Análise Comparativa de Algoritmos de Ordenação: Teoria e Desempenho Prático

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*—Este trabalho apresenta uma análise comparativa de nove algoritmos de ordenação, abordando tanto os seus fundamentos teóricos como o desempenho empírico observado em simulações computacionais. Foram considerados algoritmos clássicos, eficientes e especializados, com especial atenção às suas complexidades temporais. Através de implementações em *Python* e da execução em listas aleatórias de várias dimensões, foi possível verificar a correspondência entre os resultados experimentais e as previsões teóricas, bem como destacar situações em que a escolha do algoritmo mais adequado depende do contexto específico dos dados. Os resultados reforçam a importância de combinar análise teórica e avaliação prática na seleção de métodos de ordenação.**

Palavras-chave: arranjar referencias

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.

Serão abordados algoritmos de ordenação representativos de diferentes estratégias fundamentais, nomeadamente os métodos baseados em troca (*exchange sort*), como o Bubble Sort; em seleção (*selection sort*), como o Selection Sort; em inserção (*insertion sort*), como o Insertion Sort; e em divisão e conquista, como o Merge Sort e o Quick Sort. Para além destes, serão também analisados algoritmos baseados em distribuição, como o Counting Sort e o Radix Sort, os quais não se baseiam em comparações diretas entre elementos.

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 apresentam-se nove algoritmos de ordenação selecionados, com uma descrição individual do seu princípio de funcionamento, análise da complexidade temporal, identificação das principais vantagens e limitações, bem como a caracterização dos cenários de melhor caso, pior caso e caso médio.

A. Bubble Sort

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 ← len(arr)
2: for i from 0 to n − 1 do
3:     swapped ← 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 ← True
8:         end if
9:     end for
10:    if not swapped then
11:        break
12:    end if
13: end for
14: return arr
```

Através da análise do pseudocódigo apresentado, verifica-se que a complexidade do algoritmo depende do número de comparações e trocas realizadas. Em todos os casos, o número máximo 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 $\mathcal{O}(n^2)$.

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 numa complexidade quadrática, $\mathcal{O}(n^2)$.

Tipicamente este algoritmo não encontra aplicação em casos reais devido à sua ineficiência. No entanto, a sua rápida implementação pode ser de interesse em casos particulares, em que listas que estejam próximas da sua forma ordenada.

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 Selection Sort é um algoritmo de ordenação que pertence à família dos algoritmos *selection sort*. Neste tipo de algoritmo, a lista é ordenada progressivamente através da seleção repetida do menor elemento entre os ainda não ordenados. A cada iteração, o menor elemento da porção não ordenada da lista é identificado e trocado com o elemento na posição corrente, avançando assim a fronteira da sublista ordenada.

Algoritmo 2 Selection Sort

Entrada: lista arr
Saída: lista ordenada arr

```
1: n ← len(arr)
2: for i from 0 to n − 1 do
3:     min_idx ← i
4:     for j from i + 1 to n − 1 do
5:         if arr[j] < arr[min_idx] then
6:             min_idx ← j
7:         end if
8:     end for
9:     if min_idx ≠ 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

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

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 $n-1$) e não é adaptativo à ordenação inicial dos dados.

C. Insertion Sort

O algoritmo Insertion Sort ordena uma lista construindo, passo a passo, uma sublista ordenada à esquerda. Em cada iteração, o elemento atual é comparado com os anteriores e inserido na posição adequada, garantindo que a sublista à esquerda permanece ordenada. Para isso, os elementos maiores são deslocados uma posição para a direita, abrindo espaço para a inserção correta. O processo repete-se até que todos os elementos estejam no seu devido lugar.

Algoritmo 3 Insertion Sort

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

```
1: for i from 1 to len(arr) − 1 do
2:     key ← arr[i]
3:     j ← i − 1
4:     while j ≥ 0 and arr[j] > key do
5:         arr[j + 1] ← arr[j]
6:         j ← j − 1
7:     end while
8:     arr[j + 1] ← key
9: end for
10: return arr
```

Através da análise do pseudocódigo relativo ao algoritmo em destaque, verifica-se que a complexidade do mesmo 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 $\mathcal{O}(n^2)$.

