Kemangkusan Algoritma Pada Beberapa Variasi Quick Sort

Alifa Nurani Putri (13511074)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13511074@std.stei.itb.ac.id

***Abstrak*—Algoritma pengurutan (sort) merupakan salah satu algoritma dasar terpenting yang sering digunakan dalam penyelesaian masalah. Hingga saat ini, sudah banyak sekali algoritma pengurutan yang dibentuk dengan keunggulan dan kelemanhan masing-masing. Algoritma Quick Sort merupakan salah satu algoritma pengurutan yang dianggap mangkus. Namun, algoritma ini sangat bergantung pada data yang akan diurutkan dan algoritma yang digunakan. Makalah ini akan membahas pengaruh jenis data yang akan diurutkan, serta pengaruh pemilihan impelementasi algoritma yang digunakan ditinjau dari kompleksitasnya.**

***Kata kunci*—Kompleksitas, Sorting, Data, Algoritma.**

I. PENDAHULUAN

Dalam konteks pemrograman, untuk menyelesaikan sebuah masalah tidak cukup dengan menemukan algoritma yang hasil/ penyelesaian masalah-nya terbukti benar. Sebuah program yang baik akan menggunakan algoritma-algoritma yang efektif, efisien, dan tepat sasaran. Salah satu parameter untuk memilih algoritma dengan kualifikasi seperti itu adalah kemangkusan algoritma. Kemangkusan algoritma menggambarkan sebagaimana algoritma itu cepat dieksekusi dan sedikit memakan ruang memori.

Salah satu algoritma dasar yang kerap dipakai pada berbagai penyelesaian masalah adalah algoritma pengurutan. Hingga kini, sudah banyak sekali jenis dari algotima pengurutan, dari yang sangat sederhana hingga kompleks. Pengembangan algoritma pengurutan tersebut didasarkan bahwa performa algoritma pengurutan akan sangat mempengaruhi performa program. Sehingga sangat penting untuk mampu memilih algoritma pengurutan yang mangkus, agar program berjalan efisien.

Dari beberapa jenis algoritma pengurutan, dari yang cukup sederhana hingga cukup kompleks, algoritma Quick Sort menjadi salah satu algoritma yang dianggap mangkus. Namun, kemangkusan algoritma Quick Sort kerap kali terlalu bergantung kepada persebaran data yang akan diurutkan dan implementasi algoritma yang dipilih. Data yang sudah terurut membesar, data yang terurut mengecil, data yang teracak, dsb memiliki suatu strategi terbaik dalam pengurutan Quick Sort. Sehingga cukup susah untuk menentukan jenis Quick Sort mana yang lebih mangkus untuk digunakan, saat kita belum tahu bagaimana persebaran data yang akan diurutkan.

Tujuan makalah ini adalah untuk membahas beberapa jenis algoritma Quick Sort, dan dibandingkan kompleksitasnya/kemangkusannya terhadap jenis persebaran data yang mungkin, dan juga akan dibahas hal-hal yang harus dipilih saat pengimplementasian algoritma, yang akan mempengaruhi kemangkusan Quick Sort.

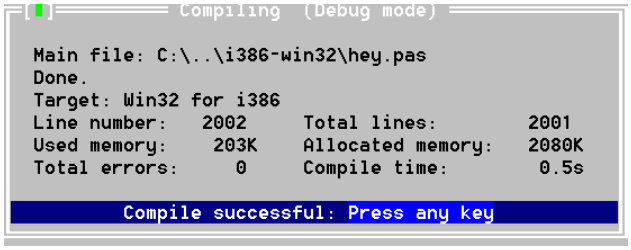
II. TEORI DASAR

A. Algoritma Mangkus

Dalam menyelesaikan masalah dengan algoritma, seringkali ditemukan beberapa macam pilihan cara, yang sama-sama menghasilkan nilai yang benar, atau sesuai dengan hasil yang diinginkan. Lantas, bagaimana cara memilih algoritma yang tepat ? Algoritma yang tepat dapat ditentukan dengan membandingkan kemangkusan dari masing-masing cara yang sudah ditemukan.

