AED2 - Aula 15
Ordenação por partes (radixSort)

Na última aula vimos o counting sort,

- cuja ideia central é contar o número de predecessores de cada chave
    - para posicionar corretamente os elementos no vetor ordenado.
- Ele é muito eficiente para ordenar conjuntos de elementos
    - cujas chaves são inteiros pequenos.
- Por inteiros pequenos, queremos dizer
    - valores inteiros entre 0 e R - 1,
        - sendo R da ordem do número de elementos do conjunto.

Quiz 1.1: Para perceber a limitação do counting sort, considere um vetor

- com 1 milhão de elementos cujas chaves são inteiros de 32 bits.
    - Qual a eficiência do counting sort neste caso?

Uma alternativa para ordenar conjuntos,

- quando suas chaves são grandes
    - é dividir as chaves em pedaços menores
        - e realizar a ordenação por etapas.
- Chamamos cada um desses pedaços de dígito
    - e seu tamanho deriva da base (radix) utilizada.
- Note que, esses dígitos não pertencem necessariamente
    - ao conjunto {0, …, 9}.
- Essa é a ideia central dos métodos de ordenação radix sort.

Os métodos radix sort também são chamados de ordenação digital,

- por ordenar as chaves dígito-a-dígito,
        - ou strings caracter-a-caracter.
- Eles variam de acordo com a ordem em que consideramos os dígitos,
    - i.e., se vamos dos mais significativos para os menos,
        - ou dos menos para os mais significativos.

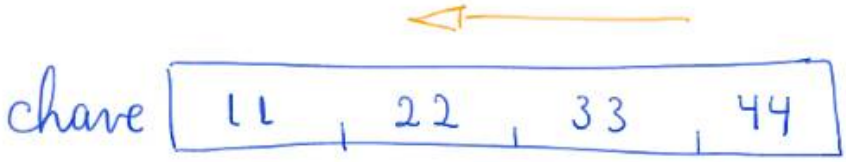
LSD radix sort

Este método é interessante para ordenar

- um vetor v[0 .. n - 1] de chaves, sendo todas do mesmo comprimento.
- Se as chaves são strings de caracteres,
    - estamos falando do problema de colocá-las em ordem lexicográfica.

Nele vamos ordenar o conjunto indo

- do dígito menos significativo até o mais significativo,
    - i.e., percorremos cada chave da direita para a esquerda.



- Em cada etapa ordenamos todo o conjunto
    - considerando um dígito por etapa
        - e usando um método de ordenação estável.
            - Por que isso é essencial?
    - De preferência, o método escolhido é o counting sort,
        - por levar tempo linear, ser estável,
            - e funcionar bem com chaves pequenas (dígitos).

Primeiro veremos uma versão do LSD radix sort

- que trabalha com vetores de caracteres (strings) de mesmo tamanho,
    - sendo cada string uma chave.

Exemplo:

Original	**3° Dígito**	**2° Dígito**	**1° Dígito**
1 2 3	1 2 3	1 2 3	1 2 3
KNG	KDA	KDA	FFU
FFU	RFD	KDV	KDA
KDV	KNG	RFD	KDV
RFD	FFU	FFU	KNG
KDA	KDV	KNG	RFD

Observe que na última coluna todas as chaves estão ordenadas.

- Para entender o motivo, considere a penúltima coluna e observe que
    - se remover dela todos os elementos que não começam com K
        - os elementos restantes já estão em ordem.
- De fato, no início da i-ésima etapa deste método
    - todas as chaves estão ordenadas
        - com relação apenas aos dígitos já considerados,
- Como a ordenação utilizada em cada etapa é estável,
    - esta propriedade invariante se mantém
        - e propaga para mais um dígito a cada nova etapa.

No algoritmo a seguir,

- W é o comprimento, em dígitos, de cada chave (string),
- R é o universo de valores que cada dígito pode assumir.
    - Note que, R < 256, já que cada dígito corresponde a um byte.

