

# Penggunaan Algoritma *Divide and Conquer* dalam Penyelesaian *Subset Guessing Problem*

Bimo Adityarahman Wiraputra

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

13517004@std.stei.itb.ac.id

**Abstrak**—*Subset Guessing Problem* merupakan suatu permasalahan dimana seorang pemain mencoba menebak isi dari suatu himpunan bagian tertentu dari bilangan asli yang tidak lebih dari suatu nilai  $n$ , dengan mengajukan pertanyaan berbentuk *query* suatu subhimpunan lainnya yang dijawab dengan kardinalitas himpunan irisan antara himpunan bagian *query* dan subhimpunan yang ditebak tersebut. Masalah ini dapat diselesaikan dengan pendekatan iteratif biasa dan dapat didekati dengan pendekatan algoritma *divide and conquer*. Makalah ini mempersembahkan beberapa algoritma yang dapat digunakan dalam penyelesaian masalah ini dan menganalisis kompleksitas waktu rata-rata yang digunakan masing-masing algoritma.

**Kata Kunci**—tebakan; *query*; himpunan bagian; *divide and conquer*

## I. PENDAHULUAN

*Competitive Programming* merupakan salah satu format kontes *programming* dimana pesertanya diminta membuat suatu program dengan spesifikasi tertentu. Biasanya program harus menerima input dan memprosesnya untuk menghasilkan suatu output yang benar atau memenuhi suatu kondisi. Pertanyaan dalam kontes *competitive programming* biasanya membutuhkan pemahaman algoritma, struktur data, maupun matematika untuk dapat diselesaikan sesuai dengan batasan waktu dan memori yang diberikan untuk program, sehingga soal *competitive programming* dapat menjadi sumber pembelajaran dan eksplorasi algoritma.

Salah satu soal yang diberikan dalam suatu kontes *competitive programming* bernama IEEEExtreme 2018 bernama Troll Coder dimana peserta harus mencari suatu *string* biner rahasia dengan mengajukan beberapa *query* berbentuk *string* biner yang dijawab dengan banyak angka satu pada hasil operasi xor antara *string* rahasia dengan *string query*. Program yang dibuat peserta dinilai berdasarkan seberapa sedikit *query* yang diajukan sampai program dapat menebak *string* rahasia tersebut.

*Subset guessing problem* adalah penyederhanaan dari permasalahan tersebut dimana kita tidak membutuhkan pengetahuan mengenai operasi dalam *string*. Kita diberitahu

suatu bilangan asli  $n$ , dan ada suatu himpunan bagian rahasia dari himpunan bilangan asli yang tidak lebih besar dari  $n$ . Tujuan kita adalah menebak himpunan rahasia tersebut dengan mengajukan sesedikit mungkin *query* yang berbentuk dengan suatu himpunan bagian lainnya, yang akan dijawab dengan tingkat kecocokan kedua himpunan bagian tersebut, atau banyak anggota dari irisan kedua himpunan.

Dalam penyelesaian permasalahan tersebut, makalah ini akan membahas beberapa algoritma yang digunakan untuk menentukan isi dari *query* pertanyaan yang ditanyakan berdasarkan nilai dari  $n$  dan jawaban dari *query* sebelumnya. Algoritma yang paling naif dapat menggunakan iterasi sederhana saja untuk menyelesaikan masalah ini. Selain itu, dalam makalah ini dapat dibuat lebih efisien dengan menggunakan algoritma *divide and conquer*. Makalah akan menghitung ekspektasi banyak *query* yang dibutuhkan masing-masing pendekatan, dibantu dengan hasil eksperimen secara langsung dengan jawaban yang dibangkitkan secara acak.

