CES-11

Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

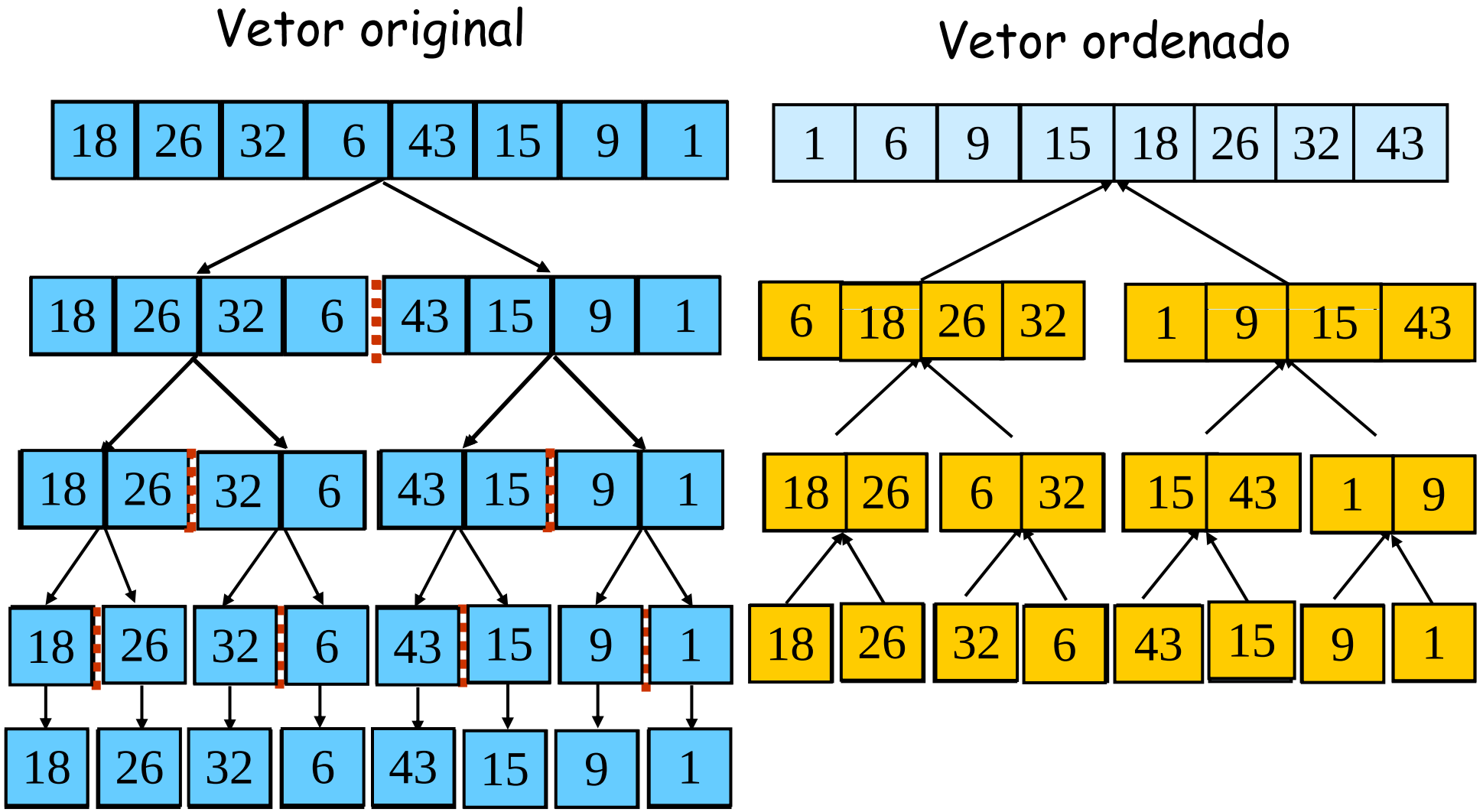
CES-11

- Algoritmos de Ordenação
  - *MergeSort*
  - *QuickSort*

MergeSort (Von Neumann, 1945)

- Este algoritmo é um exemplo do paradigma *Divisão-e-Conquista*, e por isso tem 3 fases:
  - Divisão: o vetor é dividido em duas metades;
  - Conquista: cada metade é ordenada recursivamente, dando origem a duas subsoluções;
  - Combinação: essas subsoluções são combinadas, formando a solução final.
- Condição de parada da recursão: quando for ordenar apenas um elemento. Este caso será a subsolução elementar.

