Cấu trúc dữ liệu và giải thuật (Data Structure and Algorithms)

Giải thuật là gì ?

Giải thuật (hay còn gọi là thuật toán - tiếng Anh là **Algorithms**) là một tập hợp hữu hạn các chỉ thị để được thực thi theo một thứ tự nào đó để thu được kết quả mong muốn. Nói chung thì giải thuật là độc lập với các ngôn ngữ lập trình, tức là một giải thuật có thể được triển khai trong nhiều ngôn ngữ lập trình khác nhau.

Xuất phát từ quan điểm của cấu trúc dữ liệu, dưới đây là một số giải thuật quan trọng:

- **Giải thuật Tìm kiếm**: Giải thuật để tìm kiếm một phần tử trong một cấu trúc dữ liệu.
- **Giải thuật Sắp xếp**: Giải thuật để sắp xếp các phần tử theo thứ tự nào đó.
- **Giải thuật Chèn**: Giải thuật để chèn phần từ vào trong một cấu trúc dữ liệu.
- **Giải thuật Cập nhật**: Giải thuật để cập nhật (hay update) một phần tử đã tồn tại trong một cấu trúc dữ liệu.
- **Giải thuật Xóa**: Giải thuật để xóa một phần tử đang tồn tại từ một cấu trúc dữ liệu.

Đặc điểm của giải thuật

Không phải tất cả các thủ tục có thể được gọi là một giải thuật. Một giải thuật nên có các đặc điểm sau:

- **Tính xác định**: Giải thuật nên rõ ràng và không mơ hồ. Mỗi một giai đoạn (hay mỗi bước) nên rõ ràng và chỉ mang một mục đích nhất định.
- **Dữ liệu đầu vào xác định**: Một giải thuật nên có 0 hoặc nhiều hơn dữ liệu đầu vào đã xác định.
- **Kết quả đầu ra**: Một giải thuật nên có một hoặc nhiều dữ liệu đầu ra đã xác định, và nên kết nối với kiểu kết quả bạn mong muốn.
- **Tính dừng**: Các giải thuật phải kết thúc sau một số hữu hạn các bước.

- **Tính hiệu quả**: Một giải thuật nên là có thể thi hành được với các nguồn có sẵn, tức là có khả năng giải quyết hiệu quả vấn đề trong điều kiện thời gian và tài nguyên cho phép.
- **Tính phổ biến**: Một giải thuật có tính phổ biến nếu giải thuật này có thể giải quyết được một lớp các vấn đề tương tự.
- **Độc lập**: Một giải thuật nên có các chỉ thị độc lập với bất kỳ phần code lập trình nào.

Cách viết một giải thuật ?

Bạn đừng tìm, bởi vì sẽ không có bất kỳ tiêu chuẩn nào cho trước để viết các giải thuật. Như bạn đã biết, các ngôn ngữ lập trình đều có các vòng lặp (**do, for, while**) và các lệnh điều khiển luồng (**if-else**), … Bạn có thể sử dụng những lệnh này để viết một giải thuật.

Chúng ta viết các giải thuật theo cách thức là theo từng bước một. Viết giải thuật là một tiến trình và được thực thi sau khi bạn đã định vị rõ ràng vấn đề cần giải quyết. Từ việc định vị vấn đề, chúng ta sẽ thiết kế ra giải pháp để giải quyết vấn đề đó và sau đó là viết giải thuật.

Ví dụ viết giải thuật

Bạn theo dõi ví dụ minh họa dưới đây để thấy rõ các bước và cách viết một giải thuật. Tất nhiên là ví dụ dưới đây là khá đơn giản vì đây chỉ là ví dụ minh họa mở đầu cho cách viết giải thuật thôi, nên mình nghĩ càng đơn giản sẽ càng tốt.

