# Algoritmos de Ordenação (Parte 4)

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

DAINF
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CÂMPUS PATO BRANCO

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

# Sumário

- Ordenação por Intercalação (*Mergesort*)
- Ordenação em Strings
- Ordenação em TAD

# Sumário

Ordenação por Intercalação (*Mergesort*)

# Ordenação por Intercalação (*Mergesort*)

- Divisão e conquista
- Se o vetor tiver o tamanho igual a 1, a função apenas retorna o elemento
- Caso o tamanho do vetor seja maior que 1
  1. **Divisão**: divida o vetor ao meio
  2. **Conquista 1**: ordene a primeira metade recursivamente
  3. **Conquista 2**: ordene a segunda metade recursivamente
  4. **Combinação**: intercale as duas metades

# Ordenação por Intercalação (*Mergesort*)

- Implementação

```
void mergesort(int v[], int esq, int dir){
  int meio;
  if (esq < dir){
    meio = (esq + dir) / 2;

    mergesort(v, esq, meio);
    mergesort(v, meio + 1, dir);
    merge(v, esq, meio, dir);
  }
}
```

# Ordenação por Intercalação (*Mergesort*)

- Implementação

```
void merge(int v[], int esq, int meio, int dir){
   int i, j, k;
   int n1 = meio - esq + 1;
   int n2 = dir - meio;
   int L[n1 + 1];
   int R[n2 + 1];

   for (i = 0; i < n1; i++)
     L[i] = v[esq + i];

   for (j = 0; j < n2; j++)
     R[j] = v[meio + j + 1];

   L[n1] = INT_MAX;
   R[n2] = INT_MAX;
   i = 0;
   j = 0;

   for (k = esq; k <= dir; k++)
     if (L[i] <= R[j]){
        v[k] = L[i];
        i++;
     }else{
        v[k] = R[j];
        j++;
     }
}
```

# Ordenação por Intercalação (*Mergesort*)

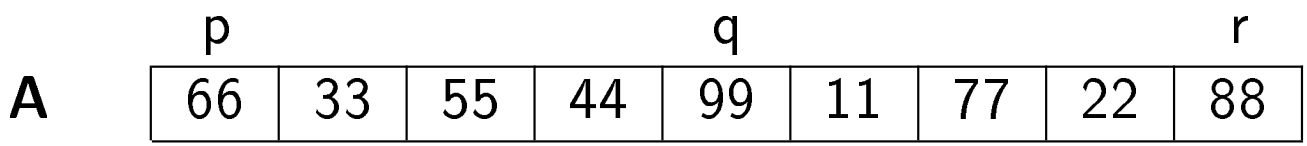
- Entrada

| | p | | | | q | | | | r |
|---|---|---|---|---|---|---|---|---|---|
| **A** | 22 | 33 | 55 | 77 | 99 | 11 | 44 | 66 | 88 |

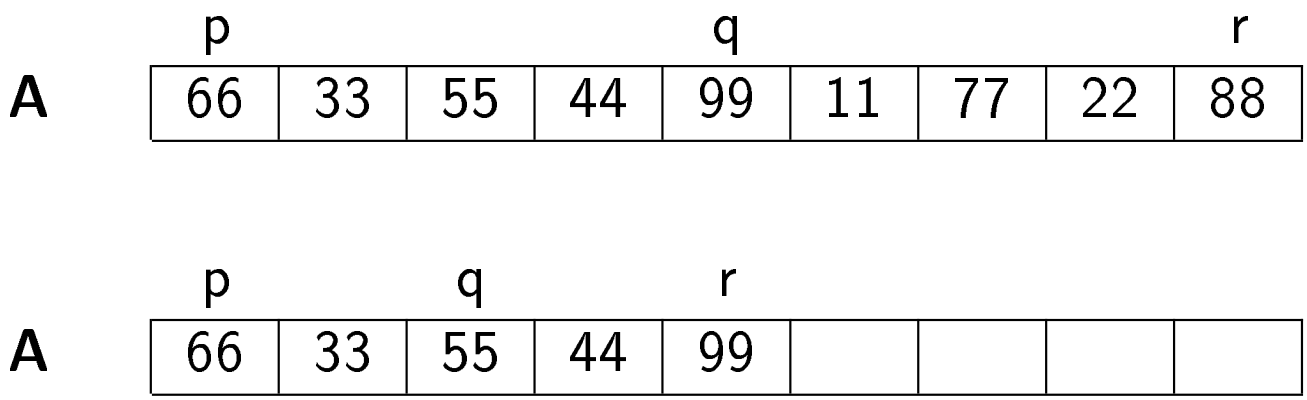
- Saída

| | p | | | | q | | | | r |
|---|---|---|---|---|---|---|---|---|---|
| **A** | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |

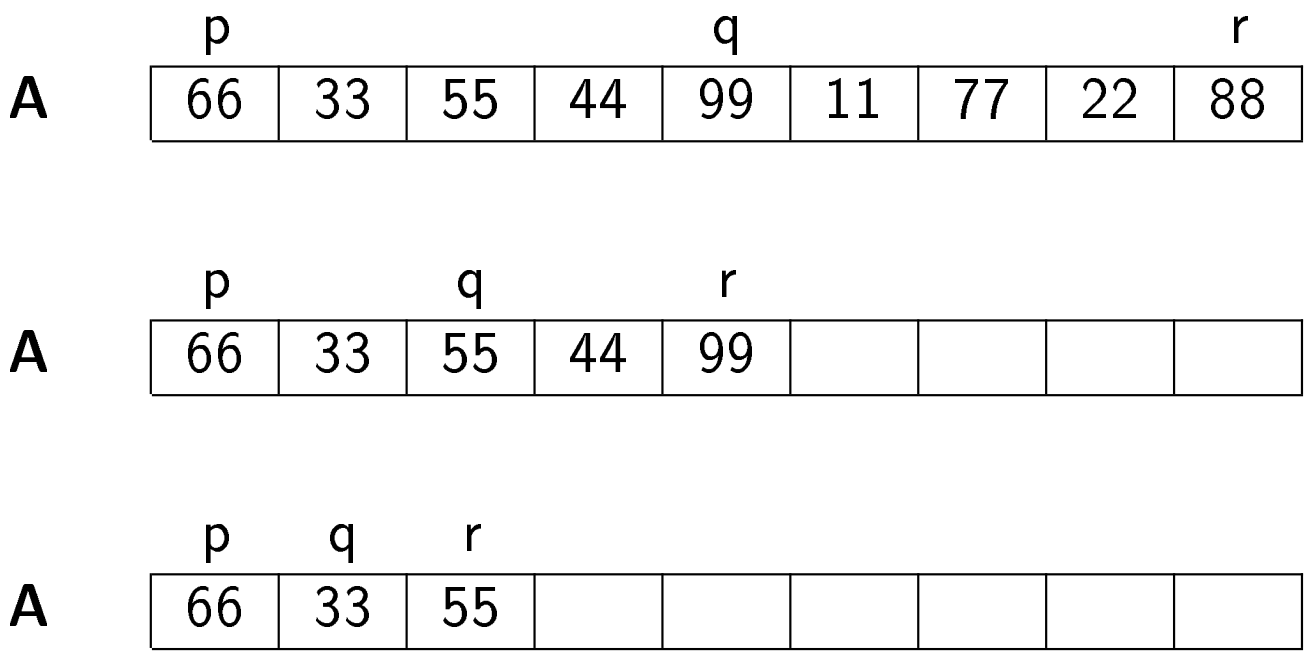
# Ordenação por Intercalação (*Mergesort*)



