Hướng dẫn học Khoa học Máy tính của Beej

Brian “Beej Jorgensen” Hall

Contents

Chapter 1

Lời Tựa

Bạn đang bắt đầu tìm hiểu về Khoa học Máy tính, hay đang cân nhắc về điều đó? Bạn đang nghĩ đến việc lấy bằng? Hay bạn đã đang theo học rồi. Cuốn hướng dẫn tổng quan này dành cho bạn!

Tôi sẽ không nói nhiều về cách viết code (cũng ít thôi). Và thật ra, tôi cũng sẽ không nói về khoa học và toán học đằng sau Khoa học Máy tính — nghe có vẻ mâu thuẫn. Điều tôi sẽ nói trong khoảng 40 trang này là *cách học* khi bạn còn là một lập trình viên mới vào nghề.

Nào, dù tôi rất muốn biết chính xác cách mọi người học (và nhồi nhét điều đó vào 40 trang), nhưng thành thật mà nói, tôi không biết.

Điều tôi có là hơn 40 năm kinh nghiệm lập trình (tự học trước khi vào đại học), 20 năm kinh nghiệm trong ngành, và hơn 8 năm kinh nghiệm giảng dạy. Cùng với bằng Cử nhân và Thạc sĩ Khoa học Máy tính.

Và tôi có những quan điểm về cách tốt nhất để học lập trình!

Nói thẳng ra luôn: bạn có thể hoàn toàn không đồng ý với những gì tôi viết ở đây. Và tôi không sao với điều đó.

Nhưng tôi đã có cơ hội chứng kiến sinh viên mắc phải đủ loại sai lầm. Và hy vọng tôi có thể giúp một số bạn đọc tránh được những sai lầm đó ngay từ đầu.

Sinh viên và giảng viên: nếu bạn thấy điều gì đó không đồng ý hoặc điều quan trọng còn thiếu, xin đừng ngần ngại cho tôi biết[1] để tôi có thể cải thiện hướng dẫn này.

Khuyến cáo: như với tất cả các hướng dẫn tôi viết, tôi không phải bậc thầy về chủ đề này. Với một chủ đề mơ hồ như cách con người học, tôi càng không phải bậc thầy hơn.

Nhưng hãy đọc và lấy những gì hữu ích, phần còn lại thì để lại cho boids[2].

1.1 Đối tượng độc giả

Sinh viên đại học mới bắt đầu học lập trình là những người tôi nhắm đến khi viết cuốn này. Vậy nên độc giả mục tiêu xấp xỉ trong phạm vi đó. Những người đang học trung học hoặc chỉ muốn học lập trình cũng có thể là một phần trong số đó.

1.1.1 Về thuật ngữ "Khoa học Máy tính"

Cuốn hướng dẫn này, ở một mức độ nào đó, được đặt tên chưa thật chuẩn. Nó không thực sự nói về những gì hầu hết mọi người coi là "khoa học máy tính". Nó nói nhiều hơn về lập trình — thứ mà các nhà khoa học máy tính, kỹ sư phần mềm, lập trình viên đều làm rất nhiều.

Tôi đặt tên nó là *Hướng dẫn về Khoa học Máy tính* vì một vài lý do:

[1] mailto:beej@beej.us
[2] https://en.wikipedia.org/wiki/Boids

- Ít nhất ở Mỹ, chương trình học mà các nhà phát triển phần mềm theo đuổi được gọi là "Khoa học Máy tính" và những người bắt đầu học chương trình đó chính là đối tượng mục tiêu.
- Khi bạn bắt đầu học Khoa học Máy tính (bằng cấp), bạn sẽ làm những thứ trong hướng dẫn này. Sau đó, khi bạn đến phần Khoa học Máy tính (khoa học thực sự) của chương trình, bạn sẽ đã vượt qua nó rồi.

Vậy nên Hướng dẫn này phù hợp để bắt đầu (hy vọng vậy), nhưng hãy hiểu rằng bạn chỉ đang chập chững bước vào một đại dương bao la chờ được khám phá.

1.2 Trang chủ chính thức

Địa chỉ chính thức của tài liệu này là:

`https://beej.us/guide/bglcs/` [3].

1.3 Sửa lỗi

Tôi cung cấp các hướng dẫn này miễn phí với hy vọng chân thành rằng mọi người sẽ thấy chúng thực sự hữu ích. Nếu có gì đó chưa thực sự hữu ích (hoặc, bạn biết đấy, "sai"), tôi rất muốn được nghe để có thể sửa chữa nhằm thực hiện sứ mệnh của mình.

- Gửi email cho tôi[4]
- Báo cáo sự cố[5]
- Gửi pull request[6]

Cảm ơn!

1.4 Chính sách Email

Nhìn chung tôi sẵn sàng hỗ trợ qua email, hãy cứ viết, nhưng tôi không thể đảm bảo sẽ trả lời. Tôi có cuộc sống khá bận rộn và đôi khi không thể trả lời câu hỏi của bạn. Khi đó, tôi thường chỉ xóa tin nhắn đi. Không có gì cá nhân cả; chỉ là tôi sẽ không bao giờ có thời gian để đưa ra câu trả lời chi tiết mà bạn cần.

Về nguyên tắc, câu hỏi càng phức tạp thì khả năng tôi trả lời càng thấp. Nếu bạn có thể thu hẹp câu hỏi trước khi gửi cho tôi, và đảm bảo đính kèm mọi thông tin liên quan (như nền tảng, trình biên dịch, thông báo lỗi, và bất cứ thứ gì bạn nghĩ có thể giúp tôi gỡ lỗi), bạn sẽ có nhiều khả năng nhận được phản hồi hơn.

Nếu không nhận được phản hồi, hãy tiếp tục mày mò, cố tìm câu trả lời, và nếu vẫn bế tắc, hãy viết lại cho tôi cùng với những thông tin bạn đã tìm được — hy vọng đủ để tôi có thể giúp được.

Sau khi đã dặn dò bạn về cách viết và không viết cho tôi, tôi chỉ muốn nói rằng tôi *thực sự* trân trọng mọi lời khen mà cuốn hướng dẫn này đã nhận được qua nhiều năm. Đó là nguồn động lực thực sự, và tôi rất vui khi biết nó đang được sử dụng tốt! `:-)` Cảm ơn!

1.5 Nhân bản

Bạn hoàn toàn được chào đón khi nhân bản trang này, dù công khai hay riêng tư. Nếu bạn nhân bản trang này công khai và muốn tôi liên kết đến từ trang chính, hãy gửi cho tôi một dòng tại `beej@beej.us`.

1.6 Lưu ý dành cho người dịch

Nếu bạn muốn dịch hướng dẫn này sang ngôn ngữ khác, hãy viết cho tôi tại `beej@beej.us` và tôi sẽ liên kết đến bản dịch của bạn từ trang chính. Hãy tự do thêm tên và thông tin liên lạc của bạn vào bản

[3] https://beej.us/guide/bglcs/
[4] mailto:beej@beej.us
[5] https://github.com/beejjorgensen/bglcs/issues
[6] https://github.com/beejjorgensen/bglcs

dịch.

Xin lưu ý các hạn chế về giấy phép trong phần Bản quyền và Phân phối bên dưới.

1.7 Bản quyền và Phân phối

1.8 Lời cảm ơn

Những điều khó khăn nhất khi viết các hướng dẫn này là:

- Tìm hiểu tài liệu đủ chi tiết để có thể giải thích được
- Tìm ra cách giải thích rõ ràng nhất — một quá trình lặp đi lặp lại dường như vô tận
- Tự đặt mình vào vị trí một *chuyên gia* được gọi là, trong khi thực ra tôi chỉ là một con người bình thường đang cố hiểu tất cả, giống như tất cả mọi người
- Kiên trì tiếp tục khi quá nhiều thứ khác thu hút sự chú ý của tôi

Nhiều người đã giúp tôi trong quá trình này, và tôi muốn ghi nhận những ai đã làm cho cuốn sách này trở thành hiện thực.

- Tất cả mọi người trên Internet đã quyết định chia sẻ kiến thức của họ dưới hình thức này hay hình thức khác. Việc chia sẻ tự do thông tin hướng dẫn là điều làm cho Internet trở thành nơi tuyệt vời như vậy.
- Tất cả những ai đã gửi sửa lỗi và pull request về mọi thứ từ hướng dẫn dễ gây nhầm lẫn đến lỗi chính tả.

Cảm ơn! ♥

Chapter 2

Mục tiêu chính

> *"Giáo dục không phải là sự chuẩn bị cho cuộc sống; giáo dục chính là cuộc sống."*
>
> —John "Not The Decimal System One" Dewey

> *"Người mù chữ của thế kỷ 21 không phải là những người không biết đọc và viết, mà là những người không biết học, không biết bỏ đi những gì đã học, và không biết học lại."*
>
> —Alvin Toffler

Chúng ta học gì ở trường? Cách trở thành lập trình viên Flutter? Cách trở thành lập trình viên React? Cách trở thành lập trình viên Rust? Cách trở thành lập trình viên JavaScript? Cách trở thành lập trình viên C++? Cách trở thành lập trình viên C? Cách trở thành lập trình viên Pascal? Cách trở thành lập trình viên LISP? Cách trở thành lập trình viên FORTRAN? Cách trở thành lập trình viên COBOL?

Bạn có thấy điều tôi vừa làm không? Ngoài việc đặt ra rất nhiều câu hỏi?

Vâng, bạn có thể muốn vào đại học để làm việc với web development hoặc embedded systems bằng những ngôn ngữ mới nhất và tốt nhất. Và nếu may mắn, bạn sẽ được làm một phần điều đó ở trường.

Nhưng đây là vấn đề:

1. Có quá nhiều công nghệ để bao quát trong bốn năm.
2. Tất cả những thứ đó sẽ sớm lỗi thời dù sao. Bạn thấy tôi đã thêm COBOL[1] vào danh sách không[2]?

Vậy là sinh viên bạn phải làm gì? Không có cách nào bạn có thể học hết tất cả.

Đây là lúc mục tiêu chính phát huy tác dụng. Nhiệm vụ của bạn với tư cách là sinh viên là làm một điều duy nhất:

Học cách giải quyết bất kỳ bài toán lập trình nào.

Dù bạn chưa từng thấy bài toán hay công nghệ đó trong đời.

Đó là toàn bộ mục tiêu.

Quan trọng là, mục tiêu không phải là học trở thành lập trình viên iPhone hay Android hay Go. Tất cả những thứ đó đều nằm trong mục tiêu chính. Bạn có thể không học lập trình Go ở trường, nhưng bạn sẽ *học cách tự học lập trình Go.*

Học cách giải quyết bất kỳ bài toán lập trình nào.

Chỉ vậy thôi. Mọi thứ khác chỉ là phụ trợ.

Khả năng tự học là một kỹ năng **bắt buộc** trong lĩnh vực phát triển phần mềm. *Thực sự* khó có khả năng công việc đầu tiên của bạn chỉ sử dụng các công nghệ bạn đã học ở trường. Và thực tế, những sinh viên mới ra trường có thể ngạc nhiên khi phát hiện ra rằng **không có** công nghệ nào họ dùng ở trường lại xuất hiện trong công việc đầu tiên của họ.

[1] https://en.wikipedia.org/wiki/COBOL

[2] Thật ra trò đùa quay lại hại tôi. Vẫn còn rất nhiều công việc liên quan đến COBOL ngoài kia.

Vậy tại sao trên đời chúng ta lại dành bốn năm học tất cả những thứ nhảm nhí này về hệ điều hành, ngôn ngữ assembly, phân tích thuật toán và—?

EERRRNT! [tiếng còi vang] Bạn không chỉ dành bốn năm làm điều đó. Bạn vừa dành bốn năm *học cách giải quyết bất kỳ bài toán lập trình nào.*

Và hãy nghĩ về điều đó. Làm thế nào bạn có thể dạy mọi người giải quyết *bất kỳ* bài toán nào? Bạn không thể dạy toàn bộ hàng tỷ ngôn ngữ lập trình[3], framework, và kỹ thuật. Vậy nên điều đó không khả thi. Và dù bạn chọn những thứ nào, chúng có thể hoặc không được dùng trong công việc của một người cụ thể.

Vì vậy, chúng ta phải đi sâu hơn vào nền tảng. Chúng ta phải luyện tập giải quyết bài toán nhiều lần đến mức phát triển và nâng cao kỹ năng giải quyết vấn đề. Bởi vì bạn sẽ đối mặt với những bài toán trong phỏng vấn hay ở nơi làm việc hoàn toàn xa lạ. Bạn sẽ không thể dựa vào bất kỳ ngôn ngữ hay thuật toán cụ thể nào đã học. Thứ duy nhất bạn có thể dùng chính là kỹ năng giải quyết vấn đề của mình.

Cuối cùng, có một hệ quả nhỏ ở đây: khi bạn đang học làm gì đó (dù bạn sẽ không bao giờ dùng nó ở nơi làm việc), *đừng gian lận.* Mục tiêu không phải là học cách xóa phần đầu của danh sách liên kết. Mục tiêu là luyện tập giải quyết bài toán lập trình! Và việc chỉ tra cứu câu trả lời sẽ tước đi cơ hội luyện tập đó của bạn. Gian lận qua hết các bài tập ở trường và bạn sẽ không bao giờ phát triển được kỹ năng nền tảng duy nhất của các lập trình viên phần mềm: khả năng giải quyết bất kỳ bài toán lập trình nào.

2.1 Suy ngẫm về chương

- Mục tiêu chính của một sinh viên khoa học máy tính là gì?
- Tại sao việc học ngôn ngữ và framework *không phải* là mục tiêu chính?
- Bạn phải làm gì để đạt được mục tiêu chính?
- Tại sao gian lận trong các lớp khoa học máy tính lại là ý tưởng tồi xét về phương diện học tập của bạn?

[3] https://en.wikipedia.org/wiki/List_of_programming_languages

Chapter 3

Tư duy phát triển

Đây là chủ đề khó nuốt với tôi, nói thật. Tôi không thích thất bại, dù không ai nhìn thấy, và càng không thích khi có người chứng kiến. Nó thắt ruột tôi lại và rồi tôi tự nói với bản thân đủ thứ tệ mà tôi chẳng bao giờ nói với người khác.

Và tôi vẫn làm vậy dù biết đó là trò thua cuộc và hoàn toàn mâu thuẫn với lời khuyên tôi sắp đưa ra trong phần này.

Thay vào đó tôi nên làm gì?

Nhà tâm lý học Carol Dweck[1] đã phổ biến thuật ngữ *tư duy phát triển* (growth mindset).

Ý chính của nó là:

- Bạn có thể mở rộng kỹ năng bằng nỗ lực
- Đón nhận việc học suốt đời
- Thử thách là cơ hội phát triển
- Học hỏi từ chỉ trích và thất bại
- Không bao giờ bỏ cuộc, không bao giờ đầu hàng![2]

Đại loại như vậy.

Đó là ngược lại với những gì tôi thường làm khi thua ván cờ Go[3] thứ mấy tỷ. Sao mình lại có thể mắc nhiều sai lầm ngu ngốc đến vậy? Mình sẽ mãi không giỏi thứ này!

Nhưng đó là điều Dweck sẽ gọi là *tư duy cố định* (fixed mindset). Đó là niềm tin sai lầm của tôi rằng dù tôi chơi bao nhiêu, tôi vẫn bị giới hạn bởi những hạn chế cố hữu của bản thân và sẽ không bao giờ vượt qua được dù học 500.000 giờ!

Và khi nói ra như vậy, nó có vẻ khá ngớ ngẩn. Không ai có thể dành 500.000 giờ làm *bất cứ điều gì* mà không trở nên tốt hơn.

> *Hãy thua 50 ván đầu tiên càng nhanh càng tốt.*
>
> —Tục ngữ cờ Go

Vậy còn 50.000 giờ thì sao? 500 giờ? 50 giờ? 5 giờ?

Nghĩ lại, có vẻ như bất kỳ lượng luyện tập nào cũng sẽ là sự cải thiện.

Ngay cả khi bạn bị bế tắc hoàn toàn, bạn vẫn đang khám phá các hướng đi. Dù chúng hóa ra là ngõ cụt, bạn ít nhất đã biết chúng là vậy!

