Penggunaan Berbagai Algoritma dalam penyelesaian *Coins in a Row Puzzle*

Widya Anugrah Putra 13519105
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail : 13519105@std.stei.itb.ac.id

***Abstract*— *Coins in a Row* adalah sebuah permainan yang dimainkan oleh dua pemain, misalnya A dan B, yang secara bergantian mengambil koin emas yang berjajar. Peraturannya adalah pemain secara bergantian mengambil koin emas hanya dari salah satu ujung jajaran koin-koinnya. Dengan menggunakan *Dynamic Programming*, kita bisa melihat berapa banyak koin yang bisa didapatkan oleh pemain pertama yang menginginkan kemenangan dengan bermain secara optimal, tetapi pemain lawan juga bermain secara optimal. Akan ditunjukkan juga mengapa algoritma *Greedy* tidak selalu memberikan hasil yang optimal dan mengapa algoritma *Naive Recursive* memiliki kompleksitas yang tidak lebih baik daripada *Dynamic Programming*.**

Keywords—Pots of Gold, Coins in a Row, Combinatorial Game Theory, Dynamic Programming, Greedy

I. Pendahuluan

Dalam buku *Mathematical Puzzles* [4], buatan Peter Winkler, puzzle pertama yang disajikan adalah puzzle *Coins in a Row*. Puzzle ini punya nama lain *Pots of Gold* dan sering sekali disajikan sebagai ujian wawancara bagi pemrogram dalam beberapa perusahaan.

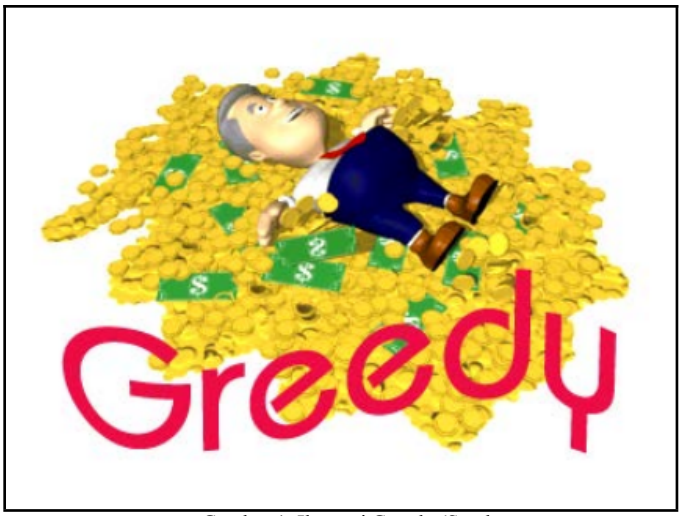
Coins in a Row adalah sebuah game yang dimainkan oleh dua orang yang saling bersaing untuk mengumpulkan koin-koin emas yang berjejeran secara bergantian. Pengambilan emas hanya bisa di ujung awal atau ujung akhir jajaran koinnya. Puzzlenya adalah bagaimana strategi pemain pertama mengambil keuntungan maksimal dari jajaran koin tersebut.

Puzzle ini dapat diselesaikan dengan berbagai metode, contohnya *Naive Recursive*, *Dynamic Programming*, dan algoritma buatan Tomasz Idziaszek [3] yang mampu menyelesaikan puzzle ini dalam kompleksitas O(N) saja. Penulis akan membahas bagaimana algoritma *Greedy* gagal mendapatkan solusi optimal, lalu algoritma *Naive Recursive* yang mampu mendapatkan solusi optimal serta algoritma *Dynamic Programming* yang memperbaiki kompleksitas algoritma *Naive Recursive*. Penulis juga akan me*review* algoritma yang ditemukan Tomasz dan memberikan implementasinya.

II. Landasan Teori

A. *Greedy*

Algoritma Greedy merupakan metode yang paling populer dan paling sederhana untuk memecahkan persoalan optimasi. Persoalan optimasi adalah persoalan mencari solusi optimal. Prinsip algoritma ini adalah *"take what you can get now!"*. Algoritma ini membentuk solusi langkah per langkah, lalu pada setiap langkah terdapat banyak pilihan yang perlu dievaluasi. Karena prinsip algoritmanya adalah *greedy/rakus*, maka diambil pilihan terbaik dari setiap langkah. Kelemahannya adalah algoritma ini tidak bisa mundur kembali ke langkah sebelumnya. Jadi pada tiap langkahnya, algoritma ini memilih *optimum local* dengan harapan bahwa langkah sisanya mengarah ke solusi *optimum global*. Solusi yang dihasilkan algoritma ini terkadang tidak optimal, sehingga hanya cocok untuk beberapa permasalahan saja.



Gambar 1: Ilustrasi Greedy (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf)

Elemen-elemen algoritma *greedy* adalah

a. Himpunan kandidat, C: berisi kandidat yang akan dipilih pada setiap langkah.
b. Himpunan solusi, S: berisi kandidat yang sudah dipilih.

c. Fungsi solusi: menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi.
d. Fungsi seleksi: memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.
e. Fungsi kelayakan: memeriksa apakah kandidat yang dipilih dapat dimasukkan dalam himpunan solusi (layak atau tidak)
f. Fungsi obyektif: memaksimumkan atau meminimumkan.