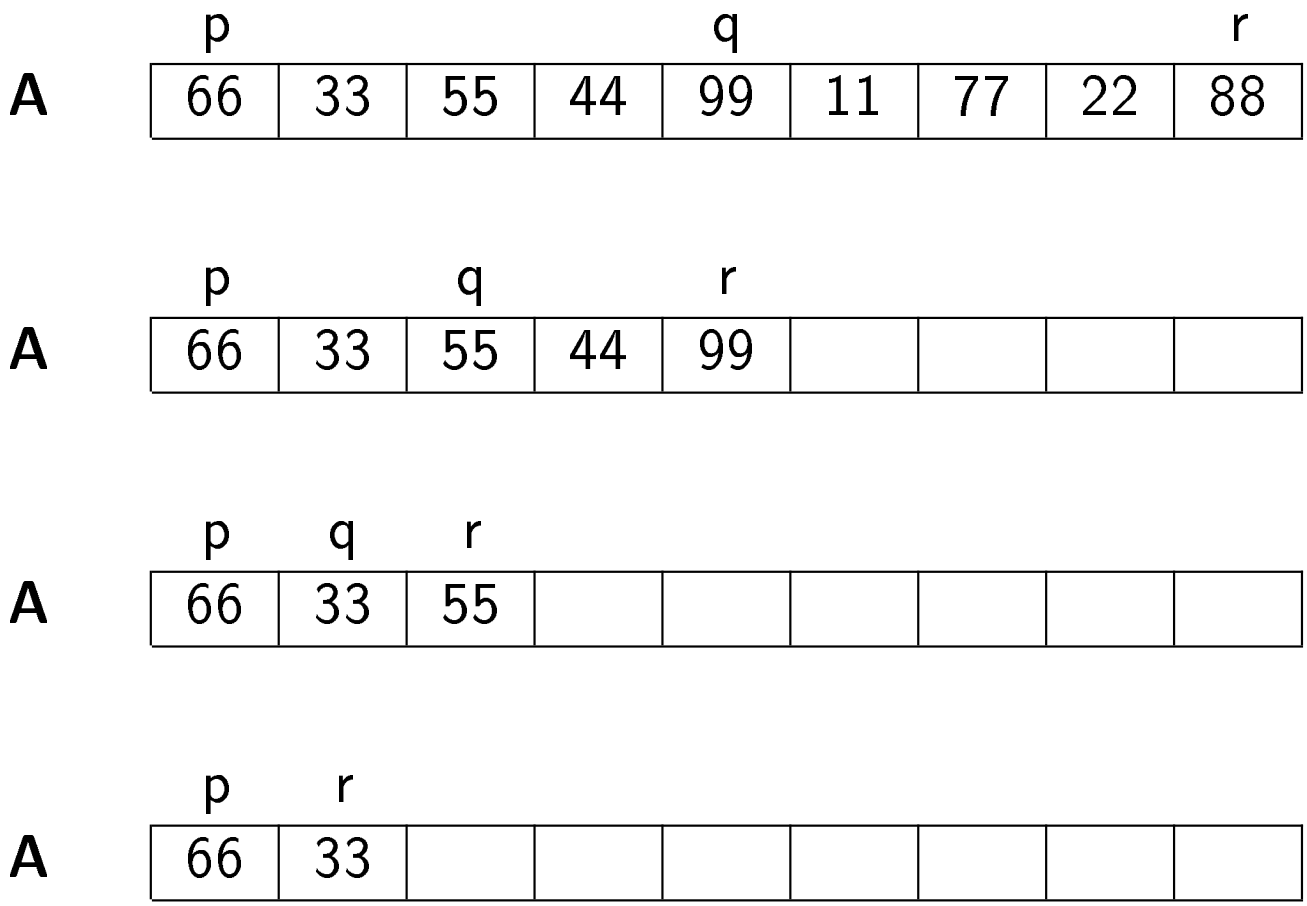
# Ordenação por Intercalação (*Mergesort*)



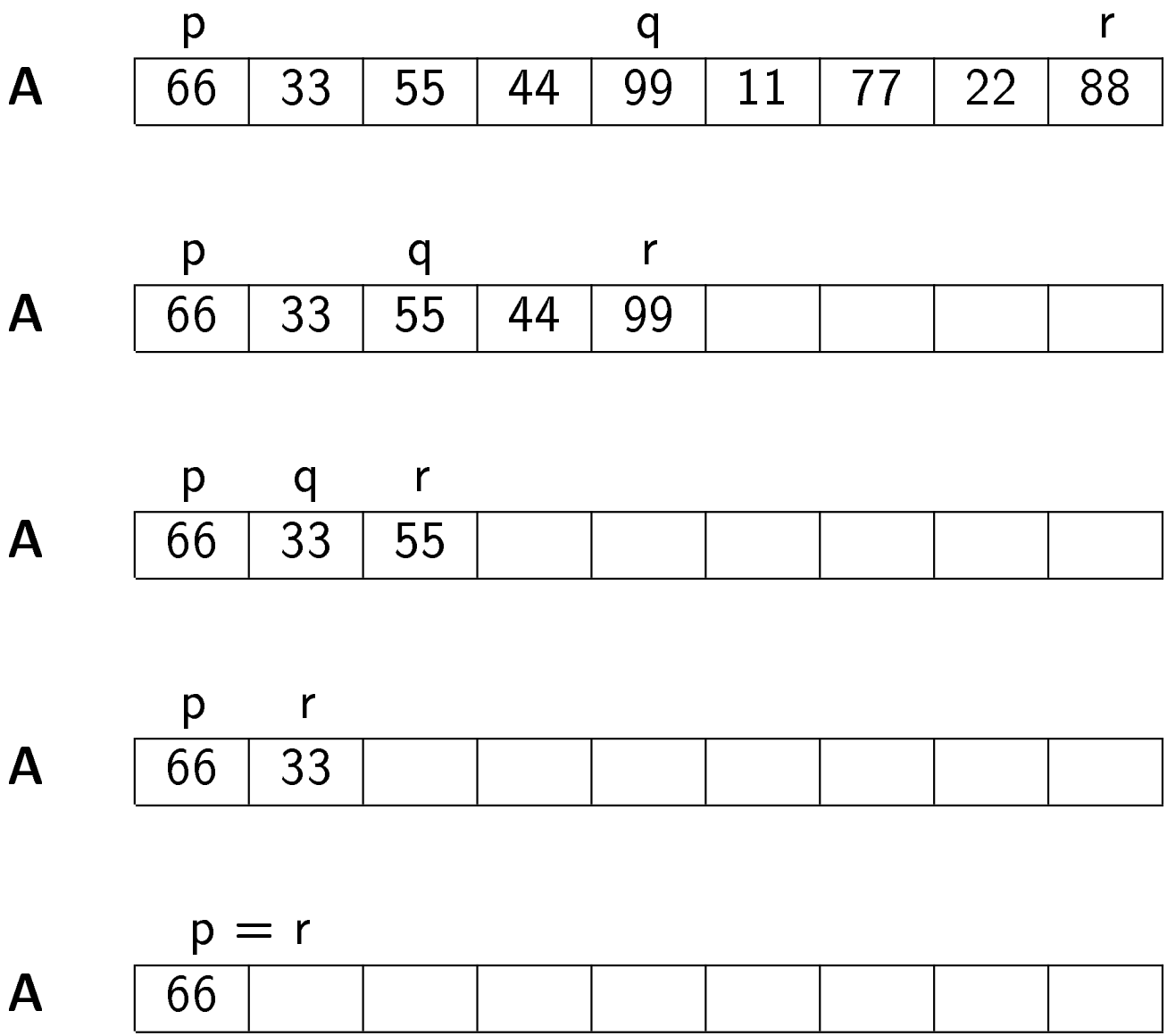
# Ordenação por Intercalação (*Mergesort*)