Algoritma yang mangkus adalah algoritma yang efisien dari segi ruang dan waktu, artinya algoritma tersebut menggunakan seminim mungkin waktu, dan sesedikit mungkin ruang memori.

Menentukan kemangkusan Algoritma secara umun terbagi menjadi dua, yaitu dengan kompleksitas algoritma dan dengan secara langsung menghitung waktu dan ruang memori saat kompilasi. Dari kedua cara tersebut, cara yang terbaik adalah dengan perhitungan kompleksitas algoritma. Kompleksitas algoritma lebih tepat dalam menentukan kemangkusan algoritma, karena perhitungannya benar-benar hanya dipengaruhi oleh algoritma tersebut. Sedangkan perhitungan langsung pada saat kompilasi dipengaruhi hal luar, seperti *compiler* dan komputer yang digunakan. Setiap komuter memiliki arsitektur yang berbeda-beda sehingga bahasa mesinnya pun berbeda-beda. *Compiler* yang digunakan saat kompilasi juga menentukan waktu yang digunakan, perbedaan *Compiler* dapat menyebabkan perbedaan waktu operasi program.



Gambar 2.1.1 Ruang Memori dan Waktu Kompilasi

B. Kompleksitas Algoritma

Untuk Mengukur Kemangkusan Algoritma secara independen, diperlukan perhitungan kompleksitas algoritma. Kompleksitas Algoritma terbagi menjadi dua, yaitu Kompleksitas waktu, dan Kompleksitas ruang. Kompleksitas tersebut ditentukan dengan besaran input yang dilambangkan dengan *n*.

1. Kompleksitas Waktu

Menghitung kompleksitas waktu untuk sebuah algoritma pada hakekatya, menghitung berapa kali tahapan komputasi dilakukan dari awal algoritma hingga algoritma selesai diproses. Pemodelan yang dilakukan, adalah dengan menganggap nilai kompleksitas waktu ini sebagai nilai fungsi yang dengan dilambangkan *T(n)* dari varibel masukan *n*.

Dalam penggunaannya, seringkali teradi penyederhanaan perhitungan kompleksitas waktu. Perhitungan yang seharusnya melibatkan seluruh tahap komputasi dari awal sampai akhir, sering disederhanakan menjadi perhitungan komputasi utama, yang biasanya dilakukan secara berulang.

Berikut adalah contoh pehitungan kompleksitas waktu pada potongan algoritma berikut :

```
1   input(n)
2   F <- 1
3   i <- 0
4
5   while (i < n) do
6       i <- i + 1;
7       F <- i*F
8   output (F)
```

Gambar 2.2.1.1 Potongan Algoritma Faktorial

Perhitungan Kompleksitas Algoritma tersebut didasarkan pada operasi mendasar faktorial yaitu perkalian (F ← i * F) . Operasi tersebut dilakukan sebanyak *n* kali. Maka $T(n) = n$.

Pada penggunaan lebih lanjut, nilai *T(n)* juga bisa dipengaruhi faktor lain, maka dari itu, kompleksitas waktu dibagi menjadi tiga jenis, yaitu :

1. $T_{max}(n)$: waktu untuk kasus terburuk
2. $T_{min}(n)$: waktu untuk kasus terbaik
3. $T_{avg}(n)$: waktu rata-rata

Dalam Menghitung Kompleksitas Waktu , dikenal notasi O-Besar,

a) Notasi O-Besar

Definisi $T(n) = O(f(n))$ (dibaca "*T(n)* adalah *O(f(n))*" *yang artinya T(n)* berorde paling besar *f(n)* bila terdapat konstanta C dan n_0 sedemikian hingga

$$T(n) <= C\ (f(n))$$

Untuk $n >= n_0$

a) Teorema O-Besar

Berikut adalah teorema O-Besar

Misalkan T_1 (n) = $O(f(n))$ dan $T_2(n)$ $(O(g(n))$, maka

(1) $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$

(2) $T_1(n)\ T_2(n) = O(f(n))\ O(g(n)) = O(f(n)\ g(n))$

(3) O(cf(n)) = O(f(n)), c adalah konstanta

(4) f(n) = O(f(n))

