

MC102 – Aula 15

Algoritmos de Ordenação Recursivos

Algoritmos e Programação de Computadores

Zanoni Dias

2019

Instituto de Computação

O Problema da Ordenação

Divisão e Conquista

Merge Sort

Quicksort

Exercícios

O Problema da Ordenação

O Problema da Ordenação

- Iremos continuar o estudo de algoritmos para o problema de ordenação visto anteriormente:

Definição do Problema

Dada uma coleção de elementos, com uma relação de ordem entre eles, ordenar os elementos da coleção de forma crescente.

- Nos nossos exemplos, a coleção de elementos será representada por uma lista de inteiros.
 - Números inteiros possuem uma relação de ordem entre eles.
- Apesar de usarmos números inteiros, os algoritmos que estudaremos servem para ordenar qualquer coleção de elementos que possam ser comparados entre si.
- Ambos os algoritmos recursivos de ordenação que veremos usam o paradigma de Divisão e Conquista.

Divisão e Conquista

Divisão e Conquista

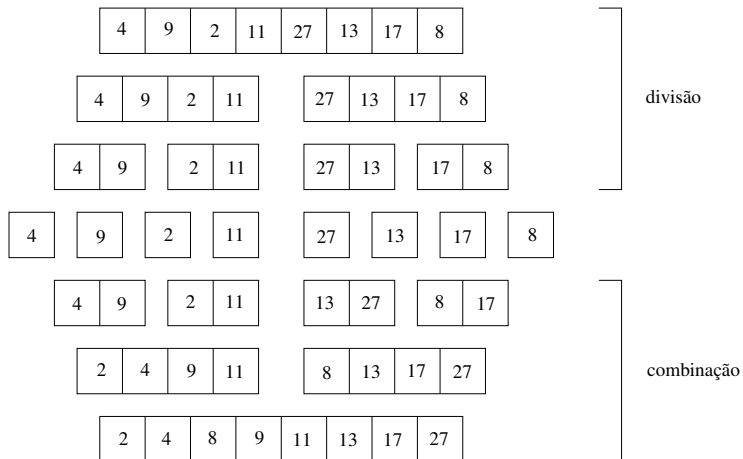
- Esta técnica consiste em dividir um problema maior recursivamente em problemas menores até que ele possa ser resolvido diretamente.
- A solução do problema inicial é dada através da combinação dos resultados de todos os problemas menores computados.
- A técnica soluciona o problema através de três fases:
 - Divisão: o problema maior é dividido em problemas menores.
 - Conquista: cada problema menor é resolvido recursivamente.
 - Combinação: os resultados dos problemas menores são combinados para se obter a solução do problema maior.

Merge Sort

Merge Sort

- O Merge Sort foi proposto por John von Neumann em 1945.
- O algoritmo Merge Sort é baseado em uma operação de intercalação (**merge**) que une duas listas ordenadas para gerar uma terceira lista também ordenada.
- O algoritmo pode ser construído a partir dos seguintes passos:
 - Divisão: a lista é dividida em duas sublistas de tamanhos quase iguais (diferindo em no máximo um elemento).
 - Conquista: cada sublista é ordenada recursivamente.
 - Combinação: as duas sublistas ordenadas são intercaladas para se obter a lista final ordenada.

Merge Sort



Merge Sort - Merge

```
1 def merge(lista1, lista2):
2     i = j = 0
3     aux = []
4
5     while (i < len(lista1)) and (j < len(lista2)):
6         if lista1[i] < lista2[j]:
7             aux.append(lista1[i])
8             i = i + 1
9         else:
10            aux.append(lista2[j])
11            j = j + 1
12
13    while i < len(lista1):
14        aux.append(lista1[i])
15        i = i + 1
16
17    while j < len(lista2):
18        aux.append(lista2[j])
19        j = j + 1
20
21    return aux
```

