# PERANCANGAN DAN ANALISIS ALGORITMA

*HEAPSORT*


**DOSEN PENGAMPU:**

Randi Proska Sandra, S.Pd, M.Sc

**OLEH:**

Falisthatiunus Rangkuti

**22343006**

**PROGRAM STUDI INFORMATIKA**

**DEPARTEMEN TEKNIK ELEKTRONIKA**

**FAKULTAS TEKNIK**

**UNIVERSITAS NEGERI PADANG**

**2024**

# HEAPSORT

## A. Penjelasan

Heapsort adalah algoritma pengurutan yang efisien dan berbasis pada struktur data heap. Tujuannya adalah untuk mengurutkan sebuah array dengan cara membangun struktur heap dan mengurutkan elemen-elemennya. Algoritma ini pertama-tama mengubah array menjadi heap dengan memastikan setiap elemen memenuhi properti heap, yang pada kasus heapsort adalah properti maks atau min. Setelah heap terbentuk, elemen teratas (root) yang merupakan elemen terbesar atau terkecil dipindahkan ke posisi terakhir array. Langkah ini disebut ekstraksi. Setelah itu, heap direstrukturisasi untuk memastikan bahwa elemen yang tersisa masih memenuhi properti heap. Proses ekstraksi dan restrukturisasi ini diulang sampai semua elemen dipindahkan dari heap ke array, dan array diurutkan dengan benar.

Algoritma heapsort memiliki kompleksitas waktu O(n log n), di mana n adalah jumlah elemen dalam array. Meskipun kompleksitas ini sama dengan algoritma pengurutan lainnya seperti mergesort dan quicksort, heapsort memiliki keuntungan dalam hal penggunaan memori karena tidak memerlukan penyimpanan tambahan seperti mergesort. Selain itu, heapsort memiliki keunggulan dalam hal kinerja dalam beberapa kasus karena struktur heap memungkinkan akses ke elemen terbesar atau terkecil dalam waktu konstan.

Proses pembangunan heap dalam heapsort membutuhkan waktu O(n), sementara setiap operasi ekstraksi dan restrukturisasi membutuhkan waktu O(log n). Sehingga total waktu yang diperlukan untuk mengeksekusi algoritma ini adalah O(n log n). Meskipun heapsort dapat digunakan untuk pengurutan dalam memori dan luar memori, implementasinya yang sederhana dan kompleksitas yang konsisten membuatnya menjadi pilihan yang populer untuk aplikasi yang memerlukan pengurutan cepat dan efisien. Namun, perlu dicatat bahwa di beberapa kasus, seperti ketika array memiliki banyak elemen yang sama, heapsort mungkin tidak optimal dan algoritma lain seperti quicksort dapat lebih efisien.

## B. Penjelasan Program

- Pseudocode

```
Procedure Tumpuk(arr: array of Comparable, n: integer,
i: integer):
    Terbesar := i
    Kiri := 2 * i + 1
    Kanan := 2 * i + 2

    // Periksa apakah anak kiri lebih besar daripada
root
    If Kiri < n and arr[Kiri] > arr[Terbesar]:
        Terbesar := Kiri

    // Periksa apakah anak kanan lebih besar daripada
root atau anak kiri
    If Kanan < n and arr[Kanan] > arr[Terbesar]:
        Terbesar := Kanan

    // Jika terbesar bukan root
    If Terbesar != i:
        Tukar(arr[i], arr[Terbesar])
```

