Analisis Algoritma Radix Sort

Arief Rahman (13511020)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
ariefrahman95@students.itb.ac.id

***Abstrak*—Makalah ini membahas tentang algoritma *radix sort*, yang merupakan salah satu bentuk algoritma *sorting* (pengurutan). Algoritma in termasuk salah satu algoritma pengurutan yang unik karena tidak menggunakan operasi perbandingan. Algoritma ini akan dilihat berdasarkan efektivitasnya, yaitu berdasarkan kompleksitas dan tingkat kesulitan *coding*-nya.**

Kata kunci*—Efektivitas, kompleksitas algoritma, *sorting

I. PENDAHULUAN

1.1 Kompleksitas Algoritma

Algoritma adalah urutan langkah-langkah penyelesaian masalah secara sistematis. Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (efficient). Algoritma yang benar sekalipun mungkin tidak berguna untuk jenis dan ukuran masukan tertentu karena waktu yang diperlukan untuk menjalankan algoritma tersebut atau ruang memori yang diperlukan untuk struktur datanya terlalu besar.

Dalam aplikasinya, setiap algoritma memiliki dua buah ciri khas yang dapat digunakan sebagai parameter pembanding, yaitu jumlah proses yang dilakukan dan jumlah memori yang digunakan untuk melakukan proses. Jumlah proses ini dikenal sebagai kompleksitas waktu yang disimbolkan dengan T(n), sedangkan jumlah memori ini dikenal sebagai kompleksitas ruang yang disimbolkan dengan S(n).

Kompleksitas waktu diukur berdasarkan jumlah operasi atau instruksi yang dieksekusi. Kompleksitas waktu tidak diukur berdasarkan *runtime* aplikasi secara nyata karena arsitektur computer dan compiler berbeda satu sama lain. Oleh karena itu, pada computer dan/atau compiler yang berbeda, suatu algoritma yang sama akan memiliki waktu eksekusi yang berbeda.

Dalam suatu algoritma, terdapat berbagai jenis operasi :

- Operasi baca tulis (*input* dan *output*).
- Operasi aritmatika (penambahan, pengurangan, perkalian, dan pembagian).
- Operasi perbandingan.
- Operasi *assignment* (pemasukan nilai).
- Operasi pemanggilan fungsi dan prosedur.

Dalam menghitung kompleksitas waktu suatu algoritma, kita hanya menghitung jumlah suatu operasi tertentu yang bersifat tipikal di algoritma tersebut.

1.2 Kompleksitas Waktu

Seperti yang telah disebutkan pada upabab sebelumnya, kompleksitas waktu diukur berdasarkan jumlah operasi tipikal yang dilakukan oleh suatu algoritma.

Kompleksitas waktu suatu algoritma dapat dibedakan menjadi tiga:

- Kasus terbaik
- Kasus terburuk
- Kasus rata-rata

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x : integer,
                            output idx : integer)
Deklarasi
  k : integer
  ketemu : boolean   { bernilai true jika x ditemukan atau false jika x
tidak ditemukan }

Algoritma:
  k←1
  ketemu ← false
  while (k ≤ n) and (not ketemu) do
    if ak = x then
      ketemu←true
    else
      k ← k + 1
    endif
  endwhile
  { k > n or ketemu }

