QuickSort

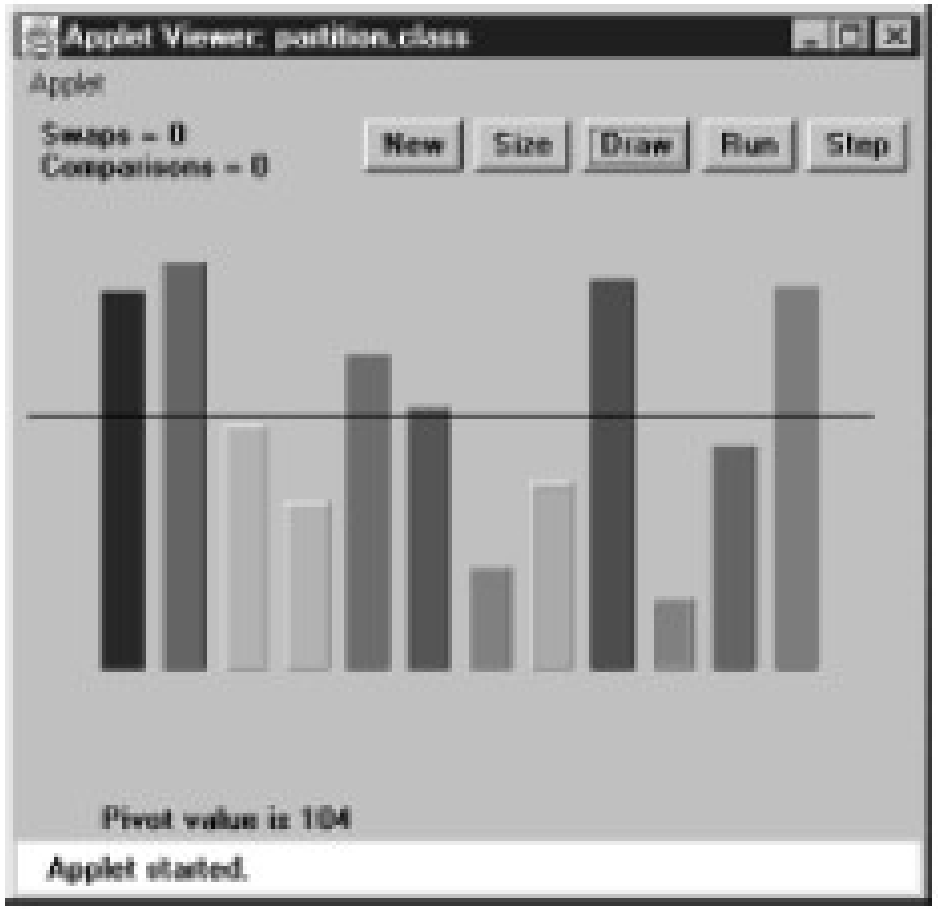
Estrutura de Dados II

Jairo Francisco de Souza

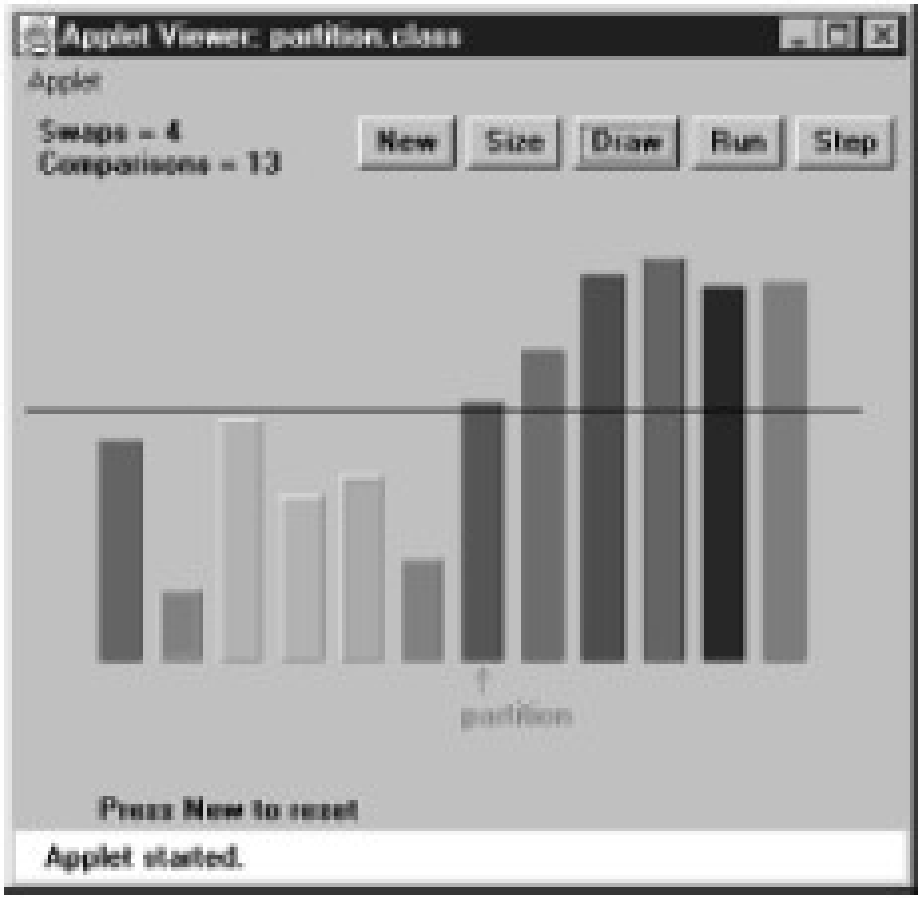
Particionamento

- Mecanismo principal dentro do algoritmo do QuickSort
- Para particionar um determinado conjunto de dados, separamos de um lado todos os itens cuja as chaves sejam maiores que um determinado valor, e do outro lado, colocamos todos os itens cuja as chaves sejam menores que um determinado valor
- Ex: Dividir as fichas de empregados entre quem mora a menos de 15km de distância da empresa e quem mora a uma distância acima de 15km

