Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Algoritmos e Estruturas de Dados I

Ordenação (parte II)

Mirtha Lina Fernández Venero
Sala 529-2, Bloco A
mirtha.lina@ufabc.edu.br
http://professor.ufabc.edu.br/~mirtha.lina/aedi.html

20 de abril de 2019

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Introdução aos Algoritmos de ordenação

Algoritmos elementares de ordenação, baseados em comparações, de custo $O(n^2)$ no caso pior

- Ordenação por intercâmbio (*bubble sort*)
- Ordenação por seleção (*selection sort*)
- Ordenação por inserção(*insertion sort*)

Algoritmos de ordenação bons

- Shellsort

Algoritmos de ordenação ótimos

- Mergesort, Heapsort, Quicksort

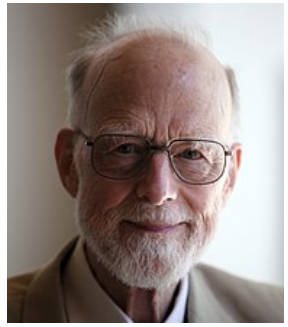
Limite assintótico da ordenação baseada em comparações é $\Omega(n \log n)$, i.e. qualquer algoritmo da ordenação baseado em comparações usa no mínimo $n \log n$ comparações no caso pior

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Sumário

Quicksort - Ordenação por Intercâmbio

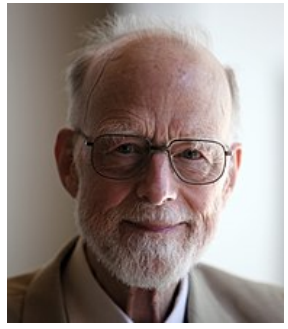


Inventado por **Sir Charles Antony Richard Hoare**, prêmio Turing em 1980 pelas contribuições no projeto de linguagens de programação

- *"my billion-dollar mistake... was the invention of the null reference in* 1965"
- *"The real value of tests is not that they detect bugs in the code but that they detect inadequacies in the methods, concentration, and skills of those who design and produce the code."*
- *"What is the central core of computer science?... It is the art of designing efficient and elegant methods of getting a computer to solve problems, theoretical or practical, small or large, simple or complex."*

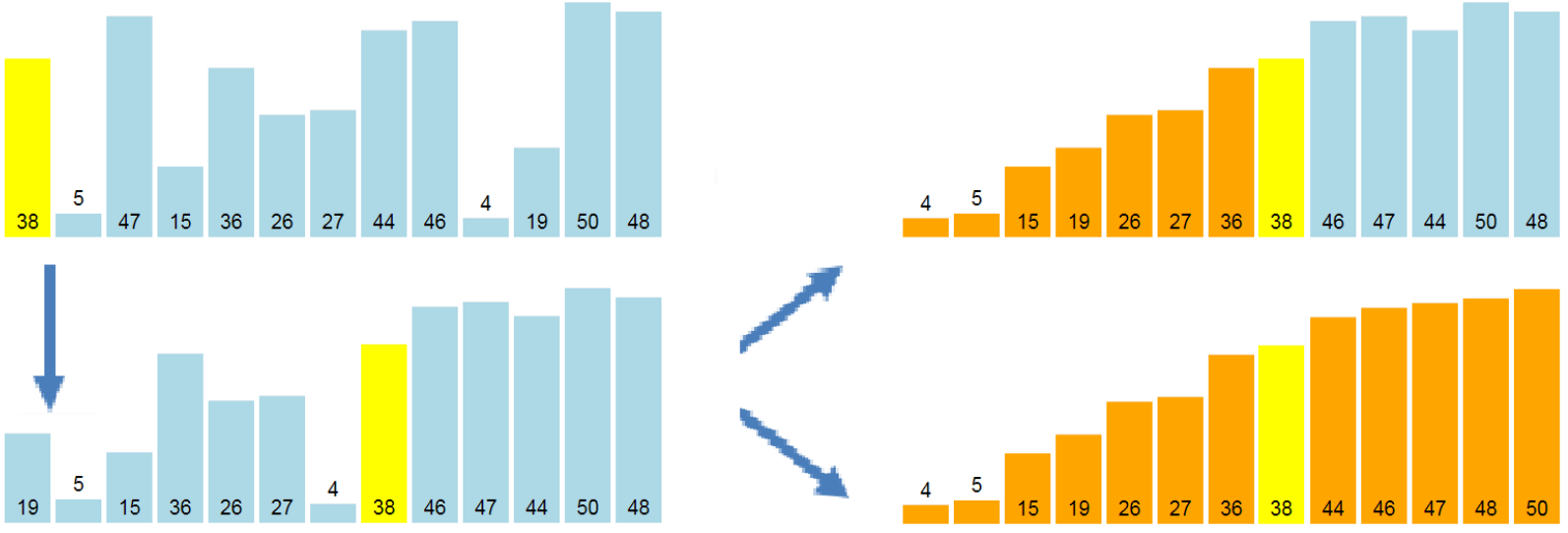
Tony Hoare Quotes

Quicksort - Ordenação por Intercâmbio



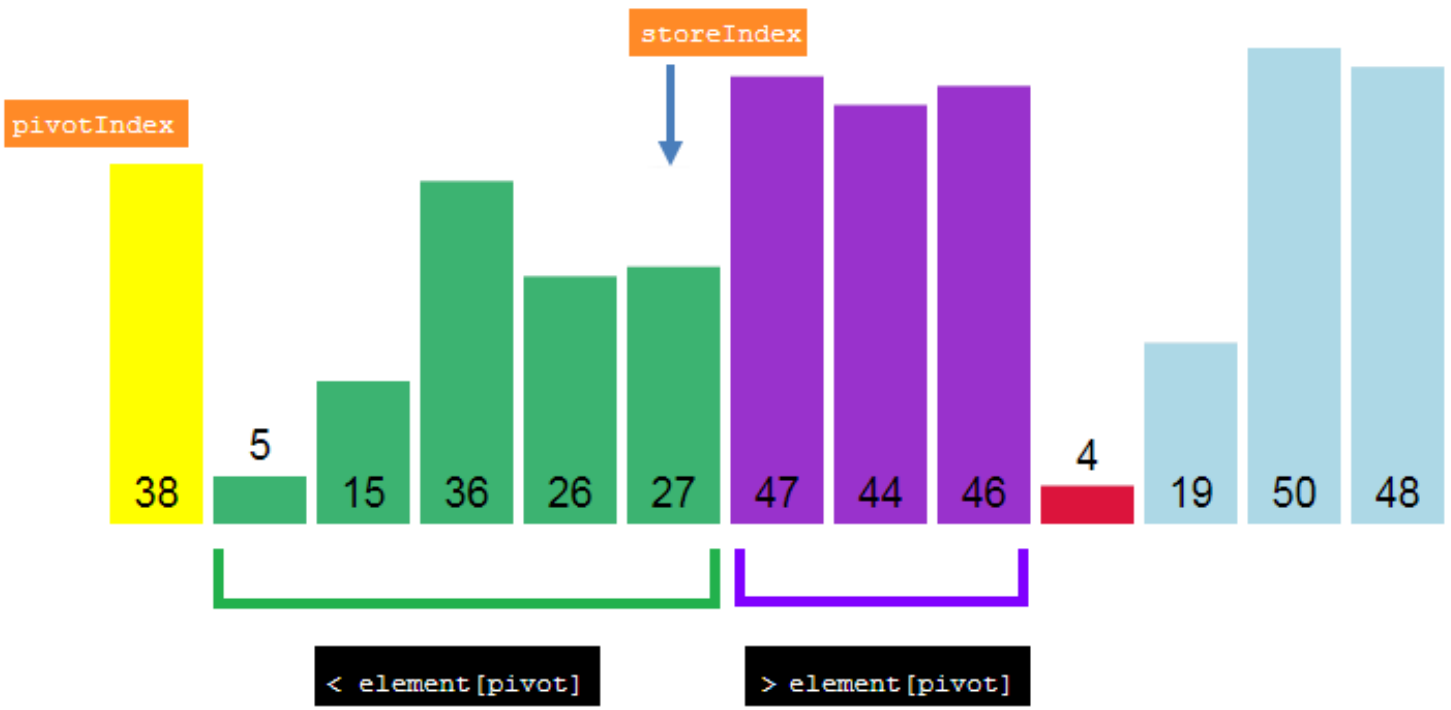
Inventado por Tony Hoare em 1959 sendo estudante e publicado em 1961

