T3. Análise do Tempo de Execução de Algoritmos Recursivos

Algoritmos com uma chamada recursiva

Exemplo: contagem de ocorrências num *array*

Relembremos a função `conta` de +T1. Tempo de Execução de Algoritmos Iterativos:

```
int conta (int k, int v[], int N) {
    int i = 0, r = 0;
    while (i<N) {
      if (v[i] == k) r++;
      i++;
    }
    return r;
}
```

É imediato escrever uma versão recursiva desta função:

```
int conta (int k, int v[], int N) {
    int r;
    if (N == 0) r = 0;
    else {
      r = conta(k, v+1, N-1);
      if (v[0] == k) r++;
    }
    return r;
}
```

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, contando o número de comparações `v[0] == k` executadas, é a seguinte:

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

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 uma comparação,
`if (v[i] == k) r++`.

Como resolver a recorrência acima? Podemos simplesmente expandir a definição:
$$\begin{aligned} T(N) &= T(N-1) + 1 \\ &= T(N-2) + 1 + 1 \\ &= \cdots \\ &= T(0) + 1 + \ldots + 1 \\ &= 0 + N * 1 \\ &= N \end{aligned}$$

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 tempo assimptótico:

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

que teria naturalmente solução $T(N) = \Theta(N)$.

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.

```
void dup_rec (int v[], int a, int b) {
    if (a>=b) return;
    for (t=a+1; t<=b ; t++)
       if (v[a]==v[t]) printf("%d igual a %d\n", v, t);
    dup_rec (v, a+1, b);
}


T(N) = 0, se N<=1
T(N) = N-1 + T(N-1), se N>1


T(N) = N-1 + N-2 + N-3 + .... 1 + T(1)
     = 1 + 2 + 3 + ... + N-2 + N-1
```

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

```
void aux (int v[], int i, int j, int x) {
    for (t=i; t<=j ; t++)
       if (v[x]==v[t]) printf("%d igual a %d\n", x, t);
}

Taux (N) = N

void dup_rec_2 (int v[], int a, int b) {
    if (a<b) {
      aux(v, a+1, b, a);
      dup_rec_2 (v, a+1, b);
    }
}

T(N) = 0, se N<=1
T(N) = Taux(N-1) + T(N-1), se N>1
     = N-1 + T(N-1), se N>1
```

```
void aux_rec (int v[], int i, int j, int x) {
    if (i>j) return;
    if (v[x]==v[i]) printf("%d igual a %d\n", i, t);
    aux_rec(v, i+1, j, x);
}

Taux_rec(N) = 0, se N=0
Taux_rec(N) = 1 + Taux_rec(N-1), se N>0
```

```
Taux_rec(N) = 1+1+...+1 = N

void dup_rec_3 (int v[], int a, int b) {
    if (a<b) {
      aux_rec(v, a+1, b, a);
      dup_rec_3 (v, a+1, b);
    }
}

T(N) = 0, se N<=1
T(N) = Taux_rec(N-1) + T(N-1), se N>1
     = 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:

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

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

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

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*.

Contemos o número de comparações `(v[a]==k)` avaliadas. 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):

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

Em que $N = b - a + 1$. Note-se que no caso recursivo é executada uma comparação (que falha) e é feita depois uma invocação da função sobre um input de tamanho N-1. Esta recorrência, que é igual à do primeiro exemplo que vimos, tem solução
$T_p(N) = N$.

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 um caso de paragem é imediatamente executado (a comparação executada tem sucesso):

$$T_m(N) = 1$$

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.

```
void merge(int A[], int p, int q, int r) {
  int n1 = q-p+1, n2 = r-q;
  for (i=0 ; i<n1 ; i++)  L[i] = A[p+i];     // L[], R[] globa
is
  for (j=0 ; j<n2 ; j++)  R[j] = A[q+j+1];
  L[n1] = INT_MAX; R[n2] = INT_MAX;

  i = 0; j = 0;
  for (k=p ; k<=r ; k++)
    if (L[i] <= R[j]) {
      A[k] = L[i]; i++;
    } else {
      A[k] = R[j]; j++;
    }
}
```

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.

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

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.

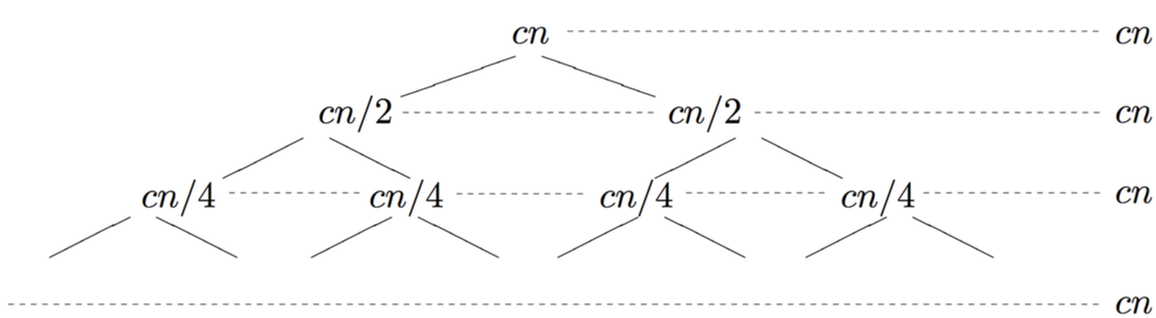
Mais uma vez resolveremos a recorrência começando por expandi-la. Mas vamos para isso considerar uma simplificação que permitirá simplificar a sua análise: admitiremos que N é uma potência de 2, o que significa que todos os valores tomados por esta variável na expansão da recorrência serão pares, e por 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:

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

Visualmente podemos comprender 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. $T(N) = N + T(N/2)$

```
int procuraBin (int v[], int a, int b, int k) {
  if (a>b) result = -1;
  else {
    m = a + (b-a) / 2;
```

```
    if (vector[m] < k) result = procuraBin (v, m+1, b, k);
    else if (vector[m] > k) result = procuraBin(v, a, m-1, k);
         else result = m;
    }
  return result;
}
```

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*:

```
void Hanoi(int nDiscos, int esquerda, int direita, int meio)
{
  if (nDiscos > 0) {
    Hanoi(nDiscos-1, esquerda, meio, direita);
    printf("mover disco de %d para %d\n", esquerda, direita);
    Hanoi(nDiscos-1, meio, direita, esquerda);
  }
}
```

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

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

```
void isort (int v[], int N) {
  int i; int t;
```

```
  if (N>1) {
    isort (v+1, N-1);
    i = 0;
    t = v[0];
    while (i<N-1 && v[i]<t) {
      v[i] = v[i+1];
      i++;
    }
    if (i>0) v[i] = t;
  }
}
```

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

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

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

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

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

a. Escreva uma recorrência que descreva o comportamento temporal do algoritmo. Desenhe a respectiva árvore de recursão para $n = 5$.
b. Efectue uma análise assimptótica do tempo de execução deste algoritmo.

c. 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.