AED2 - Aula 14
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.

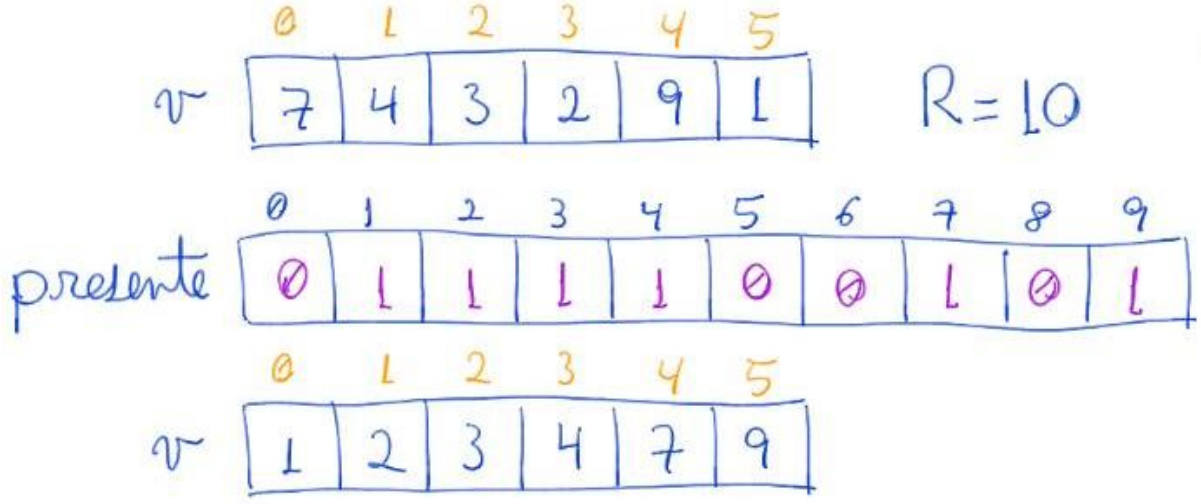
Para simplificar,

- primeiro vamos supor que não existem elementos repetidos.

Neste caso, podemos alocar um vetor auxiliar *presente*.

- Inicializar *presente* com 0.
- Percorrer v com um índice i,
    - marcando *presente*[v[i]] = 1.
- Percorrer *presente* da esquerda para a direita com um índice *valor*,
    - colocando *valor* na próxima posição livre de v
        - se *presente*[*valor*] = 1.

Exemplo:



Código:

```c
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
// desde que não existam elementos repetidos
void countingSortErrado1(int v[], int n, int R)
{
    int *presente, valor, i;
    presente = malloc(R * sizeof(int));
    for (valor = 0; valor < R; valor++)
        presente[valor] = 0;
    for (i = 0; i < n; i++)
        presente[v[i]] = 1;
    i = 0;
    for (valor = 0; valor < R; valor++)
        if (presente[valor] == 1)
            v[i++] = valor;
    free(presente);
}
```

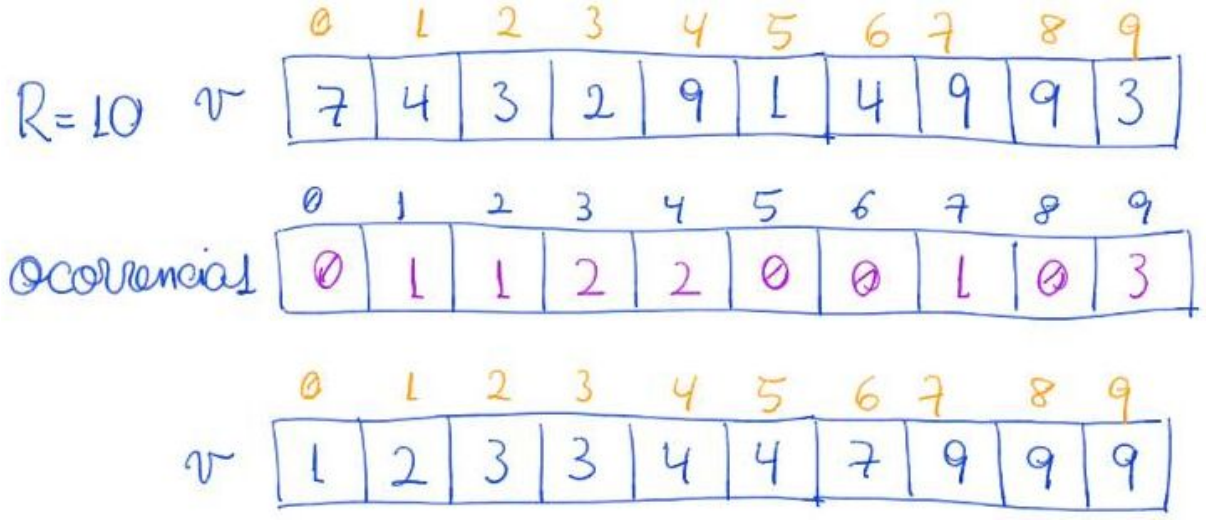
Agora vamos considerar que podem existir elementos repetidos.