## II. DASAR TEORI

### A. Teori Himpunan Dasar

Himpunan adalah suatu objek matematika yang merupakan suatu kumpulan dari beberapa objek. Sebagai salah satu konsep paling dasar dalam teori matematika, definisi dari himpunan telah berganti seiring zaman. Secara formal dalam teori himpunan modern, biasanya himpunan didefinisikan dalam aksioma ZFC (*Zermelo–Fraenkel set theory*), dimana semua sifat-sifat mengenai himpunan diturunkan dari aksioma yang ada didalamnya. Secara informal, himpunan hanyalah kumpulan dari objek-objek. Untuk setiap objek yang ada, objek tersebut dapat menjadi anggota dari suatu himpunan maupun tidak (sesuai batasan yang ada berdasarkan aksioma yang ada). Dua himpunan dikatakan sama apabila anggota dari suatu himpunan merupakan anggota dari himpunan lainnya, dan begitu juga sebaliknya. Beberapa operator dasar yang ada pada himpunan seperti operator anggota dari ( $\in$ ), himpunan bagian dari ( $\subseteq$ ) yaitu saat semua anggota dari suatu himpunan merupakan anggota dari himpunan satunya, operasi

komplemen ( $\bar{\phantom{x}}$ ) yang menghasilkan himpunan yang mengandung semua objek yang bukan merupakan anggota dari himpunan awal, operasi gabungan ( $\cup$ ) yang menghasilkan himpunan yang mengandung semua objek yang merupakan anggota dari salah satu operandnya, dan irisan ( $\cap$ ) yang mengandung semua objek yang merupakan anggota dari kedua operandnya. Beberapa sifat yang sering digunakan dalam manipulasi himpunan adalah sifat komutatif dan asosiatif dari operasi gabungan dan irisan, juga sifat distributif dari operasi gabungan terhadap irisan dan sebaliknya.

Dalam himpunan terdefinisi konsep kardinalitas, yaitu suatu ukuran banyaknya anggota dari suatu himpunan. Kardinalitas dari himpunan kosong  $\{\}$  adalah 0, sedangkan kardinalitas dari himpunan  $\{1, 2, \dots, n\}$  dengan  $n$  adalah suatu bilangan asli adalah  $n$ . Jika terdapat suatu fungsi bijektif dari suatu himpunan  $S$  ke himpunan  $\{1, 2, \dots, n\}$ , maka  $|S|$  juga bernilai  $n$ . Sedangkan apabila tidak terdapat  $n$  seperti itu, maka himpunan  $S$  disebut himpunan tak hingga. Salah satu sifat mendasar yang sering digunakan dari kardinalitas adalah Hukum De Morgan yaitu sifat  $|X \cup Y| + |X \cap Y| = |X| + |Y|$ .

### B. Algoritma Divide and Conquer

Algoritma *divide and conquer* merupakan keluarga algoritma yang memiliki ciri terbagi menjadi beberapa tahapan :

- 1) *Divide*, dimana suatu permasalahan dibagi menjadi beberapa sub permasalahan yang seragam dengan permasalahan semula dan dapat diselesaikan dengan logika algoritma yang sama
- 2) *Conquer*, memecahkan setiap sub permasalahan secara rekursif dengan memanggil algoritma ini lagi
- 3) *Combine*, yakni menggabungkan setiap solusi sub permasalahan untuk mendapatkan solusi dari permasalahan semula.

Contoh-contoh permasalahan yang dapat diselesaikan dengan algoritma *divide and conquer* adalah *sorting* dengan *mergesort* maupun *quicksort*, menghitung *min* maupun *max* dari suatu *array*, maupun melakukan perkalian matriks dengan dimensi yang besar.

Algoritma *divide and conquer* biasanya dikategorikan menjadi algoritma *easy split*, *hard join* dan algoritma *hard split*, *easy join*. Pengkategorian ini dilihat dari kesulitan tahapan *divide* dan tahapan *combine*. Sebagai contoh, dalam algoritma *mergesort*, tahapan *divide* mudah untuk dieksekusi karena *array* yang mau *disort* hanya dibagi menjadi setengah bagian awal dan setengah bagian akhir, sedangkan tahapan *combinenya* lebih kompleks karena harus menggabungkan dua *array* yang terurut, maka algoritma *mergesort* dianggap sebagai algoritma *easy split*, *hard join*. Sedangkan algoritma *quick sort* dianggap sebagai algoritma *hard split*, *easy join* karena tahapan *dividenya* mengharuskan kita mengiterasi dan menukar posisi beberapa anggota *array* dan bagian *joinnya*, kita hanya menggabungkan dua *array* yang sudah terurut

dimana salah satu *array* semua anggotanya lebih kecil dari anggota *array* kedua.

