Busca e Ordenação

Rômulo César Silva

Unioeste

Junho de 2016

unioeste
Universidade Estadual do Oeste do Paraná

Sumário

Nota:

Alguns códigos aqui listados foram adaptados de:

Paulo Feofiloff. *Algoritmos em linguagem C*. Elsivier, Rio de Janeiro, 2009.

Busca

Busca ou Pesquisa

A **busca** ou **pesquisa** visa determinar se um elemento procurado está presente ou não em um conjunto/sequência de dados.

- normalmente a busca se faz por uma **chave**, isto é, um dado que identifica o elemento procurado
- o retorno da busca pode ser tanto um valor booleano (encontrado ou não encontrado) quanto outras informações associadas ao elemento procurado, dependendo da aplicação.

Busca - exemplo

```
#define TAM 100
typedef struct {
   int matricula;
   char nome[50];
   int idade;
   char sexo;
   } Aluno;
...
 Aluno vet[TAM];
 int i;
...
 int matricula_x;
...
 while(vet[i].matricula != matricula_x && i < TAM)
   i++;
...
```

Busca

Busca Sequencial

Consiste em percorrer o conjunto/sequência de dados sequencialmente à busca do elemento procurado.

```
// busca sequencial em um vetor
// Entrada: vetor, tamanho do vetor e
//          elemento a ser pesquisado
// Retorno: posição do elemento no vetor
int busca_sequencial(int vet[], int n, int x) {
    int i = 0;
    while (i < n && vet[i] != x)
        i++;
    return i;
}
```

Busca

Busca Sequencial

A pior situação da busca sequencial é quando o elemento buscado não está presente ou é o último. Assim, sendo n o número de elementos no conjunto/sequência, o número máximo de iterações é proporcional a n.

Busca

Vetor ordenado

Um vetor de inteiros $v[0..n-1]$ é **crescente** se $v[0] \leq v[1] \leq ... \leq v[n-1]$ e **decrescente** se $v[0] \geq v[1] \geq ... \geq v[n-1]$. O vetor é **ordenado** se for crescente ou decrescente.

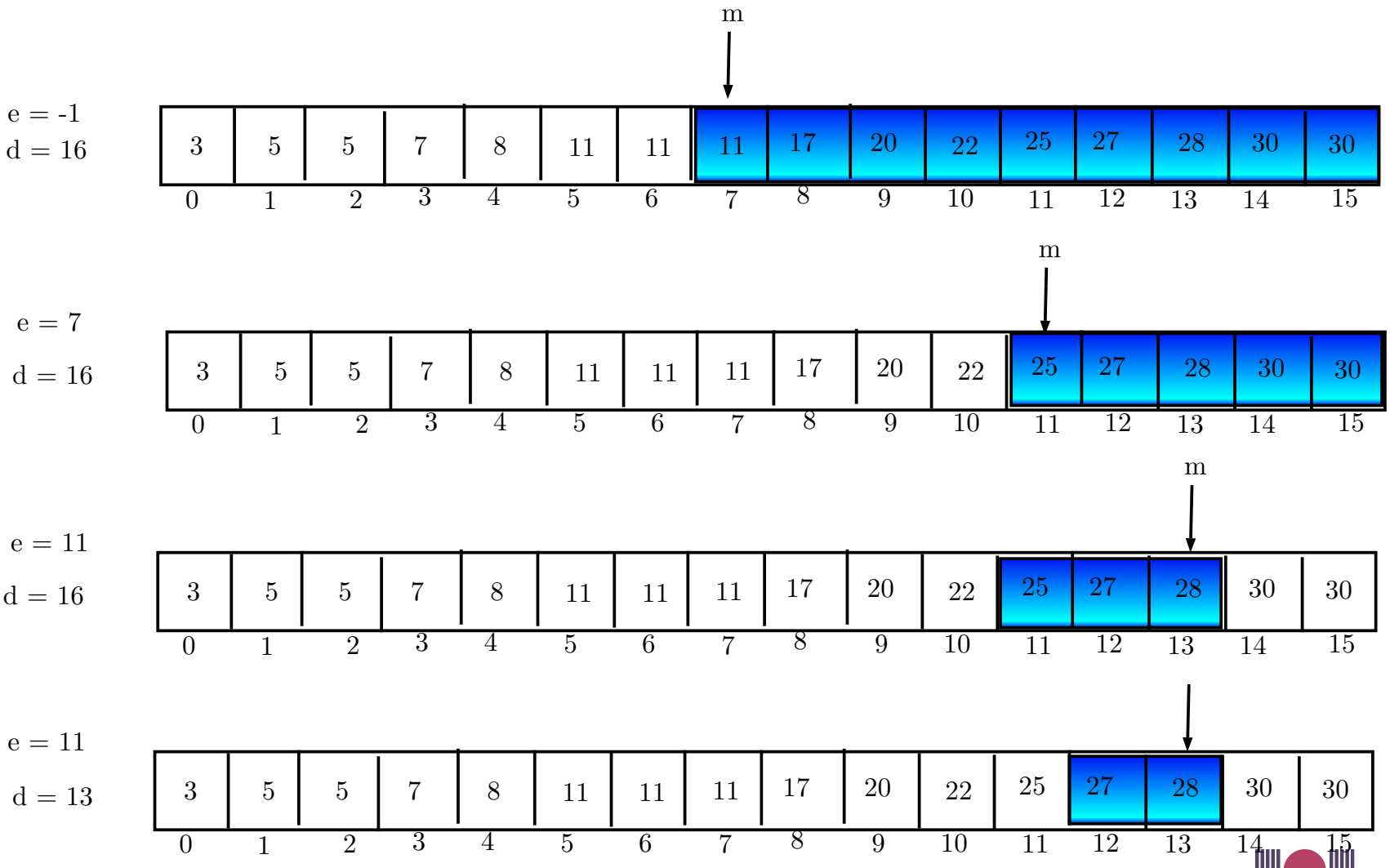
Se um vetor está ordenado, tal informação pode ser usada para facilitar o processo de busca?

Busca Binária

Semelhante ao processo que se usa para fazer busca de uma palavra no dicionário.

```
// versão iterativa
// Retorna a posição do elemento no vetor
int busca_binaria(int vet[], int n, int x) {
  int e, m, d;
  e = -1; d = n;
  while(e < d-1) {
     m = (e+d)/2;
     if (vet[m] < x)
       e = m;
     else
       d = m;
  }
  return d;
}
```

Busca Binária - exemplo



unioeste
Universidade Estadual do Oeste do Paraná

Busca Binária

```
// versão recursiva
// Retorna a posição do elemento no vetor
int busca_binaria_rec(int vet[], int n, int x) {
   return busca_binaria_aux(vet,-1,n,x);
}
int busca_binaria_aux(int vet[], int e, int d, int x) {
    if(e == d-1)
      return d;
    else {
        int m = (e+d)/2;
        if(v[m]< x)
          return busca_binaria_aux(vet,m,d,x);
        else
          return busca_binaria_aux(vet,e,m,x);
        }
}
```

Busca Binária

Analisando o algoritmo da Busca Binária, observa-se que a cada iteração ou chamada recursiva o conjunto de busca reduz-se à metade. Ou seja:

- Quando n $= 2$, o número iterações ou chamadas recursivas será no máximo 1.
- Quando n $= 4$, o número iterações ou chamadas recursivas será no máximo 2.
- Quando n $= 32$, o número iterações ou chamadas recursivas será no máximo 5.
- Quando n $= 64$, o número iterações ou chamadas recursivas será no máximo 6.

