Big O Notation

Notasi Big O digunakan untuk mengekspresikan kompleksitas dari sebuah algoritma. Maksudnya kompleksitas disini adalah, bagaimanakah efisiensi algoritma tersebut terhadap besarnya sebuah data input. Dengan kata lain, bagaimana kompleksitas waktu atau pemakaian ruang memory bertambah seiring dengan bertambahnya jumlah input yang diberikan. Sering kali ini dikaitkan dengan scenario terburuk (*worst case scenario*) sebuah algoritma dalam menyelesaikan suatu problem.

Mengapa notasi big O ini kita pelajari?

Sebab ini penting ketika kita ingin memilih struktur data maupun algoritma yang optimal dalam menyelesaikan suatu permasalahan. Sebentar, tadi disebutkan notasi big O ini hubungannya dengan algoritma. Kok muncul juga data struktur? nanti akan kita lihat.

Notasi big O sendiri aslinya notasi dalam bidang matematika untuk analisa asimptotik. Notasi big O ini sudah dipakai secara luas di bidang ilmu computer dan informatika. Agar lebih mudah dipahami, kita tidak akan membahas notasi Big O ini secara matematis, tapi langsung saja dengan contoh kode. Namun saya merekomendasikan untuk mencari tau lebih jauh bagi yang tertarik untuk mempelajari lebih dalam tentang analisa asimptotik :)

```
class BigONotation {
    private:
        // private field
        int *array;  // pointer untuk menyimpan array
        int arrayLen=0; // banyaknya elemen yang disimpan dalam array

        // variabel helper untuk mengukur waktu eksekusi
        clock_t timeStart;
        clock_t timeEnd;

        // private method
        void swapIndex(int, int); //untuk menukar nilai dari dua posisi
elemen

    public:
        // constructor & destructor
        BigONotation(int);
        ~BigONotation();

        // helper public methods
        int size(); // mengembalikan jumlah elemen dalam array
        void printArray(); // output semua elemen array

        // main methods
        void setValueAtIndex(int, int);
        bool linearSearch(int);
        void bubbleSort();
        bool binarySearch(int);
        void stdSort();
```

```
};
```

Sebelumnya saya jelaskan sedikit, kode yg kita gunakan ini terdiri dari sebuah class BigONotation. Di dalamnya terdapat private member array bertipe integer dan variabel arrayLen yg menyimpan banyaknya elemen dalam array. Di constructor diberikan parameter jumlah elemen dalam array, lalu langsung di inisialisasi dengan random number. Jadi setiap kita panggil new BigONotation(size), instance class kita langsung memiliki array integer sebanyak size dan tiap elemen bernilai random antara 0 ~ MAX_INT.

```
BigONotation::BigONotation(int num){
    array = new int[num];

    for(int i=0;i<num;i++){
        array[arrayLen++] = rand();
    }
}
```

Method yang utama yang akan kita bahas secara detil ada 5 yaitu:

```
void setValueAtIndex(int, int);
bool linearSearch(int);
void bubbleSort();
bool binarySearch(int);
void stdSort();
```

Pemberian Nilai pada Array

Ok, pertama kita akan membahas metthod setValueAtIndex. Dari namanya saja sudah kebayang ya fungsinya. Yaitu untuk men-set nilai elemen array di index tertentu. Langsung kita lihat kodenya.

```
void BigONotation::setValueAtIndex(int val, int pos){
    array[pos] = val;
}
```

Coba kita pikirkan kompleksitas dari fungsi diatas. Bagi yang sudah familiar dengan array, tentu langsung bisa melihat bahwa kompleksitas atau waktu yang diperlukan untuk memberikan nilai ke salah satu elemen dalam array adalah selalu konstan. Maksudnya adalah, kompleksitas tidak berubah seiring dengan bertambah banyaknya input, yang dalam hal ini adalah jumlah elemen dalam array. Karena konstan, maka kita nyatakan kompleksitasnya dalam notasi O(1).

Jadi jelas ya, kalau ada yang menyebut sebuah algoritma memilik kompleksitas O(1), artinya berapapun jumlah input, performa dari algoritma tersebut tidak terpengaruh. Tapi algoritma dengan kompleksitas O(1) ini biasanya hanya terbatas pada operasi-operasi yang sederhana, seperti assignment, atau aritmatika (+,-,/,* dst), komparasi/pembandingan dan sebagainya.

Ok, sampai sini kita sudah membahas tentang O(1). Mari kita ke method berikutnya, yaitu linear search.

