Algoritma Heap Sort

Paul Gunawan Hariyanto[1], Dendy Duta Narendra[2], Ade Gunawan[3]

Sekolah Teknik Elektro & Informatika
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if14023@students.if.itb.ac.id[1], if14033@students.if.itb.ac.id[2], if14049@students.if.itb.ac.id[3]

Abstrak

Dalam dunia teknologi informasi, pasti akan terjadi suatu pengolahan data atau informasi, sebelum data maupun informasi tersebut disampaikan. Salah satu bentuk dari pengolahan data adalah pengurutan data (*data sorting*). Banyak cara yang bisa gunakan dalam pengurutan data. Salah satunya adalah strategi pemecahan masalah secara *divide and conquer.* Diantara metode pengurutan data yang menggunakan teknik *divide and conquer* adalah algoritma *heap sort.* Algoritma *heap sort* menggunakan sebuah pohon *heap* -yaitu pohon yang nilai setiap simpulnya lebih besar dari nilai simpul anak yang dimilikinya- dan sebuah tabel tempat hasil pengurutan dikeluarkan.

Kata kunci*: pengurutan data, data sorting, divide and conquer, heap sort, heap.*

1. Pendahuluan

Pengurutan data (*data sorting*) merupakan bagian dari pengolahan data informasi. Dari data-data yang telah didapat, ada kalanya data tersebut harus diurutkan terlebih dahulu berdasarkan aturan yang lebih dulu ditentukan. Berdasarkan nilai maupun alphabet misalnya.

Metode-metode pengurutan data pun ada berbagai jenis. Mulai dari *binary sort*, *insertion sort, merge sort, heap sort* dll. Penggunaan metode mana yang akan dipakai nantinya tergantung dari jenis maupun kuantitas data yang diolah.

Heap sort, algoritma pengurutan data yang menggunakan teknik *divide and conquer,* merupakan salah satu metode pengurutan yang sering digunakan. Melalui makalah ini akan dibahas teknik pencarian ini beserta kelebihan dan kekurangannya.

2. Landasan Teori

2.1 Definisi *Divide and Conquer*

Algoritma *Divide* and *Conquer* menyelesaikan suatu masalah dengan membagi masalah tersebut menjadi beberapa masalah yang lebih kecil, upa-masalah (*subproblem*), menyelesaikan masing-masing upa-masalah tersebut secara rekursif, kemudian menggabungkan solusi jawaban tadi hingga mendapatkan jawaban dari persoalan yang sebenarnya.

Setelah membagi masalah menjadi lebih kecil, dan ukurannya sudah cukup ‘kecil’ untuk diselesaikan, persoalan tersebut sudah dapat langsung dicari solusinya, tanpa tergantung dari persoalan lainnya, karena memang persoalan yang satu dengan yang lain tidak berhubungan tetapi memiliki karakteristik yang sama.

2.2 Algoritma *Divide and Conquer*

Proses menyelesaikan masalah dengan *Divide and Conquer* memiliki tiga tahap utama, yaitu :

Divide : membagi masalah menjadi beberapa masalah yang lebih kecil, sehingga bisa diselesaikan
Conquer : menyelesaikan masing-masing upa-masalah tersebut
Combine : menggabungkan solusi dari upa-masalah untuk mendapatkan solusi dari persoalan semula.

```
Procedure Divide_and_Conquer (input n : integer)
Deklarasi
  r, k : integer
Algoritma
  if (n ≤ n0) then
    Conquer subprogram
  else
    Divide menjadi r persoalan, ukuran @ n/k
    for masing-masing dari r persoalan do
       Divide_and_Conquer(n/k)
    endfor
    Combine solusi dari r persoalan
  endif
```

Keterangan :
- n : ukuran dari masukan
- r : jumlah persoalan yang dibagi
- k : faktor pembagi (rasio)
- n0 : batas ukuran dari *subproblem*

2.3 Kompleksitas

Pemecahan masalah dengan algoritma *Divide and Conquer* biasanya menghasilkan algoritma yang efisien, namun tentu saja waktu pencarian solusi sangat bergantung dari jumlah masukan.

