Técnicas e Análise de Algoritmos

Busca e Ordenação - Parte 04

Professor: **Jeremias Moreira Gomes**

E-mail: jeremias.gomes@idp.edu.br

Introdução

Introdução

- Definições de Ordenação
    - Parcial e Total
- Características de um algoritmo de ordenação
- Ordenações Quadráticas
- Ordenações Linearítmicas

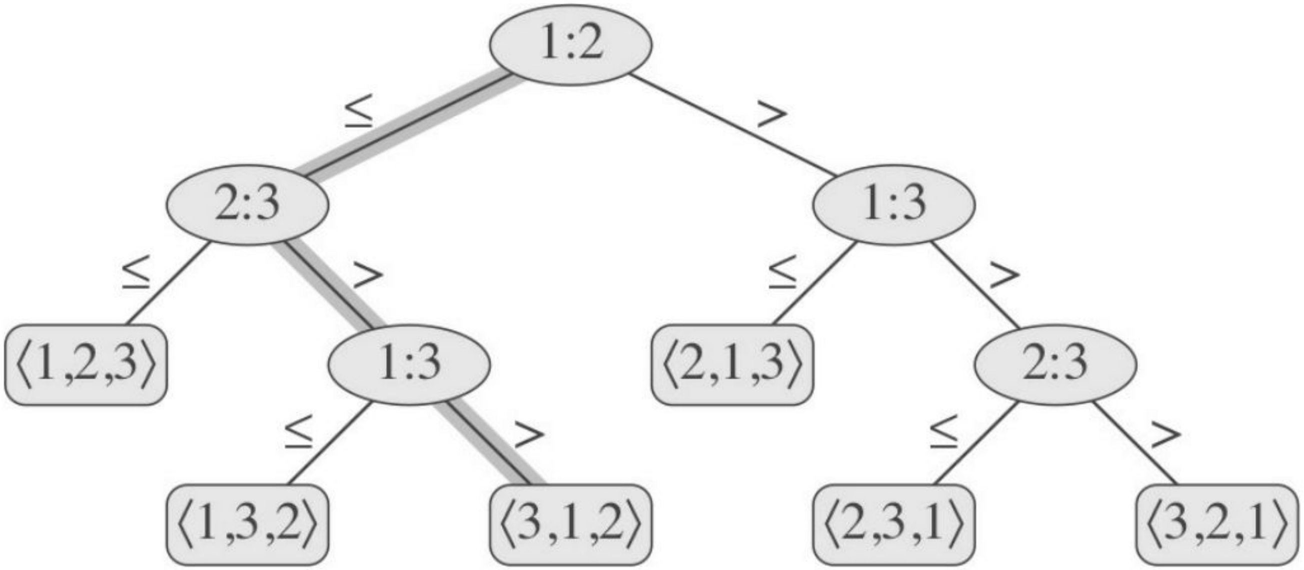
Algoritmos Baseados em Comparação

Algoritmos Baseados em Comparação

- Todos os algoritmos (de ordenação) vistos até agora, são algoritmos baseados na comparação entre pares
  - São literalmente chamados de algoritmos de ordenação por comparação (ou classificação por comparação)
- Alguns algoritmos apresentados possuem a ordem de complexidade quadrática e outros linearítmicas, mas **existe um limite inferior para um algoritmo de ordenação**?

Algoritmos Baseados em Comparação

- Algoritmos baseados em comparação podem ser vistos como uma árvore de decisão



Exemplo de um insertion sort com 3 elementos

Algoritmos Baseados em Comparação

- No caso da árvore de decisão, cada folha é uma permutação diferente possível para aquela quantidade de elementos
  - **O total de folhas é n!**
  - Todo algoritmo de ordenação correto tem que ser capaz de encontrar um caminho nessa árvore que resulte na ordenação do conjunto de elementos da raiz até a folha correta

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (1/3)
    - Se uma árvore tem altura h com f folhas acessíveis a partir de uma ordenação por comparação:

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (2/3)
    - Cada uma das n! permutações é uma folha diferente, então
        - $n! \leq f$
        - Uma árvore binária de altura h não tem mais do que 2^h folhas: $2^h \geq f$

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (3/3)
    - Então:

$$n! \leq f \leq 2^h$$

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (3/3)
    - Então:

$$n! \leq f \leq 2^h$$

$$2^h \geq n!$$

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (3/3)
    - Então:

$$n! \leq f \leq 2^h$$

$$2^h \geq n!$$

$$h \geq lg(n!)$$

Algoritmos Baseados em Comparação

- A complexidade de um algoritmo de ordenação baseado em comparações é o tamanho da árvore da raiz até a sua respectiva folha (3/3)
  - Então:

$$n! \leq f \leq 2^h$$

$$2^h \geq n!$$

$$h \geq lg(n!)$$

$$= \Omega(nlg(n))$$

Ordenação em Tempo Linear

Ordenação em Tempo Linear

- Apesar de os algoritmos de ordenação baseada em comparação terem um limite inferior, eles não são a única classe de algoritmos de ordenação
- Existe uma classe de algoritmos que não utilizam essa base de comparação, mas são muito mais eficientes, se as condições para a utilização do algoritmo forem satisfeitas