  if ketemu then   { x ditemukan }
     idx←k
  else
     idx← 0       { x tidak ditemukan }
  endif
```

Gambar 1.2.1 Algoritma *sequential search*

Kompleksitas algoritma pada gambar 1.2.1 dapat ditentukan berdasarkan jumlah operasi perbandingan yang dieksekusi.

- Untuk kasus terbaik ($a_1 = x$), T(n) = 1.
- Untuk kasus terburuk (a_n = x atau x tidak ditemukan), T(n) = n.
- Untuk kasus rata-rata, jika x ditemukan pada posisi ke-j, maka :
  T(n) = (1 + 2 + 3 + … + n) / n = (n + 1) / 2.

1.3 Kompleksitas Waktu Asimptotik

Pada umumnya, kita tidak terlalu membutuhkan kompleksitas waktu yang detil dari suatu algoritma. Biasanya kita hanya membutuhkan hampiran dari kompleksitas waktu yang sebenarnya, atau lebih umum disebut "orde" dari algoritma tersebut. Kompleksitas waktu ini disebut kompleksitas waktu asimptotik yang dinotasikan dengan "O" (baca : *Big "O"* atau "O-besar"). Kompleksitas waktu asimptotik secara umum didapat dari pengambilan *term* terbesar dari suatu persamaan komplesitas waktu T(n). Misalkan:

$$T(n) = 3n^2 - 8n + 10$$

Dari persamaan di atas, notasi O-besar-nya adalah $O(n^2)$ karena n^2 adalah *term* terbesar pada persamaan T(n) tersebut.

Tabel 1 menunjukkan kelompok algoritma berdasarkan notasi O-besar:

Kelompok Algoritma	Nama
O(1)	konstan
O(log n)	logaritmik
O(n)	lanjar
O(n log n)	n log n
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
O(n!)	faktorial

Tabel 1 Pengelompokan Algoritma Berdasarkan Notasi O-Besar

Kompleksitas waktu asimptotik memiliki spektrum, seperti spektrum cahaya. Berikut adalah urutan kompleksitas waktu asimptotik algoritma, dari yang terkecil hingga ke yang terbesar:

$O(1) < O(\log n) < O(n) < O\ (n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

II. ALGORITMA RADIX SORT

2.1 Konsep Dasar *Radix Sort*

Algoritma *radix sort* adalah salah satu algoritma pengurutan yang paling mangkus karena tidak menggunakan perbandingan secara langsung. Untuk kasus bilangan bulat (*integer*), algoritma ini akan mengurutkan data dengan mengelompokkan data-data berdasarkan digit yang memiliki *significant position* dan *value* yang sama. Kelompok digit ini ditampung dalam suatu variable "*bucket*". Struktur datanya direpresentasikan dengan *array*. Algoritma ini pertama kali diperkenalkan pada tahun 1887 oleh Herman Hollerith pada mesin tabulasi.

Ada dua jenis *radix sort* saat ini :

- LSD (*least significant digit*) *radix sort*, yaitu *radix sort* yang mengurutkan data dimulai dari digit terkecil. Algoritma ini cenderung stabil karena tetap mengikuti urutan awal data-data sebelum diurutkan.
- MSD (*most significant digit*) *radix sort*, yaitu *radix sort* yang mengurutkan data dimulai dari digit terbesar. Algoritma ini lebih susah untuk direalisasikan daripada LSD *radix sort*, dan biasanya tidak mengikuti urutan awal data-data yang akan diurutkan.

Makalah ini hanya akan membahas LSD *radix sort*, mengingat batasan makalah yang diberikan.

Misalkan terdapat sebuah *array* dengan elemen "34 52 124 8 4 10". Dengan menggunakan algoritma *radix sort*:

- **Pass pertama :**
  Urutkan *array* berdasarkan digit terkecil (satuan) menjadi :
  1<u>0</u> 5<u>2</u> 3<u>4</u> 12<u>4</u> <u>4</u> <u>8</u>
  Catatan : 124 ditulis sebelum 4 karena pada urutan awal, 124 terletak sebelum 4.
  Pada *pass* ini, terbentuk *array of bucket size* berikut:
  1 (ukuran *bucket* digit 0 : 01<u>0</u>)
  1 (ukuran *bucket* digit 2 : 05<u>2</u>)
  3 (ukuran *bucket* digit 4 : 03<u>4</u>, 12<u>4</u>, 00<u>4</u>)
  1 (ukuran *bucket* digit 8 : 00<u>8</u>)

- **Pass kedua :**
  Setelah itu, urutkan *array* berdasarkan digit selanjutnya (puluhan) menjadi :
  4 8 <u>1</u>0 12<u>4</u> <u>3</u>4 <u>5</u>2
  Catatan : Bilangan yang tidak memiliki puluhan dianggap memiliki digit puluhan "0".
  Pada *pass* ini, terbentuk *array of bucket size* berikut:
  2 (ukuran *bucket* digit 0 : 0<u>0</u>4, 0<u>0</u>8)
  1 (ukuran *bucket* digit 1 : 0<u>1</u>0)
  1 (ukuran *bucket* digit 2 : 1<u>2</u>4)
  1 (ukuran *bucket* digit 3 : 0<u>3</u>4)
  1 (ukuran *bucket* digit 5 : 0<u>5</u>2)

- **Pass ketiga :**
  Terakhir, urutkan *array* berdasarkan digit terbesar (ratusan) menjadi :
  4 8 10 34 52 <u>1</u>24
  Pada *pass* ini, terbentuk *array of bucket size* berikut:
  5 (ukuran *bucket* digit 0 : <u>0</u>04, <u>0</u>08, <u>0</u>10, <u>0</u>34, <u>0</u>52)
  1 (ukuran *bucket* digit 1 : <u>1</u>24)

Pada contoh di atas, setiap pengurutan digit dilakukan dalam sekali *pass*. Oleh karena itu, pada contoh di atas, terjadi tiga kali *pass*.

Ada banyak variasi yang digunakan dalam LSD *radix sort*. Perbedaan yang paling mencolok terletak pada representasi *bucket* untuk menyimpan digit data-data. Secara umum, *bucket* LSD *radix sort* direpresentasikan dengan *array*, tetapi dalam beberapa kasus, *bucket* dapat direpresentasikan dengan menggunakan *queue*.

Misalkan sebuah *array* dengan elemen "170 45 75 90 802 24 2 66".

- **Pass pertama :** Data-data dikelompokkan ke dalam *queue* berdasarkan digit terkecil dari kanan ke kiri.
  Queue yang terbentuk:
  0 : 17<u>0</u>, 09<u>0</u>
  1 : NIL
  2 : 00<u>2</u>, 80<u>2</u>
  3 : NIL
  4 : 02<u>4</u>
  5 : 04<u>5</u>, 07<u>5</u>
  6 : 06<u>6</u>
  7 : NIL
  8 : NIL
  9 : NIL
- Isi *queue* dikembalikan ke *array* mulai dari *head* (0) :

170, 090, 002, 802, 024, 045, 075, 066

- **Pass kedua** : Kelompokkan data-data ke *queue* berdasarkan digit selanjutnya:
  0 : 002, 802
  1 : NIL
  2 : 024
  3 : NIL
  4 : 045
  5 : NIL
  6 : 066
  7 : 170, 075
  8 : NIL
  9 : 090
  Isi *array* setelah pemasukan dari *queue*:
  002, 802, 024, 045, 066, 170, 075, 090

- **Pass ketiga** : Kelompokkan data-data ke *queue* berdasarkan digit terbesar:
  0 : 002, 024, 045, 066, 075, 090
  1 : 170
  2 : NIL
  3 : NIL
  4 : NIL
  5 : NIL
  6 : NIL
  7 : NIL
  8 : 802
  9 : NIL
- Isi *array* setelah pemasukan dari *queue*:
  002, 024, 045, 066, 075, 090, 170, 802
  Array telah terurut membesar.

Berdasarkan contoh-contoh di atas, kita dapat menentukan langkah-langkah yang harus dilakukan dalam algoritma *radix sort* secara umum:
- Ambil digit terkecil dari setiap data.
- Kelompokkan data-data berdasarkan digit. Untuk digit data sama, jangan ubah urutan data-data.
- Ulangi langkah pertama untuk digit selanjutnya.

2.2 Realisasi *Radix Sort*

Realisasi *radix sort* di C dapat dilihat pada gambar 2.2.1. Representasi *bucket* berupa *array* digunakan di algoritma ini.

