Implementasi Algoritma Backtracking untuk Menyelesaikan Masalah Sum of Subset

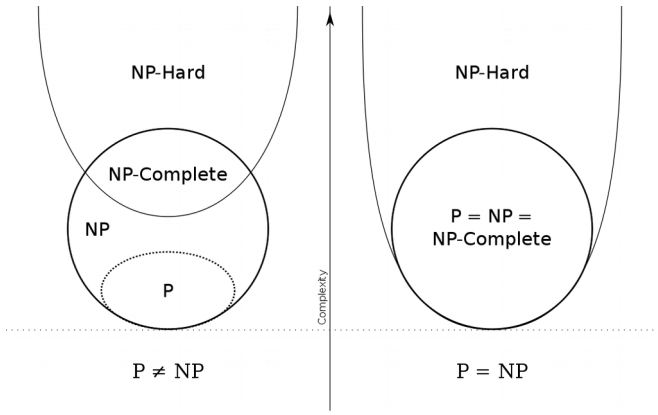
Arnold Pangihutan Sianturi
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517022@std.stei.itb.ac.id

***Abstrak*—Masalah sum of subset merupakan satu dari beberapa masalah yang tergolong mudah untuk mendeskripsikan NP-Complete (atau NPC), sebab sum of subset termasuk ke dalam kelas masalah NPC. Pada bidang ilmu komputer, sum of subset adalah salah satu masalah yang penting dalam teori kompleksitas dan kriptografi. Meskipun tergolong mudah dibanding masalah lain pada kelas masalah NPC, masalah ini tidak semudah kelihatannya, untuk diselesaikan. Dalam makalah ini akan dibahas penyelesaian masalah sum of subset dengan menggunakan algoritma runut-balik (*backtracking*).**

Keywords—sum of subset, backtracking

I. PENDAHULUAN

NP adalah adalah himpunan persoalan keputusan yang dapat diselesaikan oleh algoritma non-deterministik dalam waktu polinom.



Gambar 1. Diagram keterhubungan NP, NPC dan P (sumber: https://id.wikipedia.org/wiki/Masalah_P_versus_NP, diakses pada 25 April 2019)

Sebuah persoalan X dikatakan NPC jika:

1) X termasuk ke dalam kelas NP

2) Setiap persoalan di dalam NP dapat direduksi dalam waktu polinom menjadi X

Cara termudah untuk membuktikan sebuah persoalan X adalah NPC adalah menemukan sebuah metode sederhana (algoritma dalam waktu polinom) untuk mentransformasikan persoalan yang sudah dikenal NPC menjadi persoalan X tersebut. Dengan kata lain, untuk menunjukkan bahwa X adalah NPC, caranya adalah sebagai berikut:

1) Tunjukkan bahwa X adalah anggota NP

2) Pilih *instance*, Y, dari sembarang persoalan NPC.

3) Tunjukkan sebuah algoritma dalam waktu polinom untuk mentransformasikan (mereduksi) Y menjadi *instance* persoalan X

Di balik keunikan dari persoalan-persoalan ini, ternyata persoalan di dalam NPC dikatakan persoalan yang paling sukar karena persoalan ini dipecahkan dalam waktu/kompleksitas polinomial, dan jika ada persoalan NPC dipecahkan dalam waktu polinomial, maka semua persoalan di dalam NP dapat dipecahkan dalam waktu polinomial.

Di dalam NPC, salah satu persoalan yang termasuknya adalah *subset sum problem*. Masalah utama dari *sum of subset* adalah jika terdapat n bilangan real dan ingin dihitung semua kombinasi yang mungkin dari himpunan bilangan tersebut. Kombinasi yang diinginkan yaitu kombinasi yang jumlah seluruh elemennya sama dengan M (tertentu). Selain memberikan hasil yang benar, efisiensi dari waktu eksekusi ataupun penggunaan memori dari algoritma adalah hal yang penting bagi sebuah algoritma. Setiap masalah yang biasa dihadapi di dunia informatika dapat diselesaikan dengan beberapa algoritma yang sudah ada (tidak hanya satu algoritma). Namun, tidak semua algoritma bisa menyelesaikan suatu masalah dengan efektif dan efisien (mangkus). Tingkat kemangkusan suatu algoritma dalam memecahkan masalah ditentukan dengan menghitung operasi khas pada algoritma tersebut lalu menyatakannya dengan notasi Big-O.

