Aula 17 – QuickSort

Norton T. Roman & Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

2023

Ordenação

- Vimos que algoritmos de ordenação, no modelo incremental, utilizam a indução fraca

Ordenação

- Vimos que algoritmos de ordenação, no modelo incremental, utilizam a indução fraca
  - Assume-se $T(n-1)$ para ordenar um arranjo de n elementos

Ordenação

- Vimos que algoritmos de ordenação, no modelo incremental, utilizam a indução fraca
  - Assume-se $T(n-1)$ para ordenar um arranjo de n elementos
  - Ex: Inserção e Seleção

Ordenação

- Vimos que algoritmos de ordenação, no modelo incremental, utilizam a indução fraca
  - Assume-se $T(n-1)$ para ordenar um arranjo de n elementos
  - Ex: Inserção e Seleção
- Também vimos que, na divisão e conquista, é necessário utilizar a indução forte

Ordenação

- Vimos que algoritmos de ordenação, no modelo incremental, utilizam a indução fraca
  - Assume-se $T(n-1)$ para ordenar um arranjo de n elementos
  - Ex: Inserção e Seleção
- Também vimos que, na divisão e conquista, é necessário utilizar a indução forte
  - Assumir $T(k)$, onde $c_{base} \leq k \leq (n-1)$, para ordenar um arranjo de n elementos

Projeto por Indução Forte

Padrão Divisão e Conquista:

```
OrdenaçãoD&C(A, ini, fim):
  Entrada: Arranjo A de n valores
  Saída: Arranjo A ordenado

  n = fim - ini + 1
  se n == 1 então retorne
  senão:
    <comandos iniciais: a divisão> (cálculo de q)
    OrdenaçãoD&C(A, ini, q)
    OrdenaçãoD&C(A, q+1, fim)
    <comandos finais: a combinação da conquista>
  retorne
```

Projeto por Indução Forte

Primeira Alternativa

Projeto por Indução Forte

Primeira Alternativa

- **Base**: $n = 1$. Um conjunto de um único elemento está ordenado

Projeto por Indução Forte

Primeira Alternativa

- **Base**: $n = 1$. Um conjunto de um único elemento está ordenado
- **H.I.**: Sei ordenar um conjunto de $1 \leq k < n$ valores

Projeto por Indução Forte

Primeira Alternativa

- **Base**: $n = 1$. Um conjunto de um único elemento está ordenado
- **H.I.**: Sei ordenar um conjunto de $1 \leq k < n$ valores
- **Passo**:

Projeto por Indução Forte

Primeira Alternativa

- **Base**: $n = 1$. Um conjunto de um único elemento está ordenado
- **H.I.**: Sei ordenar um conjunto de $1 \leq k < n$ valores
- **Passo**:
  - Seja S um conjunto de $n \geq 2$ valores, e x um elemento qualquer de S, e sejam S_1 e S_2 os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente

Projeto por Indução Forte

Primeira Alternativa

- **Base**: $n = 1$. Um conjunto de um único elemento está ordenado
- **H.I.**: Sei ordenar um conjunto de $1 \leq k < n$ valores
- **Passo**:
  - Seja S um conjunto de $n \geq 2$ valores, e x um elemento qualquer de S, e sejam S_1 e S_2 os subconjuntos de $S - x$ dos elementos menores ou iguais a x e maiores que x, respectivamente
  - Pela H.I., sabemos ordenar os conjuntos S_1 e S_2

Projeto por Indução Forte

Primeira Alternativa

- **Passo** (cont.):
  - Podemos então ter S ordenado concatenando S_1 ordenado, x e S_2 ordenado ($S_1 + x + S_2$)

Projeto por Indução Forte

Primeira Alternativa

- **Passo** (cont.):
  - Podemos então ter S ordenado concatenando S_1 ordenado, x e S_2 ordenado $(S_1 + x + S_2)$
- Esta indução dá origem ao algoritmo de divisão e conquista **QuickSort**

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Dividir:

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Dividir:
  - Particione (reorganize) o arranjo $A[p..r]$ em dois sub-arranjos (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ de modo a que todo elemento de $A[p..q-1]$ seja menor ou igual a $A[q]$, e que todo elemento de $A[q+1..r]$ seja maior ou igual a $A[q]$

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Dividir:
  - Particione (reorganize) o arranjo $A[p..r]$ em dois sub-arranjos (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ de modo a que todo elemento de $A[p..q-1]$ seja menor ou igual a $A[q]$, e que todo elemento de $A[q+1..r]$ seja maior ou igual a $A[q]$
  - O índice q é calculado como parte desse procedimento de particionamento

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Conquistar:

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Conquistar:
  - Ordene os sub-arranjos $A[p..q-1]$ e $A[q+1..r]$ recursivamente

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Conquistar:
  - Ordene os sub-arranjos $A[p..q-1]$ e $A[q+1..r]$ recursivamente
- Combinar:

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Conquistar:
  - Ordene os sub-arranjos $A[p..q-1]$ e $A[q+1..r]$ recursivamente
- Combinar:
  - Uma vez que os sub-arranjos são ordenados no próprio arranjo A, não é preciso fazer nada para combiná-los

QuickSort

Passos para ordenar um subarranjo $A[p..r]$

- Conquistar:
  - Ordene os sub-arranjos $A[p..q-1]$ e $A[q+1..r]$ recursivamente
- Combinar:
  - Uma vez que os sub-arranjos são ordenados no próprio arranjo A, não é preciso fazer nada para combiná-los
  - O arranjo $A[p..r]$ todo está ordenado

QuickSort

Código

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

QuickSort

Código

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

- Embora combinar não custe nada, dividir pode ser caro...