Particionamento



Particionamento



Particionamento

- A linha horizontal no exemplo anterior representa seu valor pivô, que é exatamente o valor utilizado para que possam ser separados os dois grupos de valores (um grupo de valores menores e um grupo de valores maiores que o pivô)
- No QuickSort este pivô é uma chave do vetor de ordenação escolhida respeitando alguns critérios
- Pode-se escolher qualquer valor para o pivô, dependendo do modo que serão construídas as partições

Particionamento

Apesar de termos dois grupos de valores, não quer dizer que os valores estejam ordenados nestes grupos.

Porém, só o fato de estarem separados pelo pivô numa classificação de maior/menor que o pivô, já facilita o trabalho de ordenação.

A cada passo que um novo pivô é escolhido os grupos ficam mais ordenados que antes

Algoritmo de Particionamento

O algoritmo trabalha começando com 2 “ponteiros”, um em cada ponta do array

O “ponteiro” da esquerda **leftPtr** move-se para a direita e o “ponteiro” da direita **rightPtr** move-se para a esquerda

leftPtr é inicializado com o índice zero e será incrementado e **rightPtr** é inicializado com índice do último elemento do vetor e será decrementado

Parada e troca de valores

Quando o **leftPtr** encontra um item de menor valor que o pivô, ele é incrementado já que o item está na posição correta. Entretanto, se encontrar um item de maior valor que o pivô, ele pára.

Igualmente, quando o **rightPtr** encontra um item de maior valor que o pivô, ele é decrementado já que o item está na posição correta. Entretanto, se encontra um item de menor valor que o pivô, ele pára.

Quando ambos os “ponteiros” param, é necessário fazer a troca dos dois elementos encontrados nas posições erradas.

Parada e troca de valores

```
while( theArray[++leftPtr] < pivot )    // find bigger item
    ;  // (nop)
while( theArray[--rightPtr] > pivot )   // find smaller item
    ;  // (nop)
swap(leftPtr, rightPtr);                // swap elements
```

Tratando exceções

E se todos os elementos são menores que o pivô?

O “ponteiro” **leftPtr** irá percorrer todo o array até ultrapassar seu limite e disparar uma exceção do tipo ***Array Index Out of Bounds***!

Para resolver basta adicionarmos mais uma condição nos nossos ***while***!

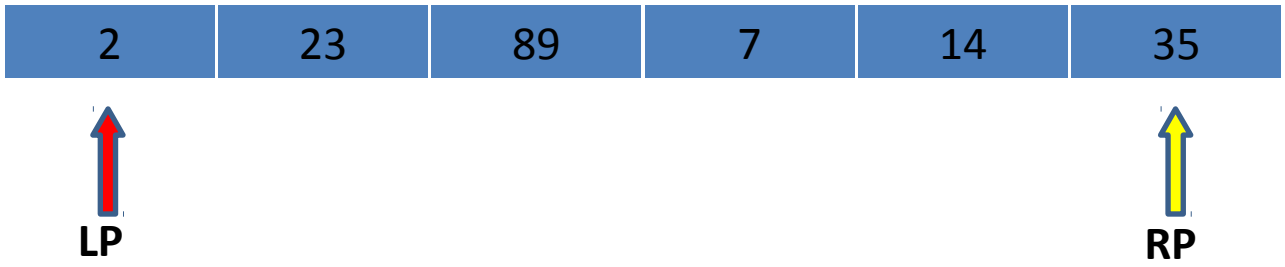
```
while(leftPtr < right &&      // find bigger item
      theArray[++leftPtr] < pivot)
   ;  // (nop)

while(rightPtr > left &&      // find smaller item
      theArray[--rightPtr] > pivot)
   ;  // (nop)
```

Troca dos elementos

```
public void swap(int dex1, int dex2)   // swap two elements
    {
    double temp;
    temp = theArray[dex1];              // A into temp
    theArray[dex1] = theArray[dex2];    // B into A
    theArray[dex2] = temp;              // temp into B
    }  // end swap(
```

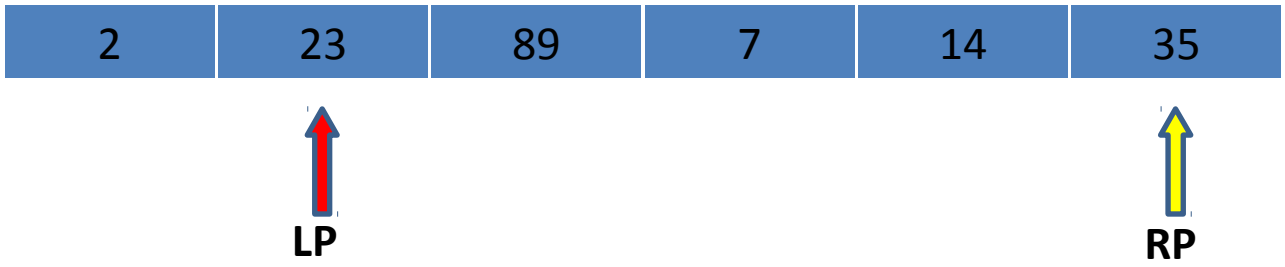
Exemplo

Particionar a seguinte lista com pivô = 15:



Exemplo

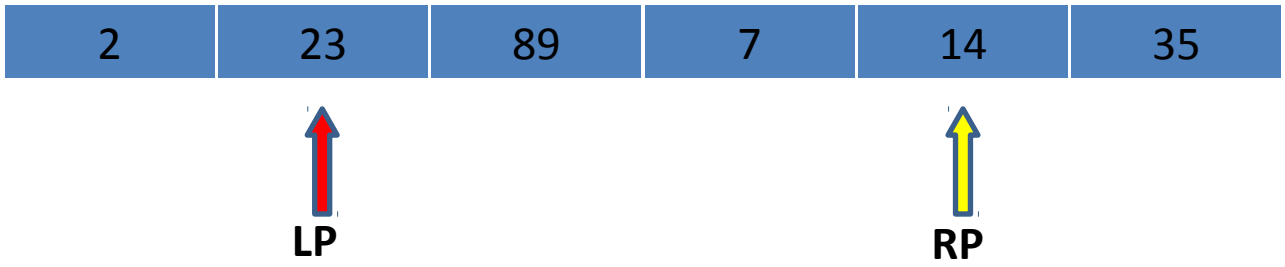
pivô = 15:



23 > 15 logo LP pára e RP começa a se mover!

Exemplo

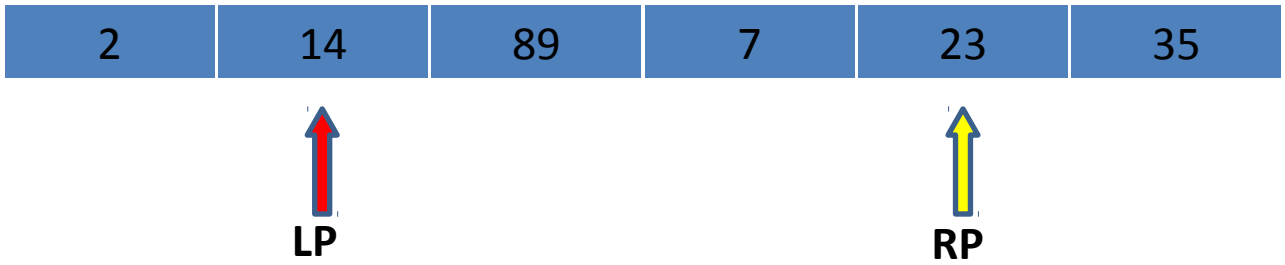
pivô = 15:



14 < 15 logo RP pára! Logo é necessário fazer a troca dos elementos: ***swap*(1,4)**

Exemplo

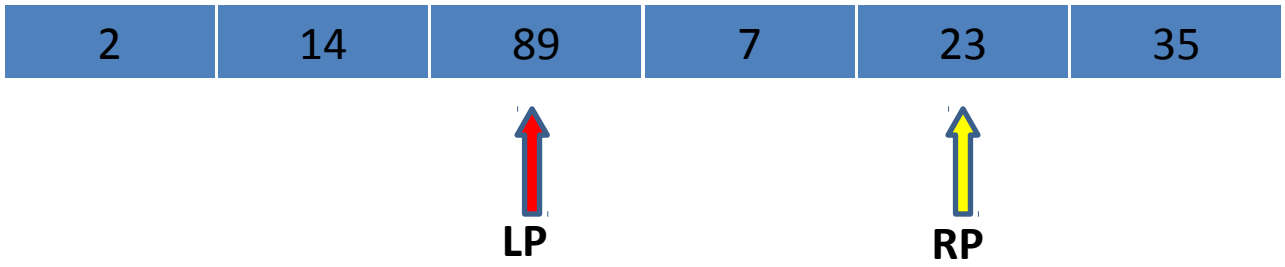
pivô = 15:



O **LP** volta a caminhar no vetor!

Exemplo

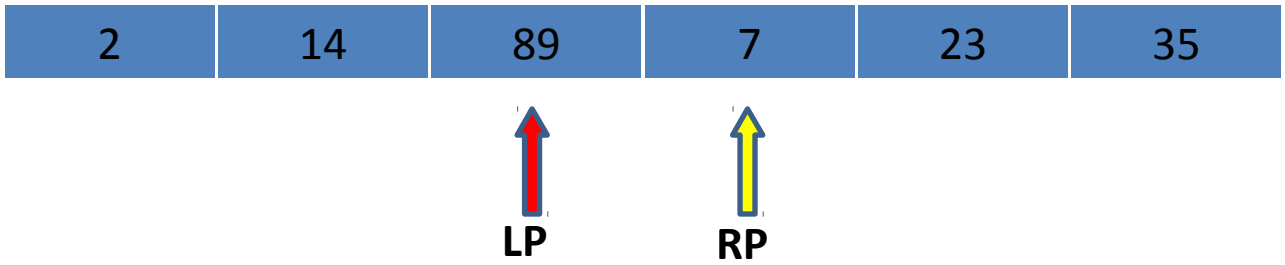
pivô = 15:



89 > 15 logo LP pára e RP volta a se mover!

