Phân tích và thiết kế bài toán

Bởi:

Khoa CNTT ĐHSP KT Hưng Yên

CÁC BƯỚC CƠ BẢN ĐỂ GIẢI QUYẾT BÀI TOÁN

Xác định bài toán

Input → Process → Output (Dữ liệu vào → Xử lý → Kết quả ra)

Việc xác định bài toán tức là phải xác định xem ta phải giải quyết vấn đề gì?, với giả thiết nào đã cho và lời giải cần phải đạt những yêu cầu gì. Khác với bài toán thuần tuý toán học chỉ cần xác định rõ giả thiết và kết luận chứ không cần xác định yêu cầu về lời giải, đôi khi những bài toán tin học ứng dụng trong thực tế chỉ cần tìm lời giải tốt tới mức nào đó, thậm chí là tồi ở mức chấp nhận được. Bởi lời giải tốt nhất đòi hỏi quá nhiều thời gian và chi phí.

Ví dụ 2.1:

Khi cài đặt các hàm số phức tạp trên máy tính. Nếu tính bằng cách khai triển chuỗi vô hạn thì độ chính xác cao hơn nhưng thời gian chậm hơn hàng tỉ lần so với phương pháp xấp xỉ. Trên thực tế việc tính toán luôn luôn cho phép chấp nhận một sai số nào đó nên các hàm số trong máy tính đều được tính bằng phương pháp xấp xỉ của giải tích số.

Xác định đúng yêu cầu bài toán là rất quan trọng bởi nó ảnh hưởng tới cách thức giải quyết và chất lượng của lời giải. Một bài toán thực tế thường cho bởi những thông tin khá mơ hồ và hình thức, ta phải phát biểu lại một cách chính xác và chặt chẽ để hiểu đúng bài toán.

Ví dụ 2.2:

Bài toán: Một dự án có n người tham gia thảo luận, họ muốn chia thành các nhóm và mỗi nhóm thảo luận riêng về một phần của dự án. Nhóm có bao nhiêu người thì được trình lên bấy nhiêu ý kiến. Nếu lấy ở mỗi nhóm một ý kiến đem ghép lại thì được một bộ ý kiến triển khai dự án. Hãy tìm cách chia để số bộ ý kiến cuối cùng thu được là lớn nhất.

Phát biểu lại: Cho một số nguyên dương n, tìm các phân tích n thành tổng các số nguyên dương sao cho tích của các số đó là lớn nhất.

Trên thực tế, ta nên xét một vài trường hợp cụ thể để thông qua đó hiểu được bài toán rõ hơn và thấy được các thao tác cần phải tiến hành. Đối với những bài toán đơn giản, đôi khi chỉ cần qua ví dụ là ta đã có thể đưa về một bài toán quen thuộc để giải.

Tìm cấu trúc dữ liệu biểu diễn bài toán

Khi giải một bài toán, ta cần phải định nghĩa tập hợp dữ liệu để biểu diễn tình trạng cụ thể. Việc lựa chọn này tuỳ thuộc vào vấn đề cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu vào. Có những thuật toán chỉ thích ứng với một cách tổ chức dữ liệu nhất định, đối với những cách tổ chức dữ liệu khác thì sẽ kém hiệu quả hoặc không thể thực hiện được. Chính vì vậy nên bước xây dựng cấu trúc dữ liệu không thể tách rời bước tìm kiếm thuật toán giải quyết vấn đề.

Các tiêu chuẩn khi lựa chọn cấu trúc dữ liệu

- Cấu trúc dữ liệu trước hết phải biểu diễn được đầy đủ các thông tin nhập và xuất của bài toán
- Cấu trúc dữ liệu phải phù hợp với các thao tác của thuật toán mà ta lựa chọn để giải quyết bài toán.
- Cấu trúc dữ liệu phải cài đặt được trên máy tính với ngôn ngữ lập trình đang sử dụng

Đối với một số bài toán, trước khi tổ chức dữ liệu ta phải viết một đoạn chương trình nhỏ để khảo sát xem dữ liệu cần lưu trữ lớn tới mức độ nào.

Xác định thuật toán

Thuật toán là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy thao tác trên cấu trúc dữ liệu sao cho: Với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

Các đặc trưng của thuật toán

- Tính đơn nghĩa

Ở mỗi bước của thuật toán, các thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng, lộn xộn, tuỳ tiện, đa nghĩa.