- Para tanto, vamos usar o número de ocorrências de um elemento.

Nesta nova abordagem, vamos alocar um vetor auxiliar *ocorrencias*.

- Inicializar *ocorrencias* com 0.
- Percorrer v com um índice i,
    - fazendo *ocorrencias*[v[i]] += 1.
- Assim, para cada *valor* em [0, R), ao final do laço
    - *ocorrencias*[*valor*] possuirá o número de ocorrências de *valor*.
- Percorrer *ocorrencias* da esquerda para a direita com um índice *valor*,
    - colocando *ocorrencias*[*valor*] cópias de *valor*
        - nas próximas posições livres de v.

Exemplo:



Código:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
// copia ao invés de rearranjar
void countingSortErrado2(int v[], int n, int R)
{
    int *ocorrencias, valor, i, repet;
    ocorrencias = malloc(R * sizeof(int));
    for (valor = 0; valor < R; valor++)
        ocorrencias[valor] = 0;
    for (i = 0; i < n; ++i)
        ocorrencias[v[i]] += 1;
    i = 0;
    for (valor = 0; valor < R; valor++)
        for (repet = 0; repet < ocorrencias[valor]; repet++)
            v[i++] = valor;
    free(ocorrencias);
}
```

Apesar de aparentarem estar corretos,

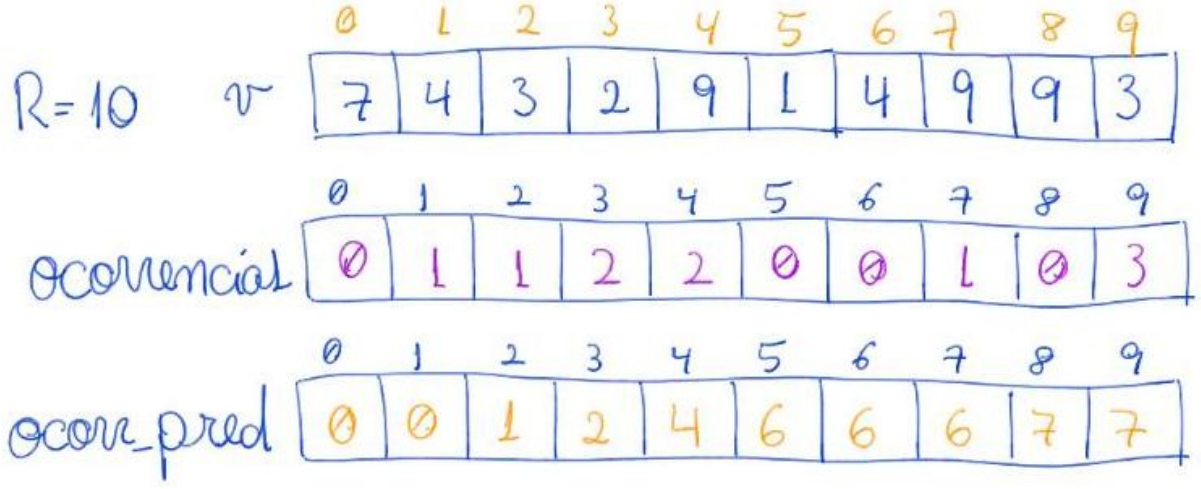
- tanto este último algoritmo quanto o primeiro,
    - apresentam um erro fundamental.
- Eles não estão ordenando os elementos originais,
    - mas apenas criando cópias das chaves destes.
- Esse é um problema grave quando as chaves sendo ordenadas
    - são parte de elementos que possuem outras informações,
        - como registros ou ponteiros, por exemplo.

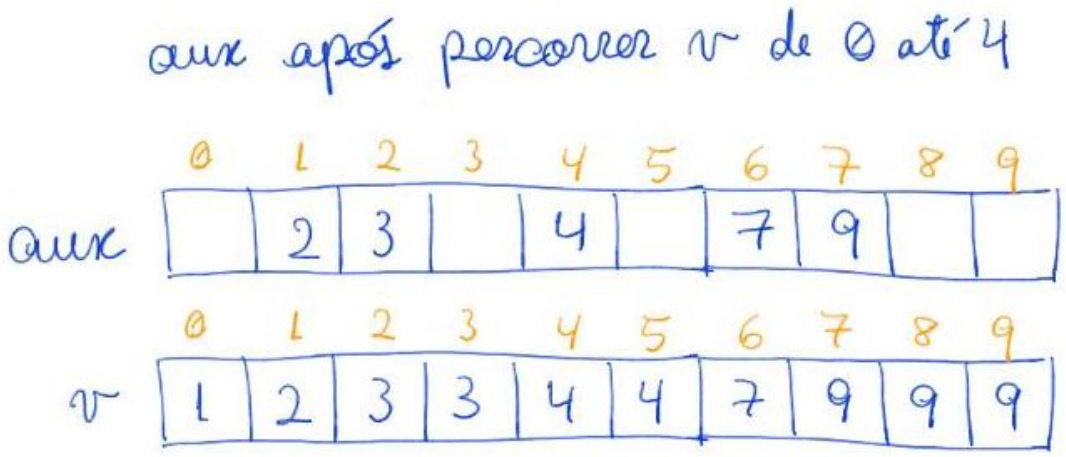
    - Ou ainda, quando são partes de uma chave maior,
        - como veremos na aplicação do counting sort
            - como subrotina do LSD radix sort.

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 o número de ocorrência dos predecessores
    - usando a ocorrência de cada chave.
- Sendo *ocorrencias*[*valor*] o número de ocorrências da chave *valor*,
    - o número de ocorrências dos predecessores de *valor* é
        - *ocorr_pred*[*valor*] = *ocorrencias*[0] + … + *ocorrencias*[*valor* - 1]
- Podemos usar uma definição recursiva
    - *ocorr_pred*[*valor*] = *ocorr_pred*[*valor* - 1] + *ocorrencias*[*valor* - 1],
        - se *valor* > 0.
    - *ocorr_pred*[0] = 0.
- Esta definição deriva da seguinte observação
    - *ocorr_pred*[*valor*] = *ocorrencias*[0] + … + *ocorrencias*[valor - 2] + *ocorrencias*[valor - 1]
    - *ocorr_pred*[*valor - 1*] = *ocorrencias*[0] + … + *ocorrencias*[valor - 2]
- Portanto,
    - *ocorr_pred*[*valor*] = (*ocorrencias*[0] + … + *ocorrencias*[valor - 2]) + *ocorrencias*[valor - 1] = *ocorr_pred*[*valor - 1*] + *ocorrencias*[valor - 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:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
void countingSort(int v[], int n, int R)
{
    int valor, i;
    int *ocorrencias, *ocorr_pred, *aux;
    ocorrencias = malloc(R * sizeof(int));
    ocorr_pred = malloc(R * sizeof(int));
    aux = malloc(n * sizeof(int));

    for (valor = 0; valor < R; valor++)
        ocorrencias[valor] = 0;
    for (i = 0; i < n; i++)
        ocorrencias[v[i]] += 1;
    ocorr_pred[0] = 0;
    for (valor = 1; valor < R; valor++)
        ocorr_pred[valor] = ocorr_pred[valor - 1] +
ocorrencias[valor - 1];
    // Os elementos iguais a valor devem
    // começar no índice ocorr_pred[valor]
    for (i = 0; i < n; i++)
    {
        valor = v[i];
        aux[ocorr_pred[valor]] = v[i];
        ocorr_pred[valor]++; // atualiza o número de predecessores
```