Exemplo

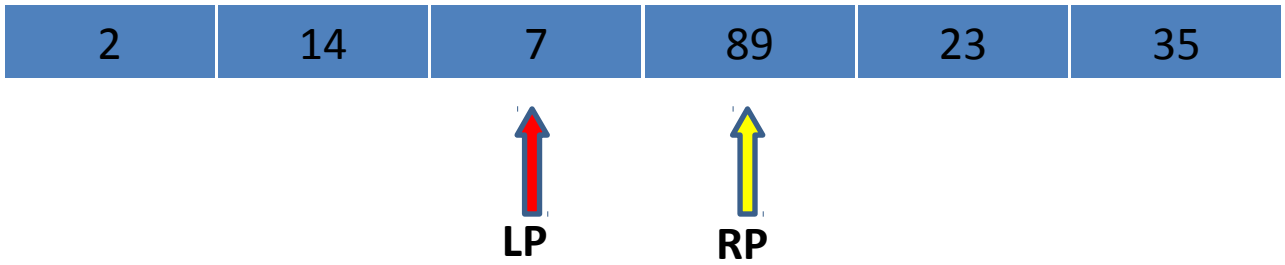
pivô = 15:



7 < 15 logo RP pára! Logo é necessário fazer a troca dos elementos: ***swap*(2,3)**

Exemplo

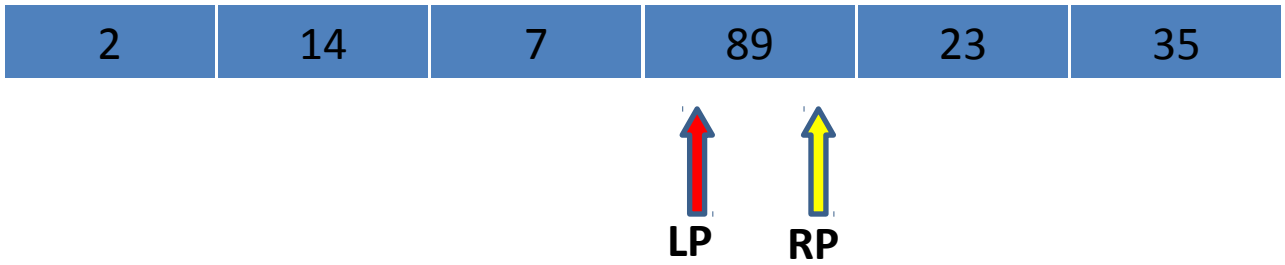
pivô = 15:



O **LP** volta a caminhar no vetor!

Exemplo

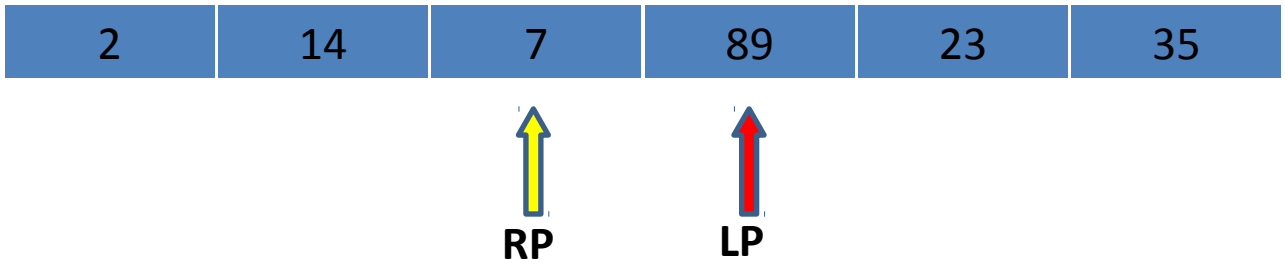
pivô = 15:



89 > 15 logo LP pára e RP volta a se mover!

Exemplo

pivô = 15:



7 < 15 logo RP pára! A condição **LP** >= **RP** é satisfeita e o particionamento termina!

Algoritmo

```
public int partitionIt(int left, int right, double pivot)
   {
   int leftPtr = left - 1;           // right of first elem
   int rightPtr = right + 1;         // left of pivot
   while(true)
      {
      while(leftPtr < right &&       // find bigger item
            theArray[++leftPtr] < pivot)
         ;  // (nop)

      while(rightPtr > left &&       // find smaller item
            theArray[--rightPtr] > pivot)
         ;  // (nop)
      if(leftPtr >= rightPtr)        // if pointers cross,
         break;                      //    partition done
      else                           // not crossed, so
         swap(leftPtr, rightPtr);    //    swap elements
      }  // end while(true)
   return leftPtr;                   // return partition
   }  // end partitionIt()
```

Eficiência do Particionamento

Roda em O(n)

Número de comparações é independente de como os dados estão arranjados

Número de trocas, por outro lado, é dependente do arranjo das informações:

Se a ordem está invertida e o pivô divide exatamente em dois conjuntos de mesmo tamanho temos N/2 trocas

No caso de dados randômico há um pouco menos de (N/2 trocas)

Exercício

Particione o seguinte conjunto de valores mostrando passo a passo o processo.

PIVÔ=99:

149	192	47	152	159	195	61	66	17	167	118	64	27	80	30	105

QuickSort

Algoritmo mais popular de ordenação

Na maioria dos casos ele roda em O(NlogN) para ordenações internas ou em memória. Para ordenar informações em arquivos em disco existem métodos melhores.