Códigos:

```c
typedef unsigned char byte;

// Rearranja em ordem lexicográfica um vetor v[0 .. n - 1]
// de strings. Cada v[i] é uma string v[i][0 .. W - 1]
// cujos elementos pertencem ao conjunto 0 .. R - 1.
void ordenacaoDigital(byte *v[], int n, int W) {
    int *ocorr_pred, digito, valor, i, R = 256;
    byte **aux;
    ocorr_pred = malloc((R + 1) * sizeof(int));
    aux = malloc(n * sizeof(byte *));
    for (digito = W - 1; digito >= 0; digito--) {
        for (valor = 0; valor <= R; valor++)
            ocorr_pred[valor] = 0;
        for (i = 0; i < n; i++) {
            valor = v[i][digito];
            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++) {
            // note a diferença entre o valor analisado e copiado
            valor = v[i][digito];
            aux[ocorr_pred[valor]] = v[i];
            ocorr_pred[valor]++; // atualiza número de predecessores
        }
        // aux[0..n-1] está em ordem crescente considerando
        // apenas os digitos entre digito .. W - 1
        for (i = 0; i < n; i++)
            v[i] = aux[i];
    }
```

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

Invariante e corretude:

- No início de cada iteração do laço externo as strings estão ordenadas
    - com relação aos subvetores v[i][digito + 1 .. W - 1], para i = 0, …, n - 1.

Eficiência de tempo:

- Uma observação importante para analisar o método radix sort
    - é que nenhum dígito é verificado mais de uma vez.
    - Assim, radix sort é linear no número total de dígitos.
- O((n + R) * W), sendo
    - n o número de chaves,
    - R o número de valores que cada dígito pode assumir,
    - W o número de dígitos em cada chave.
- Este método é preferível às ordenações O(n lg n) quando
    - R = O(n) e W = o(lg n),
        - sendo que o(lg n) significa assintoticamente menor que lg n.

Quiz: Considere ordenar 1 milhão de elementos com chaves de 5 dígitos.

- Compare lg do número de elementos com o número de dígitos das chaves
    - para decidir se o LSD radix é preferível ao quickSort, por exemplo.

Eficiência de espaço:

- Memória adicional da ordem de (n + R), i.e., O(n + R),
    - por conta dos vetores auxiliares do countingSort.

Estabilidade:

- É estável, pois como o método que ordena cada dígito tem que ser estável,
    - em nenhuma etapa elementos com a mesma chave serão invertidos,
        - já que eles coincidem em todos os dígitos.