```c
#include <stdio.h>
#define MAX 5
#define SHOWPASS
void print(int *a, int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("%d\t", a[i]);
}

void radixsort(int *a, int n)
{
  int i, b[MAX], m = a[0], exp = 1;
  for (i = 0; i < n; i++)
  {
    if (a[i] > m)
      m = a[i];
  }

  while (m / exp > 0)
  {
    int bucket[10] =
    {  0 };
    for (i = 0; i < n; i++)
      bucket[a[i] / exp % 10]++;
    for (i = 1; i < 10; i++)
      bucket[i] += bucket[i - 1];
    for (i = n - 1; i >= 0; i--)
      b[--bucket[a[i] / exp % 10]] = a[i];
    for (i = 0; i < n; i++)
      a[i] = b[i];
    exp *= 10;

    #ifdef SHOWPASS
      printf("\nPASS   : ");
      print(a, n);
    #endif
  }
}


int main()
{
  int arr[MAX];
  int i, n;

  printf("Enter total elements (n < %d) : ", MAX);
  scanf("%d", &n);

  printf("Enter %d Elements : ", n);
  for (i = 0; i < n; i++)
    scanf("%d", &arr[i]);


  printf("\nARRAY  : ");
  print(&arr[0], n);

  radixsort(&arr[0], n);

  printf("\nSORTED : ");
  print(&arr[0], n);
  printf("\n");

  return 0;
}
```

Gambar 2.2.1 Realisasi *Radix Sort* dengan Bahasa Pemrograman C

Realisasi *radix sort* di Python dapat dilihat pada gambar 2.2.2. Seperti pada contoh sebelumnya, representasi *bucket* berupa *array* digunakan di algoritma ini.