> *"Đó chẳng phải là cùng một thứ sao, như 'flammable' và 'inflammable'? Trời ơi, tôi học được điều đó theo cách khó nhất."*
>
> —Woody, *Cheers*

[1] https://en.wikipedia.org/wiki/Mindset#Fixed_and_growth_mindset
[2] Vâng, tôi là fan của *Galaxy Quest*.
[3] https://en.wikipedia.org/wiki/Go_(game)

> *"Tuyệt vời. Thêm một cái gọi là trải nghiệm học hỏi nữa."*
>
> —Mẹ tôi, người luôn mỉa mai

Mỗi thành công là một trải nghiệm học hỏi. Mỗi thất bại cũng là một trải nghiệm học hỏi. Mỗi trải nghiệm học hỏi cải thiện kỹ năng của bạn. Đừng sợ thất bại — hãy dùng nó để lên cấp.

3.1 Sự kiên trì

> *"Bậc thầy đã thất bại nhiều lần hơn số lần người mới bắt đầu thậm chí đã thử."*
>
> —Stephen McCranie

Tôi nghĩ đây là một trong những phẩm chất chính của bất kỳ ai vươn lên xuất sắc ở bất kỳ lĩnh vực nào. (Khái quát quá chứ?)

Ai sẽ đi xa hơn, người bỏ cuộc sau thất bại, hay người thất bại và thất bại và thất bại và thất bại nhưng vẫn đứng dậy, phủi bụi, và thử lại?

Đây là điều phân biệt các lập trình viên xuất sắc với những lập trình viên tầm thường. Các lập trình viên xuất sắc đã bị *quật ngã* hết lần này đến lần khác, và *họ vẫn tiếp tục tấn công* và tấn công và tấn công cho đến khi giải quyết được vấn đề. Và họ học được điều gì đó từ mỗi lần thất bại.

> *"Tôi chưa bao giờ thất bại. Tôi chỉ tìm ra 10.000 cách không hiệu quả."*
>
> —Thomas Edison

Và có điều liên quan mà sinh viên có thể không nhận ra: các giảng viên của bạn cũng đã thất bại và thất bại và thất bại và thất bại! Đúng, họ có thể live-code việc xóa linked list trong lớp và làm đúng ngay từ lần đầu... nhưng thực ra đó không phải lần đầu tiên, phải không? Đó là, như khoảng, lần thứ 100 họ làm điều đó. Bạn tự code phần xóa linked list từ đầu 100 lần thì bạn cũng sẽ làm đúng ngay từ lần đầu!

Khi nhìn giảng viên làm mọi thứ trông dễ dàng như không, bạn có thể nản lòng vì tài liệu này có vẻ *không thể* với bạn. Và bạn bắt đầu nghĩ rằng giảng viên và một số bạn học có năng lực lập trình tự nhiên mà bạn đơn giản là không được sinh ra với nó. Kỹ năng thần kỳ nào họ đã được ban tặng từ trong bụng mẹ mà bạn sẽ không bao giờ có? (Bạn có nghe thấy tư duy cố định đang nói không?)

Nhưng đây là bí mật: không có sự khác biệt nào giữa bạn và giảng viên của bạn ngoài số lần thất bại bạn đã trải qua. Bạn cần phải thất bại nhiều hơn nữa để đạt được trình độ như họ!

Rất, rất ít người là lập trình viên "thiên bẩm". 99,999% trong chúng ta phải làm việc thực sự chăm chỉ để học những thứ này.

Sự kiên trì không ngừng nghỉ là một trong những phẩm chất hàng đầu của các lập trình viên giỏi nhất thế giới.

3.2 Bạn phải thực sự muốn nó

Khi tôi đang suy nghĩ về việc lấy bằng Tiến sĩ (cuối cùng tôi không làm), cố vấn của tôi khuyên, "Bạn phải thực sự muốn nó." Hàm ý là việc lấy bằng Tiến sĩ đòi hỏi quá nhiều công sức nên tôi phải có đủ động lực để bỏ thời gian ra.

Và tôi nghĩ đây là lời khuyên hay nói chung về việc học Khoa học Máy tính.

Có quá nhiều thứ thực sự khó phải mày mò, bạn phải có đủ động lực để làm. Nó không dễ dàng. Chút nào.

Tôi thích dùng ví dụ về việc tôi làm kế toán. Tôi biết mình đủ thông minh để làm (học phụ ngành toán!), nhưng tôi cũng biết tôi *ghét* nó. Người bạn cùng phòng cũ của tôi trở thành kế toán và tôi đã xui xẻo mở một cuốn sách hướng dẫn học của cô ấy. Tôi sẽ không bao giờ mắc sai lầm đó nữa.

Não tôi tắt ngay lập tức khi người ta bắt đầu nói về nó. Hãy tưởng tượng một thứ gì đó chán ngắt đến phát khóc mà bạn ghét. Rồi tưởng tượng dành bốn năm nghiên cứu nó tỉ mỉ đến tận xương tủy.

Nói cho bản thân mình và phóng chiếu rộng ra, tôi nghĩ sẽ khó hơn nhiều khi tìm động lực để bỏ công sức làm điều bạn ghét.

Tôi đã bị chỉ trích khá nhiều trên Hacker News vì khẳng định này một thời gian trước. Nhiều người bình luận rằng họ ghét lập trình nhưng vẫn có sự nghiệp với nó.

Thứ nhất, đó là... điều đáng tiếc. Nhưng thứ hai, tôi vẫn lập luận rằng những người yêu thích khoa học máy tính dễ lấy bằng hơn những người ghét nó.

Và những người có vẻ là "thiên bẩm" chính là những người *thực sự* yêu thích nó. Họ nghĩ về khoa học máy tính *suốt* (vì sao lại không nhỉ?), và do đó họ luyện tập rất nhiều. Nhiều hơn hẳn so với những người ghét nó.

Vậy nên, dù rõ ràng bạn có thể có sự nghiệp trong lĩnh vực béo bở mà bạn không thích, (a) nó sẽ khó hơn cho bạn so với những người thích nó và (b) có lẽ bạn nên cân nhắc một lĩnh vực mà bạn thực sự thích?

Bạn phải thực sự muốn nó. Bạn có đủ muốn để vượt qua lượng nỗ lực khổng lồ cần thiết để học nó không? Có thể bạn ghét lập trình, nhưng bạn đủ muốn tiền. Có thể bạn không quan tâm đến tiền, nhưng bạn muốn lập trình mỗi giây trong ngày.

Chỉ cần đảm bảo bạn có đủ động lực để thực hiện điều đó.

3.3 Nó không dễ dàng

> *Khi bạn đọc cuộn giấy này*
> *Bạn sẽ dần dần trở nên*
> *Bối rối bởi những từ in trên đó*
>
> —"Scroll of Learning Disability", Tạp chí *Dragon*

Khoa học Máy tính *khó.* Thực sự khó. Nếu bạn quen với việc mọi thứ tương đối dễ vì bạn là người thông minh nhất lớp, thì có thể bạn sẽ sốc khi đến một số chủ đề này và bị thua tan tành.

Nhưng nó khó như vậy với tất cả mọi người. Bạn không được ban phát một khả năng đặc biệt thiếu khả năng học khoa học máy tính. Bạn chỉ phải tiếp tục đập đầu vào vấn đề giống như tất cả mọi người.

Bạn không nghi ngờ gì rằng tung 11 quả bóng cùng một lúc là khó với tất cả mọi người, đúng không? Thế nhưng, người ta vẫn làm được[4].

> *"Bạn biết cách đến Carnegie Hall không?"*
> *"Luyện tập!"*
>
> —Câu đùa không rõ nguồn gốc

Lặp, lặp, lặp, lặp.

Khi bạn ở trường, thực ra bạn còn khó khăn hơn khi đi làm. Ở trường, bạn liên tục học những thứ mới và, ngay khi bạn *vừa mới* nắm được một chủ đề, bạn đã chuyển sang học thứ khác cũng vô cùng thách thức. Bằng đại học là bốn năm liên tục chỉ vừa đủ bám víu vào lượng tài liệu mới không ngừng được trình bày cho bạn.

Và khi bạn cuối cùng lấy được bằng và có việc làm, ba tháng đầu tiên của công việc đó cũng là một trải nghiệm học hỏi tốc độ cao, vừa học vừa lo.

Nhưng sau ba tháng, bạn bắt đầu nắm được code. Và nó trở nên dễ hơn. Cảm giác bám víu bằng đầu móng tay bắt đầu giảm bớt. Và — tôi biết điều này có thể khó tin bây giờ — rốt cuộc công việc có thể trở nên dễ đến mức nhàm chán. Và rồi bạn sẽ tìm việc khác với những thách thức mới thú vị hơn.

3.4 Suy ngẫm về chương

- Đối chiếu *tư duy phát triển* với *tư duy cố định.*

[4] https://www.youtube.com/watch?v=cJsgM-3L38U

- Bạn đã từng bị giới hạn bởi tư duy cố định chưa? Một số kỹ thuật nào bạn có thể dùng để chuyển sang tư duy phát triển?
- Tại sao *sự kiên trì* là phẩm chất quan trọng mà các lập trình viên cần có?
- Sự khác biệt chính giữa giảng viên làm mọi thứ trông dễ dàng và sinh viên đang vật lộn với cùng tài liệu là gì? Sinh viên cần làm gì để cũng làm mọi thứ trông dễ dàng?

Chapter 4

Giải quyết vấn đề

Đây là ý tưởng được "đánh cắp" hoàn toàn từ cuốn sách *How to Solve It*[1] của George Pólya. Đó là cuốn sách về cách tiếp cận các bài toán toán học. Và vì khoa học máy tính về bản chất cũng là toán học, nó hoàn toàn áp dụng được!

Thực ra không hoàn toàn áp dụng được, và tôi chỉ nói vậy vì nghe hay. Nhưng có thể uốn nắn cho phù hợp khá dễ. Và tôi khuyên bạn nên đọc cuốn sách này.

Vậy nó là gì? Ngắn gọn và khá dễ thuộc:

1. **Hiểu** vấn đề
2. **Lên kế hoạch** giải quyết nó
3. **Viết code** cho giải pháp
4. **Nhìn lại** những gì có thể làm tốt hơn

Chỉ vậy thôi.

Và nếu bạn nghĩ về nó, đó thực chất chỉ là quy trình bốn bước để giải quyết hầu như bất kỳ vấn đề nào. Đèn phòng khách không sáng à? Bạn có thể giải quyết nó bằng các bước này.

> *"Toán học không phải là về con số. Nếu bạn nghĩ toán học là về con số, bạn có thể nghĩ rằng Shakespeare chỉ là về các từ ngữ. Bạn có thể nghĩ rằng khiêu vũ chỉ là về giày. Bạn có thể nghĩ rằng âm nhạc chỉ là về nốt nhạc. Toán học không phải là về con số. Toán học là về logic, về vẻ đẹp, về các kết nối, về cách bạn đi từ nơi này đến nơi khác."*
>
> —Cliff Stoll

Lập trình không phải là về viết code. **Bước 1, 2, và 4 không liên quan đến việc viết code**[2]. Những bước này đều diễn ra trong đầu bạn và trên giấy. Tôi sẽ lập luận rằng ***giải quyết các bài toán lập trình không liên quan đến máy tính***. Nghe có vẻ vô lý rõ ràng, nhưng đó chính là nơi hành động thực sự xảy ra! Đặc biệt ở bước 2, đưa ra *Kế hoạch*. Đó là phần khó. Đó là lý do bạn được trả lương cao. Ai cũng có thể gõ một chương trình vào sau khi vấn đề đã được giải quyết.

Bước 3, *Viết code*, đơn giản chỉ là ghi lại giải pháp. Công việc khó không phải là ghi lại giải pháp; công việc khó là đưa ra nó ngay từ đầu!

Ngoài ra, bạn khó có thể tiến hành tuyến tính qua các bước này. Bạn có thể phải quay lại và xem xét lại các bước trước từ thời gian này sang thời gian khác, nhưng hy vọng tiến trình tổng thể của bạn là theo hướng về phía trước.

Cuối cùng, với tư cách là sinh viên, bạn phải cưỡng lại sự thôi thúc tra cứu câu trả lời. Mục tiêu ở đây là để bạn tập luyện cơ giải quyết vấn đề (vì không ai sẽ thuê bạn làm lập trình viên mà không có những kỹ năng đó). Hầu như mọi bài toán mà giảng viên nào cũng có thể nghĩ ra đều đã được đề cập rộng rãi trên Internet hoặc có thể được giải bởi AI. Hãy nhớ rằng nhận được câu trả lời đúng không phải là điểm

[1] https://en.wikipedia.org/wiki/How_to_Solve_It

[2] Đôi khi thực ra có thể, nhưng chỉ để viết các chương trình nhỏ, dùng một lần, khám phá mang tính bằng chứng khái niệm.

mấu chốt; điểm mấu chốt là luyện tập giải quyết vấn đề để bạn có thể đưa ra câu trả lời cho bất kỳ bài toán nào được đặt ra cho bạn.[3]

Hãy cùng khám phá các bước.

4.1 Hiểu vấn đề

Tất cả mọi người, kể cả sinh viên, đều thích bỏ qua bước *Hiểu*. Các giảng viên khôn ngoan sẽ thêm một bài kiểm tra cần nộp trước khi bắt đầu viết code để xác minh bạn hiểu vấn đề[4]. Nhưng bạn cần tự ép buộc mình thực hiện bước này trước, bất kể thế nào.

Về mặt logic, nếu bạn không hiểu đầy đủ vấn đề, không có bước nào còn lại quan trọng. Bạn có thể đưa ra kế hoạch tốt nhất và viết code hoàn hảo cho nó, nhưng nếu bạn không hiểu vấn đề ngay từ đầu, bạn vừa viết code cho giải pháp hoàn hảo của sai vấn đề![5]

Và đó là lãng phí thời gian và tiền bạc của người thuê bạn, trong trường hợp tốt nhất. Hoặc, nếu bạn vẫn còn là sinh viên, là lãng phí thời gian bạn lẽ ra có thể dùng để học cho bài kiểm tra toán rời rạc sắp tới. Hay gì đó.

Vậy đừng cắt xén bước này — nó cực kỳ quan trọng về mặt nền tảng.

Bạn nên làm gì? Dưới đây là một số ý tưởng.

- Đọc vấn đề chậm và có phương pháp. Các mô tả này có xu hướng ngắn gọn và dễ gây nhầm lẫn. Chúng không dễ đọc, vì vậy đừng cố lướt qua. Đọc lại các câu nhiều lần đến khi hiểu ý nghĩa của chúng.
- Ghi chú. Đặc biệt ghi lại những điểm không nhất quán, bỏ sót, lỗi, và các câu hỏi còn tồn đọng.
- Viết lại vấn đề bằng lời của bạn. Đây là kỹ thuật rất hiệu quả giúp bạn tìm ra các khoảng trống trong sự hiểu biết của mình.
- Nghiên cứu khi cần thiết.
- Đặt câu hỏi làm rõ[6].

Bạn đã hoàn thành bước này khi bạn có thể dạy người khác vấn đề là gì. (Bạn không cần dạy giải pháp — chỉ cần vấn đề.)

Thật rất, *rất* khó để viết một mô tả đầy đủ về một vấn đề, và những cái bạn nhận được ở nơi làm việc hay ngay cả trong trường học chắc chắn sẽ còn thiếu sót. Không tin tôi? Hãy viết xuống các quy tắc của trò tic-tac-toe ("noughts and crosses" nếu bạn gọi như vậy) và tôi đảm bảo tôi sẽ tìm thấy điều gì đó bạn chưa viết xuống[7].

Do đó, bạn rất có thể sẽ phải hỏi các câu hỏi làm rõ.

> **Đến phần giai thoại!** Tôi đã tham gia một dự án viết phần front-end của một hệ thống và người khác viết back-end. Có một tài liệu rất cụ thể mô tả chính xác sự tương tác giữa hai phần.
>
> Một trong các tương tác liên quan đến việc truyền kết quả của một phép tính. Có một ví dụ trong tài liệu cho câu trả lời, ở dạng hex, của phép tính toán. Nó đại loại như:
>
> > Với đầu vào `"abc"` và `"xyz"`, kết quả sẽ là số `f319c2c6dcfb`.
>
> Tôi đã viết code (dễ vì pseudocode của thuật toán đã được cung cấp), và nó tính ra câu trả lời đúng. Tôi thêm code để truyền số 6 byte đó đến server và xong.
>
> Nhưng người viết server nhắn cho tôi nói tôi đang gửi rác.

