Tugas Kecil Pengurutan Data dengan Algoritma *Divide and Conquer*

Dery Rahman A, 13515097, K-01

Senin, 27 Februari 2017

1 Deskripsi Permasalahan

Tugas kecil ini adalah mengimplementasikan 4 algoritma sorting *Divide and Conquer* (yaitu *MergeSort*, *InsertionSort*, *SelectionSort*, dan *QuickSort*), lalu mengujinya dengan data yang besar. Input berupa koleksi data yang di*generate* secara random mulai dari *data set* berukuran 1000, 5.000, 10.000, 50.000, 100.000, 500.000, hingga 1.000.000. Koleksi data random didapatkan dengan menggunakan fungsi atau kelas Random.

Untuk mengukur kecepatan algoritma, gunakan method untuk mendapatkan waktu (*current Time*) sebelum dan sesudah algoritma pengurutan dieksekusi, lalu hitung selisih antara kedua waktu. Proses *generate* koleksi data secara random tidak perlu diukur kecepatannya. Sebelum mengukur kecepatan eksekusi algoritma, lakukan dulu pengecekan kebenaran hasil pengurutan yang diberikan oleh setiap algoritma.

Lakukanlah perbandingan kecepatan eksekusi untuk keempat algoritma tersebut dengan menampilkannya dalam bentuk tabel dan chart. Berikanlah analisis sesuai dengan konsep yang dipelajari di kelas.

2 Algoritma *Divide and Conquer*

Divide and Conquer merupakan metode pemecahan masalah dengan cara membagi *problem* menjadi *subproblem* yang lebih kecil. Kemudian *subproblem* tersebut diselesaikan secara independen dan selanjutnya menggabungkan solusi dari masing-masing *subproblem* tersebut menjadi solusi *problem* semula. Secara umum, algoritma *Divide and Conquer* dibagi menjadi 3 proses :

1. *Divide* : Membagi *problem* menjadi beberapa *subproblem* yang memiliki ukuran yang lebih kecil dari *problem* semula, untuk kemudian dapat diselesaikan secara independen.
2. *Conquer* : Memecahkan masing-masing *subproblem* tersebut (secara rekursif).
3. *Combine* : Menggabungkan solusi dari *subproblem* tersebut menjadi solusi *problem* semula.

Dalam metode pengurutan menggunakan *Divide and Conquer*, dapat dilakukan dengan 2 pendekatan :

1. Mudah membagi, sulit menggunakan : urut-gabung *Merge sort* dan urut-sisip *Insertion sort*
2. Sulit membagi, mudah menggunakan : urut-cepat *Quick sort* dan urut-seleksi *Selection sort*

3 *Source* Program

```cpp
#include <bits/stdc++.h>
using namespace std;

void Merge(int* A, int left, int mid, int right){
  int ATemp[right-left+1];
  int it1 = left, it2 = mid+1, it = 0;
  while(it1 <= mid || it2 <= right){
    if (it1 > mid){ // salin sisa array kanan
      while (it2 <= right) ATemp[it++] = A[it2++];
      break;
    }
    if (it2 > right){ // salin sisa array kiri
      while (it1 <= mid) ATemp[it++] = A[it1++];
      break;
    }
    if((A[it1] < A[it2]))
      ATemp[it++] = A[it1++];
    else
      ATemp[it++] = A[it2++];
  }

  // salin kembali ke A
  it = 0;
  for(int i = left; i <= right; i++, it++)
    A[i] = ATemp[it];
}
void MergeSort(int* A, int i, int j){
  if(i<j) {
    int mid = (i + j)/2;
    MergeSort(A, i, mid);
    MergeSort(A, mid + 1, j);
    Merge(A, i, mid, j);
  }
}

int main() {
  int N;
  cin >> N;
  int A[N];
  for(int i = 0; i < N; i++){
    cin >> A[i];
  }
  clock_t tStart = clock();
  MergeSort(A, 0,N-1);
  double time_taken = (double)(clock() - tStart)/CLOCKS_PER_SEC;
  for(int i = 0; i < N; i++){
    cout << A[i] << endl;
  }
  printf("\nTime taken: %.6fs\n", time_taken);
  return 0;
}
```