Không nên lẫn lộn tính đơn nghĩa và tính đơn định: Người ta phân loại thuật toán ra làm hai loại: Đơn định (Deterministic) và Ngẫu nhiên (Randomized). Với hai bộ dữ liệu giống nhau cho trước làm input, thuật toán đơn định sẽ thi hành các mã lệnh giống nhau

và cho kết quả giống nhau, còn thuật toán ngẫu nhiên có thể thực hiện theo những mã lệnh khác nhau và cho kết quả khác nhau. Ví dụ như yêu cầu chọn một số tự nhiên x: $a \leq x \leq b$, nếu ta viết x = a hay x = b hay x = (a + b) div 2, thuật toán sẽ luôn cho một giá trị duy nhất với dữ liệu vào là hai số tự nhiên a và b. Nhưng nếu ta viết x = a + Random(b - a + 1) thì sẽ có thể thu được các kết quả khác nhau trong mỗi lần thực hiện với input là a và b tuỳ theo máy tính và bộ tạo số ngẫu nhiên.

- Tính dừng

Thuật toán không được rơi vào quá trình vô hạn, phải dừng lại và cho kết quả sau một số hữu hạn bước.

- Tính đúng

Sau khi thực hiện tất cả các bước của thuật toán theo đúng quá trình đã định, ta phải được kết quả mong muốn với mọi bộ dữ liệu đầu vào. Kết quả đó được kiểm chứng bằng yêu cầu bài toán.

- Tính phổ dụng

Thuật toán phải dễ sửa đổi để thích ứng được với bất kỳ bài toán nào trong một lớp các bài toán và có thể làm việc trên các dữ liệu khác nhau.

Tính khả thi

- Kích thước phải đủ nhỏ: Ví dụ: Một thuật toán sẽ có tính hiệu quả bằng 0 nếu lượng bộ nhớ mà nó yêu cầu vượt quá khả năng lưu trữ của hệ thống máy tính.
- Thuật toán phải chuyển được thành chương trình: Ví dụ một thuật toán yêu cầu phải biểu diễn được số vô tỉ với độ chính xác tuyệt đối là không hiện thực với các hệ thống máy tính hiện nay
- Thuật toán phải được máy tính thực hiện trong thời gian cho phép, điều này khác với lời giải toán (Chỉ cần chứng minh là kết thúc sau hữu hạn bước). Ví dụ như xếp thời khoá biểu cho một học kỳ thì không thể cho máy tính chạy tới học kỳ sau mới ra được.

Ví dụ 2.3:

Input: 2 số nguyên tự nhiên a và b không đồng thời bằng 0

Output: Ước số chung lớn nhất của a và b

Thuật toán sẽ tiến hành được mô tả như sau: (Thuật toán Euclide)

Bước 1 (Input): Nhập a và b: Số tự nhiên

Bước 2: Nếu b ≠ 0 thì chuyển sang bước 3, nếu không thì bỏ qua bước 3, đi làm bước 4

Bước 3: Đặt r = a mod b; Đặt a = b; Đặt b = r; Quay trở lại bước 2.

Bước 4 (Output): Kết luận ước số chung lớn nhất phải tìm là giá trị của a. Kết thúc thuật toán.

Khi mô tả thuật toán bằng ngôn ngữ tự nhiên, ta không cần phải quá chi tiết các bước và tiến trình thực hiện mà chỉ cần mô tả một cách hình thức đủ để chuyển thành ngôn ngữ lập trình. Viết sơ đồ các thuật toán đệ quy là một ví dụ.

Đối với những thuật toán phức tạp và nặng về tính toán, các bước và các công thức nên mô tả một cách tường minh và chú thích rõ ràng để khi lập trình ta có thể nhanh chóng tra cứu.

Đối với những thuật toán kinh điển thì phải thuộc. Khi giải một bài toán lớn trong một thời gian giới hạn, ta chỉ phải thiết kế tổng thể còn những chỗ đã thuộc thì cứ việc lắp ráp vào.

Tính đúng đắn của những mô-đun đã thuộc ta không cần phải quan tâm nữa mà tập trung giải quyết các phần khác.

Lập trình

Sau khi đã có thuật toán, ta phải tiến hành lập trình thể hiện thuật toán đó. Muốn lập trình đạt hiệu quả cao, cần phải có kỹ thuật lập trình tốt. Kỹ thuật lập trình tốt thể hiện ở kỹ năng viết chương trình, khả năng gỡ rối và thao tác nhanh. Lập trình tốt không phải chỉ cần nắm vững ngôn ngữ lập trình là đủ, phải biết cách viết chương trình uyển chuyển và phát triển dần dần để chuyển các ý tưởng ra thành chương trình hoàn chỉnh. Kinh nghiệm cho thấy một thuật toán hay nhưng do cài đặt vụng về nên khi chạy lại cho kết quả sai hoặc tốc độ chậm.