[3] Tôi không nghĩ AI có thể giải quyết *tất cả* các vấn đề được đặt ra cho nó, nghĩa là bạn vẫn sẽ có việc làm, nhưng chúng có thể giải quyết các bài toán tương đối cơ bản thường được dùng trong chương trình khoa học máy tính. Đây là năm 2025 và chúng ta sẽ xem ghi chú chú thích này "già" đến mức nào.

[4] Tôi cũng cần phải khôn ngoan hơn.

[5] Một lần trên một trang web thử thách lập trình tôi đã viết code cho giải pháp hoàn hảo của *hai* bài toán không phải bài tôi cần giải. Tôi đã hiểu sai đến hai lần. Mất gấp ba lần thời gian cần thiết để đưa ra giải pháp thực sự.

[6] Đây thực ra là một phần trong mô tả công việc của bạn với tư cách là lập trình viên. Mọi người sẽ mong đợi và dựa vào bạn làm điều này tại nơi làm việc. Vì vậy đừng ngại làm; hãy ngại *không* làm.

[7] Bạn chưa nói cách chọn ai đi trước, liệu có thể chơi với ba người không, hay là "X" của tôi không thể chiếm nhiều hơn một ô. Ví dụ vậy.

> Chúng tôi trao đổi qua lại, và hóa ra họ nghĩ spec có nghĩa tôi sẽ gửi một chuỗi chứa những chữ số hex đó, còn tôi nghĩ spec có nghĩa tôi sẽ gửi các byte thô của kết quả. Spec không giúp ích gì — nó mơ hồ và cả hai chúng tôi đều không nhận ra điều đó.
>
> Cách dễ nhất là tôi chuyển số sang chuỗi và gửi đi, vì vậy tôi thêm dòng code đó và vấn đề biến mất. Nhưng chúng tôi có thể đã phát hiện ra sớm hơn nếu xem xét spec kỹ hơn.

4.2 Đưa ra kế hoạch

> *"Lập trình là nghệ thuật nói với một người khác điều mình muốn máy tính thực hiện."*
>
> —Donald Knuth

Đã đến lúc bắt đầu viết code? **KHÔNG!** Chưa phải lúc, bạn ơi![8]

Bước này, đưa ra kế hoạch, là nơi lập trình thực sự diễn ra. Như tôi đã nói trước đó, đây là điều làm cho công việc trở nên khó khăn, và là lý do nó được trả lương tốt. Để trở thành một lập trình viên tàm tạm, bạn cần xuất sắc trong việc đưa ra kế hoạch vững chắc.

Trước tiên bạn phải *Hiểu* vấn đề đủ để có thể giải thích cho người khác. Hoặc ít nhất bạn *nghĩ* bạn hiểu. Bạn có thể học thêm sau. Nhưng hiểu vấn đề không giống như biết cách giải quyết nó.

Nếu vấn đề thực sự đơn giản và quen thuộc, bạn có thể nghĩ ra kế hoạch gần như ngay lập tức. Các lập trình viên có kinh nghiệm đối mặt với các nhiệm vụ quen thuộc không cần bỏ nhiều thời gian lập kế hoạch khi đã hiểu đầy đủ vấn đề.

Nhưng các lập trình viên có kinh nghiệm đối mặt với các nhiệm vụ không quen thuộc *cần* bỏ thời gian vào đó. Dù bạn giỏi đến đâu — nếu bạn chưa gặp vấn đề này, bạn sẽ cần phải lập kế hoạch cho giải pháp.

Bút chì và giấy thực sự có thể hữu ích cho bước này. Dưới đây là một số điều cần suy nghĩ.

- Dữ liệu sẽ chạy qua hệ thống và được biến đổi như thế nào, từ đầu vào đã biết đến đầu ra mong muốn?
- Trong những phép biến đổi đó, các hệ thống con nào sẽ thực hiện mỗi bước?
- Bạn cần những công nghệ nào để thực hiện các bước này?
- Những điều không biết đã biết là gì?

Một điều lớn cần lưu ý là chúng ta đang nói về việc phân tách vấn đề thành các thành phần con của nó. Cách làm điều này không phải lúc nào cũng rõ ràng, mặc dù bạn càng có nhiều kinh nghiệm thì nó càng dễ hơn.

Một mẹo bạn có thể dùng là nghĩ, "Dự án này sẽ dễ nếu chỉ cần dữ liệu đầu vào ở *dạng này*." Sau đó xem liệu bạn có thể nghĩ ra đoạn code nào chuyển đổi dữ liệu sang dạng đó không. Chúng ta sẽ đi sâu hơn ở chương sau.

Một lợi ích khác của việc chia dự án thành các phần nhỏ hơn là nó có thể tự nhiên gợi ý cách chia code thành các hàm hoặc lớp nhỏ hơn. Điều này làm cho code dễ bảo trì và đọc hơn.

Bạn sẽ phải thực hiện một số nghiên cứu trong giai đoạn này để tìm hiểu những công cụ nào bạn có trong tay. Hãy mong đợi điều đó.

Thực sự phổ biến trong giai đoạn lập kế hoạch khi nhận ra rằng bạn đã không hiểu đầy đủ vấn đề. Khi điều này xảy ra, hãy quay lại giai đoạn *Hiểu* để làm rõ, rồi quay trở lại lập kế hoạch.

Bạn biết mình đã xong lập kế hoạch khi bạn có thể mô phỏng việc chạy trên giấy và trong đầu và bạn tự tin là nó hoạt động. Và bạn có thể viết pseudocode cấp cao. Chia sẻ kế hoạch của bạn với vài con vịt cao su[9] để xem nó có vững chắc không.

[8] Tính đến năm 2025, tôi làm việc tại Đại học Bang Oregon. Hãy tiến lên, Beavs!

[9] *Rubber ducking* là việc chia sẻ ý tưởng với một con vịt cao su thực sự hoặc đại diện thay thế, có thể thực ra là một người. Nó giúp bạn làm rõ suy nghĩ và đạt được đột phá trong giải quyết vấn đề. Ngoài ra, Ducks là đội bóng bầu dục của Đại học Oregon, đối thủ lâu năm của Bang Oregon. Boo Ducks!

4.3 Viết code cho giải pháp

Đây là phần dễ! Bạn phải dịch các kế hoạch pseudocode thành code.

Nếu bạn đã hiểu vấn đề và đưa ra kế hoạch vững chắc, code sẽ hoạt động. Thậm chí có thể ngay từ lần đầu, nếu bạn rất may mắn[10].

Nếu bạn *không* hiểu vấn đề và không đưa ra kế hoạch, thì đây không phải là phần dễ. Bạn sẽ thấy vô số rắc rối, và có thể không hoàn thành dự án. *Đó* là mức độ quan trọng của việc hiểu và lập kế hoạch!

Tất nhiên, chúng ta chỉ là người và sẽ gõ sai và mắc lỗi ngớ ngẩn, nhưng điều đó *dễ* debug hơn nhiều so với việc bạn có kế hoạch tồi, hoặc tệ hơn, hiểu vấn đề sai. Ôi thôi!

Những phần khó của việc viết code là:

- Biết cú pháp ngôn ngữ.
- Biết những thư viện nào có sẵn.
- Không mắc lỗi vặt.
- Theo sát kế hoạch.

Chúng ta sẽ nói về cách học ngôn ngữ sau, và hai điểm bullet cuối có thể được giải quyết bằng cách cẩn thận hơn, sử dụng thứ gì đó như lập trình theo cặp[11], hoặc nhờ AI hỗ trợ.

Trong khi viết code, bạn có thể tìm thấy chỗ mà kế hoạch của mình không hoạt động. Điều này xảy ra khá thường xuyên. Khi xảy ra, hãy quay lại giai đoạn lập kế hoạch, sửa kế hoạch, rồi quay lại giai đoạn viết code để thực hiện nó.

Một lần nữa, giai đoạn này được cho là tương đối dễ. Nếu bạn đang gặp khó khăn khi cố làm cho nó hoạt động ngoài việc chỉ sửa lỗi cú pháp (tức là có gì đó sai về cấu trúc hơn), sự hiểu biết hoặc kế hoạch của bạn có thể có lỗi. Bạn nên quay lại giai đoạn lập kế hoạch để sửa, và có thể quay lại giai đoạn hiểu, nếu cần.

Bạn đã xong giai đoạn này khi code hoạt động và vượt qua tất cả các bài kiểm tra.

4.4 Nhìn lại để cải thiện

> "Lập trình là một nghề thủ công. Hãy tự hào về công việc của bạn."
>
> —Chính tác giả

Cuối cùng nhưng chắc chắn không kém quan trọng, hãy nhìn lại và nhìn với con mắt phê phán vào những gì bạn đã làm. Vâng, nó hoạt động và vượt qua tất cả các bài kiểm tra. Nhưng nó có thanh lịch như có thể không? Nó có dễ đọc và dễ bảo trì như có thể không? Có những chỗ nào code mong manh và có thể thất bại với đầu vào không mong đợi không?

Dù kỹ năng của bạn ở mức nào, luôn còn chỗ để cải thiện. Và một trong những cách hàng đầu chúng ta học là nhìn lại code tệ hại mình đã viết và xem mình có thể làm tốt hơn như thế nào.

Code review thật tuyệt vời nếu bạn có thể thuyết phục ai đó bỏ thời gian làm điều đó cho bạn. Họ sẽ đưa ra gợi ý về những thứ bạn có thể cải thiện, và bạn có thể sửa chúng ngay bây giờ và ghi nhớ cho lần sau. Và bạn có thể không đồng ý và không thực hiện những sửa chữa đó; điều đó cũng ổn.

Một lần nữa, chúng ta có thể tận dụng AI để hỗ trợ điều này. Khi bạn đã giải quyết được vấn đề và có nó hoạt động, hãy hỏi AI về các gợi ý cải thiện[12] và xem có điều gì đáng theo không. Kỹ thuật này hiệu quả với các đoạn code nhỏ, không phải các dự án lớn, nhưng điều đó làm cho nó trở thành một trợ lý tuyệt vời cho công việc đại học.

> **Nhưng, rất quan trọng đối với sinh viên đại học, bạn cần giải quyết vấn đề trước**, và chỉ sau đó mới nhờ AI giúp bạn cải thiện nó. Mục tiêu của bạn trong luyện tập ở trường (giống như luyện tập ở phòng gym) là được luyện tập với phản hồi, không phải để người khác làm việc cho bạn.

[10] Điều này rất hiếm khi xảy ra với tôi. Khi xảy ra, tôi nhanh chóng nhìn ra cửa sổ để đảm bảo bầu trời trong xanh và tôi sắp không bị sét đánh. Và tôi mua vé số. Đó chắc chắn là lúc vận may của tôi cạn kiệt.

[11] https://en.wikipedia.org/wiki/Pair_programming

[12] Đảm bảo giảng viên và/hoặc người sử dụng lao động của bạn cho phép điều này.

Code có thể tệ theo nhiều cách. Nó có thể có lỗi. Nó có thể không hiệu quả. Nó có thể có định dạng kém. Và nó đơn giản là không thể đọc được. Hãy nhớ rằng để code của bạn có thể bảo trì được, nó cần phải dễ hiểu đối với những người khác. Hãy làm nó trông gọn gàng. Trình biên dịch có thể hài lòng với code không thể đọc được, nhưng con người không nên chấp nhận điều đó.

Bạn không bao giờ hoàn thành giai đoạn này. Thực tế, bạn xong khi bạn bỏ cuộc và quyết định đã học đủ về việc cải thiện code. Nhưng bạn có thể sẽ không tìm ra *Giải Pháp Tối Thượng Cho Vấn Đề Mãi Mãi* vì bạn vẫn đang xây dựng kỹ năng của mình suốt đời.

Dù vậy, đây là giai đoạn nơi **rất nhiều** việc học xảy ra. Đây là nơi bạn có thể xây dựng hiệu quả kỹ năng của mình với nỗ lực tương đối thấp. Và bạn sẽ thiệt thòi nếu không dành chỉ vài phút sau một dự án để tận dụng điều đó.

Ở nơi làm việc, điều này có dạng là một *post-mortem*, nơi những người tham gia dự án đã hoàn thành nhìn lại và nghiên cứu những gì đã đúng và sai.

4.5 Suy nghĩ như kẻ phản diện

Khi giải quyết vấn đề, tôi muốn bạn suy nghĩ như một kẻ phản diện; đó là, nghĩ như một người sẽ lạm dụng hệ thống mà bạn đang thiết kế và xây dựng.

Ví dụ, một hàm căn bậc hai thực tế có thể được kiểm tra kỹ lưỡng. Cho nó một số bình phương hoàn hảo, một số bình phương không hoàn hảo, một số phân số, v.v. Hoạt động hoàn hảo. Bạn sẽ không truyền số âm vào, đúng không? Điều đó thật ngớ ngẩn. Nhưng bạn biết ai sẽ làm vậy không? Một kẻ phản diện!

> **Tôi từng ghé thăm một cửa hàng trực tuyến cho phép bạn đặt số lượng âm.** Trang thanh toán nói họ sẽ ghi có vào tài khoản của tôi hàng chục nghìn đô la.
>
> Tuy nhiên tôi không bao giờ đủ can đảm để thanh toán, vì tôi không muốn chịu trách nhiệm vận chuyển tất cả sản phẩm đó cho họ.
>
> Hơn nữa, tôi không phải kẻ phản diện thực sự... tôi chỉ đang nghĩ như vậy thôi!

Hãy chuẩn bị cho những điều bất ngờ về dữ liệu mà code của bạn sẽ nhận. Hãy chuẩn bị cho các tác nhân độc hại cung cấp dữ liệu nhằm cố gắng truy cập trái phép hoặc thao túng hệ thống theo những cách không mong muốn. Hãy kiểm tra những thứ đó trong code của bạn.

Điều này áp dụng cho tất cả các giai đoạn từ hiểu đến nhìn lại.

> **Một người quen của tôi ở đại học đã làm việc cho quân đội** xem xét kế hoạch cho những thứ như xe tăng chưa đi vào sản xuất. Công việc của anh ấy là suy nghĩ như kẻ phản diện và đặt câu hỏi như, "Làm thế nào bạn sẽ đưa cờ lê vào giữa những ống đó để vặn chặt bu lông đó?"

Suy nghĩ như kẻ phản diện không chỉ có thể phát hiện ra những vấn đề bạn có thể không xem xét đến, mà còn dẫn bạn đến sự hiểu biết sâu sắc hơn về dự án, tạo ra code bền vững và dễ bảo trì hơn.

4.6 Áp dụng trong phỏng vấn

Quy trình bốn bước từ chương này chính xác là những gì bạn nên sử dụng trong các thử thách code khi phỏng vấn.

Thấy đấy, những bài toán phỏng vấn này khá hiểm. Chúng hoàn toàn không có câu trả lời rõ ràng.

Và tại sao lại như vậy? Có phải vì bạn chưa học đủ không? **Không.** Hoàn toàn không phải vậy. Dù bạn học bao nhiêu, người phỏng vấn vẫn sẽ nghĩ ra một câu hỏi không có câu trả lời rõ ràng với bạn, những kẻ tàn ác đó.

Và tại sao họ làm vậy? Họ muốn bạn thất bại? Không hề. *Họ muốn thấy bạn áp dụng kỹ năng giải quyết vấn đề.*

> **Điều này không phải là phổ quát, nhân tiện.** Có hàng triệu phong cách phỏng vấn, và một số trong đó thực sự chỉ muốn xem bạn nhận được bao nhiêu câu trả lời đúng. Nhưng tôi sẽ lập luận

> rằng họ không phỏng vấn tối ưu. Và hơn nữa, tôi sẽ lập luận rằng cách duy nhất để có câu trả lời đúng là áp dụng các bước giải quyết vấn đề, vì vậy bạn cũng có thể làm như vậy trong mọi trường hợp.

