# Capítulo 3
# Divisão e Conquista

Manoel Ribeiro Filho

# Projeto de Algoritmos

Existem uma grande variedade de técnicas de projetos de algoritmos.

A ordenação por inserção, que apresentamos, usa a técnica chamada **incremental** :

Tendo ordenado o subarranjo A[1 .. j-1], inserimos o elemento A[j] em seu lugar apropriado, o que produz o subarranjo ordenado A[1 ..j]

Nesta seção estudaremos uma abordagem de projeto alternativa a **divisão e conquista**.
Usaremos tal abordagem para projetar um algoritmo de ordenação cujo tempo de execução do pior caso é muito menor que o da ordenação por inserção.

# 3.1 A abordagem de Divisão e Conquista

Dividem o problema em vários subproblemas, semelhantes ao problemas original, mas de menor tamanho.
Resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

O paradigma de divisão e conquista envolve três passos em cada nível de recursão:

**Divisão** do problema em determinado número de subproblemas que são instâncias menores do problema original

**Conquista os** subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.

**Combinação** Combina as soluções dadas aos subprogramas para resolver o problema original

# Ordenação por **Divisão e Conquista** chamado de **MERGE-SORT**

Obedece rigorosamente o paradigma de divisão e conquista.
Seus três passos são:

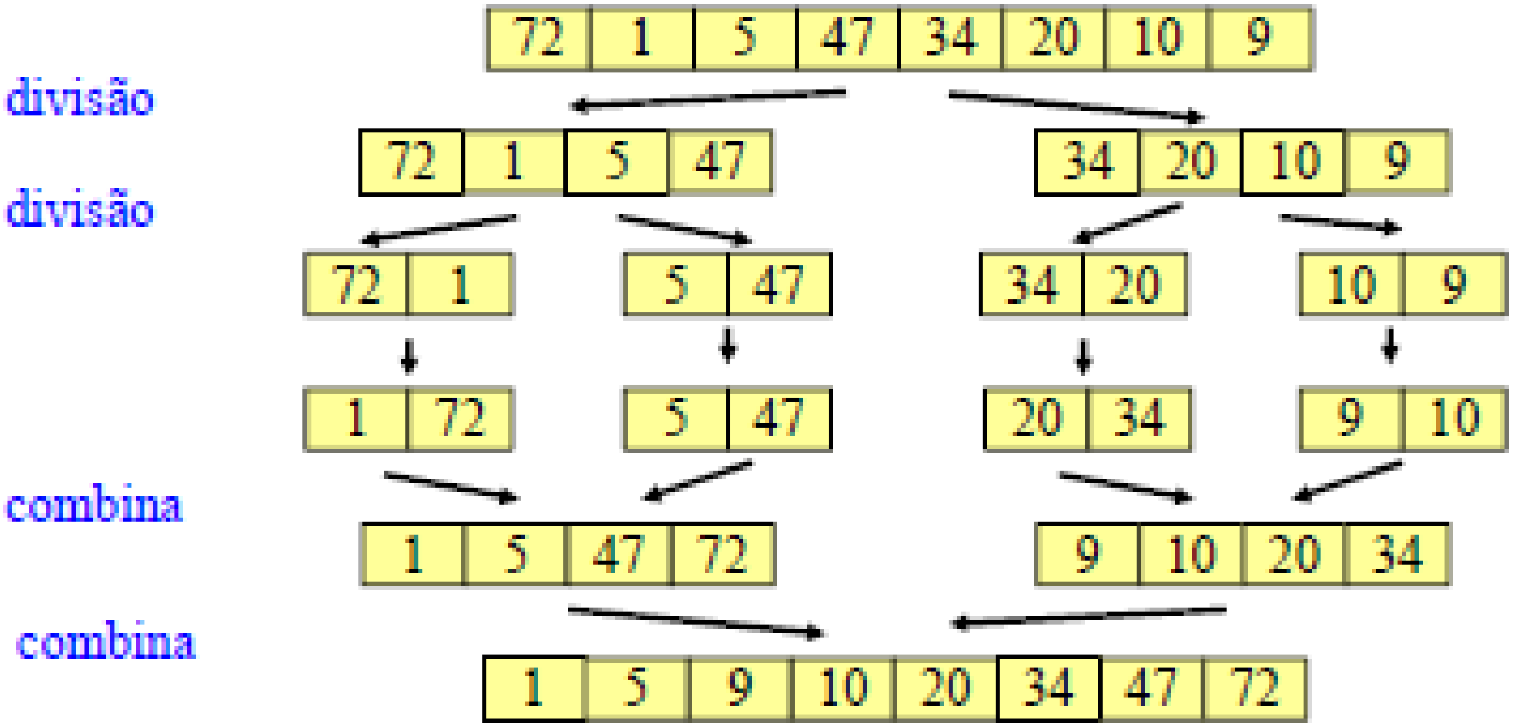
**Divisão**: divide a sequencia de n elementos que deve ser ordenada em duas subsequências de n/2 elementos cada uma.

