Tugas 4 : Analisis Kompleksitas Waktu

Heap Sort

Mata Kuliah : Analisis Algoritma

M. Ahsan Nurrijal	140810160004
Hasna Karimah	140810160020
Syifa Fauziyah N. I.	140810160026

S-1 Teknik Informatika

Fakultas Matematika & Ilmu Pengetahuan Alam

Universitas Padjadjaran

Jalan Raya Bandung - Sumedang Km. 21 Jatinangor 45363

I. Algoritma Kerja

Penjelasan

Heap Sort termasuk kedalam 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 sub-masalah tersebut

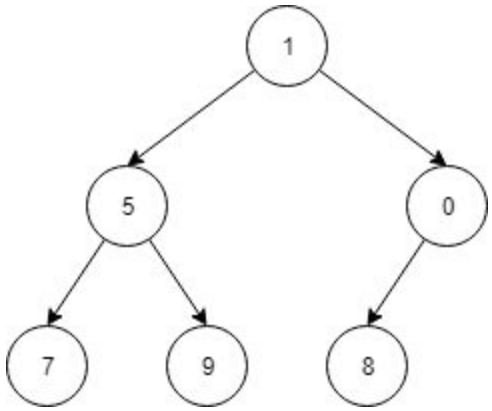
Combine : menggabungkan solusi dari sub-masalah untuk mendapatkan solusi dari persoalan semula.

Algoritma kerja

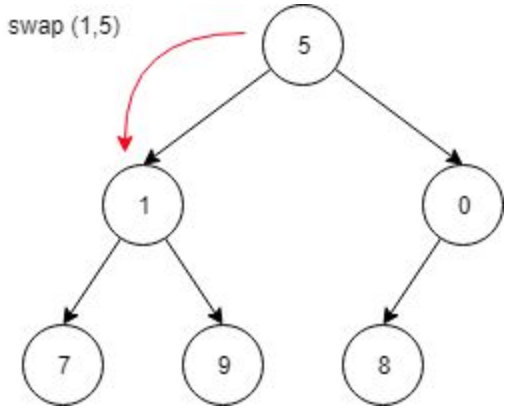
Terdapat array a dengan n = 6

1	5	0	7	9	8

Langkah 1 : Langkah pertama adalah membuat array yang sebelumnya menjadi sebuah tree dengan array pertama menjadi MaxHeap (bagian paling atas)

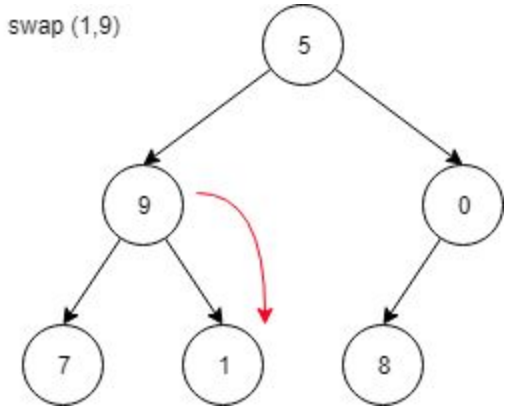


Langkah 2: Langkah selanjutnya adalah mencari nilai terbesar dari *child node* dari angka 1, dan diantara 5 dan 0 yang merupakan *child note* dari 1, angka 5 adalah yang terbesar, maka 1 dan 5 di *swap*



5	1	0	7	9	8

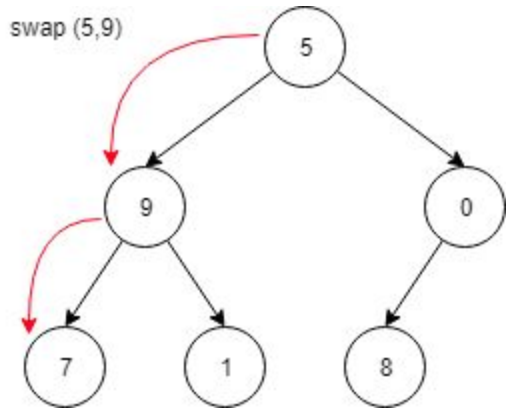
Langkah 3: Selanjutnya, angka 1 kembali mencari nilai *child node* terbesar diantara 7 dan 9. Dikarenakan angka 9 yang terbesar, maka 9 di swap dengan 1.



