Revisão – Métodos de Ordenação

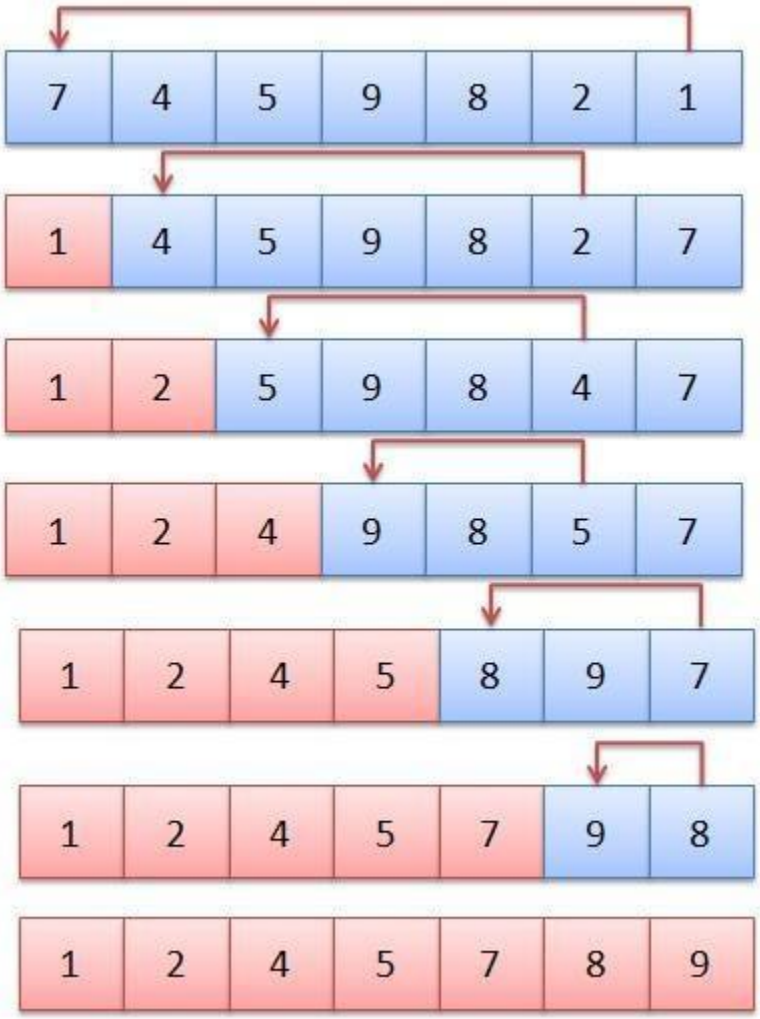
Seleção, Troca, Distribuição, Inserção e Intercalação

🎯 Objetivo da Revisão

- Relembrar os principais algoritmos de ordenação estudados
- Compreender quando cada método é indicado
- Comparar vantagens e desvantagens em tempo, espaço e aplicabilidade
- Revisar a lógica de implementação e identificar padrões comuns

📌 Ordenação por Seleção (Selection Sort)

- Ideia: selecionar o menor elemento e colocá-lo na posição correta
- Complexidade: $O(n^2)$
- Vantagens: simples, poucas trocas
- Desvantagens: ineficiente para grandes dados, não é estável
- Quando usar: quando simplicidade importa mais que desempenho