Listing 1: Implementasi *Merge Sort*

```cpp
#include <bits/stdc++.h>
using namespace std;

void InsertionSort(int* A, int i, int j){
  while(i<j){
    int k = i+1;
    while(A[k] < A[k-1] && (k > 0)){
      swap(A[k], A[k-1]);
      k--;
    }
    i++;
  }
}

int main() {
  int N;
  cin >> N;
  int A[N];
```

```
  for(int i = 0; i < N; i++){
    cin >> A[i];
  }
  clock_t tStart = clock();
  InsertionSort(A, 0,N-1);
  double time_taken = (double)(clock() - tStart)/CLOCKS_PER_SEC;
  for(int i = 0; i < N; i++){
    cout << A[i] << endl;
  }
  printf("\nTime taken: %.6fs\n", time_taken);
  return 0;
}
```

Listing 2: Implementasi *Insertion Sort*

```
#include <bits/stdc++.h>
using namespace std;

void Partisi(int* A, int i, int j, int* k){
  int mid = (i + j)/2;
  int P = A[mid];
  int p = i, q = j;
  do {
    while(A[p] < P) p++;
    while(A[q] > P) q--;
    if(p < q){
      if(p == mid) mid = q;
      else if (q == mid) mid = p;
      swap(A[p], A[q]);
      if(p != mid) p++;
      if(q != mid) q--;
    }
  } while (p < q);
  *k = q;
}

void QuickSort(int* A, int i, int j) {
  if(i < j){
    int k;
    Partisi(A, i, j, &k);
    QuickSort(A, i, k-1);
    QuickSort(A, k+1, j);
  }
}

int main(){
  int N;
  cin >> N;
  int A[N];
  for(int i = 0; i < N; i++){
    cin >> A[i];
  }
  clock_t tStart = clock();
  QuickSort(A, 0,N-1);
  double time_taken = (double)(clock() - tStart)/CLOCKS_PER_SEC;
  for(int i = 0; i < N; i++){
    cout << A[i] << endl;
  }
  printf("\nTime taken: %.6fs\n", time_taken);
  return 0;
}
```

Listing 3: Implementasi *Quick Sort*

```
#include <bits/stdc++.h>
using namespace std;

void SelectionSort(int* A, int i, int j) {
  while(i<j){
    int k = i;
    int idxmin = k;
    while(k <= j){
      if(A[k] < A[idxmin]) idxmin = k;
      k++;
```

```
    }
    swap(A[i], A[idxmin]);
    i++;
  }
}

int main(){
  int N;
  cin >> N;
  int A[N];
  for(int i = 0; i < N; i++){
    cin >> A[i];
  }
  clock_t tStart = clock();
  SelectionSort(A, 0,N-1);
  double time_taken = (double)(clock() - tStart)/CLOCKS_PER_SEC;
  for(int i = 0; i < N; i++){
    cout << A[i] << endl;
  }
  printf("\nTime taken: %.6fs\n", time_taken);
  return 0;
}
```

Listing 4: Implementasi *Selection Sort*

```
#include <bits/stdc++.h>
using namespace std;

int main(){
  int N;
  int random;
  cin >> N;
  cout << N << endl;
  srand(time(NULL));
  for(int i = 0; i < N; i++){
    cout << rand() % (N*1000) << endl;
  }
  return 0;
}
```

Listing 5: Implementasi generate program random

```
#!/bin/bash
g++ -o RandomGenerator RandomGenerator.cpp

./RandomGenerator < 1000.in > rand_1000.in
./RandomGenerator < 5000.in > rand_5000.in
./RandomGenerator < 10000.in > rand_10000.in
./RandomGenerator < 50000.in > rand_50000.in
./RandomGenerator < 100000.in > rand_100000.in
./RandomGenerator < 500000.in > rand_500000.in
./RandomGenerator < 1000000.in > rand_1000000.in
```

