Pengkajian Algoritma Pengurutan-Tanpa-Pembandingan *Counting Sort* dan *Radix Sort*

Dominikus Damas Putranto, NIM 13506060
Program Studi Teknik Informatika ITB, Bandung 40132
email: if16060@students.if.itb.ac.id

Abstract *– Pengurutan sangat penting dalam Teknologi Informasi yang tak lepas dari pengolahan data. Pemecahan permasalahan pengolahan data dapat menjadi lebih efektif dan efisien bila data sudah dalam keadaan terurut. Seperti dalam proses pencarian data (searching), algoritma pencarian tingkat lanjut yang lebih efektif daripada cara konvensional seperti Binary Search ataupun Interpolation Search membutuhkan data yang sudah terurut. Contoh lain di mana data terurut dibutuhkan adalah dalam penggabungan data menggunakan metode merging.*

Makalah ini akan mengkaji 2 algoritma pengurutan yang dalam proses pengurutannya tidak melakukan pembandingan data. Kedua algoritma tersebut adalah algoritma Counting Sort dan algoritma Radix Sort. Pengkajian yang dilakukan adalah mengenai ide dasar, algoritma pengurutan, implementasi, dan analisis kompleksitas algoritma.

Kata Kunci*: algoritma pengurutan, sorting algorithm, pengurutan tanpa pembandingan, non-comparison sort, comparison sort, radix sort ,counting sort*

1. PENDAHULUAN

Pengurutan sangat penting dalam Teknologi Informasi yang tak lepas dari pengolahan data. Pemecahan permasalahan pengolahan data dapat menjadi lebih efektif dan efisien bila data sudah dalam keadaan terurut. Seperti dalam proses pencarian data (searching), algoritma pencarian tingkat lanjut yang lebih efektif daripada cara konvensional seperti Binary Search ataupun Interpolation Search membutuhkan data yang sudah terurut. Contoh lain di mana data terurut dibutuhkan adalah dalam penggabungan data menggunakan metode merging.

Di antara banyak algoritma pengurutan tersebut terdapat satu kategori algoritma pengurutan yang dalam prosesnya tidak melakukan pembandingan antardata. Yang sering disebut *Non-Comparison Sorting Algorithm*, atau dalam bahasa Indonesia Algoritma Pengurutan-Tanpa-Pembandingan. Dua contoh dari algoritma-algoritma tersebut adalah *Counting Sort* dan *Radix Sort* yang akan dibahas dalam makalah ini.

Dalam beberapa kasus, pengurutan tanpa pembandingan ini memiliki kompleksitas waktu yang lebih baik daripada metode pengurutan dengan pembandingan. Dan walaupun hal itu dapat terjadi dengan berbagai batasan-batasan tertentu, eksplorasi lebih lanjut yang dilakukan semakin lama semakin membuat batasan-batasan tersebut berkurang, atau dengan kata lain semakin lama, algoritma-algoritma pengurutan tanpa pembandingan ini semakin dapat digunakan secara umum (*generalized*). Perbandingan antara *Comparison Sort* dengan *Non Comparison Sort* akan dibahas lebih jauh juga dalam makalah ini.

2. PENGURUTAN

2.1. Pengertian Algoritma Pengurutan [5]

Dalam ilmu komputer, algoritma pengurutan adalah algoritma yang meletakkan elemen-elemen suatu kumpulan data dalam urutan tertentu. Yang pada kenyataannya ‘urutan tertentu’ yang umum digunakan adalah secara terurut secara numerikal ataupun secara leksikografi (urutan secara abjad sesuai kamus).