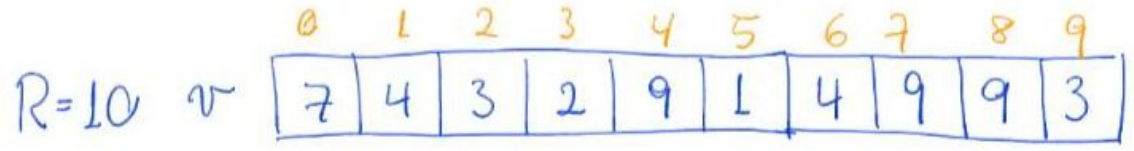
```
    }
    // aux[0 .. n-1] está em ordem crescente
    for (i = 0; i < n; ++i)
        v[i] = aux[i];

    free(ocorrencias);
    free(ocorr_pred);
    free(aux);
}
```

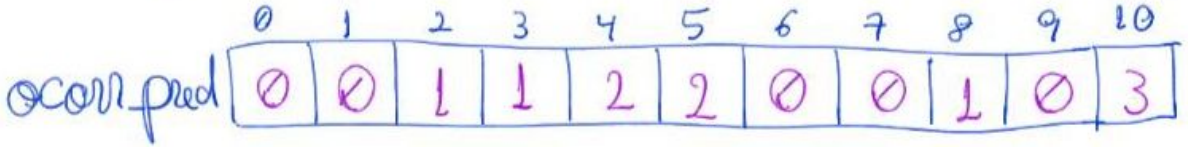
Esta última versão do counting sort está correta.

