Perbandingan Algoritma *Brute Force*, *Divide and conquer*, dan *Dynamic Programming* untuk Solusi *Maximum Subarray Problem*

Reinhard Denis Najogie - 13509097
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13509097@stei.itb.ac.id

Abstrak—*Maximum subarray problem* adalah permasalahan mencari rentang indeks *array* berisi angka dimana jumlah angka pada rentang tersebut bernilai maksimal. Tentunya kita tertarik pada kasus dimana bilangan pada *array* bisa bernilai negatif, karena jika tidak, maka solusi *maximum subarray* problem adalah elemen pertama hingga elemen terakhir *array*. *Subarray* yang diingin kan disini adalah *subarray* yang kontigu, atau dengan kata lain solusi rentang indeks [i..j] berarti indeks i,i+1,i+2,..j. Dalam makalah ini akan dibahas dan dibandingkan tiga pendekatan untuk mencari solusi *maximum subarray* problem, yaitu *brute force*, *divide and conquer*, serta *dynamic programming*.

Kata kunci—*maximum subarray, brute force, divide and conquer, dynamic programming*

I. Pendahuluan

Salah satu struktur data dasar yang sudah banyak dikenal adalah *array*. *Array* adalah kumpulan obyek yang bertipe sama dan memiliki indeks untuk menandakan dan membedakan obyek yang satu dengan yang lain, secara sederhana bisa kita ibaratkan dengan tabel dengan baris dan banyak kolom atau sebaliknya, satu kolom dan banyak baris. Dengan memiliki indeks, operasi pengaksesan obyek ke-i pada *array* bisa dilakukan sangat cepat, yaitu dalam O(1). Sedangkan pencarian nilai obyek *array* bisa dilakukan dengan pencarian linier dengan kompleksitas O(N) atau dengan binary search dengan kompleksitas O(lg N) jika obyek sudah terurut berdasarkan aturan tertentu. *Array* juga bisa memiliki lebih dari satu dimensi, layaknya tabel yang bisa memiliki banyak baris dan banyak kolom. Pada makalah ini kita membatasi hanya akan membahas mm*array* yang berisi obyek bertipe bilangan dan berdimensi satu.

Pada *maximum subarray problem*, kita akan mencari rentang dan jumlah terrbesar dari sebuah *array*. Melalui observasi sederhana akan terlihat bahwa permasalahan ini hanya menarik jika bilangan yang ada pada *array* bisa bernilai negatif, karena jika isi *array* semuanya bilangan bernilai positif atau 0, maka solusi nya adalah keseluruhan *array* tersebut. *Subarray* pada *maximum subarray problem* juga harus kontigu, yang berarti *subarray* berupa suatu rentang yang berurut dan dibatasi dari kiri dan kanan, tidak boleh terputus. Sebagai ilustrasi, perhatikan *array* berikut:

Nilai	1	2	-5	4	7	-2
Indeks	0	1	2	3	4	5

Tabel 1 – Contoh *Array* (1)

Dari contoh *array* di atas, kita bisa melihat bahwa *subarray* dengan nilai maksimal adalah dari indeks 3 hingga indeks 4, yakni *subarray* [4,7] yang total jumlahnya adalah 11. Melalui contoh di atas mungkin kita memiliki hipotesis bahwa kita hanya perlu mencari *subarray* yang tidak memiliki nilai negatif, akan tetapi hipotesis ini tidak benar, seperti kasus yang dapat kita lihat pada *array* berikut:

Nilai	1	5	-3	4	-2	1
Indeks	0	1	2	3	4	5

Tabel 2 – Contoh *Array* (2)

Ternyata *subarray* maksimal adalah dari indeks 0 hingga indeks 3, yakni *subarray* [1,5,-3,4] dengan jumlah 7.

Dapat kita lihat dari kedua ilustrasi di atas bahwa pencarian solusi untuk *maximum subarray problem* tidak sesederhana mencari *subarray* yang tidak mengandung nilai negatif. Oleh karena itu diperlukan analisis lebih dalam dan metode penyelesaian yang lebih baik untuk pemecahan masalah ini. Dalam makalah ini akan dibahas tiga metode, yakni *brute force*, *divide and conquer*, dan *dynamic programming*.