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

Quicksort

Particionando o arranjo

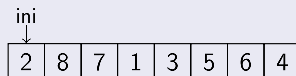
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

2	8	7	1	3	5	6	4

Quicksort

Particionando o arranjo

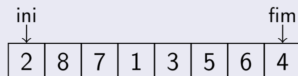
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



Quicksort

Particionando o arranjo

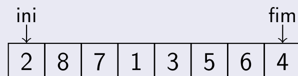
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

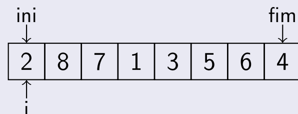


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

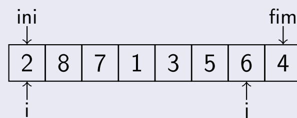


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

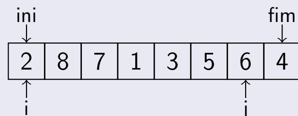
ini							fim
2	8	7	1	3	5	6	4
i						j	

x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

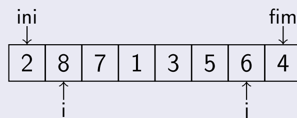


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

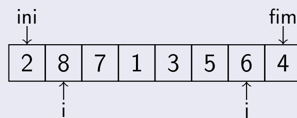


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

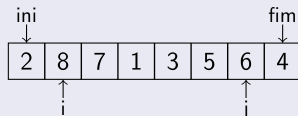


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

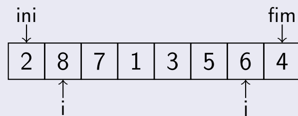


x = 4

Quicksort

Particionando o arranjo

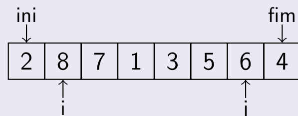
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

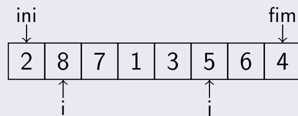


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

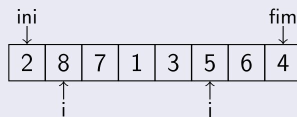


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

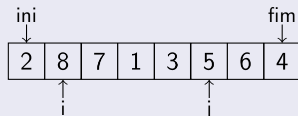


x = 4

Quicksort

Particionando o arranjo

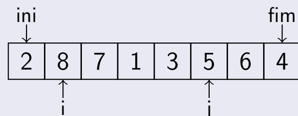
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

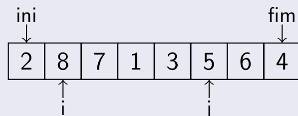


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

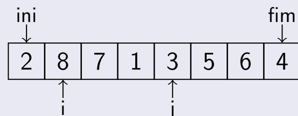


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

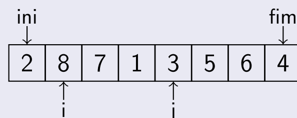


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

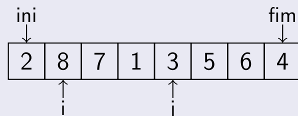


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

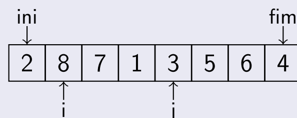


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

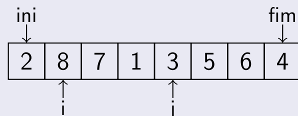


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

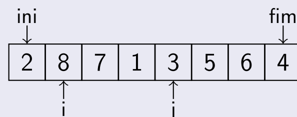


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

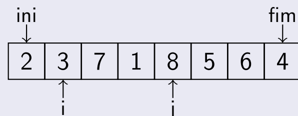


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

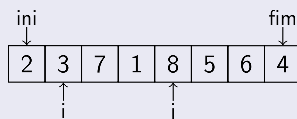


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Quicksort

Particionando o arranjo

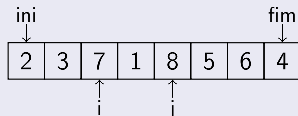
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

