DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Estrutura de Dados

Métodos de Ordenação sem comparação de chaves

Professores: Luiz Chaimowicz e Raquel Prates

Até Agora...

Todos os algoritmos de ordenação com base em:

- Comparação
- Troca

- Ordenação Comparativa
- Existem métodos que não requerem comparações de chaves. Por exemplo:
  - Counting Sort
  - Bucket Sort
  - Radix Sort

Ordenação por Contagem

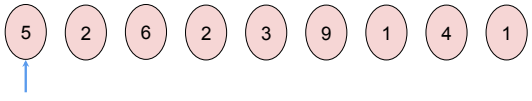
Considere o seguinte problema:

- Dada uma lista/vetor de elementos entre [0, max)
- Max é conhecido antecipadamente

Ordenação por Contagem

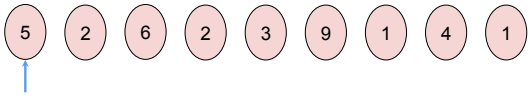
- Dada uma lista/vetor de elementos entre [0, max)
  - Max é conhecido antecipadamente
    - max = 9

Contagem



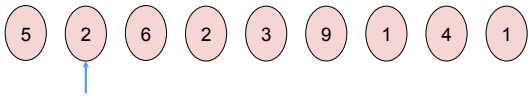
0	1	2	3	4	5	6	7	8	9

Contagem



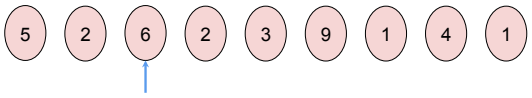
0	1	2	3	4	5	6	7	8	9
					1				

Contagem



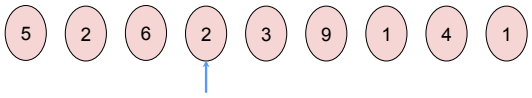
0	1	2	3	4	5	6	7	8	9
		1			1				

Contagem



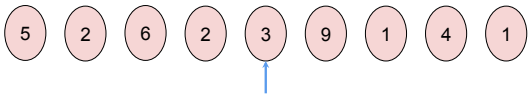
0	1	2	3	4	5	6	7	8	9
		1			1	1			

Contagem



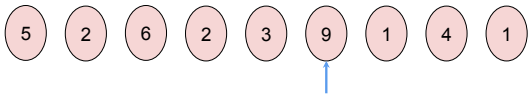
0	1	2	3	4	5	6	7	8	9
		2			1	1			

Contagem



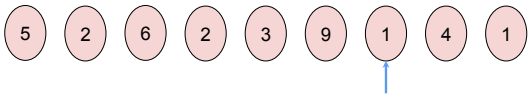
0	1	2	3	4	5	6	7	8	9
		2	1		1	1			

Contagem



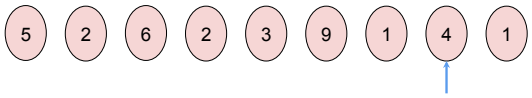
0	1	2	3	4	5	6	7	8	9
		2	1		1	1			1

Contagem



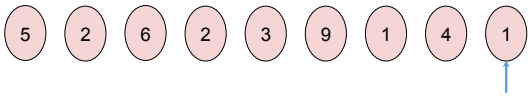
0	1	2	3	4	5	6	7	8	9
	1	2	1		1	1			1

Contagem



0	1	2	3	4	5	6	7	8	9
	1	2	1	1	1	1			1

Contagem



0	1	2	3	4	5	6	7	8	9
	2	2	1	1	1	1			1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	2	2	1	1	1	1	0	0	1

- Como ordenar?

