Disciplina: AlgProgOO2

Prof. Dr. Anderson Viçoso de Araújo

Aula 06: Algoritmos de Ordenação

Introdução

- O problema é ordenar uma sequência de números (em um **vetor** ou lista encadeada) de forma crescente
  - $v[0] \leq v[1] \leq \cdots \leq v[n-1]$
  - **Quanto mais rápido melhor, não é?**
- Existe várias maneiras de ordenar uma sequência de números
  - Inteiros **ou não**
- A maioria dos algoritmos de ordenação são ***in-place***
  - Transforma a entrada utilizando uma quantidade pequena e constante de espaço adicional. A entrada é substituída pela saída durante a execução do algoritmo

Algoritmos de Ordenação

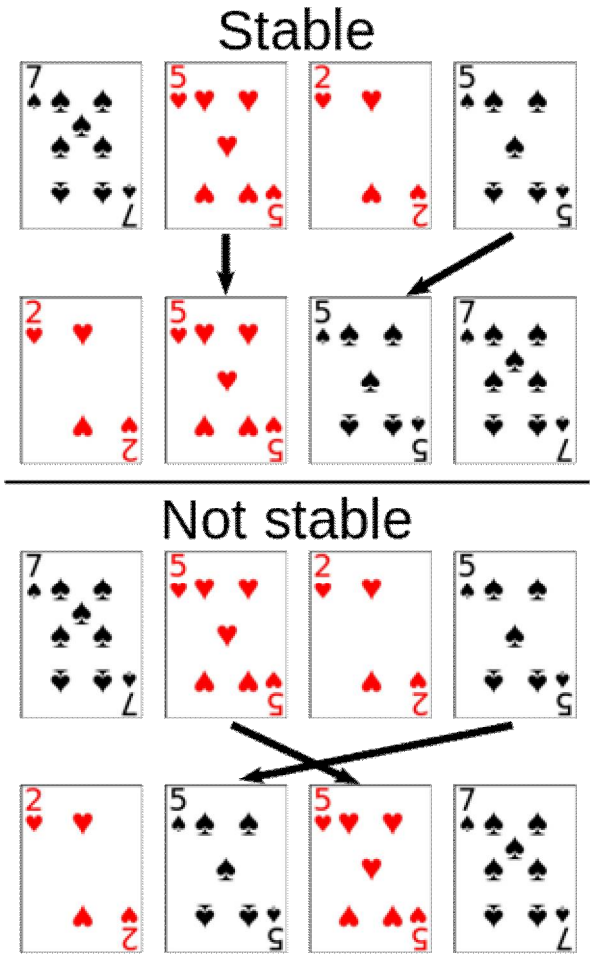
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- ...

E se tivéssemos que ordenar um vetor agora, sem pensar muito...

Como faríamos?

Características dos Algoritmos de Ordenação

- **Estabilidade**
  - Mantém a ordem dos números iguais durante a ordenação
- **Complexidade**
  - Tempo: O(?)
  - Espaço: O(?) em relação a quantidade de estruturas de dados (ED) utilizadas durante a ordenação
    - Auxiliar: EDs utilizadas além da entrada



Bubble Sort

- Ordenação “por bolha”
- É considerado um dos algoritmos mais simples para ordenação
- Em geral, se pensarmos em ordenar uma sequência é uma das primeiras ideias que temos
- A ideia é percorrer o vetor diversas vezes, a cada passagem fazendo o maior elemento “flutuar" para o final da sequência
  - Ou o menor para o início
- **MUITO RUIM!**

Bubble Sort - Algoritmo

```
bubblesort( a[] )
    for i from 0 to N - 1
        for j from 0 to N - 1
            if a[j] > a[j + 1]
                swap(a[j], a[j + 1])
end
```

- Este é o código mais simples possível para o *bubble sort*. Pode ter otimizações.

Bubble Sort - Exemplo

6 5 3 1 8 7 2 4

Bubble Sort – Complexidade

- Estável
- Tempo
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: O(n)*
- Espaço
  - Pior caso: O(n) total, O(1) auxiliar

- *Já otimizado para verificar se não ocorreu nenhuma troca.