Algoritmo criado em 1962 por C.A.R. Hoare

Particionamento é a base do algoritmo e é chamado de maneira recursiva

É importante ainda a escolha do pivô e como é feito o processo de ordenação

QuickSort

```
public void recQuickSort(int left, int right)
    {
    if(right-left <= 0)          // if size is 1,
        return;                  //    it's already sorted
    else                         // size is 2 or larger
        {
                                                 // partition range
        int partition = partitionIt(left, right);
        recQuickSort(left, partition-1);   // sort left side
        recQuickSort(partition+1, right);  // sort right side
        }

    }
```

QuickSort

3 passos básicos:

Particionamento do array ou subarray em um grupo de chaves menores (lado esquerdo) e um grupo de chaves maiores (lado direito)

Chamada recursiva para ordenar/particionar o lado esquerdo

Chamada recursiva para ordenar/particionar o lado direito

QuickSort

Após o particionamento temos um grupo dentro do array do lado esquerdo com valores menores e um grupo dentro do array do lado direito com valores maiores. Se ordenarmos cada um desses grupos, temos no final um array totalmente ordenado

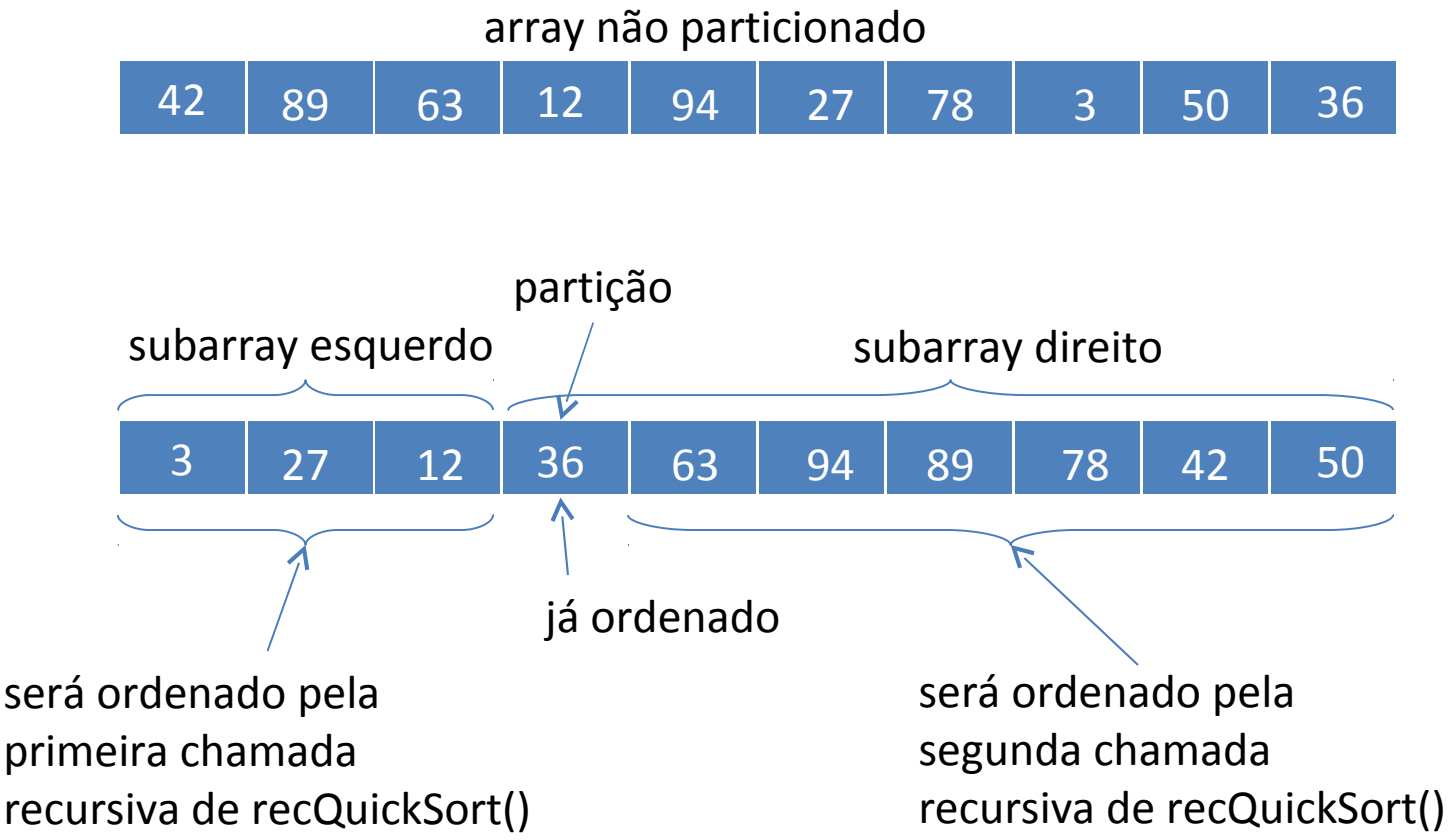
Como ordená-los? Utilizando as chamadas recursivas para particionar os subarrays. O método recebe como parâmetros onde começa e onde termina cada subarray que deve ser ordenado. O método checa antes se o subarray contém somente um elemento, este é o critério de parada para as chamadas recursivas

QuickSort

Se o subarray contém dois ou mais elementos, o algoritmo chama o método ***partitionInt()*** para particioná-lo. Este número retorna o índice da partição

O índice da partição marca exatamente o limite entre os dois lados do subarray (um lado com valores menores e outro lado com valores maiores)

QuickSort


array não particionado
42
89
63
12
94
27
78
3
50
36
partição
subarray esquerdo
subarray direito
3
27
12
36
63
94
89
78
42
50
já ordenado
será ordenado pela primeira chamada recursiva de recQuickSort()
será ordenado pela segunda chamada recursiva de recQuickSort()

QuickSort

Uma vez o array particionado, e a chamada recursiva ao método ***recQuickSort()*** para o lado esquerdo os parâmetros são ***left*** e ***partition-1*** já para o lado direito são ***partition+1***. E o índice ***partition***?

