Tugas Mandiri 1
Perancangan dan Analisis Algoritma

“Transform-and Conquer: Heapsort”

Dosen Pengampu :

Randi Proska Sandra, M.Sc.

Disusun oleh:

Carel Habsian Osagi (23343061)

PROGRAM STUDI INFORMATIKA(NK)

DEPARTEMEN TEKNIK ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2024

A. PENJELASAN PROGRAM/ALGORITMA

Heapsort adalah salah satu algoritma pengurutan yang menggunakan struktur heap, yaitu representasi binary tree dalam bentuk array yang memenuhi sifat heap. Algoritma ini termasuk dalam kategori Transform-and-Conquer, di mana data diubah menjadi bentuk heap terlebih dahulu sebelum dilakukan proses pengurutan. Heapsort sering digunakan karena memiliki kompleksitas waktu O(n log n) yang konsisten dalam kasus terbaik, rata-rata, dan terburuk, menjadikannya salah satu algoritma pengurutan yang lebih stabil dibandingkan algoritma seperti Quicksort, yang dalam kasus terburuk bisa mencapai $O(n^2)$.

Cara Kerja Heapsort

1. **Membangun Max-Heap (Heap Construction)**

   Tahap pertama adalah mengubah array input menjadi max-heap, yaitu sebuah binary tree di mana setiap induk lebih besar dari anak-anaknya. Proses ini dilakukan dengan heapify dari bawah ke atas (bottom-up approach) untuk memastikan setiap subtree dalam array memenuhi sifat heap. Heapify adalah operasi yang memastikan bahwa setiap node memiliki nilai yang lebih besar dari anak-anaknya, dilakukan dalam O(log n) waktu per elemen. Dengan memulai dari node non-daun terakhir hingga ke akar, seluruh array dapat diubah menjadi max-heap dalam O(n) waktu.

2. **Penghapusan Maksimum & Pengurutan (Heap Sort Process)**

   Setelah max-heap terbentuk, langkah selanjutnya adalah menukar elemen terbesar (akar heap) dengan elemen terakhir dalam array. Elemen terbesar yang telah ditukar dipindahkan keluar dari heap, sementara heap dikurangi ukurannya. Setelah itu, dilakukan heapify ulang untuk memastikan bahwa struktur heap tetap valid setelah perubahan elemen. Proses ini diulang hingga seluruh elemen dipindahkan ke posisi yang benar dalam array, menghasilkan array yang telah terurut. Karena setiap iterasi membutuhkan O(log n) untuk menyesuaikan heap dan dilakukan sebanyak n - 1 kali, maka total kompleksitas untuk tahap ini adalah O(n log n).

Keunggulan Heapsort

- Kompleksitas waktu stabil pada O(n log n) dalam semua kasus, baik terbaik, rata-rata, maupun terburuk. Hal ini membuatnya lebih stabil dibandingkan Quicksort, yang dalam beberapa kondisi bisa menjadi $O(n^2)$.
- Heapsort merupakan in-place sorting, sehingga tidak membutuhkan memori tambahan karena pengurutan dilakukan langsung dalam array. Berbeda dengan

Merge Sort yang memerlukan ruang tambahan sebesar O(n), Heapsort lebih efisien dalam penggunaan memori.

- Cocok untuk sistem dengan batasan memori karena tidak membutuhkan penyimpanan tambahan, sehingga sangat berguna dalam situasi di mana efisiensi ruang menjadi faktor utama.

Kelemahan Heapsort

- Kurang cache-friendly karena akses memori dalam Heapsort tidak berurutan dan bersifat acak, berbeda dengan Quicksort yang bekerja dengan cara membagi array menjadi bagian yang lebih kecil dan lebih sesuai dengan arsitektur cache modern. Oleh karena itu, dalam banyak kasus praktis, Quicksort lebih cepat daripada Heapsort meskipun kompleksitas teoritisnya sama.
- Tidak stabil secara default, artinya elemen dengan nilai yang sama bisa berubah urutannya setelah pengurutan. Namun, dapat dimodifikasi agar menjadi stabil dengan menggunakan teknik tambahan seperti menambahkan indeks sebagai pembanding sekunder.