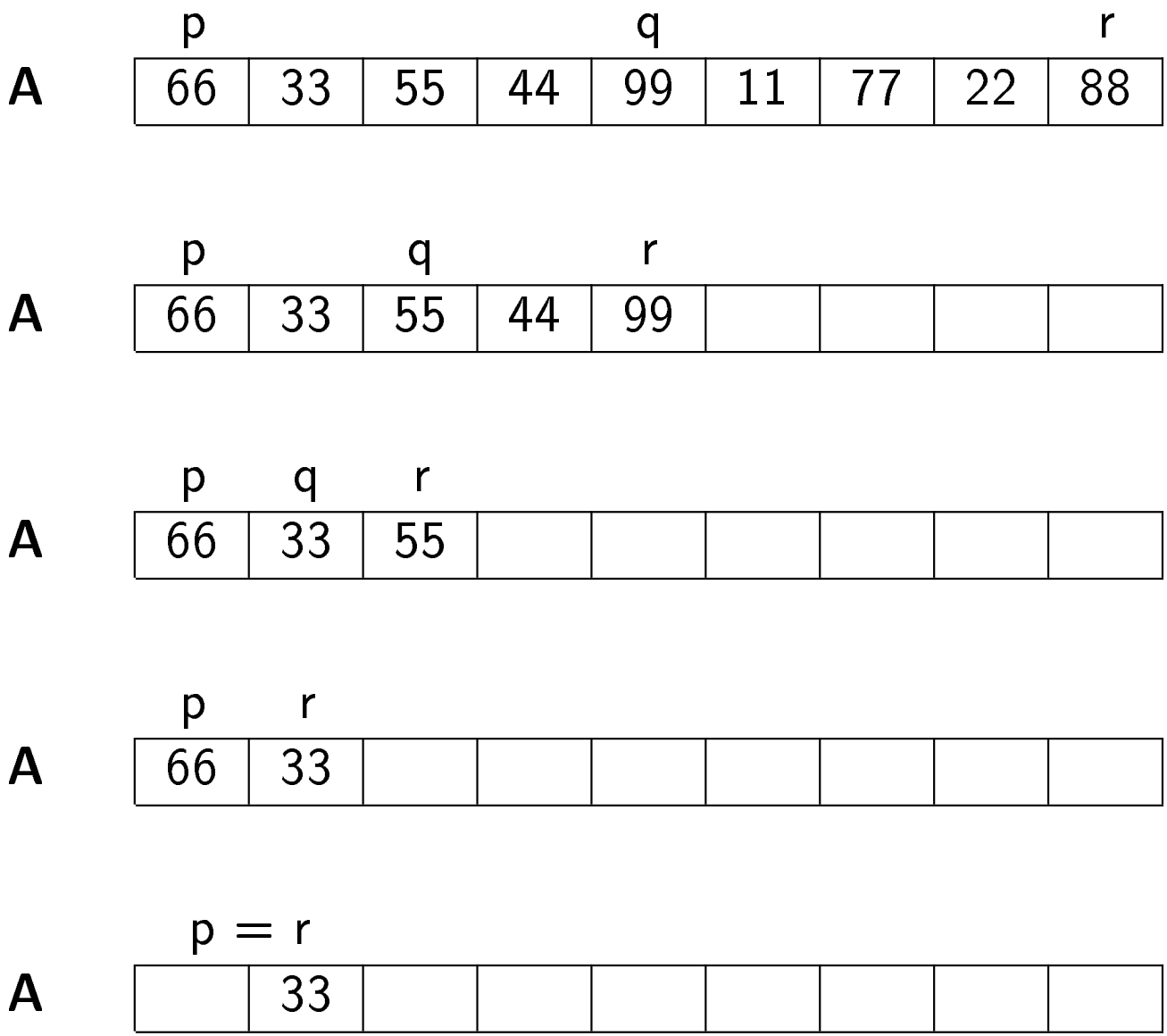
# Ordenação por Intercalação (*Mergesort*)



