Algoritma Pengurutan Data

Kompleksitas dan Penerapannya

Salvian Reynaldi (13511007)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
salvianreynaldi@students.itb.ac.id

Algoritma-algoritma pengurutan data dalam kajian ilmu informatika, seringkali digunakan untuk menyelesaikan berbagai macam persoalan informatika. Dengan semakin bervariasinya permasalahan-permasalahan tadi, efisiensi dari algoritma-algoritma pengurutan menjadi amat penting, karena algoritma pengurutan biasanya merupakan sebagian saja dari solusi total sebuah permasalahan di dunia informatika, sehingga alangkah baiknya jika proses pengurutan data tidak menjadi batu sandungan dalam proses penyelesaian masalah secara keseluruhan. Algoritma-algoritma pengurutan data yang ada saat ini dapat dikelompokkan menjadi 2, yaitu menurut berbasis perbandingan atau tidaknya, atau juga menurut stabil atau tidaknya.

Kata kunci: Algoritma pengurutan data, berbasis perbandingan, efisiensi, kompleksitas algoritma, stabil.

I. PENDAHULUAN

Dalam ilmu komputer, algoritma pengurutan adalah algoritma yang mengatur ulang urutan elemen-elemen dari kumpulan data agar data tersebut sesuai dengan urutan tertentu. Urutan yang paling sering digunakan adalah alfabetik-numerik, dan juga urutan kamus (leksikografi). Pengurutan data ini biasanya berhubungan erat dengan beberapa proses lainnya, terutama penukaran, pencarian, dan penggabungan. Karena itu, sangat dibutuhkan algoritma pencarian yang sangkil dan mangkus dalam mengatasi masalah yang dihadapi.

Algoritma-algoritma pengurutan data ini sendiri masihlah merupakan algoritma yang terus dikembangkan dan diteliti untuk mendapatkan algoritma yang lebih mangkus lagi, karena data yang ada di dunia ilmu komputer ini juga semakin beragam jenisnya dan semakin kompleks juga permasalahannya.

II. TEORI DASAR

A. Algoritma dan Pengurutan Data

"An algorithmis a finite sequence of precise instructions for performing a computation or for solving a problem"
(Rosen, 2012)

"Algoritma adalah urutan logis langkah-langkah penyelesaian masalah secara sistematis"
(Munir, 2012)

Dari kedua definisi algoritma diatas terdapat beberapa kesamaan, yaitu urutan (*sequence*), perintah atau langkah (*instruction*), dan penyelesaian masalah (*problem solving*). Jadi dapat kembali diartikan bahwa algoritma adalah suatu urutan perintah-perintah yang dilakukan untuk menyelesaikan permasalahan, dalam hal ini permasalahan yang komputasional. Algoritma ini memiliki beberapa sifat/properti penting yang umum, diantaranya:
masukan (*input*), keluaran (o*utput*), kejelasan langkah (*definiteness*), ketepatan (*correctness*), keterbatasan lingkup (*finiteness*), kesangkilan (*effectiveness*), dan berlaku untuk semua masukan yang sesuai (*generality*).
(Rosen, 2012)

Tidak semua masalah mampu diselesaikan oleh algoritma-algoritma yang sudah ada sekarang. Salah satu contoh problem yang belum terselesaikan sampai sekarang adalah ***Halting Problem***.
Salah satu properti dari algoritma adalah kesangkilan. Oleh karenanya, dalam algoritma pengurutan data, kesangkilan juga amatlah penting. Namun hal ini bukan berarti bahwa kemangkusan dari algoritma juga tidak penting, hal ini juga penting, terutama untuk data-data yang banyak jumlahnya. Karena itulah, algoritma-algoritma pengurutan yang ada, dapat dibagi ke dalam 2 kelompok, yaitu pengurutan berbasis perbandingan (*comparison-based*) sorting, dan yang tidak berbasiskan perbandingan. Biasanya algoritma yang berbasiskan perbandingan bisa digunakan untuk segala macam data, baik berupa beragam tipe data, ataupun beragam variasi dari data yang bertipe sama. Beberapa contoh dari algoritma ini adalah pengurutan gelembung (*bubble sort*), pengurutan dengan sisipan (*insertion sort*), dan pengurutan dengan pemilihan (*selection sort*). Sementara itu, algoritma-algoritma yang tidak berbasiskan perbandingan biasanya mengharuskan data yang akan diurutkan untuk memiliki properti-properti tertentu, dengan keuntungan sampai sekarang sebagian besar algoritma ini bekerja dengan lebih cepat dibandingkan yang berbasis perbandingan. Salah satu contohnya algoritma yang tidak berbasis perbandingan ini adalah pengurutan berhitung (*counting sort*).