2. Kompleksitas Ruang

Kompleksitas Ruang menghitung ruang memori yang digunakan dalam algoritma. Kompleksitas ruang juga dimodelkan sebagai fungsi dengan lambang *S(n)* dengan input *n*. Hal yang mempengaruhi besarnya nilai kompleksitas ruang yaitu :

- Bagian Konstan, yang terdiri dari memori yang dibutuhkan untuk sourcecode, variabel aggregate, konstanta, dll. Nilai konstanta tidak terlalu dipikirkan dalama perhitungan kompleksitas algoritma.
- Bagian Variabel (Karakteristik Instans), yaitu bagian yang kebutuhan memorinya bergantung pada penyelesaian permasalahan.

Apabila dirumuskan, nilai kompleksitas ruang adalah sebagai berikut :

$$S(n) = c + Sp$$

c : konstanta

Sp : Karakteristik instans

Berikut adalah contoh perhitungan kompleksitas ruang dari suatu potongan algoritma.

```
1   Algorithm jumlah(a,n)
2   //a adalah array dengan ukuran n
3   {
4       s:=0.0;
5       for i=1 to n do
6       s:=s+a[i];
7       return s;
8   }
9
```

Gambar 2.2.2.1 Potongan Algoritma Jumlah

Pada potongan algoritma tersebut, karakteristik instans ditentukan oleh *n*. Variabel a harus mampu menampung float sejumlah *n* elemen, maka, ruang memori yang dibutuhkan adalah n word. Sedangkan variabel s, i , dan n masing-masing memutuhkan 1 word. Sehingga karakteristik Instans dari potongan algoritma tersebut adalah :

$Sp(n) <= (n + 3)$

C. Algoritma Rekursif

Suatu pendefinisian disebut rekursif bila dalam pendefinisiannya menyebutkan dirinya sendiri. Dalam kajian algoritma, rekursif merupakan suatu alternatif cara untuk menyelesaikan permasalahan. Rekursif biasa digunakan dalam suatu subprogram. Untuk Membangun suatu definisi rekursif, terdapat dua poin utama yaitu :

- Basis
  Basis merupakan titik dasar pendefinisian. Pada nilai basis yang ditentukan (biasanya 0 atau 1) dilakukan pendefinisian langsung tanpa rekursif.
- Rekurens
  Rekurens adalah pendefinisian nilai selain basis, pendefinisian dilakukan secara rekursif, dan akan terdefinisi jika nilai tersebut bisa mencapai nilai basis.

Berikut adalah contoh algoritma rekursif :

```
1  function factorial (N : integer) -> integer
2
3      if (N = 0) then {Basis 0}
4          -> 0
5      else  {rekurens}
6          -> N * factorial(N-1)
7
```

Gambar 2.4.1 Fungsi faktorial rekursif

Pada contoh tersebut, pada baris ke 6 fungsi tersebut memanggil dirinya sendiri, inilah pendefinisian rekursif.

D. Kompleksitas Rekursif

Kompleksitas Suatu Algoritma Rekursif memiliki perbedaan tipe dengan algoritma non-rekursif. Untuk Kompleksitas Waktu, funsgi *S(n)* terbagi menjadi dua kondisi, yaitu kondisi basis dan rekurens, sama halnya dengan badan algoritma rekursif itu sendiri. Misal untuk algoritma fungsi faktorial rekursif yang terdapat pada gambar di atas, Kompleksitas waktunya adalah sebagai berikut :

$T(n) = 1 + T(n\text{-}1)$
$= 1 + 1 + T(n\text{-}2) = 2 + T(n\text{-}2)$
$= 2 + 1 + T(n\text{-}3) = 3 + T(n\text{-}3)$
$= \ldots$
$= n + T(0)$

Jadi $T(n) = n$

[6]

$$T(n) = \begin{cases} 0 & ,n = 0 \\ T(n-1) + 1 & ,n > 0 \end{cases}$$

Gambar 2.3.1 Kompleksitas Waktu Faktorial Rekursif

D. Algoritma Pengurutan

Algoritma pengurutan adalah salah satu algoritma klasik yang sangat berguna dalam pengelolaan data. Secara umum, algoritma pengurutan dibagai menjadi 2 bagian yaitu :

- Pengurutan Internal
  Pengurutan data yang tesimpan di media internal komputer.
- Pengurutan Eksternal
  Pengurutan data yang disimpan dalam memori sekunder.