PIVÔ! Como escolher? E seu papel?

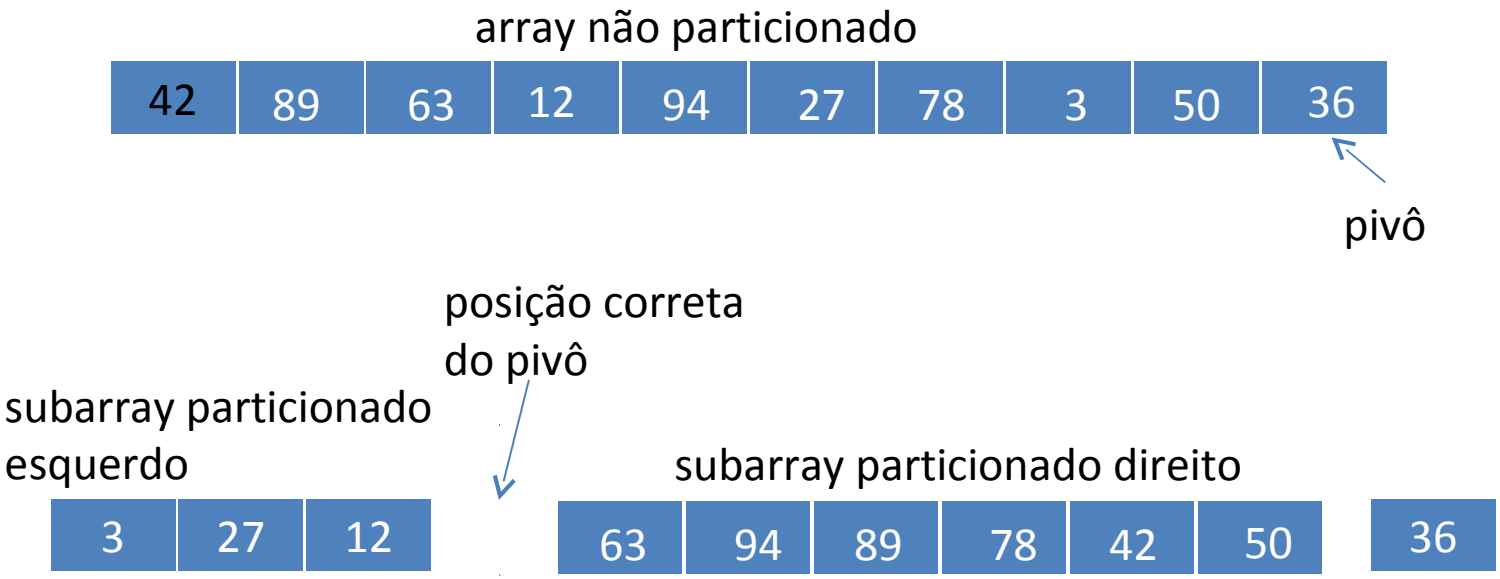
Escolha do Pivô

O pivô deve ser algum dos valores que compõem o array

O pivô pode ser escolhido aleatoriamente. Para simplificar, vamos escolher como pivô sempre o elemento que está na extrema direita de todo subarray que será particionado

Após o particionamento, se o pivô é inserido no limite entre os dois subarrays particionados, ele já estará automaticamente em sua posição correta na ordenação