Exemplo para n=8



Algoritmo

```
MergeSort(v, aux, i, f) {
  if (i < f) {
    m = ⌊(i+f)/2⌋;
    MergeSort(v, aux, i, m);
    MergeSort(v, aux, m+1, f);
    merge(v, aux, i, f);
  }
}
```

Chamada inicial:

```
MergeSort(vet, vaux, 1, n)
```

Convém que o vetor **vaux** seja alocado junto com o vetor **vet**

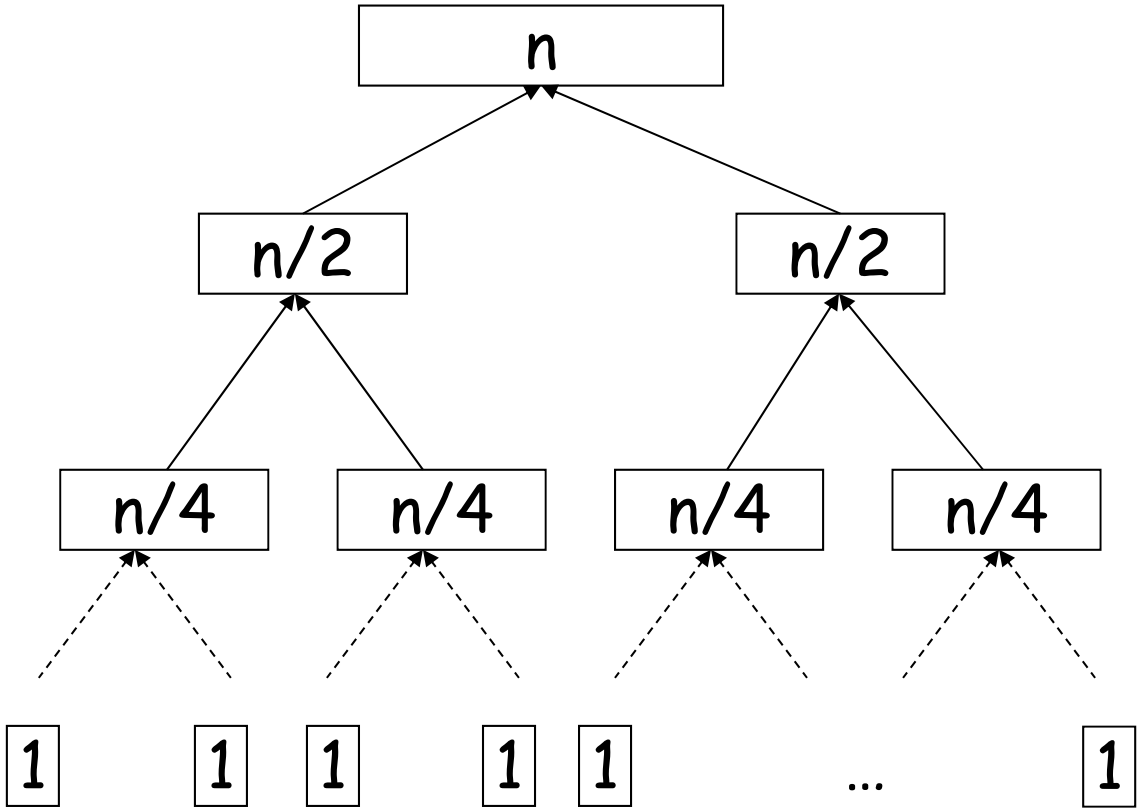
```
merge(v, aux, i, f) {
    m = ⌊(i+f)/2⌋;
    i1 = i;
    i2 = i;
    i3 = m+1;
    while (i2 <= m && i3 <= f)
        if (v[i2] < v[i3])
            aux[i1++] = v[i2++];
        else
            aux[i1++] = v[i3++];
    while (i2 <= m)
        aux[i1++] = v[i2++];
    while (i3 <= f)
        aux[i1++] = v[i3++];
    for (j=i; j<=f; j++)
        v[j] = aux[j];
}
```

Complexidade de tempo de `merge`: $O(f-i)$

Complexidade de tempo do *MergeSort*

- $T(1) = 1$ (tempo constante)
- $T(n) = 2T(n/2) + n,\ n>1$
- Supondo $n = 2^k$:
  - $T(n) = 2(2T(n/2^2) + n/2) + n = 2^2T(n/2^2) + 2n$
  - $T(n) = 2^2(2T(n/2^3) + n/2^2) + 2n = 2^3T(n/2^3) + 3n$
  - Generalizando: $T(n) = 2^kT(n/2^k) + kn$
- Substituindo $n = 2^k$:
  - $T(n) = n + n.\lg n$
  - $T(n) = O(n.\log n)$
- A generalização para n qualquer não muda a ordem de T(n).