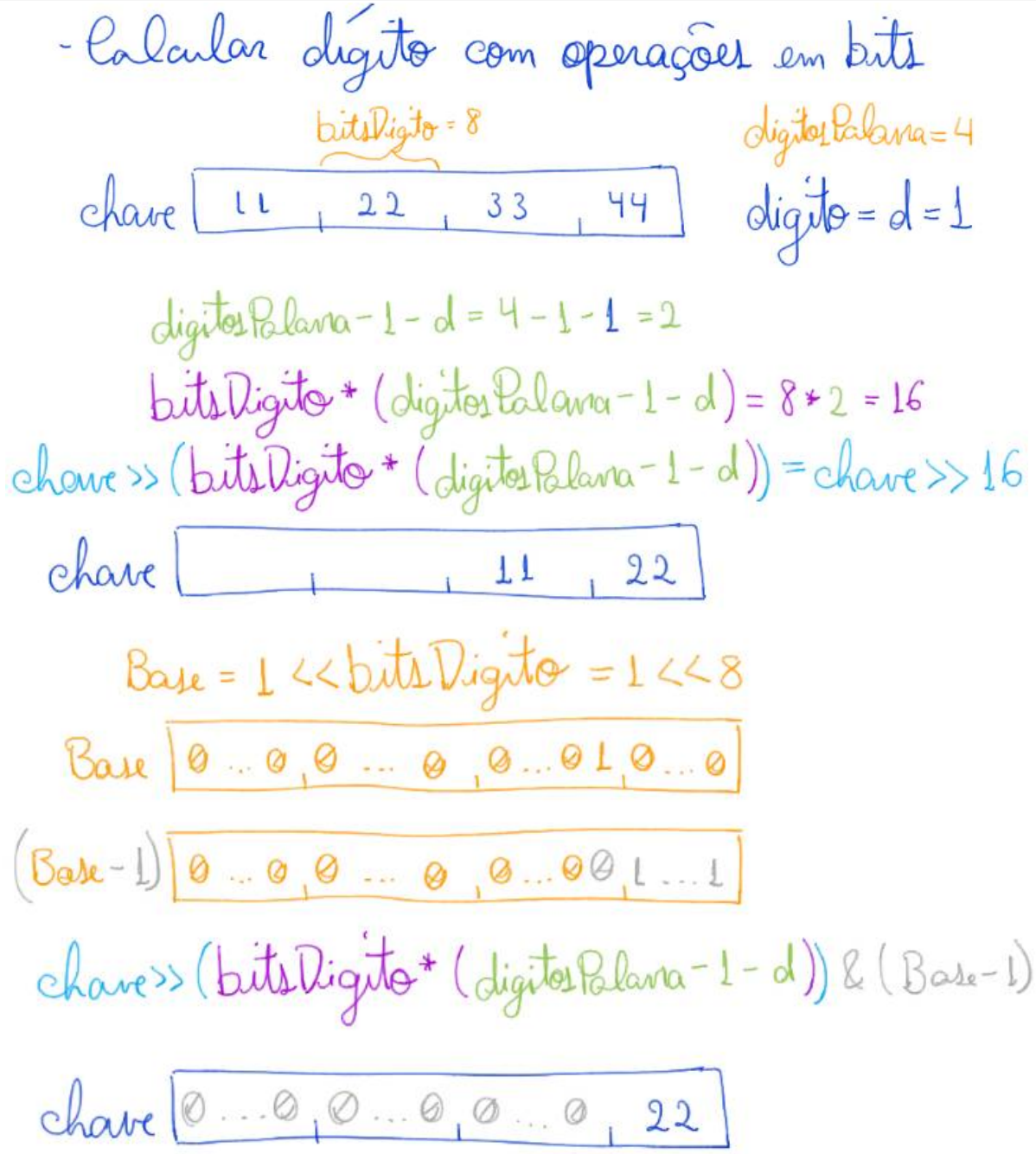
Agora veremos uma versão semelhante do LSD radix sort

- que trabalha com vetores de inteiros, sendo cada inteiro uma chave.
- Esta versão manipula explicitamente os bits das chaves
    - para obter cada dígito.

Códigos:

```
const int bitsPalavra = 32;
const int bitsDigito = 8;
const int digitosPalavra = bitsPalavra / bitsDigito;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito

int pegaDigito(int chave, int digito) {
    return (int)((chave >>
      (bitsDigito * (digitosPalavra - 1 - digito))) & (Base - 1));
}
```



```c
int pegaDigito2(int chave, int digito) {
    return (int)(chave /
     exp2(bitsDigito * (digitosPalavra - 1 - digito))) % Base;
}

void LSDradixSort(int v[], int n) {
    int digito, valor, i;
    int *ocorr_pred, *aux;
    ocorr_pred = malloc((Base + 1) * sizeof(int));
    aux = malloc(n * sizeof(int));

    for (digito = digitosPalavra - 1; digito >= 0; digito--) {
        for (valor = 0; valor <= Base; valor++)
            ocorr_pred[valor] = 0;
        for (i = 0; i < n; i++) {
            valor = pegaDigito(v[i], digito);
            ocorr_pred[valor + 1] += 1;
        }
        // agora ocorr_pred[valor] é o número
        // de ocorrências de valor - 1
        for (valor = 1; valor <= Base; 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) {
            // note a diferença entre o valor analisado e copiado
            valor = pegaDigito(v[i], digito);
            aux[ocorr_pred[valor]] = v[i];
            ocorr_pred[valor]++; // atualiza o número de
predecessores
        }
        // aux[0..n-1] está em ordem crescente considerando
        // apenas os digitos entre digito .. digitosPalavra - 1
        for (i = 0; i < n; i++)
            v[i] = aux[i];
    }
    free(ocorr_pred);
    free(aux);
}
```

