Implementasi Algoritma *Radix Sort* dalam Berbagai Kasus Bilangan Dibandingkan Algoritma Pengurutan yang lain

Dean Fathony Alfatwa[1], Eriek Rahman Syah P[2], Fahris Mumtaza Ahsan[3]

Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

NIM : 135 03 003[1], 135 03 032[2], 135 03 049[3]

Abstrak

Teknologi informasi sudah berkembang sangat pesat pada masa ini. Pencarian informasi yang berjumlah besar dalam waktu yang cepat sangat dibutuhkan sebagai upaya efisiensi waktu. Pencarian sebuah dokumen akan lebih cepat apabila informasi-informasi mengenai dokumen yang dicari tersebut diurutkan terlebih dahulu, daripada pencarian pencarian dokumen tanpa pengurutan. Sehingga proses pengurutan (*sorting*) merupakan salah satu bagian penting dalam proses pencarian informasi. Beberapa algoritma pengurutan (*sorting*) yang sering digunakan yaitu *bubble sort*, *bidirectional bubble sort*, *heap sort*, *shell sort*, *tree sort*, *radix sort*, *merge sort*, *insertion sort*, *quick sort*, dan *selection sort*. Dalam makalah ini hanya akan dibahas dan dianalisis algoritma *radix sort* jika dibandingkan dengan algoritma pengurutan lain.

***Kata kunci**: pengurutan, sorting, bubble sort, bidirectional bubble sort, heap sort, shell sort, tree sort, radix sort, merge sort, insertion sort, quick sort, selection sort*

1. Pendahuluan

Teknologi informasi sudah berkembang sangat pesat pada masa ini. Pencarian informasi yang berjumlah besar dalam waktu yang cepat sangat dibutuhkan sebagai upaya efisiensi waktu. Pencarian sebuah dokumen akan lebih cepat apabila informasi-informasi mengenai dokumen yang dicari tersebut diurutkan terlebih dahulu, daripada pencarian pencarian dokumen tanpa pengurutan. Sehingga proses pengurutan (*sorting*) merupakan salah satu bagian penting dalam proses pencarian informasi.

2. Pengurutan (*Sorting*)

2.1. Pengertian Pengurutan (*Sorting*)

Sort menurut Kamus Komputer dan Istilah Teknologi Informasi adalah penyortiran, biasa digunakan juga dalam arti pengurutan. Penyortiran ini disusun berdasarkan kriteria tertentu.[1]

2.2. Klasifikasi Pengurutan (*Sorting*)

Algoritma-algoritma pengurutan (*sorting*) dapat diklasifikasi berdasarkan teknik yang digunakan dalam algoritma tersebut. Pengklasifikasiannya adalah sebagai berikut :

1. *Brute Force*
   Brute force adalah sebuah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah (*problem statement*) dan definisi konsep yang dilibatkan. Algoritma *Brute force* memecahkan masalah dengan sangat sederhana, langsung dan dengan cara yang jelas (*obvious way*). [8]
   Yang termasuk di dalam algoritma *Brute force* adalah :
   a. *Bubble sort*
      Merupakan algoritma pengurutan paling tua dengan metode pengurutan paling sederhana. Pengurutan yang dilakukan dengan membandingkan masing-masing *item* dalam suatu list secara berpasangan, menukar *item* jika diperlukan, dan mengulaginya sampai akhir list secara berurutan, sehingga tidak ada lagi *item* yang dapat ditukar.[5]
   b. *Bidirectional bubble sort*
      Merupakan variansi dari algoritma *bubble sort*, dimana *item* dalam suatu list dibandingkan secara berpasangan, menukarnya jika diperlukan, dan mempunyai alternatif melalui list secara berurutan dari awal sampai akhir kemudian dari akhir sampai awal lagi hingga tidak ada pertukaran (*swap*) yang dapat dilakukan.[6]
2. *Divide and Conquer*
   Divide and Conquer adalah metode pemecahan masalah yang bekerja dengan membagi masalah (*problem*) menjadi beberapa upa-masalah (*subproblem*) yang lebih kecil, kemudian menyelesaikan masing-masing upa-masalah secara independen dan akhirnya menggabung solusi masing-masing upa-masalah sehingga menjadi solusi masalah semula.[8]