Escolha do Pivô

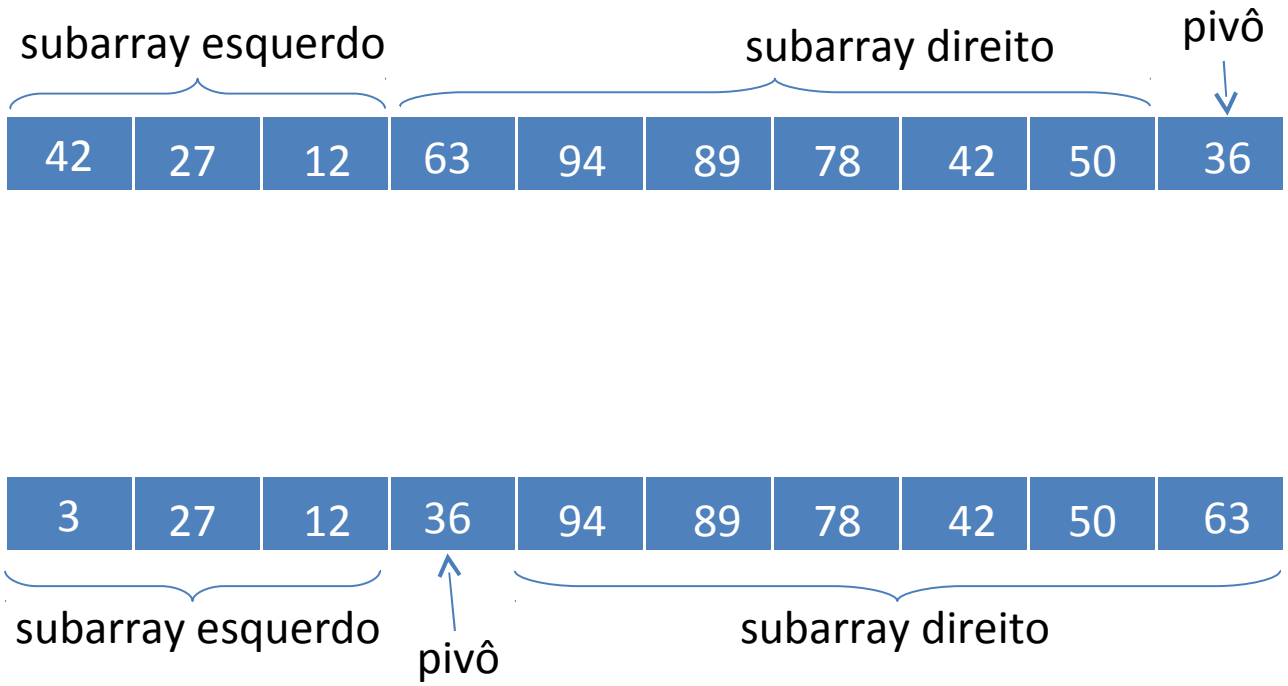


Escolha do Pivô

Como inserir então o pivô no seu lugar correto? Deslocando todos os outros elementos maiores que ele uma posição no array?

Uma possibilidade é simplesmente trocar o pivô pelo primeiro elemento do subarray que só possui valores maiores que o pivô

Escolha do Pivô



Pivô = 36
Após o particionamento troca-se o 36 pelo 63 que é o primeiro elemento do grupo dos valores maiores que o pivô

Novo Algoritmo com Pivô

```
public void recQuickSort(int left, int right)
    {
    if(right-left <= 0)          // if size <= 1,
        return;                  //    already sorted
    else                         // size is 2 or larger
       {
       double pivot = theArray[right];    // rightmost item
                                          // partition range
       int partition = partitionIt(left, right, pivot);
       recQuickSort(left, partition-1);   // sort left side
       recQuickSort(partition+1, right);  // sort right side
       }

    }  // end recQuickSort()
```

Novo Algoritmo Particionamento

```
public int partitionIt(int left, int right, double pivot)
   {
   int leftPtr = left-1;             // left    (after ++)
   int rightPtr = right;             // right-1 (after --)
   while(true)
      {                              // find bigger item
      while(theArray[++leftPtr] < pivot)
         ;  // (nop)
                                     // find smaller item
      while(rightPtr > 0 && theArray[--rightPtr] > pivot)
         ;  // (nop)

      if(leftPtr >= rightPtr)        // if pointers cross,
         break;                      //    partition done
      else                           // not crossed, so
         swap(leftPtr, rightPtr);    //    swap elements
      }  // end while(true)
   swap(leftPtr, right);             // restore pivot
   return leftPtr;                   // return pivot location
   }  // end partitionIt()
```

Degeneração para $O(n^2)$

O que acontece quando pegamos um array para ordenar e ele está em ordem decrescente?

Problema está na seleção do pivô. O ideal é que o pivô seja um valor mediano. Sendo assim, haveria duas partições com tamanhos bem próximos e o algoritmo funcionaria sempre muito rápido

Pivô Média

Muitos métodos são estudados para se alcançar o melhor pivô

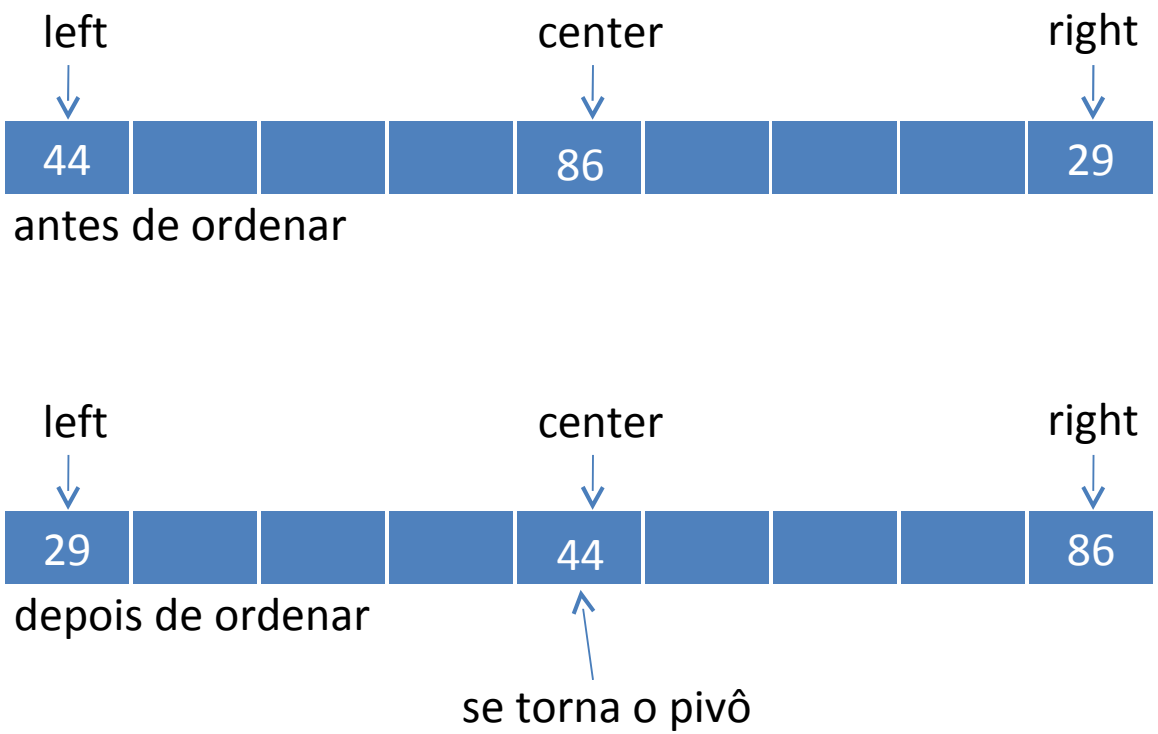
O método precisa ser simples para não degenerar o algoritmo e ao mesmo tempo ter uma boa chance de evitar os valores extremos