Thật tự nhiên khi bạn đang căng thẳng trong buổi phỏng vấn và đối mặt với một bài toán ban đầu có vẻ không thể, bạn đóng băng như con nai trước đèn pha xe. Tất cả kiến thức của bạn đột nhiên biến mất trong đám sương mù và bạn biết mình sẽ không bao giờ có được công việc này và—

ĐỪNG HOẢNG LOẠN. Hãy tự nhắc mình những từ này: "Điều duy nhất quan trọng bây giờ là *hiểu vấn đề*." Quên giải pháp đi. Điều đó không quan trọng. Viết code? Không quan trọng. Chỉ cần tập trung vào bước một: hiểu vấn đề.

Có hai lý do chính cho điều này. Một là người phỏng vấn hy vọng bạn sẽ bắt đầu từ đó. (Và điều đó nên là lý do đủ.) Nhưng còn một lý do khác là bằng cách bắt đầu với việc hiểu vấn đề, não bạn sẽ khởi động lại và tự động bắt đầu nghĩ ra các chiến lược để giải quyết vấn đề... và đó là bước hai của framework giải quyết vấn đề! Bạn đã đi đúng hướng rồi.

Hãy cố nghĩ (bạn, kẻ phản diện ơi) về bất cứ điều gì mơ hồ trong mô tả vấn đề. Hãy hỏi câu hỏi để làm rõ những điều đó. Giới hạn của đầu vào là gì? Giới hạn của đầu ra là gì? Bạn làm gì trong điều kiện lỗi? Có thể gợi ý một ví dụ đầu vào và đầu ra và xác minh với người phỏng vấn rằng bạn hiểu đúng.

Điều này cho người phỏng vấn thấy bạn chú ý đến chi tiết, một đặc điểm quan trọng cần có. Và nó cũng cho họ thấy bạn bắt đầu dự án bằng cách đưa ra những lựa chọn có cân nhắc, chủ ý.

Và nghe này: đối với nhiều người phỏng vấn, việc thấy bạn tiếp cận vấn đề một cách có hệ thống thực sự quan trọng hơn việc bạn đưa ra câu trả lời đúng. Và ở chiều ngược lại, không thể hiện kỹ năng giải quyết vấn đề khi đưa ra câu trả lời có thể khiến bạn thất bại, dù câu trả lời có đúng!

> **Khi tôi phỏng vấn tại Activision**, có hai câu hỏi tôi đã không trả lời đúng. Nhưng tôi đã cố gắng hết sức để giải quyết chúng thành tiếng, thể hiện cách tôi tiếp cận vấn đề. Tôi được nhận.
>
> (Những câu hỏi mà tôi đã hỏng là: "Cách nhanh nhất để đảo ngược các bit trong một byte là gì?" và "Tối ưu hóa phép tính này xây dựng lưới khoảng cách giữa tất cả các cầu thủ bóng đá trên sân.")

Vì vậy hãy đi qua toàn bộ quy trình với người phỏng vấn. Và đừng quên bước phản ánh cuối cùng! Bạn sẽ làm gì tốt hơn? Giải pháp có thể được cải thiện như thế nào? Những tính năng nào có thể được thêm vào trong tương lai? Người phỏng vấn thích những thứ đó, và bạn thích làm người phỏng vấn vui lòng, đúng không?

4.7 Chi phí theo giai đoạn

Một lưu ý liên quan đến framework giải quyết vấn đề là *chi phí thay đổi phần mềm tăng theo cấp số nhân khi bạn càng tiến xa trong quá trình phát triển.*

Khi bạn đang ở giai đoạn *Hiểu*, các thay đổi thực sự rẻ. Chúng miễn phí. Bạn thậm chí chưa nghĩ ra kế hoạch, và bạn chỉ đang phác thảo ý tưởng.

Khi đến giai đoạn *Kế hoạch*, các thay đổi vẫn khá rẻ. Không miễn phí — nếu bạn cần thực hiện thay đổi, nó có thể ảnh hưởng đến các phần khác của kế hoạch, và những phần đó sẽ cần được lập kế hoạch lại, hoặc có thể cần hiểu thêm.

Tiếp theo, khi đến giai đoạn *Viết code*, bây giờ các thay đổi bắt đầu gây đau. Có thể một thay đổi đòi hỏi phải bỏ đi và làm lại code trị giá hàng nghìn hoặc hàng triệu đô la chi phí lập trình viên. Các công ty thực hiện các thay đổi như thế này liên tục, dù vậy. Họ chỉ phân tích chi phí-lợi ích và quyết định xem có đáng không.

Cuối cùng, sau khi code được triển khai, bây giờ các thay đổi *thực sự* tốn kém. Không chỉ chúng ta phải lập kế hoạch lại, lập trình lại, xây dựng lại, kiểm tra lại và triển khai lại một loạt code, mà còn khách hàng ghét thực tế là chúng ta yêu cầu cập nhật, vì vậy chúng ta có tất cả các loại chi phí phụ ẩn liên quan đến thay đổi.

Từ góc độ sinh viên, bạn không lo nhiều về số tiền dự án phần mềm của bạn sẽ tốn cho công ty. Bạn lo lắng hơn rằng bạn sẽ có đủ thời gian để hoàn thành nó (cùng với mọi thứ khác — giảng viên của bạn có

biết bạn có nhiều hơn một lớp học không?) với điểm số tử tế.

Vì vậy, những gì bạn cần làm là tập trung sự chú ý vào *Hiểu* và *Kế hoạch* nơi các thay đổi rẻ *về mặt thời gian.* Điều này sẽ cho bạn kết quả tốt nhất một cách nhanh chóng và hiệu quả (và hy vọng trước hạn chót) với ít đau đớn lập trình nhất.

4.8 Suy ngẫm về chương

- Bốn giai đoạn của giải quyết vấn đề là gì?
- Giai đoạn nào liên quan đến việc ngồi ở bàn phím viết code?
- Làm thế nào bạn có thể làm cho giai đoạn *Viết code* khó hơn mức cần thiết?
- Hậu quả của một sai lầm không bị phát hiện trong giai đoạn *Hiểu* là gì?
- Đối với mỗi giai đoạn, làm thế nào bạn biết khi nào mình đã hoàn thành nó?
- Điều gì xảy ra nếu bạn bỏ qua giai đoạn *Nhìn lại*?
- Beej có ý gì khi nói “suy nghĩ như kẻ phản diện”?

Chapter 5

Phân tách vấn đề

> *"Có những trò chơi bên ngoài trò chơi."*
>
> —Stringer Bell, *The Wire*

Nếu các bước giải quyết vấn đề (*Hiểu*, *Kế hoạch*, *Viết code*, *Nhìn lại*) có phần tiếp theo, chương này sẽ là nó.

Những bước đó thực sự giúp bạn vượt qua mọi vấn đề, nhưng hóa ra những vấn đề đó tồn tại như các vấn đề fractal bên trong vấn đề bên trong vấn đề.

Ví dụ, có thể bạn muốn đóng một cái bàn. Đó là toàn bộ vấn đề:

- Đóng một cái bàn

Thế nhưng có thể chưa đủ nếu bạn chưa đóng nhiều bàn. Vì vậy bạn phân tách vấn đề thành các bài toán con phụ thuộc nhau.

- Đóng một cái bàn
    - Gắn chân vào mặt bàn
        - Làm mặt bàn
        - Làm chân bàn

Và có thể vẫn chưa đủ. Làm thế nào để làm chân bàn? Làm thế nào để làm mặt bàn?

- Đóng một cái bàn
    - Gắn chân vào mặt bàn
        - Làm mặt bàn
            - Đánh nhám bề mặt
                - Dán viền vào mặt bàn
                    - Cắt mặt bàn chính
                        - Học sử dụng cưa bàn
                    - Cắt viền
        - Làm chân bàn
            - Cắt chân bàn
            - Tiện chân bàn trên máy tiện
                - Học sử dụng máy tiện

Và cứ tiếp tục như vậy. *Chúng ta tiếp tục phân tách vấn đề cho đến khi chúng ta thu được bước đủ nhỏ để biết mình có thể thực hiện được.*

Khi bạn mới bắt đầu, bạn có thể phải rút gọn vấn đề xuống từng dòng code đơn lẻ. Các lập trình viên có kinh nghiệm thường không cần phân tách đến tận mức đó vì họ đã thành thục trong các bước con.

Ví dụ, một thợ mộc với kinh nghiệm vừa phải có thể chỉ cần phân tách việc đóng bàn thành bộ bước thứ hai của chúng ta ở trên, và không cần đi vào chi tiết như vậy.

Như mọi thứ khác, phân tách vấn đề là một kỹ năng, và bạn sẽ giỏi hơn khi luyện tập.

Khi phân tách vấn đề, hãy nhớ lại cân nhắc trước đó của chúng ta: "Bài toán này sẽ dễ nếu dữ liệu đầu vào ở *dạng này*." Đó là gợi ý rằng bạn nên tách ra một bài toán con chuyển đổi dữ liệu đầu vào sang dạng đó, từ đó làm cho bài toán trở nên dễ.

Và khi bạn có một bài toán con, hãy giả vờ đó là toàn bộ vấn đề, chỉ trong một chút. Tập trung chặt vào nó và xem liệu bạn có thể giải quyết nó một cách độc lập không. Nếu không, hãy tự hỏi điều gì sẽ làm cho nó dễ giải quyết, và tách ra thành một bài toán con.

Lặp lại.

Bạn càng luyện tập phân tách vấn đề và viết code cho giải pháp, bạn sẽ càng giỏi hơn. Sớm thôi bạn sẽ không cần phân tách vấn đề xa đến mức bạn cần trước đây, và, giống như một chuyên gia, bạn sẽ bắt đầu nhận ra các mẫu mà bạn có thể tái sử dụng.

Tuy nhiên, không phải lúc nào cũng rõ ràng *cách* phân tách một vấn đề.

Một kỹ thuật là tưởng tượng một hiện thân vật lý của thứ bạn đang cố code. (Ví dụ, bạn đang viết một thuật toán sắp xếp? Hãy tưởng tượng một đống khối chữ cái trên bàn và bạn phải sắp xếp chúng.)

Và sau đó, để đẩy xa hơn, hãy tưởng tượng bạn đang dạy một người bạn không có kiến thức kỹ thuật cách giải quyết nó. Bạn sẽ mô tả các bước như thế nào? Các điều kiện? Khi nào họ xong?

Nếu bạn có thể sắp xếp vật lý các cuốn sách trên kệ (dù "sách" là gì), bạn có thể viết một thuật toán để làm chính xác điều tương tự. Bạn chỉ cần phân tách các bước.

5.1 Pseudocode

Một trong những công cụ lớn mà các lập trình viên dùng để khám phá ý tưởng là viết pseudocode. Đây là "code dành cho con người". Máy tính không thể đọc được nó. (Mặc dù một số người có thể lập luận rằng Python khá gần với pseudocode.)

Nhưng bạn có thể dùng nó để phác thảo các bước của một thuật toán hoặc quy trình để kiểm tra độ hợp lý hoặc chỉ khám phá cách bạn có thể làm điều gì đó.

Bạn có thể viết một số pseudocode để chèn một giá trị vào một danh sách đã được sắp xếp.

```
find correct spot in list
insert the value there
```

Nhưng điều đó không thực sự mô tả đủ. Chúng ta có thể phải phân tách tiếp.

```
find correct spot in list
    → find first entry larger than the new one

insert the value there
    → shift all the higher values to the right
    → insert the new value in the newly-opened spot
```

Đang tiến triển.

```
find correct spot in list
    find first entry larger than the new one
        → loop through items, stop when you find a larger one

insert the value there
    shift all the higher values to the right
    insert the new value in the newly-opened spot
```

Và bây giờ nó đang trở nên rõ ràng hơn một chút.

```
find correct spot in list
    find first entry larger than the new one
        loop through items, stop when you find a larger one
        → record the index before it

insert the value there
    shift all the higher values to the right
    insert the new value in the newly-opened spot
        → set the list item at the index to the new value
```

Và chúng ta đang đến rất gần mức có thể dịch pseudocode sang code thực tế. Có thể vẫn chưa rõ ràng cách chúng ta sẽ dịch chuyển tất cả các giá trị sang phải, và chúng ta nên phân tách điều đó thêm một chút.

Đôi khi các lập trình viên thêm pseudocode vào code thực của họ dưới dạng comment và triển khai code thực bên dưới chúng.

Đây là công cụ mạnh mẽ để sử dụng trong giai đoạn *Kế hoạch*. Nó thực sự có thể giúp củng cố suy nghĩ của bạn về quy trình tổng thể.

5.2 Bằng chứng khái niệm

Điều gì sẽ xảy ra nếu bạn đã phân tách bài toán lớn hơn thành các bài toán con nhỏ hơn, nhưng đơn giản bạn không biết liệu một trong những thứ đó có khả thi để thực hiện không.

Ví dụ, "Bạn có thể render một hình ảnh vào HTML canvas và sau đó lưu hình ảnh đó trực tiếp vào thư viện ảnh trên điện thoại di động không?" Có thể bạn chưa bao giờ làm điều đó, và bạn không biết liệu điện thoại và công nghệ web có khả năng đó không.

Một cách chắc chắn để tìm hiểu là viết code bằng chứng khái niệm.

Vì vậy bạn tạo một trang web, thêm canvas, vẽ thứ gì đó đặc trưng lên đó, như một hình chữ nhật, và sau đó thêm code để tải xuống khi nhấn nút.

Điều này trước đây đòi hỏi rất nhiều đọc sách và, sau đó, tìm kiếm trên web. Và thường vẫn còn như vậy. Nhưng phổ biến hơn bây giờ chúng ta dựa vào AI[1] để trả lời các câu hỏi "nó có thể và làm thế nào", và thậm chí đưa ra một số code bằng chứng khái niệm.

Khi bạn có code hoạt động, bạn biết hai điều:

1. Nó hoạt động!
2. Cách viết code để thực hiện điều đó.

Thường thì code bằng chứng khái niệm chưa sẵn sàng cho môi trường sản xuất, nhưng tạo thành cốt lõi của những gì bạn sẽ cuối cùng bàn giao.

Một cách sử dụng khác của code bằng chứng khái niệm là để trình bày cho mọi người thấy sản phẩm hoàn thiện sẽ trông như thế nào hoặc hoạt động như thế nào. Đôi khi người ta viết code một triển khai giả, nơi chỉ một phần nhỏ của giao diện người dùng hoạt động nhưng người xem có thể nắm được ý tưởng về cách phần mềm cuối cùng sẽ hoạt động.

Thường thì phần lớn code bạn đã viết cho bằng chứng khái niệm sẽ bị bỏ đi, và bạn có thể cảm thấy kháng cự khi loại bỏ công việc đó. Nhưng đừng lo lắng về nó. Phần quan trọng của bằng chứng khái niệm là kiến thức thu được trong khi làm việc, không phải bản thân công việc.

> *"Hãy lên kế hoạch vứt bỏ một cái; dù sao bạn cũng sẽ làm vậy."*
>
> —Fred Brooks, *The Mythical Man-Month*

[1] Một lần nữa, nếu được phép trong môi trường làm việc hoặc trường học của bạn.

5.3 Suy ngẫm về chương

- Tại sao việc phân tách một vấn đề lại quan trọng?
- Bạn phải phân tách vấn đề đến đâu trước khi có thể bắt đầu viết code?
- Pseudocode là gì và tại sao chúng ta viết nó?
- Bằng chứng khái niệm là gì và tại sao nó hữu ích?

Chapter 6

Công Cụ Phù Hợp Cho Công Việc

"Ngôn ngữ lập trình nào là tốt nhất?"

Đây là câu hỏi phổ biến của những người mới bắt đầu học lập trình. Và thường thì mọi người đều có ý kiến riêng, đặc biệt khi họ mới vào nghề.

Nhưng bất kỳ ai đã hoạt động trong lĩnh vực này một thời gian sẽ nói với bạn rằng không có ngôn ngữ lập trình "tốt nhất" nếu không biết bối cảnh của vấn đề bạn đang giải quyết.

Tôi không biết ai lại tuyên bố rằng lập trình Bash[1] là ngôn ngữ lập trình tốt nhất. Nhưng chắc chắn có những bài toán mà thực sự nó lại là giải pháp tốt nhất!

