

Những cách tiếp cận bài toán: Phần 1

Những cách tiếp cận bài toán: Phần 1

Bài viết bởi [leadhyena_inran](#) [🔗](#).

Nguồn: [Topcoder](#) [🔗](#)

Giải bài là một nghệ thuật, nó có thể gây khó khăn cho cả những coder kì cựu nhất cũng như những người mới học. Để tìm cách giải phù hợp, chúng ta phải biết kết hợp nhiều phương pháp khác nhau như cảm nhận bằng trực giác, sáng tạo và thậm chí là may mắn. Cảm giác bất lực trước những bài khó thường làm chúng ta nản chí, và có thể bỏ cuộc giữa chừng. Tuy nhiên, không gì là không thể nếu bạn có đủ quyết tâm cũng như kĩ năng cần thiết. Bởi lí do đó, bài hướng dẫn này sẽ đưa ra những kĩ năng để giúp các bạn chọn những hướng tiếp cận rõ ràng cho những bài toán như vậy.

Tư duy theo lối mòn (Pattern Mining and Wrong Mindset)

Chúng ta rất dễ hướng các suy nghĩ vào việc tìm giải thuật dựa trên từng dạng của đề bài. Với những người đã làm khá nhiều bài tập, họ sẽ nhận ra được dạng của những bài toán khác nhau, và khi đọc thấy một bài toán nào đó quen thuộc, thì suy nghĩ hiện ra ngay tức khắc trong đầu là: *"Ồ, đây là dạng bài X, nên sử dụng thuật toán này, áp dụng kĩ thuật này"*. Cách này khá hữu ích đối với những người mới học: nếu chưa thể giải được, thì họ sẽ tập trung vào luyện tập với những dạng đề bài này để khiến nó trở thành một dạng bài quen thuộc.

Tuy nhiên, cách tiếp cận này cũng có nhược điểm khá lớn. Có những lúc ta tưởng rằng mình đang code đúng hướng nhờ vào lối tư duy như vậy, nhưng khi test thử thì không khớp một ví dụ nào cả. Lúc đó đọc kĩ lại đề bài mới nhận ra: mình chưa gặp "dạng" này bao giờ và bế tắc hoàn toàn. Bạn sẽ trải nghiệm điều này nhiều lần khi giải các bài toán "gốc" và đọc đáo, và kể cả những coder "lão làng" cũng có lúc bị chính kinh nghiệm của mình đánh lừa.

Tư duy lối mòn thường khiến ta ảo tưởng rằng mọi bài toán đều đã được "phân loại" và ta chỉ cần áp dụng "đúng thuật", trong khi thực tế điều này bất khả thi. Khi ta tạm quên hết những lối mòn đó và sử dụng hết khả năng suy nghĩ, phân tích, trình độ của ta sẽ được cải thiện rất nhiều.

Coding Kata

Đây là thử thách đầu tiên: hãy chọn bất cứ bài tập nào trong Practice Rooms mà bạn chưa làm. Hốt nó, cho dù phải mất bao nhiêu thời gian, và hoàn thành nó (có thể sử dụng editorial nếu quá "bí"). Sau khi làm xong (accepted), hãy xem bạn mất bao nhiêu thời gian để giải. Tiếp theo, xóa hết code và **gõ** lại lần nữa, check thời gian sau khi làm xong. Tiếp tục lần thứ 3 như thế.

Thời gian hoàn thành lần đầu tiên là thời gian để giải bài toán khi bạn không có bất cứ hướng suy nghĩ hay cách tiếp cận nào về bài toán. Thời gian hoàn thành lần thứ 2 là thời gian lần đầu tiên trừ đi thời gian bạn dùng để đọc và hiểu đề (đừng quá ngạc nhiên với số lỗi bạn lặp lại ở lần này). Còn thời gian hoàn thành lần cuối là thời gian bạn có thể giải được trong lúc thi thật nếu tìm thấy được cách tiếp cận đúng ngay sau khi đọc đề bài. Việc giải bài với thời gian của lần cuối cùng hoàn toàn có thể làm được (thậm chí không cần khả năng gõ phím nhanh). Nhưng điều mà bạn cũng nhận ra ở lần giải thứ 3 là cảm giác bạn đã biết trước chiến thuật, nên code thế nào, dễ bị bug nhất ở những đoạn nào, ... Đó là cảm giác để có được một cách tiếp cận đúng.