Listing 6: Implementasi file *Bash* generate program random

```
#!/bin/bash
g++ -o MergeSort MergeSort.cpp
g++ -o InsertionSort InsertionSort.cpp
g++ -o QuickSort QuickSort.cpp
g++ -o SelectionSort SelectionSort.cpp

# 1000
./MergeSort < rand_1000.in > sort_1000_MergeSort.out
./InsertionSort < rand_1000.in > sort_1000_nsertionSort.out
./QuickSort < rand_1000.in > sort_1000_QuicSort.out
./SelectionSort < rand_1000.in > sort_1000_SelectionSort.out

# 5000
./MergeSort < rand_5000.in > sort_5000_MergeSort.out
./InsertionSort < rand_5000.in > sort_5000_InsertionSort.out
./QuickSort < rand_5000.in > sort_5000_QuickSort.out
./SelectionSort < rand_5000.in > sort_5000_SelectionSort.out

```

```
# 10000
./MergeSort < rand_10000.in > sort_10000_MergeSort.out
./InsertionSort < rand_10000.in > sort_10000_InsertionSort.out
./QuickSort < rand_10000.in > sort_10000_QuickSort.out
./SelectionSort < rand_10000.in > sort_10000_SelectionSort.out

# 50000
./MergeSort < rand_50000.in > sort_50000_MergeSort.out
./InsertionSort < rand_50000.in > sort_50000_InsertionSort.out
./QuickSort < rand_50000.in > sort_50000_QuickSort.out
./SelectionSort < rand_50000.in > sort_50000_SelectionSort.out

# 100000
./MergeSort < rand_100000.in > sort_100000_MergeSort.out
./InsertionSort < rand_100000.in > sort_100000_InsertionSort.out
./QuickSort < rand_100000.in > sort_100000_QuickSort.out
./SelectionSort < rand_100000.in > sort_100000_SelectionSort.out

# 500000
./MergeSort < rand_500000.in > sort_500000_MergeSort.out
./InsertionSort < rand_500000.in > sort_500000_InsertionSort.out
./QuickSort < rand_500000.in > sort_500000_QuickSort.out
./SelectionSort < rand_500000.in > sort_500000_SelectionSort.out

# 1000000
./MergeSort < rand_1000000.in > sort_1000000_MergeSort.out
./InsertionSort < rand_1000000.in > sort_1000000_InsertionSort.out
./QuickSort < rand_1000000.in > sort_1000000_QuickSort.out
./SelectionSort < rand_1000000.in > sort_1000000_SelectionSort.out
```

Listing 7: Implementasi file *Bash* generate seluruh algoritma

```
#include <bits/stdc++.h>
using namespace std;

int main(){
  int x, prev;

  bool urut = true;
  scanf("%d", &prev);
  while(scanf("%d", &x) && urut){
    if(prev > x) urut=false;
  }

  if(urut) cout << "TERURUT" << endl;
  else cout << "BELUM TERURUT" << endl;
  return 0;
}
```

Listing 8: Program cek kebenaran terurut

```
#!/bin/bash
g++ -o CekSort CekSort.cpp

# cek urut data acak
./CekSort < sort_1000_MergeSort.out
./CekSort < sort_5000_MergeSort.out
./CekSort < sort_10000_MergeSort.out
./CekSort < sort_50000_MergeSort.out
./CekSort < sort_100000_MergeSort.out
./CekSort < sort_500000_MergeSort.out
./CekSort < sort_1000000_MergeSort.out

./CekSort < sort_1000_InsertionSort.out
./CekSort < sort_5000_InsertionSort.out
./CekSort < sort_10000_InsertionSort.out
./CekSort < sort_50000_InsertionSort.out
./CekSort < sort_100000_InsertionSort.out
./CekSort < sort_500000_InsertionSort.out
./CekSort < sort_1000000_InsertionSort.out

./CekSort < sort_1000_QuickSort.out
./CekSort < sort_5000_QuickSort.out
./CekSort < sort_10000_QuickSort.out
```

