SORTING DENGAN ALGORITMA DIVIDE AND CONQUER

Ibnu Alam (13506024)

Teknik Informatika Institut Teknologi Bandung
Jalan Ganesha 10
if16024@students.if.itb.ac.id

ABSTRAK

Kemangkusan program adalah salah satu tujuan utama pembelajaran Strategi Algoritmik. Algoritma *Divide and Conquer* adalah salah satu dasar dari konsep kemangkusan program. Dalam makalah ini dibahas dua buah penggunaan algoritma *Divide and Conquer* dalam sorting, yaitu Merge Sort dan *Quick Sort.*

Kata kunci: Divide and Conquer, Merge Sort, Quick Sort.

1. PENDAHULUAN

Divide and Conquer (D&C) adalah algoritma pemrograman yang melakukan pemecahan masalah menjadi dua sub-masalah secara rekursif sampai setiap sub-masalah cukup sederhana untuk diselesaikan secara langsung. Tiap solusi dari masing-masing sub-masalah akan digabungkan untuk mendapatkan solusi dari masalah utama tersebut. Algoritma D&C menjadi basis dari algoritma-algoritma efisien untuk masalah-masalah seperti *sorting* (*quick sort*, *merge sort*) dan transformasi diskrit *Fourier*.

Dasar dari algoritma ini dikemukakan pertama kali oleh Anatolii Karatsuba Alexeevich pada tahun 1960 sebagai algoritma perkalian dua buah angka n-digit dengan kompleksitas algoritma $O(n.^2\log 3)$. Algoritma ini sekarang dikenal dengan nama Algoritma Karatsuba, membuktikan bahwa ada algoritma perkalian yang lebih cepat dari $O(n^2)$ [Kolmogorov, 1956]. Buah pikiran Kolmogorov dan penemuan Karatsuba kemudian membantu merintis penelitian performa algoritma.

Algoritma Divide and Conquer secara natural diimplementasikan secara rekursif sebagai pemanggilan prosedur dalam dirinya sendiri. Sub-masalah-sub-masalah akan dikerjakan dalam *procedure-call stack.* Setiap sub-masalah yang merupakan hasil pembagian dari masalah utama biasanya dibagi tanpa menimbulkan *overlapping* sehingga tidak ada pengerjaan redundan. Semuanya akan dimasukkan ke dalam stack dan dikerjakan mulai dari sub-masalah terkecil. Tetapi, selain pengimplementasian rekursif, dengan metode lain sub-masalah yang dibentuk dapat juga disimpan dalam struktur data yang dibuat sendiri seperti *stack*, *queue*, dan *priority queue* . Pengimplementasian non-rekursif ini menambah kebebasan dalam pemilihan sub-masalah mana yang akan dipecahkan berikutnya. Konsep ini dikembangkan dalam algoritma-algoritma seperti *Breadth First Search* dan optimalisasi *Branch and Bound*.

2. METODE

Terdapat dua metode sorting paling umum yang dipakai dalam implementasi algoritma Divide and Conquer, yaitu *quick sort* dan *merge sort* (di luar kedua ini masih ada metode lain seperti *insertion sort*). Keduanya berfungsi untuk mengurutkan sebuah *array* berisi nilai-nilai yang acak dengan cara mengurutkan sebagian dari *array* terlebih dahulu sebelum mengurutkan semua *array* secara keseluruhan.

Pembahasan mendalam akan menjelaskan lebih, tetapi pada dasarnya perbedaan antara kedua metode sorting itu sebagai berikut:

- Pembagian masalah menjadi sub-masalah di *mergesort* lebih mudah daripada *quicksort* karena *mergesort* hanya perlu membagi dua masalah sedangkan pada *quicksort* terdapat partisi yang rumit untuk membagi masalah.
- Sebaliknya, *mergesort* rumit pada saat penggabungan karena harus mengurutkan kembali gabungan dari sub-masalah, sedangkan *quicksort* tinggal menempelkan *array* yang sudah dipecah menjadi satu kembali.
- *Quick sort* melakukan proses langsung di dalam array masalah sehingga tidak memerlukan memory tambahan untuk menyimpan sub-masalah. *Merge sort* melakukan sebaliknya.

2.1 Merge Sort

Algoritma dari *merge sort* adalah:

DIVIDE : Bagi tabel di tengah menjadi 2 tabel submasalah (Tabel [i..j] dibagi menjadi Tabel [i..pivot] dan tabel [pivot+1..j] dimana pivot = (i+j) div 2)
CONQUER: Gabungkan 2 submasalah dan urutkan, lakukan sampai semua sub-masalah bergabung lagi menjadi satu.