Bahasan makalah ini dibatasi pada wilayah pengurutan internal saja. Algoritma pengurutan sangat membutuhkan kemangkusan yang tinggi, terlebih lagi ketika diharuskan menangani kasus besar yang melibatkan banyka data. Performa dari algoritma pengurutan yang dipilih akan sangat menentukan kompleksitas waktu sistem. Dengan begitu, perkembangan algoritma penurutan terus berjalan, seiring dengan kebutuhan akan kemangkusan yang menjadi tujuan pada pendesain program.

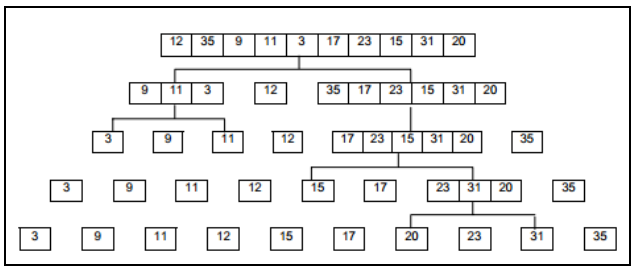
E. Quick Sort

Metode Pengurutan Quick atau yang juga dikenal sebagai metode partisi menggunakan teknik penukaran dua elemen dengan jarak yang cukup besar.

Teknis penukaran dengan metode Quick secara umum adalah sebagai berikut :

- Pemilihan data tertentu sebagai pivot. Pivot dipilih sebagai pembatas nilai, bahwa akan diatur selanjutnya agar nilai elemen-elemen di sebelah kiri pivot bernilai kurang dari pivot. Dan nilai-nilai di sebelah kanan pivot akan lebih besar dari pivot.
- Pivot diletakkan pada posisi ke-j sedemikian hingga data ke 1 sampai j-1 lebih kecil dari pivot. Dan pada data ke j+1 sampai N bernilai lebih besar dari pivot. Cara pengelompokan ini dilakukan dengan penukaran.
- Setelah proses ini berlangsung, data terbagi menjadi 3, bagian < pivot, pivot. Dan bagian > pivot. Setiap bagian melakukan proses poin I dan II, pada bagiannya masing masing. Proses ini berlangsung berulang sampai, tidak ada lagi data yang masih berkelompok.

Berikut adalah ilustrasi metode Quick Sort.



Gambar 2.4.1 Ilustrasi Quick Sort[7]

Berikut bagaimana perjalanan algoritma QuickSort pada gambar di atas, dari proses awal, hingga data terurut secara lengkap :

- Kondisi awal

12	25	9	11	3	17	23	15	31	20

- Penentuan Pivot
  Pengurutan pada ilustrasi tersebut mengambil nilai pivot = 12, yaitu data ke-1.
- Proses selanjutnya adalah membuat data-data yang berada disebelah kiri pivot adalah data-data yang bernilai lebih kecil dari pivot, sedangkan data-data di sebelah kanan pivot adalah data-data yang lebih besar dari pivot. Caranya adalah dengan melakukan traversal, lalu pengecekan satu, satu dan langsung dipindahkan jika tidak sesuai. Sehingga posisi menjadi

9	11	3	12	35	17	23	15	31	20

- Sehingga muncullah tiga kelompok dalam data tersebut yaitu :
  - Kelompok kiri < pivot : 9,11,3
  - Pivot
  - Kelompok kanan > pivot : 35,17, 23, 15,31, 20
- Proses selanjutnya merupakan bagian rekurens, yaitu melakukan proses-proses di atas terhadap kelompok kiri, hingga setiap elemen tidak ada lagi yang berkelompok. Setelah itu proses akan berganiti ke kelompok kanan, hingga data pada kelomplok kanan juga tidak ada yang masih berkelompok.

