Notas sobre Correção do MergeSort e do InsertionSort

1 Introdução

Quando pretendemos analisar um algoritmo, começamos por colocar duas questões essenciais:

1. Está correto?
2. O que podemos esperar em termos de desempenho computacional?

Depois desta análise, é frequente surgir uma terceira questão (nesta disciplina, vai surgir quase sempre):

3. Podemos fazer melhor?

Estas notas pretendem mostrar como podemos, de um modo formal, responder à primeira questão para alguns algoritmos utilizados nas aulas TP. Para isso, vamos usar como estudo de caso dois algoritmos de ordenamentos utilizados nas aulas desta semana. Começamos com o método da Inserção Linear (*InsertionSort*) e continuamos com o método do *MergeSort*. Este último vai também auxiliar a mostrar mais um exemplo da estratégia de desenho de algoritmos conhecida por "*dividir-e-conquistar*" (*Divide and Conquer*).

2 Inserção Linear - InsertionSort

No exercício pré-aula são apresentados dois modos diferentes (entre os muitos que existem) para a implementação do método da Inserção Linear em código `Python`. Um deles foi a implementação (direta) do algoritmo dos slides da aula:

```
Insertionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada
    for j = 2, …, n:
        este = a[j]
        i = j-1
       while i > 0 and a[i] > este:
           a[i+1] = a[i]
           i = i-1
       endwhile
       a[i+1] = este
   endfor
```

Comecemos por analisar a questão da correção deste algoritmo.

2.1 Correção do método de Inserção Linear

Após compreender como trabalha o *InsertionSort* (podem ver o exemplo nos slides para perceber o que está a acontecer em cada iteração), pensamos imediatamente que o algoritmo está "obviamente" correto. No entanto, se não soubermos como funciona e só tivermos o pseudocódigo (ou um código), a correção não é assim tão óbvia. Muitos dos algoritmos que vamos estudar, *nem sempre é óbvio* como e porque funciona, pelo que temos de mostrar que o algoritmo devolve a resposta correta para qualquer input.

Vamos começar com uma prova de correção para o algoritmo do *InsertionSort*. A nossa prova vai assentar na noção, que já conhecem, de *invariante de ciclo* (*loop invariant*). Recordando, esta é uma propriedade que vai permitir construir, passo a passo (iterativamente ou em cada iteração do corpo do ciclo), a solução procurada. Diz-se invariante porque, para as iterações passadas é válida e as iterações futuras não alteram a propriedade, mas apenas aumentam a dimensão (complexidade) da solução. Posto de outro modo, é válida antes e à saída do ciclo. No entanto, não vamos usar explicitamente o ciclo para validar a propriedade, mas antes mostrar a sua validade usando o *Método de Indução*.

Neste caso, a nossa propriedade (invariante) é que, após a iteração *i*, a sequência `A[1 .. i+1]` está ordenada.

Tal propriedade é óbvia quando $i = 1$ (porque a sequência composta por um único elemento $A[1]$ está, claramente, ordenada). Vamos mostrar que, para qualquer $i > 1$, se a propriedade é verdadeira para $i - 1$, então vai ser válida para *i* (isto é, que a propriedade é *hereditária*). Sendo assim, e se é verdade para i=1 e é hereditária, então, é verdade para qualquer i, em particular, para i=n e, portanto a sequência `A[1 .. n]` está ordenada.

- **Hipótese de Indução (H.I.):** Se A[1..i-1] está ordenada, então, após a iteração *i* do ciclo exterior, A[1..i] está ordenada.

- **Caso base:** Quando $i = 1$, A[1] contém um só elemento, logo, está ordenada.

- **Passo de Indução:** Assumimos que A[1..i-1] está ordenada. Queremos agora mostrar que A[1..i-1**+1**] está ordenada após a *i*-ésima iteração.

  Suponha-se que j^* é o maior inteiro em $\{0,...,i-1\}$ tal que A[j*] < A[i]. Neste caso, é accionado o ciclo interior que vai pegar em A[i]

  $$A[1] ,..., A[j^*],..., A[i-1], A[i]$$

  e colocar em ordem:

  $$,A[1], ...,A[j^*], A[i], A[j^*+1], ..., A[i-1].$$

  Porque $A[i] > A[j^*]$, usando a H.I. temos que $A[j^*] \geq A[j]$ para todo o $j \leq j^*$, e, portanto, $A[i]$ é maior que todos os elementos ({0, ..., j*}). De modo semelhante, pela escolha de j^* temos $A[i] \leq A[j^*+1] \leq A[j]$ para todo o $j \geq j^*+1$. Portanto, $A[i]$ está colocado no local correto. Como todos os restantes elementos estavam, por hipótese, já ordenados, após a iteração *i*, `A[1..i]` está ordenado.

- **Conclusão:** Usando indução, concluímos que a hipótese indutiva é válida para todo o *i*. Isto implica que após a $(n-1)$-iésima iteração, temos que A[1..n] está ordenado.

Reforçamos que a propriedade do algoritmo que acabámos de mostrar ser verdadeira para qualquer dimensão n, é "óbvia". No entanto, precisamente por ser óbvia, este exercício é importante para que se compreenda a estrutura do argumento utilizado, que iremos usar frequentemente nesta unidade curricular.

