# AED2 - Aula 15
# Ordenação por contagem (counting sort)

## Ordenação por contagem

Este método é especializado na ordenação de

- vetores de inteiros pequenos

e não é baseado na comparação entre elementos do vetor,

- por isso pode vencer o limitante inferior Omega (n lg n) visto na última aula.

Para desenvolvermos a ideia do algoritmo

- vamos supor que no vetor v de tamanho n
  - só existem inteiros entre 0 e R - 1.

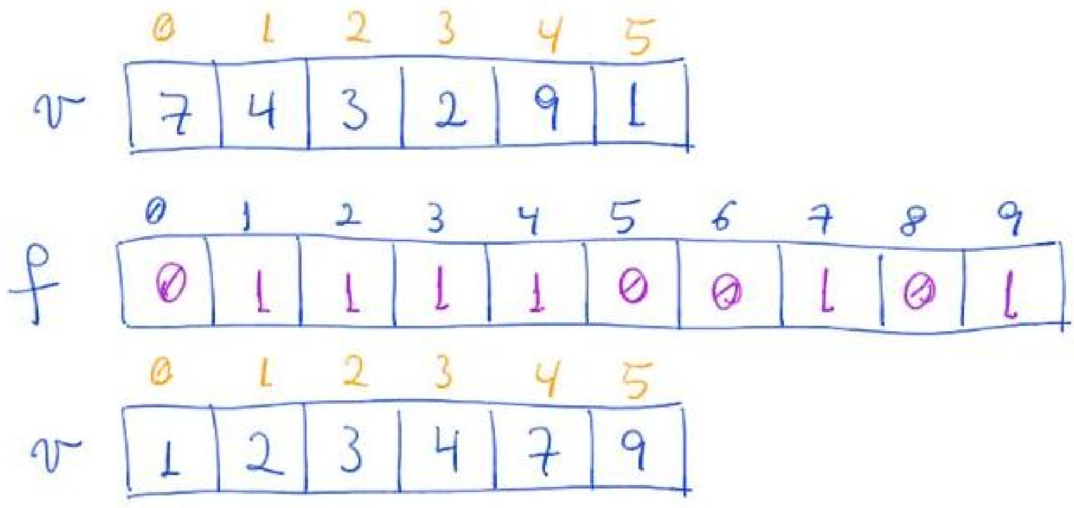
Para simplificar,

- primeiro vamos supor que não existem elementos repetidos.

Neste caso, podemos alocar um vetor auxiliar f

- inicializar f com 0
- percorrer v com um índice i
  - marcando f[v[i]] = 1
- limpar o vetor v
- percorrer f da esquerda para a direita com um índice r
  - colocando r na próxima posição livre de v
    - se v[r] = 1

Exemplo:




Código:

```c
// versao do counting sort que nao trata repeticoes
void countingSortErrado1(int v[], int n, int R)
{
   int *f, r, i;
   f = malloc(R * sizeof(int));
   for (r = 0; r < R; r++)
       f[r] = 0;
   for (i = 0; i < n; i++)
       f[v[i]] = 1;
   i = 0;
   for (r = 0; r < R; r++)
       if (f[r] == 1)
           v[i++] = r;
   free(f);
}
```

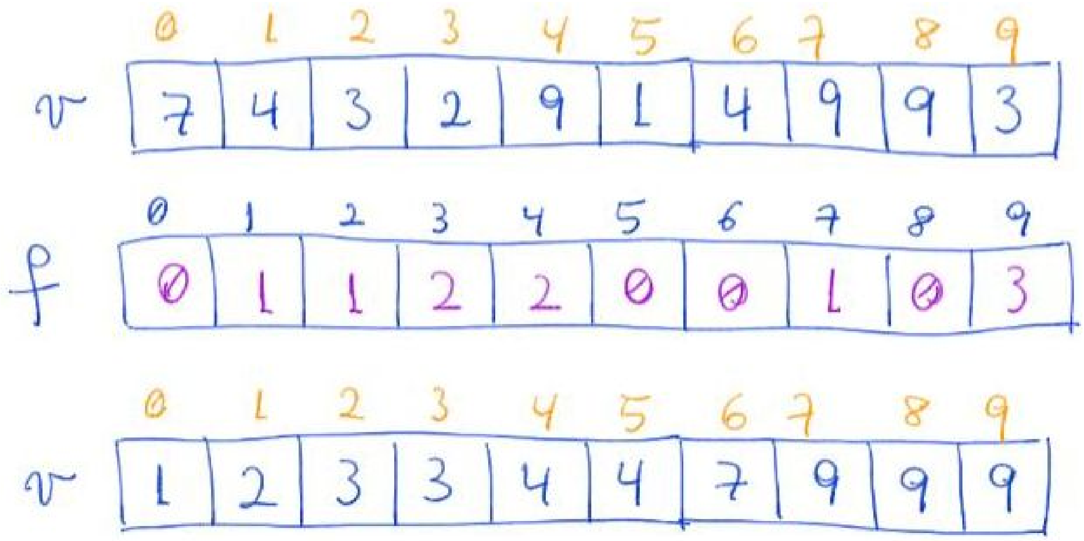
Agora vamos considerar que podem existir elementos repetidos.

