TUGAS EKSPLORASI MANDIRI
PERANCANGAN DAN ANALISIS ALGORITMA
HEAPSORT

DOSEN PENGAMPU:

Randi Proska Sandra, S.Pd, M.Sc

OLEH:

Brian Makmur

23343029

Informatika (NK)

PROGRAM STUDI INFORMATIKA

DEPARTEMEN ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2025

A. RANGKUMAN MATERI

Heapsort adalah algoritma pengurutan yang menggunakan struktur data heap untuk menyusun elemen dalam urutan yang diinginkan. Algoritma ini ditemukan oleh J.W.J. Williams pada tahun 1964 dan terdiri dari dua tahap utama: pembangunan heap (heap construction) dan penghapusan maksimum secara berturut-turut (maximum deletions).

1. **Pembangunan Heap**
   Tahap pertama dalam heapsort adalah membangun heap dari array yang diberikan. Heap adalah struktur data berbentuk pohon biner yang memenuhi sifat **heap property**, yaitu setiap elemen induk memiliki nilai lebih besar atau sama dengan anak-anaknya (untuk max-heap). Heap dapat dibangun dengan dua pendekatan utama:
   - **Metode Bottom-Up**: Semua elemen ditempatkan dalam pohon biner lengkap, kemudian dilakukan penyusunan ulang mulai dari simpul internal terakhir hingga ke akar. Metode ini lebih efisien dibandingkan dengan metode top-down.
   - **Metode Top-Down**: Setiap elemen baru ditambahkan satu per satu ke dalam heap dan diatur posisinya dengan cara sift-up (menggeser ke atas sesuai dengan aturan heap).
2. **Penghapusan Maksimum dan Pengurutan**
   Setelah heap terbentuk, elemen terbesar (yang berada di akar) dipindahkan ke akhir array, kemudian ukuran heap dikurangi dan proses **heapify** dilakukan kembali untuk mempertahankan sifat heap. Proses ini diulangi hingga semua elemen terurut dengan benar dalam array. Teknik ini memastikan bahwa elemen terbesar selalu berada di posisi akhir setelah setiap iterasi, sehingga menghasilkan urutan yang benar.
3. **Efisiensi dan Kompleksitas Waktu**
   Heapsort memiliki kompleksitas waktu **O(n log n)** baik dalam kasus terbaik, rata-rata, maupun terburuk, menjadikannya lebih konsisten dibandingkan quicksort yang bisa mencapai **$O(n^2)$** dalam skenario terburuk. Namun, dalam implementasi praktis, heapsort biasanya lebih lambat dibandingkan quicksort karena overhead dalam pemeliharaan heap. Selain itu, heapify yang sering digunakan dalam proses ini memiliki kompleksitas **O(log n)** untuk setiap elemen, sehingga secara keseluruhan tetap efisien.
4. **Keunggulan dan Kelemahan**
   - **Keunggulan**:
     - **In-Place Sorting**: Tidak membutuhkan memori tambahan seperti mergesort.
     - **Stabilitas Waktu Eksekusi**: Kompleksitas waktu yang tetap di **O(n log n)**.
     - **Cocok untuk Struktur Data Prioritas**: Banyak digunakan dalam implementasi antrian prioritas karena heap dapat digunakan untuk dengan mudah mengekstrak elemen terbesar atau terkecil.
   - **Kelemahan**:
     - **Tidak Stabil**: Posisi elemen yang memiliki nilai sama dapat berubah.
     - **Overhead Perhitungan**: Proses heapify memerlukan lebih banyak pertukaran elemen dibandingkan quicksort.

- **Kurang Efisien untuk Data Kecil**: Untuk dataset yang kecil, insertion sort atau quicksort lebih efisien dibandingkan heapsort karena overhead heapify lebih besar dibandingkan keuntungan sorting dalam **O(n log n)**.

B. PSEUDOCODE

```
Fungsi SUSUN_HEAP(A, n, i):
    terbesar ← i
    kiri ← 2 * i + 1
    kanan ← 2 * i + 2
    Jika kiri < n dan A[kiri] > A[terbesar]:
        terbesar ← kiri
    Jika kanan < n dan A[kanan] > A[terbesar]:
        terbesar ← kanan
    Jika terbesar ≠ i:
        Tukar A[i] dengan A[terbesar]
        Panggil SUSUN_HEAP(A, n, terbesar)


Fungsi PENGURUTAN_HEAP(A):
    n ← panjang(A)
    // Tahap 1: Bangun Max Heap
    Untuk i dari (n // 2 - 1) hingga 0 dengan langkah -1:
        Panggil SUSUN_HEAP(A, n, i)
    // Tahap 2: Ekstraksi Elemen Maksimum Berulang Kali
    Untuk i dari (n - 1) hingga 1 dengan langkah -1:
        Tukar A[i] dengan A[0]
```