**Conquista**: Ordena as duas subsequências recursivamente, utilizando a ordenação por intercalação.

**Combinação**: Intercala as duas subsequências ordenadas para produzir a resposta ordenada

A recursão finaliza quando a sequência a ser ordenada tiver comprimento 1, visto que, nesse caso não há nenhum trabalho a ser feito, já que toda sequencia de comprimento 1 já está ordenada.

# MergeSort



72 1 5 47 34 20 10 9
divisão
72 1 5 47
34 20 10 9
divisão
72 1
5 47
34 20
10 9
1 72
5 47
20 34
9 10
combina
1 5 47 72
9 10 20 34
combina
1 5 9 10 20 34 47 72

A operação chave do algoritmo MERGE-SORT, é a **intercalação** de suas sequências ordenadas, no passo de combinação.

Para executar essa intercalação vamos chamar um procedimento (função) auxiliar Intercala(A, p, q, r), onde A é um arranjo (vetor) e p, q e r são índices dos elementos do vetor tal que p<=q<r.

O procedimento considera que os subvetores A[p .. q] e A[q+1 .. r] estão já ordenados.

Esse procedimento os intercala( ou mescla) para formar um único subvetor ordenado que substitui o arranjo atual A[p .. r].

# Ordenação por intercalação

Q que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p \dots q]$ e $A[q{+}1 \dots r]$ crescentes, rearranjar $A[p \dots r]$ de modo que ele fique em ordem crescente.

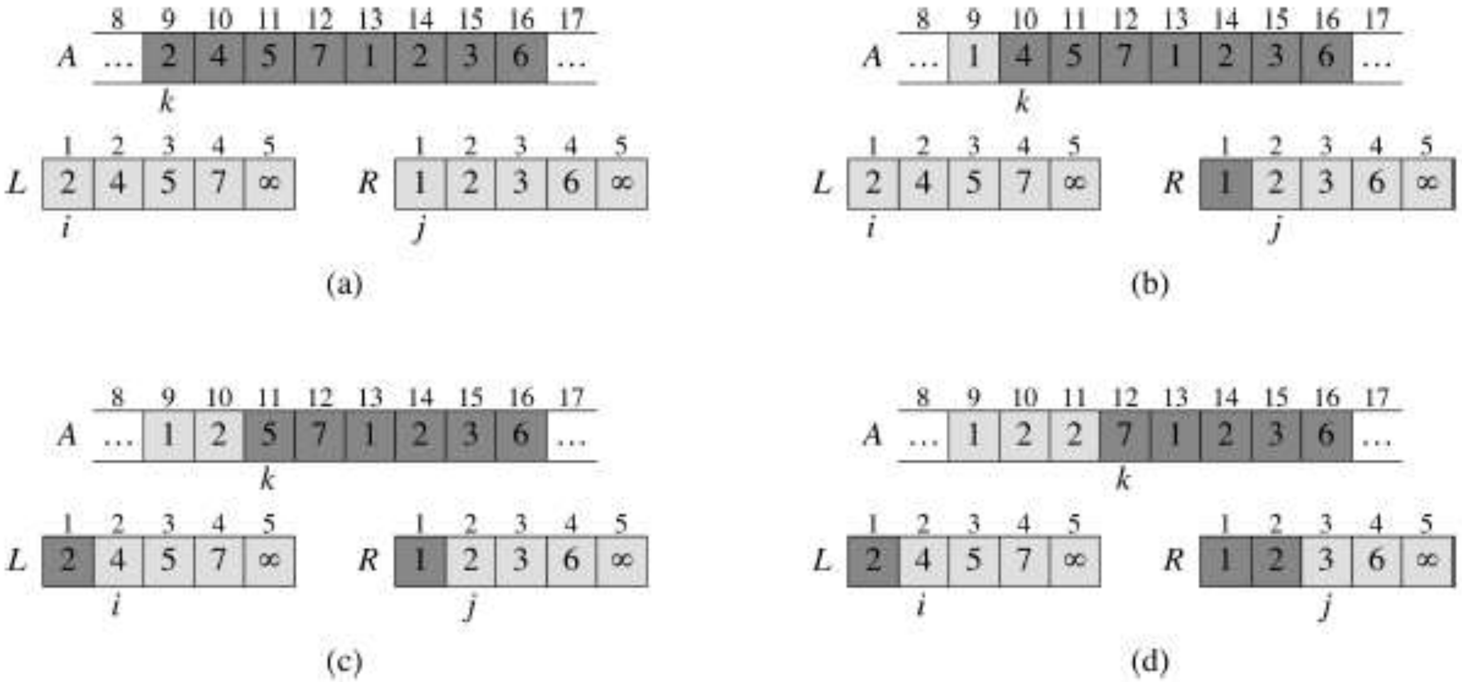
Entrada:

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

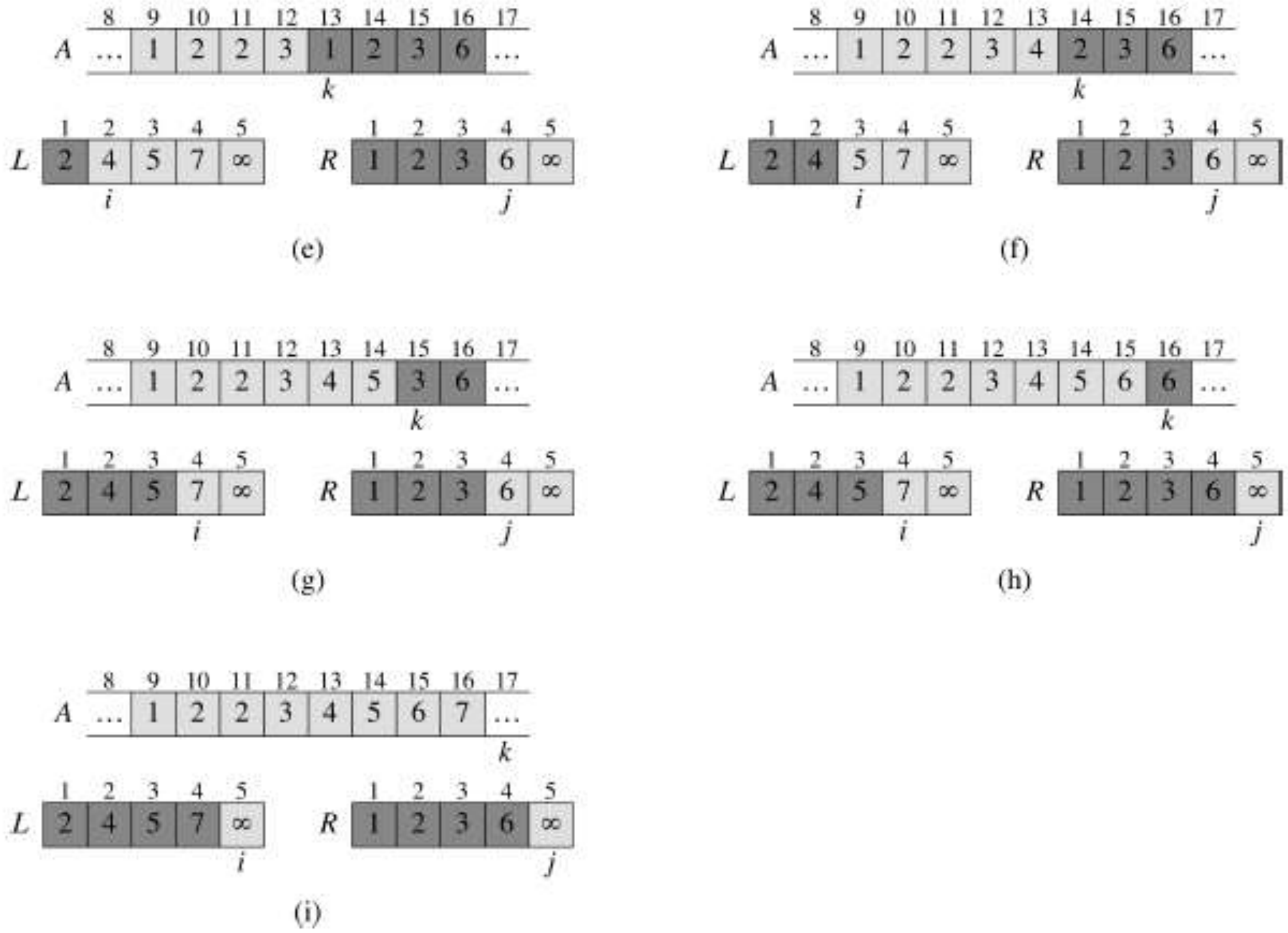
Saída:

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

# Intercalação com sentinela



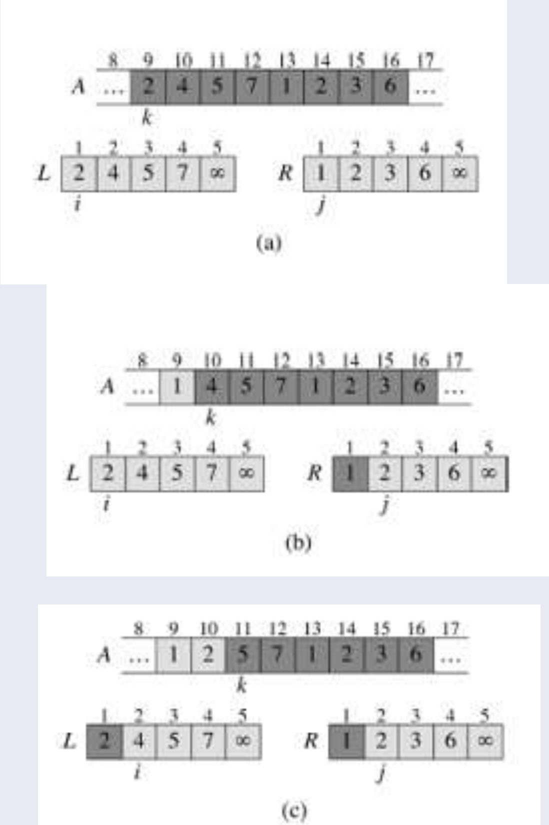
# Intercalação com sentinela



# Intercalação com sentinela

INTERCALA($A, p, q, r$)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
sejam $L[1..n_1 + 1]$ e $R[1..n_2 + 1]$ novos vetores
**para** $i \leftarrow 1$ **até** $n_1$ **faça**
    $L[i] \leftarrow A[p + i - 1]$
**para** $j \leftarrow 1$ **até** $n_2$ **faça**
    $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
**para** $k \leftarrow p$ **até** $r$ **faça**
    **se** $L[i] \leq R[j]$ **então**
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    **senão**
        $A[k] = R[j]$
        $j \leftarrow j + 1$



(a) (b) (c)

O procedimento Intercala (Merge em inglês) funciona da maneira ilustrada a seguir.
A linha 1 calcula o comprimento $n1$ do subarranjo $A[p .. q]$ e a linha 2 calcula o comprimento $n2$ do subarranjo $A[q + 1.. r]$. No nosso exemplo,


p=9, q=12 e r=16

Criamos os arranjos $L$ e $R$ (de "*left*" e "*right*", em inglês, ou "esquerda" e "direita") de comprimentos $n1 + 1$ e $n2 + 1$, respectivamente, na linha 3; a posição extra em cada arranjo conterá a sentinela.

O laço **for** das linhas 4 e 5 copia o subarranjo $A[p .. q]$ em $L[1 ..n1]$, e o laço **for** das linhas 6 e 7 copia o subarranjo $A[q + 1 .. r]$ em $R[1 .. n2]$.

As linhas 8 e 9 colocam as sentinelas nas extremidades dos arranjos $L$ e $R$.

A linha 10 faz i=1 e a linha 11 faz j=1

Operação das linhas 12 a 18 na chamada Intercala($A$, 9, 12, 16) quando o subarranjo $A$[9 .. 16] contém a sequência ⟨2, 4, 5, 7, 1, 2,3, 6⟩.
Depois de copiar e inserir sentinelas, o arranjo $L$ contém ⟨2, 4, 5, 7, ∞⟩ e o arranjo $R$ contém (1, 2, 3, 6, ∞).
Posições sombreadas em tom mais claro em $A$ contêm seus valores finais, e posições sombreadas em tom mais claro em $L$ e $R$ contêm valores que ainda têm de ser copiados de volta em $A$.
Juntas, as posições sombreadas em tom mais claro sempre incluem os valores contidos originalmente em $A$ [9 ..16], além das duas sentinelas.
Posições sombreadas em tom mais escuro em $A$ contêm valores que serão copiados, e posições em tom mais escuro em $L$ e $R$ contêm valores que já foram copiados de volta em $A$. **(a)-(h)** Os arranjos $A$, $L$ e $R$ e seus respectivos índices $k$, $i$, e $j$ antes de cada iteração do laço das linhas 12 a 18.

