🟦 Heapsort

1️⃣ Grundidee

Heapsort nutzt eine **Heap-Datenstruktur** (meist Max-Heap), um das größte Element effizient zu bestimmen. Der Algorithmus besteht aus zwei Phasen:

1. Aufbau eines Heaps aus der Liste
2. Wiederholtes Entfernen des Maximums und Einfügen ans Listenende

---

2️⃣ Voraussetzungen

- ❌ keine
- funktioniert auf **beliebigen vergleichbaren Elementen**

3️⃣ Laufzeiten & Eigenschaften

Eigenschaft	Wert
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)
Speicherbedarf	O(1)
In-place	ja
Stabil	nein

Hinweis: Die Laufzeit ist unabhängig von der Eingabereihenfolge.

4️⃣ Schritt-für-Schritt-Beispiel

Ausgangsliste:

```
[5, 3, 4, 1]
```

Heap-Aufbau (Max-Heap: Jeder Knoten ist grösser gleich seinen Kindern)

```
        5
       / \
      3   4
     /
    1
```

Sortieren

- Tausche 5 mit letztem Element → [1, 3, 4, 5]
- Heapify Rest → [4, 3, 1, 5]
- Tausche 4 → [1, 3, 4, 5]
- Heapify → [3, 1, 4, 5]

Ergebnis:

```
[1, 3, 4, 5]
```

Heap Sort – Algorithmus in Worten

Der Algorithmus beginnt damit, die gesamte Liste in einen Max-Heap umzuwandeln.
Dabei steht das größte Element immer an der Wurzel des Heaps.
Anschließend wird das größte Element mit dem letzten Element der Liste vertauscht.
Dadurch befindet sich das größte Element an seiner endgültigen Position.
Der betrachtete Bereich der Liste wird um dieses letzte Element verkleinert.
Der Heap wird anschließend wiederhergestellt, sodass erneut das größte Element an der Wurzel steht.
Dieser Vorgang wird so lange wiederholt, bis nur noch ein Element übrig ist.
Am Ende ist die gesamte Liste sortiert.\

5️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- Garantierte Laufzeit O(n log n)
- Kein zusätzlicher Speicher
- Schlechtere Cache-Lokalität als Quicksort

6️⃣ Vor- und Nachteile

Vorteile

- in-place
- garantierte Laufzeit
- kein zusätzlicher Speicher

Nachteile

- nicht stabil
- langsamer als Quicksort in der Praxis

🧠 Merksatz für die Prüfung

Heapsort sortiert mithilfe eines Heaps in O(n log n), ist in-place, aber nicht stabil.

7️⃣ Python-Implementierung

In [1]:
```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

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

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

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

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

    return arr
```

Beispielaufgaben

Implementieren einer Sink funktion

Element an Index i wird in einem Max-Heap "gesunken", bis die Heap-Eigenschaft wiederhergestellt ist.

Heap-Index-Regeln:\

- Linkes Kind: 2*i + 1\

- Rechtes Kind: 2*i + 2\
- Elternteil: (i - 1) // 2

Algorithmus für sink(k):

1. Solange k mindestens 1 Kind hat
2. Bestimme das grössere Kind
3. Wenn heap[k] >= heap[grösseres Kind] => fertig
4. Sonst tausche heap[k] mit heap[grösseres Kind] und setze k auf Index des grösseren Kindes
5. Wiederhole

Hier der Code dazu:

In [1]:
```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def sink(self, k):
        n = len(self.heap)

        while True:
            left = 2 * k + 1
            right = 2 * k + 2
            largest = k

            if left < n and self.heap[left] > self.heap[largest]:
                largest = left

            if right < n and self.heap[right] > self.heap[largest]:
                largest = right

            if largest == k:
                break

            self.swap(k, largest)
            k = largest
```

Implementieren einer swim Funktion

Wenn ein Element zu Gross ist, muss es "aufsteigen", um die Heap-Eigenschaft wiederherzustellen.

Idee: Vergleiche mit Parent und tausche, solange heap[k] > heap[parent].

Laufzeit: O(log n)

In [ ]:
```python
class MaxHeap:
    def __init__(self):
        self.heap = []

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def swim(self, k):
        while k > 0:
            parent = (k - 1) // 2
            if self.heap[k] <= self.heap[parent]:
                break
            self.swap(k, parent)
            k = parent
```

Implementieren der Einfügeoperation

Fügt ein neues Element in den Max-Heap ein.

Idee: Neues Element ans Ende hängen und dann swim.

Laufzeit: O(log n)

In [ ]:
```python
class MaxHeap:
    def __init__(self):
        self.heap = []

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def swim(self, k):
        while k > 0:
            parent = (k - 1) // 2
            if self.heap[k] <= self.heap[parent]:
                break
            self.swap(k, parent)
            k = parent

    def insert(self, x):
        self.heap.append(x)
        self.swim(len(self.heap) - 1)
```

Implementieren der Löschoperation

Entfernt das Maximum (Wurzel) aus dem Max-Heap.

Vorgehen:

1. Tausche Wurzel mit letztem Element
2. Entferne letztes Element (altes Maximum)
3. Sink die neue Wurzel, um Heap-Eigenschaft wiederherzustellen

Laufzeit: O(log n)

```
In [2]: class MaxHeap:
    def __init__(self):
        self.heap = []

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def sink(self, k):
        n = len(self.heap)
        while True:
            left = 2 * k + 1
            right = 2 * k + 2
            largest = k

            if left < n and self.heap[left] > self.heap[largest]:
                largest = left
            if right < n and self.heap[right] > self.heap[largest]:
                largest = right

            if largest == k:
                break

            self.swap(k, largest)
            k = largest

    def extract_max(self):
        if not self.heap:
            raise IndexError("extract_max from empty heap")

        max_val = self.heap[0]
        last = self.heap.pop()  # entfernt letztes Element

        if self.heap:
            self.heap[0] = last
            self.sink(0)

        return max_val
```

Implementieren der Heapify-Funktion

Wandelt eine unsortierte Liste in einen Max-Heap um.

Idee: Von unten nach oben sink auf alle Nicht-Blatt-Knoten.

Laufzeit: O(n)

```
In [3]: class MaxHeap:
    def __init__(self):
        self.heap = []

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

```python
    def sink(self, k):
        n = len(self.heap)
        while True:
            left = 2 * k + 1
            right = 2 * k + 2
            largest = k

            if left < n and self.heap[left] > self.heap[largest]:
                largest = left
            if right < n and self.heap[right] > self.heap[largest]:
                largest = right

            if largest == k:
                break

            self.swap(k, largest)
            k = largest

    def heapify(self, arr):
        self.heap = list(arr)
        n = len(self.heap)
        # Letzter innerer Knoten:
        for k in range((n // 2) - 1, -1, -1):
            self.sink(k)
```

Implementieren des Heapsort-Algorithmus

Sortiert eine Liste mithilfe von Heapsort.

Idee: Heap bauen, dann wiederholt Maximum extrahieren und ans Ende setzen.

Wenn aufsteigend, dann einfach am Ende out[::-1] zurückgeben.

In [ ]:

```python
def heapsort(arr):
    h = MaxHeap()
    h.heapify(arr)
    out = []
    while h.heap:
        out.append(h.extract_max())
    return out  # absteigend sortiert
```