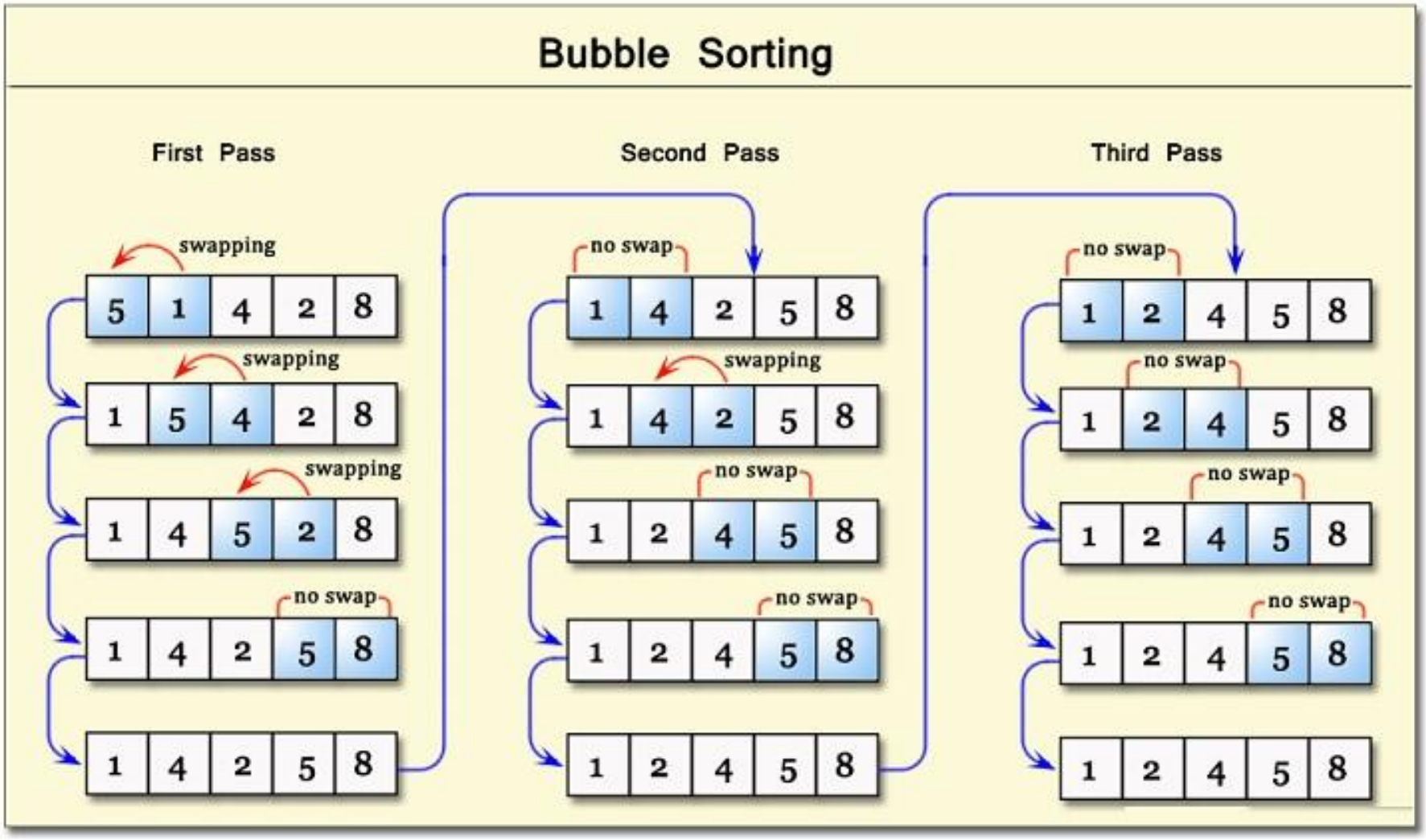

7 4 5 9 8 2 1
1 4 5 9 8 2 7
1 2 5 9 8 4 7
1 2 4 9 8 5 7
1 2 4 5 8 9 7
1 2 4 5 7 9 8
1 2 4 5 7 8 9

Implementação em C

```
#include <stdio.h>
#include <stdlib.h>


#define MAX 100


//---------------------------------------------
// SELECTION SORT (Ordenação por Seleção)
//---------------------------------------------
void selectionSort(int arr[], int n) {
   int i, j, min, temp;
   for (i = 0; i < n - 1; i++) {
      min = i; // assume que o menor está na posição i
      for (j = i + 1; j < n; j++) {
         if (arr[j] < arr[min]) {
            min = j; // atualiza o índice do menor valor
         }
      }
      // troca o menor valor encontrado com o valor na posição i
      temp = arr[i];
      arr[i] = arr[min];
      arr[min] = temp;
   }
}
```

📌 Ordenação por Troca (Bubble Sort)

- Ideia: percorre o vetor, trocando vizinhos fora de ordem
- Complexidade: $O(n^2)$
- Vantagens: simples, estável
- Desvantagens: muito ineficiente em grandes listas
- Quando usar: didático ou listas pequenas