B. Kompleksitas Algoritma

"Algoritma yang bagus adalah algoritma yang mangkus"
(Munir, 2012)

Ya, algoritma yang mangkus akan memiliki tingkat kompleksitas yang lebih rendah (walau mungkin tidak lebih sederhana algoritmanya) sehingga disebut algoritma ini lebih baik. Disebut lebih baik karena berarti algoritma ini akan menghasilkan keluaran lebih cepat untuk masukan yang sama dibandingkan algoritma yang lebih tinggi kompleksitasnya. Hal ini terjadi karena setiap algoritma akan membutuhkan waktu dan ruang.

"Kebutuhan waktu suatu algoritma biasanya dihitung dalam satuan detik, mikrodetik, dan sebagainya, sedangkan ruang memori yang digunakannya dapat dihitung dalam satuan byte atau kilobyte"
(Munir, 2012)

Rata-rata komputer saat ini dapat mengerjakan kira-kira 10^8-10^9 proses sederhana dalam satu detik. Beberapa proses sederhana ini diantaranya operasi dasar matematik (penambahan, pengurangan, perkalian, dan pembagian), inisiasi dan pemberian nilai, perbandingan boolean, serta pemrosesan masukan dan keluaran (*input*/*output*) program. Kebutuhan ruang dinilai kurang *critical* sekarang untuk permasalahan algoritma, karena perkembangan ruang penyimpanan komputer di saat ini sangat cepat, sehingga masalah ruang dapat lebih mudah diatasi untuk kebanyakan kasus, dan masalah masalah ini juga lebih berkaitan dengan struktur data yang dipakai algoritma. Hal ini tidak berarti perkembangan kecepatan memroses komputer juga tidak berkembang, namun menggunakan algoritma yang lebih mangkus akan lebih berpengaruh daripada menggunakan prosesor yang lebih mangkus (setidaknya untuk sekarang ini). Misal, dengan perkembangan kecepatan perangkat keras prosessor saat ini yang sudah, misalnya, 10 kali lebih cepat dibanding 15 tahun yang lalu, hanya akan membuat jumlah proses yang dapat dikerjakan komputer per detiknya naik menjadi 10 kalinya juga. Namun dengan mengimplementasikan algoritma yang lebih mangkus, akan didapatkan faktor pengali yang lebih besar dari 10, bisa 100, 1000, bahkan lebih. Misalkan dengan input n sebesar 10.000 algoritma dengan kompleksitas $O(n^{2)}$ akan melakukan kira-kira 10^8 proses. Namun dengan algoritma berkompleksitas $O(n \log_2 n)$, program akan melakukan kira-kira hanya 133.000 proses saja. Sekitar 700 kali lebih sedikit daripada algoritma pertama bukan?

III. ALGORITMA-ALGORITMA PENGURUTAN DATA

Algoritma pengurutan data, seperti sudah disebutkan di subbab sebelumnya, dapat dibagi menjadi 2 kelompok, berbasis perbandingan dan tidak berbasis perbandingan. Algoritma pengurutan data juga dapat dikelompokkan menurut parameter yang lain, menjadi *stable sort*, dan *unstable sort*. Algoritma yang stabil akan mempertahankan urutan input jika ada elemen-elemen yang memiliki kesamaan pada bagian yang dibandingkan, sementara pada algoritma yang tidak stabil urutan elemennya dapat berubah.

A. Algoritma Berbasis Perbandingan

Algoritma berbasis perbandingan membaca setiap elemen dan melakukan perbandingan 'lebih besar dari', 'lebih kecil dari', atau 'sama dengan'. Beberapa contoh algoritma ini diantaranya:

a. PENGURUTAN GELEMBUNG (*bubble sort*)

"Bubble sort adalah algoritma pengurutan yang populer tetapi tidak mangkus. Cara kerjanya adalah dengan menukar semua elemen bertetangga yang tidak sesuai urutan"
Diterjemahkan dari:
(Leiserson, Cormen, Rivest, & Stein, 2009)

Konon pengurutan ini dinamakan *bubble* karena sifatnya seperti gelembung, akan mengapungkan nilai yang terkecil ke atas lebih dahulu. Berikut adalah salah satu contoh pseudocode dari *bubble sort* yang mengurutkan isi sebuah larik bernama data, yang berisi N elemen, secara menaik:

```
for i = 1 to N-1 do
  ditukar = false
  for j = N downto i+1 do
    if data[j-1]>data[j] then
      swap(data[j-1],data[j])
      ditukar=true;
  if not(ditukar) then break;
```

algoritma ini akan membandingkan setiap elemen yang bersebelahan yang ada di larik, dan mengapungkan elemen terkecil mulai dari yang paling akhir letaknya di larik, sehingga pada akhirnya memaksa elemen yang letaknya "lebih kiri" supaya lebih kecil nilainya daripada elemen yang "lebih kanan". Jadi kompleksitas waktu bubble sort adalah linear untuk kasus terbaiknya (terjadi ketika data sudah/hampir terurut), karena penukaran yang terjadi akan lebih sedikit, dan pengulangan akan diterminasi lebih dini (di*break*). Sementara untuk kasus rata-rata dan kasus terburuknya, kompleksitasnya adalah

$$T(N) = N - 1 + N - 2 + \cdots + 1 = \frac{N(N-1)}{2} = O(N^2).$$

Bubble Sort disini termasuk algoritma yang *stable*.

