Aplikasi Pohon Pencarian Biner Seimbang sebagai Memo Table Dynamic Programming

Reinhard Benjamin Linardi, 13515011
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515011@std.stei.itb.ac.id

***Abstract*—Memo Table merupakan bagian penting dari teknik *memoization* pada Dynamic Programming. Namun, jika range dan jumlah data yang dimasukkan sangat besar, penggunaan tabel (array) sebagai memo table sangat tidak efektif dari segi kompleksitas waktu dan memori. Salah satu alternatif dalam mengatasi masalah ini adalah menggunakan pohon pencarian biner seimbang sebagai memo table. Bagaimana caranya?**

***Keywords*—Dynamic Programming, Kompleksitas, Memo Table, Pohon Pencarian Biner Seimbang**

I. PENDAHULUAN

Pemrograman merupakan kemampuan yang wajib dimiliki oleh setiap mahasiswa Teknik Informatika. Salah satu teknik dalam pemrograman adalah rekursi, yaitu menyelesaikan masalah dengan cara menyelesaikan *subprolem* dari masalah tersebut. Rekursi merupakan teknik yang sangat bermanfaat karena dapat dibuat dengan kode yang relatif pendek, yaitu dengan membuat subprogram yang memanggil dirinya sendiri dengan parameter yang berbeda.

Sayangnya, rekursi memiliki kekurangan. Untuk relasi rekurens yang mengandung *overlapping subproblem*, rekursi akan menghitung ulang suatu *subproblem* yang sudah pernah dihitung sebelumnya, mengakibatkan kompleksitas waktu yang dibutuhkan menjadi eksponensial. Untuk mengatasi hal ini, digunakan teknik pemrograman lain yang disebut *dynamic programming*, yang menyimpan *return value* dari *state* yang telah dikunjungi. *Return value* disimpan dalam sebuah *memo table*, pada umumnya berbentuk array dengan dimensi sesuai dengan jumlah parameter dari fungsi *dynamic programming*. Teknik penyimpanan inilah yang disebut *memoization*, setiap parameter menandakan suatu *state* spesifik dari fungsi tersebut, yang akan disimpan dalam *memo table* untuk *state* yang sama.

Dengan menyimpan hasil perhitungan, maka saat fungsi menemukan *state* yang sudah pernah dihitung, fungsi secara langsung akan mengembalikan nilai dari *memo table* tanpa menghitung ulang, sehingga kompleksitas perhitungan menjadi O(1). Kompleksitas fungsi secara total menjadi linear dan jauh lebih cepat dari eksponensial jika memakai rekursi saja.

Tetapi, masalah belum berhenti sampai disitu. Pengunaan tabel sebagai *memo table* memiliki kelemahan. Bila fungsi menerima input 1 dan 10^9, maka akan dibuat tabel yang berukuran 10^9 (1 GB) karena representasi tabel yang kontigu, padahal yang terpakai hanya indeks pertama dan terakhir saja. Tabel juga tidak efisien jika dibuat dinamis sebab harus melakukan penggeseran data jika ada data baru. Jika data tidak digeser, harus dilakukan *sequential search* untuk mencari data, tidak dapat dilakukan pemanggilan langsung. Hal ini akan membebani kompleksitas fungsi *dynamic programming* sehingga menjadi tidak efisien lagi dalam menyelesaikan masalah.

II. LANDASAN TEORI

2.1 Pohon

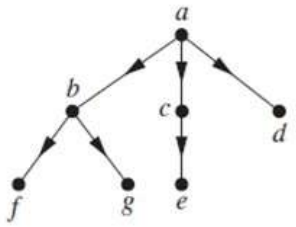
A. Definisi Pohon

Pohon adalah graf tak-berarah terhubung yang tidak mengandung sirkuit. Misalkan $G = (V,E)$ adalah graf tak-berarah sederhana dan jumlah simpulnya n. Maka semua pernyataan di bawah ini ekivalen :

1. G adalah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki $m = n - 1$ buah sisi.
4. G tidak mengandung sirkuit dan memiliki $m = n - 1$ buah sisi.
5. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit.
6. G terhubung dan semua sisinya adalah jembatan.