```python
#python2.6 <
from math import log

def getDigit(num, base, digit_num):
    # pulls the selected digit
    return (num // base ** digit_num) % base

def makeBlanks(size):
    # create a list of empty lists to hold the split by digit
    return [ [] for i in range(size) ]

def split(a_list, base, digit_num):
    buckets = makeBlanks(base)
    for num in a_list:
        # append the number to the list selected by the digit
        buckets[getDigit(num, base, digit_num)].append(num)
    return buckets

# concatenate the lists back in order for the next step
def merge(a_list):
    new_list = []
    for sublist in a_list:
       new_list.extend(sublist)
    return new_list

def maxAbs(a_list):
    # largest abs value element of a list
    return max(abs(num) for num in a_list)

def radixSort(a_list, base):
    # there are as many passes as there are digits in the longest number
    passes = int(log(maxAbs(a_list), base) + 1)
    new_list = list(a_list)
    for digit_num in range(passes):
        new_list = merge(split(new_list, base, digit_num))
    return new_list
```

Gambar 2.2.1 Realisasi *Radix Sort* dengan Bahasa Pemrograman Python

Realisasi *radix sort* di Pascal dapat dilihat pada gambar 2.2.3. Representasi *bucket* berupa *queue* digunakan pada contoh ini.

```
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;
  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;
```

Gambar 2.2.3 Realisasi *Radix Sort* dengan Bahasa Pemrograman Pascal

Realisasi *radix sort* di C++ dapat dilihat pada gambar 2.2.4. Representasi *bucket* berupa *array* digunakan pada contoh ini.

```
#include<iostream>
#include<vector>
#include<iterator>
#include<algorithm>

typedef std::vector<unsigned int> input_type;

void radix_sort(input_type & x)
{
    if ( x.empty() ) return; // at least one element

    typedef std::vector< std::vector< input_type::value_type > > buckets_type;
    buckets_type buckets(10); // allocate buckets
    // for sorting decimal numbers

    int pow10 = 1; // pow10 holds powers of 10 (1, 10, 100, ...)

    // find maximum in the array to limit the main loop below
    input_type::value_type max = *std::max_element(x.begin(), x.end());

    //begin radix sort
    for(; max != 0 ; max/=10, pow10*=10)
    {
        // 1. determine which bucket each element should enter
        // for each element in 'x':
        for(input_type::const_iterator elem = x.begin(); elem != x.end(); ++elem)
        {
                // calculate the bucket number:
                size_t const bucket_num = ( *elem / pow10 ) % 10;
                // add the element to the list in the bucket:
                buckets[ bucket_num ].push_back( *elem );
        }

        // 2. transfer results of buckets back into main array
        input_type::iterator store_pos = x.begin();
        // for each bucket:
        for(buckets_type::iterator bucket = buckets.begin(); bucket != buckets.end(); ++bucket)
        {
                // for each element in the bucket:
                for(buckets_type::value_type::const_iterator bucket_elem = bucket->begin();
                        bucket_elem != bucket->end(); ++bucket_elem)
                {
                        // copy the element into next position in the main array
                        *store_pos++ = *bucket_elem;
                }
                bucket->clear(); // forget the current bucket's list
        }
    }
}

int main(){

    input_type input;

    // read numbers from standard input (ends with end-of-file: ^Z / ^D, or
    // with a new line that contains something that's not a number)
    std::cout << "Enter positive numbers to sort:" << std::endl;
    std::copy(std::istream_iterator<unsigned int>(std::cin),
        std::istream_iterator<unsigned int>(), std::back_inserter(input));

    if ( input.end() != std::find(input.begin(), input.end(), 0) )
    {
        std::cerr << "Zero isn't positive" << std::endl;
        return 1;
    }

    std::cout << " ** Elements before sorting: " << std::endl;
    std::copy(input.begin(), input.end(),
        std::ostream_iterator<unsigned int>(std::cout, " "));

    radix_sort(input);

    std::cout << std::endl << " ** Elements after sorting: " << std::endl;
    std::copy(input.begin(), input.end(),
        std::ostream_iterator<unsigned int>(std::cout, " "));
    std::cout << std::endl;

