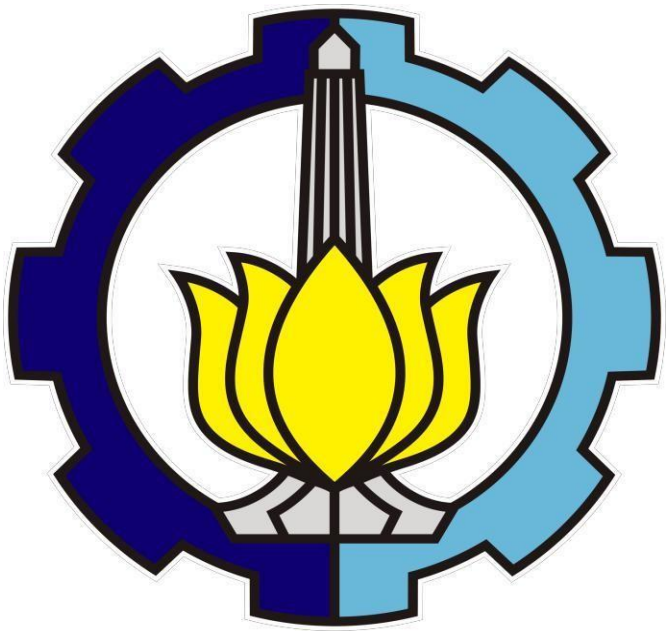
TUGAS 3

DESAIN DAN ANALISIS ALGORITMA



Oleh:

Muhammad Nanda Setiawan (06111740000037)

Dosen:

Drs. Bandung Arry Sanjoyo, MI.Komp.
19630605 198903 1 003

DEPARTEMEN MATEMATIKA

FAKULTAS SAINS DAN ANALITIKA DATA (FSAD)

INSTITUT TEKNOLOGI SEPULUH NOPEMBER

SURABAYA

2020

A. Deskripsi Tugas

Pelajari perbedaan strategi dari masing-masing metoda sorting berikut dan buatlah implementasinya. Kemudian, lakukan analisis dari masing-masing metoda. Lakukan run program untuk ukuran data 20, 50, 200, 500, 1000, 10.000, 100.000, 500.000, 1.000.000, 2.000.000. Buatlah ploting dari waktu yang diperlukan untuk me-running data-data tersebut.

1. Bubble Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort

B. Pembahasan

1. Strategi Sorting

a. Bubble Sort

Bubble sort (metode gelembung) adalah metode/algoritma pengurutan dengan dengan cara melakukan penukaran data dengan tepat disebelahnya secara terus menerus sampai bisa dipastikan dalam satu iterasi tertentu tidak ada lagi perubahan.

Metode ini diinspirasikan oleh gelembung sabun yang berada dipermukaan air. Karena berat jenis gelembung sabun lebih ringan daripada berat jenis air, maka gelembung sabun selalu terapung ke atas permukaan. Algoritma ini adalah salah satu algoritma pengurutan yang paling simpel, baik dalam hal pengertian maupun penerapannya. Ide dari algoritma ini adalah mengulang proses pembandingan antara tiap-tiap elemen array dan menukarnya apabila urutannya salah. Pembandingan elemen-elemen ini akan terus diulang hingga tidak perlu dilakukan penukaran lagi.

Algoritma-nya:

```
BubbleSort(A, n){
1.  for (i=0; i<n; i++){
2.      for (j=0; j<n-1; j++){
3.          if (A[j]>A[j+1]){
4.              temp = arr[j]
5.              A[j]= A[j+1]
6.              A[j+1]=temp
7.          }
8.      }
9.  }
  }
```

b. Insertion Sort

Insertion sort adalah sebuah metode pengurutan data dengan menempatkan setiap elemen data pada pisisinya dengan cara melakukan perbandingan dengan data – data yang ada. Ide algoritma dari metode insertion sort ini dapat dianalogikan

sama seperti mengurutkan kartu, dimana jika suatu kartu dipindah tempatkan menurut posisinya, maka kartu yang lain akan bergeser mundur atau maju sesuai kondisi pemindahanan kartu tersebut. Dalam pengurutan data, metode ini dipakai bertujuan untuk menjadikan bagian sisi kiri array terurutkan sampai dengan seluruh array diurutkan.

Algoritma-nya:

```
InsertionSort(A, n){
1.  for (j=1; j<n; j++){
2.      key = arr[j]
3.      i=j-1
4.      while(i>=0 && A[i]>key){
5.              A[i+1] = A[i]
6.              i--
7.       }
8.       A[i+1]=key
9.   }
10. return A
}
```

c. Merge Sort

Algoritma pengurutan data merge sort dilakukan dengan menggunakan cara divide and conquer yaitu dengan memecah kemudian menyelesaikan setiap bagian kemudian menggabungkannya kembali. Pertama data dipecah menjadi 2 bagian dimana bagian pertama merupakan setengah (jika data genap) atau setengah minus satu (jika data ganjil) dari seluruh data, kemudian dilakukan pemecahan kembali untuk masing-masing blok sampai hanya terdiri dari satu data tiap blok.

Setelah itu digabungkan kembali dengan membandingkan pada blok yang sama apakah data pertama lebih besar daripada data ke-tengah+1, jika ya maka data ke-tengah+1 dipindah sebagai data pertama, kemudian data ke-pertama sampai ke-tengah digeser menjadi data ke-dua sampai ke-tengah+1, demikian seterusnya sampai menjadi satu blok utuh seperti awalnya. Sehingga metode merge sort merupakan metode yang membutuhkan fungsi rekursi untuk penyelesaiannya.

Algoritma dirumuskan dalam 3 langkah berpola divide-and-conquer.

Proses rekursi yang ada pada metode ini berhenti jika mencapai elemen dasar. Hal ini terjadi bilamana bagian yang akan diurutkan menyisakan tepat satu elemen. Sisa pengurutan satu elemen tersebut menandakan bahwa bagian tersebut telah terurut sesuai rangkaian.