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	2	2	1	1	1	1	0	0	1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	2	2	1	1	1	1	0	0	1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	1	2	1	1	1	1	0	0	1

1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	0	2	1	1	1	1	0	0	1

1 1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	0	2	1	1	1	1	0	0	1

1 1

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	1	1	0	0	1

1 1 2

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	0	0	1	1	1	1	0	0	1

1 1 2 2

Sabendo das contagens

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

1 1 2 2 3 4 5 6 9

CountingSort (Ordenação por Contagem)

- Cria um vetor de contadores
- Conta todos os elementos
- Só funciona com limites de entrada bem definidos
  - Sem saber a entrada não temos como alocar a contagem de forma eficaz
  - Custo de memória

CountingSort

```
void countingsort(int *values, int n, int max) {
  int *counts = (int *) calloc(max, sizeof(int));
  int i, j;
  for (i = 0; i < n; i++)
       counts[values[i]]++;
  i = 0;
  for(j = 0; j < max; j++)
       while(counts[j] > 0) {
         values[i++] = j;
       counts[j]--;
    }
  free(counts);
}
```


Complexidade

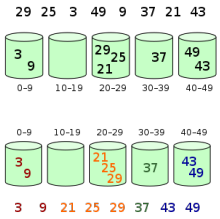
- Considere *n* o número de elementos e *k* o valor do maior elemento (max)
- Tempo
  - Uma passagem pelo vetor para contagem: O(n)
  - Uma passagem pelo "contador" para imprimir os elementos: O(k) [com O(n) impressões]
  - Total: O(n+k)
- Espaço
  - Memória extra: O(k)

Counting Sort - Considerações

- Vantagem
  - Ordenação em O(n)
- Desvantagens
  - Muita memória extra
  - Na maioria dos casos é impossível saber a escala de elementos a priori

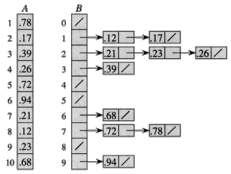
Bucket Sort

- Separa os elementos em *buckets* (baldes) de tamanho menor
- Ordena separadamente cada um dos baldes usando um dos algoritmos tradicionais



Bucket Sort

- Alternativamente, pode-se implementar os *buckets* como listas encadeadas e já inserir de forma ordenada em cada lista. Ou utilizar Insertion sort em cada bucket.
- Ex: itens uniformemente distribuídos no intervalo [0,1]



Exemplo: https://www.cs.usfca.edu/~galles/visualization/BucketSort.html

Complexidade

- Considere *n* o número de elementos e *k* buckets (listas ordenadas)
- Tempo
  - Pior caso: $O(n^2)$
  - Caso médio (entrada uniformemente distribuída): $O(n + n^2/k + k)$
- Espaço
  - Memória extra: O(n+k)
    - k listas, mas com n elementos no total

Bucket Sort - Considerações

- Observações:
  - Método útil para entradas uniformemente distribuídas em um intervalo de valores. Se a entrada tiver muitos itens com valores próximos (clustering), alguns backets terão mais elementos do que a média. No pior caso, todos os elementos podem pertencer a um único bucket.
- Vantagem:
  - Ordenação em tempo quasi-linear quando k cresce
- Desvantagem:
  - Muita memória extra para armazenar os bins

Radixsort

- **Radix Sort** é uma classe de algoritmos que usa a representação binária das chaves para a ordenação.
  - Digital Sort
- Idéia Geral:
  - chaves cujo bit mais a esquerda é 0, vem antes que chaves cujo bit é 1
  - Repetindo-se isso para todos os bits de forma adequada é possível ordenar

Radixsort

- Requer a representação binária da chave
- Como extrair os bits (importante técnica de programação)
  - Formas eficientes em C: and (&) e shift(>>)
    - Ex: Bit0= x & 00000001; x >> 1;
  - Formas simples: divisão e resto
    - Considerando bits indexados de 0 a n da dir para esq, para extrair o bit i de um número X, temos **$(X / 2^i) \% 2$**

Radix Exchange Sort

- Algoritmo analisa os bits da esquerda para a direita
- Funcionamento similar ao do Quicksort, mas a partição é feita comparando-se bits ao invés de chaves
- Chamadas recursivas ordenando os subvetores pelo bit i-1

Radix Exchange Sort

```
quicksortB(int a[], int l, int r, int w) {
  int i = l, j = r;

  if (r <= l || w > 0) return;
  while (j != i) {
      while (digit(a[i], w) == 0 && (i < j)) i++;
      while (digit(a[j], w) == 1 && (j > i)) j--;

      swap(a[i], a[j]);
  }
  if (digit(a[r], w) == 0) j++;
  quicksortB(a, l, j-1, w+1);
  quicksortB(a, j, r, w+1);
 }

void sort(Item a[], int l, int r) {
   quicksortB(a, l, r, numbits - 1);
}
```