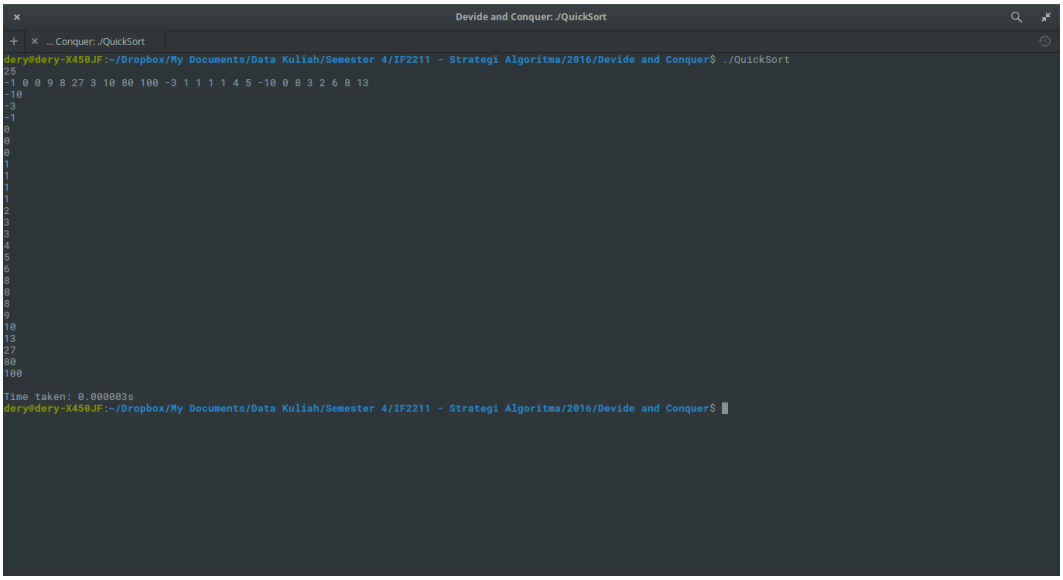
```
./CekSort < sort_50000_QuickSort.out
./CekSort < sort_100000_QuickSort.out
./CekSort < sort_500000_QuickSort.out
./CekSort < sort_1000000_QuickSort.out

./CekSort < sort_1000_SelectionSort.out
./CekSort < sort_5000_SelectionSort.out
./CekSort < sort_10000_SelectionSort.out
./CekSort < sort_50000_SelectionSort.out
./CekSort < sort_100000_SelectionSort.out
./CekSort < sort_500000_SelectionSort.out
./CekSort < sort_1000000_SelectionSort.out
```

