CHƯƠNG II: PHÂN TÍCH-THIẾT KẾ THUẬT GIẢI

1. Phân tích thuật giải

1.1. Thuật giải và chương trình

Khi hai chương trình máy tính giải quyết cùng một vấn đề nhưng theo cách khác nhau, câu hỏi tự nhiên đặt ra là "***chương trình nào tốt hơn***?"

Trước hết, cần nhớ rằng có sự khác biệt quan trọng giữa chương trình và thuật giải mà chương trình đó thể hiện. Thuật giải là một dãy các *lệnh*, trình tự xác định để giải quyết một bài toán hay vấn đề được cho. Thuật giải là giải pháp nhằm giải quyết các thể hiện có thể có của bài toán. Nghĩa là, với đầu vào cụ thể, thuật giải sẽ cho ra kết quả tương ứng. Trong khi đó, chương trình là một thuật giải được mã hóa bằng các *chỉ thị* một ngôn ngữ lập trình. Có thể có nhiều chương trình khác nhau, mã hoá cùng một thuật giải, tùy thuộc vào người lập trình và ngôn ngữ lập trình người đó sử dụng.

thuật giải là cách giải một bài toán được cho và chương trình là mã hoá của thuật giải.

1.1. Ký hiệu O-lớn (Big-Oh)

Để mô tả hiệu quả của thuật giải, mà không phụ thuộc vào chương trình hay máy tính, cần xác định số thao tác mà thuật toán cần để thi hành thuật giải đó. Nếu xem mỗi thao tác là một *đơn vị tính cơ bản*, thì thời gian thực hiện một thuật toán được biểu thị bằng số thao tác cần thiết để giải bài toán được cho. Việc *chọn đúng đơn vị tính cơ bản* phụ thuộc vào cách thực hiện thuật giải. Chẳng hạn, *lệnh gán* là đơn vị tính cơ bản của thuật giải tính tổng n số nguyên tự nhiên đầu tiên, và hiệu quả của thuật giải này có thể xem là tổng số lệnh gán phải được thực hiện. Ta thử "đếm" số lệnh gán này: số lệnh khởi tạo là 1 (Sum = 0) và n lệnh cộng dồn (Sum = Sum + i, i = 1, 2,…, n). Kết quả đếm này có thể biểu thị bằng hàm T: T(n) = 1 + n. Tham số n được gọi là "*kích thước của bài toán*" và "*T(n) là chi phí (về thời gian) cần thiết để giải một bài toán cỡ n*".

Bằng cách biểu diễn chi phí tính toán T(n) như vậy, ta có thể khẳng định chi phí tính tổng của 100000 số lớn hơn chi phí tính tổng của 1000 số: *đầu vào lớn thì chi phí lớn*. Như vậy, ta có thể ký hiệu O-lớn, đọc là Big-Oh như sau.

O-lớn là kí hiệu biểu diễn sự biến thiên của chi phí tính toán theo kích thước đầu vào của thuật toán hay quy mô bài toán.

Cần lưu ý là khi gọi kí hiệu **O**-lớn là ta quan tâm đến dáng điệu của biến thiên hơn là giá trị chi phí cụ thể. Nghĩa là việc xác định chính xác số lượng thao tác không quan trọng bằng việc xác định *phần chi phối nhất* của hàm T(n). Nói cách khác, khi quy mô (đầu vào) lớn, một số thành phần của hàm T(n) có xu hướng "lấn át" các phần còn lại. Thuật ngữ *lấn át* được dùng để so sánh các thuật toán giải cùng bài toán. Ta thấy, trước hết, bằng cách đếm như trên, T(n) luôn có dạng tổng $T(n) = f_1(n) + \cdots + f_k(n)$, và thành phần lấn át nhất sẽ là phần có "*bậc luỹ thừa*" cao nhất. Ký hiệu **Big-O**, viết là **O(g(n))**, dùng để xấp xỉ số thao tác thực sự thuật giải phải làm. Hàm g(n) là biểu diễn giản lược của phần lấn át trong hàm T(n).

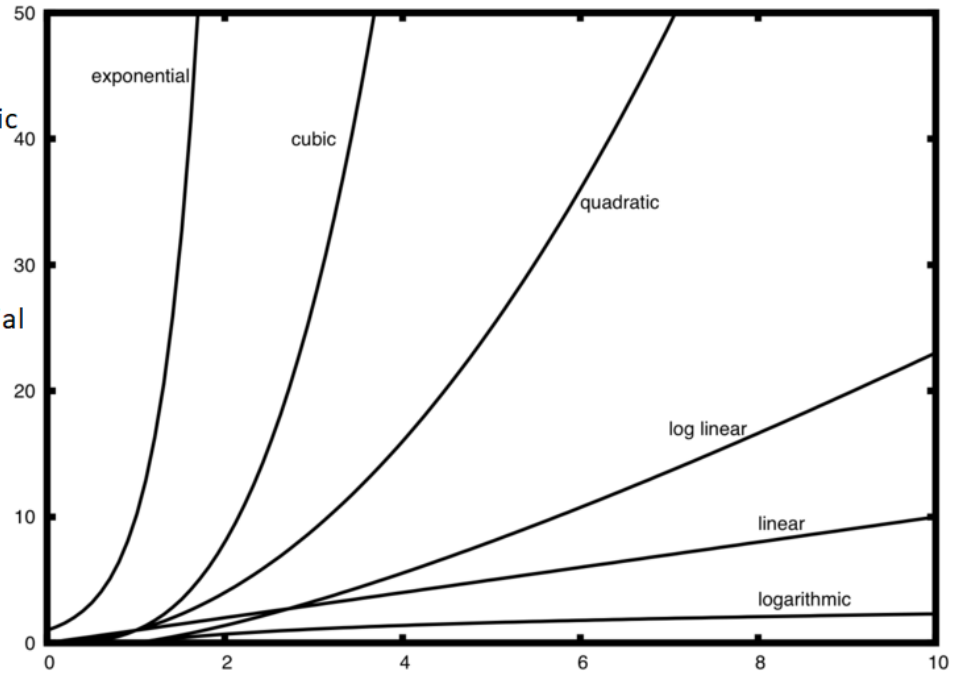
Trong ví dụ độ phức tạp thuật giải tính tổng, T(n) = 1 + n. Khi n lớn, hằng số 1 sẽ ít có ý nghĩa trong kết quả cuối cùng, 1000000 và 1000001 có thể xem như nhau. Vì chỉ cần giá trị gần đúng của T(n), ta có thể bỏ qua giá trị 1, và đơn giản nói rằng *thời gian chạy là O(n)*. Cần lưu ý là giá trị 1 chắc chắn có ý nghĩa đối với T(n). Tuy nhiên, khi n lớn, ước lượng sẽ dễ dàng và chính xác hơn khi bỏ đi số 1.

