Programação 2 _ T4

Algoritmos de pesquisa e ordenação em listas.

Rui Camacho
(slides por Luís Teixeira)
Li.EEC 2021/2022

PESQUISA EM LISTAS

Problema *(pesquisa de valor numa lista)*:

- Verificar se um valor existe na listas, no caso de existir, indicar a sua posição.
- Possíveis variantes para o caso de listas com valores repetidos:
  - (a) indicar a posição da primeira ocorrência
  - (b) indicar a posição da última ocorrência
  - (c) indicar a posição de uma ocorrência qualquer

PESQUISA SEQUENCIAL

Algoritmo *(pesquisa sequencial)*:

- Percorrer sequencialmente todas as posições da lista, da primeira para a última (a) ou da última para a primeira (b), até encontrar o valor pretendido ou chegar ao fim do vetor
  - (a) caso se pretenda saber a posição da primeira ocorrência
  - (b) caso se pretenda saber a posição da última ocorrência

IMPLEMENTAÇÃO DA PESQUISA SEQUENCIAL EM C

```
/* Procura um valor inteiro (x) num vetor (v). Retorna o índice
da sua primeira ocorrência, se encontrar; senão, retorna -1. */

int pesquisaSequencial(int v[], int tamanho, int x)
{
   int i;
   for (i = 0; i < tamanho; i++)
      if (v[i] == x)
         return i; // encontrou

   return -1; // não encontrou
}
```

PESQUISA EM LISTAS ORDENADAS

Problema (*pesquisa de valor numa lista ordenada*):

- verificar se um valor (*x*) existe numa lista (*l*) previamente ordenada e, no caso de existir, indicar a sua posição
- no caso de listas com valores repetidos, consideramos a variante em que basta indicar a posição de uma ocorrência qualquer

PESQUISA BINÁRIA

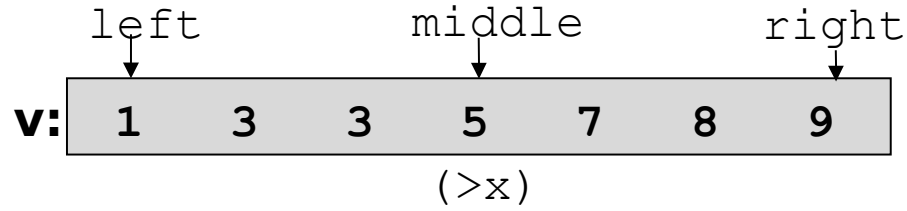
Algoritmo (*pesquisa binária*):

- Comparar o valor que se encontra a meio da lista com o valor procurado, podendo acontecer uma de três coisas:
  - 1) é igual ao valor procurado ⇒ está encontrado
  - 2) é maior do que o valor procurado ⇒ continuar a procurar (do mesmo modo) na sub-lista à esquerda da posição inspeccionada
  - 3) é menor do que o valor procurado ⇒ continuar a procurar (do mesmo modo) na sub-lista à direita da posição inspeccionada.
- Se a lista a inspecionar se reduzir a uma lista vazia, conclui-se que o valor procurado não existe.

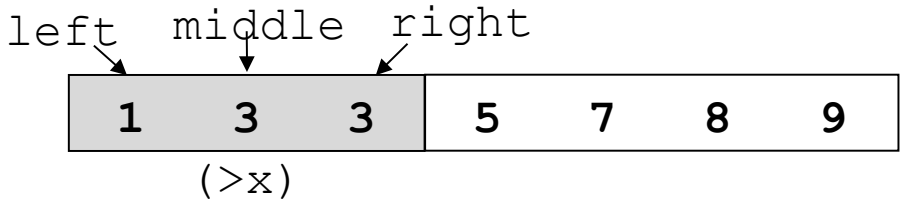
EXEMPLO DE PESQUISA BINÁRIA

x: 2

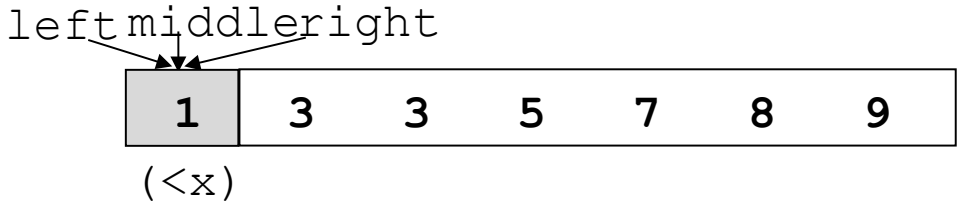
1ª iteração



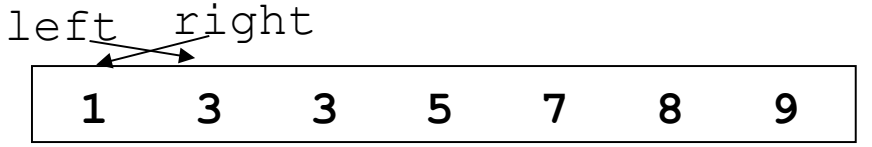
2ª iteração



3ª iteração



4ª iteração



vetor a inspecionar vazio ⇒ o valor 2 não existe no vetor inicial !

