

# algoritmos 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 cois<sup>os</sup> + "sort"VS "Sort"+  $\mathcal{O}$  VS normal math  $\mathcal{O}$

**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 optimização 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 optimizações na própria ordenação podem levar a avanços significativos em problemas computacionalmente intensivos.

## II. ALGORITMOS DE ORDENAÇÃO

Nesta secção serão apresentados os algoritmos seleccionados, detalhando-se individualmente o seu princípio básico de funcionamento, a complexidade temporal, as vantagens e desvantagens, e a aplicabilidade prática em diferentes cenários da optimização 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.

O algoritmo Bubble Sort ordena uma lista iterando repetidamente sobre os elementos e trocando pares adjacentes se estiverem fora de ordem. A cada iteração, o maior elemento "bolha" para o fim da lista não ordenada. O processo repete-se até que nenhuma troca seja necessária, sinalizando que a lista está ordenada.

---

### Algoritmo 1 Bubble Sort

---

**Entrada:** lista *arr*

**Saída:** lista ordenada *arr*

---

```
1:  $n \leftarrow \text{len}(arr)$ 
2: for  $i$  from 0 to  $n - 1$  do
3:    $swapped \leftarrow \text{False}$ 
4:   for  $j$  from 0 to  $n - 2 - i$  do
5:     if  $arr[j] > arr[j + 1]$  then
6:       Swap  $arr[j]$  and  $arr[j + 1]$ 
7:        $swapped \leftarrow \text{True}$ 
8:     end if
9:   end for
10:  if not  $swapped$  then
11:    break
12:  end if
13: end for
14: return  $arr$ 
```

---

A complexidade do algoritmo depende do número de comparações e trocas realizadas. Em todos os casos, o número máximo de comparações realizadas é:

$$(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}$$

o que corresponde a uma complexidade de ordem  $\mathcal{O}(n^2)$ .

No melhor caso, quando a lista já está ordenada, a flag *swapped* permite terminar após a primeira iteração sem trocas. Neste cenário, o algoritmo apenas percorre uma vez a lista e não realiza nenhuma troca, resultando numa complexidade de tempo linear:

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

No pior caso, quando a lista está em ordem inversa, o algoritmo realiza o número máximo possível de trocas e comparações, resultando em:

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

Apesar da sua simplicidade, o Bubble Sort é ineficiente para listas grandes. No entanto, é estável e pode ser útil apenas para fins didáticos ou em contextos onde a simplicidade do código é prioritária.

Tipicamente este algoritmo não encontra aplicação em casos reais devido à sua ineficiência e complexidade de ordem  $\mathcal{O}(n^2)$ . 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  $\mathcal{O}(n)$ .

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.

Em resumo, a complexidade do Bubble Sort é:

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

## 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 itens 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*

---

```

1:  $n \leftarrow \text{len}(arr)$ 
2: for  $i$  from 0 to  $n - 1$  do
3:    $min\_idx \leftarrow i$ 
4:   for  $j$  from  $i + 1$  to  $n - 1$  do
5:     if  $arr[j] < arr[min\_idx]$  then
6:        $min\_idx \leftarrow j$ 
7:     end if
8:   end for
9:   if  $min\_idx \neq i$  then
10:    Swap  $arr[i]$  and  $arr[min\_idx]$ 
11:   end if
12: end for
13: 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  $(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$ , 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*

---

```

1: for  $i$  from 1 to  $\text{len}(arr) - 1$  do
2:    $key \leftarrow arr[i]$ 
3:    $j \leftarrow i - 1$ 
4:   while  $j \geq 0$  and  $arr[j] > key$  do
5:      $arr[j + 1] \leftarrow arr[j]$ 
6:      $j \leftarrow j - 1$ 
7:   end while
8:    $arr[j + 1] \leftarrow key$ 
9: end for
10: 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 deslocamentos neste caso é dado por  $(1+2+\dots+n-1) = \frac{n(n-1)}{2}$ , 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*

---

```

1: max_val ← max(arr)
2: count ← array of zeros with length max_val + 1
3: for each num in arr do
4:   count[num] ← count[num] + 1
5: end for
6: sorted_arr ← empty list
7: for i from 0 to max_val do
8:   if count[i] > 0 then
9:     Append count[i] copies of i to sorted_arr
10:  end if
11: end for
12: return sorted_arr
```

---

A complexidade do algoritmo depende de dois fatores: o tamanho da lista  $n$  e o maior valor presente, denotado por  $k = \max(arr)$ . 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*