- Para tanto, vamos usar o conceito de frequência de um elemento.

Nesta nova abordagem, vamos alocar um vetor auxiliar f

- inicializar f com 0
- percorrer v com um índice i
    - fazendo f[v[i]] += 1
    - Assim, f[r] possui o número de ocorrências de r
- limpar o vetor v
- percorrer f da esquerda para a direita com um índice r
    - colocando v[r] cópias de r nas próximas posições livres de v

Exemplo:




Código:

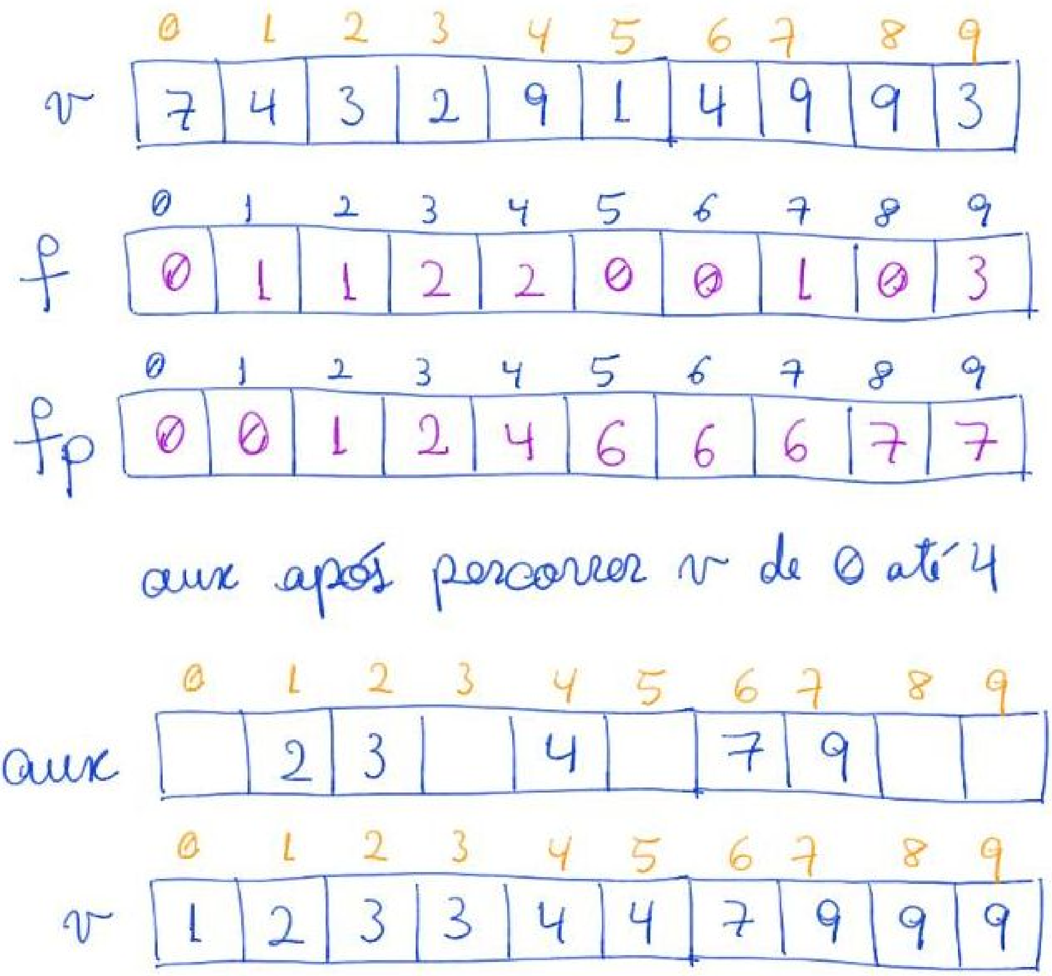
```c
void countingSortErrado2(int v[], int n, int R)
{
   int *f = malloc(R * sizeof(int));
   for (int r = 0; r < R; ++r)
       f[r] = 0;
   for (int i = 0; i < n; ++i)
       f[v[i]] += 1;
   int i = 0;
   for (int r = 0; r < R; ++r)
       for (int k = 0; k < f[r]; ++k)
           v[i++] = r;
   free(f);
}
```