Ở nhiều môn võ, có một thuật ngữ luyện tập gọi là *kata*. Người tập sẽ thi triển những đòn thế được kịch bản sẵn và cố phòng thủ trước các đòn tấn công - cũng đã được kịch bản. Thoạt nghe thì phương pháp này chả có tác dụng gì vì sẽ chẳng lúc nào đánh nhau được "tiện" như vậy. Hơn nữa có vẻ như nó khuyến khích lối tư duy, luyện tập theo lối mòn được đề cập ở trên. Thế nhưng *kata* sẽ cho người tập cảm giác của việc có một kế hoạch rõ ràng và khuyến khích ý chí. Tin học cũng như vậy; chỉ sau khi code 3 lần một dạng bài trong một thời gian đủ nhiều ta mới nhuần nhuyễn hết mọi mặt của bài toán - định hướng, trình bày code và debug.

Chiến thuật định hướng giải bài (Approach Tactics)

Bây giờ bạn đã biết việc định hướng giải bài nó như thế nào và nó bao gồm những gì, bạn sẽ nhận ra rằng trước đây bạn đã biết khá nhiều cách để tiếp cận một bài toán. Dạng như "*Tôi dùng Quy hoạch động (dynamic programming) cho bài đó*", "*Tôi có thể dùng tham lam (greedy) để giải bài này không?*", "*Đừng nói với tôi rằng duyệt (brute-force) có thể ăn bài này*". Thật ra bạn đang dùng sai những từ ngữ ở đây, vì bạn không thể phân loại mỗi bài tập chỉ với tham hay duyệt. Có rất rất nhiều dạng bài tập, và cũng rất nhiều cách tiếp cận, và thậm chí với mỗi hướng tiếp cận có thể tồn tại rất nhiều cách thể hiện khác nhau. Cách dùng từ như vậy chỉ tổng quát được những bước đi lớn để đến được đáp án.

Trong TopCoder có nhiều bài có lời giải hoàn chỉnh và chú thích cận kẽ. Lần tham gia contest tiếp theo, bạn hãy thử tìm một bài viết tốt và nghiên cứu các bước tiếp cận cụ thể của bài toán mà tác giả đã chỉ ra. Bạn sẽ bắt đầu chú ý rằng có một tính chất nào đó của bài toán sẽ gợi ra cho bạn một cách tiếp cận, hay những cách biến đổi, chuyển hướng, phân tích, ... làm bạn đến gần hơn với đáp án, hay ít nhất giúp bạn không nghĩ theo hướng sai. Khi tìm một cách tiếp cận, hãy thử tất cả những hướng đi bạn có, như kiểu bạn đã gõ sẵn code hết trong đầu - bạn thuyết phục bản thân rằng hướng đi của bạn là đúng.


Những coder có kiến thức toán cơ bản tốt sẽ có thể biết được hướng suy nghĩ này, bởi vì những hướng tiếp cận đó rất giống với kĩ năng chứng minh toán học. Với những kì thủ họ sẽ thấy nó giống như việc nghĩ trước các nước cờ hay những người thiết kế ứng dụng có thể đã quen với nó trong khi làm việc với các [design patterns](#) (thuật ngữ trong lập trình ứng dụng, chỉ những phương pháp thiết kế hiệu quả đã được nghiên cứu kĩ)...

Để luyện tập cách tư duy và tìm ra phương pháp phù hợp với mình, ghi chú lại hướng giải của các bài toán là rất quan trọng. Sau đó hãy viết một bài phân tích dựa trên những gì bạn đã làm, càng chi tiết càng tốt, để những người khác có thể biết được cách bạn suy nghĩ như thế nào. Việc viết lại như thế không chỉ giúp bạn hiểu rõ hơn cách tiếp cận của mình, mà còn tìm ra được những sai lầm của bản thân và cải thiện nó. Hãy nhớ rằng, rất khó để cải thiện được những điều mà bạn còn chưa hiểu.

Chia nhỏ vấn đề


Hãy bàn về một trong những cách tiếp cận phổ biến nhất: chia nhỏ vấn đề ra, còn được gọi là tiếp cận từ trên xuống (top-down programming). Ý tưởng cơ bản ở đây là code của bạn phải được thực hiện theo thứ tự, nên

hãy bắt đầu bằng việc suy nghĩ xem phần chương trình chính sẽ cần những gì trước khi xây dựng những hàm con. Điều này cho phép bạn thấy được tổng thể chương trình, cũng như chia nhỏ vấn đề từ những thứ phức tạp cho đến đơn giản.

