Algoritmos de ordenação: Heap sort

Gabriel Grahl Musskopf
gmusskopf@edu.unisinos.br

Estruturas Avançadas de Dados I, UNISINOS
Prof. Marcio Garcia Martins
Outubro, 2023

Objetivo

Este texto tem como objetivo elaborar sobre o algoritmo de ordenação *heap sort*, introduzindo e abordando o contexto histórico, apresentando as características e funcionamento em múltiplos conjuntos de entradas e sua complexidade temporal.

Introdução

O algoritmo *heap sort* foi desenvolvido em 1964, por J.W.J Williams, e faz parte de algoritmos de ordenação por seleção. Essa família de algoritmos é baseada no princípio de seleção repetida da maior ou menor chave da lista não ordenada de elementos restantes, e adicionados em uma lista de elementos ordenados. Outros algoritmos conhecidos desta família são o *selection sort* e *smooth sort*.

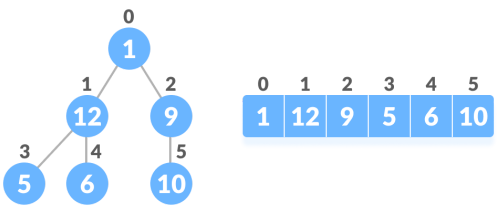
Heap sort é baseado na estrutura de dados *heap*, que também dá origem ao nome do algoritmo. Para a compreensão do algoritmo abordado, farei uma introdução do conceito de *heap*.

Heap

Na computação, a estrutura de dados *heap* é uma estrutura baseada em árvore, muito utilizada na implementação de filas com prioridade, pois baseia-se na ideia de remover repetidamente o elemento de maior ou menor valor, dependendo do tipo da estrutura. Os possíveis tipos são: heap *máxima*, onde o nó tem um valor maior que o filho, ou *heap mínima*, onde o nó tem valor menor que o filho.

Uma implementação comum para essa estrutura é uma árvore binária, um nó raíz com referências para uma sub-árvore à esquerda e direita. Essa implementação foi desenvolvida pelo mesmo autor do algoritmo *heap sort* durante sua elaboração.

Figura 1



Fonte: Programiz. Heap Sort Algorithm

Conforme visualizado na figura 1, a estrutura pode ser representada por uma árvore binária, porém, é implementada utilizando um *array*, sem a necessidade de uma estrutura de árvores com referência e recursão. Isso se deve a característica da árvore binária de que todos os filhos podem ser encontrados dada uma posição i do pai utilizando a equação $D = 2i + 2$, para o filho à direita, e $E = 2i + 1$, para o filho à esquerda. O caminho inverso também é válido, bastando rearranjar a fórmula.

Algoritmo

O algoritmo de ordenação abordado tem objetivamente duas diretivas:

1. Organizar o *heap*, também chamado de *heapify*, lembrando da variação *max* ou *min heap*;
2. Trocar o elemento da última posição não ordenada com o elemento da primeira posição. Com esse movimento, o elemento posto no fim está na sua posição final;
3. Remover o elemento posicionado, e realizar a primeira diretiva novamente para organizar o *heap*.

O algoritmo completo usado na implementação pode ser encontrado no apêndice 1, visto que contém informações adicionais à lógica do *heapsort* que trariam complexidade adicional nesse momento. As funções responsáveis pela ordenação são encontradas abaixo:

Figura 2

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    random = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if random < n and arr[largest]
< arr[random]:
        largest = random

    if largest != i:
        arr[i], arr[largest] =
arr[largest], arr[i]
        heapify(arr, n, largest)

def heapsort(arr):
    n = len(arr)

