Algoritmos de Ordenação

MergeSort

Professora:

Fátima L. S. Nunes

SISTEMAS DE INFORMAÇÃO

EACH

Algoritmos de Ordenação

- Grande parte das operações de Sistemas de Computação é constituída por buscas em bases de dados.
- Exemplos?

Algoritmos de Ordenação

Grande parte das operações de Sistemas de Computação é constituída por buscas em bases de dados.

- Exemplos?
  - consultar saldo bancário, fornecendo número da conta corrente;
  - consultar nota no sistema Júpiter, fornecendo número USP;
  - consultar preço de um livro em uma loja, fornecendo seu código.

Algoritmos de Ordenação

- Grande parte das operações de Sistemas de Computação é constituída por buscas em bases de dados.
- Para essas operações, os dados devem estar ordenados.
- Algoritmos de ordenação constituem uma classe muito estudada de algoritmos.
- Por quê?

Algoritmos de Ordenação

- Grande parte das operações de Sistemas de Informações é constituída por buscas em bases de dados.
- Para essas operações, os dados devem estar ordenados.
- Algoritmos de ordenação constituem uma classe muito estudada de algoritmos.
  - é impossível pensar em buscas sem ordenação;
  - buscas exigem que os dados estejam organizados;
  - volume de dados geralmente é grande.

Algoritmos de Ordenação

- Que algoritmos de ordenação já conhecemos ?

Algoritmos de Ordenação

- Que algoritmos de ordenação já conhecemos (ICC1)?
  - *Insertion Sort* (Ordenação por Inserção)
  - *Selection Sort* (Ordenação por Seleção)
  - *Bubble Sort* (Ordenação pelo método da Bolha)

  - Já analisamos a complexidade do algoritmo *Insertion Sort*: $O(n^2)$

MergeSort

- Ordenação por intercalação:
    - Dada uma sequência de *n* elementos:
        - divide o arranjo em duas subsequências de *n/2* elementos;
        - classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação;
        - faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

MergeSort

- Ordenação por intercalação:
  - Dada uma sequência de *n* elementos:
    - divide o arranjo em duas subsequências de *n/2* elementos;
    - classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação;
    - faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Que técnica de algoritmo é esta?

MergeSort

- Ordenação por intercalação:
  - Dados uma sequência de *n* elementos:
    - Dividir: divide o arranjo em duas subsequências de *n/2* elementos;
    - Conquistar: classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação;
    - Combinar: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Que técnica de algoritmo é esta? Dividir e Conquistar!

MergeSort

- Recursão não funciona quando a sequência a ser ordenada tem comprimento 1: neste caso não há trabalho a ser feito.
- Operação chave do algoritmo MergeSort:
  - intercalação de duas sequências ordenadas
- Algoritmo (analogia com baralho):
  1. há duas pilhas com cartas ordenadas;
  2. menor carta está com a face para cima em cada pilha;
  3. pegar menor delas e coloca em nova pilha, com face para baixo;
  4. repete 3 até terminar uma das pilhas;
  5. junta cartas da pilha restante.

MergeSort

- Algoritmo:

```
merge (A,p,q,r) // A=arranjo; p,q,r=índices, p≤ q < r
// A[p..q] e A [q+1..r] estão ordenados
// intercala os subarranjos para formar novo arranjo A

// define subarranjos
n1 ← q−p+1
n2 ← r−q

// popular subarranjos
criar arranjos L[1..n1+1] e R [1..n2+1]
para i ← 1 até n1
 L[i] ← A[p+i-1]
para j ← 1 até n2
 faça R[j] ← A[q+j]

// continua...
```

MergeSort

- Algoritmo:

```
merge (A,p,q,r) // A=arranjo;p,q,r=índices, p≤ q < r

// definir sentinela (para evitar testar se chegou fim)
L[n1+1] ← ∞
L[n2+1] ← ∞

//  mesclar subarranjos
para k ← p até r
 se L[i] ≤ R[j]
      A[k] ← L[i]
      i ← i + 1
 senão
      A[k] ← R[j]
      j ← j + 1
 fim se
fim para
```

MergeSort

- Complexidade do algoritmo Merge:
- A cada passo:
  - verifica 2 posições atuais (cartas superiores da pilha de baralho) e decide qual pegar;
  - tempo constante em cada passo;
  - Quantas vezes executa este passo?

MergeSort

- Complexidade do algoritmo Merge:
- A cada passo:
  - verifica 2 posições atuais (cartas superiores da pilha de baralho) e decide qual pegar;
  - tempo constante em cada passo;
  - Quantas vezes executa este passo? n vezes
  - Portanto, complexidade = O(n) , onde n=r-p+1

MergeSort

- Agora que já sabemos como mesclar os subarranjos, vamos definir o algoritmo recursivo da ordenação por intercalação:
- Dados A, p, r:
  - ordena elementos do subarranjo A [p..r];
  - se $p \geq r$, o subarranjo tem no máximo um elemento (já está ordenado);
  - caso contrário, faz a divisão: calcula um índice q que particiona A [p..r] em dois subarranjos: A[p..q] contendo n/2 elementos e A[q+1..r] contendo n/2 elementos.

