

MC-202 — Unidade 14

Ordenação em tempo linear

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2023

Ordenação em $O(n \lg n)$

Vimos dois algoritmos de ordenação $O(n \lg n)$:

- MergeSort
- HeapSort

E o caso médio do QuickSort é $O(n \lg n)$...

Dá para fazer melhor que $O(n \lg n)$?

- **Não**, se considerarmos algoritmos que usam comparação
 - algoritmos que precisam saber apenas se $v[i] < v[j]$
 - algoritmos de ordenação “genéricos”
- **Sim**, se não usarmos comparações
 - algoritmos que sabem a estrutura da chave
 - ex: a chave é um número inteiro com **32 bits**

Algoritmos baseados em comparações

Um algoritmo de ordenação baseado em comparações

- recebe uma sequência de n valores
- precisa decidir qual das $n!$ permutações é a correta
- usando apenas comparações entre pares de elementos
 - pode ordenar `int`, `float`, `strings`, `structs`, etc...
 - desde que tenha uma função de comparação

Os algoritmos que vimos são baseados em comparações

Quantas comparações um tal algoritmo precisa fazer no mínimo para ordenar o vetor?

- Quão rápido pode ser um algoritmo baseado em comparações?

Árvore de decisão

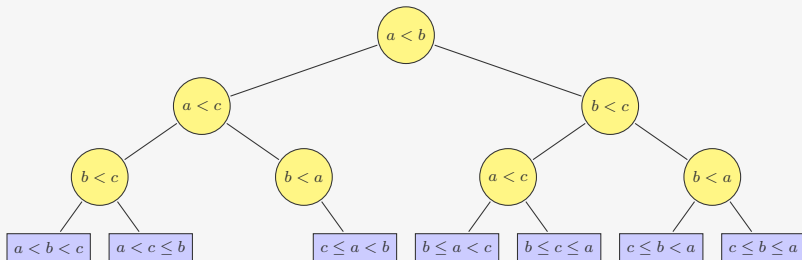
Um algoritmo baseado em comparações:

- compara dois elementos $v[i]$ e $v[j]$
- e toma decisões diferentes dependendo do resultado

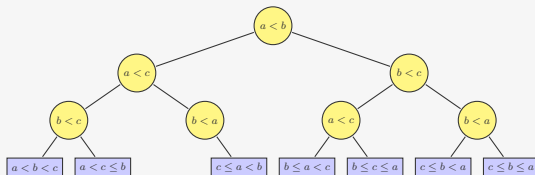
Podemos pensar a execução do algoritmo como uma árvore:

- cada nó interno representa um teste se $v[i] < v[j]$
- **subárvore esquerda**: comparações feitas se for **verdade**
- **subárvore direita**: comparações feitas se for **falso**

Ex: SelectionSort de (a, b, c)



Árvore de decisão



Qual é a altura mínima h de uma árvore de decisão?

- Temos pelo menos $n!$ folhas (uma para cada permutação)
- Uma árvore de altura h tem no máximo 2^h folhas
- Seja l o número de folhas, temos que $n! \leq l \leq 2^h$

Ou seja,

$$h \geq \lg(n!) \geq \lg \left(\frac{n}{e} \right)^n = n (\lg n - \lg e)$$

Não dá para fazer um algoritmo baseado em comparações melhor do que $O(n \lg n)$

Ordenação em tempo linear

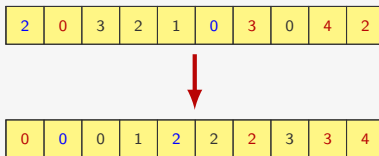
Quando falamos de ordenação em tempo linear:

- São algoritmos que não são baseados em comparação
- Eles não servem para qualquer tipo de chave
 - Já que não usamos apenas comparações
 - Eles usam a estrutura da chave de alguma forma
 - Ex: número inteiros entre 0 e $R-1$
 - Ex: número inteiros de 32 bits

Veremos dois algoritmos de ordenação em tempo linear

Ordenação Estável

Um algoritmo de ordenação é *estável* se ele mantém a ordem relativa original dos itens com chaves de ordenação duplicadas



Algoritmos estáveis:

- InsertionSort, BubbleSort e MergeSort

Algoritmos não-estáveis:

- SelectionSort, Quicksort e Heapsort

Ordenação por Contagem — CountingSort

Se temos números inteiros entre 0 e $R-1$:

- Contamos o número de ocorrências de cada número
 - Fazemos um histograma dos números
- Colocamos os números na posição correta (de maneira estável)