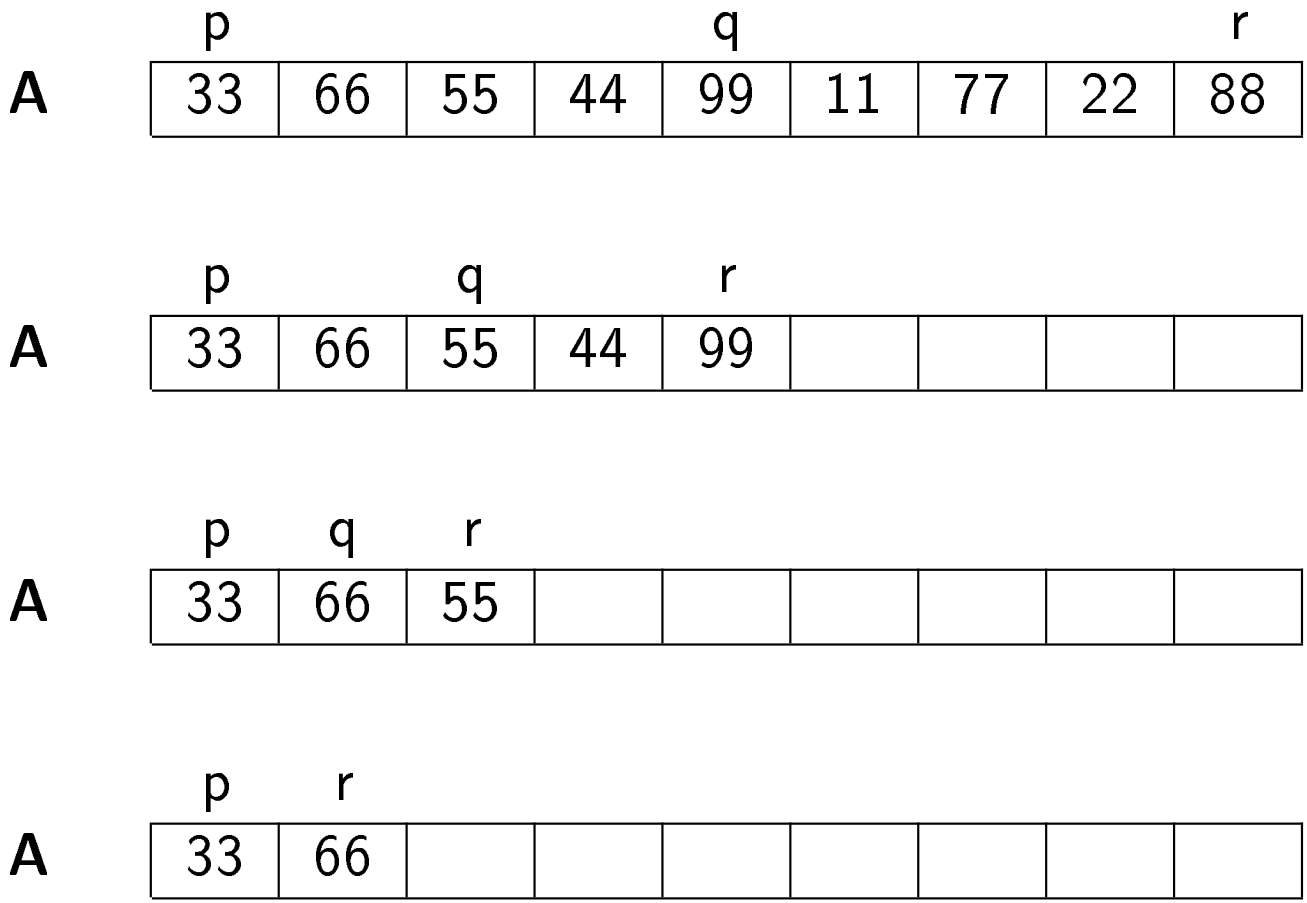
# Ordenação por Intercalação (*Mergesort*)



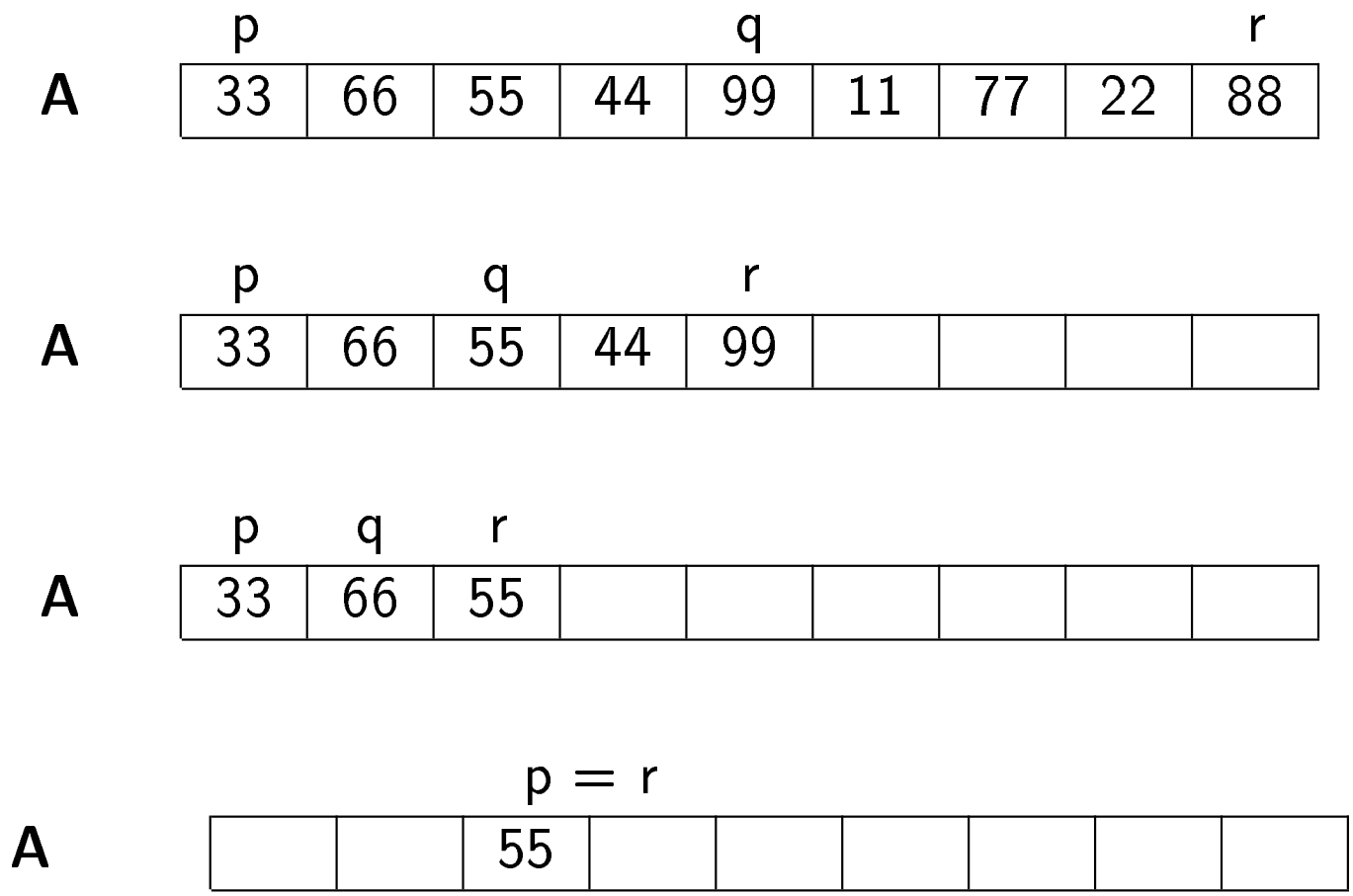
# Ordenação por Intercalação (*Mergesort*)