Contoh-contoh persoalan yang diselesaikan dengan algoritma *greedy*:

a. Minimum spanning tree
b. Shortest path
c. Huffman code
d. Knapsack problem, dll

Contoh permasalahan sederhana yang mampu diselesaikan dengan algoritma Greedy adalah permasalahan pembelian saham jika pada hari ke-i hanya bisa membeli saham sebanyak i saja. Contoh kasusnya:

Misal, harga saham per lembar untuk 3 hari berturut-turut adalah [10, 7, 19] dan seseorang hanya diberikan uang 45 satuan. Berapa lembar saham maksimal yang dapat dibeli?

Mari definisikan elemen-elemennya:

a. Himpunan kandidat, C: himpunan harga-harga saham per lembar tiap hari
b. Himpunan solusi, S: himpunan multiset yang berisi frekuensi pembelian lembar saham pada hari tertentu
c. Fungsi solusi, jika himpunan solusi ditambah dengan harga saham termurah yang masih bisa dibeli sudah melebihi uang yang tersedia.
d. Fungsi seleksi, sorting harga saham berdasarkan harganya, jika ada harga yang sama, sorting terhadap hari terbesar. Ambil elemen terdepan jika batasan banyak saham belum dilewati.
e. Fungsi kelayakan, harga saham ditambah dengan himpunan solusi sekarang belum melebihi uang yang diberikan di awal.
f. Fungsi obyektif-nya adalah memaksimalkan lembar saham yang dapat dibeli.

Dari elemen-elemen algoritma greedy di atas, kita dapat menyelesaikan contoh persoalan pada kasus di atas.

Jika diurutkan berdasarkan harga saham, maka urutan pertamanya adalah harga saham pada hari kedua, yaitu 7 satuan. Lalu harga saham pada hari pertama yaitu 10 satuan, dan terakhir harga saham pada hari ketiga yaitu 19 satuan.

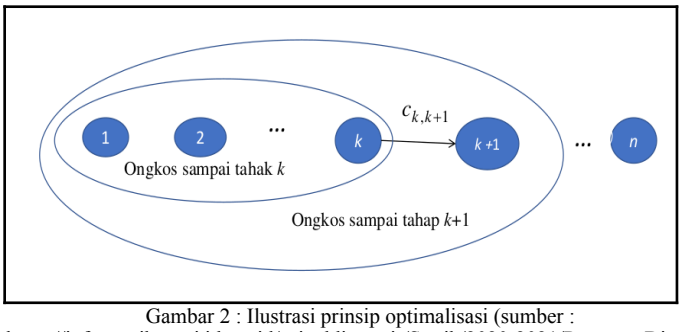
Secara greedy kita bisa mengambil 2 lembar saham sekaligus pada hari kedua (karena 0 + 2 * 7 masih belum melebihi 45). Sehingga sekarang biaya membeli sahamnya adalah 14. Lalu kita bisa mengambil 1 lembar saham pada hari pertama (karena 14 + 1 * 10 masih belum melebihi 45). Sehingga sekarang biaya pembelian sahamnya adalah 24. Terakhir, kita coba untuk membeli selembar saham pada hari ketiga, karena 24 + 19 masih di bawah 45, kita ambil satu lembar dulu. Total biaya sekarang adalah 43. Lalu kita coba beli selembar lagi, ternyata tidak bisa karena 43 + 19 lebih dari 45, sehingga solusinya berhasil ditemukan. Total lembar saham yang telah dibeli adalah 2 + 1 + 1 = 4 lembar saham.

B. *Dynamic Programming*

Dynamic Programming atau Program Dinamis adalah metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan tahapan (stage) sedemikian sehingga solusi persoalan dapat dipandang sebagai serangkaian keputusan yang saling mempengaruhi. Istilah *Dynamic Programming* pertama kali diperkenalkan pada tahun 1950-an oleh profesor di Princeton University yang bernama Richard Bellman. Penerapan *Dynamic Programming* banyak digunakan pada proses optimalisasi suatu permasalahan.

Perbedaan *Dynamic Programming* dengan algoritma *Greedy* adalah *Dynamic Programming* memilih rangkaian keputusan mana yang baik dalam langkah ke-k, yang mempertimbangkan rangkaian pilihan langkah sebelumnya (langkah $k-1$) dan hubungan antar langkahnya, sedangkan algoritma *Greedy* tidak mempedulikan pengambilan keputusan pada langkah sebelumnya, dan selalu mengambil pilihan yang terbaik pada langkah sekarang. Sehingga pada algoritma *Greedy* hanya menghasilkan 1 rangkaian keputusan, sedangkan pada algoritma *Dynamic Programming* rangkaian keputusan yang dipertimbangkan lebih dari 1.

