BUCKET SORT

Luciano Brum

Eugênio Pierazzoli

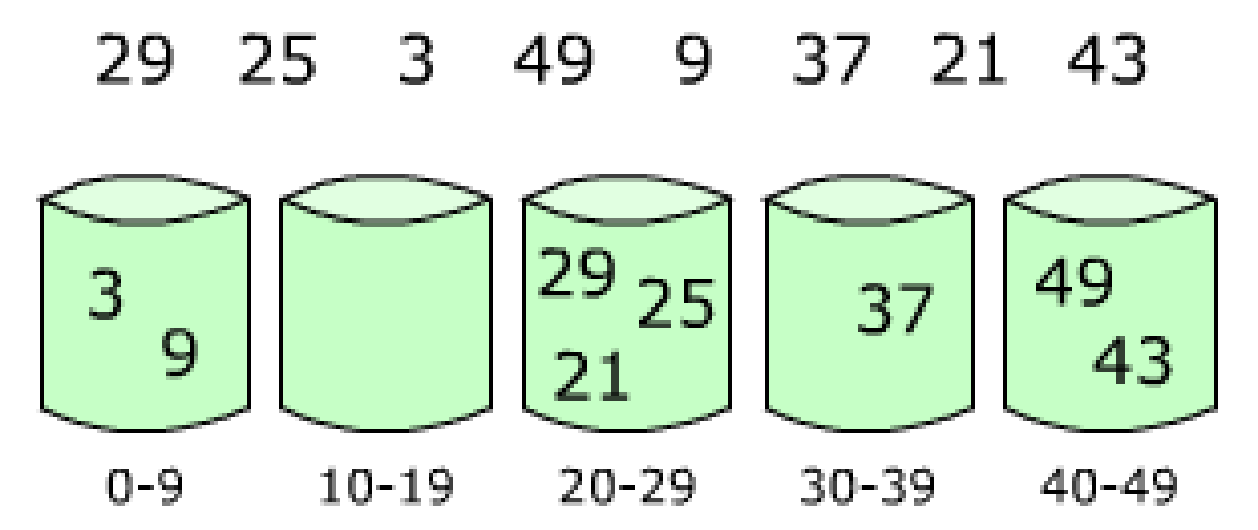
INTRODUÇÃO

- Bucket Sort é um algoritmo para ordenação de vetores em que não é baseado em comparações e tem complexidade linear.
- (Lembrando que para ordenação de vetores baseados em comparações, o pior caso é O(n log n).)

PRINCÍPIO DE FUNCIONAMENTO

- O algoritmo recebe um vetor inicial, e aloca os seus elementos em ‘k’ baldes (vetores).
- Um número de baldes e faixas de valores são atribuídas a cada balde. O tamanho dos baldes pode ser fixo ou variável caso se conheça a distribuição de dados.
- Se o elemento pertence a essa faixa, ele é alocado ao balde.

