempo linear como cN e o termo constante como c .

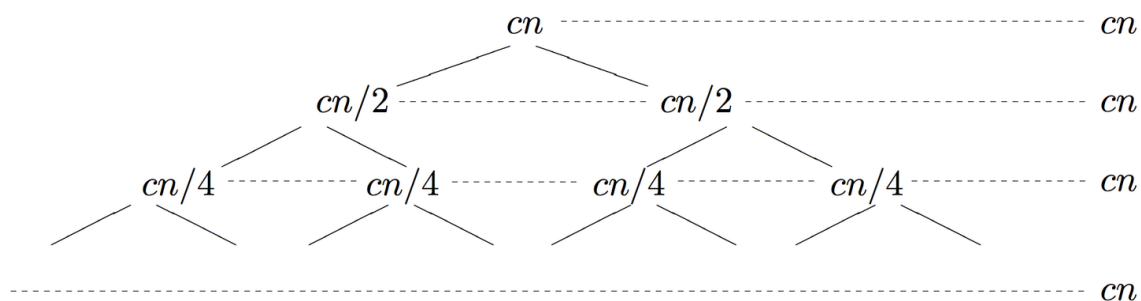
$$T(N) = c, \text{ se } N = 1$$

$$T(N) = 2T(N/2) + cN, \text{ se } N > 1$$

Efectuemos então a expansão:

$$\begin{aligned} T(N) &= 2T(N/2) + cN \\ &= 4T(N/4) + 2c(N/2) + cN = 4T(N/4) + 2cN \\ &= 8T(N/8) + 3cN \\ &= \dots \\ &= NT(1) + (\log N)cN \\ &= Nc + (\log N)cN \\ &= cN \log N + cN \\ &= \Theta(N \log N) \end{aligned}$$

Visualmente podemos compreender a execução do algoritmo através de uma árvore de recursividade em que todos os níveis contribuem para o tempo de execução com o mesmo custo cN , tendo a árvore $\log N + 1$ níveis (o último dos quais corresponde à execução dos casos de paragem, sobre os arrays singulares).



Em conclusão, o algoritmo merge sort executa em tempo $T(N) = \Theta(N \log N)$, sem distinção de casos.

Métodos de resolução de recorrências

Não existe um método universal para resolução de recorrências. No entanto, muitas das recorrências que caracterizam o tempo de execução de algoritmos de divisão e

conquista podem ser resolvidas pelo chamado *Master theorem*, que não estudaremos aqui.

https://en.wikipedia.org/wiki/Master_theorem

Adicionalmente, é possível *provar* indutivamente que uma determinada pseudo-solução (por exemplo calculada informalmente por expansão da recorrência, como nos exemplos anteriores) é de facto uma verdadeira solução de uma recorrência. O método de prova que se utiliza para isto designa-se por *método de substituição*, e veremos um exemplo da sua aplicação em [+T4. Tópicos sobre Algoritmos de Ordenação \(?\)](#).

Exercícios Adicionais

1. Para cada uma das seguintes recorrências, apresente um exemplo de um algoritmo cujo tempo de execução seja caracterizado por ela, e resolva-a. k é uma constante; assuma também que $T(1)$ é constante (caso de paragem).
 - a. $T(N) = k + T(N - 1)$
 - b. $T(N) = N + T(N - 1)$
 - c. $T(N) = k + T(N/2)$
 - d. $T(N) = k + 2 * T(N/2)$
 - e. $T(N) = N + 2 * T(N/2)$
 - f. $T(N) = N + T(N/2)$

```
1 int procuraBin (int v[], int a, int b, int k) {
2     if (a>b) result = -1;
3     else {
4         m = a + (b-a) / 2;
5         if (vector[m] < k) result = procuraBin (v, m+1, b, k);
6         else if (vector[m] > k) result = procuraBin(v, a, m-1, k);
7         else result = m;
```

```

8     }
9     return result;
10  }

```

$$T(16) = k + T(8) = 2k + T(4) = 3k + T(2) = 4k + T(1) = 4k$$

$$T(N) = \sum_{i=1}^{\log N} k = \theta(\log N)$$

2. Considere o seguinte algoritmo para o problema conhecido por *Torres de Hanói*:

```

1 void Hanoi(int nDiscos, int esquerda, int direita, int meio)
2 {
3     if (nDiscos > 0) {
4         Hanoi(nDiscos-1, esquerda, meio, direita);
5         printf("mover disco de %d para %d\n", esquerda, direita);
6         Hanoi(nDiscos-1, meio, direita, esquerda);
7     }
8 }

```

- Escreva uma relação de recorrência que expresse a complexidade deste algoritmo (por exemplo, em função do número de linhas impressas).
- Desenhe a árvore de recursão do algoritmo e obtenha a partir dessa árvore um resultado sobre a sua complexidade assintótica.

3. Considere a seguinte versão recursiva do algoritmo *insertion sort*.

```

1 void isort (int v[], int N) {
2     int i; int t;
3     if (N>1) {
4         isort (v+1, N-1);
5         i = 0;

```

```

6      t = v[0];
7      while (i < N-1 && v[i] < t) {
8          v[i] = v[i+1];
9          i++;
10     }
11     if (i > 0) v[i] = t;
12 }
13 }

```

- Identifique o melhor e pior casos de execução desta função.
- Para esses casos, apresente uma relação de recorrência que traduza o *número de comparações entre elementos do vector* em função do tamanho do vector.

$$T_{mc}(N) = 1 + T(N - 1) = N - 1$$

$$T_{pc}(N) = N - 1 + T(N - 1) = \frac{(N-1)N}{2}$$

- Considere o seguinte algoritmo para o cálculo dos números de Fibonacci.
Assuma que as operações aritméticas elementares se efectuam em tempo $\Theta(1)$.

```

1  int fib (int n)
2  {
3      if (n==0 || n==1) return 1;
4      else return fib(n-1) + fib(n-2);
5  }

```

- Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.
- Efectue uma análise assintótica do tempo de execução deste algoritmo.
- Apesar de traduzir exactamente a definição da sequência de números de Fibonacci, este algoritmo é muito ineficiente, como deverá ter concluído na alínea anterior. Escreva em C um algoritmo alternativo eficiente e analise o seu tempo de execução.



Criado com o Dropbox Paper. [Saiba mais](#)