Pada program dinamis, rangkaian keputusan yang optimal dibuat dengan menggunakan Prinsip Optimalitas. Prinsip Optimalitas : *jika solusi total optimal, maka bagian solusi sampai tahap ke-k juga optimal*. Prinsip optimalitas berarti bahwa jika kita bekerja dari tahap k ke tahap k+1, kita dapat menggunakan hasil optimal dari tahap k tanpa harus kembali ke tahap awal. Ongkos pada tahap k + 1 = (ongkos yang dihasilkan pada tahap k) + (ongkos dari tahap k ke tahap k+1, atau $C_{k,k+1}$) .



Gambar 2 : Ilustrasi prinsip optimalisasi (sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf)

Beberapa ciri atau karakteristik dari persoalan program dinamis adalah

1. Persoalan dapat dibagi menjadi beberapa tahap (stage), yang pada setiap tahap hanya diambil satu keputusan.
2. Masing-masing tahap terdiri dari sejumlah status (state) yang berhubungan dengan tahap tersebut. Secara umum, status merupakan bermacam kemungkinan masukan yang ada pada tahap tersebut.

3. Hasil dari keputusan yang diambil pada setiap tahap ditransformasikan dari status yang bersangkutan ke status berikutnya pada tahap berikutnya.
4. Cost pada suatu tahap meningkat secara teratur (steadily) dengan bertambahnya jumlah tahapan.
5. Ongkos pada suatu tahap bergantung pada ongkos tahap-tahap yang sudah berjalan dan ongkos pada tahap tersebut.
6. Adanya hubungan rekursif yang mengidentifikasikan keputusan terbaik untuk setiap status pada tahap K memberikan keputusan terbaik untuk setiap status pada tahap K + 1.
7. Prinsip optimalitas berlaku pada persoalan tersebut.

Ada dua pendekatan *Dynamic Programming*

1. Pendekatan maju (*forward* atau *bottom-up*)

   Perhitungan dilakukan dari tahap 1, 2, …, n - 1, n

2. Pendekatan mundur (*backward* atau *top-down*)

   Perhitungan dilakukan dari tahap n, n-1, …, 2, 1

Langkah-langkah pengembangan algoritma *Dynamic Programming*

1. Karakteristik-kan struktur solusi optimal.

   Penentuan tahap, variabel, keputusan, status, dsb.

2. Definisikan secara rekursif nilai solusi optimal.

   Penentuan hubungan nilai optimal suatu tahap dengan tahap sebelumnya. Penulis menganggap langkah ini sebagai langkah yang paling sulit dalam mengidentifikasi penggunaan DP.

3. Hitung nilai solusi optimal secara maju atau mundur.

   Kebanyakan menggunakan tabel

4. Rekonstruksi solusi optimal (opsional)

   Rekonstruksi solusi secara mundur.

Salah satu permasalahan yang dapat diselesaikan dengan *Dynamic Programming* adalah barisan ke N fibonacci.

1. Karakteristik-kan struktur solusi optimal

   Tahap ke-k adalah tahap mencari bilangan ke k dari baris fibonacci.

2. Definisikan secara rekursif

   $$f_n = f_{n-1} + f_{n-2}, f_0 = 0 \text{ dan } f_1 = 1$$

3. Hitung nilai solusi optimal secara maju atau mundur

   Akan dihitung menggunakan pendekatan maju, tetapi untuk menghemat space, kita bisa mengganti tabel dengan beberapa variabel yang secara bergantian berisi bilangan fibonacci ke n-1 dan bilangan fibonacci ke n-2.

```
const Fibo = (n) => {
  if (n === 0) return 0;
  if (n === 1) return 1;
  let a = 0;
  let b = 1;
  for (let i = 2; i <= n; i++){
    b = a + b;
    a = b - a;
  }
  return b;
}
```

Gambar 3 : solusi bilangan fibonacci dengan pendekatan maju *dynamic programming* (sumber : koleksi pribadi)

C. *Coins in a Row*

Coin in a Row atau juga dikenal dengan *Pots of Gold* adalah sebuah puzzle pertama yang ditulis dalam buku *Mathematical Puzzles* [4] (Winkler, Peter. 2004). Puzzle ini berbunyi:

"*On a table is a row of fifty coins, of various denominations. Alice picks a coin from one of the ends and puts it in her pocket, then Bob chooses a coin from one of the (remaining) ends, and the alternation continues until Bob pockets the last coin. Prove that Alice can play so as to guarantee at least as much as Bob.*"

Solusinya cukup sederhana, yaitu Alice tinggal menghitung lebih besar mana antara kumpulan koin berurutan genap atau berurutan ganjil. Penjelasannya sebagai berikut:

Misal dari koin pertama sampai koin-50 diberi angka sesuai urutannya, maka Alice pada tahap pertama berhak untuk mengambil koin urutan pertama, sehingga meninggalkan koin kedua dan koin ke-50 untuk diambil oleh Bob, atau mengambil koin ke-50 sehingga meninggalkan koin pertama dan ke-49 untuk diambil Bob.