Bài toán: Thiết kế một giải thuật để cộng hai số và hiển thị kết quả.

```
Bước 1: Bắt đầu
Bước 2: Khai báo ba số a, b & c
Bước 3: Định nghĩa các giá trị của a & b
Bước 4: Cộng các giá trị của a & b
Bước 5: Lưu trữ kết quả của Bước 4 vào biến c
Bước 6: In biến c
Bước 7: Kết thúc
```

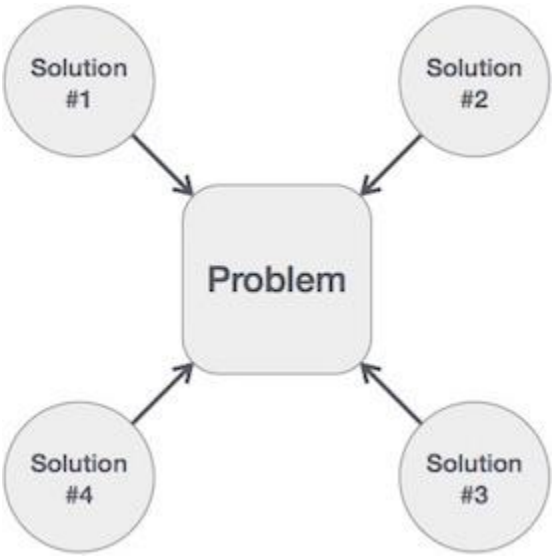
Các giải thuật nói cho lập trình viên cách để viết code. Ngoài ra, bạn cũng có thể viết một giải thuật cho bài toán trên như sau:

```
Bước 1: Bắt đầu
Bước 2: Lấy giá trị của a & b
Bước 3: c ← a + b
Bước 4: Hiển thị c
Bước 5: Kết thúc
```

Trong khi thiết kế và phân tích các giải thuật, thường thì phương thức thứ hai được sử dụng để miêu tả một giải thuật. Cách thứ hai này giúp dễ dàng phân tích giải thuật khi đã bỏ qua các phần định nghĩa không cần thiết. Nhìn vào cách thứ hai này, người ta có thể biết các phép tính nào đang được sử dụng và cách tiến trình thực thi.

Tất nhiên, việc viết **tên** các bước là tùy ý.

Chúng ta viết một giải thuật để tìm giải pháp để xử lý một bài toán nào đó. Một bài toán có thể được giải theo nhiều cách khác nhau.



Do đó, một bài toán có thể sẽ có nhiều lời giải. Vậy lời giải nào sẽ là thích hợp nhất cho bài toán đó. Mời bạn tiếp tục theo dõi.

Phân tích giải thuật

Hiệu quả của một giải thuật có thể được phân tích dựa trên 2 góc độ: trước khi triển khai và sau khi triển khai:

- **Phân tích lý thuyết**: Có thể coi đây là phân tích chỉ dựa trên lý thuyết. Hiệu quả của giải thuật được đánh giá bằng việc giả sử rằng tất cả các yếu tố khác (ví dụ: tốc độ vi xử lý, …) là hằng số và không ảnh hưởng tới sự triển khai giải thuật.
- **Phân tích tiệm cận**: Việc phân tích giải thuật này được tiến hành sau khi đã tiến hành trên một ngôn ngữ lập trình nào đó. Sau khi chạy và kiểm tra đo lường các thông số liên quan thì hiệu quả của giải thuật dựa trên các thông số như thời gian chạy, thời gian thực thi, lượng bộ nhớ cần dùng, …

Chương này chúng ta sẽ tìm hiểu phân tích lý thuyết. Còn phân tích tiệm cận chúng ta sẽ cùng tìm hiểu ở chương tiếp theo.

Độ phức tạp giải thuật (Algorithm Complexity)

Về bản chất, độ phức tạp giải thuật là một hàm ước lượng (có thể không chính xác) số phép tính mà giải thuật cần thực hiện (từ đó dễ dàng suy ra thời gian thực hiện của giải thuật) đối với bộ dữ liệu đầu vào (Input) có kích thước **n**. Trong đó, n có thể là số phần tử của mảng trong trường hợp bài toán sắp xếp hoặc tìm kiếm, hoặc có thể là độ lớn của số trong bài toán kiểm tra số nguyên tố, …

Giả sử X là một giải thuật và n là kích cỡ của dữ liệu đầu vào. Thời gian và lượng bộ nhớ được sử dụng bởi giải thuật X là hai nhân tố chính quyết định hiệu quả của giải thuật X:

- **Nhân tố thời gian**: Thời gian được đánh giá bằng việc tính số phép tính chính (chẳng hạn như các phép so sánh trong thuật toán sắp xếp).
- **Nhân tố bộ nhớ**: Lượng bộ nhớ được đánh giá bằng việc tính lượng bộ nhớ tối đa mà giải thuật cần sử dụng.
  Độ phức tạp của một giải thuật (một hàm f(n)) cung cấp mối quan hệ giữa thời gian chạy và/hoặc lượng bộ nhớ cần được sử dụng bởi giải thuật.

