MC102 – Aula27
Recursão IV - QuickSort

Eduardo C. Xavier

Instituto de Computação – Unicamp

19 de Junho de 2018

Introdução

Vamos usar a técnica de recursão para resolver o problema de ordenação.

- Problema:
  - Temos um vetor v de inteiros de tamanho n.
  - Devemos deixar v ordenado em ordem crescente de valores.
- Veremos um algoritmo baseado na técnica **dividir-e-conquistar** que usa recursão.

Introdução

Vamos usar a técnica de recursão para resolver o problema de ordenação.

- Problema:
  - Temos um vetor v de inteiros de tamanho n.
  - Devemos deixar v ordenado em ordem crescente de valores.
- Veremos um algoritmo baseado na técnica **dividir-e-conquistar** que usa recursão.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Dividir e Conquistar

- Temos que resolver um problema P de tamanho n.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P.

Quick-Sort

- O Quick-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim* com n elementos ($n = fim - ini + 1$).
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:**
    - Escolha em elemento especial do vetor chamado *pivô*.
    - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
  - **Conquistar:** Nada a fazer, já que o vetor estará ordenado devido à como foi feito a fase de divisão.

Quick-Sort

- O Quick-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim* com n elementos ($n = fim - ini + 1$).
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:**
    - Escolha em elemento especial do vetor chamado *pivô*.
    - Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
  - **Conquistar:** Nada a fazer, já que o vetor estará ordenado devido à como foi feito a fase de divisão.

Quick-Sort

- O Quick-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim* com n elementos ($n = fim - ini + 1$).
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:**
    - ★ Escolha em elemento especial do vetor chamado *pivô*.
    - ★ Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
  - **Conquistar:** Nada a fazer, já que o vetor estará ordenado devido à como foi feito a fase de divisão.

Quick-Sort

- O Quick-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim* com n elementos ($n = fim - ini + 1$).
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:**
    - ★ Escolha em elemento especial do vetor chamado *pivô*.
    - ★ Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
  - **Conquistar:** Nada a fazer, já que o vetor estará ordenado devido à como foi feito a fase de divisão.

Quick-Sort

- O Quick-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim* com n elementos ($n = fim - ini + 1$).
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:**
    - ★ Escolha em elemento especial do vetor chamado *pivô*.
    - ★ Particione o vetor em uma posição *pos* tal que todos elementos de *ini* até $pos - 1$ são menores ou iguais do que o *pivô*, e todos elementos de *pos* até *fim* são maiores ou iguais ao *pivô*.
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores (um de *ini* até $pos - 1$ e o outro de *pos* até *fim*).
  - **Conquistar:** Nada a fazer, já que o vetor estará ordenado devido à como foi feito a fase de divisão.

Quick-Sort

- Note a similaridade do Quick-Sort com o Merge-Sort.
- Porém o maior trabalho do Merge-Sort está na fase de conquista onde é necessário fazer a fusão.
- No Quick-Sort o maior trabalho está na fase de Divisão pois é necessário fazer um particionamento do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer" o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posições e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posições e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posições e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

Dado um valor p como pivô, como fazer o particionamento?

- Podemos "varrer"o vetor do início para o fim até encontrarmos um elemento maior que o pivô.
- Varremos o vetor do fim para o início até encontrarmos um elemento menor ou igual ao pivô.
- Trocamos então estes elementos de posições e continuamos com o processo até termos verificado todas as posições do vetor.

