algortimos de ordenacao ? referencias

Hugo Veríssimo
Optimização Combinatória 24/25
Universidade de Aveiro
Aveiro, Portugal
hugoverissimo@ua.pt

João Cardoso
Optimização Combinatória 24/25
Universidade de Aveiro
Aveiro, Portugal
joaopcardoso@ua.pt

***Resumo*—rever italico ou não no nome dos coisos + "sort"VS "Sort"+ mathcal O VS normal math O**

Palavras-chave: zzz, aaa

I. Introdução

Os algoritmos de ordenação são essenciais em várias áreas da optimização combinatória e da investigação operacional, sendo utilizados para organizar dados de forma eficiente, melhorar o desempenho de algoritmos de procura e reduzir a complexidade computacional de problemas em larga escala.

Em contextos de otimização combinatória, onde para vários problemas se torna fundamental a ordenação prévia de dados, como nas heurísticas para o problema Knapsack, no algoritmo de Kruskal para construção de árvores geradoras mínimas, ou em algoritmos gulosos para cobertura de conjuntos, a escolha do método de ordenação pode impactar significativamente a eficiência global da solução.

Este trabalho visa explorar os principais algoritmos de ordenação, destacando as suas características, o seu funcionamento, a complexidade computacional associada e uma análise comparativa entre os mesmos. Considerando as diferentes implementações e constrangimentos associados, exploramos os cenários onde cada algoritmo poderá ser mais útil. O objetivo é compreender como a ordenação pode ser utilizada de forma estratégica para melhorar a eficiência de algoritmos em diversos cenários, e como otimizações na própria ordenação podem levar a avanços significativos em problemas computacionalmente intensivos.

II. Algoritmos de Ordenação

Nesta secção serão apresentados os algoritmos selecionados, detalhando-se individualmente o seu princípio básico de funcionamento, a complexidade temporal, as vantagens e desvantagens, e a aplicabilidade prática em diferentes cenários da otimização combinatória. POR REVER SE REALMENTE SE FAZ ISTO TUDO E MUDAR ?!?!?!??!

A. Bubble Sort acrescentar informação sobre a família Exchange sorts

O algoritmo *Bubble sort* consiste em comparar um dado item com todos os elementos de uma lista, em que o item vai se reposicionando com elementos comparativamente menores até encontrar um item maior ou chegar ao fim da lista. O nome do algoritmo prende-se com a forma como o elemento comparado vai ascendendo (ou afundando, consoante seja usado), tal como uma bolha na água.

Tipicamente este algoritmo não encontra aplicação em casos reais devido à sua ineficiência e complexidade de ordem $O(n^2)$. No entanto, a sua rápida implementação pode ser de interesse em casos particulares, em que listas que estejam próximas da sua forma ordenada passam pelo algoritmo com uma complexidade de ordem $O(n)$.

Algoritmo 1 Bubble Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

$n \leftarrow \text{len}(arr)$
for i from 0 to $n-1$ **do**
 $swapped \leftarrow$ False
 for j from 0 to $n-2-i$ **do**
  if $arr[j] > arr[j+1]$ **then**
   Swap $arr[j]$ and $arr[j+1]$
   $swapped \leftarrow$ True
  end if
 end for
 if not $swapped$ **then**
  break
 end if
end for
return arr

FALAR DA COMPLEXIDADE DO ALGORITMO A PARTIR DA ANÁLISE DO MESMO + BEST AND WORST CASE

Existem algoritmos baseados nesta abordagem, tal como o *Cocktail sort*, que aplica a metodologia em sentido crescente e decrescente, tornando-o mais rápido por norma, mas com o mesmo grau de complexidade no pior cenário.

B. Selection Sort

O algoritmo de *Selection sort* tem um procedimento bastante diferente do algoritmo anterior. Faz parte da família de algoritmos de *Selection sorts*, em que a lista de items ordenados vai sendo gerada por comparação a partir da lista original desordenada. No algoritmo em questão, procura-se o menor elemento disponível e coloca-se no início da nova lista. Estes passos repetem-se na lista desordenada até não existirem mais e a lista ordenada ser criada.