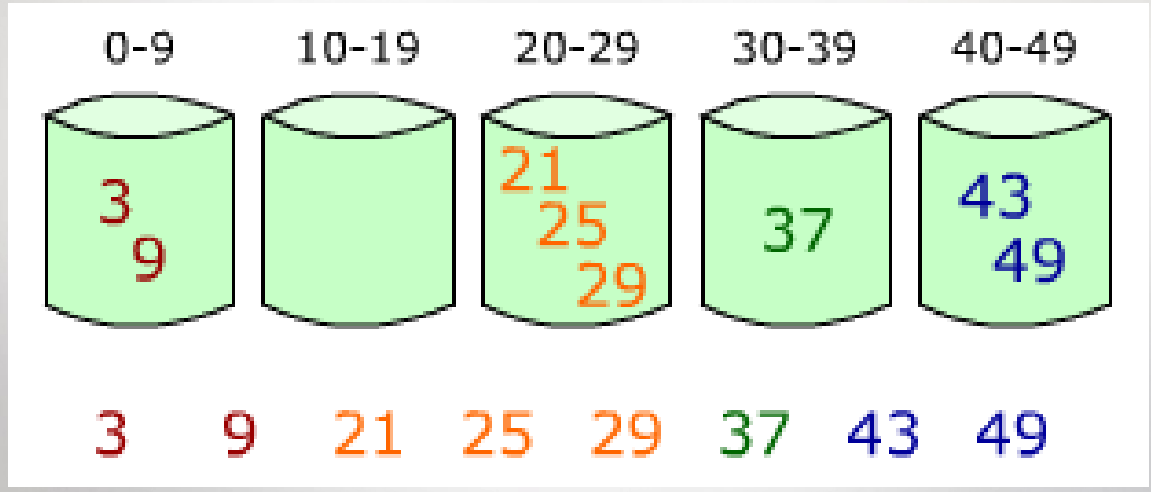
INSERÇÃO NOS BALDES



ORDENAÇÃO INTERNA

- A partir da alocação nos baldes, podemos utilizar algum algoritmo qualquer de ordenação para ordenar os elementos de cada balde, ou o próprio bucket sort recursivamente.

RETORNO DOS BALDES



RETORNO DOS DADOS

- Após a ordenação, basta percorrermos os baldes em ordem listando os elementos contidos em cada um e, de fato, teremos o vetor inicial ordenado.

VETOR FINAL

IMPLEMENTAÇÃO

- Serão demonstradas duas implementações distintas do Bucket Sort em linguagem C.

A explicação do código será feita concorrentemente com sua apresentação.

IMPLEMENTAÇÃO 1

```
#define tam_bucket 100 //tam_bucket é o tamanho de cada balde da estrutura bucket.
#define num_bucket 10 //num_bucket é o número de baldes, isto é, o tamanho do vetor de bucket
#define max 10 //max é o tamanho do vetor a ser ordenado.

 typedef struct {
         int topo; //conta o número de elementos no balde
         int balde[tam_bucket]; //vetor de elementos dentro do balde (tam_bucket é o tamanho)
 }bucket;

 void bubble(int v[],int tam){  //Algoritmo qualquer para ordenação individual de cada balde//
         int i,j,temp;
                 for(j=0;j<tam-1;j++){
                         for(i=0;i<tam-1;i++){
                                 if(v[i+1]<v[i]){
                                         temp=v[i];
                                         v[i]=v[i+1];
                                         v[i+1]=temp;
                                 }
                         }
                 }
 }//*****************************************************************************************//
```

IMPLEMENTAÇÃO 1

```
void bucketsort(int v[],int tam){          //Aqui começa o BucketSort...
        bucket b[num_bucket];              //Declarada uma estrutura com n baldes(num_bucket define o nº de baldes...
        int i,j,k;
/* 1 */ for(i=0;i<num_bucket;i++)          //inicializa os "topos"
                b[i].topo=0;
```

IMPLEMENTAÇÃO 1

```
/* 2 */ for(i=0;i<tam;i++){//verifica em que balde o elemento deve ficar
                j=(num_bucket)-1;//j <= número de baldes - 1
                while(1){//Loop que não para até encontrar um break;
                        if(j<0)//para j < 0, sai do while;
                                break;
                        if(v[i]>=j*10){//se o elemento do vetor é maior que j*10 faça...do contrário, j <= j-1...
                                b[j].balde[b[j].topo]=v[i]; //balde J[topo] <= v[i], que é a posição mais alta...
                                (b[j].topo)++;//incrementa o topo...
                                break;//cai fora do loop WHILE e repete tudo de novo para i <= i + 1...
                        }
                        j--;//se o elemento v[i] for menor que j*10, j <= j-1 e repete o while(1)...
                }
        }
```

IMPLEMENTAÇÃO 1

```
/* 3 */ for(i=0;i<num_bucket;i++)//ordena os baldes utilizando BubbleSort;
                if(b[i].topo)
                        bubble(b[i].balde,b[i].topo);

        i=0;
/* 4 */ for(j=0;j<num_bucket;j++){//põe os elementos dos baldes de volta no vetor;
                for(k=0;k<b[j].topo;k++){
                        v[i]=b[j].balde[k];
                        i++;
                }
        }
}
```

IMPLEMENTAÇÃO 2

- A implementação 2 possui apenas uma mudança:

O algoritmo utilizado para ordenar cada balde individualmente é o **Merge Sort**.