Ví dụ khác, giả sử rằng với thuật toán **A**, số thao tác chính xác đếm được là $T(n) = 5n^2 + 27n + 1005$. Khi n nhỏ, n = 1 hay n = 2 chẳng hạn, hằng số 1005 là phần lấn át của hàm T(n). Nhưng khi n lớn, n = 10000 chẳng hạn, số hạng n^2 trở nên trội nhất. Trong thực tế, khi n lớn, vai trò của hai số hạng còn lại ảnh hưởng không đáng kể vào kết quả cuối cùng. Để tính *xấp xỉ* T(n) khi n lớn, ta có thể bỏ qua hai số hạng kia và chỉ tập trung vào $\mathbf{5n^2}$. Ngoài ra, hệ số 5 cũng sẽ trở nên không đáng kể khi n lớn. Như vậy, ta nói hàm T(n) có độ phức tạp là $f(n) = n^2$, hay đơn giản là $\mathbf{O(n^2)}$.

Đôi khi hiệu suất của một thuật giải phụ thuộc vào giá trị chính xác của dữ liệu thay vì chỉ đơn giản phụ thuộc vào quy mô hay kích thước bài toán. Với các thuật giải này, ta cần mô tả hiệu suất của chúng theo (***i***) trường hợp *tốt nhất*, (***ii***) trường hợp *xấu nhất* hoặc (***iii***) hiệu suất *trung bình*. Hiệu suất trong trường hợp xấu nhất đề cập đến tập dữ liệu cụ thể mà thuật giải hoạt động kém hiệu quả nhất. Trong khi với tập dữ liệu khác, cùng thuật giải đó lại có hiệu suất cực kỳ tốt. Thường thì trong hầu hết trường hợp, hiệu quả thuật giải nằm đâu đó giữa hai điểm cực biên này (trường hợp trung bình). Hiểu rõ những khác biệt này để không bị nhầm lẫn vào một trường hợp cụ thể.

Hình sau tóm tắt một số hàm biểu diễn độ phức tạp phổ biến và tốc độ tăng trưởng của chúng. Qua đó, khi phân tích, có thể quyết định hàm nào trong số các hàm này là phần lấn át trong hàm đếm T(n), khi n lớn.

Order	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic
$O(n)$	linear
$O(n \log_b n)$	Log linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential



Ta thử phân tích thuật giải được biểu diễn qua chương trình sau (không cần quan tâm chương trình nó làm gì).

```
a = 2
b = 4
c = 7
for i in range(n):
   for j in range(n):
      x = i * i
      y = j * j
      z = i * j
for k in range(n):
   w = a*k + 45
   v = b*b
d = 33
```

Số phép gán là tổng của bốn số hạng. Số hạng đầu tiên là hằng số, **3** (tương ứng với 3 phép gán đầu tiên). Số hạng thứ hai là $\mathbf{3n^2}$ (vì có ba câu lệnh được thực hiện n^2 lần trong 2 vòng lặp lồng nhau). Số hạng thứ ba là **2n** (hai câu lệnh được lặp n lần). Cuối cùng, số hạng thứ tư là hằng số, **1** (phép gán cuối cùng). Vậy

$T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4.$

Dễ dàng thấy rằng số hạng $\mathbf{n^2}$ sẽ chiếm ưu thế khi n lớn, và do đó đoạn mã này có độ phức tạp là $\mathbf{O(n^2)}$. Nhắc lại rằng có thể bỏ qua tất cả các số hạng khác cũng như hệ số của số hạng ưu thế khi n lớn.

1.2. Các quy tắc tính độ phức tạp tính toán

Quy tắc phép toán cơ sở. Độ phức tạp tính toán được xác định dựa trên phép toán cơ sở, là thao tác được thực hiện thường xuyên nhất trong thuật giải.

Quy tắc này phải được áp dụng đầu tiên khi phân tích độ phức tạp một thuật giải.

Quy tắc tổng. Độ phức tạp tổng là độ phức tạp của thành phần trội nhất trong tổng các độ phức tạp thành phần.

Quy tắc tổng được áp dụng cho các khối lệnh tuần tự.

Quy tắc tích. Độ phức tạp tích là tích các độ phức tạp.

Quy tắc tích được áp dụng cho các vòng lặp lồng nhau.

2. Thiết kế thuật giải

Ta sẽ học cách thiết kế thuật giải qua việc xem xét một số bài toán cụ thể, qua đó hình dung và tổng quát hóa cách thiết kế thuật giải để giải bài toán được cho. Qua các ví dụ, ta cũng sẽ học cách xây dựng hàm đếm số thao tác cơ sở của các thuật giải cụ thể để xác định độ phức tạp thời gian tính toán của một thuật giải.

2.1. Bài toán các từ đảo chữ

Trong một từ điển, một từ là 'đảo chữ' của từ khác nếu từ này là sắp xếp lại trật tự các chữ cái của từ kia. Ví dụ, 'DUY AN' và 'UY DAN' là cặp từ đảo ngữ. Để đơn giản, giả sử hai từ có độ dài bằng nhau và được tạo thành từ các ký hiệu trong tập hợp 26 chữ cái viết hoa. Ta cần viết một hàm luận lý (boolean) nhận vào hai chuỗi và trả về TRUE, nếu chúng là đảo ngữ của nhau và FALSE, nếu ngược lại.

Gọi một chuỗi là chuỗi nguồn – **source** và chuỗi còn lại là chuỗi đích – **target**.

2.2.1. Giải pháp 1 – **Solution1**

Suy nghĩ đầu tiên là kiểm tra từng ký tự trong chuỗi đích – target xem ký tự đó có xuất hiện trong chuỗi nguồn – source hay không. Nếu có thì *đánh dấu*. Nếu "đánh dấu" được mọi ký tự, thì hai chuỗi nguồn – source và đích – target là đảo chữ của nhau.