Perhatikan bahwa ketika Alice mengambil koin berurutan ganjil pada tahap pertama, Bob hanya bisa mengambil koin berurutan genap pada tahap yang sama. Begitu pula sebaliknya. Karena Alice berperan sebagai pemain pertama dalam game ini, ia memiliki kontrol game sehingga dapat memaksa dirinya mengambil semua koin urutan ganjil dan Bob mengambil semua koin urutan genap atau sebaliknya. Sehingga Alice hanya perlu mempertimbangkan lebih banyak mana koin berurutan ganjil atau koin berurutan genap.

Lalu berkembang versi lain puzzle ini, selain menentukan apakah Alice bisa *unbeatable* dalam game ini dengan banyak koin genap, ditanyakan juga kemungkinan terbesar koin yang diambil Alice sehingga dia tidak kalah terhadap Bob. Alice dan Bob bermain secara optimal.

Misal urutan koinnya adalah sebagai berikut : 4, 6, 2, 3

Maka solusinya sebagai berikut.

	Tahap	Urutan Koin	Alice	Bob
	0	4 6 2 3	-	-
	1	4 6 2	3	-

		6 2	-	4
	2	2	6	-
		-	-	2
Total	-	-	9	6

Tabel 1 : Solusi optimal, Alice tidak dikalahkan Bob (Sumber : koleksi pribadi)

Ternyata Alice mampu mengambil urutan koin sehingga ia tidak kalah dari Bob. Hal ini disebabkan karena ketika banyak koinnya berjumlah genap, kontrol yang dimiliki Alice untuk mendapatkan hasil yang maksimal sedikit lebih besar daripada Bob. Namun apa yang terjadi jika banyak koinnya ganjil? Simak contoh kasus di bawah.

Misal urutan koinnya adalah

1 5 1 5 1

Maka solusi terbaik yang dapat diambil Alice adalah

	Tahap	Urutan Koin	Alice	Bob
	0	1 5 1 5 1	-	-
	1	5 1 5 1	1	-
		1 5 1	-	5
	2	1 5	1	-
		1	-	5
	3	-	1	-
Total	-	-	3	10

Tabel 2 : Solusi optimal, tetapi Alice dikalahkan Bob (Sumber : koleksi pribadi)

Ternyata Alice tidak berdaya terhadap urutan koin yang seperti ini. Hal ini disebabkan kontrol yang dipegang Alice berbeda ketika banyaknya koin genap.

III. Studi Kasus Coins in a Row

D. Penyelesaian Puzzle dengan strategi Greedy

Setelah melihat puzzle ini, kebanyakan orang pasti akan mencoba *Greedy* sebagai salah satu strategi untuk menyelesaikan masalah puzzle ini. Salah satu metode *Greedy* yang mungkin dipilih adalah mengambil koin terbesar dari salah satu ujung yang tersedia.

Elemen-elemen algoritmanya adalah

1. Himpunan kandidatnya adalah seluruh koin yang tersedia dalam barisan.
2. Himpunan solusinya adalah koin-koin yang sudah terpilih.
3. Fungsi solusinya adalah ketika banyak anggota himpunan solusi sudah setengah dari himpunan kandidat awal.
4. Fungsi seleksinya adalah koin terbesar dari kedua ujung yang tersedia.
5. Fungsi kelayakannya adalah semua koin layak.
6. Fungsi objektifnya adalah memaksimumkan jumlah koinnya.

Namun, pada beberapa kasus, algoritma *Greedy* tidak memberikan hasil yang optimal. Misal saja Bob juga bermain dengan strategi *Greedy*, karena Alice dan Bob diharapkan sama-sama optimal (padahal *Greedy* sudah diklaim tidak memberikan hasil yang optimal pula, bagaimana jika Bob malah menemukan solusi lain yang lebih baik daripada greedy?). Lalu urutan koin yang didapat ternyata sama dengan salah satu contoh di atas, yaitu 4 6 2 3. Maka urutan tahap-tahapnya sebagai berikut.

	Tahap	Urutan Koin	Alice	Bob
	0	4 6 2 3	-	-
	1	6 2 3	4	-
		2 3	-	6
	2	2	3	-
		-	-	2
Total	-	-	7	8

Tabel 3 : Solusi tidak optimal metode *Greedy* (Sumber : koleksi pribadi)

Ternyata *Greedy* masih belum cukup. Mari mencoba strategi algoritma yang lain. *Dynamic Programming* misalnya.

E. Penyelesaian Puzzle dengan strategi Dynamic Programming

Misal pada suatu tahap, koin yang tersedia berada pada indeks ke i hingga indeks ke j. Kita harus mengambil koin sedemikian mungkin hingga mendapatkan hasil maksimum, sementara musuh berusaha untuk meminimalkan hasil koin kita juga dengan cara yang sama-sama optimal. Misal pada tahap tersebut

1. Kita memilih koin ke-i, sehingga koin yang tersedia bagi musuh adalah koin ke-i+1 hingga koin ke-j.
    a. Musuh bisa memilih koin ke-i+1 untuk meminimalkan koin kita, sehingga tersisa koin ke-i+2 hingga koin ke-j. Selanjutnya game akan lanjut ke tahap berikutnya.
    b. Musuh bisa memilih koin ke-j untuk meminimalkan koin kita, sehingga tersisa

koin ke-i+1 hingga koin ke-j-1. Selanjutnya game akan lanjut ke tahap berikutnya.

