SESSION 9 ASSIGNMENT

ALGORITMA BACKTRACKING

NIM	: 20230040040
Full Name	: Bilqis Zahra
Class	: TI23G
Lecturer	: Alun Sujjada, S.Kom, M.T
Course	: Kompleksitas Algoritma

Assignment
Lakukan implementasi dan analisis terhadap masalah berikut menggunakan algoritma backtracking. Tuliskan hasil praktik dalam bentuk laporan essay, lengkap dengan kode program, penjelasan, dan analisisnya!

ALGORITMA BACKTRACKING

Algoritma backtracking adalah metode penyelesaian masalah yang menggunakan pendekatan rekursif dengan mencoba setiap kemungkinan langkah untuk menemukan solusi terbaik. Biasanya, algoritma ini digunakan pada masalah kombinatorik atau optimasi, di mana tujuannya adalah mencari solusi optimal dari berbagai kemungkinan yang tersedia. Cara kerja backtracking adalah dengan menjelajahi setiap opsi satu per satu, dan jika suatu opsi tidak membawa ke solusi yang benar, algoritma akan mundur ke langkah sebelumnya untuk mencoba opsi lain. Pendekatan ini sering digunakan pada masalah yang membutuhkan pengambilan keputusan bertahap dan tidak memungkinkan untuk memeriksa semua kemungkinan secara langsung karena jumlahnya yang terlalu banyak.

BAGIAN 1 : Implementasi Masalah "Subset Sum"

1. **Bagaimana Algoritma Bekerja**
   Algoritma mencoba semua kemungkinan kombinasi elemen dari array untuk mencapai target.
   a. Algoritma memutuskan apakah akan menyertakan elemen tersebut dalam subset saat ini atau melewatinya.
   b. Setiap kombinasi subset diuji hingga algoritma mencapai target atau melampaui batas array.

2. **Kondisi Stopping**
   a. **Berhasil :** Jika jumlah elemen dalam subset sama dengan nilai target, subset disimpan dalam hasil.
   b. **Gagal :** Jika jumlah elemen dalam subset lebih besar dari target atau indeks melampaui panjang array, algoritma berhenti dan kembali ke langkah sebelumnya.

3. **Proses Backtracking**
   a. Jika subset gagal memenuhi kriteria (misalnya, jumlah melebihi target), elemen terakhir dikeluarkan menggunakan pop() untuk mencoba kombinasi lain.

b. Dengan cara ini, algoritma menghapus cabang yang tidak valid dan kembali ke tingkat sebelumnya dalam pohon pencarian.

CODE PROGRAM

bagian1.py > ...

```python
def subset_sum_backtrack(array, target):
    result = []

    def backtrack(current_subset, current_sum, index):
        # Kasus dasar: jika jumlah saat ini sama dengan target, tambahkan ke hasil
        if current_sum == target:
            result.append(list(current_subset))
            return

        # Jika jumlah melebihi target atau indeks melampaui panjang array, berhenti
        if current_sum > target or index >= len(array):
            return

        # Tambahkan elemen saat ini ke subset
        current_subset.append(array[index])
        backtrack(current_subset, current_sum + array[index], index)

        # Keluarkan elemen terakhir untuk mencoba kombinasi lain
        current_subset.pop()
        backtrack(current_subset, current_sum, index + 1)

    backtrack([], 0, 0)
    return result

# Contoh masukan
array = [2, 4, 6, 8]
target = 10

# Panggil fungsi dan tampilkan hasil
result = subset_sum_backtrack(array, target)
print("Semua subset yang jumlahnya", target, ":", result)
```

OUTPUT

```
Semua subset yang jumlahnya 10 : [[2, 2, 2, 2, 2], [2, 2, 2, 4], [2, 2, 6], [2, 4, 4], [2, 8], [4, 6]]
```

Uji Program dengan Input yang Berbeda

1. **Input:** array = [1, 3, 5, 7], target = 8
   - **Hasil:** [[1, 7], [3, 5]]
2. **Input:** array = [2, 3, 5], target = 10
   - **Hasil:** [[5, 5], [2, 3, 5]]
3. **Input:** array = [1, 2], target = 3
   - **Hasil:** [[1, 2]]

Algoritma ini efektif untuk masalah subset kecil. Namun, untuk array besar atau target tinggi, kompleksitasnya meningkat secara eksponensial karena eksplorasi semua kemungkinan kombinasi.