Algoritmo 2 Selection Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

$n \leftarrow \text{len}(arr)$
for i from 0 to $n-1$ **do**
  $min\_idx \leftarrow i$
  for j from $i+1$ to $n-1$ **do**
    if $arr[j] < arr[min\_idx]$ **then**
      $min\_idx \leftarrow j$
    end if
  end for
  if $min\_idx \neq i$ **then**
    Swap $arr[i]$ and $arr[min\_idx]$
  end if
end for
return arr

A complexidade do algoritmo pode ser analisada com base no número de comparações e trocas realizadas. O número de comparações é sempre o mesmo, independentemente da ordem inicial dos dados. Para uma lista com n elementos, o número total de comparações realizadas é dado por $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$, o que corresponde a uma complexidade de ordem $\mathcal{O}(n^2)$ tanto no melhor como no pior caso.

O número de trocas, por outro lado, depende da configuração inicial da lista. No melhor caso, quando a lista já está ordenada, o índice do menor elemento nunca difere do índice atual, pelo que nenhuma troca é efetuada. Assim, o número de trocas é zero. No pior caso, quando a lista está em ordem inversa, cada iteração do algoritmo encontra um menor elemento que precisa de ser trocado, resultando em $n-1$ trocas. Neste caso, o número de trocas é linear, ou seja, de ordem $\mathcal{O}(n)$.

Em resumo, o algoritmo Selection Sort tem complexidade temporal $\mathcal{O}(n^2)$ em todos os casos, realiza poucas trocas (até $n-1$), não é adaptativo à ordenação inicial dos dados e não é um algoritmo estável.

Dentro desta família de algoritmos, há um que devemos destacar, chamado *Gnome sort*, que consiste nas operações do *Selection sort*, mas sem ter o segundo ciclo dentro do primeiro, resultando num número de operações semelhante ao *Insertion sort*, e por consequência pior que o *Selection sort*.

C. *Insertion Sort*

O algoritmo Insertion Sort ordena uma lista construindo progressivamente uma sublista ordenada à esquerda. A cada passo, o elemento atual é comparado com os anteriores e inserido na posição correta, deslocando os elementos maiores uma posição à frente.

Algoritmo 3 Insertion Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

for i from 1 to $\text{len}(arr)-1$ **do**
  $key \leftarrow arr[i]$
  $j \leftarrow i-1$
  while $j \geq 0$ and $arr[j] > key$ **do**
    $arr[j+1] \leftarrow arr[j]$
    $j \leftarrow j-1$
  end while
  $arr[j+1] \leftarrow key$
end for
return arr

A complexidade do algoritmo pode ser analisada considerando o número de comparações e movimentações de elementos. No pior caso, a lista está em ordem inversa. Cada novo elemento precisa de ser comparado com todos os anteriores e movido para o início da lista. Assim, o número total de comparações e deslocações neste caso é dado por $(1+2+\ldots+n-1) = \frac{n(n-1)}{2}$, o que resulta numa complexidade temporal de ordem $\mathcal{O}(n^2)$.

No melhor caso, a lista já está ordenada. O algoritmo ainda percorre os elementos, mas como nenhuma comparação resulta em troca, cada elemento é apenas comparado uma vez. Neste cenário, o número total de comparações é linear, e a complexidade temporal é $\mathcal{O}(n)$.

No caso médio, assume-se que metade dos elementos anteriores são maiores que o atual, levando a aproximadamente metade das comparações do pior caso, o que continua a resultar em complexidade $\mathcal{O}(n^2)$.

Em termos de eficiência prática, o Insertion Sort é eficiente para listas pequenas ou quase ordenadas, e tem a vantagem de ser estável, além de funcionar em tempo linear no melhor caso.

Resumindo, o algoritmo apresenta complexidade:

$$\begin{cases} \mathcal{O}(n) & \text{melhor caso (lista ordenada)} \\ \mathcal{O}(n^2) & \text{pior caso (lista inversamente ordenada)} \\ \mathcal{O}(n^2) & \text{caso médio} \end{cases}$$

D. *Counting Sort*

O algoritmo Counting Sort é um algoritmo de ordenação não comparativo, apropriado para listas de números inteiros não negativos e com amplitude limitada. A ideia principal consiste em contar quantas vezes cada valor ocorre e depois reconstruir a lista ordenada a partir dessas contagens.