Một ví dụ áp dụng cách tiếp cận này. [Round 2, MatArith, topcoder Invitational 2002](#) , bài toán yêu cầu bạn phải tính giá trị biểu thức có sử dụng ma trận. Để có được những con số chúng ta phải xử lý chuỗi (vì input là string), tính toán dựa trên đó, và trả ngược về string. Nên bạn cần một hàm `print` để in kết quả, một hàm `parse` để xử lý chuỗi và một hàm `calc` để xử lý toán học. Không cần suy nghĩ quá phức tạp, Nếu chúng ta có đủ 3 hàm đó thì vấn đề sẽ được giải quyết chỉ trong 1 dòng code:

```
1 | public String[] calculate(String[] A, String[] B, String[] C, String eval){
2 |     return print(calc(parse(A), parse(B), parse(C), eval));
3 | }
```

Cái hay của các tiếp cận này là tạo ra một hệ thống cấp bậc các hàm, hàm trên sẽ dựa vào kết quả của hàm dưới để chạy. Bây giờ công việc đã được chia làm 3 phần: tạo hàm `parse`, hàm `print`, và hàm `calc`. Nếu bạn chia bài toán thành những bài toán con đủ nhỏ, bạn sẽ không phải tốn nhiều thời gian vào việc suy nghĩ cho những bước không đơn giản, bởi vì nó sẽ trở nên atomic (sẽ được đề cập phía dưới). Thay vào đó chúng ta có thể tập trung vào hàm cộng và hàm nhân ma trận hay hàm đọc dữ liệu vào sao cho chính xác.

Cách tiếp cận này có thể ứng dụng khá hiệu quả vào những bài toán đệ quy. Toàn bộ ý tưởng của đệ quy đó là chia vấn đề lớn thành những vấn đề nhỏ hơn nhưng giống chính xác với vấn đề gốc, vậy chúng ta chỉ cần giải được một 1 vấn đề gốc là xong. Cách tiếp cận này cũng được sử dụng rất phổ biến trong [functional programming](#)  (một lớp các ngôn ngữ lập trình mà chương trình được xây dựng chỉ từ các hàm). Có khá nhiều bài viết đã bàn sâu về vấn đề này, nhưng ý tưởng cơ bản chính là nếu ta chia nhỏ bài toán một cách đúng đắn, chương trình sẽ tự động gửi tham số qua mỗi lời gọi hàm, không cần thiết lưu trữ dữ liệu giữa các bước. Nhưng có một nhược điểm là sẽ khó debug hơn.

Kế hoạch debug

Bất cứ khi nào có một cách tiếp cận, chúng ta phải luôn có một kế hoạch debug cho cách tiếp cận đó. Có rất nhiều khả năng xảy ra làm cho chương trình chúng ta chạy sai, bằng cách suy nghĩ trước các hướng chương trình sẽ chạy sai, chúng ta có thể ngăn chặn bug trước khi code. Hơn thế nữa, nếu không vượt qua được các test mẫu, chúng ta sẽ biết ngay vị trí thích hợp để bắt đầu tìm lỗi. Cuối cùng, bằng cách chú ý đến điểm trọng yếu nhất, sẽ dễ dàng hơn để chứng minh cách tiếp cận của bạn là đúng.

Với trường hợp tiếp cận top-down, việc chia nhỏ những phần code thành những hàm con giúp chúng ta cô lập những phần code lỗi và chỉ phải sửa lại code ở một nơi. Cũng như việc test được một tổ hợp các hàm để bị bug nhất. Copy và paste nhiều lúc cũng làm việc sửa lỗi tốn nhiều công sức hơn thay vì viết riêng một hàm con. Một kinh nghiệm nữa cho cách tiếp cận từ trên xuống là chúng ta nên tìm bugs bên trong hàm trước khi tìm giữa những lời gọi hàm. Những cách này tạo nên một chiến thuật debug: biết được vị trí nào cần test trước tiên, chỗ nào bạn nghĩ là sai và cần kiểm tra, hay chứng minh những đoạn code chuẩn và bỏ qua chúng. Nếu luyện tập thường xuyên, bạn sẽ thấy rằng những kĩ năng này vô cùng hữu ích.

Atomic Code

Nếu chúng ta đi đến một phần code mà chúng ta không thể chia nhỏ được nữa, thì phần đó gọi là **atomic code**. Hi vọng các bạn đều biết cách code những phần như vậy, vì đó sẽ là những phần bạn sẽ code thường xuyên nhất. Nhưng đừng lo nếu bạn chưa thuần thục nó, những atomic code khó như vậy thường làm cho bài toán trở nên thú vị hơn, và thỉnh thoảng thấy trước được điều đó sẽ giúp bạn có một bước tiến lớn trong việc tìm cách tiếp cận đúng đắn, hạn chế được việc phung phí thời gian cho những cách tiếp cận sai.