MergeSort

- Dados A, p, r:
  - ordena elementos do subarranjo A [p..r];
  - se p≥r, o subarranjo tem no máximo um elemento (já está ordenado);
  - caso contrário, faz a divisão: calcula um índice q que particiona A [p..r] em dois subarranjos: A[p..q] contendo n/2 elementos e A[q+1..r] contendo n/2 elementos.

```
mergeSort (A,p,r)

???
```

MergeSort

```
mergeSort (A,p,r)

se p < r
     q ← ⌊(p+r)/2⌋
     mergeSort(A, p, q)
     mergeSort(A, q+1,r)
     merge(A,p,q,r)
```

MergeSort

```
mergeSort (A,p,r)

se p < r
      q ← ⌊(p+r)/2⌋
      mergeSort(A, p, q)
      mergeSort(A, q+1,r)
      merge(A,p,q,r)
```

Arranjo inicial

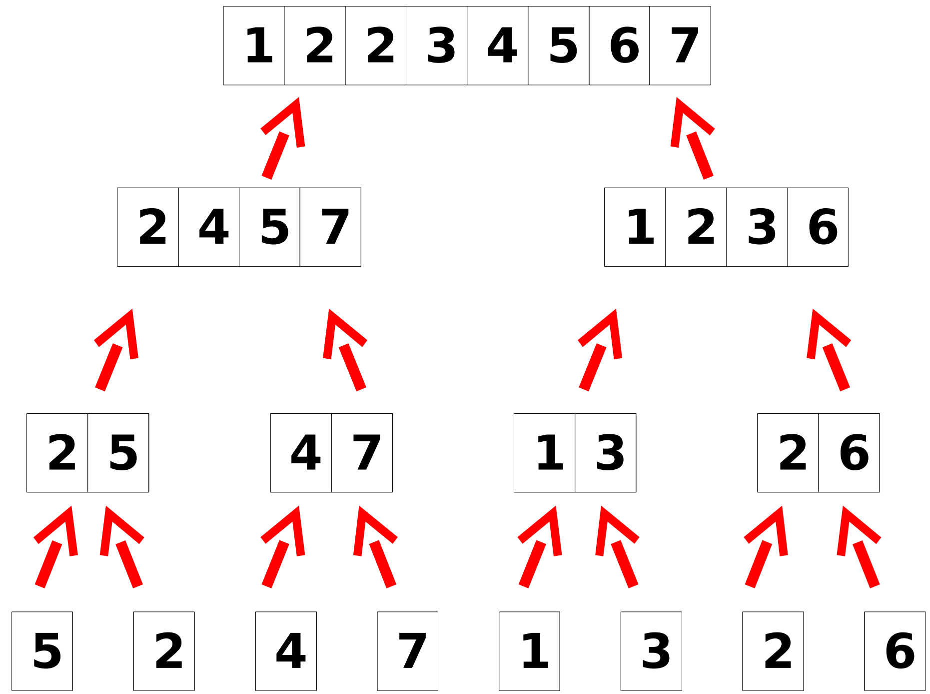
5	**2**	**4**	**7**	**1**	**3**	**2**	**6**

Divide até obter subarranjos com tamanho 1.
Então, começa a mesclar...

MergeSort

mergeSort (A,p,r)

```
se p < r
        q ← ⌊(p+r)/2⌋
        mergeSort(A, p, q)
        mergeSort(A, q+1,r)
        merge(A,p,q,r)
```



MergeSort

- Analisando a complexidade do MergeSort
- É um algoritmo recursivo.
- Portanto, devemos usar recorrência.
- Temos que definir T(n):
  - se ***n*** suficientemente pequeno (exemplo: $n \leq c$ para alguma constante *c*), a solução direta demorará um tempo constante, que consideraremos O(1)

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

MergeSort

- Analisando a complexidade do MergeSort
- É um algoritmo recursivo.
- Portanto, devemos usar recorrência.
- Temos que definir T(n):

```
mergeSort (A,p,r)
se p < r
 q ← ⌊(p+r)/2⌋
 mergeSort(A, p, q)
 mergeSort(A, q+1,r)
 merge(A,p,q,r)
```

  - se ***n*** suficientemente pequeno (exemplo: ***n*** ≤ ***c*** para alguma constante *c*), a solução direta demorará um tempo constante, que consideraremos O(1)
  - supomos que o problema seja dividido em ***a*** subproblemas, cada um com tamanho ***1/b*** do tamanho do problema original.

MergeSort

- Analisando a complexidade do MergeSort
- É um algoritmo recursivo.
- Portanto, devemos usar recorrência.
- Temos que definir T(n):
  - se ***n*** suficientemente pequeno (exemplo: $n \leq c$ para alguma constante *c*), a solução direta demorará um tempo constante, que consideraremos O(1)
  - supomos que o problema seja dividido em ***a*** subproblemas, cada um com tamanho ***1/b*** do tamanho do problema original.
  - no caso da ordenação por intercalação: a=b=???