5	9	0	7	1	8

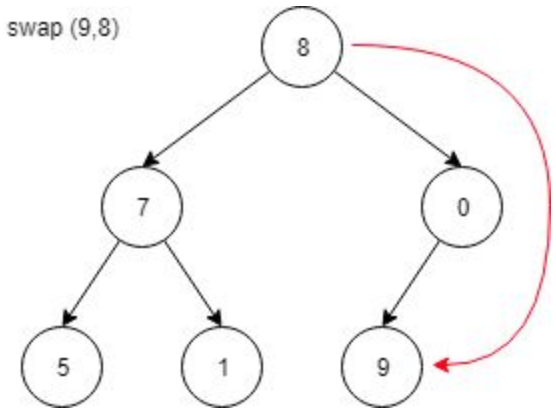
Langkah 4: Sekarang, angka 5 menjadi maxHeap, dan sekarang kita akan mencari nilai terbesar dari *child node* angka 5, dan yang terbesar adalah 9. Oleh karena itu,

5 dan 9 di swap. Dan karena pada *child node* setelahnya 7 lebih besar dari 5, makan 5 swap dengan 7.



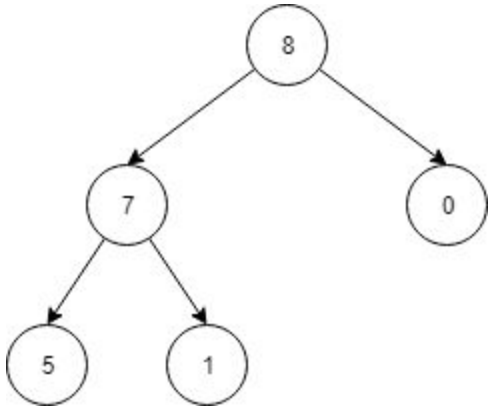
5	9	0	7	1	8

Langkah 5 : Dikarenakan 9 adalah maxHeap dan yang tertinggi daripada nilai yang lain, maka, 9 akan di swap dengan nilai yang paling akhir yaitu 8.



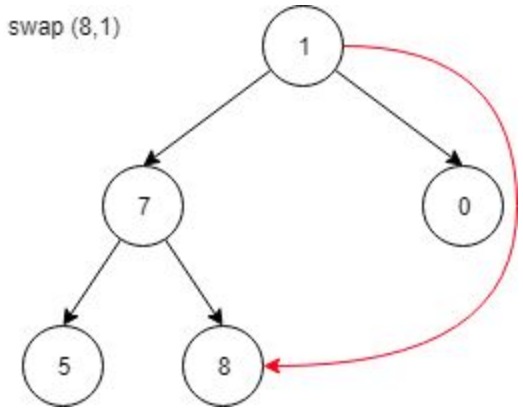
8	7	0	5	1	9

Langkah 6 : 9 adalah nilai max dari semuanya dan tidak akan berganti tempat dengan yang lain lain, oleh karena itu 9 dihilangkan dari tree



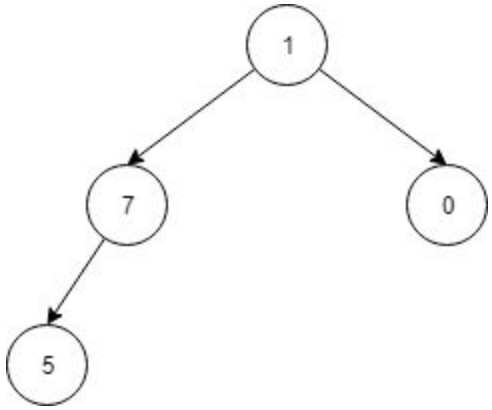
8	7	0	5	1	9

Langkah 7: Dikarenakan 1 adalah maxHeap dan yang tertinggi daripada nilai yang lain, maka, 8 akan di swap dengan nilai yang paling akhir yaitu 1.