Algoritma dengan waktu polinomial memimiliki fungsi Big-O = $O(n^m)$. Karena seperti itu, maka *Sum of subset* merupakan masalah yang tergolong yang sukar untuk dpipecahkan dengan waktu polinomial.

II. DASAR TEORI

A. Algoritma *Bactracking*

Algoritma *backtracking* merupakan algoritma yang berbasis pada Depth-First-Search (DFS) untuk mencari solusi persoalan secara lebih mangkus. Secara sistematis, *backtracking* mencari solusi persoalan di antara semua kemungkinan solusi yang ada. Dengan metode *backtracking*,

tidak perlu memeriksa semua kemungkinan yang ada. Hanya pencarian yang mengarah ke solusi saja yang selalu dipertimbangkan. Keuntungannya, waktu pencarian akan lebih sedikit. *Backtracking* alami dinyatakan dalam algoritma rekursif. Kadang-kadang disebutkan pula bahwa *backtracking* merupakan bentuk tipikal dari algoritma rekursif.

Istilah *backtracking* sendiri pertama kali diperkenalkan oleh D.H. Lehmer pada tahun 1950. Selanjutnya, R.J. Walker, Golomb, dan Baumert menyajikan uraian umum tentang *backtracking* dan penerapannya pada berbagai persoalan. Saat ini algoritma backtracking banyak diterapkan untuk program *games* (seperti permainan tic-tac-toe, menemukan jalan keluar dalam sebuah labirin, catur, dan lain-lain) dan masalah-masalah pada bidang kecerdasan buatan (*artificial intelligence*).

1. Properti Umum Metode *Backtracking*

Untuk menerapkan metode backtracking, properti berikut didefinisikan:

a. Solusi persoalan.

Solusi dinyatakan sebagai vektor dengan n-tuple: X = (x1, x2, …, xn), xi himpunan berhingga Si. Mungkin saja S1 = S2 = … = Sn. Contoh: Si = {0, 1}, xi = 0 atau 1.

b. Fungsi pembangkit nilai xk

Dinyatakan sebagai: T(k).

T(k) membangkitkan nilai untuk xk, yang merupakan komponen vektor solusi.

c. Fungsi pembatas

Dinyatakan sebagai: B(x1, x2, …, xk).

Fungsi pembatas menentukan apakah (x1, x2, …, xk) mengarah ke solusi. Jika ya, maka pembangkitan nilai untuk xk+1 dilanjutkan, tetapi jika tidak, maka (x1, x2, …, xk) dibuang dan tidak dipertimbangkan lagi dalam pencarian solusi.

Fungsi pembatas tidak selalu dinyatakan sebagai fungsi matematis, ia dapat dinyatakan sebagai predikat yang bernilai true atau false, atau dalam bentuk lain yang ekivalen.

2. Pengorganisasian Solusi

Semua kemungkinan solusi dari persoalan disebut ruang solusi (*solution space*). Secara formal dapat dinyatakan, bahwa jika $x_i \in S_i$, maka

$S_1 \times S_2 \times \ldots \times S_n$

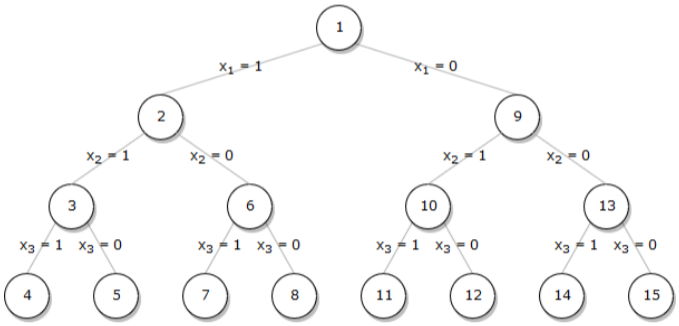
disebut ruang solusi. Jumlah anggota di dalam ruang solusi adalah $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_n|$.