Contoh sederhana:

Pemecahan

(17, 12, 6, 19, 23, 8, 5, 10)
(17, 12, 6, 19), (23, 8, 5, 10)
(17, 12), (6, 19), (23, 8), (5, 10)
(17), (12), (6), (19), (23), (8), (5), (10)

Penggabungan dan pengurutan
(12, 17), (6, 19), (8, 23), (5, 10)
(6, 12, 17, 19), (5, 8, 10, 23)
(5, 6, 8, 10, 12, 17, 19, 23)

Implementasi dalam Java:

```
public abstract class MergeSort extends Object {
    protected Object           toSort[];
    protected Object           swapSpace[];

    public void sort(Object array[]) {
        if(array != null && array.length > 1)
        {
            int             maxLength;

            maxLength = array.length;
            swapSpace = new Object[maxLength];
            toSort = array;
            this.mergeSort(0, maxLength - 1);
            swapSpace = null;
            toSort = null;
        }
    }

    public abstract int compareElementsAt(int beginLoc, int endLoc);

    protected void mergeSort(int begin, int end) {
        if(begin != end)
        {
            int           mid;

            mid = (begin + end) / 2;
            this.mergeSort(begin, mid);
            this.mergeSort(mid + 1, end);
            this.merge(begin, mid, end);
        }
    }

    protected void merge(int begin, int middle, int end) {
        int           firstHalf, secondHalf, count;

        firstHalf = count = begin;
        secondHalf = middle + 1;
```

```
        while((firstHalf <= middle) && (secondHalf <= end))
        {
            if(this.compareElementsAt(secondHalf, firstHalf) < 0)
                swapSpace[count++] = toSort[secondHalf++];


    else
                swapSpace[count++] = toSort[firstHalf++];
        }
        if(firstHalf <= middle)
        {
            while(firstHalf <= middle)
                swapSpace[count++] = toSort[firstHalf++];
        }
        else
        {
            while(secondHalf <= end)
                swapSpace[count++] = toSort[secondHalf++];
        }
        for(count = begin;count <= end;count++)
            toSort[count] = swapSpace[count];
    }
}
```

2.2 Quick Sort

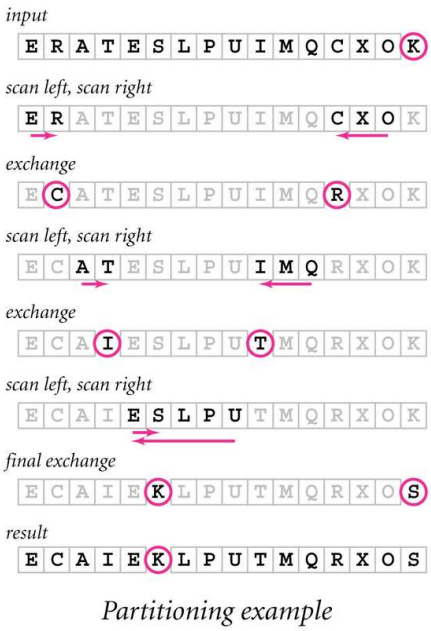
Seperti yang telah dinyatakan sebelumnya, *quick sort* mempartisi tabel sehingga (asumsi pengurutan kecil-besar dari kiri ke kanan):

- Elemen T[i] berada di tempatnya yang benar (tidak perlu dipindahkan/diurutkan lagi)
- Tidak ada elemen yang lebih besar nilainya di kiri i
- Tidak ada elemen yang lebih kecil nilainya di kanan i

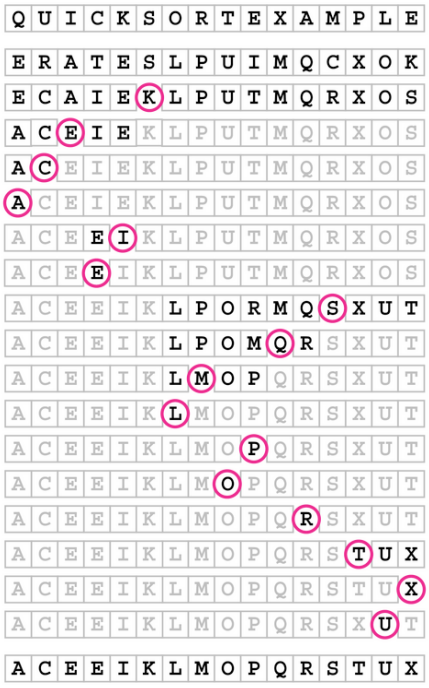
Algoritma dari *quick sort*:

Partisi tabel (secara implisit juga melakukan pengurutan tabel). Jika belum urut, partisi kembali partisi dari tabel (rekursif)

Gabungkan semua partisi.



