Bab 6
Algoritma *Sorting*

6.1 Tujuan

Sorting adalah proses menyusun elemen – elemen dengan tata urut tertentu dan proses tersebut terimplementasi dalam bermacam aplikasi. Kita ambil contoh pada aplikasi perbankan. Aplikasi tersebut mampu menampilkan daftar *account* yang aktif. Hampir seluruh pengguna pada sistem akan memilih tampilan daftar berurutan secara *ascending* demi kenyamanan dalam penelusuran data.

Beberapa macam algoritma *sorting* telah dibuat karena proses tersebut sangat mendasar dan sering digunakan. Oleh karena itu, pemahaman atas algoritma – algoritma yang ada sangatlah berguna.

Setelah menyelesaikan pembahasan pada bagian ini, anda diharapkan mampu :

1. Memahami dan menjelaskan algoritma dari *insertion sort, selection sort, merge sort* dan *quick sort*.
2. Membuat implementasi pribadi menggunakan algoritma yang ada

6.2 *Insertion Sort*

Salah satu algoritma *sorting* yang paling sederhana adalah *insertion sort*. Ide dari algoritma ini dapat dianalogikan seperti mengurutkan kartu. Penjelasan berikut ini menerangkan bagaimana algoritma *insertion sort* bekerja dalam pengurutan kartu. Anggaplah anda ingin mengurutkan satu set kartu dari kartu yang bernilai paling kecil hingga yang paling besar. Seluruh kartu diletakkan pada meja, sebutlah meja ini sebagai meja pertama, disusun dari kiri ke kanan dan atas ke bawah. Kemudian kita mempunyai meja yang lain, meja kedua, dimana kartu yang diurutkan akan diletakkan. Ambil kartu pertama yang terletak pada pojok kiri atas meja pertama dan letakkan pada meja kedua. Ambil kartu kedua dari meja pertama, bandingkan dengan kartu yang berada pada meja kedua, kemudian letakkan pada urutan yang sesuai setelah perbandingan. Proses tersebut akan berlangsung hingga seluruh kartu pada meja pertama telah diletakkan berurutan pada meja kedua.

Algoritma *insertion sort* pada dasarnya memilah data yang akan diurutkan menjadi dua bagian, yang belum diurutkan (meja pertama) dan yang sudah diurutkan (meja kedua). Elemen pertama diambil dari bagian array yang belum diurutkan dan kemudian diletakkan sesuai posisinya pada bagian lain dari array yang telah diurutkan. Langkah ini dilakukan secara berulang hingga tidak ada lagi elemen yang tersisa pada bagian array yang belum diurutkan.

6.2.1 Algoritma

```
void insertionSort(Object array[], int startIdx, int endIdx) {
     for (int i = startIdx; i < endIdx; i++) {
       int k = i;
       for (int j = i + 1; j < endIdx; j++) {
              if (((Comparable) array[k]).compareTo(array[j])>0) {
                     k = j;
              }
       }
       swap(array[i],array[k]);
     }
}
```

6.2.2 Sebuah Contoh

Data		***1st Pass***		***2nd Pass***		***3rd Pass***		***4th Pass***
Mango		Mango		Apple		Apple		Apple
Apple		Apple		Mango		Mango		Banana
Peach		Peach		Peach		Orange		Mango
Orange		Orange		Orange		Peach		Orange
Banana		Banana		Banana		Banana		Peach

Gambar 1.1.2: Contoh insertion sort

Pada akhir modul ini, anda akan diminta untuk membuat implementasi bermacam algoritma *sorting* yang akan dibahas pada bagian ini.

6.3 *Selection Sort*

Jika anda diminta untuk membuat algoritma *sorting* tersendiri, anda mungkin akan menemukan sebuah algoritma yang mirip dengan *selection sort.* Layaknya *insertion sort*, algoritma ini sangat rapat dan mudah untuk diimplementasikan.

Mari kita kembali menelusuri bagaimana algoritma ini berfungsi terhadap satu paket kartu. Asumsikan bahwa kartu tersebut akan diurutkan secara *ascending*. Pada awalnya, kartu tersebut akan disusun secara linier pada sebuah meja dari kiri ke kanan, dan dari atas ke bawah. Pilih nilai kartu yang paling rendah, kemudian tukarkan posisi kartu ini dengan kartu yang terletak pada pojok kiri atas meja. Lalu cari kartu dengan nilai paling rendah diantara sisa kartu yang tersedia. Tukarkan kartu yang baru saja terpilih dengan kartu pada posisi kedua. Ulangi langkah – langkah tersebut hingga posisi kedua sebelum posisi terakhir dibandingkan dan dapat digeser dengan kartu yang bernilai lebih rendah.

Ide utama dari algoritma *selection sort* adalah memilih elemen dengan nilai paling rendah dan menukar elemen yang terpilih dengan elemen ke-*i*. Nilai dari *i* dimulai dari 1 ke *n*, dimana *n* adalah jumlah total elemen dikurangi 1.