Kompleksitas algoritma ini dapat dihitung dalam relasi berikut :

$$T(n) = \begin{cases} g(n) & , n \le n0 \\ 2T(n/2) + f(n) & , n > n0 \end{cases}$$

Keterangan :

T(n) = waktu komputasi Divide and Conquer dengan ukuran masukan sebanyak n

g(n) = waktu komputasi untuk penyelesaian *subproblem*
f(n) = waktu komputasi untuk menggabungkan solusi masing-masing *subproblem*

Untuk kasus rata-rata (*average case*), algoritma *Divide and Conquer* akan memiliki kompleksitas O(n log n)

2.4 Keuntungan

Algoritma Divide and Conquer memiliki keuntungan sebagai berikut :

a. Dapat menyelesaikan persoalan sulit

Kebanyakan persoalan yang susah dipecahkan dapat diselesaikan dengan algoritma ini, karena algoritma ini 'hanya' memerlukan pembagian masalah menjadi masalah yang lebih kecil, menyelesaikan masalah-masalah yang sudah dibagi tersebut, kemudian menggabungkan solusi-solusi tadi menjadi solusi dari persoalan awal.

Walaupun demikian, tetap saja harus dipikirkan cara yang tepat untuk membagi dan menggabungkan persoalan, supaya dapat diselesaikan. Dan kadang kala algoritma ini adalah satu-satunya cara untuk menyelesaikan persoalan tersebut.

b. Efisiensi algoritma

Divide and Conquer biasanya memberikan desain algoritma yang efisien. Contohnya, jika persoalan tersebut dibagi, diselesaikan dan kemudian digabungkan kembali, maka akan ada k sebagai faktor pembagi, dan ada persoalan yang berukuran n/k untuk tiap levelnya. Pencarian solusi untuk level basis memerlukan kompleksitas O(1), sehingga algoritma ini akan memiliki kompleksitas O(n log n). Hal ini diterapkan di beberapa persoalan, walaupun terkadang terdapat pendekatan-pendekatan lain dalam pengimplementasiannya.

c. Dapat berjalan secara paralel

Algoritma *Divide and Conquer* dapat dikerjakan secara paralel pada mesin multi-prosesor, karena persoalan yang telah dibagi dapat dikerjakan secara bebas dan sama sekali tidak bergantung pada persoalan lainnya. Sangat menguntungkan apabila persoalan yang dikerjakan berukuran besar, dapat dibagi untuk diselesaikan pada prosesor yang berbeda.

2.5 Kerugian

Kerugian utama pada Algoritma *Divide and Conquer* adalah sifat rekursifnya. Kebanyakan implementasi dari algoritma ini memakai rekursi, sehingga terkadang menjadikannya lambat. Pemanggilan prosedur yang sama berkali-kali, mengakibatkan pemakaian memori untuk menyimpan urutan pemanggilan prosedur. Semakin besar ukuran masukannya, akan dilakukan lebih banyak lagi pemanggilan prosedur untuk membagi masukan tersebut, sehingga memperbanyak pemakaian memori.

3. Heap Sort

3.1 Penjelasan Singkat

*Heap So*rt adalah sebuah algoritma pengurutan yang paling lambat dari algoritma yang memiliki kompleksitas O(n log n). Tetapi tidak seperti algoritma Merge Sort dan Quick Sort, algoritma Heap Sort tidak memerlukan rekursif yang besar atau menggunakan banyak tabel (array). Oleh karena itu, Heap Sort adalah pilihan yang baik untuk sebuah kumpulan data yang besar.

Algoritma ini dimulai dengan membangun sebuah *array heap* dengan membangun tumpukan dari kumpulan data, lalu memindahkan data terbesar ke bagian belakang dari sebuah tabel hasil. Setelah itu, *array heap* dibangun kembali, kemudian mengambil elemen terbesar untuk diletakkan di sebelah item yang telah dipindahkan tadi. Hal ini diulang sampai *array heap* habis.