IMPLEMENTAÇÃO DA PESQUISA BINÁRIA EM C

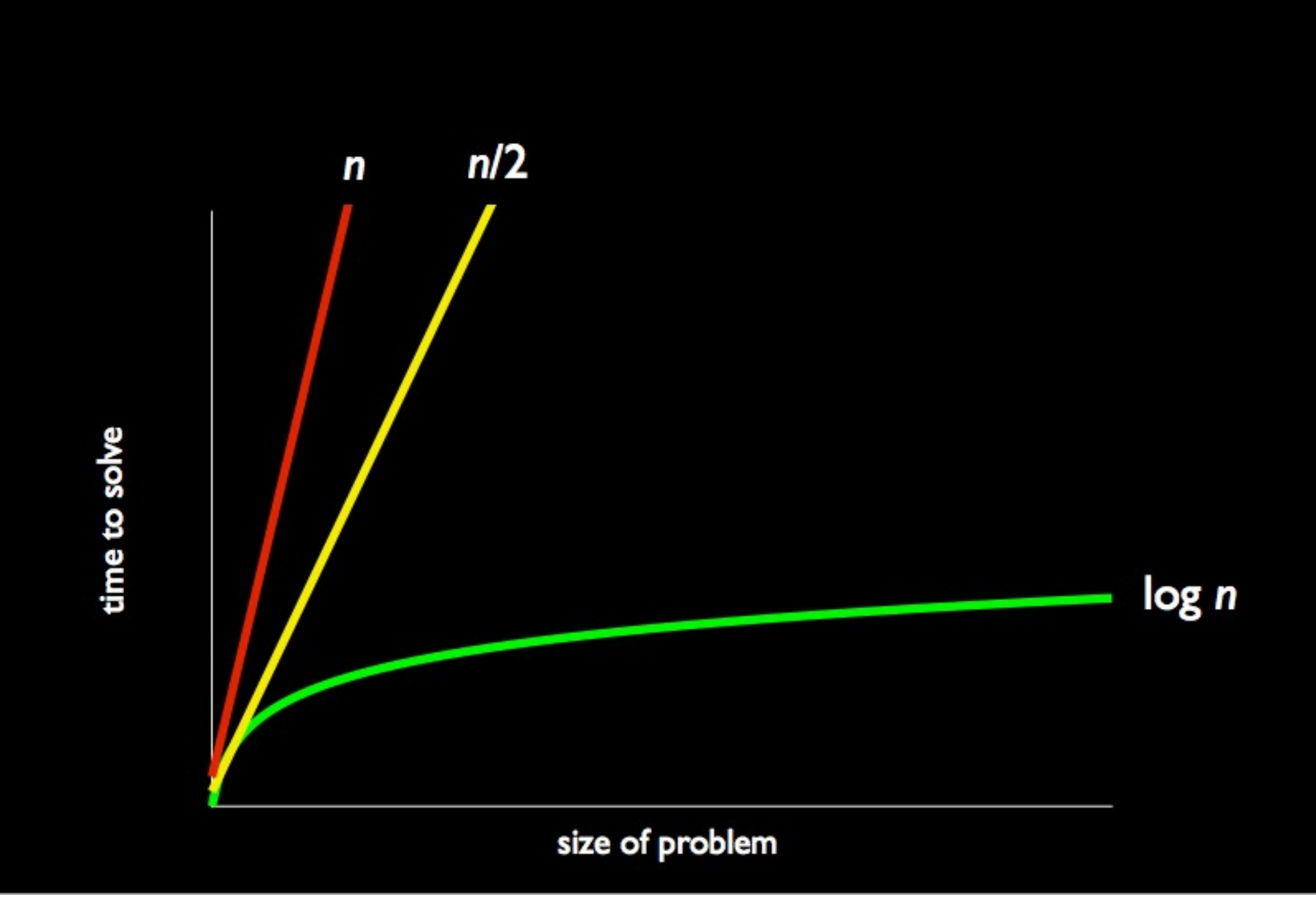
```
/* Procura um valor inteiro (x) num vetor (v) previamente ordenado.
Retorna o índice de uma ocorrência, se encontrar; senão, retorna -1. */

int pesquisaBinaria(int v[], int tamanho, int x)
{
    int left = 0, right = tamanho - 1;
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if (x == v[middle])
        return middle; // encontrou
        else if (x < v[middle])
        right = middle - 1;
        else left = middle + 1;
    }
    return -1;  // não encontrou
}
```