6.3.1 Algoritma

```
void selectionSort(Object array[], int startIdx, int endIdx) {
       int min;
       for (int i = startIdx; i < endIdx; i++) {
              min = i;
              for (int j = i + 1; j < endIdx; j++) {
                     if (((Comparable)array[min]).compareTo(array[j])>0) {
                            min = j;
                     }
              }
       swap(array[min], array[i]);
       }
}
```

6.3.2 Sebuah Contoh

Data		***1st Pass***		***2nd Pass***		***3rd Pass***		***4th Pass***
Maricar		Hannah		Hannah		Hannah		Hannah
Vanessa		*Vanessa*		Margaux		Margaux		Margaux
Margaux		*Margaux*		*Vanessa*		Maricar		Maricar
Hannah		Maricar		*Maricar*		*Vanessa*		Rowena
Rowena		Rowena		Rowena		*Rowena*		Vanessa

Figure 1.2.2: Contoh selection sort

6.4 *Merge Sort*

Sebelum mendalami algoritma *merge sort*, mari kita mengetahui garis besar dari konsep *divide and conquer* karena *merge sort* mengadaptasi pola tersebut.

6.4.1 Pola Divide and Conquer

Beberapa algoritma mengimplementasikan konsep rekursi untuk menyelesaikan permasalahan. Permasalahan utama kemudian dipecah menjadi sub-masalah, kemudian solusi dari sub-masalah akan membimbing menuju solusi permasalahan utama.

Pada setiap tingkatan rekursi, pola tersebut terdiri atas 3 langkah.

1. Divide
   Memilah masalah menjadi sub masalah
2. Conquer
   Selesaikan sub masalah tersebut secara rekursif. Jika sub-masalah tersebut cukup ringkas dan sederhana, pendekatan penyelesaian secara langsung akan lebih efektif
3. Kombinasi
   Mengkombinasikan solusi dari sub-masalah, yang akan membimbing menuju penyelesaian atas permasalahan utama

6.4.2 Memahami Merge Sort

Seperti yang telah dijelaskan sebelumnya, *Merge sort* menggunakan pola *divide and conquer*. Dengan hal ini deskripsi dari algoritma dirumuskan dalam 3 langkah berpola *divide-and-conquer*. Berikut menjelaskan langkah kerja dari *Merge sort.*

1. Divide
   Memilah elemen – elemen dari rangkaian data menjadi dua bagian.
2. Conquer
   Conquer setiap bagian dengan memanggil prosedur *merge sort* secara rekursif
3. Kombinasi
   Mengkombinasikan dua bagian tersebut secara rekursif untuk mendapatkan rangkaian data berurutan

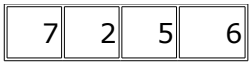
Proses rekursi berhenti jika mencapai elemen dasar. Hal ini terjadi bilamana bagian yang akan diurutkan menyisakan tepat satu elemen. Sisa pengurutan satu elemen tersebut menandakan bahwa bagian tersebut telah terurut sesuai rangkaian.

6.4.3 Algoritma

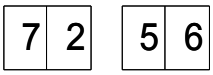
```
void mergeSort(Object array[], int startIdx, int endIdx) {
       if (array.length != 1) {
              //Membagi rangkaian data, rightArr dan leftArr
              mergeSort(leftArr, startIdx, midIdx);
              mergeSort(rightArr, midIdx+1, endIdx);
              combine(leftArr, rightArr);
       }
}
```

6.4.4 Sebuah Contoh

Rangkaian data:

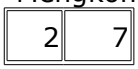


Membagi rangkaian menjadi dua bagian:
LeftArr RightArr



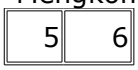
Membagi *LeftArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan

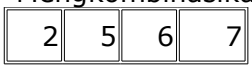


Membagi *RightArr* menjadi dua bagian:
LeftArr RightArr

Mengkombinasikan



Mengkombinasikan *LeftArr* dan *RightArr*.



Gambar 1.3.4: Contoh merge sort

6.5 *Quicksort*

Quicksort ditemukan oleh C.A.R Hoare. Seperti pada *merge sort*, algoritma ini juga berdasar pada pola divide-and-conquer. Berbeda dengan *merge sort*, algoritma ini hanya mengikuti langkah – langkah sebagai berikut :

1. Divide
   Memilah rangkaian data menjadi dua sub-rangkaian A[p…q-1] dan A[q+1…r] dimana setiap elemen A[p…q-1] adalah kurang dari atau sama dengan A[q] dan setiap elemen pada A[q+1…r] adalah lebih besar atau sama dengan elemen pada A[q]. A[q] disebut sebagai elemen pivot. Perhitungan pada elemen q merupakan salah satu bagian dari prosedur pemisahan.

2. Conquer
   Mengurutkan elemen pada sub-rangkaian secara rekursif

Pada algoritma *quicksort*, langkah ”kombinasi” tidak di lakukan karena telah terjadi pengurutan elemen – elemen pada sub-array

6.5.1 Algoritma

```
void quickSort(Object array[], int leftIdx, int rightIdx) {
       int pivotIdx;
       /* Kondisi Terminasi */
       if (rightIdx > leftIdx) {
              pivotIdx = partition(array, leftIdx, rightIdx);
              quickSort(array, leftIdx, pivotIdx-1);
              quickSort(array, pivotIdx+1, rightIdx);
       }
}
```

6.5.2 Sebuah Contoh

Rangkaian data:

3	1	4	1	5	9	2	6	5	3	5	8

Pilih sebuah elemen yang akan menjadi elemen pivot.