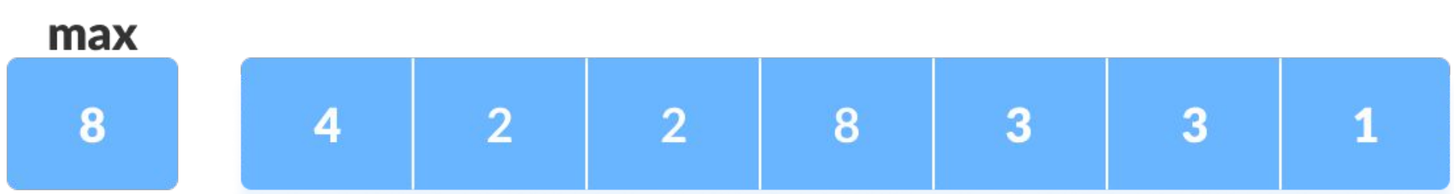
Counting Sort

Counting Sort

- Algoritmo baseado na contagem dos elementos a serem ordenados
- A condição básica para utilização do algoritmo são duas:
  - Conhecer os valores máximos e mínimos da lista
  - O intervalo dado por esses limites ser praticável em memória
    - Ter correspondência de índices para todo o domínio
    - Ser possível alocar uma lista do tamanho desse intervalo

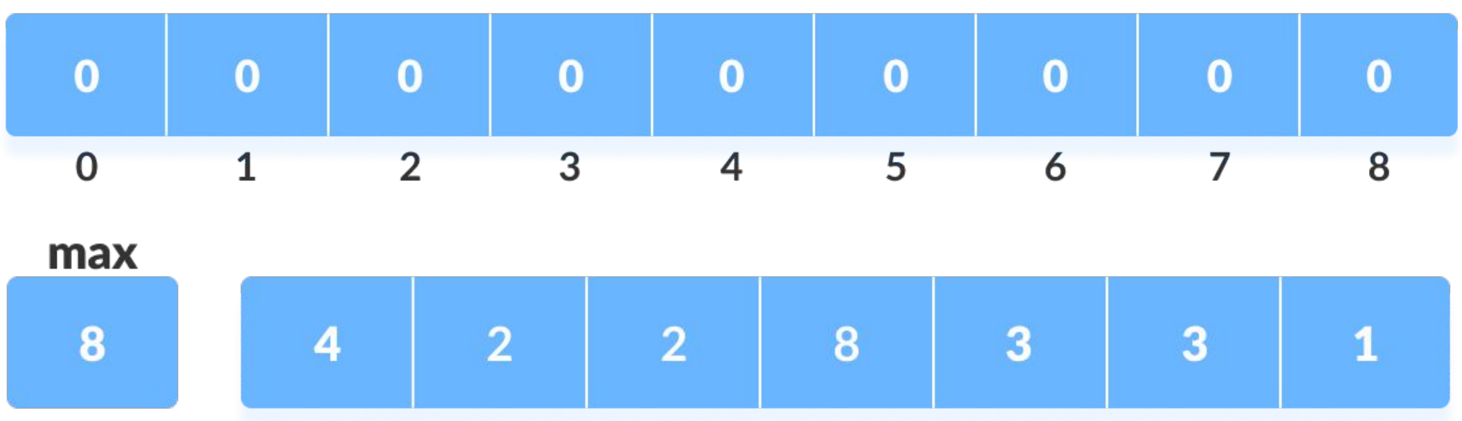
Counting Sort

- Assim, para entender a ideia inicial, vamos abordar o problema problema de ordenação
  - A lista só possui valores inteiros positivos (> 0)
  - Sabemos o maior valor dessa lista



