

HeapSort

O que é o HeapSort?

O HeapSort é um algoritmo de ordenação eficiente que utiliza uma estrutura de dados chamada **heap** (montículo). É um algoritmo **híbrido** que combina as vantagens da ordenação por seleção com a eficiência de uma estrutura de dados especializada.

Conceitos Fundamentais

1. O que é um Heap?

Um heap é uma árvore binária completa que satisfaz a **propriedade de heap**:

- **Heap máximo**: O valor de cada nó é maior ou igual ao valor de seus filhos
- **Heap mínimo**: O valor de cada nó é menor ou igual ao valor de seus filhos

2. Representação do Heap

Um heap é normalmente representado como um **array**, onde:

- Raiz está no índice 0
- Filho esquerdo de i está em $2*i + 1$
- Filho direito de i está em $2*i + 2$
- Pai de i está em $(i-1)//2$

Funcionamento do HeapSort

O algoritmo possui duas fases principais:

Fase 1: Construção do Heap

Transformamos o array em um heap máximo.

Fase 2: Ordenação

Extraímos repetidamente o maior elemento (raiz) e reconstruímos o heap.

Implementação em Python

python

```
def heapify(arr, n, i):
    """
    Transforma uma subárvore em heap máximo
    arr: array a ser ordenado
    n: tamanho do heap
    i: índice do nó raiz da subárvore
    """
    maior = i # Inicializa o maior como raiz
    esquerda = 2 * i + 1
    direita = 2 * i + 2

    # Verifica se o filho esquerdo existe e é maior que a raiz
    if esquerda < n and arr[esquerda] > arr[maior]:
        maior = esquerda

    # Verifica se o filho direito existe e é maior que o maior até agora
    if direita < n and arr[direita] > arr[maior]:
        maior = direita

    # Se o maior não é a raiz, troca e continua heapificando
    if maior != i:
        arr[i], arr[maior] = arr[maior], arr[i] # Troca
        heapify(arr, n, maior) # Heapifica recursivamente a subárvore

def heapsort(arr):
    """
    Implementação do algoritmo HeapSort
    """
    n = len(arr)

    # Fase 1: Construção do Heap Máximo
    # Começa do último nó não-folha até a raiz
    print("Fase 1: Construção do Heap Máximo")
    for i in range(n // 2 - 1, -1, -1):
```

```

    print(f"Heapificando a partir do índice {i}: {arr}")
    heapify(arr, n, i)
    print(f"Após heapificar: {arr}")
    print("-" * 50)

# Fase 2: Extração dos elementos um por um
print("Fase 2: Extração e Ordenação")
for i in range(n - 1, 0, -1):
    # Move a raiz (maior elemento) para o final
    arr[i], arr[0] = arr[0], arr[i]
    print(f"Movendo maior elemento para posição {i}: {arr}")

    # Heapifica o heap reduzido
    heapify(arr, i, 0)
    print(f"Heap após reconstrução: {arr}")
    print("-" * 50)

# Exemplo prático passo a passo
def exemplo_detalhado():
    print("=== EXEMPLO DETALHADO DO HEAPSORT ===\n")

    # Array de exemplo
    arr = [4, 10, 3, 5, 1]
    print(f"Array original: {arr}")
    print("\n" + "="*60 + "\n")

    heapsort(arr)

    print(f"\nArray ordenado: {arr}")

# Versão simplificada para uso prático
def heapsort_simples(arr):
    """
    Versão simplificada do HeapSort
    """
    n = len(arr)

    # Constrói o heap máximo
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extrai elementos um por um
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # Troca

```

```

        heapify(arr, i, 0) # Heapifica o heap reduzido

    return arr

# Teste com visualização gráfica
def visualizar_heap(arr, titulo=""):
    """
    Função auxiliar para visualizar o heap como árvore
    """
    print(f"{titulo}: {arr}")
    n = len(arr)
    nivel = 0
    i = 0

    while i < n:
        elementos_nivel = 2 ** nivel
        fim = min(i + elementos_nivel, n)
        nivel_str = " " * (n // 2 - nivel)
        print(nivel_str + " ".join(f"{arr[j]:2}" for j in range(i, fim)))
        i = fim
        nivel += 1
    print()

# Exemplo com visualização
def exemplo_com_visualizacao():
    print("=== HEAPSORT COM VISUALIZAÇÃO ===\n")

    arr = [4, 10, 3, 5, 1, 8, 7]
    print("Array original:")
    visualizar_heap(arr, "Estado inicial")

    n = len(arr)

    # Construção do heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
        visualizar_heap(arr, f"Após heapificar a partir de índice {i}")

    # Ordenação
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        visualizar_heap(arr, f"Após mover maior para posição {i}")
        heapify(arr, i, 0)
        visualizar_heap(arr, f"Após reconstruir heap")

```

```

    print("Array final ordenado:", arr)

# Executar exemplos
if __name__ == "__main__":
    # Exemplo detalhado
    exemplo_detalhado()

    print("\n" + "="*80 + "\n")

    # Exemplo com visualização
    exemplo_com_visualizacao()

    # Teste prático
    print("\n" + "="*80 + "\n")
    print("TESTE PRÁTICO:")
    teste = [64, 34, 25, 12, 22, 11, 90]
    print(f"Antes: {teste}")
    resultado = heapsort_simples(teste.copy())
    print(f"Depois: {resultado}")

```

Análise de Complexidade

Complexidade de Tempo:



- **Pior caso:** $O(n \log n)$
- **Melhor caso:** $O(n \log n)$
- **Caso médio:** $O(n \log n)$



Complexidade de Espaço:

- **$O(1)$** - Ordenação in-place (não usa espaço adicional significativo)




Vantagens e Desvantagens

Vantagens:

-  Eficiente: $O(n \log n)$ em todos os casos
-  Ordenação in-place

-  Estável na maioria das implementações
-  Não requer espaço adicional significativo

Desvantagens:

-  Não é adaptativo (não aproveita arrays parcialmente ordenados)
-  Mais complexo de implementar que algoritmos simples
-  Pior desempenho de cache comparado ao QuickSort

Comparação com Outros Algoritmos

python

```
# Comparação prática
import time
import random

def comparar_desempenho():
    tamanhos = [100, 1000, 5000]

    for tamanho in tamanhos:
        arr = [random.randint(1, 1000) for _ in range(tamanho)]

        # HeapSort
        copia = arr.copy()
        inicio = time.time()
        heapsort_simples(copia)
        tempo_heap = time.time() - inicio

        # QuickSort (built-in do Python)
        copia2 = arr.copy()
        inicio = time.time()
        copia2.sort()
        tempo_quick = time.time() - inicio

        print(f"Tamanho {tamanho}:")
        print(f"  HeapSort: {tempo_heap:.6f} segundos")
        print(f"  QuickSort: {tempo_quick:.6f} segundos")
        print()

# comparar_desempenho() # Descomente para testar
```

Casos de Uso Práticos

O HeapSort é ideal para:

- Sistemas embarcados com memória limitada
- Aplicações que requerem garantia de desempenho $O(n \log n)$
- Situações onde a estabilidade não é crítica
- Implementação de filas de prioridade

Este algoritmo combina a simplicidade conceitual da ordenação por seleção com a eficiência de estruturas de dados avançadas, sendo uma excelente escolha para muitas aplicações práticas.