Gambar 1 – Ilustrasi teknik partisi



Full quicksort example

Gambar 2 - contoh quick sort

Implementasi dalam Java:

```
public class QuickSort {
    private static long comparisons = 0;
    private static long exchanges   = 0;

   /***********************************************************************
    *  Quicksort code from Sedgewick 7.1, 7.2.
    ***********************************************************************/
    public static void quicksort(double[] a) {
        shuffle(a);                        // to guard against worst-case
        quicksort(a, 0, a.length - 1);
    }

    // quicksort a[left] to a[right]
    public static void quicksort(double[] a, int left, int right) {
        if (right <= left) return;
        int i = partition(a, left, right);
        quicksort(a, left, i-1);
        quicksort(a, i+1, right);
    }

    // partition a[left] to a[right], assumes left < right
    private static int partition(double[] a, int left, int right) {
        int i = left - 1;
        int j = right;
        while (true) {
            while (less(a[++i], a[right]))      // find item on left to swap
                ;                               // a[right] acts as sentinel
```

```
            while (less(a[right], a[--j]))      // find item on right to swap
                if (j == left) break;           // don't go out-of-bounds
            if (i >= j) break;                  // check if pointers cross
            exch(a, i, j);                      // swap two elements into place
        }
        exch(a, i, right);                      // swap with partition element
        return i;
    }

    // is x < y ?
    private static boolean less(double x, double y) {
        comparisons++;
        return (x < y);
    }

    // exchange a[i] and a[j]
    private static void exch(double[] a, int i, int j) {
        exchanges++;
        double swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }

    // shuffle the array a[]
    private static void shuffle(double[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int r = i + (int) (Math.random() * (N-i));   // between i and N-1
            exch(a, i, r);
        }
    }



    // test client
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);

        // generate N random real numbers between 0 and 1
        long start = System.currentTimeMillis();
        double[] a = new double[N];
        for (int i = 0; i < N; i++)
            a[i] = Math.random();
        long stop = System.currentTimeMillis();
        double elapsed = (stop - start) / 1000.0;
        System.out.println("Generating input:  " + elapsed + " seconds");

        // sort them
        start = System.currentTimeMillis();
        quicksort(a);
        stop = System.currentTimeMillis();
        elapsed = (stop - start) / 1000.0;
        System.out.println("Quicksort:   " + elapsed + " seconds");

        // print statistics
        System.out.println("Comparisons: " + comparisons);
        System.out.println("Exchanges:   " + exchanges);
    }
}
```

3. KOMPLEKSITAS ALGORITMA

Merge Sort:

$T(N) <= 0$, *if* $N = 1$

Otherwise,

$T(N) <= T(N/2) + T(N/2) + N$

Solusi:

$T(N) = O(N^2 \log N)$

Quick Sort:

$$C_N = N + 1 + \frac{1}{N}\sum_{k=1}^{N}(C_k + C_{N-k})$$

$$= N + 1 + \frac{2}{N}\sum_{k=1}^{N} C_{k-1}$$

$$N C_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

$$C_N \approx 2(N+1)\ln N \approx 1.39 N \log_2 N.$$

$$\frac{C_N}{N+1} \approx \sum_{k=1}^{N}\frac{2}{k} \approx \int_{k=1}^{N}\frac{2}{k} = 2\ln N$$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

$$= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$= \vdots$$

$$= \frac{C_2}{3} + \sum_{k=3}^{N}\frac{2}{k+1}$$

$$N C_N = (N+1)C_{N-1} + 2N$$

4. KESIMPULAN

Merge Sort dan *Quick Sort* adalah salah satu penggunaan dari algoritma Divide and Conquer. Keduanya merupakan *tool* yang kuat dalam pemrosesan pengurutan data dalam program. Secara kompleksitas *Quick Sort* lebih cepat daripada *Merge Sort*.

REFERENSI

[1] Munir, Rinaldi. "Diktat Kuliah IF 2251 Strategi Algoritmik", 2005.