Hãy thử dùng phép so sánh xem! Cái gì tốt hơn: tua vít hay búa? Đây là câu hỏi đánh lừa! Cả hai đều là công cụ tốt nhất, nhưng chỉ trong bối cảnh công việc mà chúng được thiết kế cho.

Các ngôn ngữ lập trình khác nhau có điểm mạnh và điểm yếu khác nhau. Nhiệm vụ của bạn với tư cách là một lập trình viên là chọn búa cho đinh và tua vít cho vít, và phải thành thạo cả hai công cụ đó. "Tôi không quen với máy cưa bàn và không thực sự thích nó, nên tôi sẽ dùng cái búa này để cắt tấm ván ép" không phải là giải pháp mà sếp của bạn đang tìm kiếm.

> **Người hướng dẫn lặn biển của tôi đã cho tôi một lời khuyên.** Tôi đang xem xét mua một số thiết bị và đang phân vân giữa đồ tiêu chuẩn và thiết bị kiểu DIR[a].
>
> Ông nói, "Hãy trở thành người thợ lặn giỏi nhất trong bất kỳ thiết bị nào. Tôi không quan tâm nếu anh đang lặn với bộ collar ngựa từ những năm 1960 và bình J-valve."
>
> [a] https://en.wikipedia.org/wiki/Doing_It_Right_(scuba_diving)

Những lập trình viên giỏi nhất có rất nhiều công cụ trong tay, và họ biết cách sử dụng chúng.

Vì vậy, khi bạn thấy mình nói, "Ngôn ngữ này tệ và tôi ghét nó so với ngôn ngữ kia mà tôi yêu thích!" hãy nghĩ đến những trường hợp sử dụng của ngôn ngữ mà bạn ghét đó. Bởi vì nó được tạo ra với một lý do, và việc xác định lý do đó có thể giúp bạn biết khi nào bạn *nên* dùng nó.

Và vì bạn đang nóng lòng muốn biết, ngôn ngữ yêu thích của tôi là Rust.

Hay Python. Hay C. Hay JavaScript. Hay SQL. Hay AWK. Hay Bash. Hay... thật ra còn tùy thuộc vào vấn đề tôi đang giải quyết!

6.1 Hãy Có Chính Kiến

Tôi vừa nói xong rằng bạn nên đối xử công bằng với tất cả các ngôn ngữ và không thiên vị phải không?

Đúng vậy. Nhưng tôi thề điều này không mâu thuẫn với điều đó.

Tại sao cần có chính kiến? Đó là vì bạn nên cân nhắc kỹ lưỡng và có đủ thông tin khi đưa ra lựa chọn. Và một cách tốt để làm điều đó là có chính kiến cùng với lý do cụ thể. Nó buộc bạn phải đào sâu vào

[1] https://en.wikipedia.org/wiki/Shell_script

điểm mạnh và điểm yếu của các công nghệ khác nhau, và điều đó giúp bạn chọn ra công cụ tốt nhất trong bộ công cụ của mình cho một công việc cụ thể.

Điều này không có nghĩa là bạn nên cư xử thô lỗ. Đây không phải là chuyện ai đúng hay ai sai — không có gì trong số này được khắc vào đá bất biến. Mọi người đều vừa đúng vừa sai cùng lúc. Rất kiểu Schrödinger của chúng ta.

Bạn nên có chính kiến về những công cụ mình chọn. Từ tất cả mọi thứ trong tay, bạn chọn ngôn ngữ *này* và hệ thống build *này* để sử dụng. Bạn chọn editor *này* hoặc thư viện *kia*. Bạn nên có lý do giải thích *tại sao* chúng là tốt nhất, dù có nhiều ngôn ngữ hay công cụ khác cũng sẽ đáp ứng được yêu cầu.

Bạn nên có chính kiến về những cấu trúc dữ liệu mình chọn. Lập trình là một công việc sáng tạo — bạn có thể biến tất cả mọi thứ thành máy Turing[2] nếu muốn và giải quyết vấn đề theo cách đó, nhưng có rất nhiều lựa chọn khác. Hãy chọn những lựa chọn tốt nhất và có lý do giải thích *tại sao* bạn chọn chúng trong số tất cả các lựa chọn khác có sẵn.

Bạn nên có chính kiến về những thuật toán mình chọn. Mergesort chẳng hạn? Cứ dùng đi. Nhưng thực ra có những lúc insertion sort tầm thường lại tốt hơn! Binary search? Tại sao dùng nó nếu linear search là đủ tốt cho công việc? Hãy có lý do giải thích *tại sao* bạn chọn thuật toán đó khi bạn có thể đã dùng một thuật toán khác.

Khi quyết định, bạn có thể đang xem xét những yếu tố như hiệu suất, hay khả năng đọc của code, hay thời gian phát triển cần thiết, hay chi phí công cụ, hay công nghệ mà codebase hiện tại đang dùng, hay tất cả các yếu tố khác.

Bạn luôn phải sẵn sàng học hỏi thêm. Dù bạn chọn gì với lý do của mình, một nửa Internet sẽ không đồng ý với bạn. Và bạn biết không? Chính bạn trong tương lai thậm chí cũng có thể không đồng ý với bạn!

Khi học hỏi thêm, bạn sẽ tìm hiểu về những trường hợp cụ thể mà một công nghệ nào đó là tốt nhất, trong khi trước đây bạn nghĩ nó sẽ không phù hợp. Hãy cởi mở để học hỏi và thay đổi khi thích hợp. Điều này có thể là điều gì đó lớn như việc lựa chọn framework cho sản phẩm định nghĩa toàn bộ công ty của bạn, hoặc điều gì đó nhỏ như việc các biến mảng có nên đặt tên số nhiều hay không.

Khi trưởng thành, các lựa chọn của bạn sẽ thay đổi. Và, miễn là bạn có lý do đằng sau đó, đó là điều tốt.

6.2 Suy Ngẫm Về Chương

- Mô tả quan điểm điển hình của một lập trình viên có kinh nghiệm về ngôn ngữ lập trình nào là tốt nhất.
- Tại sao cần có chính kiến về công nghệ?
- Những yếu tố nào cần xem xét khi quyết định công nghệ nào là công cụ phù hợp cho công việc?
- Những yếu tố nào cần xem xét khi quyết định công nghệ nào là công cụ phù hợp cho công việc mà không được đề cập trong chương này?
- Tại sao việc tiếp tục học hỏi lại quan trọng đến vậy khi lựa chọn công nghệ?

[2] https://en.wikipedia.org/wiki/Turing_machine

Chapter 7

Thủ Thuật và Kỹ Năng Để Học Tốt Hơn

Có một số mẹo và thủ thuật giúp tối đa hóa tốc độ học tập bền vững của bạn. Các lập trình viên biết những điều này có ích, nhưng chúng ta vẫn cứng đầu bỏ qua chúng mãi.

Nhưng phòng khi bạn muốn học nhanh hơn, đây là một vài điều có tác dụng với một số người. Không đảm bảo gì cả; mỗi người mỗi khác, và bạn có thể có con đường riêng hiệu quả hơn.

> **Âm nhạc** là một trong những điều như vậy. Có người thề rằng sự im lặng hoặc tiếng ồn trắng, hồng, hay nâu mới hiệu quả. Có người nghe nhạc cổ điển, electronica, hay metal. Hãy làm điều gì phù hợp với bạn.

7.1 Flow

Flow[1] là một trạng thái tinh thần mà bạn đạt được khi tập trung cao độ và các ý tưởng kết nối tự do với nhau. Không có gì làm gián đoạn.

Các lập trình viên thích đạt trạng thái flow để tối đa hóa năng suất.

Dưới đây là những đặc điểm của trạng thái đó, lấy thẳng từ Wikipedia:

1. Tập trung mãnh liệt và chú tâm vào khoảnh khắc hiện tại
2. Hành động và nhận thức hòa làm một
3. Mất đi ý thức tự quan sát bản thân
4. Cảm giác kiểm soát và làm chủ tình huống hoặc hoạt động
5. Sự bóp méo về trải nghiệm thời gian, khi nhận thức chủ quan về thời gian bị thay đổi
6. Trải nghiệm hoạt động như một phần thưởng tự thân

Có lẽ bạn đã từng trải qua điều này trong một khía cạnh nào đó của cuộc sống. Hãy nhớ rằng nó có thể rất có lợi cho các lập trình viên.

7.2 Đọc Trước

Tôi biết khi còn là sinh viên, nếu có bài gì nộp vào Chủ nhật, tôi thường đọc lần đầu tiên vào đúng Chủ nhật. Ai khác cũng vậy không? Ừ thì vậy.

Nhưng đây là một ý tưởng khác: đọc bài tập ngay khi bạn nhận được nó. Có thể bạn chưa biết đủ để hiểu hết, nhưng không sao. Cứ đọc qua thôi, thậm chí không cần đọc kỹ.

Và một ý tưởng khác: đọc bài tập khi bạn đã làm xong một số phần đọc và nghe giảng khác, nhưng là trước vài ngày so với lúc bạn định bắt đầu làm. Lại cứ đọc qua thôi, không cần lo giải quyết ngay. Có thể đọc ngay **và** đọc lại giữa tuần!

[1] https://en.wikipedia.org/wiki/Flow_(psychology)

Điều này giúp kích hoạt não của bạn với thông tin đó. Bạn sẽ không ghi nhớ hết hay hiểu hết, nhưng não sẽ bắt đầu nghiền ngẫm nó ở hậu trường và giúp bạn dễ xử lý bài hơn khi cuối cùng bạn bắt tay vào lúc 9 giờ tối Chủ nhật.

Và khi bạn bắt đầu quy trình giải quyết vấn đề bốn giai đoạn, tài liệu sẽ có vẻ quen thuộc hơn và dễ tiếp cận hơn.

Một biến thể mạnh mẽ của điều này là hoàn thành giai đoạn *Hiểu* sớm. Chưa cần *Lên kế hoạch* hay *Viết code* vội.

7.3 Không Copy-Paste Code

Bạn cần giải một bài toán, và nhìn kìa trên mạng! Có giải pháp sẵn rồi! Đến lúc mang kỹ năng `CTRL-c` / `CTRL-v` ra dùng!

Trước đây người ta thường tìm thấy giải pháp trên trang lập trình Stack Overflow[2]. Và vẫn còn dùng. Nhưng ngày nay họ có xu hướng nhờ AI hơn.

Người mới học lập trình không nên làm vậy. Hãy nhớ mục tiêu chính: phát triển kỹ năng giải quyết vấn đề xuất sắc. Copy-paste code **không hề** giúp ích gì cho mục tiêu này.

Một ngoại lệ cho quy tắc này là nếu bạn đã vật lộn với bài toán một thời gian, như đề cập trong phần *Quy tắc 30 phút* bên dưới. Nhưng ngay cả khi đó, bạn **phải** hiểu 100% hoàn toàn đoạn code bạn đang copy trước khi dùng nó.

7.4 Quy Tắc 30 Phút

Nếu bạn đã bí 30 phút và **thực sự** đã cố tấn công vấn đề từ nhiều hướng khác nhau mà vẫn không ra, đã đến lúc tìm kiếm sự trợ giúp.

Vâng, bạn có thể làm lâu hơn 30 phút mà không cần giúp đỡ (ai cũng làm vậy), nhưng 30 phút là sự cân bằng tốt giữa việc cố gắng chăm chỉ và sử dụng hiệu quả thời gian học tập hạn chế của bạn.

Đây là một câu chuyện. Một trong những cuốn sách yêu thích của tôi để học lập trình có tên là *Cấu trúc và Diễn giải Chương trình Máy tính*[3], hay SICP viết tắt. (Tôi đặc biệt thích phiên bản Scheme — thực sự giúp bạn học đệ quy.)

Các bài toán lập trình trong cuốn sách đó có thể rất thử thách. Nhưng tôi tự đặt giới hạn thời gian sáu giờ cho mỗi bài. (Tôi thường không làm cả sáu giờ liền một lúc.) Sau khi hết giờ, tôi xem đáp án.

Và đây là lý do tại sao điều đó có tác dụng và tại sao nó quan trọng: trong khi bạn nỗ lực với một bài toán, bạn đang bận xây dựng một khung tư duy xung quanh nó, cố gắng hỗ trợ nó. Và nếu bạn tìm được đáp án, tuyệt! Nhưng dù không tìm được, *bạn vẫn đã xây dựng được khung tư duy đó.*

Vì vậy, khi bạn từ bỏ sau giới hạn thời gian, rất thường xuyên giải pháp bạn đọc khớp gọn vào khung tư duy bạn đã xây. Chỉ là một bước nhỏ từ khung tư duy của bạn đến giải pháp, và việc bước được bước nhỏ đó *dễ hơn nhiều* so với việc cố grok[4] toàn bộ từ đầu.

So sánh với việc bạn chỉ tìm đáp án ngay lập tức mà không xây dựng khung tư duy. Giải pháp không có chỗ để "đặt" trong não bạn, và không kết nối với bất cứ điều gì khác. Và đó là bước dài hơn nhiều để đạt được sự hiểu biết.

Sự vật lộn rất quan trọng! Đừng bỏ qua sự vật lộn! Nhưng đồng thời, đừng kéo dài mãi. Khi hết thời gian, hãy xin giúp đỡ từ bạn học, gia sư, giảng viên, hoặc (nếu được phép) AI.

Ngoài ra, sau đó có thể đi dạo một chút.

[2] https://stackoverflow.com/
[3] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs
[4] https://en.wikipedia.org/wiki/Grok#In_computer_programmer_culture

7.5 Đi Dạo

Vâng, tôi nói thật. Bạn đang bí, không có gì có vẻ hoạt động, và không có thêm hướng tấn công nào nảy ra trong đầu. Bạn làm gì?

Đi dạo.

Khi bạn thử các cách tiếp cận khác nhau cho một bài toán, nó giống như bạn đang để lại những rãnh mòn trên đường, và não bạn có xu hướng tập trung vào những rãnh hiện có thay vì thử điều gì mới. Bạn bị khóa chặt vào những cách tiếp cận đã thử nhưng không hiệu quả. "Giá mà tôi chỉ có thể tinh chỉnh cái gần đúng này..." nhưng bạn không thấy cách nào.

Đứng dậy và đi bộ. Có thể bạn đang ở nơi làm việc và điều này chỉ có nghĩa là bạn đi vòng quanh hành lang. Hoặc có thể bạn có thể ra ngoài ban công, ra trước cửa, hoặc lên mái nhà.

Điều này giải phóng tâm trí bạn khỏi những rãnh mòn đó và khám phá các cách tiếp cận mới.

Đã có những lần tôi quyết định đi dạo và mới bước ra cửa được hai bước thì một cách tiếp cận mới cho bài toán đã nảy ra trong đầu.

Phương pháp thoát khỏi bế tắc này đã được kiểm chứng và đáng tin.

7.6 Vịt Cao Su

Hãy nói chuyện với ai đó về vấn đề. Đây là một kỹ thuật hiệu quả đến nỗi nó thậm chí hoạt động khi bạn đang nói chuyện với một con vịt cao su vô tri, từ đó có tên gọi *rubber ducking* (nói chuyện với vịt cao su).

Ý tưởng cơ bản là bạn sẽ dẫn dắt người kia qua các bước giải quyết vấn đề, thực chất là dạy họ. Giúp họ hiểu vấn đề và nhờ họ giúp lên kế hoạch.

Điều thực sự tuyệt vời về kỹ thuật này là: *nó hoạt động ngay cả khi người kia (hoặc con vịt) không có chuyên môn kỹ thuật.*

Một trong những lý do là để hiểu một vấn đề và đưa ra kế hoạch, bạn thực sự không cần biết gì về lập trình. Họ vẫn có thể giúp được.

Và đây là điều *thực sự* tuyệt vời: họ thậm chí không cần nói gì cả. Chỉ hành động dạy ai đó về vấn đề thôi thường đã đủ để bạn tự tìm ra câu trả lời. Có thể là một phần hiểu biết bạn bị bỏ lỡ, hoặc có một lỗ hổng không rõ ràng trong kế hoạch của bạn. Nói chuyện qua có thể giúp bạn tìm ra những điều đó.

Có một lần, kết hợp giữa đi dạo và rubber ducking, một đồng nghiệp của tôi bước đến chỗ tôi làm, giơ tay lên như muốn hỏi, dừng lại một nhịp, rồi nói, "Thôi không cần, tôi tự tìm ra rồi." Chỉ vậy thôi là đủ.