IMPLEMENTAÇÃO 2

```
/* 3 */ for(i=0;i<num_bucket;i++)//ordena os baldes utilizando mergesort;
                if(b[i].topo)
                        mergesort(b[i].balde,0,b[i].topo);
```

IMPLEMENTAÇÃO 2

```
void MergeSort(int A[], int ini, int fim){

     if(ini<fim){
          int meio=(fim+ini)/2;
          MergeSort(A,ini,meio);
          MergeSort(A,meio+1,fim);
          Merge(A,ini,meio,fim);
     }


}
```

IMPLEMENTAÇÃO 2

```
void Merge(int A[], int ini, int meio, int fim){
    int tamanhodireita=fim-meio;
    int tamanhoesquerda=meio-ini+1;
    int dir[tamanhodireita];
    int esq[tamanhoesquerda];
    int i, j;
    for(i=0;i<tamanhoesquerda;i++){
        esq[i]=A[ini+i];}
    for(j=0;j<tamanhodireita;j++){
        dir[j]=A[meio+1+j];}
    int origem=ini;
    int e=0;
    int d=0;
    while((e<tamanhoesquerda)&&(d<tamanhodireita)){
        if(dir[d]<esq[e]){
            A[origem]=dir[d];
            d++;}
        else{
            A[origem]=esq[e];
            e++;}
        origem++;}
    for(i=d;i<tamanhodireita;i++){
    A[origem]=dir[i];
    origem++;}
    for(i=e;i<tamanhoesquerda;i++){
        A[origem]=esq[i];
        origem++;}
}
```

IMPLEMENTAÇÃO

- O algoritmo pressupõe algo sobre os dados elementos no vetor. (que eles estão distribuídos uniformemente.). Portanto, este não é um algoritmo de ordenação genérico, pois além de não ser baseado em comparações, não é eficiente em muitos casos.

IMPLEMENTAÇÃO

- Problema 1: E se um balde encher?
- R: Isso nunca vai acontecer devido ao fato de os dados serem uniformemente distribuídos nos baldes, mas no pior caso, ainda mesmo que raro, pode acontecer de todos os dados caírem em um só balde. Para evitar encher o balde, cada balde tem o tamanho do vetor, para caso caia no pior caso.

IMPLEMENTAÇÃO

- Problema 2: O algoritmo apresentado funciona para números negativos no vetor?

- R: Não! Um tratamento adicional é necessário para alocar números negativos nos baldes.

IMPLEMENTAÇÃO 3

```
/* 2 */ for(i=0;i<tam;i++){
                    j=num_bucket/2;
                    j2=num_bucket;
                    while(1){
                            if(j2<0){
                                break;
                            }
                            if(j2<(num_bucket/2)){
                                if(v[i]>=j*10){
                                b[j2].balde[b[j2].topo]=v[i];
                               (b[j2].topo)++;
                                break;
                                }
                            }
                            if((v[i]>=j*10)&&(j2>=num_bucket/2)){
                                    b[j2].balde[b[j2].topo]=v[i];
                                    (b[j2].topo)++;
                                    break;
                            }
                            j2--;
                            j--;
                    }
            }
```

IMPLEMENTAÇÃO 3

```
                    if((v[i]>=j*10)&&(j2>=num_bucket/2)){
                            b[j2].balde[b[j2].topo]=v[i];
                            (b[j2].topo)++;
                            break;
                    }
                    j2--;
                    j--;
            }
}
```

IMPLEMENTAÇÃO 3

- J recebe metade do tamanho dos baldes, e J2 recebe o tamanho dos baldes. J decrementa junto com J2.
- J2 indica o balde a ser colocado o elemento. Note que elementos positivos só serão inseridos até a metade superior da lista de baldes. Quando J2 for menor que metade do tamanho dos baldes, começa o tratamento para números negativos.