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
swapping
swapping
no swap
no swap
no swap
no swap
no swap
no swap
no swap
no swap
5 1 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8

```
// BUBBLE SORT (Ordenação por Troca)
//-----------------------------------------
void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // troca se o elemento for maior que o próximo
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

📌 Ordenação por Distribuição

- **(Counting Sort / Radix Sort / Bucket Sort)**
- Ideia: classifica dados em baldes ou contagens
- Complexidade: O(n) em casos ideais
- Vantagens: muito eficiente em inteiros limitados
- Desvantagens: exige memória extra, ruim para dados genéricos
- Quando usar: inteiros em faixas conhecidas

Algoritmo básico

Couting Sort

// 1. encontra o maior elemento

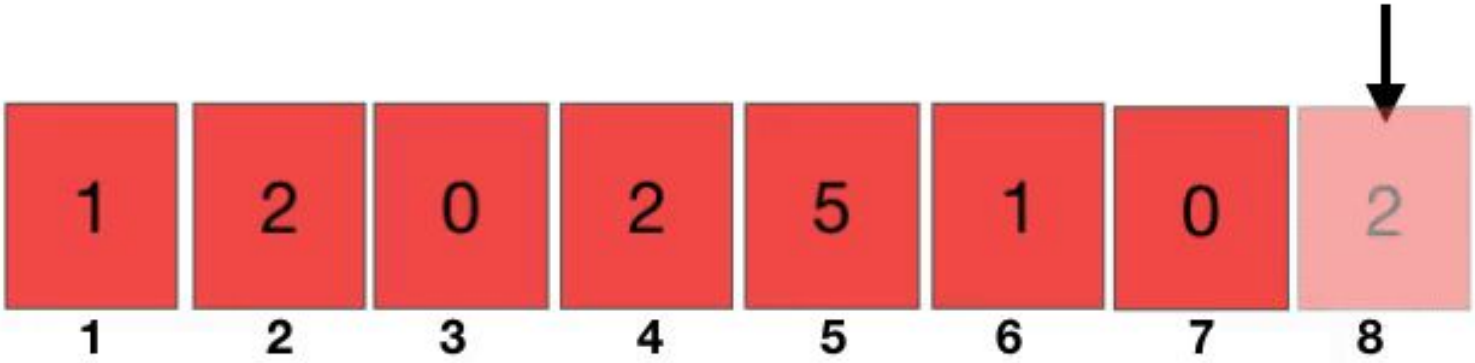
// 2. inicializa o vetor count com zeros

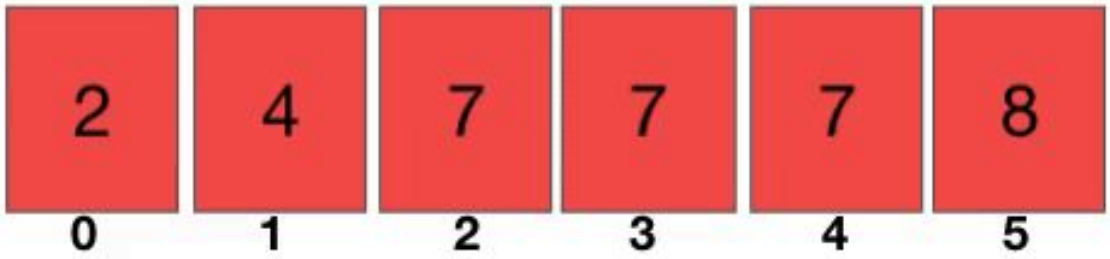
// 3. conta a frequência de cada elemento

// 4. calcula posições acumuladas

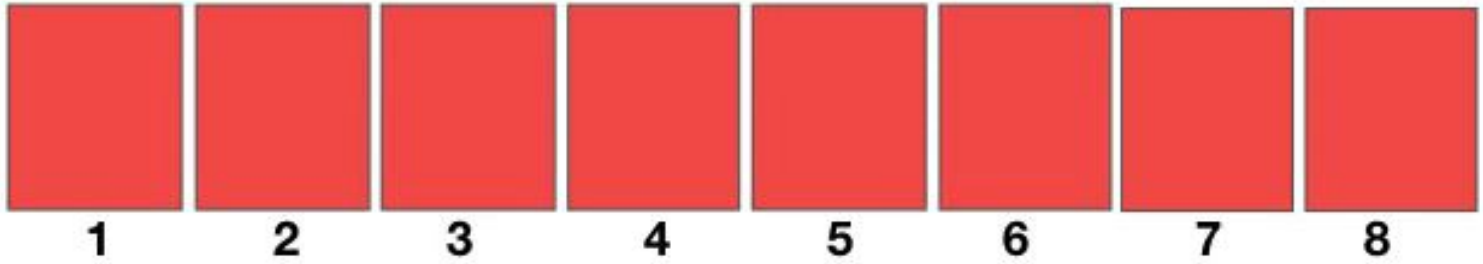
// 5. constrói o array de saída (em ordem estável)

// 6. copia o vetor ordenado de volta





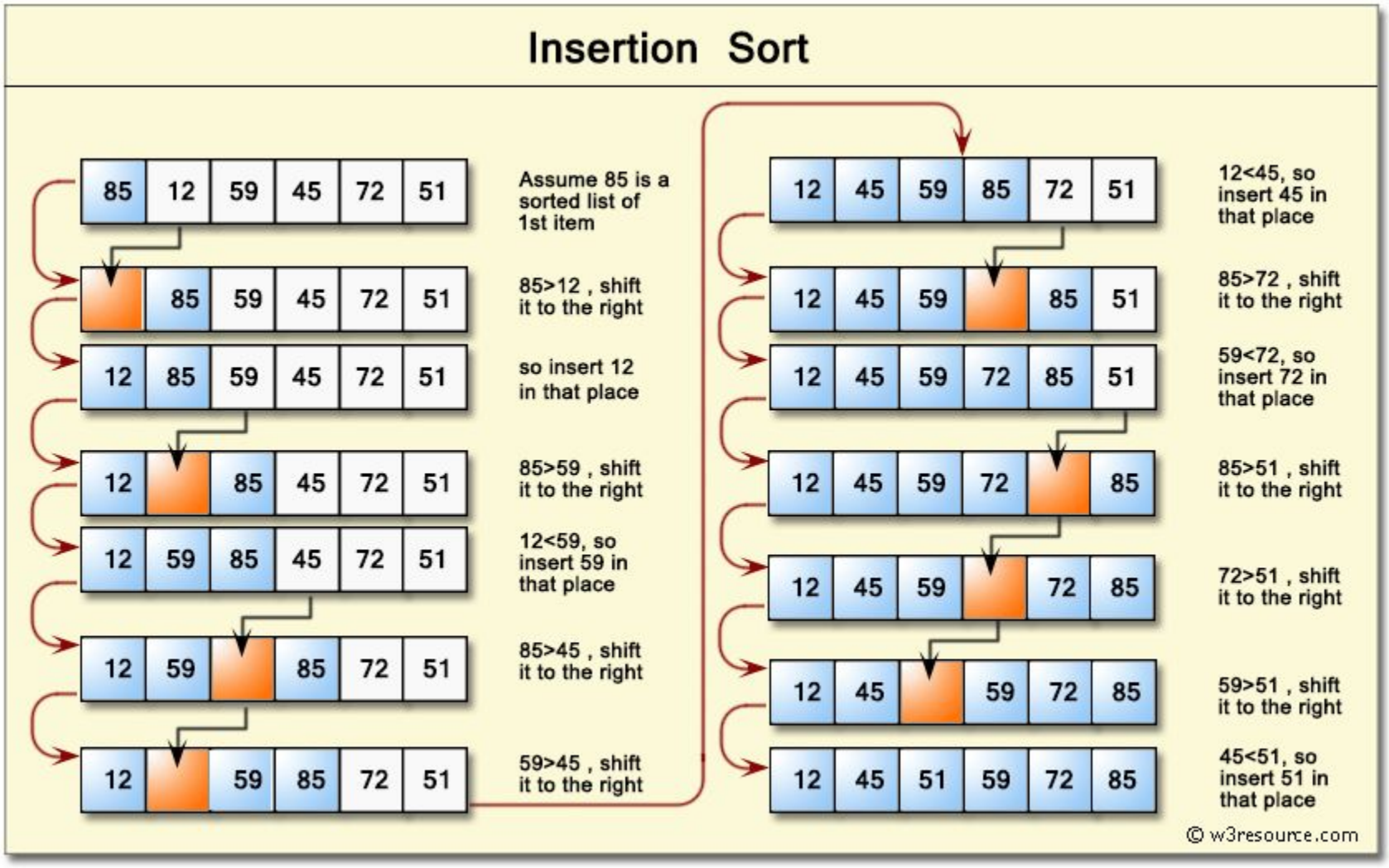
Modified temp