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

Exemplo:

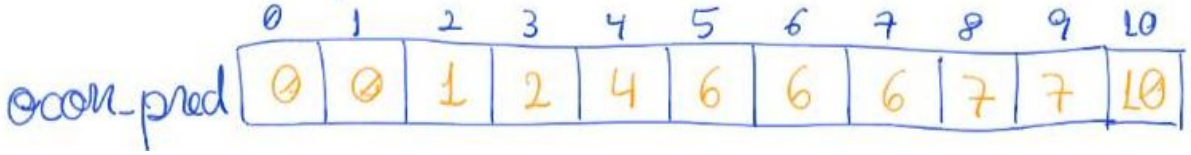


ocorr_pred[valor] é o # de ocorrências de valor - 1

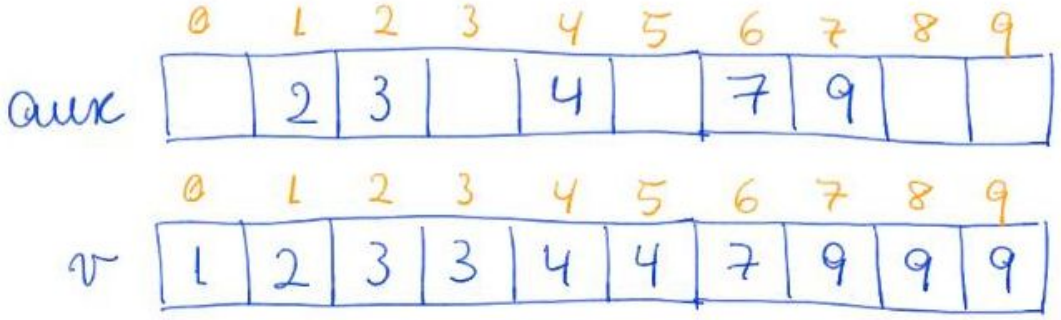


para (valor = 1 até n) faça (ocorr_pred[valor] += ocorr_pred[valor - 1])

ocorr_pred[valor] passa a ler o # de ocorrências dos predecessores de valor



aux após percorrer v de 0 até 4



Código:

```
// ordena um vetor v[0 .. n-1] de inteiros em [0, R)
void countingSort2(int v[], int n, int R)
{
    int valor, i;
    int *ocorr_pred, *aux;
    ocorr_pred = malloc((R + 1) * sizeof(int));
    aux = malloc(n * sizeof(int));
```

```
    for (valor = 0; valor <= R; valor++)
        ocorr_pred[valor] = 0;
    for (i = 0; i < n; i++)
    {
        valor = v[i];
        ocorr_pred[valor + 1] += 1;
    }
    // agora ocorr_pred[valor] é o número
    // de ocorrências de valor - 1
    for (valor = 1; valor <= R; valor++)
        ocorr_pred[valor] += ocorr_pred[valor - 1];
    // agora ocorr_pred[valor] é o número de
    // ocorrências dos predecessores de valor.
    // Logo, a carreira de elementos iguais a valor
    // deve começar no índice ocorr_pred[valor].
    for (i = 0; i < n; i++)
    {
        valor = v[i];
        aux[ocorr_pred[valor]] = v[i];
        ocorr_pred[valor]++; // atualiza o número de predecessores
    }
    // aux[0 .. n-1] está em ordem crescente
    for (i = 0; i < n; ++i)
        v[i] = aux[i];

    free(ocorr_pred);
    free(aux);
}
```

Curiosidade:

- Note que, *ocorr_pred* 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 *valor* = v[i] = R - 1
            - e *ocorr_pred*[*valor* + 1] recebe um incremento.

Eficiência de tempo:

- countingSort leva tempo da ordem de n + R, i.e., O(n + R),
    - já que cada laço itera por R ou n vezes,
        - e não temos laços aninhados.
- Se R é pequeno (da ordem de n no máximo),
    - isso é melhor que a eficiência O(n log n) de algoritmos como
        - mergeSort, quickSort e heapSort.
- Por isso, countingSort é o método preferido para ordenar
    - vetores cujas chaves são inteiros pequenos.

Eficiência de espaço:

- O(n + R) já que precisamos de
    - um vetor *ocorr_pred* de tamanho proporcional a R
        - e um vetor aux de tamanho proporcional a n.

Estabilidade:

- countingSort é estável.

Quiz:

- A ordem de alguns laços do algoritmo é decisiva
    - para manter a estabilidade, enquanto outros laços
        - poderiam ser invertidos ou seguir uma ordem arbitrária.
- Identifique a relevância da ordem de cada laço do algoritmo.

Curiosidade:

- A estabilidade do countingSort é a propriedade chave
    - que permite aplicá-lo ao LSD radix sort,
        - que veremos na próxima aula.