7.7 Ghi Lại Câu Hỏi

Khi nghiền ngẫm mô tả bài toán hoặc học một công cụ hay ngôn ngữ, về cơ bản có hai loại câu hỏi nảy sinh.

1. **Câu hỏi chặn tiến độ** là những câu hỏi bạn cần được trả lời ngay vì chúng đang chặn tiến độ của bạn. Bạn không thể làm gì khác cho đến khi có câu trả lời.
2. **Câu hỏi không chặn tiến độ** là những điều nảy sinh trong quá trình phát triển, thú vị, nhưng bạn có thể tiếp tục mà không cần biết câu trả lời ngay.

Tôi thích ghi lại các câu hỏi không chặn tiến độ và tìm câu trả lời sau. Những thứ như, "Ngôn ngữ này có hỗ trợ destructuring assignments không?" hay "Thư viện có thể cung cấp số ngẫu nhiên trong khoảng số nguyên không?" hay "Những giao thức mạng nào khác được tích hợp sẵn trong thư viện chuẩn?"

Đó là những thứ tôi tò mò, nhưng không cần biết câu trả lời ngay.

Quay lại và tìm câu trả lời sau có thể giúp xây dựng bức tranh đầy đủ hơn về các hệ thống bạn đang làm việc và giúp bạn trở thành lập trình viên hiệu quả hơn.

7.8 Xây Dựng Tấm Thảm Kiến Thức

Đây là nơi mọi thứ hội tụ lại.

Khi bạn mới bắt đầu lập trình, bạn đang đứng giữa thế giới kiến thức bao la chưa được khám phá. Bạn đã học cách in `Hello, world!` ra màn hình, nhưng đó là tất cả.

Vì vậy bạn bắt đầu lập bản đồ. Bạn thấy có các hàm, biến, thao tác I/O và bạn thấy chúng kết nối với nhau như thế nào. Và bạn học về mạng máy tính, thấy nó kết nối với hệ thống I/O của OS, và bạn nối chúng trên bản đồ.

Khi bản đồ lớn dần, bạn vẽ các kết nối giữa nhiều thứ bạn đã học, và dần dần bạn thấy rằng thế giới phát triển phần mềm kết nối với nhau nhiều hơn là tách rời. Nhiều bài toán rất giống nhau với nhiều bài toán khác.

Và khi bạn biết nhiều bài toán như vậy, đó là rất nhiều sức mạnh bạn có thể mang vào các thử thách mới. "Ồ, bài toán x này nhắc tôi nhớ đến bài toán y. Có thể tôi giải nó theo cách tương tự."

> **Bạn có một nhóm 10 người được đánh số từ 0 đến 9** và họ đều đang xếp hàng tại cửa sổ ngân hàng. Mô phỏng của bạn cần họ theo thứ tự ngẫu nhiên không có trùng lặp trong thời gian $O(n)$. Bạn code thế nào?
>
> Có thể trước đó bạn đã viết chương trình xáo trộn bộ bài sử dụng thuật toán nổi tiếng Fisher–Yates[a]... đợi đã! Tất cả những gì bạn cần làm là tạo danh sách người được đánh số từ `0` đến `9` theo thứ tự, rồi xáo trộn danh sách như xáo bài!
>
> Đây là cùng một bài toán!
>
> [a] https://en.wikipedia.org/wiki/Fisher–Yates_shuffle

Người mới học lập trình giải các bài toán lập trình thuần túy bằng logic và lý luận.

Các lập trình viên có kinh nghiệm cũng dùng logic và lý luận, nhưng họ chủ yếu dựa nhiều vào nhận dạng mẫu. Mẫu code nào tôi biết giải tốt nhất loại bài toán tôi đang gặp?

Tóm lại, họ dựa vào tấm thảm kiến thức liên kết chặt chẽ mà họ đã xây dựng qua nhiều năm lập trình.

Khi học code, hãy tìm cách mà thứ bạn đang học hiện tại kết nối với phần còn lại của thế giới lập trình bạn đã khám phá. Tạo ra những kết nối đó để bạn có thể khai thác chúng sau.

7.9 Nhận và Cho Đi Code Review

Việc đưa code của bạn ra ngoài và nhờ ai đó tư vấn cách cải thiện có thể rất khó. Dễ bị coi là công kích cá nhân.

Nhưng hãy chống lại cái thôi thúc đó, và coi mỗi code review bạn nhận như một món quà. Có người sẵn lòng bỏ thời gian giúp bạn trở thành lập trình viên tốt hơn, thường là không tốn gì của bạn.

> **Hãy cư xử tử tế trong code review** dù bạn là người được review hay người review. Hãy hỗ trợ trong phản hồi của mình, khiêm tốn, và đừng coi phản hồi tiêu cực là chuyện cá nhân.

Và nếu không tìm được người để giúp, hãy đưa code vào AI chatbot và nhờ nó review.

Ngay cả khi bạn không thành thạo về chủ đề đó, điều đó không nên ngăn bạn thực hiện code review khi có thể. Chỉ đọc code của người khác thôi cũng giúp bạn tiếp xúc với các phong cách code và mẫu thuật toán khác nhau mà bạn có thể chưa biết đến.

Nhưng hãy nhớ luôn có chính kiến về phản hồi bạn nhận được, và dùng sự phán xét phê phán khi quyết định có áp dụng hay không.

7.10 Tham Gia Câu Lạc Bộ

Lập trình về mặt lịch sử và nhìn chung là hoạt động cá nhân — ít nhất là phần bạn ngồi gõ phím. Và ngay cả trong các giai đoạn *Hiểu* và *Lên kế hoạch* cũng có thể hấp dẫn khi làm việc một mình.

Tham gia một nhóm người có cùng chí hướng nơi bạn có thể trao đổi ý tưởng (hoặc chỉ xả hơi) có thể thực sự giúp não bạn thoát khỏi những rãnh mòn và giúp bạn tiếp cận bài toán theo những cách bạn chưa nghĩ đến hoặc thậm chí chưa biết đến.

Nó cũng tốt cho networking, và nhiều câu lạc bộ có những buổi thuyết trình thú vị bạn có thể tham dự. Hay thậm chí tốt hơn, hãy thuyết trình tại đó!

Câu lạc bộ có thể làm cho cuộc chiến cảm giác ít cô đơn hơn nhiều. Chúng ta đều cùng chung hành trình này.

7.11 Suy Ngẫm Về Chương

- Bạn (cá nhân) làm gì để đạt trạng thái *flow*?
- Những lợi thế của việc đọc bài tập sớm hơn rất nhiều so với lúc bạn định làm là gì?
- Những bất lợi của việc làm một bài toán quá ít thời gian trước khi nhờ giúp đỡ là gì?
- Những bất lợi của việc làm một bài toán quá lâu trước khi nhờ giúp đỡ là gì?
- Khi nào thì copy-paste code được chấp nhận? Điều gì xảy ra nếu bạn dùng nó không đúng cách?
- Tại sao đi dạo là ý tưởng hay khi bị bế tắc?
- Một con vịt cao su vô tri có thể là người bạn đồng hành lập trình tốt như thế nào?
- Tác giả muốn nói gì về *tấm thảm kiến thức* và nó liên quan đến kỹ năng của bạn với tư cách là lập trình viên như thế nào?
- Bạn thu được gì khi nhận code review? Bạn thu được gì khi cho code review?

Chapter 8

Debug

Trước khi bắt đầu, cách tốt nhất để debug một chương trình là ngay từ đầu không có bug. Dù chúng ta chỉ là con người và chắc chắn mak mstakes, cách tốt nhất để tránh bug là tuân theo khung giải quyết vấn đề. Hãy nhớ rằng trận chiến lập trình nằm ở các giai đoạn *Hiểu* và *Lên kế hoạch.* Bạn hoàn thành những giai đoạn đó càng đầy đủ và chính xác, bạn sẽ càng có ít bug khi viết code.

Như vậy, hãy nói về những việc cần làm khi điều không tránh khỏi xảy ra.

8.1 Mô Hình Tư Duy

Đây là một trong những điều quan trọng nhất để trở thành lập trình viên: *có mô hình tư duy về quá trình tính toán.*

Tức là, bạn phải có khả năng đọc code và biết điều gì sẽ xảy ra.

```
def foo(n):
    i = 0

    while i < n:
        i = i + 1 + (i % 2)

    print(i)

foo(5)
```

Hãy đọc đoạn code Python vô nghĩa ở trên. Tính toán kết quả đầu ra trong đầu. Rồi xem bạn có đúng không. (Tôi viết code đó, và vẫn mất tôi một phút tốt đẹp để mô phỏng câu trả lời trong đầu. Nhưng tôi đã đúng!)

Nếu bạn không thể "chạy" code trong đầu, bạn không thể debug. Vâng, tôi nói thẳng thế. Tôi chắc một số người không đồng ý với tôi, nhưng tôi muốn nhấn mạnh tầm quan trọng của điều này.

Debug là nghệ thuật tìm ra phần code nơi mô hình tư duy của bạn về quá trình tính toán và thực tế của quá trình tính toán không khớp nhau. Và sau đó sửa nó.

Nếu bạn không có mô hình tư duy, bạn không có gì để so sánh và sẽ tiến triển rất ít.

Làm thế nào để cải thiện mô hình tư duy về tính toán của bạn?

- **Nghiên cứu code**. Trace qua nó. Có rất nhiều code ngoài kia để luyện tập, ví dụ trên GitHub, trên HackerRank, codebase của bạn học, code của chính bạn viết bốn tháng trước mà bạn đã quên nó hoạt động thế nào, v.v.
- **Dự đoán kết quả đầu ra**. Khi bạn học code, hãy cố đoán xem nó sẽ hoạt động thế nào khi chạy.

- **Trace thủ công một lần chạy**. Dùng bảng trắng để theo dõi thủ công các biến nhận giá trị gì, hàm nào được gọi, và đang thực thi dòng code nào.
- **Viết đặc tả**. Nghiên cứu một đoạn code và "reverse engineer" nó. Tìm hiểu nó làm gì, rồi viết một đặc tả dễ đọc cho người mô tả hoàn hảo thuật toán hoặc codebase ở mức độ mà người đọc có thể tái triển khai từ đầu.
- **Chạy từng bước với debugger**. Để máy tính chỉ cho bạn thấy chương trình đang chạy như thế nào. Chúng ta sẽ nói thêm về điều đó bên dưới.

Bạn chắc chắn sẽ cải thiện kỹ năng này qua thực hành.

8.2 Tái Hiện Bug

Trước tiên hãy làm điều này: xem bạn có thể khiến bug xảy ra ổn định không. Có thể tái hiện ("repro") bug là bước đầu tiên để có thể dẹp nó.

Đôi khi đây thực sự là phần khó. Bạn thấy điều gì đó sai một lần (hoặc ai đó báo cáo họ thấy nó), và bây giờ bạn không thể repro nó.

> **Tại thời điểm này bạn có thể bị cám dỗ sử dụng một kỹ thuật lập trình được gọi là "cầu nguyện"** khi bạn đang cầu nguyện rằng hoặc bạn không thực sự thấy nó hoặc bug sẽ không bao giờ nhô mặt xấu xí của nó lên nữa trên đời này. Nhưng đây là sự thật đáng buồn: nếu ai đó thấy bug hiếm gặp này *chỉ một lần* khi kiểm thử, hàng nghìn hay hàng triệu người sẽ thấy bug khi nó đưa vào production. Định luật Murphy *và* thống kê? Bạn không có cơ hội chống lại điều đó.

Nếu bạn không thể repro nó, bạn chỉ đang bắn vào bóng tối khi cố sửa nó. Mục tiêu của bạn lúc này là chỉ cần khiến nó xảy ra. Làm việc ngược lại một cách logic. Điều gì *nhất định* đã xảy ra trong luồng chương trình để thấy những gì người báo bug thấy? Tìm ở đó trước. Ngay cả khi bạn chắc chắn điều kiện nào đó *không thể* đúng, nếu nó nhất định phải đúng để thấy bug, thì nó *nhất định* đã đúng! Tiếp tục trace ngược lại, tìm nơi mô hình tư duy của bạn và code không khớp và dùng điều đó để tự repro.

Nếu bạn chỉ có thể repro nó lẻ tẻ, bạn có cơ hội, nhưng sẽ khó. Mục tiêu của bạn là khiến nó repro ổn định. Điều này giúp bạn không phải mãi mãi cố gắng repro hiếm gặp. Nếu bạn có thể khiến nó xảy ra ổn định, đó là tiết kiệm thời gian lớn và giúp bạn thu hẹp nơi bug có thể nằm.

Khi bạn tìm ra cách repro ổn định, bây giờ bạn có thể xử lý có hệ thống hơn. Bạn muốn tìm số bước tối thiểu có thể khiến bug xuất hiện. Ví dụ, bạn đang chơi game và thấy bug khi bạn hoàn thành 20 vòng đường đua rồi lái vào cây. Có thể thử chỉ lái vào cây trước. Nếu may mắn, bạn đã tiết kiệm được 20 vòng và thu hẹp bug xuống còn cái cây. Nếu không có gì xảy ra, bạn biết 20 vòng là phần không thể thiếu của bug. Có thể thử một vòng rồi đến cây. Có repro không?

Tìm số bước tối thiểu không chỉ giúp bạn thu hẹp hơn nữa vị trí bug trong code, mà còn giúp kiểm thử các bản sửa dễ hơn vì bạn không phải mất nhiều thời gian để repro vấn đề.

8.3 Tìm Bug

Có một bug ở đâu đó. Bạn biết điều đó vì khi bạn cung cấp đầu vào cho code, nó xử lý, và cuối cùng cho ra kết quả bất ngờ.

Ở đâu đó trong quá trình lớn đó, thực tế của tính toán không khớp với mô hình tư duy của bạn về tính toán.

Ban đầu, tất cả những gì bạn có thể biết là ở đâu đó trong 10.000 dòng code có điều gì đó sai. Vậy là bạn đã thu hẹp xuống mức đó. Mô hình tư duy của bạn nói rằng nếu đầu vào là `2`, đầu ra sẽ là `3490`. Nhưng thay vào đó đầu ra là `299792458`.

Do đó bạn biết bug nằm ở đâu đó giữa đầu vào và đầu ra.

Bạn *có thể* chỉ bắt đầu thay đổi ngẫu nhiên trong code để xem nó có được sửa không. Đôi khi được gọi là *shotgun debugging* hoặc *prayer debugging*, và nó rất, rất, rất, rất, *rất* hiếm khi hiệu quả. Thường xuyên hơn nhiều bạn chỉ làm rối tung mọi thứ và khiến vấn đề khó tìm hơn. Nó giống như cố sửa vấn đề điện của xe bằng cách ngẫu nhiên thêm và cắt dây. Không đáng để debug theo cách này.

Dù vậy, đây là kỹ thuật rất phổ biến được thực hành, uổng công, bởi sinh viên trên khắp thế giới. Tuy nhiên bạn không nên dùng nó.

Thay vào đó, đã đến lúc xử lý có hệ thống. Ở đâu đó trong đường ống tính toán đó, các giá trị trung gian được tính toán không khớp với mô hình tư duy của bạn. Nhiệm vụ của bạn là tìm ra nơi đó.

Vì vậy bạn bắt đầu thăm dò *bên trong* chương trình ở các điểm khác nhau để xem nơi nào mọi thứ đi lệch. Binary search rất hay — nhảy đến giữa quá trình và kiểm tra các giá trị trong một lần chạy (xem bên dưới). Nếu chúng như mong đợi, bug vì vậy phải nằm giữa giữa chương trình và phần cuối! Bạn vừa giảm một nửa không gian tìm kiếm. Bây giờ làm lại cho đến khi bạn thu hẹp đủ để thấy bug.

Tôi muốn tranh luận, dù một số người có thể không đồng ý, *bug chưa được tìm thấy cho đến khi bạn hiểu nó.* Tức là, bạn **phải** hiểu chính xác chương trình của bạn cho ra kết quả `299792458` thay vì kết quả mong đợi `3490` như thế nào. Hiểu đầy đủ điều đó có nhiều lợi ích:

- Bạn có thể tự tin hơn rằng bạn đã sửa bug thực sự.
- Bạn sẽ học cách nhận ra các mẫu dẫn đến bug này, giúp bạn tránh tốt hơn trong tương lai.
- Bạn đang rèn luyện kỹ năng giải quyết vấn đề trong khi làm điều này.