Poderíamos examinar todo o array e encontrar o valor mediano dentre todos?

Pivô Média

- Uma solução simples e atraente é obter o valor mediano entre três elementos do array:
  - 1º elemento
  - Elemento no meio do array
  - Último elemento
- Processo chamado **“média-dos-três”**
  - Agilidade no processo e possui altas taxas de sucesso
  - Ganho de desempenho no algoritmo

Pivô média


left
center
right
44
86
29
antes de ordenar
left
center
right
29
44
86
depois de ordenar
se torna o pivô

Pivô Média

O teste **rightPtr > left** do segundo loop dentro do método **partitionIt** pode ser retirado

O método média-de-três, além de escolher o pivô, ainda ordena estes 3 elementos

Ganho nas partições?

Assim, além de evitar a degradação para O(n2), ainda ganha-se tempo nos loops internos (menos uma comparação em cada iteração) e reduz a quantidade de elementos para serem particionados

Algoritmo

```
public void recQuickSort(int left, int right)
{
   int size = right-left+1;
   if(size <= 3) // manual sort if small
     manualSort(left, right);
   else // quicksort if large
   {
     double median = medianOf3(left, right);
     int partition = partitionIt(left, right, median);
     recQuickSort(left, partition-1);
     recQuickSort(partition+1, right);
   }
}
```

Algoritmo

```
public double medianOf3(int left, int right)
{
  int center = (left+right)/2;

  // order left & center
  if( theArray[left] > theArray[center] )
    swap(left, center);
  // order left & right
  if( theArray[left] > theArray[right] )
    swap(left, right);
  // order center & right
  if( theArray[center] > theArray[right] )
    swap(center, right);

  swap(center, right-1); // put pivot on right

  return theArray[right-1]; // return median value
}
```

Algoritmo

```
public int partitionIt(int left, int right, double pivot)
{
  int leftPtr = left; // right of first elem
  int rightPtr = right - 1; // left of pivot

  while(true) {

    while(theArray[++leftPtr] < pivot) // find bigger
      ; // (nop)
    while(theArray[--rightPtr] > pivot) // find smaller
      ; // (nop)

    if(leftPtr >= rightPtr) // if pointers cross,
      break; // partition done
    else // not crossed, so
      swap(leftPtr, rightPtr); // swap elements
  } // end while(true)

  swap(leftPtr, right-1); // restore pivot

  return leftPtr; // return pivot location
}
```

Algoritmo

```
public void manualSort(int left, int right)
{
  int size = right-left+1;
  if(size <= 1)
    return; // no sort necessary
  if(size == 2)
  { // 2-sort left and right
    if( theArray[left] > theArray[right] )
      swap(left, right);
    return;
  }
  else // size is 3
  { // 3-sort left, center (right-1) & right
    if( theArray[left] > theArray[right-1] )
      swap(left, right-1); // left, center
    if( theArray[left] > theArray[right] )
      swap(left, right); // left, right
    if( theArray[right-1] > theArray[right] )
      swap(right-1, right); // center,right
  }
}
```

Algoritmo

É necessário um algoritmo específico para ordenar um array de até 3 elementos pois utilizando a média-de-três só é possível utilizar a partir de 4 elementos

Segundo Knuth, o ideal é que arrays com tamanho inferior a 10 sejam ordenados utilizando o InsertionSort, caso contrário o QuickSort ordena normalmente

Algoritmo

```
public void recQuickSort(int left, int right)
{
  int size = right-left+1;
  if(size < 10) // insertion sort if small
    insertionSort(left, right);
  else // quicksort if large
  {
    double median = medianOf3(left, right);
    int partition = partitionIt(left, right, median);
    recQuickSort(left, partition-1);
    recQuickSort(partition+1, right);
  }
}
```

Recursão?

Alguns programadores/autores discutem a remoção da recursão do QuickSort e substituir por uma abordagem iterativa com loops

Apesar disto, esta idéia é pouco adotada atualmente pois as linguagens mais modernas não encontram gargalos nesse tipo de situação, muito menos o hardware

Eficiência do algoritmo

Complexidade: O(NlogN) para o melhor caso e para o caso médio

No pior caso, pode degenerar para $O(n^2)$

Correspondente aos algoritmos de divisão e conquista como o MergeSort também

Estudo da estabilidade

- O algoritmo é considerado instável, pois há a possibilidade de elementos com mesma chave mudar de posição no processo de ordenação
- O QuickSort é baseado no particionamento de vetores.
- Então, considerando o vetor [0 2^1 3 2^2 5 1] e o pivô como 2, teremos o seguinte particionamento:
  - Encontrando os ponteiros: [0 2^1 3 2^2 5 1]
  - Alterando as posições: [0 1 3 2^2 5 2^1]