2. Kita memilih koin ke-j, sehingga koin yang tersedia bagi musuh adalah koin ke-i hingga koin ke-j-1.
   a. Musuh bisa memilih koin ke-i untuk meminimalkan koin kita, sehingga tersisa koin ke-i+1 hingga koin ke-j-1. Selanjutnya game akan berlanjut ke tahap berikutnya.

   b. Musuh bisa memilih koin ke-j-1, sehingga tersisa koin ke-i hingga koin ke-j-2. Selanjutnya game akan lanjut ke tahap berikutnya.

Dari penjelasan di atas, sudah terbentuk bagaimana tiap tahap dari persoalan dan statusnya masing-masing. Namun bagaimana dengan hubungan antar tahapnya? Kita tahu jika tinggal 2 koin saja (i+1 == j), maka kita pilih yang maksimum dari keduanya. Jika tinggal 1 koin (i == j), ambil koin tersebut. Dan jika i > j, berikan 0.

Misal untuk mencari solusi, kita menggunakan fungsi Solusi(coins, i, j) dengan i dan j menandakan indeks awal dan akhir dari kumpulan koin yang tersedia atau belum diambil pada barisan coins.

Perhatikan kasus pertama, yaitu ketika kita memilih koin ke-i, musuh akan memilih antara koin ke-i+1 dan koin ke-j sehingga solusi dari kumpulan koin yang tersedia sangat minimal. Ingat bahwa musuh bermain sama cerdasnya dengan kita.

Maka dari kasus-1, musuh akan meninggalkan koin sisa yang bisa kita ambil sebanyak

$$min(Solusi(coins,\ i+2,\ j)\ ,\ Solusi(coins,\ i+1,\ j-1))$$

Dengan cara yang sama, pada kasus-2, musuh akan meninggalkan koin sisa yang bisa kita ambil sebanyak

$$min(Solusi(coins,\ i,\ j-2)\ ,\ Solusi(coins,\ i+1,\ j-1))$$

Sehingga pada suatu tahap tertentu, kita harus memilih kasus mana yang akan menghasilkan solusi terbesar. Sehingga kita akan punya

$$max(coins_i + koinSisaKasus1, coins_j + koinSisaKasus2)$$

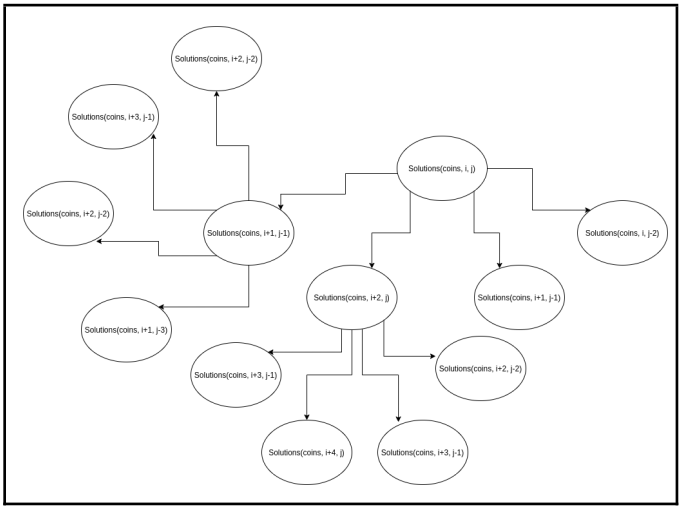
Lebih jelasnya perhatikan potongan fungsi dalam Javascript ini

```
const Solution = (coins, i, j) => {
  if (i > j) return 0;
  if (i === j) return coins[j];
  if (i + 1 === j) return Math.max(coins[i], coins[j]);
  return Math.max(
    coins[i] + Math.min(
      Solution(coins, i+1, j-1),
      Solution(coins, i+2, j)),
    coins[j] + Math.min(
      Solution(coins, i, j-2),
      Solution(coins, i+1, j-1)));
}
```

Gambar 4 : fungsi Solusi yang memanfaatkan hubungan antartahap secara rekursif dan naif (Sumber : koleksi pribadi)

Namun, fungsi di atas masih belum termasuk *Dynamic Programming*, melainkan masih bersifat *Naive Recursive*. yang masih mengalami masalah overlap antar solusinya.

Perhatikan gambar di bawah ini



Gambar 5: Recursion tree yang terbentuk (Sumber: koleksi pribadi)

Jika gambar terlalu kecil, gambar sudah tersedia di pranala ini. Terlihat pada gambar recursion tree yang belum lengkap di atas, terdapat subproblem yang sama tetapi berulang kali dipanggil. Hal ini menyebabkan kompleksitas algoritmanya menjadi eksponensial. Namun, *Dynamic Programming* mampu menyederhanakan kompleksitasnya menjadi $O(N^2)$.

Kemampuan *Dynamic Programming* dalam menyimpan solusi *subproblem* yang sudah pernah dikerjakan mampu memecahkan permasalahan *overlap* ini. Biasanya solusi *subproblem* yang sudah pernah dikerjakan ditaruh ke dalam tabel, yang isi tabel cenderung bertambah seiring dengan bertambahnya *subproblem* yang sudah dikerjakan. Oleh sebab itu metode ini dinamakan *Dynamic Programming* atau Program Dinamis.