BAGIAN 2 : Analisis Masalah "Permutasi"

1. **Bagaimana Algoritma Backtracking Digunakan untuk Membentuk Permutasi?**

    - Algoritma mencoba setiap elemen di array sebagai langkah pertama dalam permutasi.
    - Dengan menggunakan rekursi, algoritma menambahkan elemen berikutnya dari array yang belum digunakan hingga membentuk urutan lengkap (path dengan panjang sama dengan array asli).
    - Setelah mencapai permutasi lengkap, algoritma kembali (backtrack) untuk mengeksplorasi kombinasi lain dengan menandai elemen yang terakhir digunakan sebagai "tidak digunakan" dan menghapusnya dari jalur (path).

2. **Bagaimana Pengulangan Elemen Dicegah?**

    - Sebuah array used digunakan untuk melacak elemen yang sudah termasuk dalam permutasi saat ini.
    - Sebelum menambahkan elemen ke dalam jalur, algoritma memeriksa apakah elemen tersebut sudah digunakan. Jika ya, elemen tersebut dilewati.

3. **Apakah Algoritma Dapat Dioptimalkan?**

    - **Optimalisasi untuk Input yang Sudah Terurut**
      Jika input mengandung elemen duplikat, algoritma dapat diperluas dengan memeriksa kondisi tambahan untuk menghindari pengulangan. Misalnya, jika elemen saat ini sama dengan elemen sebelumnya dan elemen sebelumnya belum digunakan dalam jalur saat ini, algoritma dapat melewati elemen tersebut.
    - **Mengurangi Rekursi**
      Algoritma saat ini menggunakan array boolean used, yang memerlukan pemeriksaan tambahan di setiap langkah. Alternatifnya, elemen dapat langsung dihapus dan ditambahkan kembali ke array saat backtrack.
    - **Iterasi Efisien**
      Penggunaan struktur data seperti set atau heap dapat membantu jika ada kebutuhan untuk memproses elemen dalam urutan tertentu atau menghindari duplikasi secara langsung.

CODE PROGRAM

```python
def permute(array):
    def backtrack(path, used):
        # 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]

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

OUTPUT

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

UJI COBA

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

Algoritma ini sangat cocok untuk array kecil hingga menengah. Namun, untuk array besar, jumlah permutasi meningkat secara eksponensial, sehingga optimalisasi diperlukan untuk efisiensi waktu dan memori.

BAGIAN 3 : Kesimpulan dan Eksperimen

1. **Bandingkan kinerja algoritma backtracking pada kedua masalah di atas!**
   Menguji dengan panjang array yang berbeda dan jumlah langkah yang diperlukan

   a. **[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

   b. **[Bagian 2]**
   Input: array = [1, 2, 3, 4,5]
      - **Hasil:** [[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]]
      - **Jumlah Langkah :** 120 langkah

2. DISCUSSION

Algoritma backtracking memiliki beberapa kelebihan dalam menyelesaikan masalah kombinatorial, terutama dalam hal kesederhanaan dan fleksibilitas. Kelebihannya terletak pada kemampuannya untuk menjelajahi semua kemungkinan solusi dengan cara yang sistematis, memungkinkan penemuan solusi yang valid tanpa harus menghasilkan semua kombinasi terlebih dahulu. Ini sangat berguna dalam masalah seperti pencarian subset, permutasi, dan masalah penempatan, di mana solusi dapat dibangun secara bertahap.

Namun, keterbatasan utama dari algoritma ini muncul ketika dihadapkan pada input yang lebih besar. Kompleksitas waktu yang tinggi dapat menjadi masalah, karena jumlah kemungkinan kombinasi atau permutasi tumbuh secara eksponensial dengan bertambahnya elemen. Hal ini dapat menyebabkan waktu eksekusi yang sangat lama dan penggunaan memori yang besar, sehingga membuat algoritma ini tidak efisien untuk dataset yang besar.

Untuk meningkatkan efisiensi algoritma backtracking, beberapa modifikasi dapat diterapkan. Salah satunya adalah menggunakan teknik pemangkasan (pruning), di mana cabang-cabang yang tidak mungkin menghasilkan solusi valid dapat diabaikan lebih awal. Misalnya, jika jumlah saat ini melebihi target dalam masalah subset sum, kita dapat menghentikan pencarian lebih lanjut di cabang tersebut. Selain itu, menerapkan heuristik atau strategi pencarian yang lebih cerdas, seperti memilih elemen dengan cara tertentu atau menggunakan struktur data yang lebih efisien, juga dapat membantu mengurangi ruang pencarian dan mempercepat proses. Dengan cara ini, meskipun backtracking tetap memiliki batasan, modifikasi tersebut dapat membuatnya lebih praktis untuk digunakan dalam situasi yang lebih kompleks.