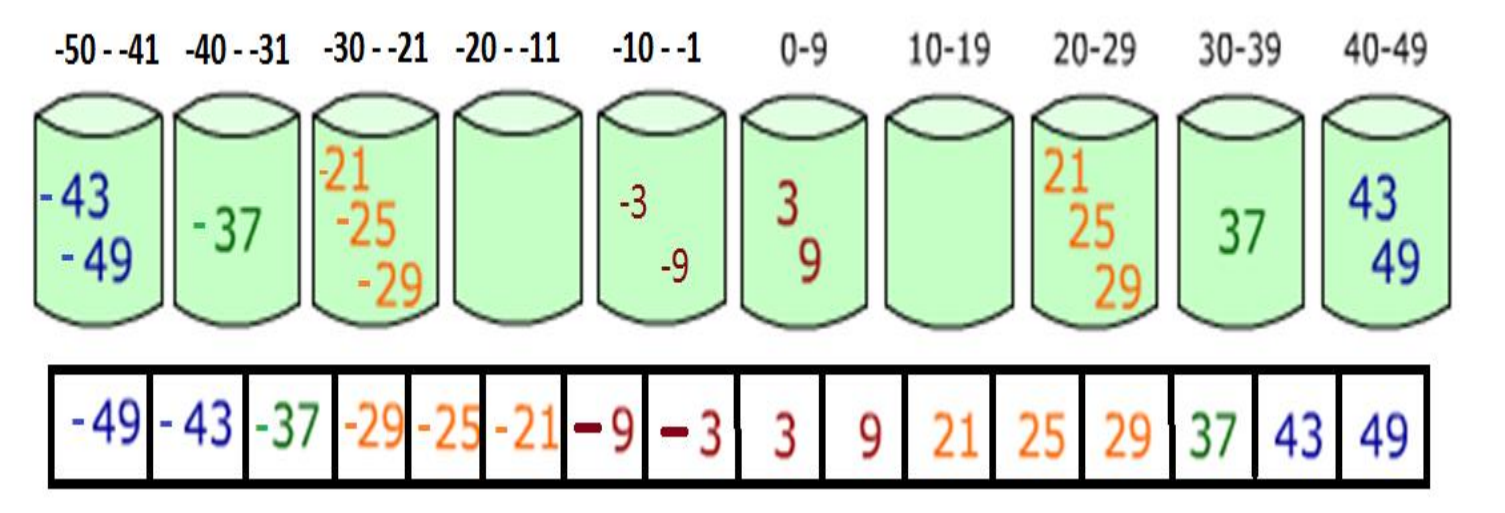
IMPLEMENTAÇÃO 3

```
j=num_bucket/2;
j2=num_bucket;
while(1){
        if(j2<0){
            break;
        }
        if(j2<(num_bucket/2)){
            if(v[i]>=j*10){
            b[j2].balde[b[j2].topo]=v[i];
           (b[j2].topo)++;
            break;
            }
        }
```

IMPLEMENTAÇÃO 3

- A variável J tanto no caso dos números positivos como nos negativos, serve para definir o intervalo a qual o valor pertence, e atribui-lo ao balde correspondente. Exemplificando, veja a figura abaixo:

IMPLEMENTAÇÃO 3



IMPLEMENTAÇÃO 3

- Resumindo:
- J define o intervalo o qual o elemento pertence.
- J2 define o balde associado a este intervalo o qual será inserido o elemento.

DETALHES DA IMPLEMENTAÇÃO

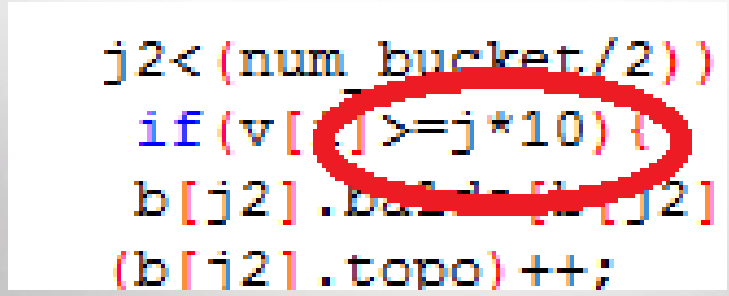
- Relação entre as variáveis tam_bucket, num_bucket e J:
- Exemplo: Para dados entre 0 – 1.000.000 serem distribuídos uniformemente nos baldes, devemos definir o número de baldes em tam_bucket, e se definirmos como 10, J deverá ser ajustado.
- 10 baldes induz que devemos ter 10 intervalos iguais, ou seja, 0 – 100.000, 100.000 – 200.000,...,900.000 – 1.000.000.(J*100.000)