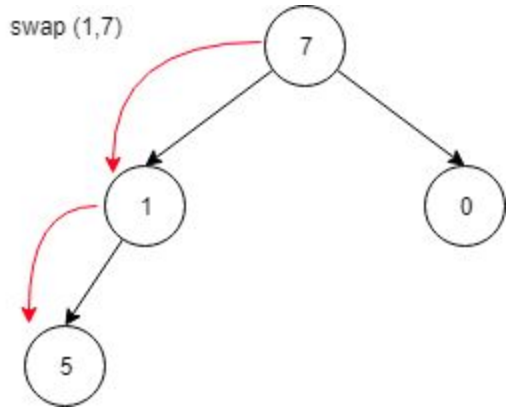
1	7	0	5	8	9

Langkah 8: 8 adalah nilai max dari semuanya dan tidak akan berganti tempat dengan yang lain lain, oleh karena itu 8 dihilangkan dari tree



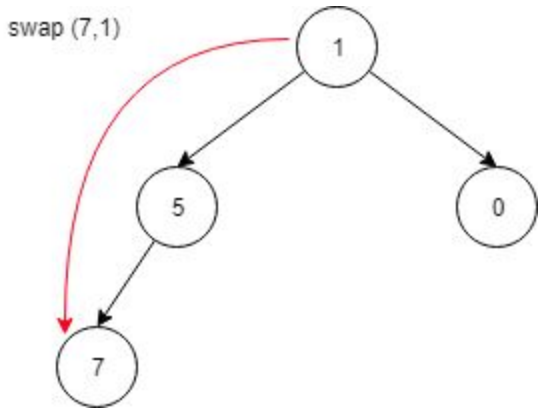
1	7	0	5	8	9

Langkah 9 : Selanjutnya, angka 1 mencari nilai *child node* terbesar diantara 7 dan 0. Dikarenakan angka 7 yang terbesar, maka 7 di swap dengan 1. lalu 1 dibandingkan dengan *child node* yaitu 5. karena 5 lebih besar dari 1 maka di *swap*.



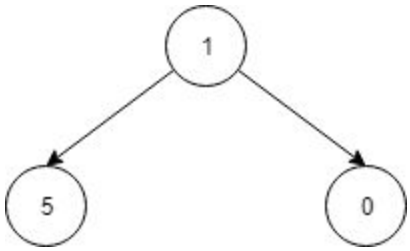
7	1	0	5	8	9

Langkah 10 : Dikarenakan 7 adalah maxHeap dan yang tertinggi daripada nilai yang lain, maka, 7 akan di swap dengan nilai yang paling akhir yaitu 1.



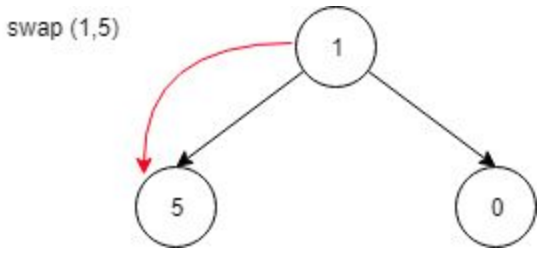
1	5	0	7	8	9

Langkah 11 : 7 adalah nilai max dari semua nilai dan tidak akan berganti tempat dengan yang lain, maka angka 7 dihilangkan dari tree.