Isto é, o número de iterações ou chamadas recursivas é proporcional à $log_2 n$.

Busca Sequencial X Busca Binária

Seja n o número de elementos no vetor. O número de iterações (para o pior caso):

n	Busca Sequencial	Busca Binária
4	4	2
32	32	5
64	64	6
128	128	7
1024	1024	10

Ordenação

Problema da Ordenação

Rearranjar ou permutar os elementos de um vetor $vet[0..n-1]$ de tal modo que ele se torne crescente.

Ordenação

Existem vários métodos de ordenação. Alguns aspectos a serem levados em conta ao se escolher um método:

- os dados a serem ordenados estão na memória principal ou em memória secundária?
- no caso de elementos com chaves iguais é necessário preservar a ordem original?

Ordenação

Ordenação Interna/Externa

A ordenação feita toda em memória principal é dita ordenação interna. A ordenação feita em memória secundária é dita ordenação externa.

- Os métodos mais apropriados para ordenação externa são aqueles que realizam menos trocas de elementos.

Ordenação

Estabilidade

Um método de ordenação é dito **estável** se preserva a ordem original quando há elementos com chaves iguais.

Ex. de situação em que a estabilidade é importante: suponha uma estrutura de data com 2 campos: dia e mês. Ordenando-se primeiro pelo mês e depois uma nova ordenação pelo dia, a ordem cronológica das datas somente é garantida se o método de ordenação utilizado for estável!

- Há métodos não-estáveis que com algumas alterações tornam-se estáveis.

Ordenação

In-place

Um método de ordenação é dito **in-place** se a memória adicional requerida é independente do tamanho do vetor que está sendo ordenado.

Ordenação

Análise de Complexidade

A análise da complexidade dos algoritmos é feita em função do tamanho da entrada:

- **Memória**: se há necessidade de uso de vetores auxiliares ou não.
- **Tempo**: número de comparações e trocas efetuadas

Ordenação

Análise de Complexidade

A análise da complexidade de algoritmos em geral é feita para 3 situações:

- **pior caso**: é a mais interessante do seguinte ponto de vista: *se um algoritmo é eficiente no pior caso, ele o é para os demais*.
- **caso médio**: representa em geral as ocorrências mais esperadas para a entrada, dependendo da análise de probabilidades, nem sempre fácil de ser realizada.
- **melhor caso**: representa as condições em que o algoritmo faz menos operações.

Ordenação

Métodos de ordenação mais comuns usando comparação:

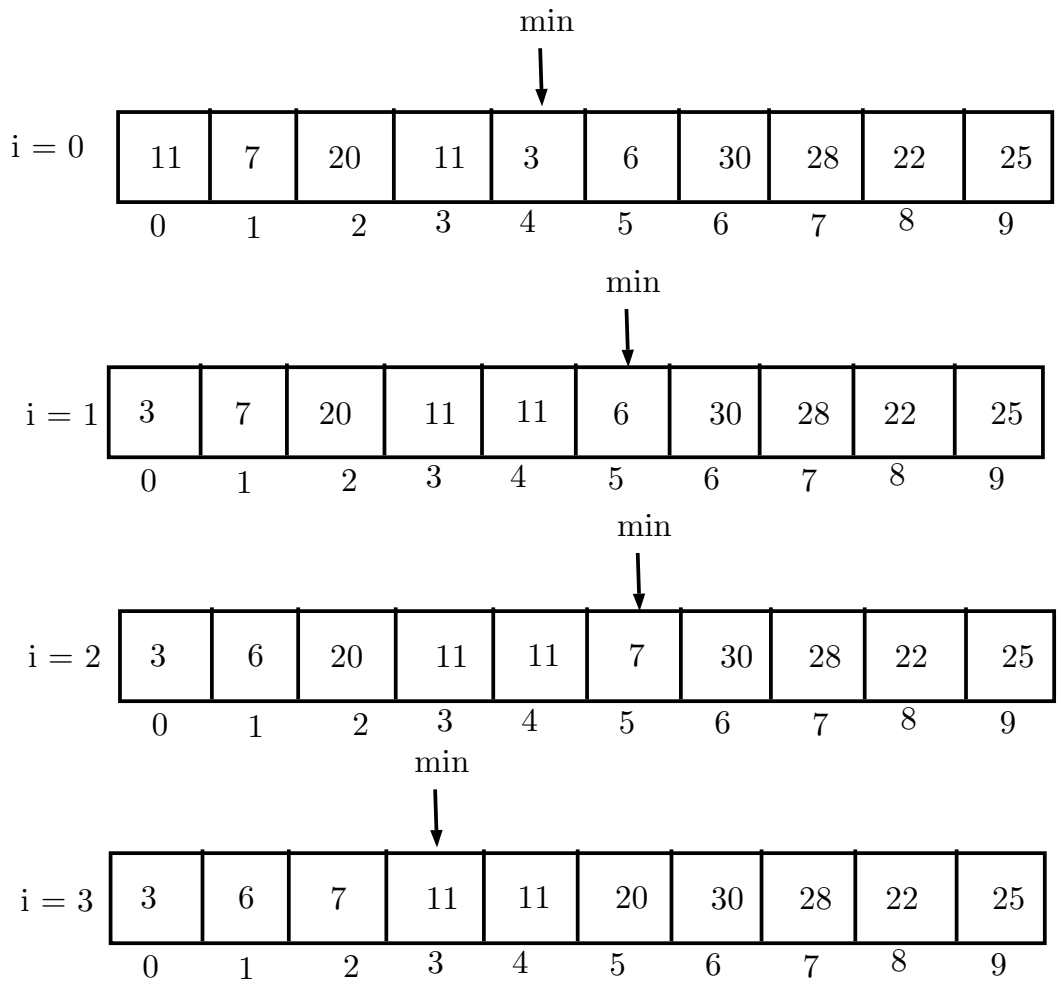
- SelectionSort ou Ordenação por Seleção
- InsertionSort ou Ordernação por Inserção
- BubbleSort ou Método da Bolha
- MergeSort ou Ordenação por Intercalação
- QuickSort
- ShellSort
- HeapSort