Algoritmo 4 Counting Sort

Entrada: lista *arr*
Saída: lista ordenada *sorted_arr*

```
max_val ← max(arr)
count ← array of zeros with length max_val + 1
for each num in arr do
    count[num] ← count[num] + 1
end for
sorted_arr ← empty list
for i from 0 to max_val do
    if count[i] > 0 then
        Append count[i] copies of i to sorted_arr
    end if
end for
return sorted_arr
```

A complexidade do algoritmo depende de dois fatores: o tamanho da lista n e o maior valor presente, denotado por $k = \max(\text{arr})$. A primeira parte do algoritmo percorre a lista original e preenche um vetor auxiliar `count` de tamanho $k+1$, incrementando as posições correspondentes. Esta etapa tem complexidade $\mathcal{O}(n)$. Em seguida, o algoritmo percorre o vetor `count`, que tem tamanho $k+1$, e reconstrói a lista ordenada. Esta etapa tem complexidade $\mathcal{O}(k)$.

Assim, a complexidade total do algoritmo é $\mathcal{O}(n + k)$, tanto no melhor como no pior caso. Não há distinção entre diferentes ordens iniciais da lista, pois o algoritmo não realiza comparações entre os elementos.

O melhor desempenho é alcançado quando k é da mesma ordem de n, resultando em tempo linear. No entanto, se k for muito maior que n, o algoritmo torna-se ineficiente em termos de espaço e tempo, pois aloca um vetor proporcional a k, independentemente de quantos desses valores realmente aparecem.

Em resumo, a complexidade do Counting Sort é:

$$\mathcal{O}(n + k)$$

em todos os casos, sendo eficiente apenas quando $k \in \mathcal{O}(n)$. O algoritmo é estável e apropriado para ordenar valores inteiros com gama pequena e conhecida.

E. Radix Sort

O algoritmo Radix Sort ordena números inteiros processando-os dígito a dígito, a partir do dígito menos significativo até ao mais significativo. Em cada passo, aplica-se um algoritmo de ordenação estável, geralmente o Counting Sort, para ordenar os elementos com base no dígito atual. O número de iterações é igual ao número de dígitos do maior número presente na lista.

Algoritmo 5 Radix Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function COUNTINGSORTRADIX(arr, exp)
    n ← len(arr)
    output ← array of zeros of length n
    count ← array of zeros of length 10
    for i from 0 to n − 1 do
        index ← (arr[i] ÷ exp) mod 10
        count[index] ← count[index] + 1
    end for
    for i from 1 to 9 do
        count[i] ← count[i] + count[i − 1]
    end for
    for i from n − 1 to 0 do
        index ← (arr[i] ÷ exp) mod 10
        output[count[index] − 1] ← arr[i]
        count[index] ← count[index] − 1
    end for
    for i from 0 to n − 1 do
        arr[i] ← output[i]
    end for
end function
max_num ← max(arr)
exp ← 1
while max_num ÷ exp > 0 do
    COUNTINGSORTRADIX(arr, exp)
    exp ← exp × 10
end while
return arr
```

Seja n o número de elementos da lista, k o valor máximo presente, e d o número de dígitos de k na base decimal. Em cada uma das d iterações, o Counting Sort é aplicado com complexidade $\mathcal{O}(n + b)$, onde b é a base usada (tipicamente 10). Como b é constante, o custo de cada iteração é $\mathcal{O}(n)$.

Assim, a complexidade total do algoritmo Radix Sort é:

$$\mathcal{O}(d \cdot n)$$

como $d = \lfloor \log_{10}(k) \rfloor + 1$, pode também escrever-se como $\mathcal{O}(n \cdot \log k)$ para base 10. Esta complexidade mantém-se tanto no melhor como no pior caso, já que todos os elementos são sempre processados integralmente, independentemente da ordem inicial.

O Radix Sort é eficiente quando os números têm número limitado de dígitos (isto é, $d \in \mathcal{O}(1)$), caso em que o tempo de execução é linear em n. No entanto, o algoritmo não é comparativo e apenas funciona diretamente para inteiros ou representações discretizáveis. Além disso, o consumo de espaço extra é $\mathcal{O}(n + b)$ por iteração.

Em resumo, o algoritmo apresenta complexidade:

$$\mathcal{O}(n \cdot d) = \mathcal{O}(n \cdot \log k)$$

em todos os casos, e é estável, não comparativo e adequado para listas grandes de inteiros com número de dígitos moderado.

F. Quick Sort

O algoritmo Quick Sort é um algoritmo de ordenação baseado no paradigma de dividir-para-conquistar. Escolhe um elemento como pivô e reorganiza a lista de forma que todos os elementos menores ou iguais ao pivô fiquem à esquerda, e os maiores à direita. Em seguida, aplica-se recursivamente o mesmo procedimento às duas sublistas geradas.

Algoritmo 6 Quick Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function PARTITION(items, low, high)
    pivot ← items[high]
    i ← low − 1
    for j from low to high − 1 do
        if items[j] ≤ pivot then
            i ← i + 1
            Swap items[i] and items[j]
        end if
    end for
    Swap items[i + 1] and items[high]
    return i + 1
end function
function QUICKSORT(items, low, high)
    if low < high then
        pivot_index ← PARTITION(items, low, high)
        QUICKSORT(items, low, pivot_index − 1)
        QUICKSORT(items, pivot_index + 1, high)
    end if
end function
QUICKSORT(arr, 0, len(arr) − 1)
return arr
```