2.2. Klasifikasi Algoritma Pengurutan Berdasarkan Proses [5]

Banyak klasifikasi yang dapat digunakan untuk mengklasifikasikan algoritma-algoritma pengurutan, misalnya secara kompleksitas, teknik yang dilakukan, stabilitas, memori yang digunakan, rekursif/tidak, ataupun proses yang terjadi. Namun agar relevan dengan topik masalah ini, pengklasifikasian yang disajikan adalah menurut langkah yang terjadi dalam proses pengurutan tersebut. Pengklasifikasiannya adalah sebagai berikut :

1. *Exchange Sort*
   Dalam prosesnya, algoritma-algoritma pengurutan yang diklasifikasikan sebagai *exchange sort* melakukan pembandingan antar data, dan melakukan pertukaran apabila urutan yang didapat belum sesuai. Contohnya adalah : *Bubble sort, Cocktail sort, Comb sort, Gnome sort, Quicksort*.

2. *Selection Sort*
   Prinsip utama algoritma dalam klasifikasi ini

adalah mencari elemen yang tepat untuk diletakkan di posisi yang telah diketahui, dan meletakkannya di posisi tersebut setelah data tersebut ditemukan. Algoritma yang dapat diklasifikasikan ke dalam kategori ini adalah : *Selection sort, Heapsort, Smoothsort, Strand sort.*

3. *Insertion Sort*
   Algoritma pengurutan yang diklasifikasikan ke dalam kategori ini mencari tempat yang tepat untuk suatu elemen data yang telah diketahui ke dalam subkumpulan data yang telah terurut, kemudian melakukan penyisipan (*insertion*) data di tempat yang tepat tersebut. Contohnya adalah : *Insertion sor, Shell sort, Tree sort, Library sort, Patience sorting.*

4. *Merge Sort*
   Dalam algoritma ini kumpulan data dibagi menjadi subkumpulan-subkumpulan yang kemudian subkumpulan tersebut diurutkan secara terpisah, dan kemudian digabungkan kembali dengan metode *merging.* Dalam kenyataannya algoritma ini melakukan metode pengurutan *merge sort* juga untuk mengurutkan subkumpulan data tersebut, atau dengan kata lain, pengurutan dilakukan secara rekursif. Contohnya adalah : *Merge sort.*

5. *Non-Comparison Sort*
   Sesuai namanya dalam proses pengurutan data yang dilakukan algoritma ini tidak terdapat pembandingan antardata, data diurutkan sesuai dengan *pigeon hole principle.* Dalam kenyataanya seringkali algoritma *non-comparison sort* yang digunakan tidak murni tanpa pembandingan, yang dilakukan dengan menggunakan algoritma-algoritma pengurutan cepat lainnya untuk mengurutkan subkumpulan-subkumpulan datanya. Contohnya adalah : *Radix sort, Bucket sort, Counting sort, Pigeonhole sort, Tally sort.*

3. PENGURUTAN TANPA PEMBANDINGAN

Sebenarnya klasifikasi-klasifikasi di atas tersebut bila ditelaah lebih lanjut dapat kita rampatkan menjadi 2 klasifikasi saja, yaitu pengurutan dengan pembandingan (*comparison sort*), dan pengurutan tanpa pembandingan (*non comparison sort*). Dan kedua klasifikasi tersebut akan dikaji lebih lanjut. Tentu saja membahas pengurutan dengan pembandingan (*comparison sort*) saja tidak cukup, namun harus disertai pula dengan pembahasan mengenai pengurutan tanpa pembandingan (*non comparison sort*) sebagai perbandingan.

3.1. Pengurutan Tanpa Pembandingan

Pengurutan tanpa pembandingan (*non comparison sort*) adalah algoritma pengurutan di mana dalam prosesnya tidak melakukan pembandingan antardata.

Secara umum yang proses yang dilakukan dalam metode ini adalah mengklasifikasikan data sesuai dengan kategori terurut yang tertentu, dan dalam tiap kategori dilakukan pengklasifikasian lagi, dan seterusnya sesuai dengan kebutuhan, kemudian subkategori-subkategori tersebut digabungkan kembali, yang secara dilakukan hanya dengan metode sederhana *concatenation.*

Secara kompleksitas, dalam berbagai kasus tertentu, algoritma tanpa pembandingan ini dapat bekerja dalam waktu linier, atau dengan kata lain memiliki kompleksitas $O(n)$. Yang akan ditelaah lebih detail pada masing-masing algoritma, yaitu *radix sort dan counting sort.*

Salah satu kelemahan dari metode ini adalah selalu diperlukannya memori tambahan.