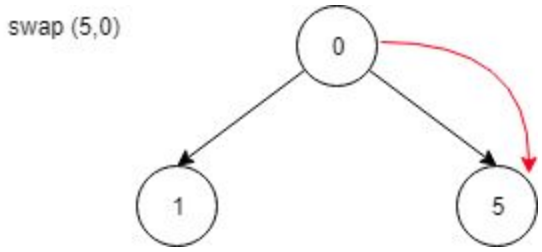
1	5	0	7	8	9

Langkah 12 : Selanjutnya, angka 1 mencari nilai *child node* terbesar di antara 5 dan 0. Dikarenakan angka 5 yang terbesar, maka 5 di swap dengan 1.



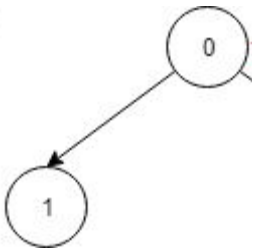
1	5	0	7	8	9

Langkah 13 : Dikarenakan 5 adalah maxHeap dan yang tertinggi daripada nilai yang lain, maka, 5 akan di swap dengan nilai yang paling akhir yaitu 0.



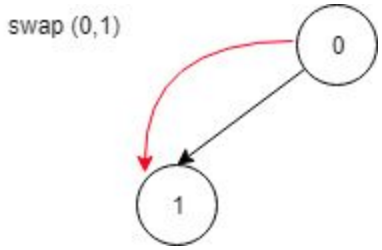
0	1	5	7	8	9

Langkah 14 : 5 adalah nilai max dari semua nilai dan tidak akan berganti tempat dengan yang lain, maka angka 5 dihilangkan dari tree.



0	1	5	7	8	9

Langkah 15 : Selanjutnya, angka 0 dibandingkan dengan child nodenya yaitu 1. Dikarenakan angka 1 lebih besa, maka 1 di swap dengan 0.



0	1	5	7	8	9

II. Kompleksitas Algoritma

Algoritma 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
  }
}
```

Algoritma

Untuk mencari left child, right child, dan parent digunakan rumus sebagai berikut :

- Left Child : 2i (Contoh : Left child dari 1 adalah 2 x 1 = 2)

- Right Child : 2i + 1 (Contoh : Right Child dari 1 adalah (2 x 1) + 1 = 3)
- Parent : ⌊ i/2 ⌋ (Contoh : Parent dari 3 adalah 3 / 2 = 1)

HeapSort(A)

```
1.    Deklarasi array A
2.    Deklarasi Elemen
3.    Input elemen array A
4.    Input nilai-nilai elemen array A
5.    Build-Max-Heap(A)
6.    For i = Elemen – 1 selama i > 0
7.          Tukar A[i] dengan A[0]
8.          Elemen – 1
9.          Max-Heapfy(A, 1)
10.   End for
```

Build-Max-Heap(A)

```
1.    For i = (Elemen - 1) / 2 selama i ≥ 0
2.          Max-Heapfy(A, i)
3.    End for
```

Max-Heapfy(A, i)