- refinado e analisado por Robert Sedgewick em 1976
- exemplo clássico de algoritmo recursivo de divisão e conquista
- a conquista é feita de forma inteligente particionando a sequência antes de dividir sem precisar memória auxiliar



Particionamento

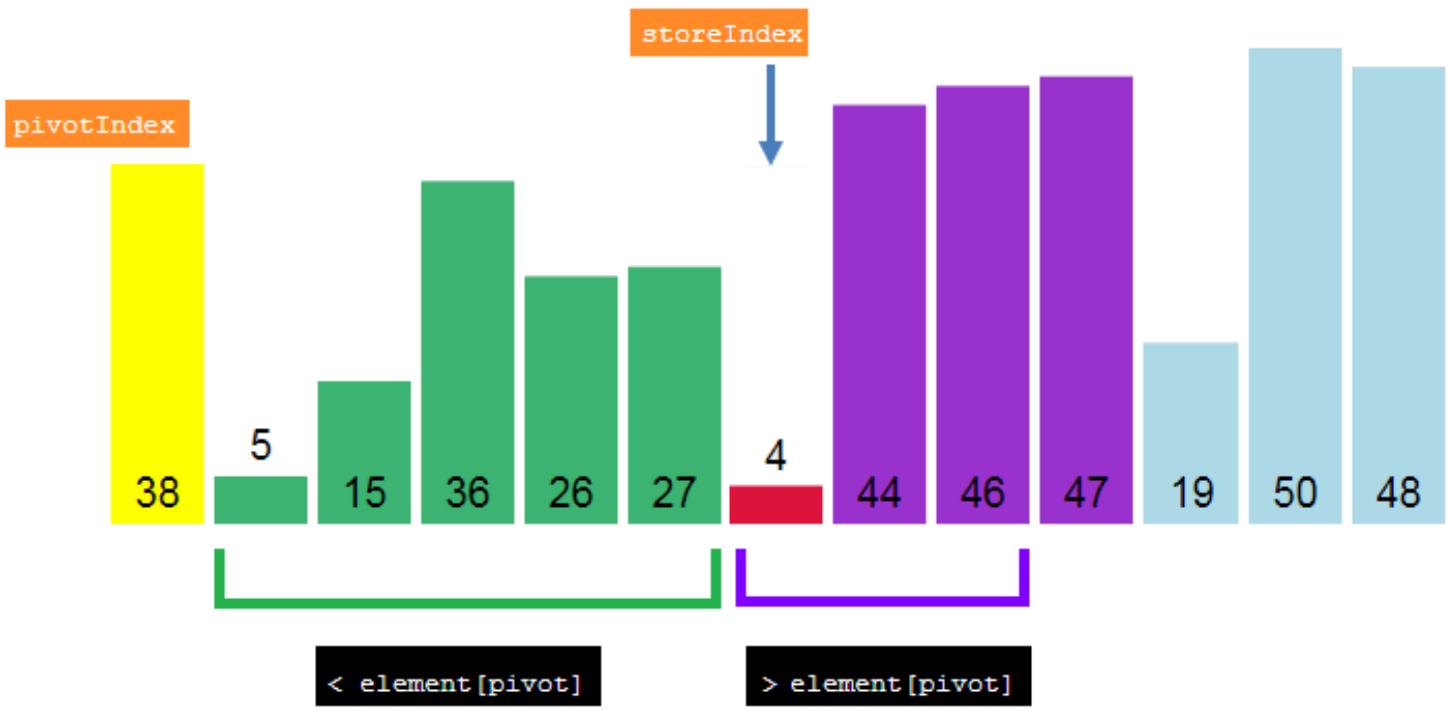
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

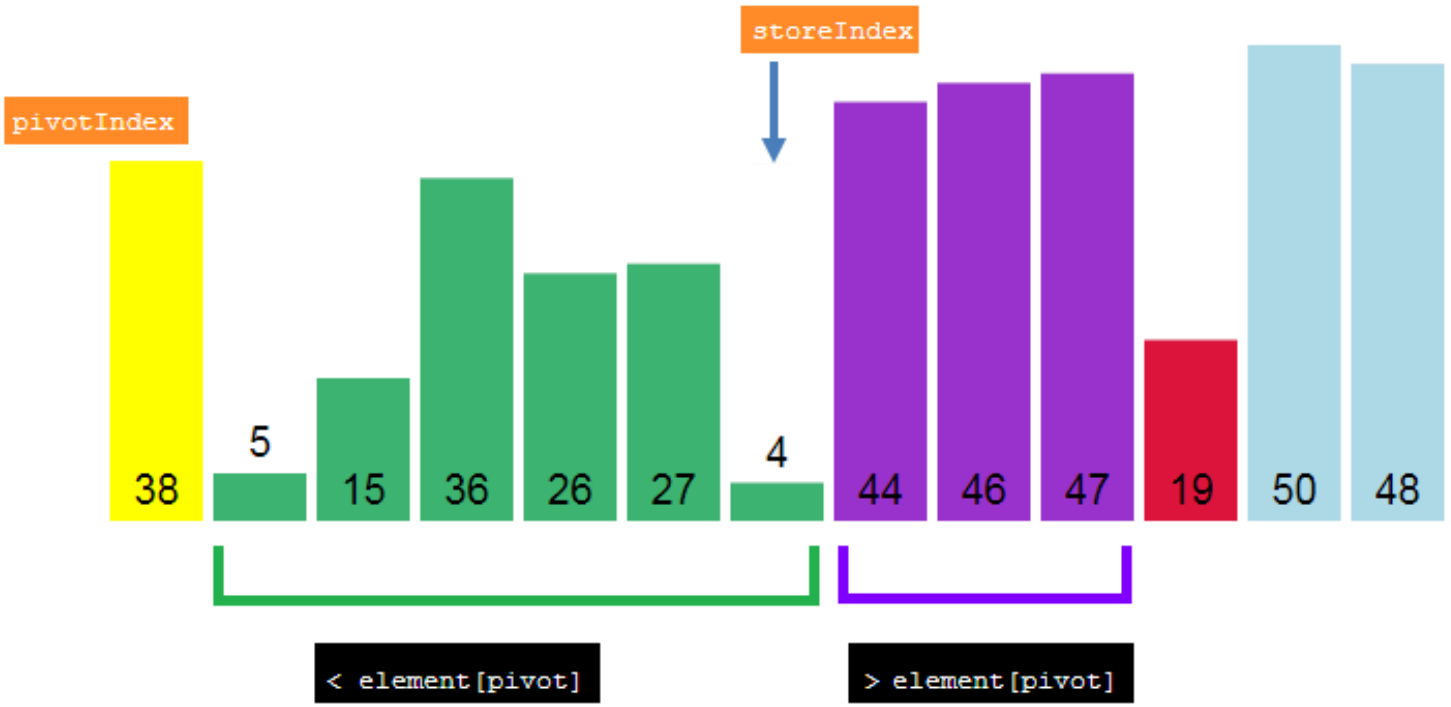
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