# 3.2 Invariante do Laço para o algoritmo de Intercalação

No início de cada iteração do laço **for** das linhas 12 a 18, o subarranjo $A[p \mathinner{..} k-1]$ contém os $k-p$ menores elementos de $L[1 \mathinner{..} n1+1]$ e $R[1 \mathinner{..} n2+1]$, em sequência ordenada. Além disso, $L[i]$ e $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em $A$.
Devemos mostrar que esse invariante de laço é válido antes da primeira iteração do laço **for** das linhas 12 a 18, que cada iteração do laço mantém o invariante e que o invariante fornece uma propriedade útil para mostrar correção quando o laço termina.

**Inicialização:** Antes da primeira iteração do laço, temos $k = p$, de modo que o subarranjo $A[p \mathinner{..} k-1]$ está vazio. Esse subarranjo vazio contém os $k - p = 0$ menores elementos de $L$ e $R$ e, uma vez que $i = j = 1$, tanto $L[i]$ quanto $R[j]$ são os menores elementos de seus arranjos que não foram copiados de volta em $A$.

**Manutenção:** Para ver que cada iteração mantém o invariante de laço, vamos supor primeiro que $L[i] \leq R[j]$.
Então, $L[i]$ é o menor elemento ainda não copiado de volta em $A$. Como $A[p \mathinner{..} k-1]$ contém os $k-p$ menores elementos, depois de a linha 14 copiar $L[i]$ em $A[k]$, o subarranjo $A[p \mathinner{..} k]$ conterá os $k-p+1$ menores elementos. O incremento de $k$ (na atualização do laço **for**) e de $i$ (na linha 15) restabelece o invariante de laço para a próxima iteração. Se, em vez disso, $L[i] > R[j]$, então as linhas 16 e 17 executam a ação apropriada para manter o invariante de laço.

**Término:** No término, $k = r + 1$. Pelo invariante de laço, o subarranjo $A[p \mathinner{..} k-1]$, que é $A[p \mathinner{..} r]$, contém os $k-p = r-p+1$ menores elementos de $L[1 \mathinner{..} n1+1]$ e $R[1 \mathinner{..} n2+1]$ em sequência ordenada. Os arranjos $L$ e $R$ juntos contêm $n1 + n2 + 2 = r - p + 3$ elementos. Todos os elementos, exceto os dois maiores, foram copiados de volta em $A$, e esses dois maiores elementos são as sentinelas.

# 3.3 Complexidade do algoritmo da Intercalação

Para ver que o procedimento Merge é executado no tempo $\Theta(n)$, onde $n = r - p + 1$, observe que cada uma das linhas 1 a 3 e 8 a 11 demora um tempo constante, que os laços **for** das linhas 4 a 7 demoram o tempo $\Theta(n1 + n2) = \Theta(n)$ e que há $n$ iterações do laço **for** das linhas 12 a 18, cada uma demorando um tempo constante, a complexidade do algoritmo de intercalação é $\Theta(n)$

# 3.4 O Algoritmo Merge Sort - Classificar Intercalando

Agora podemos usar o procedimento Intercala como uma subrotina no algoritmo de ordenação por intercalação.

O procedimento MERGE-SORT(A , p , r) ordena os elementos do vetor A[p .. r] .

Se p >= r, o vetor tem no máximo um elemento e, portanto já está ordenado (caso base)

Caso contrário, a etapa de divisão simplesmente calcula um índice q que subdivide A[p .. r] em dois subarranjos: A[p .. q] , contendo [n/2] elementos , e A[q+1 .. r] , contendo [n/2] elementos. (caso recursivo)

## Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGESORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGESORT(A, p, q)
              MERGESORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