Algoritma-nya:

```
MergeSort(A, p, r){
1.  if(l<r){
2.      int m = (l+r)/2;
3.      MergeSort(A, l, m);
```

```
4.      MergeSort(A, m+1, r);
5.      merge(A, l, m, r);
6.  }
}
```

d. Quick Sort

Algoritma quick sort mengurutkan dengan sangat cepat, namun algoritma ini sangat komplex dan diproses secara rekursif. Algoritma ini merupakan suatu algoritma pengurutan data yang menggunakan teknik pemecahan data menjadi partisi-partisi, sehingga metode ini disebut juga dengan nama *partition exchange* sort. Untuk memulai irterasi pengurutan, pertama-tama sebuah elemen dipilih dari data, kemudian elemen-elemen data akan diurutkan diatur sedemikian rupa. Metode ini mempunyai efektifitas yang tinggi dengan teknik menukarkan dua elemen dengan jarak yang cukup besar.

Algoritma-nya:

```
QuickSort(A, p, r){
1.  if (p < r){
2.      Partition(A, p, r)
3.      QuickSort(A, p, q-1)
4.      QuickSort(A, q+1,  r)
5.  }
}

Partition(A, p, r){
1.  x = A[r]
2.  i = p-1
3.  for (j=p; j<r; j++){
4.      if (A[j] <= x){
5.          i++
6.          Tukar A[i] dengan A[j]
7.      }
8.  }
9.  Tukar A[i+1] dengan A[r]
10. return i+1
}
```

e. Heap Sort

Heap sort adalah sebuah metode sorting (pengurutan) angka pada sebuah array dengan cara menyerupai binary tree, yaitu dengan cara memvisualisasikan sebuah array menjadi sebuah binary tree yang nantinya pada binary tree tersebut nilai pada masing-masing index array akan diurutkan. Pada heap sort terdapat 3 bagian yaitu Node, Edge, dan leaf dimana node itu adalah setiap index yang berada pada array, edge adalah garis yang menghubungkan tiap node dan leaf adalah setiap node yang

tidak memiliki child node (node turunan). Selain itu juga ada yang bernama root yaitu node awal pada sebuah heap,

Heap tree terbagi menjadi 2 jenis yaitu Max-Heap dan Min-Heap, dimana max-heap adalah kondisi heap tree yang memiliki nilai tertinggi berada di node root dan setiap child node memiliki nilai yang lebih kecil dari nilai yang dimiliki parent nodenya. Sedangkan pada min-heap adalah kondisi kebalikan dengan max-heap, pada min-heap nilai terkecil berada di node root dan setiap child node memiliki nilai yang lebih besar dari nilai yang dimiliki parent nodenya. Pada metode heap sort jenis heap tree yang digunakan adalah Max-Heap.

Algoritmanya:

```
HeapSort (A, length){
1.  BuildMaxHeap(A)
2.      for (i=length-1; i>0; i--){
3.          Swap(A[0] , A[i])
4.          MaxHeapify(A, i, 0)
5.      }
}

BuildMaxHeap (A){
1.  length = A.length;
2.  for (i=length/2-1; i>=0; i--)
3.      MaxHeapify(A, length, i)
}

MaxHeapify(A, n, i){
1.  l=2*i+1
2.  r=2*i+2
3.  if (l<n && A[l]>A[i])
4.      largest = l
5.  else
6.      largest = i
7.  if (r<n && A[r]>A[largest])
8.      largest = r
9.  if (largest !=i ){
10.     Swap(A[largest], A[i])
11.     MaxHeapify(A, n, largest)
12. }
}
```

2. Analisis Kompleksitas Waktu

a. Bubble Sort

Diketahui Algoritma BubbleSort seperti berikut:

1. for (i=0; i<n; i++){

```
2.      for (j=0; j<n-1; j++){
3.          if (A[j]>A[j+1]){
4.              temp = arr[j]
5.              A[j]= A[j+1]
6.              A[j+1]=temp
7.          }
8.      }
9.  }
```

Sebelum menghitung kompleksitasnya, tentukan dahulu proses utama yang dilakukan oleh algoritmanya. Dari algoritma terlihat bahwa proses utamanya adalah melakukan pembandingan, yaitu pada line 3. Asumsikan dalam melakukan pembandingan dalam waktu konstan yaitu $O(1) = 1$, sehingga:

$$
\begin{aligned}
Time\ complexity &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} 1 \\
&= \sum_{i=0}^{n-1} \big((n-i-1)-0\big) + 1 \\
&= \sum_{i=0}^{n-1} n - i \\
&= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i \\
&= n \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i \\
&= n(\big((n-1)-0\big)+1) - \frac{n(n+1)}{2} \\
&= n^2 - \frac{n^2+2}{2} \\
&= \frac{n^2-n}{2} \\
&= O(n^2)
\end{aligned}
$$

Jika diperhatikan lebih lanjut, apabila diberikan input berupa array yang sudah terurut (*Best case*) maupun input berupa array setengah terurut (*Average case*), algoritma ini akan tetap melakukan kerja pembandingan yang sama banyak, sehingga Time complexity untuk Base case dan Average case sama dengan Worst case atau dapat ditulis $\Omega(n) = \Theta(\mathrm{n}) = O(n) = n^2$

b. Insertion Sort

Diketahui algoritma Insertion Sort sebagai berikut:

```
1.  for (j=1; j<n; j++){
```

```
2.      key = arr[j]
3.      i=j-1
4.      while(A[i]>key && i>=0){
5.              A[i+1] = A[i]
6.              i--
7.       }
8.       A[i+1]=key
9.   }
10. return A
```

- ***Best case***

  Kasus *best case* terjadi apabila input yang diberikan berupa Array n element yang sudah terurut. Ini berarti kerja utama yang dilakukan pada baris 4. Dalam statement pada baris 4 menggunakan operator logic &&(dan) dalam perbandingannya, sehingga jika salah satu argumen tidak terpenuhi akan langsung bernilai salah. Maka, yang dilakukan cukup membandingkan *A[i]* dengan *key*. Pembandingan ini dilakukan dengan *loping* mulai dari indeks j=1 sampai j<n atau j=n-1. Dari sini dapat dilakukan perhitungan kompleksitas waktunya sebagai berikut:

  $$Time\ complexity = \sum_{j=1}^{n-1} 1$$
  $$= \big((n-1)-1\big)+1$$
  $$= n-1$$
  $$= \Omega(n)$$

  Terlihat bahwa untuk *best case* insertion sort memiliki $Time\ complexity = \Omega(n)$

  .