III. PEMBAHASAN

A. Pengaruh Variasi Persebaran Data terhadap Kemangkusan Algoritma Quick Sort

Persebaran data yang akan diurut sangat memengaruhi kompleksitas/kemangkusan waktu algoritma, karena dengan metoda pengurutan Quick Sort ini jenis persebaran data menentukan apakah posisi pivot efektif atau tidak.

Secara umum, persebaran data yang akan diurutkan adalah sebagai berikut :

1.Terurut Membesar

Data yang disajikan terurut membesar merupakan data yang dianggap 'bonus' dalam segala jenis pengurutan. Karena tidak penukaranan yang harus dilakukan, karena sudah sesuai dengan hasil yang diinginkan.

2. Terurut Mengecil

Data terurut mengecil membutuhkan pratinjau yang cukup detail dalam menentukan lokasi pivot. Masalahnya adalah ketika pivot yang dipilih adalah elemen pertama/terakhir, maka terjadi pertukaran besar-besaran, dan akhirnya kelompok data kiri (< pivot) tidak terbentuk, sehingga data menajdi tidak ideal, proses traversal dilakukan banyak walau rekurensnya hanya terjadi pada satu sisi saja.

3. Acak

Data bersifat acak adalah data yang ideal, karena nilai lokasi pivot tidak akan terlalu mempengaruhi kompleksitas, karena persebaran data yang cukup rata.

B. Pengaruh Variasi Posisi Pivot terhadap Kompleksitas Algoritma Quick Sort

Pada metoda pengurutan Quick Sort ini nilai pivot akan mempengaruhi proses penukaran yang akan dilakukan, jika pivot berada pada posisi yang tidak tepat tentu kompleksitas waktunya akan lebih besar, karena terjadi lebih banyak operasi komputasi yang dilakukan.

Berikut adalah beberapa cara pemilihan pivot yang mungkin dilakukan, juga bagaimana posisi pivot tersebut mempengaruhi kemagkusan algoritma :

1. Elemen Pertama

Jika pivot diletakkan di elemen pertama, kemangkusan tidak tentu, seperti yang telah dijelaskan sebelumnya bahwa ternyata jika data dalam posisi terurut mengecil. Keunggulan dari posisi pivot ini adalah, tidak perlu algoritma untuk mencari posisi pivot, karena posisi elemen pertama adalah posisi eksak, yang tidak bergantung pada data.

2. Elemen Terakhir

Penempatan pada elemen terakhir akan berpengaruh sama dengan penempatan pada elemen pertama.

3. Elemen Tengah

Jenis Penempatan ini merupaan salah satu jenis penempatan pivot yang cukup umum. Terutama untuk data yang tersebar acak. Pada posisi ini perbandingan kanan-kiri menjadi berimbang. Keunggulannya adalah algoritma pencarian posisi pivotnya sangat mudah yaitu :

(Posisi data terkiri + Posisi data terkanan) div 2

4. Pivot Acak

Penempatan posisi pivot jenis ini baik, karena tidak bergantung pada persebaran data, kecenderungannya sering membuat posisi cukup berimbang. Namun, dilihat dari kompleksitas waktu, algoritma menjadi tidak mangkus, karena setiap pencarian pivot, yang notabene-nya secara rekursif terulang, akan mengeksekusi fungsi pembangkit random terlebih dahulu. Fungsi pembangkit random sendiri memiliki kompleksitas algoritma yang bermacam-macam.

5. Privot Median

Jika ditinjau hanya dari letak posisinya saja, pivot jenis ini adalah yang terbaik. Bayangkan saja, nilai tengah merupakan nilai pembagi rata. Sehingga, komposisi yang terbentuk akan bersifat seimabang. Proses penukaran selanjutnya akan berjalan efektif. Namun, cara ini juga mempunyai kelemahan yang sama dengan privot acak. Yaitu dibutuhkan fungsi pencari nilai median. Fungsi pencari nilai median mempunyai kompleksitas waktu

tersendiri yang akan memperbesar kompleksitas waktu pengurutan karena, pencarian nilai median ini akan dieksekusi pada setiap loop utama.