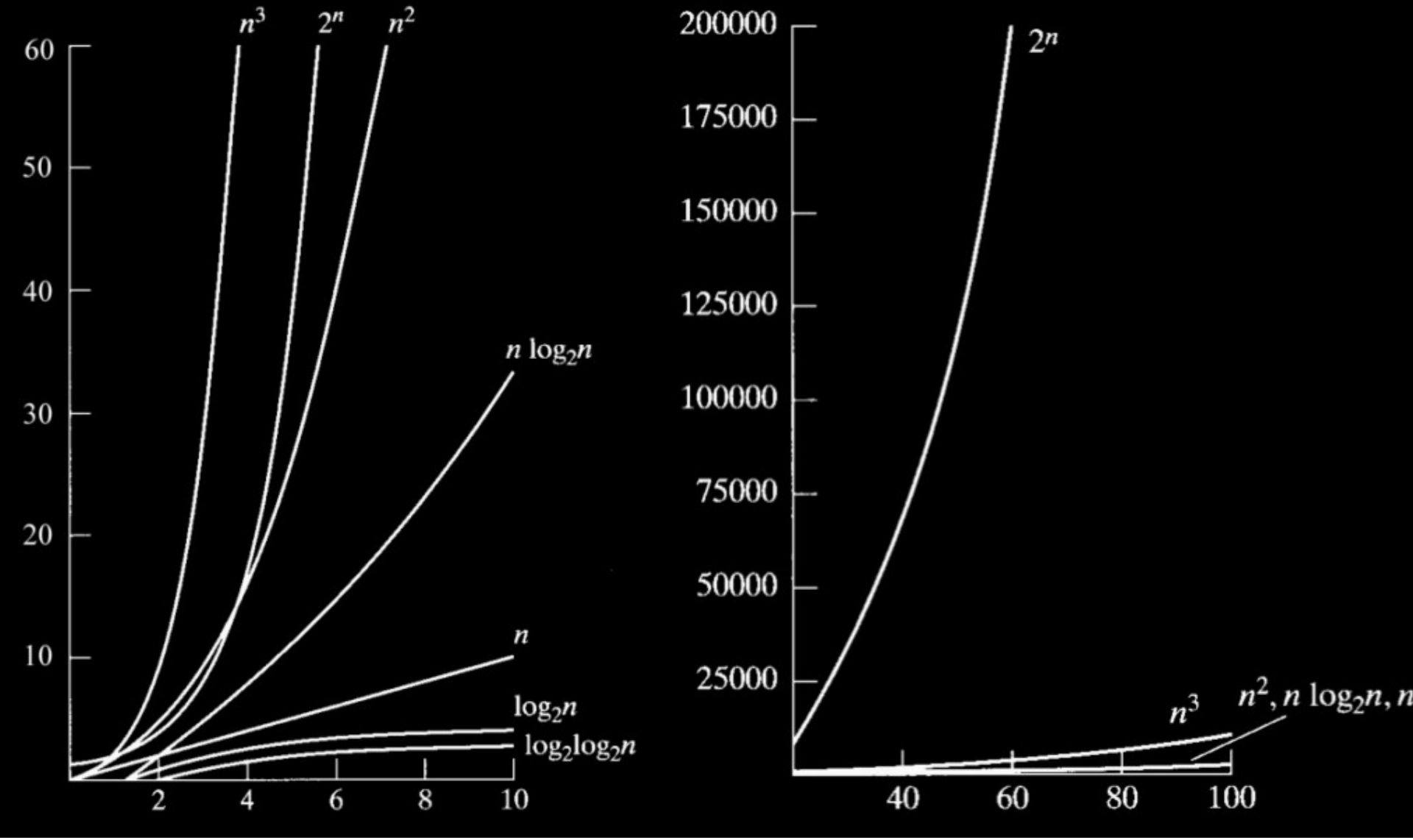
COMPLEXIDADE DE ALGORTIMOS

- Como se comporta quando a dimensão do problema aumenta?

- Notação big-O
  - Algoritmo de tempo constante (”ordem 1”): O(1)
  - Algoritmo de tempo linear ("ordem N”): O(N)
  - Algoritmo de tempo quadrático ("ordem N^2”): O(N^2)

- Complexidade analisada em mais detalhe na aula T11


n
n/2
log n
time to solve
size of problem


n^3
2^n
n^2
$n \log_2 n$
n
$\log_2 n$
$\log_2 \log_2 n$
60
50
40
30
20
10
2
4
6
8
10
200000
175000
150000
125000
100000
75000
50000
25000
2^n
n^3
$n^2, n \log_2 n, n$
40
60
80
100

O(1)

```
/**  indica o comprimento da lista
  *  parametro: lst apontador para a lista
  *  retorno: numero de elementos da lista ou -1 se lista = NULL */


int lista_tamanho(lista *lst)
{
    if(lst == NULL)
        return -1;
    return lst->tamanho;
}
```

O(N)

```
/**
*  devolve a posicao do primeiro elemento da lista com a string especificada
*  parametro: lst apontador para a lista
*  parametro: str string a pesquisar
*  retorno: elemento ou NULL se nao encontrar elemento ou ocorrer um erro */

l_elemento* lista_pesquisa(lista *lst, const char* str)
{
    l_elemento *aux;

    if(lst == NULL || str == NULL)
     return NULL;

    for (aux = lst->inicio; aux != NULL; aux = aux->proximo)
    {
     if (strcmp(aux->str, str) == 0)
     {
          return aux;
     }
    }

    return NULL;
}
```