Apesar de aparentar estar correto

- este último algoritmo
    - assim como o primeiro
- apresenta um erro fundamental
    - ele não está ordenando os elementos originais,
    - mas apenas criando cópias das chaves destes
- Esse é um problema grave quando
    - as chaves sendo ordenadas
        - são parte de elementos que possuem outras informações
            - registros ou ponteiros, por exemplo
        - ou ainda quando são partes de uma chave maior
            - como veremos na aplicação do counting sort
                - para o LSD radix sort
- Para resolver esse problema
    - ou seja, para copiar os elementos originais e manter estabilidade
        - é preciso saber a quantidade de elementos
            - que aparece antes de cada chave.
- Para isso, vamos calcular a frequência dos predecessores
    - usando a frequência de cada chave.
    - Sendo f[r] a o número de ocorrência da chave r
        - a frequência dos predecessores de r é
            - fp[r] = f[0] + … + f[r - 1]
    - Podemos usar uma definição recursiva
        - fp[r] = fp[r - 1] + f[r - 1], se r > 0
        - fp[0] = 0
    - Esta definição deriva da seguinte observação
        - fp[r] = f[0] + … + f[r - 2] + f[r - 1]
        - fp[r - 1] = f[0] + … + f[r - 2]
        - Portanto,
            - fp[r] = (f[0] + … + f[r - 2]) + f[r - 1] = fp[r - 1] + f[r - 1]

- Também precisaremos de um vetor auxiliar
    - aux[0 .. n - 1]
    - para podermos copiar um elemento de uma posição em v
        - para uma posição diferente em aux
            - sem corromper elementos ainda não copiados de v.

Exemplo:




Código:

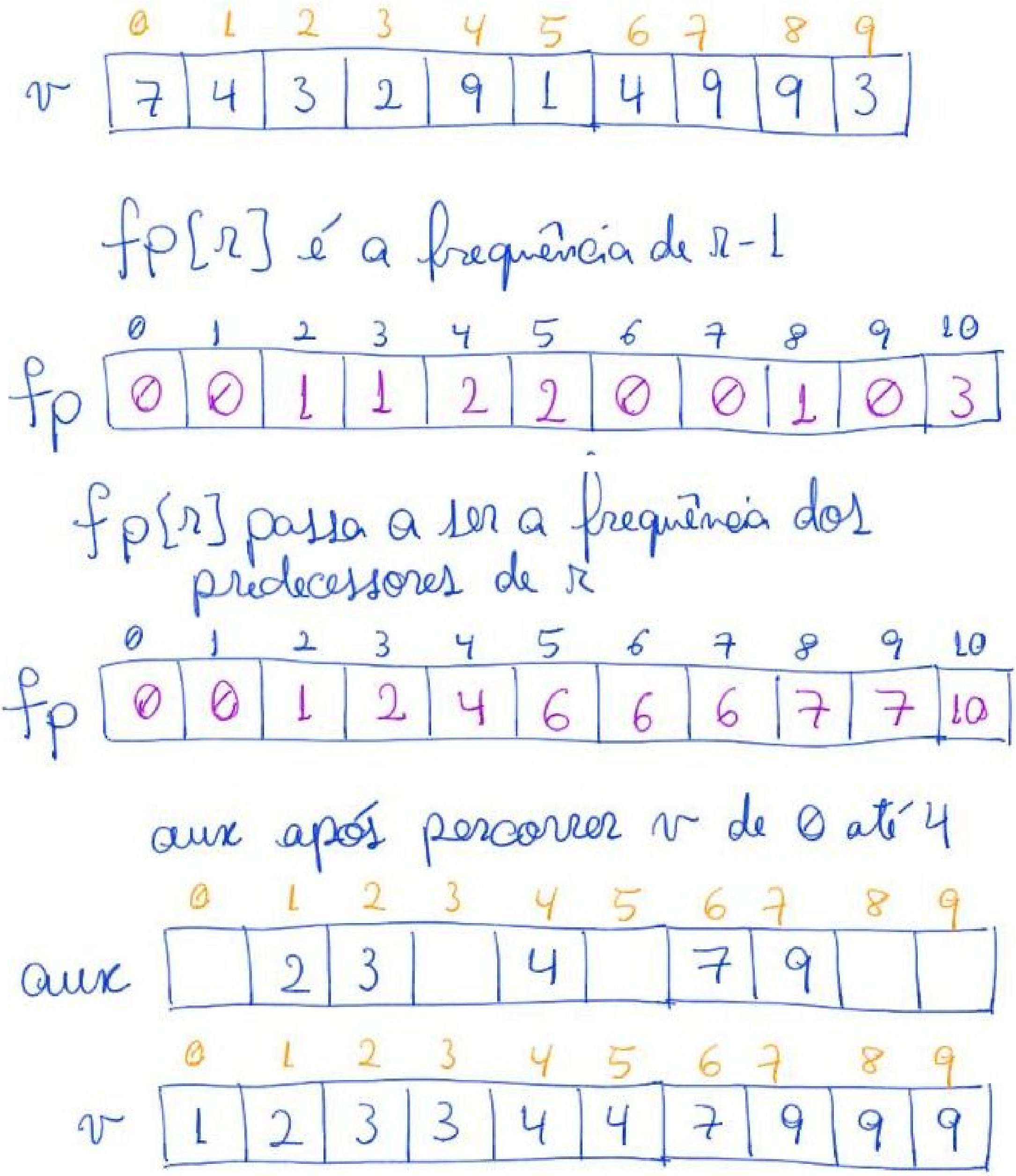
```
// Rearranja v[0..n-1] em ordem crescente
// supondo que os elementos do vetor
// pertencem ao universo 0..R-1.
void countingSort(int v[], int n, int R)
{
   int r, i;
   int *f, *fp, *aux;
   f = malloc(R * sizeof(int));
   fp = malloc(R * sizeof(int));
   aux = malloc(n * sizeof(int));
```

```
    for (r = 0; r < R; ++r)
        f[r] = 0;
    for (int i = 0; i < n; ++i)
        f[v[i]] += 1;
    // agora f[r] é a frequência de r
    fp[0] = 0;
    for (r = 1; r < R; ++r)
        fp[r] = f[r - 1] + fp[r - 1];
    // fp[r] é a freq dos predecessores de r
    // logo, a carreira de elementos iguais a r
    // deve começar no índice fp[r]
    for (i = 0; i < n; ++i)
    {
        r = v[i];
        aux[fp[r]] = v[i];
        fp[r]++; // *
    }
    // aux[0..n-1] está em ordem crescente
    for (i = 0; i < n; ++i)
        v[i] = aux[i];
    free(f);
    free(fp);
    free(aux);
}
```