2.2 Estratégia algorítmica "Diminuir e conquistar"

A nível de estratégia geral de desenho de algoritmos, o método Inserção Linear, de algum modo, também tenta simplificar tarefa a resolver, mas usando um processo de resolução incremental – estratégia incremental – que é conhecido por "Diminuir e conquistar" (*Decrease and conquer*). Em cada iteração, é colocado um elemento **em ordem** na subsequência ordenada (conquistar) e removido (decrementado) um elemento da parte ainda não ordenada (diminuir) de acordo com o esquema no exemplo na figura seguinte:

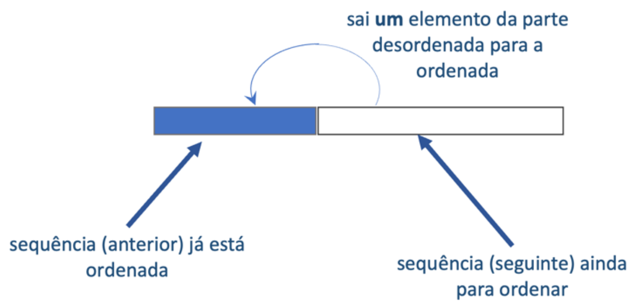

Figura 1-Esquema de funcionamento genérico de um exemplo "diminuir para conquistar".

[Nos slides da aula ainda é realçada a estratégia do tipo "pesquisa" exaustiva ou "método da força bruta".]

3 MergeSort

Já o algoritmo MergeSort serve como um bom exemplo para o paradigma *Dividir e conquistar* (*Divide-and-conquer*) sendo, sem dúvida, uma ideia interessante em termos de algoritmo para resolver o problema do ordenamento. O seu pseudocódigo pode ser o seguinte:

```
mergesort(a, ini, fim):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e posições inicial e final da sequência
    //Output: sequência ordenada
    if ini < fim:
        meio = maior inteiro contido em (ini+fim/2)
        mergesort(a, ini, meio)
        mergesort(a, meio+1,  fim)
        merge(a, ini, meio, fim)
   endif
```

O módulo `merge()` recebe (indicações) para uma subsequência esquerda, *L=A[ini..meio],* e uma subsequência direita *R=A[meio+1..fim]*, ambas *já ordenadas* , e funde as duas em ordem, ou seja, devolve uma sequência com todos os elementos indicados por ordem crescente (ver os slides da aula).

```
merge(a, ini, meio, fim):
    // fusão de duas sequências ordenadas: L = a[ini..meio] e R = a[meio+1..fim]
       i ← ini; j ← meio+1
       for k = ini … fim:
```

```
        if j > fim: aux[k] ← a[i]; i←i+1
        else if i > meio: aux[k] ← a[j]; j←j+1
        else if a[i] < a[j]: aux[k] ← a[i]; i←i+1
        else:  aux[k] ← a[j]; j←j+1
    for k = ini … fim:
          a[k] ← aux[k]
```

3.1 Correção do *MergeSort*

Tal como no caso do *Inserção Linear*, vamos usar o Método de Indução para mostrar que o algoritmo devolve sempre a resposta desejada, qualquer que seja o *input*. A nova **propriedade invariante** (recursiva) vai ser que, no final de uma chamada recursiva, o *MergeSort* devolve sempre uma sequência ordenada.

3.1.1 Correção da função merge

Devemos começar por procurar a propriedade invariante do ciclo `for` do algoritmo do *merge*. Como o for vai construindo (ver ilustração nos slides da aula) uma sequência auxiliar - `aux`, passo a passo (ou seja, para k a variar de `ini` a `fim`), temos que o nosso invariante do ciclo será:

$$aux[k] \leq a[t],\ t \in \{i, \ldots, meio\} \quad \text{e} \quad aux[k] \leq a[t],\ t \in \{j, \ldots, fim\}.$$

Ou seja, o elemento que é copiado para a posição k de aux é sempre o menor (mínimo) de todos os que **ainda não foram copiados.** Quando uma das partes terminar (ou o lado esquerdo foi todo copiado ou o lado direito foi todo copiado), os elementos ainda não copiados são:

- maiores que todos os que já foram copiados
- continuam a estar ordenado por ordem crescente
- são copiados nessa mesma ordem

Donde, no final do *merge* temos que:

$$a[k] \leq a[k+1],\ \forall k \in \{ini, \ldots, fim\}.$$

3.1.2 Correção do Mergesort por indução

- **Hipótese de Indução (H.I.):** Sempre que o MergeSort devolve uma sequência de tamanho i, a sequência está ordenada por ordem crescente.

- **Caso base:** Consideremos $i = 1$. Se o MergeSort devolve uma sequência de tamanho $0 \leq k \leq 1$, a sequência está ordenada, uma vez que toda a sequência de tamanho 0 ou 1, por definição, está ordenada.

- **Passo de Indução:** Consideremos que temos duas sequências obtidas no final de duas chamadas recursivas ao MergeSort, L and R, de tamanho $\lfloor i/2 \rfloor$. Como $\lfloor i/2 \rfloor \leq i$, pela hipótese de indução, ambas estão ordenadas por ordem crescente. Usando o merge, e como acabámos de provar, obtemos uma sequência ordenada de tamanho $length(L) + length(R) = \lfloor i/2 \rfloor + \lfloor i/2 \rfloor = i$ e, portanto, o Mergesort devolve uma sequência totalmente ordenada de tamanho i.

- **Conclusão:** A hipótese indutiva é válida para qualquer $i \geq 0$. Em particular, dada uma sequência de tamanho $i = n$, o MergeSort devolve uma permutação ordenada dessa sequência.

3.2 Ordem de desempenho do MergeSort

Nas próximas aulas iremos mostrar que a ordem de complexidade do Mergesort é $O(n \log n)$.