



Algoritmos de Ordenação

- **Problema:** encontrar um número de telefone em uma lista telefônica
 - simplificado pelo fato dos nomes estarem em ordem alfabética
 - e se estivesse sem uma ordem?
- **Problema:** busca de um livro em uma biblioteca
 - a cada livro é atribuída uma posição relativa a outros e portanto pode ser recuperado em um tempo hábil



Algoritmos de Ordenação

Terminologia básica

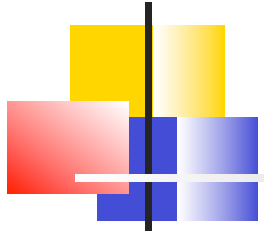
- seqüência de n ítems ou elementos (registros)
 - $r[0], r[1], \dots, r[n-1]$
- a chave $a[i]$ está associada ao elemento $r[i]$
 - usualmente a chave é um campo do registro
 - os elementos estão classificados por pela chave

se $i < j$ então $a[i]$ precede $a[j]$



Terminologia básica

- a ordenação tem por objetivo reorganizar as chaves de forma que obedecem a uma regra (ex.: ordem numérica ou alfabética)
- se os elementos representam registros de um arquivo, sendo representado por uma lista sequencial em memória principal
 - a ordenação é interna
- se o arquivo estiver em memória secundária
 - a ordenação é externa



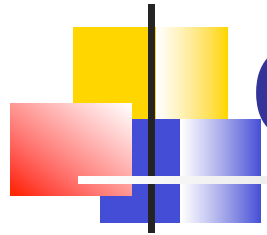
Terminologia básica

- a ordenação é **estável** se preserva a ordem relativa das chaves iguais (senão, **instável**)
- os algoritmos podem operar sobre o elemento ou sobre uma tabela de ponteiros
 - registro é muito grande: ordenação **indireta**
 - os registros não são rearrumados e sim os índices
 - as chaves tanto podem ficar armazenadas com os registros ou com os ponteiros