Dạng atomic code bạn viết thường xuyên nhất đó là dạng tối giản nhất, nên phần lớn sẽ có lệnh sẵn trong thư viện để hỗ trợ nó. Việc sử dụng thành thạo thư viện là một lợi thế lớn, nó giúp chúng ta viết được atomic code trong sáng và rõ ràng hơn. Còn cách nào để tiết kiệm thời gian hơn khi bạn biết rằng một phần atomic code khá khó xoi lại là một hàm hay một lớp nào đó trong thư viện sẵn có?

Mình sẽ gọi dạng atomic code thứ hai mà bạn sẽ viết là language techniques (một dạng kĩ năng về ngôn ngữ). Đó là những phần code nhỏ mà bạn thuộc lòng để giải quyết một phép xử lí nào đó, như tìm chỉ số của phần tử nhỏ nhất đầu tiên trong mảng, hay tách một xâu ra thành những chuỗi con mà không có khoảng trắng. Những kĩ thuật trên rất cần thiết cho việc tìm cách tiếp cận bài toán, bởi vì nếu bạn biết rõ cách hoạt động của những thao tác cơ bản đó, sẽ làm cho việc quá trình chia nhỏ nhanh đến atomic hơn, nó cũng sẽ hạn chế việc tạo ra bug không đáng có. Thêm nữa, nếu bạn cần phải cài một hàm hơi tương tự như một language technique mà bạn đã biết, thì việc đó sẽ trở nên vô cùng dễ dàng dựa trên việc chỉ cần sửa đổi nó chút ít. Luyện tập với những language technique đó nên là việc làm hàng ngày, và bất kì atomic code nào cũng sẽ có thể bay ra khỏi bàn tay (và bay vào màn hình) ngay khi bạn vừa nghĩ đến nó.

Về việc sử dụng thư viện cá nhân, tôi thấy có nhiều người sử dụng thư viện bằng cách chèn những đoạn code thường dùng vào phần đầu của file. Điều này hoàn toàn hợp lí và hợp pháp. Những ưu điểm của cách làm này là có thể tạo ra được nhiều phần trước cho những atomic code thường gặp, có nhiều lợi ích cho việc tiếp cận top-down, cũng như bottom-up (sẽ đề cập sau). Tuy nhiên, theo ý kiến cá nhân, nhược điểm của cách làm này không tương xứng với hiệu quả. Nó có thể làm chậm code của bạn đi, bởi vì bạn phải tìm cách làm cho input của đề bài khớp với đầu vào của thư viện của bạn, hoặc thư viện của bạn không được định nghĩa đủ tốt để làm điều đó. Điều hiển nhiên là những thư viện bạn tự viết có thể bị bugged, và việc debug thư viện của bạn lúc đang thi đấu sẽ rất nguy hiểm bởi vì bạn phải sửa lại những bài bạn đã submit trước khi xem thêm bất kì bài nào khác. Thêm nữa, thư viện cá nhân không được sử dụng ở các kì thi onsite, ví dụ như thi HSG QG. Cuối cùng, việc sử dụng thư viện cá nhân sẽ làm bạn phải học kĩ năng sử dụng nó chứ không phải kĩ năng của ngôn ngữ, khiến việc đọc hiểu được bài giải của người khác khó hơn. Nếu sử dụng đúng lúc, thì nó sẽ là một vũ khí hiệu quả, nhưng nó không phải là vũ khí vạn năng trong mọi trường hợp.

Có những lúc bạn sẽ gặp những atomic code mà bạn không thể chia nhỏ vấn đề được nữa. Đây là lúc bạn phải bắt đầu phân tích cách tiếp cận hiện tại của bạn. Tôi có nên chia nhỏ vấn đề theo cách khác không? Có nên lưu trữ dữ liệu bằng cấu trúc khác? Đây có phải là nhân tố để làm cho bài toán trở nên khó hơn? Bạn phải xét đến những thứ đó trước khi tìm ra đáp án. Thậm chí ở thời điểm bạn nhận ra bạn không tìm ra được hướng suy nghĩ tiếp theo, thì vẫn luôn có cách để thoát ra khỏi tình huống đó, mình sẽ giới thiệu nó ở [phần tiếp theo](#).

Được cung cấp bởi [Wiki.js](#)