b. PENGURUTAN SISIPAN (*insertion sort*)

Pengurutan dengan sisipan bekerja dengan cara menyisipkan setiap elemen yang baru diterima, ke dalam kelompok elemen yang sudah terurut, di posisi yang tepat, agar kelompok elemen tadi tetap terurut. Hal ini mirip saat melakukan pengurutan kartu ketika

kita bermain kartu (misalnya pada permainan *Big Two* alias *Cap Sa*) dimana kita memasukkan kartu tertentu agar ke kelompok kartu yang sudah terurut, agar kelompok kartu tadi tetap terurut. Berikut potongan kodenya:

```
for j = 2 to N do
  key = data[j]
  i = j-1
  while (i>0) and (data[i]>key) do
    data[i+1]=data[i]
    i= i-1
  data[i+1]=key
```

kompeksitas dari algoritma ini adalah linear juga untuk data yang hampir/sudah terurut (karena tidak ada pergeseran elemen yang terjadi / tidak masuk ke pengulangan while), sementara untuk kasus rata-rata dan kasus terburuknya, kompleksitasnya menjadi:

$$T(N) = 1 + 2 + \cdots + N - 1 = \frac{N(N-1)}{2} = O(N^2).$$

Insertion Sort disini termasuk algoritma yang *stable*.

c. PENGURUTAN DENGAN SELEKSI (*selection sort*)

Pengurutan yang paling manusiawi dan mudah dimengerti mungkin adalah pengurutan dengan pemilihan ini. Algoritma ini sangat naif, karena ia memilih elemen yang terkecil/terbesar dari seluruh elemen di larik yang belum terurut, untuk menempati posisi yang telah disesuaikan agar terurut. Contoh algoritma pengurutan ini adalah sebagai berikut:

```
for i = 1 to N do
  k = i
  for j = i+1 to N do
    if data[j] < data[k] then
      k = j
  swap(data[i],data[k])
```

jumlah operasi penukaran yang terjadi akan linear, sementara jumlah operasi if akan dijalankan adalah kuadratik. Pada Selection sort, apakah data sudah/hampir/tidak terurut pada awalnya tidak mengubah kompleksitas selection, yang tertulis seperti di bawah ini, karena tetap akan dicari sebuah elemen dari seluruh data yang tersisa untuk menempati setiap posisi terurut di larik.

$$T(N) = N - 1 + N - 2 + \cdots + 1 = \frac{N(N-1)}{2} = O(N^2).$$

Selection Sort disini termasuk algoritma yang *unstable*.

Ketiga algoritma pengurutan di atas waktu pelaksanaannya adalah kuadratik, sehingga dengan asumsi komputer dapat mengeksekusi 1 milyar proses per detik, jika diberi batas 1 detik untuk mengurutkan, Nilai N maksimal hanya boleh sekitar 30.000 saja, karena selebihnya akan membutuhkan waktu lebi. Bayangkan betapa lamanya mengurutkan nama seluruh penduduk Kota Bandung dengan ketiga algoritma tadi: asumsikan penduduk kota Bandung saat ini 3 juta jiwa, maka ketiga algoritma akan melakukan kira-kira 9 triliun proses, yang akan memakan waktu kira-kira 9000 detik (dua setengah jam).

Kuadratik bukanlah waktu pelaksanaan terlama dari algoritma sorting yang ada. *Bogo Sort* {O(N.N!)}, dan *Stooge Sort* {$O(N^{2.7})$} adalah contoh algoritma sorting yang lebih lambat dari algoritma pengurutan $O(N^2)$ sehingga sering disebut sebagai algoritma pengurutan candaan, dan tidak akan dibahas disini.

Kedua algoritma pengurutan berbasis perbandingan terakhir yang akan dibahas disini menggunakan sebuah paradigma pemecahan masalah yang bernama pecah-dan-kuasai (*divide & conquer*). Paradigma ini membagi sebuah persoalan menjadi beberapa persoalan yang lebih kecil (misalnya menjadi 2 bagian), lalu menyelesaikannya satu per satu untuk kemudian digabungkan kembali untuk membentuk solusi yang sebenarnya.