DETALHES DA IMPLEMENTAÇÃO

- 100 baldes induz que devemos ter 100 intervalos iguais, ou seja, 0 – 10.000, 10.000 – 20.000,...,990.000 – 1.000.000. (J*10.000)
- 1000 baldes induz que devemos ter 1000 intervalos iguais, ou seja, 0 – 1.000, 1.000 – 2.000,...,999.000 – 1.000.000.(J*1.000)
- 10000 baldes induz que devemos ter 10000 intervalos iguais, ou seja, 0 – 100, 100 – 200,...,999.900 – 1.000.000.(J*100)

DETALHES DA IMPLEMENTAÇÃO

- Resumindo: Para uma entrada fixa, quanto maior o número de baldes, menor será o intervalo destes baldes. O intervalo é definido justamente aqui:



DETALHES DA IMPLEMENTAÇÃO

- A seguir será apresentada uma animação do Bucket Sort para diversos casos apresentados aqui. Posteriormente um manual de uso será demonstrado.

ANIMAÇÃO

1º - São criados os Baldes no Intervalo do vetor.

Balde 0-7	Balde 8-15	Balde 16-23

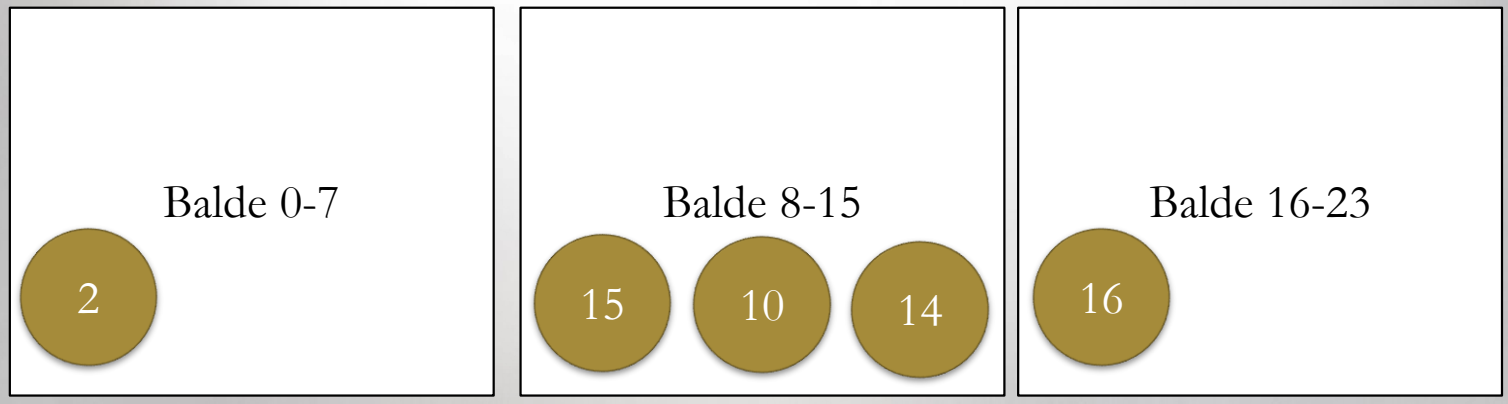
ANIMAÇÃO

2º - Os Valores são inseridos no Balde correspondente.