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

MergeSort

- Analisando a complexidade do MergeSort
- É um algoritmo recursivo.
- Portanto, devemos usar recorrência.
- Temos que definir T(n):
  - se ***n*** suficientemente pequeno (exemplo: ***n*** ≤ ***c*** para alguma constante *c*), a solução direta demorará um tempo constante, que consideraremos O(1)
  - supomos que o problema seja dividido em ***a*** subproblemas, cada um com tamanho ***1/b*** do tamanho do problema original.
  - no caso da ordenação por intercalação: a=b=2

```
mergeSort (A,p,r)
se p < r
 q ← ⌊(p+r)/2⌋
 mergeSort(A, p, q)
 mergeSort(A, q+1,r)
 merge(A,p,q,r)
```

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo $\Rightarrow$ constante $\Rightarrow$ D(n) = O(1)
- Conquistar: resolvemos recursivamente dois subproblemas, cada um com tamanho n/2 $\Rightarrow$ 2T(n/2)
- Combinar: já vimos que o método ***merge*** *em um subarranjo com* ***n*** *elementos tem o tempo* O(n)

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n) = \begin{cases} O(1), n \leq c \\ aT(n/b) + D(n) + C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemo ... nas, cada um com tamanho
- Combinar: já vimos que o método ***merge*** *em u ... ubarranjo com n elementos tem o tempo* O(n)

D(n) + C(n) = O(1) + O(n) = O(n)

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

- Portanto:

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

- Quantas vezes teremos que dividir o problema?

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

- Portanto:

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

- Quantas vezes teremos que dividir o problema?
- Para facilitar, vamos supor que n é potência de 2