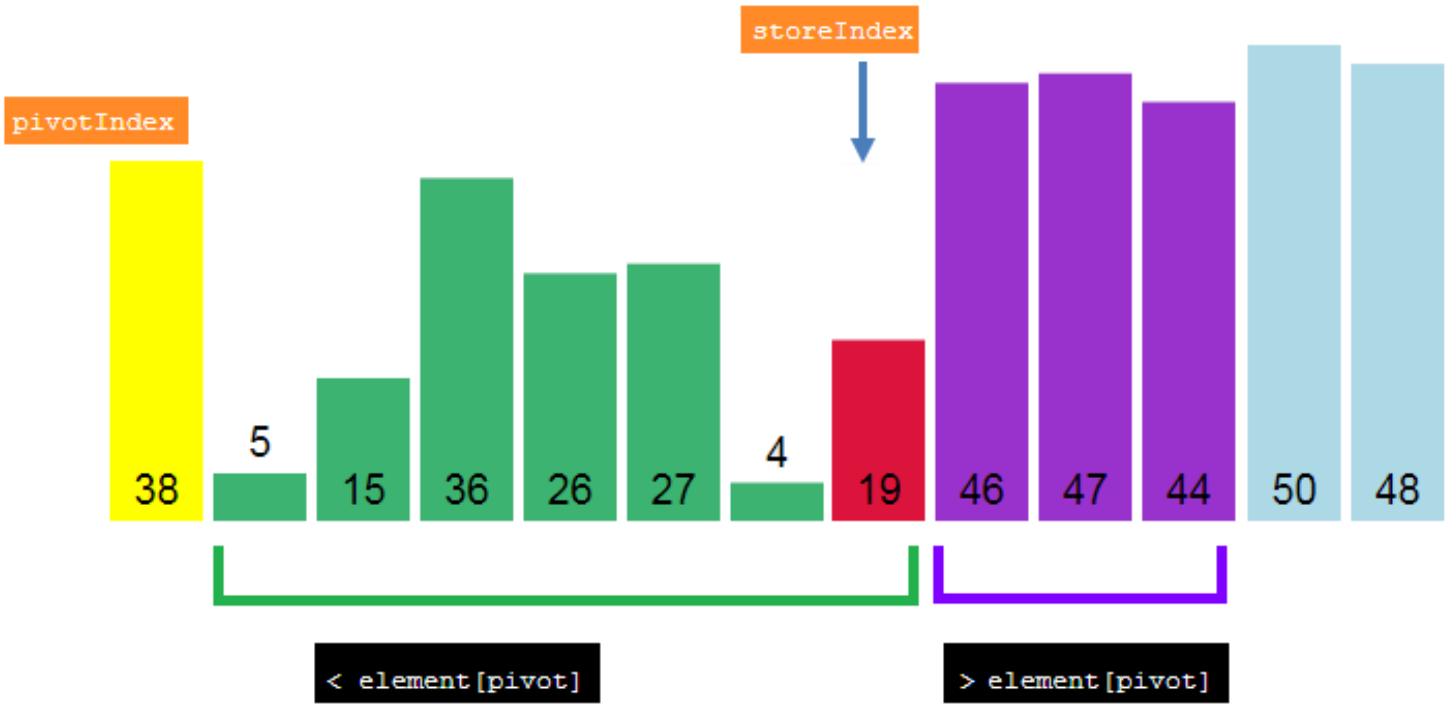
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

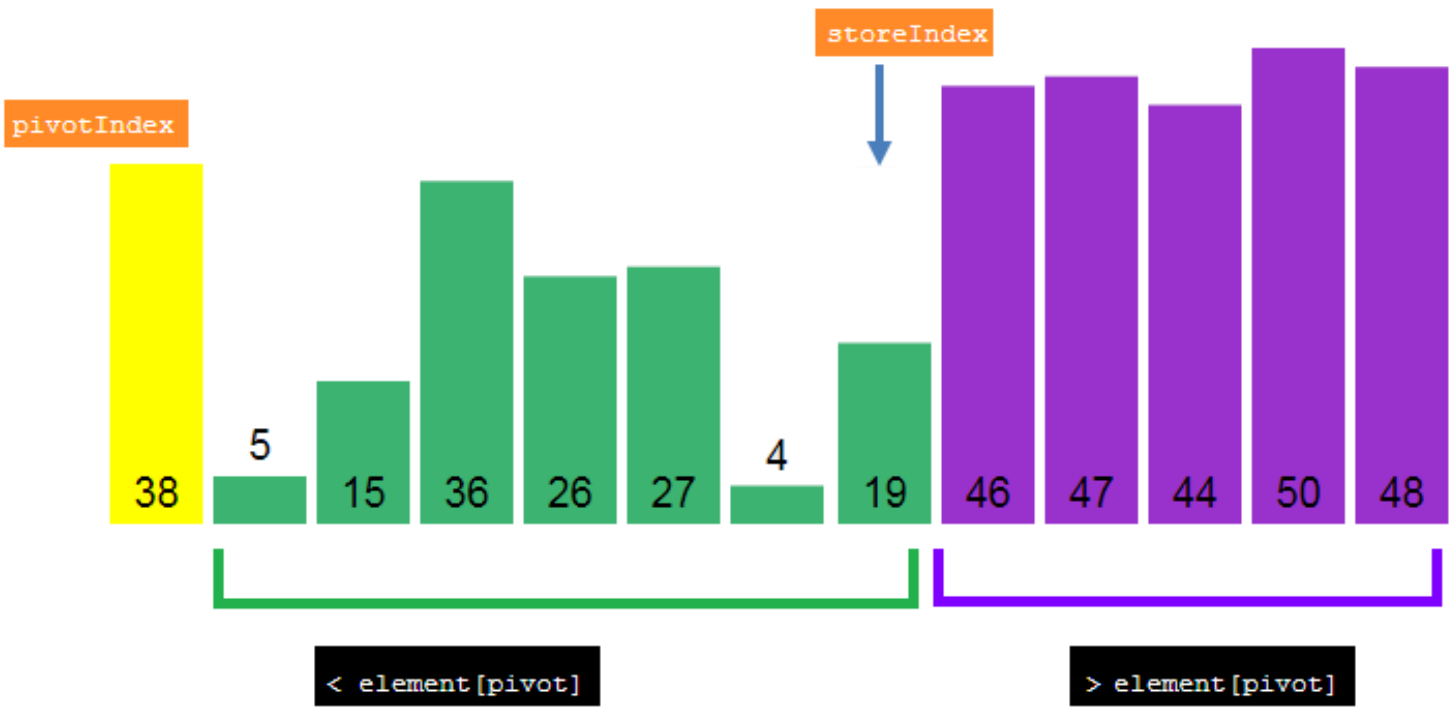
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

