CES-11

Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

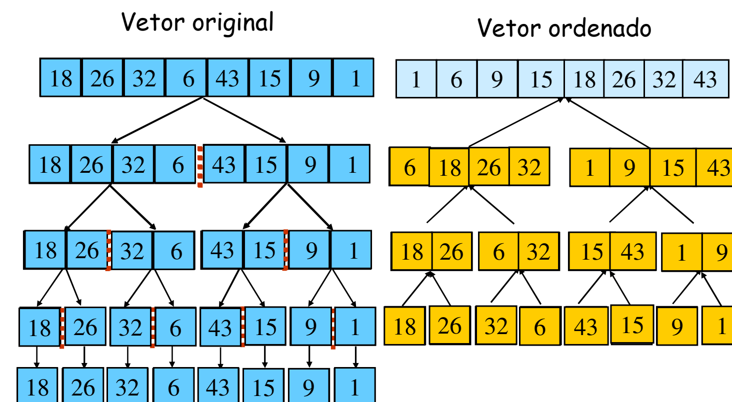
CES-11

- 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 sub-soluções;
  - Combinação: essas sub-soluções são combinadas, formando a solução final.
- Condição de parada da recursão: quando for ordenar apenas um elemento. Este caso será a sub-solução elementar.

Exemplo para n=8



Algoritmo

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

Chamada inicial:

```
MergeSort(1, n)
```

Vetor aux deve ser alocado fora!

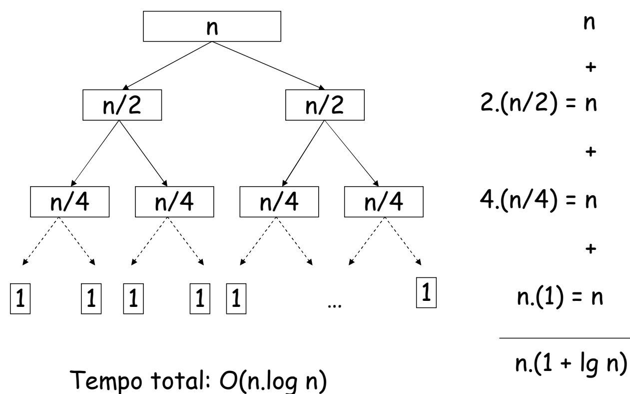
```
merge(i, f) {
   m = (i+f)/2;
   i1 = i;
   i2 = i;
   i3 = m+1;
   while (i2 <= m && i3 <= f)
       if (v[i2] < v[i3])
          aux[i1++] = v[i2++];
       else
          aux[i1++] = v[i3++];
   while (i2 <= m)
       aux[i1++] = v[i2++];
   while (i3 <= f)
       aux[i1++] = v[i3++];
   for (j=i; 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 $n = 2^k$:
  - $T(n) = 2(2T(n/4) + n/2) + n = 2^2T(n/2^2) + 2n$
  - $T(n) = 2^2(2T(n/2^3) + n/2^2) + 2n = 2^3T(n/2^3) + 3n$
  - *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).*

Árvore de recursão



n
+
2.(n/2) = n
+
4.(n/4) = n
+
n.(1) = n

n.(1 + lg n)

Tempo total: O(n.log n)

Conclusões

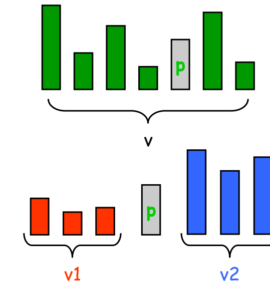
- *MergeSort* necessita de espaço linear para armazenar o vetor temporário.
- Seu tempo de execução é da mesma ordem do *lower bound* do problema, isto é, O(n.log n).
- Podemos dizer que:
  - Qualquer algoritmo que, através de comparações, ordena n números em tempo O(n.log n) é ótimo.
  - A ordenação 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:
  - Escolha um elemento p para ser o pivô em v.
  - Particione o vetor v – {p} em dois grupos distintos:
    - v1 = {x ∈ v – {p} | x < p}
    - v2 = {x ∈ v – {p} | x ≥ p}
- Conquista: ordene recursivamente v1 e v2.
- Combinação: junte v1, p e v2 para obter v ordenado.



Algoritmo básico para o *QuickSort*

```
QuickSort(min, max) {
    if (min < max) {
        p = Partition(min, max);
        Quicksort(min, p-1);
        Quicksort(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 8 9 3 5 6
- Busca: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- Troca: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- Busca: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- Troca: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- Busca: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- Troca: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- Busca: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- Troca com o pivô: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

↑ Posição do pivô

Algoritmo para particionamento

```
int Partition(left, right) {
    pivot = v[left];
    l = left + 1;
    r = right;
    while (true) {
        while (l < right && v[l] < pivot) l++;
        while (r > left && v[r] >= pivot) r--;
        if (l >= r) break;
        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 = |v1|)
- 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!!
- Comprove essas complexidades resolvendo as recorrências.

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 último e o central:
  - O pivô será o valor intermediário entre os três.
  - Estes três valores são rearranjados de tal modo que o menor fique na primeira posição do vetor, o intermediário no centro e o maior no final.
- Quando esta técnica é utilizada, tornam-se muito raros os casos em que o *QuickSort* gasta tempo quadrático.

Pilha de execução

- Nos casos em que o *QuickSort* gasta tempo quadrático, a pilha de execução, devido às chamadas recursivas, chega a exigir espaço O(n).
- Dependendo do tamanho do vetor, esse espaço pode se esgotar, e o programa é abortado...
- Através de uma pequena alteração no código do *QuickSort*, é possível eliminar uma dessas recursões.
- Neste caso, se a recursão restante corresponder à metade menor do vetor, garante-se que a pilha de execução consuma no máximo espaço O(log n).

QuickSort com uma única recursão

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

- Tempo continua $O(n^2)$
- Espaço consumido na pilha de execução: $O(\log n)$