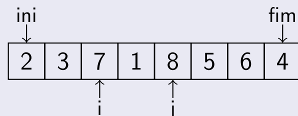


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

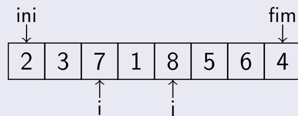


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

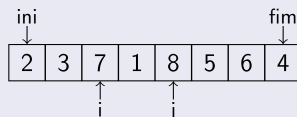


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

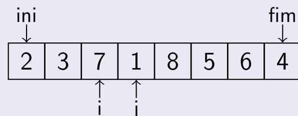
ini							fim
2	3	7	1	8	5	6	4
		i		j			

x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

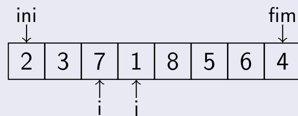


x = 4

Quicksort

Particionando o arranjo

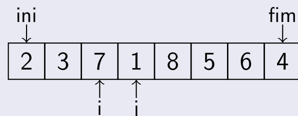
```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



x = 4

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

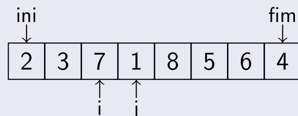


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

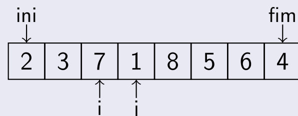


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

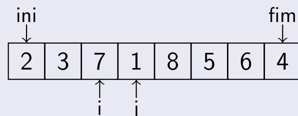


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

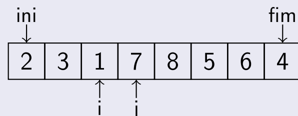


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

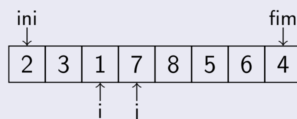


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

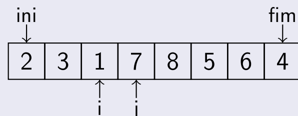


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

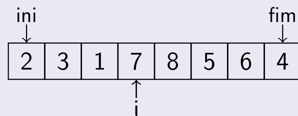


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

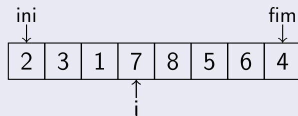


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

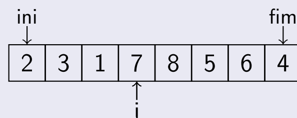


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

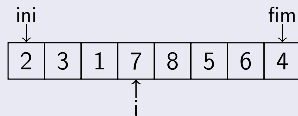


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

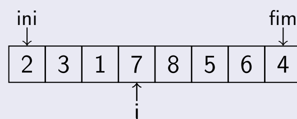


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

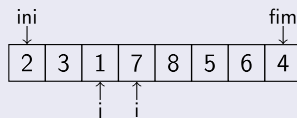


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

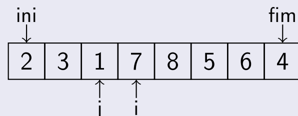


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

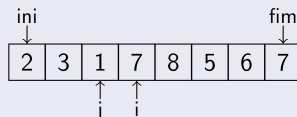


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

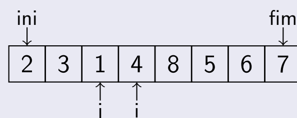


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

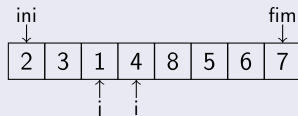


x = 4

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```

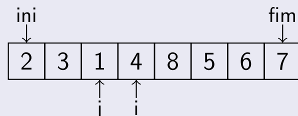


x = 4

i é o pivô a partir do qual a partição é feita

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivô
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
    else
      if (A[j] > x) j--;
        else { //troca A[i] e A[j]
          temp = A[i];
          A[i] = A[j];
          A[j] = temp;
        }
  }
  A[fim] = A[i]; //reposiciona o pivô
  A[i] = x;
  return(i);
}
```



`x = 4`

i é o pivô a partir do qual a partição é feita

À esquerda de i há os valores $\leq i$, enquanto que à direita há os $> i$

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

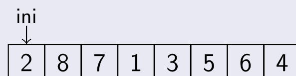
2	8	7	1	3	5	6	4

Chamadas:

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

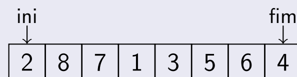


Chamadas:

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



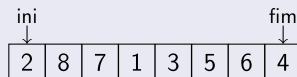
Chamadas:

-

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



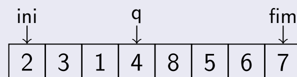
Chamadas:

-

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



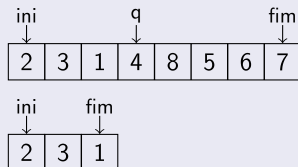
Chamadas:

-

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

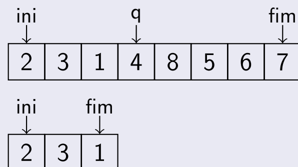


Chamadas:

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

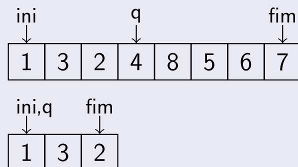


Chamadas:

Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

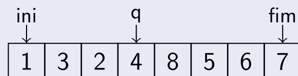


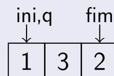
Chamadas:

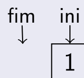
Quicksort

Ordenando

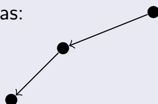
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







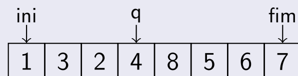
Chamadas:

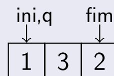


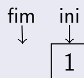
Quicksort

Ordenando

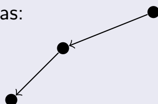
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







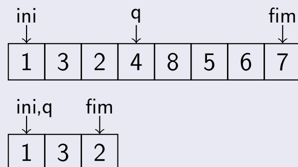
Chamadas:



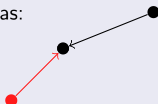
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



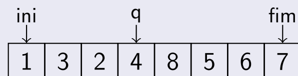
Chamadas:

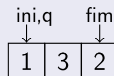


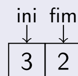
Quicksort

Ordenando

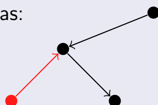
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







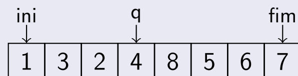
Chamadas:

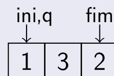


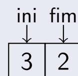
Quicksort

Ordenando

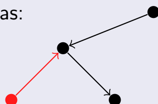
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







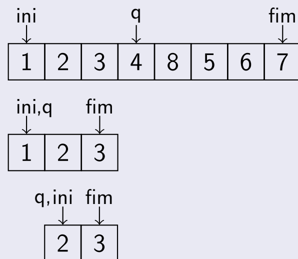
Chamadas:



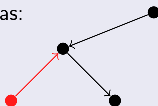
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



Chamadas:

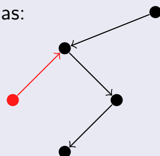


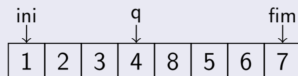
Quicksort

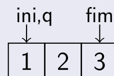
Ordenando

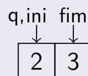
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

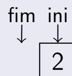
Chamadas:









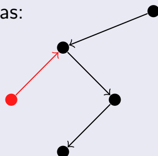


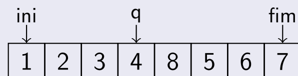
Quicksort

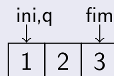
Ordenando

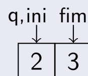
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

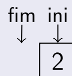
Chamadas:







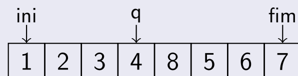


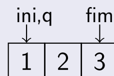


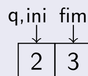
Quicksort

Ordenando

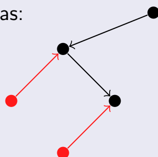
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







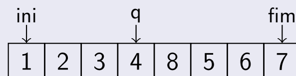
Chamadas:

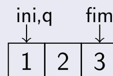


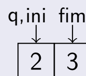
Quicksort

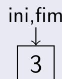
Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

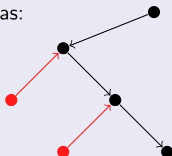








Chamadas:

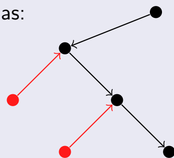


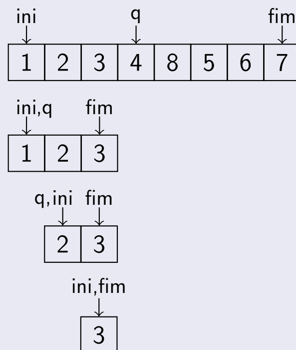
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

Chamadas:

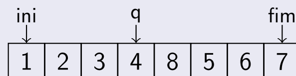


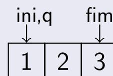


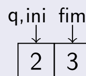
Quicksort

Ordenando

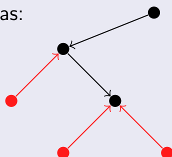
```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```







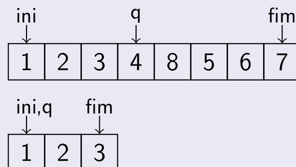
Chamadas:



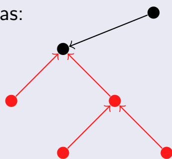
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



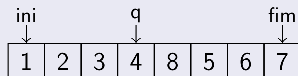
Chamadas:



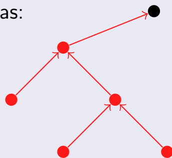
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



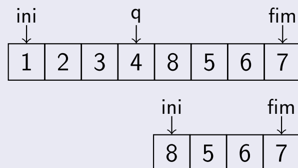
Chamadas:



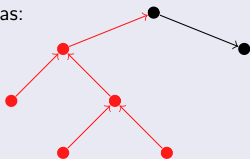
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



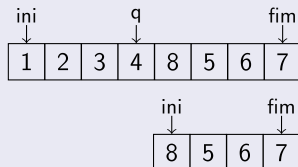
Chamadas:



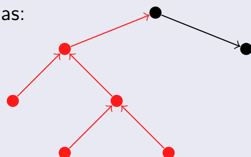
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



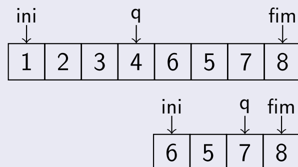
Chamadas:



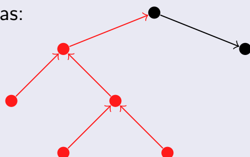
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



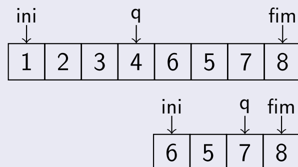
Chamadas:



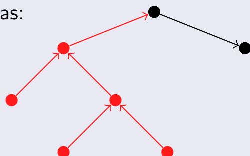
Quicksort

Ordenando

```
void quickSort(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```



Chamadas:



E assim por diante . . .