Listing 9: Implementasi file *Bash* cek terurut

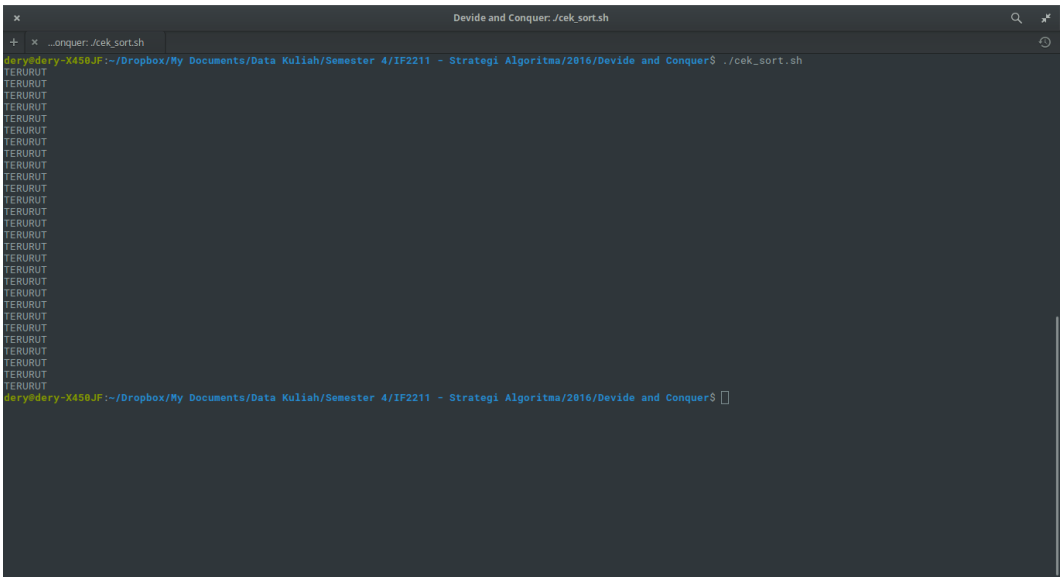
4 Contoh *Input* dan *Output*

Input	*Output*
10 4 7 0 0 2 3 0 5 6 0	0 0 0 0 2 3 4 5 6 7
15 10 2 90 1 1 -1 0 2 0 9 10 9 2 -10 -1	-10 -1 -1 0 0 1 1 2 2 2 9 9 10 10 90
10 90 87 65 43 33 22 12 8 6 1	1 6 8 12 22 33 43 65 87 90
15 90 87 65 43 33 22 12 8 6 1 0 -1 -5 -8 -10	-10 -8 -5 -1 0 1 6 8 12 22 33 43 65 87 90

Tabel 1: Contoh *input* dan *output*



Gambar 1: Contoh *input* dan *output*



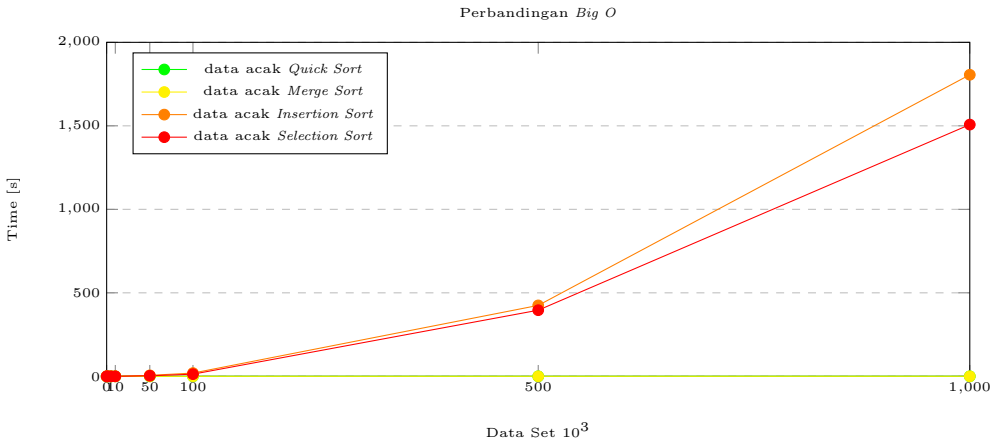
Gambar 2: Cek keterurutan

5 Hasil Perbandingan dan Analisis

Data set yang digunakan merupakan sekumpulan angka random berukuran 1000, 5000, 10000, 50000, 100000, 500000, dan 1000000. Uji yang dilakukan menggunakan sekumpulan angka acak, terurut menurun, dan hampir terurut. Keempat algoritma pengurutan akan dieksekusi dan kemudian dicatat waktu untuk melakukan pengurutan tersebut. Berikut tabel perbandingan waktu pengurutan dari keempat algoritma tersebut :