---

d. PENGURUTAN CEPAT (*quick sort*)

Algoritma Quicksort membagi data ke dalam beberapa partisi (contohnya disini 2), dengan menggunakan bantuan data pembatas (pivot), sehingga isi partisi kiri adalah elemen data yang lebih kecil dari pivot, dan isi dari partisi kanan adalah elemen data yang lebih besar dari pivot. Hal ini terus dilakukan sampai isi partisi hanya 1 elemen, sehingga partisi itu tidak perlu lagi diurutkan.

```
procedure Quicksort(a,b:integer)
left,right,pivot,tmp:integer
begin
  if b>a then
    left=a
    right=b
    pivot=data[⌊(a+b)/2⌋]
    while left≤right do
      while data[left]<pivot do
        left=left+1
      while data[right]>pivot do
        right=right-1
      if left≤right then
        swap(data[left],data[right])
        left=left+1
        right=right-1
  quicksort(a,right)
  quicksort(left,b)
end;
Quicksort(1,N) {pemanggilan pertama}
```

Ada 2 hal yang dilakukan algoritma diatas:

- Membagi larik menjadi 2 partisi
- Membandingkan lalu menukar nilai

Kompleksitas dari *quick sort* untuk kasus terburuk adalah melakukan N proses, dan melakukan

pemanggilan rekursif *quick sort* 2 kali yang masing-masing larik ukurannya 1 dan (N-1) elemen, sehingga: $T(N) = N + T(N-1)$; $T(N \leq 1) = 0$. Maka dengan melakukan *bottom-up*:
T(1) = 0
T(2) = N + 0 = N
T(3) = N + N = 2N
dst. didapat

$$T(N) = N(N-1) = O(N^2).$$

Kasus terburuk ini dapat terjadi karena pivot terpilih gagal membagi data menjadi 2 partisi sama besar. Misalnya dengan data awal berisi 8 elemen, pembagiannya menghasilkan 1-7 elemen, 1-1-6, 1-1-1-5, dst. Sementara untuk kasus rata-rata dan terbaik, *quick sort* akan melakukan N proses dan pemanggilan rekursif *quick sort* 2 kali dengan masing-masing besar partisi adalah N/2, sehingga $T(N) = N + 2T\left(\left\lfloor\frac{N}{2}\right\rfloor\right)$; $T(N \leq 1) = 0$.
Dengan menggunakan *Master Theorem* (Leiserson, Cormen, Rivest, & Stein, 2009), $T(N) = a\,T(N/b) + F(N)$, untuk kompleksitas quicksort ini $F(N) = N; a = b = 2$; lalu F(N) memenuhi bentuk 2 teorema ini, yaitu $F(N) = N = \Theta\left(N^{\log_b a}\right) = \Theta(N)$, maka

$$T(N) = \Theta\left(N^{\log_b a} \log N\right) = \Theta(N \log N).$$

Ada beberapa varian *quick sort* yang telah dioptimasi, misalnya *quick sort* dengan 3 partisi; quicksort dengan median data sebagai nilai pivot; *intro sort*, sorting yang dipakai di STL beberapa bahasa pemrograman, merupakan pengurutan *hybrid*, yang melakukan *quick sort* sampai level kedalaman tertentu, dan menggantinya dengan *heap sort* jika level tadi sudah tercapai.
sehingga sorting ini memiliki T(N)=O(N log N) untuk semua kasus.

Quick Sort disini termasuk algoritma yang *unstable.*

e. PENGURUTAN MENGGABUNG (*merge sort*)

Merge sort, sejenis dengan *quick sort*, juga membagi data yang ada ke dalam beberapa (misalnya 2) partisi. Bedanya di merge sort tidak dilakukan proses perbandingan elemen dengan pivot, jadi antara data dibagi 2 secara naif atau menggunakan pembatas pembagian yang terdefinisi. Berikut potongan kode salah satu varian *Merge Sort*:

```
MERGE(A, p, q, r)
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

(Leiserson, Cormen, Rivest, & Stein, 2009)

Pada Merge Sort, umumnya terjadi 2 proses utama:

- Pemartisian
- Penggabungan

Kompleksitas *Merge Sort* tidak jauh berbeda dengan *Quick Sort* cara mendapatkannya, yaitu karena untuk semua kasus berlaku $T(N) = N + 2T\left(\left\lfloor\frac{N}{2}\right\rfloor\right)$; $T(N \leq 1) = 0$ (data selalu dibagi menjadi 2 partisi yang sama besar), maka dengan *Master Theorem* (Leiserson, Cormen, Rivest, & Stein, 2009) didapatkan

$$T(N) = \Theta\left(N^{\log_b a} \log N\right) = \Theta(N \log N).$$

B. Algoritma Tidak Berbasis Perbandingan

Algoritma pengurutan yang tidak berbasis perbandingan biasanya hanya dapat mengurutkan integer saja. Keuntungannya adalah, algoritma-algoritma ini kompleksitasnya lebih kecil dari O(N log N), namun biasanya memliliki syarat-syarat lainnya juga selain hanya dapat mengurutkan interger.

a. PENGURUTAN BERHITUNG (*counting sort*)

Count Sort menggunakan sistem yang mirip dengan sistem penurusan / *tally* (misalnya saat *voting* presiden oleh MPR sebelum zaman reformasi, atau proses pemilihan ketua murid di SD, SMP, SMA), yaitu mencatat satu-satu input yang masuk, untuk dimasukkan ke "tempatnya" masing-masing. Berikut ini salah satu varian algoritma *Counting Sort*, yang mengurutkan N angka yang diinputkan (*range* angka input adalah 1-1000):