SelectionSort

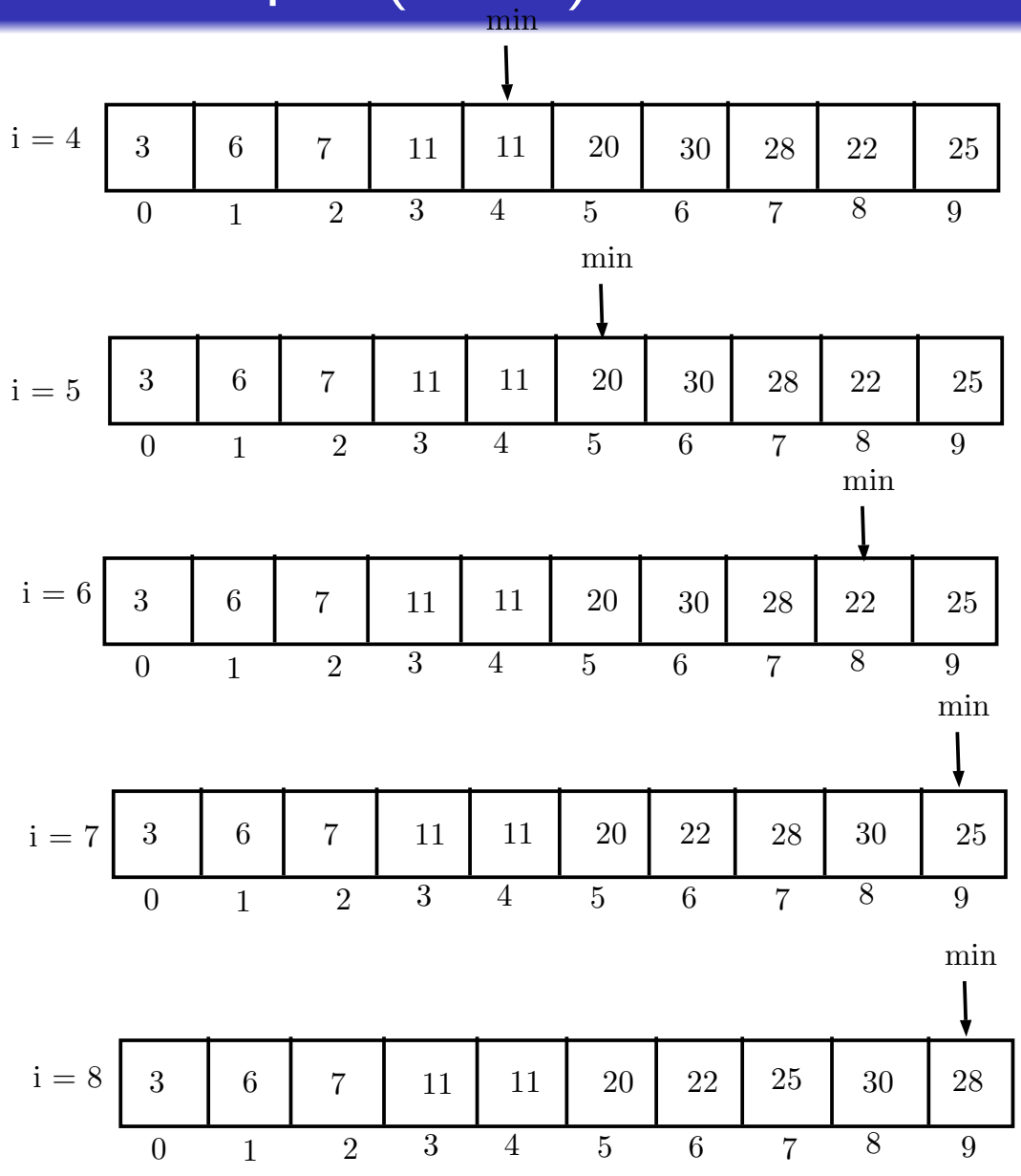
Baseado na ideia de escolher o menor elemento do vetor, depois o segundo menor, e assim por diante.

```
void SelectionSort(int vet[], int n) {
int i, j, min, x;
for ( i = 0; i < n-1; i++) {
  min = i;
  for (j = i+1; j < n; j++)
    if(vet[j] < vet[min]) min = j;
  //troca o elem da posicao i com o da posição min
  x = vet[i]; vet[i] = vet[min]; vet[min] = x;
  }
}
```

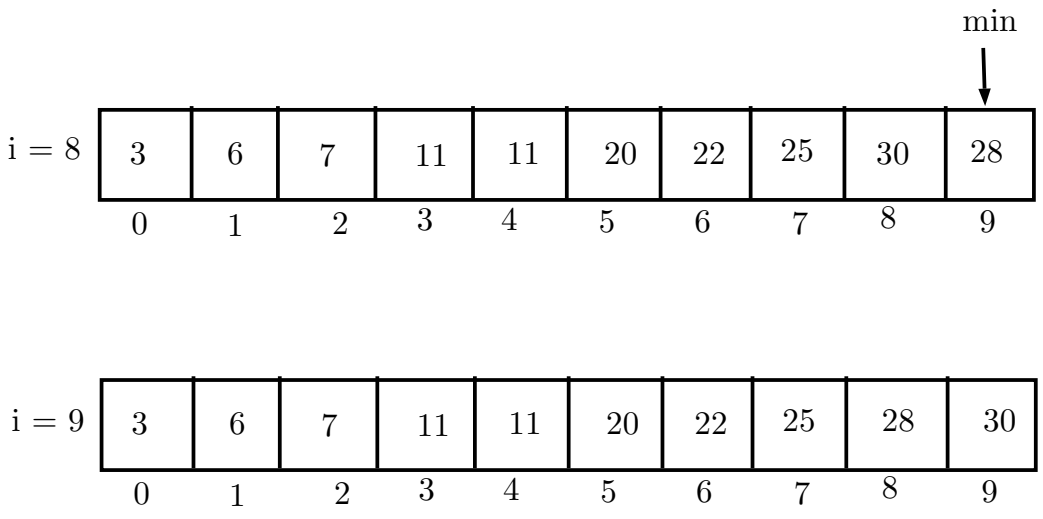
Selection Sort - exemplo



Selection Sort - exemplo (cont.)



Selection Sort - exemplo (cont.)



Selection Sort - análise de complexidade

No pior caso:

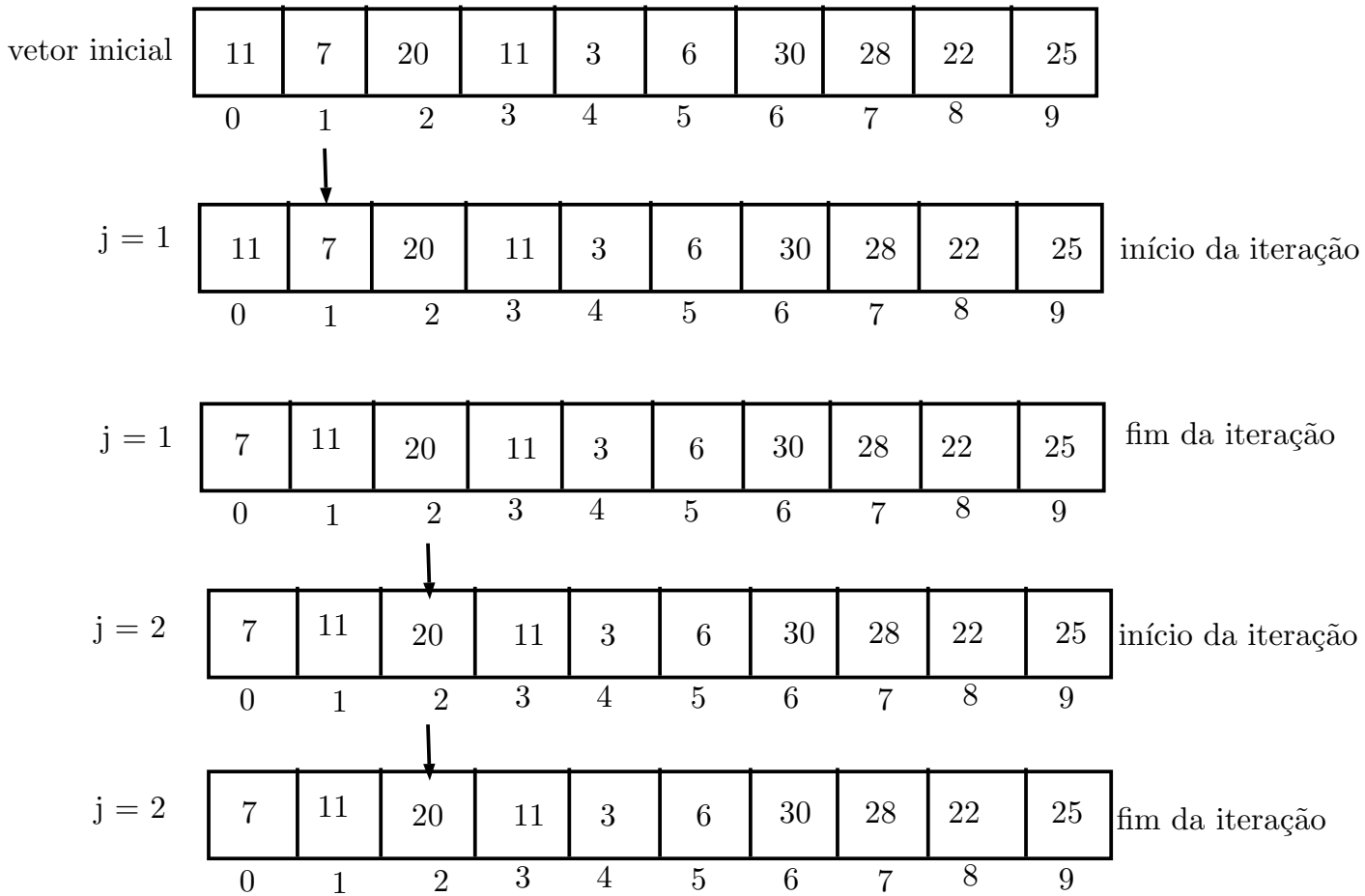
- Número de comparações: $O(n^2)$
- Número de trocas: $O(n)$

InsertionSort

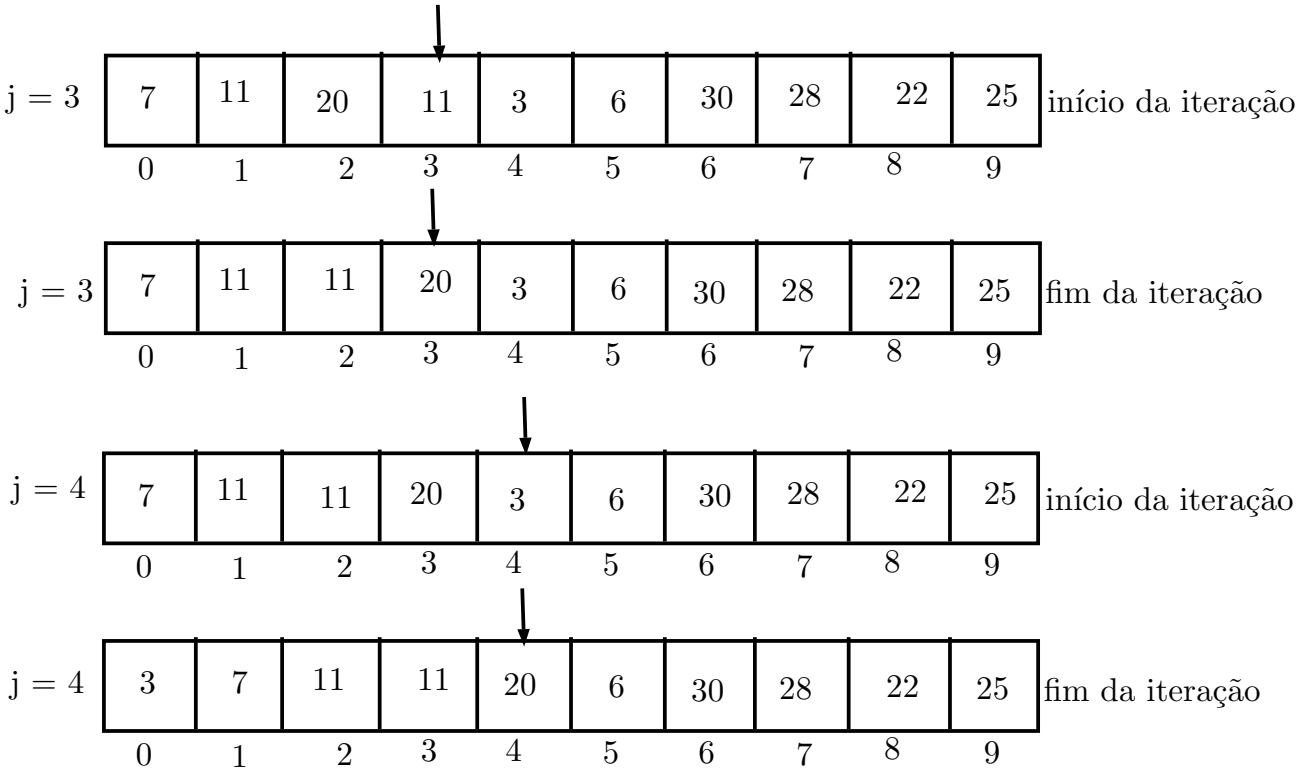
Baseado na ideia de colocar cada elemento na posição correta em relação aos seus antecessores.

```
void InsertionSort(int vet[], int n) {
int i, j, x;
for (j = 1; j < n; j++) {
  x = vet[j];
  for (i = j-1; i >= 0 && vet[i] > x; i--)
    vet[i+1] = vet[i];
  //coloca x na posição correta
  vet[i+1] = x;
  }
}
```