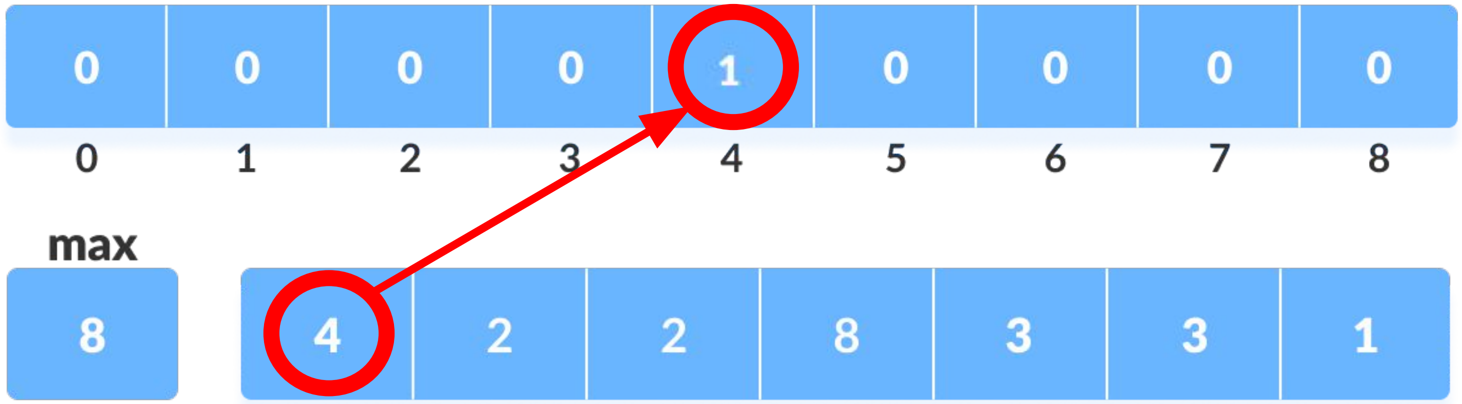
Counting Sort

- O primeiro passo, é alocar um novo vetor auxiliar com o tamanho do intervalo de valores possíveis (nesse caso, de 1 a 8) e inicializados com valor zero



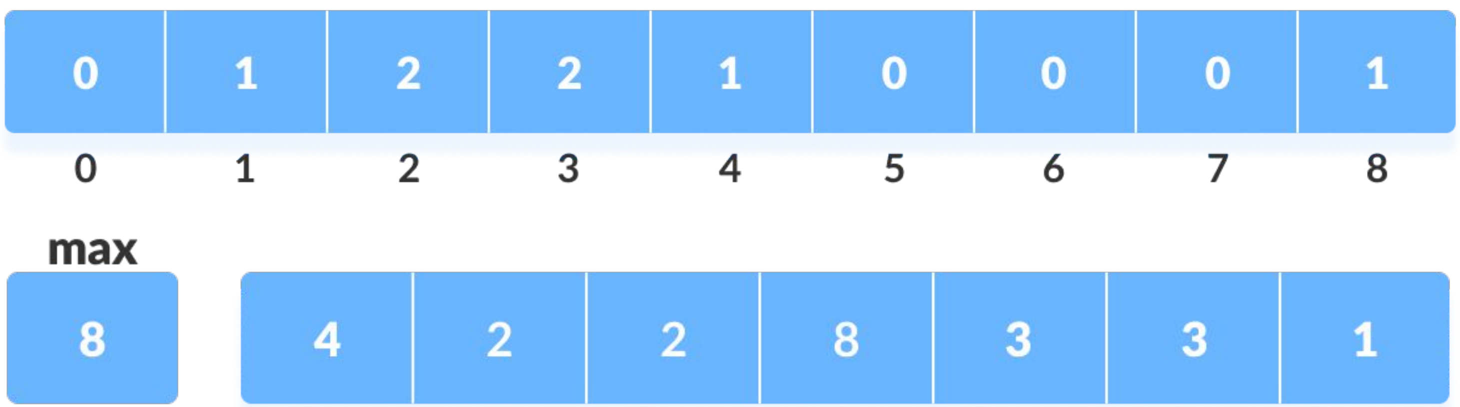
Counting Sort

- O próximo passo é, para cada elemento da lista, adicionar uma unidade no vetor auxiliar
  - `auxiliar[vetor[0]] += 1`