3	1	4	1	5	9	2	6	5	3	5	8

Inisialisasi elemen kiri sebagai elemen kedua dan elemen kanan sebagai elemen akhir.

	kiri										kanan
3	1	4	1	5	9	2	6	5	3	5	8

Geser elemen kiri kearah kanan sampai ditemukan nilai yang lebih besar dari elemen pivot tersebut. Geser elemen kanan ke arah kiri sampai ditemukan nilai dari elemen yang tidak lebih besar dari elemen tersebut.

		kiri							kanan		
3	1	4	1	5	9	2	6	5	3	5	8

Tukarkan antara elemen kiri dan kanan

		kiri								kanan	

3	1	3	1	5	9	2	6	5	4	5	8

Geserkan lagi elemen kiri dan kanan.

				kiri		kanan					
3	1	3	1	5	9	2	6	5	4	5	8

Tukarkan antar elemen kembali.

				kiri		kanan					
3	1	3	1	2	9	5	6	5	4	5	8

Geserkan kembali elemen kiri dan kanan.

				kanan	kiri						
3	1	3	1	2	9	5	6	5	4	5	8

Terlihat bahwa titik kanan dan kiri telah digeser sehingga mendapatkan nilai elemen kanan < elemen kiri. Dalam hal ini tukarkan elemen pivot dengan elemen kanan.

				pivot							
2	1	3	1	**3**	9	5	6	5	4	5	8

Gambar 1.4.2: Contoh quicksort

Kemudian urutkan elemen sub-rangkaian pada setiap sisi dari elemen pivot.

6.6 *Latihan*

6.6.1 Insertion Sort

Impelementasikan algoritma *insertion sort* dalam Java untuk mengurutkan serangkaian data integer. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui *command line*.

6.6.2 Selection Sort

Impelementasikan algoritma *selection sort* dalam Java untuk mengurutkan serangkaian data integer. Lakukan percobaan terhadap hasil implementasi anda terhadap rangkaian data integer yang dimasukkan oleh pengguna melalui *command line*.

6.6.3 Merge Sort

Gunakan implementasi *merge sort* berikut ini terhadap serangkaian data integer.

```
class MergeSort {
	static void mergeSort(int array[], int startIdx,
	int endIdx) {
	if(startIdx == _____) {
		return;
	}
	int length = endIdx-startIdx+1;
	int mid = _____;
	mergeSort(array, _____, mid);
	mergeSort(array, _____, endIdx);
	int working[] = new int[length];
	for(int i = 0; i < length; i++) {
		working[i] = array[startIdx+i];
	}
	int m1 = 0;
	int m2 = mid-startIdx+1;
	for(int i = 0; i < length; i++) {
		if(m2 <= endIdx-startIdx) {
			if(m1 <= mid-startIdx) {
				if(working[m1] > working[m2]) {
					array[i+startIdx] = working[m2++];
				} else {
					array[i+startIdx] = _____;
				}
			} else {
				array[i+startIdx] = _____;
			}
		} else {
			array[_____] = working[m1++];
		}
	}
}

	public static void main(String args[]) {
		int numArr[] = new int[args.length];
		for (int i = 0; i < args.length; i++) {
			numArr[i] = Integer.parseInt(args[i]);
		}
		mergeSort(numArr, 0, numArr.length-1);
		for (int i = 0; i < numArr.length; i++) {
			System.out.println(numArr[i]);
		}
	}
}
```

6.6.4 Quicksort

Gunakan implementasi *quicksort* berikut ini terhadap serangkaian data integer.

```
class QuickSort {
	static void quickSort (int[] array, int startIdx,
	int endIdx) {
		// startIdx adalah index bawah
		// endIdx is index atas
		// dari array yang akan diurutkan
		int i=startIdx, j=endIdx, h;
		//pilih elemen pertama sebagai pivot
		int pivot=array[_____];

		// memilah
			do {
				while (array[i]_____pivot) {
					i++;
				}
			while (array[j]>_____) {
				j--;
			}
			if (i<=j) {
				h=_____;
				array[i]=_____;
				array[j]=_____;
				i++;
				j--;
			}
		} while (i<=j);

		// rekursi
		if (startIdx<j) {
			quickSort(array, _____, j);
		}
		if (i<endIdx) {
			quickSort(array, _____, endIdx);
		}
	}

	public static void main(String args[]) {
		int numArr[] = new int[args.length];
		for (int i = 0; i < args.length; i++) {
			numArr[i] = Integer.parseInt(args[i]);
		}
		quickSort(numArr, 0, numArr.length-1);
		for (int i = 0; i < numArr.length; i++) {
			System.out.println(numArr[i]);
		}
	}
}
```