Linier Search

```
bool BigONotation::linearSearch(int x){
    bool found = false;

    timeStart = clock();
    for(int i=0; i< arrayLen; i++){
        if(x == array[i]){
            cout << "[Linear search] Found at index: " << i << endl;
            found = true;
            break;
        }
    }
    timeEnd = clock();

    if(!found){
        cout << "[Linear search] Not found in array" << endl;
    }

    cout  << "[Linear search] Time elapsed: "
          << (double)(timeEnd - timeStart)*1000/CLOCKS_PER_SEC
          << "ms"  << endl;

    return found;
}
```

Saya jelaskan sedikit, method linear search ini adalah fungsi untuk mencari sebuah nilai di dalam array. Karena bertipe bool, kalau nilai yg dicari ada maka return *value*-nya true, jika tidak maka false.

Bagian utama dari metode ini adalah yang ini:

```
    for(int i=0; i< arrayLen; i++){
        if(x == array[i]){
            cout << "[Linear search] Found at index: " << i << endl;
            found = true;
            break;
        }
    }
```

Kita lihat di bagian ini kita menggunakan for loop untuk memeriksa satu persatu apakah nilai yang ingin kita cari (x) sama dengan elemen array di posisi i.

```
if(x == array[i])
```

Sebagainama saya sebutkan tadi, ini adalah operasi pembandingan, yang kompleksitasnya adalah? ya, O(1) atau konstan.

Tapi ada yang merasa aneh ngga? di cuplikan kode diatas ada perintah break, atau keluar dari loop jika nilai yang dicari sudah ditemukan. Jadi belum tentu loop nya berulang sebanyak jumlah elemen dalam array. Lalu apakah kompleksitasnya masih bisa disebut O(n) walaupun loop nya belum tentu berulang sebanyak n kali? Ingat bahwa notasi big O ini fungsinya adalah untuk mengekspresikan kompleksitas dari worst case sebuah algoritma. Jadi walaupun belum tentu pengulangannya sebanyak

n, tetapi kita sebut O(n), sebab kompleksitasnya bertambah secara linear seiring bertambahnya data input.

Bagaimana kalau kasusnya begini, ada algoritma yang diimplementasikan seperti ini:

```
for (int i=0;i<n;i++){
    int a = i *2; // O(1)
    int b = a-5 // O(1)
}
```

Kita lihat di dalam loop sebanyak n ada dua operasi konstan. Bagaimanakah kita mengekspresikan kompleksitas algoritma diatas? karena ada dua operasi konstant sebanyak n, maka totalnya adalah 2n. Jadi kompleksitasnya O(2n) , betul atau salah? Tidak benar, jadi ingat2 saja, bahwa konstanta akan selalu dihilangkan dalam menulis notasi big O. jadi 2n,3n,4n, ... tetap ditulis n.

Bubble Sort

```
void BigONotation::bubbleSort(){
    timeStart = clock();

    for(int i=0; i<arrayLen;i++){
        for(int j=0;j<arrayLen;j++){
            if(array[j]>array[j+1])
                swapIndex(j, j+1);
        }
    }
    timeEnd = clock();

    cout  << "[Bubble sort] Time elapsed: "
          << (double)(timeEnd - timeStart)*1000/CLOCKS_PER_SEC
          << "ms" << endl;

}
```

Saya jelaskan lagi sedikit, bubble sort ini adalah fungsi untuk mengurutkan data dalam array. Kenapa harus di urut? karena nanti kita akan memperkenalkan metode search yang lebih optimal dari linear search yg kita bahas diatas.

Logikanya mudah, kita cek setiap elemen dari array, kita bandingkan elemen tersebut dengan elemen sesudahnya, misal kita sedang melihat elemen ke-2, kita bandingkan dengan elemen ke-3, ke-3 dan ke-4 dst.

Kalau elemen sesudahnya lebih kecil dari elemen yg sedang kita cek, maka kita melakukan swap atau tukar nilai elemen tersebut dengan elemen sesudahnya. kita ulangi proses ini sebanyak jumlah elemen dalam array, dan kita akan mendapatkan array kita akan menjadi berurutan ascending pada akhirnya. Kenapa algoritma ini bekerja? ada analisa dan bukti secara matematisnya, tapi silakan cari sendiri ya :D

Untuk menukar dua nilaii elemen array saya buat fungsi private swapIndex dalam class BigONotation. Mengenai implementasi swap ini tidak akan saya bahas dulu, tapi kalau tertarik silakan ditanyakan. Tapi mungkin materi ini akan di cover di kulgram yang lain, mudah2an.

okay, kita balik ke bubble sort

```
    for(int i=0; i<arrayLen;i++){
        for(int j=0;j<arrayLen;j++){
            if(array[j]>array[j+1])
                swapIndex(j, j+1);
        }
    }
```

Dari kode diatas kita dapati nested loop, atau loop dalam loop. sebagaimada di linear search, pada loop terdalam ada dua operasi, yaitu komparasi if(array[j]>array[j+1]) dan swap. Komparasi kompleksitasnya O(1).