C. Pengaruh ke-rekursif-an terhadap Kompleksitas Algoritma Quick Sort

Pada bagian ini akan dijelaskan algoritma Quick-Sort dalam bentuk Pseudo Code. Untuk algoritma ini, pemilihan pivot pada posisi tengah data. Dalam implementasinya, algoritma Quick-Sort ini bisa dimplementasikan dalam 2 pendekatan, yaitu pendekatan rekursif dan non-rekursif.

1. Quick Sort Rekursif

Algoritma Quick Sort dengan pendekatan rekursif tidak membutuhkan struktur data khusus, karena proses pengurutan setiap kelompok akan berjalan secara rekursif. Algoritma Quick Sort dengan pendekatan rekursif, terbagi menjadi dua bagian, yaitu quicksort dan partisi. Partisi ,merupakan bagian pengelompokkan, sedangakan quicksort sendiri adalah bagian yang melakukan proses rekurdifnya.

Berikut adalah pseudocode dari algoritma Quick Sort secara rekursif :

```
procedure quicksort (input/output a :
array [1..n] of integer, input i,j  :
integer)

KAMUS

k : integer

ALGORITMA

if (i < j) then
   partisi(a,i,j,k)
   quicksort(a,i,k)
   quicksort(a,k+1,j)

-----------------------------------------

procedure partisi (input/output a :
array[1..n] of integer, input i,j :
integer, output q : integer)

KAMUS

pivot, temp : integer

ALGORITMA

pivot ← a [(i+j) div 2] {jika pivot =
elemen tengah}
p ← i
q ← j
repeat
   while (a[p] < pivot) do
        p ← p + 1
   while (a[q] > pivot) do
        q ← q - 1
   if (p <= q) then
        temp  ← a[p]
        a[p]  ← a[q]
        a[q]  ← temp
   p ← p + 1
   q ← q - 1
until (p > q)
```

Gambar 3.3.1.1 Algoritma QuickSort Non-Rekursif

- Kompleksitas Ruang

Untuk algoritma rekursif, jika ditinjau dari penggunaan struktur data, kompleksitas ruang terbialang unggul, karena dalam pendekaran rekursif, tidak digunakan struktur data tersendiri (misal : stack), sehinggan tidak perlu pembentukan tipe bentukan atau ADT yang akan memakan ruang memori. Namun, dalam implementasinya, ke-rekursifan justru membutuhkan tambahan memori tersendiri karena setiap loop, terjadi pemanggilan dirinya sendiri, dan di dalam dirinya sendiri membutuhkan alokasi memori. Sehingga, semakin membesar jumlah data, kompleksitas ruang suatu algoritma rekursif akan membesar pula.

- Kompleksitas Waktu

Kompleksitas waktu Quick Sort terbagi menjadi 3 bagian yang bergantung pada persebaran data.

1. Kasus Terbaik

Kasus terbaik akan terjadi untuk proprorsi seimbang, artinya kelompok kiri dan kelompok kanan mempunyai jumlah anggota kelompok yang sama. Biasanya kasus ini terjadi jika memakai pivot berupa nilai median, tanpa menghiraukan kompleksitas waktu pencarian nilai median. Kompleksitas waktu untuk kasus terbaik dalah :

$$T(n) = \begin{cases} a & ,n = 1 \\ 2\,T\left(\frac{n}{2}\right) + cn & ,n > 1 \end{cases}$$

Gambar 3.3.1.2 Kompleksitas Kasus Terbaik

Penyelesaian :

$T(n) = 2\,T(n/2) + cn$
$= 2\,(2\,T(n/4) + cn/2\,) + cn = 4T(n/4) + 2\,cn$
$= 4\,(2\,T(n/8) + cn/4) + 2\,cn = 8T(n/8) + 3cn$
$= \ldots$
$= 2^k\,T(n/2^k) + kcn$