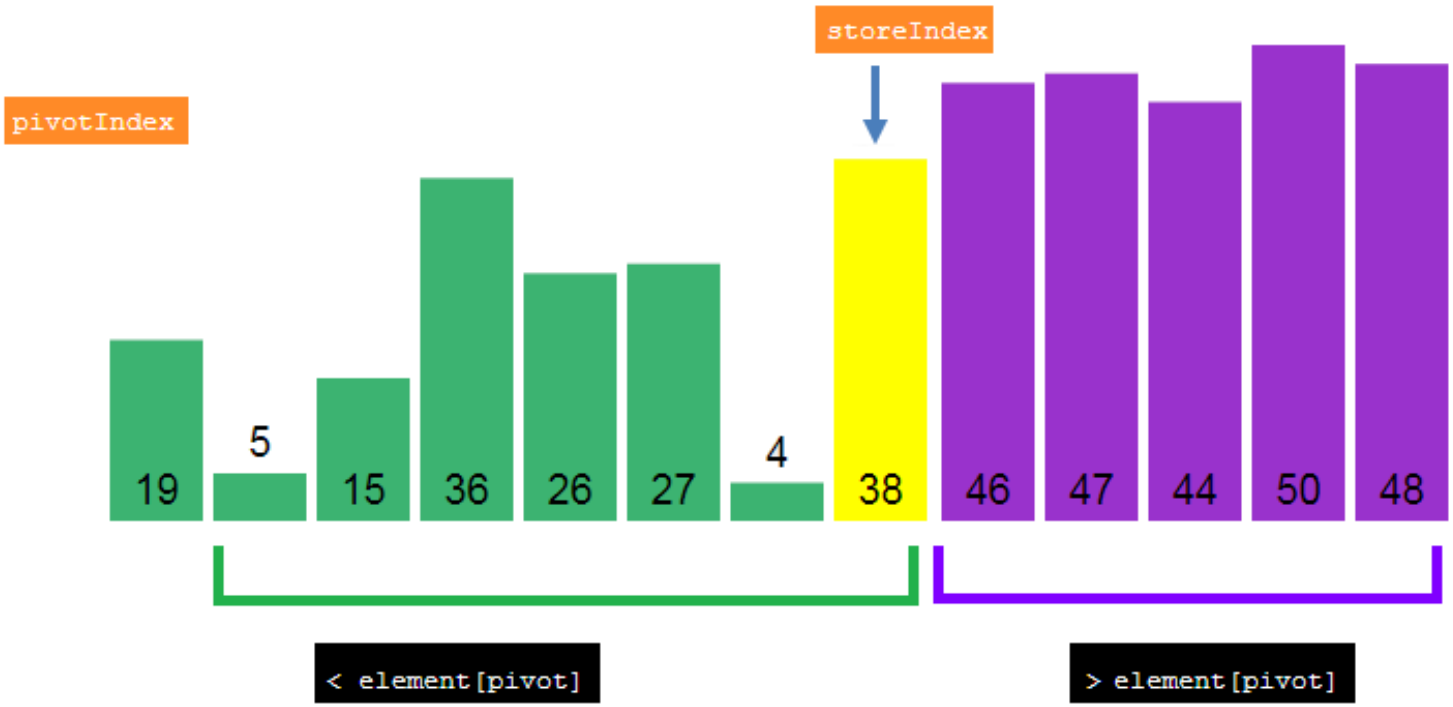
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

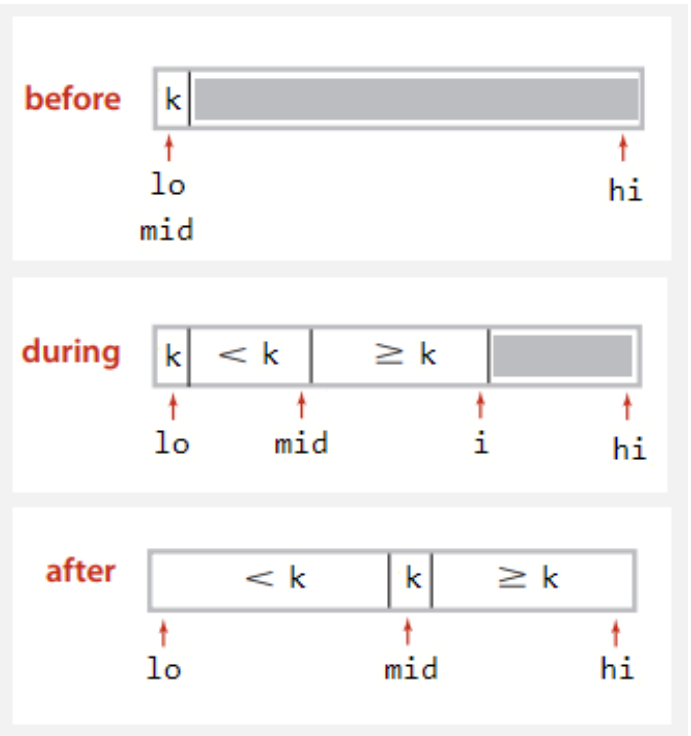
- escolher um elemento pivô de forma tal que no final ele seja colocado na posição certa
- separar os elementos menores e maiores ou iguais ao pivô usando trocas



https://visualgo.net/en/sorting

Particionamento

1. escolher um elemento arbitrário `k` para ser o item de particionamento (pivô)
2. colocar `k` na primeira posição e usar e usar uma variável auxiliar para marcar sua posição (`mid`)
3. percorrer o restante do arranjo até acharmos um elemento menor estrito do que `k`
4. Se acharmos esse elemento na posição `i`, incrementar a posição marcada do pivô e trocar os elementos em `i` e `mid`
5. no final trocar o pivô com o elemento em `mid`



https://algs4.cs.princeton.edu/lectures/23Quicksort-2x2.pdf

Particionamento

- a parte mais importante do Quicksort, tem custo linear
- a escolha do elemento pivô influencia como é feito o particionamento e o desempenho do Quicksort
- pode ser usado um elemento aleatório ou a mediana duma amostra do vetor (e.g. primeiro, último, meio)
- pode mudar a ordem relativa de registros com a mesma chave (portanto não é estável) **Exemplo?**