Bagaimana dengan swap? Kalau kita lihat definisi fungsi swapIndex

```
void BigONotation::swapIndex(int a, int b){
    array[a] = array[a]^array[b];
    array[b] = array[a]^array[b];
    array[a] = array[a]^array[b];
}
```

kita lihat ada 3 operasi aritmatika. ditambah dengan operasi pembandingan di atas, jadi total ada 4 operasi pada inner loop. sehingga kompleksitasnya adalah O(4) benar atau salah? ya, seperti pada kompleksitas linear, semua konstan direduksi menjadi O(1). paham ya?

Sekarang kita lihat loop dalam bubble sort. Outer loop sebanyak n elemen array. demikian juga dengan inner loop. kalau ada n elemen array, maka total iterasinya adalah n*n atau n^2 (n kuadrat). karena ada paling banyak 4 operasi di inner loop, total maksimal operasinya adalah 4n^2, sehingga kompleksitas algoritma ini adalah?

Ya betul O(n^2). ingat, semua konstanta diabaikan. Sampai sini mari kita coba eksekusi programnya, mari kita bandingkan hasilnya antara O(n) dan O(n^2).

```
[Linear search] Not found in array
[Linear search] Time elapsed: 0.041ms
[Linear search] Not found in array
[Linear search] Time elapsed: 3.937ms
[Linear search] Found at index: 357
[Linear search] Time elapsed: 0.028ms
[Linear search] Found at index: 539896
[Linear search] Time elapsed: 2.004ms
[Bubble sort] Time elapsed: 720.292ms
```

kita fokus pada hasil dari Linear search dan Bubble sort. di main function kita buat dua instance class BigONotation, d1 dan d2.

```
BigONotation *d1 = new BigONotation(10000);
BigONotation *d2 = new BigONotation(1000000);
```

Dari hasli eksekusi kita lihat untuk d1 dengan elemen sebanyak 10000, hasil pencarian nilai cuma 0.0 sekian millisec (per seribu detik). cepat sekali ya. Sementara untuk d2 (1000000) elemen atau 100 kali lebih banyak dari d1, hasil pencariannya meningkat jadi 2-4 ms. masih bisa dibilang cepat.

Tapi coba lihat bubble sort untuk d1. langsung melonjak jd 720 ms, hampir satu detik. Kelihatannya masih cepat, tapi coba silakan jalankan bubble sort untuk d2. Di komputer saya, sampai udah males nunggunya :D

Ternyata O(n^2) performanya jauh lebih buruk dari O(n). Jadi sebisa mungkin hindari algoritma dengan kompleksitas O(n^2). apalain n^3, n^4 dan seterusnya.

Binary Search

```
bool BigONotation::binarySearch(int x){
    int start = 0, end = arrayLen-1;
    int searchIteration = 0;
    bool found = false;

    timeStart = clock();
    while(end > start+1){
        ++searchIteration;
        int mid = (start + end)/2;
        if (array[mid] < x){
            start = mid;
        }
        else if(array[mid] > x){
            end = mid;
        }
        else{
            cout  << "[Binary search] Found at iteration "
                  << searchIteration
                  << endl;
            found = true;
            break;
        }
    }
    if(start != end){
        if(array[start] == x){
            cout  << "[Binary search] Found at iteration "
                  << searchIteration
                  << endl;
            found = true;
        }
    }

    if(!found){
        cout  << "[Binary search] Not found in array after iteration "
              << searchIteration << endl;
    }
    timeEnd = clock();

    cout  << "[Binary search] Time elapsed: "
          << (double)(timeEnd - timeStart)/CLOCKS_PER_SEC << "ms" << endl;

    return found;
}
```

Ini sama fungsinya dengan linear search, tapi ada syaratnya, yaitu data harus sudah di sort.

Strateginya adalah, kita pilih satu index dalam array (biasanya yang tengah2), sebut saja index tengah. lalu kita bandingkan value yang mau dicari dengan nilai array pada index tersebut. Kalau nilainya sama, berarti ketemu dan pencarian selesai.

Kalau nilai yg dicari lebih besar dari nilai array di index tengah, dengan memanfaatkan situasi dimana array sudah berurut, maka kita cuma perlu mencari di bagian array setelah index tengah.