Outro modo de calcular o tempo



$$
\begin{aligned}
& n \\
& + \\
2.(n/2) &= n \\
& + \\
4.(n/4) &= n \\
& + \\
& \cdots \\
& + \\
n.(1) &= n \\
\hline
& n.(1 + \lg n)
\end{aligned}
$$

Tempo total: $\Theta(n.\log n)$

Conclusões

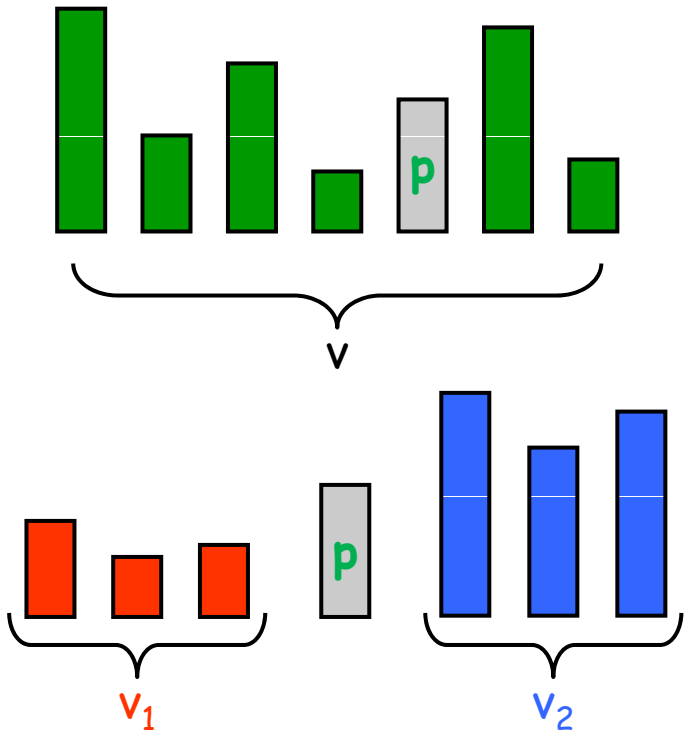
- *MergeSort* necessita de espaço linear para armazenar o vetor temporário (convém que seja alocado uma única vez, fora da função recursiva).
- É possível escrever uma versão do *MergeSort* não recursiva e sem pilha, cuja execução é mais rápida.
- Seu tempo de execução é da mesma ordem do *lower bound* do problema, isto é, O(n.log n).
- Como o *upper* e o *lower bounds* da ordenação são de mesma ordem, podemos dizer que:
  - Qualquer algoritmo que realiza comparações e ordena n números em tempo O(n.log n) é ótimo.
  - A ordenação através de comparações é um problema computacionalmente resolvido.

CES-11

- Algoritmos de Ordenação
  - *MergeSort*
  - *QuickSort*

QuickSort (Hoare, 1961)

- Na prática, *QuickSort* é o algoritmo de ordenação mais rápido.
- Também segue o paradigma da *Divisão-e-Conquista*.
- Divisão:
  1) Escolha um elemento p para ser o pivô em v.
  2) Particione o vetor v – {p} em dois grupos distintos:
     - $v_1 = \{x \in v - \{p\} \mid x < p\}$
     - $v_2 = \{x \in v - \{p\} \mid x \geq p\}$
- Conquista: ordene recursivamente v_1 e v_2.
- Combinação: junte v_1, p e v_2 para obter v ordenado.



Algoritmo básico para o *QuickSort*

```
QuickSort(v, min, max) {
    if (min < max) {
        p = Partition(v, min, max);
        Quicksort(v, min, p-1);
        Quicksort(v, p+1, max);
    }
}
```

- Pontos-chave: a escolha do pivô e o algoritmo de particionamento.
- Há várias técnicas eficientes.

Um possível particionamento

- Escolha como pivô o primeiro elemento do vetor.
- Começando da esquerda, encontre o primeiro elemento do vetor igual ou maior que o pivô (ou seja, um valor que deverá ir para o lado direito do vetor).
- Vindo do direita, encontre o primeiro elemento menor que o pivô (idem: deverá ir para o lado esquerdo).
- Troque esses dois elementos.
- Continue o mesmo procedimento até que os pontos de busca se encontrem em alguma posição do vetor.
- No final, troque o pivô com o último valor encontrado vindo da direita.