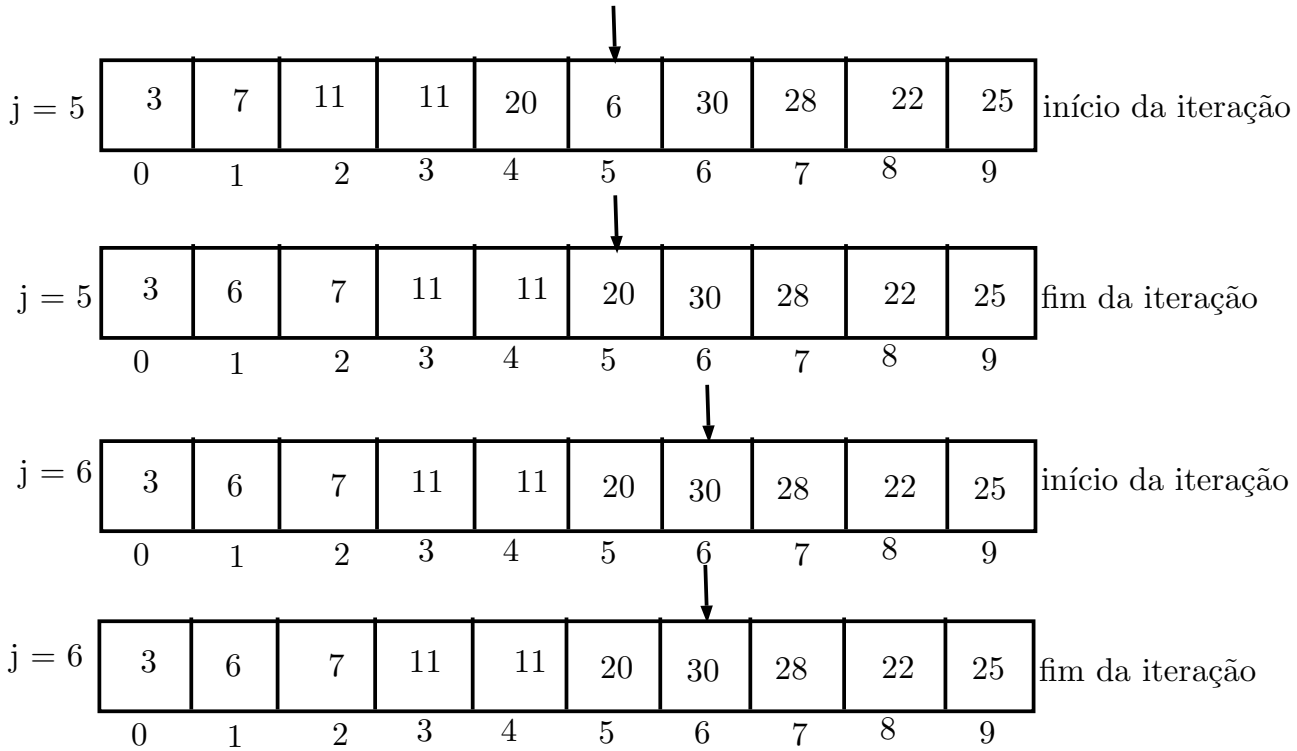
Insertion Sort - exemplo



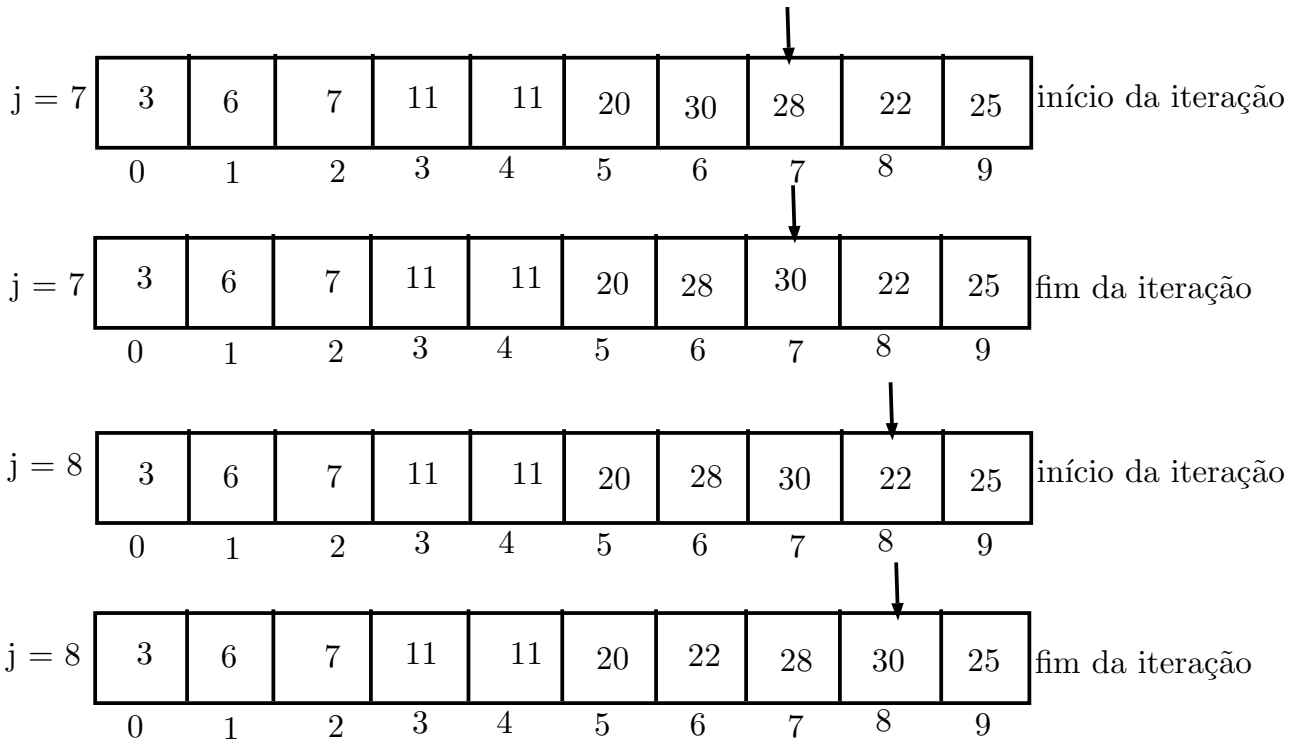
Insertion Sort - exemplo (cont.)



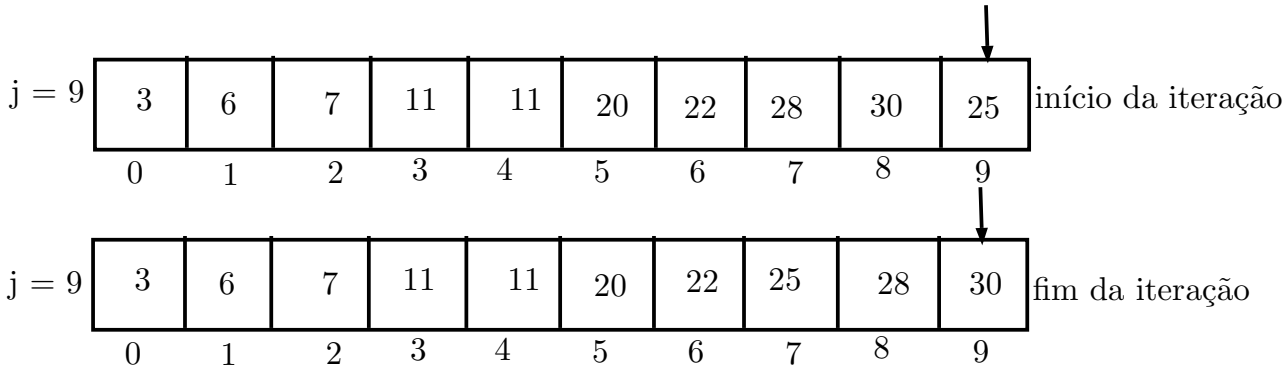
Insertion Sort - exemplo (cont.)