Jadi secara umum, algoritma ini memerlukan dua buah tabel; satu tabel untuk menyimpan *heap*, dan satu tabel lainnya untuk menyimpan hasil. Walaupun lebih lambat dari Merge Sort atau Quick Sort, algoritma ini cocok untuk digunakan pada data yang berukuran besar.

3.2 Algoritma Heap Sort

3.2.1 Pseudo-code

```
function heapSort(a, count) {
  var int start := count ÷ 2 - 1,
          end := count - 1
  while start ≥ 0
    sift(a, start, count)
    start := start - 1
  while end > 0
    swap(a[end], a[0])
    sift(a, 0, end)
    end := end - 1
}
```

```
function sift(a, start, count) {
  var int root := start, child
  while root * 2 + 1 < count {
    child := root * 2 + 1
    if child < count - 1 and
    a[child] < a[child + 1]
      child := child + 1
    if a[root] < a[child]
      swap(a[root], a[child])
      root := child
    else
      return
  }
}
```

Keterangan :

swap : prosedur yang telah didefiniskan, untuk menukar isi dari argumen 1 dengan argumen 2.

3.2.2 Contoh implementasi

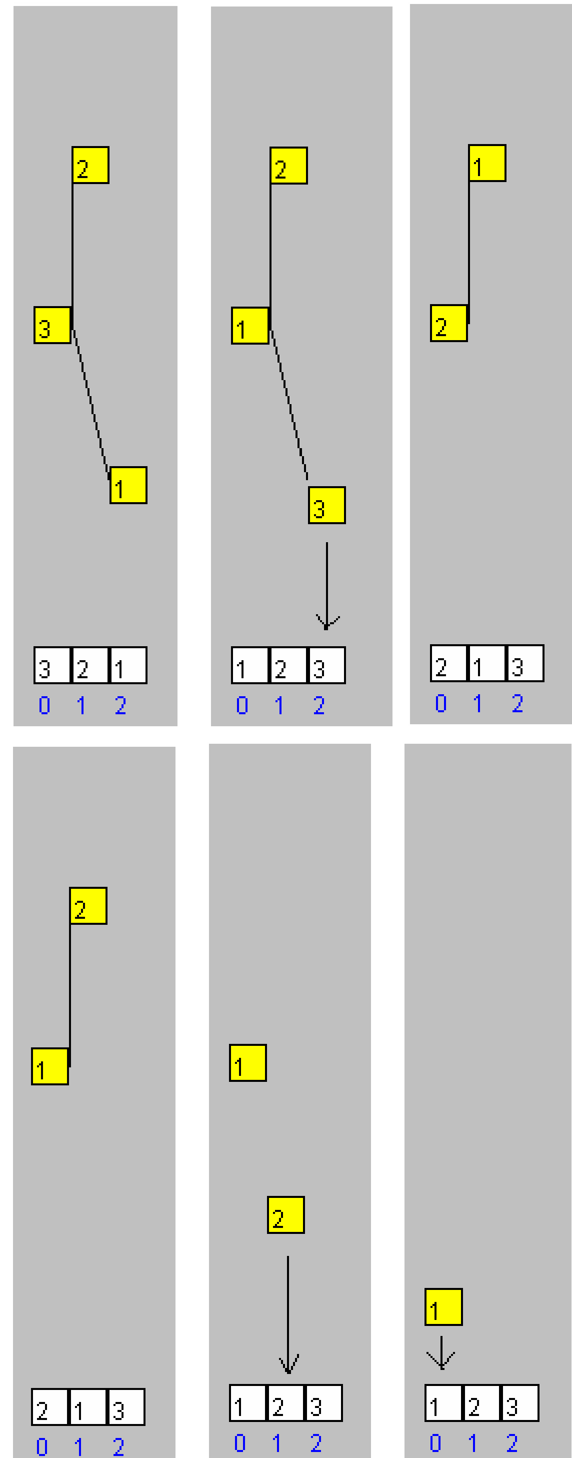
```
void heapSort(int numbers[], int array_size)
{
  int i, temp;

  for (i = (array_size / 2)-1; i >= 0; i--)
    siftDown(numbers, i, array_size);

  for (i = array_size-1; i >= 1; i--)
  {
    temp = numbers[0];
    numbers[0] = numbers[i];
    numbers[i] = temp;
    siftDown(numbers, 0, i-1);
  }
}


void siftDown(int numbers[], int root, int
bottom)
{
  int done, maxChild, temp;

  done = 0;
  while ((root*2 <= bottom) && (!done))
  {
    if (root*2 == bottom)
      maxChild = root * 2;
    else if (numbers[root * 2] >
numbers[root * 2 + 1])
      maxChild = root * 2;
    else
      maxChild = root * 2 + 1;

    if (numbers[root] < numbers[maxChild])
    {
      temp = numbers[root];
      numbers[root] = numbers[maxChild];
      numbers[maxChild] = temp;
      root = maxChild;
    }
    else
      done = 1;
  }
}
```