- ***Worst case***

  Kasus *Worst case* terjadi apabila diberikan input berupa array dalam keadaan kondisi terurut menurun. Hal ini akan menyebabkan statement *while* pada baris 4 selalu bernilai *true* sehingga akan melakukan kerja utama pada baris 5. Akibatnya program akan melakukan 2 kali *looping*, yaitu *outer looping for* dari indeks j=1 sampai n-1 dan *inner looping* dari i=0 sampai i=j-1. Maka perhitungan kompleksitas waktunya sebagai berikut:

  $$Time\ complexity = \sum_{j=1}^{n-1}\sum_{i=0}^{j-1} 1$$
  $$= \sum_{j=1}^{n-1}\big((j-1)-0\big)+1$$

$$
\begin{aligned}
&= \sum_{j=1}^{n-1} j \\
&= 1 + 2 + \cdots + n - 1 \\
&= (1 + 2 + \cdots + n - 1) + n - n \\
&= \sum_{j=1}^{n} j - n \\
&= \frac{n(n+1)}{2} - n \\
&= \frac{n^2 - n}{2} \\
&= O(n^2)
\end{aligned}
$$

Terlihat bahwa untuk *worst case* insertion sort memiliki $Time\ complexity = O(n)$

- ***Average case***
  Kasus *average case* terjadi apabisa algoritma dibeikan input berupa Array dengan keadaan setengah terurut naik. Ini akan menyebabkan statement pada baris 4 bisa bernilai benar dan bisa bernilai salah dalam *outer looping for*-nya. Sehingga untuk menghitung kompleksitas waktu nya seperti pada kasus *worst case* dengan melakukan dua kali menghitung dengan notasi sigma dengan batas yang sama. Perbedaannya terdapat pada *inner sum*. Jika pada worst case bernilai 1, maka untuk *average case* nilainya ada Expekstasi dari i atau $E[i] = (0 + 1).p(i)$.dimana 0 menyatakan kondisi *false* untuk statement pada baris 4 dalam melakukan pembandingan dan 1 menyatakan kondisi *true*-nya. Dari sini terlihat bahwa $E[i]$ akan bernilai suatu konstanta yang mengakibatkan hasil akhir perhitungan kompleksitasnya juga akan menghasilkan n^2 atau dapat ditulis $\Theta(\mathrm{n}) = n^2$

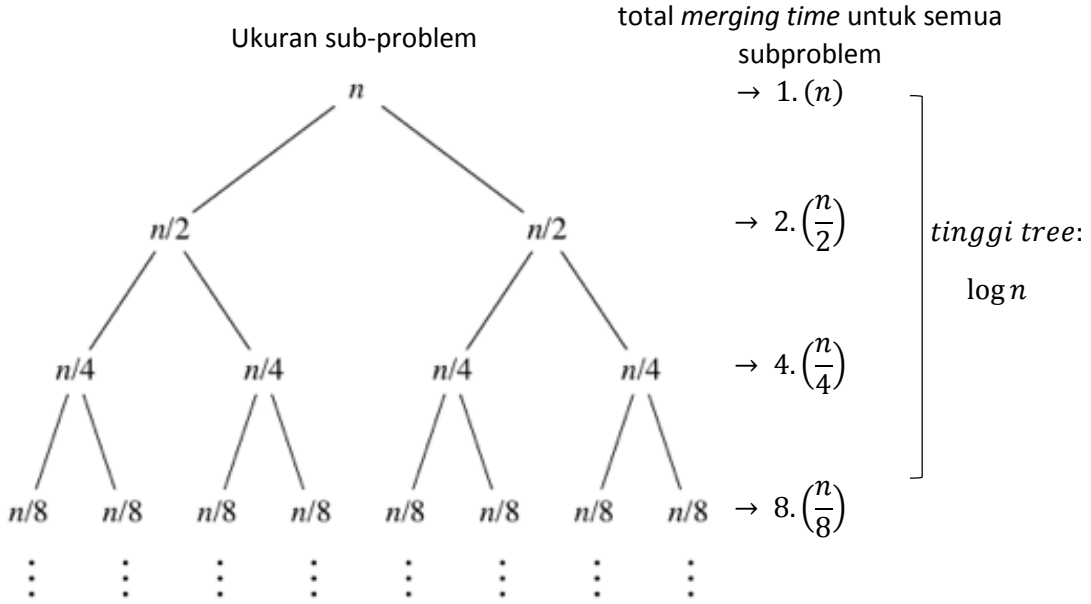
c. Merge Sort
Mengingat bahwa merge sort berjalan di $\Theta\ (n)$ saat menggabungkan n elemen, akan ditunjukkan bahwa merge sort berjalan di $\Theta\ (n * log2\ n)$. Diasumsikan input berupa array dengan n elemen. Perhitungan dalam strateginya sebagai berikut

1. Langkah *divide* membutuhkan waktu yang konstan, terlepas dari ukuran subarray. Bagaimanapun, langkah membagi hanya menghitung titik tengah q dari indeks p dan r. Ingat bahwa dalam notasi $big - \Theta$, waktunya konstan sebesar $\Theta\ (1)$.
2. Langkah *conquer*, di mana secara rekursif menyortir dua subarrays masing-masing sekitar $n/2$ elemen, membutuhkan sejumlah waktu, tetapi akan dihitung waktu ketika mempertimbangkan subproblem.
3. Langkah gabungan menggabungkan total n elemen, dengan waktu $\Theta\ (n)$.

Andaikan langkah *divide* dan gabungan bersama, running time Θ (1) untuk langkah *divide* didominasi oleh running time $\Theta\ (n)$ dari langkah gabungan. Diasumsikan running time untuk langkah *divide* dan gabungan adalah Θ (n).

Kemudian, agar lebih sederhana asumsikan bila $n > 1$. n selalu genap, sehingga untuk $n/2$ adalah bilangan bulat (perhitungan untuk kasus di mana n ganjil tidak mengubah hasil dalam hal notasi $big - \Theta$). Jadi, sekarang didapat bahwa running time untuk algoritma *merge sort* pada suatu n-elemen subarray sebagai jumlah dari dua kali running time *merge sort* pada $(n\ /\ 2)$- elemen subarray (untuk langkah c*onquer*) ditambah n, n (pada langkah *divide* dan gabungan).