```
Data: array[1..1000] of integer;

{input dan mengurutkan}
for i = 1 to N do
  input(x)
  data[x] = data[x] + 1
```

```
{output}
for i = 1 to 1000 do
  for j = 1 to data[i] do
    output(i)
```

Simple sekali algoritma *counting sort* ini. Dia memasukkan sesuatu yang diinput ke tempatnya, misalnya dalam hal ini inputnya adalah integer, dan integer akan dimasukkan ke tempatnya masing-masing (misalnya input 1, dimasukkan ke "kotak" 1 pada larik). Kelemahan dari *Counting Sort* adalah pada variasi data, yang harus sedikit, misalnya, *range* data yang diinput hanya akan 1 sampai 100, atau data hanyalah bilangan-bilangan 2^n dengan batas $0 \leq n \leq 100$. Hal ini dikarenakan jika data terlalu bervariasi, akan perlu dibuat tempat baru untuk menampung data varian itu, sehingga akan membuat kompleksitas ruangnya menjadi tidak mangkus. Algoritma *Counting Sort* juga dapat langsung digunakan untuk membuang data-data duplikat yang terdapat di dalam larik.
Kompleksitas waktu algoritma *Counting Sort* pada contoh di atas adalah $\boldsymbol{T(N) = 2N = O(N)}$ untuk proses input dan pengurutannya, dan $\boldsymbol{T(N) = x_1 + x_2 + \cdots + x_{1000} = N = O(N)}$ pada proses outputnya.

Counting Sort disini termasuk algoritma yang *stable.*

b. PENGURUTAN RADIX (*radix sort*)

Pengurutan radix melakukan pengurutan untuk tiap-tiap subelemen (digit untuk integer, atau karakter untuk string) secara berurutan. Secara sederhana, beginilah algoritmanya (larik A, dan banyak subelemen maksimal d):