Sebaliknya kalau nilai yg dicari lebih kecil, maka kita yakin nilai yang kita cari akan ada di elemen-elemen sebelum index tengah. Misal, ada array 2, 5, 7, 9, 12.

Kita ingin mencari nilai 2. maka pertama-tama kita ambil index tengah yaitu nilai 7. karena 2 lebih kecil dari 7, kita hanya perlu mencari di bagian kiri dari 7. ada 2 yaitu 2 dan 5. kita ambil index tengah 2 (walaupun 2 bukan tengah, tapi karena cuma ada dua nilai, jadi kita harus milih 2 atau 5, dan kita pilih 2). Ternyata nilai index tengah sekarang sama dengan nilai yang kita cari, jadi ketemu dan selesai.

Kita lihat disini karena kita mengambil index tengah dengan membagi dua segmen array pada setiap iterasi, jika ada n elemen array, maka kita dapatkan jumlah iterasinya sebanyak log n.

```
while(end > start+1){
    ++searchIteration;
    int mid = (start + end)/2;
    if (array[mid] < x){
        start = mid;
    }
    else if(array[mid] > x){
        end = mid;
    }
    else{
        cout  << "[Binary search] Found at iteration "
              << searchIteration
              << endl;
        found = true;
        break;
    }
}
```

disini kita punya dua variabel start dan end, di beri nilai awal start = 0, dan end = arrayLen-1. Start dan end adalah index awal dan akhir dari array. Lalu kita masuk ke while loop, sampai kita dapatkan kondisi start == end, atau start + 1 == end

```
while(end > start+1){
```

lalu kita hitung index tengah mid, yaitu nilai tengah antara start dan end.

```
int mid = (start + end)/2;
```

lalu kita cek 3 kasus: array[mid] < x , array[mid] > x, dan array[mid] == x. Untuk array[mid] < x, berarti x pasti berada di antara index mid dan end, sehingga kita perbarui nilai start = mid. Sebaliknya,untuk array[mid] > x, berarti x pasti berada di

antara index start dan mid, sehingga kita perbarui nilai end = mid. Untuk array[mid] == x, berarti sudah ketemu dan selesai.

Diatas tadi kita bahas maksimal iterasi algoritma ini adalah log(n) untuk n adalah jumlah data dalam array, sehingga kita sebut kompleksitasnya O(log n). Anyway, langsung saja kita lihat eksekusinya kita menerapkan binary search terhadap d1. Hasilnya:

```
[Binary search] Found at iteration 12
[Binary search] Time elapsed: 1.8e-05ms
[Binary search] Not found in array after iteration 13
[Binary search] Time elapsed: 6e-06ms
```

Bandingkan dengan liner search.

```
[Linear search] Not found in array
[Linear search] Time elapsed: 0.055ms
...
[Linear search] Found at index: 102436
[Linear search] Time elapsed: 0.496ms
```

kalau di linear search kita dapatkan hasilnya dalam order 0.sekian milisec, untuk binary searc kita dapatkan ordernya 10^-6 sampai dengan 10^-5 (0.00000sekian) millisec. Jauh sekali bedanya, bias 10ribu kali lebih cepat dengan jumlah elemen yang sama.

std::sort

```
void BigONotation::stdSort(){
    timeStart = clock();

    sort(array, array+arrayLen);

    timeEnd = clock();

    cout  << "[Std sort] Time elapsed: "
          << (double)(timeEnd - timeStart)/CLOCKS_PER_SEC << "ms" << endl;
}
```

kalau lihat disini http://en.cppreference.com/w/cpp/algorithm/sort, kompleksitas-nya adalah O(N log N), detil implementasinya mungkin bisa google quicksort algorithm ndak akan saya bahas karena lebih ribet lagi.

Langsung saja kita lihat hasil eksekusi programmya, kita coba jalankan stdSort() untuk d2:

```
[Bubble sort] Time elapsed: 700.726ms
...
[Std sort] Time elapsed: 0.098007ms
```

Dengan jumlah data 100 kali lipat lebih banyak, waktu yang diperlukan untuk sorting 10ribu kali lebih cepat, itulah kehebatan O(n log n) pada dasarnya O(n log n) ~ O(n).

Okeh kita sudah bahas semua notasi big O yang sering muncul kita sudah membahas bagaimana menganalisa sebuah fungsi/algoritma dan menuliskan kompleksitasnya dengan notasi big O.