Một ký tự xuất hiện trong chuỗi thứ hai sẽ được thay thế (đánh dấu) bằng giá trị đặc biệt là "*". Tuy nhiên, vì kiểu chuỗi (string) trong Python là bất biến, nên bước đầu tiên trong giải pháp này là chuyển chuỗi target có kiểu string sang target_list có kiểu list. Mỗi ký tự từ chuỗi source

được so sánh với các ký tự trong danh sách target_list, và nếu tìm thấy, đánh dấu "*".

```
def Solution1(source,target):
    stillOK = True # cờ nhớ để xem có cần lặp lại hay thôi
    if len(source) != len(target): # nếu 2 chuỗi không dài như nhau thì không xét nữa
        stillOK = False
    target_list = list(target) #chuyển đổi kiểu: từ string sang list
    pos1 = 0
    while pos1 < len(source) and stillOK: #duyệt tối đa n kí tự của chuỗi s1
        pos2 = 0
        found = False
        while pos2 < len(target_list) and not found: #duyệt hết chuỗi hay đến khi tìm thấy
            if source[pos1] == target_list[pos2]:
                found = True
            else:
                pos2 = pos2 + 1
        if found:
            target_list[pos2] = '*'
        else:
            stillOK = False
        pos1 = pos1 + 1
    return stillOK
#main
print(Solution1('DUY AN,'DAN UY'))
```

Để phân tích thuật giải này, cần lưu ý rằng mỗi ký tự trong n ký tự của source cần một lần lặp để duyệt qua tối đa n ký tự trong danh sách target_list. Mỗi vị trí, trong n vị trí, trong danh sách target_list sẽ được truy cập một lần để so sánh với một ký tự của source. Số lượt truy cập là tổng:

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n.$

Khi n lớn, số hạng n^2 sẽ lấn át số hạng $\frac{1}{2}n$ còn lại, nên số hạng $\frac{1}{2}n$ này và thừa số $\frac{1}{2}$ của số hạng $\frac{1}{2}n^2$ có thể bỏ qua. Cuối cùng, độ phức tạp của giải pháp 1 sẽ là **O(n²).**

2.2.2. Giải pháp 2 – **Solution2**

Nhận xét là dù source và target có khác nhau, nhưng chúng là đảo chữ nếu chúng chỉ gồm các ký tự giống nhau. Vì vậy, nếu bắt đầu bằng cách sắp xếp từng chuỗi theo thứ tự từ điển, từ 'A' đến 'Z', và so sánh 2 chuỗi sau khi sắp xếp để được kết quả. Tuy nhiên, do cả source và target đều có kiểu bất biến string, nên trước hết, cần chuyển sang các danh sách để có thể sắp xếp.

```
def Solution2(source,target):
    list1 = list(source) #chuyển kiểu string sang list
    list2 = list(target)

    list1.sort() #gọi phương thức sort để sắp xếp danh sách
    list2.sort()

    pos = 0
    matches = True
    while pos < len(source) and matches:
        if list1[pos]==list2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(Solution2('SONG AN','ONG SAN'))
```

Thoạt nhìn, độ phức tạp thuật giải này có vẻ là O(n), vì chỉ có một phép lặp **while** để so sánh tối đa n ký tự sau hai thao tác sắp xếp. Tuy nhiên, hai lệnh gọi đến phương thức sắp xếp cũng cần chi phí thực thi. Giả sử chi phí sắp xếp tốt nhất là O(nlogn). Vì thao tác sắp xếp chiếm ưu thế

trong việc lặp lại nên thuật giải này sẽ có cùng độ phức tạp với sắp xếp, **O(nlogn).**

2.2.3. Giải pháp 3 – **Solution3**

Một kỹ thuật đơn giản để giải quyết một mọi vấn đề là *vét cạn* mọi khả năng. Đối với bài toán đảo chữ, có thể tạo ra danh sách tất cả các hoán vị các ký tự của source và kiểm tra xem liệu target có là một trong các hoán vị này hay không.

Khi tạo tất cả các hoán vị có thể có từ source, có n ký tự có thể cho vị trí thứ nhất, n-1 ký tự có thể cho vị trí thứ hai, n-2 cho vị trí thứ ba, v.v. Tổng số chuỗi hoán vị ứng viên có thể có sẽ là

n * (n – 1) * (n – 2) * ... * 3 * 2 * 1 = **n!**.

Mặc dù một số chuỗi có thể trùng lắp, nhưng chương trình máy tính không thể biết trước điều này, và vì vậy nó sẽ vẫn tạo ra n! chuỗi khác nhau.

n! phát triển nhanh hơn 2n. Thực vậy, có thể hình dung là nếu source dài 20 ký tự, sẽ có

20! = 2,432,902,008,176,640,000

chuỗi ứng viên.

Đây có lẽ sẽ không phải là một giải pháp dùng được, nhất là khi n lớn. Ta bỏ qua không cài đặt giải pháp này ở đây[1].

2.2.4. Giải pháp 4 – **Solution4**

Nhận xét là hai chuỗi là đảo chữ sẽ có cùng số chữ a, cùng số chữ b, cùng số chữ c, v.v. Để quyết định hai chuỗi có phải là đảo chữ của nhau hay không, trước tiên ta sẽ đếm số lần xuất hiện của từng ký tự. Vì có 26 ký tự có thể có, ta có thể sử dụng danh sách 26 bộ đếm, mỗi bộ đếm cho một ký tự tương ứng trong bảng chữ cái. Duyệt qua toàn bộ chuỗi, khi gặp một ký tự cụ thể, bộ đếm tại vị trí tương ứng sẽ tăng lên 1. Cuối cùng, nếu hai danh sách các bộ đếm giống hệt nhau, hai chuỗi là đảo chữ của nhau.

```
def Solution4(source,target):

    count1 = [0]*26 #tạo 26 bộ đếm

    count2 = [0]*26
```

[1] Trong phần sau, ta sẽ được học về cách tạo ra tất cả các hoán vị của chuỗi n phần tử.

```
    for i in range(len(source)):
        pos = ord(source[i])-ord('A')
        count1[pos] = count1[pos] + 1

    for i in range(len(target)):
        pos = ord(target[i])-ord('A')
        count2[pos] = count2[pos] + 1

    j = 0
    stillOK = True
    while j<26 and stillOK:
        if count1[j]==count2[j]:
            j = j + 1
        else:
            stillOK = False

    return stillOK
print(Solution4('MINH AN','MAN NHI'))
```

Giải pháp 4 này cũng có một số lần lặp lại. Tuy nhiên, không giống như giải pháp 1, không có các vòng lặp lồng vào nhau. Hai lần lặp đầu tiên được sử dụng để đếm các ký tự đều tính trên kích thước đầu vào n, là chiều dài các chuỗi. Lần lặp thứ ba, so sánh hai danh sách đếm, luôn thực hiện 26 bước vì có 26 ký tự có thể có trong chuỗi. Cộng tất cả số bước lại,

T(n) = 2n + 26

bước. Độ phức tạp sẽ là **O(n).**

Ta đã tìm ra một thuật toán có độ phức tạp tuyến tính để giải quyết vấn đề này.