Untuk proses strategi algoritma ini dapat digambarkan dengan tree sebagai berikut:



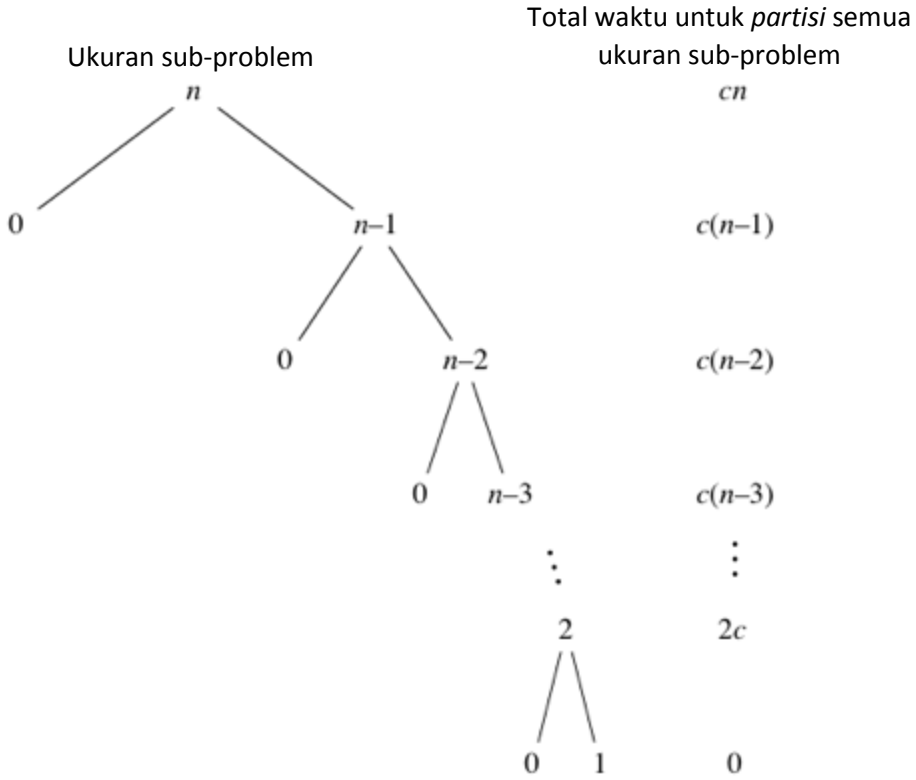
Dari ilustrasi terlihat bahwa perhitungan kompleksitas waktu algoritma ini dapat diperumum sebagai berikut:

$$
\begin{aligned}
Time\ complexity &= \sum_{i=0}^{\log n} 2^i \left(\frac{n}{2^i}\right) \\
&= \sum_{i=0}^{\log n} n \\
&= n.((\log n - 0) + 1) \\
&= n.(\log n + 1) \\
&= n.\log n + n \\
&= O(n \log n)
\end{aligned}
$$

d. Quick Sort

- ***Worst case***

Untuk kasus *worst case* terjadi apabila hasil dari proses partisi sangat tidak seimbang. Khususnya, asumsikan bahwa pivot yang dipilih dari fungsi *partition* adalah elemen yang terkecil atau yang terbesar antara n-element subarray. Maka, salah satu partisi tidak memiliki elemen dan partisi lainnya memiliki n-1 elemen (kecuali pivotnya). Ini akan mengakibatkan pemanggilann rekursif akan berada di subarray ukuran 0 dan n-1. Dalam kondisi yang sangat tidak seimbang seperti ini, pemanggilan rekursif dengan n-1 element sebanyak $c(n-1)$ untuk suatu konstanta c. Kemudian untuk pemanggilan rekursif dengan n-2 element sebanyak $c(n-2)$, dst. pemanggilan ini dapat digambarkan dengan tree sebagai berikut:

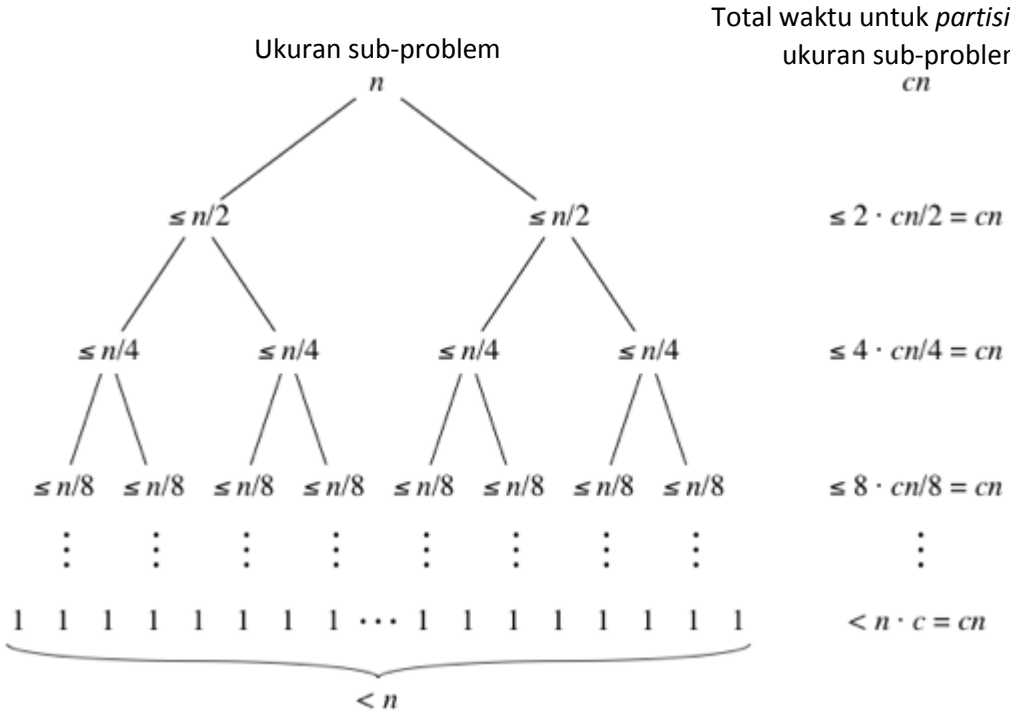


Dari ilustrasi tree di atas, total waktu untuk melakukan *paritsi* ditiap levelnya:

$$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n + (n-1) + (n-2) + \cdots + 2)$$
$$= c((n+1)\left(\frac{n}{2}\right) - 1)$$
$$= O(cn^2)$$
$$= O(n^2)$$

