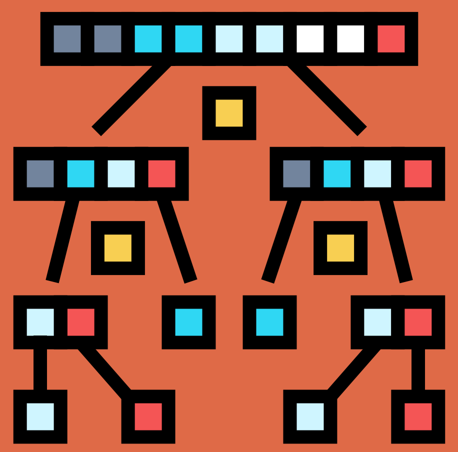
QuickSort

Análise de Algoritmo

EDUARDO HENRIQUE MACHADO
NATÁLIA ALMADA

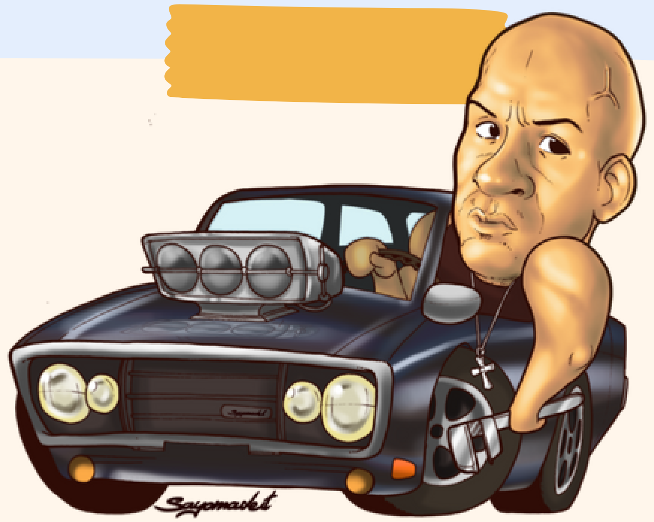
QuickSort

Charles Antony Richard Hoar - 1960

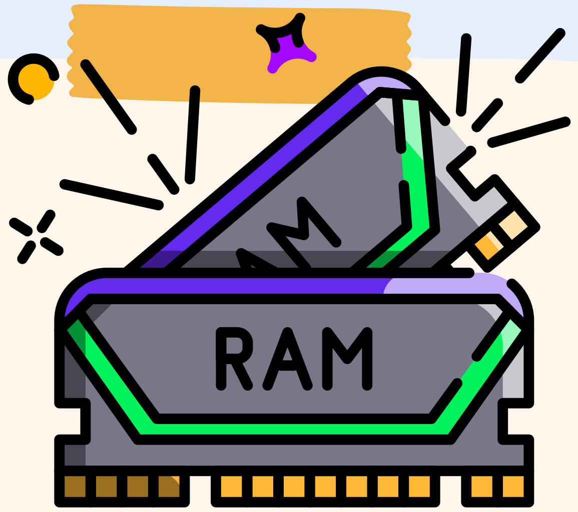


O algoritmo Quicksort implementa a estratégia da divisão e conquista., ou seja: Subdivide o problema original em subproblemas menores, que são resolvidos recursivamente.