Maximum subarray problem sendiri banyak berguna dalam bidang statistik, terutama pada cabang ilmu yang menganalisis naik-turunnya suatu nilai pada periode waktu tertentu, contohnya adalah grafik harga saham. Dengan mengetahui solusi dari *maximum subarray*

problem maka kita akan mengetahui pada selang waktu kapan harga saham tertentu mengalami kenaikan paling banyak atau penurunan paling banyak. Jika perhitungan ini memerlukan data yang cukup besar, misalnya data pergerakan harga saham selama 20 tahun terakhir, maka diperlukan algoritma yang efisien untuk melakukan perhitungan dalam waktu yang cukup cepat. Dalam bab II akan dibahas algoritma-algoritma untuk menyelesaikan permasalahan ini beserta perbandingan kompleksitasnya.

II. Dasar Teori

Pada bab ini akan dijelaskan mengenai prinsip pendekatan yang akan kita gunakan untuk mencari solusi dari *maximum subarray* problem, yakni *brute force*, *divide and conquer*, dan *dynamic programming*.

Pertama, *brute force*. *Brute force* adalah teknik paling sederhana untuk menyelesaikan permasalahan komputasi pada umumnya. Prinsip *brute force* adalah mencoba semua kemungkinan dari instansi masalah dan mencari solusi yang diinginkan, tanpa analisis yang dalam. Solusi dengan *brute force* biasanya mudah dimengerti dan mudah diimplementasi, tetapi seringkali kurang efisien secara waktu maupun memori yang digunakan.

Kedua, *divide and conquer*. *Divide and conquer* adalah metode penyelesaian masalah dengan membagi masalah utama menjadi masalah-masalah yang lebih kecil (tahap *divide*). Masalah-masalah yang lebih kecil ini akan diselesaikan dan digabung untuk menyelesaikan masalah yang sesungguhnya (tahap *conquer*/*combine*). *Divide and conquer* pada implementasinya seringkali harus menggunakan algoritma yang rekursif, karena itu untuk merancang algoritma *divide and conquer* kita seharusnya sudah mahir merancang algoritma rekursif.

Ketiga, *dynamic programming*. Metode ini juga membagi suatu masalah menjadi beberapa submasalah. Yang membedakan *dynamic programming* dengan *divide and conquer* adalah pada *dynamic programming*, sub-masalah ini tidak digabung untuk menyelesaikan masalah secara keseluruhan, melainkan disimpan untuk tidak dihitung lagi jika diperlukan perhitungan sub-masalah yang sama nantinya. Ada dua properti yang diperlukan suatu permasalahan untuk memiliki solusi *dynamic programming*, yaitu *overlapping subproblem* dan *optimal substructure*. *Overlapping subproblem* adalah kondisi dimana kita memanggil *subproblem* yang sama berkali-kali untuk menyelesaikan problem utama. *Optimal substructure* adalah jika solusi optimal untuk problem utama bisa dibentuk dari solusi optimal untuk *subproblem* yang ada. Properti ini juga diperlukan oleh algortima *greedy* untuk menghasilkan solusi yang benar.

III. Solusi *Maximum Subarray Problem*

A. Solusi Brute Force

Sekarang kita akan merancang solusi paling sederhana untuk persoalan *maximum subarray* problem—*brute force*, sesuai dengan prinsip KIS (Keep it Simple) untuk mencari solusi paling mudah terlebih dahulu untuk suatu persoalan sebelum mencari solusi yang lebih sulit.

Karena batasan bahwa solusi harus berupa *subarray* yang kontigu, maka solusi *brute force* nya adalah dengan melakukan loop untuk semua kemungkinan indeks awal dan indeks akhir. Karena untuk mencari kemungkinan indeks awal memerlukan N pencarian (dimana N=jumlah elemen pada *array*), dan untuk setiap N indeks awal diperlukan pencarian paling banyak N-1 indeks akhir, maka algoritma ini akan berjalan dengan kompleksitas O(N^2).