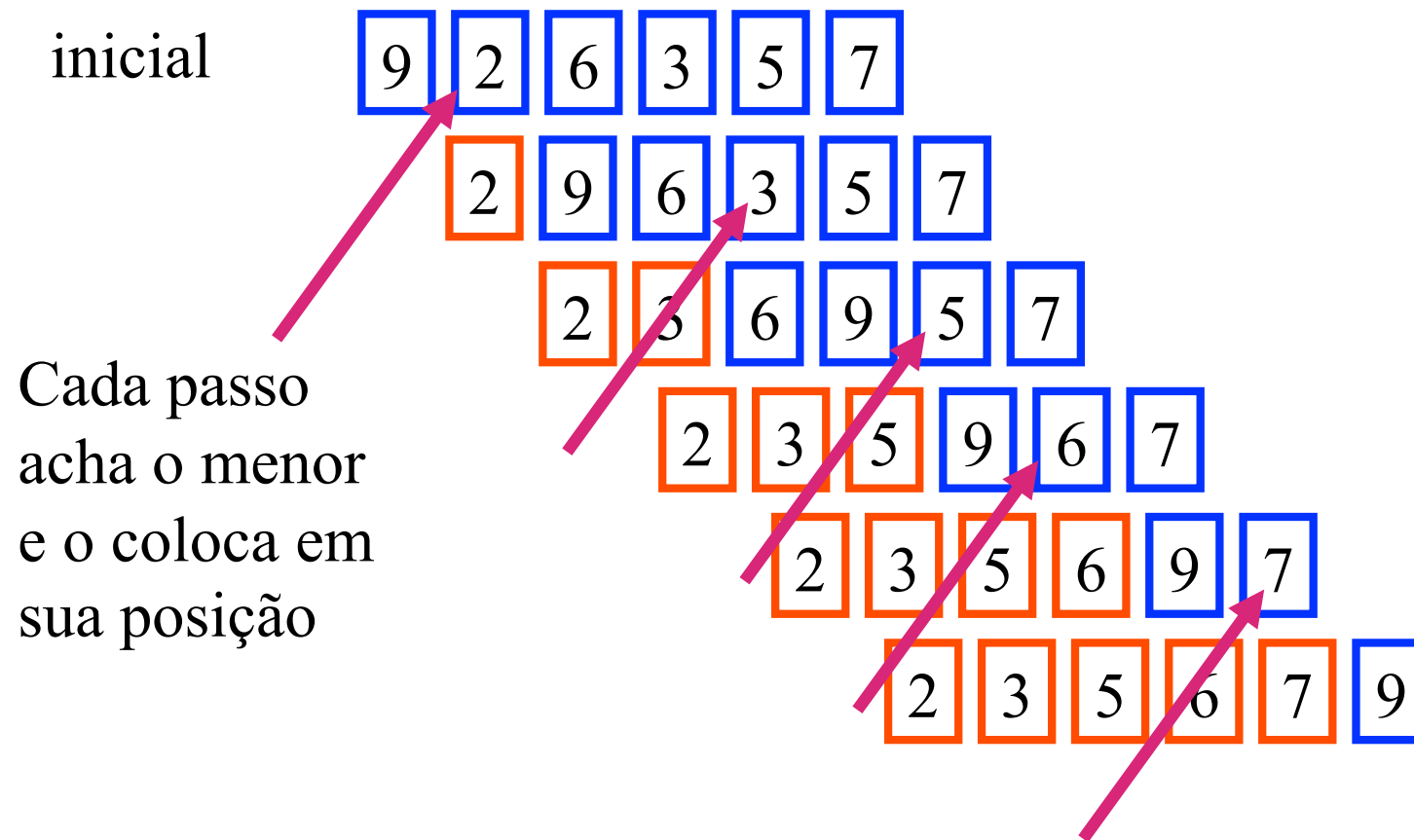


Ordenação por Seleção

- Um dos algoritmos de sort mais simples:
 - ache **primeiro** o menor elemento da lista e troque sua posição com o **primeiro** elemento
 - ache o **segundo** menor elemento e troque sua posição com o **segundo** elemento
 - continue até que toda lista esteja ordenada
 - para cada i de 0 a $n-2$, troca o menor elemento da sub-lista que vai de i a $N-1$ com $a[i]$



Ordenação por Seleção





Sort por Seleção

Ord_Seleção()

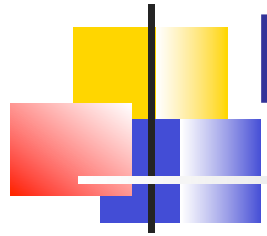
```
{ int t, i, j, min;
  for (i=0; i < n-2; i++){
    min = i;
    for (j=i+1; j < n-1; j++){
      if (a[j] < a[min]) min = j;
    }
    /* troca */
    t = a[min];
    a[min] = a[i];
    a[i] = t;
  }
}
```

complexidade?

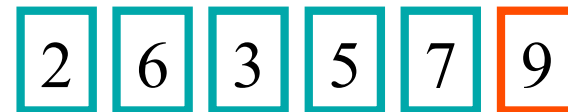
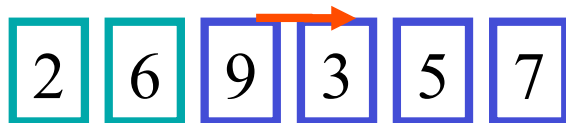
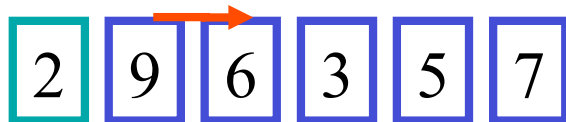
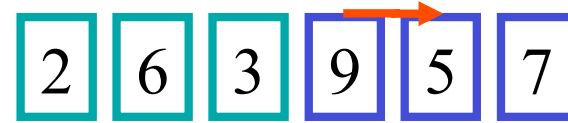
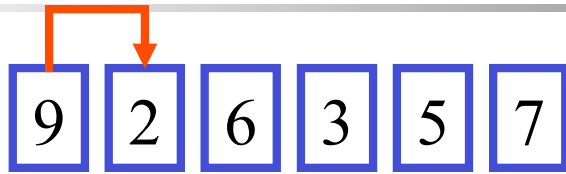