```
Radix(A,d)
For i = 1 to d
  Urutkan A berdasarkan digit ke-i
```

Diterjemahkan dari:
(Leiserson, Cormen, Rivest, & Stein, 2009)

Ada 2 teknik untuk menggunakan radix sort, yaitu teknik subelemen-paling-insignifikan (*Least Significant Character* / *Least Significant Digit*) dan teknik subelemen-paling-signifikan (*Most Significant Character* / *Most Significant Digit*).

Teknik Subelemen-Paling-Insignifikan
Misalkan array A berisi angka-angka berikut ini: 3, 9, 1, 2, 19, 28, 91, 827, 918, 736
Pada contoh kali ini digunakan struktur data antrian (*queue*). Dan dibuat larik yang berisi 10 elemen bertipe antrian.
Pertama-tama lakukan *enque* angka-angka tadi masing-masing ke dalam queue yang sesuai menurut satuannya dan sesuai dengan urutan inputnya, sehingga larik yang elemennya antrian akan menjadi seperti ini:

0:
1: 1, 91
2: 2
3: 3
4:
5:
6: 736
7: 827
8: 28, 918
9: 9, 19

Masukkan lagi angka-angka dari *queue* ke dalam larik A, dengan melakukan *deque* dimulai dari digit 0. Larik A kini menjadi: 1, 91, 2, 3, 736, 827, 28, 918, 9, 19.
Lakukan lagi pengurutan untuk digit puluhan:

0: 1, 2, 3, 9
1: 918, 19
2: 827, 28
3: 736,
4,5,6,7,8:
9: 91

Kembalikan lagi ke larik A, lakukan lagi pengurtan untuk digit ratusan, kembalikan lagi ke larik A, hasil akhirnya tentu menjadi: 1, 2, 3, 9, 19, 28, 91, 736, 827, 918. Karena dalam melakukan *enque* selain sesuai digitnya, urutan masukan juga diperhatikan, maka pengurutan radix dengan teknik SPI pada dasarnya akan menjadi pengurutan yang stabil.

Teknik Subelemen-Paling-Signifikan
Misalkan array A berisi angka-angka berikut ini: 1, 2, 3, 9, 19, 28, 91, 827, 736, 918
Kembali digunakan struktur data antrian (*queue*). Dan dibuat larik yang berisi 10 elemen bertipe antrian.
Pertama-tama lakukan *enque* angka-angka tadi masing-masing ke dalam queue yang sesuai menurut digit ratusan dan sesuai dengan urutan inputnya, sehingga larik yang elemennya antrian akan menjadi seperti ini:

0: 1, 91, 2, 3, 28, 9, 19
1,2,3,4,5,6:
7: 736
8: 827
9: 918

Biarkan antrian yang hanya berisi 1 elemen, lanjutkan pengurutan pada antrian dengan 2 atau lebih elemen, kali ini dibuat antrian baru, dan dilakukan *enque* sesuai digit puluhan dan urutan inputnya.

0: 1, 2, 3, 9
1: 19
2: 28
9: 91

Biarkan antrian yang hanya berisi 1 elemen, lanjutkan pengurutan pada antrian dengan 2 atau lebih elemen, kali ini dibuat antrian baru, dan dilakukan *enque* sesuai digit satuan dan urutan inputnya, hasilnya akan sama. Lalu lakukan deque ke larik A. Larik A menjadi terurut.

Karena pada pengurutan radix dengan teknik SPS ini nilai-nilai tidak dikembalikan lagi ke larik A setiap kali selesai melakukan *enque*, maka pengurutan ini akan menjadi tidak stabil (urutan input tidak dipertahankan), namun sebenarnya ada varian pengurutan radix dengan teknik SPS yang stabil.

Jadi kompleksitas *Radix Sort* baik dengan teknik SPI maupun SPS akan sama, dan karena akan diurutkan beberapa subelemen dari tiap data, maka akan ada 2 parameter: $\boldsymbol{T(N,k) = kN = O(kN)}$, dengan N adalah banyak data dan k adalah banyak subelemen maksimal dari sebuah elemen yang ada di larik.

c. PENGURUTAN BUCKET (*bucket sort*)

Bucket sort pada dasarnya merupakan generalisasi dari *counting sort*, yaitu meyediakan "wadah-wadah" untuk mengelompokkan setiap elemen data.
Misalkan array A berisi angka-angka berikut ini: 1, 2, 3, 9, 19, 28, 91, 827, 736, 918. Dibuat misalnya 3 *bucket* yang masing-masing ketentuannya:

Bucket 1: 0-9
Bucket 2: 10-99
Bucket 3: 100-999

Maka *bucket* tadi masing-masing diisi dengan angka dari larik A yang sesuai. Bucket 1 akan berisi 1, 2, 3, 9. Bucket 2 isinya 19, 28, 81. Bucket 3 diisi dengan 827, 736, 918.
Hal ini dilakukan secara rekursif, yaitu *bucket* 1 dibagi lagi menjadi beberapa *bucket*, yang memiliki syarat-syarat tertentu lagi (yang juga harus sesuai syarat *bucket* 1 tentunya), sampai setiap bucket hanya berisi 1 jenis elemen saja. Setelah itu tentu tinggal angka-angka di *bucket* dimasukkan lagi ke larik A, dimulai dengan *bucket* terkiri, misalnya bucket 1.1.1.1, 1.1.1.2, dst...