Thông thường, ta không nên cụ thể hoá ngay toàn bộ chương trình mà nên tiến hành theo phương pháp tinh chế từng bước (Stepwise refinement):

- Ban đầu, chương trình được thể hiện bằng ngôn ngữ tự nhiên, thể hiện thuật toán với các bước tổng thể, mỗi bước nêu lên một công việc phải thực hiện.
- Một công việc đơn giản hoặc là một đoạn chương trình đã được học thuộc thì ta tiến hành viết mã lệnh ngay bằng ngôn ngữ lập trình.
- Một công việc phức tạp thì ta lại chia ra thành những công việc nhỏ hơn để lại tiếp tục với những công việc nhỏ hơn đó.

Trong quá trình tinh chế từng bước, ta phải đưa ra những biểu diễn dữ liệu. Như vậy cùng với sự tinh chế các công việc, dữ liệu cũng được tinh chế dần, có cấu trúc hơn, thể hiện rõ hơn mối liên hệ giữa các dữ liệu.

Phương pháp tinh chế từng bước là một thể hiện của tư duy giải quyết vấn đề từ trên xuống, giúp cho người lập trình có được một định hướng thể hiện trong phong cách viết chương trình. Tránh việc mò mẫm, xoá đi viết lại nhiều lần, biến chương trình thành tờ giấy nháp.

Kiểm thử

Chạy thử và tìm lỗi

Chương trình là do con người viết ra, mà đã là con người thì ai cũng có thể nhầm lẫn. Một chương trình viết xong chưa chắc đã chạy được ngay trên máy tính để cho ra kết quả mong muốn. Kỹ năng tìm lỗi, sửa lỗi, điều chỉnh lại chương trình cũng là một kỹ năng quan trọng của người lập trình. Kỹ năng này chỉ có được bằng kinh nghiệm tìm và sửa chữa lỗi của chính mình.

Có ba loại lỗi:

- Lỗi cú pháp: Lỗi này hay gặp nhất nhưng lại dễ sửa nhất, chỉ cần nắm vững ngôn ngữ lập trình là đủ. Một người được coi là không biết lập trình nếu không biết sửa lỗi cú pháp.
- Lỗi cài đặt: Việc cài đặt thể hiện không đúng thuật toán đã định, đối với lỗi này thì phải xem lại tổng thể chương trình, kết hợp với các chức năng gỡ rối để sửa lại cho đúng.
- Lỗi thuật toán: Lỗi này ít gặp nhất nhưng nguy hiểm nhất, nếu nhẹ thì phải điều chỉnh lại thuật toán, nếu nặng thì có khi phải loại bỏ hoàn toàn thuật toán sai và làm lại từ đầu.

Xây dựng các bộ test

Có nhiều chương trình rất khó kiểm tra tính đúng đắn. Nhất là khi ta không biết kết quả đúng là thế nào?. Vì vậy nếu như chương trình vẫn chạy ra kết quả (không biết đúng sai thế nào) thì việc tìm lỗi rất khó khăn. Khi đó ta nên làm các bộ test để thử chương trình của mình.

Các bộ test nên đặt trong các file văn bản, bởi việc tạo một file văn bản rất nhanh và mỗi lần chạy thử chỉ cần thay tên file dữ liệu vào là xong, không cần gõ lại bộ test từ bàn phím. Kinh nghiệm làm các bộ test là:

Bắt đầu với một bộ test nhỏ, đơn giản, làm bằng tay cũng có được đáp số để so sánh với kết quả chương trình chạy ra.

Tiếp theo vẫn là các bộ test nhỏ, nhưng chứa các giá trị đặc biệt hoặc tầm thường. Kinh nghiệm cho thấy đây là những test dễ sai nhất.

Các bộ test phải đa dạng, tránh sự lặp đi lặp lại các bộ test tương tự.

Có một vài test lớn chỉ để kiểm tra tính chịu đựng của chương trình mà thôi. Kết quả có đúng hay không thì trong đa số trường hợp, ta không thể kiểm chứng được với test này.