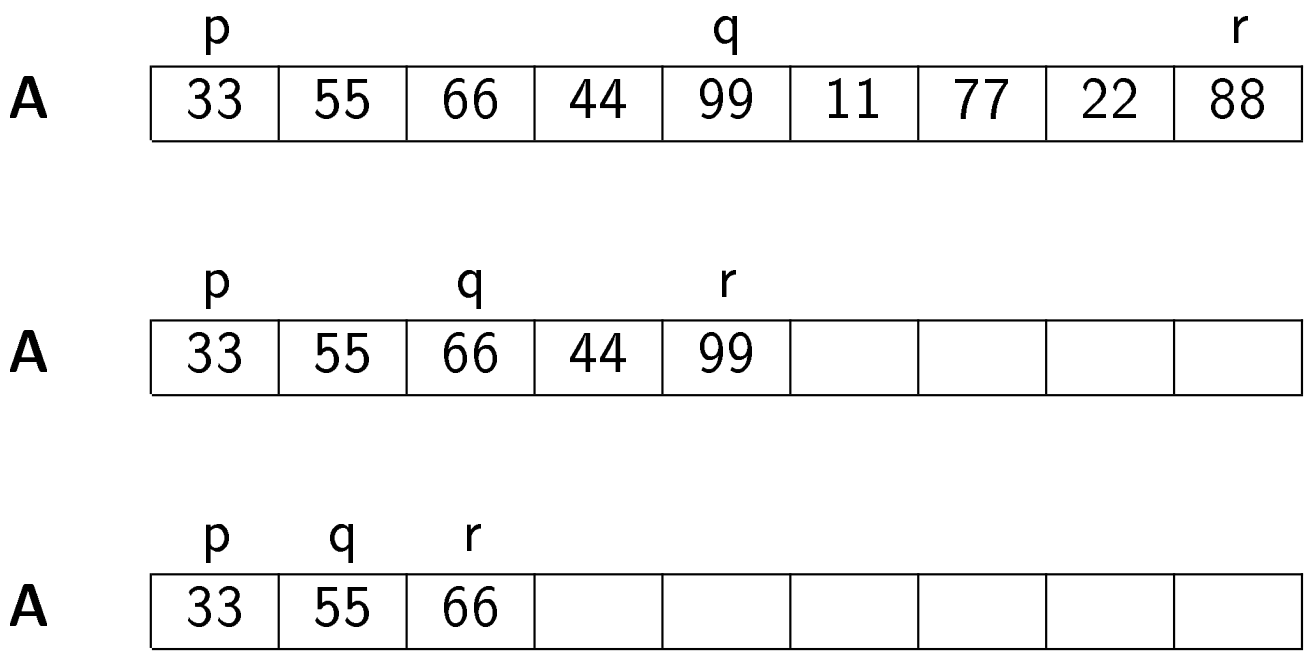
# Ordenação por Intercalação (*Mergesort*)



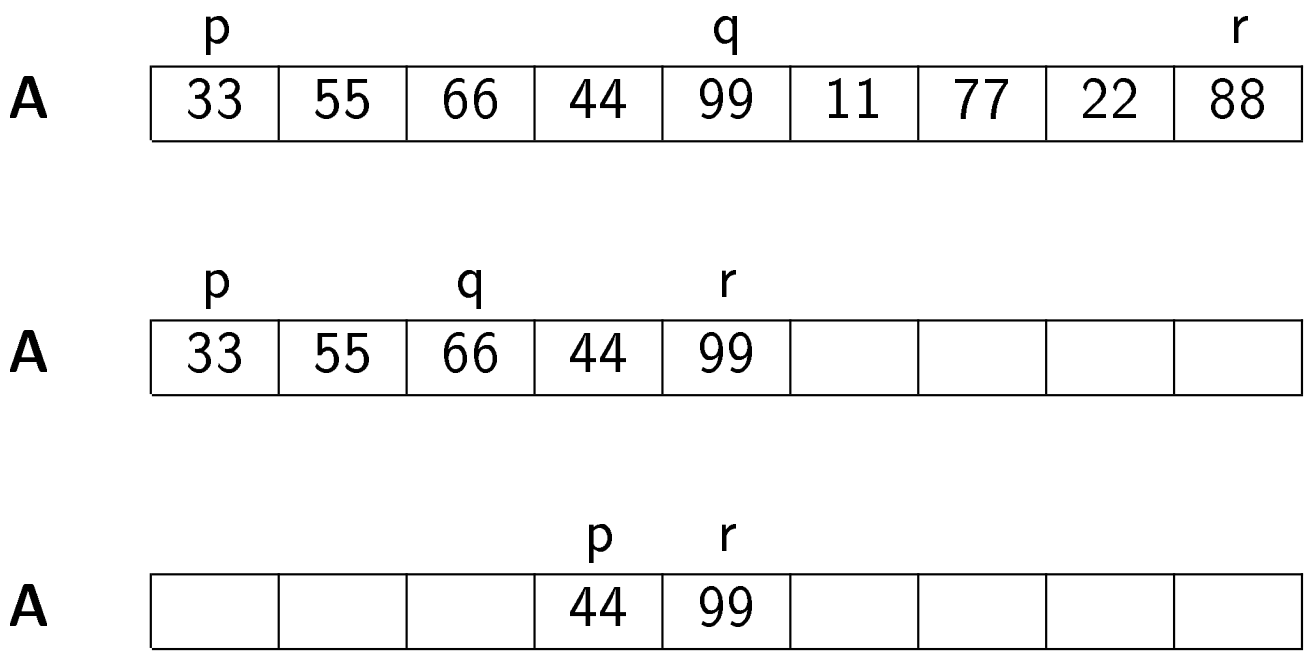
# Ordenação por Intercalação (*Mergesort*)



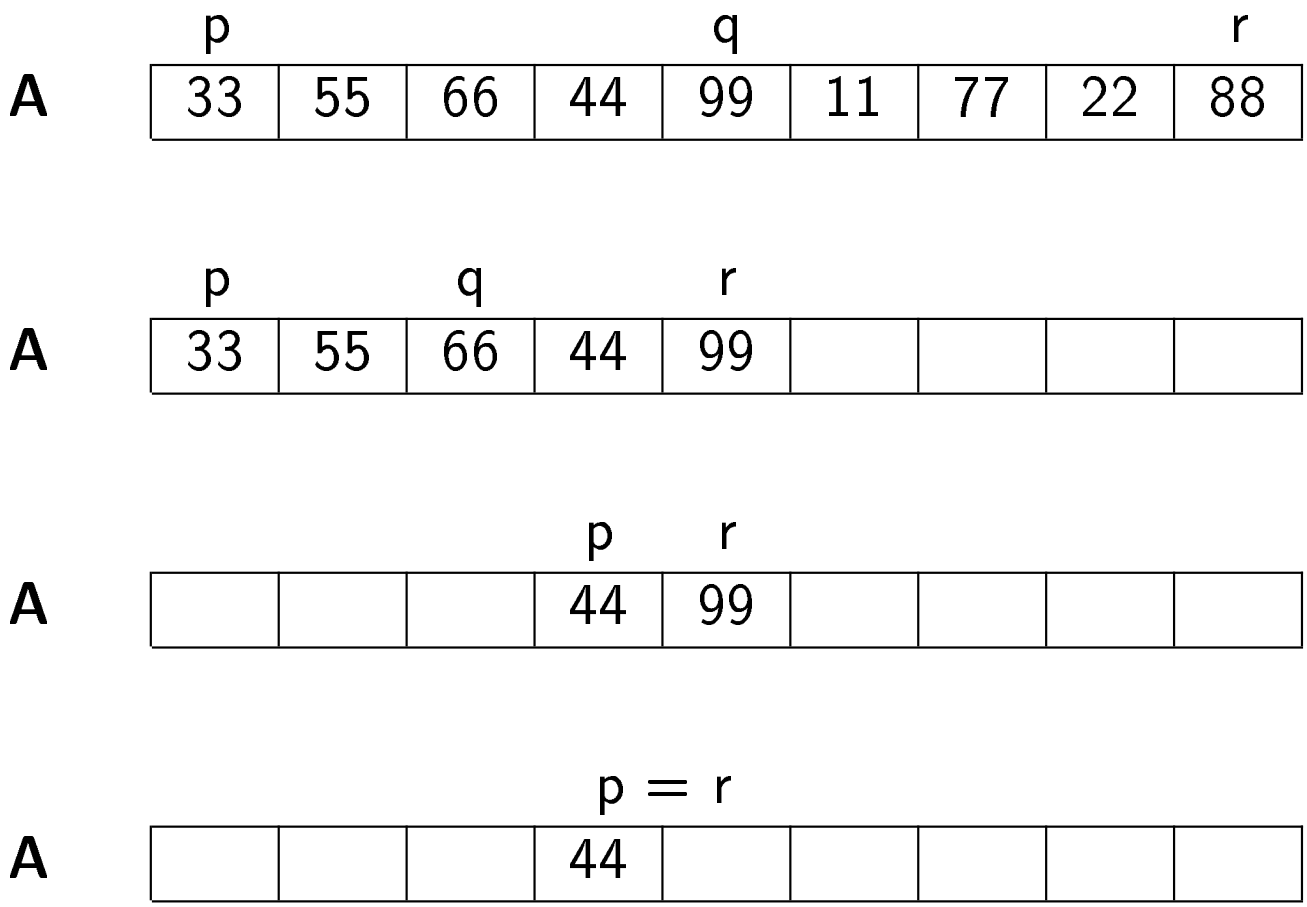
# Ordenação por Intercalação (*Mergesort*)