Esta última versão do counting sort está correta

- no entanto, ela desperdiça memória por alocar espaço para f e para fp.
    - Observe que só usamos f para calcular os valores de fp.
- Assim, uma melhoria envolve alocar um único vetor fp,
    - usá-lo inicialmente para armazenar a frequência das chaves,
    - e reaproveitá-lo para armazenar a frequência dos predecessores.
- Isso é possível,
    - mas exigirá algumas mudanças sutis.
    - Em particular, vamos armazenar a frequência da chave r
        - em fp[r + 1]
    - Com isso, a princípio
        - a posição fp[r] terá a frequência de r - 1
    - Lembrando que
        - fp[r] = fp[r - 1] + f[r - 1], se r > 0
    - Para que fp[r] passe a armazenar a frequência dos predecessores
        - basta somar a ele fp[r - 1]
            - já que a frequência de r - 1 (f[r - 1]) já está lá.

Exemplo:




Código:

```
// Rearranja v[0..n-1] em ordem crescente
// supondo que os elementos do vetor
// pertencem ao universo 0..R-1.
void countingSort2(int v[], int n, int R)
{
   int r;
   int *fp, *aux;
   fp = malloc((R + 1) * sizeof(int));
   aux = malloc(n * sizeof(int));

   for (r = 0; r <= R; ++r)
       fp[r] = 0;
```

```c
    for (int i = 0; i < n; ++i)
    {
        r = v[i];
        fp[r + 1] += 1;
    }
    // agora fp[r] é a frequência de r-1
    for (r = 1; r <= R; ++r)
        fp[r] += fp[r - 1];
    // agora fp[r] é a freq dos predecessores de r
    // logo, a carreira de elementos iguais a r
    // deve começar no índice fp[r]
    for (int i = 0; i < n; ++i)
    {
        r = v[i];
        aux[fp[r]] = v[i];
        fp[r]++; // *
    }
    // aux[0..n-1] está em ordem crescente
    for (int i = 0; i < n; ++i)
        v[i] = aux[i];

    free(fp);
    free(aux);
}
```

Curiosidade:

- Note que, fp foi alocado com uma posição a mais,
    - mas o único motivo para tanto é evitar que, no segundo laço
        - seja acessada uma posição de memória inválida,
        - quando r = v[i] = R - 1 e fp[r + 1] recebe um incremento.

Eficiência de tempo:

- countingsort leva tempo da ordem de n + R.
    - se R é pequeno (da ordem de n no pior caso),
        - isso é melhor que a eficiência O(n log n) de algoritmos como
            - mergeSort, quickSort e heapSort.

Estabilidade:

- countingsort é estável.
- Essa propriedade é a base da aplicação do countingsort para o LSD radix.