Eficiência de tempo:

- Uma observação importante para analisar os método radix sort
    - é que nenhum dígito é verificado mais de uma vez.
    - Assim, radix sort é linear no número de dígitos.
- O((n + Base) * digitosPalavra), sendo:
    - n o número de chaves.
    - Base o número de valores que cada dígito pode assumir.
        - Note que, sendo bitsDigito o número de bits de cada dígito,
            - Base = 2^bitsDigito.
    - digitosPalavra é o número de dígitos em cada chave.
        - Note que, sendo digitosPalavra o número de bits por chave,
            - digitosPalavra = bitsPalavra / bitsDigito.
- Se os dígitos forem pequenos e as chaves grandes
    - a complexidade de tempo pode superar (n lg n) assintoticamente.
- Já se os dígitos forem grandes em relação às chaves
    - a eficiência tende a O(n).
- Por exemplo, tome n = 1 bilhão = 10^9.
    - Se bitsDigito = 1 (Base = 2^1 = 2) e bitsPalavra = 64,
        - temos digitosPalavra = 64/ 1 = 64,
            - que é maior que lg 10^9 <= lg 2^(10*3) ~= 30.
    - Se bitsDigito = 16 (Base = 2^16 = 65536) e bitsPalavra = 32,
        - temos digitosPalavra = 32 / 16 = 2,
            - que é muito menor que lg 10^9 ~= 30.

Quiz 1.2: Para perceber a vantagem do radix sort, considere um vetor

- com 1 milhão de elementos cujas chaves são inteiros de 32 bits.
    - Qual a eficiência do radix sort com digito de 8 bits?

Eficiência de espaço:

- Memória adicional da ordem de (n + Base), i.e, O(n + Base),
    - por conta dos vetores auxiliares do countingSort.

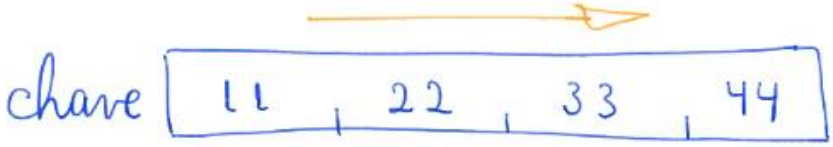
Estabilidade:

- É estável, pois como o método que ordena cada dígito tem que ser estável,
    - em nenhuma etapa elementos com a mesma chave serão invertidos,
        - já que eles coincidem em todos os dígitos.