Dengan rekursi ini, seperti layaknya *Worst Case* pada *Quick Sort*, bahkan lebih parah, kompleksitasnya bisa menjadi bahkan melebihi $T(N) = O(N^2)$. Oleh karena itu, algoritma pengurutan menggunakan *bucket* jarang digunakan sendirian, melainkan digabung dengan algoritma lainnya, sehingga data hanya sekali saja dibagi menjadi beberapa *bucket*. Proses pengurutan selanjutnya dilakukan oleh algoritma pengurutan yang lain, biasanya *insertion sort* (karena *insertion sort* berjalan cepat untuk data yang setiap elemennya posisinya sudah "hampir benar", dan proses agar setiap elemen posisinya hampir benar inilah yang dilakukan oleh *bucket sort*), lalu hasil pengurutan di tiap *bucket* digabungkan secara langsung, ujung kanan *bucket* kiri dengan ujung kiri *bucket* kanan).
Namun mungkin juga dipasangkan dengan algoritma pengurutan lain, contohnya pada *radix sort* yang sudah dijelaskan sebelumnya, sebenarnya telah dilakukan pemecahan data ke dalam 10 *bucket* (10 buah antrian). Algoritma *hybrid* antara *bucket sort* dengan pengurutan lainnya akan memiliki kompleksitas yang bergantung pada algoritma apa yang dipasangkan dengan *bucket sort*. Namun diperkirakan T(N) tidak akan jauh dari N, karena itu masih diekspektasikan O(N). Stabil atau tidaknya pengurutan *bucket* versi *hybrid* tergantung pada algoritma pengurutan apa yang digunakan bersama dengan *bucket sort*.

IV. PENERAPAN ALGORITMA PENGURUTAN DATA

a. PENCARIAN NAMA

Setelah pengurutan selesai dilakukan, kerap kali proses yang berikutnya dilakukan untuk memecahkan masalah adalah pencarian. Misalnya diminta mencari apakah sebuah nama ada di dalam data yang diinputkan, yaitu data nama-nama pemakai di situs jejaring sosial *Facebook*. Dengan menggunakan teknik pencarian linier, kompleksitas algoritma pencarian akan menjadi $T(N) = O(N)$. Jika diasumsikan jumlah pengguna facebook ada 1 milyar orang, maka pencarian nama pengguna sederhana dengan teknik pencarian linier akan memakan waktu kira-kira lebih dari satu detik (kompleksitas membandingkan 2 buah string adalah $T(k) = O(k)$ dengan k adalah panjang karakter string yang dibandingkan). Namun jika digunakan teknik *Binary Search*, kompleksitas untuk melakukan pencarian dapat dikurangi menjadi hanya $T(N) = O(log\ N)$ saja, sehingga hanya akan dilakukan maksimal sekitar 30 kali proses pencarian saja, yang akan jauh di bawah 1 detik, untuk mengetahui hasilnya. Tetapi, teknik pencarian biner mensyaratkan data sudah terurut, sehingga di sinilah peran algoritma pengurutan dibutuhkan. Disini dapat digunakan algoritma pengurutan yang kompleksitasnya $T(N) = O(N)\ atau\ T(N) = O(N \log N)$, seperti *quick sort*, *radix sort*, dsb. Mengapa tidak menggunakan algoritma yang kompleksitasnya $T(N) = O(N^2)$? Karena jika digunakan algoritma yang demikian, *bottleneck* dari program secara keseluruhan bukanlah terletak di proses pencariannya lagi, melainkan di proses pengurutaannya, dengan kata lain lebih baik untuk menggunakan teknik pencarian linier saja sekalian.

b. PERSOALAN PEMROGRAMAN KOMPETITIF

Contoh lainnya adalah problem UVa 10327 – Flip Sort, dengan deskripsi sebagai berikut: diberikan N (N≤1000) buah bilangan, tentukan jumlah minimal penukaran pasangan bilangan yang harus dilakukan agar angka-angka tadi menjadi terurut menaik. Misal, untuk membuat 3 bilangan (2 3 1) menjadi (1 2 3), diperlukan minimal 2 penukaran, misalnya 1 ditukar dengan 2, lalu 2 ditukar dengan 3.

Dalam persoalan ini sebenarnya yang harus dilakukan adalah simulasi proses pengurutan berbasis komparasi, karena algoritma ini juga melakukan proses perbandingan dan penukaran, dan proses itu sudah dilakukan minimal secara natural. Karena N lebih kecil dari 1000, dan time limit yang diberikan adalah 3 detik, maka algoritma pengurutan $T(N) = O(N^2)$ masih bisa dilakukan dalam batas waktu.

Dengan menggunakan potongan kode berikut ini (dalam bahasa C++)