Yang termasuk di dalam algoritma *Divide and Conquer* adalah :

a. *Merge sort*
   Sebuah algoritma pengurutan yang membagi *item* yang akan diurutkan menjadi dua bagian, dan secara rekursif mengurutkan masing-masing bagian tersebut, lalu menggabungkannya sampai berakhir.[6]
b. *Insertion sort*
   Sebuah algoritma pengurutan dengan mengambil *item* dan memasukkannya ke dalam struktur data yang secara berulang kali dalam susunan yang tepat. [6]
c. *Quick sort*
   Sebuah algoritma pengurutan yang mengambil sebuah elemen dalam larik sebagai *pivot*, mempartisi elemen sisanya menjadi elemen yang lebih besar dan elemen yang lebih kecil daripada pivot, dan secara rekursif mengurutkan hasil dari partisi tersebut. [6]
d. *Selection sort*
   Sebuah algoritma pengurutan yang secara berulang mencari *item* yang belum terurut dan mencari paling sedikit satu untuk dimasukkan ke dalam lokasi akhir. [6]
e. *Shell sort*
   Merupakan algoritma yang stau jenis dengan *insertion sort*, dimana pada setiap nilai i dalam n/i *item* diurutkan. Pada setiap pergantian nilai, i dikurangi sampai 1 sebagai nilai terakhir. [6]
f. *Radix Sort*
   Secara kompleksitas waktu, *radix sort* termasuk ke dalam *Divide and Conquer*. Namun dari segi algoritma untuk melakukan proses pengurutan, *radix sort* tidak termasuk dalam *Divide and Conquer*. *Radix sort* merupakan sebuah algoritma pengurutan yang mengatur pengurutan nilai tanpa melakukan beberapa perbandingan pada data yang dimasukkan. [6]

3. Algoritma *Radix Sort*

3.1. Pengertian [7]

Radix Sorting adalah metode sorting yang ajaib yang mana mengatur pengurutan nilai tanpa melakukan beberapa perbandingan pada data yang dimasukkan. Secara harfiah, radix dapat diartikan sebagai posisi dalam angka. Pada sistem desimal, radix adalah digit dalam angka desimal. Seperti contoh, angka “42” mempunyai 2 digit yaitu 4 dan 2. Radix Sort memperoleh namanya dari digit-digit tersebut, karena metode ini pertama kalinya mengurutkan nilai-nilai input berdasarkan radix pertamanya, lalu pengurutan dilakukan berdasarkan radix keduanya, begitu juga seterusnya.

3.2. Implementasi Radix Sort [7]

Radix Sort merupakan algoritma pengurutan yang cepat, mudah, dan sangat efektif. Namun banyak orang yang berpikir bahwa algoritma ini memiliki banyak batasan di mana untuk kasus-kasus tertentu tidak dapat dilakukan dengan algoritma ini, seperti pengurutan bilangan pecahan, bilangan negatif, adanya kompleksitas bit dan word, dan pengurutan pada *multiple keys*. Radix sort hanya bisa digunakan pada bilangan integer, untuk bilangan pecahan, bisa dilakukan dengan perantara bucket sort atau metode berbasis perbandingan yang lain.
Dalam perilakunya yang melihat digit-digit angka sebagai pengontrolnya, Radix Sort dapat dimplementasikan dalam pengurutan bilangan desimal dan bilangan bit.