Độ phức tạp bộ nhớ (Space complexity) trong phân tích giải thuật

Nhân tố bộ nhớ của một giải thuật biểu diễn lượng bộ nhớ mà một giải thuật cần dùng trong vòng đời của giải thuật. Lượng bộ nhớ (giả sử gọi là **S(P)**) mà một giải thuật cần sử dụng là tổng của hai thành phần sau:

- Phần cố định (giả sử gọi là **C**) là lượng bộ nhớ cần thiết để lưu giữ dữ liệu và các biến nào đó (phần này độc lập với kích cỡ của vấn đề). Ví dụ: các biến và các hằng đơn giản, …
- Phần biến đổi (giả sử gọi là **SP(I)**) là lượng bộ nhớ cần thiết bởi các biến, có kích cỡ phụ thuộc vào kích cỡ của vấn đề. Ví dụ: cấp phát bộ nhớ động, cấp phát bộ nhớ đệ qui, …
  Từ trên, ta sẽ có `S(P) = C + SP(I)`. Bạn theo dõi ví dụ đơn giản sau:

```
Giải thuật: tìm tổng hai số A, B
Step 1 -  Bắt đầu
Step 2 -  C ← A + B + 10
Step 3 -  Kết thúc
```

Ở đây chúng ta có ba biến A, B và C và một hằng số. Do đó: `S(P) = 1+3`.

Bây giờ, lượng bộ nhớ sẽ phụ thuộc vào kiểu dữ liệu của các biến và hằng đã cho và sẽ bằng tích của tổng trên với bộ nhớ cho kiểu dữ liệu tương ứng.

Độ phức tạp thời gian (Time Complexity) trong phân tích giải thuật

Nhân tố thời gian của một giải thuật biểu diễn lượng thời gian chạy cần thiết từ khi bắt đầu cho đến khi kết thúc một giải thuật. Thời gian yêu cầu có thể được biểu diễn bởi một hàm **T(n)**, trong đó T(n) có thể được đánh giá như là số các bước.

Ví dụ, phép cộng hai số nguyên n-bit sẽ có n bước. Do đó, tổng thời gian tính toán sẽ là **T(n) = c*n**, trong đó **c** là thời gian để thực hiện phép cộng hai bit. Ở đây, chúng ta xem xét hàm T(n) tăng tuyến tính khi kích cỡ dữ liệu đầu vào tăng lên.

Phân tích tiệm cận trong Cấu trúc dữ liệu và Giải thuật

Phân tích tiệm cận của một giải thuật là khái niệm giúp chúng ta ước lượng được thời gian chạy (Running Time) của một giải thuật. Sử dụng phân tích tiệm cận, chúng ta có thể đưa ra kết luận tốt nhất về các tình huống trường hợp tốt nhất, trường hợp trung bình, trường hợp xấu nhất của một giải thuật. Để tham khảo về các trường hợp này, bạn có thể tìm hiểu chương **Cấu trúc dữ liệu là gì ?**.

Phân tích tiệm cận tức là tiệm cận dữ liệu đầu vào (Input), tức là nếu giải thuật không có Input thì kết luận cuối cùng sẽ là giải thuật sẽ chạy trong một lượng thời gian cụ thể và là hằng số. Ngoài nhân tố Input, các nhân tố khác được xem như là không đổi.

Phân tích tiệm cận nói đến việc ước lượng thời gian chạy của bất kỳ phép tính nào trong các bước tính toán. Ví dụ, thời gian chạy của một phép tính nào đó được ước lượng là một hàm **f(n)** và với một phép tính khác là hàm **g(n^2)**. Điều này có nghĩa là thời gian chạy của phép tính đầu tiên sẽ tăng tuyến tính với sự tăng lên của n và thời gian chạy của phép tính thứ hai sẽ tăng theo hàm mũ khi n tăng lên. Tương tự, khi n là khá nhỏ thì thời gian chạy của hai phép tính là gần như nhau.