Lampiran

File Main.cpp:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>

using namespace std;

class BigONotation {
    private:
        // private field
        int *array;  // pointer untuk menyimpan array
        int arrayLen=0; // banyaknya elemen yang disimpan dalam array

        // variabel helper untuk mengukur waktu eksekusi
        clock_t timeStart;
        clock_t timeEnd;

        // private method
        void swapIndex(int, int); //untuk menukar nilai dari dua posisi
elemen

    public:
        // constructor & destructor
        BigONotation(int);
        ~BigONotation();


        // helper public methods
        int size(); // mengembalikan jumlah elemen dalam array
        void printArray(); // output semua elemen array

        // main methods
        void setValueAtIndex(int, int);
        bool linearSearch(int);
        void bubbleSort();
        bool binarySearch(int);
        void stdSort();
};

int main(){
    srand(time(NULL));

    // coding session #1
    BigONotation *d1 = new BigONotation(10000);
    BigONotation *d2 = new BigONotation(1000000);

    d1->linearSearch(5000);
    d2->linearSearch(5000);

    d1->setValueAtIndex(5000, rand()%d1->size());
    d1->setValueAtIndex(5000, rand()%d1->size());
```

```
    d1->setValueAtIndex(5000, rand()%d1->size());

    d2->setValueAtIndex(5000, rand()%d2->size());
    d2->setValueAtIndex(5000, rand()%d2->size());
    d2->setValueAtIndex(5000, rand()%d2->size());

    d1->linearSearch(5000);
    d2->linearSearch(5000);

    d1->bubbleSort();

    d1->binarySearch(5000);
    d1->binarySearch(100);

    d2->stdSort();
    //d3->printArray();


    delete d1;
    delete d2;
    // delete d3;
}

BigONotation::BigONotation(int num){
    array = new int[num];

    for(int i=0;i<num;i++){
        array[arrayLen++] = rand();
    }
}

BigONotation::~BigONotation(){
    delete array;
}

int BigONotation::size(){
    return arrayLen;
}

void BigONotation::swapIndex(int a, int b){
    array[a] = array[a]^array[b];
    array[b] = array[a]^array[b];
    array[a] = array[a]^array[b];
}

void BigONotation::printArray(){
    for(int i=0;i<arrayLen;i++)
        cout << array[i] << " ";
    cout << endl;
}

void BigONotation::setValueAtIndex(int val, int pos){
    array[pos] = val;
}

bool BigONotation::linearSearch(int x){
    bool found = false;

    timeStart = clock();
```

```
    for(int i=0; i< arrayLen; i++){
        if(x == array[i]){
            cout << "[Linear search] Found at index: " << i << endl;
            found = true;
            break;
        }
    }
    timeEnd = clock();

    if(!found){
        cout << "[Linear search] Not found in array" << endl;
    }

    Cout  << "[Linear search] Time elapsed: "
          << (double)(timeEnd - timeStart)*1000/CLOCKS_PER_SEC
          << "ms"  << endl;

    return found;
}

void BigONotation::bubbleSort(){
    timeStart = clock();

    for(int i=0; i<arrayLen;i++){
        for(int j=0;j<arrayLen;j++){
            if(array[j]>array[j+1])
                swapIndex(j, j+1);
        }
    }
    timeEnd = clock();

    cout  << "[Bubble sort] Time elapsed: "
          << (double)(timeEnd - timeStart)*1000/CLOCKS_PER_SEC
          << "ms" << endl;

}

bool BigONotation::binarySearch(int x){
    int start = 0, end = arrayLen-1;
    int searchIteration = 0;
    bool found = false;

    timeStart = clock();
    while(end > start+1){
        ++searchIteration;
        int mid = (start + end)/2;
        if (array[mid] < x){
            start = mid;
        }
        else if(array[mid] > x){
            end = mid;
        }
        else{
            cout  << "[Binary search] Found at iteration "
                  << searchIteration << endl;
            found = true;
            break;
        }
    }
```

```
    if(start != end){
        if(array[start] == x){
            cout   << "[Binary search] Found at iteration "
                   << searchIteration << endl;
            found = true;
        }
    }

    if(!found){
        cout  << "[Binary search] Not found in array after iteration "
              << searchIteration << endl;
    }
    timeEnd = clock();

    cout  << "[Binary search] Time elapsed: "
          << (double)(timeEnd - timeStart)/CLOCKS_PER_SEC << "ms" << endl;

    return found;
}

void BigONotation::stdSort(){
    timeStart = clock();

    sort(array, array+arrayLen);

    timeEnd = clock();

    cout   << "[Std sort] Time elapsed: "
           << (double)(timeEnd - timeStart)/CLOCKS_PER_SEC << "ms" << endl;
}
```

File makefile:

```
main: main.cpp
      g++ -std=c++11 -o main main.cpp
clean: main
      rm -rf main
test: main
      ./main
```