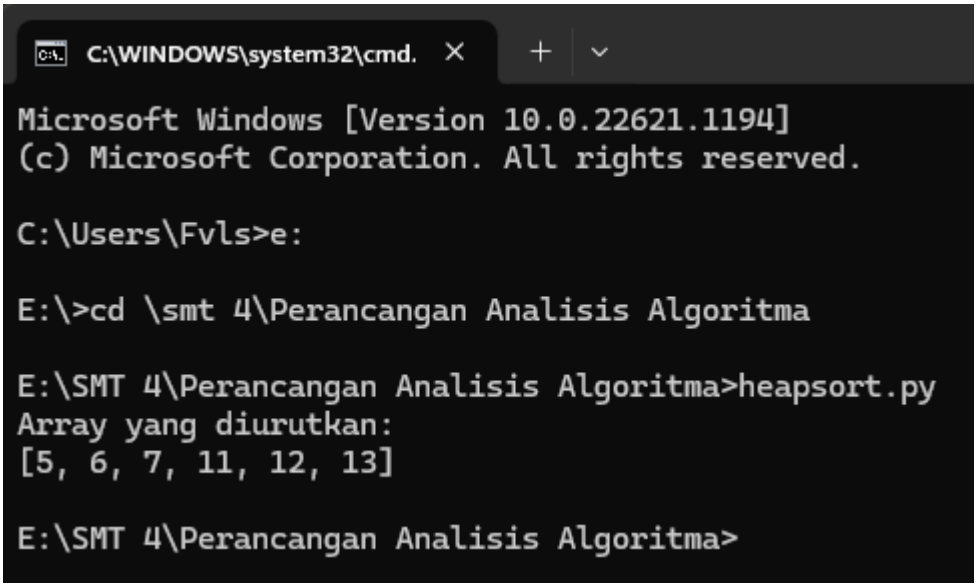
```
        // Rekursif tumpuk pada sub-pohon yang
terpengaruh
        Tumpuk(arr, n, Terbesar)

Procedure HeapSort(arr: array of Comparable):
    N := Panjang(arr)

    // Bangun heap awal
    For i from N/2 - 1 down to 0:
        Tumpuk(arr, N, i)

    // Ekstraksi elemen satu per satu dari heap
    For i from N - 1 down to 0:
        // Tukar elemen teratas dengan elemen terakhir
        Tukar(arr[0], arr[i])

        // Panggil tumpuk untuk membangun heap pada
bagian yang tersisa
        Tumpuk(arr, i, 0)
```

- Program

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

    # Periksa apakah anak kiri lebih besar
daripada root
    if kiri < n and arr[kiri] > arr[terbesar]:
        terbesar = kiri

    # Periksa apakah anak kanan lebih besar
daripada root atau anak kiri
    if kanan < n and arr[kanan] > arr[terbesar]:
        terbesar = kanan

    # Jika terbesar bukan root
    if terbesar != i:
        arr[i], arr[terbesar] = arr[terbesar],
arr[i]

        # Rekursif tumpuk pada sub-pohon yang
terpengaruh
        tumpuk(arr, n, terbesar)

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

    # Bangun heap awal
    for i in range(n//2 - 1, -1, -1):
        tumpuk(arr, n, i)

    # Ekstraksi elemen satu per satu dari heap
```

```
    for i in range(n-1, 0, -1):
        # Tukar elemen teratas dengan elemen terakhir
        arr[0], arr[i] = arr[i], arr[0]