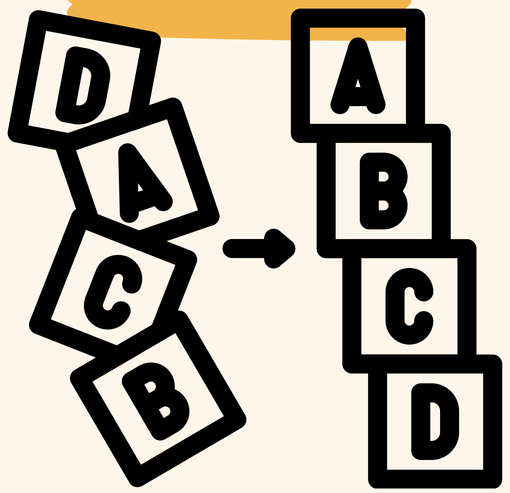
Vantagens



Rápido e eficiente na execução média



Funciona bem em ambientes de memória virtual



Vantagem em ordenar *in place*

Não requer muito espaço

Pontos em destaque

Rapidez	Memória Virtual	In Place	Economia de espaço
Dividir o problema em subproblemas permite a ordenação eficaz de conjuntos de dados de forma rápida e precisa.	Localidade Espacial: acessa elementos em regiões contíguas do array durante a partição e a troca, Localidade Temporal: eusa partes do array durante a execução; O menor uso de memória adicional significa que menos páginas de memória precisam ser alocadas	Ele realiza a ordenação dos elementos dentro do array original, sem a necessidade de alocar espaço adicional significativo para uma cópia dos dados.	Não requer espaço adicional proporcional ao tamanho do array para ordenar os dados. O espaço extra necessário é apenas para o armazenamento da pilha de chamadas recursivas (ou a pilha explícita usada em versões iterativas)

Complexidade

Melhor e Médio Caso:

O melhor caso ocorre quando todas as chamadas de Divide devolvem um índice q a meio caminho entre p e r, e portanto o vetor A[p .. q−1] tem $\lceil (n-1)/2 \rceil$ elementos e o vetor A[q+1 .. r] tem $\lfloor (n-1)/2 \rfloor$ elementos.

- Nível 0 (Raiz): Custo é n
- Nível 1: O array é dividido em duas sublistas de tamanho $\frac{n}{2}$.
  Custo total neste nível é $2 * \frac{n}{2} = n$
- Nível 2: Cada sublista é dividida em duas partes de tamanho $\frac{n}{4}$
  Custo total neste nível é $4 * \frac{n}{4} = n$

Complexidade

Melhor e Médio Caso:

O melhor caso ocorre quando todas as chamadas de Divide devolvem um índice q a meio caminho entre p e r, e portanto o vetor A[p .. q−1] tem $\lceil (n - 1)/2 \rceil$ elementos e o vetor A[q+1 .. r] tem $\lfloor (n - 1)/2 \rfloor$ elementos.

- Nível k:
- Existem 2^k sublistas, cada uma de tamanho $n*(2^k)$

Custo total neste nível é $2 * \frac{n^k}{2^k} = n$

Número Total de Níveis:

O número de níveis é $\log_2 n$ (cada nível da recursão reduz o problema pela metade)

Complexidade

Melhor e Médio Caso:

O melhor caso ocorre quando todas as chamadas de Divide devolvem um índice q a meio caminho entre p e r, e portanto o vetor A[p .. q−1] tem $\lceil (n - 1)/2 \rceil$ elementos e o vetor A[q+1 .. r] tem $\lfloor (n - 1)/2 \rfloor$ elementos.

- T(n): O tempo total necessário para ordenar um array de n elementos.
- 2xT($\frac{n}{2}$): O custo total das duas chamadas recursivas. Cada chamada recursiva é responsável por ordenar uma das duas sublistas de tamanho .
- O(n): O custo para particionar o array de n elementos. Esse é o custo de reorganizar os elementos em torno do pivô.

Complexidade

Melhor e Médio Caso:

Nos fazendo chegar a essa recorrencia:

$$T(n)=2xT\left(\frac{n}{2}\right)+O(n)$$

Complexidade

Melhor e Médio Caso:

Somatória Total dos Custos

$$T(n) = \sum_{i=0}^{\log_2 n - 1} c \times n$$

$$T(n) = c \times n \times \log_2 n$$

Complexidade

Melhor e Médio Caso:

Complexidade:

$$O(n \log n)$$

Complexidade

Pior Caso:

Ocorre quando todas as chamadas de Divide devolvem q = p ou q = r. Como Divide faz <u>n-1 comparações</u>, a intuição sugere que $T^*(n)$ obedece a <u>recorrência</u> $T^*(n) = n-1 + T^*(0) + T^*(n-1)$.