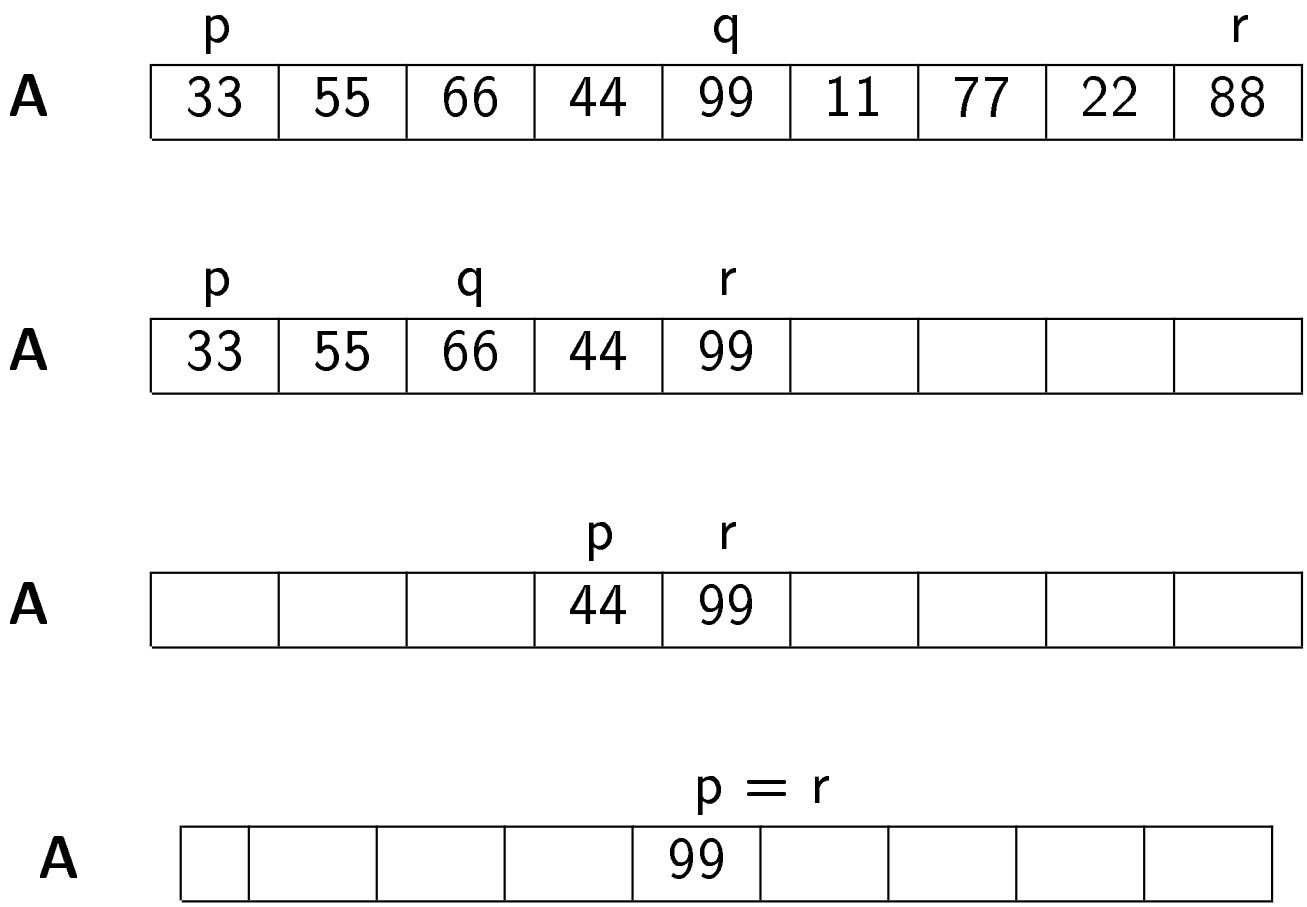
# Ordenação por Intercalação (*Mergesort*)



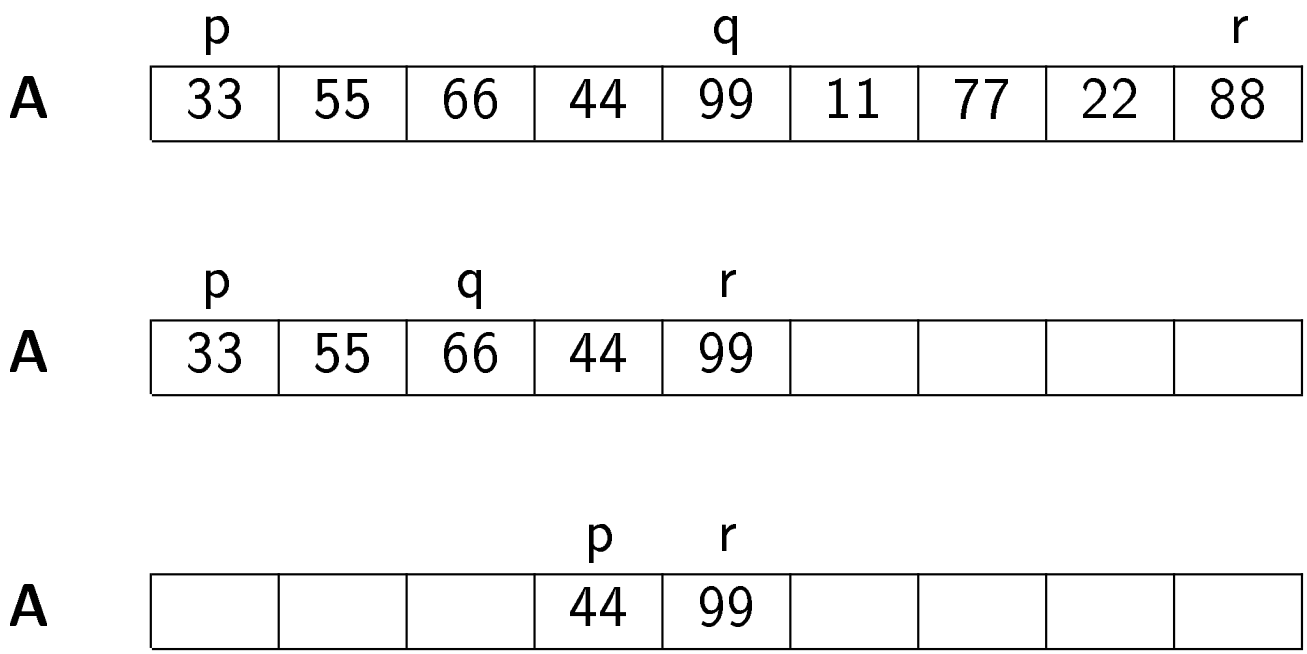
# Ordenação por Intercalação (*Mergesort*)