Kondisi ini berhenti saat $n = 1$

$$n / 2^k = 1 \rightarrow k = {}^2\log n$$

$T(n) = n\,T(1) + cn\,{}^2\log n$
$= na + cn\,{}^2\log n$
$= O\,(n\,{}^2\log n)$

Sehingga

$$T_{min}(n) = 2T(n/2) + cn = na + cn\,{}^2\log n = O\,(n\,{}^2\log n)$$

2. Kasus Terburuk

Kasus terburuk terjadi jika semua elemen kecuali privot masuk di salah satu kemompok , dan kelompok yang

satunya kosong, dan data belum terurut membesar. Dengan begitu akan nanyak pengulangan yang dilakukan. Dari segi penempatan pivot, kasus terburuk ini terjadi ketika pivot merupakan nilai maksimum atau minimum dari data, dan itu terus berlangusng setiap partisi dilakukan. Semakin lama waktu yang dibutuhkan.

$$T(n) = \begin{cases} a & ,n = 1 \\ T(n-1) + cn & ,n > 1 \end{cases}$$

Gambar 3.3.1.3 Kompleksitas Kasus Terburuk

Penyelesaian :

$T(n) = cn + T(n - 1)$
$= cn + \{ c(n -1) + T(n - 2)\}$
$= cn + c(n - 1) + \{c(n - 2) + T(n - 3)\}$
$= \ldots$
$= cn + c(n - 1) + c(n - 2) + c(n -3) + \ldots + c2 + T(1)$
$= c\{ n + (n-1) + (n-2) + (n-3) + \ldots + 2\} + a$
$= c\{ (n - 1)(n - 2)/2\} + a$
$= cn^2/2 + cn/2 + (a - c)$
$= O(n^2)$

sehingga

$$T_{max}(n) = T(n-1) + cn \log n = O(n^2)$$

3. Kasus Rata-Rata

Kasus ini terjadi bila pivot tidak terpilih secara acak dari data, dimana tidak diketahui bagaimana perbandingan nilai pivot dengan nilai data lainnya, Kasus Rata-rata ini memiliki kompleksitas yang sama dengan kasus terbaik, yaitu kasus dengan pemilihan pivot berupa nilai median dari data.

$$T_{avg}(n) = O(n^2 \log n)$$

A. Quick Sort Non-Rekursif

Algoritma Quick Sort dengan pendekatan non-rekursif membutuhkan dua buah stack untuk menyimpan posisi batas-batas kelompok kiri dan kanan. Stack pada hal ini direpresentasikan sebagai array.

```
procedure QuickSortNonRekursif (input N :
integer, input/output Data : array [1..N] of
integer )

KAMUS

    constant M : 100;
    type tump : < Kiri : integer,
                  Kanan : integer
                >
    Tumpukan : array [1..M] of tump
    i, j, L, R, x, ujung = integer;


ALGORITMA

    i ← 1; j ← 1; L ← 1;  R ← 1; x ← 1; ujung ← 1
    Tumpukan[1].Kiri ← 0
    Tumpukan[1].Kanan ← N-1


    while (ujung ≠ 0) do
          L ← Tumpukan[ujung].Kiri
          R ← Tumpukan[ujung].Kanan
          ujung ← ujung-1;
          while (R > L) do
                i ← L
                j ← R
                x ← Data[(L + R) / 2]
                while (i <= j) do
                    while (Data[i] < x)
                          i ← i + 1
                    while (Data[j] > x)
                          j ← j - 1
                    if (i <= j)
                        Tukar(Data[i], Data[j])
                        I ← i + 1
                        J ← j - 1

                if (L < i)
                    ujung ← ujung + 1
                    Tumpukan[ujung].Kiri ← i;
                    Tumpukan[ujung].Kanan ← R;
                R = j;


procedure Tukar (input/output a,b : integer)

KAMUS

    temp : integer

ALGORITMA

    temp ← a
    a ←  b
```

Gambar 3.1.2.1 Algoritma QuickSort Non-Rekursif

- *Kompleksitas Waktu*

Kompleksitas waktu dari suatu algoritma rekursif, secara umum tidak berbeda jika dibandingkan dengan penyelesaian dengan non-rekursif. Penjelasannya adalah sebagai berikut :