Mari selesaikan permasalahan ini dengan 2 pendekatan, yaitu *Bottom-up* dan *Top-down*.

Pendekatan *Bottom-up* atau tabulasi atau pendekatan maju, merupakan pendekatan yang dimulai dari basis, lalu merekonstruksi tahap-tahapnya (*subproblem*) hingga mencapai tahap terakhir (problem awal). Relasi antar tahapannya masih sama dengan relasi yang ada pada metode *Naive Recursive*. Berikut potongan fungsinya dalam Javascript.

```
const calculate = (array, i, j) => {
  return i <= j ? array[i][j] : 0;
}
const Solution = (coins) => {
  const n = coins.length;
  if (n === 1 ) return coins[0];
  if (n === 2) return Math.max(coins[0], coins[1]);
```

```
const T = new Array(n).
  fill(0).
  map(() =>
  new Array(n).fill(0));
for (let x = 0; x < n; x++){
  let i = 0;
  let j = x;
  while (j < n) {
    const start =
      coins[i] +
      Math.min(
        calculate(T, i+2, j),
        calculate(T, i+1, j-1));
    const end =
      coins[j] +
      Math.min(
        calculate(T, i+1, j-1),
        calculate(T, i, j-2));
    T[i][j] = Math.max(start,end);
    i++;
    j++;
  }
 }
 return T[0][n-1];
}
```

Gambar 6: Pendekatan maju (Sumber : Koleksi pribadi)

Mari berpikir dengan perspektif yang berbeda pada pendekatan ini. Pada potongan fungsi di atas, pengisian matriks 2D dilakukan terurut dari diagonal utama lalu membentuk matriks segitiga atas. Elemen matriks pada indeks baris ke i dan indeks kolom ke j adalah solusi optimal jika jajaran koinnya hanya *subset* dari jajaran koin asli dengan elemen pertama *subset*-nya adalah elemen berindeks i di dalam jajaran koin asli dan elemen terakhir *subset*-nya adalah elemen berindek j di dalam jajaran koin asli.

Dari *subproblem* yang lebih kecil, dapat dikonstruksi menjadi *problem* awal menggunakan relasi rekursif yang sudah didefinisikan. Hingga untuk mengambil solusi optimal dari jajaran koin asli, kita hanya perlu mengambil elemen pojok kanan atasnya, atau elemen dengan indeks baris 0 dan indeks kolom ke-n-1 (*subset*-nya berupa jajaran koin asli itu sendiri).

Mari lakukan pendekatan yang terakhir. Pendekatan *Top-down* atau memoisasi atau *memory function* atau pendekatan mundur, adalah pendekatan Program dinamis dalam memecah masalah dari problem utama (tahap akhir) ke *subproblem* yang lebih kecil (tahap-tahap pendahulunya). Pengerjaannya secara rekursif. Lalu apa perbedaannya dengan metode *Naive Recursive* di atas? Perbedaannya adalah tiap-tiap *subproblem* yang telah dikerjakan disimpan dalam tabel/matriks, lalu sebelum mengerjakan *subproblem* baru, dicek apakah *subproblem* tersebut pernah dikerjakan atau tidak, jika pernah dikerjakan maka jangan kerjakan lagi dengan cara rekursif, tetapi panggil saja elemen tabel/matriks yang bersesuaian. Jika ternyata belum pernah dikerjakan, baru *subproblem* tersebut dikerjakan secara rekursif, lalu di akhir simpan hasilnya ke dalam tabel/matriks. Berikut potongan fungsinya dalam Javascript.

```
const Solution = (coins, i, j, lookup) => {
  if (i > j) return 0;
  if (i === j) return coins[i];
  if (i+1 === j) return Math.max(coins[i], coins[j]);
  if (lookup[i][j] === -1){
    const start =
      coins[i] +
      Math.min(
        Solution(coins, i+1, j-1),
        Solution(coins, i+2, j));
    const end =
      coins[j] +
      Math.min(
        Solution(coins, i, j-2),
        Solution(coins, i+1, j-1));
    lookup[i][j] = Math.max(start, end);
  }
  return lookup[i][j];
}
```

Gambar 7 : Pendekatan mundur (Sumber : Koleksi pribadi)

Matriks lookup adalah matriks yang akan menyimpan jawaban solusi *subproblem* yang telah diselesaikan. Matriksnya harus berukuran NxN yang diinisialisasi dengan -1 di semua elemennya. Contoh pemanggilannya adalah

```
console.log(Solution([4, 6, 2, 3], 0, 3,
new Array(4).fill(new Array(4).fill(-1))));
```

Gambar 8 : Contoh pemanggilan solusi dengan matriks NxN yang berisi -1 (Sumber: koleksi pribadi)\

F. Penyelesaian Puzzle dengan strategi algoritma yang ditemukan Tomasz Idziaszek

Algoritma yang ditemukan Tomasz didasari oleh 3 teorema, pertama adalah *Greedy Move Principle*, lalu ada *Fusion Principle*, dan ada *Fruitless Move Principle*. Ternyata dengan menggunakan 2 teorema pertama, kita bisa menyelesaikan puzzle ini dengan merepresentasikan jajaran koinnya dengan stack (berbeda dengan pendekatan yang dilakukan Tomasz, ia menggunakan representasi graph).