A complexidade do algoritmo depende da qualidade da escolha do pivô em cada chamada recursiva. No melhor caso, o pivô divide o array em duas partes aproximadamente iguais a cada etapa. A profundidade da recursão será então $\log n$, e em cada nível realiza-se um número total de operações proporcional a n, resultando numa complexidade total de:

$$\mathcal{O}(n \log n)$$

No pior caso, o pivô escolhido é sempre o menor ou o maior elemento, fazendo com que uma das sublistas tenha tamanho zero e a outra contenha $n-1$ elementos. O número de chamadas recursivas será então n, e o número total de comparações será da ordem de:

$$\mathcal{O}(n^2)$$

O caso médio, assumindo uma escolha aleatória ou razoável do pivô, também resulta em complexidade esperada de $\mathcal{O}(n \log n)$. É importante destacar que a versão apresentada do algoritmo não é estável e realiza ordenação in-place, com consumo de memória adicional proporcional à profundidade da pilha de chamadas recursivas (em média $\mathcal{O}(\log n)$).

Em resumo, a complexidade do Quick Sort é:

$$\begin{cases} \mathcal{O}(n \log n) & \text{melhor caso (divisões equilibradas)} \\ \mathcal{O}(n^2) & \text{pior caso (divisões altamente desequilibradas)} \\ \mathcal{O}(n \log n) & \text{caso médio (pivôs aleatórios ou bons)} \end{cases}$$

G. Merge Sort

O algoritmo Merge Sort é um algoritmo de ordenação baseado no paradigma de dividir-para-conquistar. A cada passo, divide recursivamente a lista ao meio até obter sublistas de tamanho 1, e depois combina essas sublistas de forma ordenada, reconstruindo a lista original em ordem crescente.

Algoritmo 7 Merge Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function MERGESORT(arr)
    if len(arr) > 1 then
        mid ← len(arr) ÷ 2
        left ← arr[0 : mid]
        right ← arr[mid :]
        MERGESORT(left)
        MERGESORT(right)
        i, j, k ← 0, 0, 0
        while i < len(left) and j < len(right) do
            if left[i] ≤ right[j] then
                arr[k] ← left[i]
                i ← i + 1
            else
                arr[k] ← right[j]
                j ← j + 1
            end if
            k ← k + 1
        end while
        while i < len(left) do
            arr[k] ← left[i]
            i ← i + 1
            k ← k + 1
        end while
        while j < len(right) do
            arr[k] ← right[j]
            j ← j + 1
            k ← k + 1
        end while
    end if
end function
MERGESORT(arr)
return arr
```

A análise da complexidade do Merge Sort baseia-se no facto de que a cada nível da recursão a lista é dividida ao meio (número de níveis $\log n$), e que em cada nível o tempo necessário para combinar as sublistas (fase de merge) é proporcional ao número total de elementos n. Assim, a complexidade total é:

$$\mathcal{O}(n \log n)$$

Este tempo mantém-se tanto no melhor como no pior caso, uma vez que o número de divisões e de fusões é sempre o mesmo, independentemente da ordem inicial dos dados.

O melhor caso ocorre quando as sublistas já estão ordenadas entre si, mas mesmo assim o algoritmo precisa de as percorrer totalmente para realizar as comparações, mantendo a complexidade em $\mathcal{O}(n \log n)$.

O pior caso ocorre quando todos os elementos estão intercalados de forma que cada comparação resulta num movimento de elementos entre as sublistas. Ainda assim, o número total de operações permanece proporcional a $n \log n$, pois todos os elementos são comparados e copiados exatamente uma vez por nível de recursão.

O algoritmo é estável, mas não é in-place, pois exige espaço adicional proporcional a n para armazenar as sublistas temporárias durante a fusão.

Em resumo, a complexidade do Merge Sort é:

$$\begin{cases} \mathcal{O}(n \log n) & \text{melhor caso} \\ \mathcal{O}(n \log n) & \text{pior caso} \\ \mathcal{O}(n \log n) & \text{caso médio} \end{cases}$$

H. Heap Sort

O algoritmo Heap Sort ordena uma lista transformando-a inicialmente numa estrutura de heap máximo, em que o maior elemento está na raiz. Em seguida, remove repetidamente o elemento da raiz (o maior), coloca-o no fim da lista e reestrutura o heap com os elementos restantes. Este processo continua até a lista estar totalmente ordenada.

Algoritmo 8 Heap Sort

Entrada: lista *arr*
Saída: lista ordenada *arr*