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivo
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
  else
    if (A[j] > x) j--;
    else { //troca A[i] e A[j]
      temp = A[i];
      A[i] = A[j];
      A[j] = temp;
      i++; j--;
    }
  }
  A[fim] = A[i]; //reposiciona o pivo
  A[i] = x;
  return i;
}
```

Quicksort

Particionando o arranjo

```
int particao (int A[], int ini, int fim) {
  int i, j, temp;
  int x = A[fim]; //pivo
  i = ini;
  j = fim - 1;
  while (i <= j) {
    if(A[i] <= x) i++;
  else
    if (A[j] > x) j--;
    else { //troca A[i] e A[j]
      temp = A[i];
      A[i] = A[j];
      A[j] = temp;
      i++; j--;
    }
  }
  A[fim] = A[i]; //reposiciona o pivo
  A[i] = x;
  return i;
}
```

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não
  - E isso, por sua vez, depende do pivô usado

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não
  - E isso, por sua vez, depende do pivô usado
- Pior caso: O particionamento resulta em $n - 1$ elementos de um lado e 0 do outro

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não
  - E isso, por sua vez, depende do pivô usado
- Pior caso: O particionamento resulta em $n - 1$ elementos de um lado e 0 do outro
  - Note que uma posição foi reservada para o pivô, por isso, a soma do tamanho dos dois sub-arranjos é $n - 1$

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não
  - E isso, por sua vez, depende do pivô usado
- Pior caso: O particionamento resulta em $n - 1$ elementos de um lado e 0 do outro
  - Note que uma posição foi reservada para o pivô, por isso, a soma do tamanho dos dois sub-arranjos é $n - 1$
- E quando isso ocorre?

Quicksort

Desempenho

- O desempenho do QuickSort depende do particionamento ser balanceado ou não
  - E isso, por sua vez, depende do pivô usado
- Pior caso: O particionamento resulta em $n - 1$ elementos de um lado e 0 do outro
  - Note que uma posição foi reservada para o pivô, por isso, a soma do tamanho dos dois sub-arranjos é $n - 1$
- E quando isso ocorre?
  - Quando o arranjo já está ordenado

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?

```
void quickSort(int A[], int ini,
                        int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?
- Supondo que esse particionamento se repita em cada chamada recursiva, temos

```
void quickSort(int A[], int ini,
                        int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

$$T(n) = \begin{cases} & \text{se } n \leq 1 \\ & \text{para } n \geq 2 \end{cases}$$

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?
- Supondo que esse particionamento se repita em cada chamada recursiva, temos

```
void quickSort(int A[], int ini,
                          int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ & \text{para } n \geq 2 \end{cases}$$

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?
- Supondo que esse particionamento se repita em cada chamada recursiva, temos

```
void quickSort(int A[], int ini,
                        int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ \Theta(n) & \text{para } n \geq 2 \end{cases}$$

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?
- Supondo que esse particionamento se repita em cada chamada recursiva, temos

```
void quickSort(int A[], int ini,
                        int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ \Theta(n) + T(n-1) & \text{para } n \geq 2 \end{cases}$$

Quicksort

Desempenho: pior caso

- E qual o custo do Quicksort?
- Supondo que esse particionamento se repita em cada chamada recursiva, temos

```
void quickSort(int A[], int ini,
                         int fim) {
  int q;
  if (ini < fim) {
    q = particao(A, ini, fim);
    quickSort(A, ini, q-1);
    quickSort(A, q+1, fim);
  }
}
```

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

Quicksort

Desempenho: pior caso

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

- Por que o particionamento é $\Theta(n)$?

Quicksort

Desempenho: pior caso

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

- Por que o particionamento é $\Theta(n)$?
  - Para fazer o particionamento, serão necessárias sempre n comparações

Quicksort

Desempenho: pior caso

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

- Por que o particionamento é $\Theta(n)$?
    - Para fazer o particionamento, serão necessárias sempre n comparações
- E quanto é $T(0)$?

Quicksort

Desempenho: pior caso

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

- Por que o particionamento é $\Theta(n)$?
  - Para fazer o particionamento, serão necessárias sempre n comparações
- E quanto é $T(0)$?
  - Cai no mesmo caso do $T(1)$ no código, ou seja, $O(1)$

Quicksort

Desempenho: pior caso

- Então a relação fica:

$$T(n) = \Theta(n) + T(n-1) + O(1)$$

Quicksort

Desempenho: pior caso

- Então a relação fica:

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) + O(1) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Desempenho: pior caso

- Então a relação fica:

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) + O(1) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Quicksort

Desempenho: pior caso

- Então a relação fica:

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) + O(1) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

- Ou seja, no pior caso, com o particionamento desbalanceado ao máximo, o tempo é o mesmo da Seleção, Inserção e Bolha

Quicksort

Desempenho: pior caso

- Então a relação fica:

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) + O(1) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

- Ou seja, no pior caso, com o particionamento desbalanceado ao máximo, o tempo é o mesmo da Seleção, Inserção e Bolha
  - Com o agravante de que, se a entrada estiver ordenada, a inserção roda em tempo $O(n^2)$