Bubble Sort

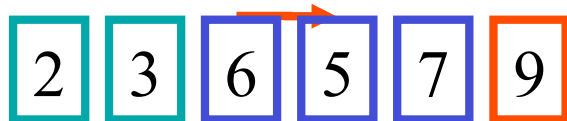
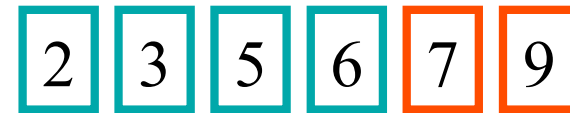
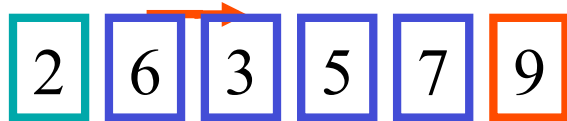
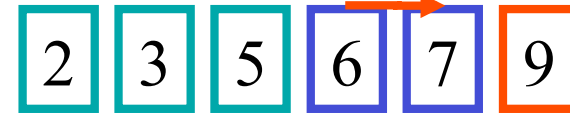
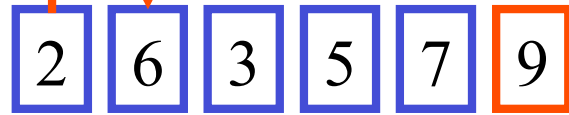
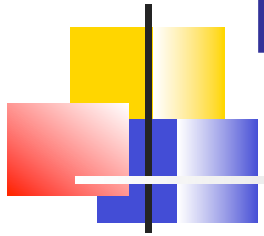
- percorrer a lista trocando elementos adjacentes, se necessário
- quando nenhuma troca for efetuada ao percorrer toda a lista → está classificada

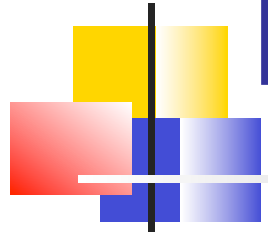


Bubble Sort

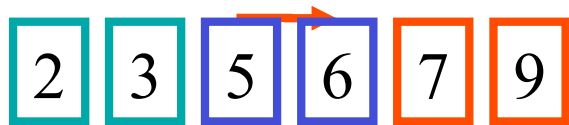
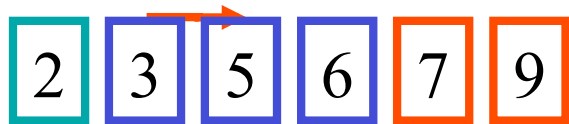
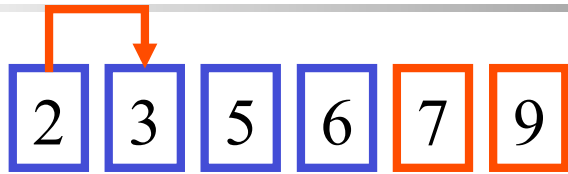


Bubble Sort





Bubble Sort





Bubble Sort()

```
int i, j, t, trocas;
for (i = n-1; i>0; i--) {
    trocas = 0;
    for (j = 1; j <= i; j++){
        if (a[j-1] > a[j]){
            /* troca */
            t = a[j-1]; a[j-1] = a[j]; a[j] = t;
            trocas++;
        }
    }
    if (trocas == 0) break;
}
}
```

complexidade?



Pensar

- escreva um algoritmo de *merge* de duas listas sequenciais ordenadas, analisando sua complexidade
- escreva um algoritmo que some dois polinômios, sendo estes, representados por listas sequenciais.
 - especifique as listas
 - a complexidade