Fonte: Algorithms in C
Robert Sedgewick

Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
```

Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
```


Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
[ 011 010 000 001|110 101 111]
```


Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
[ 011 010 000 001|110 101 111]
[ 001 000|010 011]
```


Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
[ 011 010 000 001|110 101 111]
[ 001 000|010 011]
[ 000|001]
          [010|011]
```


Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
[ 011 010 000 001|110 101 111]
[ 001 000|010 011]
[ 000|001]
          [010|011]
                  [101|110 111]
                      [110|111]
```


Exemplo

```
[   3   2   5   6   2   0   7]
[ 011 010 101 110 001 000 111]
[ 011 010 101 110 001 000 111]
[ 011 010 000 001|110 101 111]
[ 001 000|010 011]
[ 000|001]
          [010|011]
                  [101|110 111]
                      [110|111]
[   0   1   2   3   5   6   7]
```

Complexidade

- Considere *n* o número de elementos e *k* o número de bits de cada chave
- O *radix exchange sort* faz *k* passagens pelo vetor de *n* elementos: O(*n.k*) comparações de bits.
  - Considerando k = log(n), temos O(*n.log(n)*) comparações de bits
- A eficiência do algoritmo depende do custo para se extrair os bits...

Radix sort

- Vantagens
  - Pode ser implementado in place
  - Complexidade linear se k=O(1), i.e. valor max = 2^k
- Desvantagens
  - Aplicável apenas a domínios com representação binária (já quicksort, por ex, pode ordenar qq conj onde elementos são comparáveis)
  - Método não estável (pode ser resolvido com memória auxiliar extra)
- Observações
  - Atualmente mais usado para *strings* binárias e inteiros com representações de tamanho fixo
  - Há diferentes implementações
    - Ordem dos dígitos: MSD, LSD
    - Trie-based: build a trie and traverse in pre order

The topic of the efficiency of radix sort compared to other sorting algorithms is somewhat tricky and subject to quite a lot of misunderstandings. Whether radix sort is equally efficient, less efficient or more efficient than the best comparison-based algorithms depends on the details of the assumptions made. Radix sort efficiency is O(d·n) for n keys which have d or fewer digits. Sometimes d is presented as a constant, which would make radix sort better (for sufficiently large n) than the best comparison-based sorting algorithms, which are all O(n·log(n)) number of comparisons needed. However, in general d cannot be considered a constant. In particular, under the common (but sometimes implicit) assumption that all keys are distinct, then d must be at least of the order of log(n), which gives at best (with densely packed keys) a time complexity O(n·log(n)). That would seem to make radix sort at most equally efficient as the best comparison-based sorts (and worse if keys are much longer than log(n)).

The counter argument is the comparison-based algorithms are measured in number of comparisons, not actual time complexity. Under some assumptions the comparisons will be constant time on average, under others they will not. Comparisons of randomly-generated keys takes constant time on average, as keys differ on the very first bit in half the cases, and differ on the second bit in half of the remaining half, and so on, resulting in an average of two bits that need to be compared. In a sorting algorithm the first comparisons made satisfies the randomness condition, but as the sort progresses the keys compared are clearly not randomly chosen anymore. For example, consider a bottom-up merge sort. The first pass will compare pairs of random keys, but the last pass will compare keys that are very close in the sorting order.

The deciding factor is how the keys are distributed. The best case for radix sort is that they are taken as consecutive bit patterns. This will make the keys as short as they can be, still assuming they are distinct. This makes radix sort O(n·log(n)), but the comparison based sorts will not be as efficient, as the comparisons will not be constant time under this assumption. If we instead assume that the keys are bit patterns of length k·log(n) for a constant k > 1 and base 2 log, and that they are uniformly random, then radix sort will still be O(n·log(n)), but so will the comparison based sorts, as the "extra" length makes even the keys that are consecutive in the sorted result differ enough that comparisons are constant time on average. If keys are longer than O(log(n)), but random, then radix sort will be inferior. There are many other assumptions that can be made as well, and most require careful study to make a correct comparison.