Panggil SUSUN_HEAP(A, i, 0)

// Program utama

A ← [12, 11, 13, 5, 6, 7]

PENGURUTAN_HEAP(A)

Cetak A

C. SOURCE CODE

```
def susun_heap(A, n, i):
    terbesar = i
    kiri = 2 * i + 1
    kanan = 2 * i + 2

    if kiri < n and A[kiri] > A[terbesar]:
        terbesar = kiri
    if kanan < n and A[kanan] > A[terbesar]:
        terbesar = kanan
    if terbesar != i:
        A[i], A[terbesar] = A[terbesar], A[i]
        susun_heap(A, n, terbesar)

def pengurutan_heap(A):
    n = len(A)
    # Tahap 1: Konstruksi Heap
    for i in range(n // 2 - 1, -1, -1):
```

```
            susun_heap(A, n, i)

        # Tahap 2: Penghapusan Maksimum Berulang
        for i in range(n - 1, 0, -1):
            A[i], A[0] = A[0], A[i]
            susun_heap(A, i, 0)


A = [12, 11, 13, 5, 6, 7]
pengurutan_heap(A)
print(A)
```

D. ANALSIS KEBUTUHAN WAKTU

1) Analisis menyeluruh dengan memperhatikan operasi/instruksi yang di eksekusi berdasarkan operator penugasan atau assignment (=, +=, *=, dan lainnya) dan operator aritmatika (+, %, * dan lainnya)

   Heap Sort terdiri dari dua tahap utama:

   1. **Konstruksi Heap** (Membangun Max Heap)
   2. **Penghapusan Maksimum Berulang** (Heap Sort)

   Tahap 1: Konstruksi Heap (Build Max Heap)

   Dilakukan dengan **heapify** pada setiap elemen dari tengah ke akar (dimulai dari index *n/2 - 1* hingga 0).

   - Setiap pemanggilan `susun_heap()` bisa melakukan pertukaran elemen dan rekursi hingga tinggi heap ($O(\log n)$).
   - Jumlah total pemanggilan `susun_heap()` adalah sekitar $O(n)$, sehingga tahap ini berjalan dalam $O(n)$.

   Tahap 2: Heap Sort (Penghapusan Maksimum Berulang)

   - Mengambil elemen maksimum (root) dan menukarnya dengan elemen terakhir.
   - Kemudian memanggil `susun_heap()` untuk menyesuaikan heap dengan ukuran yang berkurang.
   - Pemanggilan `susun_heap()` dilakukan kali (n -1), dengan masing-masing membutuhkan waktu $O(\log n)$.

- Maka kompleksitasnya adalah *O*(n log n).

Total Kompleksitas

- Tahap 1: *O*(n)
- Tahap 2: *O*(n log n)
- Sehingga totalnya adalah *O*(n log n).

2) Analisis berdasarkan jumlah operasi abstrak atau operasi khas

Mari kita hitung jumlah operasi utama berdasarkan karakteristik Heap Sort:

Konstruksi Heap

- Untuk setiap elemen dari index *n/2* hingga 0, kita melakukan heapify.
- Heapify memerlukan paling banyak *O*(log n) per elemen.
- Karena ada sekitar *n/2* elemen yang perlu diperiksa, jumlah operasi adalah sekitar: *O*(n)

Penghapusan Maksimum Berulang

- Proses penghapusan membutuhkan:
  - Swap antara root dan elemen terakhir: *O*(1) .
  - Heapify ulang setelah setiap penghapusan: *O*(log n) .
  - Ini dilakukan untuk setiap elemen dalam array (n - 1 kali).

Sehingga total jumlah operasi tetap *O*(n log n).

3) Analisis menggunakan pendekatan best-case (kasus terbaik), worst-case (kasus terburuk), dan average-case (kasus rata-rata)

Kasus	Analisis	Kompleksitas
Best-Case	Array sudah terurut, sehingga hanya sedikit operasi dilakukan	O(n log n)
Worst-Case	Array dalam urutan terbalik, sehingga setiap elemen harus diubah dan diproses sepenuhnya	*O*(n log n)
Average-Case	Secara rata-rata setiap elemen akan membutuhkan per operasi heapify	*O*(n log n)

Heap Sort memiliki kompleksitas waktu tetap *O*(n log n) pada semua kasus, karena sifat dari binary heap.

E. REFERENSI

Levitin, A. (2012). Introduction to the design & analysis of algorithms (3rd ed.). Pearson Education, Inc

F. LAMPIRAN GITHUB

https://github.com/TuanMudaBrian/Brian-Makmur-23343029-Heapsort.git