Sebagai gambaran, tinjau persoalan Knapsack 0/1 untuk n = 3. Solusi persoalan dinyatakan sebagai vektor (x_1, x_2, x_3) dengan $x_i \in \{0, 1\}$. Ruang solusinya adalah

{0, 1} x {0,1} x {0, 1} = {(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)}.

Dengan kata lain, pada persoalan Knapsack 0/1 dengan n = 3 terdapat $2^n = 2^3 = 8$ kemungkinan solusi, yaitu (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), dan (1, 1, 1).

Algoritma *backtracking* memperbaiki pencarian solusi secara *exhaustive search* dengan mencari solusi secara sistematis. Untuk memfasilitasi pencarian ini, maka ruang solusi diorganisasikan ke dalam struktur pohon. Tiap simpul pohon menyatakan status (*state*) persoalan, sedangkan sisi (cabang) dilabeli dengan nilai-nilai x_i. Lintasan dari akar ke daun menyatakan solusi yang mungkin. Seluruh lintasan dari akar ke daun membentuk ruang solusi. Pengorganisasian pohon ruang solusi diacu sebagai pohon ruang status. Tinjau kembali persoalan Knapsack 0/1 untuk n = 3. Ruang solusinya diorganisasikan sebagai pohon ruang status pada Gambar 2. Lintasan dari 1 sampai 4 misalnya, menyatakan solusi X = (1, 1, 1). Perhatikanlah bahwa simpul-simpul diberi urutan nomor sesuai dengan prinsip pencarian DFS. Metode backtracking menerapkan DFS dalam pencarian solusi.



Gambar 2. Ruang solusi untuk persoalan Knapsack 0/1 dengan n = 3 (sumber: http://supeeerblog.blogspot.com/2013/01/teknik-backtracking-dan-sumofsub.html, diakses pada 25 April 2019)

3. Prinsip Pencarian Solusi dengan Metode *Backtracking*

Di sini kita hanya akan meninjau pencarian solusi pada pohon ruang status yang dibangun secara dinamis. Langkah-langkah pencarian solusi adalah sebagai berikut:

a. Solusi dicari dengan membentuk lintasan dari akar ke daun. Aturan pembentukan yang dipakai adalah mengikuti metode pencarian mendalam (DFS). Simpul-simpul yang sudah dilahirkan dinamakan simpul hidup (*live node*). Simpul hidup yang sedang diperluas dinamakan simpul-E (*Expand-node*). Simpul dinomori dari atas ke bawah sesuai dengan urutan kelahirannya (berdasarkan prinsip DFS).

b. Tiap kali simpul-E diperluas, lintasan yang dibangun olehnya bertambah panjang. Jika lintasan yang sedang dibentuk tidak mengarah ke solusi, maka simpul-E tersebut "dibunuh" sehingga menjadi simpul mati (dead node). Fungsi yang digunakan untuk membunuh simpul-E adalah dengan menerapkan fungsi pembatas (bounding function). Simpul yang sudah mati tidak akan pernah diperluas lagi.

c. Jika pembentukan lintasan berakhir dengan simpul mati, maka proses pencarian diteruskan dengan membangkitkan simpul anak yang lainnya. Bila tidak ada simpul anak yang dapat dibangkitkan, maka pencarian solusi dilanjutkan dengan melakukan backtrack ke simpul hidup terdekat (simpul orangtua). Selanjutnya simpul ini menjadi simpul-E

yang baru. Lintasan baru dibangun kembali sampai lintasan tersebut membentuk solusi.

d. Pencarian dihentikan bila kita telah menemukan solusi atau tidak ada lagi simpul hidup untuk backtracking.