O(N²)

```
/**
*  ordena uma lista por ordem alfabetica
*  parametro: lst apontador para a lista
*  retorno: -1 se ocorrer um erro ou 0 se for bem sucedido */

int lista_ordena(lista *lst)
{
    l_elemento *next, *curr, *min;
    char *aux;
    if(lst == NULL) return -1;
    if(lst->tamanho == 0) return 0;
    for(curr = lst->inicio; curr->proximo != NULL; curr=curr->proximo) {
     min = curr;
     next = curr->proximo;
     while(next != NULL) {
                if(strcmp(next->str, min->str) < 0) min = next;
                next = next->proximo;
     }
     if (min != curr){
           aux = curr->str;
           curr->str = min->str;
           min->str = aux;
     }
    }
    return 0;
}
```

ORDENAÇÃO DE LISTAS

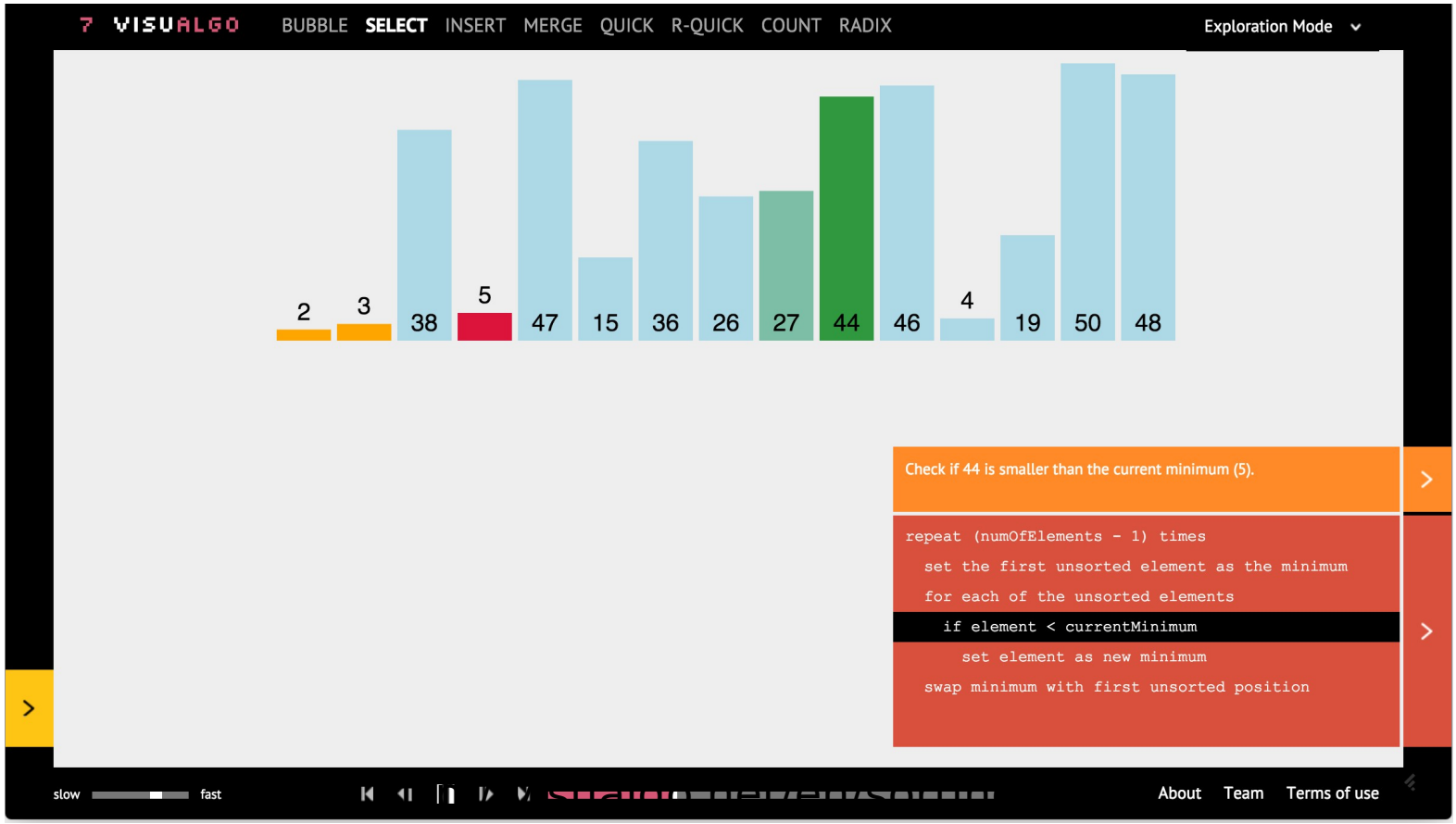
- Problema (*ordenação de lista*)
  - Dado uma lista (*l*) com **N** elementos, rearranjar esses elementos por ordem crescente (ou melhor, por ordem não decrescente, porque podem existir valores repetidos)