Greedy Move Principle kira-kira berbunyi: “Jika terdapat koin pada ujung barisan koin yang nilainya tidak lebih kurang dari seluruh nilai koin lainnya pada barisan, maka koin tersebut dapat diambil”. Prinsip ini memperhatikan fakta bahwa jika salah satu koin terbesar dalam barisan yang masih ada diambil, maka kita tidak akan pernah rugi walau misalnya pengambilan koin tersebut menyebabkan musuh mengambil koin lain yang mungkin sama besar, atau lebih baiknya kurang dari koin yang kita ambil. Namun, prinsip ini tidak bisa berdiri sendiri, sehingga kita perlu prinsip kedua.

Fusion Principle kira-kira berbunyi: “Jika terdapat 3 koin yang berurutan dan koin kedua lebih besar daripada koin pertama maupun koin ketiga, maka kita dapat meleburkan 3 koin tersebut menjadi koin baru dengan nilai koinnya

sebanyak nilai koin pertama dikurangi nilai koin kedua lalu ditambah nilai koin ketiga.

Kedua prinsip di atas dapat diterapkan pada stack yang akan menyimpan koin-koin. Lalu dari stack tersebut diurutkan berdasarkan *non-increasing order*. Lalu kita dapat menghitung profit (selisih) yang didapat dari musuh dengan rumus:

$$profit = f(v_1) - f(v_2) + \ldots + (-1)^{k+1} f(v_k)$$

Setelah mendapat profit terbesar yang dapat kita ambil, kita bisa menentukan dengan mudah berapa koin maksimum yang dapat kita ambil. Berikut potongan algoritma dengan pendekatan yang ditemukan Tomasz.

```
const Solution = (coins) => {
  const stack = [];
  for (let i of coins){
    stack.push(i);
    let fail = false;
    while (stack.length >= 3 && !fail){
      let a = stack.pop();
      let b = stack.pop();
      let c = stack.pop();
      if (a < b && b > c){
        stack.push(a - b + c);
      }else{
        fail = true;
        stack.push(c);
        stack.push(b);
        stack.push(a);
      }
    }
  }
  stack.sort((a,b) => b - a);
  let selisih = 0;
  let sign = 1;
  for (let i of stack){
    selisih += i*sign;
    sign = -1*sign;
  }
  const total = coins.reduce((a,b) => a+b);
  return (total+selisih)/2;
}
```

Gambar 9 : Pendekatan yang ditemukan Tomasz Idziaszek (Sumber : koleksi pribadi, terinspirasi dari Tomasz Idziaszek)

G. *Pengujian Algoritma*

Diberikan barisan koin dan kita akan mencoba menyelesaikannya dengan berbagai algoritma, dimulai dari algoritma yang ditemukan Tomasz hingga ke *Greedy*. Barisan koinnya adalah 20, 30, 2, 3, 4, 10, 234, 345, 12, 24, 45, 34, 1, 3, 6, 87, 26, 45, 89, 89, 23, 52, 63, 87, 80, 43, 22, 12, 45, 12, 1, 3. Dengan panjang barisannya 32 koin.

Dengan algoritma yang ditemukan Tomasz, kita mendapatkan

```
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node Tomasz.js
879
Tomasz's Algorithm: 11.586ms
```

Gambar 10 : Pengujian menggunakan algoritma Tomasz (Sumber : koleksi pribadi)

Ternyata didapatkan nilai maksimal yang dapat diambil adalah 879, dengan lama yang dibutuhkan untuk menghitung hasilnya adalah 11.568 ms.

Sekarang kita uji algoritma DP secara maju dan secara mundur.

```
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node DPforward.js
879
DP Forward Approach: 10.888ms
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node DPbackward
879
DP Backward Approach: 13.13ms
```

Gambar 11 : Pengujian menggunakan algoritma DP dengan dua pendekatan (Sumber : koleksi pribadi)

Ternyata hasil yang didapatkan sama, yaitu sama-sama 879, sama juga dengan hasil yang didapatkan oleh algoritma Tomasz di atas. Lalu mengapa waktu yang dibutuhkan cenderung sama dengan algoritma Tomasz di atas, padahal algoritma DP membutuhkan kompleksitas waktu $O(N^2)$ sedangkan algoritma buatan Tomasz hanya butuh kompleksitas waktu O(N)? Karena panjang barisannya masih tergolong rendah, hanya 32, jadi tidak terlihat perbedaannya.