---

```

1: function COUNTINGSORTRADIX(arr, exp)
2:   n ← len(arr)
3:   output ← array of zeros of length n
4:   count ← array of zeros of length 10
5:   for i from 0 to n - 1 do
6:     index ← (arr[i] ÷ exp) mod 10
7:     count[index] ← count[index] + 1
8:   end for
9:   for i from 1 to 9 do
10:    count[i] ← count[i] + count[i - 1]
11:  end for
12:  for i from n - 1 to 0 do
13:    index ← (arr[i] ÷ exp) mod 10
14:    output[count[index] - 1] ← arr[i]
15:    count[index] ← count[index] - 1
16:  end for
17:  for i from 0 to n - 1 do
18:    arr[i] ← output[i]
19:  end for
20: end function
21: max_num ← max(arr)
22: exp ← 1
23: while max_num ÷ exp > 0 do
24:   COUNTINGSORTRADIX(arr, exp)
25:   exp ← exp × 10
26: end while
27: 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  $arr$

---

```

1: function PARTITION( $items, low, high$ )
2:    $pivot \leftarrow items[high]$ 
3:    $i \leftarrow low - 1$ 
4:   for  $j$  from  $low$  to  $high - 1$  do
5:     if  $items[j] \leq pivot$  then
6:        $i \leftarrow i + 1$ 
7:       Swap  $items[i]$  and  $items[j]$ 
8:     end if
9:   end for
10:  Swap  $items[i + 1]$  and  $items[high]$ 
11:  return  $i + 1$ 
12: end function
13: function QUICKSORT( $items, low, high$ )
14:   if  $low < high$  then
15:      $pivot\_index \leftarrow$  PARTITION( $items, low, high$ )
16:     QUICKSORT( $items, low, pivot\_index - 1$ )
17:     QUICKSORT( $items, pivot\_index + 1, high$ )
18:   end if
19: end function
20: QUICKSORT( $arr, 0, len(arr) - 1$ )
21: 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*

---

```
1: function MERGESORT(arr)
2:   if len(arr) > 1 then
3:     mid ← len(arr) ÷ 2
4:     left ← arr[0 : mid]
5:     right ← arr[mid :]
6:     MERGESORT(left)
7:     MERGESORT(right)
8:     i, j, k ← 0, 0, 0
9:     while i < len(left) and j < len(right) do
10:      if left[i] ≤ right[j] then
11:        arr[k] ← left[i]
12:        i ← i + 1
13:      else
14:        arr[k] ← right[j]
15:        j ← j + 1
16:      end if
17:      k ← k + 1
18:    end while
19:    while i < len(left) do
20:      arr[k] ← left[i]
21:      i ← i + 1
22:      k ← k + 1
23:    end while
24:    while j < len(right) do
25:      arr[k] ← right[j]
26:      j ← j + 1
27:      k ← k + 1
28:    end while
29:  end if
30: end function
31: MERGESORT(arr)
32: 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*

---

```
1: function HEAPIFY(arr, n, i)
2:   largest ← i
3:   left ← 2i + 1
4:   right ← 2i + 2
5:   if left < n and arr[left] > arr[largest] then
6:     largest ← left
7:   end if
8:   if right < n and arr[right] > arr[largest] then
9:     largest ← right
10:  end if
11:  if largest ≠ i then
12:    Swap arr[i] and arr[largest]
13:    HEAPIFY(arr, n, largest)
14:  end if
15: end function
16: function HEAPSORT(arr)
17:   n ← len(arr)
18:   for i from  $\lfloor n/2 \rfloor - 1$  down to 0 do
19:     HEAPIFY(arr, n, i)
20:   end for
21:   for i from n - 1 down to 1 do
22:     Swap arr[0] and arr[i]
23:     HEAPIFY(arr, i, 0)
24:   end for
25:   return arr
26: end function
27: 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*