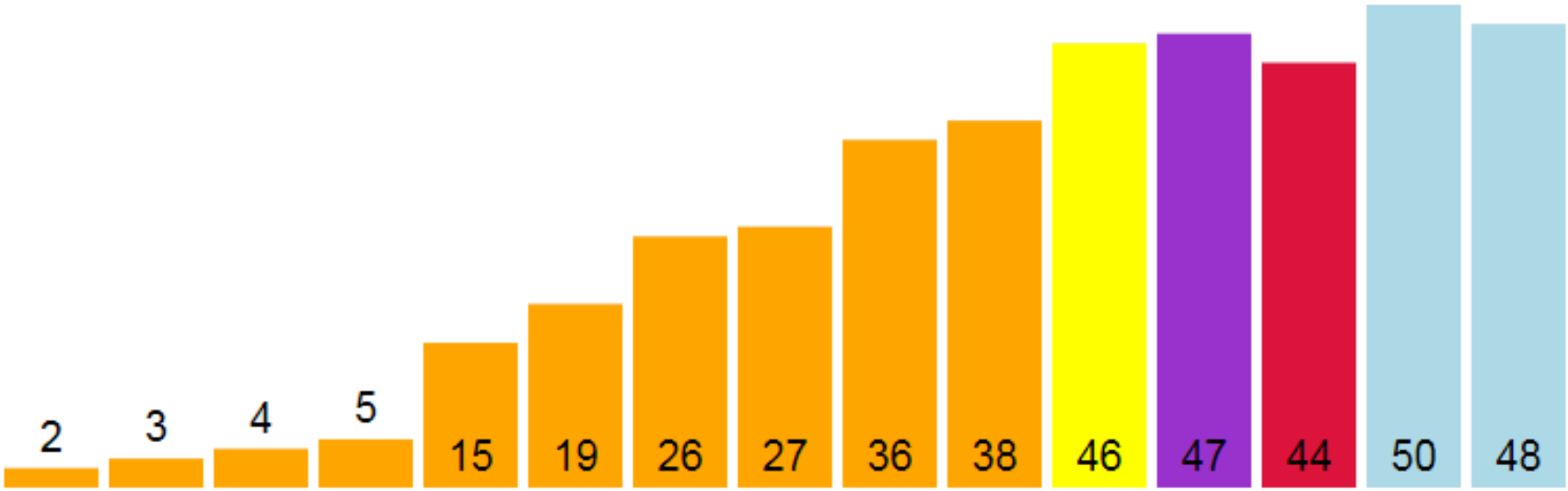
Quicksort: Particiona e divide

```
int QuickSort(int v[], int lo, int hi) {
  int comp = 0;
  if (lo < hi) {
    int mid = partition(v, lo, hi, &comp);
    comp += QuickSort(v, lo, mid - 1);
    comp += QuickSort(v, mid + 1, hi);
  }
  return comp;
}

long int quickSort(int v[], int n) {
  return QuickSort(v, 0, n - 1);
}
```

https://repl.it/@mirthalina/optimalSortingAlgorithms

Quicksort: Particiona e divide




```
for each (unsorted) partition
set first element as pivot
  storeIndex = pivotIndex + 1
  for i = pivotIndex + 1 to rightmostIndex
    if element[i] < element[pivot]
      swap(i, storeIndex); storeIndex++
  swap(pivot, storeIndex - 1)
```

https://visualgo.net/en/sorting

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Análise do Quicksort

- **Caso melhor**: o vetor é sempre particionado na metade

$$T_{best}(n) = 2 * T_{best}(n/2) + O(n) = \Theta(n \log n)$$

- **Caso pior**: A sequência já está ordenada (crescente ou decrescente)

$$T_{worst}(n) = T_{worst}(n-1) + O(n) = \Theta(n^2)$$

Como evitar o caso pior? Inicialmente embaralhar o arranjo!

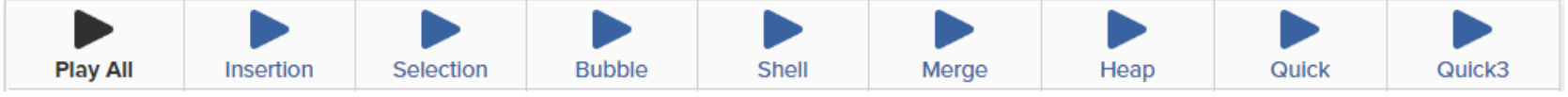
- Para sub-arranjos pequenos, usar ordenação por inserção. Nesses casos também é possível ignorar a chamada e ordenar por inserção somente uma vez no final
- primeiro ordenar a partição menor para garantir que o algoritmo seja *in-place*, i.e. $O(\log n)$ chamadas na pilha

Mergesort

- ótimo ($\Theta(n \log n)$) e estável porém não adaptativo
- usa memória auxiliar $O(n)$. As variantes *in-place* usam mais comparações e movimentações ou não são estáveis
- implementação elegante tanto de forma recursiva como iterativa; tanto em arranjos quanto em listas ligadas (*in-place!*); tanto em memória interna como externa

Quicksort

- algoritmo ótimo *in-place* mais rápido (se implementado de forma correta e eficiente)
- não é estável nem adaptativo



Quickselect

Problema: Achar o k-ésimo menor elemento de um conjunto

- ligado a outros problemas, e.g. achar o minimo, máximo, mediana, o k-ésimo maior, os primeiros k menores

Quickselect

Problema: Achar o k-ésimo menor elemento de um conjunto

- ligado a outros problemas, e.g. achar o minimo, máximo, mediana, o k-ésimo maior, os primeiros k menores
- complexidade $\Omega(n)$ e $O(n * \log n)$

Quickselect

Problema: Achar o k-ésimo menor elemento de um conjunto

- ligado a outros problemas, e.g. achar o minimo, máximo, mediana, o k-ésimo maior, os primeiros k menores
- complexidade $\Omega(n)$ e $O(n * \log n)$

Como resolver sem ordenar?

Quickselect

Problema: Achar o k-ésimo menor elemento de um conjunto

- ligado a outros problemas, e.g. achar o minimo, máximo, mediana, o k-ésimo maior, os primeiros k menores
- complexidade $\Omega(n)$ e $O(n * \log n)$

Como resolver sem ordenar?

Solução Quickselect: Usar o particionamento para colocar um elemento na posição correta j. Se $j \neq k$ continuar com o particionamento somente na metade que inclui k

- **Caso médio**: o vetor é particionado na metade, logo usando o Teorema Mestre

$$T_{avg}(n) = T_{avg}(n/2) + O(n) = \Theta(n)$$

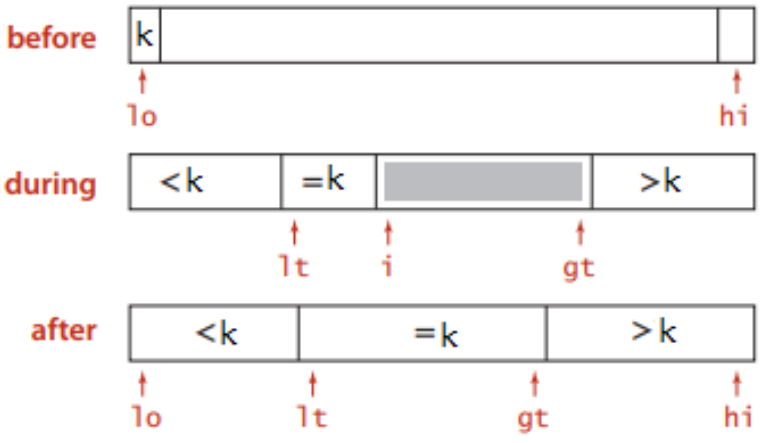
3-way Partitioning e 3-way Quicksort

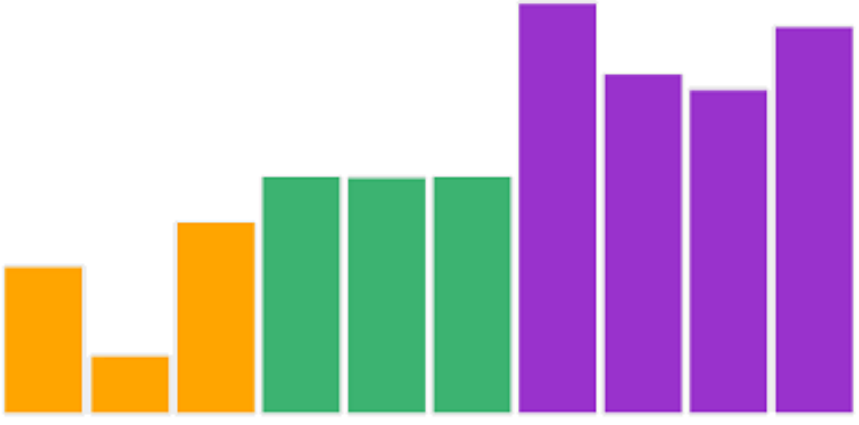
Problema: Ordenar um conjunto de três elementos com muitas repetições (e.g. aprovados, reprovados com F e reprovados com O; números positivos, negativos e zeros; etc)