Lưu ý rằng chương trình chạy qua được hết các test không có nghĩa là chương trình đó đã đúng. Bởi có thể ta chưa xây dựng được bộ test làm cho chương trình chạy sai. Vì vậy nếu có thể, ta nên tìm cách chứng minh tính đúng đắn của thuật toán và chương trình, điều này thường rất khó.

Tối ưu chương trình

Một chương trình đã chạy đúng không có nghĩa là việc lập trình đã xong, ta phải sửa đổi lại một vài chi tiết để chương trình có thể chạy nhanh hơn, hiệu quả hơn. Thông thường, trước khi kiểm thử thì ta nên đặt mục tiêu viết chương trình sao cho đơn giản, miễn sao chạy ra kết quả đúng là được, sau đó khi tối ưu chương trình, ta xem lại những chỗ nào viết chưa tốt thì tối ưu lại mã lệnh để chương trình ngắn hơn, chạy nhanh hơn. Không nên viết tới đâu tối ưu mã đến đó, bởi chương trình có mã lệnh tối ưu thường phức tạp và khó kiểm soát.

Việc tối ưu chương trình nên dựa trên các tiêu chuẩn sau:

- Tính tin cậy

Chương trình phải chạy đúng như dự định, mô tả đúng một giải thuật đúng. Thông thường khi viết chương trình, ta luôn có thói quen kiểm tra tính đúng đắn của các bước mỗi khi có thể.

- Tính uyển chuyển

Chương trình phải dễ sửa đổi. Bởi ít có chương trình nào viết ra đã hoàn hảo ngay được mà vẫn cần phải sửa đổi lại. Chương trình viết dễ sửa đổi sẽ làm giảm bớt công sức của lập trình viên khi phát triển chương trình.

- Tính trong sáng

Chương trình viết ra phải dễ đọc dễ hiểu, để sau một thời gian dài, khi đọc lại còn hiểu mình làm cái gì?. Để nếu có điều kiện thì còn có thể sửa sai (nếu phát hiện lỗi mới), cải tiến hay biến đổi để được chương trình giải quyết bài toán khác. Tính trong sáng của chương trình phụ thuộc rất nhiều vào công cụ lập trình và phong cách lập trình.

- Tính hữu hiệu

Chương trình phải chạy nhanh và ít tốn bộ nhớ, tức là tiết kiệm được cả về không gian và thời gian. Để có một chương trình hữu hiệu, cần phải có giải thuật tốt và những tiểu xảo khi lập trình. Tuy nhiên, việc áp dụng quá nhiều tiểu xảo có thể khiến chương trình trở nên rối rắm, khó hiểu khi sửa đổi. Tiêu chuẩn hữu hiệu nên dừng lại ở mức chấp nhận được, không quan trọng bằng ba tiêu chuẩn trên. Bởi phần cứng phát triển rất nhanh, yêu cầu hữu hiệu không cần phải đặt ra quá nặng.

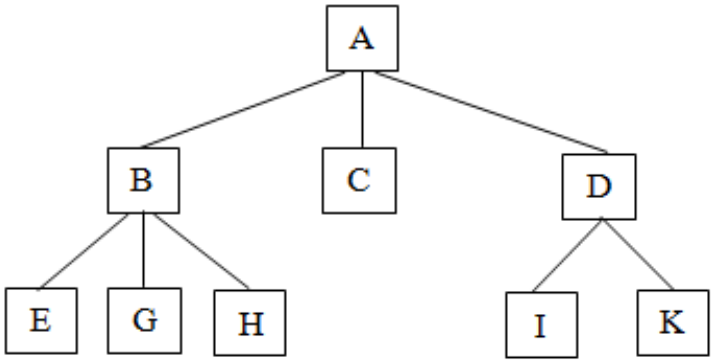
Từ những phân tích ở trên, chúng ta nhận thấy rằng việc làm ra một chương trình đòi hỏi rất nhiều công đoạn và tiêu tốn khá nhiều công sức. Chỉ một công đoạn không hợp lý sẽ làm tăng chi phí viết chương trình. Nghĩ ra cách giải quyết vấn đề đã khó, biến ý tưởng đó thành hiện thực cũng không dễ chút nào.