```
function HEAPIFY(arr, n, i)
    largest ← i
    left ← 2i + 1
    right ← 2i + 2
    if left < n and arr[left] > arr[largest] then
        largest ← left
    end if
    if right < n and arr[right] > arr[largest] then
        largest ← right
    end if
    if largest ≠ i then
        Swap arr[i] and arr[largest]
        HEAPIFY(arr, n, largest)
    end if
end function
function HEAPSORT(arr)
    n ← len(arr)
    for i from ⌊n/2⌋ − 1 down to 0 do
        HEAPIFY(arr, n, i)
    end for
    for i from n − 1 down to 1 do
        Swap arr[0] and arr[i]
        HEAPIFY(arr, i, 0)
    end for
    return arr
end function
HEAPSORT(arr)
```

A construção inicial do heap é feita percorrendo os nós não-folha da árvore e aplicando a operação `Heapify` de forma decrescente. Esta construção tem complexidade $\mathcal{O}(n)$, resultado de uma análise amortizada do número de operações necessárias em cada nível da árvore.

Após a construção do heap, o algoritmo realiza $n - 1$ extrações sucessivas do máximo. Cada extração envolve uma troca seguida de uma chamada à função `Heapify`, que tem complexidade $\mathcal{O}(\log n)$, pois percorre a altura do heap. Assim, a fase de ordenação tem complexidade total $\mathcal{O}(n \log n)$.

Como o processo é sempre o mesmo independentemente da ordem inicial dos dados, a complexidade do algoritmo é a mesma no melhor, pior e caso médio. O Heap Sort não é estável, mas tem a vantagem de ser in-place, necessitando apenas de espaço adicional constante.

Em resumo, a complexidade do Heap Sort é:

$$\begin{cases} \mathcal{O}(n \log n) & \text{melhor caso} \\ \mathcal{O}(n \log n) & \text{pior caso} \\ \mathcal{O}(n \log n) & \text{caso médio} \end{cases}$$

I. Timsort

O algoritmo Timsort é um algoritmo híbrido que combina Insertion Sort e Merge Sort, sendo utilizado como algoritmo de ordenação padrão em linguagens como Python e Java. O seu funcionamento baseia-se na identificação de subsequências ordenadas (chamadas *runs*), que são ordenadas individualmente com Insertion Sort e depois fundidas com Merge Sort.

Algoritmo 9 Timsort

Entrada: lista *arr*

Saída: lista ordenada *arr*