Data set	Uji	*Merge Sort*	*Insertion Sort*	*Quick Sort*	*Selection Sort*
1000	Acak	0.000137s	0.001699s	0.000101s	0.001579s
	Menurun	0.000891s	0.003074s	0.000027s	0.001263s
	Hampir Terurut	0.000073s	0.000019s	0.000050s	0.001318s
5000	Acak	0.000876s	0.040218s	0.000618s	0.033150s
	Menurun	0.000437s	0.070670s	0.000157s	0.031154s
	Hampir Terurut	0.000429s	0.000422s	0.000301s	0.032235s
10000	Acak	0.001672s	0.156521s	0.001300s	0.129070s
	Menurun	0.000891s	0.361655s	0.000352s	0.124023s
	Hampir Terurut	0.000911s	0.001791s	0.000685s	0.127724s
50000	Acak	0.009722s	5.916973s	0.007432s	3.195484s
	Menurun	0.005047s	9.434425s	0.002699s	3.012494s
	Hampir Terurut	0.004995s	0.039329s	0.003436s	3.339433s
100000	Acak	0.025999s	19.539330s	0.015889s	12.901558s
	Menurun	0.010179s	37.485172s	0.004609s	12.384136s
	Hampir Terurut	0.018262s	0.171896s	0.008987s	15.756708s
500000	Acak	0.107728s	424.713904s	0.083176s	395.992637s
	Menurun	0.055096s	925.932211s	0.024505s	403.825625s
	Hampir Terurut	0.058310s	4.967224s	0.042333s	394.206782s
1000000	Acak	0.213972s	1805.759814s	0.201507s	1507.416152s
	Menurun	0.120716s	3761.200232s	0.051493s	1514.037618s
	Hampir Terurut	0.117752s	20.672915s	0.097526s	1481.563584s

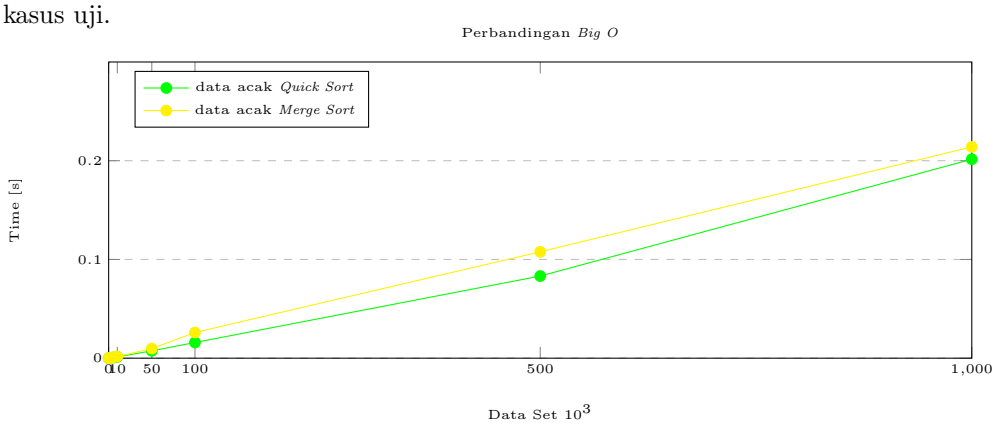
Tabel 2: Hasil eksekusi keempat algoritma



Gambar 3: Kompleksitas waktu *Merge Sort*, *Insertion Sort*, *Quick Sort*, dan *Selection Sort*

