

AED2 - Aula 16

Radix sort e bucket sort

Radix sort

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.

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 precisam pertencer ao conjunto $\{0, \dots, 9\}$.
- 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.

MSD radix sort

- **Bônus, conteúdo extra**

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 R subconjuntos,

- sendo R o número de possibilidades de valor de um dígito.
- Um caso particular, e mais simples, deste método
 - em que o dígito tem tamanho 1
 - e R vale $2^1 = 2$
 - é o binary quicksort.

Códigos:

```

const int bitword = 32;
const int bitsdigit = 1;
const int digitword = bitword / bitsdigit;
const int Base = 1 << bitsdigit;

int digit(int a, int d)
{
    // return (int)((a >> (bitsdigit * (digitword - 1 - d))) & (Base - 1));
    return (int)((a >> (bitword - 1 - d)) & (Base - 1));
}

int digit2(int a, int d)
{
    // return (int)(a / exp2(bitsdigit * (digitword - 1 - d))) % Base;
    return (int)(a / exp2(bitword - 1 - d)) % Base;
}

// p indica a primeira posicao, r indica a ultima, d indica o digito corrente
void quicksortBin(int v[], int l, int r, int d)
{
    int i, j;
    i = l;
    j = r;
    if (r <= l || d > bitword)
        return;
    while (j > i)
    {
        while (digit(v[i], d) == 0 && i < j)
            i++;
        while (digit(v[j], d) == 1 && j > i)
            j--;
        troca(&v[i], &v[j]);
    }
    if (digit(v[r], d) == 0)
        j++;
    quicksortBin(v, l, j - 1, d + 1);
    // quicksortBin(v, l, j - 1, d + bitsdigit);
    quicksortBin(v, j, r, d + 1);
    // quicksortBin(v, j, r, d + bitsdigit);
}

```

```
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.
- $O(n * \text{bitsword})$

Eficiência de espaço:

- memória adicional da ordem de bitsword,
 - por conta da profundidade máxima da recursão.

Estabilidade:

- Não é estável.

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,
 - de preferência 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, em cada 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:

```
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 R)
{
    int *fp, d, r, i;
    byte **aux;
    fp = malloc((R + 1) * sizeof(int));
    aux = malloc(n * sizeof(byte *));

    for (d = W - 1; d >= 0; d--)
    {
        for (r = 0; r <= R; r++)
```

```

    fp[r] = 0;
for (int i = 0; i < n; i++)
{
    r = v[i][d];
    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 (i = 0; i < n; i++)
{
    r = v[i][d];
    aux[fp[r]] = v[i];
    fp[r]++;
}
// aux[0..n-1] está em ordem crescente considerando
// apenas os dígitos entre d .. W - 1
for (i = 0; i < n; i++)
    v[i] = aux[i];
}
free(fp);
free(aux);
}

```

Invariante e corretude:

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

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.
- $O((n + R) * W)$, 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 $o(\lg n)$ indica assintoticamente menor que $\lg n$.

Eficiência de espaço:

- memória adicional da ordem de $(n + R)$.

Estabilidade:

- É estável,
 - desde que o método escolhido para ordenar cada dígito seja estável.
 - Isso porque, neste caso, em nenhuma etapa
 - elementos com a mesma chave serão invertidos.

Agora veremos uma versão semelhante do LSD radix sort

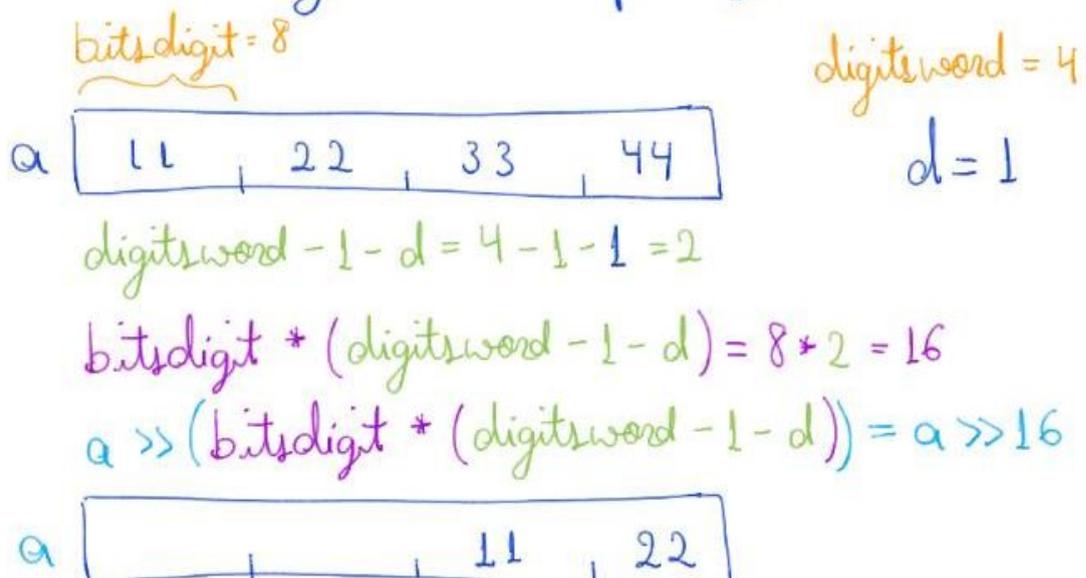
- que trabalha com vetores de inteiros,
 - sendo cada inteiro uma chave.
- Esta versão manipula explicitamente os bits das chaves
 - para obter cada dígito.

Códigos:

```
const int bitword = 32;
const int bitsdigit = 8;
const int digitword = bitword / bitsdigit;
const int Base = 1 << bitsdigit;

int digit(int a, int d)
{
    return (int)((a >> (bitsdigit * (digitword - 1 - d))) & (Base - 1));
}
```

- Calcular dígito com operações em bits



$$\text{Base} = 1 \ll \text{bitsdigit} = 1 \ll 8$$

$$\text{Base} \quad \boxed{0 \dots 0, 0 \dots 0, 0 \dots 01, 0 \dots 0}$$

$$(\text{Base} - 1) \quad \boxed{0 \dots 0, 0 \dots 0, 0 \dots 00, 1 \dots 1}$$

$$a \gg (\text{bits digit} * (\text{digitword} - 1 - d)) \& (\text{Base} - 1)$$

$$a \quad \boxed{0 \dots 0, 0 \dots 0, 0 \dots 0, 22}$$

```
int digit2(int a, int d)
{
    return (int)(a / exp2(bitsdigit * (digitword - 1 - d))) % Base;
}
```

```
void LSDradixSort(int v[], int n)
{
    int r, i, d;
    int *fp, *aux;
    fp = malloc((Base + 1) * sizeof(int));
    aux = malloc(n * sizeof(int));

    for (d = digitword - 1; d >= 0; d--)
    {
        for (r = 0; r <= Base; r++)
            fp[r] = 0;
        for (i = 0; i < n; i++)
        {
            r = digit(v[i], d);
            fp[r + 1] += 1;
        }
        // agora fp[r] é a frequência de r-1
        for (r = 1; r <= Base; 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 (i = 0; i < n; ++i)
        {
            r = digit(v[i], d);
            aux[fp[r]] = v[i];
            fp[r]++; // *
        }
        // aux[0..n-1] está em ordem crescente considerando
```

```

// apenas os dígitos entre d .. digitword - 1
for (i = 0; i < n; i++)
    v[i] = aux[i];
}
free(fp);
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.
- $O((n + \text{Base}) * \text{digitword})$, sendo:
 - n o número de chaves.
 - Base o número de valores que cada dígito pode assumir.
 - Note que, sendo bitsdigit o número de bits que cada dígito tem,
 - $\text{Base} = 2^{\text{bitsdigit}}$.
 - digitword o número de dígitos em cada chave.
 - Note que, sendo bitword o número de bits em cada chave,
 - $\text{digitword} = \text{bitword} / \text{bitsdigit}$.
- Se os dígitos forem pequenos e as chaves grandes
 - a complexidade de tempo pode superar assintoticamente $(n \lg n)$.
- Já se os dígitos forem grandes em relação às chaves
 - a eficiência tende a $O(n)$.
- Por exemplo, tome $n = 1 \text{ bilhão} = 10^9$.
 - Se bitsdigit = 1 (Base = $2^1 = 2$) e bitword = 32,
 - temos digitword = $32 / 1 = 32$,
 - que é maior que $\lg 10^9 \leq \lg 2^{(10*3)} \approx 30$.
 - Se bitsdigit = 16 (Base = $2^{16} = 65536$) e bitword = 32,
 - temos digitword = $32 / 16 = 2$,
 - que é uma pequena constante muito menor que $\lg 10^9$.

Eficiência de espaço:

- memória adicional da ordem de $(n + \text{Base})$.

Estabilidade:

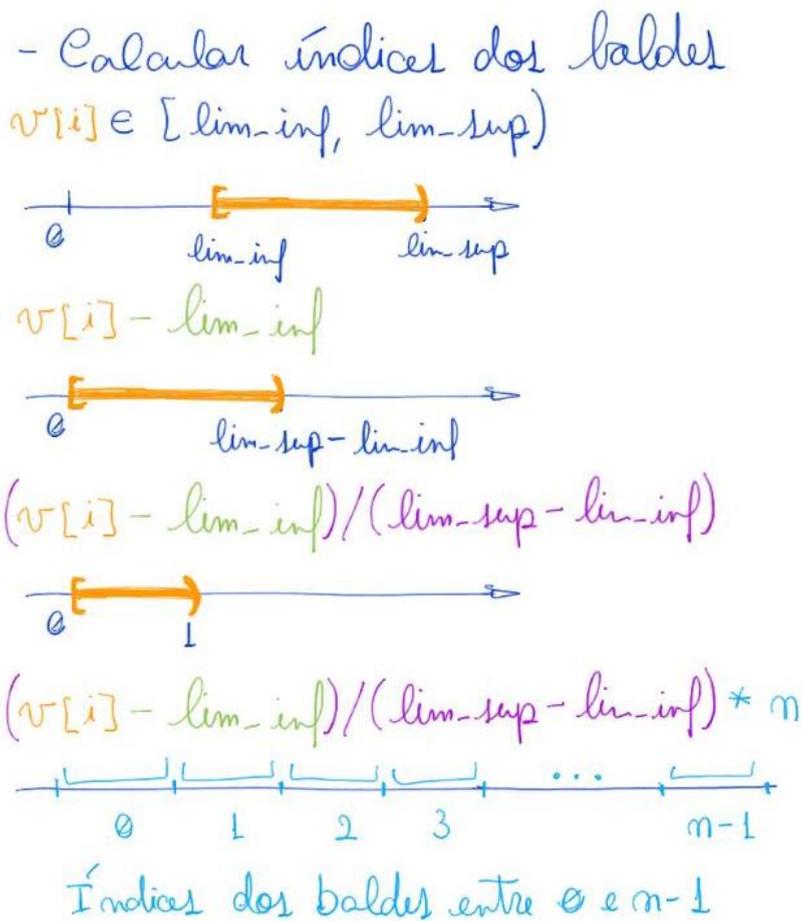
- É estável.

Bucket sort

Este é um método eficiente para ordenar um conjunto com n elementos

- distribuídos uniformemente num intervalo de tamanho k.
- Primeiro, dividimos o intervalo de tamanho k em n baldes

- e associamos com cada balde
 - uma fração de valor igual a k/n .
- Então colocamos cada elemento em seu respectivo balde.
- Em seguida ordenamos os elementos de cada balde,
 - o que deve levar tempo constante,
 - já que cada balde deve ter um número pequeno de elementos,
 - uma vez que estes vieram de uma distribuição uniforme.
- Por fim, percorremos os baldes em ordem
 - e os elementos de cada balde, também em ordem,
 - copiando eles de volta para o vetor original.



Códigos:

```
#define lim_inf 0
#define lim_sup 1

void bucketSort(double v[], int n)
{
    int i, j, k;
    Celula *p, *nova, *morta;
    Celula **b = (Celula **)malloc(n * sizeof(Celula *));
```

```

// inicializando cada balde com uma lista com um nó cabeça
for (j = 0; j < n; j++)
{
    b[j] = (Celula *)malloc(sizeof(Celula));
    b[j]->prox = NULL;
}
// coloca cada elemento no balde correspondente
for (i = 0; i < n; i++)
{
    j = (int)(v[i] * n);
    // j = (int)((double)(v[i] - lim_inf) / (lim_sup - lim_inf) * n);
    p = b[j];
    // já insere o elemento na ordem correta dentro do balde
    while (p->prox != NULL && p->prox->chave < v[i])
        p = p->prox;
    nova = (Celula *)malloc(sizeof(Celula));
    nova->chave = v[i];
    nova->prox = p->prox;
    p->prox = nova;
}
//põe os elementos dos baldes de volta no vetor
i = 0;
for (j = 0; j < n; j++)
{
    p = b[j]->prox;
    while (p != NULL)
    {
        v[i++] = p->chave;
        p = p->prox;
    }
}
for (j = 0; j < n; j++)
{
    p = b[j];
    while (p != NULL)
    {
        morta = p;
        p = p->prox;
        free(morta);
    }
}
free(b);
}

```

Eficiência de tempo esperado é $O(n)$,

- apenas se as chaves forem uniformemente distribuídas.
- Sem essa hipótese o tempo de pior caso do algoritmo é $O(n^2)$,
 - pois muitos elementos podem se acumular num mesmo balde,

- e a ordenação deste pode levar tempo quadrático,
 - dependendo do método utilizado.
- Diante disso, vale a pena usar um método de ordenação $O(n \lg n)$,
 - para ordenar cada balde?
 - Em geral não, pois são esperados poucos elementos por balde,
 - e métodos como insertionSort são melhores para n pequeno.

Eficiência de espaço:

- memória adicional da ordem de n ,
 - pois são utilizados n baldes e
 - n células, uma para cada elemento.

Estabilidade:

- A estabilidade depende da implementação da ordenação intrabalde.
- O código que estudamos não é estável, mas uma pequena modificação
 - na inserção/ordenação intra balde pode corrigir isso.
 - Que modificação é essa?