- ***Best case***

  Kasus terbaik terjadi ketika hasil proses partisi sangat seimbang. Ini terjadi ketika subarray memiliki jumlah elemen ganjil dan pivot tepat berada di tengah subarray, dan tiap partisi memiliki (n-1)/2 element. Ilustrasi untuk kasus ini dapat digambarkan dengan tree sebagai berikut:



Hasil perhitungan dari ilustrasi ini akan sama dengan merge sort, yaitu $\Theta(n \log n)$

e. Heap Sort

Untuk menggunakan algoritma Heap Sort, input berupa array terlebih dahulu dibentuk menjadi *max-heap*. Untuk membuat *max-heap* dari input array yang tidak terurut membutuhkan sebanyak $O(n)$ pemanggilan fungsi *MaxHeapify*. Untuk setiap kali proses *MaxHeapify* sendiri melakukan running time sebanyak $O(\log n)$. Sehingga, total running time untuk fungsi *BuildMaxHeap* adalah $O(n \log n)$.

Heap Sort running time berada di $O(n \log n)$, sebab pemanggilan *BuildMaxHeap* memerlukan waktu $O(n \log n)$ dan tiap pemanggilan $O(n)$ dari *MaxHeapify* memerlukan waktu $O(\log n)$.

3. Implementasi

Implementasi kelima algoritma dilakukan dengan memberikan input berupa 5 array berbeda. kelima array ini berukuran n-elemen dengan isi elemen yang sama agar menjamin kesamarataan input untuk setiap algoritma. Elemen yang diisi pada setiap array di n input menggunakan pembangkitan angka acak. Untuk menunjukkan bahwa array disorting dengan benar, akan diambil sampel untuk array berukuran 20 sebelum disorting dan ditunjukkan hasil sortingannya diakhir untuk tiap algoritma yang digunakan.