Sekarang mari kita uji algoritma *Naive Recursive* dengan kasus yang sama. Didapatkan hasil

```
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node NaiveCoinsinaRow.js
879
Naive Recursive Approach  with 32 coins: 28.998s
```

Gambar 12 : Pengujian menggunakan algoritma *Naive Recursive* (Sumber : koleksi pribadi)

Hasil yang didapatkan masih sama yaitu 879, tetapi waktu yang dibutuhkan jauh lebih lama, yaitu 28.998 s. Hal ini disebabkan oleh permasalahan *overlapping subproblem* sehingga kompleksitas algoritmanya menjadi eksponensial.

Selanjutnya mari kita uji dengan algoritma *Greedy* tetapi musuh menggunakan algoritma *Greedy* juga untuk meminimalkan koin kita. Didapatkan

```
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node GreedyCoinsinaRow.js
749
Greedy with enemy using greedy too: 10.928ms
```

Gambar 13 : Pengujian dengan menggunakan algoritma *Greedy* dengan musuh menggunakan *Greedy* juga (Sumber: koleksi pribadi)

Ternyata koin yang dihasilkan berbeda, yaitu 749, lebih sedikit dari hasil algoritma sebelumnya. Hal ini membuktikan bahwa algoritma *Greedy* tidak optimal. Bagaimana jika musuh malah menggunakan DP? Didapatkan

```
wiwid@wiwid ~/Desktop/kuliah/Semester4/STIMA/Makalah node GreedyButEnemyOptimum.js
654
Greedy with Enemy using DP: 21.779ms
```

Gambar 14 : Pengujian dengan menggunakan algoritma *Greedy* dengan musuh menggunakan DP (Sumber : koleksi pribadi)

Ternyata koin yang didapatkan lebih sedikit, yaitu 654, sehingga semakin menunjukkan bahwa algoritma DP lebih baik daripada *Greedy*.

IV. Kesimpulan

Coins in a Row merupakan salah satu puzzle yang bagus. Puzzle ini dapat mengecoh pemainnya yang mengira strategi terbaik dalam menyelesaikan puzzle ini adalah strategi *Greedy*. Ternyata strategi *Greedy* tidak selalu menghasilkan jawabannya yang paling optimal. Lalu, puzzle ini ternyata juga dapat diselesaikan dengan pendekatan *Naive Recursive*, hasil yang dihasilkan tepat, tetapi kompleksitas algoritmanya sangat tinggi. Sehingga diperlukan *Dynamic Programming* untuk memangkas kompleksitas algoritmanya menjadi $O(N^2)$. Lalu berkat penemuan Tomasz Idziaszek, permasalahan ini dapat dipangkas menjadi lebih sederhana, dengan penyelesaiannya hanya memerlukan kompleksitas algoritma $O(N)$.

Penulis berharap ada lagi orang seperti Tomasz yang mampu menyederhanakan permasalahan yang ada menjadi lebih sederhana. Misalnya menyederhanakan permasalahan klasik sepert *Travelling Salesman Problem* hingga mendapatkan solusi dengan kompleksitas yang lebih baik.

Link Video Youtube dan Github

Video terdapat di pranala https://youtu.be/hZcUjCfRcDM dan repositori untuk pengujian algoritma bisa didapatkan dalam https://github.com/widyaput/MakalahStima.

Ucapan Terima kasih

Penulis bersyukur kepada Tuhan Yang Maha Esa karena atas berkat-Nya, penulis mampu menyelesaikan makalah berjudul "Penggunaan Berbagai Algoritma dalam penyelesaian *Coins in a Row Puzzle*". Penulis mengucapkan terima kasih kepada orang tua, sahabat-sahabat, dan teman-teman penulis yang setia memberikan dukungan, motivasi, dan masukan terhadap penyelesaian tugas pembuatan makalah ini. Penulis juga berterima kasih kepada para dosen mata kuliah IF2211 Strategi Algoritma pada semester II 2020/2021, Bapak Rinaldi Munir, Ibu Masayu Leylia Khodra, dan Ibu Nur Ulfa Maulidevi, atas ilmu yang sudah diberikan pada penulis.

Referensi

[1] Munir, Rinaldi . 2021. "Algoritma Greedy Bagian 1" . https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf. (Diakses pada 10 Mei 2021, Pukul 20.00 WIB)

[2] Munir, Rinaldi. 2020. "Program Dinamis Bagian 1". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf . (Diakses pada 10 Mei 2021, Pukul 20.00 WIB)

[3] Idziaszek, Tomasz. 2012. "An Optimal Algorithm for Calculating the Profit in the Coins in a Row Game". https://www.mimuw.edu.pl/~idziaszek/termity/termity.pdf. (Diakses pada 10 Mei 2021, Pukul 20.00 WIB)

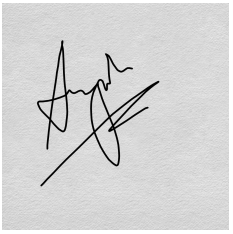
[4] Winkler, Peter. 2004. "Mathematical Puzzles: A Connoisseur's Collection".

[5] Techiedelight.com, 2021. "Pots of Gold Game Problem using Dynamic Programming". https://www.techiedelight.com/pots-gold-game-dynamic-programming/ (Diakses pada 10 Mei 2021, Pukul 20.00 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, 11 Mei 2021



Widya Anugrah Putra