Selection Sort

- Ordenação por **seleção**
  - **Selecionar o menor elemento**
- Algoritmo considerado simples e *geralmente* pior que o seu similar *Insertion Sort*
- Pode ser considerado bom para vetores bem pequenos (10-20 elementos)

Selection Sort - Funcionamento

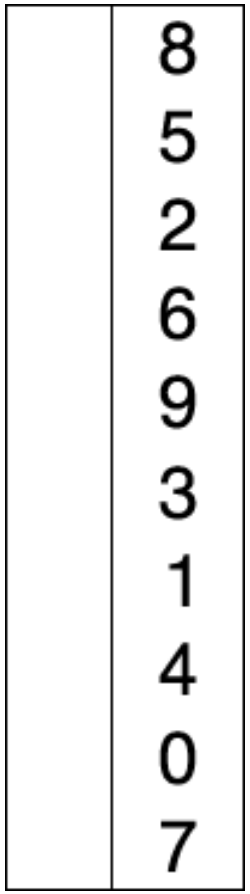
1. Dividir o vetor em 2 partes: a parte ordenada e a desordenada
2. Todos elementos começam na parte desordenada
3. **Selecionar o menor elemento** do vetor não ordenado e trocá-lo com o elemento na primeira posição do vetor não ordenado
4. Com o menor já na sua posição correta, podemos começar novamente da posição posterior a ele
    1. Mudando a posição da parte ordenada e repetindo a operação de busca do menor (voltar para 3)

Selection Sort - Algoritmo

```
for i = 0 to N – 1
      min = i
      for  j = i + 1 to N
            if a[j] < a[min]
                  min = j
            end if
      end for j

      if min != i
            swap(a[i],a[min])
end for i
```

Selection Sort - Exemplo



Selection Sort – Complexidade

- Não estável
- Tempo
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: $O(n^2)$
- Espaço
  - Pior caso: O(n) total, O(1) auxiliar

Insertion Sort

- Ordenação por **inserção**
- Mais eficiente na prática que o *Bubble Sort* e *Selection Sort*
- Também é considerado rápido para vetores pequenos como o *Selection Sort*
- Percorre um vetor de elementos e a medida que avança vai deixando os elementos mais a esquerda ordenados

Insertion Sort - Funcionamento

1. Dividir o vetor em 2 partes: a parte ordenada e a desordenada
2. Começa com o primeiro elemento do vetor na lista dos ordenados
3. Percorre o vetor a partir do primeiro elemento da lista desordenada comparando elementos dois a dois até encontrar um elemento que deve ser trocado (com o menor a direita)
4. Leva o menor pra esquerda até encontrar a posição dele na lista ordenada
5. Continua percorrendo o vetor (volta para 3)

Insertion Sort - Algoritmo

```
for i = 1 to N
    j = i
    while j > 0 and a[j-1] > a[j]
        swap (a[j], a[j-1])
        j = j - 1
    end while
end for
```

Insertion Sort - Exemplo

6 5 3 1 8 7 2 4

Insertion Sort - Complexidade

- Estável
- Tempo
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n^2)$
  - Melhor caso: $O(n)$
- Espaço
  - Pior caso: O(n) total, O(1) auxiliar

Dividir e Conquistar

Dividir e Conquistar

- Técnica que consiste em dividir um problema recursivamente em problemas menores até que possa ser resolvido diretamente
- Dividido em três passos:
  1. **Dividir**: Se o tamanho da entrada é menor que um limite (um ou dois elementos) resolver diretamente. Caso contrário, divida a entrada em dois ou mais subconjuntos disjuntos
  2. **Recursão**: Resolver recursivamente os subproblemas associados com os subconjuntos
  3. **Conquistar**: Pegar as soluções dos problemas menores e agrupá-las na solução do problema original

Merge Sort

- Usa a técnica Dividir e Conquistar
- Algoritmo Criado por **Von Neumann** em 1945
  - Matemático – Arquitetura de Von Neumann
- Pode ser tão lento quanto os algoritmos mais simples*, mas **em geral é muito rápido**
- Precisa da função ***merge*** para executar
  - Ela agrupa em ordem dois vetores já ordenados
- Pode ser mais facilmente paralelizado
  - Faz mais sentido em vetores grandes