```
function INSERTIONSORT(arr, left, right)
    for i from left + 1 to right do
        key ← arr[i]
        j ← i − 1
        while j ≥ left and arr[j] > key do
            arr[j + 1] ← arr[j]
            j ← j − 1
        end while
        arr[j + 1] ← key
    end for
end function
function MERGE(arr, left, mid, right)
    left_part ← arr[left : mid + 1]
    right_part ← arr[mid + 1 : right + 1]
    i, j, k ← 0, 0, left
    while i < len(left_part) and j < len(right_part) do
        if left_part[i] ≤ right_part[j] then
            arr[k] ← left_part[i]
            i ← i + 1
        else
            arr[k] ← right_part[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i < len(left_part) do
        arr[k] ← left_part[i]
        i, k ← i + 1, k + 1
    end while
    while j < len(right_part) do
        arr[k] ← right_part[j]
        j, k ← j + 1, k + 1
    end while
end function
function TIMSORT(arr)
    MIN_RUN ← 32
    n ← len(arr)
    for start from 0 to n − 1 in steps of MIN_RUN do
        INSERTIONSORT(arr, start, min(start + MIN_RUN − 1, n − 1))
    end for
    size ← MIN_RUN
    while size < n do
        for left from 0 to n − 1 in steps of 2 × size do
            mid ← min(n − 1, left + size − 1)
            right ← min(n − 1, left + 2 × size − 1)
            if mid < right then
                MERGE(arr, left, mid, right)
            end if
        end for
        size ← size × 2
    end while
    return arr
end function
TIMSORT(arr)
```

O parâmetro `MIN_RUN` define o tamanho mínimo das *runs* a serem ordenadas com Insertion Sort. Esta escolha permite explorar a eficiência do Insertion Sort em listas pequenas e parcialmente ordenadas. Em seguida, as *runs* são fundidas de forma semelhante ao Merge Sort, com fusões sucessivas em potências de dois.

No melhor caso, quando a lista já está parcialmente ordenada e contém grandes *runs* crescentes, o custo de ordenação por Insertion Sort é quase linear, e o número de fusões necessárias é reduzido. Neste cenário, o tempo de execução é:

$$\mathcal{O}(n)$$

No pior caso, quando os dados não contêm nenhuma ordenação prévia relevante, o algoritmo comporta-se essencialmente como o Merge Sort, realizando fusões completas a cada nível. Assim, a complexidade no pior caso é:

$$\mathcal{O}(n \log n)$$

O caso médio também resulta em complexidade $\mathcal{O}(n \log n)$, mas com desempenho prático superior a algoritmos puros devido à utilização adaptativa do Insertion Sort e à minimização de operações de fusão desnecessárias.

Além disso, o Timsort é estável e eficiente em termos práticos, sendo adequado para listas grandes e dados parcialmente ordenados. O consumo de memória adicional é proporcional ao número de *runs* e ao processo de fusão, tal como no Merge Sort.

Em resumo, a complexidade do Timsort é:

$$\begin{cases} \mathcal{O}(n) & \text{melhor caso (lista já parcialmente ordenada)} \\ \mathcal{O}(n \log n) & \text{pior caso (dados desordenados)} \\ \mathcal{O}(n \log n) & \text{caso médio} \end{cases}$$

III. Simulação de Ordenações

ns q usar python para simluar os algs e contar ops basicas

Tabela I: comparao das complexidades teoricas CONFIRMAR where k is range of input values and n is number of elemenets to sort

Algoritmo	**Complexidade**
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Counting Sort	$O(n+k)$
Radix Sort	$O(nk)$
Quick Sort	$O(n \log n)$*
Merge Sort	$O(n \log n)$
Heap Sort	$O(n \log n)$
Timsort	$O(n \log n)$

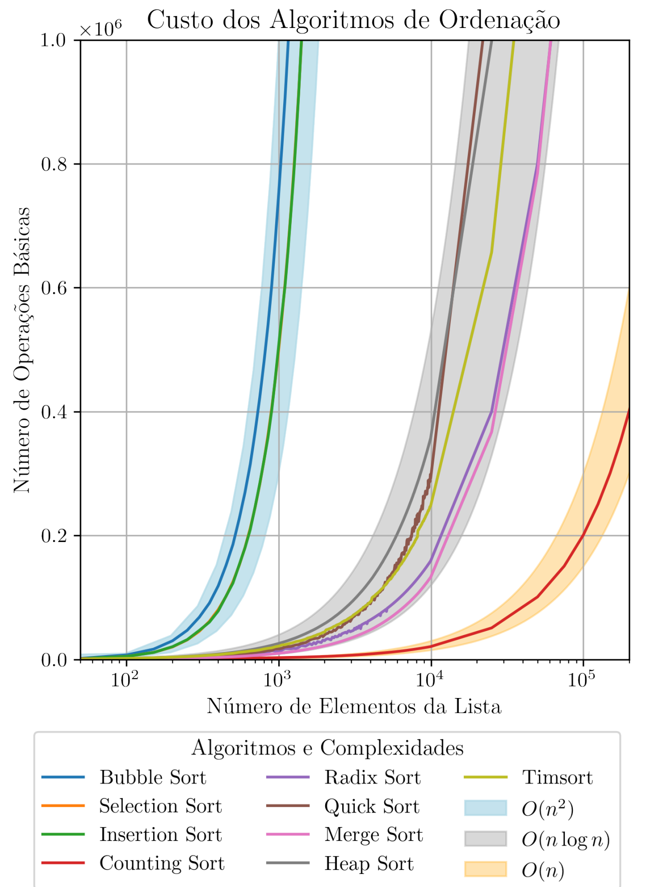

Figura 1: complxidades baseadas no num de ops basicas em listas geradas com numeros inteiros positivos aleatorios até 1000

Referências