Karena algoritma *divide and conquer* memanggil algoritma yang sama secara rekursif, biasanya didapat persamaan kompleksitas waktu dalam bentuk yang rekursif. Kompleksitas waktu ini biasanya dapat diselesaikan dengan Teorema Master yang memecahkan hubungan rekursi kompleksitas waktu. Teorema Master menyebutkan bahwa apabila ada suatu fungsi kompleksitas  $T(n)$  yang memiliki hubungan rekursi :

$T(n) = aT(n/b) + cn^d$ , dimana dalam hal ini  $n = b^k$ ,  $k$  bilangan asli, maka didapat

$$T(n) \text{ adalah : } \begin{array}{ll} O(n^d) & \text{jika } a < b^d \\ O(n^d \log n) & \text{jika } a = b^d \\ O(n^{\log_b a}) & \text{jika } a > b^d \end{array}$$

### III. ALGORITMA PENYELESAIAN MASALAH

Secara formal, *Subset guessing problem* adalah suatu permasalahan dimana kita diberi tahu suatu nilai bilangan asli  $n$ . Terdapat suatu himpunan rahasia  $X$  yang tidak diketahui kita, tapi diketahui merupakan suatu himpunan bagian dari himpunan  $\{1, 2, \dots, n\}$ . Kita dapat mengajukan pertanyaan dalam bentuk *query* berbentuk suatu himpunan  $Y$ , himpunan bagian juga dari himpunan  $\{1, 2, \dots, n\}$ . *Query* akan dijawab dalam bentuk nilai dari  $|X \cap Y|$ , yaitu banyak anggota dari himpunan irisan antara  $X$  dan  $Y$ . Berdasarkan informasi yang didapat dari jawaban *query* tersebut, kita bertujuan untuk menebak isi dari himpunan  $X$  dengan tepat menggunakan *query* dengan banyak seminimal mungkin.

Dalam pendekatan mencari cara penyelesaian *subset guessing problem*, mungkin terpikir untuk menjalankan algoritma *brute force* untuk mencari solusi. Terdapat  $2^n$  kemungkinan solusi dari himpunan  $X$ . Untuk mengecek bahwa suatu himpunan  $Y$  adalah himpunan  $X$ , kita dapat menggunakan dua *query* untuk mengecek  $|X \cap Y| = |Y|$ , yang membuktikan bahwa semua anggota  $Y$  adalah anggota dari  $X$ , dan  $|X \cap Y^c| = 0$ , yang membuktikan bahwa semua anggota  $X$  adalah anggota  $Y$ . Dengan optimasi, kita membutuhkan maksimal  $2^n$  *query* untuk mencari himpunan rahasia  $X$ , dan apabila diuji secara acak, rata-rata akan dibutuhkan  $2^{n-1}$  *query* untuk menemukan himpunan rahasia tersebut.

Algoritma *brute force* memberikan solusi yang sangat tidak efektif, yaitu secara eksponensial. Dalam makalah ini, algoritma *brute force* tidak akan diujikan karena kompleksitas yang terlalu besar.

Pendekatan lain yang mungkin adalah dengan mencari apakah suatu nilai  $i$  merupakan anggota dari himpunan  $X$  untuk semua  $i$  dalam  $\{1, 2, \dots, n\}$ . Cara termaif yang mungkin

terpikir adalah dengan menanyakan *query* berbentuk  $\{i\}$ , karena hasil dari *query*-nya,  $|X \cap \{i\}|$ , akan bernilai satu apabila  $i$  anggota dari  $X$  dan nol jika tidak. Maka kita bisa menentukan keanggotaan setiap bilangan, sehingga didapat algoritma naif:

#### A. Algoritma Naif

```
def solveA:
    ans = {}
    for i = 1 to n:
        if ask({i}) == 1:
            ans.add(i)
    return ans
```

Karena nilai  $i$  pasti diiterasi untuk semua bilangan yang tidak lebih dari  $n$ , maka algoritma ini menggunakan tepat  $n$  *query* untuk semua kasus.  $P(n) = n$  dimana  $P(n)$  dalam konteks makalah ini menandakan rata-rata banyak *query* yang dibutuhkan algoritma untuk menentukan secara pasti nilai dari himpunan  $X$ .