Insertion Sort - exemplo (cont.)



Insertion Sort - exemplo (cont.)



Insertion Sort - análise de complexidade

No pior caso:

- Número de comparações: $O(n^2)$
- Número de trocas: $O(n^2)$

Bubble Sort

Baseado na ideia de fazer trocas sucessivas tal que os elementos menores ("menos densos") vão para o início ("emergem") do vetor enquanto os maiores ("mais densos") vão para o fim ("submergem").

```
void Bubblesort(int vet[], int n){
     int i,j,temp;
     for(i = n - 1; i >= 0; i--)
        for(j = 1; j <= i; j++)
          if(vet[j-1] > vet[j]) {
             temp = vet[j-1];
             vet[j-1] = vet[j];
             vet[j] = temp;
          }
}
```

Bubble Sort - exemplo

11	7	20	11	3	6	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	20	11	3	6	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	20	11	3	6	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	11	20	3	6	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	11	3	20	6	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	11	3	6	20	30	28	22	25
0	1	2	3	4	5	6	7	8	9

Bubble Sort - exemplo (cont.)

7	11	11	3	6	20	30	28	22	25
0	1	2	3	4	5	6	7	8	9

7	11	11	3	6	20	28	30	22	25
0	1	2	3	4	5	6	7	8	9

7	11	11	3	6	20	28	22	30	25
0	1	2	3	4	5	6	7	8	9

7	11	11	3	6	20	28	22	25	30
0	1	2	3	4	5	6	7	8	9

...

Bubble Sort - análise de complexidade

No pior caso:

- Número de comparações: $O(n^2)$
- Número de trocas: $O(n^2)$

MergeSort

Baseado na ideia de ordenar cada metade do vetor separadamente e depois fazer a intercalação dos subvetores ordenados para obter o vetor totalmente ordenado.

```
//e: índice da esquerda
//d: índice da direita
// ordena o vetor subvetor vet[e..d].
// Chamada inicial: MergeSort(vet,0,n)
void MergeSort(int vet[],int e, int d) {
     if( e < d-1) {
        int m = (e+d)/2;
        MergeSort(vet,e,m);
        MergeSort(vet,m,d);
        Intercala(vet,e,m,d);
  }
}
```

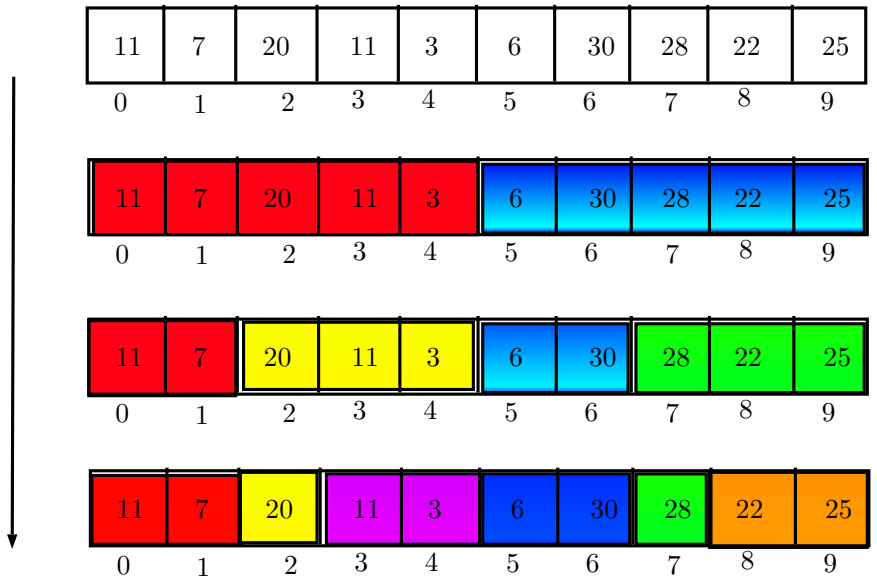
MergeSort

Intercalação no MergeSort:

```
void Intercala(int vet[],int e, int m, int d) {
  int i, j, k, *vet_aux;
  vet_aux = malloc((d-e)*sizeof(int));
  i = e; j = m; k = 0;
  while( i < m && j < d) {
    if(vet[i] <= vet[j]) vet_aux[k++] = vet[i++];
    else vet_aux[k++] = vet[j++];
  }
  while (i < m) vet_aux[k++] = vet[i++];
  while (j < d) vet_aux[k++] = vet[j++];
  for(i = e; i < d; i++) vet[i] = vet_aux[i-e];
  free(vet_aux);
}
```

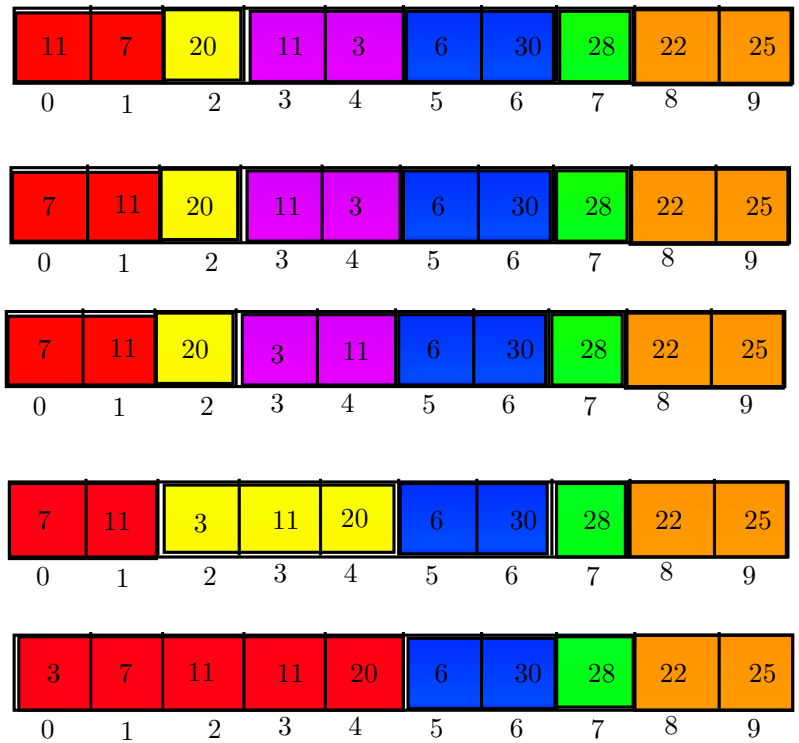
Mergesort - exemplo

Fase de divisão:



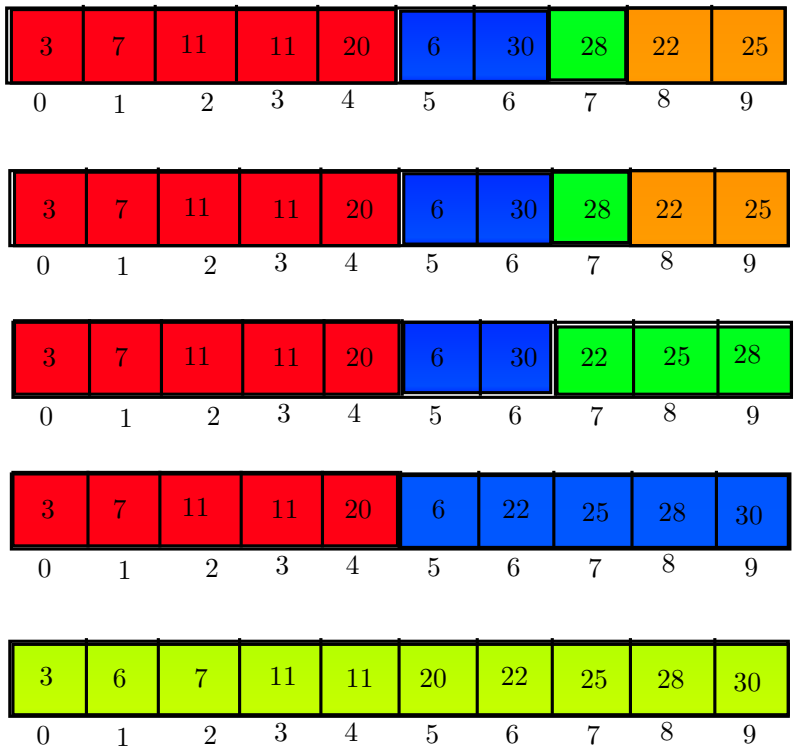
Mergesort - exemplo (cont.)

Fase de intercalação:



Mergesort - exemplo (cont.)

Fase de intercalação:



Mergesort - análise de complexidade

No pior caso:

- Número de comparações: $O(n \lg n)$
- Número de trocas: $O(n \lg n)$

Obs.: Há gasto de memória adicional com o uso de vetor auxiliar.

QuickSort

Baseado na ideia de escolher um elemento x do vetor, denominado *pivô*, e separar os demais em duas partes: os menores ou iguais a x, e o maiores que x. Ordenar cada parte separadamente aplicando o mesmo processo e depois concatenar as duas partes tal que x fique entre a primeira e a segunda parte ordenada.

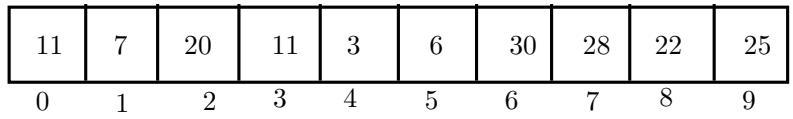
```
// versão recursiva
// chamada inicial: Quicksort(vet,0,n-1)
void QuickSort(int vet[],int e, int d) {
     int j;
     if( e < d) {
        j = Separa(e,d,vet);
        QuickSort(vet,e,j-1);
        QuickSort(vet,j+1,d);
  }
}
```

QuickSort

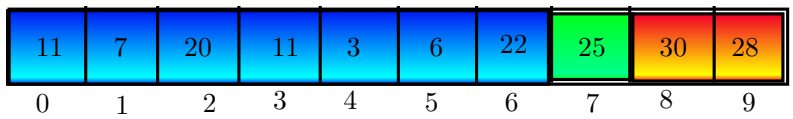
Processo de separação dos elementos em relação ao pivô:

```
// Rearranja vet tal que:
//       vet[e..j-1] <= vet[j] < vet[j+1...d]
// retorna a posição correta do pivô
void Separa(int vet[],int e, int d) {
     int pivo,j,k,temp;
     pivo = vet[d]; j = e;
     for( k = e; k < d; k++)
       if(vet[k] <= pivo) {
         temp = vet[j]; vet[j] = vet[k]; vet[k] = temp;
         j++;
       }
     vet[d] = vet[j]; vet[j] = pivo;
     return j;
}
```

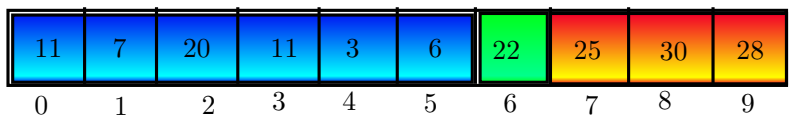
Quicksort - exemplo



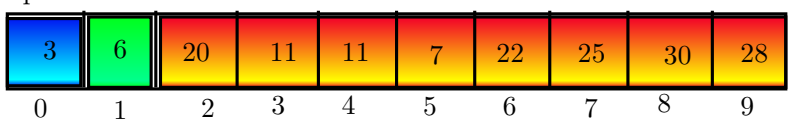
pivô = 25



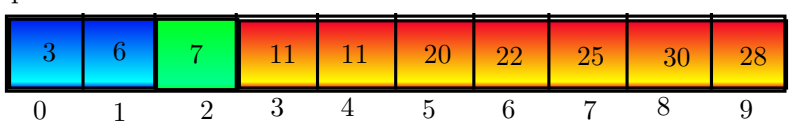
pivô = 22



pivô = 6

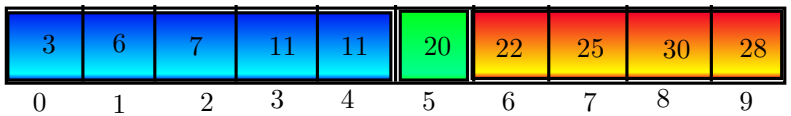


pivô = 7

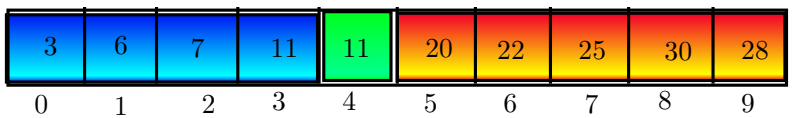


Quicksort - exemplo (cont.)

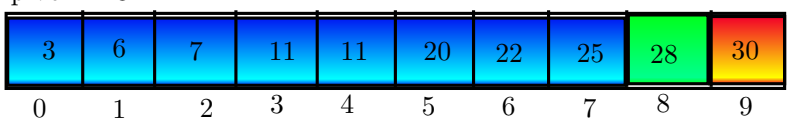
pivô = 20



pivô = 11



pivô = 28



Quicksort - análise de complexidade

No pior caso (ocorre quando o vetor já está quase ordenado):

- Número de comparações: $O(n^2)$
- Número de trocas: $O(n^2)$

No caso médio:

- Número de comparações: $O(n \lg n)$
- Número de trocas: $O(n \lg n)$

Obs.: A escolha do pivô influencia a eficiência do algoritmo. A melhor situação é quando o pivô escolhido divide igualmente o subvetor analisado, isto é, sua posição fica a meio do caminho.