3.2 *Comparison Sort* sebagai Perbandingan

3.2.1. *Comparison Sort*

Pengurutan dengan pembandingan dalam proses pengurutan melakukan pembandingan antardata, yang kemudian diikuti dengan pertukaran data bila tidak sesuai dengan syarat keterurutan tertentu.

3.3.2. Kompleksitas *Comparison Sort* [2]

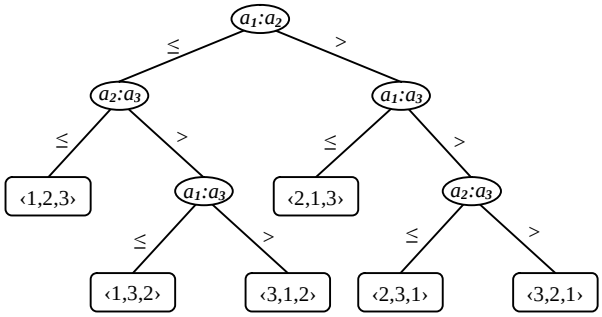
Contoh dari algoritma pengurutan yang termasuk *comparison sort* adalah *Merge sort* dan *Quick sort, Merge sort* memiliki kompleksitas $O(n \log n)$ dalam kasus terburuknya, sedangkan *Quick sort* dapat mencapainya dalam kasus rata-rata.

Kenyataannya, dalam prosesnya mengurutkan n data, semua pengurutan dengan pembandingan (*comparison sort*) harus melakukan $\Omega(n \log n)$ pembandingan data dalam kasus terburuknya.

Bukti dari pernyataan di atas adalah sebagai berikut :

Dalam pengurutan dengan pembandingan, akan dilakukan pembandingan antardata, misal dalam sekumpulan data $(a_1, a_2, \dots, a_n)$, kita akan melakukan pengetesan apakah $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i > a_j$, ataupun $a_i \geq a_j$ untuk menentukan urutan relatifnya. Pembandingan $a_i = a_j$ dapat dihilangkan karena tidak berguna dalam proses pengurutan ini, dan pembandingan $a_i < a_j$, $a_i \leq a_j$, $a_i > a_j$, ataupun $a_i \geq a_j$ sebenarnya sama saja karena dalam proses pengurutan kita hanya ingin mendapatkan urutan antara a_i dan a secara relatif. Jadi dapat diasumsikan semua pembandingan dilakukan dengan menggunakan $a_i \leq a_j$.

Dan semua pembandingan yang dilakukan dalam pengurutan dengan pembandingan (*comparison sort*) dapat digambarkan dengan pohon keputusan. Misalnya untuk data dengan 3 elemen, pohon keputusannya adalah sebagai berikut :



Tentu saja penggambaran dengan menggunakan pohon keputusan tersebut adalah untuk penggambaran pengurutan dengan pembandingan (*comparison sort*) secara umum. Kontrol, pergerakan data, dan aspek-aspek lainnya yang terdapat dalam algoritma-algoritma implementasinya diabaikan dalam pembuktian ini.

Dalam pohon keputusan di atas tersebut, setiap simpul dalam dinotasikan dengan $a_i : a_j$ untuk i dan j dalam rentang $1 \le i, j \le n$. Yang dalam contoh ini n adalah 3.

Dan eksekusi dari pengurutan yang dilakukan dengan menelusuri jalur dari akar hingga daunnya. Dan kemungkinan terburuk terjadi bila jalur yang ditempuh adalah jalur paling panjang, atau dengan kata lain bila menempuh jarak sepanjang kedalaman dari pohon tersebut.