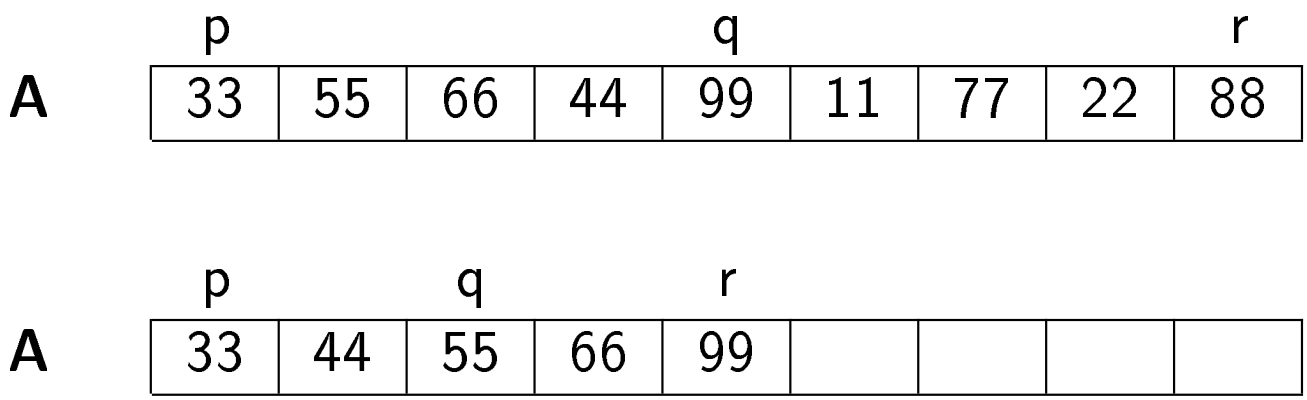
# Ordenação por Intercalação (*Mergesort*)



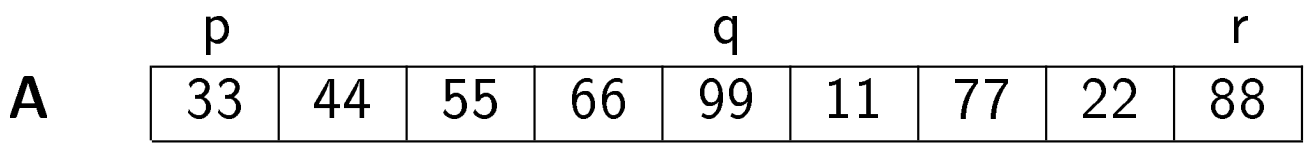
# Ordenação por Intercalação (*Mergesort*)



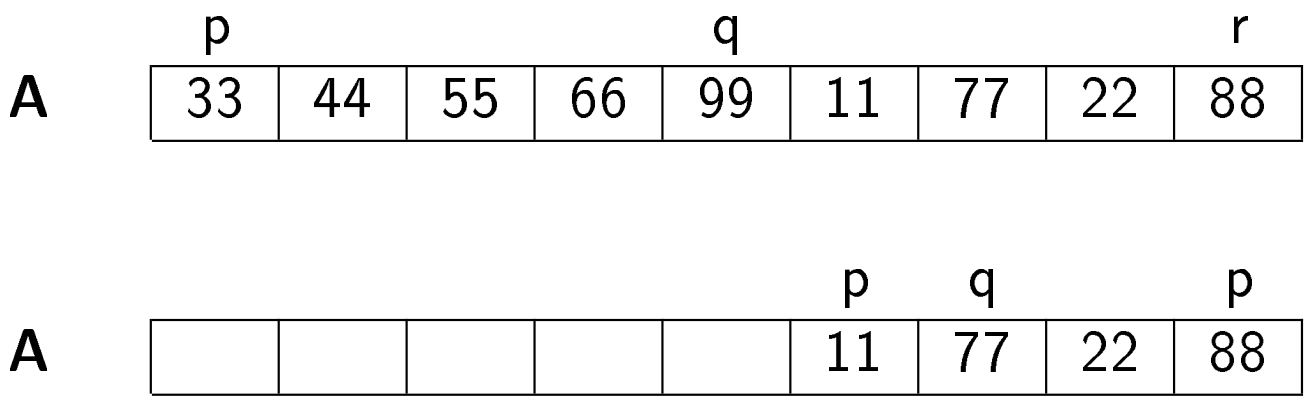
# Ordenação por Intercalação (*Mergesort*)



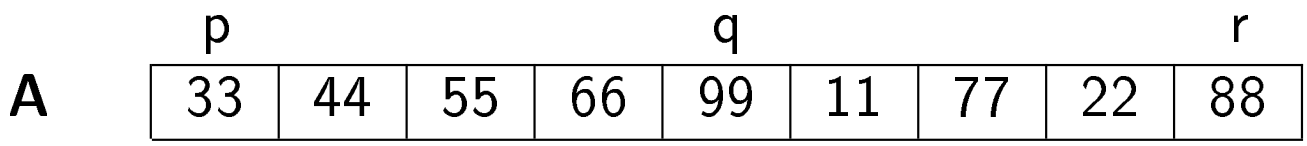
# Ordenação por Intercalação (*Mergesort*)

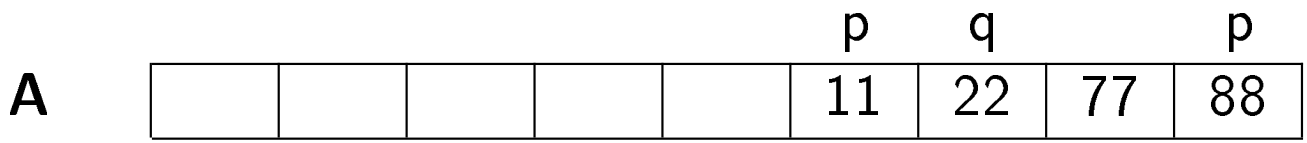


# Ordenação por Intercalação (*Mergesort*)



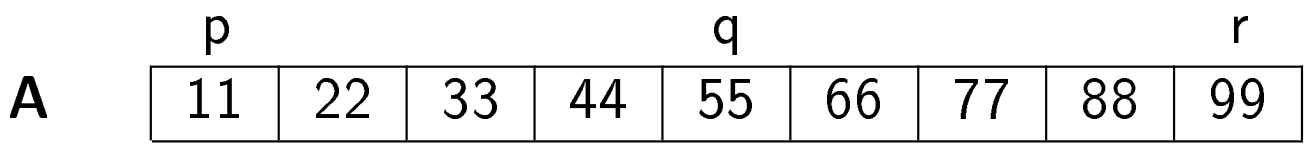
# Ordenação por Intercalação (*Mergesort*)




...

