

# MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2015

# Roteiro

1 Ordenação

2 Divisão e Conquista

3 Merge Sort

4 Quicksort

# Ordenação

- Conforme estudamos anteriormente, o problema de ordenação pode ser definido da seguinte forma:

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 um vetor 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.

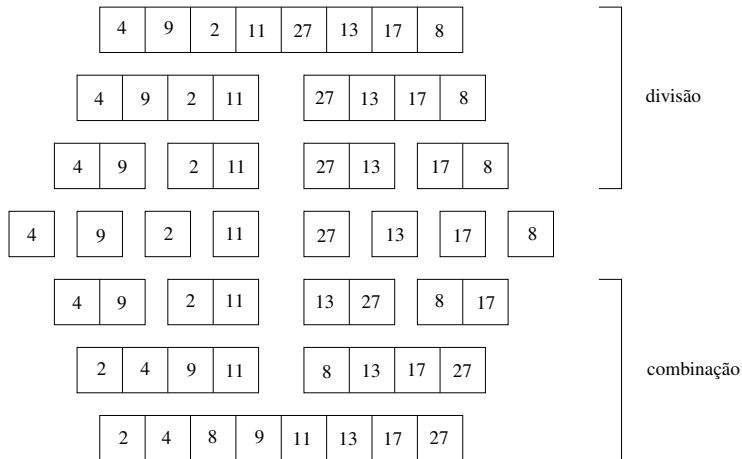
# Divisão e Conquista

- 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

- 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 dois vetores ordenados para gerar um terceiro vetor também ordenado.
- O algoritmo pode ser construído a partir dos seguintes passos:
  - ▶ Divisão: o vetor é dividido em dois subvetores de tamanhos aproximadamente iguais.
  - ▶ Conquista: cada subvetor é ordenado recursivamente.
  - ▶ Combinação: os dois subvetores ordenados são intercalados para se obter o vetor final ordenado.

# Merge Sort



# Merge Sort

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

void merge(int v[], int aux[], int inicio1, int inicio2, int fim2) {
    int i = inicio1, j = inicio2, fim1 = inicio2 - 1, k = 0;

    /* Enquanto existirem elementos nas duas partes...*/
    while ((i <= fim1) && (j <= fim2))
        /* ... verifica qual dos dois elementos iniciais eh o menor */
        if (v[i] < v[j])
            aux[k++] = v[i++]; /* ou copia o elemento inicial da primeira parte */
        else
            aux[k++] = v[j++]; /* ou copia o elemento inicial da segunda parte */

    while (i <= fim1) /* Se ainda existir elementos na primeira parte ... */
        aux[k++] = v[i++]; /* ... copia os elementos restantes no vetor auxiliar */

    while (j <= fim2) /* Se ainda existir elementos na segunda parte ... */
        aux[k++] = v[j++]; /* ... copia os elementos restantes no vetor auxiliar */

    for (i = 0; i < k; i++) /* Copia os elementos do vetor auxiliar ... */
        v[i + inicio1] = aux[i]; /* ... de volta para o vetor original */
}
```

# Merge Sort

```
void mergesort(int v[], int aux[], int inicio, int fim) {
    int meio = (inicio + fim) / 2;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        mergesort(v, aux, inicio, meio);          /* Ordena a primeira parte */
        mergesort(v, aux, meio + 1, fim);        /* Ordena a segunda parte */
        merge(v, aux, inicio, meio + 1, fim);    /* Intercala as duas partes */
    }
}
```



# Merge Sort

```
int main() {
    int v[MAX], aux[MAX], n, i;

    printf("Entre com o tamanho do vetor: ");
    scanf("%d", &n);

    if (n > MAX) {
        n = MAX;
    }

    printf("Entre com os %d valores inteiros:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);

    mergesort(v, aux, 0, n - 1);

    /* Imprime o vetor ordenado */
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

# Merge Sort

```
int main() {
    int *v, *aux, n, i;

    printf("Entre com o tamanho do vetor: ");
    scanf("%d", &n);
    v = malloc(n * sizeof(int));
    aux = malloc(n * sizeof(int));

    printf("Entre com os %d valores inteiros:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);

    mergesort(v, aux, 0, n - 1);

    /* Imprime o vetor ordenado */
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);
    free(aux);
    return 0;
}
```

# Merge Sort - Análise de complexidade

- Seja  $T(n)$  o custo de ordenar um vetor 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 do vetor.
  - ▶ A ordenação recursiva dos  $\lfloor n/2 \rfloor$  últimos elementos do vetor.
  - ▶ Intercala os dois subvetores previamente ordenados.
- 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 (recebendo um ponteiro para um vetor auxiliar), o Merge Sort necessita de espaço linear de memória adicional.

# Merge Sort - Exercício

## Exercício

*Escreva uma versão recursiva da função merge. Sua função deve receber ponteiros para três vetores de inteiros A, B e C, tal que A e B são vetores ordenados de tamanhos  $n$  e  $m$ , respectivamente. Sua função deve copiar em C os elementos dos vetores A e B, de forma a gerar um vetor ordenado de tamanho  $n + m$ . Sua função não deve usar um vetor auxiliar, nem qualquer tipo de repetição (for, while, etc).*

*Protótipo:*

```
void merge(int *A, int *B, int *C, int n, int m);
```

# 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 o vetor em duas partições:
  - ▶ Valores menores que o pivô são colocados antes do pivô no vetor, enquanto valores maiores são colocados depois.
- O algoritmo pode ser construído a partir dos seguintes passos:
  - ▶ Divisão: o vetor é dividido em duas partições, usando o *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).

