

Algoritmos de ordenação

Vinicius A. Matias

May 30, 2021

1 Introdução

Ordernar elementos de uma estrutura de dados é uma das tarefas mais curiosas e da computação. Diferentes implementações podem ser realizadas, das mais triviais às mais eficientes, assumindo heurísticas dos algoritmos ou não, e cada um dos algoritmos tem seu valor para um melhor entendimento da complexidade computacional.

2 Insertion Sort

O método de ordenação por inserção é um dos mais simples. Ele consiste em verificar de dois em dois elementos se o valor corrente é maior que o valor antecessor no arranjo, e se for, trocar e repetir o processo até o arranjo ficar ordenado.

Esse algoritmo pode ser pensado no caso de segurarmos um conjunto de cartas em uma mão e queiramos ordená-las. Aplicando o algoritmo da ordenação por inserção, verificamos a segunda carta, vemos se a primeira é maior que ela e se sim, trocamos as cartas. Na terceira carta vemos se a segunda é maior, e se for, verificamos se ela também é maior que a primeira. Esse processo é repetido para todas as cartas, passando por cada carta da esquerda para a direita e comparando da direita para a esquerda.

2.1 Implementação iterativa

A implementação do algoritmo iterativo pode ser vista na Listing 1.

Como a operação de interesse desse algoritmo é a comparação entre elementos do arranjo e o valor corrente, devemos identificar que existem dois laços envolvidos para computar essa operação. O laço mais externo roda entre $i = 0$ e $i < \text{len}(A)$, ou seja, $n - 1$ vezes. No melhor dos casos não haverá necessidade de trocar os elementos pois o arranjo já está ordenado, e nesse

caso só será realizada uma verificação por volta do laço, levando à $n - 1$ iterações no melhor caso.

Listing 1: Insertion Sort iterativo

```
1 def insertion_sort(A):
2     end = len(A)
3     i = 0
4     j = 0
5
6     while i < end:
7         value = A[i]
8         j = i
9         while j > 0 and A[j-1] > value:
10             A[j] = A[j-1]
11             j = j-1
12         A[j] = value
13         i += 1
14
15     return A
```

No pior caso, para cada uma das $n - 1$ voltas do laço será necessário verificar todos os elementos anteriores à posição atual (segundo laço). Isso implica que a primeira execução fará uma verificação, a segunda duas, a terceira três até a n -ésima, realizando n comparações. Somando o número de comparações teremos algo como $1 + 2 + 3 + \dots + n - 1$, que pode ser vista como uma soma de Progressão Aritmética. A soma desta PA que cresce de 1 em um pode ser definida como $\frac{n(0+n-1)}{2} = \frac{n^2-n}{2}$, isto é, o algoritmo de ordenação por inserção iterativo tem crescimento assintótico $\mathcal{O}(n^2)$.

2.2 Demonstração de crescimento assintótico $\mathcal{O}(n^2)$

$f(n) = \frac{n^2-n}{2} \in \mathcal{O}(n^2)$ se existem constantes $n_0 \geq 0$ e $c \geq 0$ que satisfazem a inequação:

$$0 \leq \frac{n^2-n}{2} \leq cn^2$$

Resolvendo $0 \leq \frac{n^2-n}{2}$ notamos que a inequação é verdadeira para qualquer $n \in \mathcal{R}$

Resolvendo $\frac{n^2-n}{2} \leq cn^2$:

$$\begin{aligned}\frac{n(n-1)}{2} &\leq cn^2 \\ \frac{n-1}{2} &\leq cn \\ \frac{-1}{2} &\leq cn - \frac{n}{2} \\ \frac{-1}{2} &\leq n(c - \frac{1}{2})\end{aligned}$$

Para a inequação ser verdadeira, $c - \frac{1}{2} \geq 0$ deve ser verdade, logo, $c \geq \frac{1}{2}$.

Assim, a inequação $\frac{-1}{2} \leq n(c - \frac{1}{2})$ é verdadeira para qualquer valor $n \geq 0$ e $c \geq \frac{1}{2}$, como exemplo:

$$n_0 = 1 \text{ e } c = 1$$

$$\text{Isso prova que } \frac{n^2-n}{2} \in \mathcal{O}(n^2)$$

2.3 Implementação recursiva

A listing 2 apresenta uma implementação recursiva do Insertion Sort.

O algoritmo é baseado na indução fraca, garantindo que sabe-se ordenar um arranjo com um elemento, pois ele já está ordenado (caso base). Cada chamada recursiva um dos $n - 1$ sub arranjos possíveis.

Listing 2: Insertion Sort recursivo

```

1  def insertion_sort_rec(A, n):
2      if n == 1: return
3
4      insertion_sort_rec(A, n-1)
5      i = n - 1
6      aux = 0
7
8      while i > 0 and A[i-1] > A[i]:
9          aux = A[i]
10         A[i] = A[i-1]
11         A[i-1] = aux
12         i -= 1
13
14     return A

```

Considerando que a operação de interesse é a comparação entre elementos do arranjo $A[i - 1] > A[i]$, podemos definir duas equações de recorrência, uma para o melhor caso e outra para o pior caso.

Melhor caso: O arranjo está ordenado, portanto serão feitas $n - 1$ chamadas recursivas para um arranjo de tamanho n , e uma comparação em cada uma dessas chamadas. Para o caso base não é feita nenhuma comparação entre elementos do arranjo.

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + 1, & n > 1 \end{cases}$$

A resolução dessa equação de recorrência diz que $T(n) = n - 1$

Pior caso: São feitas $n - 1$ comparação para cada n passado na recorrência.

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

E a resolução dessa equação de recorrência resulta em $T(n) = (n^2 - n)/2$, e que $T(n) \in \mathcal{O}(n^2)$

2.4 Equação de recorrência para o pior caso

Demonstraremos que o resultado da equação de recorrência abaixo é $T(n) \in \mathcal{O}(n^2)$

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

Notando que $T(n)$ varia as chamadas recursivas de 1 em 1, calcularemos as equações de recorrência para $n - 1$, $n - 2$ e $n - 3$

$$T(n) = T(n-1) + n - 1$$

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

$$T(n-2) = T(n-2-1) + n-2-1 = T(n-3) + n-3$$

$$T(n-3) = T(n-3-1) + n-3-1 = T(n-4) + n-4$$

Podemos aplicar esses valores na equação de recorrência $T(n)$:

$$T(n) = T(n-1) + n - 1$$

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

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

$$T(n) = T(n-3) + n - 3 + 2n - 2 - 1$$

$$T(n) = T(n-3) + 3n - 3 - 2 - 1$$

$$T(n) = T(n-4) + n - 4 + 3n - 3 - 2 - 1$$

$$T(n) = T(n-4) + 4n - 4 - 3 - 2 - 1$$

[...]

$$T(n) = T(n-i) + in + \sum_{j=1}^i -j$$

A operação $\sum_{j=1}^i -j$ é uma soma de Progressão Aritmética, podendo ser reescrita como $\frac{i*(-1-i)}{2}$

$$\text{Assim, } T(n) = T(n-i) + in + \frac{i*(-1-i)}{2}$$

Quando $i = n$:

$$T(n) = T(n-i) + in + \frac{i*(-1-i)}{2}$$

$$T(n) = T(n-n) + n^2 + \frac{n*(-1-n)}{2}$$

$$T(n) = T(0) + n^2 + \frac{(-n-n^2)}{2}$$

$$T(n) = n^2 + \frac{(-n-n^2)}{2}$$

$$T(n) = \frac{(-n - n^2 + 2n^2)}{2}$$

$$T(n) = \frac{(n^2 - n)}{2}$$

E como foi demonstrado no item 2.2, $\frac{(n^2 - n)}{2} \in \mathcal{O}(n^2)$, portanto:

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

3 Selection Sort

A ordenação por seleção parte do último elemento de um arranjo e compara com todos os anteriores para verificar se há um elemento maior que ele e, caso exista, capturar o maior de todo o subarranjo. O índice do maior valor do subarranjo é capturado e é comparado com o índice elemento que se partiu a ordenação (da direita para a esquerda, então, o último, penúltimo etc) para verificar se são iguais, caso forem iguais não há necessidade de trocar de posições pois o maior elemento do subarranjo já está mais à direita do arranjo. Caso sejam diferentes, o algoritmo troca a posição do então último elemento do arranjo pelo maior elemento encontrado.

3.1 Implementação iterativa

Uma implementação em Python do algoritmo de seleção pode ser vista na Listing 3.

Listing 3: Selection Sort iterativo

```

1 def selection_sort(A):
2     n = len(A)
3     fim = n-1
4
5     while fim > 0:
6         max = fim
7         for j in range(fim):
8             if A[j] > A[max]:
9                 max = j
10        if fim != max:
11            temp = A[fim]
12            A[fim] = A[max]
13            A[max] = temp
14        fim -= 1
15
16    return A
```

A operação de interesse aqui é a comparação entre cada elemento de um subarranjo com o máximo encontrado até então. Note que tanto o loop mais externo quanto o mais interno (que compreende a comparação $A[j] > A[\text{max}]$) são

executados sempre a mesma quantidade de vezes para um mesmo n , implicando que o melhor e o pior caso sejam iguais.

O loop externo é executado $n - 1$ vezes e a quantidade de iterações do loop interno segue uma progressão aritmética ($n - 1, n - 2, \dots, 2, 1$). A soma dessas iterações é dada por $\frac{(n-1)*(n-1+1)}{2}$, logo, $\frac{n^2-n}{2}$ comparações para qualquer caso. Isso implica que esta implementação $\in \Theta(n^2)$

3.2 Implementação recursiva

Uma implementação recursiva do algoritmo de ordenação por seleção é exibido na Listing 4.

Listing 4: Selection Sort recursivo

```

1 def selection_sort_rec(A, n):
2     if n == 1: return A
3
4     max = n-1
5     for i in range(n):
6         if A[i] > A[max]:
7             max = i
8
9     if max != n-1:
10        temp = A[max]
11        A[max] = A[n-1]
12        A[n-1] = temp
13
14    return selection_sort_rec(A, n-1)
```

Partindo do caso base, sabe-se ordenar um arranjo de apenas um elemento (é o próprio arranjo). Para se ordenar para mais um elemento devem ser seguidas as diretrizes do algoritmo, como foi exibido acima. A equação de recorrência para esse ordenador (para o número de comparações entre elementos do arranjo) pode ser definida como:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n > 1 \end{cases}$$

Levando à $T(n) = \frac{n^2-n}{2}$ e, tanto no melhor quanto no pior caso, à uma implementação $\in \Theta(n^2)$.

4 Bubble Sort

O Bubble Sort é possivelmente o método de ordenação mais simples dos aqui estudados, contudo também o método que tem pior desempenho em aplicações reais. O algoritmo consiste

em passar por todos os possíveis pares de elementos e comparar se um é maior que o outro.

4.1 Implementação iterativa

O método bolha consiste apenas em uma troca de elementos em pares, uma das implementações iterativas possíveis está na Listing 5.

Listing 5: Bubble Sort iterativo

```

1  def bubble_sort(A, n):
2      i = n-1
3
4      while i > 0:
5          j = 1
6          while j <= i:
7              if A[j-1] > A[j]:
8                  temp = A[j-1]
9                  A[j-1] = A[j]
10                 A[j] = temp
11                 j += 1
12             i -= 1
13
14     return A

```

Sendo a comparação de interesse destacada em $A[j-1] > A[j]$, assim como no selection sort essa operação será realizada $n-1, n-2, \dots, 2, 1$ vezes, mudando agora quais pares de elementos são comparados. A complexidade assintótica se mantém como $\Theta(n^2)$, pois a quantidade de operações segue a mesma soma de PA que resulta em $\frac{(n^2-n)}{2}$.

4.2 Implementação recursiva

Listing 6: Bubble Sort recursivo

```

1  def bubble_sort_rec(A, n):
2      i = 1
3      while i < n:
4          if A[i] < A[i-1]:
5              temp = A[i]
6              A[i] = A[i-1]
7              A[i-1] = temp
8              i += 1
9      return bubble_sort(A, n-1)

```

Ainda que não seja comum, o bubble sort pode ser implementado recursivamente, trocando o laço mais externo por um chamada à própria função para $n-1$. A equação de recorrência é a mesma do selection sort recursivo, levando à uma complexidade $\mathcal{O}(n^2)$.

5 Merge Sort

O Merge Sort é um método de ordenação que utiliza a técnica de Divisão e Conquista também chamado de ordenação por intercalação. Ele parte da hipótese de indução forte de que sabemos ordenar um conjunto de elementos no intervalo $[1, n]$. A ordenação é feita dividindo o arranjo no meio para gerar dois subarranjos (esquerdo e direito). Para cada subarranjo será feito o mesmo processo de dividir entre esquerdo e direito até uma chamada para um arranjo de apenas um elemento, neste caso armazenaremos o valor e poderemos partir para a chamada do lado direito. Após algumas chamadas recursivas, teremos um elemento do arranjo esquerdo e um elemento do arranjo direito. Estes dois serão passados para a função merge, que será responsável por comparar os dois elementos e ordená-los.

Esse processo cresce conforme a pilha é desempilhada, havendo comparações dos elementos do arranjo esquerdo com o direito até um limite de $n/2$ comparações, que é o caso da primeira chamada recursiva realizada. O Merge Sort permite que ordenemos arranjos "da esquerda" e "da direita", e isso permite que na próxima comparação, os dois subarranjos passados para o merge estejam ordenados, cabendo à cada chamada do merge comparar o novo esquerdo e direito e ordená-los.

O Merge Sort normalmente é implementado em duas funções: uma para a ordenação de dois arranjos diferentes e gerar um único (*merge()*) e uma para realizar as chamadas recursivas para dividir os arranjos (*merge_sort()*).

Listing 7: Merge sort

```

1  def merge_sort(A):
2      if len(A) > 1:
3          meio = len(A) // 2
4          esq = merge_sort(A[:meio])
5          dir = merge_sort(A[meio:])
6          A = merge(esq, dir)
7      return A

```

Diferentes implementações podem ser obtidas para retornar o mesmo resultado. Muitos materiais utilizam variáveis de início e fim para gerir o meio do arranjo, e sendo passadas nas chamadas recursivas. Dependendo das estruturas que a linguagem fornece, a implementação pode ser mais sucinta, como é o caso de Python. O pseudocódigo do livro Algoritmos (Cormen et al.), já mencionados em outro relatório fornece

um pseudocódigo do Merge Sort que tipicamente rege as implementações em Java e C. Em Python podemos utilizar os recursos da linguagem para pegar os índices que separam o arranjo esquerdo e o arranjo direito quando temos o meio calculado.

Listing 8: Merge

```

1  def merge(esq, dir):
2      i, j = 0, 0
3      ordenados = list()
4
5      while i < len(esq) and j < len(dir):
6          if esq[i] <= dir[j]:
7              ordenados.append(esq[i])
8              i += 1
9          else:
10             ordenados.append(dir[j])
11             j += 1
12
13     if i < len(esq):
14         ordenados += esq[i:]
15
16     elif j < len(dir):
17         ordenados += dir[j:]
18
19     return ordenados

```

O merge funciona verificando o arranjo esquerdo e o direito para identificar quais valores são menores (do arranjo esquerdo ou do direito) e até quando. Os valores são armazenados em uma lista que será retornada ao no método *merge.sort()*. Também são verificados se houveram valores que não foram inseridos na comparação, ou seja, os valores do arranjo esquerdo ou direito que eram maiores que o maior valor do outro subarranjo.

Para analisar a complexidade do merge sort precisamos notar que ele realiza duas chamadas recursivas para $n/2$ e uma chamada ao merge para cada n . Considerando a operação de interesse a comparação $esq[i] \leq dir[j]$ do laço da função merge, teremos um custo para essa função de $n - 1$ no pior caso e $n/2$ no melhor caso. A equação de recorrência para o pior caso é:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T(n/2) + n - 1, & n > 1 \end{cases}$$

E calculando essa equação de divisão e conquista pelo Teorema Mestre (cláusula 2) descobrimos que a complexidade assintótica do merge sort para todos os casos $\in \Theta(n * \log n)$. Vale lembrar que para a generalização é necessário calcular a equação de recorrência para o melhor

caso também, que terá a mesma complexidade que o pior caso.

É importante observar que ainda que a complexidade do Merge Sort seja menor que a dos outros estudados até então, esse algoritmo tem um custo de criar novos arranjos para manipular os elementos durante a execução do método.

6 Quick Sort

O quick sort segue a mesma hipótese de indução forte que o merge sort (sabemos ordenar um novo elemento em qualquer arranjo até então ordenado). O método consiste em dividir o arranjo em uma partição central onde todos os elementos à esquerda dela são menores que esse valor e à direita são maiores. O processo é repetido até termos apenas um elemento para procurar a partição, e então é realizada a volta das chamadas recursivas. O método de ordenação é desenvolvido utilizando duas funções:

Listing 9: Quick Sort

```

1  def quick_sort(A, ini, fim):
2      if ini < fim:
3          p = particao(A, ini, fim)
4          quick_sort(A, ini, p-1)
5          quick_sort(A, p+1, fim)
6      return A

```

Como pode ser visto, há um processo de divisão custoso (chamada à função partição) e um processo de conquista pela volta às chamadas recursivas. Não há um custo para junção, como visto no merge sort. A função em questão realiza as chamadas recursivas para o mesmo arranjo à esquerda da partição e à direita dela, atualizando diretamente no arranjo passado na entrada. A função *particao()* que realiza a magia é mostrada na sequência.

Para identificar o elemento que divide o arranjo entre maiores e menores que um valor é chamada a função de partição e retornado o tal elemento da divisão, chamado de pivô. Com um arranjo qualquer nós começamos definindo os índices **válidos** que limitarão o início e fim do processo de comparação no arranjo (diferente do merge sort, os índices de fim do arranjo deve ser uma posição válida, e não o tamanho).

Enquanto os índices forem diferentes haverá um processo de comparar os elementos à esquerda do arranjo com o pivô para verificar se são menores que ele, e a mesma coisa para os

elementos à direita serem maiores. Caso à esquerda seja realmente menor, está tudo certo por enquanto e vamos para a próxima posição. Caso o elemento à esquerda seja maior que o pivô (isto é, deve ir para a direita do arranjo) devemos verificar se o elemento à direita é maior ou menor que o pivô. Se o elemento à direita (j) for maior, ele está na posição correta e passamos a comparação do elemento à esquerda (i) para o elemento à esquerda do mais à direita (j-1). Se o da esquerda for maior que o pivô e o à direita for menor, trocamos eles de lugar e atualizamos seus índices. Ao fim é criado atualizado o pivô como o elemento que ficou ao meio da "ordenação" atual. Esse algoritmo de partição também é chamado de partição de Hoare.

Listing 10: Partição

```

1  def particao(A, ini, fim):
2      pivo = A[fim]
3      i = ini
4      j = fim - 1
5
6      while i <= j:
7          if A[i] <= pivo:
8              i += 1
9          elif A[j] > pivo:
10             j -= 1
11         else:
12             aux = A[i]
13             A[i] = A[j]
14             A[j] = aux
15             i += 1
16             j -= 1
17
18     A[fim] = A[i]
19     A[i] = pivo
20     return i

```

Essa implementação da partição tem como melhor caso para o número de comparações entre o pivô e um elemento do arranjo duas possibilidades. A primeira é quando o pivô é o maior elemento do subarranjo (realiza apenas a primeira comparação do método partição), e a segunda é quando em uma passada do laço realiza as duas comparações e cai no *else*, isto é, quando deve haver uma troca e, portanto, avança tanto do lado esquerdo quanto do lado direito do subarranjo (pula uma das comparações que poderiam ser feitas). Para o primeiro melhor caso serão feitas $n - 1$ comparações, e para o segundo caso serão feitas também $2(n - 1)/2 = n - 1$ comparações.

O pior caso é quando o primeiro *if* é verdadeiro e o segundo é falso, ou seja, quando o

pivô é menor que todos os elementos à esquerda e menor que todos à direita (logo, o menor elemento do subarranjo). Realiza-se nesse caso $2n - 2$ comparações. Tanto no melhor quanto no pior caso, $T(n) \in \Theta(n)$

O quick sort como um todo tem uma equação de recorrência para o melhor caso definida como:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T(n/2) + n - 1, & n > 1 \end{cases}$$

Com $T(n) \in \Theta(n * \log n)$

E como pior caso:

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n - 1) + 2n - 2, & n > 1 \end{cases}$$

Com $T(n) \in \Theta(n^2)$

Uma prova mais longa pode provar que o caso médio do quick sort $\in \Theta(n * \log n)$

7 Heap Sort

Heaps em computação são utilizados como um espaço de memória para armazenar valores e como uma estrutura de dados para uma regra particular. No Heaps sort, essa estrutura é auxiliar ao método. A ordenação desse método utiliza indução fraca (a mesma do selection sort), mas a estrutura Heap utiliza indução forte. A organização do Heap permite otimizar a ordenação no método, contrariando a ordenação por seleção. O Heap vai simular uma árvore binária completa em um arranjo.

Cada posição do arranjo representa um nó, todos os níveis são completos, com exceção das folhas. Para encontrar as relações de um nó na posição i segue-se o método:

$$PAI : \lfloor (i - 1)/2 \rfloor$$

$$ESQUERDA : 2 * i + 1$$

$$DIREITA : 2 * i + 2$$

Aqui utilizaremos o conceito de Heap máximo, onde um nó pai é maior ou igual que seus filhos. Em um heap de altura h o número de nós varia de $2^h \leq n \leq 2^{h+1}$, sendo a altura definida como a distância até as folhas para um nó, portanto a altura de um heap de n nós é $\lceil \log 2n \rceil$.

A implementação do Heap será baseada em duas funções. A primeira é uma função recursiva que verifica se um nó pai é maior que seus filhos à esquerda e à direita. Caso seja, trocam-se de valores e se faz a verificação novamente (por recursão para o novo nó pai).

Listing 11: Manutenção do Heap

```

1 def refazer_heap_max(A, i_pai, n_heap):
2     esq = 2*i_pai + 1
3     dir = 2*i_pai + 2
4
5     if esq < n_heap and A[esq] > A[i_pai]:
6         maior = esq
7     else:
8         maior = i_pai
9
10    if dir < n_heap and A[dir] > A[maior]:
11        maior = dir
12
13    if maior != i_pai:
14        aux = A[maior]
15        A[maior] = A[i_pai]
16        A[i_pai] = aux
17    refazer_heap_max(A, maior, n_heap)

```

Essa função será passada para a construção do Heap (máximo). O primeiro nó pai é o que está no meio do arranjo, e a partir daí é chamado o método anterior para reorganizar o heap, ou seja, deixar um nó pai maior que seus filhos. Esse processo vai ser realizado até o primeiro elemento do arranjo ($n/2$ iterações).

Listing 12: Construção do Heap

```

1 def construir_heap_max(A):
2     n_heap = len(A)
3     i = (n_heap // 2) - 1
4     while i >= 0:
5         refazer_heap_max(A, i, n_heap)
6         i -= 1

```

Com o Heap implementado, podemos utilizar o Heap Sort para ordenar os valores de uma lista / arranjo. O método se baseia no Selection Sort, mas a estrutura do heap influencia em uma redução na quantidade de iterações necessárias, logo, na complexidade assintótica.

Listing 13: Heap Sort

```

1 def heap_sort(A):
2     construir_heap_max(A)
3     n_heap = len(A)
4     i = len(A) - 1
5     while i > 0:
6         aux = A[0]
7         A[0] = A[i]
8         A[i] = aux
9         n_heap -= 1
10        i -= 1
11    refazer_heap_max(A, 0, n_heap)

```

A operação de interesse para a análise da complexidade assintótica do Heap Sort

é a comparação entre elementos do arranjo, passo esse que acontece no método `refazer_heap_max()`. As comparações $A[esq] > A[i_pai]$ e $A[dir] > A[maior]$ acontecem, no pior caso, em $2/3$ dos elementos, apenas uma vez. O pior caso geral a recursão acontece para metade dos elementos, fazendo duas verificações por vez.

$$T(n) = \begin{cases} 0, & n \leq 1 \\ T(n/2) + 2, & n > 1 \end{cases} \in \mathcal{O}(\log n)$$

A função `construir_heap_max()` pode ser reformulada como uma função recursiva para fazer uma equação de recorrência. Neste caso seria feita uma chamada à esquerda e outra à direita, ambas para metade dos elementos. O custo seria acrescido pelas comparações do método de manutenção do heap.

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T(n/2) + \mathcal{O}(\log n), & n > 1 \end{cases}$$

$$T(n) \in \Theta(n)$$

O Heap Sort tem o custo $\Theta(n)$ para criar o Heap e o custo $\mathcal{O}(\log n)$ para realizar a manutenção do Heap. Assim, o número de operações n_op pode ser identificado como:

$$n_op \in \Theta(n) + \mathcal{O}(\log n), \text{ e como } \Theta(n) \text{ cresce com } \mathcal{O}(n):$$

$$n_op \in \mathcal{O}(n) + \mathcal{O}(\log n)$$

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

8 Counting Sort

A ordenação por comparações (algoritmos vistos até aqui) não conseguem ordenar um arranjo em tempo linear. Para tal é importante buscar adaptações ao método de pensar a ordenação de arranjos e assim buscar algum método de complexidade $\mathcal{O}(n)$. Uma das possibilidades é por meio da ordenação por contagem.

O Counting Sort deve receber, além do arranjo, um parâmetro adicional $k \in \mathcal{O}(n)$, isto é, os n elementos de um arranjo estão no intervalo $0 \leq k$. É possível realizar modificações para o Counting Sort aceitar valores negativos (como referenciando eles ao fim do arranjo), mas esse método não é focado para esses casos. A ordenação por contagem consegue ter complexidade linear, mas a restrição de que os elementos estejam no intervalo entre 0 e n tipicamente deixa o método para contextos bem específicos,

mas quando entram neste escopo acabam se tornando interessantes.

Outro problema desse algoritmo é que se torna necessário alocar em memória espaço para todos elementos possíveis de se entrar no arranjo ordenado (entre 0 e k), mesmo que não sejam utilizados.

Um ponto positivo da implementação é que ela é estável, isto é, elementos de mesmo valor que aparecem em posições diferentes no arranjo original são ordenados na mesma sequência.

Listing 14: Counting Sort

```

1 def counting_sort(A, k):
2     n = len(A)
3     ordenado = [None] * n
4     quantidade = [0] * k
5
6     for j in range(n):
7         quantidade[A[j]] += 1
8     for i in range(1, k):
9         quantidade[i] += quantidade[i-1]
10    for j in reversed(range(0, n)):
11        ordenado[quantidade[A[j]]-1] = A[j]
12        quantidade[A[j]] -= 1
13
14    return ordenado

```

A ordenação por contagem não realiza nenhuma comparação, logo, a operação de interesse é a de atribuição de valores. Analisando o algoritmo podemos perceber que a quantidade de operações de interesse $\in \Theta(n + k)$ e pela restrição $k \in \mathcal{O}(n)$, que o método ideal utiliza, o algoritmo Counting Sort tem crescimento assintótico $\mathcal{O}(n)$.

9 Radix Sort

O último método de ordenação a ser analisado é o Radix Sort, ou ordenação por raiz. Esse método de ordenação linear baseia-se na extração de cada parte de um valor (normalmente uma string) a partir do dígito menos significativo (de trás pra frente). Para cada um dos dígitos será chamado um ordenador com complexidade $\mathcal{O}(n)$, visto que não faz sentido utilizar um ordenador de custo não linear em um ordenador que visa custo linear. Aqui exemplificaremos o uso do Radix Sort pela ordenação de strings de tamanho fixo com auxílio dos valores unicode de seus dígitos (limitando o universo aos primeiros 256 valores unicode). O método usará como ordenador uma adaptação do counting sort que ordena strings.

Listing 15: Radix Sort para strings de tamanho fixo

```

1 def radix_sort(A, n_digitos):
2     n_unicode = 256
3     for i in reversed(range(n_digitos)):
4         A = counting_sort_str(A, i, n_unicode)
5     print(A)
6     return 0

```

O método recebe o vetor de strings, com cada entrada de tamanho $n_digitos$. Para cada dígito (do menos significativo ao mais significativo) é chamado o counting sort para ordenar a string pelos valores unicode.

Listing 16: Counting Sort para valores unicode de caracteres

```

1 def counting_sort_str(A, indice, k):
2     n = len(A)
3     ordenado = [''] * n
4     quantidade = [0] * k
5
6     for j in range(n):
7         quantidade[ord(A[j][indice])] += 1
8     for i in range(1, k):
9         quantidade[i] += quantidade[i-1]
10    for j in reversed(range(0, n)):
11        ordenado[quantidade[ord(A[j][indice])]-1] = A[j]
12        quantidade[ord(A[j][indice])] -= 1
13    return ordenado

```

O comando `ord` em python retorna o valor unicode de um caractere. O método acima permite executar o counting sort sobre caracteres de uma string e retornar o arranjo ordenado com base em um dos dígitos das strings.

A complexidade do Radix Sort, portanto, é dependente da complexidade do ordenador secundário chamado. Utilizando um ordenador de complexidade $\Theta(n)$ como o Counting Sort (considerando, obviamente, que suas restrições para tamanho de k tenham sido obedecidas), a complexidade do Radix Sort será $\Theta(d * \Theta(n))$, onde d é a quantidade de divisões feitas no radix sort (no exemplo dado, seria a quantidade de dígitos de uma string). Se, e somente se d for um valor independente de n , poderemos considerar $d \in \Theta(1)$, e assim o Radix Sort terá complexidade $\Theta(n)$.