Tinjau persoalan Knapsack 0/1 dengan instansiasi: n = 3 (w_1, w_2, w_3) = (35, 32, 25) (p_1, p_2, p_3) = (40, 25, 50) M = 30 Solusi dinyatakan sebagai X = (x1, x2, x3), xi {0, 1}. Fungsi pembatas yang digunakan adalah kendala (constraint):

$$\sum_{i=1}^{k} w_i x_i \leq M$$

Jadi, jika vektor solusi (x_1, x_2, …, x_k) tidak memenuhi ketidaksamaan tersebut, maka lintasan di dalam pohon pencarian yang sisi-sisinya dilabeli dengan (x_1, x_2, …, x_k) tidak mengarah ke solusi sehingga lintasan tersebut tidak diperpanjang lagi. Sebaliknya, jika memenuhi, maka lintasan tersebut dapat diteruskan untuk dikembangkan. Mulailah dari status awal (simpul akar) yaitu belum ada objek yang dimasukkan ke dalam knapsack. Beri nomor simpul akar ini dengan angka satu. Simpul 1 sekarang menjadi simpul hidup sekaligus simpul-E. Simpul 1 diperluas menjadi simpul 2 dan meng-*assign* sisi (1, 2) dengan $x_1 = 1$.

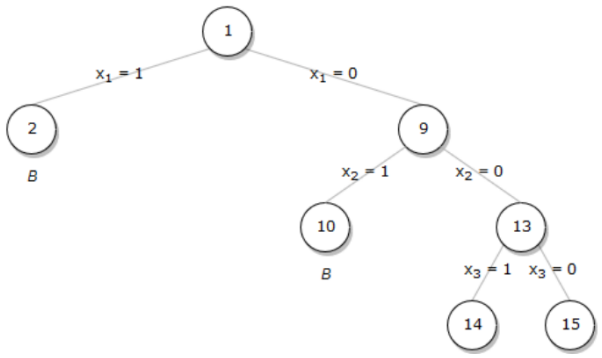
Sampai saat ini solusi adalah X = (1, x_2, x_3). Solusi ini tidak layak karena berat objek 1 lebih besar daripada kapasitas *knapsack*. Oleh karena itu, simpul 2 dimatikan (pada Gambar 2 simpul 2 ditandai dengan B, yang artinya simpul itu mati karena tidak memenuhi kendala. Dengan demikian, anak-anak simpul 2 tidak pernah diperluas lagi). Lakukan *backtrack* ke simpul 1. Simpul 1 diperluas menjadi simpul 9 dan meng-assign sisi (1,9) dengan $x_1 = 0$.

Sampai saat ini solusi adalah X = (0, x_2, x_3). Solusi ini layak sehingga simpul 9 dapat diperluas. Sekarang simpul 9 menjadi simpul-E. Simpul 9 diperluas menjadi simpul 10 dan meng-assign sisi (9, 10) dengan $x_2 = 1$.

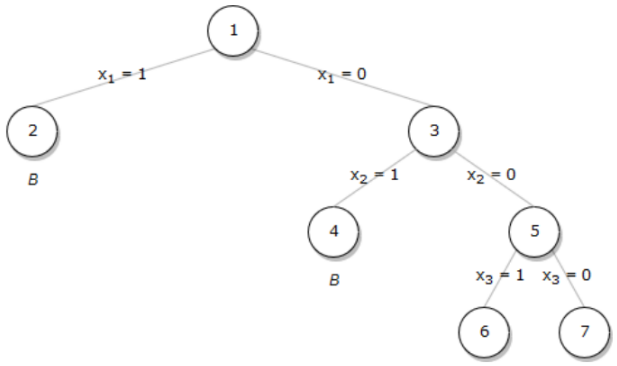
Sampai saat ini solusi adalah X = (0, 1, x_3). Solusi ini tidak layak karena berat objek 2 lebih besar daripada kapasitas *knapsack*. Oleh karena itu, simpul 10 dimatikan. Lakukan *backtrack* ke simpul 9. Simpul 9 diperluas menjadi simpul 13 dan meng-*assign* sisi (9, 13) dengan $x_2 = 0$.