3.3 Contoh kasus

3.3.1 Kasus terbaik (*Best case*)

Jika masukan yang diberikan dalam Heap Sort adalah berurutan mengecil, maka akan diperoleh kasus terbaik untuk algoritma ini. Karena dalam pembuatan pohon *heap*-nya hanya memerlukan sekali *pass* saja, dan tidak dilakukan pertukaran elemen. Pertukaran hanya dilakukan pada saat sudah tercapai pohon *heap* yang akarnya berelemen terbesar, untuk menaruhnya pada tabel hasil.

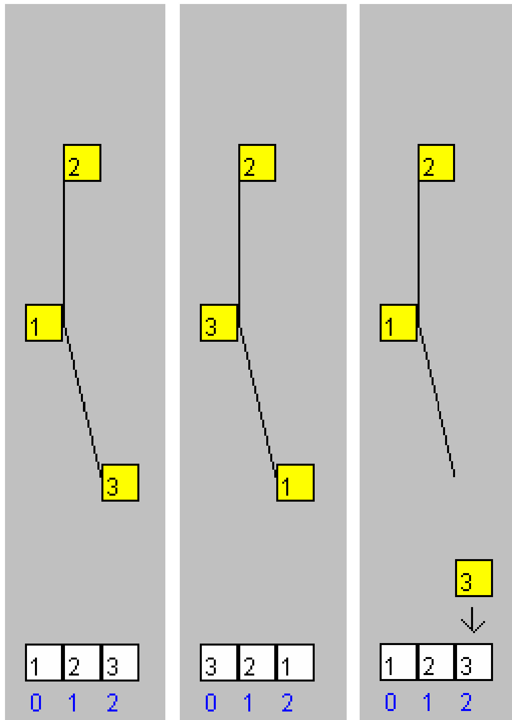
Berikut adalah contoh gambar algoritma Heap Sort dengan masukan : 1 2 3

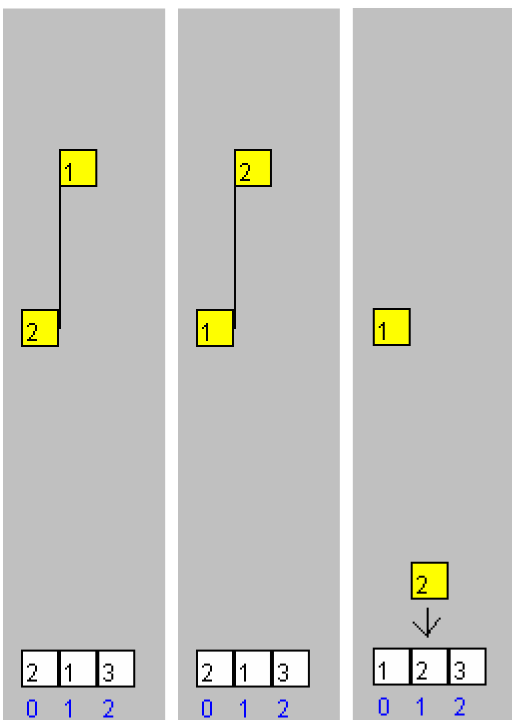


3.3.2 Kasus terburuk (*Worst case*)

Sedangkan kasus terburuk terdapat pada masukan yang telah berurut membesar. Hal ini diakibatkan karena pertukaran elemen yang terjadi merupakan yang terbanyak dari kasus lainnya, dimana elemen terbesar yang seharusnya berada di akar, terdapat di anak pohon yang paling dalam.