Balde 0-7	Balde 8-15	Balde 16-23

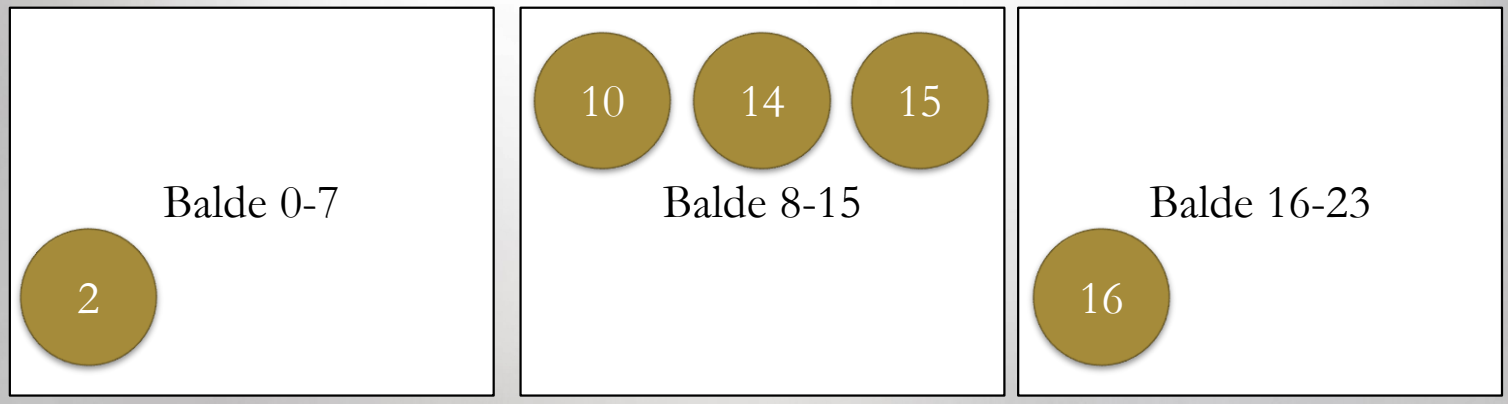
ANIMAÇÃO

3º - Os Baldes não vazios são ordenados usando o bucket sorte ou outro método desejável.



ANIMAÇÃO

4º - Os dados ordenados dos baldes retornam ao vetor.