MSD radix sort

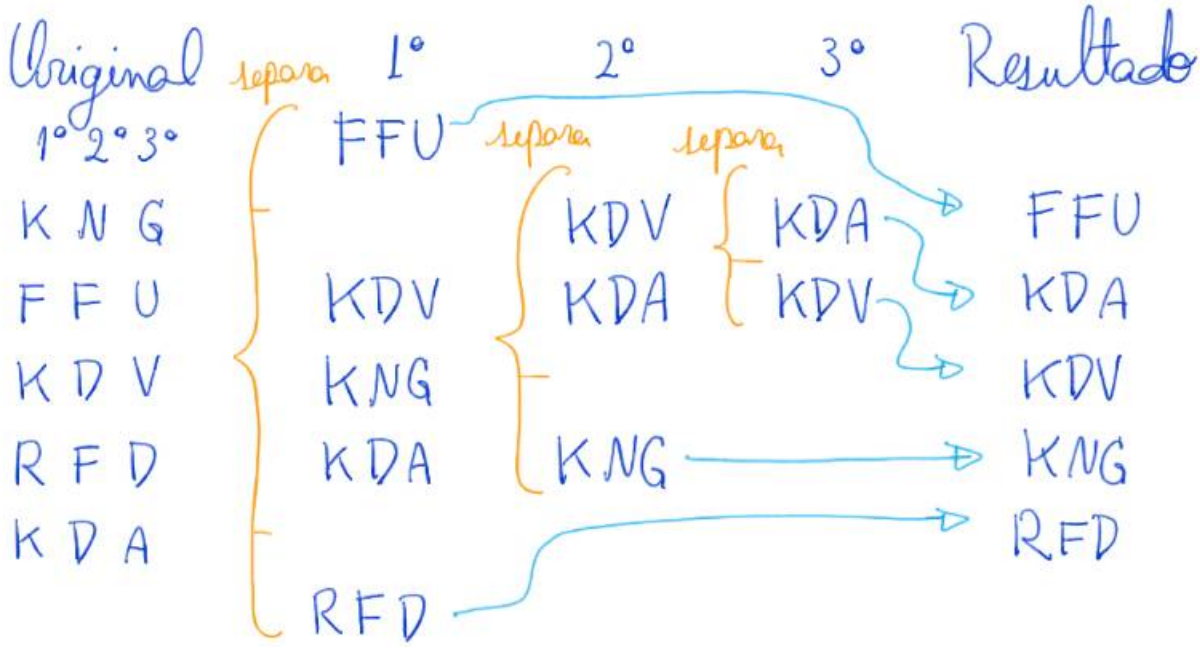
Neste método vamos ordenar o conjunto indo

- do dígito mais significativo até o menos significativo,
    - i.e., percorremos cada chave da esquerda para a direita.



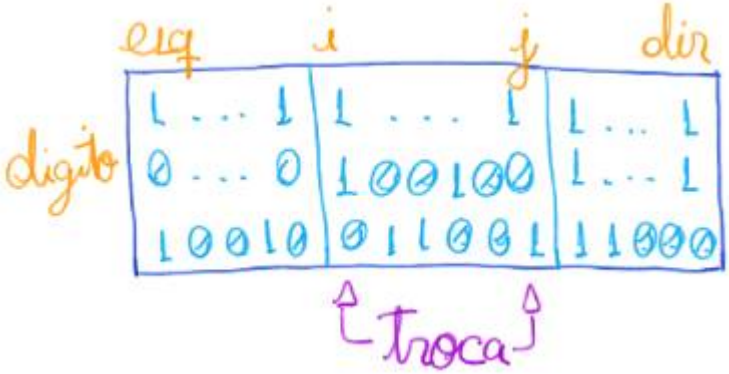
- Em cada etapa podemos usar uma variante do algoritmo da separação,
    - que estudamos junto do quickSort.
- Esta variante só considera o dígito corrente
    - e divide o conjunto em Base subconjuntos,
        - sendo Base o número de possibilidades de valor de um dígito.
- Este método pode lidar com chaves de comprimentos variados,
    - e nem sempre precisa avaliar todos os dígitos de todas as chaves,
        - o que pode melhorar sua eficiência.