Perhitungan runtime tiap algoritma dilakukan menghitung selisih waktu saat akan menjalankan dan setelah selesai menggunakan algoritma. Kemudian, hasilnya disajikan dalam bentuk tabel untuk tiap ukuran input n. Dengan menggunakan bahasa pemrograman Jawa, Source Codenya sebagai berikut:

```java
package tugas3;
/**
 *
 * @author Muhammad Nanda Setiawan-06111740000037
 */
public class Tugas3 {
    public static void main(String[] args) {
        int[] input_n ={20, 50, 200, 500, 1000, 10000, 100000,
                        500000, 1000000, 2000000};
        int[] sampel_awal = new int[20];
        int[] sampel_alg1_akhir = new int[20];
        int[] sampel_alg2_akhir = new int[20];
        int[] sampel_alg3_akhir = new int[20];
        int[] sampel_alg4_akhir = new int[20];
        int[] sampel_alg5_akhir = new int[20];

        System.out.println("Perbandingan Runnung Time hasil
        implementasi 5 Algoritma Sorting untuk beberapa ukuran
        input N");

        //tabel
        String s="n    ";
        System.out.println("=====================================
        ============================================================
        ============================================================
        ==========");
        System.out.println("| " +String.format("%9S",s) +" |"
         +"\t" +String.format("%17s",("Bubble Sort "))+ " \t|"
         +"\t" +String.format("%17s",("Insertion Sort "))+ " \t|"
         +"\t" +String.format("%17s",("Merge Sort "))+ " \t|"
         +"\t" +String.format("%17s",("Quick Sort "))+ " \t|"
         +"\t" +String.format("%17s",("Heap Sort "))+ " \t|"
        );
        System.out.println("=====================================
        ============================================================
        ============================================================
        ===========");

        //proses utama
        for (int n:input_n){
            //inisialisasi array untu digunakan disetiap algoritma
            int[] arr_alg1 = new int[n];
            int[] arr_alg2 = new int[n];
            int[] arr_alg3 = new int[n];
            int[] arr_alg4 = new int[n];
            int[] arr_alg5 = new int[n];

            /*membangkitkan angka acak agar menghailkan input yang
            sama disetiap array untuk diimplementasikan disetiap
            Algoritma Sorting*/
            for (int i=0; i<n; i++){
                int a= (int)(Math.random()*(2*n+1))+(-n);

                 arr_alg1[i]=arr_alg2[i]=arr_alg3[i]=arr_alg4[i]=arr
                 _alg5[i]=a;
                if (n==20)
                    sampel_awal[i]=a;
            }

            //Proses penggunaan algoritma dan pencatatan running
            time-nya
```

```
        System.out.print("| " +String.format("%9d",n) +" |");
        float a, b, c;
        a = System.nanoTime();
        BubbleSort(arr_alg1, n);
        b = System.nanoTime();
        System.out.print("\t" +RunningTime(a,b) +"\t|");

        a = System.nanoTime();
        InsertionSort(arr_alg2, n);
        b = System.nanoTime();
        System.out.print("\t" +RunningTime(a,b) +"\t|");

        a = System.nanoTime();
        MergeSort(arr_alg3, 0, n-1);
        b = System.nanoTime();
        System.out.print("\t" +RunningTime(a,b) +"\t|");

        a = System.nanoTime();
        QuickSort(arr_alg4, 0, n-1);
        b = System.nanoTime();
        System.out.print("\t" +RunningTime(a,b) +"\t|");

        a = System.nanoTime();
        HeapSort(arr_alg5, n);
        b = System.nanoTime();
        System.out.print("\t" +RunningTime(a,b) +"\t|");
        System.out.println("");

        //untuk sampel hasil sorting tiap algoritma
        if (n==20){
            isi(arr_alg1, sampel_alg1_akhir);
            isi(arr_alg2, sampel_alg2_akhir);
            isi(arr_alg3, sampel_alg3_akhir);
            isi(arr_alg4, sampel_alg4_akhir);
            isi(arr_alg5, sampel_alg5_akhir);
        }
    }

    System.out.println("=====================================
    ============================================================
    ============================================================
    ============");
    System.out.print("\n\nSampel Array Awal untuk n=20 \t: { ");
    PrintSample(sampel_awal);
    System.out.print("}");
    System.out.print("\n\t1. Hasil Sorting dgn Algoritma Bubble
     Sort\t : ");
    PrintSample(sampel_alg1_akhir);
    System.out.print("\n\t2. Hasil Sorting dgn Algoritma
     Insertion Sort\t : ");
    PrintSample(sampel_alg2_akhir);
    System.out.print("\n\t3. Hasil Sorting dgn Algoritma Merge
     Sort\t : ");
    PrintSample(sampel_alg3_akhir);
    System.out.print("\n\t4. Hasil Sorting dgn Algoritma Hasil
     Quick Sort\t : ");
    PrintSample(sampel_alg4_akhir);
    System.out.print("\n\t5. Hasil Sorting dgn Algoritma Heap
     Sort\t : ");
    PrintSample(sampel_alg5_akhir);
    System.out.println("");
```

```
}

public static void isi (int[] awal, int[] akhir){
    for (int i=0; i<awal.length; i++)
        akhir[i]=awal[i];
}

public static void PrintSample (int[] arr){
    for (int i=1; i<=arr.length; i++)
        System.out.print(arr[i-1]+" ");
}

public static String RunningTime(float a, float b){
    float c = (b-a)/1000000000;
    if (c<60)
        return ""+String.format("%12.9f", c) +" det";
    else if (c<3600)
        return ""+String.format("%12.9f",(c/60)) +" mnt";
    else
        return ""+String.format("%12.9f",(c/3600)) +" jam";
}

public static void BubbleSort(int[] arr, int length){
    for (int i=0; i<length; i++){
        for (int j=0; j<length-1; j++){
            if (arr[j]>arr[j+1]){
                int temp = arr[j];
                arr[j]= arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
}

public static int[] InsertionSort(int[] arr, int n){
    for (int j=1; j<n; j++){
        int key = arr[j];
        int i=j-1;
        while(arr[i]>key && i>=0){
            arr[i+1]=arr[i];
            i--;
        }
        arr[i+1]=key;
    }
    return arr;
}

public static void MergeSort(int A[], int l, int r){
    if(l<r){
        int m = (l+r)/2;
        MergeSort(A, l,m);
        MergeSort(A, m+1, r);
        merge(A, l, m, r);
    }
}
public static void merge(int[]A, int l, int m, int r){
    int p1 = m-l+1;
    int p2 = r-m; //r-p1

    //mendefinisikan dua buah array kosong untuk hasil split
    array utama
```

```
        int Kr[]=new int [p1];
        int Kn[]=new int [p2];

        //split/membagi array yg besar ke dalam dua buah array
        for (int i=0; i<p1; i++)
            Kr[i]= A[l+i];
        for (int i=0; i<p2; i++)
            Kn[i]= A[m+1+i];
        int j=0,i=0,k=l;

        //proses merge-nya
        while(i<p1 && j<p2){
            if (Kr[i] <= Kn[j]){
                A[k]=Kr[i];
                i++;
            }
            else{
                A[k]=Kn[j];
                j++;
            }
            k++;
        }

        //menambah sisa elemen array Kr ke array hasil merge
        while(i<p1){
            A[k]=Kr[i];
            i++;
            k++;
        }

        //menambah sisa elemen array Kn ke array hasil merge
        while(j<p2){
            A[k]=Kn[j];
            j++;
            k++;
        }
    }
    public static void QuickSort(int[] arr, int p , int r){
        if (p<r){
            int q = Partition(arr, p, r);
            QuickSort(arr, p, q-1);
            QuickSort(arr, q+1, r);
        }
    }
    public static int Partition(int[] arr, int p, int r){
        int x = arr[r];
        int i = p-1;

        for (int j=p; j<r; j++){
            if (arr[j]<=x){
                i++;
                int exc = arr[i];
                arr[i] = arr[j];
                arr[j] = exc;
            }
        }

        int exc = arr[i+1];
        arr[i+1] = arr[r];
        arr[r] = exc;
        return i+1;
```

```
    }

    public static void HeapSort (int[] arr, int length){
        int largest;
        BuildMaxHeap(arr);

        for (int i=length-1; i>0; i--){
            largest = arr[0];
            arr[0] = arr[i];
            arr[i] = largest;
            MaxHeapify(arr, i, 0);
        }
    }
    public static void BuildMaxHeap (int[] arr){
       int length = arr.length;
       for (int i=length/2-1; i>=0; i--)
           MaxHeapify(arr, length, i);
    }
    public static void MaxHeapify(int[] arr, int n, int i){
        int l=2*i+1;
        int r=2*i+2;
        int largest, largest_val;

        if (l<n && arr[l]>arr[i])
            largest = l;
        else
            largest=i;
        if (r<n && arr[r]>arr[largest])
            largest = r;
        if (largest!=i){
            largest_val = arr[largest];
            arr[largest] = arr[i];
            arr[i] = largest_val;
            MaxHeapify(arr, n, largest);
        }
    }
}
```

4. Hasil Run

Setelah melakukan running dari program didapat hasilnya sebagai berikut:

Source History

Output - Tugas3 (run)

```
run:
Perbandingan Runnung Time hasil implementasi 5 Algoritma Sorting untuk beberapa ukuran input N
```