- $$\begin{aligned} T^*(n) &= T^*(n-1) + n - 1 \\ &= T^*(n-2) + (n-2) + (n-1) \\ &= T^*(n-3) + (n-3) + (n-2) + (n-1) \\ &= T^*(n-4) + (n-4) + (n-3) + (n-2) + (n-1) \\ &\;\vdots \\ &= T^*(0) + 0 + 1 + 2 + \dots + (n-2) + (n-1) \\ &= n(n-1)/2. \end{aligned}$$

Complexidade

Pior Caso:

O melhor caso ocorre quando todas as chamadas de Divide devolvem um índice q a meio caminho entre p e r, e portanto o vetor A[p .. q−1] tem $\lceil (n-1)/2 \rceil$ elementos e o vetor A[q+1 .. r] tem $\lfloor (n-1)/2 \rfloor$ elementos.

- Nível 0 (Raiz): Custo é n
- Nível 1: O array é dividido em uma sublista de tamanho *n-1* e uma de tamanho 0 .
  Custo total neste nível é *n-1*
- Nível 2: A sublista de tamanho n-1 é dividida em uma sublista de tamanho n-2 e uma de tamanho 0.
  Custo total neste nível é n-2

Complexidade

Pior Caso:

O melhor caso ocorre quando todas as chamadas de Divide devolvem um índice q a meio caminho entre p e r, e portanto o vetor A[p .. q−1] tem $\lceil (n-1)/2 \rceil$ elementos e o vetor A[q+1 .. r] tem $\lfloor (n-1)/2 \rfloor$ elementos.

- Nível k:
- A cada nível, o custo é n−k, até que a sublista tenha tamanho 1)

Custo total neste nível é 2 * $\underline{n}$ = n

Número Total de Níveis:

O número de níveis é n

Complexidade

Pior Caso:

Somatória Total dos Custos

$$T(n) = \sum_{i=0}^{n-1} (n - i)$$

$$T(n) = n + (n - 1) + (n - 2) + \ldots + 1$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Complexidade

Melhor e Médio Caso:

Complexidade:

$$O(n^2)$$

Complexidade

O que significa isso?

$$O(n \log n)$$

log n geralmente está em base 2

n (log n) é a quantidade de vezes que você pode dividir n pela metade até chegar a 1

n: representa o número de operações necessárias para combinar as sublistas em cada nível de divisão

$$O(n^2)$$

$O(n^2)$ significa que o tempo de execução do algoritmo cresce proporcionalmente ao quadrado do tamanho da entrada

Complexidade

Para n = 8?

$O(n \log n)$

log n geralmente está em base 2

Então, log(8) = 3.

O tempo de execução do algoritmo seria proporcional a:

$8 * 3 = 24$

$O(n^2)$

Então, o tempo de execução seria proporcional a:

$8^2 = 64$

Complexidade

Para n = 1024?

$O(n \log n)$

log n geralmente está em base 2

Então, $\log(1024) = 10$

O tempo de execução do algoritmo seria proporcional a:

$$1024 * 10 = 10240$$

Em ms ≅ 10,24 segundos

$O(n^2)$

Então, o tempo de execução seria proporcional a:

$$1024^2 = 1048576$$

Em ms ≅ 1048,5 segundos

≅ 17,47minutos

Tem melhor?

MergeSort

O MergeSort mantém uma complexidade de O (nlogn) em todos os casos devido à sua estrutura constante de divisão e mesclagem. O array é dividido sempre exatamente ao meio, ordena e depois mescla.

O Quicksort mantém uma complexidade de O (nlogn) apenas quando se comporta bem ou conforme o esperado, divide o array baseando na escolha de um pivô, os maiores vao pra uma lista e os menores para outra. Sem mesclagem.

Tem melhor?

MergeSort

O MergeSort é mais estável. O quicksort não.

O MergeSort usa O(n) memória extra. O quicksort O(logn).

DEPENDE

QuickSort é geralmente mais eficiente devido ao menor overhead de memória e melhor desempenho em cache, mas requer boas estratégias de escolha de pivô para evitar o pior caso.


THANKS