Pseudocode untuk algoritma ini adalah sebagai berikut:

```
function brute_force(A : array of integer, n : integer) :
<integer, integer, integer>

var
        i, j, maks, sum, l, r : integer
begin
        maks = -∞
        sum = 0
        l = 0
        r = 0

        for i = 0 to n-1
                sum = 0
                for j = i to n-1
                        sum = sum + A[j]
                        if (sum > maks)
                                l = i
                                r = j
                                maks = sum

        return (l, r, maks)
end
```

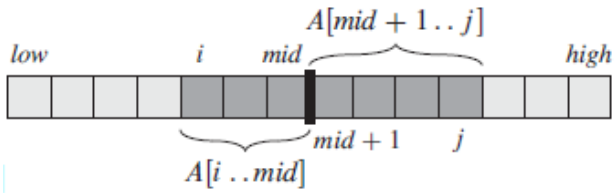
Gambar 1 – Pseudocode algoritma *brute force*

B. Solusi Divide and Conquer

Setelah merancang algoritma *brute force* di atas, kita tahu bahwa kompleksitas algoritmanya adalah O(N^2). Kita akan mencoba merancang algoritma yang lebih kecil kompleksitasnya dengan metode *divide and conquer*.

Sesuai dengan prinsip *divide and conquer*, kita mulai dengan membagi permasalahan utama kita menjadi submasalah-submasalah yang lebih mudah diselesaikan.

Misalkan kita ingin mencari *maximum subarray* dari sebuah *array* A, yaitu A[*low*..*high*] dimana *low* dan *high* menandakan indeks awal dan akhir A. Kita akan membagi *array* ini menjadi 2 pada titik *mid*, yaitu pertengahan antara *low* dan *high*, sehingga kita akan dapatkan A[*low*..*mid*] dan A[*mid*+1..*high*]. Pembagian *subarray* ini dapat dilihat dengan ilustrasi berikut:



Gambar 2 – *Subarray* saat pembagian dengan algortima *divide and conquer*

Dari gambar di atas, kita bisa lihat bahwa ada tiga kemungkinan letak *subarray* dari A. Untuk setiap *subarray* A[i..j] dari *array* A[*low*..*mid*], bisa berlaku:

1. Terletak seluruhnya pada *subarray* A[*low*..*mid*], sehingga *low* <= i <= j <= *mid*,
2. Terletak seluruhnya pada *subarray* A[*mid*+1..*high*], sehingga *mid* < i <= j <= *high*,
3. Melewati titik *mid*, sehingga *low* <= i <= *mid* < j <= *high*.

Oleh karena itu *maximum subarray* dari A[*low*..*high*] haruslah terletak pada tepat satu dari tiga kemungkinan di atas. Untuk mencari solusi untuk permasalahan utama, kita harus mencari *subarray* yang memiliki jumlah maksimal dari ketiga kemungkinan di atas. Untuk dua kemungkinan pertama, kita bisa menggunakan rekursif biasa, karena kedua kasus di atas adalah submasalah dari permasalahan utama kita. Sedangkan untuk kasus terakhir kita harus buat satu fungsi tambahan untuk menghitungnya.

Sekarang kita akan mencari cara untuk menghitung maximum subararay pada kasus terakhir. Jika kita perhatikan pada gambar 2, *subarray* yang melewati titik mid dapat kita bagi dua menjadi A[i..mid] dan A[mid+1..j], dimana low <= i <= mid < j <= high. Dengan ini kita hanya perlu *maximum subarray* A[i..mid] dan A[mid+1..j] dan menggabungkan keduanya. Cara ini dapat diimplementasikan sebagai berikut:

```
function max_crossing_subarray(A : array of integer, low : integer, mid : integer, high : integer) : <integer, integer, integer>

var
        lsum, rsum, maxl, maxr, sum, i : integer

begin
        lsum = -∞
        sum = 0

        for i = mid downto low
                sum = sum + A[i]
                if (sum > lsum)
                        lsum = sum
                        maxl = i

        rsum = -∞
        sum = 0

        for i = mid + 1 to high
                sum = sum + A[i]
                if(sum > rsum)
                        rsum = sum
                        maxr = i

        return (maxl, maxr, lsum + rsum)
end
```