Ex: Se queremos ordenar 2, 0, 3, 2, 1, 0, 3, 0, 4, 2

- Temos três ocorrências do número 0
- Temos uma ocorrência do número 1
- Temos três ocorrências do número 2
- Temos duas ocorrências do número 3
- Temos uma ocorrência do número 4

Basta colocar, em ordem:

- três 0's, um 1, três 2's, dois 3's e um 4
- Ou seja, 0, 0, 0, 1, 2, 2, 2, 3, 3, 4

CountingSort

```
1 #define MAX 10000
2 #define R 5
3
4 int aux[MAX];
5
6 void countingsort(int *v, int l, int r) {
7     int i, count[R + 1];
8     for (i = 0; i <= R; i++)
9         count[i] = 0;
10    for (i = l; i <= r; i++)
11        count[v[i] + 1]++;
12    for (i = 1; i <= R; i++)
13        count[i] += count[i-1];
14    for (i = l; i <= r; i++) {
15        aux[count[v[i]]] = v[i];
16        count[v[i]]++;
17    }
18    for (i = l; i <= r; i++)
19        v[i] = aux[i-l];
20 }
```

count

aux

v

CountingSort

```
1 #define MAX 10000
2 #define R 5
3
4 int aux[MAX];
5
6 void countingsort(int *v, int l, int r) {
7     int i, count[R + 1];
8     for (i = 0; i <= R; i++)
9         count[i] = 0;
10    for (i = l; i <= r; i++)
11        count[v[i] + 1]++;
12    for (i = 1; i <= R; i++)
13        count[i] += count[i-1];
14    for (i = l; i <= r; i++) {
15        aux[count[v[i]]] = v[i];
16        count[v[i]]++;
17    }
18    for (i = l; i <= r; i++)
19        v[i] = aux[i-l];
20 }
```

count

aux

v

CountingSort

```
1 #define MAX 10000
2 #define R 5
3
4 int aux[MAX];
5
6 void countingsort(int *v, int l, int r) {
7     int i, count[R + 1];
8     for (i = 0; i <= R; i++)
9         count[i] = 0;
10    for (i = l; i <= r; i++)
11        count[v[i] + 1]++;
12    for (i = 1; i <= R; i++)
13        count[i] += count[i-1];
14    for (i = l; i <= r; i++) {
15        aux[count[v[i]]] = v[i];
16        count[v[i]]++;
17    }
18    for (i = l; i <= r; i++)
19        v[i] = aux[i-l];
20 }
```

count

aux

v

CountingSort

```
1 #define MAX 10000
2 #define R 5
3
4 int aux[MAX];
5
6 void countingsort(int *v, int l, int r) {
7     int i, count[R + 1];
8     for (i = 0; i <= R; i++)
9         count[i] = 0;
10    for (i = l; i <= r; i++)
11        count[v[i] + 1]++;
12    for (i = 1; i <= R; i++)
13        count[i] += count[i-1];
14    for (i = l; i <= r; i++) {
15        aux[count[v[i]]] = v[i];
16        count[v[i]]++;
17    }
18    for (i = l; i <= r; i++)
19        v[i] = aux[i-l];
20 }
```

count	3	4	7	9	10	10
-------	---	---	---	---	----	----

aux	0	0	0	1	2	2	2	3	3	4
-----	---	---	---	---	---	---	---	---	---	---

v	0	0	0	1	2	2	2	3	3	4
---	---	---	---	---	---	---	---	---	---	---

Ordenando datas

Ideia: ordena por ano, depois por mês e depois por dia

30/09/2017	26/06/ 2000	19/01/2010
01/12/2005	03/04/ 2000	03/04/2000
09/09/2003	13/12/ 2000	10/04/2004
26/06/2000	21/09/ 2002	01/04/2014
19/01/2010	09/09/ 2003	28/05/2007
03/04/2000	10/04/ 2004	26/06/2000
01/04/2014	01/12/ 2005	01/06/2006
13/12/2000	17/07/ 2005	17/07/2005
21/09/2002	 01/06/ 2006	 27/08/2014
28/05/2007	28/05/ 2007	21/09/2002
27/08/2014	19/01/ 2010	09/09/2003
10/04/2004	01/04/ 2014	30/09/2017
01/06/2006	27/08/ 2014	28/10/2014
17/07/2005	28/10/ 2014	13/12/2000
28/10/2014	30/09/ 2017	01/12/2005