Những cấu trúc dữ liệu và giải thuật đề cập tới trong chuyên đề này là những kiến thức rất phổ thông, một người học lập trình không sớm thì muộn cũng phải biết tới. Chỉ hy vọng rằng khi học xong chuyên đề này, qua những cấu trúc dữ liệu và giải thuật hết sức mẫu mực, chúng ta rút ra được bài học kinh nghiệm: Đừng bao giờ viết chương trình khi mà chưa suy xét kỹ về giải thuật và những dữ liệu cần thao tác, bởi như vậy ta dễ mắc phải hai sai lầm trầm trọng: hoặc là sai về giải thuật, hoặc là giải thuật không thể triển khai nổi trên một cấu trúc dữ liệu không phù hợp. Chỉ cần mắc một trong hai lỗi đó thôi thì nguy cơ sụp đổ toàn bộ chương trình là hoàn toàn có thể, càng cố chữa càng bị rối, khả năng hầu như chắc chắn là phải làm lại từ đầu

MODUL HÓA VÀ VIỆC GIẢI QUYẾT BÀI TOÁN

Trong thực tế các bài toán được giải trên máy tính điện tử ngày càng nhiều và càng phức tạp. Các giải thuật ngày càng có qui mô lớn và khó thiết lập.

Để đơn giản hoá bài toán người ta tiến hành phân chia bài toán lớn thành các bài toán nhỏ. Có nghĩa là nếu bài toán lớn là một modul chính thì cần chia nó ra thành các modul con, đến lượt nó mỗi modul con này lại có thể chia tiếp ra thành các modul con khác ứng với các phần việc cơ bản mà người ta đã biết cách giải quyết. Việc tổ chức lời giải của bài toán có thể được thực hiện theo cấu trúc phân cấp như sau :



Chiến lược giải quyết bài toán theo kiểu như vậy gọi là chiến lược "chia để trị" (devide and conquare). Để thể hiện chiến lược này người ta sử dụng phương pháp thiết kế từ trên "*đỉnh - xuống*" (top - down design). Đó là cách phân tích tổng quát toàn bộ mọi vấn đề, xuất phát từ dữ kiện và các mục tiêu đề ra, để đề cập đến những công việc chủ yếu rồi sau đó mới đi dần vào giải quyết các phần cụ thể một cách chi tiết hơn(gọi đó là cách thiết kế từ khái quát đến chi tiết) .

Ví dụ : Chủ tịch hội đồng xét cấp học bổng của nhà trường yêu cầu chúng ta:

" Dùng máy tính điện tử để quản lý và bảo trì các hồ sơ về học bổng của các sinh viên ở diện được tài trợ, đồng thời thường kỳ phải lập các báo cáo tổng kết để đệ trình lên Bộ"

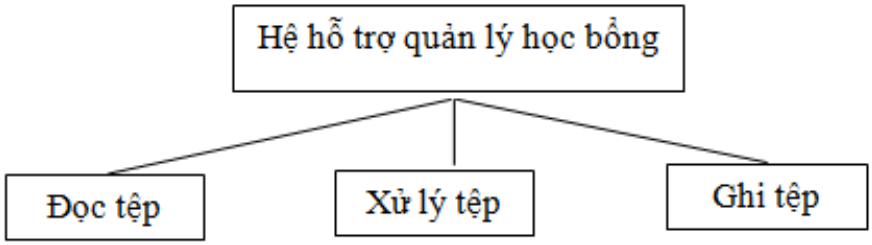
Như vậy trước hết ta phải hình dung được cụ thể hơn đầu vào và đầu ra của bài toán.

Có thể coi như ta đã có 1 tập hồ sơ (file) bao gồm các bản ghi (records) về các thông tin liên quan đến học bổng của sinh viên như : Mã SV, Điểm TB, điểm đạo đức, khoản tiền tài trợ. Và chương trình lập ra phải tạo điều kiện cho người sử dụng giải quyết được các yêu cầu sau:

1. Tìm lại và hiển thị được bản ghi của bất kỳ sinh viên nào tại thiết bị cuối (terminal) của người dùng.
2. Cập nhật (update) được bản ghi của một sinh viên cho trước bằng cách thay đổi điểm trung bình, điểm đạo đức, khoản tiền tài trợ nếu cần.
3. In bảng tổng kết chứa những thông tin hiện thời (đã được cập nhật mỗi khi có thay đổi) gồm số liệu, điểm trung bình, điểm đạo đức, khoản tiền tài trợ, nếu cần.