- Ideias base:
  - Existem diversos algoritmos de ordenação com complexidade **O(N^2)** que são muito simples **(**por ex: Ordenação por Inserção, BubbleSort)
  - Existem algoritmos de ordenação mais difíceis de codificar que têm complexidade **O(N log N)**

ALGORITMOS DE ORDENAÇÃO DE VECTORES

- Algoritmos:
  - Ordenação por Seleção - $O(N^2)$ já abordado em Prog1
  - Ordenação por Inserção - $O(N^2)$
  - BubbleSort - $O(N^2)$
  - ShellSort - $O(N^2)$ variante mais popular
  - MergeSort - O(N logN)
  - Ordenação por Partição (QuickSort) - O(N logN)
  - HeapSort - O(N logN)

VISUALIZAÇÃO DE ALGORITMOS DE ORDENAÇÃO



ORDENAÇÃO POR SELEÇÃO

- Algoritmo (*ordenação por seleção*):
  - Algoritmo tem N-1 fases: de 1 a N-1.
    - (sendo N o tamanho do vetor)
  - Em cada fase F:
    - Procura-se (sequencialmente) a posição M com o menor elemento guardado nas posições de F a N;
    - Troca-se o valor guardado na posição F com o valor guardado na posição M.
      - (excepto se M for igual a F)

ORDENAÇÃO POR SELEÇÃO

indice	início	**passo 1**	**passo 2**	**passo 3**	**passo 4**	**passo 5**	**passo 6**	**passo 7**
0	7	7	2	2	2	2	2	2
1	21	21	21	7	7	7	7	7
2	10	10	10	10	10	10	10	10
3	15	15	15	15	15	11	11	11
4	2	2	7	21	21	21	13	13
5	13	13	13	13	13	13	21	15
6	11	11	11	11	11	15	15	21

ORDENAÇÃO POR SELEÇÃO

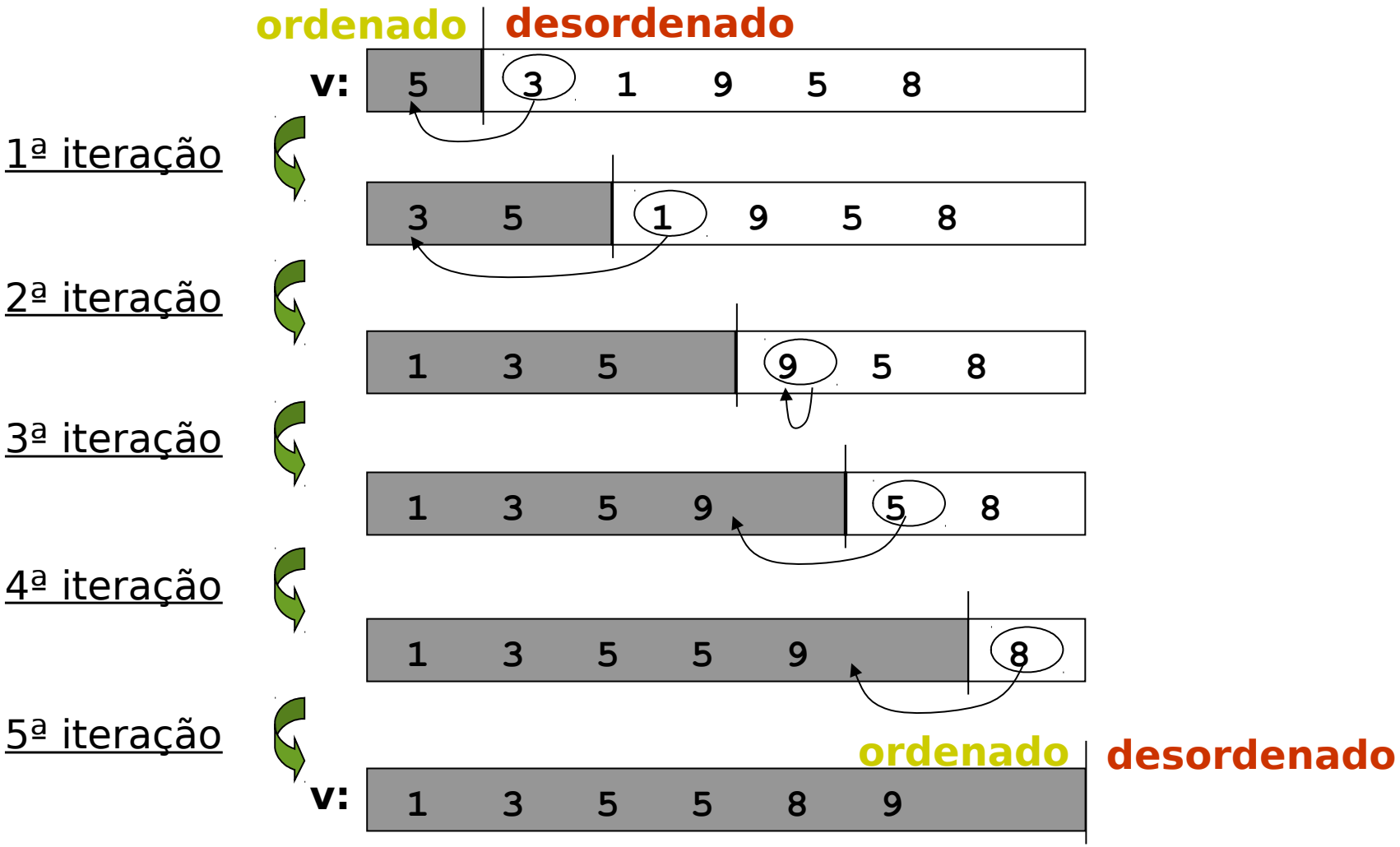
```
#include <stdio.h>
void ordenacaoSelecao(int v[ ], int n){
    int passo, imin, i, aux;
    for(passo=0; passo < n - 1; passo++) {
    imin = passo;
    i = passo + 1;
    while (i < n){
            if (v[i] < v[imin]) imin = i;
            i++;
    }
    if (imin != passo) {
            aux = v[passo];
        v[passo] = v[imin];
        v[imin] = aux;
    }
    }
}
```

```
int main() {
    int i=0, v[7] = {17, 3, 31, 9, -3, 35, 10};
    ordenacaoSelecao(v, 7);
    while (i < 7) printf("%d ", v[i++]);
}
```