        # Panggil tumpuk untuk membangun heap pada bagian yang tersisa
        tumpuk(arr, i, 0)

# Contoh penggunaan
arr = [12, 11, 13, 5, 6, 7]
heapsort(arr)
print("Array yang diurutkan:")
print(arr)
```

- Screenshot




- Penjelasan
  1. **Fungsi tumpuk(arr, n, i):**
     - Fungsi ini menerima tiga parameter:
       - **arr**: Array yang akan diurutkan.
       - **n**: Ukuran heap saat ini.
       - **i**: Indeks root dari sub-heap yang akan di-tumpuk.
     - Fungsi ini bertanggung jawab untuk memperbaiki sifat heap maksimum dari sub-heap dengan root di indeks **i**.
     - Pertama, fungsi ini menentukan indeks dari anak kiri (**kiri**) dan anak kanan (**kanan**) dari node **i**.
     - Kemudian, fungsi membandingkan nilai node **kiri** dan **kanan** dengan nilai node **terbesar** (yang awalnya diatur sebagai **i**). Jika nilai node anak lebih besar dari nilai node terbesar saat ini, **terbesar** diubah menjadi indeks anak yang memiliki nilai lebih besar.
     - Jika nilai **terbesar** tidak sama dengan **i**, maka pertukaran dilakukan antara elemen di posisi **i** dengan elemen di posisi **terbesar**. Setelah itu, fungsi rekursif dipanggil untuk menumpuk sub-heap pada indeks **terbesar**.

- Proses ini dilakukan secara rekursif hingga semua sub-heap memenuhi sifat heap maksimum.

2. **Fungsi heapsort(arr):**
   - Fungsi ini menerima satu parameter, yaitu array yang akan diurutkan.
   - Pertama, ukuran array (**n**) dihitung.
   - Selanjutnya, dilakukan pembangunan heap awal dengan memanggil fungsi **tumpuk** untuk setiap sub-heap, dimulai dari sub-heap terakhir hingga root heap.
   - Setelah heap awal dibangun, dilakukan ekstraksi elemen satu per satu dari heap. Elemen teratas (elemen maksimum) ditukar dengan elemen terakhir heap yang belum diurutkan. Setelah pertukaran, heap dikurangi satu elemen dan dilakukan operasi **tumpuk** untuk memperbaiki sifat heap maksimum.
   - Proses ini diulang sampai seluruh array terurut.
3. **Contoh Penggunaan:**
   - Sebuah array **arr** dibuat dengan elemen **[12, 11, 13, 5, 6, 7]**.
   - Fungsi **heapsort** dipanggil dengan array ini sebagai argumen.
   - Setelah pengurutan selesai, array yang diurutkan dicetak.

## C. Analisis

1. **Analisis dengan Memperhatikan Operasi/Instruksi:**
   - Dalam algoritma Heapsort, terdapat dua tahap utama: pembangunan heap awal dan ekstraksi elemen dari heap.
   - **Pembangunan Heap Awal:** Tahap ini melibatkan iterasi melalui setengah dari elemen array (dari n/2 hingga 0) dan melakukan operasi tumpukan pada setiap elemen. Dalam operasi tumpukan, kita membandingkan elemen dengan kedua anaknya dan menukar jika perlu, yang membutuhkan beberapa operasi penugasan dan perbandingan. Oleh karena itu, kompleksitas waktu untuk tahap ini adalah O(n).
   - **Ekstraksi Elemen dari Heap:** Tahap ini melibatkan iterasi dari n-1 hingga 1, di mana kita pertukarkan elemen teratas heap dengan elemen terakhir yang belum diurutkan dan kemudian memanggil operasi tumpukan untuk memastikan sifat heap dipertahankan. Ini juga membutuhkan beberapa operasi penugasan dan perbandingan, yang dapat dihitung sebagai O(log n). Karena tahap ini dieksekusi n-1 kali, maka kompleksitasnya adalah O(n log n).
   - Secara total, kompleksitas waktu algoritma ini adalah O(n + n log n), yang biasanya disederhanakan menjadi O(n log n), karena tahap ekstraksi dominan.
2. **Analisis Berdasarkan Jumlah Operasi Abstrak:**
   - Jumlah operasi abstrak dalam Heapsort adalah jumlah pertukaran elemen dan panggilan rekursif ke operasi tumpukan.
   - Pada setiap iterasi pembangunan heap awal, kita melakukan beberapa pertukaran elemen (maksimal log n pada setiap iterasi), dan ini dieksekusi sebanyak n/2 kali. Oleh karena itu, jumlah operasi pertukaran dalam tahap ini adalah O(n log n).
   - Pada tahap ekstraksi, kita melakukan pertukaran elemen pada setiap iterasi (maksimal log n pada setiap iterasi), yang dieksekusi sebanyak n-1 kali. Jumlah operasi dalam tahap ini juga O(n log n).

- Total jumlah operasi dalam algoritma Heapsort adalah O(n log n).

3. **Analisis Menggunakan Pendekatan Best-case, Worst-case, dan Average-case:**
   - **Best-case:** Dalam Heapsort, baik kasus terbaik maupun kasus terburuk adalah sama, yaitu O(n log n). Ini karena algoritma tidak bergantung pada keadaan awal array dan akan selalu menghasilkan urutan yang sama.
   - **Worst-case:** Kebutuhan waktu terburuk terjadi ketika setiap iterasi heapify pada tahap pembangunan heap membutuhkan waktu O(log n). Dalam hal ini, kompleksitasnya adalah O(n log n).
   - **Average-case:** Secara rata-rata, Heapsort memiliki kompleksitas O(n log n) karena pembangunan heap awal dan ekstraksi elemen dilakukan dalam O(n log n).

## D. Referensi


Schaffer, R., & Sedgewick, R. (1993). The analysis of heapsort. *Journal of Algorithms, 15*(1), 76-100.

Heap Sort Algorithm


## E. Lampirkan Link Github

Heapsort_Algorithm_GitHub_Falis