    return 0;
}
```

Gambar 2.2.4 Realisasi *Radix Sort* dengan Bahasa Pemrograman C++

2.3 Kompleksitas Algoritma *Radix Sort*

Kompleksitas waktu algoritma *radix sort* sama untuk

semua kasus (kasus terbaik, kasus terburuk, dan kasus rata-rata), karena operasi yang signifikan di algoritma ini adalah operasi assignment, bukan berupa operasi perbandingan. Penulis akan menggunakan algoritma Pascal untuk menentukan kompleksitas algoritma *radix sort*.

Berdasarkan operasi *assignment*, kita dapat menentukan kompleksitas waktu algoritma *radix sort*.

- Kalang luar pertama ("mengkopi A ke B") = O(n).
- Kalang dalam pertama ("inisialisasi KatRadix") = O(1).
- Kalang dalam kedua ("dikategorikan") = O(n).
- Kalang dalam ketiga ("dikonkat") = O(n).
- Kalang luar kedua = O(d)

Kompleksitas waktu asimptotik algoritma di atas adalah O(nd).

Secara umum, kompleksitas algoritma asimptotik algoritma *radix sort* adalah O(kN). N merupakan jumlah data, sedangkan k adalah jumlah digit. k dapat memiliki nilai yang berbeda-beda, tergantung dari jenis data yang sedang diproses.

2.4 Kelebihan & Kekurangan Algoritma *Radix Sort*

Algoritma *radix sort* memiliki kelebihan dan kekurangan yang berbeda dibandingkan dengan algoritma pengurutan yang lain. Beberapa kelebihan algoritma *radix sort* adalah sebagai berikut :

- Algoritma sangat mangkus. Hal ini dapat dilihat dari kompleksitas waktu asimptotiknya yang sangat kecil (O(kN)). Hal ini mengakibatkan algoritma *radix sort* sangat efektif untuk data dalam jumlah yang sangat besar sekalipun.
- Konsep algoritma mudah dipahami. Algoritma *radix sort* mengurutkan data berdasarkan digit, tidak melalui proses perbandingan yang cenderung sulit dipahami.

Walaupun memiliki banyak kelebihan dibandingkan algoritma pengurutan yang lain, algoritma ini memiliki kekurangan : realisasi program rumit dan kurang fleksibel untuk digunakan pada tipe data lain.

Realisasi program untuk algoritma *radix sort* tidak semudah memahami konsep dasarnya. Secara umum, algoritma ini membutuhkan *bucket* untuk mengelompokkan data-data yang sedang diurutkan. Inisialisasi *bucket* untuk kasus ini tidak mudah dilakukan.

Pada awalnya, *radix sort* hanya dapat digunakan untuk data bertipe bit dan desimal. Tetapi, seiring waktu, *radix sort* mulai dikembangkan untuk tipe data yang lain. Saat ini *radix sort* sudah dapat digunakan untuk tipe data berupa bilangan pecahan dan bilangan negatif. Akan tetapi, modifikasi terhadap algoritma ini menjadi signifikan. Pada umumnya, pengembangan algoritma *radix sort* untuk tipe data lain dibantu dengan menggunakan *counting sort*, yang merupakan salah satu algoritma pengurutan yang tidak menggunakan perbandingan. Penulis tidak akan membahas algoritma ini mengingat batasan makalah yang diberikan.

III. KESIMPULAN

- Algoritma *radix sort* melakukan pengurutan terhadap data berdasarkan digit (*radix*) dari data-data yang sedang diproses.
- Algoritma *radix sort* adalah salah satu algoritma tanpa perbandingan yang mangkus dan sederhana untuk dipahami, tetapi rumit dalam realisasinya.
- Algoritma *radix sort* dapat direpresentasikan dalam berbagai cara, terutama bagian *bucket*-nya. *Bucket* data dapat direpresentasikan dalam bentuk *array* atau *queue*.
- Algoritma *radix sort* dapat digunakan untuk tipe data selain bit dan desimal, tetapi diperlukan bantuan algoritma pengurutan lain, seperti *counting sort*.
- Kompleksitas algoritma *radix sort* secara umum adalah O(kN). k bergantung terhadap jenis data yang diproses untuk pengurutan.

REFERENSI

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2009-2010/Makalah0910/MakalahStrukdis0910-032.pdf
Tanggal akses 16 dan 17 Desember 2012

[2] http://www.scribd.com/doc/53014751/Algoritma-Radix-Sort
Tanggal akses 17 Desember 2012

[3] http://tid3ustj.blogspot.com/2011/11/pengkajian-algoritma-pengurutan-tanpa.html
Tanggal akses 17 Desember 2012

[4] http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=395781&KC=&FT=E&locale=en_EP
Tanggal akses 17 Desember 2012

[5] http://www.drdobbs.com/architecture-and-design/algorithm-improvement-through-performanc/221600153
Tanggal akses 17 Desember 2012

[6] R. Sedgewick, "Algorithms in C++", third edition, 1998, hal. 424-427

[7] H. Casanova et al, "Parallel Algorithms". Chapman & Hall, 2008.

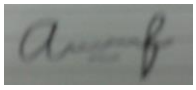
[8] Ir. Rinaldi Munir, MT, Diktat kuliah IF2153 Matematika Diskrit (Edisi Keempat), Teknik Informatika ITB, 2003.C.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 29 April 2010

ttd



Arief Rahman (13511020)