```
int n,ans;
int main () {
  while(scanf("%d",&n)!=EOF)  {
    int data[n+2];
    ans=0;
    for (int i=0;i<n;i++)
      scanf("%d",&data[i]);
    for (int i=0;i<n-1;i++)
      for (int j=n-1;j>=i+1;j--)
        if (data[j-1]>data[j])  {
          swap(data[j-1],data[j]);
          ans++;
        }
    printf("Minimum exchange operations : %d\n",ans);
  }
  return 0;
}
```

diperoleh *verdict accepted* dengan *run time* sebesar 0,092s.

10327 Flip Sort	Accepted	C++	0.092	2012-12-17 12:21:02

c. PENGURUTAN MULTIPRIORITAS

Misalkan setiap elemen di larik itu tidak hanya memiliki 1 nilai, misalkan saja ada 2 nilai (seperti tipe data *point*). Untuk melakukan pengurutan berdasarkan X dulu baru Y secara menaik, maka dengan teknik pengurutan bergelembung, tinggal mengubah bagian yang akan menukar elemen, yaitu bagian kondisional dari

```
If (data[j-1] > data[j]) then
```

menjadi

```
If (P[j-1].X>P[j].X) or
   ((P[j-1].X=P[j].X) and
     (P[j-1].Y>P[j].Y)) then
```

saja (misalkan P adalah sebuah larik yang elemennya bertipe data *Point*, yang telah dibuat sebelumnya). Sementara untuk teknik pengurutan cepat , maka bagian yang diubah adalah bagian yang membagi data ke dalam partisi, misalnya dari

```
while data[left]<pivot do
        left=left+1
      while data[right]>pivot do
        right=right-1
```

menjadi

```
while (P[left].X<pivX) or
   ((P[left].X=pivX) and
    (P[left].Y<PivY)) do
      left=left+1
while P[right].X>pivX
   ((P[right].X=pivX) and
    (P[right].Y>PivY)) do
      right=right-1
```

Perlu diingat pivot juga harus memiliki X dan Y (atau sama tipenya dengan tipe elemen yang akan diurutkan).

Namun misalkan dibatasi X dan Y dari setiap titik adalah bilangan bulat [0-999] saja, maka sebenarnya ada cara lain selain membuat tipe data baru yaitu *point*. Misalkan pertama dimasukkan bilangan X dan Y dari masing-masing titik. Maka yang dimasukkan ke dalam array bukanlah X,Y, namun misalkan diminta mengurutkan menurut X dulu baru Y, maka yang dimasukkan adalah A=1000X + Y, lalu cukup lakukan algoritma pengurutan seperti dengan larik bilangan bulat biasa. Hal ini akan benar karena X yang telah dikali 1000 akan menjadi lebih signifikan nilainya dibanding dengan Y, jadi mengurutkan bilangan 1000X + Y akan berarti mengurutkan berdasarkan X dahulu. Setelah proses pengurutan dilakukan, kita juga tetap dapat memperoleh nilai X dan Y dari masing-masing angka A, yaitu:

$X = \left\lfloor \frac{A}{1000} \right\rfloor ; Y = (A - 1000X) = A \; mod \; 1000.$

d. PEMILU / PEMILUKADA

Dalam Pemilihan Umum / Pemilihan Umum Kepala Daerah, tentu akan ada calon yang menang dan yang kalah, dengan kata lain harus ada hasilnya. Misalkan ada 5 calon yang dapat dipilih untuk menempati jabatan di Indonesia, yang jumlah penduduknya kira-kira 250 juta jiwa. Data ini disimpan dengan menggunakan larik biasa, misalkan, dan nilai larik ke-i adalah nomor calon yang dipilih oleh penduduk ke-i, tentu hal ini akan tidak mangkus kompleksitas ruangnya, perlu dibuat larik yang ukurannya 250 juta elemen. Lalu untuk mengetahui pemenangnya, larik harus diurutkan dengan menggunakan algoritma yang ada, algoritma O(N log N) pun tetap dirasa kurang mangkus untuk data yang sebesar ini.

Hal ini berbeda tentunya jika digunakan struktur data larik biasa juga. Dengan 5 elemen saja, dan elemen ke-i isinya adalah berapa banyak pemilih calon ke-i (seperti teknik pengurutan berhitung). Dengan ini selain hanya membutuhkan larik 5 elemen, proses pengurutan juga berjalan dengan sendirinya (alamiah *counting sort*).

V. KESIMPULAN

Tidak ada algoritma pengurutan data yang terbaik untuk semua kasus. Setiap algoritma pengurutan yang ada memiliki kelebihan dan kekurangannya masing-masing. Untuk Algoritma pengurutan yang berbasis perbandingan, biasanya algoritma berkompleksitas O(N log N) akan bekerja lebih cepat dibandingkan $O(N^2)$, namun hal itu pun tidak berlaku untuk semua kasus (acak, hampir terurut, terurut terbalik, atau banyak elemen sama). Sementara algoritma yang tidak berbasis perbandingan memang cepat, namun kemampuannya terbatas (hanya berfungsi jika range kecil, variansi sedikit, atau untuk tipe data tertentu saja).

Hal ini mengharuskan kita untuk pandai-pandai dalam memilih algoritma mana yang akan digunakan untuk memecahkan masalah yang kita hadapi. Semua algoritma pengurutan baik, tapi belum tentu sangkil; kalau sangkil juga belum tentu mangkus untuk menyelesaikan permasalahan yang ada jika kita salah menggunakannya.

REFERENSI

Leiserson, C. E., Cormen, T. H., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). London: McGraw-Hill.

Munir, R. (2012). *Matematika Diskrit* (edisi ke-5). Bandung: Informatika Bandung.

Rosen, K. H. (2012). *Discrete Mathematics and Its Applications* (7th ed.). New York: McGraw-Hill.

Universidad de Valladolid Online Judge (http://uva.onlinejudge.org/), diakses tanggal 17 Desember 2012 pukul 19:33

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, 18 Desember 2012

ttd

Salvian Reynaldi