B. PSEUDOCODE

```
FUNGSI SusunHeap(data, ukuran_heap, indeks):
    terbesar ← indeks   // Anggap node indeks sebagai yang terbesar
    kiri ← 2 * indeks + 1   // Indeks anak kiri
    kanan ← 2 * indeks + 2  // Indeks anak kanan

    JIKA kiri < ukuran_heap DAN data[kiri] > data[terbesar] MAKA
        terbesar ← kiri

    JIKA kanan < ukuran_heap DAN data[kanan] > data[terbesar] MAKA
        terbesar ← kanan

    JIKA terbesar ≠ indeks MAKA
        TUKAR data[indeks] dengan data[terbesar]
        PANGGIL SusunHeap(data, ukuran_heap, terbesar)

FUNGSI Heapsort(data):
    ukuran_data ← PANJANG(data)

    // Bangun Max-Heap dari bawah ke atas
    UNTUK i ← ukuran_data // 2 - 1 hingga 0 DENGAN LANGKAH -1 LAKUKAN
        PANGGIL SusunHeap(data, ukuran_data, i)

    // Ekstraksi elemen satu per satu dari heap
    UNTUK i ← ukuran_data - 1 hingga 1 DENGAN LANGKAH -1 LAKUKAN
        TUKAR data[i] dengan data[0]  // Pindahkan elemen terbesar ke
akhir
        PANGGIL SusunHeap(data, i, 0)  // Heapify ulang
```

```
// Contoh penggunaan
data ← [3, 8, 9, 5, 1, 2]
CETAK "Data sebelum diurutkan: ", data

PANGGIL Heapsort(data)

CETAK "Data setelah diurutkan: ", data
```

Penjelasan Pseudocode

1. Fungsi SusunHeap(data, ukuran_heap, indeks)

   - Mengubah data menjadi max heap dengan membandingkan anak kiri dan kanan.
   - Jika ada anak yang lebih besar dari indeks, maka dilakukan pertukaran dan proses diulang rekursif.

2. Fungsi Heapsort(data)

   - Membangun max heap dari array yang diberikan.
   - Menukar elemen terbesar (root heap) ke posisi akhir.
   - Heapify ulang untuk memastikan sisa elemen tetap dalam bentuk heap.

C. SOURCE CODE

heapsort.py

```
# Fungsi untuk membentuk Max-Heap
def susun_heap(data, ukuran_heap, indeks):
    terbesar = indeks
    kiri = 2 * indeks + 1
    kanan = 2 * indeks + 2

    if kiri < ukuran_heap and data[kiri] > data[terbesar]:
        terbesar = kiri
    if kanan < ukuran_heap and data[kanan] > data[terbesar]:
        terbesar = kanan
    if terbesar != indeks:
        data[indeks], data[terbesar] = data[terbesar], data[indeks]
        susun_heap(data, ukuran_heap, terbesar)

# Fungsi Heapsort
def heapsort(data):
    ukuran_data = len(data)

    # Bangun max heap
```

```
    for i in range(ukuran_data // 2 - 1, -1, -1):
        susun_heap(data, ukuran_data, i)

    # Pengurutan dengan menukar elemen terbesar ke akhir
    for i in range(ukuran_data - 1, 0, -1):
        data[i], data[0] = data[0], data[i]
        susun_heap(data, i, 0)

# Contoh penggunaan
data = [32, 54, 34, 76, 87, 45, 67, 57]
print("Data sebelum diurutkan:", data)
heapsort(data)
print("Data setelah diurutkan:", data)
```

D. ANALISIS KEBUTUHAN WAKTU

1) Tahap Kerja Algoritma Heapsort

Heapsort terdiri dari dua tahap utama: membangun max-heap dan melakukan pengurutan.