MergeSort

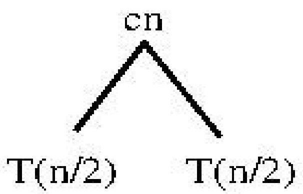
```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

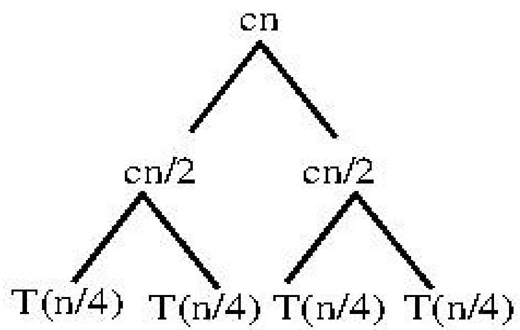
- Portanto:

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

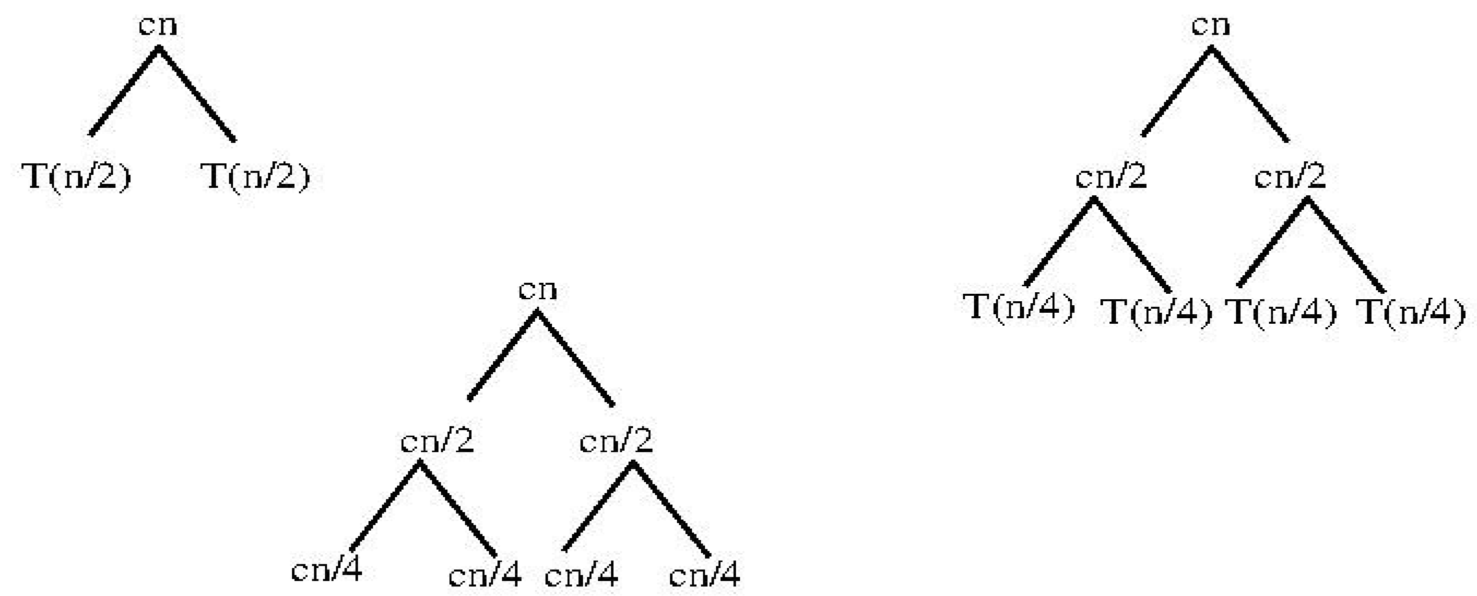
- Quantas vezes teremos que dividir o problema?
- Para facilitar, vamos supor que n é potência de 2
- T(n) pode ser representado da seguinte forma

Merge Sort - análise

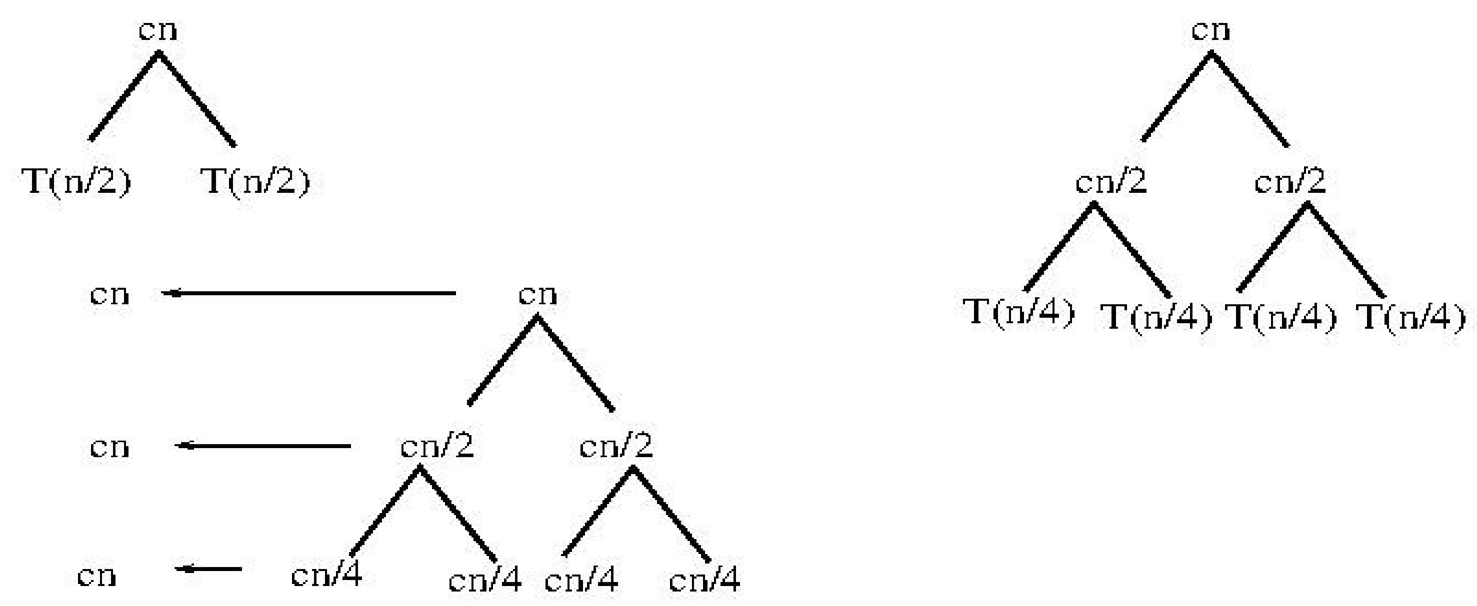