3-way Partitioning e 3-way Quicksort

Problema: Ordenar um conjunto de três elementos com muitas repetições (e.g. aprovados, reprovados com F e reprovados com O; números positivos, negativos e zeros; etc)

Solução 3-way Partitioning: Adaptação do particionamento para lidar com chaves repetidas

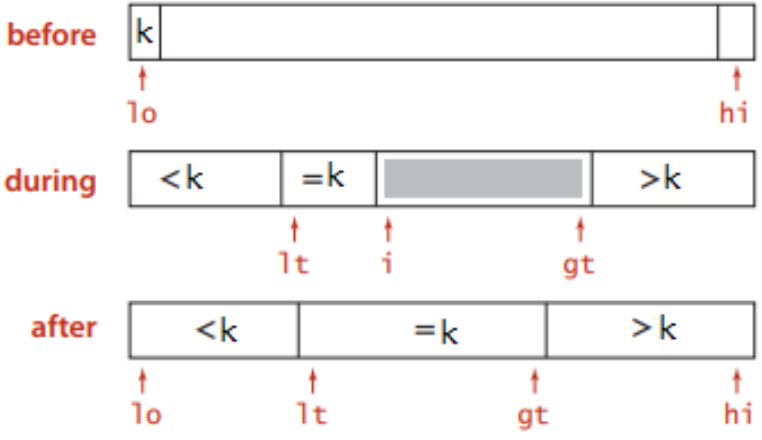


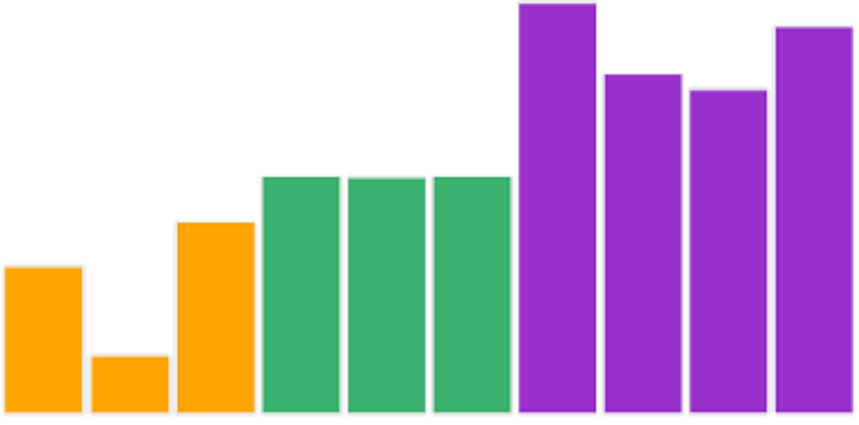


3-way Partitioning e 3-way Quicksort

Problema: Ordenar um conjunto de três elementos com muitas repetições (e.g. aprovados, reprovados com F e reprovados com O; números positivos, negativos e zeros; etc)

Solução 3-way Partitioning: Adaptação do particionamento para lidar com chaves repetidas



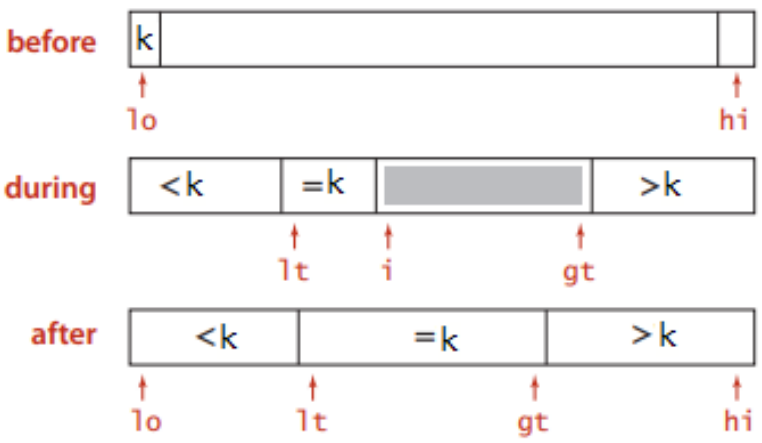


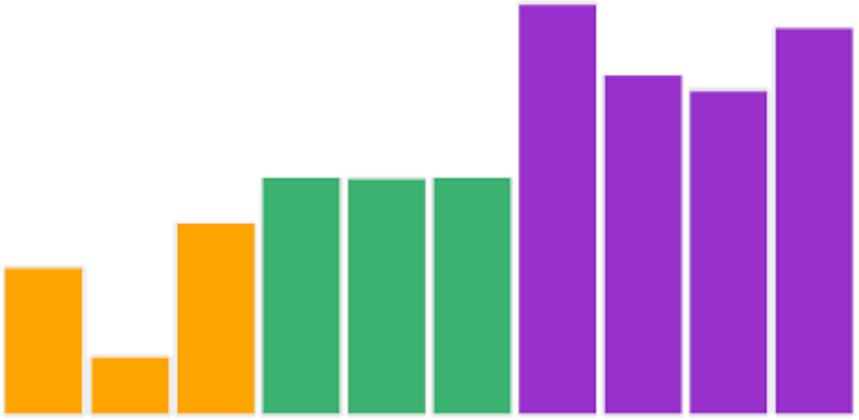
3-way Partitioning + Quicksort = 3-way Quicksort

3-way Partitioning e 3-way Quicksort

Problema: Ordenar um conjunto de três elementos com muitas repetições (e.g. aprovados, reprovados com F e reprovados com O; números positivos, negativos e zeros; etc)

Solução 3-way Partitioning: Adaptação do particionamento para lidar com chaves repetidas





3-way Partitioning + Quicksort = 3-way Quicksort

Dá para melhorar?

