Nama : Elia Jose Alvaro Rahayaan

Kelas : TI23G

NIM : 20230040163

Mata Kuliah : Kompleksitas Algoritma

TUGAS
Susunlah implementasi dan analisis untuk menyelesaikan masalah berikut menggunakan algoritma backtracking. Sajikan hasil pengamatan dalam bentuk laporan esai yang mencakup kode program, penjelasan terperinci, serta analisis mendalam dari hasil yang diperoleh!

ALGORITMA BACKTRACKING

Backtracking adalah sebuah metode pemecahan masalah yang menggunakan pendekatan rekursif dengan menjelajahi setiap kemungkinan solusi untuk menemukan hasil terbaik. Algoritma ini sering diterapkan pada permasalahan kombinatorik atau optimasi, di mana tujuannya adalah menentukan solusi optimal dari berbagai pilihan yang tersedia. Proses backtracking bekerja dengan mencoba setiap opsi satu per satu. Jika suatu opsi ternyata tidak mengarah pada solusi yang diinginkan, algoritma akan kembali ke langkah sebelumnya untuk mengevaluasi pilihan lain. Pendekatan ini sangat berguna untuk masalah yang memerlukan pengambilan keputusan secara bertahap dan tidak memungkinkan untuk mengevaluasi semua kemungkinan secara langsung karena jumlahnya terlalu besar.

BAGIAN 1: Implementasi Masalah "Subset Sum"

1. **Cara Kerja Algoritma**
   Algoritma mengeksplorasi setiap kemungkinan kombinasi elemen dalam array untuk mencapai target yang diinginkan.

   a. Algoritma menentukan apakah suatu elemen akan dimasukkan ke dalam subset yang sedang dibangun atau dilewati.
   b. Setiap kombinasi subset diperiksa sampai algoritma menemukan kombinasi yang memenuhi target atau seluruh elemen dalam array telah diproses.

2. Kondisi Penghentian

   a. Sukses: Jika total elemen dalam subset mencapai nilai target, subset tersebut ditambahkan ke dalam daftar hasil.

b. Gagal: Jika total elemen dalam subset melebihi nilai target atau indeks melampaui panjang array, proses dihentikan, dan algoritma kembali ke langkah sebelumnya.

3. Mekanisme Backtracking

a. Jika subset tidak memenuhi kriteria (misalnya, total nilai elemen melampaui target), elemen terakhir dihapus dari subset menggunakan operasi pop() untuk mencoba kombinasi lainnya.
b. Pendekatan ini memungkinkan algoritma untuk mengabaikan cabang pencarian yang tidak valid dan kembali ke tahap sebelumnya, memeriksa opsi lain dalam struktur pohon pencarian.

CODE PROGRAM

Kode program nya adalah seperti berikut :

```python
def find_subsets_with_target_sum(arr, target_sum):
    solutions = []

    def explore(subset, total, start):
        # Jika total saat ini sama dengan target_sum, simpan subset
        if total == target_sum:
            solutions.append(subset[:])  # Salin subset saat ini
            return

        # Jika total melebihi target atau indeks di luar jangkauan, hentikan eksplorasi
        if total > target_sum or start >= len(arr):
            return

        # Coba tambahkan elemen saat ini ke subset
        subset.append(arr[start])
        explore(subset, total + arr[start], start)  # Tetap di indeks yang sama

        # Kembalikan keadaan sebelum mencoba elemen berikutnya
        subset.pop()
        explore(subset, total, start + 1)  # Lanjutkan ke elemen berikutnya

    explore([], 0, 0)
    return solutions

# Contoh penggunaan
numbers = [2, 4, 6, 8, 10]
target_value = 10
result = find_subsets_with_target_sum(numbers, target_value)
print(f"Semua subset dengan jumlah {target_value}: {result}")
```

OUTPUT

Output dari code tersebut adalah sebagai berikut :

```
Semua subset dengan jumlah 10: [[2, 2, 2, 2, 2], [2, 2, 2, 4], [2, 2, 6], [2, 4, 4], [2, 8], [4, 6], [10]]
PS C:\Users\riang\vscode Elia>
```

1. **Input**: array = [2, 4, 6, 8], **target**: 10
   - **Hasil**: [[2, 8], [4, 6]]
2. **Input**: array = [1, 5, 9, 12], **target**: 14
   - **Hasil**: [[1, 5, 9]]
3. **Input**: array = [3, 6, 7], **target**: 9

- **Hasil**: [[3, 6]]

Algoritma ini sangat cocok untuk menyelesaikan masalah dengan ukuran subset yang kecil. Namun, ketika berhadapan dengan array yang lebih besar atau nilai target yang tinggi, kompleksitasnya bertambah secara eksponensial karena harus memeriksa semua kemungkinan kombinasi.