Khi bạn hiểu đầu ra sai được tạo ra như thế nào, hãy sửa vấn đề một cách dứt khoát và chính xác, và biết tại sao cách sửa sẽ hoạt động.

Cuối cùng, nếu bạn chỉ đang nộp báo cáo bug (tức là người khác sẽ sửa), có khả năng cung cấp cho họ các bước tối thiểu cần thiết để repro sẽ làm bạn trở thành anh hùng của họ trong ngày đó. Hãy xem xét từ góc độ ngược lại; bạn thích sửa bug với chuỗi bước dài dòng mơ hồ để repro, hay một bug với vài bước khiến nó repro mỗi lần? Càng cụ thể, người sửa bug càng vui, dù đó là bạn hay người khác.

8.4 Debug Bằng Print

Cách thăm dò phần mềm truyền thống tốt bụng khi đang chạy được gọi là *print debugging*, hoặc nếu bạn là lập trình viên C, *printf debugging* (đọc là "print eff").

Về cơ bản đây chỉ là việc đặt các câu lệnh in một cách chiến thuật bên trong code để xem chương trình của bạn đang ở trạng thái nào.

Có một số cách dùng phổ biến:

- In bất cứ thứ gì để xem liệu một phần code có thực sự thực thi không.

  ```
  print("A")

  x = foo()

  print("B")

  if x == 3:
      print("C")
      x *= 2
  else:
      print("D")
      bar()

  print("E")
  ```

  Chú ý rằng khi tôi chạy code, tôi có thể thấy nó chạy đến đâu trước khi crash, và tôi có thể xác định `x` có bằng `3` không.

- In một số giá trị cụ thể để xem chúng là gì.

  Đây là ví dụ nơi chúng ta lấy dữ liệu từ một cảm biến nào đó trong vòng lặp để xử lý. Chúng ta nghi ngờ một số dữ liệu bị lỗi (có thể cảm biến bị hỏng) nên đang in ra để xem chúng ta nhận được gì.

```
while not done:
    data = get_sensor_data()

    print(f"Got sensor data: {data}")

    process_sensor_data(data)

    done = data < 0
```

> **Đừng dùng tục ngữ trong các câu lệnh debug của bạn.** Định luật Murphy nói rằng nếu bạn dùng tục ngữ, bạn sẽ quên xóa nó, và dù nó ở trong phần code bạn chắc chắn sẽ không bao giờ chạy, nó sẽ không tránh khỏi hiện lên màn hình trong khi bạn đang demo cho khách hàng trước mặt sếp vào ngày sếp đang đánh giá bạn để tăng lương.
>
> Tôi biết rõ hơn ai hết lập trình có thể bực bội đến mức nào. Và khi tôi cảm thấy như vậy và quên thở sâu và lấy lại bình tĩnh, tôi in cái này:
>
> ```
> print("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
> ```
>
> Không chỉ nó nổi bật rõ ràng trên màn hình, mà nó còn rất dễ gõ khi bực bội và giúp giải tỏa năng lượng tiêu cực của tôi. Và nếu khách hàng thấy nó, đó chỉ là lỗi nhỏ.
>
> Một người bạn khác của tôi đề xuất in các emoji không gây phản cảm, cũng là cách thú vị để xả stress.

Bây giờ, print debugging bị coi là phương tiện debug kém hơn so với dùng debugger thật sự (như trong phần sau). Nhưng ai cũng làm vậy vào một lúc nào đó, và một số người còn thề bằng nó.

Nơi tôi nghĩ nó thực sự tỏa sáng là khi bạn cần thu thập nhiều dữ liệu về lần chạy để thấy mẫu lớn hơn nổi lên, hoặc khi bạn cần bắt một sự kiện hiếm gặp. Nếu có gì đó xảy ra một lần trong 10.000 lần chạy, việc chạy từng bước với debugger tiêu chuẩn sẽ mất mãi. Bạn có thể thêm một số câu lệnh in và script chạy 10.000 lần và xem kết quả đầu ra để thấy khi nào nó xuất hiện.

Một điều cần chú ý là nếu bạn đang in nhiều, có thể khó phân tích kết quả bằng mắt, và thông báo lỗi có thể bị mất trong đó. Tôi khuyên nên chuyển hướng kết quả vào file rồi mở nó trong editor để tìm kiếm.

Và cuối cùng, đừng quên xóa tất cả các câu lệnh in trước khi bạn giao nộp công việc!

8.5 Debugger

Debugger là các công cụ giúp bạn tìm bug. Có nhiều debugger khác nhau, nhưng hầu hết tất cả đều có chung một tập tính năng. Các tính năng chính là:

- Thêm *breakpoint* nơi chương trình sẽ dừng chạy và bạn sẽ có quyền kiểm soát trong debugger
- "Single step" qua chương trình từng dòng một
- Kiểm tra giá trị của các biến

Các tính năng bổ sung phổ biến là:

- Bước vào một hàm
- Tiếp tục ra khỏi một hàm
- Bước qua một hàm
- Đặt giá trị của các biến
- Kiểm tra call stack
- Đặt breakpoint kích hoạt theo điều kiện

Hiếm gặp hơn là *time-travel debugger*. Ngoài việc cho phép bạn bước tiến qua chương trình, chúng còn cho phép bạn bước lùi! Điều này rất hay nếu bạn bước qua bug vô tình và muốn bước lại để xem nó.

Tất cả các IDE[1] lớn đều có chức năng debugger. (Đó là một phần của "tích hợp", chữ "I" trong "IDE".)

[1] https://en.wikipedia.org/wiki/Integrated_development_environment

Cũng có các debugger độc lập mà bạn có thể chạy. Và tất cả các ngôn ngữ chính thống đều có hỗ trợ debugger nào đó.

Như bạn có thể tưởng tượng, với những tính năng đó, debugger thực sự mạnh mẽ.

Nếu bạn nghi ngờ có bug trong hàm `foo()`, bạn có thể đặt breakpoint ở đó, chạy code, rồi có quyền kiểm soát debugger khi `foo()` thực thi. Sau đó bạn có thể bước qua từng dòng, xem cách các giá trị của biến thay đổi. Và không cần thêm bất kỳ câu lệnh in nào.

Chú ý rằng trong VS Code, việc thiết lập debugger có thể là chuyện đơn giản, hoặc bạn có thể phải chỉnh sửa một số file JSON khó hiểu để nó hoạt động.

Dù sao đi nữa, học cách dùng debugger là kỹ năng quý giá có thể giúp bạn tiết kiệm rất nhiều thời gian trong khi cố tìm con quỷ nhỏ đó trong code.

8.6 Suy Ngẫm Về Chương

- *Mô hình tư duy về tính toán* là gì và nó quan trọng như thế nào cho việc debug?
- Tại sao việc tìm trường hợp tái hiện tối thiểu của bug lại là bước quan trọng trong việc sửa bug đó?
- Bạn thu hẹp vị trí bug trong code như thế nào?
- So sánh print debugging với dùng debugger. Bạn nghĩ điểm mạnh và điểm yếu tương đối của chúng là gì?

Chapter 9

Học Một Ngôn Ngữ Mới

Ngôn ngữ đầu tiên bạn học là khó nhất. Không chỉ học ngôn ngữ, bạn còn đang học các *khái niệm* được dùng trong ngôn ngữ đó. Ý tôi về khái niệm là những thứ như cách tổ chức hàm và truyền đối số, cách chạy vòng lặp, cách làm điều kiện, v.v.

Tất cả các ngôn ngữ đều có chung các khái niệm này (nhiều hay ít — sẽ nói thêm sau), và vì vậy học ngôn ngữ thứ hai chỉ là học cách áp dụng những khái niệm bạn đã biết.

Nó giống như nếu bạn đã biết tiếng Tây Ban Nha, học tiếng Ý không phải bước nhảy lớn.

Chương này nói về việc học thêm ngôn ngữ sau khi bạn đã học ngôn ngữ đầu tiên. Điều này quan trọng vì bạn sẽ học ngôn ngữ mới trong suốt sự nghiệp của mình. May mắn thay, học ngôn ngữ mới tự nó là một kỹ năng, và bạn càng học nhiều ngôn ngữ mới, nó càng trở nên dễ hơn.

Có hai phần lớn để học ngôn ngữ mới trong mô hình bạn đã biết (thủ tục, hướng đối tượng, hàm, v.v.)

1. **Học cú pháp**. Như `if` và `while`, và cách khai báo biến và hàm, v.v.
2. **Học thư viện chuẩn**. Đây là chức năng tích hợp sẵn mà bạn có thể tận dụng, như đọc và ghi file, in ra màn hình, hoặc kết nối với web server.

Học cú pháp thường là phần dễ hơn trong hai. Hầu hết các ngôn ngữ có cú pháp tương đối đơn giản.

Để so sánh, bạn có thể học động từ, danh từ, tính từ là gì và cách phân tích câu[1]. Nhưng đó chưa đủ để viết một tác phẩm văn học xuất sắc. Bạn cũng cần biết những từ nào có trong tay.

Và đó là phần phức tạp hơn. Nhiều thư viện chuẩn có *rất nhiều* chức năng tích hợp sẵn. Hãy lướt qua thư viện chuẩn Python xem ví dụ[2].

9.1 Học Cú Pháp

Nhảy thẳng vào! Theo một hướng dẫn và viết các chương trình "toy". Đây là những chương trình chỉ luyện tập một khía cạnh nào đó của ngôn ngữ.

Chúng ta làm điều kiện trong Rust như thế nào? Hãy viết một chương trình toy để tìm hiểu.

```
fn main() {
    if (1 == 2) {
        println!("Something is horribly wrong.");
    } else {
        println!("That's correct.");
    }
}
```

Đợi đã — những dấu ngoặc đơn đó có cần thiết quanh `if` không?

[1] https://en.wikipedia.org/wiki/Sentence_diagram
[2] https://docs.python.org/3/library/index.html

```
$ rustc foo.c
  warning: unnecessary parentheses around `if` condition
```

Không cần! Đây là chương trình toy; chúng ta chỉ dùng nó để học.

Để học tất cả cú pháp cần thiết, hãy lấy các khái niệm bạn đã biết và tìm hiểu cách áp dụng chúng trong ngôn ngữ mới.

- Hàm main
- Biến
- Điều kiện
- Vòng lặp
- Class
- v.v.

Lúc đầu sẽ gây bực bội vì bạn phải tra cứu từng. Thứ. Một. Giống như với một ngôn ngữ người mới, bạn biết về mặt khái niệm rằng bạn muốn đi đến siêu thị, nhưng bạn phải tra tất cả những từ tiếng Ý đó nếu bạn không biết tiếng Ý.

Tin tốt là cú pháp của ngôn ngữ máy tính *đơn giản hơn nhiều* so với ngôn ngữ người. Và bạn có thể nắm vững nó khá nhanh.

9.2 Học Thư Viện

Thư viện chuẩn là chức năng đã được tích hợp sẵn đi kèm với ngôn ngữ. Điều này tiện lợi vì bạn biết rằng tất cả mọi người đã cài ngôn ngữ đều có sẵn tất cả các hàm này và không phải tải xuống bất kỳ phụ thuộc bên thứ ba nào thêm.

Nhưng bạn cần quen thuộc với thư viện chuẩn của ngôn ngữ đó để biết ngôn ngữ có thể làm gì sẵn có, và để không tái phát minh bánh xe khi không cần thiết.

Một khuyến nghị là lướt qua thư viện chuẩn của ngôn ngữ bạn đang dùng. Bạn không phải biết chính xác cách sử dụng chức năng IMAP[3] của Python, nhưng biết nó có sẵn trong trường hợp bạn cần là rất có giá trị. Ít nhất, nó cho bạn biết Python là ứng viên để lựa chọn ngôn ngữ nếu bạn cần làm việc với IMAP.

Rồi khi bạn thực sự cần một số chức năng đó, bạn có thể đào sâu vào tài liệu và ví dụ để xem nó hoạt động thế nào.

Tôi có xu hướng học thư viện theo từng mảnh, chỉ học chi tiết những gì tôi cần để hoàn thành công việc. Tôi biết phần còn lại của những gì nó *có thể* làm (vì tôi đã lướt qua tài liệu), nhưng tôi chỉ biết từng mảnh và miếng đủ để code với chúng.

Và điều đó ổn, vì các thư viện rất đồ sộ, và khó có thể đạt được sự thành thạo mọi thứ trong đó. Bạn chỉ cần có thể học những gì bạn cần để hoàn thành công việc.

9.3 Học Một Mô Hình Mới

Trước tiên, *mô hình lập trình* là gì? Đó là cách mô hình hóa bài toán để bạn có thể đưa ra giải pháp. Tôi biết điều đó mơ hồ nhưng hãy kiên nhẫn với tôi vài đoạn.

Hãy tưởng tượng đang làm khai thuế. (Xin lỗi.) Khi làm, đó là một chuỗi các bước liên tiếp. Điền tên. Điền thu nhập. Nếu thu nhập lớn hơn một giá trị nào đó, làm x. Còn lại làm y. Đó là một thủ tục bạn đang theo. Bạn có thể mô hình hóa nó như một chuỗi các bước.

Hãy tưởng tượng bạn đang mô phỏng một thế giới fantasy 3D. Trong thế giới đó, bạn có thể có một loại sinh vật gọi là orc, và có thể có nhiều sinh vật loại đó chạy xung quanh. Và chúng đều có tọa độ riêng, và điểm máu[4] riêng của chúng, nhưng chúng đều có cùng hành vi khi bạn tiến lại gần. Bạn có thể mô hình hóa chúng như một tập hợp các đối tượng độc lập nhưng có hành vi tương tự.

[3] https://en.wikipedia.org/wiki/Internet_Message_Access_Protocol
[4] https://en.wikipedia.org/wiki/Health_(game_terminology)#Hit_points

Đây là hai cách khác nhau để giải quyết bài toán, hoặc bằng cách mô hình hóa chúng như một chuỗi các bước, hoặc như các đối tượng.

Chúng ta gọi các mô hình khác nhau này là *mô hình lập trình*. Ví dụ đầu là "lập trình thủ tục" (kiểu vậy; tôi đang lướt qua một chút), và ví dụ thứ hai là "lập trình hướng đối tượng".

Có rất nhiều mô hình[5], nhưng Bộ Ba Lớn là thủ tục, hướng đối tượng, và hàm.

Đây là điều đáng lo: học một mô hình mới thì khó. Khó hơn nhiều so với chỉ học một ngôn ngữ khác trong cùng mô hình.

Nếu bạn biết tiếng Tây Ban Nha, học tiếng Ý tương đối dễ. Nhưng học tiếng Trung, đó là điều khác! Không dễ như vậy. Tiếp tục phép so sánh, đó là mô hình khác. Bạn phải học các kỹ thuật và khái niệm mới mà bạn thậm chí có thể chưa biết đến từ các ngôn ngữ Roman.

> **Tôi đã học Erlang một thời gian trước**. Erlang[a] là một ngôn ngữ hàm, và tôi còn yếu với mô hình hàm.
>
> Ví dụ, trong Erlang, khi bạn đặt một "biến", bạn không bao giờ có thể thay đổi nó. Và mọi cách tôi biết để mô hình hóa bài toán đều liên quan đến việc thay đổi biến!
>
> Ý tôi là, bạn làm gì *bất cứ thứ gì* nếu không thể thay đổi biến?!
>
> Nhưng rõ ràng, các hệ thống lớn đã được triển khai thành công trong Erlang, vậy nên có cách. Nhưng tôi phải thay đổi cách suy nghĩ về cách mô hình hóa bài toán, và việc học cách mới đó là một thách thức đáng kể.
>
> [a] https://www.erlang.org/

Lời khuyên chính của tôi ở đây là dùng *rất nhiều* ví dụ để thấy ngôn ngữ đó thực hiện các tác vụ cơ bản thế nào. Tức là, thu thập và nghiên cứu nhiều chương trình toy.

Rồi đặt ra các thách thức liên quan (hoặc tìm một số trên mạng, hoặc nhờ AI tạo) cho phép bạn luyện tập để xây dựng kỹ năng và tìm ra các lỗ hổng trong hiểu biết của mình.