Merge Sort - Merge

```
1 def merge(lista1, lista2):
2     i = j = 0
3     aux = []
4
5     while (i < len(lista1)) and (j < len(lista2)):
6         if lista1[i] < lista2[j]:
7             aux.append(lista1[i])
8             i = i + 1
9         else:
10            aux.append(lista2[j])
11            j = j + 1
12
13    aux = aux + lista1[i:]
14
15
16
17    aux = aux + lista2[j:]
18
19
20
21    return aux
```

Merge Sort

```
1 def merge(lista1, lista2):
2     ...
3
4 def merge_sort(lista, inicio, fim):
5     if fim - inicio > 1:
6         meio = (inicio + fim) // 2
7
8         merge_sort(lista, inicio, meio)
9         merge_sort(lista, meio, fim)
10
11        lista1 = lista[inicio:meio]
12        lista2 = lista[meio:fim]
13
14        lista[inicio:fim] = merge(lista1, lista2)
```

Merge Sort

```
1 def merge(lista1, lista2):
2     ...
3
4 def merge_sort(lista, inicio, fim):
5     ...
6
7 def main():
8     lista = [4, 9, 2, 11, 27, 13, 17, 8]
9     n = len(lista)
10    merge_sort(lista, 0, n)
11    print(lista)
12
13 main()
14 # [2, 4, 8, 9, 11, 13, 17, 27]
```

Merge Sort - Análise de complexidade

- Seja $T(n)$ o custo de ordenar uma lista de n elementos usando o Merge Sort.
- Para $n > 1$, temos que o algoritmo executa:
 - A ordenação recursiva dos $\lceil n/2 \rceil$ primeiros elementos da lista.
 - A ordenação recursiva dos $\lfloor n/2 \rfloor$ últimos elementos da lista.
 - Intercala as duas sublistas previamente ordenadas.
- A seguinte recorrência define o tempo de execução do Merge Sort:

$$T(1) = c_1$$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + M(n) + c_2$$

- É fácil verificar que $M(n)$, o tempo de execução da função **merge**, é proporcional à função $f(n) = n$.

Merge Sort - Análise de complexidade

- É possível mostrar que $T(n)$, o tempo de execução do Merge Sort, é proporcional à função $f(n) = n \log n$, tanto no melhor quanto no pior caso.
- Da forma como a função **merge** foi implementada utilizando uma lista auxiliar, o Merge Sort necessita de espaço linear de memória adicional.

Quicksort

Quicksort

- O Quicksort foi desenvolvido por Charles A. R. Hoare em 1959.
- O algoritmo Quicksort é baseado em uma operação de particionamento (**partition**) que, com base num elemento pivô, divide a lista em duas partições:
 - Valores menores que o pivô são colocados antes do pivô na lista, enquanto valores maiores são colocados depois.
- O algoritmo pode ser construído a partir dos seguintes passos:
 - Divisão: a lista é dividida em duas partições, usando a função **partition**.
 - Conquista: cada partição é ordenada recursivamente.
 - Combinação: nada precisa ser feito, já que os números menores que o pivô estão antes do pivô (e ordenados), enquanto os maiores estão depois do pivô (e também ordenados).

Partition com Listas Auxiliares

```
1 def partition(lista, inicio, fim):
2     pivo = lista[inicio]
3     menores = []
4     maiores = []
5
6     for k in range(inicio + 1, fim):
7         if lista[k] <= pivo
8             menores.append(lista[k])
9         else:
10            maiores.append(lista[k])
11
12    lista[inicio:fim] = menores + [pivo] + maiores
13
14    return inicio + len(menores)
```

Partition e Quicksort

[45] 53 13 25 89 75 46 32 20 [11]
inicio fim-1

```
1 def partition(lista, inicio, fim):  
2     j = inicio  
3  
4     for i in range(inicio + 1, fim):  
5         if lista[i] <= lista[inicio]:  
6             j = j + 1  
7             (lista[i], lista[j]) = (lista[j], lista[i])  
8  
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])  
10  
11     return j
```