Quicksort

Desempenho: melhor caso

- No melhor particionamento possível, teremos 2 sub-problemas, de tamanhos $n/2$ e $(n/2 - 1)$

Quicksort

Desempenho: melhor caso

- No melhor particionamento possível, teremos 2 sub-problemas, de tamanhos $n/2$ e $(n/2 - 1)$
  - Mais uma vez, uma posição foi reservada para o pivô

Quicksort

Desempenho: melhor caso

- No melhor particionamento possível, teremos 2 sub-problemas, de tamanhos $n/2$ e $(n/2 - 1)$
  - Mais uma vez, uma posição foi reservada para o pivô
- Nesse caso, repetindo o procedimento anterior teremos

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

Quicksort

Desempenho: melhor caso

- No melhor particionamento possível, teremos 2 sub-problemas, de tamanhos $n/2$ e $(n/2 - 1)$
  - Mais uma vez, uma posição foi reservada para o pivô
- Nesse caso, repetindo o procedimento anterior teremos

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

  - E, pelo teorema mestre (caso 2), temos que $T(n) = \Theta(n \log n)$

Quicksort

Particionamento Balanceado

- O tempo de execução do caso médio do QuickSort é muito mais próximo do melhor caso do que do pior caso

Particionamento Balanceado

- O tempo de execução do caso médio do QuickSort é muito mais próximo do melhor caso do que do pior caso
- Para entender isto, considere um particionamento que sempre produza uma divisão proporcional de 9 para 1

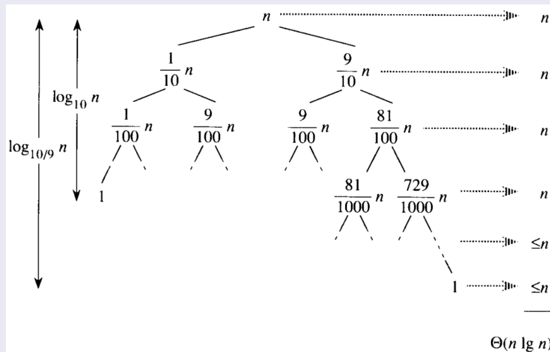
Quicksort

Particionamento Balanceado

- O tempo de execução do caso médio do QuickSort é muito mais próximo do melhor caso do que do pior caso
- Para entender isto, considere um particionamento que sempre produza uma divisão proporcional de 9 para 1
  - Isso parece bastante desbalanceado, possuindo a recorrência $T(n) = T(9n/10) + T(n/10) + cn$

Particionamento Balanceado

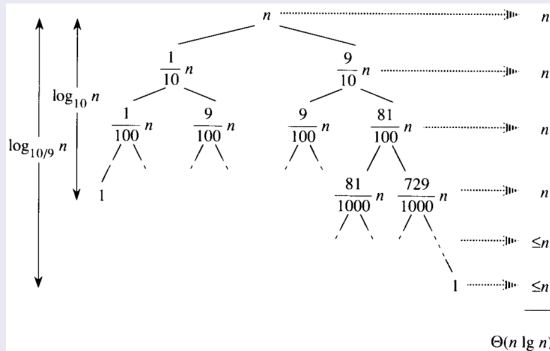
- Expandindo a árvore de recursão temos



Quicksort

Particionamento Balanceado

- Expandindo a árvore de recursão temos



Ou seja, o custo total do quicksort, mesmo com esse particionamento, é $O(n \log n)$

Quicksort

Intuição para o caso médio

Quicksort

Intuição para o caso médio

- No caso médio, todas as permutações dos valores de entrada são igualmente prováveis

Quicksort

Intuição para o caso médio

- No caso médio, todas as permutações dos valores de entrada são igualmente prováveis
- Para um arranjo de entrada aleatório é improvável que o particionamento ocorra sempre do mesmo modo em todo nível (como suposto na análise informal anterior)

Quicksort

Intuição para o caso médio

- No caso médio, todas as permutações dos valores de entrada são igualmente prováveis
- Para um arranjo de entrada aleatório é improvável que o particionamento ocorra sempre do mesmo modo em todo nível (como suposto na análise informal anterior)
  - O esperado é que haja algumas divisões boas e outras ruins, distribuídas aleatoriamente ao longo da árvore

Quicksort

Intuição para o caso médio

- Suponha, no entanto, que partições boas e ruins se alternem nos níveis da árvore

Quicksort

Intuição para o caso médio

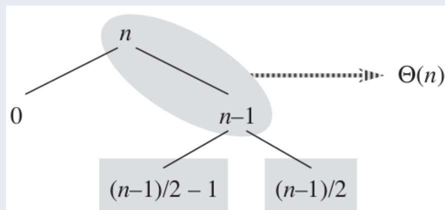
- Suponha, no entanto, que partições boas e ruins se alternem nos níveis da árvore
- Suponha também que as boas partições são do melhor caso, enquanto que as ruins são do pior caso

Quicksort

Intuição para o caso médio

- Suponha, no entanto, que partições boas e ruins se alternem nos níveis da árvore
- Suponha também que as boas partições são do melhor caso, enquanto que as ruins são do pior caso