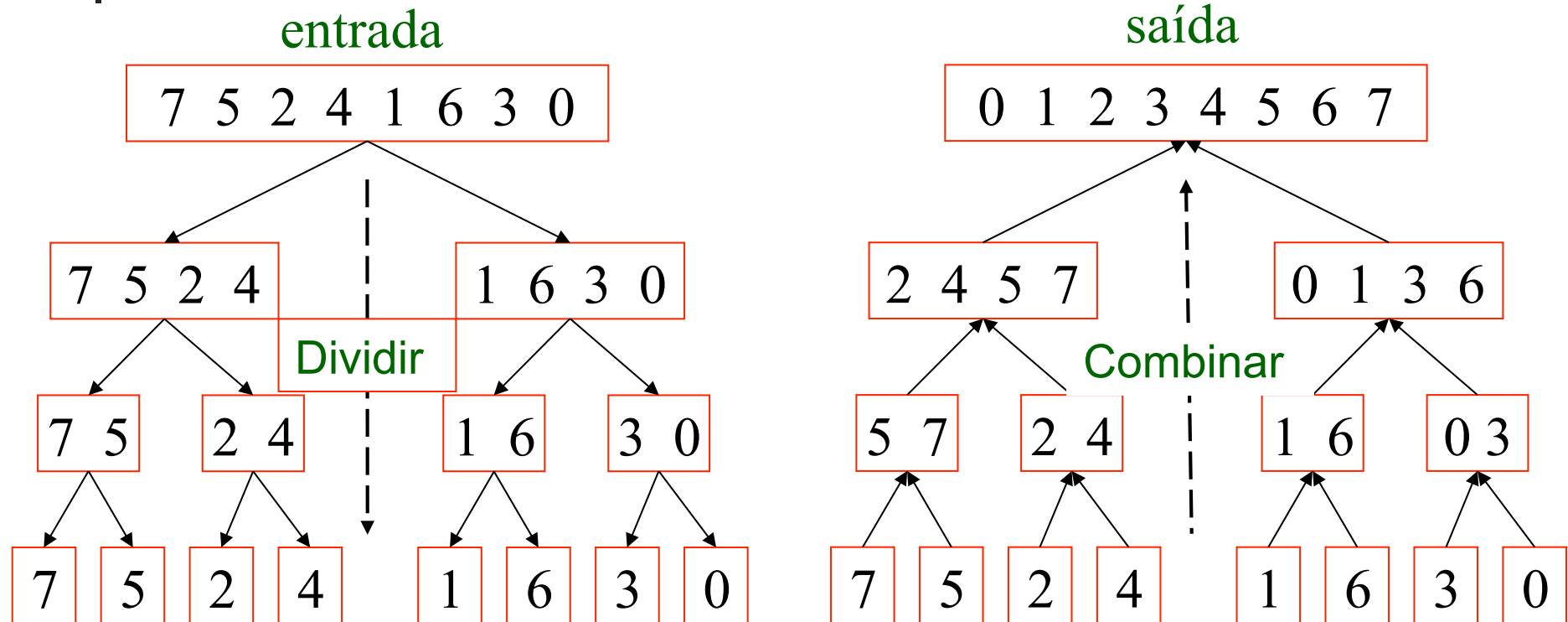


MergeSort

- Seja uma lista A de n elementos. O algoritmo consiste das seguintes fases
 - **Dividir** A em 2 sub-listas de tamanho $\approx n/2$
 - **Conquistar**: ordenar cada sub-lista chamando *MergeSort* recursivamente
 - **Combinar** as sub-lista ordenadas formando uma única lista ordenada
- **caso base**: lista com um elemento



MergeSort





MergeSort

MergeSort ordena as posições $e, e+1, \dots, d-1, d$ da lista A

```
MergeSort (e,d)
{
  if (e < d ){
    meio = (e+d)/2;
    MergeSort (e, meio);
    MergeSort (meio+1, d);
    Merge (e, meio, d);
  }
```




MergeSort

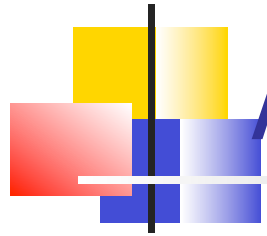
Merge (e, m, d) {
???

}



MergeSort

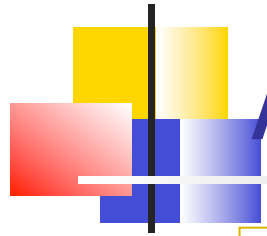
- algoritmo de ordenação estável
 - Se dois elementos são iguais eles nunca são trocados de ordem
 - Importante, por exemplo, se os elementos já estão ordenados segundo alguma chave secundária
- poderíamos eliminar a lista **B**?
- a copia final de **B** para **A** é necessária?