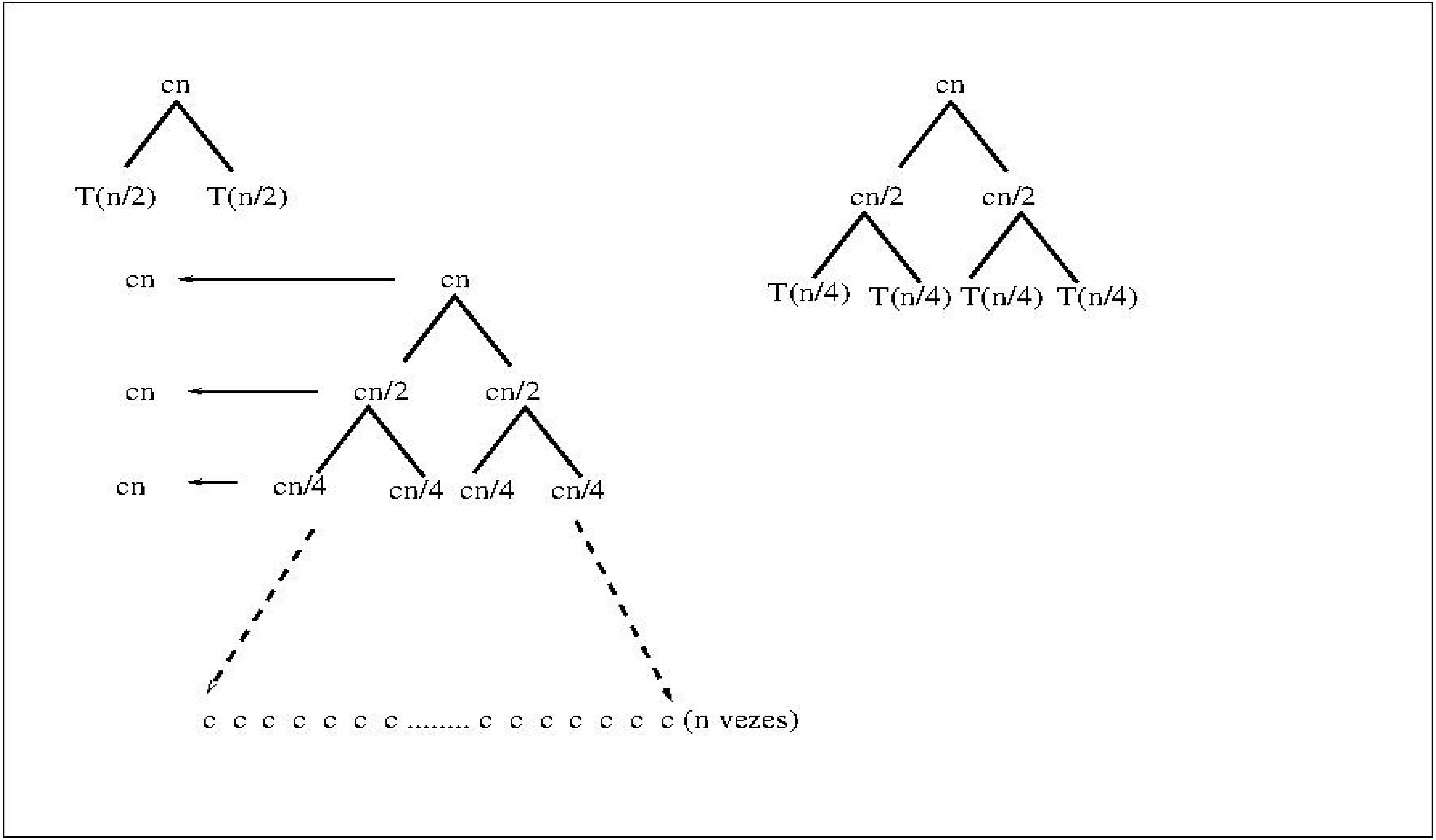
Merge Sort - análise



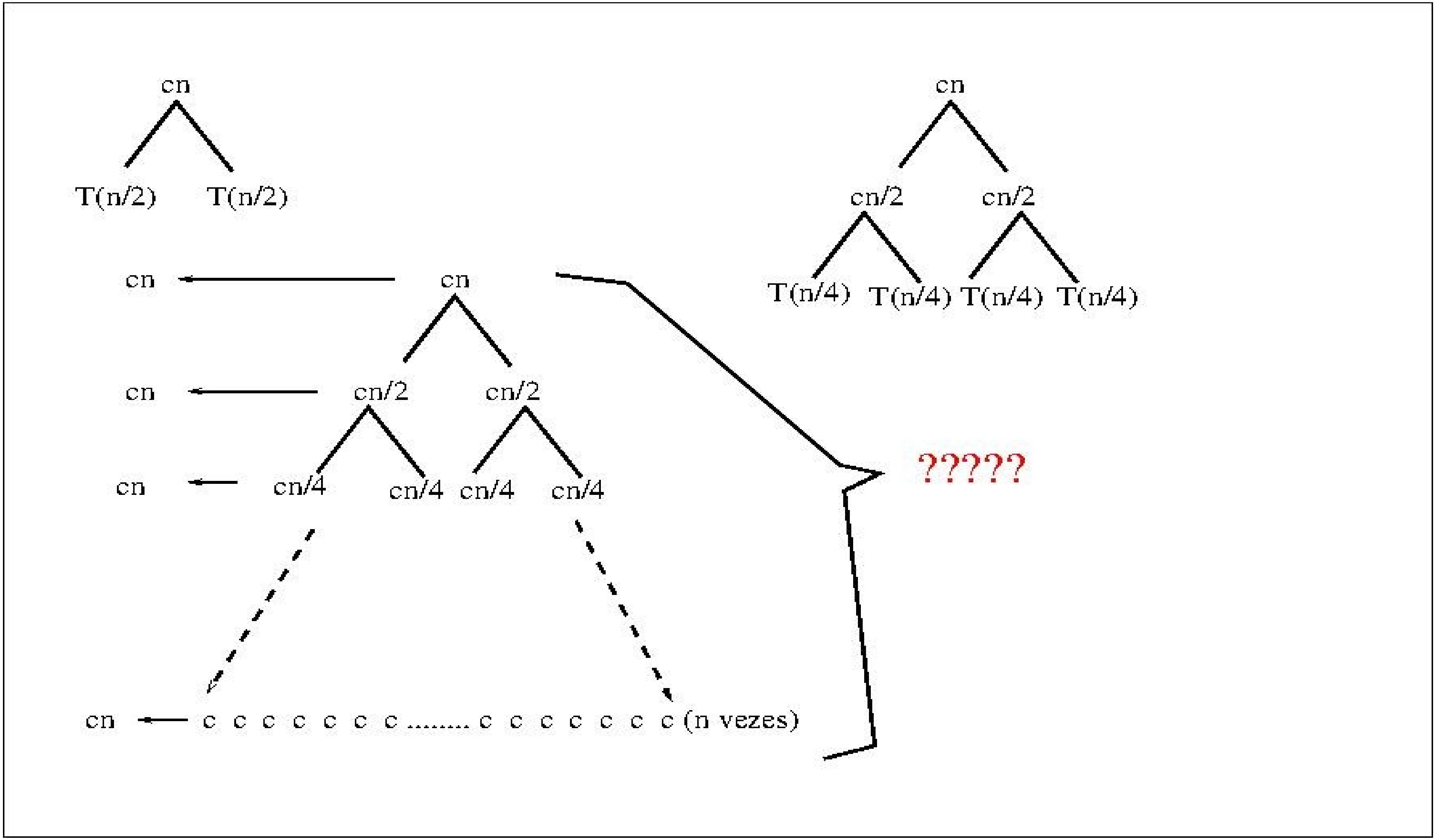
Merge Sort - análise



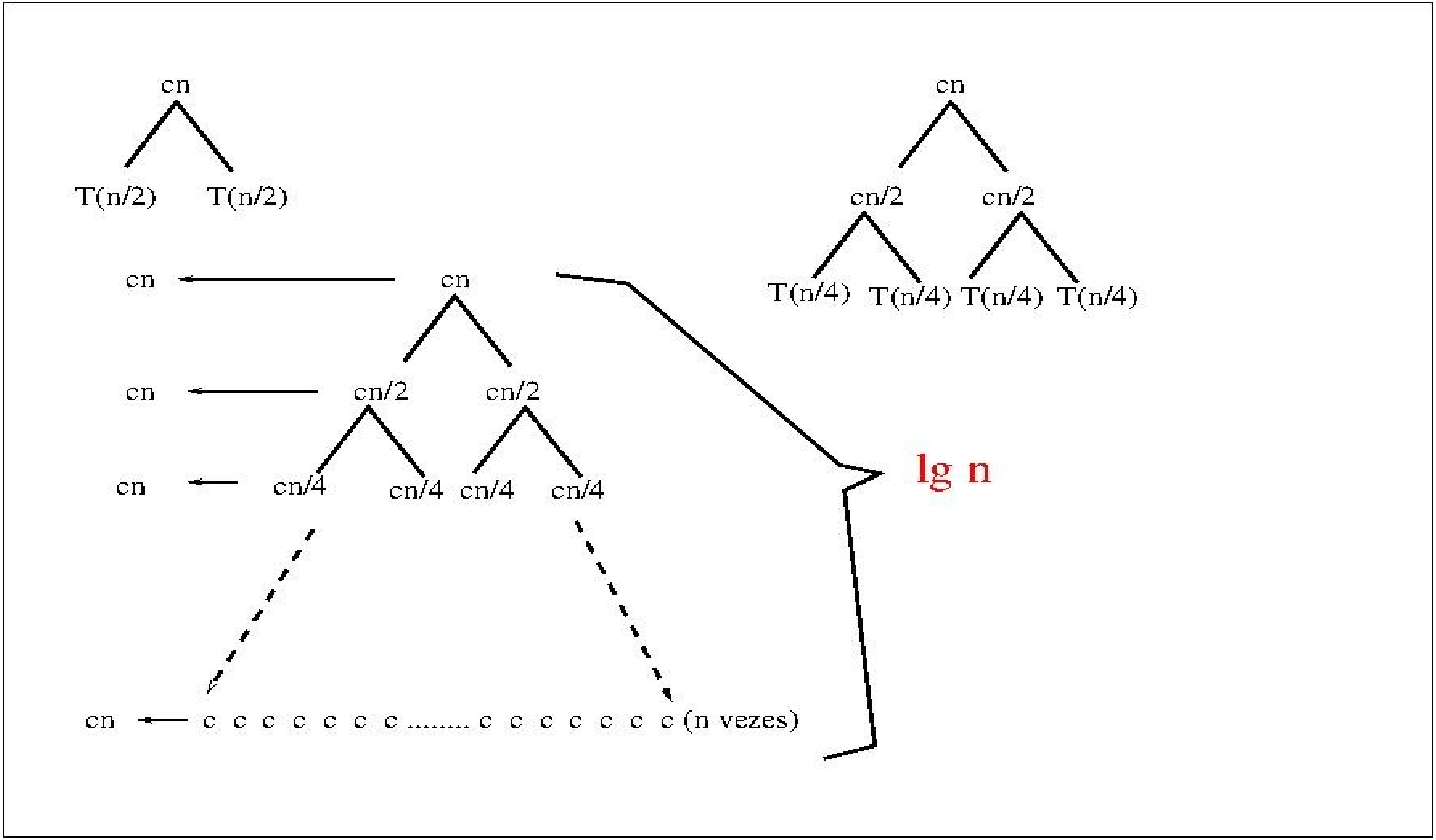
Merge Sort - análise



Merge Sort - análise



Merge Sort - análise



Merge Sort - análise

- Na verdaden temos lg n + 1 níveis
- Nivel 0 temos 2^0 nós
- Nivel 1 temos 2^1 nós
- Nivel i temos 2^i nós
- Somando todos os níveis, temos ….

Merge Sort - análise

- Na verdaden temos lg n + 1 níveis
- Nivel 0 temos 2^0 nós
- Nivel 1 temos 2^1 nós
- Nivel i temos 2^i nós
- Somando todos os níveis, temos $T(n) = cn(\lg n + 1) = cn \lg n + cn = \ldots$

Merge Sort - análise

- Na verdaden temos lg n + 1 níveis
- Nivel 0 temos 2^0 nós
- Nivel 1 temos 2^1 nós
- Nivel i temos 2^i nós
- Somando todos os níveis, temos T(n) = cn(lg n +1) = cn lg n + cn = O(n lg n)

Merge Sort em C

// A função recebe vetores crescentes v[p..q-1] e v[q..r-1] e rearranja v[p..r-1]
//em ordem crescente.