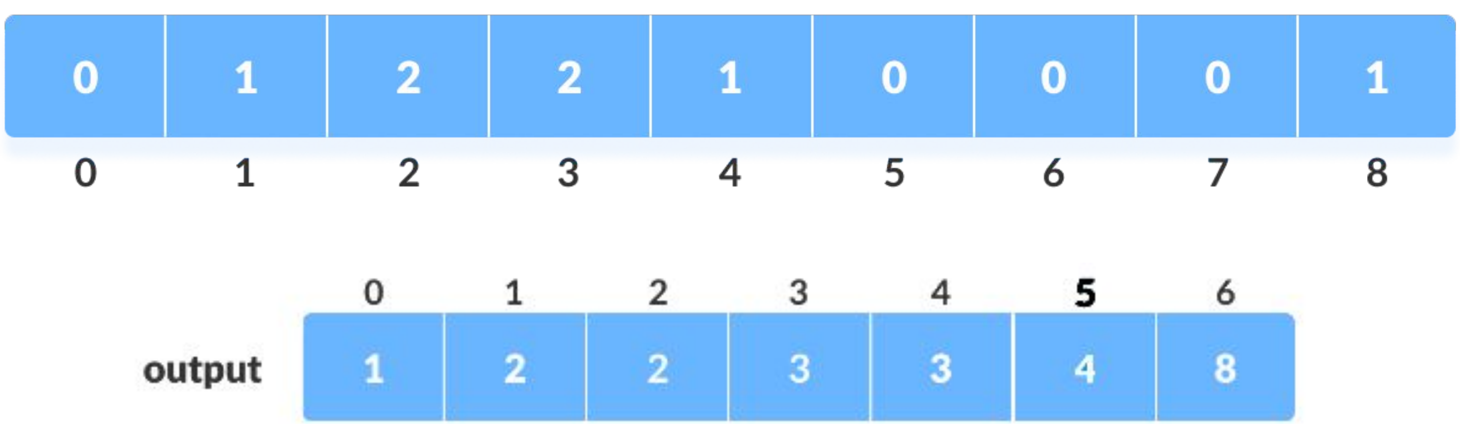
Counting Sort

- Continuar incrementando a contagem até percorrer todos os elementos da lista
  - `auxiliar[vetor[i]] += 1`



Counting Sort

- Após esse processo, o vetor auxiliar possui as quantidades de cada elemento (do seu índice) no vetor original
- Assim, basta escrever essas quantidades na ordem de cada índice



Counting Sort

```
void couting_sort(vector<int> &v, int maior) {
    vector<int> auxiliar(maior + 1, 0);
    for (int i = 0; i < v.size(); i++) {
        auxiliar[v[i]]++;
    }

    for (int i = 1, j = 0; i < auxiliar.size(); i++) {
        for (int k = 0; k < auxiliar[i]; k++) {
            v[j] = i;
            j++;
        }
    }
}
```

Counting Sort

- O algoritmo percorre n de elementos da lista uma vez contando
- Em seguida, ele percorre o intervalo k de elementos uma vez
- Para cada k, ele percorre a quantidade de vezes que esse elemento aparece, porém apesar de serem dois laços aninhados, a soma de todas as iterações desses laços internos é sempre igual ao número de elementos do intervalo mais a quantidade de n de elementos, fazendo com que a complexidade seja
  - `O(n + n + k) = O(2n + k) = O(n + k)`

Counting Sort

- O valor de k é importante, porque ele dita a quantidade de espaço auxiliar utilizado, então a complexidade de espaço é O(k)
  - Imagine aplicar esse algoritmo no seguinte vetor
    - [7, 2, 2, 1, 8, 238794619378641, 3, 10]
    - Por isso o valor de k deve ser razoável em relação ao uso de memória

Counting Sort

- Sobre as propriedades do algoritmo
    - Não é in-place
        - Precisa de memória extra para ordenar os elementos
    - É estável
        - Apesar de realizar contagem, a primeira ocorrência é a que aparece na resposta, possibilitando replicar essa propriedade no algoritmo

Radix Sort

Radix Sort

- Radix Sort ou Ordenação Digital (nome quase não visto), é um algoritmo de ordenação utilizado por máquinas de ordenação de cartões antigas
    - Esse ordenador era programado de acordo com a coluna do cartão (cartões possuíam 80 colunas)
- Essa ordenação utiliza uma ideia não intuitiva de **ordenação por dígito**

Radix Sort - Exemplo

329

457

657

839

436

720

355

Radix Sort - Exemplo

329

457

657

839

436

720

355

Por qual dígito ordenar primeiro?

Radix Sort - Exemplo

329	720
457	355
657	436
839	457
436	657
720	329
355	839

Radix Sort - Exemplo

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

Radix Sort - Exemplo

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

Radix Sort - Exemplo

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

- No caso desse algoritmo, a ordenação por dígitos é feita por meio de algum outro algoritmo de ordenação
    - O principal requisito é que este algoritmo **seja estável**
    - Assim, como são conhecidos os limites da quantidade de dígitos disponíveis, o counting sort se torna um candidato ideal para ser utilizado

Radix Sort

```
{
    for (int d = 0; d < nr_digitos; d++) {
        counting_sort(tab, d);
    }
}
```

Radix Sort

- A complexidade do Radix Sort depende diretamente do algoritmo de ordenação utilizado
  - Usando como auxiliar o Counting Sort, e tendo cada dígito na faixa de 0 a k - 1
  - A passagem em cada dígito leva O(n + k) e possuindo k dígitos, fica O(d(n + k))
  - Assim, como d é constante e pequeno, a ordenação é linear

Conclusão