Jika Kompleksitas waktu yang dihitung merupakan operasi utama, yaitu operasi perbandingan dan operasi pertukaran, maka berapa kali terjadinya pertukaran dan perbandingan itu tidak dipengaruhi bagaimana algoritma itu diproses. Jumlah perbandingan dan pertukaran itu bergantung pada hal-hal yang disebutkan di atas, yaitu persebaran data, dan posisi pivot. Sehingga untuk kasus seperti ini, rekursif atau tidaknya algoritma bukanlah suatu poin penting untuk menentukan kemangkusan algoritma ditinjau dari kompleksitas waktu yang digunakan. Dalam implementasi langsung, dan prakteknya, mungkin tetap ada selisih waktu yang dibutuhkan untuk eksekusi algoritma, namun tidak signifikan sehingga nilai *O(n)* dari algoritma quick sort rekursif akan sama dengan algoritma quick sort non-rekursif.

- *Kompleksitas Ruang*

Untuk prosedur yang dirancang dengan pendekatan non-rekursif, dibutuhkan suatu struktur data tersendiri. Biasanya dan seperti contoh diatas dibutuhkan struktur data yang merepresentasikan stack, namun pada pseudo code diatas, stack tersebut diimplementasikan dalam array, hanya dalam proses didalamnya, bekerja layaknya stack, sehingga tidak terlebih dahulu dubuat ADT stack-nya. Pseudo Code diatas membutuhkan tambahan ruang memori saat pembentukan tipe bentukan, dibanding dengan pendekatan rekursif yang tidak perlu membangun suatu tipe bentukan. Namun, sama dengan penjelasan pada bagian rekursif, ruang memori untuk pendekatan non-rekursif dalam implementasinya akan lebih unggul, terutama jika melibatkan jumlah data yang banyak, karena tidak ada penambahan khusus/besar ruang memori setiap loop data.

IV. KESIMPULAN

Kemangkusan algoritma Quick Sort bergantung pada beberapa poin yaitu :

1. Persebaran Data, yang berpengaruh terhadap kemangkusan pada setiap posisi pivot.
2. Posisi Pivot, yang berpegaruh pada perhitungan kompleksitas waktu.
3. Ke-rekursif-an yang berpengaruh pada ruang memori.
4. Jumlah Data, yang berpengaruh terhadap ruang memori jika diambil pendekatan rekursif.

Sehingga untuk menemukan jenis algoritma Quick Sort yans sesuai, jika mungkin, analisis terlebih dahulu karakteristik data yang akan diproses.

REFERENSI

[1] Munir, Rinaldi, Draft Diktat Kuliah IF2091 Struktur Diskrit, Bandung: Program Studi Teknik Informatika- Sekolah Teknik Elektro dan Informatika- Institut Teknologi Bandung, 2008

[2] Liem, Inggriani, Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural), Bandung : Kelompok Kehalian Rekayasa Perangkat Lunak & Data (DSE) - STEI –ITB, 2007

[3] Munir, Rinaldi, Slide Bahan Kuliah IF 2091 Struktur Diskrit-Kompleksitas Algoritma

[4] Liem, Inggriani, Draft Diktat Struktur Data, Bandung : Program Studi Teknik Informatika- Institut Teknologi Badnung, 2008

[5] http://id.shvoong.com/internet-and-technologies/software/2063318-kompleksitas-algoritma-ruang-dan-waktu/, 15 Deseember 2012, 22:17 WIB

[6] http://www.ittelkom.ac.id/staf/zka/Materi%20Desain%20Analisis%20Algoritma/M06Kompleksitas%20Waktu%20untuk%20Algoritma%20Rekursif.pdf, 16 Desember 2012, 00:03 WIB

[7] http://lecturer.eepis-its.edu/~entin/Struktur%20Data%20&%20Algoritma/buku/Data%20Structure%20-%20Bab%206.pdf, 16 Desember 2012, 17:32 WIB

[8] http://dc303.4shared.com/doc/gBQtIPrr/preview.html, 17 Dsember 2012, 22:22 WIB

[9] Slide Algoritma *Divide and Conquer*, Bagian 1, Teknik Informatika, Universitas Ahmad Dahlan

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

Alifa Nurani Putri
13511074