Sampai saat ini solusi adalah X = (0, 0, x_3). Solusi ini layak sehingga simpul 13 dapat diperluas menjadi simpul 14 dan meng-assign sisi (13, 14) dengan $x_3 = 1$; sampai saat ini solusi adalah X = (0, 0, 1). Solusi ini layak, dan arena simpul 14 adalah simpul daun berarti kita sudah mencapai simpul solusi dengan X = (0, 0, 1) dan F = 50. Ini adalah solusi terbaik sampai sejauh ini. Karena simpul 13 masih hidup, maka simpul 13 masih dapat diperluas menjadi simpul 15 dan meng-*assign* sisi (13, 15) dengan $x_3 = 0$.

Sampai saat ini solusi adalah X = (0, 0, 0). Solusi ini layak, dan karena simpul 15 adalah simpul daun berarti kita sudah mencapai simpul solusi dengan X = (0, 0, 0) dan F = 0. Namun, karena total keuntungan solusi yang kedua lebih kecil dari total keuntungan solusi pertama, maka solusi optimumnya adalah X = (0, 0, 1) dan F = 50.



Gambar 3. Pohon dinamis yang dibentuk selama pencarian untuk persoalan Knapsack 0/1 dengan n = 3, w = (35, 32, 25), p = (40, 25, 50) (sumber: http://supeeerblog.blogspot.com/2013/01/teknik-backtracking-dan-sumofsub.html, pada 25 April 2019)



Gambar 4. Penomoran ulang simpul-simpul sesuai urutan pembangkitannya
(sumber:http://supeeerblog.blogspot.com/2013/01/teknik-backtracking-dan-sumofsub.html, pada 25 April 2019)

4. Skema Umum Algoritma *Backtracking*

Di bawah ini disajikan skema umum algoritma backtracking dalam dua versi, versi rekursif dan versi iteratif. Skema dalam versi rekursif lebih tepat karena algoritma backtracking lebih alami dinyatakan dalam bentuk rekursif. Algoritma di bawah ini akan menghasilkan semua solusi.

```
procedure RunutBalikR(input k: integer)
{Mencari semua solusi persoalan dengan metode backtracking; skema rekursif
Masukan: k, yaitu indeks komponen vektor solusi, x[k]
Keluaran: solusi x = (x[1], x[2], …, x[n]) }

Algoritma:
    for tiap x[k] yang belum dicoba sedemikian sehingga
    (x[k] ← T(k)) and B(x[1], x[2], …, x[k]) = true) do
            if (x[1], x[2], …, x[k]) adalah lintasan dari akar
            ke daun then
                    CetakSolusi(x)
            endif
            RunutBalikR(k+1)
            {tentukan nilai untuk x[k+1]}
    endfor
```

Pemanggilan prosedur pertama kali: RunutbalikR(1).

Prosedur CetakSolusi adalah sebagai berikut:

```
procedure RunutBalikR(input k: integer)
{Mencari semua solusi persoalan dengan metode backtracking; skema rekursif
Masukan: k, yaitu indeks komponen vektor solusi, x[k]
Keluaran: solusi x = (x[1], x[2], …, x[n]) }

Algoritma:
    for tiap x[k] yang belum dicoba sedemikian sehingga
    (x[k] ← T(k)) and B(x[1], x[2], …, x[k]) = true) do
        if (x[1], x[2], …, x[k]) adalah lintasan dari akar
        ke daun then
            CetakSolusi(x)
        endif
        RunutBalikR(k+1)
        {tentukan nilai untuk x[k+1]}
    endfor
```

Setiap simpul dalam pohon ruang status berasosiasi dengan sebuah pemanggilan rekursi. Jika jumlah simpul dalam pohon ruang status adalah 2n atau n!, maka untuk kasus terburuk, algoritma backtracking membutuhkan waktu dalam $O(p(n)2^n)$ atau O(q(n)n!), dengan p(n) dan q(n) adalah polinom derajat n yang menyatakan waktu komputasi setiap simpul.