Dengan teorema : "Semua pohon keputusan yang mengurutkan n elemen memiliki kedalaman $\Omega(n \log n)$", yang dibuktikan dengan :

> **Bukti**. Misal pohon keputusan memiliki kedalaman h dan melakukan pengurutan n elemen. Maka pohon ini akan memiliki setidaknya $n!$ daun, dan karena pohon keputusan adalah pohon biner yang dengan ketinggian h paling banyak akan memiliki 2^h daun, maka :
>
> $n! \le 2^h$, yang ekuivalen dengan
>
> $h \ge \log (n!)$, dan dengan *Stirling's approximation*, akan didapatkan
>
> $n! > \log (n / e)^n$, di mana $e = 2{,}71828 \ldots$
>
> $h \ge \log (n / e)^n$
>
> $= n \log n - n \log e$
>
> $= \Omega(n \log n)$

Dan dengan telah jelasnya keterkaitan antara metode pengurutan ini dengan pohon keputusan, maka telah terlihat dengan jelas mengapa semua *comparison sort* harus melakukan $\Omega(n \log n)$ pembandingan data dalam kasus terburuknya.

4. *COUNTING SORT*

4.1. Ide Dasar *Counting Sort*

Untuk dapat melakukan pengurutan dengan *counting sort* rentang nilai untuk kumpulan data yang akan diurutkan harus diketahui, katakanlah *k*.

Ide dasar dari *counting sort* adalah menentukan, untuk setiap elemen *x*, jumlah elemen yang lebih kecil daripada *x*, yang kemudian informasi ini digunakan untuk menentukan posisi *x*. Contoh sederhana saja, jika terdapat 12 elemen yang lebih kecil daripada *x*, maka *x* akan mendapatkan posisinya di posisi 13.

Tentu saja, sedikit modifikasi harus dilakukan agar metode ini dapat menangani kasus di mana terdapat elemen-elemen lain yang nilainya sama dengan *x*. Di mana tentu saja kita tidak dapat menempatkan semua elemen yang nilainya sama dengan *x* di posisi yang sama.

4.2. Implementasi *Counting Sort*

Misal *array* data yang akan diurutkan adalah *A*. *Counting sort* membutuhkan sebuah *array C* berukuran *k*, yang setiap elemen *C*[*i*] merepresentasikan jumlah elemen dalam *A* yang nilainya adalah *i*. Di *array* inilah penghitungan (*counting*) yang dilakukan dalam pengurutan ini disimpan.

Misal kita akan melakukan pengurutan pada *array* A sebagai berikut, dengan *n* adalah 10 dan diasumsikan bahwa rentang nilai setiap *A*[*i*] adalah 1..5 :

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

Dan *array C* setelah diinisialisasi adalah :

C

0	0	0	0	0
1	2	3	4	5

Kemudian proses penghitungan pun dimulai, proses ini linier, dilakukan dengan menelusuri *array A*,
Langkah 1 : pembacaan pertama mendapat elemen *A*[1] dengan isi 1, maka C[1] ditambah 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

C

1	0	0	0	0
1	2	3	4	5

Langkah 2 : pembacaan kedua mendapat elemen *A*[2] dengan isi 3, maka C[3] ditambah 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

C

1	0	1	0	0
1	2	3	4	5

Langkah 3 : pembacaan ketiga mendapat elemen *A*[3] dengan isi 5, maka C[5] ditambah 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

C

1	0	1	0	1
1	2	3	4	5

Langkah 4 : pembacaan keempat mendapat elemen *A*[4] dengan isi 4, maka C[4] ditambah 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

C

1	0	1	1	1
1	2	3	4	5

.
.
.