```
void merge(int v[], int p, int q, int r) {
   int i, j, k, *w;
   w = malloc((r-p) * sizeof (int));
   i = p; j = q;
   k = 0;

   while (i < q && j < r) {
      if (v[i] <= v[j])  w[k++] = v[i++];
      else  w[k++] = v[j++];
   }
   while (i < q)  w[k++] = v[i++];
   while (j < r)  w[k++] = v[j++];
   for (i = p; i < r; ++i)  v[i] = w[i-p];
   free (w);
}
```

Merge Sort em C

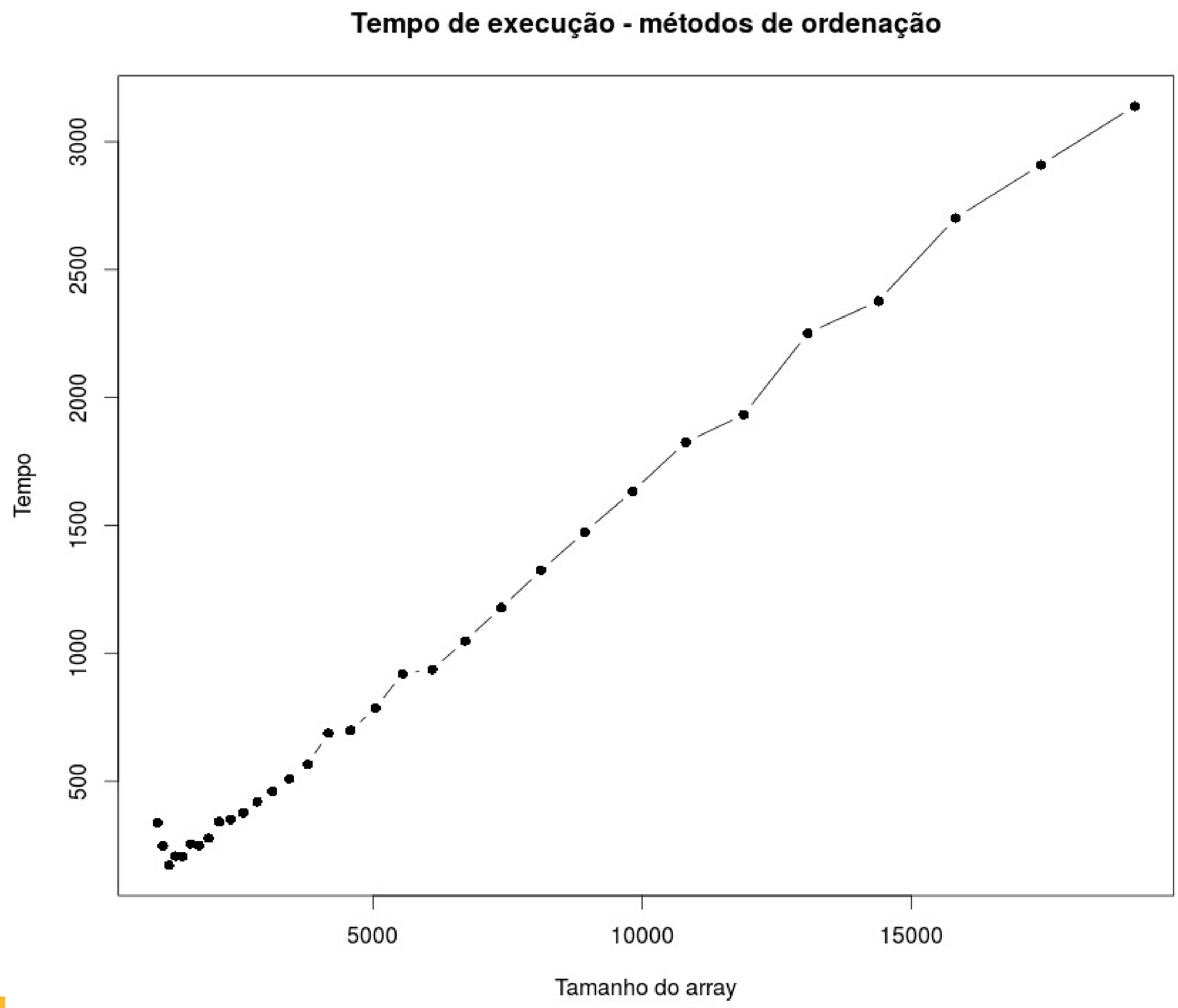
```
// A função mergesort rearranja o vetor v[p..r-1]
// em ordem crescente.