    for i in range(n//2, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0],
arr[i]
        heapify(arr, i, 0)
```

Fonte: o autor

Complexidade

Para a análise da complexidade, faz sentido pensar no algoritmo como duas etapas: organizar o *heap* e realizar a seleção.

Na primeira fase, no caso de um *max heap*, o pior caso é quando um elemento na raiz deve ser movido até a folha, tendo a complexidade idêntica a árvore binária, $O(log\ n)$. Essa organização acontece para, no máximo, $\frac{n}{2}$ elementos, obtendo então $T(n) = \frac{n}{2} log\ n$, e complexidade assintótica de $O(n\ log\ n)$.

Na fase de seleção, o elemento da raiz é trocado com o último elemento do *heap*, o que requer tempo constante, e após isso, o *heap* é reorganizado, terminando com a mesma complexidade da etapa anterior.

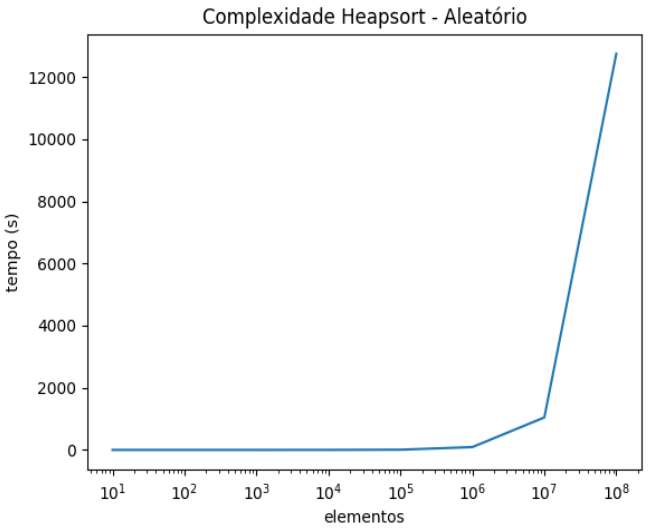
Como os eventos ocorrem em sequência, a complexidade assintótica final é $O(n\ log\ n)$.

Características

Para a demonstração deste algoritmo será feita a comparação em três cenários: array distribuído de forma aleatória e ordenado de forma crescente e decrescente.

A ordenação foi aplicada para X amostras, chamadas de S, com o tamanho de cada array dado por $N(i) = 10^i$, sendo $1 \leq i \leq S$. O tempo do algoritmo no ambiente que foi executado está explicitado nos gráficos abaixo

Figura 3



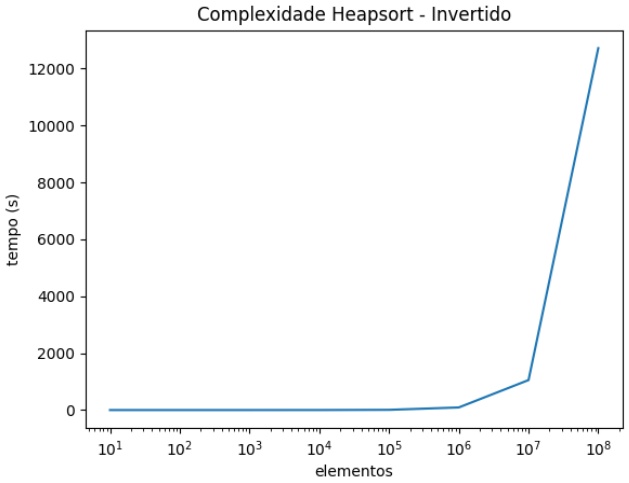
Fonte: o autor

Figura 4

Elementos	Tempo
10	0,5us
100	1 ms
1000	19 ms
10000	567 ms
100000	6,9 s
1000000	1 min 31 s
10000000	17 min 25 s
100000000	3h 32min 31 s

Fonte: o autor

Figura 5



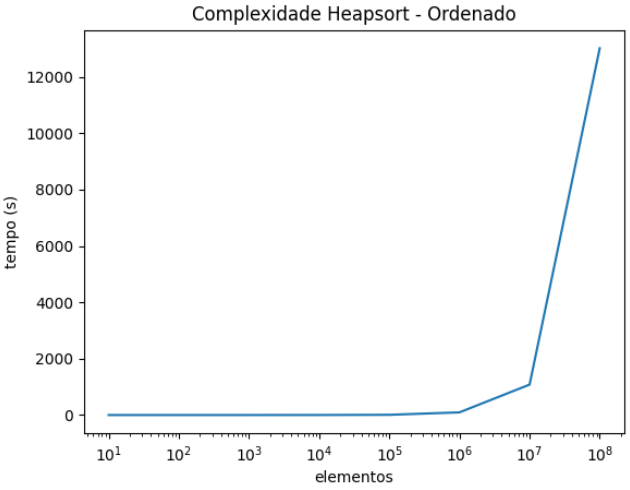
Fonte: o autor

Figura 6

Elementos	Tempo
10	0,5us
100	1 ms
1000	20 ms
10000	567 ms
100000	7 s
1000000	1 min 31 s
10000000	17 min 33 s
100000000	3h 32min 1 s

Fonte: o autor

Figura 7



Fonte: o autor

Figura 8

Elementos	Tempo
10	0,1ms
100	1,1 ms
1000	23 ms
10000	545 ms
100000	7,37 s
1000000	1 min 34 s
10000000	17 min 59 s
100000000	3h 36min 57 s

Fonte: o autor

Referências

[1] Williams, J. W. J. (1964), 'Algorithm 232: Heapsort', Communications of the ACM 7 (6), 347–348.

[2] Kumar, Suman & Chakraborty, Soubhik. (2009), Empirical Study on the Robustness of Average Complexity & Parameterized Complexity Measure for Heapsort Algorithm.

[3] Hulín, Matej. (2017), Performance analysis of Sorting Algorithms

[4] Heap Sort Algorithm. programiz. https://www.programiz.com/dsa/heap-sort Acessado em 26 out. 2023

Apêndice

[1] Algoritmo completo usado na implementação. Código fonte pode ser encontrado em: https://github.com/gabrielmusskopf/uni/blob/main/estrutura-de-dados-avancada/heapsort.py