BAGIAN 2: Analisis Masalah "Permutasi"

1. **Bagaimana Algoritma Backtracking Digunakan untuk Membentuk Permutasi?**
   a. Algoritma memilih setiap elemen dalam array sebagai langkah awal pembentukan permutasi.
   b. Dengan pendekatan rekursif, elemen-elemen berikutnya yang belum digunakan ditambahkan ke jalur (path) hingga membentuk permutasi lengkap dengan panjang yang sama seperti array asli.
   c. Setelah permutasi lengkap dibuat, algoritma mundur (backtrack) untuk mencoba kombinasi lain dengan menandai elemen terakhir sebagai "belum digunakan" dan menghapusnya dari jalur.
2. **Bagaimana Algoritma Mencegah Pengulangan Elemen?**
   a. Sebuah array boolean used digunakan untuk mencatat elemen yang telah dimasukkan ke dalam permutasi saat ini.
   b. Sebelum menambahkan elemen baru ke jalur, algoritma memeriksa apakah elemen tersebut sudah digunakan. Jika sudah, elemen tersebut dilewati untuk menghindari pengulangan.
3. **Apakah Algoritma Dapat Dioptimalkan?**
   a. **Optimalisasi untuk Input yang Mengandung Duplikasi**
   Jika array memiliki elemen yang sama, algoritma dapat diubah untuk memeriksa kondisi tambahan guna menghindari duplikasi. Contohnya, jika elemen saat ini identik dengan elemen sebelumnya dan elemen sebelumnya belum dipakai dalam jalur saat ini, elemen tersebut dapat dilewati.
   b. **Mengurangi Rekursi yang Tidak Perlu**
   Saat ini, algoritma menggunakan array boolean used, yang memerlukan pemeriksaan tambahan pada setiap langkah. Sebagai alternatif, elemen dapat dihapus langsung dari array saat digunakan dan ditambahkan kembali ketika mundur (backtracking).
   c. **Iterasi Lebih Efisien**
   Dengan memanfaatkan struktur data seperti set atau heap, elemen dapat diproses dalam urutan tertentu atau duplikasi dapat dihindari secara langsung, sehingga meningkatkan efisiensi.

```python
def permute(array):
    result = []

    def backtrack(path, used):
        # Jika panjang path sama dengan panjang array, tambahkan ke hasil
        if len(path) == len(array):
            result.append(list(path))
            return

        # Iterasi melalui elemen array
        for i in range(len(array)):
            # Lewati elemen yang sudah digunakan
            if used[i]:
                continue

            # Tandai elemen sebagai digunakan dan tambahkan ke path
            used[i] = True
            path.append(array[i])

            # Lanjutkan eksplorasi
            backtrack(path, used)

            # Backtrack: lepaskan elemen terakhir dan tandai sebagai tidak digunakan
            path.pop()
            used[i] = False

    # Awali dengan path kosong dan semua elemen belum digunakan
    backtrack([], [False] * len(array))
    return result

# Contoh input
array = [1, 2, 3, 4, 5]

# Panggil fungsi dan tampilkan hasil
result = permute(array)
print("Semua permutasi dari", array, ":", result)
```