9.4 Suy Ngẫm Về Chương

- Tại sao học ngôn ngữ lập trình đầu tiên lại khó hơn học ngôn ngữ thứ hai?
- Sự khác biệt giữa học cú pháp và học thư viện là gì?
- Sự khác biệt giữa ngôn ngữ lập trình và mô hình lập trình là gì?
- Tại sao học một mô hình mới thường khó hơn học một ngôn ngữ lập trình mới trong mô hình bạn đã biết?

[5] https://en.wikipedia.org/wiki/Programming_paradigm

Chapter 10

Sử Dụng AI

Tính đến thời điểm viết bài này vào năm 2025, AI là thứ nóng hổi mới nhất[1]. Ai cũng dùng nó, và không ai ngừng nói về nó.

Câu hỏi mà không ai có vẻ có câu trả lời là điều gì sẽ xảy ra. Sam Altman[2] và những người khác sẽ nói với bạn rằng AI chỉ trong giây lát sẽ thống trị thế giới và giải quyết mọi thứ, và con người sẽ trở nên lỗi thời.

Tôi muốn nghĩ rằng điều đó sẽ không xảy ra, và ngay cả khi AI bắt đầu giải quyết mọi thứ, người ta vẫn sẽ muốn dùng óc sáng tạo của mình để đẩy ranh giới xa hơn mức AI có thể.

Điều đó có nghĩa gì với bạn với tư cách là sinh viên khoa học máy tính (tính đến thời điểm viết bài này)?

Hãy nói về cách bạn nên dùng AI với tư cách sinh viên và trong công việc, vì đó là hai điều khác nhau.

Nhưng trước đó, hãy nói về những gì bạn *không* nên làm.

10.1 Cách *Không* Nên Dùng AI Khi Còn Là Sinh Viên

Tôi đã đề cập ở chỗ khác rằng tôi thích nghiên cứu cuốn sách SICP[3] để cải thiện kỹ năng. Và cách tôi tự đặt giới hạn thời gian sáu giờ để giải các bài toán được đưa ra.

Bây giờ, những bài toán này hoàn toàn không phải bài toán thực tế. Đây là các bài toán luyện tập được thiết kế sẵn. Và — hãy nhớ điều này — câu trả lời đều có sẵn trên Internet ở nhiều nơi.

Vậy tại sao tôi dành đến sáu giờ? Thật lãng phí, phải không? Tại sao không chỉ clone repo của ai đó và tuyên bố đã triển khai xong?

Hay nếu không vậy, tại sao không chỉ sao chép những gì bạn bè tôi làm?

Hay nếu không vậy, tại sao không thuê người viết câu trả lời cho tôi?

Dĩ nhiên, bạn biết câu trả lời. Tôi không làm vậy vì nếu tôi làm, tôi chưa học được gì. Cụ thể hơn, *tôi chưa vật lộn với bài toán* nên sự học hỏi từ đó sẽ bị mất.

Tôi nghĩ bạn thấy điều này đang dẫn đến đâu: chỉ nhờ AI giải bài toán sẽ không tăng kỹ năng của bạn chút nào. Đó chỉ là nhờ người khác làm công việc khó.

Giống như tôi đến phòng tập và nhờ robot nâng tạ cho tôi. Đúng là tôi có thể bước ra nói tạ đã được nâng thành công, nhưng tôi không thu được gì từ trải nghiệm đó.

Tôi muốn bạn kể tên một hoạt động, ngoài việc ở phòng tập, bao gồm nâng tạ cả ngày. Câu trả lời: không có cái nào (nói chung cho phần lớn dân số). Vậy tại sao chúng ta dành thời gian nâng tạ ở phòng tập nếu không có hoạt động nào khác liên quan đến điều đó?

[1] Không giống như cách diễn đạt "new hotness" (cái mới nóng hổi), hiện đã trở nên nhàm. "Nhàm" cũng đã nhàm, nhưng nó tự tham chiếu nên tôi nghĩ nó đã chạm đáy.

[2] https://en.wikipedia.org/wiki/Sam_Altman

[3] https://en.wikipedia.org/wiki/Structure_and_Interpretation_of_Computer_Programs

Tất nhiên, những quả tạ chỉ là công cụ chúng ta dùng hướng đến mục tiêu lớn hơn là trở nên mạnh mẽ hơn nói chung.

Trường học cũng y như vậy. Các bài toán lập trình bạn có trong trường là những quả tạ. Chúng không thực. Chúng được thiết kế để cho bạn luyện tập để khi bạn ra làm việc, bạn có đủ sức mạnh để làm việc.

Và vì các bài toán không thực, AI có thể giải tất cả chúng *rất* dễ dàng. Có rất nhiều tài liệu đào tạo cho chúng để học.

Nhưng đừng bị lừa. Chỉ vì AI có thể giải các bài toán trường học của bạn không có nghĩa là nó có thể giải các bài toán thực tế bạn sẽ gặp trong công việc. (Tính đến thời điểm này, nó không thể.)

(Và nếu nó có thể giải tất cả những bài toán đó, bạn nghĩ các lập trình viên sẽ kiếm được bao nhiêu? Có lý do tại sao làm lập trình viên được trả cao, và đó là vì công việc khó. Nếu nó dễ như gõ prompt AI, nó sẽ trả lương tối thiểu. Điều đó nên gợi ý cho bạn rằng nếu *tất cả* những gì bạn có thể làm là prompt AI, bạn sẽ không nhận được gig lương cao.)

Nhưng điều đó không có nghĩa là bạn không nên giỏi dùng AI; chỉ là khi bạn còn là sinh viên, bạn phải dùng nó đúng cách để tối đa hóa phát triển kỹ năng.

TLDR của phần này là: đừng bao giờ nhờ AI giải toàn bộ dự án lập trình của bạn. Có lẽ nó có thể làm được, nhưng bạn sẽ không học được gì. Mục tiêu của dự án không phải là hoàn thành bài toán; mà là rèn luyện trong khi bạn hoàn thành nó.

10.2 Cách Dùng AI Khi Còn Là Sinh Viên

Trước tiên: nếu trường hay giảng viên của bạn cấm AI, đó là quy tắc. Và bạn phải bỏ qua những gì tôi đã viết ở đây. Xin lỗi. Tôi sẽ giả định họ chưa làm điều ngốc nghếch đó, và tiếp tục.

Vậy bạn nên dùng nó thế nào? Hãy dùng nó như bạn đang làm việc với một gia sư tốt biết nhiều thứ khá tốt. Gia sư AI chắc chắn mắc lỗi và đưa ra lời khuyên kém thỉnh thoảng, vì vậy bạn nên nhìn mọi thứ bạn học từ nó bằng con mắt phê phán.

Khi bạn bị mắc kẹt ở một phần nhỏ của dự án, hãy hỏi về điều đó. Khi bạn không nhớ cú pháp, hãy hỏi về điều đó. Hỏi về cách thành ngữ để viết vòng lặp xóa phần tử khỏi mảng đang lặp. Hỏi về một toán tử cụ thể làm gì hoặc cách dùng nó. Khi bạn có câu hỏi về ngôn ngữ hoặc thư viện, hãy hỏi. Những miếng nhỏ vừa miệng như vậy hoàn toàn có thể hỏi. Nó có thể nhanh hơn nhiều so với tìm kiếm Internet thông thường.

Một nơi khác tôi thấy AI làm tốt là với gợi ý. Bạn có thể đưa vào lượng code bằng một hàm và nói với AI, "Có gì đó sai với đây [mô tả vấn đề]. Nhưng đừng cho tôi câu trả lời và đừng cho tôi code đã sửa. Chỉ cho tôi gợi ý về cách tôi cần suy nghĩ về bài toán để tự tìm ra giải pháp."

Và khi bạn đã hoàn thành dự án (và nó hoạt động hoàn toàn), thì bạn có thể thoải mái nhờ AI một giải pháp để so sánh. Hay tốt hơn nữa, đưa giải pháp của bạn vào AI và nhờ cải tiến.

Hãy nhớ rằng một số "cải tiến" sẽ không phải cải tiến gì cả, và bạn sẽ muốn bỏ qua chúng. Hãy nhìn bằng con mắt phê phán. Hãy có chính kiến và có lý do cho lời khuyên nào bạn chấp nhận và lời khuyên nào bạn từ chối.

Cuối cùng, còn có một lúc khác ổn để dùng AI giải toàn bộ dự án: khi giảng viên của bạn nói bạn có thể. Điều này xảy ra khi họ đang cho bạn luyện tập thực tế hơn so với nâng tạ. Cả hai đều là cách sử dụng thời gian học tập có giá trị.

10.3 Cách Dùng AI Trong Công Việc

Trước tiên, một lưu ý: lần cuối tôi làm việc trong production (và tôi đã làm 20 năm), AI như chúng ta biết ngày nay không tồn tại. Điều đó nói rằng, tôi dùng nó để hoàn thành công việc nhanh hơn ngày nay. Đó là mức độ "chuyên môn" của tôi.

Tôi đã đề cập trước đây rằng lập trình viên mới giải bài toán bằng lý luận logic, nhưng những chuyên gia, ngoài ra, còn nhận ra các mẫu. Là lập trình viên có kinh nghiệm hơn, bạn hiểu rõ hơn về các khối

xây dựng nào cấu thành bài toán. Bạn nhận ra những mảnh bạn cần, và lý luận logic về cách lắp ghép chúng.

Nói cách khác, các lập trình viên có kinh nghiệm giỏi hơn ở *Hiểu* và *Lên kế hoạch*. (Họ giỏi hơn ở tất cả các giai đoạn, nhưng hãy nhớ rằng *Hiểu* và *Lên kế hoạch* là nơi trận chiến diễn ra.)

Do đó, họ có thể tận dụng AI để giúp viết code cho những khối xây dựng đó. Họ có thể nói những thứ như, "Tôi cần lọc những kết quả đó cho bất cứ thứ gì khớp với biểu thức chính quy này" — và sau đó họ nhờ AI code khối xây dựng đó, họ thành thạo kiểm tra xem code có đúng không và điều chỉnh để phù hợp với nhu cầu, rồi tiếp tục.

Ngay cả với các công nghệ họ không quen, điều này có thể giúp họ hoàn thành công việc. Nhưng họ vẫn cần dựa vào chuyên môn để biết khi nào cần học thêm. Tức là, các lập trình viên có kinh nghiệm có mũi để ngửi code nguy hiểm và biết khi nào họ cần tiến cẩn thận và thu thập thêm kiến thức.

Ở khía cạnh này, AI có thể thực sự hữu ích cho công việc proof-of-concept và tạo prototype nhanh nơi code thường dùng xong bỏ.

> *"Nào hãy nhanh lên! Không có gì đáng sợ ở đây!"*
> *"Đó chính xác là điều khiến tôi sợ."*
>
> — Satipo và Indiana Jones, *Raiders of the Lost Ark*

Lại nói, với tư cách sinh viên, bạn không thể mang kinh nghiệm đó (mà bạn chưa có được) vào cuộc, và nếu bạn chỉ cố dùng AI như một lập trình viên dày dạn, bạn sẽ có code đầy bug, mong manh mà bạn không biết cách sửa. Và tệ nhất là bạn sẽ không phát triển được kỹ năng cần thiết.

Nhưng khi bạn tích lũy thêm kinh nghiệm, bạn hoàn toàn có thể dựa vào AI để viết một lượng lớn boilerplate code cho bạn mà bạn đã biết logic đằng sau, dù sao.

10.4 AI và Thị Trường Việc Làm

Khi bạn đọc cái này, tôi muốn bạn biết rằng tôi, tác giả, đã nghĩ Yahoo![4] là ý tưởng ngớ ngẩn khi nó mới ra mắt. Tôi vẫn hơi nghĩ vậy, nhưng nó đã kiếm được hàng tỷ đô từ đó, nên tôi đã sai. Ít nhất là về mặt tư bản.

Một điều mà các lập trình viên lớn tuổi đã nghe trong suốt sự nghiệp là chúng ta đang trên bờ vực của cuộc cách mạng "không code", và công cụ này hay công cụ kia rốt cuộc sẽ đặt các lập trình viên ra ngoài vì mọi người ở khắp nơi có thể tạo ra phần mềm.

Mỗi dự đoán như vậy đều có điểm chung là tất cả đều sai một cách đáng kinh ngạc.

Nhưng mọi thứ chỉ sai cho đến khi chúng không còn sai nữa, và LLM như ChatGPT chắc chắn là những con thú mới lạ không chơi theo quy tắc cũ.

> **Tôi nhận ra chiêu pump cổ phiếu, dù vậy.** Tất cả các công ty "không code" đó đang nói khoác để nhận được lợi nhuận lớn cho nhà đầu tư. OpenAI và các tay chơi AI khác cũng nói khoác như vậy.
>
> Điều đó không có nghĩa là họ sẽ không đạt được những lợi nhuận đó; nhưng có nghĩa là nó có mùi quen thuộc.

Và LLM đang thể hiện chuyên môn coding đáng kinh ngạc. Tôi liên tục ngạc nhiên với những gì chúng có thể làm. Nhưng liệu chúng có làm được hết không?

Tôi có một thí nghiệm tư duy cho bạn. Giả sử có một AI giỏi đến mức tôi có thể nói với nó, "AI, hãy thiết kế và triển khai một tập đoàn sẽ nghiền nát tất cả đối thủ và làm tôi trở thành người giàu nhất trên Trái Đất," và nó sẽ thực sự làm được điều đó thành công.

Nhưng vấn đề là mọi người đều có thể truy cập cùng một AI và họ đều có thể đưa ra cùng yêu cầu đó. Điều đó đưa chúng ta đến đâu? Chúng ta trở lại điểm xuất phát khi tất cả đều bình đẳng.

Là một người theo chủ nghĩa tư bản, tôi không thích bình đẳng. Tôi muốn có lợi thế so với đối thủ. Vậy tôi bắt đầu nghĩ, "Chúng ta có thể làm gì hơi khác so với những gì AI đang nói với đối thủ của tôi?"

[4] https://www.yahoo.com/

Và chỉ như vậy, con người lại tham gia vào cuộc chơi!

Tôi nghĩ xu hướng đó sẽ tiếp tục, có thể mãi mãi.

Điều sẽ xảy ra, tôi dự đoán, là các công việc boilerplate đơn giản tồn tại hiện nay sẽ bị thu hẹp đáng kể. AI có thể giải nhiều bài toán đơn giản đó, và bạn không cần một đội ngũ kỹ sư lớn đằng sau chúng. Các bài toán mới lạ hơn vẫn sẽ cần nhiều công việc của con người.

Nhưng có lẽ không nhiều công việc như trước. AI có thể giúp đỡ theo nhiều cách, như đã nêu ở trên, vì vậy nó giúp đẩy nhanh mọi việc.

Quay ngược thời gian, hãy xem xét khi hầu hết các chương trình được viết bằng ngôn ngữ assembly[5] và cần nhiều kiến thức chuyên môn để viết những chương trình dễ lỗi này. Rồi trình biên dịch trở nên phổ biến và bây giờ không ai[6] viết bằng assembly nữa; những công việc đó đã bị xóa sổ, bị phá hủy bởi cách code nhanh hơn, dễ hơn mà trình biên dịch mang lại.

Tóm lại, tôi nghĩ chúng ta sẽ tiếp tục đẩy tiến, và AI sẽ trở thành công cụ rất hữu ích, nhưng chỉ khi có con người ở vị trí lái. Tôi nghĩ vậy. Chúng ta sẽ thấy.

> *"Và... Luôn nhìn vào mặt sáng của cuộc sống..."*
>
> — Người hát chính Crucifee, *Monty Python's Life of Brian*

10.5 Suy Ngẫm Về Chương

- So sánh những cách hữu ích và không hữu ích mà sinh viên có thể dùng AI.
- Điều gì xảy ra nếu bạn dùng AI sai cách với tư cách sinh viên?
- Mọi thứ khác nhau thế nào với AI trong công việc so với ở trường?
- Một số nguy hiểm của việc dùng AI trong công việc là gì?

[5] https://en.wikipedia.org/wiki/Assembly_language

[6] À, một số ít người có. Bạn nên thử cho vui. Đó là điều gì khác.

Index