*Para vetores pequenos

Merge Sort - Funcionamento

1. Divide o vetor na metade (recursivamente) até chegar ao caso base
    1. Ordenar dois números
2. Função ***Merge***
    1. Combina dois vetores ordenados em um vetor único ordenado

Merge Sort - Algoritmo

```
mergesort(a[], start, end)
     if start < end
          middle = (start+end)/2
          mergesort(a, start, middle)
          mergesort(a, middle+1, end)

          merge(a, start, middle, end)
end
```

Merge Sort – Função *Merge*

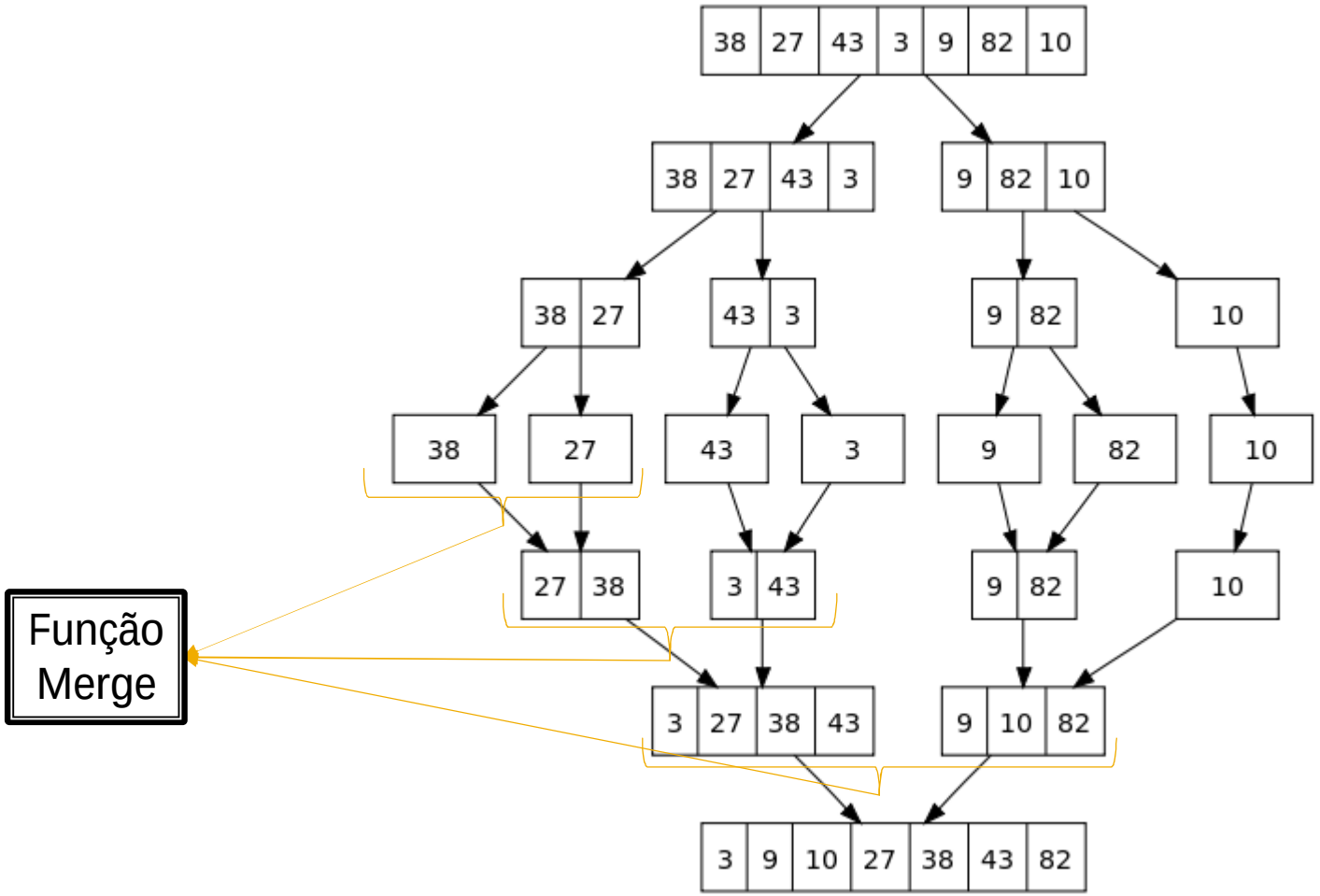
```
merge(a[], start, middle, end)
      for i = start to end
            b[i] = a[i]
      i = start
      j = middle + 1
      k = start

      while i <= middle and j <= end
            if b[i] <= b[j]
                  a[k++] = b[i++]
            else
                  a[k++] = b[j++]

      while i <= middle
            a[k++] = b[i++]

end
```

