Problem 1. *Phân tích và so sánh ưu điểm và nhược điểm giữa 2 thuật toán và chỉ rõ các ưu / nhược điểm trên trong các trường hợp cụ thể.*

Solution.

1 Phân tích 2 thuật toán:

- Giống nhau: Backtracking và Brute force đều là các phương pháp giải quyết bài toán bằng cách tìm tất cả các cấu hình $C = (x_1, x_2, x_3, ..., x_n)$ $(x_i \in S_i, \forall i \in \{1, 2, 3, ..., n\})$ thỏa mãn các ràng buộc của bài toán.

- Khác nhau:

  – Ở đây ta cần hiểu rõ các khái niệm explicit constraint và implicit constraint. Explicit constraint là miền giá trị của các phần tử trong cấu hình C, tức là các tập hợp S_i chứa các phần tử x_i. Implicit constraint là các ràng buộc giữa hai hay nhiều phần tử trong cấu hình, ví dụ $x_i \neq x_j, \forall (i, j) : i \neq j$ hay $x_i < x_j, \forall (i, j) : i < j$

  – Brute force chỉ quan tâm tới explicit constraint, tức là thay toàn bộ các giá trị của S_i vào x_i với mọi $i \in \{1, 2, 3, ..., n\}$ và chỉ kiểm tra các implicit constraint sau khi đã xây dựng đủ một cấu hình.

  – Backtracking kiểm tra các implicit constraint mỗi khi thêm một phần tử vào cấu hình. Nếu không thỏa mãn (và việc thêm phần tử vào cấu hình cũng không làm thỏa mãn) implicit constraint thì ngay lập tức hoặc thay đổi phần tử thành một giá trị khác hoặc quay lại phần tử trước đó nếu không còn giá trị nào. Đây có thể coi là một cải tiến của phương pháp Brute force.

Ví dụ minh họa sự khác nhau giữa Brute force và backtracking: Xét bài toán n quân hậu, ta cần tìm các cấu hình $C = (x_1, x_2, x_3, ..., x_n)$ với explicit constraint $x_i \in S_i = \{1, 2, 3, ..., n\}, \forall i \in \{1, 2, 3, ..., n\}$ và implicit constraint $x_i \neq x_j \wedge x_i - i \neq x_j - j \wedge x_i + i \neq x_j + j, \forall (i, j) : i \neq j$.

– Thuật toán sử dụng phương pháp Brute force:

```
procedure BRUTE_FORCE
    C ← (1, 1, 1, ...1)
    while C ≠ NULL do
        if IMPLICIT_CONSTRAINT(C) then
            OUTPUT(C)
        end if
        C ← NEXT(C)
    end while
end procedure
```

Trong đó thuật toán tìm $next(C)$:

function NEXT(C)
  $C_n \leftarrow C_n + 1$
  $i \leftarrow n$
  while $C_i > n$ **do**
    if $i = 1$ **then**
      return $NULL$
    end if
    $C_i \leftarrow 1$
    $i \leftarrow i - 1$
    $C_i \leftarrow C_i + 1$
  end while
end function
return C

– Thuật toán sử dụng phương pháp Backtracking:

procedure BACKTRACK(C)
  if not IMPLICIT_CONSTRAINT(C) **then**
    return
  end if
  if LEN(C) $= n$ **then**
    OUTPUT(C)
  end if
  $S \leftarrow C + (1)$
  while $S \neq NULL$ **do**
    BACKTRACK(S)
    if S[LEN(S)] $= n$ **then**
      $S \leftarrow NULL$
    else
      S[LEN(S)] $\leftarrow S$[LEN(S)] $+ 1$
    end if
  end while
end procedure

2 Ưu điểm và nhược điểm:

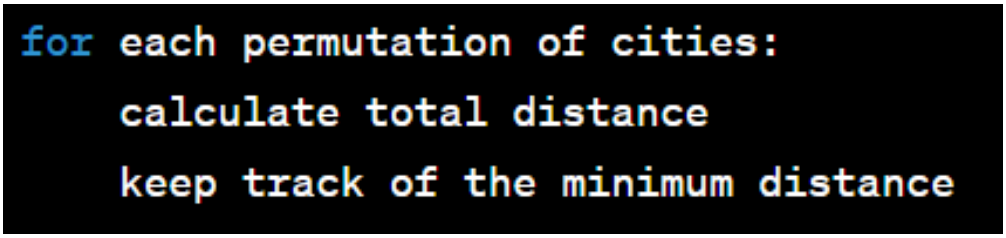
2.1 Brute Force:

- **Ưu điểm:**
  - **Cài đặt:** đơn giản, không phức tạp vì sẽ duyệt qua toàn bộ các trường hợp và chọn ra các trường hợp thỏa yêu cầu bài toán.
  - **Ứng dụng:** Có thể giải quyết hầu hết các bài toán
  - **Dễ tiếp cận** (không yêu cầu kiến thức chuyên sâu về thuật toán)

- **Khuyết điểm:**
  - **Vấn đề hiệu suất:** không phù hợp khi phải giải quyết với các bài toán lớn vì thuật toán sẽ duyệt qua toàn bộ các trường hợp

    VD: tìm kiếm một phần tử trong bộ dữ liệu lớn
  - Brute Force không phải lúc nào cũng cho ra giải pháp tối ưu cho bài toán

    VD: trong bài toán TSP (Traveling Salesman Problem), Brute Force sẽ duyệt qua tất cả các thành phố => một số lượng khổng lồ



2.2 Backtracking:

- **Ưu điểm:**
  - **Độ hiệu quả:** Backtracking sẽ cải thiện độ hiệu quả bằng cách loại bỏ những sự lựa chọn không cần thiết (nhất là các bài toán có không gian lớn)

    Ví dụ:

```python
#hàm check nếu có tồn tại dãy con thỏa target
def is_subset_sum(arr, n, target):
    if target == 0: # nếu target = 0 => trường hợp thỏa
        return True
    if n == 0 and target != 0: #nếu đây là phần tử đầu của mảng mà target vẫn không bằng 0 => trường hợp sai
        return False
    if arr[n - 1] > target: #nếu phần tử hiện tại lớn hơn target, bỏ qua và duyệt tiếp phần tử khác
        return is_subset_sum(arr, n - 1, target)
    #chọn hoặc không chọn phần tử hiện tại
    return is_subset_sum(arr, n - 1, target) or is_subset_sum(arr, n - 1, target - arr[n - 1])

#backtracking sẽ hiệu quả vì thuật toán trong mỗi lần duyệt sẽ loại bỏ các trường hợp không thỏa (arr[n - 1] > target)
set = [3, 34, 4, 12, 5, 2]
target_sum = 9
result = is_subset_sum(set, len(set), target_sum)
print(result) #True (vì có tồn tại [3,4,2])
```

  - **Tính linh hoạt:** Có thể giải quyết các bài toán yêu cầu tối ưu hóa và cover nhiều trường hợp

Ví dụ:

```python
#In ra các trường hợp mảng hoán vị
def generate_permutations(elements, current_permutation):
    if not elements:
        # Base case: hoán vị hoàn tất
        result.append(current_permutation.copy())
        return

    for i in range(len(elements)):
        # Chọn phần tử tiếp theo để thêm vào mảng hoán vị
        current_element = elements[i]

        # Truy vấn các phần tử còn lại
        remaining_elements = elements[:i] + elements[i+1:]
        current_permutation.append(current_element)
        generate_permutations(remaining_elements, current_permutation)

        # Backtracking
        current_permutation.pop()

#Backtracking có tính linh hoạt cao vì có thể thay đổi kích thước input bất kì, miễn sao vẫn thỏa mãn được một số ràng buộc nhất định
elements = [1, 2, 3]
result = []
generate_permutations(elements, [])
print(result)  # [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

- **Khuyết điểm:**
  - Sử dụng đệ quy để xét từng trường hợp:
    - Nếu trường hợp thỏa yêu cầu, duyệt tiếp.
    - Ngược lại, sẽ quay về bước trước đó và duyệt trường hợp khác.

    => **Phức tạp**, cần hàm đệ quy.
  - Cần phải theo dõi quá trình thực hiện backtracking cẩn thận, bao gồm trường hợp hiện tại và các quyết định được thực hiện ở mỗi bước để có thể backtrack khi cần thiết => **Khó duy trì code** (độ phức tạp có thể tăng theo thời gian và không gian).

    Ví dụ: Trong bài toán N-queens, quá trình đệ quy backtracking bao gồm các quá trình như update tình hình chessboard, tìm hiểu các sự lựa chọn và backtrack hủy di các lựa chọn không phù hợp. Vì thế, khi kích thước chessboard tăng => độ phức tạp của các quá trình cũng sẽ tăng.

□