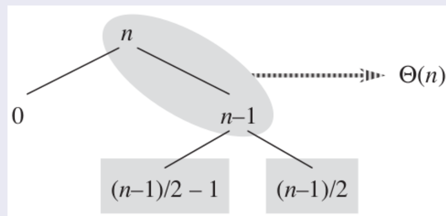
2 níveis consecutivos da árvore de recursão (pior e melhor caso):



Quicksort

Intuição para o caso médio

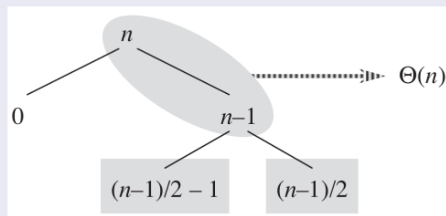
- Então temos:



Quicksort

Intuição para o caso médio

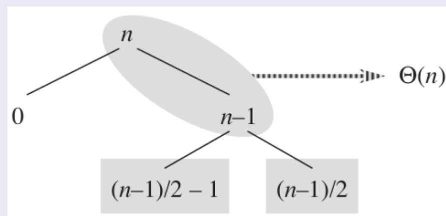
- Então temos:
  - Primeira divisão (ruim): $T(0) + T(n-1)$



Quicksort

Intuição para o caso médio

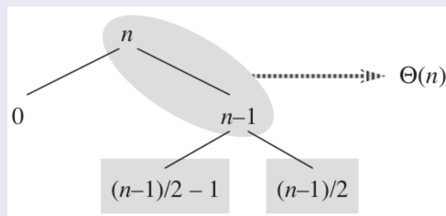
- Então temos:
  - Primeira divisão (ruim): $T(0) + T(n-1)$
  - Segunda divisão (boa): $T(\frac{n1}{2}) + T(\frac{n1}{2})$



Quicksort

Intuição para o caso médio

- Então temos:
  - Primeira divisão (ruim): $T(0) + T(n-1)$
  - Segunda divisão (boa): $T(\frac{n1}{2}) + T(\frac{n1}{2})$

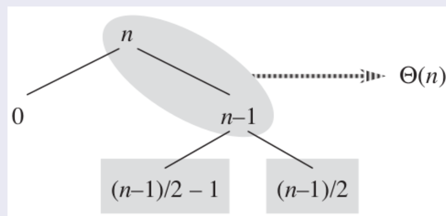


- E, combinando, temos $T(0) + T(\frac{n1}{2}) + T(\frac{n1}{2})$

Quicksort

Intuição para o caso médio

- Então temos:
  - Primeira divisão (ruim): $T(0) + T(n-1)$
  - Segunda divisão (boa): $T(\frac{n1}{2}) + T(\frac{n1}{2})$

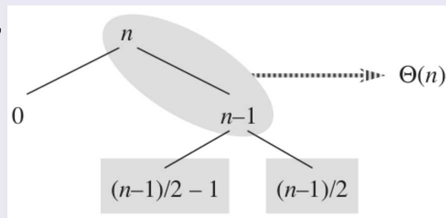


- E, combinando, temos $T(0) + T(\frac{n1}{2}) + T(\frac{n1}{2})$
- Ou seja, o custo da divisão ruim ($T(n-1)$) é absorvido pela divisão boa ($T(\frac{n1}{2})$)

Quicksort

Intuição para o caso médio

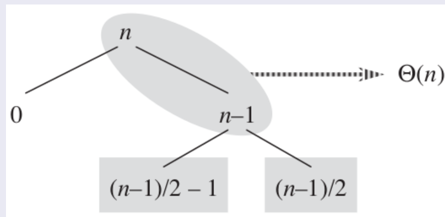
- Portanto, o tempo de execução do QuickSort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao custo para as divisões boas sozinhas



Quicksort

Intuição para o caso médio

- Portanto, o tempo de execução do QuickSort, quando os níveis se alternam entre divisões boas e ruins, é semelhante ao custo para as divisões boas sozinhas



- Ainda é $T(n) = O(n\,log\,n)$, mas com uma constante maior

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis

Quicksort

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis
- Ex: Cheques são registrados na ordem temporal (data) em que são descontados

Quicksort

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis
- Ex: Cheques são registrados na ordem temporal (data) em que são descontados
  - O banco pode querer uma listagem ordenada pelo número do cheque (e.g., 86781, 86782,...)

Quicksort

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis
- Ex: Cheques são registrados na ordem temporal (data) em que são descontados
  - O banco pode querer uma listagem ordenada pelo número do cheque (e.g., 86781, 86782,...)
  - Normalmente, os cheques são emitidos e descontados na ordem do talão, mas nem sempre. Às vezes, o comerciante pode dar um prazo a mais (cheque pré-datado) e demorar para descontar o cheque

Quicksort

Aproximando-se do caso médio

- Nem sempre as possíveis entradas são equiprováveis
- Ex: Cheques são registrados na ordem temporal (data) em que são descontados
  - O banco pode querer uma listagem ordenada pelo número do cheque (e.g., 86781, 86782,...)
  - Normalmente, os cheques são emitidos e descontados na ordem do talão, mas nem sempre. Às vezes, o comerciante pode dar um prazo a mais (cheque pré-datado) e demorar para descontar o cheque
  - Ou seja: a sequência temporal dos cheques está quase ordenada por número de cheque

Quicksort

Aproximando-se do caso médio

- Nesta situação, o QuickSort vai ter um desempenho ruim

Quicksort

Aproximando-se do caso médio

- Nesta situação, o QuickSort vai ter um desempenho ruim
- É então necessário aproximar-se do caso médio

Quicksort

Aproximando-se do caso médio

- Nesta situação, o QuickSort vai ter um desempenho ruim
- É então necessário aproximar-se do caso médio
  - Como?