| 5 | 2 | 4 | 7 | 1 | 3 | 2 | 6 |
|---|---|---|---|---|---|---|---|

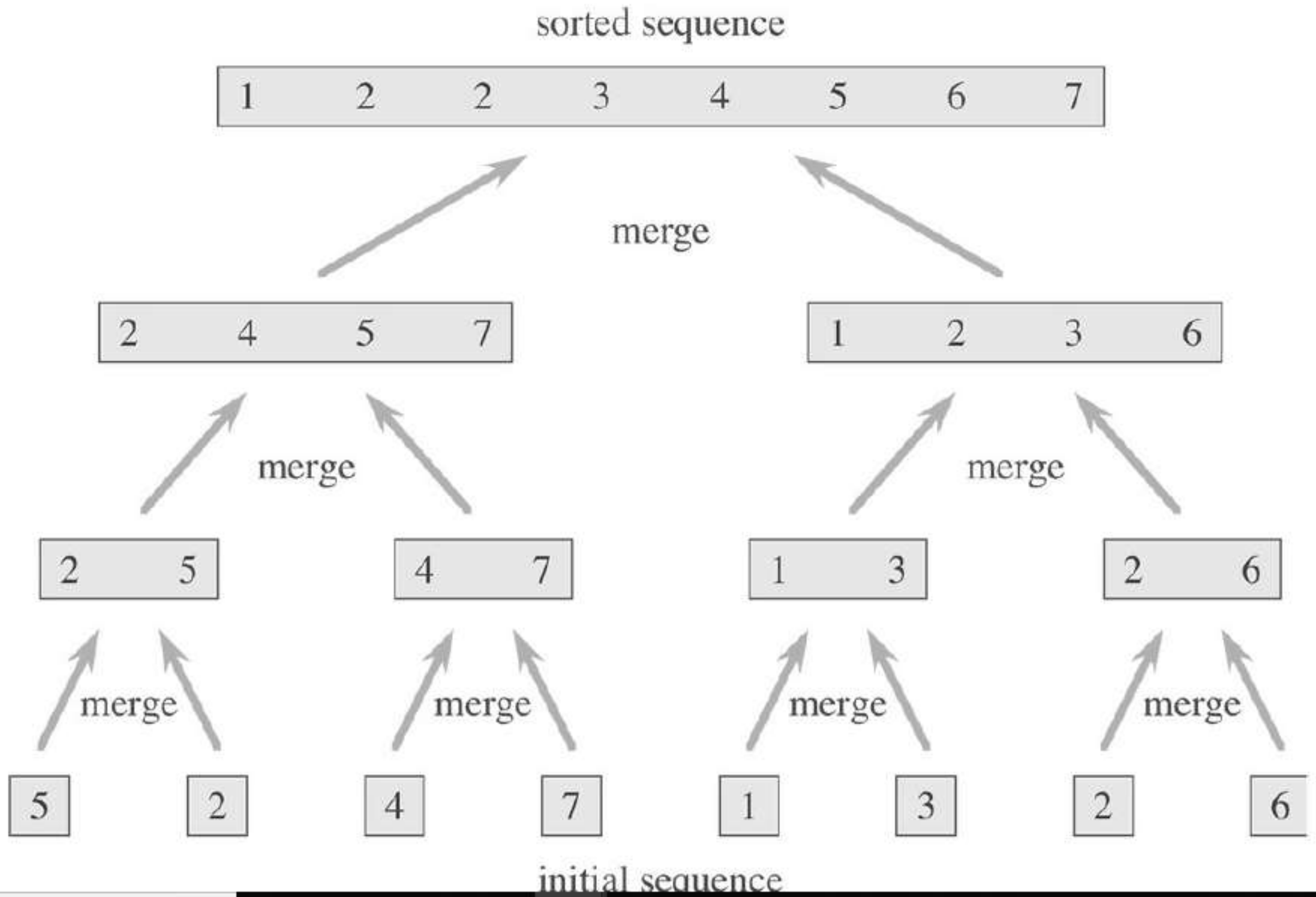
Supor o vetor A de entrada, com p=1 e r=8, como p<r, na primeira interação, em divisão inteira q=4 , e começa o processo de decomposição de uma função recursiva de uma linguagem de programação. Na primeira passagem os parâmetros, da chamada recursiva são: Mergesort(A,1,4) e Mergesort(A,5,8), e começa o processo de decomposição, E a função Intercala não é chamada, ela só será chamada quando p>=r, e iniciar o processo de composição .

5 2 4 7 — 5 2 — 4 7 — 5 — 2 — 4 — 7

1 3 2 6 — 1 3 — 2 6 — 1 — 3 — 2 — 6

Agora vai se iniciar o processo de composição, no qual é chamada a função intercala.

O algoritmo consiste em intercalar pares de sequências com 1 item para formar sequências ordenadas de comprimento 2, intercalar pares de sequências de comprimento 2 para formar sequências ordenadas de comprimento 4, e assim por diante, até que duas sequências de comprimento *n*/2 sejam intercaladas para formar a sequência ordenada final de comprimento *n*, mostrada na próxima transparência.


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence