Pohon Indeks Biner atau Pohon Fenwick untuk menyelesaikan persoalan penjumlahan interval

Eric 13512021
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13512021@std.stei.itb.ac.id

***Abstrak*—Pengembangan ilmu matematika diskrit menghasilkan struktur-struktur efisien yang dapat diaplikasikan dalam algoritma dalam ilmu komputer untuk mempercepat waktu eksekusi sebuah algoritma. Struktur data yang dipakai di dalam implementasi suatu algoritma sangat memengaruhi kemangkusan algoritma tersebut. Makalah ini membahas salah satu contoh pengaruh struktur data, Pohon Indeks Biner atau Pohon Fenwick, dalam menyelesaikan persoalan penjumlahan interval. Makalah ini mengulas beberapa pendekatan algoritma dengan struktur data berbeda yang dapat digunakan dan memberikan kelebihan dan kelemahan suatu algoritma. Makalah ini juga membahas kompleksitas waktu dan ruang dari setiap pendekatan. Setelah membaca makalah ini diharapkan pembaca dapat mengerti prinsip kerja Pohon Indeks Biner atau Fenwick Tree.**

***Kata Kunci*— kompleksitas, operasi *bitwise*, Penjumlahan Interval, pohon, Pohon Fenwick, Pohon Indeks Biner.**

I. PENDAHULUAN

Persoalan penjumlahan interval merupakan masalah yang klasik dalam bidang komputer sains, penjumlahan interval biasanya merupakan langkah awal untuk algoritma-algoritma yang lebih kompleks. Sebelum diulas lebih jauh mengenai persoalan penjumlahan interval ini, selanjutnya akan disebut sebagai PI, ada baiknya kita mendefinisikan persoalan PI dengan jelas terlebih dahulu.

Persoalan PI adalah persoalan dimana kita memiliki sekumpulan nilai. Misalkan nilai tersebut adalah sebanyak n dan direkam dalam sebuah array A[1..n]. Misalkan terdapat dua buah indeks i dan j, dimana i ≤ j Persoalannya adalah berapakah jumlah nilai dari indeks ke-i sampai indeks ke-j? Persoalan ini dapat juga dituliskan dalam notasi sigma sebagai $PI_A(i,j) = \sum_{k=i}^{j} A_k$.

Persoalan PI ini dapat diselesaikan dengan banyak metode, seperti metode trivial/naif (*brute-force*), *Dynamic Programming*, *Segment Tree*, dan lain-lain. Salah satu metode yang dapat dipakai untuk menyelesaikan PI adalah dengan menggunakan Pohon Indeks Biner (*Binary Indeks Tree*) atau Pohon Fenwick (*Fenwick Tree*). Pohon ini mempunyai dua buah nama, dimana nama pertama lebih mendeskripsikan prinsip kerja Pohon. Pohon ini juga memiliki nama kedua karena orang yang pertama kali menuliskan makalah tentang pohon ini bernama Peter M. Fenwick, beliau menuliskan makalah tersebut pada tahun 1994.

Contoh Persoalan PI yang sederhana adalah sebagai berikut. Diberikan Sekumpulan bilangan $A_1,A_2,...,A_n$. Berapa kemungkinan indeks i dan j, dimana i ≤ j, sehingga $A_i,A_{i+1},...,A_{j-1},A_j = 0$? Persoalan ini memiliki cara naif dengan komplesitas waktu $O(n^3)$. Dengan algoritma yang lebih baik, persoalan ini dapat diselesaikan dalam $O(n^2)$. Dapatkah anda mencari solusi $O(n^2)$ nya?

Dalam makalah ini semua kode diberikan dalam bahasa C. Penulis memilih untuk menuliskan dalam bahasa C karena bahasa C merupakan bahasa yang sangat populer, terdapat di hampir semua platform, dan telah memiliki standar yang diikuti oleh seluruh *compiler C*.

II. DASAR TEORI

A. Operasi *Bitwise*

Operasi *Bitwise* adalah operasi matematika yang bekerja secara bit per bit untuk dua buah angka. Operasi yang dilakukan untuk setiap bit ekivalen dengan operasi logika yang sesuai, dengan 1 sebagai *true*, dan 0 sebagai *false*.

Contoh operasi *bitwise*:

- **not**
  not 7_{10} = not 0111_2 = 1000_2 = 8_{10}
- **and**
  5_{10} and 3_{10} = 0101_2 and 0011_2 = 0001_2 = 1_{10}
- **or**
  5_{10} or 3_{10} = 0101_2 or 0011_2 = 0111_2 = 7_{10}
- **xor**
  5_{10} xor 3_{10} = 0101_2 xor 0011_2 = 0110_2 = 6_{10}

Selain operasi *bitwise* di atas, masi terdapat dua jenis operasi *bitwise* yaitu operasi *shift left* dan operasi *shift right*.

Operasi *Bitwise* dapat dilakukan dengan sangat cepat oleh komputer karena operasi tersebut didukung langsung oleh *processor*. Kecepatan operasi *bitwise* lebih cepat daripada operasi penjumlahan, pengurangan, dan lain-lain.

B. Pohon berakar atau pohon

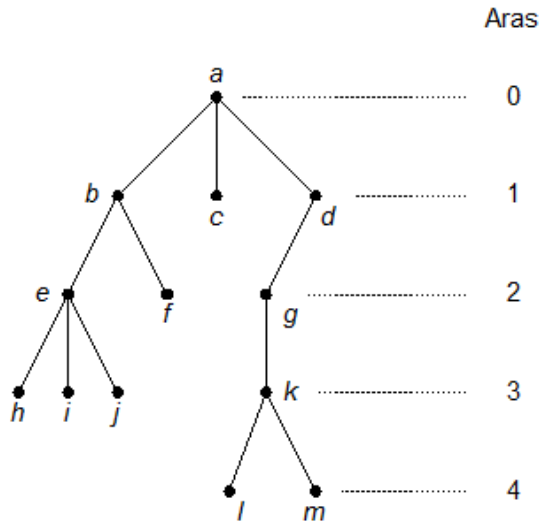
Pohon berakar atau pohon dalam bidang komputer sains, merupakan Struktur Data Abstrak yang sangat umum dan memiliki banyak aplikasi di dalam kehidupan nyata.

Pohon sering digunakan untuk merepresentasikan struktur yang memiliki hirarki tertentu, misalkan terdiri dari beberapa objek dengan tingkatan tertentu dan terdapat hubungan antar objek. Struktur ini disebut Pohon karena bentuknya yang seperti pohon dalam biologi meskipun terbalik.

Sebagian besar struktur di kehidupan nyata memiliki hirarki tertentu, seperti struktur jabatan di dalam sebuah perusahaan, struktur keluarga, stuktur evolusi makhluk hidup, dsb. Beberapa persoalan yang tidak secara langsung terlihat sebagai pohon juga dapat diubah ke dalam bentuk pohon.

Pohon merupakan stuktur yang rekursif, karena sebuah Pohon mengandung informasi, dan beberapa sub-Pohon. Sub-Pohon tersebut juga merupakan Pohon. Pohon juga merupakan sebuah bentuk khusus dari Graf, tetapi karena kegunaan pohon sangat banyak sehingga terdapat pembahasan khusus untuk pohon. Pohon adalah graf yang terdiri dari n simpul dan n-1 sisi berarah, tetapi karena arahnya selalu dari simpul dengan aras lebih rendah ke aras yang lebih tinggi, maka sering digambarkan tanpa arah.

Untuk kemudahan memehami pohon, gambar 2.1 di bawah ini.



Gambar 2.1 Contoh sebuah pohon
(slide mata kuliah IF2120, http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Pohon%20(2013).ppt diakses tanggal 14 Desember 2013)

Beberapa definisi yang akan sering dipakai di dalam pohon:

- *Ayah*, simpul yang memiliki aras lebih rendah yang berhubungan dengan suatu simpul.
- *Anak*, simpul yang memiliki aras lebih tinggi yang berhubungan dengan suatu simpul.
- *Akar*, simpul yang tidak memiliki ayah (memiliki aras terendah).
- *Daun*, simpul yang tidak memiliki anak.
- *Tinggi*, aras maksimum dari suatu pohon.

Pada gambar 2.1 di atas:

- g adalah ayah dari k
- c adalah anak dari a
- a adalah akar dari pohon tersebut
- h,i,j,f,c,l, dan m adalah daun dari pohon tersebut

Terdapat beberapa variasi dari pohon:

- Pohon terurut, pohon yang memiliki urutan anak.
- Pohon n-*ary*, pohon yang setiap simpulnya memiliki maksimal n anak.
- Pohon biner, pohon 2-*ary* terurut.
- Pohon biner seimbang, pohon biner yang memiliki selisih tinggi sub-pohon kiri dan sub-pohon kanan maksimal 1.

C. Penjumlahan Prefiks

Penjumlahan Prefiks adalah penjumlahan interval yang dimulai dari 1. Dalam notasi sigma dituliskan sebagai $\sum_{i=1}^{n} A_i$. Penjumlahan prefiks biasanya merupakan soal ujian ataupun dapat digunakan sebagai bagian dari algoritma lain yang lebih kompleks.

III. PEMBAHASAN

A. Pendekatan trivial/naif (*brute force*)

Metode paling trivial untuk menyelesaikan persoalan Penjumlahan Interval adalah dengan menjumlahkan elemen per elemen dari indeks ke-i sampai indeks ke-j. Berikut ini adalah potongan kode-nya dalam bahasa C.

```
/* Mencari PI */
  int sum = 0, k;
  for (k = i; k <= j; k++)
    sum += A[k];

/* Mengubah nilai A[x] */
  A[x] = NILAI_BARU;
```

Algoritma di atas akan berjalan dalam kompleksitas waktu O(n) untuk setiap kalinya. Algoritma ini memiliki kelebihan yaitu tidak memerlukan tahap persiapan, sehingga perubahan nilai A_k dapat dilakukan secara langsung dalam O(1). Algoritma ini juga tidak memerlukan memori tambahan. Algoritma ini cocok digunakan jika nilai PI tidak sering diperlukan.

B. Pendekatan dengan *Dynamic Programming*

Penjumlahan Interval dapat diselesaikan dalam O(1) dengan pendekatan *Dynamic Programming* (DP). Namun pendekatan ini memerlukan tabel prefiks, tabel prefiks dapat dibangun dalam O(n). Prinsipnya adalah menggunakan identitas penjumlahan

$\sum_{k=i}^{j} A_k = \sum_{k=0}^{j} A_k - \sum_{k=0}^{i-1} A_k$.

Dengan menambahkan elemen ke-0, apa pun nilainya, biasanya nol. Berikut ini adalah potongan kode-nya dalam bahasa C.

```
/* Mempersiapkan tabel prefiks */
  int Pref[n+1], k;
  Pref[0] = 0; /* Elemen ke-0 */
  for (k = 1; k <= n; k++)
    Pref[k] = Pref[k-1] + A[k];

/* Mencari PI */
  int sum = Pref[j] – Pref[i-1];

/* Mengubah nilai A[x] */
  A[x] = NILAI_BARU;
  for (k = x; k <= n; k++)
    Pref[k] = Pref[k-1] + A[k];
```

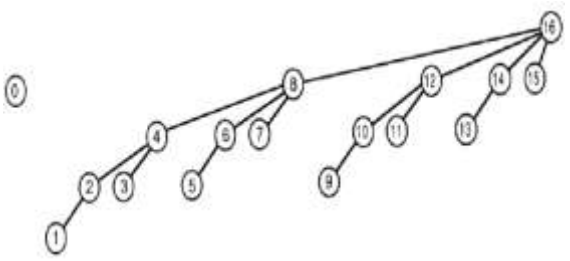
Setelah melakukan tahap persiapan awal dalam kompleksitas waktu O(n), algoritma di atas dapat mencari PI dalam O(1). Namun perubahan nilai pada A menyebabkan kita harus membangun ulang sebagian/seluruh tabel Prefiks dalam waktu O(n). Pendekatan DP sangat cocok untuk data-data yang tidak banyak berubah, tetapi nilai PI sering diperlukan. Algoritma di atas juga memerlukan tambahan memori sebesar O(n).

C. Pendekatan dengan Pohon Fenwick

a. Struktur pohon Fenwick dan keterhubunganya

Pohon Fenwick adalah pohon yang dibuat sedemikian rupa sehingga penjumlahan prefiks dapat dihitung dalam kompleksitas waktu O(log n). Pohon Fenwick hanya dapat bekerja untuk indeks yang berupa bilangan bulat lebih besar dari 0. Representasi abstrak pohon Fenwick adalah sebagai berikut:

- setiap simpul yang indeksnya habis dibagi 2^n dengan n sebesar mungkin akan memiliki n anak. Contoh: simpul dengan indeks ganjil habis dibagi oleh 2^0 sehingga akan memiliki 0 anak, simpul dengan indeks 12 habis dibagi oleh 2^2 sehingga akan memiliki 2 anak.
- Anak dari simpul didapat dengan mengurangi indeks simpul dengan bilangan 2^n untuk n dari 0 sampai dengan na-1, dengan na sebagai banyak anak. Contoh:
  - 8 memiliki 3 anak, yaitu $8\text{-}2^0=7$, $8\text{-}2^1=6$, $8\text{-}2^2=4$.
  - 12 memiliki 2 anak, yaitu $12\text{-}2^0=11$ dan $12\text{-}2^1=10$.
- Ayah dari simpul didapat dengan menambahkan bit paling kanan dengan indeks simpul. Contoh:
  - ayah dari 3 adalah $11_2 + 1_2 = 100_2 = 4$
  - ayah dari 4 adalah $100_2 + 100_2 = 1000_2 = 8$
  - ayah dari 6 adalah $110_2 + 10_2 = 1000_2 = 8$
  - ayah dari 7 adalah $111_2 + 1_2 = 1000_2 = 8$



Gambar 3.1 Contoh sebuah pohon Fenwick dengan 16 elemen dengan indeks 1 sampai 16 (0 tidak dianggap). (sumber: http://i51.photobucket.com/albums/f390/konaht/BIT2-1.png diakses tanggal 15 Desember 2013).

b. Sifat-sifat pohon Fenwick dan kompleksitas berbagai operasi

Setiap simpul bertanggung jawab terhadap dirinya sendiri dan seluruh simpul yang merupakan *descendant* dari simpul tersebut. Contoh:

- simpul dengan indeks 8 bertanggung jawab terhadap simpul dengan indeks 1,2,3,4,5,6,7,8.
- simpul dengan indeks 7 bertanggung jawab terhadap simpul dengan indeks 7.

Jadi, nilai yang terkandung pada simpul dengan indeks 8 merupakan jumlah dari $A_1,A_2,...,A_8$. Sedangkan nilai yang terkandung pada simpul dengan indeks 7 hanya merupakan nilai dari A_7. Dengan cara yang seperti ini penjumlahan prefiks dapat dihitung dalam kompleksitas waktu O(log n). Misalkan kita ingin menghitung $\sum_{k=1}^{8} A_k$, kita hanya perlu mencari nilai dari simpul dengan indeks 8. Untuk menghitung $\sum_{k=1}^{7} A_k$, kita dapat menjumlahkan nilai dari simpul dengan indeks 4,6, dan 7.

Untuk menghitung penjumlahan prefiks dalam bentuk $\sum_{k=1}^{x} A_k$, seseorang harus menjumlahkan nilai simpul dengan indeks tertentu pada pohon Fenwick. Indeks tersebut dapat ditentukan dengan operasi *bitwise*, kita hanya perlu menjumlahkan simpul $x_1 = x$, $x_2 = x_1 – lsb(x_1)$, $x_3 = x_2 – lsb(x_2)$, sampai x_n dimana x_n hanya memiliki 1 bit yang bernilai 1. Fungsi lsb(x) (*Least Significant Byte*) pada persamaan sebelumnya menghasilkan nilai bit terkecil pada x yang bernilai 1. Contoh:

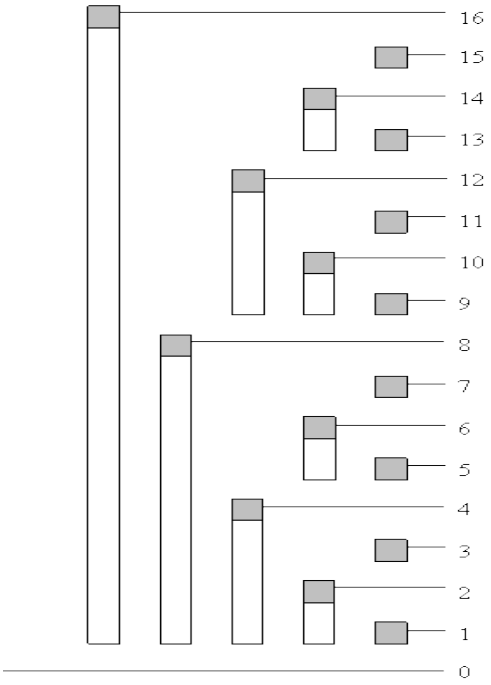
- untuk menghitung $\sum_{k=1}^{7} A_k$, kita hanya perlu menjumlahkan simpul ke $7 = 111_2$, $111_2\text{-}1_2 = 110_2 = 6$, $110_2\text{-}10_2 = 100_2 = 4$.
- Untuk menghitung $\sum_{k=1}^{2^y} A_k$, kita hanya perlu membaca nilai A_{2^y} pada tabel karena 2^y hanya memiliki 1 bit yang bernilai 1.

Operasi untuk mendapatkan penjumlahan prefiks pada pohon Fenwick memiliki komplesitas waktu O(log n).

Untuk mengubah nilai ke-k pada tabel, pohon Fenwick harus diubah juga. Pohon Fenwick diubah dengan menambahkan nilai (nilai_baru-nilai_lama) dari tabel. Kita hanya perlu mengubah simpul dengan indeks yang bertanggung jawab untuk indeks ke-k tersebut. Simpul yang bertanggung jawab untuk indeks ke-k adalah simpul dengan indeks ke-k, dan seluruh *ancestor* dari simpul

tersebut. Operasi ini membutuhkan kompleksitas waktu O(log n). Hal ini mengakibatkan membangun pohon Fenwick dari data memiliki kompleksitas waktu O(n log(n)). Contoh: kita ingin mengubah A_7 dari 11 menjadi 17, maka selain A_7 diubah menjadi 17, pohon juga harus diubah. Pohon diubah dari simpul dengan indeks 7, nilai pada simpul tersebut ditambah dengan 6. Selain itu seluruh *ancestor* dari simpul tersebut (simpul 8 dan 16) juga ditambah dengan 6.

Pohon Fenwick sangat mudah diimplementasikan sebagai array. Implementasi Pohon Fenwick dengan array hanya memiliki kompleksitas ruang O(n). Array dengan indeks ke-i menandakan simpul dengan indeks ke-i. Kita tidak perlu menyimpan informasi tentang keterhubungan pohon karena keterhubungan pohon dapat diperoleh dengan sifat-sifat pohon yang telah dijelaskan di bagian sebelumnya.



Gambar 3.2 Visualisasi pohon dalam bentuk array (sumber: http://community.topcoder.com/i/education/binaryIndexedTrees/BITimg.gif diakses tanggal 16 Desember 2013)

Perhatikan gambar 3.2 di atas untuk menvisualisasikan bagaimana bentuk pohon di dalam array. Setiap interval seolah-olah menjadi simpul dengan indeks yang sejajar dengan kotak yang diarsir. Ayah dari sebuah simpul merupakan interval yang berada di sebelah kirinya.

c. Contoh Implementasi dalam menyelesaikan masalah penjumlahan interval

Pendekatan dengan pohon Fenwick juga memakai prinsip identitas penjumlahan

$\sum_{k=i}^{j} A_k = \sum_{k=0}^{j} A_k - \sum_{k=0}^{i-1} A_k$ untuk mendapatkan penjumlahan interval tertentu. Berikut ini adalah potongan kodenya dalam bahasa C.

```
/* membangun pohon Fenwick */
  int Fenwick[n+1], k, m;
  for (k = 1; k <= n; k++)
    Fenwick[k] = 0;
  for (k = 1; k <= n; k++)
    /* update Fenwick */
    /* m&(-m) menghasilkan lsb(m) */
    for (m = k; m <= n; m += m&(-m))
      Fenwick[m] += A[k];

/* mendapatkan PIA(i,j) */
  int sum = 0;
  for (k = j; k > 0; k -= k&(-k))
    sum += Fenwick[k];
  for (k = i-1; k > 0; k -= k&(-k))
    sum -= Fenwick[k];

/* mengubah nilai A[x] */
  int beda = NILAI_BARU - A[x];
  A[x] = NILAI_BARU;
  for (k = x; k <= n; k += k&(-k))
    Fenwick[k] += beda;
```

Algoritma di atas membangun Pohon Fenwick dalam waktu O(n log (n)). Setelah itu mencari PI dapat dilakukan dalam waktu O(log n). Untuk melakukan pengubahan nilai diperlukan waktu O(log n). Algoritma ini memerlukan ruang tambahan O(n).

IV. VARIANSI POHON FENWICK DAN PERBANDINGANNYA DENGAN SEGMENT TREE

Segment Tree adalah struktur pohon yang sangat efisien untuk permasalahan Penjumlahan Interval, *Segment Tree* juga dapat digunakan untuk permasalahan *Range Minimum Query* (RMQ). *Segment Tree* memiliki kompleksitas waktu O(log n) untuk Penjumlahan Interval, dan memiliki kompleksitas waktu O(log n) untuk pengubahan nilai. Membangun suatu *Segment Tree* memerlukan waktu O(n log (n)). *Segment Tree* memiliki kompleksitas ruang O(n log (n)). Pengubahan nilai pada *Segment Tree* sama seperti pada Pohon Fenwick, yaitu dengan menambahkan perbedaan nilai yang baru dengan nilai sebelumnya. *Segment Tree* dapat melakukan "pengubahan nilai" sekaligus dalam suatu interval (*Range Update*) dalam waktu O(log n), Misalnya $A_i,A_{i+1},...,A_{j-1},A_j$ masing-masing ditambah dengan 3.

Beberapa kelebihan pohon Fenwick daripada *Segment Tree* antara lain:

- Mudah diimplementasi dalam bentuk array
- Memiliki kompleksitas ruang yang lebih baik, yaitu O(n)
- Kode pendek, efisien, elegan, tidak mudah nge-*bug*

Beberapa kelebihan *Segment Tree* daripada pohon Fenwick antara lain:

- Lebih mudah dipahami daripada pohon Fenwick
- Bisa melakukan *Range Update* apapun

Sifat pengubahan nilai sekaligus dalam suatu interval merupakan sifat yang cukup menarik. Terdapat beberapa versi modifikasi dari pohon Fenwick yang dapat melakukan *Range Update* penambahan dan pengurangan, tetapi untuk *Range Update* perkalian dan pembagian belum diketahui versi pohon Fenwick yang dapat melakukannya.

V. KESIMPULAN

Pohon Fenwick sangat berguna untuk persoalan Penjumlahan Interval. Penjumlahan Interval sendiri memiliki banyak kegunaan dalam algoritma yang lebih kompleks. Pohon Fenwick memiliki aplikasi dalam algoritma kompresi data, pohon ini digunakan pertama kali dalam algoritma kompresi data Arithmetic Coding.

VII. UCAPAN TERIMA KASIH

Saya ingin mengucapkan terima kasih kepada Tuhan atas segala yang diberikan-Nya sehingga makalah ini dapat selesai tepat waktu. Saya juga ingin mengucapkan terima kasih kepada Orang tua saya yang telah membesarkan saya sehingga dapat menjadi mahasiswa ITB. Saya juga ingin mengucapkan terima kasih kepada dosen mata kuliah IF-2120 Dr. Ir. Rinaldi Munir dan Dra. Harlili atas ilmu yang telah dibagikannya melalui kuliah sehingga saya mampu menyelesaikan makalah saya ini. Saya juga ingin berterima kasih kepada teman-teman saya atas semangat dan dukungan yang diberikan selama penulisan makalah ini.

REFERENCES

[1] Munir,rinaldi."Diktat Kuliah IF2120 Matematika Diskrit", edisi keempat,Program Studi Teknik Informatika STEI ITB,2006.

[2] Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Hal. 308–423.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, dan Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Hal. 214-217, 253–320.

[4] "Summation" URL: http://planetmath.org/summation diakses tanggal 15 Desember 2013 pukul 18.00 WIB

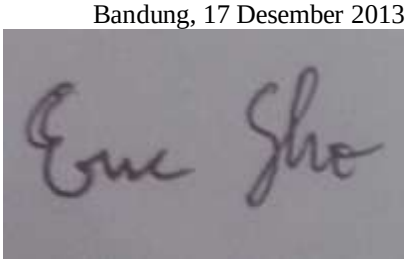
[5] "Binary Indexed Trees" URL: www.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees diakses tanggal 16 Desember 2013 pukul 17.00 WIB

[6] "Bitwise Operators" URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators diakses tanggal 15 Desember 2013 pukul 21.00 WIB

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, 17 Desember 2013



Eric/13512021