Resumindo, o algoritmo apresenta complexidade:

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

Em termos de eficiência prática, o Insertion Sort torna-se bastante eficiente para listas pequenas ou quase ordenadas, onde o número de deslocamentos é reduzido. Além disso, por ser estável e não requerer memória adicional significativa, é frequentemente utilizado como componente auxiliar em algoritmos de ordenação mais complexos, como o Timsort ou em fases finais de ordenação híbrida.

D. Counting Sort

O algoritmo Counting Sort é um método de ordenação não comparativo, especialmente eficiente para listas de números inteiros não negativos com amplitude limitada. Em vez de comparar elementos entre si, constrói uma tabela de frequências que regista quantas vezes cada valor ocorre. A partir dessa informação, determina-se a posição exata de cada elemento na lista final, permitindo uma ordenação estável e com complexidade linear nos melhores casos, quando o intervalo dos valores é conhecido e reduzido.

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
```

Atendendo o pseudocódigo apresentado, conclui-se que 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 intervalo de variação relativamente pequeno em relação ao número de elementos, permitindo uma ordenação eficiente em tempo linear, quando esta condição é satisfeita.

E. Radix Sort

O algoritmo Radix Sort ordena números inteiros ao processá-los dígito a dígito, começando pelo dígito menos significativo (unidades) e avançando até ao mais significativo (milhares, etc.). Em cada etapa, os elementos são reorganizados com base no dígito atual, utilizando um algoritmo de ordenação estável, tipicamente o Counting Sort, para garantir

que a ordem relativa dos elementos com dígitos iguais é preservada. O processo repete-se tantas vezes quantos os dígitos do maior número na lista, até a ordenação estar completa.

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 dada por

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

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.

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.

Para além disso, o consumo de espaço extra é $\mathcal{O}(n+b)$ por iteração.

Assim, o Radix Sort é eficiente quando os números têm uma quantidade limitada 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 valores que possam ser representados por dígitos. Por exemplo, números irracionais como π ou e, que possuem infinitas casas decimais não periódicas, não podem ser tratados diretamente, pois não têm uma representação discreta e finita compatível com a lógica dígito-a-dígito do algoritmo.

F. Quick Sort

O algoritmo Quick Sort é um método de ordenação baseado no paradigma de dividir-para-conquistar, amplamente valorizado pela sua eficiência prática e adaptabilidade a arquiteturas modernas. Escolhe um elemento como pivô e reorganiza a lista colocando à esquerda os elementos menores ou iguais ao pivô e à direita os maiores. Este processo de partição divide o problema original em subproblemas independentes, permitindo aplicar o mesmo procedimento de forma recursiva a cada sublista.

A estrutura do Quick Sort é particularmente favorável ao paralelismo, já que, após a partição, as duas sublistas podem ser ordenadas simultaneamente em processos ou threads distintos, sem dependências entre si.

Algoritmo 6 Quick Sort

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

```
1: function Partition(items, low, high)
2:     pivot ← items[high]
3:     i ← low − 1
4:     for j from low to high − 1 do
5:         if items[j] ≤ pivot then
6:             i ← 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 ← 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
```

Tendo em conta o pseudocódigo apresentado, a complexidade do algoritmo depende da qualidade da escolha do pivô em cada chamada recursiva.

No melhor caso, o pivô divide a lista 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 $\mathcal{O}(n \log n)$.

Em resumo, a complexidade do Quick Sort é:

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

G. Merge Sort

O algoritmo Merge Sort é um método de ordenação baseado no paradigma de dividir-para-conquistar, tal como o Quick Sort. No entanto, em vez de utilizar um pivô para partição, o Merge Sort divide a lista ao meio de forma determinística, recursivamente, até obter sublistas de tamanho 1. Em seguida, procede à fusão ordenada dessas sublistas, reconstruindo progressivamente a lista original em ordem crescente.

É, portanto, semelhante ao Quick Sort na abordagem recursiva e na divisão do problema, mas sem recurso a um pivô. A divisão é sempre equitativa e a ordenação ocorre na fase de combinação, não na de separação.

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:

$$\Big\{ \mathcal{O}(n \log n) \quad \text{em todos os casos}$$

H. Heap Sort

O algoritmo Heap Sort ordena uma lista aproveitando as propriedades de uma estrutura chamada heap máxima, onde o maior elemento está sempre no topo. Embora represente uma árvore, esta estrutura é implementada como um vetor e obedece à regra de que cada elemento é maior ou igual aos seus filhos. O processo começa por reorganizar a lista para formar essa heap. Em seguida, remove-se o maior elemento (no topo), troca-se com o último da lista e ajusta-se a heap com os elementos restantes. Este ciclo repete-se até que todos os elementos estejam no lugar, resultando numa lista 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 ⌊n/2⌋ − 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)
```

Com base no pseudocódigo do algoritmo, a construção inicial da heap é feita percorrendo os nós que não são folhas e aplicando a operação `Heapify` de baixo para cima. Esta fase tem complexidade $\mathcal{O}(n)$, resultado de uma análise amortizada sobre o número de operações em cada nível da árvore.

Depois da heap construída, o algoritmo realiza $n - 1$ extrações do maior elemento. Cada extração envolve uma troca e uma chamada à função `Heapify`, que percorre a altura da heap, com custo $\mathcal{O}(\log n)$. Assim, esta fase tem complexidade total $\mathcal{O}(n \log n)$.

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

Em resumo, a complexidade do Heap Sort é:

$$\Big\{ \mathcal{O}(n \log n) \quad \text{em todos os casos}$$

I. Timsort

O algoritmo Timsort é um algoritmo híbrido sofisticado que combina as estratégias do Insertion Sort e do Merge Sort, tirando partido das vantagens de ambos. É utilizado como algoritmo de ordenação padrão em linguagens como Python e Java, devido à sua elevada eficiência em dados reais.

O funcionamento do Timsort assenta na identificação de subsequências já ordenadas na lista original, conhecidas como *runs*. Estas *runs* são inicialmente tratadas com Insertion Sort, aproveitando a sua eficiência em listas pequenas ou quase ordenadas. Em seguida, as diferentes *runs* são fundidas de forma eficiente com Merge Sort, mantendo a estabilidade e garantindo boa performance mesmo em casos adversos.

Graças a esta abordagem adaptativa, o Timsort consegue desempenhos excelentes em listas parcialmente ordenadas, combinando velocidade, estabilidade e robustez.

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:

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

III. Resultados

A partir da análise individual de cada algoritmo, torna-se possível uma comparação global das suas complexidades.

Tabela I: Comparação das complexidades temporais teóricas médias dos algoritmos de ordenação considerados. Assume-se que n é o número de elementos a ordenar, k o maior valor presente e d o número máximo de algarismos de um elemento.

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	$\mathcal{O}(n \times d)$
Quick Sort	$\mathcal{O}(n \log n)$
Merge Sort	$\mathcal{O}(n \log n)$
Heap Sort	$\mathcal{O}(n \log n)$
Timsort	$\mathcal{O}(n \log n)$

A Tabela I apresenta a referida comparação, tendo em conta a complexidade média teórica de cada um dos algoritmos analisados.

De modo a analisar estas complexidades de uma forma prática, os algoritmos foram implementados em *Python*, aplicados a listas de diferentes dimensões, com elementos inteiros positivos até 1000. Para cada execução, foi contabilizado o número de operações básicas realizadas, permitindo comparar o comportamento empírico com a complexidade teórica. Para além disso, para cada tamanho de lista, foram geradas 5 listas aleatórias, de modo a criar robustez face aos melhores e piores casos.

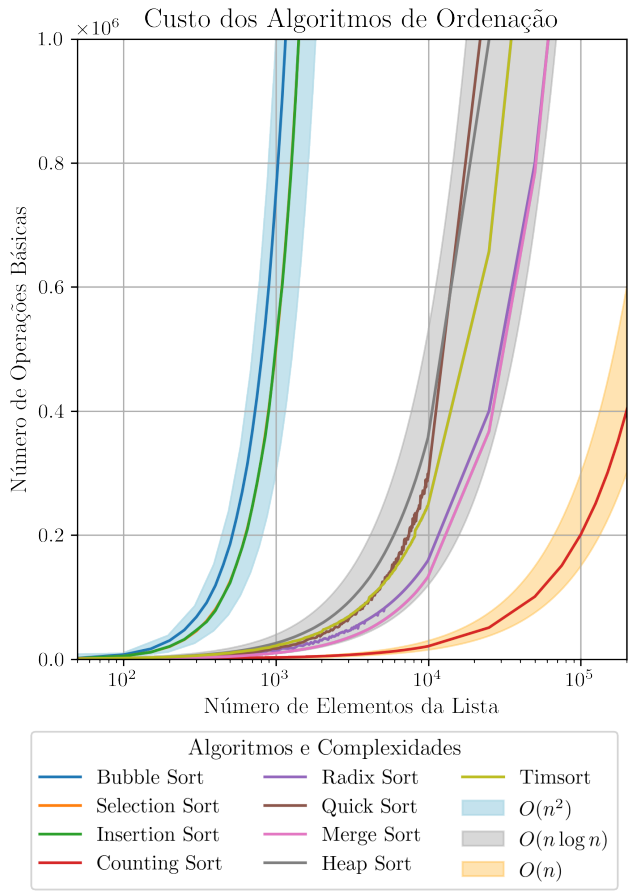

Figura 1: Análise do número de operações básicas dos diferentes algoritmos face a diferentes tamanhos de listas.

A partir da análise da figura 1, que contém os resultados práticos da implementação dos algoritmos analisados, torna-se possível então a comparação entre o comportamento empírico com a complexidade teórica.

No caso do Counting Sort, é possível verificar que a sua complexidade se comporta como $\mathcal{O}(n)$, uma vez que o valor máximo presente na lista ($k = 1000$) é significativamente menor do que os tamanhos das listas utilizadas nesta simulação. Assim, o termo adicional k não afeta substancialmente o crescimento do tempo de execução.

Relativamente aos algoritmos Bubble Sort, Selection Sort e Insertion Sort, observa-se um crescimento quadrático consistente com a complexidade teórica $\mathcal{O}(n^2)$. Nota-se que o desempenho do Selection Sort não se destaca visualmente no gráfico, por se sobrepor ao do Insertion Sort, evidenciando o mesmo comportamento assintótico.

Quanto aos restantes algoritmos, com exceção do Radix Sort, todos apresentam um crescimento compatível com a complexidade $\mathcal{O}(n \log n)$, conforme previsto teoricamente. Já o Radix Sort, cuja complexidade é dada por $\mathcal{O}(n \cdot b)$, onde b representa o número de dígitos necessários para representar o maior valor k, pode também ser expressa como $\mathcal{O}(n \cdot \log_b k)$, assumindo uma base b fixa. Na representações gráfica, observa-se que o seu desempenho tende a aproximar-se dos algoritmos de complexidade $\mathcal{O}(n \log n)$, devido à presença de um fator logarítmico multiplicativo que, apesar de ser pequeno, afeta o crescimento linear.

Este resultado demonstra que, apesar das complexidades teóricas associadas a cada algoritmo, o seu desempenho prático depende fortemente das características da lista a ser ordenada, como a distribuição dos valores ou a presença de padrões específicos nos dados.

IV. Conclusão

Neste trabalho, foram analisados e comparados nove algoritmos de ordenação, abrangendo abordagens clássicas (como Bubble Sort, Selection Sort e Insertion Sort), algoritmos mais eficientes baseados em paradigmas de dividir-para-conquistar (como Quick Sort, Merge Sort e Heap Sort), e algoritmos especializados como Counting Sort, Radix Sort e Timsort. Para cada algoritmo, foram descritos o funcionamento, a complexidade teórica, as vantagens e limitações, e o comportamento prático observado através de simulações em *Python*.

A análise empírica permitiu confirmar, de forma geral, as previsões teóricas. Algoritmos quadráticos apresentaram um crescimento acentuado com o aumento do tamanho da lista, tornando-se inviáveis para grandes volumes de dados. Por outro lado, algoritmos com complexidade $\mathcal{O}(n \log n)$ revelaram-se muito mais eficientes e escaláveis. O caso do Radix Sort destacou a importância de considerar não apenas a complexidade assintótica, mas também os fatores constantes ocultos e as condições específicas dos dados, uma vez que, apesar da sua complexidade quase linear, o seu desempenho prático aproxima-se frequentemente dos algoritmos $\mathcal{O}(n \log n)$ devido ao custo das múltiplas passagens.

Conclui-se, assim, que a escolha do algoritmo de ordenação mais adequado não deve ser feita apenas com base na complexidade teórica, mas também considerando as características concretas dos dados, o contexto de utilização e os requisitos práticos de desempenho e memória.

Referências