Demikian dilakukan terus menerus hingga semua elemen *A* telah diakses. Hingga setelah langkah ke 10 didapatkan *array C* sebagai berikut :

C

1	2	3	1	3
1	2	3	4	5

Lalu *array C* diproses sehingga setiap elemen *C*, *C*[*i*] tidak lagi merepresentasikan jumlah elemen dengan nilai sama dengan *i*, namun setiap *C*[*i*] menjadi merepresentasikan jumlah elemen yang lebih kecil atau sama dengan *i*.

Dan setelah proses tersebut dilakukan didapatkan *C* sebagai berikut :

C

1	3	6	7	10
1	2	3	4	5

Setelah *C* didapat dilakukan proses penempatan sesuai dengan posisi yang didapat. Proses ini dilakukan dengan menelusuri kembali *A* dari belakang. Mengapa dari belakang? Karena kita mengharapkan hasil pengurutan yang *stable*, yang akan sangat penting dalam pengurutan data majemuk. Dan juga agar dapat diterapkan dalam metode *Radix sort*.

Dalam proses ini kita mengakses elemen *A*[*i*], kemudian memposisikannya di posisi sebagaimana tercatat dalam *C*[*A*[*i*]], kemudian kita mengurangkan *C*[*A*[*i*]] dengan 1, yang dengan jelas untuk memberikan posisi untuk elemen berikutnya dengan yang isinya sama dengan *A*[*i*].

Proses ini memerlukan sebuah array bantu *B* yang ukurannya sama dengan array *A*, yaitu *n*. Yang pada awalnya semua *B*[*i*] diinisialisasi dengan nil.

B

-	-	-	-	-	-	-	-	-	-
1	2	3	4	5	6	7	8	9	10

Langkah 1 : elemen *A*[10] adalah 5, maka karena *C*[5] adalah 10, maka *B*[10] diisi dengan 5, dan *C*[5] dikurangi 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

B

-	-	-	-	-	-	-	-	-	5
1	2	3	4	5	6	7	8	9	10

C

1	3	6	7	9
1	2	3	4	5

Langkah 2 : elemen *A*[9] adalah 3, maka karena *C*[3] adalah 6, maka *B*[6] diisi dengan 3, dan *C*[3] dikurangi 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

B

-	-	-	-	-	3	-	-	-	5
1	2	3	4	5	6	7	8	9	10

C

1	3	5	7	9
1	2	3	4	5

Langkah 3 : elemen *A*[8] adalah 3, maka karena *C*[3] adalah 5, maka *B*[5] diisi dengan 3, dan *C*[3] dikurangi 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

B

-	-	-	-	3	3	-	-	-	5
1	2	3	4	5	6	7	8	9	10

C

1	3	4	7	9
1	2	3	4	5

Langkah 3 : elemen *A*[7] adalah 2, maka karena *C*[2] adalah 3, maka *B*[3] diisi dengan 2, dan *C*[2] dikurangi 1.

A

1	3	5	4	5	2	2	3	3	5
1	2	3	4	5	6	7	8	9	10

B

-	-	2	-	3	3	-	-	-	5
1	2	3	4	5	6	7	8	9	10

C

1	2	4	7	9
1	2	3	4	5

.
.
.

Demikian proses dilakukan hingga elemen *A*[1] selesai diproses, sehingga didapatkan hasil akhir

B

1	2	2	3	3	3	4	5	5	5
1	2	3	4	5	6	7	8	9	10

Yang telah berupa array dengan elemen-elemennya terurut secara *non-decreasing.*

4.3. Algoritma dan Kompleksitas Waktu *Counting Sort*

Dari implementasi yang telah dilakukan tersebut, penulis mengimplementasikannya manjadi sebuah program menggunakan bahasa *Pascal* sebagai berikut :

```
procedure CountingSort (A : TArray;
var B : TArray);

var
  C   : array [1..k] of byte;
  i   : integer;

begin

  for i:=1 to k do
    C[i] := 0;

  for i:=1 to n do
    C[A[i]] := C[A[i]] + 1;

  for i:=2 to k do
    C[i] := C[i] + C[i-1];

  for i:=n downto 1 do begin
    B[C[A[i]]] := A[i];
    C[A[i]] := C[A[i]] - 1;
  end;

end;
```