Untuk mempercepat algoritma iterasi di atas, kita dapat menggunakan observasi bahwa jika kita mengajukan *query* dengan isi dua bilangan berbeda, mengasumsikan kemungkinan mereka terkandung dalam  $X$  atau tidak adalah sama, maka kemungkinan muncul angka nol atau dua dalam *query* tersebut adalah  $1/4 + 1/4 = 1/2$ . Jika dalam *query* muncul angka nol maupun angka dua, jelas bahwa itu menandakan bahwa kedua bilangan tidak masuk (atau keduanya masuk) ke dalam himpunan  $X$ . Maka perhatikan bahwa ada  $1/2$  kemungkinan kita memproses dua bilangan dengan satu *query* saja, yang mengoptimasi algoritma di atas. (Apabila *query* menghasilkan angka satu, maka dicek salah satu bilangan dan jawaban diisi seperti algoritma naif. Bilangan yang satunya lagi dihitung dari hasil *query* dua bilangan apakah masuk atau tidak ke dalam  $X$ ).

#### B. Algoritma Pemilihan Dua

```
def solveB:
    ans = {}
    i = 1
    while i <= n:
        x = ask({i, i+1})
        if x == 2:
            ans.add(i)
            ans.add(i+1)
        elif x == 1:
            if ask({i}) == 1:
                ans.add(i)
            else ans.add(i+1)
        i += 2
    if i <= n:
        if ask({n}) == 1:
```

```
        ans.add(n)
    return ans
```

Kasus terburuk dari algoritma di atas adalah saat jawaban *query* hanya menghasilkan nilai satu, sehingga dibutuhkan dua *query* untuk setiap dua bilangan, atau dibutuhkan total  $n$  *query*. Sedangkan kasus terbaik dari algoritma di atas adalah saat jawaban *query* tidak pernah menghasilkan nilai satu sehingga dibutuhkan satu *query* untuk setiap dua bilangan, atau dibutuhkan  $1/2 * n$  *query*.

Dalam ekspektasi peluang, karena kita mengecek keanggotaan dari dua bilangan dengan satu *query* dengan probabilitas  $1/2$ , dan dua *query* dengan probabilitas  $1/2$ , maka didapat ekspektasi dari banyak *query*  $P(n) = 1/2 + 2 * 1/2 + P(n-2) = 3/2 + P(n-2)$ , maka didapat  $P(n) \approx 3/4 * n$ .

Untuk memperbaiki lagi dari algoritma di atas, perhatikan bahwa setelah mendapati hasil *query* dua bilangan menghasilkan nilai satu, kita tidak perlu langsung mengecek nilai per satuan. Kita dapat mengganti salah satu bilangan saja dari bilangan yang baru dicek, dan mengajukan *query* dua bilangan lagi. Lakukan ini sampai ditemukan jawaban nol atau dua (atau tidak ada lagi bilangan yang bisa dicek). Maka perhatikan bahwa bilangan pertama dan terakhir diketahui nilainya dari dari *query* terakhir yang diajukan, dan untuk semua bilangan diantaranya, karena hasil *query*-nya satu, maka status keanggotaannya berbeda dengan bilangan pertama, karenanya kita dapat mengetahui status keanggotaannya. Apabila tidak ada lagi bilangan yang bisa dicek, terpaksa kita mengecek salah satu bilangan untuk mengetahui status keanggotaan semua bilangan sisanya.