Exemplo

- Escolha do pivô: 4 3 6 9 2 4 3 1 2 1 4 9 3 5 6
- Busca: 4 3 6 9 2 4 3 1 2 1 4 9 3 5 6
- Troca: 4 3 3 9 2 4 3 1 2 1 4 9 6 5 6
- Busca: 4 3 3 9 2 4 3 1 2 1 4 9 6 5 6
- Troca: 4 3 3 1 2 4 3 1 2 9 4 9 6 5 6
- Busca: 4 3 3 1 2 4 3 1 2 9 4 9 6 5 6
- Troca: 4 3 3 1 2 2 3 1 4 9 4 9 6 5 6
- Busca: 4 3 3 1 2 2 3 1 4 9 4 9 6 5 6
- Troca com o pivô: 1 3 3 1 2 2 3 4 4 9 4 9 6 5 6

Posição do pivô

Algoritmo para particionamento

```
int Partition(v, left, right) {
    pivot = v[left];
    l = left + 1;
    r = right;
    cont = true;
    while (cont) {
        while (l < right && v[l] < pivot) l++;
        while (r > left && v[r] >= pivot) r--;
        if (l >= r)
           cont = false;
        else {
           aux = v[l];
           v[l] = v[r];
           v[r] = aux;
        }
    }
    v[left] = v[r];
    v[r] = pivot;
    return r;
}
```

Tempo: $O(n)$

Mostrar animação
(o pivô será o último elemento)

Análise de tempo do *QuickSort*

- T(n): tempo do *QuickSort* para ordenar v[1..n]
- $T(n) = c.n + T(i) + T(n-i-1)$, onde $0 \leq i < n$ (obs.: $i = |v_1|$)
- Melhor caso: $i = n/2$ (balanceamento perfeito)
  - $T(n) = T(n/2) + T((n/2)-1) + c.n \approx 2T(n/2) + c.n$
  - $T(n) = O(n.\log n)$
- Pior caso: $i = 0$ ou $i = n-1$
  - $T(n) = T(n-1) + c.n$
  - $T(n) = O(n^2)$
- No pior caso, o *QuickSort* é quadrático!!

Casos práticos

- Embora haja casos em que o desempenho do *QuickSort* seja quadrático, seu tempo é usualmente O(n.log n).
- Além disso, as constantes são tão boas que o *QuickSort* é o melhor algoritmo de ordenação conhecido.
- No mundo real, a grande maioria das ordenações é realizada através deste algoritmo, principalmente quando n é grande.
- Para se encontrar um particionamento ótimo, seria preciso escolher a mediana como pivô. E para encontrar a mediana, seria necessário ordenar o vetor...
- Na verdade, é possível encontrar a mediana em tempo O(n), mas com um algoritmo nada trivial. Isso garantiria sempre tempo O(n.log n).

Mediana de três

- Uma alternativa é encontrar a chamada *mediana de três*.
- Comparam-se três elementos do vetor: o primeiro, o central e o último:
  - O pivô será a mediana entre os três.
  - Este valor é trocado com o que estava na posição inicial e o particionamento é feito do mesmo modo anterior.
- Quando esta técnica é utilizada, tornam-se muito raros os casos em que o *QuickSort* gasta tempo quadrático.

Pilha de execução

- Nos piores casos do *QuickSort* (vetor ordenado, por exemplo), devido às chamadas recursivas, a pilha de execução chega a exigir espaço O(n).
  - Isso ocorre porque pode haver até n recursões pendentes.
- Dependendo do tamanho do vetor, esse espaço pode se esgotar, e o programa será abortado...
- Através de uma pequena alteração no código do *QuickSort*, é possível eliminar uma das chamadas recursivas:
  - A ideia é chamar a recursão apenas na menor metade de cada subvetor.
  - Desse modo, cada subvetor na pilha de execução será menor que a metade do subvetor imediatamente abaixo.
  - Isso garante que a altura da pilha de execução não ultrapasse lg n.

QuickSort com uma única recursão

```
QuickSort(v, min, max) {
    while (min < max) {
        p = Partition(v, min, max);
        if (p-min < max-p) {
            QuickSort(v, min, p-1);
            min = p+1;
        }
        else {
            QuickSort(v, p+1, max);
            max = p-1;
        }
    }
}
```

- O tamanho da pilha de execução passa a ser $O(\log n)$: o caso que consome mais espaço é aquele em que o vetor é sempre quebrado em duas metades iguais, gastando tempo $O(n.\log n)$.
- Por outro lado, nos casos em que o algoritmo gasta tempo $O(n^2)$, a pilha de execução cresce $O(1)$.