Análise da Complexidade

- Merge:
 - Cada chamada une um total de $d - e$ elementos, que no máximo, é n
 - uma comparação a cada iteração
 - total de passos do primeiro **while**: não mais que n
 - do outros, no pior caso, $n/2$
 - a cópia de A para B – n passos

Logo, complexidade é $O(n)$



Análise da Complexidade

- MergeSort

- Para $n = 1$, $T(1) = 1$
- Para $n > 1$, executa-se recursivamente:
 - uma vez *piso*($n/2$) com elementos
 - outra vez com os *teto*($n/2$) elementos restantes
 - Merge, que executa n operações

$$T(n) = T(n/2) + T(n/2) + n$$



Recursividade

Uma escada é igual a um degrau seguido de uma escada.

Descrição recursiva

fatorial de um número **n** é o produto de todos os números compreendidos no intervalo que vai de um até **n**.

notação mais formal:

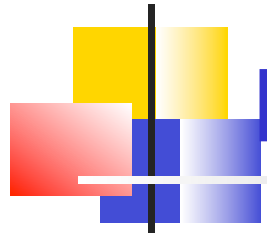
$$n! = 1 * 2 * 3 * \dots * n$$



Recursividade

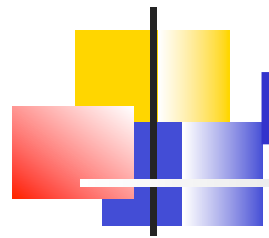
O Fatorial de um número natural n é :

- igual a 1 **se $n=0$** ;
- igual ao produto deste número pelo fatorial de seu antecessor, **se $n > 0$**



Elementos da Descrição Recursiva

- **Definição geral** : Toda definição recursiva tem duas partes,
 - se aplica a um valor qualquer do domínio do problema, onde o conceito que está sendo definido deve ser utilizado.
fatorial de n - usa-se o fatorial do *antecessor de n* (valor mais simples)
- **Definição independente** : um valor tão simples que a sua definição é dada de forma independente - base da recursão
- **Obtenção de valores mais simples** : uma função deve ser usada
fatorial, subtração de n por 1 - antecessor de n
- **Função auxiliar** : para obter um valor usando o valor considerado e o valor definido recursivamente.
no fatorial - função de multiplicação.



Elementos da Descrição Recursiva

- **Garantia de atingir o valor independente** : É fundamental que a aplicação sucessiva da função chegue à base da recursão.

Esta condição é fundamental para garantir que ao avaliarmos uma expressão atingiremos a base da recursão.



Recursividade

passo	redução	justificativa
0	fat 5	expressão proposta
1	5 * fat 4	substituindo fat pela definição geral
2	5*(4 * fat 3)	idem
3	5*(4* (3 * fat 2))	idem
4	5*(4*(3*(2 * fat 1)))	idem
5	5*(4*(3*(2*(1 * fat 0))	idem
6	5*(4*(3*(2*(1 * 1))))	usando a definição específica
7	5*(4*(3*(2*1)))	usando a primitiva de multiplicação
8	5*(4*(3*2))	idem
9	5*(4*6)	idem
10	5 * 24	idem
11	120	idem



Recursividade

- Descrever a função que determina o elemento de valor máximo uma lista de números.
- Descrever a função que verifica se um dado valor ocorre em uma lista.



Recorrência

- O tempo de execução de um algoritmo recursivo pode ser descrito por uma **recorrência**
 - equação ou inequação que descreve uma função em termos de sua entrada para valores pequenos

Recorrência que descreve a `MergeSort()` (outra forma de resolver)

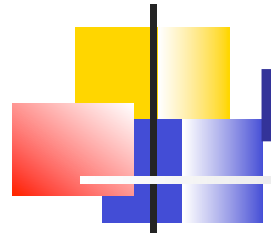
$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + O(n) & \text{se } n > 1 \end{cases}$$

????