B. Pohon Berakar

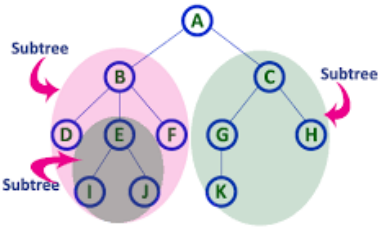
Pohon yang satu buah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah dinamakan pohon berakar (*rooted tree*). Dalam penggambaran pohon berakar seringkali arah dihilangkan sebagai konvensi / perjanjian.



Gambar 1 : Pohon Berakar

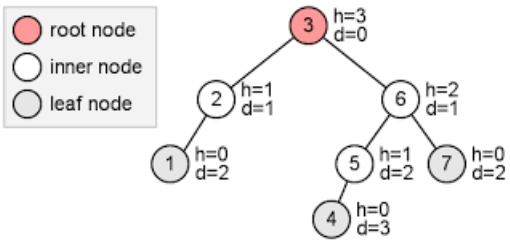
Terminologi pada pohon berakar :

1. Anak : *b*, *c*, dan *d* adalah anak dari *a*.
2. Orangtua : *a* adalah orangtua dari *b*, *c*, *d*.
3. Lintasan : Lintasan dari *a* ke *f* adalah *a*, *b*, *f*. Panjang lintasan adalah 2.
4. Saudara kandung : *f* adalah saudara kandung *g*.
5. Upapohon (*subtree*) : Berikut ini merupakan contoh dari upapohon.



Gambar 2 : Upapohon

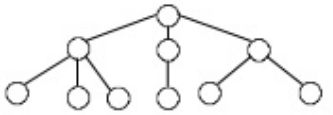
6. Derajat : Jumlah anak atau upapohon dari simpul tersebut. Derajat yang dimaksud adalah derajat keluar. Contoh : *a* memiliki derajat 2, *b* memiliki derajat 3.
7. Daun (*leaf*) : Simpul berderajat 0. Contoh daun : simpul *d*, *i*, *j* ,*k*, *f*, dan *h*.
8. Simpul dalam (*internal nodes*) : Simpul yang memiliki anak disebut simpul dalam. Contoh : simpul *b*, *e*, *g*, *c*.
9. Aras (*level*) , tingkat, atau kedalaman (*d*), dan tinggi (*h*) :



Gambar 3 : Aras/tingkat/kedalaman, dan tinggi pohon

C. Pohon *n-ary*

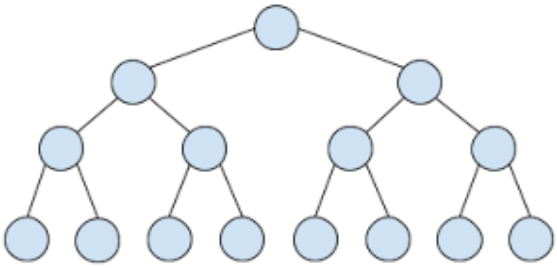
Pohon *n-ary* merupakan pohon berakar yang setiap simpulnya mempunyai paling banyak *n* buah anak. Pohon *n-ary* disebut teratur atau penuh (*full*) jika setiap simpulnya mempunyai tepat *n* anak. Berikut adalah ilustrasi pohon *n-ary* :



Gambar 4 : Pohon *n-ary*

D. Pohon Biner

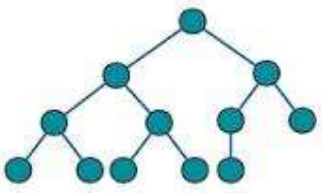
Pohon biner merupakan pohon *n-ary* dengan $n = 2$. Pohon biner merupakan salah satu pohon yang sangat banyak aplikasinya. Pada pohon biner, setiap simpul mempunyai paling banyak 2 buah anak, yaitu anak kiri (*left child*) dan anak kanan (*right child*). Pohon biner merupakan pohon terurut karena adanya perbedaan urutan anak.



Gambar 5 : Pohon biner

E. Pohon Biner Seimbang

Pohon biner seimbang merupakan pohon yang mempunyai tinggi upapohon kiri dan tinggi upapohon kanan seimbang, yaitu berbeda maksimal 1.



Gambar 6 : Pohon biner seimbang

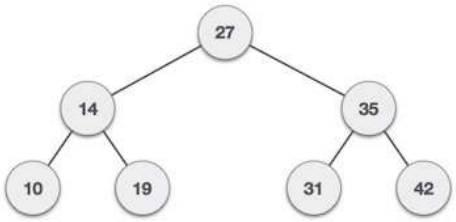
2.2 Pohon Pencarian Biner

Pohon pencarian biner (*binary search tree*) merupakan pohon biner yang setiap simpulnya mempunyai pasangan nilai (*value*) dan kunci (*key*), serta penambahan simpul baru pada pohon pencarian biner harus memenuhi aturan berikut :

key(left subtree) ≤ *key(node)* ≤ *key(right subtree)*

Pengaturan simpul seperti ini akan menjaga sifat *binary search* dari pohon pencarian biner, sehingga :

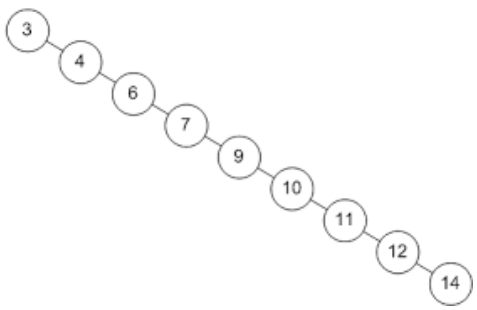
1. Pencarian dilakukan dengan mencari *key* dari *value* yang diinginkan, secara tidak langsung memanfaatkan sifat *binary search*, kompleksitas : Θ(log n).
2. Penambahan simpul dilakukan dengan mencari posisi *key* yang tepat, secara tidak langsung memanfaatkan sifat *binary search*, kompleksitas : Θ(log n).
3. Penghapusan simpul adalah pencarian simpul dengan *key* yang diinginkan, kompleksitas : Θ(log n).



Gambar 7 : Pohon pencarian biner

2.3 Pohon pencarian biner seimbang

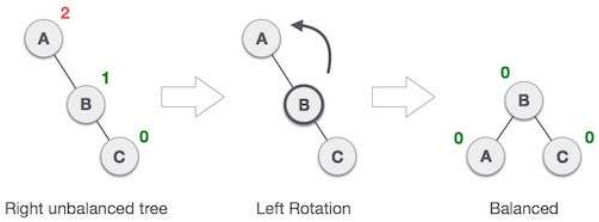
Pohon pencarian biner seimbang (*balanced binary search tree*) merupakan pengembangan dari pohon pencarian biner. Jika kita memasukkan input yang terurut membesar dan menyimpannya pada pohon pencarian biner, maka akan terbentuk pohon condong kanan :



Gambar 8 : Pohon condong kanan

Pohon pencarian biner yang tidak seimbang, pada kasus terburuknya condong kiri/kanan, menyebabkan sifat *binary search* pohon tidak berguna dan kompleksitas pencarian, penambahan, maupun pegnhapusan menjadi O(n). Oleh karena itu, dibutuhkan pohon pencarian biner yang dapat menyeimbangkan dirinya sendiri (*self-balancing*) yang dikenal dengan pohon pencarian biner seimbang. Salah satu variasinya adalah AVL Tree yang ditemukan oleh Adelson, Velski, dan Landis.

AVL Tree memanfaatkan ide pohon biner seimbang, dimana perbedaan tinggi upapohon maksimal 1. Perbedaan ini dikenal dengan istilah *balance factor*, yaitu *height(left subtree) – height(right subtree)*. Jika *balance factor* lebih dari 1, maka akan dilakukan rotasi untuk menyeimbangkan pohon. Dengan begitu, kompleksitas pencarian, penambahan, maupun penghapusan simpul menjadi O(log n).



Gambar 9 : AVL Tree

2.4 Dynamic Programming

Dynamic Programming merupakan salah satu teknik pemrograman yang menyimpan *return value* dari *state* yang sudah pernah dikunjungi sebelumnya. *Dynamic Programming* banyak digunakan untuk permasalahan yang harus diselesaikan dengan *recursive complete search*, tetapi mengandung *overlapping subproblem*. Untuk memahami *dynamic programming*, diperlukan dasar pemahaman rekursi yang kuat. Mari kita mulai dengan *overlapping subproblem*.

A. *Overlapping Subproblem*

Perhatikan potongan kode di bawah ini.

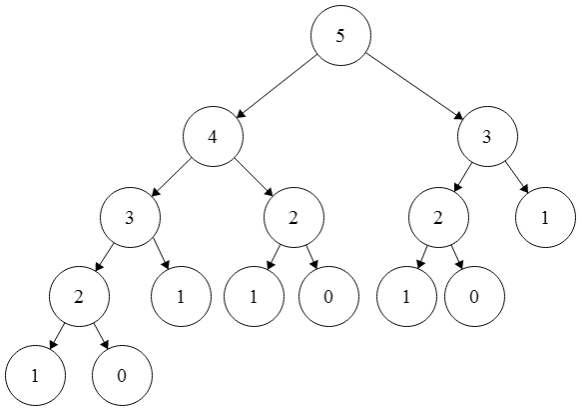
```
int fibo(int n)

   if (n==0) return 0;
   if (n==1) return 1;

   return fibo(n-1) + fibo(n-2);
```

Kode di atas merupakan fungsi untuk menghitung barisan Fibonacci ke-*n* dengan teknik rekursi. Basisnya, barisan ke-0 adalah 0, dan barisan ke-1 adalah 1. Rekurensnya, barisan ke-*n* adalah penjumlahan barisan ke-(*n-1*) dan barisan ke-(*n-2*).

Jika kita memanggil `fibo(5)`, maka pohon pemanggilan fungsinya adalah :



Gambar 10 : Pohon pemanggilan `fibo(5)` dengan rekursi

Kita melakukan pemanggilan terhadap `fibo(3)` sebanyak 2 kali, dan `fibo(2)` sebanyak 3 kali. Padahal, kita sudah pernah menghitung `fibo(3)` dan `fibo(2)` sebelumnya. Untuk `fibo(6)` akan dilakukan pemanggilan terhadap `fibo(4)` sebanyak 2 kali, `fibo(3)` sebanyak 3 kali, dan `fibo(2)` sebanyak 5 kali! Inilah yang disebut *overlapping subproblem*, *subproblem* yang sama muncul berkali-kali dan menjadi salah satu penyebab tingginya kompleksitas waktu yang dibutuhkan jika menggunakan rekursi saja.

B. Memoization

Cara yang digunakan dalam *dynamic programming* untuk mengatasi masalah di atas adalah membuat tabel yang menyimpan nilai dari *state* yang telah dikunjungi sebelumnya, yang disebut *memoization*. Untuk kasus barisan Fibonacci, kita dapat menggunakan array untuk menyimpan *return value* dari *state* yang telah dikunjungi. Isi dari `memo` indeks ke-*n* berisi barisan Fibonacci ke-*n*. Array inilah yang disebut *memo table*.

```
int memo[101];

memo[0] = 0;
memo[1] = 1;
```

Untuk membedakan apakah Fibonacci ke-*n* sudah pernah dihitung, kita dapat menginisialisasi semua isi `memo` dengan suatu elemen dummy, yang menandakan bahwa *state* tersebut belum pernah dihitung. Untuk kasus ini, kita dapat menggunakan nilai -1 karena barisan Fibonacci selalu bernilai positif.

```
for(int i=2;i<=100;i++)
    memo[i] = -1;
```

C. *Dynamic Programming*

Untuk mengubah rekursi menjadi *dynamic programming*, yang perlu ditambahkan adalah mengecek apakah *state* sudah pernah dihitung. Jika sudah, langsung kembalikan nilai dari *memo table*. Jika belum, cek apakah *state* merupakan basis. Jika basis, langsung *return* nilai dari basis itu, jika tidak, hitung terlebih dahulu nilainya, lalu masukkan nilainya ke dalam *memo table*, lalu *return* nilai tersebut.

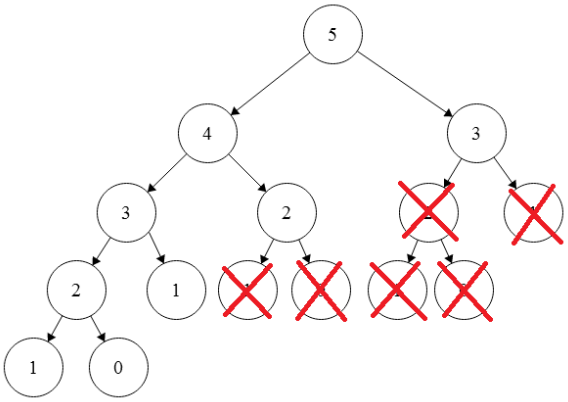
```
int fibo(int n)

    if (memo[n]!=-1) return memo[n];

    if (n==0) return 0;
    if (n==1) return 1;

    memo[n] = fibo(n-1) + fibo(n-2);
    return memo[n];
```

Sebagai catatan, teknik *dynamic programming* yang dibahas di sini adalah *top down*, yaitu memecah masalah menjadi *subproblem* sampai mencapai basis lalu mendapatkan hasilnya. Pada teknik ini, inisialisasi `memo` pada *state* basis tidak diperlukan, karena kasus basis pada fungsi akan langsung mengembalikan suatu nilai. Setelah mengubah rekursi menjadi *dynamic programming*, maka pohon pemanggilan fungsi menjadi :



Gambar 11 : Pohon pemanggilan `fibo(5)` dengan *dynamic programming*

Dengan teknik *dynamic programming*, setiap *state* hanya perlu dihitung sekali, dan selanjutnya akan mengembalikan nilai dari *memo table* dengan kompleksitas O(1). Maka, kompleksitas program pun menjadi linear, tidak eksponensial seperti memakai rekursi saja. Untuk nilai *n* yang besar, teknik *dynamic programming* sangat efisien dari segi kompleksitas waktu yang dibutuhkan.

III. APLIKASI POHON PENCARIAN BINER SEIMBANG SEBAGAI MEMO TABLE DYNAMIC PROGRAMMING

Pada bagian pendahuluan, telah dijelaskan bahwa masalah utama dalam pemakaian tabel sebagai *memo table* dari fungsi *dynamic programming* adalah tidak dapat menyimpan data dengan *range* atau jumlah yang besar. Maka, solusi dari masalah ini adalah mengaplikasikan pohon pencarian biner seimbang sebagai *memo table* dari fungsi *dynamic programming*. Ada 2 alasan utama yang mendasari pemilihan pohon pencarian biner seimbang sebagai *memo table* fungsi *dynamic programming*, yaitu :

1. Pohon pencarian biner seimbang memiliki kompleksitas O(log n) untuk pencarian, penambahan dan penghapusan simpul. Hal ini sangat baik untuk struktur data yang tidak kontigu.

2. Pohon pencarian biner seimbang memiliki kompleksitas ruang O(n). Karena tidak kontigu, maka memori yang dipakai hanya sebanyak jumlah input saja, tidak bergantung pada *range* dari masukan.

Untuk mengimplementasikan pohon pencarian biner seimbang, kita dapat menggunakan *library* yang ada di bahasa pemrograman. Pohon pencarian biner seimbang dikenal juga dengan nama *map* atau *dictionary*. Sebagai contoh, pada C++, pohon pencarian biner seimbang terdapat pada Standard Template Library (STL) dengan nama **std::map**. Berikut merupakan contoh potongan kode penggunaan *map* dalam C++ :

Pertama, kita gunakan "#include<map>" atau "#include<bits/stdc++.h>" untuk memasukkan map dalam program kita. Gunakan juga "using namespace std;" untuk mempermudah. Selanjutnya, deklarasi dapat dibuat dengan menggunakan :

```
map <key,value> name;
```

map dicetak tebal pada penulisan di atas, karena merupakan keyword dari C++ untuk menandakan struktur data *map*. Untuk `key` dan `value`, diganti dengan tipe data dari kunci dan tipe data dari nilai. Sedangkan, `name` merupakan nama variabel dari *map* itu sendiri. Pada kasus barisan Fibonacci, tipe data dari kunci dan nilai adalah integer. Pasangan kunci dan nilai dalam kasus ini adalah suku ke-*n* dan nilai barisan Fibonacci ke-*n*. Mengapa? Karena yang kita ingin ketahui adalah apakah barisan Fibonacci ke-*n* sudah pernah dihitung atau belum, maka kita mempunyai data suku ke-*n* untuk mencari informasi tersebut. Maka suku ke-*n* menjadi kunci di *map* dan yang menjadi nilai adalah barisan Fibonacci ke-*n*.

Untuk menambahkan simpul, maupun memperbarui informasi nilai pada simpul, dapat dilakukan operasi pemberian nilai (*assign*) pada simpul tersebut, contoh :

```
memo[0] = 0; //penambahan simpul
memo[1] = 1;
memo[2] = -1;

/* pembaruan nilai simpul(update) */

memo[2] = 1;
```

Pada contoh di atas, dibuat simpul baru dengan pasangan kunci dan nilai (0,0), (1,1) dan (2,-1). Selanjutnya, simpul dengan kunci 2, diperbarui nilainya menjadi 1.

Karena struktur data *map* tidak kontigu, kita tidak memerlukan elemen dummy sebagai penanda bahwa simpul belum dihitung, cukup dengan mencari apakah ada simpul dengan kunci *k* atau tidak. Kita dapat menggunakan fungsi **std::map::count** untuk mencari tahu apakah simpul dengan kunci *k* ada di *map* :

```
if(memo.count(k)) printf("Yes\n");
else printf("No\n");
```

`memo` merupakan nama *map* yang kita deklarasikan, dan `memo.count(k)` akan mencari apakah ada simpul dengan kunci *k* pada `memo`. Jika ada, akan mengembalikan 1, jika tidak ada, akan mengembalikan 0.

Dengan mengetahui struktur data *map* dan fungsi **std::map::count**, maka kita dapat membuat fungsi *dynamic programming* barisan Fibonacci dengan memanfaatkan *map* sebagai *memo table* seperti contoh di bawah ini :

```
#include<bits/stdc++.h>

using namespace std;

map <int,int> memo;

int fibo(int n)
{
    if(memo.count(n))return memo[n];
    if(n==0 || n==1) return n;

    return memo[n] = fibo(n-1) +
    fibo(n-2);
}

int main()
{
    int x;

    cout << "Fibonacci ke-";
    cin >> x;
    cout << x << " = " << fibo(x);

    return 0;
}
```

IV. KESIMPULAN

Pohon merupakan salah satu jenis graf yang sangat banyak aplikasinya, tidak hanya pada bidang Matematika, tetapi juga bidang-bidang lain, termasuk Informatika. Pohon dapat digunakan sebagai struktur data *map*, yang merupakan pohon pencarian biner seimbang dan simpulnya menyimpan pasangan nilai dan kunci. Struktur data ini sangat efisien dari segi kompleksitas waktu dan ruang sehingga dapat dimanfaatkan untuk menyelesaikan persoalan seperti *dynamic programming* yang merupakan teknik pemrograman penting di Informatika.

V. UCAPAN TERIMA KASIH

Pertama-tama, penulis mengucapkan syukur kepada Tuhan Yang Maha Esa, oleh karena penyertaan-Nya penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga ingin mengucapkan terima kasih untuk Bapak Dr. Ir. Rinaldi Munir, M.T. dan Ibu Dra. Harlili S., M.Sc., selaku dosen mata kuliah Matematika Diskrit yang telah memberikan ilmu dan menginspirasi penulis dalam penulisan makalah ini.

REFERENSI

[1] Rinaldi Munir, Matematika Diskrit. Bandung : Penerbit Informatika, Palasari.
[2] https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm diakses pada 4 Desember 2016 pukul 11:26
[3] https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm diakses pada 4 Desember 2016 pukul 13:52
[4] https://community.topcoder.com/tc?module=Static&d1=features&d2=040104 diakses pada 4 Desember 2016 pukul 14:14
[5] http://www.cplusplus.com/reference/map/map/ diakses pada 6 Desember 2016 pukul 22:24
[6] http://www.cplusplus.com/reference/map/map/map/ diakses pada 6 Desember 2016 pukul 23:03
[7] http://www.cplusplus.com/reference/map/map/count/ diakses pada 6 Desember 2016 pukul 23:34

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, 7 Desember 2016

Reinhard Benjamin Linardi
13515011