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Sumário

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

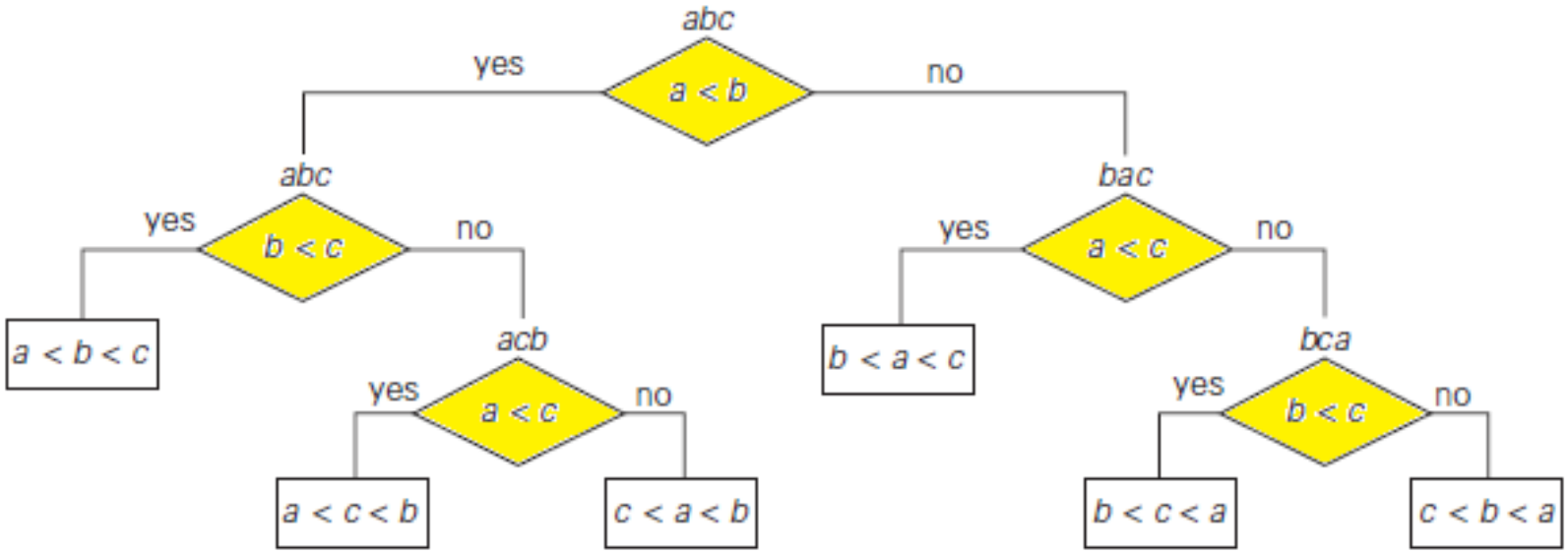
Além do $\Omega(n * \log n)$

	inplace?	stable?	best	average	worst	remarks
selection	✔		½ n^2	½ n^2	½ n^2	n exchanges
insertion	✔	✔	n	¼ n^2	½ n^2	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	½ $n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2\, n \ln n$	½ n^2	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2\, n \ln n$	½ n^2	improves quicksort when duplicate keys
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail
?	✔	✔	n	n	n	?

Adaptado de https://algs4.cs.princeton.edu/lectures/23Quicksort-2x2.pdf

Além do $\Omega(n * \log n)$

- **Limite assintótico da ordenação baseada em comparações** é $\Omega(n \log n)$, i.e. qualquer algoritmo da ordenação baseado em comparações usa no mínimo $n \log n$ comparações no caso pior



The algorithm´s work on a particular input of size n can be traced by a path from the root to a leaf in its **decision tree**, and the number of comparisons made by the algorithm on such a run is equal to the length of this path. Hence, the number of comparisons in the worst case is equal to the height of the algorithm´s decision tree. For any binary tree with l leaves and height h, $h \geq \log_2 l$. Anany Levitin. Introduction to the Design and Analysis of Algorithms

Além do $\Omega(n * \log n)$

- **Limite assintótico da ordenação baseada em comparações** é $\Omega(n \log n)$, i.e. qualquer algoritmo da ordenação baseado em comparações usa no mínimo $n \log n$ comparações no caso pior
- Para quebrar o limite é necessário ter informação adicional sobre a chave e evitar as comparações entre elas usando técnicas como a contagem, agrupamento, distribuição

Exemplo: (Exercício DNA ordenado) Dada uma sequência de ADN, imprimir a sequência ordenada.

AAGAAAACTGAAAACACATGGCTTTTT

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Além do $\Omega(n * \log n)$

- **Limite assintótico da ordenação baseada em comparações** é $\Omega(n \log n)$, i.e. qualquer algoritmo da ordenação baseado em comparações usa no mínimo $n \log n$ comparações no caso pior
- Para quebrar o limite é necessário ter informação adicional sobre a chave e evitar as comparações entre elas usando técnicas como a contagem, agrupamento, distribuição

Exemplo: (Exercício DNA ordenado) Dada uma sequência de ADN, imprimir a sequência ordenada.

AAGAAAACTGAAAACACATGGCTTTTT

AAAAAAAAAAAACCCCGGGGTTTTTTT

Universidade Federal do ABC

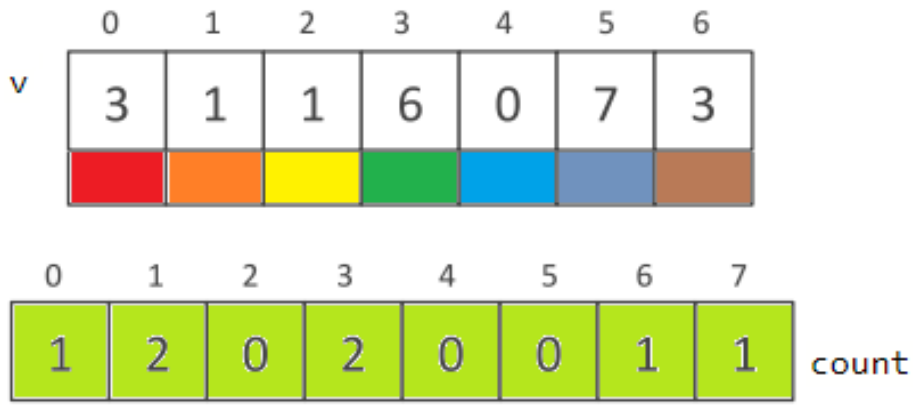
CMCC Centro de Matemática, Computação e Cognição

Sumário

Ordenação por contagem (Counting Sort)

Algoritmo desenvolvido por Harold H. Seward em 1954

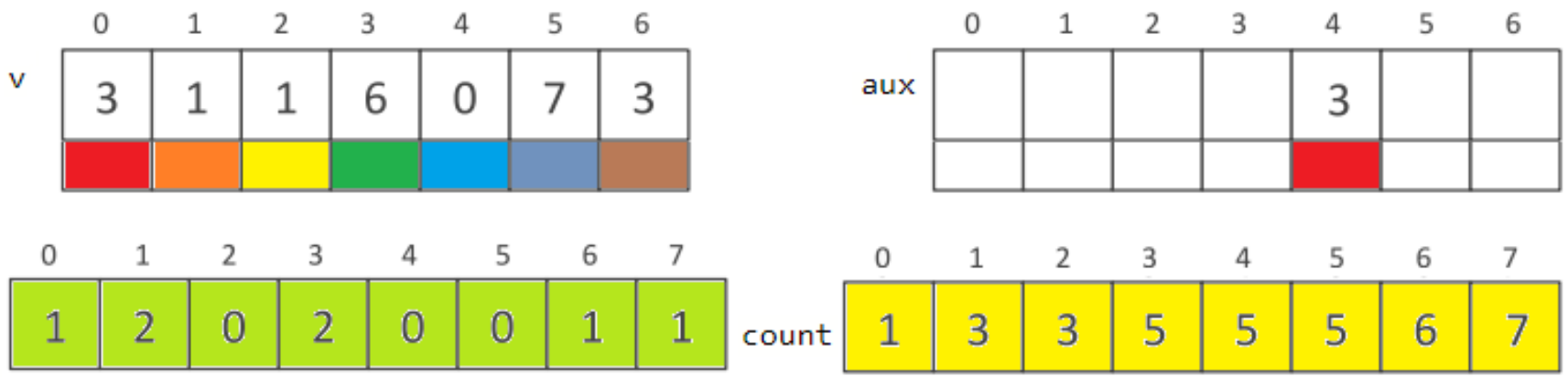
- As chaves são (ou podem ser convertidas em tempo $O(1)$) inteiros entre zero e $K > 0$
- Calcula quantas vezes cada elemento usando um vetor de contadores



Ordenação por contagem (Counting Sort)