#### C. Algoritma Pemilihan Dua Iteratif

```
def solveC:
    ans = {}
    i = 1
    while i <= n:
        j = i + 1
        flag = False
        while j <= n and not flag:
            x = ask({i, j})
            if x == 1:
                ++j
                continue
            elif x == 0:
                for k = i+1 to j-1:
                    ans.add(k)
            else:
                ans.add(i)
                ans.add(j)
            i = j+1
            flag = True
```

```

if not flag:
    if ask({i}) == 0:
        for k = i+1 to n:
            ans.add(k)
    else:
        ans.add(i)

```

Kasus terburuk dari algoritma di atas adalah saat semua *query* menghasilkan nilai satu, sehingga kita harus mengecek semua pasangan bilangan dengan satu, dan mengecek bilangan satu, dengan total *query* sebanyak  $n$ . Kasus terbaiknya sama dengan algoritma B dimana setiap dua bilangan diproses dengan satu *query*. Maka dibutuhkan total  $1/2 * n$  *query*.

Dalam ekspektasi peluang, kita memproses dua bilangan dengan satu *query* dengan kemungkinan  $1/2$ , tiga bilangan dengan dua *query* dengan kemungkinan  $1/4$ , empat bilangan dengan tiga *query* dengan kemungkinan  $1/8$ , dan seterusnya. Maka didapat ekspektasi,

$$\begin{aligned}
 P(n) &= (1/2) * (1 + P(n-2)) + (1/4) * (2 + P(n-3)) + \dots \\
 &= 1/2 + 1/4 + \dots + 1/2 * P(n-2) + 1/4 * (1 + P(n-3)) + \dots \\
 &\approx 1 + 1/2 * P(n-2) + 1/2 * P(n-1)
 \end{aligned}$$

Karena  $P(1) = 1, P(2) = 2$ , perhatikan bahwa aproksimasi  $P(n) \approx 2/3 * n$  mendekati nilai awal dan memenuhi persamaan rekurens di atas, maka didapat ekspektasi  $2/3 * n$ .

Dalam algoritma-algoritma iterasi di atas, program mencari solusi untuk suatu sub permasalahan khusus (satu bilangan pertama, dua bilangan pertama, maupun beberapa bilangan pertama), maka algoritma di atas mungkin pantas kita sebut sebagai algoritma *decrease and conquer* yang berkurang sebagai suatu konstan atau pun sebanyak suatu variabel. Selain algoritma di atas, kita juga dapat menggunakan algoritma *divide and conquer* untuk menyelesaikan permasalahan *subset guessing problem*. Misal fungsi `recurD(start, end, cnt)` merupakan fungsi yang mengembalikan himpunan anggota  $X$  yang berada di dalam range dari start sampai end, dengan cnt menandakan banyak anggota  $X$  diantara start sampai end. Maka kita mendapat kasus dasar saat banyak anggota himpunan  $X$  diantara start sampai end ada sebanyak nol ataupun  $end - start + 1$ , maka kita dapat langsung mengambil tidak sama sekali maupun semua dari bilangan di range tersebut, lalu kita tinggal menggabungkan jawaban dari rekursinya.

### C. Algoritma Divide and Conquer

```

def recurD(start, end, cnt):
    ans = {}

    if end < start return {}
    if cnt == 0 return ans

```

```

if cnt == end - start + 1:
    for i = start to end:
        ans.add(i)
    return ans

mid = (start + end) / 2
x = ask(start, mid)
ans1 = recurD(start, mid, x)
ans2 = recurD(mid+1, end, cnt-x)

return ans1 + ans2

```

Kasus terburuk dari algoritma di atas adalah saat semua pemanggilan rekursif, tidak ada yang bernilai nol ataupun  $end - start + 1$ , sehingga untuk semua segmen dipanggil, banyak *query*  $n$ . Kasus terbaik dari algoritma di atas adalah saat langsung dipotong di awal, yaitu saat cnt di awal bernilai nol maupun  $n$ , banyak *query* sebanyak satu.

Sedangkan rumus rekursi ekspektasi banyak *query* yang digunakan algoritma *divide and conquer* ini cukup kompleks karena mempertimbangkan peluang yang saling berkaitan. Penulis merasa tidak mampu menghitung secara teoritis fungsi ekspektasinya sehingga membandingkan hasil algoritma *divide and conquer* secara eksperimen.

