**INSTITUTO FEDERAL DE SÃO PAULO**

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FERNANDA MARTINS DA SILVA

# *Bucket Sort* e *Counting Sort*

Listex 11 - Exercício 03 - Projeto e Análise de Algoritmos

**São João da Boa Vista/SP**

### *Bucket Sort*

O *Bucket Sort* é um algoritmo de ordenação que divide os elementos de um array em vários grupos chamados de "baldes" (*bucket* do inglês), e ordena cada balde individualmente, seja usando outro algoritmo de ordenação ou recursivamente.

**Funcionamento**

Os elementos do array original são distribuídos em vários baldes com base em uma função de distribuição, que geralmente mapeia os valores do array para o intervalo [0, 1) ou [0, n), onde n é o número de baldes. Cada balde é ordenado individualmente, podendo ser feito usando outro algoritmo de ordenação, como *Insertion Sort*, ou aplicando o *Bucket Sort* recursivamente. Depois que cada balde é ordenado, são concatenados na ordem correta para formar o *array* ordenado.

**Implementação**

```
void bucketSort(float arr[], int n){

   //Cria um array de buckets, onde cada bucket é uma lista
dinâmica
   int i, j, k;
   int bucketCount = 10; // Número exemplo de buckets
   float buckets[bucketCount][n];
   int bucketSizes[bucketCount];

   //Inicializa os tamanhos dos buckets para zero
   for(i = 0; i < bucketCount; i++){
       bucketSizes[i] = 0;
   }

   //Distribui os elementos do array original em buckets
   for(i = 0; i < n; i++){
       int bucketIndex = (int)(bucketCount * arr[i]);
       buckets[bucketIndex][bucketSizes[bucketIndex]] = arr[i];
       bucketSizes[bucketIndex]++;
   }

   //Ordena cada bucket individualmente usando Insertion Sort
   for(i = 0; i < bucketCount; i++){
```

```
        insertionSort(buckets[i], bucketSizes[i]);
    }

    //Concatena todos os buckets ordenados de volta ao array
original
    k = 0;
    for(i = 0; i < bucketCount; i++){
        for(j = 0; j < bucketSizes[i]; j++){
            arr[k++] = buckets[i][j];
        }
    }
}
```

**Complexidade**

O *Bucket Sort* pode ser muito eficiente quando os dados são distribuídos uniformemente, com a complexidade tendendo para O(n). No entanto, se os dados não forem bem distribuídos, a eficiência do algoritmo pode se degradar para $O(n^2)$, tornando o *Bucket Sort* altamente dependente da distribuição dos dados e da escolha do número de *buckets*.

## *Counting Sort*

*Counting Sort* é um algoritmo de ordenação eficiente para certos tipos de dados, especialmente quando os valores a serem ordenados estão em um intervalo limitado e conhecido. Ele funciona contando o número de ocorrências de cada valor distinto no *array* de entrada e, em seguida, utilizando essa contagem posiciona cada valor corretamente no *array* de saída.

**Funcionamento**

Primeiro, ele encontra o valor máximo e mínimo no *array* de entrada para determinar o intervalo de valores, depois, um *array* de contagem (frequências) que terá um índice para cada valor possível no intervalo do *array* de entrada. O mesmo percorre o array de entrada e, para cada valor, incremente o contador correspondente no array de contagem. Depois, modifica o *array* de contagem para que cada valor seja somado ao valor do índice anterior, transformando-o em um *array* de prefixos somados. Finalizando, percorre o *array* de entrada

de trás para frente e coloca cada valor na posição correta no array de saída, copiando os valores do array de saída de volta para o array original.

**Implementação**

```
void countingSort(int arr[], int n){

   int i;
   int max = arr[0];
   int min = arr[0];

   //Encontra o valor máximo e mínimo no array
   for(i = 1; i < n; i++){
       if(arr[i] > max){
           max = arr[i];
       }else if(arr[i] < min){
           min = arr[i];
       }
   }

   int range = max - min + 1;
   int count[range];
   int output[n];

   //Inicializa o array de contagem com zeros
   for(i = 0; i < range; i++){
       count[i] = 0;
   }

   //Faz a contagem de frequências dos elementos
   for(i = 0; i < n; i++){
       count[arr[i] - min]++;
   }

   //Acumule as contagens
   for(i = 1; i < range; i++){
       count[i] += count[i - 1];
   }

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

```
        count[arr[i] - min]--;
    }

    //Copia os valores ordenados de volta ao array original
    for(i = 0; i < n; i++){
        arr[i] = output[i];
    }
}
```

**Complexidade**

*Counting Sort* é frequentemente usado como uma sub-rotina em algoritmos mais complexos, como o Radix Sort, e é útil em situações onde os dados estão em um intervalo limitado. O algoritmo requer espaço adicional para o *array* de contagem e o *array* de saída, e tem complexidade O(n).