```
//-----------------------------------------------
// COUNTING SORT (Ordenação por Distribuição)
//-----------------------------------------------
void countingSort(int arr[], int n) {
   int i, max = arr[0];


   // encontra o maior elemento
   for (i = 1; i < n; i++) {
      if (arr[i] > max)
         max = arr[i];
   }


   int *count = (int*)calloc(max + 1, sizeof(int));
   int *output = (int*)malloc(n * sizeof(int));


   // conta a frequência de cada elemento
   for (i = 0; i < n; i++) {
      count[arr[i]]++;
   }


   // calcula posições acumuladas
   for (i = 1; i <= max; i++) {
      count[i] += count[i - 1];
   }


   // constrói o array de saída
   for (i = n - 1; i >= 0; i--) {
      output[count[arr[i]] - 1] = arr[i];
      count[arr[i]]--;
   }


   // copia para o array original
   for (i = 0; i < n; i++) {
      arr[i] = output[i];
   }


   free(count);
   free(output);
}
```

📌 Ordenação por Inserção (Insertion Sort)

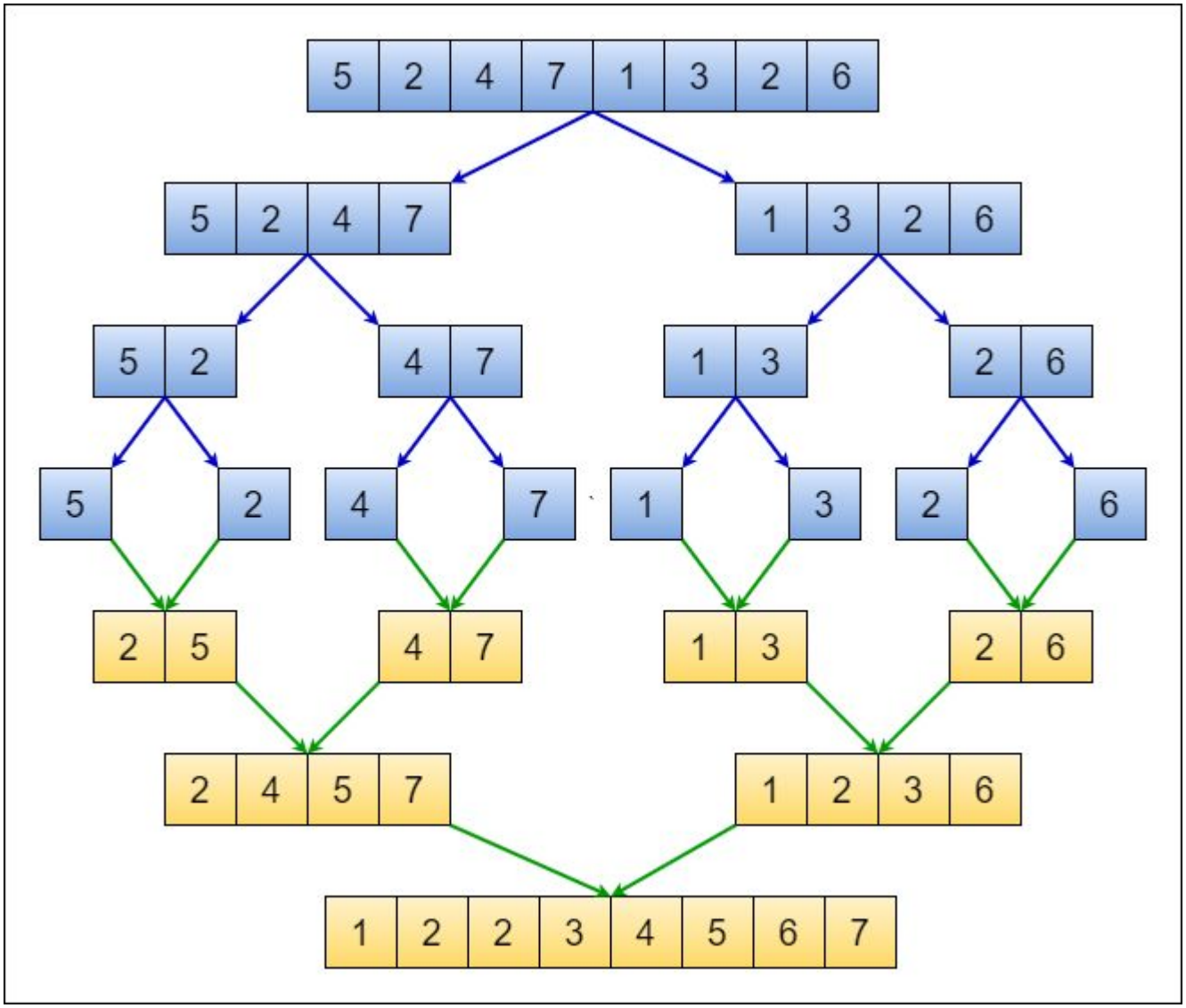
- Ideia: constrói lista ordenada inserindo elementos na posição correta
- Complexidade: $O(n^2)$ no pior caso, $O(n)$ no melhor
- Vantagens: eficiente para listas pequenas, estável
- Desvantagens: ineficiente em listas grandes
- Quando usar: listas pequenas ou quase ordenadas


Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