# Quicksort

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

void troca(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int partition(int v[], int inicio, int fim) {
    int j = inicio; /* j indica o ultimo elemento da primeira particao */
    int i;          /* i indica o elemento atual a ser verificado      */

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]); /* Posiciona o pivo entre as duas particoes */

    return j; /* Retorna a posicao final do pivo */
}
```

# Quicksort

```
void quicksort(int v[], int inicio, int fim) {
    int pivo;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        pivo = partition(v, inicio, fim);    /* Particiona o vetor      */
        quicksort(v, inicio, pivo - 1);    /* Ordena a primeira particao */
        quicksort(v, pivo + 1, fim);      /* Ordena a segunda particao */
    }
}
```



# Quicksort

```
int main() {
    int v[MAX], n, i;

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    if (n > MAX) {
        n = MAX;
    }

    printf("Entre com os %d valores inteiros:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);

    quicksort(v, 0, n - 1);

    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    return 0;
}
```

# Quicksort

```
int main() {
    int *v, n, i;

    printf("Entre com o valor de n: ");
    scanf("%d", &n);

    v = malloc(n * sizeof(int));

    printf("Entre com os %d valores inteiros:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);

    quicksort(v, 0, n - 1);

    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");

    free(v);

    return 0;
}
```

# Quicksort - Partition

[45] 23 13 25 89 75 46 32 20 [11]  
início fim

```
void quicksort(int v[], int inicio, int fim) {
    int pivo;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        pivo = partition(v, inicio, fim);    /* Particiona o vetor */
        quicksort(v, inicio, pivo - 1);
        quicksort(v, pivo + 1, fim);
    }
}
```

# Quicksort - Partition

[45] [23] 13 25 89 75 46 32 20 11  
j i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio; /* j indica o ultimo elemento da primeira particao */
    int i;          /* i indica o elemento atual a ser verificado */

    for (i = inicio + 1; i <= fim; i++)
        if (v[i] <= v[inicio])
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] [23] 13 25 89 75 46 32 20 11  
j i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 [13] 25 89 75 46 32 20 11  
j i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 [25] 89 75 46 32 20 11  
          j   i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 [89] 75 46 32 20 11  
          j      i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```



# Quicksort - Partition

[45] 23 13 25 89 [75] 46 32 20 11  
                  j                  i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 89 75 [46] 32 20 11  
                  j                  i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 89 75 46 [32] 20 11  
                  j                  i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 32 75 46 89 [20] 11  
                                  j                                  i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 32 20 46 89 75 [11]  
                                  j                                  i

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        /* Se o elemento atual for menor ou igual que o pivo... */
        if (v[i] <= v[inicio]) /* (pivo = primeiro elemento) */
            /* ... posiciona o elemento atual na primeira particao */
            troca(&v[++j], &v[i]);

    troca(&v[inicio], &v[j]);

    return j;
}
```

# Quicksort - Partition

[45] 23 13 25 32 20 [11] 89 75 46  
início j

```
int partition(int v[], int inicio, int fim) {
    int j = inicio;
    int i;

    for (i = inicio + 1; i <= fim; i++)
        if (v[i] <= v[inicio])
            troca(&v[++j], &v[i]);

    /* Posiciona o pivo (primeiro elemento) entre as duas partes */
    troca(&v[inicio], &v[j]);

    return j;
}
```







# Quicksort

[11 23 13 25 32 20] 45 [89 75 46]  
quicksort quicksort

```
void quicksort(int v[], int inicio, int fim) {
    int pivo;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        pivo = partition(v, inicio, fim);
        quicksort(v, inicio, pivo - 1);
        quicksort(v, pivo + 1, fim);
    }
}
```

# Quicksort

[11 13 20 23 25 32] 45 [89 75 46]  
quicksort quicksort

```
void quicksort(int v[], int inicio, int fim) {
    int pivo;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        pivo = partition(v, inicio, fim);
        quicksort(v, inicio, pivo - 1);    /* Ordena a primeira particao */
        quicksort(v, pivo + 1, fim);
    }
}
```

# Quicksort

[11 13 20 23 25 32] 45 [46 75 89]  
quicksort quicksort

```
void quicksort(int v[], int inicio, int fim) {
    int pivo;

    /* Se existirem pelo menos dois elementos para serem ordenados... */
    if (inicio < fim) {
        pivo = partition(v, inicio, fim);
        quicksort(v, inicio, pivo - 1);
        quicksort(v, pivo + 1, fim);          /* Ordena a segunda particao */
    }
}
```

## Quicksort - Análise de complexidade

- Melhor caso: ocorre quando o `partition` sempre divide o vetor 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 o vetor 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 um vetor aleatório qualquer, é extremamente raro o Quicksort se comportar como no seu pior caso.
- No entanto, o Quicksort, devido à escolha do primeiro elemento do vetor como pivô, apresenta seu pior comportamento quando recebe como entrada um dos casos mais simples possíveis para qualquer algoritmo de ordenação: um vetor já ordenado.
- Uma forma de contornar este caso (vetor ordenado) e evitar partições de tamanho zero é utilizar como pivô a mediana de três elementos do vetor: o primeiro, o do meio e o último.
- Uma outra alternativa bastante utilizada é definir o pivô como um elemento do vetor 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 um vetor auxiliar, consumindo apenas o espaço para armazenar as variáveis locais na pilha de recursão.

# Quicksort - Exercícios

## Exercício

*Implemente uma função de partição que use o método da mediana de três elementos do vetor para definir o pivô.*

## Exercício

*Implemente uma função de partição que use um elemento do vetor escolhido aleatoriamente como pivô.*

*Dica: use a função `int rand()` da biblioteca `stdlib.h`.*