2.2.5. Nhận xét - **notes**

Ví dụ bài toán đảo chữ này cũng minh hoạ một vấn đề khác: *chi phí không gian*. Mặc dù giải pháp cuối cùng, giải pháp 4, có thể chạy trong thời gian tuyến tính, nhưng nó đã sử dụng thêm bộ nhớ phụ để giữ hai danh

sách số lượng ký tự. Nói cách khác, thuật giải này đã hy sinh không gian để đạt được thời gian.

Đây là trường hợp thường xảy ra trong thiết kế thuật giải. Trong nhiều trường hợp, cần phải đưa ra quyết định giữa thời gian và không gian. Trong trường hợp này, chi phí không gian không đáng kể. Tuy nhiên, nếu bảng chữ cái cơ bản có hàng triệu ký tự, thì cần quan tâm hơn. Tóm lại, khi được chọn thuật giải, sẽ phải quyết định cách sử dụng tốt nhất các tài nguyên máy tính cho một vấn đề đang phải giải.

2.3. Cấu trúc dữ liệu và độ phức tạp

Độ phức tạp thuật giải nhiều khi cũng phụ thuộc vào cấu trúc dữ liệu mà thuật giải sử dụng. Ta sẽ xem xét 2 cấu trúc phổ biến và rất mạnh của Python: danh sách (**list**) và từ điển (**dictionary**).

2.3.1. Danh sách - **list**

Hai thao tác phổ biến với cấu trúc danh sách là *xác định chỉ số* và *gán* giá trị cho phần tử tại vị trí có chỉ số xác định. Cả hai thao tác này đều mất cùng một khoảng thời gian cho dù danh sách có lớn đến đâu. Các thao tác này độc lập với kích thước của danh sách, và là **O(1)**.

Độ phức tạp của thao tác xác định chỉ số của danh sách và độ phức tạp của thao tác gán giá trị cho một phần tử trong danh sách là O(1).

Một nhiệm vụ rất phổ biến khác là *thác triển một danh sách*. Có hai cách để tạo một danh sách dài hơn: phương thức *thêm* (**append**) và toán tử *nối* (**'+'**).

Phương thức thêm có độ phức tạp O(1), còn toán tử nối là O(k) trong đó k là kích thước của danh sách được nối.

Điều này quan trọng và cần biết vì nó có thể giúp tạo các chương trình hiệu quả hơn khi chọn công cụ phù hợp (ở trên là các phương thức).

Ta xem xét bốn cách khác nhau để tạo ra danh sách n số nguyên bắt đầu từ 0. Trước tiên, ta sẽ dùng một vòng lặp *for* và tạo danh sách bằng cách nối (**test1**), sau đó ta sẽ sử dụng *append* thay vì nối (**test2**). Tiếp theo, ta sẽ thử tạo danh sách bằng cách sử dụng tính năng "*hiểu*" (nội dung) danh sách (**test3**) và cuối cùng, có lẽ là cách rõ ràng nhất, sử dụng hàm ***range*** được bọc trong lệnh gọi hàm tạo danh sách – list (**test4**).

```
def test1(): #toán tử nối
    l = []
    for i in range(1000):
        l = l + [i]

def test2():#phương thức nối
    l = []
```

```
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

Để trực quan, dùng hàm Timer trong lớp **timeit** để đo thời gian thực thi của 4 cách tạo danh sách này.

```
import timeit

t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")

concat  6.54352807999 milliseconds
append  0.306292057037 milliseconds
comprehension  0.147661924362 milliseconds
list range  0.0655000209808 milliseconds
```

Cách đo thời gian chạy của một thuật giải cũng là cách thực nghiệm để đánh giá hiệu quả thời gian thực thi của một thuật giải.

2.3.2. Từ điển – **dictionary**

Từ điển khác với danh sách ở chỗ có thể truy cập các mục trong từ điển theo khóa (**key**) thay vì theo vị trí (**index**). Điều quan trọng cần chú ý là các thao tác *get item* và *set item* trên từ điển là **O(1).** Một thao tác từ điển quan trọng khác là thao tác kiểm tra thành viên. Kiểm tra xem một khóa có trong từ điển hay không cũng là **O(1).**

Ta sẽ lặp lại thử nghiệm một từ điển có chứa các số làm khóa. Trong thử nghiệm này, ta sẽ thấy rằng việc xác định xem một số có trong từ điển hay không, không chỉ nhanh hơn nhiều, mà thời gian cần để kiểm tra cũng sẽ không đổi ngay cả khi từ điển ngày càng lớn.

```
import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
```

```
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

3. Kỹ thuật đệ quy và bài toàn tìm kiếm

3.1. Định nghĩa đệ quy

Một khái niệm được định nghĩa thông qua chính nó được gọi là khái niệm có tính đệ quy.

Trong toán học, một hàm định nghĩa qua chính nó được gọi là hàm đệ quy. Chẳng hạn, phần tử thứ n của dãy Fibonacci:

0, 1, 1, 2, 3, 5, ...

có định nghĩa hình thức như sau:

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & n \geq 2 \end{cases}.$$

Trong tin học, một chương trình con gọi lại chính nó gọi là hàm hay thủ tục đệ quy. Chẳng hạn, hàm sau tính phàn tử thứ n của dãy Fibonacci theo công thức thuật toán trên.

```
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n-1)+fibo(n-2) # gọi lại hàm fibo
```

3.2. Kỹ thuật đệ quy

Một định nghĩa đệ quy, *hàm hay thủ tục đệ quy – recursive function/procedure*, gồm (**i**) một mệnh đề cơ sở để định nghĩa trạng thái kết thúc, và (**ii**) một mệnh đề đệ quy, ở đấy hàm hay thủ tục được định nghĩa lại ở quy mô nhỏ hơn. Ví dụ, để định nghĩa hàm tính giai thừa một số nguyên dương n, ta có thể phát biểu:

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1)!, & n > 0 \end{cases}.$$

Định nghĩa trên có thể được cài đặt bằng hàm ***facto***(n) trong Python như sau.

```
def facto(n):
    if n == 0: # (i) mệnh đề cơ sở
        return 1
    else:
        return n*facto(n-1) # (ii) mệnh đề đệ quy
```

Trong định nghĩa hay chương trình tính giai thừa, bằng cách thay 1 bằng 0 trong *mệnh đề cơ sở* và thay phép "*" bằng "+" trong *mệnh đề đệ quy*, ta có định nghĩa chương trình tính tổng n số tự nhiên đầu tiên. Tuy nhiên, vì dãy n số tự nhiên đầu tiên là cấp số cộng, công sai bằng 1, có thể tính tổng này với độ phức tạp nhỏ nhất O(1). Nên thay vì tính tổng n số tự nhiên đầu tiên bằng kỹ thuật đệ quy, ta sẽ tính tổng của n phần tử trong danh sách **A**.

```
def sum(n,a):
    if n == 0: #(i)
        return 0
    else:
        return a[n-1]+sum(n-1,a) #(ii)

A = [1,3,7,2,5]
print(sum(len(A),A))
```

Từ các ví dụ trên, ta thấy,

kỹ thuật đệ quy, cài đặt hàm hay thủ tục, là **kỹ thuật mô tả** mệnh đề cơ sở (i) và mệnh đề đệ quy (ii).