1.2.1. Implementasi dalam bilangan bit [4]

0	I	II	III
523	523	523	088
153	153	235	153
088	554	153	235
554	235	554	523
235	088	088	554

Radix Sort yang sangat sederhana untuk bit yaitu seperti pseudo-code di bawah ini :

```
for i = 0 to source_n do
      dest[source[i]].append(
      source[i])
endfor
```

di mana :

`source` adalah List of bytes

`source_n` adalah jumlah bytes yang diurutkan

`dest[256]` adalah 256 list of bytes.

Lalu muncul permasalahan yang lain yaitu bagaimana bila kita memperluas proses dengan tujuan untuk mengurutkan tidak hanya bilangan bit namun juga word, dwords, atau bahkan string karakter. Sebenarnya cara ini sangat mudah yaitu kita hanya mengurutkan lagi berdasarkan radix selanjutnya, panggil kembali yang berlaku sebagai bit dalam bilangan. Radix pertama adalah LSB (Least Significant Byte) dan radix yang terakhir adalah MSB (Most Significant Byte). Di antara keduanya kita lakukan semua pass yang dibutuhkan. Pass yang kedua tidak akan merusak apa yang telah dilakukan pada tahap pertama karena metode sorting bersifat stabil.[7]

Kita dapat mengambil contoh untuk mengurutkan bilangan heksadesimal berikut :
`0xBC, 0xAB, 0xBA, 0xAC, 0xBB, 0xAA`
Pass pertama menghasilkan :

```
0xBA
0xAA
0xAB
0xBB
0xBC
0xAC
```

Dapat dilihat bahwa list diurutkan berdasarkan kolom terakhir (LSB). Pass kedua akan mengurutkan list ini berdasarkan kolom pertama (MSB). Selama pengurutan ini stabil, 3 bilangan 0xAA, 0xAB dan 0xAC yang membagi MSB yang sama akan d*item*ukan dalam output yang sama dengan input. Dan selama input ditentukan oleh pass pertama, list terurut berdasarkan LSB dan berikut adalah hasil urutannya :

```
0xAA
0xAB
0xAC
0xBA
0xBB
0xBC
```

1.2.2. Implementasi dalam bilangan desimal [4]

Misalkan di dalam list terkandung 5 bilangan yaitu 523, 153, 088, 554, dan 235
Tahap-tahap pengurutannya adalah sebagai berikut :

Kita memulai dengan mengurutkan digit terakhir dari bilangan. Digit yang dicetak merah menunjukkan digit mana yang sedang berjalan dalam proses pengurutan. Perunutan dilakukan dari digit terakhir sampai pada digit pertama, kemudian dapat dilihat hasilnya yaitu pada tahap IV bahwa list tersebut sudah terurut.
Dalam contoh yang lain yaitu kita gunakan kunci biner dari list yang berisi bilangan : 51, 41, 19, 6, 48, 25, dan 23. [2]
Tahap-tahap pengurutannya adalah sebagai berikut :

0		I	
110011	51	000110	6
101001	41	110000	48
010011	19	110011	51
000110	6	101001	41
110000	48	010011	19
011001	25	011001	25
010111	23	010111	23

II		III	
110000	48	110000	48
101001	41	101001	41
011001	25	011001	25
000110	6	110011	51
110011	51	010011	19
010011	19	000110	6
010111	23	010111	23

IV		V		VI	
110000	48	000110	6	000110	6
110011	51	101001	41	010011	19
010011	19	110000	48	010111	23
000110	6	110011	51	011001	25
010111	23	010011	19	101001	41
101001	41	010111	23	110000	48
011001	25	011001	25	110011	51

Dapat dilihat bahwa hasil terakhir merupakan list dari bilangan-bilangan yang telah terurut.