ORDENAÇÃO POR INSERÇÃO

- Algoritmo iterativo de **N-1** passos
- Em cada passo *p:*
  - coloca-se um elemento na ordem, sabendo que elementos dos índices inferiores (entre **0** e **p-1**) já estão ordenados
- Algoritmo (*ordenação por inserção*):
  - Considera-se o vetor dividido em dois sub-vetores (esquerdo e direito), com o da esquerda ordenado e o da direita desordenado
  - Começa-se com um elemento apenas no sub-vetor da esquerda
  - Move-se um elemento de cada vez do sub-vetor da direita para o sub-vetor da esquerda, inserindo-o na posição correcta por forma a manter o sub-vetor da esquerda ordenado
  - Termina-se quando o sub-vetor da direita fica vazio

ORDENAÇÃO POR INSERÇÃO



ORDENAÇÃO POR INSERÇÃO (IMPLEMENTAÇÃO)

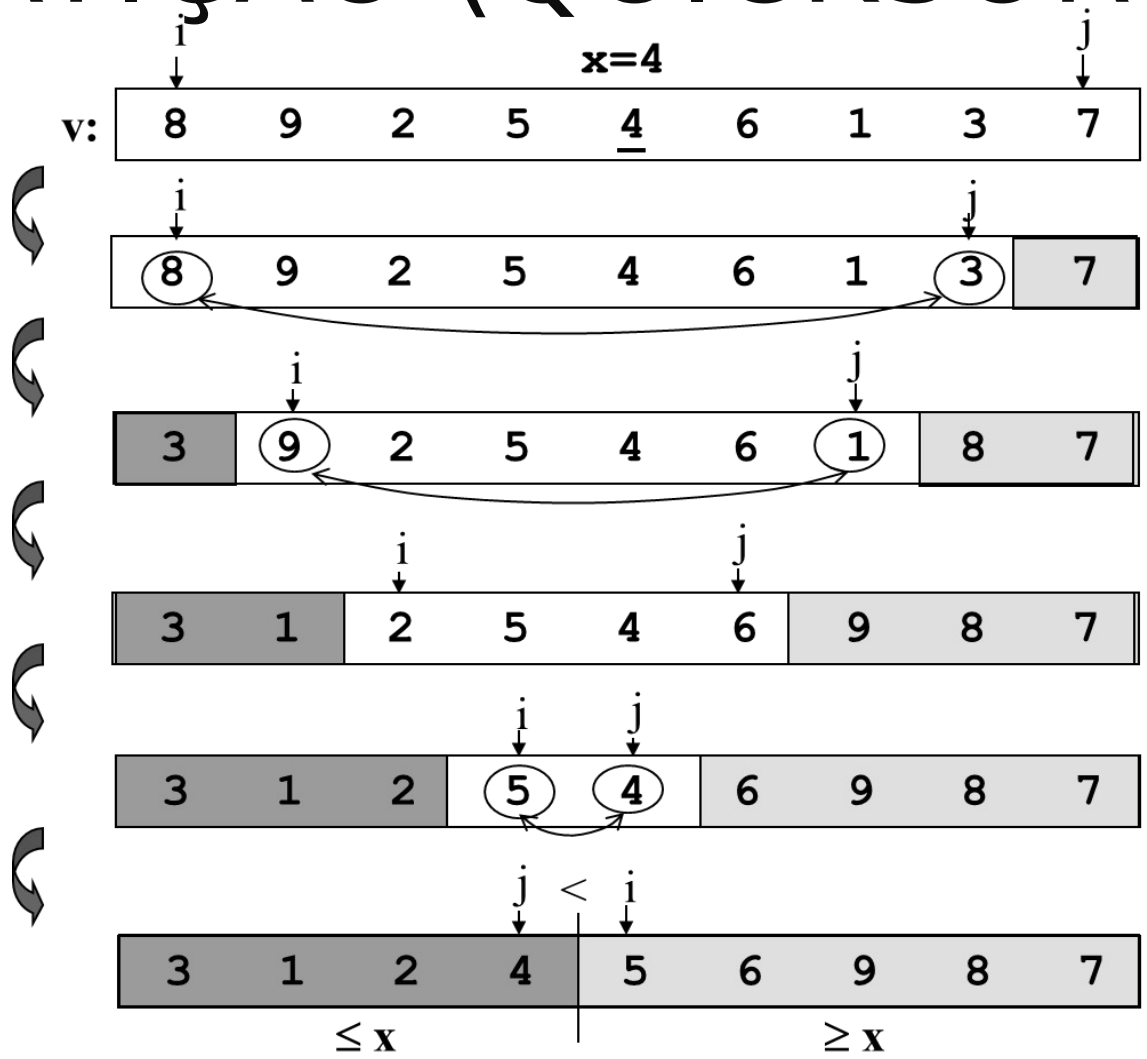
```
/* Ordena elementos do vetor v de inteiros. */

void ordenacaoInsercao(int v[], int tamanho)
{
    int i, j, tmp;
    for (i = 1; i < tamanho; i++)
    {
    tmp = v[i];
    for (j = i; j>0 && tmp<v[j-1]; j--)
       v[j] = v[j-1];
    v[j] = tmp;
    }
}
```