Partition e Quicksort

[45] 53 13 25 89 75 46 32 20 11
j

```
1 def partition(lista, inicio, fim):  
2     j = inicio  
3  
4     for i in range(inicio + 1, fim):  
5         if lista[i] <= lista[inicio]:  
6             j = j + 1  
7             (lista[i], lista[j]) = (lista[j], lista[i])  
8  
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])  
10  
11     return j
```

Partition e Quicksort

[45] [53] 13 25 89 75 46 32 20 11
j i

```
1 def partition(lista, inicio, fim):  
2     j = inicio  
3  
4     for i in range(inicio + 1, fim):  
5         if lista[i] <= lista[inicio]:  
6             j = j + 1  
7             (lista[i], lista[j]) = (lista[j], lista[i])  
8  
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])  
10  
11     return j
```

Partition e Quicksort

[45] [53 13] 25 89 75 46 32 20 11
j i

```
1 def partition(lista, inicio, fim):  
2     j = inicio  
3  
4     for i in range(inicio + 1, fim):  
5         if lista[i] <= lista[inicio]:  
6             j = j + 1  
7             (lista[i], lista[j]) = (lista[j], lista[i])  
8  
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])  
10  
11     return j
```

Partition e Quicksort

[45 53] [13] 25 89 75 46 32 20 11
j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13] [53] 25 89 75 46 32 20 11
j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```


Partition e Quicksort

[45 13] [53 25] 89 75 46 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 53] [25] 89 75 46 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 25] [53] 89 75 46 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 25] [53 89] 75 46 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 25] [53 89 75] 46 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 25] [53 89 75 46] 32 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```


Partition e Quicksort

[45 13 25 53] [89 75 46 32] 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```


Partition e Quicksort

[45 13 25 32] [89 75 46 53] 20 11
 j i

```
1 def partition(lista, inicio, fim):
2     j = inicio
3
4     for i in range(inicio + 1, fim):
5         if lista[i] <= lista[inicio]:
6             j = j + 1
7             (lista[i], lista[j]) = (lista[j], lista[i])
8
9     (lista[inicio], lista[j]) = (lista[j], lista[inicio])
10
11    return j
```

Partition e Quicksort

[45 13 25 32] [89 75 46 53 20] 11
 j i

```
1 def partition(lista, inicio, fim):
2     ...
3
4 def quick_sort(lista, inicio, fim):
5     if fim - inicio > 1:
6         pivo = partition(lista, inicio, fim)
7         quick_sort(lista, inicio, pivo)
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[45 13 25 32 89] [75 46 53 20] 11
 j i

```
1 def partition(lista, inicio, fim):
2     ...
3
4 def quick_sort(lista, inicio, fim):
5     if fim - inicio > 1:
6         pivo = partition(lista, inicio, fim)
7         quick_sort(lista, inicio, pivo)
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[45 13 25 32 20] [75 46 53 89] 11
 j i

```
1 def partition(lista, inicio, fim):
2     ...
3
4 def quick_sort(lista, inicio, fim):
5     if fim - inicio > 1:
6         pivo = partition(lista, inicio, fim)
7         quick_sort(lista, inicio, pivo)
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[45 13 25 32 20] [75 46 53 89 11]
 j i

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[45 13 25 32 20 75] [46 53 89 11]
 j i

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```


Partition e Quicksort

[45 13 25 32 20 11] [46 53 89 75]
início j

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```


Partition e Quicksort

[11 13 25 32 20 45] [46 53 89 75]
início j

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[11 13 25 32 20 45] [46 53 89 75]
j

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```


Partition e Quicksort

[11 13 25 32 20] [45] 46 53 89 75
pivo

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

```
[11  13  20  25  32] [45] 46  53  89  75  
      quicksort      pivo
```

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[11 13 20 25 32] [45] [46 53 89 75]
pivo

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[11 13 20 25 32] [45] [46 53 75 89]
 pivo quicksort

```
1 def partition(lista, inicio, fim):
2     ...
3
4 def quick_sort(lista, inicio, fim):
5     if fim - inicio > 1:
6         pivo = partition(lista, inicio, fim)
7         quick_sort(lista, inicio, pivo)
8         quick_sort(lista, pivo + 1, fim)
```

Partition e Quicksort

[11 13 20 25 32] [45] [46 53 75 89]
pivo