```
1.    Deklarasi left = (i + 1) * 2 – 1
2.    Deklarasi right = (i + 1) * 2
3.    Deklarasi largest
4.    if(left < elemen dan A[left] > A[i])
5.          largest = left
6.    end if
7.    else
8.          largest = i
9.    end else
10. if(right < elemen dan A[right] > A[i])
11.       largest = right
12. end if
13. if(largest != i)
14.       Tukar A[i] dengan A[largest]
15.       Max-Heapfy(A, i)
16. end if
```

Kompleksitas waktu

Algoritma heap sortmembutuhkan n langkah untuk membentuk heap. Selain itu, heap sort juga membutuhkan (n-1) log n langkah untuk mengeluarkan elemen dari heapdan menambahkannya pada arraybaru. Oleh karena itu dibutuhkan total langkah sebesar

$$T(n) = n + (n-1)\log log\ n = n \log log\ n \qquad (1)$$

Adapun rinciannya sebagai berikut :

Dimisalkan T(n) adalah waktu yang dibutuhkan untuk menjalankan algoritma **heap sort** pada array dengan n elemen. Maka persamaan T(n) adalah

$$T(n) = T_{building}(n) + \sum_{k=1}^{n-1} T_{heapify}(k) + \theta(n-1) \qquad (2)$$

Proses heapify juga digunakan didalam buidheap. Oleh karena itu heapify akan dijabarkan terlebih dahulu

$$T_{heapify}(n) = \theta(1) + T_{heapify}(size\ of\ subtree) \qquad (3)$$

Jika suatu heap A memiliki ukuran n, maka ukuran dari subtree heap tersebut kurang dari atau sama dengan 2n/3. Dengan demikian, nila nilai tersebut disubstitusikan pada persamaan (14), maka didapat pada persamaan (3), maka didapat

$$T_{heapify}(n) = \theta(1) + T_{heapify}\left(\tfrac{2n}{3}\right)$$

$$T_{heapify}(n) = \theta(n \log log\ n) \qquad (4)$$

Kompleksitas waktu buildheap sebesar O(n log n). Jika persamaan (4) disubstitusikan pada persamaan (2) maka didapat

$$T(n) = T_{building}(n) + \sum_{k=1}^{n-1} T_{heapify}(k) + \theta(n-1)$$

$$T(n) = \theta(n \log log\ n) + \sum_{k=1}^{n-1} \log log\ k + \theta(n-1)$$

$$T(n) = \theta(n \log log\ n)$$

Didapat nilai kompleksitas waktu asimptotik θ(n log n) = O(n log n). Kompleksitas waktu ini berlaku untuk semua kondisi baik best case, worst case, maupun average case.

III. Code Program

```
#include<iostream>
```

```
#include <chrono>
using namespace std;
using namespace std::chrono;

// A function to heapify the array.
void MaxHeapify(int a[], int i, int n)
{
	int j, temp;
	temp = a[i];
	j = 2*i;

	while (j <= n)
	{
	   if (j < n && a[j+1] > a[j])
	   j = j+1;
	   // Break if parent value is already greater than child value.
	   if (temp > a[j])
		  break;
	   // Switching value with the parent node if temp < a[j].
	   else if (temp <= a[j])
	   {
		  a[j/2] = a[j];
		  j = 2*j;
	   }
	}
	a[j/2] = temp;
	return;
}
void HeapSort(int a[], int n)
{
	int i, temp;
	for (i = n; i >= 2; i--)
	{
	   // Storing maximum value at the end.
	   temp = a[i];
	   a[i] = a[1];
	   a[1] = temp;
	   // Building max heap of remaining element.
	   MaxHeapify(a, 1, i - 1);
	}
}
void Build_MaxHeap(int a[], int n)
{
	int i;
	for(i = n/2; i >= 1; i--)
	   MaxHeapify(a, i, n);
}
int main()
{
	int n, i;
	cout<<"\nEnter the number of data element to be sorted: ";
	cin>>n;
	n++;
	int arr[n];
```

```
       for(i = 1; i < n; i++)
       {
          cout<<"Enter element "<<i<<": ";
          cin>>arr[i];
       }
       // Building max heap.
       auto start = high_resolution_clock::now();
       Build_MaxHeap(arr, n-1);
       HeapSort(arr, n-1);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(stop - start);

       // Printing the sorted data.
       cout<<"\nSorted Data ";

       for (i = 1; i < n; i++)
          cout<<"->"<<arr[i];

       cout << "Time taken by function: "
      << duration.count() << " nanoseconds" << endl;

       return 0;
}
```

Output :

```
Enter the number of data element to be sorted: 6
Enter element 1: 1
Enter element 2: 5
Enter element 3: 0
Enter element 4: 7
Enter element 5: 9
Enter element 6: 8

Sorted Data ->0
->1
->5
->7
->8
->9
Time taken by function: 826 nanoseconds
```