## IV. HASIL EKSPERIMENTASI

Penulis melakukan eksperimentasi menggunakan kode C++ agar efisien dalam penggunaan waktu. Program menerima *input* nilai  $n$ , dan  $m$ , yaitu banyak eksperimentasi untuk setiap cara, yang nanti hasilnya akan dirata-ratakan untuk mendapat nilai sampel dari banyak *query* yang ditanyakan. Berikut adalah beberapa eksperimentasi dengan nilai  $n$  yang bervariasi.

A.  $n = 10, m = 1000$  :

Cara A : 10  
 Cara B : 7.457  
 Cara C : 6.896  
 Cara D : 7.38

B.  $n = 20, m = 1000$  :

Cara A : 20  
 Cara B : 14.918  
 Cara C : 13.573  
 Cara D : 14.776

Cara C : 1333.94

Cara D : 1438.38

C.  $n = 50, m = 1000$  :

Cara A : 50

Cara B : 37.432

Cara C : 33.52

Cara D : 36.996

D.  $n = 100, m = 1000$  :

Cara A : 100

Cara B : 74.976

Cara C : 67.041

Cara D : 74.07

E.  $n = 200, m = 1000$  :

Cara A : 200

Cara B : 150.12

Cara C : 133.572

Cara D : 147.901

F.  $n = 500, m = 1000$  :

Cara A : 500

Cara B : 375.003

Cara C : 333.64

Cara D : 360.343

G.  $n = 1000, m = 1000$  :

Cara A : 1000

Cara B : 749.974

Cara C : 666.938

Cara D : 719.776

H.  $n = 2000, m = 1000$  :

Cara A : 2000

Cara B : 1500.8

I.  $n = 5000, m = 1000$  :

Cara A : 5000

Cara B : 3750.81

Cara C : 3332.36

Cara D : 3677.61

#### J. Analisis Kompleksitas

Melihat hasil data, banyak *query* rata-rata yang digunakan dalam algoritma A, B, dan C memenuhi prediksi dari hitungan matematika di bagian sebelumnya, yaitu sekitar  $n$ ,  $3/4 * n$ ,  $2/3 * n$ . Sedangkan melihat data, kita lihat banyak *query* rata-rata yang digunakan dalam algoritma D ada sekitar  $0.72 * n$  terutama untuk nilai  $n$  yang mendekati perpangkatan dua.

Walaupun semua algoritma mempunyai kompleksitas kasus terburuk yang sama, jika diurutkan melalui keefektifan rata-ratanya, kita miliki algoritma C merupakan algoritma tercepat, diikuti algoritma D yang merupakan algoritma *divide and conquer*, lalu algoritma B dan akhirnya algoritma A.

#### V. KESIMPULAN

Permasalahan *subset guessing problem* merupakan salah satu permasalahan yang dapat diselesaikan dengan algoritma *bruteforce*, iteratif, maupun *divide and conquer*. Walaupun *divide and conquer* memiliki struktur yang lebih kompleks dibanding iteratif biasa, permasalahan ini lebih efektif diselesaikan menggunakan algoritma iteratif biasa dibanding algoritma *divide and conquer*.

#### UCAPAN TERIMA KASIH

Puji dan syukur penulis panjatkan ke hadirat Allah SWT yang hanya karena berkat dan rahmat-Nya, penulis dapat menyelesaikan makalah ini dengan baik dan tepat waktu. Penulis juga mengucapkan terima kasih kepada Ibu Ulfa, selaku dosen pengajar mata kuliah Strategi Algoritma penulis. Ucapan terima kasih juga penulis berikan kepada orang tua, teman-teman, dan semua pihak yang mendukung penulis dalam merampungkan makalah ini.

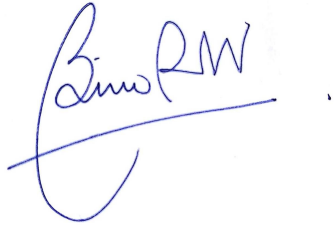
#### REFERENSI

- [1] Munir, Rinaldi. Diktat Kuliah IF2211 Strategi Algoritma. Program Studi Teknik Informatika ITB.
- [2] IEEE. IEEEExtreme Programming Competition 12.0.

## 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, 26 April 2019



Bimo AdityarahmanWiraputra, 13517004