```
//-----------------------------------------
// INSERTION SORT (Ordenação por Inserção)
//-----------------------------------------
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i]; // valor atual a ser inserido
        j = i - 1;
        // move os elementos maiores que a chave uma posição à frente
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // insere a chave na posição correta
    }
}
```

📌 Ordenação por Intercalação (Merge Sort)

- Ideia: divide vetor, ordena e intercala subvetores
- Complexidade: O(n log n)
- Vantagens: ótimo desempenho, estável
- Desvantagens: requer memória extra
- Quando usar: grandes volumes de dados


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

```
//---------------------------------------------
// MERGE SORT (Ordenação por Intercalação)
//---------------------------------------------
void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;


   // cria arrays temporários
   int L[n1], R[n2];


   // copia dados para os arrays L[] e R[]
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];


   // intercala os arrays temporários de volta em arr[l..r]
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }


   // copia os elementos restantes de L[]
   while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }


   // copia os elementos restantes de R[]
   while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}


void mergeSort(int arr[], int l, int r) {
   if (l < r) {
      int m = l + (r - l) / 2; // ponto do meio
      mergeSort(arr, l, m);
```

Resumo Comparativo

- Seleção: $O(n^2)$, não estável, sem memória extra, simples
- Troca: $O(n^2)$, estável, sem memória extra, didático
- Distribuição: $O(n)$, estável, exige memória extra, inteiros limitados
- Inserção: $O(n^2)$, estável, sem memória extra, listas pequenas
- Intercalação: $O(n \log n)$, estável, exige memória extra, grandes volumes

 Sugestão de Estudo

- 1. Revisar ideia central de cada algoritmo
- 2. Implementar pelo menos uma vez em C
- 3. Comparar tempos de execução com vetores grandes
- 4. Refletir sobre quando escolher cada algoritmo

```
//---------------------------------------------
// FUNÇÃO PARA IMPRIMIR ARRAY
//---------------------------------------------
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}


//---------------------------------------------
// MAIN PARA TESTAR OS ALGORITMOS
//---------------------------------------------
int main() {
   int arr[MAX], n, choice;


   printf("Digite o tamanho do vetor (max %d): ", MAX);
   scanf("%d", &n);


   printf("Digite os elementos do vetor:\n");
   for (int i = 0; i < n; i++)
      scanf("%d", &arr[i]);


   printf("\nEscolha o algoritmo de ordenacao:\n");
   printf("1 - Selection Sort\n");
   printf("2 - Bubble Sort\n");
   printf("3 - Insertion Sort\n");
   printf("4 - Counting Sort\n");
   printf("5 - Merge Sort\n");
   scanf("%d", &choice);


   switch (choice) {
      case 1: selectionSort(arr, n); break;
      case 2: bubbleSort(arr, n); break;
      case 3: insertionSort(arr, n); break;
      case 4: countingSort(arr, n); break;
      case 5: mergeSort(arr, 0, n - 1); break;
      default: printf("Opcao invalida!\n"); return 0;
   }
```