Quicksort - implementações

Existem várias versões de implementação do *Quicksort*, tanto recursivas quanto iterativas. Abaixo um exemplo de outra implementação. Pesquise na Internet outras versões.

```
void QuicksortCLR(int vet[], int p, int r) {
  int pivo = vet[p]; i = p -1; j = r+1; temp;
  if(p < r) {
    while(1) {
      do --j; while (vet[j] > pivo);
      do ++i; while (vet[i] < pivo);
      if(i >= j) break;
      temp = vet[i], vet[i] = vet[j], vet[j] = temp;
      }
      QuicksortCLR(vet,p,j);
      QuicksortCLR(vet,j+1,r);
   }
}
```

Heapsort

Utiliza uma estrutura de dados denominada *heap*, que simula uma árvore binária. Será visto apenas mais adiante, após o estudo de árvore binária.

Animações de Algoritmos de Ordenação

Acesse os links:

- http://nicholasandre.com.br/sorting
- http://www.sorting-algorithms.com

Heapsort

Heapsort

O algoritmo **Heapsort** usa uma estrutura de dados denominada **heap** para fazer a ordenação. Aqui, iremos supor que os índices do vetor são $1..n$ e não de $0..n$.

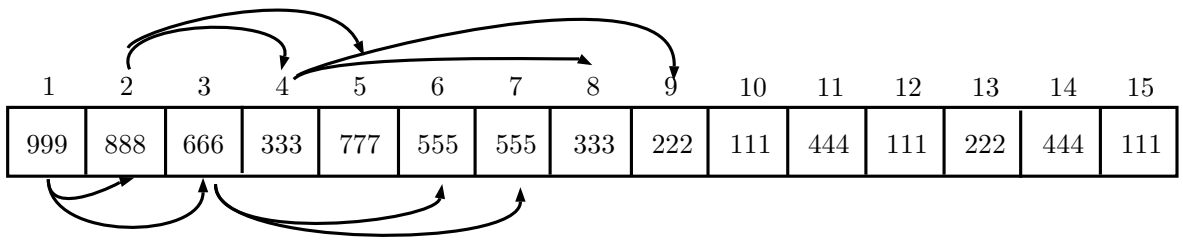
Heap

Heap

Um *heap* é uma estrutura de dados baseada em vetor, podendo ser de 2 tipos:

1. **max-heap**: é um vetor $v[1..m]$ tal que $v[\lfloor \frac{1}{2} f \rfloor] \geq v[f]$ para $f = 2, 3, ..., m$.
2. **min-heap**: é um vetor $v[1..m]$ tal que $v[\lfloor \frac{1}{2} f \rfloor] \leq v[f]$ para $f = 2, 3, ..., m$.

Max-Heap - exemplo



Heapsort

Heapsort

O **Heapsort** baseia-se na construção/rearranjo de um *heap* e sucessivas trocas com o elemento da primeira posição.

Observe que em um **max-heap** $v[1..m]$ o maior elemento está na posição 1. Assim, já se sabe que sua posição será a última, podendo então trocá-lo com o elemento da última posição. Em seguida monta-se um max-heap considerando até a penúltima posição e novamente troca-se o elemento da primeira com a penúltima posição. E assim por diante.

Heap

Heap

Os índices $1..m$ de um *heap* pode ser visto como uma árvore binária tal que:

- o índice 1 é a raiz da árvore
- o **pai** de um índice f é $\lfloor \frac{1}{2} f \rfloor$
- o **filho esquerdo** de um índice p é $2p$ e o **filho direito** é $2p + 1$

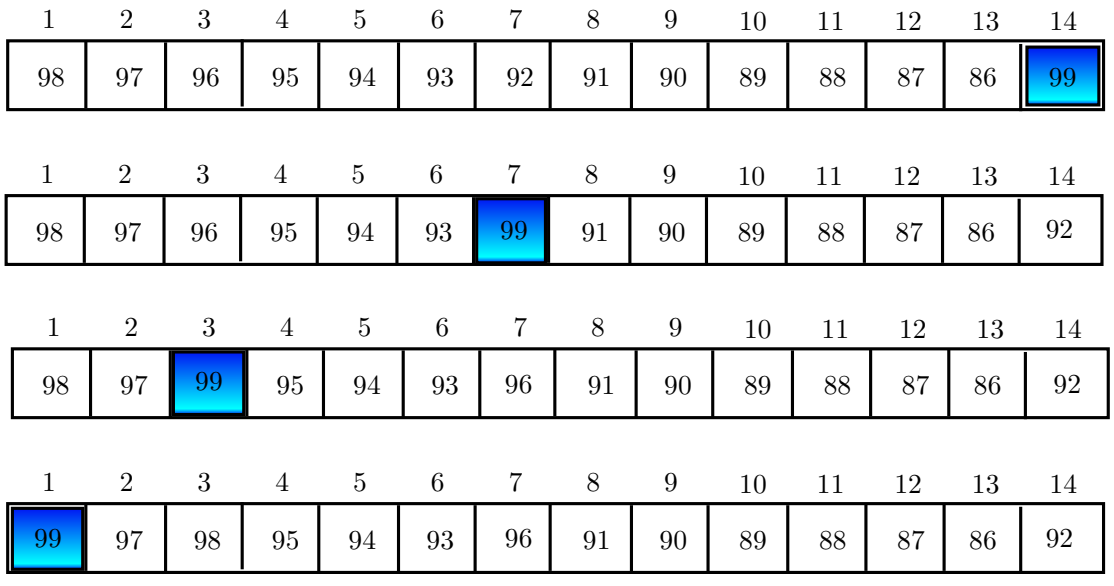
Heap - inserção

A inserção é feita "subindo" em direção à raiz procurando a posição em que o elemento deve entrar.

```
// insere  vet[m+1] no max-heap vet[1..m]
// tornand vet[1..m+1] um max-heap
void insereHeap(int m, int vet[]) {
  int f = m+1;
  while (f > 1 && vet[f/2] < vet[f]) {
    int t = vet[f/2]; vet[f/2] = vet[f]; vet[f] = t;
    f = f/2;
  }
}
```

Max-Heap - exemplo de inserção

Inserção de *vet*[14] no *max-heap vet*[1..13]:



Heapsort - função auxiliar

```
// rearranja  vet[1..m] p/ ser um max-heap
void sacodeHeap(int m, int vet[]) {
  int t, f = 2;
  while (f <= m) {
    if(f < m && vet[f] < v[f+1]) ++f;
    if(vet[f/2] >= vet[f]) break;
    t = vet[f/2]; vet[f/2] = vet[f]; vet[f] = t;
    f *= 2;
  }
}
```

Heapsort

```
void Heapsort(int n, int vet[]) {
  int m;
  for(m = 1; m < n; m++)
    insereHeap(m,vet);
  for(m = n; m > 1; m--) {
    int t = vet[1]; vet[1] = vet[m]; vet[m] = t;
    sacodeHeap(m-1,vet);
  }
}
```

Bibliografia I

[Feofiloff 2009] Paulo Feofiloff.
Algoritmos em linguagem C. Elsivier, Rio de Janeiro, 2009.

[Cormen 1997] Cormen,T.; Leiserson, C.; Rivest, R.
Introduction to Algorithms. McGrawHill, New York, 1997.