Berikut adalah contoh gambar dari algoritma Heap Sort dengan masukan : 1 2 3

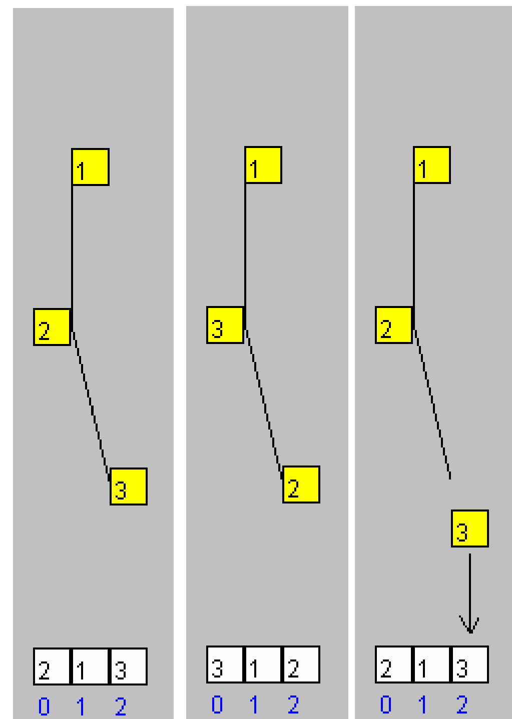


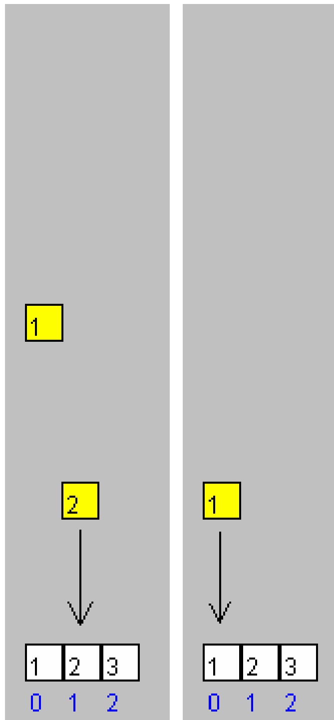


3.3.3 Kasus acak (*Random case*)

Kasus acak mempunyai urutan masukan yang tidak terdapat pada dua kasus di atas, yaitu tidak diketahui di mana letak elemen terbesar yang akan menjadi elemen akar.

Berikut adalah contoh algoritma Heap Sort dengan masukan : 2 1 3





4. Kesimpulan

Meskipun lebih lambat dari algoritma pengurutan data yang lain, algoritma *heap sort* memiliki kelebihan ketika menangani data dalam skala yang besar/*massive*. Karena algoritma ini memiliki kelebihan tidak menggunakan banyak tabel, tetapi hanya satu tabel yang dipakai untuk menyimpan hasil dari pengurutan tersebut.

5. Daftar Pustaka

[1] Heap sort visualization Java applet. ***http://www2.hawaii.edu/~copley/665/HSApplet.html***. Diakses tanggal 18 Mei 2006 pukul 22.00 WIB

[2] Heapsort - Wikipedia, the free encyclopedia. ***http://en.wikipedia.org/wiki/Heap_sort***. *D*iakses tanggal 18 Mei 2006 pukul 21.00 WIB

[3] Munir M. T., Rinaldi. 2005. *"Diktat Kuliah IF2251, Strategi Algoritmik"*. Bandung : Institut Teknologi Bandung.

[4] Three Divide and Conquer Sorting Algorithms. ***http://www.ics.uci.edu/~eppstein/161/960118.html***. Diakses tanggal 18 Mei 2006 10.00 WIB