ORDENAÇÃO POR PARTIÇÃO (QUICKSORT)

- Algoritmo (ordenação por partição):
  1. Caso básico: Se o número (*n*) de elementos do vetor (*v*) a ordenar for 0 ou 1, não é preciso fazer nada
  2. Passo de partição:
     - 2.1. Escolher um elemento arbitrário (**x**) do vetor (chamado pivot)
     - 2.2. Partir o vetor inicial em dois sub-vetores (esquerdo e direito), com valores $\leq$ **x** no sub-vetor esquerdo e valores $\geq$ **x** no sub-vetor direito
  3. Passo recursivo: Ordenar os sub-vetores esquerdo e direito, usando o mesmo método recursivamente

- Algoritmo recursivo baseado na técnica *divisão e conquista*
  - nota: quando parte do vetor a ordenar é de dimensão reduzida, usa um método de ordenação mais simples (p.ex. 'insertionSort')

ORDENAÇÃO POR PARTIÇÃO (QUICKSORT)



ORDENAÇÃO POR PARTIÇÃO (QUICKSORT)

- Escolha pivot determina eficiência
  - *pior caso*: pivot é o elemento mais pequeno
    $O(N^2)$
  - *melhor caso*: pivot é o elemento médio
    $O(N \log N)$
  - *caso médio*: pivot corta vetor arbitrariamente
    $O(N \log N)$

- Escolha do pivot
  - um dos elementos extremos do vetor:
    - má escolha: $O(N^2)$ se vetor ordenado
  - elemento aleatório:
    - envolve uso de mais uma função pesada
  - mediana de três elementos (extremos do vetor e ponto médio)
    - recomendado