---

```

1: function INSERTIONSORT(arr, left, right)
2:   for i from left + 1 to right do
3:     key  $\leftarrow$  arr[i]
4:     j  $\leftarrow$  i - 1
5:     while j  $\geq$  left and arr[j] > key do
6:       arr[j + 1]  $\leftarrow$  arr[j]
7:       j  $\leftarrow$  j - 1
8:     end while
9:     arr[j + 1]  $\leftarrow$  key
10:  end for
11: end function
12: function MERGE(arr, left, mid, right)
13:  left_part  $\leftarrow$  arr[left : mid + 1]
14:  right_part  $\leftarrow$  arr[mid + 1 : right + 1]
15:  i, j, k  $\leftarrow$  0, 0, left
16:  while i < len(left_part) and j < len(right_part)
17:  do
18:    if left_part[i]  $\leq$  right_part[j] then
19:      arr[k]  $\leftarrow$  left_part[i]
20:      i  $\leftarrow$  i + 1
21:    else
22:      arr[k]  $\leftarrow$  right_part[j]
23:      j  $\leftarrow$  j + 1
24:    end if
25:    k  $\leftarrow$  k + 1
26:  end while
27:  while i < len(left_part) do
28:    arr[k]  $\leftarrow$  left_part[i]
29:    i, k  $\leftarrow$  i + 1, k + 1
30:  end while
31:  while j < len(right_part) do
32:    arr[k]  $\leftarrow$  right_part[j]
33:    j, k  $\leftarrow$  j + 1, k + 1
34:  end while
35: end function
36: function TIMSORT(arr)
37:  MIN_RUN  $\leftarrow$  32
38:  n  $\leftarrow$  len(arr)
39:  for start from 0 to n - 1 in steps of MIN_RUN
40:  do
41:    INSERTIONSORT(arr, start,
42:    min(start + MIN_RUN - 1, n - 1))
43:  end for
44:  size  $\leftarrow$  MIN_RUN
45:  while size < n do
46:    for left from 0 to n - 1 in steps of  $2 \times \textit{size}$  do
47:      mid  $\leftarrow$  min(n - 1, left + size - 1)
48:      right  $\leftarrow$  min(n - 1, left +  $2 \times \textit{size}$  - 1)
49:      if mid < right then
50:        MERGE(arr, left, mid, right)
51:      end if
52:    end for
53:    size  $\leftarrow$  size  $\times$  2
54:  end while
55:  return arr
56: end function
57: 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    | $\mathcal{O}(n^2)$        |
| Selection Sort | $\mathcal{O}(n^2)$        |
| Insertion Sort | $\mathcal{O}(n^2)$        |
| Counting Sort  | $\mathcal{O}(n + k)$      |
| Radix Sort     | $\mathcal{O}(nk)$         |
| Quick Sort     | $\mathcal{O}(n \log n)^*$ |
| Merge Sort     | $\mathcal{O}(n \log n)$   |
| Heap Sort      | $\mathcal{O}(n \log n)$   |
| Timsort        | $\mathcal{O}(n \log n)$   |

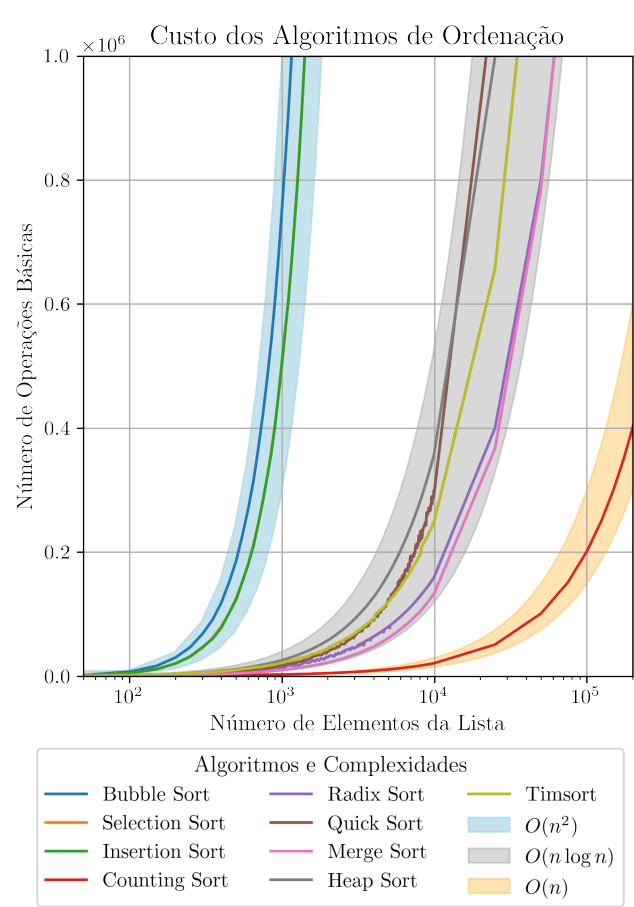


Figura 1: complxidades baseadas no num de ops basicas em listas geradas com numeros inteiros positivos aleatorios até 1000

### REFERÊNCIAS