```
1 def partition(lista, inicio, fim):  
2     ...  
3  
4 def quick_sort(lista, inicio, fim):  
5     if fim - inicio > 1:  
6         pivo = partition(lista, inicio, fim)  
7         quick_sort(lista, inicio, pivo)  
8         quick_sort(lista, pivo + 1, fim)
```


Quicksort

```
1 def partition(lista, inicio, fim):
2     ...
3
4 def quick_sort(lista, inicio, fim):
5     ...
6
7 def main():
8     lista = [45, 53, 13, 25, 89, 75, 46, 32, 20, 11]
9     n = len(lista)
10    quick_sort(lista, 0, n)
11    print(lista)
12
13 main()
14 # [11, 13, 20, 25, 32, 45, 46, 53, 75, 89]
```

Quicksort - Análise de complexidade

- Melhor caso: ocorre quando o **partition** sempre divide a lista em duas partições de tamanhos aproximadamente iguais.
- A seguinte recorrência define o tempo de execução do Quicksort no melhor caso:

$$T(1) = c_1$$

$$T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + P(n) + c_2$$

- É fácil ver que $P(n)$, o tempo de execução da função **partition**, é proporcional à função $f(n) = n$.
- É possível mostrar que $T(n)$, o tempo de execução do Quicksort no melhor caso, é proporcional à função $f(n) = n \log n$.

Quicksort - Análise de complexidade

- Pior caso: ocorre quando o **partition** sempre divide a lista em duas partições de tamanhos muito diferentes.
- A seguinte recorrência define o tempo de execução do Quicksort no pior caso:

$$T(1) = c_1$$

$$T(n) = T(n - 1) + P(n) + c_2$$

- Como sabemos, $P(n)$, o tempo de execução da função **partition**, é proporcional à função $f(n) = n$.
- É possível mostrar que $T(n)$, o tempo de execução do Quicksort no pior caso, é proporcional à função $f(n) = n^2$.

Quicksort - Análise de complexidade

- Caso médio: a probabilidade de uma partição de um tamanho qualquer ocorrer é igual a $1/n$.
- A seguinte recorrência define o tempo de execução do Quicksort no caso médio:

$$T(1) = c_1$$
$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-1-i)] + P(n) + c_2$$

- Como sabemos, $P(n)$, o tempo de execução da função **partition**, é proporcional à função $f(n) = n$.
- É possível mostrar que $T(n)$, o tempo de execução do Quicksort no caso médio, é proporcional à função $f(n) = n \log n$.

Quicksort

- Dado uma lista aleatória qualquer, é extremamente raro o Quicksort se comportar como no seu pior caso.
- No entanto, o Quicksort, devido à escolha do primeiro elemento da lista como pivô, apresenta seu pior comportamento quando recebe como entrada um dos casos mais simples possíveis para qualquer algoritmo de ordenação: uma lista já ordenada.
- Uma forma de contornar este caso (lista ordenada) e evitar partições de tamanho zero é utilizar como pivô a mediana de três elementos da lista: o primeiro, o do meio e o último.
- Uma outra alternativa bastante utilizada é definir o pivô como um elemento da lista escolhido de forma aleatória.
- Uma vantagem do Quicksort em relação ao Merge Sort é em relação ao uso de memória auxiliar: o Quicksort não usa uma lista auxiliar, consumindo apenas o espaço para armazenar as variáveis locais na pilha de recursão.

Exercícios

Exercícios

1. Implemente uma versão recursiva do função `merge`.
2. Implemente uma função de partição que use o método da mediana de três elementos da lista para definir o pivô.
3. Implemente uma função de partição que use um elemento da lista escolhido aleatoriamente como pivô.

Dica: use a função `randint` da biblioteca `random`.

```
1 import random
2 # Gera um número aleatório entre 1 e 10 (inclusive)
3 n = random.randint(1, 10)
4 print(n)
5 # 6
```