Waktu yang dibutuhkan untuk mengurutkan data menggunakan *counting sort* bisa didapatkan dari perhitungan sebagai berikut :

Kalang pertama membutuhkan waktu $O(k)$,
kalang kedua membutuhkan waktu $O(n)$,
kalang ketiga membutuhkan waktu $O(k)$, dan
kalang keempat membutuhkan waktu $O(n)$.

Jadi secara total membutuhkan waktu $O(k+n)$, yang seringkali dianggap $k = O(n)$, sehingga waktu total yang dibutuhkan menjadi $O(n)$.

4.4. Analisis mengenai *Counting Sort*

Dengan contoh dan algoritma yang telah dibahas, jelas bahwa *counting sort* melakukan pengurutan tanpa melakukan pembandingan antardata.

Syarat agar pengurutan ini dapat dilakukan adalah diketahuinya rentang nilai data-datanya. Data-data yang akan diurutkan juga harus berupa bilangan bulat (*integer*).

Counting sort bisa efisien bila *k* tidak jauh lebih besar daripada *n*. Di mana semakin besar *k* maka memori tambahan yang diperlukan menjadi sangat besar. Contoh di mana *counting sort* dapat menjadi efisien adalah bila mengurutkan siswa-siswa dalam sebuah sekolah berdasar nilainya, dengan nilai adalah bilangan bulat dengan rentang 0..100. Dan contoh di mana *counting sort* akan sangat buruk kinerjanya adalah untuk data yang rentangnya sangat besar, misal dengan rentang $0..2^{32}-1$.

Counting sort memiliki properti yang penting, yaitu ke-*stable*-an. Di mana data-data dengan nilai yang sama akan diurutkan berdasar urutan kemunculan pada *array* asal. Properti ini sangat penting dalam pengurutan data majemuk.

Dengan kompleksitas $O(n)$, metode pengurutan ini sangat cepat dan efektif. Yang diimbangi dengan kelemahan yaitu dibutuhkan memori tambahan sebagai *array* bantu dalam prosesnya.

4. *RADIX SORT*

4.1. Ide dasar *Radix Sort*

Sebelum memulai membahas lebih lanjut, akan lebih baik jika diadakan penyamaan persepsi mengenai istilah *radix*, dalam makalah ini *radix* bermakna harafiah posisi dalam angka [1]. Di mana sederhananya, dalam representasi desimal, *radix* adalah digitnya.

Ide dasar dari metode *Radix sort* ini adalah mengkategorikan data-data menjadi subkumpulan-subkumpulan data sesuai dengan nilai *radix*-nya, mengkonkatenasinya, kemudian mengkategorikannya kembali berdasar nilai *radix* lainnya.

Dalam kenyataannya banyak sekali algoritma *Radix sort* yang berbeda-beda walaupun ide dasarnya sama, ada yang menggunakan queue, ada yang

menggunakan bantuan algoritma pengurutan lain untuk mengurutkan data per kategori (dan yang biasa digunakan adalah *counting sort* [2]), ataupun menggunakan rekursif.

Selain itu, ada 2 macam *Radix sort* berdasarkan urutan pemrosesan *radix*-nya, yaitu secara LSD (*Least Significant Digit*) di mana pemrosesan dimulai dari *radix* yang paling tidak signifikan, dan satu lagi secara MSD (*Most Significant Digit*), di mana pemrosesan dimulai dari *radix* yang paling signifikan. Namun dalam makalah ini yang dibahas adalah *LSD radix sort*, di mana langkah yang dilakukan lebih sederhana dibanding dengan *MSD radix sort* yang dalam pemrosesannya diperlukan penyimpanan *track* dari pemrosesan sebelumnya.