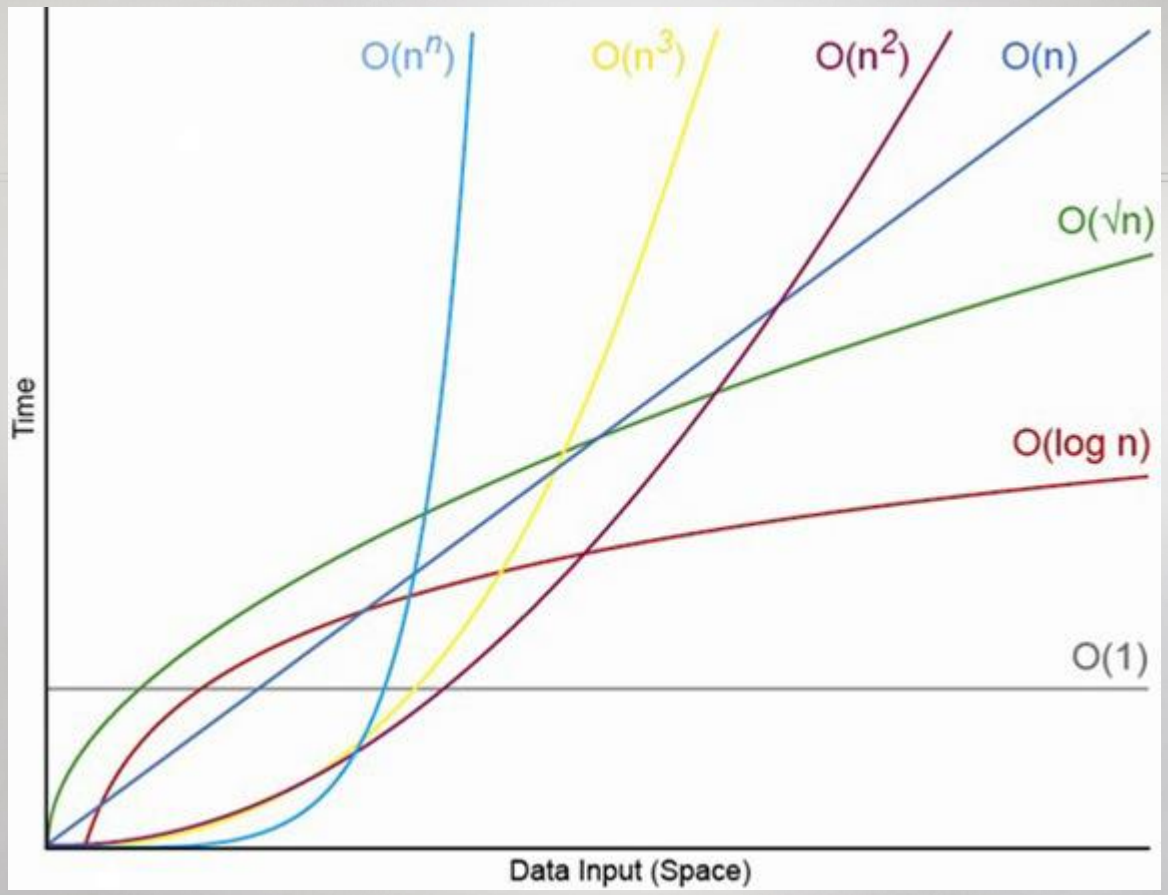


A MELHOR OTIMIZAÇÃO

- Uma forma comum de otimizar é retornar os valores dos baldes ao vetor sem ordenar e executar o *insertion sort* sobre o vetor inteiro, porque baseado em o quão próximo estão os elementos de sua posição final, o numero de comparações é relativamente pequeno e a memoria é melhor explorada.

COMPLEXIDADE

- Qual a complexidade do algoritmo BucketSort?
- Qual a complexidade do melhor caso?
- Qual a complexidade do pior caso?


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)

COMPLEXIDADE

- O bucket sort é considerado linear quando a entrada é gerada a partir de uma distribuição uniforme. A operação de criar os baldes, ler o valor e distribuir os dados para os o baldes e retornar os dados é realizada em tempo linear.

COMPLEXIDADE BÁSICA

- Vamos definir **A** como uma sequencia de entradas ***n*** (chave, elemento) sendo as chaves no intervalo de [0, N − 1]. O Bucket-sort usa as chaves (Keys nas literaturas) como índices em um array auxiliar **B** de sequencia de baldes.

- **Passos:**
- **1º PASSO:** Esvaziar a sequencia de ***A***, movendo cada *entrada (k, o)* para o balde ***B****[k]*.
- **2º PASSO:** Para *i = 0, ... , N-1,* move as entradas do balde ***B****[i]* para o fim da sequencia ***A***.

- ***Analise de complexidade dos PASSOS:***
- *O 1º Passo tem ordem de tempo* ***O(n).***
- *O 2º Passo tem ordem de tempo* ***O(n+ N).***
- ***O Bucket Sorte tem então complexidade O(n+ N).***

BUCKETSORT(A, N)

ENTRADA: *sequencia* ***A*** *de (key, elementos) itens com keys no intervalo de [0, N − 1]*