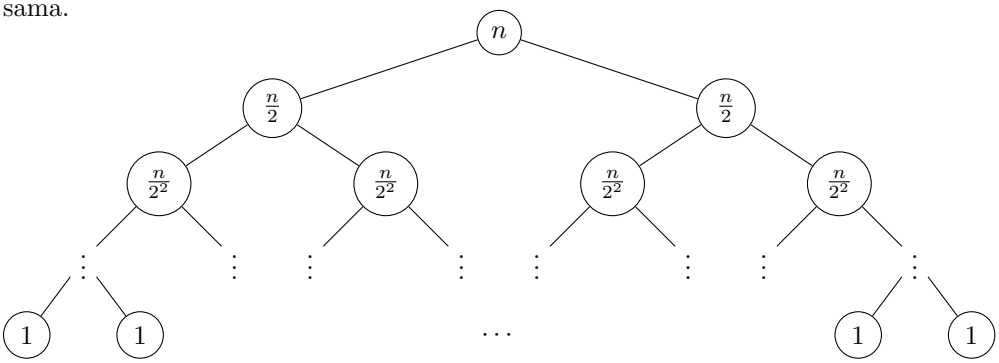
Berdasarkan percobaan diatas, algoritma *Quick Sort* merupakan algoritma tercepat untuk semua kasus uji. *Quick Sort* memiliki kompleksitas algoritma O(n $\log_2 n$). Kasus terbaik pada *Quick Sort* terjadi apabila *pivot* yang dipilih merupakan pivot median, sehingga tabel dapat terbagi menjadi 2 sama rata. Kompleksitas waktu untuk kasus terbaik pada algoritma ini sama dengan kompleksitas *Merge Sort* yaitu O(n $\log_2 n$). Sedangkan kasus terburuk terjadi apabila *pivot* yang dipilih selalu elemen maksimum/minimum, sehingga pohon yang dihasilkan merupakan *skew tree* yang mempunyai kompleksitas bernilai O(n^2). Kasus rata-rata bisa terjadi jika *pivot* yang dipilih adalah acak. Algoritma *Quick Sort* pada kasus rata-rata memiliki kompleksitas O(n $\log_2 n$).

Algoritma tercepat berikutnya adalah *Merge Sort*. Walau selisih waktu kasus uji dengan *Quick Sort* tidak begitu besar, *Merge Sort* kurang diminati karena membutuhkan tabel temporer sebagai tempat penampung sementara ketika melakukan *combine*. Kompleksitas waktu asimptotik sama seperti kompleksitas pada *Quick Sort* yaitu O(n $\log_2 n$) untuk semua kasus uji.



Gambar 4: Kompleksitas waktu *Merge Sort* dan *Quick Sort*

Untuk kasus terbaik *Quick Sort* (*pivot* yang diambil merupakan elemen median), pohon yang terbentuk merupakan pohon yang seimbang (*balance tree*). Pembagian tabel dengan mengambil elemen median sebagai *pivot* dapat membentuk upatabel yang berukuran relatif sama.



Gambar 5: *Balance Tree* pada *Quick Sort*

Kompleksitas waktu pengurutan dihitung dari jumlah perbandingan elemen-elemen tabel yaitu

$$T_{min}(n) = \text{waktu partisi} + \text{waktu pemanggilan} \textit{Quick Sort}$$

Kompleksitas prosedur partisi adalah $t(n) = cn = O(n)$ Sehingga kompleksitas waktu *Quick Sort* secara rekurens adalah

$$T(n) = \begin{cases} a & , n = 1 \\ T(n/2) + cn & , n > 1 \end{cases}$$

Dengan a dan c sebagai konstanta. Penyelesaian persamaan rekurens adalah

$T(n) = 2T(n/2) + cn$
$T(n) = 2(2T(n/4) + cn/2) + cn$
$T(n) = ...$
$T(n) = 2^k T(n/2^k) + kcn$

Persamaan terakhir dapat diselesaikan karena ukuran basis dari rekursif adalah 1,