Agar relevan dengan topik utama makalah ini yang mengedepankan pengurutan tanpa pembandingan, yang dibahas hanyalah *radix sort* yang dalam prosesnya hanya menggunakan pengkategorian berulang, tanpa menggunakan bantuan algoritma pengurutan lain.

4.2. Implementasi *Radix Sort*

Contoh implementasi yang akan dilakukan adalah implementasi pada bilangan bulat positif menggunakan salah satu algoritma pengurutan *radix sort*.

Contohnya adalah pengurutan sebuah kumpulan data bilangan bulat dengan jumlah digit maksimal 3 :

121	076	823	367	232	434	742	936	274

Pertama kali, data dibagi-bagi sesuai dengan digit terkanan :

121	076	823	367	232	434	742	936	274

Kategori digit	Isi
0	-
1	121
2	232, 742
3	823
4	434, 274
5	-
6	076, 936
7	367
8	-
9	-

Hasil pengkategorian tersebut lalu digabung kembali dengan metode konkatenasi menjadi :

121	232	742	823	434	274	076	936	367

Kemudian pengkategorian dilakukan kembali,namun kali ini berdasar digit kedua atau digit tengah, dan jangan lupa bahwa urutan pada tiap subkumpulan data harus sesuai dengan urutan kemunculan pada kumpulan data.

121	232	742	823	434	274	076	936	367

Kategori digit	Isi
0	-
1	-
2	121, 823
3	232, 434, 936
4	742
5	-
6	367
7	274, 076
8	-
9	-

Yang kemudian dikonkatenasi kembali menjadi

121	823	232	434	936	742	367	274	076

Kemudian langkah ketiga, atau langkah terakhir pada contoh ini adalah pengkategorian kembali berdasar digit yang terkiri, atau yang paling signifikan :

121	823	232	434	936	742	367	274	076

Kategori digit	Isi
0	076
1	121
2	232, 274
3	367
4	434
5	-
6	-
7	742
8	823
9	936

Yang kemudian dikonkatenasi lagi menjadi

076	121	232	274	367	434	742	823	936

Yang merupakan hasil akhir dari metode pengurutan ini. Di mana data telah terurut dengan metode *radix sort.*

4.3. Algoritma dan Kompleksitas Waktu *Radix Sort*

Implementasi dari algoritma tersebut dibuat penulis menggunakan bahasa *Pascal*, yang direalisasikan dengan menggunakan *queue* sebagai representasi tiap kategori *radix* untuk pengkategorian. *Array* A adalah *array input*, dan *array* B adalah *array* A yang sudah terurut.

```
Procedure RadixSort (A : TArray; var B :
TArray; d : byte);
var
  KatRadix  : array [0..9] of Queue;
  i, x, ctr : integer;
  pembagi   : longword;

begin

  {--- mengkopi A ke B ---}
  for i:=1 to n do
    B[i] := A[i];

  pembagi := 1;
  for x:=1 to d do begin
    {--- inisialisasi KatRadix ---}
    for i:=0 to 9 do
      InitQueue (KatRadix[i]);

    {--- dikategorikan ---}
    for i:=1 to n do
      Enqueue (KatRadix [(B[i] div
pembagi) mod 10], B[i]);
      B[i] := 0;

    {--- dikonkat ---}
    ctr := 0;
    for i:=0 to 9 do begin
      while (NOT IsQueueEmpty
(KatRadix[i])) do begin
        ctr := ctr + 1;
        B[ctr]:=DeQueue (KatRadix [i]);
      end;
    end;

    pembagi := pembagi * 10;
  end;

end;
```

Dari algoritma di atas, dapat dihitung waktu yang diperlukan untuk melakukan pengurutan menggunakan metode *radix sort*.