Quicksort

Aproximando-se do caso médio

- Nesta situação, o QuickSort vai ter um desempenho ruim
- É então necessário aproximar-se do caso médio
  - Como?
- Uma maneira de aproximar-se do caso médio é escolhendo aleatoriamente um pivô, ao invés de utilizar sempre o último elemento

Quicksort

Aproximando-se do caso médio

- Nesta situação, o QuickSort vai ter um desempenho ruim
- É então necessário aproximar-se do caso médio
  - Como?
- Uma maneira de aproximar-se do caso médio é escolhendo aleatoriamente um pivô, ao invés de utilizar sempre o último elemento
  - Técnica conhecida como **amostragem aleatória**

Aproximando-se do caso médio

- E como fazemos isso?

Quicksort

Aproximando-se do caso médio

- E como fazemos isso?
  - Trocando o elemento `A[fim]` por um elemento escolhido aleatoriamente do sub-arranjo `A[ini..fim]`

Quicksort

Aproximando-se do caso médio

- E como fazemos isso?
  - Trocando o elemento `A[fim]` por um elemento escolhido aleatoriamente do sub-arranjo `A[ini..fim]`
- Isso faz com que o pivô (`A[fim]`) tenha a mesma chance de ser qualquer elemento do sub-arranjo

Quicksort

Aproximando-se do caso médio

- E como fazemos isso?
  - Trocando o elemento `A[fim]` por um elemento escolhido aleatoriamente do sub-arranjo `A[ini..fim]`
- Isso faz com que o pivô (`A[fim]`) tenha a mesma chance de ser qualquer elemento do sub-arranjo
  - Uma vez que o pivô é escolhido aleatoriamente, esperamos que a partição do arranjo fique na média razoavelmente bem balanceada

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Retorna um número inteiro, pseudoaleatório, maior ou igual a zero

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

i é tal que
$ini \leq i \leq fim$

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Escolhe um número aleatório entre `ini` e `fim`)

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Troca de posição A[i] e A[fim]

Quicksort

Partição aleatória

```
int particaoAleatoria (int A[], int ini, int fim) {
  int i, temp;

  i = rand();
  i = i % (fim-ini+1) + ini ;

  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Note que simplesmente implementamos a troca antes de efetivamente particionar

Quicksort

Quicksort aleatório

```
void quickSortAleatorio(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particaoAleatoria(A, ini, fim);
    quickSortAleatorio(A, ini, q-1);
    quickSortAleatorio(A, q+1, fim);
  }
}
```

Quicksort

Partição usando pivô que estava no meio

```
int particaoMeio (int A[], int ini, int fim) {
  int temp, i;

  i = (fim+ini)/2;


  temp = A[fim];
  A[fim] = A[i];
  A[i] = temp;
  return particao(A, ini, fim);
}
```

Quicksort

Quicksort - pivô que estava no meio

```
void quickSortMeio(int A[], int ini, int fim) {
  int q;
  if (ini < fim) {
    q = particaoMeio(A, ini, fim);
    quickSortMeio(A, ini, q-1);
    quickSortMeio(A, q+1, fim);
  }
}
```

Quicksort

Em suma

Quicksort

Em suma

- O Quicksort trabalha com ordenação local (*in loco* ou *in-place*)

Quicksort

Em suma

- O Quicksort trabalha com ordenação local (*in loco* ou *in-place*)
  - Utiliza quantidade de memória constante, além do próprio arranjo (não precisa criar um arranjo auxiliar para realizar a ordenação)

Quicksort

Em suma

- O Quicksort trabalha com ordenação local (*in loco* ou *in-place*)
  - Utiliza quantidade de memória constante, além do próprio arranjo (não precisa criar um arranjo auxiliar para realizar a ordenação)
- Pior caso: $O(n^2)$

Quicksort

Em suma

- O Quicksort trabalha com ordenação local (*in loco* ou *in-place*)
  - Utiliza quantidade de memória constante, além do próprio arranjo (não precisa criar um arranjo auxiliar para realizar a ordenação)
- Pior caso: $O(n^2)$
- Melhor caso: $O(n \log n)$

Quicksort

Em suma

- O Quicksort trabalha com ordenação local (*in loco* ou *in-place*)
  - Utiliza quantidade de memória constante, além do próprio arranjo (não precisa criar um arranjo auxiliar para realizar a ordenação)
- Pior caso: $O(n^2)$
- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$

Referências

- Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms. 2a ed. MIT Press, 2001.

- Material baseado em slides dos professores Delano Beder e Marcos Chain

Aula 17 – QuickSort

Norton T. Roman & Luciano A. Digiampietri
digiampietri@usp.br
@digiampietri

2023