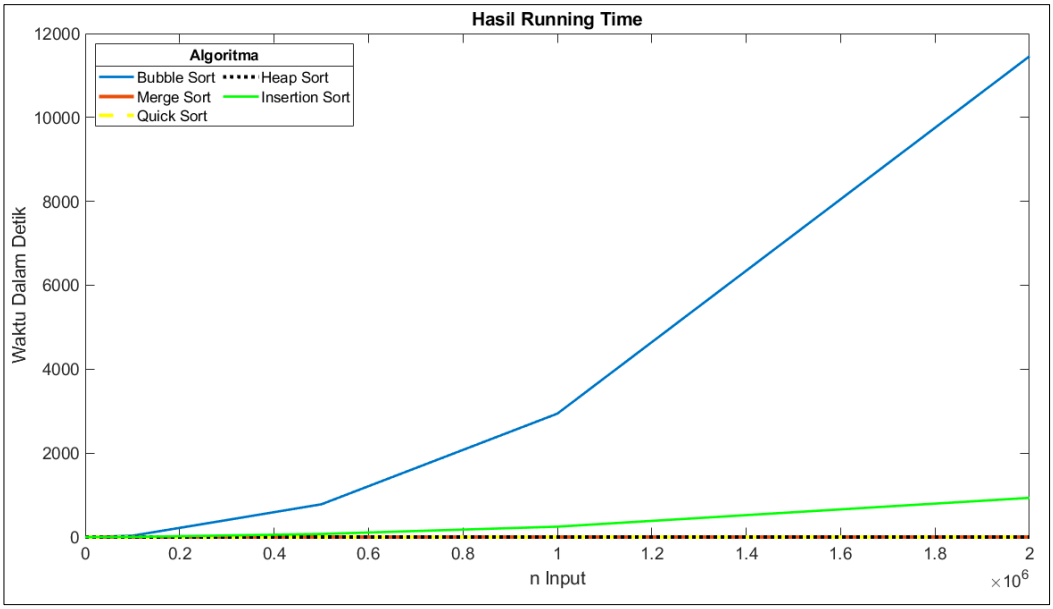
N	Bubble Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
20	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det
50	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det
200	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det
500	0,033554431 det	0,000000000 det	0,000000000 det	0,000000000 det	0,000000000 det
1000	0,000000000 det	0,000000000 det	0,033554431 det	0,000000000 det	0,000000000 det
10000	0,570425332 det	0,033554431 det	0,000000000 det	0,033554431 det	0,000000000 det
100000	31,272729874 det	3,154116631 det	0,033554431 det	0,033554431 det	0,033554431 det
500000	12,986124992 mnt	1,209637284 mnt	0,201326594 det	0,100663297 det	0,201326594 det
1000000	49,008483887 mnt	4,119925022 mnt	0,402653188 det	0,201326594 det	0,369098753 det
2000000	3,182451487 jam	15,562545776 mnt	0,905969679 det	0,369098753 det	0,838860810 det

```
Sampel Array Awal untuk n=20                          : { -18 13 12 4 -18 -4 1 -3 -9 -3 -20 -6 16 -20 2 6 15 -8 14 13 }
       1. Hasil Sorting dgn Algoritma Bubble Sort      : -20 -20 -18 -18 -9 -8 -6 -4 -3 -3 1 2 4 6 12 13 13 14 15 16
       2. Hasil Sorting dgn Algoritma Insertion Sort   : -20 -20 -18 -18 -9 -8 -6 -4 -3 -3 1 2 4 6 12 13 13 14 15 16
       3. Hasil Sorting dgn Algoritma Merge Sort       : -20 -20 -18 -18 -9 -8 -6 -4 -3 -3 1 2 4 6 12 13 13 14 15 16
       4. Hasil Sorting dgn Algoritma Hasil Quick Sort : -20 -20 -18 -18 -9 -8 -6 -4 -3 -3 1 2 4 6 12 13 13 14 15 16
       5. Hasil Sorting dgn Algoritma Heap Sort        : -20 -20 -18 -18 -9 -8 -6 -4 -3 -3 1 2 4 6 12 13 13 14 15 16
BUILD SUCCESSFUL (total time: 274 minutes 30 seconds)
```

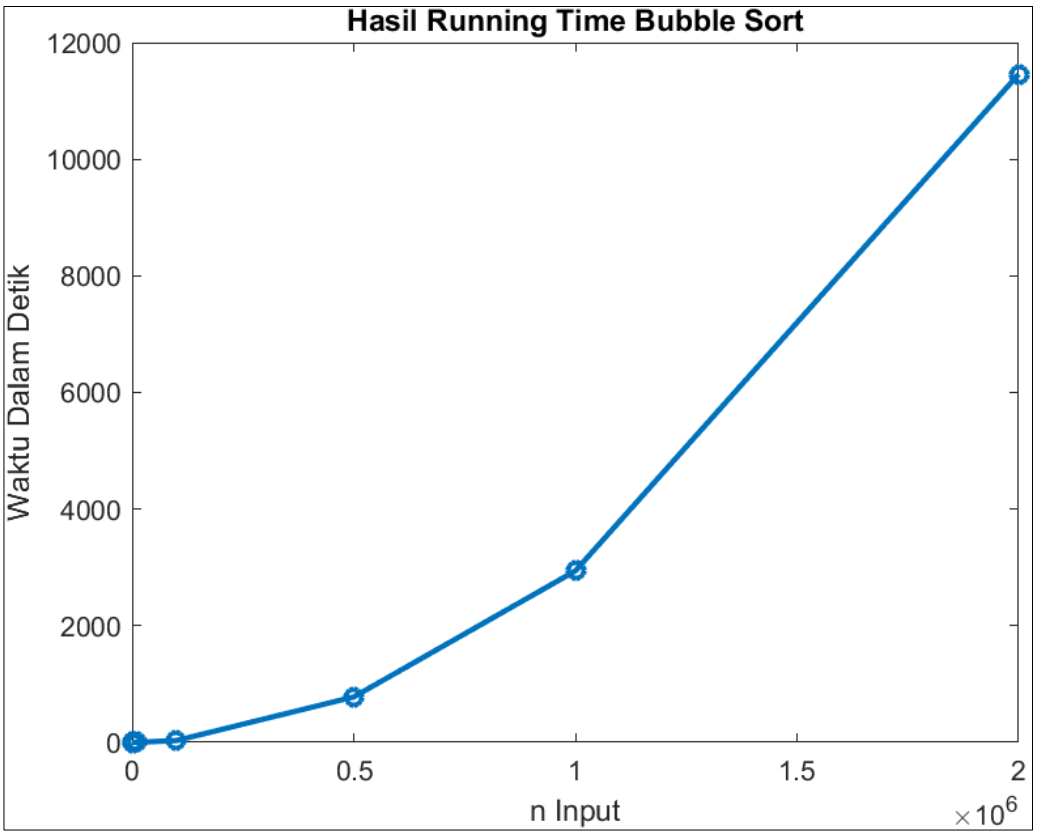
5. Plottingan Waktu

Dengan menggunakan Matlab, data hasil Running Time semua algoritma disajikan dalam bentuk grafik sebagai berikut:

a. Hasil running time semua algoritma

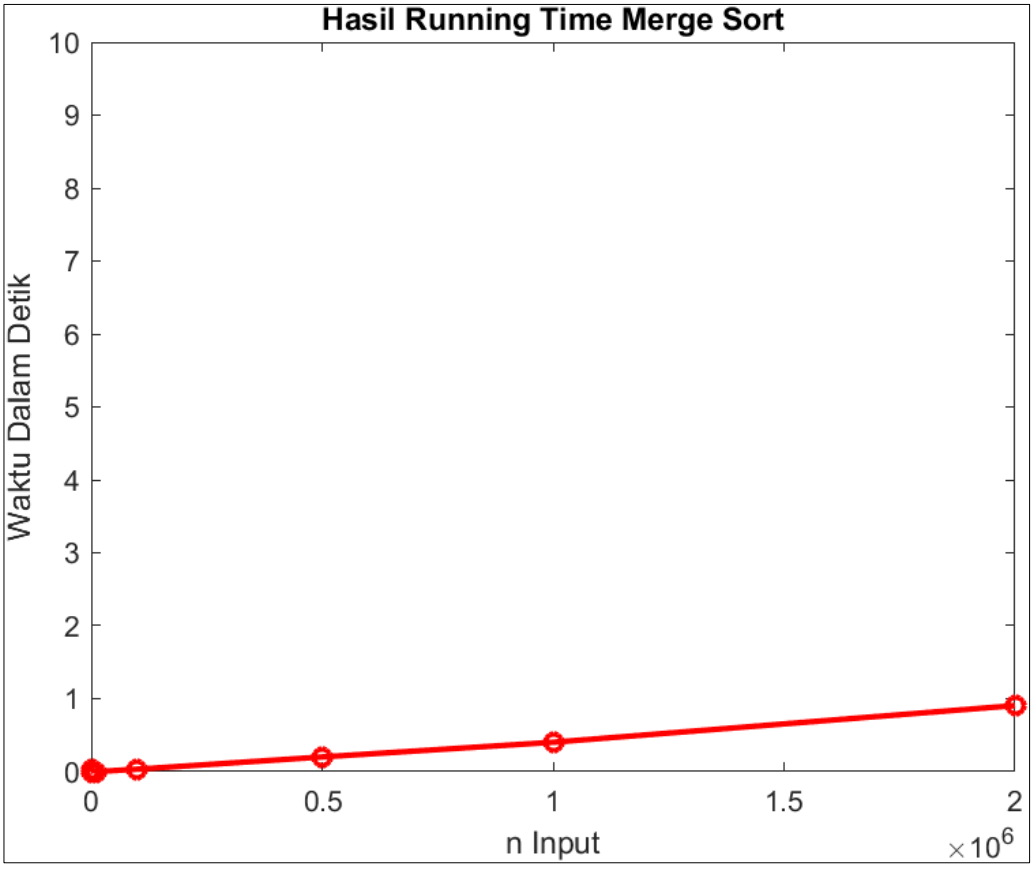


b. Hasil running time dengan Bubble Sort

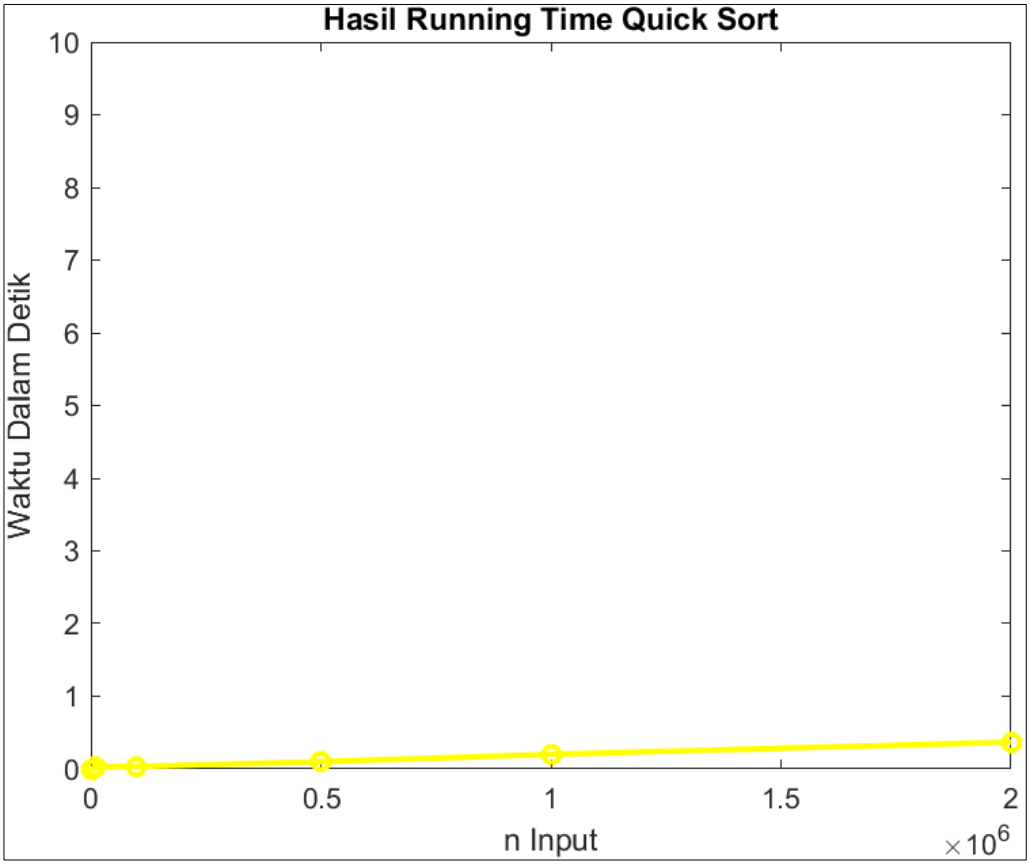


c. Hasil running time dengan Insertion Sort

d. Hasil running time dengan Merge Sort



e. Hasil running time dengan Quick Sort



f. Hasil running time dengan Heap Sort