Lưu ý là luôn phải có mệnh đề cơ sở để đảm bảo tiến trình đệ quy có thể kết thúc.

Kỹ thuật mô tả này sẽ được sử dụng để thiết kế thuật giải. Rõ ràng, lệnh lặp có thể chuyển qua đệ quy bằng cách (**1**) đặt điều kiện kết thúc lặp là mệnh đề cơ sở (**i**), và (**2**) biểu diễn thân vòng lặp dạng gọi lại chính nó (**ii**). Chẳng hạn, để tính ước chung lớn nhất của 2 số nguyên dương, ta có thể "*mô tả*" gcd(a, b) tính ước chung lớn nhất của a và b như sau:

$$gcd(a,b) = \begin{cases} a, & a = b \\ gcd(a-b,b), & a > b \\ gcd(a,b-a), & a < b \end{cases}.$$

Và cài đặt bằng Python.

```
def gcd(a,b):
    if a==b:
        return a
    elif a>b:
        return gcd(a-b, b)
    else:
        return gcd(a,b-a)
```

Lưu ý rằng,

một thuật giải không lặp luôn có thể được mô tả chỉ dùng mệnh đề cơ sở bằng cách liệt kê các khả năng, và không có mệnh đề đệ quy.

Chẳng hạn chương trình sau "mô tả" cách giải phương trình bậc nhất equal1(a, b) ≡ ax + b = 0.

```
def equal1(a,b):
```

```
    if a==0:
        if b==0:
            print("many solution")
        else:
            print("no solution")
    else:
        print("solution x = ", -b/a)
```

3.3. Bài toán tìm kiếm

Giải quyết vấn đề bằng máy tính có thể được xem như bài toán tìm kiếm, ở đó, ta cần tìm một (số) giải pháp hay lời giải của bài toán được cho.

Bài toán tìm kiếm có thể được xếp vào một trong hai dạng: (**1**) tìm kiếm lời giải bằng cách liệt kê các khả năng và chọn lời giải thích hợp, gọi tắt là **liệt kê**; hay (**2**) tìm ra cách giải, gọi là **tìm kiếm hướng đích**.

Xét các ví dụ sau.

Ví dụ 1 (**bài toán tổng con**). Cho danh sách **a** có n số nguyên dương phân biệt và một số nguyên dương **t**. In ra các số trong **a** sao cho tổng chúng bằng **t**. Chẳng hạn, với

a = [2, 5, 3, 9] và **t** = 14,

có 2 kết quả

2+3+9 = 14 và 5+9 = 14.

Có thể có nhiều cách để giải bài toán tổng con này. Một cách có thể nghĩ đến ngay là liệt kê từng danh sách con có thể có của danh sách **a** và kiểm tra xem tổng của chúng có bằng **t** hay không. Theo cách nghĩ này, có thể dùng đệ quy để mô tả cách liệt kê và kiểm tra xem dãy con vừa liệt kê đó có là lời giải hay không.

Ví dụ 2 (**bài toán n quân hậu**). Là bài toán điển hình minh hoạ cho các bài giảng về kỹ thuật quay lui, cùng với bài toán điển hình quay lui khác là bài toán mã đi tuần mà bạn có thể tự tìm hiểu. Trở lại với ví dụ bài toán n con hậu, nhiệm vụ phải làm là đặt n con hậu trên bàn cờ vua nxn ô sao cho chúng không tấn công được nhau. Nếu ta xem con hậu thứ i sẽ được đặt đúng vào ô j của hàng i trên bàn cờ, thì bằng cách "liệt kê" tất cả lời giải có thể có của một véc-tơ n thành phần, mỗi thành phần nhận giá trị là 1 trong n vị trí có thể đặt và kiểm tra xem véc-tơ nào là lời giải.

Ví dụ 3 (**Tháp Hà Nội**). Tháp Hà Nội là bài toán quen thuộc và thường được dùng để minh hoạ kỹ thuật đệ quy. Rõ ràng, giải bài toán Tháp Hà Nội chính là tìm cách chuyển n đĩa từ cọc nguồn sang cọc đích, mỗi lần chuyển 1 đĩa, và có thể dùng một cọc khác làm cọc trung gian. Bằng cách biểu

diễn trạng thái đầu là cọc nguồn chứa n đĩa, 2 cọc trung gian và đích trống; và trạng thái đích là cọc đích chứa n đĩa, còn 2 cọc nguồn và trung gian trống, thì để tìm lời giải là cách chuyển từ trạng thái nguồn về trạng thái đích mang ý nghĩa tìm kiếm hướng đích, và ta có thể mô tả như sau: để chuyển n đĩa từ cọc nguồn sang cọc đích, ta (**1**) chuyển n-1 đĩa từ cọc nguồn sang cọc trung gian; sau đó (**2**) chuyển đĩa còn lại ở cọc nguồn sang cọc đích; cuối cùng (**3**) chuyển n-1 đĩa từ cọc trung gian sang cọc đích.

Ví dụ 4 (**bài toán mê cung**). Một mê cung được mô phỏng bằng một bàn cờ n hàng, m cột. Trên bàn cờ có một số chướng ngại vật được mô phỏng bằng giá trị 0 tại ô đó, các ô không có chướng ngại là 1. Một con rô-bốt được đặt ở ô trên cùng bên trái, ô (0,0), và phải di chuyển đến ô tận cùng bên phải, ô (n-1, m-1), tại mỗi ô, rô-bốt chỉ di chuyển được 1 ô qua 4 hướng đến ô không có chướng ngại vật.

Kỹ thuật mô tả dựa trên đệ quy sẽ được sử dụng để thiết kế thuật giải giải 2 dạng bài toán tìm kiếm này.

3.4. Tìm kiếm liệt kê

3.4.1. Quy trình và thuật giải liệt kê - **enumerate**

Trước hết, ta hình thức hoá bài toán tìm kiếm dưới dạng hàm: $p: D \to \{False, True\}$, với D là miền hữu hạn. Nhiệm vụ của ta là tìm $x \in D$ sao cho $p(x) = True$, hay đơn giản dạng mệnh đề $p(x)$. D được gọi là không gian lời giải, hay không gian trạng thái. Trong trường hợp miền D vô hạn, có thể được giải bằng các kỹ thuật trí tuệ nhân tạo, máy học, mà sẽ được trình bày trong các bài học khác.