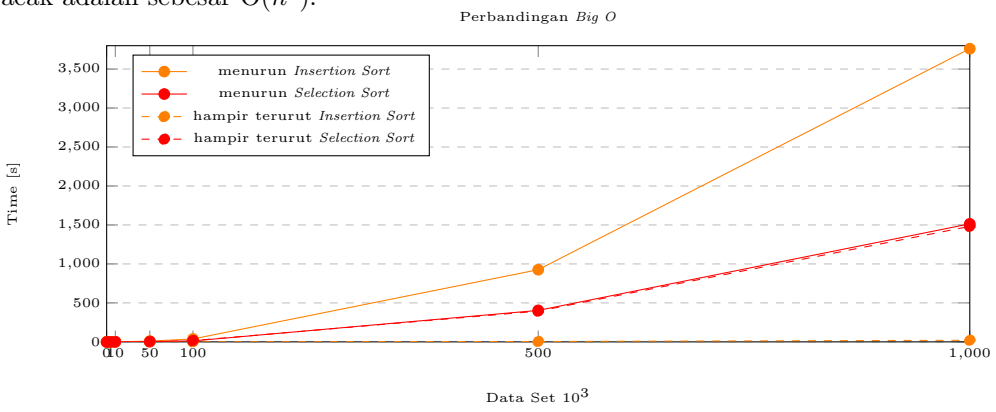
$n/a^k = 1 \rightarrow k = \log_2 n$

sehingga,

$T(n) = nT(1) + cn \log_2 n$
$T(n) = na + cn \log_2 n$
$T(n) = O(n \log_2 n)$

Persamaan rekurens tersebut menghasilkan kompleksitas $O(n \log_2 n)$. Kompleksitas ini jauh lebih baik dibanding dengan kompleksitas algoritma pada *Insertion Sort* dan *Selection Sort* yang bernilai $O(n^2)$.

Algoritma *Insertion Sort* dan *Selection Sort* yang digunakan pada percobaan kali ini menggunakan metode iteratif, hal ini dikarenakan kapasitas memori yang terbatas pada komputer, sehingga untuk melakukan pengurutan data sebesar satu juta, membutuhkan ruang *stack* yang cukup besar. Kompleksitas waktu asimptotik kedua algoritma pada kasus acak adalah sebesar O(n^2).



Gambar 6: Kompleksitas waktu *Insertion Sort* dan *Selection Sort*

Pada uji coba hampir terurut, algoritma *Insertion Sort* memiliki kompleksitas waktu yang lebih baik dibanding dengan *Selection Sort*, hal ini dikarenakan pada bagian *conquer Insertion Sort* yaitu saat suatu elemen disisipkan, iterasi akan langsung berhenti. Elemen yang terurut akan berhenti apabila posisi dari elemen tersebut sudah sesuai sehingga tidak perlu dilakukan penyisipan lebih lanjut. Elemen 1 2 3 4 5 6, contohnya, ketika elemen 4 diiterasi mundur (untuk dicari lokasi yang cocok pada pengurutan), elemen 4 dibandingkan dengan 3, kemudian langsung berhenti dikarenakan $4 > 3$. Begitu juga elemen-elemen selanjutnya, sehingga tidak perlu dilakukan pengecekan hingga indeks paling awal. Berbeda dengan *Selection Sort*, metode pengurutan ini mengecek seluruh indeks dari indeks iterasi i ke indeks paling akhir untuk menyeleksi elemen terkecil. Sehingga kompleksitas *Insertion Sort* hampir setara dengan O(n) sedangkan *Selection Sort* tetap O(n^2). Pada uji coba kasus terurut menurun, *Selection Sort* lebih unggul dibanding *Insertion Sort*. Hal ini dikarenakan *Insertion Sort* melakukan penyisipan disetiap iterasi mundurnya hingga indeks paling awal. Sedangkan *Selection Sort* hanya akan melakukan penukaran elemen, jika ditemukan elemen paling minimum disetiap iterasinya.

6 *Checklist* Pengerjaan Tugas

Tabel 3: *Checklist* Pengerjaan Tugas

Poin	Ya	Tidak
Program berhasil dikompilasi	✓	
Program berhasil running	✓	
Program dapat membaca koleksi data random dan menuliskan koleksi data terurut.	✓	
Laporan berisi hasil perbandingan kecepatan eksekusi dan analisisnya.	✓	