OUTPUT

```
Semua permutasi dari [1, 2, 3, 4, 5] : [[1, 2, 3, 4, 5], [1, 2, 3, 5, 4], [1, 2, 4, 3, 5], [1, 2, 4, 5, 3], [1,
 2, 5, 3, 4], [1, 2, 5, 4, 3], [1, 3, 2, 4, 5], [1, 3, 2, 5, 4], [1, 3, 4, 2, 5], [1, 3, 4, 5, 2], [1, 3, 5, 2,
 4], [1, 3, 5, 4, 2], [1, 4, 2, 3, 5], [1, 4, 2, 5, 3], [1, 4, 3, 2, 5], [1, 4, 3, 5, 2], [1, 4, 5, 2, 3], [1,
4, 5, 3, 2], [1, 5, 2, 3, 4], [1, 5, 2, 4, 3], [1, 5, 3, 2, 4], [1, 5, 3, 4, 2], [1, 5, 4, 2, 3], [1, 5, 4, 3,
2], [2, 1, 3, 4, 5], [2, 1, 3, 5, 4], [2, 1, 4, 3, 5], [2, 1, 4, 5, 3], [2, 1, 5, 3, 4], [2, 1, 5, 4, 3], [2, 3
, 1, 4, 5], [2, 3, 1, 5, 4], [2, 3, 4, 1, 5], [2, 3, 4, 5, 1], [2, 3, 5, 1, 4], [2, 3, 5, 4, 1], [2, 4, 1, 3, 5
], [2, 4, 1, 5, 3], [2, 4, 3, 1, 5], [2, 4, 3, 5, 1], [2, 4, 5, 1, 3], [2, 4, 5, 3, 1], [2, 5, 1, 3, 4], [2, 5,
 1, 4, 3], [2, 5, 3, 1, 4], [2, 5, 3, 4, 1], [2, 5, 4, 1, 3], [2, 5, 4, 3, 1], [3, 1, 2, 4, 5], [3, 1, 2, 5, 4]
, [3, 1, 4, 2, 5], [3, 1, 4, 5, 2], [3, 1, 5, 2, 4], [3, 1, 5, 4, 2], [3, 2, 1, 4, 5], [3, 2, 1, 5, 4], [3, 2,
4, 1, 5], [3, 2, 4, 5, 1], [3, 2, 5, 1, 4], [3, 2, 5, 4, 1], [3, 4, 1, 2, 5], [3, 4, 1, 5, 2], [3, 4, 2, 1, 5],
 [3, 4, 2, 5, 1], [3, 4, 5, 1, 2], [3, 4, 5, 2, 1], [3, 5, 1, 2, 4], [3, 5, 1, 4, 2], [3, 5, 2, 1, 4], [3, 5, 2
, 4, 1], [3, 5, 4, 1, 2], [3, 5, 4, 2, 1], [4, 1, 2, 3, 5], [4, 1, 2, 5, 3], [4, 1, 3, 2, 5], [4, 1, 3, 5, 2],
[4, 1, 5, 2, 3], [4, 1, 5, 3, 2], [4, 2, 1, 3, 5], [4, 2, 1, 5, 3], [4, 2, 3, 1, 5], [4, 2, 3, 5, 1], [4, 2, 5,
 1, 3], [4, 2, 5, 3, 1], [4, 3, 1, 2, 5], [4, 3, 1, 5, 2], [4, 3, 2, 1, 5], [4, 3, 2, 5, 1], [4, 3, 5, 1, 2], [
4, 3, 5, 2, 1], [4, 5, 1, 2, 3], [4, 5, 1, 3, 2], [4, 5, 2, 1, 3], [4, 5, 2, 3, 1], [4, 5, 3, 1, 2], [4, 5, 3,
2, 1], [5, 1, 2, 3, 4], [5, 1, 2, 4, 3], [5, 1, 3, 2, 4], [5, 1, 3, 4, 2], [5, 1, 4, 2, 3], [5, 1, 4, 3, 2], [5
, 2, 1, 3, 4], [5, 2, 1, 4, 3], [5, 2, 3, 1, 4], [5, 2, 3, 4, 1], [5, 2, 4, 1, 3], [5, 2, 4, 3, 1], [5, 3, 1, 2
, 4], [5, 3, 1, 4, 2], [5, 3, 2, 1, 4], [5, 3, 2, 4, 1], [5, 3, 4, 1, 2], [5, 3, 4, 2, 1], [5, 4, 1, 2, 3], [5,
 4, 1, 3, 2], [5, 4, 2, 1, 3], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2], [5, 4, 3, 2, 1]]
```

UJI COBA

1. **Input: [3, 2, 1]**
    a. **Output: [[3, 2, 1], [3, 1, 2], [2, 3, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3]]**
2. **Input: [4, 5]**
    a. **Output: [[4, 5], [5, 4]]**
3. **Input: [1, 2, 2] (dengan optimalisasi untuk mengatasi elemen duplikat)**
    a. **Output setelah optimalisasi: [[1, 2, 2], [2, 1, 2], [2, 2, 1]]**

Algoritma ini bekerja dengan baik untuk array berukuran kecil hingga sedang. Namun, untuk array dengan elemen yang lebih besar, jumlah permutasi yang dihasilkan meningkat secara eksponensial. Oleh karena itu, diperlukan strategi optimalisasi untuk meningkatkan efisiensi baik dalam hal waktu maupun penggunaan memori.