SAÍDA: *sequencia* ***A*** *ordenada pela ordem crescente das Keys.*

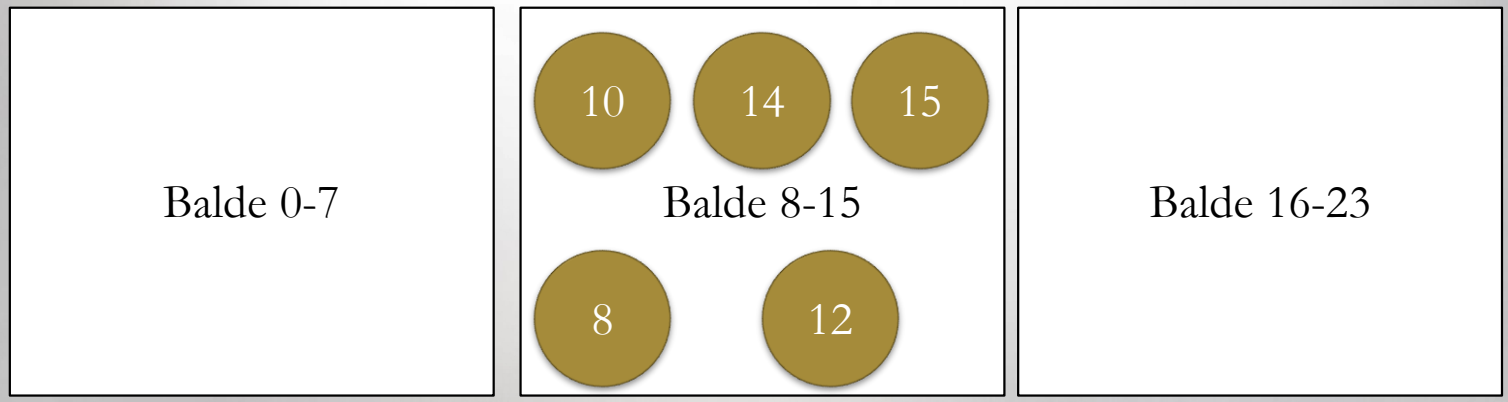
```
B ← array vazio de N intervalos
while ¬A.estaVazio()
        f ← A.primeiro()
        (k, o) ← A.remove(f)
        B[k].insereUltimo((k, o))
                for i ← 0 to (N – 1)
                        while ¬B[i]. estaVazio ()
                                f ← B[i].primeiro()
                                (k, o) ← B[i].remove(f)
                                A.insereUltimo((k, o))
```

COMPLEXIDADE TOTAL

- Como as entradas são uniformemente distribuídas sobre [O, 1), não é esperado que muitos números caiam em cada balde.
- Assim a execução seria O(n) + um somatório da execução de um algoritmo de ordenação (considerado quadrático) para cada balde.
- $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O({n_i}^2)$

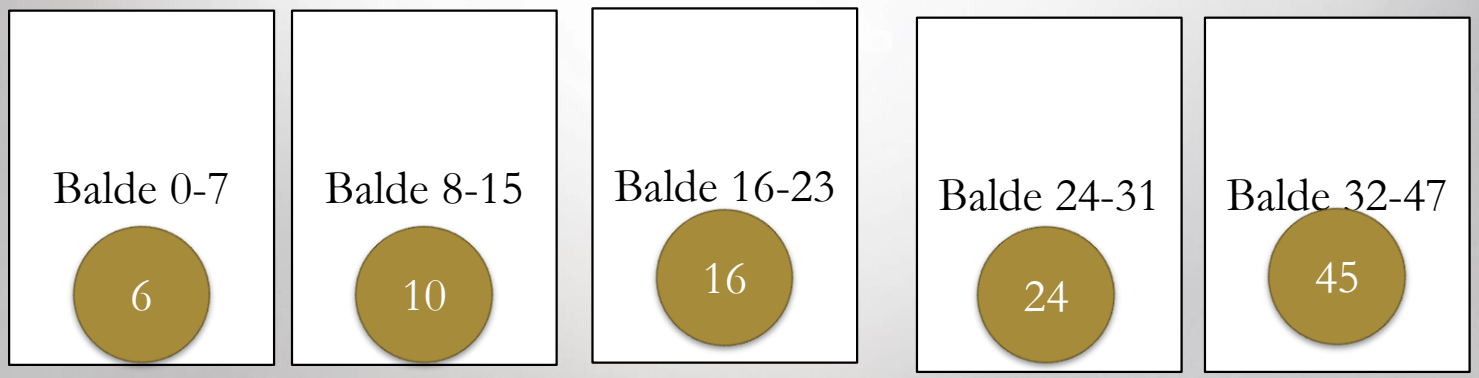
COMPLEXIDADE

❖ O pior caso de execução é o de uma distribuição não homogênea, ou seja, grande concentração de dados em poucos baldes.



COMPLEXIDADE

- O melhor caso é considerado quando os dados são distribuídos um por balde.



REFERENCIAL BIBLIOGRÁFICO

- CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L. et al. **Algoritmos: teoria e prática**. 2.ed. Editora Campus, 2002.
- AHO, A.V & ULLMAN, J.D. & HOPCROFT, J.E. **Data Structures and Algorithms**. 3.ed. Editora Addison Wesley