Kalang luar pertama : $O(n)$

Kalang dalam pertama : $O(1)$
Kalang dalam kedua : $O(n)$
Kalang dalam ketiga : $O(n)$
Sehingga kompleksitasnya adalah $O(n)$

Semua itu kalang dalam tersebut terdapat dalam kalang luar kedua, yang memiliki kompleksitas $O(d)$. Jadi secara keseluruhan prosedur *Radix Sort* di atas memiliki kompleksitas $O(nd)$.

4.4. Analisis mengenai *Radix Sort*

Dari langkah-langkah yang dilakukan dalam proses pengurutan menggunakan *radix sort*, jelas tampak bahwa *radix sort* termasuk dalam kategori pengurutan tanpa pembandingan.
Dalam contoh yang menggunakan bilangan desimal, *radix* adalah *digit*, namun dalam penggunaannya *radix* bisa saja dalam ukuran-ukuran 3 bit (oktal), maupun karakter untuk pengurutan *string*. Yang perlu diperhatikan adalah bahwa semakin besar *d* semakin besar juga ruang yang diperlukan, namun jika *d* kecil, iterasi yang perlu dilakukan menjadi semakin banyak. Jadilebih baik pemilihan *d* disesuaikan dengan kebutuhan dan memori maupun waktu yang tersedia.

Dalam penggunaannya, *radix sort* bahkan bisa dimodifikasi sehingga bisa digunakan untuk mengurutkan data-data negatif, *float*, dan pecahan untuk kasus-kasus tertentu.

Radix sort adalah algoritma pengurutan data yang cepat, efektif, dan sederhana.

5. KESIMPULAN

Dari bahasan di atas, dapat disimpulkan Algoritma Pengurutan Tanpa Pembandingan (*Non-Comparison Sorting*) adalah algoritma pengurutan yang cepat, efektif, namun penggunaannya terbatas pada kasus-kasus tertentu dan memerlukan memori tambahan dalam prosesnya.

Dan secara umum kompleksitasnya adalah linier, di mana lebih baik daripada algoritma Pengurutan Dengan Pembandingan (*Comparison Sort*) yang memiliki *lower-bound* $\Omega(n \log n)$.

Secara lebih mendalam, *Counting Sort* adalah algoritma pengurutan efektif dan efisien yang melakukan pengurutan dengan ide dasar meletakkan elemen pada posisi yang benar, di mana penghitungan posisi yang benar dilakukan dengan cara menghitung (*counting*) elemen-elemen dengan nilai lebih kecil atau sama dengan elemen tersebut. Dan memiliki kompleksitas waktu linier. Walaupun tidak dapat digunakan secara luas karena banyaknya batasan.

Radix Sort melakukan pengurutan dengan cara mengkategorikan data sesuai dengan *radix* tertentu secara berulang-ulang. Memerlukan memori tambahan yang berbanding lurus dengan *d*, di mana *d* adalah *radix* yang digunakan. Memiliki kompleksitas waktu linier juga seperti *Counting Sort*.

DAFTAR REFERENSI

[1] DF Alfatwa, ER Syah P, FM Ahsan, "Implementasi Algoritma Radix Sort dalam Berbagai Kasus Bilangan Dibandingkan Algoritma Pengurutan yang lain". Bandung. 2005.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction To Algorithms*, McGraw-Hill. 1990.

[3] Handbook UI, Algoritma-algoritma pengurutan internal
http://ranau.cs.ui.ac.id/sda/archive/1998/handout/handout24.html
Tanggal akses : 29 Desember 2007 pukul 04.00 GMT +7

[4] Wikipedia, (2007). Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Counting_Sort
Tanggal akses : 29 Desember 2007 pukul 04.00 GMT +7

[5] Wikipedia, (2007). Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Sorting_Algorithm
Tanggal akses : 29 Desember 2007 pukul 04.00 GMT +7

[6] Wikipedia, (2007). Wikipedia, the free encyclopedia.
http://en.wikipedia.org/wiki/Radix_Sort
Tanggal akses : 29 Desember 2007 pukul 04.00 GMT +7