Ordenando datas

Ordena por dia, depois por mês e depois por ano

30/09/2017	01/12/2005	19/01/2010	03/04/2000
01/12/2005	01/04/2014	01/04/2014	26/06/2000
09/09/2003	01/06/2006	03/04/2000	13/12/2000
26/06/2000	03/04/2000	10/04/2004	21/09/2002
19/01/2010	09/09/2003	28/05/2007	09/09/2003
03/04/2000	10/04/2004	01/06/2006	10/04/2004
01/04/2014	13/12/2000	26/06/2000	17/07/2005
13/12/2000	17/07/2005	17/07/2005	01/12/2005
21/09/2002	19/01/2010	27/08/2014	01/06/2006
28/05/2007	21/09/2002	09/09/2003	28/05/2007
27/08/2014	26/06/2000	21/09/2002	19/01/2010
10/04/2004	27/08/2014	30/09/2017	01/04/2014
01/06/2006	28/05/2007	28/10/2014	27/08/2014
17/07/2005	28/10/2014	01/12/2005	28/10/2014
28/10/2014	30/09/2017	13/12/2000	30/09/2017

Funciona se o algoritmo for estável!

RadixSort

Ideia:

- Usar o mesmo princípio da ordenação de datas
- Ordenar números comparando sequências de bits
 - do menos significativo para o mais significativo
 - usando ordenação estável
- Radix é o mesmo que a base do sistema numeral

Vamos ordenar números inteiros de 4 bytes, i.e., 32 bits

- Poderia ser números maiores
- Nosso radix será 256 (1 byte)
 - Poderia ser outro número
 - É melhor escolher uma potência de 2
- Precisaremos extrair o i -ésimo byte do número
 - contando da direita para esquerda

Deslocamento de bits

Desloca para a esquerda (\ll) — “multiplica por 2^k ”

- Ex: $00000101 \ll 3 == 00101000$ ($5 \ll 3 == 40$)
- Ex: $01000101 \ll 3 == 00101000$ ($69 \ll 3 == 40$)

Desloca para a direita (\gg) — divide por 2^k

- Ex: $00101000 \gg 3 == 00000101$ ($40 \gg 3 == 5$)
- Ex: $00101011 \gg 3 == 00000101$ ($43 \gg 3 == 5$)

Bits e Bytes

```
1 #define bitsword 32
2
3 #define bitsbyte 8
4
5 #define bytesword 4
6
7 #define R 256
8
9 #define digit(N,D) (((N) >> (D)*bitsbyte) % R)
```

RadixSort

```
1 void radixsort(int *v, int l, int r) {
2     int i, w, count[R+1];
3     for (w = 0; w < bytesword; w++) {
4         for (i = 0; i <= R; i++)
5             count[i] = 0;
6         for (i = l; i <= r; i++)
7             count[digit(v[i], w) + 1]++;
8         for (i = 1; i <= R; i++)
9             count[i] += count[i-1];
10        for (i = l; i <= r; i++) {
11            aux[count[digit(v[i], w)]] = v[i];
12            count[digit(v[i], w)]++;
13        }
14        for (i = l; i <= r; i++)
15            v[i] = aux[i-l];
16    }
17 }
```

CountingSort no
 w -ésimo dígito

Tempo: $O(\text{bytesword} \cdot (R + n))$

Se a chave tem k bits, tempo: $O\left(\frac{k}{\lg R}(n + R)\right)$

Comparação do algoritmos

Limite de tempo=0.05s

- `bubblesort` ordena 5.000 números em 0.034s
- `insertionsort_v4` ordena 20.000 números em 0.038s
- `quicksort_mdt_v3` ordena 640.000 números em 0.05s
- `radixsort` ordena 2.560.000 números em 0.04s

Comparação Assintótica

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Memória
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
QuickSort	$O(n \lg n)$	$O(n \lg n)$	$O(n^2)$	$O(n)$
MergeSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
HeapSort	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(1)$
RadixSort	$O((n + R) \frac{k}{\lg R})$	$O((n + R) \frac{k}{\lg R})$	$O((n + R) \frac{k}{\lg R})$	$O(R)$

onde k é o número de bits na chave de ordenação

Lembrando que RadixSort não pode ser usado sempre

- não é baseado em comparações

Conclusão

Escolher entre dois algoritmos de mesmo tempo tem resultado na prática

- Ex: `bubblesort` vs. `insertionsort`
- Ex: `heapsort` vs. `mergesort`

Otimizar o código dos algoritmos pode trazer boas melhoras

- Ex: `insertionsort` vs. `insertionsort_v2`
- Ex: `quicksort` vs. `quicksort_mdt`

No fim do dia, o que mais faz diferença é o tempo assintótico

- Se n for grande...
 - Se n for pequeno, o overhead pode não compensar
- Ex: `insertionsort_v2` vs. `heapsort`
- Ex: `quicksort_mdt` vs. `radixsort`

Exercício

Mostre um esquema para tornar qualquer algoritmo em um algoritmo estável. Quanto espaço e tempo adicional é necessário para o seu esquema?