1.2.3. Implementasi dalam bilangan pecahan [7]

Seperti yang orang-orang pikir bahwa Radix Sort tidak bisa diimplementasikan untuk mengurutkan pecahan. Dalam beberapa kasus pernyataan itu dapat dibenarkan tapi di satu sisi juga bisa dibuktikan bahwa mereka salah. Solusinya yaitu dengan mendefinisikan fungsi IR(x) sebagai *integer representation* untuk setiap bilangan pecahan x. Sebagai contoh bilangan pecahan 42.0 memiliki representasi biner 0x42280000. Dalam arti lain :
IR (42.0) = 0x42280000
Radix Sort bekerja dengan pecahan positif karena untuk setiap bilangan pecahan x dan y,
x > y > 0 => IR(x) > IR(y)
Di sini x dan y diperlakukan oleh kode pengurutan sebagai IR(x) dan IR(y) dan hasilnya akan benar.

1.2.4. Implementasi dalam bilangan pecahan negatif [7]

Terdapat kekacauan bila dengan penyelesaian pengurutan bilangan pecahan positif diimplementasikan pada kasus ini, karena bilangan pecahan negatif diurutkan pada posisi terakhir, misalnya seperti :

```
2083.000000
2785.000000
8080.000000
10116.000000
10578.000000
12974.000000
-660.000000
-4906.000000
-10050.000000
```

```
-16343.000000
```

Dapat dilihat bahwa nilai negatifnya sudah terurut namun dalam tempat dan susunan yang salah (urutan dimulai dari yang terbesar). Solusinya yang pertama yaitu perlu mencari banyak nilai negatif. Tanda suatu bilangan ditentukan oleh MSB dan yang ingin kita cari adalah nomor entitas yang memiliki MSB 1. Pada proses radix, terdapat 128 entri pada counter terakhir. Yang harus dilakukan adalah menjumlahkannya :

```
NbNegativeValues = 0;
For (i=128; i<256, i++) {
     NbNegatifValues+=
     LastCounter[i];
}
```

Untuk pengurutan yang kurang dari 128 entri, algoritma *brute force* akan bekerja lebih efisien. Namun, radix sort merupakan algoritma yang bertujuan mengurutkan sejumlah informasi yang berjumlah relatif besar. Yang perlu dilakukan untuk menutupi kelemahan algoritma ini agar mendapatkan hasil pengurutan akhir yang benar adalah memperbaiki dua hal, yaitu tempat dan susunan yang belum benar.

Pertama, perbaikan jika entri bilangan positif. Misalkan M adalah sebuah bilangan entri negatif yang telah dihitung sebelumnya, harus dipastikan bahwa pecahan positif dimulai setelah pecahan yang negatif. Perbaikan pada tempat yang salah dilakukan dengan memaksa offset positif yang pertama sebagai M.

```
Offset[0] = M;
```

Karena pengurutan untuk entri bilangan positif sudah benar, maka perbaikan untuk kasus bilangan positif sudah selesai.

Selanjutnya, perbaikan jika entri adalah bilangan negatif. Ambil kembali `-16343.0` yang merupakan hasil pengurutan terakhir dari pengurutan di atas. Seharusnya entri ini berada pada urutan paling awal setelah proses pengurutan dilakukan. Perbaikan pertama dengan memaksa membersihkan offset yang relatif terhadap entri paling akhir.

```
Offset[255] = 0;
```

Dari sini akan diurutkan sisa entri negatif dalam susunan menaik, dengan cara membalik susunan pada status awal *radix sort*.

```
for(i=0;i<127;i++){
     Offset[254-i] =
     Offset[255-i] +
     LastCounter[255-i];
}
```

Kode ini mengurutkan offset pada *range* 254-0 s.d. 255-126 yaitu dari 254 s.d. 128, atau dengan kata lain offset dalam entri negatif. Mengurutkannya dalam susunan terbalik, memulai dari offset 255 (null), akan membuat pecahan negatif dalam susunan benar.