Algoritmo desenvolvido por Harold H. Seward em 1954

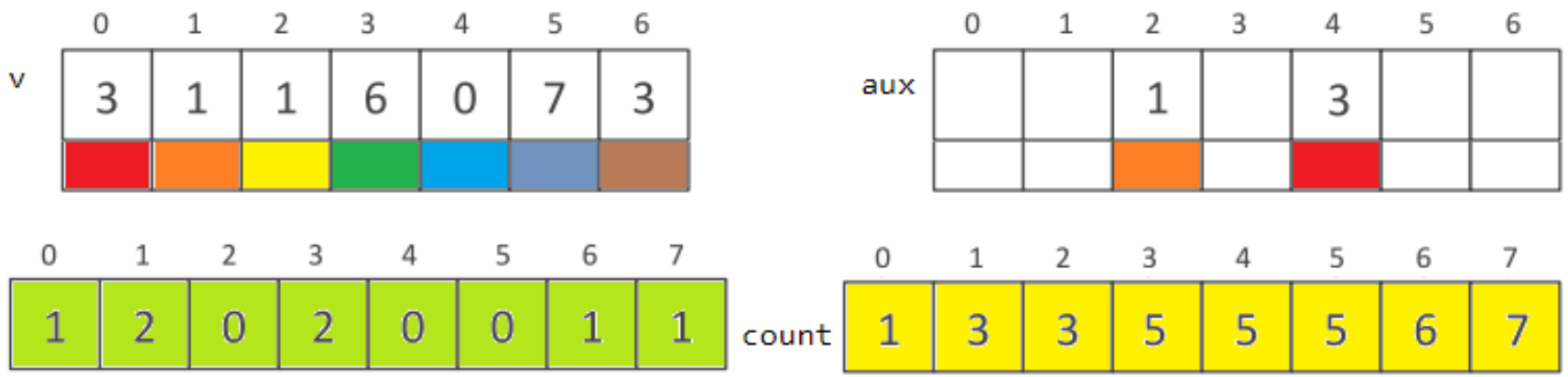
- As chaves são (ou podem ser convertidas em tempo $O(1)$) inteiros entre zero e $K > 0$
- Calcula quantas vezes cada elemento usando um vetor de contadores
- A posição de um registro no arranjo ordenado é calculada usando o número de chaves menores ou iguais que à dele
- Precisa de um vetor auxiliar para ordenar (não é *in-place*)



Ordenação por contagem (Counting Sort)

Algoritmo desenvolvido por Harold H. Seward em 1954

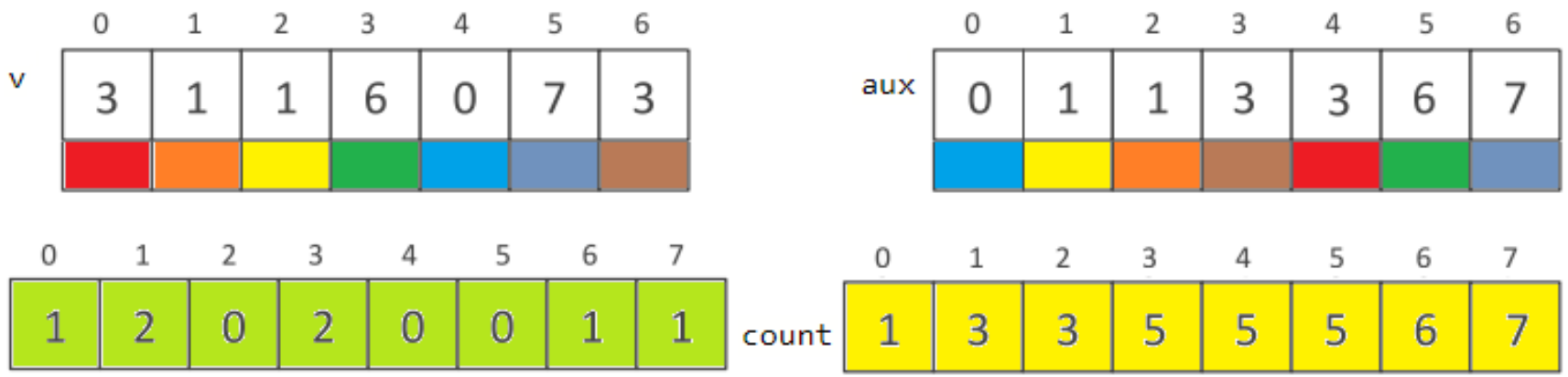
- As chaves são (ou podem ser convertidas em tempo $O(1)$) inteiros entre zero e $K > 0$
- Calcula quantas vezes cada elemento usando um vetor de contadores
- A posição de um registro no arranjo ordenado é calculada usando o número de chaves menores ou iguais que à dele
- Precisa de um vetor auxiliar para ordenar (não é *in-place*)



Ordenação por contagem (Counting Sort)

Algoritmo desenvolvido por Harold H. Seward em 1954

- As chaves são (ou podem ser convertidas em tempo $O(1)$) inteiros entre zero e $K > 0$
- Calcula quantas vezes cada elemento usando um vetor de contadores
- A posição de um registro no arranjo ordenado é calculada usando o número de chaves menores ou iguais que à dele
- Precisa de um vetor auxiliar para ordenar (não é *in-place*)



Algoritmo de Ordenação por contagem (Counting Sort)

```
 8  void countingSort(int v[], int n, int K, int aux[]) {
 9   // initializing the counters
10   int count[++K], i;
11   for (i = 0; i < K; i++)
12     count[i] = 0 ;
13
14   // counting: count[i] <- # keys ==i
15   for (i = 0; i < n; i++)
16     count[v[i]] ++;
17
18   // accumulating: count[i] <- # keys <=i
19   for (i = 1; i < K; i++)
20     count[i] += count[i-1];
21
22   //putting the keys in place
23   for (i = 0; i < n; i++)
24     aux[--count[v[i]]] = v[i];
25  }
```

https://repl.it/@mirthalina/countingSort

Algoritmo de Ordenação por contagem (Counting Sort)

```
 8  void countingSort(int v[], int n, int K, int aux[]) {
 9   // initializing the counters
10   int count[++K], i;
11   for (i = 0; i < K; i++)
12     count[i] = 0 ;
13
14   // counting: count[i] <- # keys ==i
15   for (i = 0; i < n; i++)
16     count[v[i]] ++;
17
18   // accumulating: count[i] <- # keys <=i
19   for (i = 1; i < K; i++)
20     count[i] += count[i-1];
21
22   //putting the keys in place
23   for (i = 0; i < n; i++)
24     aux[--count[v[i]]] = v[i];
25  }
```

Complexidade

$$\Theta(n + K)$$

Para garantir a estabilidade o laço deve ser decrescente i.e.

```
for (i = n-1; i >= 0; i--)
```

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

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Complexidade dos algoritmos de ordenação

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)

http://bigocheatsheet.com/

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Referências Bibliográficas

- Algorithms, Robert Sedgewick and Kevin Wayne, 4th Edition, 2011, Slides http://algs4.cs.princeton.edu/lectures/
- Introduction to Algorithms, 3rd Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, 2009
- The Art of Computer Programming 3rd Edition, Donald Knuth, Section 5.2.4: Sorting by Merging, 1997
- Introduction to the Design and Analysis of Algorithms, Anany Levitin, 3rd Edition, 2011
- Comparison Sorting Algorithms
  https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- Sorting Algorithms Animation http://www.sorting-algorithms.com/
- Animação de algoritmos de ordenação
  http://nicholasandre.com.br/sorting/