# Ordenação por Intercalação (*Mergesort*)



# Ordenação por Intercalação (*Mergesort*)

- A complexidade do *mergesort* é $O(n \log n)$ no pior caso

Ordenação em Strings

# Ordenação em Strings

- Esse tipo de ordenação é em ordem alfabética
- Comparação de strings é feita caractere por caractere
- Para diferenciar dois números, basta realizar uma única comparação
- Para diferenciar duas strings, a quantidade de comparações é de acordo com o número de caracteres

# Ordenação em Strings

- Na biblioteca *string.h* contém a função *strcmp* para a comparação de cadeias de caracteres:
  - Entrada: duas strings (*str1* e *str2*)
  - Saída:
    - -1: conteúdo de *str1* menor do que *str2*
    - 0: ambas strings são iguais
    - 1: conteúdo de *str1* maior do que *str2*
  - Exemplos
    - *strcmp("goku", "vegeta")* = -1
    - *strcmp("sheena", "sheena")* = 0
    - *strcmp("shaka", "aldebaran")* = 1

# Ordenação em Strings

- Implementação do método de comparação de strings:

```
int comparar_char(char c1, char c2){
  if (c1 == c2) return 0;
  else if (c1 < c2) return -1;
  else return 1;
}


int comparar(char s1[], char s2[]){
  int i;

  for (i = 0; (s1[i] == s2[i]) &&
       (s1[i] != '\0') &&
       (s2[i] != '\0'); i++);

  return comparar_char(s1[i], s2[i]);
}
```

# Ordenação em Strings

- Vetor de strings
  - `char vstr[n][m];`
  - `char *vstr[m];`
  - `char **vstr;`

# Ordenação em Strings

- Exemplo para vetor de string

```
char vstr[5][10];

strcpy(vstr[0], "a");
strcpy(vstr[1], "bb");
strcpy(vstr[2], "ccc");
strcpy(vstr[3], "ddd");
strcpy(vstr[4], "eeee");

printf("%s\n", vstr[0]);
printf("%s\n", vstr[1]);
printf("%s\n", vstr[2]);
printf("%s\n", vstr[3]);
printf("%s\n", vstr[4]);
```

# Ordenação em Strings

- Troca de posição entre duas strings (linhas 0 e 4):

```
strcpy(str, vstr[0]);
strcpy(vstr[0], vstr[4]);
strcpy(vstr[4], str);
```

- A ordenação de strings é custosa
  - Além da quantidade de comparações entre um par de strings, na abordagem apresentada acima ainda teremos que lidar com 3 cópias
  - Assim, serão realizadas $4 * l$ operações em cada movimentação entre strings, onde $l$ é o tamanho de uma string
  - Teria como amenizar esse problema?

# Ordenação em Strings

- O problema apresentado no slide anterior pode ser amenizado por meio do uso de ponteiros:

```
void troca(char **vstr, int p1, int p2){
  char *str;

  str = vstr[p1];
  vstr[p1] = vstr[p2];
  vstr[p2] = str;
}
```

- Para o uso dessa função (ou aplicação de ponteiros), o vetor de strings deve estar alocado dinamicamente

```
char **vstr = (char**) malloc(sizeof(char*) * 5);
char str[20];

for (i = 0; i < 5; i++)
   vstr[i] = (char*) malloc(sizeof(char) * 10);
```

# Ordenação em Strings

- Com o uso de ponteiros, a quantidade de operações para uma troca de posição entre strings passa a ser $l + 3$
  - Por mais que ainda seja necessária a comparação entre strings (complexidade na ordem de $l$ caracteres), a movimentação para a troca de posições de strings passa a ser similar em relação aos dados do tipo numérico
  - É muito menos custoso realizar $l + 3$ operações em comparação com $4l$

# Ordenação em Strings

- Ordenação por bolha (bubblesort) adaptada para vetores de strings

```
void bubblesort(char **vstr, int n){
  int i, j, x, t = 1;

  for (i = 0; (i < n - 1) && t; i++){
    t = 0;

    for (j = 0; j < n - i - 1; j++)
     if (comparar(vstr[j], vstr[j + 1]) > 0){
       troca(vstr, j, j + 1);
       t = 1;
     }
  }
}
```

- Complexidade: $O(l * n^2)$

# Ordenação em Strings

- Algoritmo *quicksort* adaptado para vetores de strings

```
void quicksort(char **vstr, int n_cima, int n_baixo){
   int i = n_cima, j = n_baixo, aux;
   char *pivo = vstr[(i + j) / 2];

   do{
      while ((comparar(vstr[i], pivo) < 0) && (i <= n_baixo))
        i++;

      while ((comparar(vstr[j], pivo) > 0) && (j >= n_cima))
        j--;

      if (i <= j){
        troca(vstr, i, j);
        i++;
        j--;
      }
   }while (i <= j);

   if (j > n_cima)
      quicksort(vstr, n_cima, j);

   if (i < dir)
      quicksort(vstr, i, n_baixo);
}
```

- Complexidade: $O(l * n * \log(n))$

# Ordenação em Strings

- A ordenação de *strings* pode ser feita utilizando qualquer um dos métodos que vimos até o momento em sala de aula

- Entretanto, a ordenação é mais custosa

Ordenação em TAD

# Ordenação em TAD

- A ordenação pode ser aplicada em *structs* também
- Assim, podemos aplicar tanto algoritmos de ordenação de números quanto de strings
- As *structs* podem ter chaves de primárias (*e.g.* registro acadêmico)
- Uma *struct* pode ter chave secundária (*e.g.* código do curso de graduação)

# Ordenação em TAD

- Assim, há diversas formas diferentes de ordenação que podem ser aplicadas em *structs*
  - Listar nome de alunos ordenados por nome
  - Listar nome de alunos ordenados por registro acadêmico
  - Etc
- Assim como para números, não há algoritmo de ordenação que se destaca em relação aos outros para a ordenação de strings e *structs*

# Ordenação em TAD

- Existem inúmeros algoritmos de ordenação que não foram apresentados em sala de aula
  - *Radix sort*
  - *Counting sort*
  - *Shake sort*
  - etc

# Referências I



Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
*Introduction to Algorithms*.
Third edition, The MIT Press, 2009.


Horowitz, E., Sahni, S. Rajasekaran, S.
*Computer Algorithms*.
Computer Science Press, 1998.


Ziviani, N.
*Projeto de Algoritmos - com implementações em Java e C++*.
Thomson, 2007.