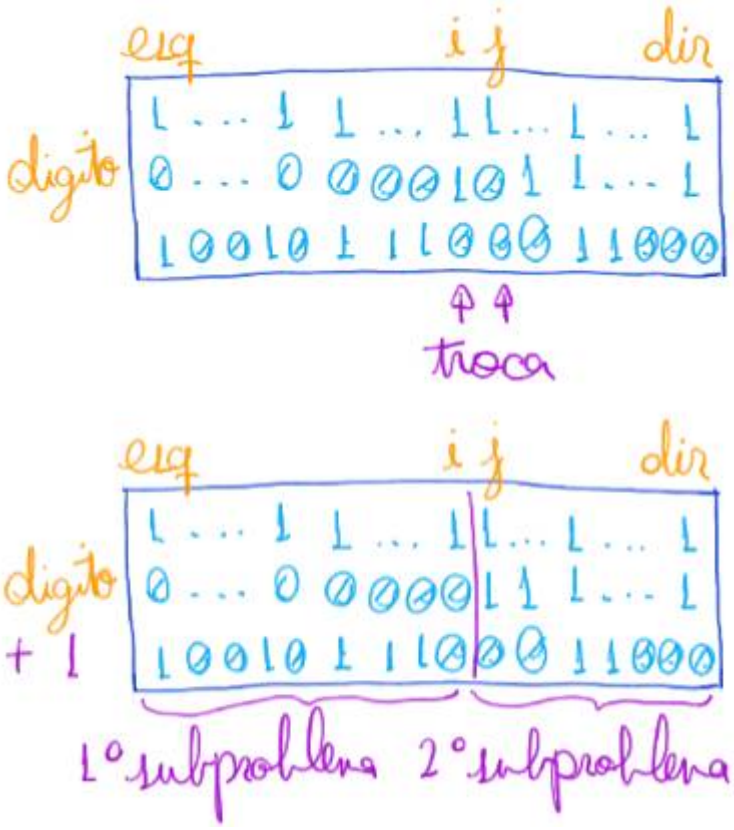
Exemplo:



Vamos focar em um caso particular, e mais simples, do MSD radix sort

- em que o dígito tem tamanho 1 e Base vale 2^1 = 2
    - chamado de binary quicksort.





Códigos:

```
const int bitsPalavra = 32;
const int bitsDigito = 1;
const int digitosPalavra = bitsPalavra / bitsDigito;
const int Base = 1 << bitsDigito; // Base = 2^bitsDigito

int pegaDigito(int chave, int digito) {
    return (int)((chave >>
      (digitosPalavra - 1 - digito)) & (Base - 1));
}

int pegaDigito2(int chave, int digito) {
    return (int)(chave / exp2(digitosPalavra - 1 - digito)) % Base;
}

// esq indica a primeira posicao, dir indica a ultima, digito indica
o digito corrente
void quicksortBin(int v[], int esq, int dir, int digito) {
    int i, j;
    i = esq;
    j = dir;
    // caso base quando subvetor é pequeno ou acabaram os dígitos
    if (dir <= esq || digito > digitosPalavra)
        return;
    // separa as chaves do subvetor de acordo com o dígito corrente
```

```
    while (j > i) {
        while (pegaDigito(v[i], digito) == 0 && i < j)
            i++;
        while (pegaDigito(v[j], digito) == 1 && j > i)
            j--;
        troca(&v[i], &v[j]);
    }
    // ajusta j para que v[esq .. j-1] tenha chaves com digito 0
    if (pegaDigito(v[dir], digito) == 0)
        j++;
    quicksortBin(v, esq, j - 1, digito + 1);
    // quicksortBin(v, esq, j - 1, digito + bitsDigito);
    quicksortBin(v, j, dir, digito + 1);
    // quicksortBin(v, j, dir, digito + bitsDigito);
}

void MSDradixSort(int v[], int n) {
    quicksortBin(v, 0, n - 1, 0);
}
```

Eficiência de tempo:

- Uma observação importante para analisar os método radix sort
    - é que nenhum dígito é verificado mais de uma vez.
    - Assim, radix sort é linear no número de dígitos.
- O(n * digitosPalavra) = O(n * bitsPalavra) no caso do binary quicksort.

Eficiência de espaço:

- Memória adicional da ordem de bitsPalavra,
    - por conta da profundidade máxima da pilha de recursão.

Estabilidade:

- Não é estável, por conta das trocas da separação.

Quiz: Como seria uma versão deste algoritmo com bitsDigito > 1?

- Dica: solução para separação inspirada
    - na contagem de predecessores do counting sort.
- Por conta disso, um termo R = Base surge na eficiência do algoritmo.