BAGIAN 3: Kesimpulan dan Eksperimen

1. **Perbandingan Kinerja Algoritma Backtracking pada Kedua Masalah**

[Bagian 1]

- **Input**: array = [2, 4, 6, 8, 10], **target**: 10
- **Hasil**: [[2, 2, 2, 2, 2], [2, 2, 2, 4], [2, 2, 6], [2, 4, 4], [2, 8], [4, 6], [10]]
- **Jumlah Langkah**: 7 langkah

[Bagian 2]

- **Input**: array = [1, 2, 3, 4, 5]
- **Hasil**: Kombinasi permutasi dari lima elemen menghasilkan 120 urutan yang berbeda, seperti:
    a. [[1, 2, 3, 4, 5], [1, 2, 3, 5, 4], ..., [5, 4, 3, 2, 1]]
- **Jumlah Langkah**: 120 langkah

Analisis

- Untuk masalah **Subset Sum** (Bagian 1), algoritma memerlukan langkah yang lebih sedikit karena hanya mencari kombinasi elemen yang memenuhi kriteria jumlah target.
- Sebaliknya, pada masalah **Permutasi** (Bagian 2), jumlah langkah meningkat drastis karena algoritma menghasilkan semua urutan kemungkinan dari elemen dalam array.
- Dengan bertambahnya ukuran array, kompleksitas algoritma backtracking pada masalah permutasi meningkat secara faktorial, sedangkan pada masalah subset, peningkatan terjadi lebih moderat tergantung pada jumlah target dan elemen dalam array.

Kesimpulan

- Algoritma backtracking efektif untuk menyelesaikan kedua masalah ini, tetapi lebih efisien digunakan pada dataset dengan ukuran kecil hingga sedang. Optimalisasi tambahan dapat diterapkan untuk memperbaiki efisiensi pada array besar atau kondisi dengan banyak elemen duplikat.

2. Diskusi

Backtracking itu seperti mencoba berbagai kunci untuk membuka sebuah peti harta karun. Kita mencoba satu per satu kombinasi sampai menemukan kunci yang pas. Ini adalah cara yang sangat fleksibel dan sederhana untuk menyelesaikan masalah seperti mencari subset, permutasi, atau menentukan posisi yang tepat dalam berbagai skenario. Dengan pendekatan yang sistematis, kita bisa memastikan tidak ada kemungkinan solusi yang terlewatkan.

Namun, masalahnya muncul kalau jumlah kunci (atau elemen) terlalu banyak. Bayangkan ada ribuan kunci yang harus dicoba—waktu dan tenaga yang dibutuhkan akan meledak seperti balon yang terlalu ditiup. Jumlah kemungkinan kombinasi atau permutasi akan tumbuh secara eksponensial, yang artinya semakin banyak elemen, semakin tak terkendali waktu proses dan penggunaan memorinya. Ini membuat backtracking menjadi kurang cocok untuk masalah dengan input yang besar.

Tapi, ada trik untuk mengatasi ini, seperti memotong cabang pohon pencarian lebih awal. Misalnya, kalau kita tahu suatu kunci tidak mungkin cocok, kenapa harus terus mencobanya? Teknik ini disebut **pruning**, dan sangat membantu untuk menghemat waktu. Contohnya, dalam masalah subset sum, jika jumlah elemen yang kita pilih sudah melebihi target, kita bisa langsung berhenti tanpa perlu melanjutkan.

Selain itu, kita juga bisa lebih pintar dalam memilih langkah berikutnya. Misalnya, memilih elemen yang lebih mungkin menghasilkan solusi terlebih dahulu atau menggunakan struktur data seperti heap atau set untuk mempercepat pencarian. Ini seperti memilih kunci yang lebih besar dulu kalau kita tahu peti harta karunnya besar.

Intinya, backtracking itu seperti metode coba-coba yang cerdas, tapi kalau tidak diatur dengan strategi yang tepat, bisa makan waktu dan energi. Dengan teknik-teknik seperti pruning dan heuristik, kita bisa membuatnya lebih efisien dan relevan untuk masalah yang lebih rumit. Jadi, meskipun bukan solusi sempurna untuk semua masalah, dengan sedikit modifikasi, backtracking tetap bisa menjadi alat yang kuat di kotak peralatan pemrograman kita.