B. Permasalahan *Sum of Subset*

Dalam dunia ilmu komputer, *sum of subset* merupakan masalah yang penting dalam teori kompleksitas dan kriptografi. Masalah *sum of subset* adalah sebagai berikut.

> *Diberikan himpunan bilangan bulat tidak kosong dan sebuah angka yang menyatakan jumlah suatu bilangan, misalnya m. Carilah sub-himpunan yang jumlahnya = m.*

Misalnya terdapat himpunan bilangan A = {-4, -1, 1, 2, 3, 8, 9} dan m = 0. Maka salah satu solusi dari masalah sum of subset tersebut adalah {-4, -1, 2, 3}.

Sum of subset merupakan masalah *NP-Complete* (*Non-Deterministic Polynomial Complete* atau NPC). Hal ini dikarenakan masalah *sum of subset* termasuk ke dalam kelompok kelas NP dan setiap persoalan di dalam kelas NP dapat direduksi menjadi masalah *sum of subset* dalam waktu polinom. Oleh karena itu, masalah sum of subset dapat dikatakan salah satu persoalan yang sukar karena belum ada persoalan NPC yang dapat dipecahlan dalam waktu polinomial. Jika ada persoalan NPC yang dapat dipecahkan dalam waktu polinomial, maka semua persoalan di dalam NP dapat dipecahkan dalam waktu polinomial. Masalah *sum of subset* dapat pula dikatakan sebagai masalah khusus dari persoalan knapsack. Hal ini berarti masalah *sum of subset* sebenarnya memiliki penyelesaian yang mirip dengan persoalan *knapsack*. Perbedaannya adalah pada persoalan *knapsack*, kita perlu mengetahui barang apa saja yang perlu diikutsertakan sambil memperhitungkan keuntungan yang diperoleh jika barang tersebut dimasukkan ke dalam *knapsack* agar memperoleh keuntungan maksimum dan tidak melebihi kapasitas *knapsack*, sedangkan pada masalah *sum of subset* kita hanya perlu mengetahui bilangan apa saja yang diikutsertakan ke dalam solusi agar bilangan- bilangan tersebut memiliki jumlah yang telah ditentukan sebelumnya. Kompleksitas dari masalah *sum of subset* bergantung kepada jumlah bilangan di dalam himpunan dan presisi dari masalah tersebut. Jika jumlah bilangan di dalam himpunan cukup kecil, maka penyelesaian masalah *sub of subset* dengan algoritma *exhaustive search* sudah tergolong mangkus. Algoritma *exhaustive search* sendiri bertujuan mencari seluruh kemungkinan penjumlahan bilangan yang mungkin dari himpunan bilangan yang diberikan lalu memeriksa setiap kemungkinan penjumlahan tersebut, apakah hasil penjumlahannya sesuai dengan jumlah bilangan yang diinginkan. Jika hasil penjumlahannya sesuai, maka penjumlahan bilangan tersebut termasuk dalam solusi, dan sebaliknya. Namun, untuk jumlah bilangan di dalam himpunan tersebut cukup besar, algoritma *exhaustive search* menjadi sangat tidak efisien karena kompleksitas waktu algoritma *exhaustive search* dalam menyelesaikan masalah *sum of subset* adalah eksponensial. Oleh karena itu, para ahli atau programmer berusaha untuk menemukan algoritma yang lebih mangkus daripada *exhaustive search*. Beberapa algoritma yang dapat digunakan untuk menyelesaikan masalah *sum of subset* adalah algoritma *backtrack*, *greedy*, dan *dynamic programming*.

III. Implementasi Algoritma *Backtracking* Terhadap Permasalahan *Sum of Subset*

Masalah utama dari Sum of Subsets adalah jika terdapat n bilangan real dan ingin dihitung semua kombinasi yang mungkin dari himpunan bilangan tersebut. Kombinasi yang diinginkan yaitu kombinasi yang jumlah seluruh elemennya sama dengan M (tertentu).