Merge Sort – Exemplo



Merge Sort – Exemplo 2

6 5 3 1 8 7 2 4

Merge Sort – Complexidade

- Estável
- Tempo:
  - Pior caso: Θ(n log n)
  - Caso médio: Θ(n log n)
  - Melhor caso: Θ(n log n) típico, Θ(n) variante natural
- Espaço:
  - Pior caso: **Θ(n)** auxiliar → Não é *in place*

Primeira vez que memória auxiliar é usada

Quick Sort

- Também usa a técnica Dividir e Conquistar
- Pode ser tão lento quanto os algoritmos mais simples*, mas **em geral é muito rápido**
- Provavelmente é o mais utilizado
- Precisa da função ***partition*** para executar
  - Faz uso do **pivô**
    - É um elemento do vetor tal que os elementos que forem maiores que ele serão considerados grandes e os outros pequenos
    - A escolha do pivô pode ser feita de diversas maneiras:
      - Aleatoriamente
      - O meio, início ou fim do vetor

*Para vetores pequenos

Quick Sort - Funcionamento

1. Escolher o **pivô**
    1. A posição central ou última do vetor
2. Função ***Partition***:
    1. Reorganizar o vetor para que os elementos menores que o pivô fiquem antes dele e os maiores depois
3. Recursivamente aplicar as etapas acima para o sub-conjunto de elementos com valores menores e maiores que o pivô

Quick Sort - Algoritmo

```
quicksort(a[], init, end)
      if init < end
            pivot = partition(a, init, end)
            quicksort(a, init, pivot - 1)
            quicksort(a, pivot, end)


Ou

            pivot = partition(a, init, end)
            if (init < pivot - 1)
                  quicksort(a, init, pivot - 1)
            if (pivot < end)
                  quicksort(a, pivot, end)
```

Quick Sort – Função *Partition*

```
partition(a[], init, end) {
        i = init
        j = end

        pivot = getPivot(a, init, end)

        while i <= j
                while a[i] < pivot
                        i++

                while a[j] > pivot
                        j--

                if i <= j
                        swap(a[i++], a[j--])


        return i
}
```

Quick Sort - Exemplo

6 5 3 1 8 7 2 4

Quick Sort - Complexidade

- Não é estável
- Tempo:
  - Pior caso: $O(n^2)$
  - Caso médio: $O(n \log n)$
  - Melhor caso: $O(n \log n)$
- Espaço
  - Pior caso: O(n) auxiliar para **a pilha de chamadas**

Acabou? Ficou a dúvida: Qual é melhor/mais rápido?

Depende...

Links Interessantes

- AlgoRythmics (*Insertion Sort*)
  - https://www.youtube.com/watch?v=ROalU379l3U
- CS50 Harvard (*Insertion Sort*)
  - https://www.youtube.com/watch?v=DFG-XuyPYUQ
- **15 Sorting Algorithms in 6 Minutes**
  - https://www.youtube.com/watch?v=kPRA0W1kECg
- Animações e comparações para diferentes vetores
  - http://www.sorting-algorithms.com
- Dicas com complexidades para diferentes procedimentos já conhecidos
  - http://bigocheatsheet.com/