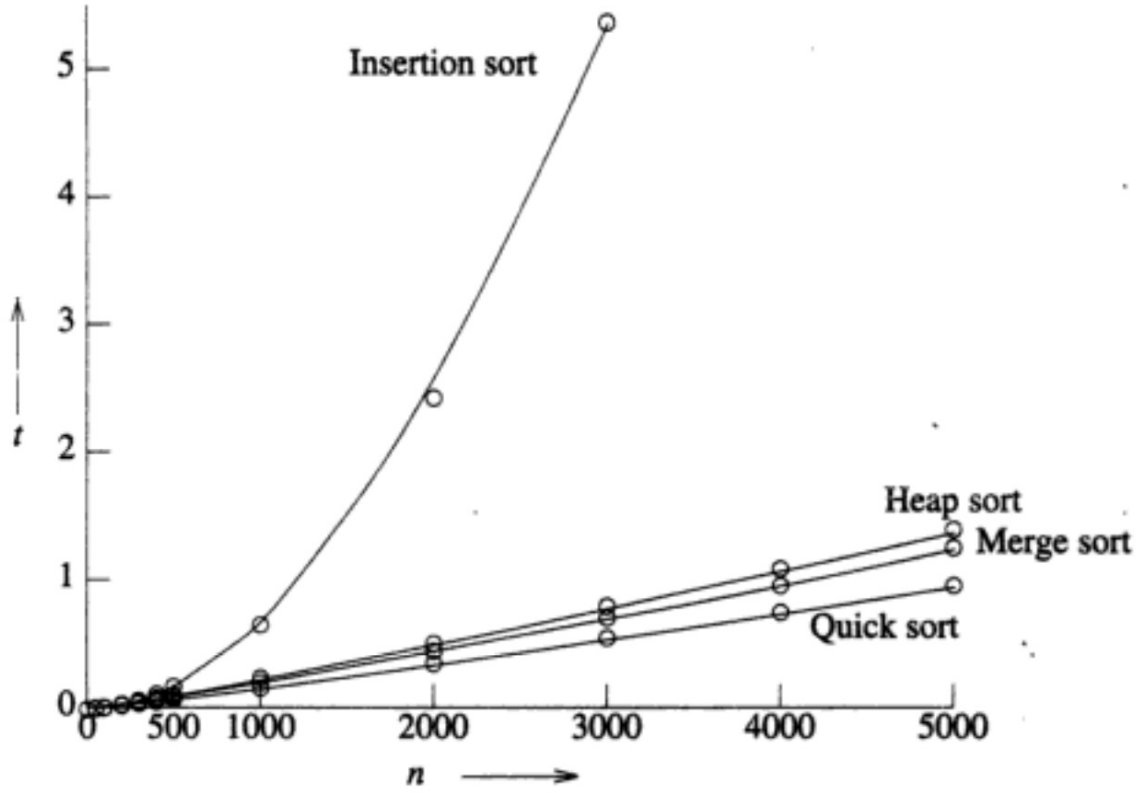
ORDENAÇÃO POR PARTIÇÃO (QUICKSORT)

```
/* Ordena elementos do vetor v de inteiros,
invocando a função recursiva com o protótipo
void quickSortIter(int v[], int left, int right)
mostrada no slide seguinte. */

void quickSort(int v[], int tamanho)
{
   quickSortIter(v, 0, tamanho-1);
}
```

ORDENAÇÃO POR PARTIÇÃO (QUICKSORT)

```
void quickSortIter(int v[], int left, int right) {
  int i, j, tamanho = right-left+1;
  if(tamanho<2) /* com tamanho 0 ou 1 esta ordenado */
    return;
  else {
    int pos = rand()%tamanho + left; /* determina pivot */
    swap(&v[pos], &v[right]);  /* coloca pivot no fim */
    i = left;   j = right-1;   /* passo de partição */
    while(1) {
    while (i < right && v[i] <= v[right]) i++;
    while (j >= 0 && v[right] <= v[j]) j--;
    if (i < j) swap(&v[i], &v[j]);
      else break;
    }
    swap(&v[i], &v[right]);  /* repoe pivot */
    quickSortIter(v, left, i-1);
    quickSortIter(v, i+1, right);
  }
}
```

ALGORITMOS DE ORDENAÇÃO



n	Insert	Heap	Merge	Quick
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Times are in milliseconds

Cada n corresponde à ordenação de 100 vetores de inteiros gerados aleatoriamente

Fonte: Sahni, "Data Structures, Algorithms and Applications in C++"

Método de ordenação por partição (*quickSort*) é na prática o mais eficiente, excepto para vetores pequenos, em que o método de ordenação por inserção (*insertionSort*) é melhor.

QUICKSORT MELHORADO

```
/* versao quicksort melhorada*/
void quickSortIter(int v[], int left, int right) {
   int i, j;
   if (right-left+1 <= 20)     /* se vetor pequeno */
      ordenacaoInsercao(v, left, right);
   else {
      int x = median3(v, left, right);  /* x é o pivot */
      i = left;   j = right-1;   /* passo de partição */
      while(1) {
         while (v[i] < x && i < right) i++;
         while (x < v[j] && j >= 0) j--;
         if (i < j) swap(&v[i], &v[j]);
         else break;
    }
      swap(&v[i], &v[right]);  /* repoe pivot */
      quickSortIter(v, left, i-1);
      quickSortIter(v, i+1, right);
   }
}
```

QUICKSORT MELHORADO

```
/* determina o valor do pivot como sendo a mediana de 3 valores:
elementos extremos e central do vetor */

int median3(int v[], int left, int right)
{
   int center = (left+right) /2;
   if (v[center] < v[left])
      swap(&v[left], &v[center]);
   if (v[right] < v[left])
      swap(&v[left], &v[right]);
   if (v[right] < v[center])
      swap(&v[center], &v[right]);
  /* coloca pivot na posicao right */
  swap(&v[center], &v[right]);
  return v[right];
}
```

QUICKSORT MELHORADO

```
/* com dimensão reduzida, método de ordenação mais simples */

void ordenacaoInsercao(int v[], int left, int right) {
    int i, j, tmp;
    for (i = left+1; i < right+1; i++) {
        tmp = v[i];
        for (j = i; j>0 && tmp<v[j-1]; j--)
           v[j] = v[j-1];
        v[j] = tmp;
        }
    }

    void swap(int *x, int *y) {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
    }
```