Recorrência

- Existem três métodos para resolver funções de recorrência
 - por substituição: intuitivo, utiliza-se indução matemática
 - iterativo: a função é substituída por uma série de somas
 - mestre: limites são definidos para a função



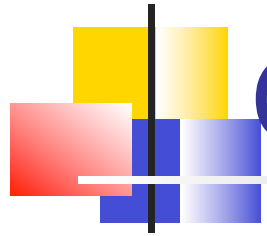
Resolvendo Recorrência

- $T(n) = \begin{cases} O(1) & \text{se } n = 0 \\ 2T(n - 1) + 1 & \text{se } n > 0 \end{cases}$



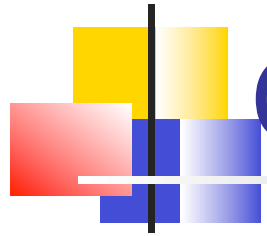
Quicksort

- eficiente
- naturalmente recursivo - implementação seria "complicada" sem recursão
- Algoritmo Básico
 - "dividir para conquistar"
 - particiona a lista de elementos em duas partes e as classifica independentemente
 - a posição exata da partição depende da lista



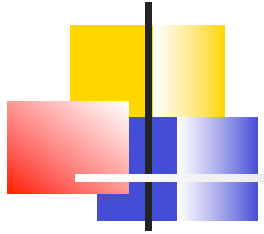
Quicksort: algoritmo básico

```
quicksort (int e, int d)
{
  int i;
  if (d > e){
    i = partition (e,d); /* importante */
    quicksort(e,i-1);
    quicksort(i+1,d);
  }
}
```



Quicksort: algoritmo básico

- A primeira chamada: `quicksort(1,N)`
- A função `partition` deve:
 - rearrumar a lista de acordo com as três condições:
 - o elemento `a[i]` está em seu lugar final na lista
 - todos os elementos em `a[e]` a `a[i-1]` são menores que `a[i]`
 - todos os elementos em `a[i+1]` a `a[d]` são maiores ou iguais a `a[i]`



Quicksort: algoritmo básico

- Partition:
 - escolha arbitrariamente um elemento **pivot** $a[r]$ – elemento que estará na sua posição ao final
 - percorra a lista da **esquerda** até que um elemento maior que $a[r]$ seja encontrado
 - percorra a lista da **direita** até um elemento menor que $a[r]$
 - esses dois elementos estão fora de posição → **troque-os**



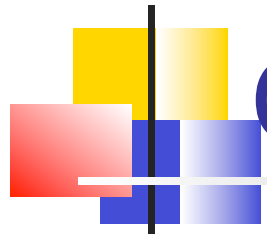
Quicksort: algoritmo básico

- parar a varredura toda vez que um elemento for igual ao pivot $a[r]$
 - Continuando dessa forma garante-se que todos os elementos da lista esquerda são menores e os a direita são maiores

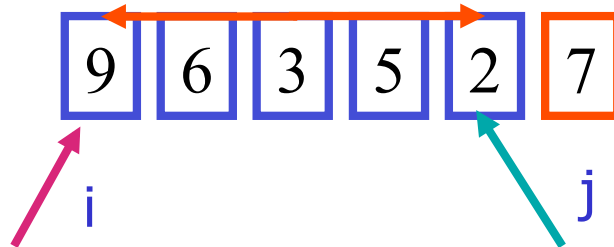


Quicksort: algoritmo básico

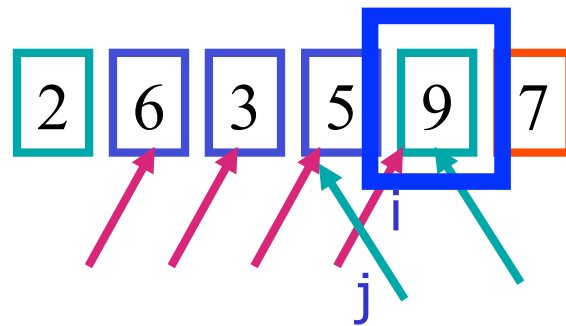
- Quando os ponteiros de varredura se cruzam, o processo está quase completo
 - falta trocar $a[r]$ com o elemento mais a esquerda da sub-lista da direita – o elemento $a[i]$
- a posição i foi definida
 - aplicar **quicksort** nas sublistas de e a $i-1$ e $i+1$ a d
 - i é o elemento que já está na sua posição