Quy trình chung để giải bài toán tìm kiếm liệt kê gồm:

Bước 1: biểu diễn không gian trạng thái $D = D_1 \times \ldots \times D_n$.

Bước 2: biểu diễn điều kiện p xác định một trạng thái $x = (x_1, \ldots, x_n) \in D = D_1 \times \ldots \times D_n$ có phải là lời giải $p(x_1, \ldots, x_n)$ hay không, **p(x)**.

Bước 3: thực thi thuật giải liệt kê enumerate.

Thuật giải liệt kê enumerate(.)

```
DEF enumerate(i) #S[0..i] là chuỗi i thành phần hợp lệ
   IF i == n: # liệt kê đủ các thành phần của lời giải
      p(S) # kiểm tra xem có là lời giải
   ELSE
```

```
    FOR j in Di:
        S[i] = j
        enumerate(i+1) # gọi hàm liệt kê cho chuỗi i+1 thành phần
#END
```

3.4.2. Liệt kê các chuỗi nhị phân – **listed_Binary**

Ta sẽ áp dụng quy trình và thuật giải liệt kê để liệt kê tất cả các chuỗi nhị phân n bit. Theo đó,

$D = \{0,1\}^n$ và $p(x_1, \dots, x_n) = True, x_i \in \{0,1\}, i = 1,2, \dots, n$.

Vì thế, $p(x)$ chỉ đơn giản là in x ra. Chương trình như minh hoạ dưới.

```
def p(S):
    print(S)
######################
def listed_Binary(i):
    if i == n:
        p(S)
    else:
        for j in range(0,2):#Di = {0,1} có thể được mở rộng Di = {0,1,…,k} cho chuỗi k-phân
            S[i] = j
            listed_Binary(i+1)
#main
n = 3
S = n*[0]
listed_Binary(0)
```

listed_Binary có thể xem như một lược đồ tổng quát. Thực vậy, nếu Di = {0, 1, 2, 3}, bằng cách thay range(0,3) cho range(0,1), ta có chương trình liệt kê các chuỗi tứ phân n kí số. Hơn nữa listed_Binary có thể được sử dụng làm cơ sở giải các bài toán dạng tìm kiếm liệt kê như sẽ minh hoạ sau.

3.4.3. Bài toán tổng con – **subset_sum**

Trở lại bài toán tổng con, chọn ra tập con các phần tử của danh sách **a** có n phần tử sao cho tổng chúng bằng **t**.

Bằng cách đặt

$p_S(a) = p(a_1, \dots, a_n) = s_1a_1 + \cdots + s_na_n$, với

$S = (s_1, \dots, s_n) \in \{0,1\}^2$

là véc-tơ nhị phân n thành phần.

Bài toán tổng con được đưa về tìm véc-tơ nhị phân $S = (s_1, \dots, s_n)$ sao cho $p_S(a) = t$, và được giải bằng cách liệt kê để sinh các dãy nhị phân S và kiểm tra mệnh đề $p_S(a) = t$.

```
#subset sum problem
a = [2, 5, 3, 9]
t = 14
def p(S):
    s = 0
    for i in range(len(S)):
        s = s + S[i]*a[i]
    if s == t: # p(S,a) = t
        print(S)
######################
def listed_Binary(i):
    if i == n:
        p(S)
    else:
        for j in range(0,2):#Di = {0,1} có thể được mở rộng Di = {0,1,…,k} cho chuỗi k-phân
            S[i] = j
            listed_Binary(i+1)
#main
n = len(a)
S = n*[0]
listed_Binary(0)
```

Tương tự bài toán n con hậu có thể được giải bằng cách (**1**) sinh ra dãy n-phân có n phần tử như ghi chú cho lược đồ listed_Binary; và (**2**) thiết kế hàm đảm bảo các con hậu đứng đúng vị trí không tấn công nhau.

Bài tập. Sửa chương trình tổng con trên để giải bài toán n con hậu.

Lược đồ liệt kê chuỗi nhị phân cơ sở, như ta thấy qua các ứng dụng trên là rất tổng quát, với độ phức tạp luôn trên $O(2^n)$. Vì thế, trong các bài toán cụ thể, lược đồ này chỉ nên là một cách trong khi chờ tìm được một thuật giải khác tốt hơn.

3.4.4. Liệt kê các hoán vị - **listed_permutation**

Hoán vị cũng là kỹ thuật thường được sử dụng để liệt kê tập S có n thành phần, trong đó các thành phần khác nhau. Rõ ràng có thể sử dụng **enumerate** để liệt kê tất cả các dãy số n-phân và chỉ in ra các dãy $x = (x_1, \dots x_n)$ có tất cả các khác nhau, theo công thức đệ quy dưới, $x_i \neq x_j, \forall 1 \leq i \neq j \leq n$.

$$p(x_1, \dots, x_n) = \begin{cases} True, & x_i \neq x_j, \forall 0 \leq i, j < n \text{ and } i \neq j \\ False, & \text{ngược lại} \end{cases}.$$

Có thể làm tốt hơn cách liệt kê như trên bằng cách làm trực tiếp trên các phần tử của tập hợp được cho. Thuật giải hoán vị được mô tả đệ quy như sau.

- Hoán vị của tập rỗng hay tập chỉ có 1 phần tử là chính nó.

• Với tập có n > 1 phần tử, với mọi phần tử m được trích ra từ tập S, nếu danh sách L là một hoán vị của n-1 phần tử còn lại, thì ghép m vào L, [m] + L, được danh sách mới là 1 hoán vị của tập S có n phần tử.

Chương trình sau minh hoạ cài đặt thuật giải **listed_permutation**.

Bài tập. Để so sánh với phương pháp liệt kê n-phân dựa trên **emenurate**, bạn thử (**a**) về mặt thực nghiệm, viết và đo thời gian thực thi của 2 phương pháp; (**b**) về mặt lý thuyết, tính và so sánh O-lớn của 2 cách tiếp cận này; và (**c**) cho nhận xét.

```
def listed_permutation(S):
    if len(S) == 0: #hoán vị của chuỗi rỗng là chuỗi rỗng
        return []
    if len(S) == 1: #hoán vị của chuỗi có 1 phần tử là chính nó
        return [S]
    L = [] #L sẽ lưu hoán vị hiện tại, khởi tạo là rỗng

    # tính các hoán vị của đầu vào S
    for i in range(len(S)):
       m = S[i]

       # trích ra L[i] (hay m) từ S. L1 là danh sách còn lại
       L1 = S[:i] + S[i+1:]

       # đệ quy tạo tất cả hoán vị với m là phần tử đầu tiên
       for p in listed_permutation(L1):
           L.append([m] + p)
    return L


#main
data = ['red','green','blue']
for p in listed_permutation(data):
    print (p)
```

Bài tập. Áp dụng hàm hoán vị để viết chương trình liệt kê tất cả các tập con k phần tử của tập có n phần tử.