Quick-Sort: Particionamento

A função retorna a posição de partição. Ela considera sempre o último elemento como o pivô.

```
def particiona(v, ini, fim):
        pivo = v[fim]

        while ini < fim:
                #encontra alguem > pivo
                while ini < fim and v[ini] <= pivo:
                        ini = ini + 1
                #encontra algume <= pivo
                while ini < fim and v[fim] > pivo:
                        fim = fim - 1
                v[ini], v[fim] = v[fim], v[ini]

        return ini
```

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

- Note que no fim da função garantimos que todos elementos de 0 até (ini -1) são menores ou iguais ao pivô, e de *ini* em diante são maiores.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

- Note que no fim da função garantimos que todos elementos de 0 até (ini -1) são menores ou iguais ao pivô, e de *ini* em diante são maiores.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

- Note que no fim da função garantimos que todos elementos de 0 até (ini -1) são menores ou iguais ao pivô, e de *ini* em diante são maiores.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

- Note que no fim da função garantimos que todos elementos de 0 até (ini -1) são menores ou iguais ao pivô, e de *ini* em diante são maiores.

Quick-Sort: Particionamento

Exemplo: (1,9,3,7,6,2,3,8,5) e pivô=5.

- (1,9,3,7,6,2,3,8,5) $\rightarrow$ (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) $\rightarrow$ (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) $\rightarrow$ (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) $\rightarrow$ Retorna posição 5.

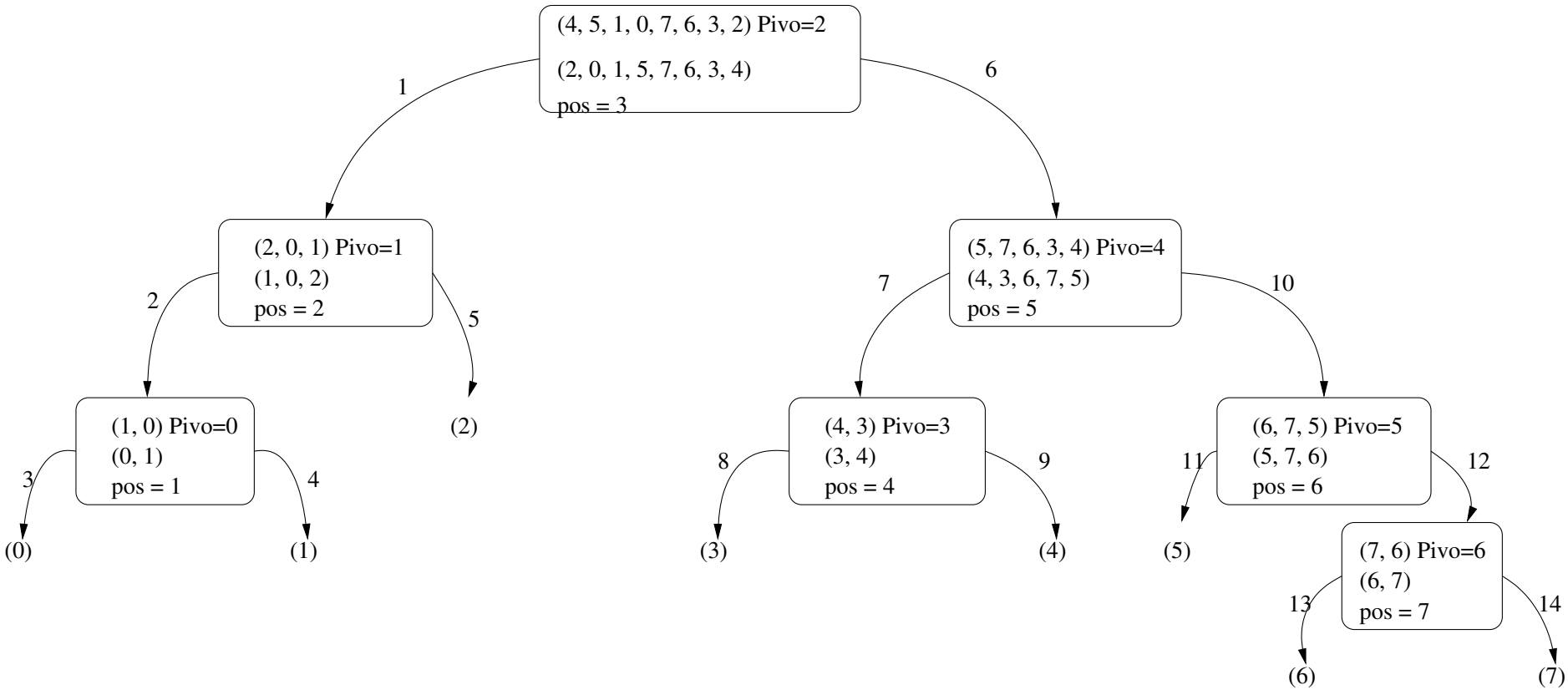
- Note que no fim da função garantimos que todos elementos de 0 até (ini -1) são menores ou iguais ao pivô, e de *ini* em diante são maiores.

Quick-Sort

```
def quickSort(v, ini, fim):
        if ini < fim:
                p = particiona(v, ini, fim)
                quickSort(v, ini, p-1)
                quickSort(v, p, fim)
```

Quick-Sort

Abaixo temos um exemplo da árvore de recursão com ordem das chamadas recursivas.



Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort

- Se o Quick-Sort particionar o vetor de tal forma que cada partição tenha mais ou menos o mesmo tamanho ele é muito eficiente.
- Porém se a partição for muito desigual ($n-1$ de um lado e 1 de outro) ele é ineficiente.
- Quando um vetor já está ordenado ou quase-ordenado, ocorre este caso ruim. Por que?

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.

Quick-Sort: Tratando o pior caso

- Podemos implementar o Quick-Sort de tal forma a diminuirmos a chance de ocorrência do pior caso.
- Ao invés de escolhermos o pivô como um elemento de uma posição fixa, podemos escolher como pivô o elemento de uma posição aleatória.

Random-Quick-Sort

- A única diferença é que escolhemos um elemento aleatório.
- Tal elemento é trocado com o que está no fim (será o pivô).

```
def rquickSort(v, ini, fim):
        pale = random.randint(ini, fim)
        v[pale], v[fim] = v[fim], v[pale]
        if ini < fim:
                p = particiona(v, ini, fim)
                quickSort(v, ini, p-1)
                quickSort(v, p, fim)
```

- A chance de ocorrer um caso ruim para o Random-Quick-Sort é desprezível.

Random-Quick-Sort

- A única diferença é que escolhemos um elemento aleatório.
- Tal elemento é trocado com o que está no fim (será o pivô).

```
def rquickSort(v, ini, fim):
        pale = random.randint(ini, fim)
        v[pale], v[fim] = v[fim], v[pale]
        if ini < fim:
                p = particiona(v, ini, fim)
                quickSort(v, ini, p-1)
                quickSort(v, p, fim)
```

- A chance de ocorrer um caso ruim para o Random-Quick-Sort é desprezível.

Random-Quick-Sort

- Implementadas as funções anteriores podemos rodar o exemplo:

```
def main():
        tam = 10
        v = [random.randint(0,tam) for i in range(tam)]
        print(v)
        rquickSort(v, 0, tam-1)
        print(v)

main()
```

Random-Quick-Sort: Eficiência

Considere um vetor v de n elementos a ser ordenado:

- Para o Random-Quick-Sort, é possível mostrar que o número esperado de operações para ordenar o vetor é proporcional a $\approx n \log n$.
- Uma vantagem em relação ao Merge-Sort é que não há uso de um vetor auxiliar.

Exercícios

1. Aplique o algoritmo de particionamento sobre o vetor $(13, 19, 9, 5, 12, 21, 7, 4, 11, 2, 6, 6)$ com pivô igual a 6.
2. Qual o valor retornado pelo algoritmo de particionamento se todos os elementos do vetor tiverem valores iguais?
3. Faça uma execução passo-a-passo do Quick-Sort com o vetor $(4, 3, 6, 7, 9, 10, 5, 8)$.
4. Modifique o algoritmo QuickSort para ordenar vetores em ordem decrescente.