3.3 Algoritma dan Kompleksitas Waktu Radix Sort [3]

Asumsikan bahwa kunci *item* yang diurutkan mempunyai rentang **k**, $\mathbf{f_i}$| **i=0..k-1**, dan setiap $\mathbf{f_i}$ memiliki nilai diskret $\mathbf{s_i}$, lalu prosedur radix sortnya dapat dituliskan seperti berikut :

`radixsort( A, n ) {`
`  for(i=0;i<k;i++) {`
`  for(j=0;j<si;j++)`
`  bin[j] = EMPTY;`
	O(s_i)

`for(j=0;j<n;j++) {`
`  move Ai to the end of bin[Ai->fi]`
`  }`
	O(n)

`for(j=0;j<si;j++)`
`  concatenate bin[j] onto the end of A;`
`  }}`
	O(s_i)

Total $\sum_{i=1}^{k} O(s_i + n) = O(kn + \sum_{i=1}^{k} s_i$ $= O(n + \sum_{i=1}^{k} s_i$	

Sekarang misalnya kita ambil contoh kuncinya yaitu bilangan integer dalam range **(0,b^k-1)**, untuk beberapa nilai konstan **k**, lalu kunci tersebut dapat dilihat sebagai **k**-digit base-**b** integers. Jadi, $\mathbf{s_i}$ **= b** untuk semua i dan kompleksitas waktunya menjadi **O(n+kb)** atau **O(n)**. Jika **k** diperbolehkan ditambah dengan **n**, maka kita mendapatkan kasus berbeda. Contohnya yaitu dibutuhkan $\log_2$**n** digit biner untuk merepresentasikan sebuah integer < **n**. Jika panjang kunci diperbolehkan ditambah dengan **n**, maka **k** = log **n**, dan kita mendapatkan :

$$\sum_{i=1}^{k} O(s_i + n) = O(n \log n + \sum_{i=1}^{\log n} 2)$$
$$= O(n \log n + 2 \log n)$$
$$= O(n \log n)$$

4. Kesimpulan

Radix Sort merupakan teknik algoritma pengurutan yang banyak mengundang kontrovensi, sebagaimana yang diketahui banyak orang bahwa kemampuannya terbatas pada bilangan integer. Namun seiring dengan eksplorasi yang dilakukan, ternyata Radix Sort juga bisa digunakan untuk mengurutkan bilangan pecahan bahkan pecahan negatif.

Metode ini men-*scanning* digit-digit pada suatu bilangan yang dirunut dari digit terakhir hingga digit pertama, tetapi bukanlah masalah untuk mengatasi bilangan pecahan yaitu dengan merepresentasikan

bilangan tersebut ke dalam representasi biner atau heksa yang memiliki konfigurasi tertentu dengan kompleksitas waktu yang sebanding dengan metode pencarian yang telah diakui oleh banyak orang sebagai metode tercepat, yaitu Quick Sort. Atau bahkan metode Radix Sort ini memiliki kompleksitas waktu yang lebih cepat dari Quick Sort dalam kasus tertentu.

Daftar Pustaka

[1] Jack Febrian, Farida Andayani. 2002. Kamus Komputer dan Istilah Teknologi Informasi. Bandung : Informatika Bandung.

[2] 1999, Jim Loy http://www.jimloy.com/computer/radix.htm

[3] John Morris, 1998 http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/radixsort.html

[4] Nils Pipenbrinck. 1997 http://www.cubic.org/docs/radix.htm

[5] Paul E. Black http://www.nist.gov. Diakses tanggal 19 Mei 2005 pukul 10.00 WIB.

[6] Paul E. Black et al http://www.nist.gov. Diakses tanggal 19 Mei 2005 pukul 10.15 WIB.

[7] Pierre Terdiman,2000 http://codercorner.com/RadixSortRevisited.htm

[8] Ir. Rinaldi Munir, M.T. 2005. Diktat Kuliah IF2251 Strategi Algoritmik. Bandung : Lab. Ilmu dan Rekayasa Komputasi, Departemen Teknik Informatika, Institut Teknologi Bandung.