Gambar 3 – Pseudocode pencarian *maximum subarray* yang melewati titik mid

Setelah memiliki fungsi untuk mencari *maximum subarray* yang melewati titik mid, sekarang kita dapat merancang algortima *divide and conquer* yang menyelesaikan masalah utamanya. Seperti yang sudah kita ketahui ada tiga kasus dimana kita bisa mendapatkan *maximum subarray*. Dua kasus pertama merupakan submasalah dari masalah utama, karena itu kita bisa langsung menghitungnya dengan rekursif, sementara untuk kasus terakhir karena kita sudah punya fungsi untuk menghitungnya maka kita tinggal memasukkan nya pada fungsi utama kita. Dari ketiga kasus kemungkinan *subarray* ini, kita akan mencari nilai jumlah terbesar dan me-return nilai itu. Kita bisa mengimplementasikan algoritma utama *divide and conquer* nya sebagai berikut:

```
function max_subarray(A : array of integer, low : integer, high : integer) : <integer, integer, integer>

var
        mid, llow, lhigh, lsum, rlow, rhigh, rsum, clow, chigh, csum : integer

begin
        if(low == high)
                return (low, high, A[low])
        else
                mid = (low + high)/2
                (llow, lhigh, lsum) = max_subarray(A, low, mid)
                (rlow, rhigh, rsum) = max_subarray(A, mid+1, high)
```

```
                (clow, chigh, csum) = max_crossing_subarray(A, low, mid, high)

                if(lsum >= rsum and lsum >= csum)
                        return (llow, lhigh, lsum)
                else if(rsum >= lsum and rsum >= csum)
                        return (rlow, rhigh, rsum)
                else
                        return (clow, chigh, csum)


end
```

Gambar 4 – Pseudocode algoritma *divide and conquer*

Sekarang kita akan menganalisis kompleksitas waktu dari algoritma *divide and conquer* yang telah kita buat. Untuk sebuah *array* dengan ukuran N, jika N=1, maka kita dapatkan T(1) = O(1), sementara untuk kasus lebih umum yakni N > 1, kita akan men-*conquer* dua submasalah yang ukurannya setengah dari masalah utama dan satu masalah tambahan yang ukurannya sama dengan masalah, sehingga kita akan dapatkan T(N) = 2T(N/2) + O(N). Menggabungkan kedua persamaan di atas, kita dapatkan persamaan rekurens untuk T:

T(N) = O(1), untuk n=1
= 2T(N/2) + O(n), untuk n>1

Yang dengan *master theorem* (atau metode lain untuk menghitung kompleksitas) bisa kita dapatkan T(N) = O(N lg N).

Bisa kita lihat bahwa metode *divide and conquer* bisa menghasilkan solusi dengan kompleksitas yang lebih kecil daripada metode *brute force* untuk *maximum subarray problem.*

C. Solusi Dynamic Programming

Kita berhasil mengurangi kompleksitas waktu solusi dari O(N^2) menjadi O(N lg N) dengan metode *divide and conquer*. Sekarang kita akan mencoba metode lain untuk menyelesaikan *maximum subarray problem*, metode itu ialah *dynamic programming*. Tentunya kita berharap bisa mendaptkan kompleksitas waktu yang lebih kecil dari hasil yang sudah kita dapatkan.

Untuk merancang algoritma *dynamic programming* untuk solusi *maximum subarray* problem, kita akan mulai dengan mencari *optimal substructure* dari permasalahan ini.

Jika kita diberikan *array* masukan A, dan kita mengetahui *maximum subarray* dari A[0..i] dan jumlah dari elemen-elemen *subarray* itu sendiri. Dengan informasi ini, kita harus mencari *maximum subarray* dari A[0..i+1] dengan jumlah elemen paling besar. Inilah permasalahan *optimal substructure* yang harus kita cari solusinya.

Sekarang misalkan A[j..k] adalah *maximum subarray*, t adalah jumlah *array* A[j..i] dan s adalah jumlah dari elemen-elemen *maximum subarray*. Jika t+a[i] > s, maka *maximum subarray* sekarang adalah a[j..i+1] dan s kita isi dengan nilai t. Jika t+a[i] < 0, kita tau dari batasan permasalahan bahwa *subarray* harus kontigu berarti kita tidak bisa menyertakan *subarray* A[j..i+1] pada *maximum subarray* karena *subarray* kita akan berkurang jumlahnya. Oleh karena itu jika t+a[i] < 0 kita isi t dengan 0 dan set batas kiri *subarray* menjadi i+1.