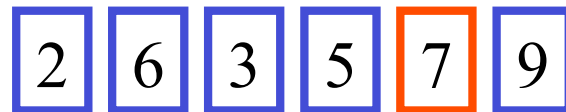
Quicksort: partition



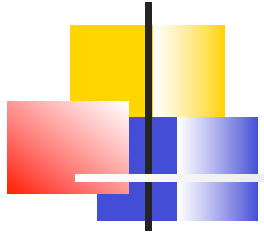
1ª iteração (**do-while** externo)



2ª iteração (**do-while** externo)



- fora **do-while** externo - trocas
- 7 - em sua posição definitiva
- **partition** retorna $i = 4$

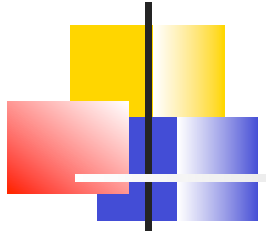


Quicksort: partition

```
partition(int e, int d)
{int v, i, j;
  v = a[d];  i = e - 1;  j = d;
  do {
    do{ i = i+1; /* esquerda*/
      } while (a[i] < v) && (i < d) ;
    do{ j = j-1; /* direita*/
      } while (a[j] > v) && (j > 0);
    t=a[i]; a[i]=a[j]; a[j]=t;
  } while (j > i)
```

```
/* para desfazer a troca extra
realizada quando j <= i e saiu
do while interno (t já tem o
valor de a[i]) */
```

```
  a[j] = a[i];
  a[i] = a[d];
  a[d] = t;
  return (i);
}
```



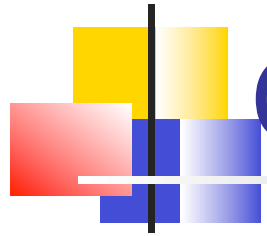
Quicksort

- É difícil imaginar um loop interno mais simples
 - simplesmente incrementam um ponteiro e fazem uma comparação
 - é o que faz o quicksort ser realmente rápido
- pode-se garantir que $a[d]$ nunca será o maior ou o menor elemento
 - o particionamento da lista seria desequilibrado em relação ao número de elementos



Quicksort: Partition

- Pegue arbitrariamente 3 elementos de a :
 - $a[d]$, $a[e]$ e $a[\lfloor (e+d)/2 \rfloor]$
- Compare os 3 elementos e defina o de valor médio
- Troque com $a[d]$
- Execute o **partition**
 - **Vantagem**: acrescentam-se um número fixo de instruções e não testes que variam com o tamanho da lista



Quicksort: características

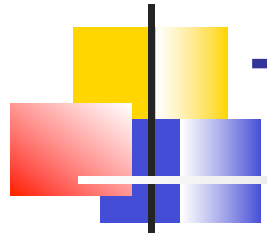
- o algoritmo não é estável
- tempo de processamento depende da seleção do pivot
- quanto ao tratamento do cruzamento dos ponteiros na presença de chaves iguais
 - tanto faz se a varredura de ambos os ponteiros parar, um continuar e outro parar, nenhum parar
 - mudanças devem ser feitas no algoritmo apresentado para listas com grande número de chaves repetidas



Exemplo

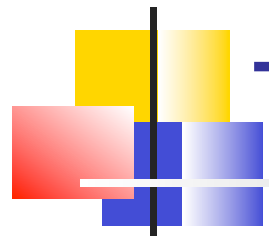
3	1	4	1	5	9	2	6	5	4
---	---	---	---	---	---	---	---	---	---

- Complexidade?



Trabalho Prático

- Analisar a complexidade do Quicksort, (pior caso e caso médio – pesquisar)
- Analisar a escolha do pivot
- Comparar em relação a complexidade e com exemplo os algoritmos de ordenação aqui estudados
 - por seleção
 - bubblesort
 - mergesort
 - quicksort
- Preparar as seguintes entradas a serem ordenadas: $n = 100$, 1000 e 10.000. Contabilizar o tempo de execução para cada instância. Cada instância deve ser gerada aleatoriamente.



Trabalho Prático (continuação)

- Pensar
 - Achar os K maiores números de uma sequência de N números não ordenados, onde $N \gg K$
 - derivar a complexidade