- **Membangun Max-Heap**: Array awal dikonversi menjadi max-heap menggunakan proses heapify dari bawah ke atas. Heapify dilakukan pada setiap node non-daun mulai dari indeks tengah hingga indeks nol. Ini memastikan bahwa setiap node memenuhi sifat heap.
- **Penghapusan Maksimum & Pengurutan**: Elemen terbesar (akar) ditukar dengan elemen terakhir dalam array. Setelah itu, heap dikurangi ukurannya, dan proses heapify diterapkan lagi untuk memastikan heap tetap valid. Langkah ini diulang sampai seluruh elemen terurut.

Setiap kali heapify diterapkan pada satu elemen, kompleksitasnya adalah O(log n). Karena ada sekitar nnn elemen yang perlu diproses, kompleksitas total membangun heap adalah O(n), dan proses pengurutan memiliki kompleksitas O(n log n), sehingga keseluruhan algoritma berjalan dalam O(n log n)

2) Analisis Berdasarkan Operator Penugasan dan Aritmatika

- **Operator Penugasan (=, +=, *=,** dll**.)**:
    - Setiap elemen dalam heap mengalami beberapa kali perpindahan saat heapify berjalan.
    - Saat heap dibangun, terjadi sekitar O(n) operasi penugasan.
    - Saat sorting, ada sekitar O(n log n) operasi penugasan karena tiap elemen ditukar dan diproses kembali dengan heapify.
- **Operator Aritmatika (+, %, *, dll.)**:

- Heapify bergantung pada perbandingan nilai anak dan induk yang dilakukan dengan operasi aritmatika.
- Setiap perbandingan dalam heapify membutuhkan operasi seperti pembagian untuk menentukan indeks anak kiri dan kanan.
- Karena setiap iterasi heapify berjalan dalam O(log n), total operasi aritmatika selama sorting adalah O(n log n).

3) Analisis Berdasarkan Jumlah Operasi Abstrak

- **Heapify**: Dilakukan sebanyak O(n) kali dengan setiap operasi berjalan dalam O(log n), sehingga totalnya O(n).
- **Penghapusan Maksimum & Pengurutan**: Dilakukan sebanyak O(n) kali, dan setiap kali membutuhkan O(log n) operasi heapify, sehingga totalnya O(n log n).
- **Keseluruhan Operasi**: Total operasi abstrak utama dalam Heapsort adalah O(n log n).

4) Analisis Best-Case, Worst-Case, dan Average-Case

- **Best-Case (Kasus Terbaik)**: Terjadi jika array sudah dalam bentuk max-heap sejak awal. Dalam kasus ini, membangun heap bisa berjalan dalam O(n), dan sorting tetap O(n log n). Total kompleksitas tetap O(n log n).
- **Worst-Case (Kasus Terburuk)**: Terjadi jika array diberikan dalam urutan yang benar-benar berlawanan dengan bentuk max-heap yang diinginkan. Heapify akan berjalan dalam kondisi paling tidak optimal, tetapi kompleksitas tetap O(n log n) karena heapify memiliki batasan waktu tersebut.
- **Average-Case (Kasus Rata-Rata)**: Secara umum, Heapsort selalu berjalan dalam O(n log n), karena membangun heap memakan waktu O(n) dan sorting berjalan dalam O(n log n). Perbedaan hanya ada dalam jumlah perbandingan dan penugasan yang dilakukan selama heapify.

E. REFERENSI

[1] Levitin, A. (2002). Introduction to the Design and Analysis of Algorithms (3rd ed.). Pearson

[2] Goodrich, M. T., & Tamassia, R. (2014). *Data Structures and Algorithms in Java* (6th ed.). Wiley.

[3] Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

[4] Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

[5] Heineman, G. T., Pollice, G., & Selkow, S. (2008). *Algorithms in a Nutshell*.

[6] Baase, S., & Van Gelder, A. (2000). *Computer Algorithms: Introduction to Design and Analysis* (3rd ed.). Pearson.

F. LAMPIRAN LINK GITHUB

https://github.com/CarelHabsi/Algoritma-Heapsort