Bentuk penyajian pohon dari persoalan *sum of subset*(yaitu DFS dan BFS), merupakan tahapan pertama dalam proses mendapatkan solusi sesungguhnya (solusi optimal). Untuk mendapatkan solusi yang optimal dari ruang penyelesaian digunakan suatu algoritma lain. Algoritma tersebut menggunakan teknik backtracking, yang selanjutnya disebut dengan algoritma subestSum.

```
procedure subsetSum(set, subset, n, subSize, total, node, sum)
{set : himpunan bilangan; subset : himpunan jawaban; n : ukuran set;
subSize : ukuran subset; total : jumlah bilangan pada subset;
node : banyak elemen pada subset; sum : jumlah yang ingin dicari}

Begin
    if total = sum then
      output(subset)
      //lanjut untuk mencari subset berikutnya
      subsetSum(set, subset, subSize-1, total - set[node], node+1,
sum)
      //return
    else
      for semua element i pada set do
        subset[subSize] ← set[i]
        subsetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)
      endfor
End
```

Implementasi algoritma ini akan direpresenasikan langsung dengan sebuah contoh sederhana.

> Diberikan sebuah himpunan (*set*) : {10, 7, 5, 18, 12, 20, 15}. Tentukan himpunan bagian (jawaban) yang memiliki total elemen sejumlah 35.

Langkah-langkah yang dilakukan adalah sebagai berikut.

1. Dari set yang diketehaui, ciptakan sebuah subset yang akan menampung jawaban. Sediakan ukuran subset sebesar ukuran set. Panggil prosedur rekursif dengan paramater subSize, total dan node masing-maing diisi dengan besar 0.

2. Masuk ke prosedur rekursif. Pertama, cek apakah total = 35. Karena nilai total di-set sebesar 0, berarti nilai total != 35. Maka, masuk ke kondisi else.

3. Di dalam kondisi else, ada *looping* yang akan menge-set setiap elemen subSet berbeda-beda namun tetap pada indeks 0. Sebelum nilai i pada loop tersebut diinkremen, prosedur subsetSum dipanggil lagi.

4. Setelah prosedur seubsetSum dipanggil lagi, masukan parameter adalah sama seperti langkah 1, dengan perbedaan subSize ditambah 1, total ditambah 'nilai subSet[i]', dan node diisi dengan i+1. Dalam prosedur rekursif pertama ini, setiap elemen subSet kembali diisi oleh angka yang berbeda namun pada indeks 1 (mirip langkah 3, namun indeknya berbeda, dan pengisian dilakukan mulai dari indeks 2 si 'set'). Dan sebelum inkremensi, prosedur subsetSum dipanggil lagi.

5. Pada setiap rekursif, akan dicek apakah ketemu total nilai subset yang memenuhi (yaitu 35). Apabila ketemu, elemen-elemen subset tersebut akan dicetak ke layar. Kemudian, prosedur subsetSum dipanggil lagi dengan masukan parameter subSize dikurangi 1, total dikurangi dengan elemen terkahir dari subset, dan node dikurangi 1. Rekursif berhenti hingga *looping* selesai.