3.5. Tìm kiếm hướng đích

3.5.1. Quy trình và thuật giải hướng đích - **enumerate**

Nếu như tìm kiếm liệt kê là tìm nghiệm trong miền xác định, thì tìm kiếm hướng đích lại là *tìm cách giải khi được cho giả thiết và kết luận*, như các bài toán chứng minh hình học hay được làm ở bậc phổ thông. Với cách so sánh vậy, có thể xem giả thiết là trạng thái đầu và kết luận là trạng thái kết thúc hay trạng thái đích. Và lời giải là dãy các trạng thái biến đổi từ trạng thái nguồn sang trạng thái đích mà ta cần tìm. Quy trình tìm kiếm hướng đích gồm các bước cơ bản sau:

Bước 1: biểu diễn được không gian trạng thái và mã hoá trạng thái nguồn S và trạng thái đích S* theo biểu diễn đó. Gọi trạng thái nguồn là trạng thái hiện hành (đầu tiên).

Bước 2: từ trạng thái hiện hành S, (**i**) xác định tập tất cả các trạng thái kế có thể chuyển tới, và (**ii**) với mọi trạng thái kế S' trong tập trạng thái kế, xem trạng thái S' là trạng thái nguồn mới, đánh dấu nó; và (**iii**) nếu S'S* là cách chuyển trạng thái từ S' đến S*, thì SS'S* là một lời giải cần tìm.

3.5.2. Bài toán mê cung – **maze**

Từ lược đồ tìm kiếm hướng đích trên, ta sẽ thiết kế thuật giải mê cung tổng quát, theo đó, ô xuất phát có thể là bất kỳ ô nào bên trái, và ô đích có thể là bất kỳ ô nào bên phải mê cung. Trong bài toán mê cung này, nếu (x, y) là trạng thái hiện hành, tức là trạng thái rô-bốt đang đứng, thì tập các trạng thái kế rô-bốt có thể di chuyển là {(x+1, y), (x, y+1), (x-1, y), (x, y-1)}.

```
#Maze
def Next(x, y):
    if x >= 0 and x < n and y >= 0 and y < n and A[x][y] == 1:
        return True
    return False

def Target(x,y):
    if x == xe and y == ye and A[x][y] == 1:
        M[x][y] = 1
        return True

def S(x, y): #hàm chuyển trạng thái
    if Target(x,y):#nếu (x, y) là trạng thái đích
        return True #thì tìm được 1 giải pháp
    # nếu không thì kiểm tra xem (x,y) có phải là trạng thái kế
    if Next(x, y):
        if M[x][y] == 1: #nếu trạng thái kế (x, y) đã được đánh dấu
            return False #thì không xét lại

        M[x][y] = 1 #đánh dấu ô (x, y) đã đi qua
        #các trạng thái có thể đi tiếp {(x+1, y), (x, y+1), (x-1, y), (x, y-1)}
        for xx in [-1,0,1]: #khi lập trình quen, có thể thay [-1,0,1] bằng range(-1,2)
            for yy in [-1,0,1]: #chẳng hạn for yy in range(-1,2)
                if S(x + xx,y + yy):
                    return True

        M[x][y] = 0 #bỏ đánh dấu để quay lui, nếu có thể
        return False

#main
A = [#maze
    [0, 0, 0, 0],
    [1, 1, 0, 1],
    [0, 1, 1, 1],
    [1, 1, 0, 1]
    ]
n = len(A)
```

```
M = [[0 for j in range(n)] for i in range(n)] #khởi tạo bảng đánh dấu các trạng thái đã qua
xs, ys = 1, 0 # trạng thái nguồn
xe, ye = 1, 3 # trạng thái đích
if S(xs, ys) == False:
    print("Khong co loi giai");
else:
    print(M) #có thể dựa vào M để xử lý kết quả
```

Các bài toán mê cung, hay hướng đích, có thể đưa về bài toán đồ thị G = (V, E), ở đó, mỗi trạng thái được xem như 1 đỉnh của đồ thị, và nếu S' ∈ V là trạng thái kế của trạng thái S ∈ V, có một cung (S, S') ∈ E, hay còn gọi 1 cạnh, nối giữa 2 đỉnh biểu diễn 2 trạng thái này. Lý thuyết đồ thị sẽ được xem xét ở một bài học khác, ở đó, sẽ được học về các dạng đồ thị và các bài toán phổ biến thường gặp khi làm việc trên đồ thị.

4. Đệ quy và lặp

4.1. Đệ quy và khử đệ quy

Như ta đã thấy ở trên, đệ quy là kỹ thuật mạnh có thể được dùng để thiết kế thuật toán giải bài toán được cho một cách tự nhiên như mô tả. Tuy nhiên, các chương trình đệ quy thường kém hiệu quả vì trình biên dịch của ngôn ngữ luôn phải sử dụng các vùng nhớ tạm để lưu trữ các trạng thái trung gian của quá trình đệ quy.

Ta cũng biết rằng, đệ quy là quá trình thực hiện một "bất biến" nào đó cho đến khi mệnh đề cơ sở được thi hành. Như vậy, hầu hết các hàm đệ quy đề có thể diễn tả dưới dạng lặp. Chẳng hạn, không khó để cài đặt hàm tính giai thừa bằng kỹ thuật lặp. Một ví dụ khác, ta sẽ cài đặt hàm tính phần tử thứ n của dãy Fibonacci có định nghĩa đệ quy bằng kỹ thuật lặp như sau.

$$f(n) = \begin{cases} 0, & n = 0\ (1) \\ 1, & n = 1\ (2) \\ f(n-1) + f(n-2), & x \geq 0\ (3) \end{cases}$$

```
def fibo_iteration(n):
    if n == 0:# (1)
        return 0
    if n == 1:# (2)
        return 1
    f1, f2 = 0, 1 # bắt đầu cho (3)
    for i in range(2,n+1):
        f3 = f1 + f2 # (3)
        f1 = f2
        f2 = f3
    return f2 # kết thúc (3)
#main
for i in range(7):
    print(fibo_iteration(i))
```

Như ta đã thấy, đệ quy là phương tiện rất đẹp và tự nhiên cho thiết kế thuật giải. Tuy nhiên, đệ quy sử dụng nhiều bộ nhớ để có thể hiện thực các chương trình tự nhiên như đệ quy. Hơn nữa, nhiều ngôn ngữ không hỗ

trợ đệ quy. Và các chương trình dịch đều phải có nhiệm vụ khử đệ quy. Vì thế, trong nhiều trường hợp, lập trình viên nên ***khử đệ quy***. Trên nguyên tắc, hầu hết các chương trình (con) đệ quy đều có thể khử để chuyển sang chương trình không đệ quy tương ứng. Tuy nhiên, trong thực tế, khử đệ quy là công việc phức tạp.