void mergesort (int v[], int p, int r)
{
   if (p < r-1) {
      int q = (p + r)/2;
      mergesort (v, p, q);
      mergesort (v, q, r);
      merge (v, p, q, r);
   }
}
```

Exercício

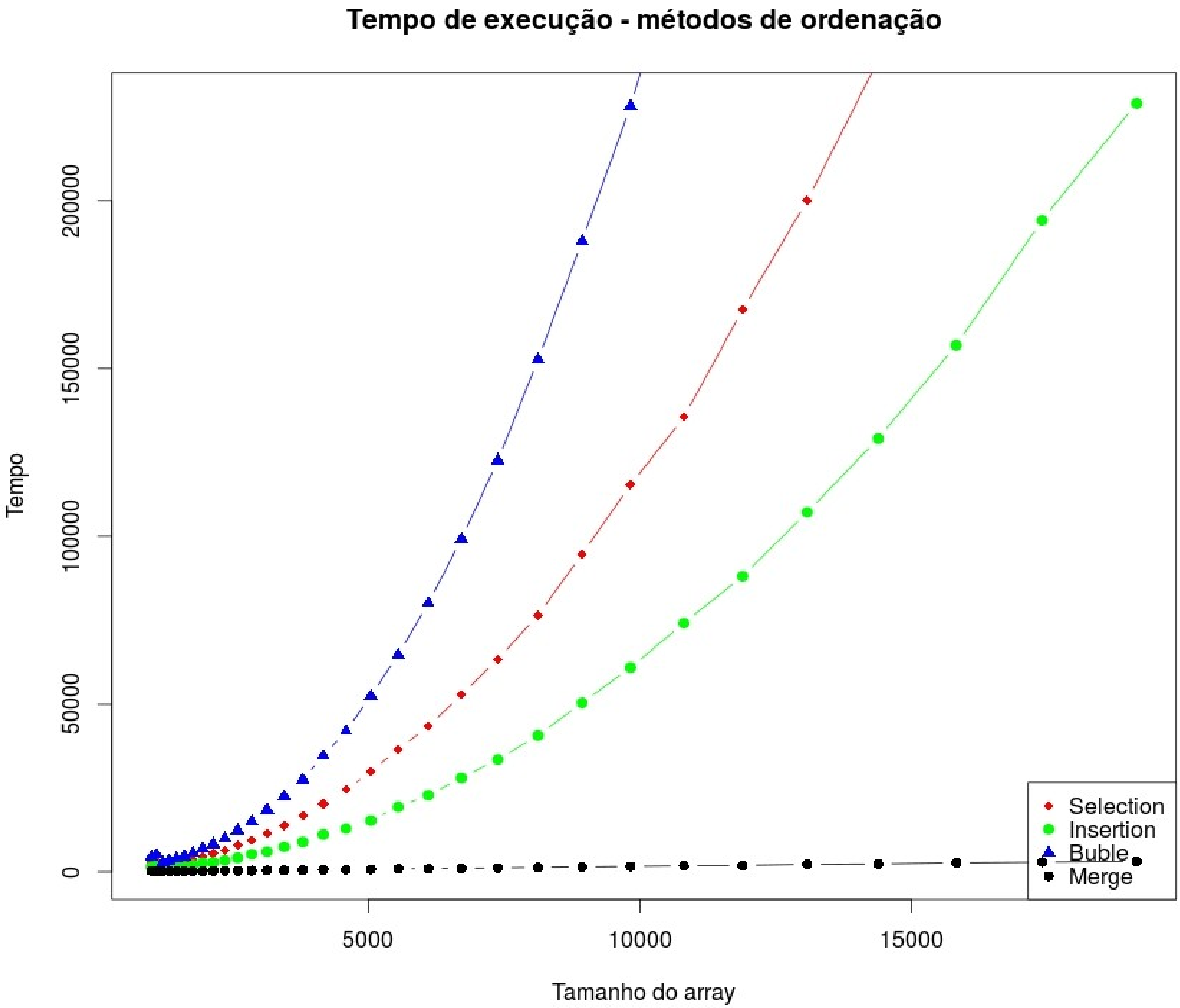
- Compare em um gráfico o tempo de execução do mergesort com os métodos vistos anteriormente.

Exercício



Tempo de execução - métodos de ordenação
Tempo
500
1000
1500
2000
2500
3000
5000
10000
15000
Tamanho do array

Exercício



Tempo de execução - métodos de ordenação
Tempo
0
50000
100000
150000
200000
5000
10000
15000
Tamanho do array
Selection
Insertion
Buble
Merge

Referências

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002.

Algoritmos de Ordenação

MergeSort

Professora:

Fátima L. S. Nunes