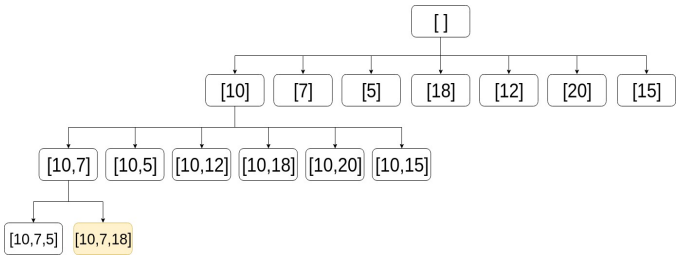
Berikut adalah kode program dalam bahasa C++.

```
#include <iostream>
using namespace std;

void printSubset(int subSet[], int size) {
  for(int i = 0; i < size; i++) {
    cout << subSet[i] << "  ";
  }
  cout << endl;
}

void subsetSum(int set[], int subSet[], int n, int subSize, int total,
                        int nodeCount ,int sum) {
  if(total == sum) {
    printSubset(subSet, subSize);     //print isi subset ke layar
    subsetSum(set,subSet,n,subSize-1,total-set[nodeCount],
                        nodeCount+1,sum);     //panggil lagi untuk
                                              mengecek subset
berikutnya
    //return
  }else {
    for( int i = nodeCount; i < n; i++ ) {     //isi subset secara melebar
      subSet[subSize] = set[i];
      subsetSum(set,subSet,n,subSize+1,total+set[i],i+1,sum);
                        //isi subset secara mendalam
    }
  }
}
```

```
void findSubset(int set[], int size, int sum) {
  int *subSet = new int[size];     //menciptakan array subSet sebesar
size
  subsetSum(set, subSet, size, 0, 0, 0, sum);
  delete[] subSet;
}

int main() {              //main
  int weights[] = {10, 7, 5, 18, 12, 20, 15};
  int size = 7;
  findSubset(weights, size, 35);
}
```

```
morty@arnoldpstr:~/makalah$ g++ subsetSum.cpp
morty@arnoldpstr:~/makalah$ ./a.out
10  7  18
10  5  20
5  18  12
morty@arnoldpstr:~/makalah$
```

Gambar 5. Keluaran program dari contoh (diakses di sumber pribadi, 26 April 2019)



Gambar 6. Pohon pengisian array subset. Seharusnya, pohon terisi secara melebar (BFS). Namun, hanya ditunjukkan hingga jawaban pertama ditemukan. (dibuat di draw.io, pada 26 April 2019)

IV. KESIMPULAN

Permasalahan *sum of subset* dapat diselesaikan dengan berbagai macam algoritma, salah satunya adalah dengan menggunakan algoritma *backtracking* (runut-balik). Algoritma ini memiliki kompleksitas waktu yang lebih baik dalam menyelesaikan masalah *sum of subset* dibandingkan dengan algoritma lainnya. Walaupun begitu, sampai saat ini belum terdapat algoritma yang cukup mangkus untuk menyelesaikan masalah *sum of subset*. Hal ini dikarenakan sampai saat ini belum ada yang berhasil menemukan cara untuk menyelesaikan masalah NP-Complete secara mangkus. Jika ditemukan suatu algoritma untuk menyelesaikan masalah *NP-Complete* dalam waktu yang wajar (polinomial), maka masalah *sum of subset* juga dapat diselesaikan dengan dalam waktu polinomial karena masalah sum of subset merupakan masalah NPC.

UCAPAN TERIMA KASIH

Pertama, penulis ingin mengucapkan puji dan syukur kepada Tuhan Yang Maha Esa karena dengan berkat-Nya penulis dapat menyelesaikan makalah dengan judul "Implementasi Algoritma Backtracking untuk Menyelesaikan Masalah Sum of Subset" dengan baik. Penulis juga berterima kasih kepada para dosen pengajar mata kuliah IF2211 Strategi Algoritma, Dr. Masayu Leylia Khodra, Dr. Nur Ulfa Maulidevi, S.T, M.Sc., dan Dr. Ir. Rinaldi Munir, M.T., atas bimbingannya selama ini dalam mengajar dan memberikan ilmu sehingga penulis mampu untuk membuat makalah ini, terlebih kepada pengimplementasian algoritma terhadap ilmu lain. Penulis juga berterima kasih kepada teman dan keluarga yang telah memberikan semangat dan dorongan kepada penulis.

REFERENSI

[1] Munir, Rinaldi. 2005. "Diktat Kuliah IF 2211 Strategi Algoritmik". Bandung:Program Studi Teknik Informatika STEI ITB.

[2] https://www.geeksforgeeks.org/subset-sum-backtracking-4/
(diakses pada 25 April 2019)

[3] http://journal.uad.ac.id/index.php/JIFO/article/view/5277/2910
(diakses pada 26 April 2019)

[4] http://supeeerblog.blogspot.com/2013/01/teknik-backtracking-dan-sumofsub.html
(diakses pada 25 April 2019)

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

Arnold Pangihutan Sianturi
13517022