

MC102 – Aula29

Recursão III - QuickSort

Instituto de Computação – Unicamp

29 de Outubro de 2012

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 .
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **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 .
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **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 .
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **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 .
- **Dividir:** Quebramos P em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior P .

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
 - ▶ Escolha um elemento do vetor especial chamado *pivô*.
 - ▶ Particione o vetor em uma posição *pos* tal que todos os elementos de *ini* até *pos* - 1 são menores ou iguais do que o *pivô*, e todos os 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 a como foi feita a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
 - ▶ Escolha em elemento do vetor especial 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 a como foi feito a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
 - ▶ Escolha um elemento do vetor especial chamado *pivô*.
 - ▶ Particione o vetor em uma posição *pos* tal que todos os elementos de *ini* até *pos* - 1 são menores ou iguais do que o *pivô*, e todos os 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 a como foi feita a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
 - ▶ Escolha um elemento do vetor especial 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 a como foi feita a fase de divisão.

Quick-Sort

- Vamos supor que devemos ordenar um vetor de uma posição *ini* até *fim*.
- **Dividir:**
 - ▶ Escolha um elemento do vetor especial chamado *pivô*.
 - ▶ Particione o vetor em uma posição *pos* tal que todos os elementos de *ini* até *pos* - 1 são menores ou iguais do que o *pivô*, e todos os 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 a como foi feita a fase de divisão.

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.

```
int particiona(int v[], int ini, int fim){
    int pivo = v[fim], aux;

    while(ini<fim){
        while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
            ini++;                               //maior que o pivô

        while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
            fim--;                               //menor ou igual ao pivô

        aux = v[ini]; //troca estes elementos de posição
        v[ini] = v[fim];
        v[fim] = aux;
    }

    //O laço para quando ini==fim, ou seja checamos o vetor inteiro
    return ini;
}
```

Quick-Sort: Particionamento

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

```
int particiona(int v[], int ini, int fim){
    int pivo = v[fim], aux;

    while(ini<fim){
        while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
            ini++;                               //maior que o pivô

        while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
            fim--;                               //menor ou igual ao pivô

        aux = v[ini]; //troca estes elementos de posição
        v[ini] = v[fim];
        v[fim] = aux;
    }

    //O laço para quando ini==fim, ou seja checamos o vetor inteiro
    return ini;
}
```

Quick-Sort: Particionamento

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

```
int particiona(int v[], int ini, int fim){
    int pivo = v[fim], aux;

    while(ini<fim){
        while( (ini < fim) && (v[ini] <= pivo) ) //para quando encontrar elemento
            ini++;                               //maior que o pivô

        while( (ini < fim) && (v[fim] > pivo) ) //para quando encontrar elemento
            fim--;                               //menor ou igual ao pivô

        aux = v[ini]; //troca estes elementos de posição
        v[ini] = v[fim];
        v[fim] = aux;
    }

    //O laço para quando ini==fim, ou seja checamos o vetor inteiro
    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) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

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) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

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) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

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) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

Quick-Sort: Particionamento

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

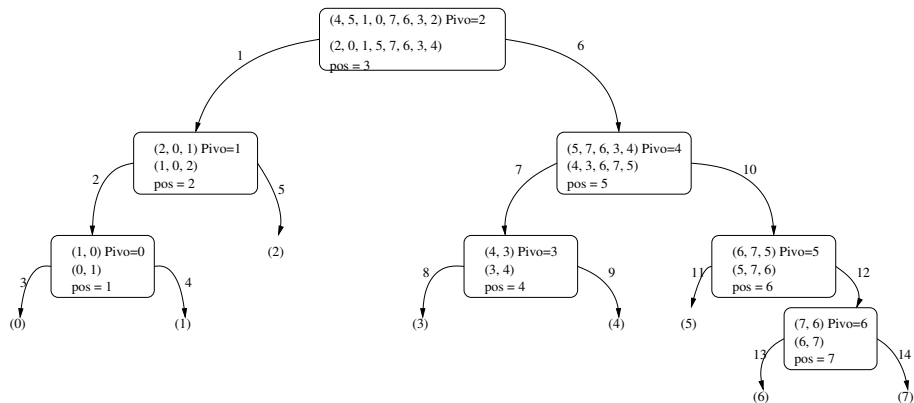
- (1,9,3,7,6,2,3,8,5) → (1,5,3,7,6,2,3,8,9)
- (1,5,3,7,6,2,3,8,9) → (1,5,3,3,6,2,7,8,9)
- (1,5,3,3,6,2,7,8,9) → (1,5,3,3,2,6,7,8,9)
- (1,5,3,3,2,6,7,8,9) → Retorna posição 5.

Quick-Sort

```
void quickSort(int v[], int ini, int fim){
    if(ini < fim){ //só faz ordenação se tiver pelo
        //menos 2 elementos
        int pos = particiona(v, ini, fim);
        quickSort(v,ini, pos-1);
        quickSort(v,pos, 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.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

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.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

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.
- Podemos usar a função *rand* em *stdlib.h* que retorna um número de forma aleatória entre 0 e *RAND_MAX*.

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).

```
void randomQuickSort(int v[],int ini, int fim){
    int j, aux;
    j = rand()%(fim-ini+1);
    aux = v[fim];
    v[fim] = v[ini+j];
    v[ini+j] = aux;

    if(ini < fim){
        int pos = particiona(v, ini, fim);
        randomQuickSort(v, ini, pos-1);
        randomQuickSort(v, pos, fim);
    }
}
```

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).

```
void randomQuickSort(int v[],int ini, int fim){
    int j, aux;
    j = rand()%(fim-ini+1);
    aux = v[fim];
    v[fim] = v[ini+j];
    v[ini+j] = aux;

    if(ini < fim){
        int pos = particiona(v, ini, fim);
        randomQuickSort(v, ini, pos-1);
        randomQuickSort(v, pos, fim);
    }
}
```

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).

```
void randomQuickSort(int v[],int ini, int fim){
    int j, aux;
    j = rand()%(fim-ini+1);
    aux = v[fim];
    v[fim] = v[ini+j];
    v[ini+j] = aux;

    if(ini < fim){
        int pos = particiona(v, ini, fim);
        randomQuickSort(v, ini, pos-1);
        randomQuickSort(v, pos, fim);
    }
}
```

Random-Quick-Sort

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

Quick-Sort

- 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.