4.2. Khử đệ quy đuôi

Một hàm được gọi là ***đệ quy trực tiếp*** nếu trong thân hàm có lời gọi lại chính hàm đó.

Chẳng hạn, lời gọi $if\ (n > 1): n! = n \times (n-1)!$, hay lời gọi $if\ (n > 2): fibo(n) = fibo(n-1) + fibo(n-2)$ là các minh hoạ về đệ quy trực tiếp. Các hàm đệ quy trực tiếp có cấu trúc

IF <điều_kiện_đệ_quy> ... Recursive_function ...,

vì thế, dạng này còn được gọi là *đệ quy đuôi*. Các hàm đệ quy đuôi có thể được chuyển sang cấu trúc lặp bằng cách sử dụng thêm một số biến bộ nhớ giữ giá trị (trạng thái) trung gian, như đã thấy trong chương trình lặp tính giá trị Fibonacci trên. Với giai thừa, hàm có vẻ đơn giản hơn vì chỉ cần 1 biến facto để giữ giá trị trung gian i!,0 ≤ i < n:

```
def factorial(n):
    facto = 1
    for i in range(1,n+1):
        facto = i*facto
    return facto
```

Bài tập. Đọc hiểu và thực hiện hàm cài đặt công thức đệ quy sau và cho biết hàm làm việc gì.

```
def binary1(n):
    if n!=0:
        binary_recursive(n//2)
        print(n%2,end='')
```

```
def binary2(n):
    while n!=0:
        n = n//2
        print(n%2, end = '')
```

4.2. Khử đệ quy bằng cấu trúc ngăn xếp

Đây là một kỹ thuật mạnh để khử đệ quy. Trong kỹ thuật này, ngoài các biến trung gian, cần sử dụng thêm cấu trúc ngăn xếp (stack) để điều khiển quá trình thực hiện chương trình. Chẳng hạn, hàm binary2 trong bài tập trên là dạng khử đệ quy của phiên bản binary1. Để thấy rõ có thể sửa để phiên bản binary2 in kết quả là ngăn xếp stack.

```
def binary_stack(n):
    stack = []
    while n!=0:
        stack = [n%2] + stack
```

```
        n = n//2
    print(stack)
```

Thực ra, cách dùng ngăn xếp để khử đệ quy không giảm được độ phức tạp lưu trữ, vì ta vẫn phải sử dụng bộ nhớ cho ngăn xếp như cách các trình dịch thực hiện, nhưng chắc chắn chương trình của ta không tối ưu bằng cách trình dịch, nên cách dùng stack khử đệ quy chỉ nên dùng khi ngôn ngữ lập trình không hỗ trợ đệ quy. May mắn thay, Python là ngôn ngữ hỗ trợ đệ quy.

Bài tập. Trước khi kết thúc bài học, hãy kiểm chứng và cài đặt thuật giải lặp giải bài toán Tháp Hà Nội nhu mô ta sau:

```
Tháp Hà Nội
Input:
- n (số đĩa)
- n đĩa đặt ở cọc 'A', cọc 'B' và cọc 'C' trống
Output:
- n đĩa đặt ở cọc 'C', cọc 'B' và cọc 'A' trống
Thuật giải
- Số bước lặp,
- Ở bước lặp lẻ, chuyển đĩa #1 theo quy_luật_chuyển
- Ở bước lặp chẵn, trong 2 cọc không chứa đĩa #1, cọc nào chứa đĩa nhỏ thì chuyển sang cọc kia.

quy_luật_chuyển áp dụng cho bước lẻ để chuyển đĩa #1:
- nếu n chẵn, chuyển đĩa #1 theo chiều kim đồng hồ "A-B-C"
- nếu n lẻ, chuyển đĩa @1 ngược chiều kim đồng hồ "A-C-B"
```

Tóm tắt

Giải quyết vấn đề bằng máy tính là xây dựng giải pháp, được gọi là thuật toán hay thuật giải, để có thể sử dụng máy tính giải bài toán được cho. Có thể có nhiều thuật toán để giải cùng một bài toán được cho; và ngay cả có thuật toán thì cũng có thể có nhiều chương trình khác nhau cài đặt nó. Có hai vấn đề người làm khoa học máy tính quan tâm khi giải quyết vấn đề bằng máy tính: thiết kế thuật giải, và phân tích tính hiệu quả của một thuật giải.

. Hiệu quả của thuật giải được đánh giá theo tài nguyên, thời gian và bộ nhớ, mà thuật giải cần để có thể giải được bài toán được cho.

. Ký hiệu O-lớn là ký hiệu được dùng để ước lượng tài nguyên thuật giải có thể sử dụng để giải bài toán theo hàm biểu diễn kích thước đầu vào.

. Các nguyên tắc tư duy chung để thiết kế thuật giải là (i) từ tổng quát đến cụ thể - topdown, (ii) chia bài toán phức tạp thành các bài toán đơn

giản hơn để giải – devide and conquer, (iii) cân đối giữa tài nguyên thời gian và tài nguyên bộ nhớ - dynamic programming.

. Đệ quy là kỹ thuật có thể được sử dụng để thiết kế thuật giải bằng cách mô tả bài toán dưới dạng đệ quy: định nghĩa một khái niệm qua chính nó.

. Giải quyết vấn đề bằng máy tính có thể xem như bài toán tìm kiếm giải pháp trong không gian giải pháp. Theo đó, thiết kế thuật toán gồm (1) biểu diễn miền xác định lời giải, và (2) tùy thuộc tính chất của miền xác định mà xây dựng các thuật giải tìm kiếm thích hợp.

. Khi không gian lời giải là hữu hạn, các kỹ thuật (a) tạo-và-thử (generate and test hay test and error) có thể được dùng để liệt-kê và kiểm-tra các lời giải khả thi, (b) quay lui (back tracking) để nhắm đích trong quá trình tìm kiếm xây dựng giải pháp.

. Khi không gian lời giải là vô hạn đếm được, các thuật giải heuristics – A^* sẽ được học để giải bài toán tìm kiếm lời giải; con nếu không gian là liên túc, các kỹ thuật xấp xỉ sẽ được học để tìm kiếm lời giải gần đúng cho bài toán.