Sebagai ilustrasi algoritma di atas perhatikan contoh berikut:

s = -∞, t = 0, j = 0, batas = (0,0)
Langkah:
1. (1 2 -5 4 7 -2
2. (1)| 2 -5 4 7 -2
3. (1 2)|-5 4 7 -2
4. 1 2 -5(| 4 7 -2
5. 1 2 -5 (4)| 7 -2
6. 1 2 -5 (4 7)|-2
7. 1 2 -5 (4 7) -2|

Penjelasan:
1. Kondisi awal *array*, batas kiri berada pada elemen pertama *array*.
2. Kita set t=1, karena t>s, set s=1, dan batas=(0,0)
3. Kita set t=3, karena t>s, set s=3, dan batas=(0,1)
4. Kita set t=-2, karena t<0, set t=0 dan j=3
5. Kita set t=4, karena t>s, set s=4, dan batas=(3,3)
6. Kita set t=11, karena t>s, set s=11, dan batas=(3,4)
7. Kita set t=9, karena t<s, maka tidak dilakukan apa-apa

Kita dapatkan *maximum subarray* ada adalah [4,7] pada indeks 3..4 dengan jumlah 11. Implementasi algoritma di atas dapat kita lihat pada pseudocode berikut:

```
function dp(A : array of integer) : <integer, integer, integer>

var
        low, high, sum, t, i, j : integer
begin
        low = 0
        high = 0
        sum = -∞
        t=0
        j=0

        for i = 0 to n-1
                t = t + A[i]
                if (t > sum)
                        low = j
                        high = i + 1
                        sum = t
                else if(t < 0)
```

```
            t = 0
            j = i +1

        return (low, high, sum)

end
```

Gambar 5 – Pseudocode algoritma *dynamic programming*

Bisa kita lihat dengan mudah bahwa kompleksitas algoritma *dynamic programming* di atas adalah O(N) dimana N adalah jumlah elemen pada *array*. Dengan begitu solusi dengan *dynamic programming* adalah solusi yang terbaik secara kompleksitas dibandingkan solusi *brute force* O(N^2) dan solusi *divide and conquer* O(N lg N) yang sudah kita temukan sebelumnya.

IV. Kesimpulan

Setelah melihat perbandingan kompleksitas dan running time dari ketiga algoritma yang sudah dibahas pada bab-bab sebelumnya, dapat diambil kesimpulan bahwa *dynamic programming* adalah solusi paling efisien untuk *maximum subarray* problem. Mungkin ada diantara pembaca ada yang menanggap perbaikan kompleksitas dari solusi maximum subaray yang sudah dibahas di atas tidak signifikan, untuk hal ini penulis memberikan kesempatan untuk pembaca bereksperimen pada komputernya masing-masing untuk mengimplementasikan ketiga algoritma di atas dan membandingkan waktu jalan program yang sesungguhnya. Jika pembaca membandingkan waktu jalan program untuk input yang cukup besar, katakanlah untuk N > 10000, maka penulis yakin akan terlihat perbedaan waktu yang cukup signifikan antara ketiga algoritma di atas (terutama algoritma *brute force* jika dibandingkan dengan algoritma *dynamic programming*)

Secara umum permasalahan pada ilmu komputer dapat memiliki banyak solusi dengan metode yang berbeda-beda. Karena itu sebaiknya kita jangan cepat puas jika sudah mendapatkan solusi untuk permasalahan tertentu. Siapa yang tahu, kita bisa menemukan algoritma yang lebih efisien untuk diterapkan.

Referensi

[1] Cormen, Leiserson, Rivest, dan Stein, "Introduction to Algorithms", 3rd ed. Massachusetss: The MIT Press, 2009, Bab 4 – "*Divide*-and-*Conquer*" dan Bab 15 – "*Dynamic Programming*".

[2] Http://en.wikipedia.org/wiki/*Divide*_and_*conquer*_algorithm - diakses tanggal 3 Desember 2011 pukul 20.00

[3] Http://en.wikipedia.org/wiki/*Dynamic_programming* - diakses tanggal 3 Desember 2011 pukul 20.00

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, 4 Desember 2011
Ttd

Reinhard Denis Najogie (13509097)