Xuất phát từ những nhận định trên, giải thuật xử lý phải giải quyết 3 nhiệm vụ chính như sau:

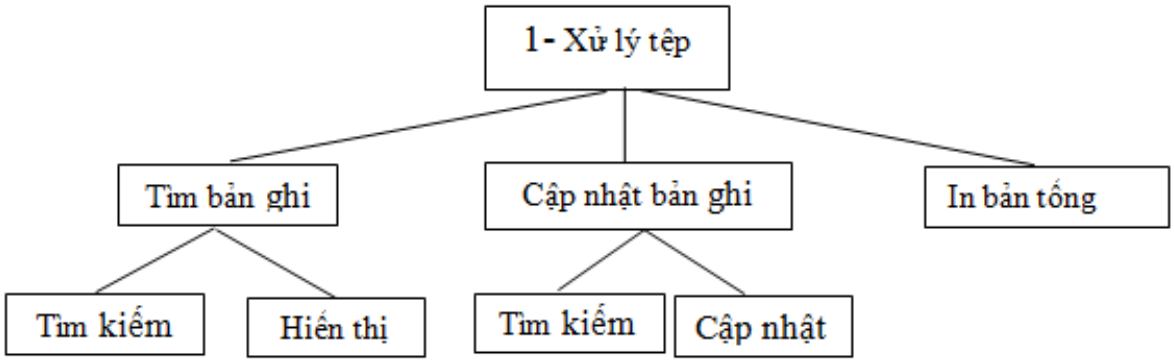
1. Những thông tin về sinh viên được học bổng, lưu trữ trên đĩa phải được đọc vào bộ nhớ trong để có thể xử lý (gọi là nhiệm vụ "đọc tệp")
2. Xử lý các thông tin này để tạo ra kết quả mong muốn (nhiệm vụ "xử lý tệp")
3. Sao chép những thông tin đã được cập nhật vào tệp trên đĩa để lưu trữ cho việc xử lý sau này(gọi là nhiệm vụ "ghi tệp").



Các nhiệm vụ ở mức đầu này tương đối phức tạp thường chia thành các nhiệm vụ con. Chẳng hạn, nhiệm vụ "xử lý tệp" sẽ được phân thành 3 nhiệm vụ con tương ứng giải quyết 3 yêu cầu chính được nêu trên:

1. Tìm lại bản ghi của một sinh viên cho trước.
2. Cập nhật thông tin trong bản ghi sinh viên.
3. In bảng tổng kết những thông tin về các sinh viên được học bổng.

Những nhiệm vụ con này cũng có thể lại được chia nhỏ thành các nhiệm vụ theo sơ đồ sau:



Cách thiết kế giải thuật theo kiểu top - down này sẽ giúp cho việc giải quyết bài toán được định hướng rõ ràng, dễ dàng thực hiện và nó chính là nền tảng cho việc lập trình cấu trúc.

PHƯƠNG PHÁP TINH CHỈNH DẦN TỪNG BƯỚC (Stepwise refinement)

Tinh chỉnh từng bước là phương pháp thiết kế giải thuật gắn liền với lập trình. Nó phản ánh tinh thần của quá trình modul hoá bài toán và thiết kế kiểu top - down.

Phương pháp này được tiến hành theo sơ đồ:

CTDL → CTDL lưu trữ → Cách cài đặt DL hợp lý → CTDL tiền định.

Trong quá trình thực hiện giải thuật ban đầu chương trình được thực hiện bằng ngôn ngữ tự nhiên phản ánh ý chính của công việc cần làm. Đến các bước sau những ý đó sẽ được chi tiết hoá dần dần tương ứng với những công việc nhỏ hơn. Ta gọi đó là các bước tinh chỉnh, sự tinh chỉnh này sẽ được hướng về phía ngôn ngữ lập trình mà ta đã chọn. Càng ở các bước sau lời lẽ đặc tả các công việc xử lý sẽ được thay thế bởi các câu lệnh hướng tới câu lệnh của ngôn ngữ lập trình.

Ví dụ 2.4: Giả sử ta muốn lập chương trình sắp xếp một dãy n số nguyên khác nhau theo thứ tự tăng dần.

Giải thuật có thể được phác thảo một cách thủ công đơn giản như sau:“ Coi các phần tử của dãy số như các phần tử của một véc tơ (có cấu trúc mảng một chiều) và dãy này được lưu trữ bởi một vec tơ lưu trữ gồm n từ máy kế tiếp ở bộ nhớ trong (a1, a2, . . ., an) mỗi từ a_i lưu trữ một phần tử thứ i ($1 \leq i \leq n$) của dãy số. Qui ước dãy số được sắp xếp rồi vẫn để tại chỗ cũ như đã cho.

Từ các số đã cho chọn ra một số nhỏ nhất, đặt nó vào cuối dãy đã được sắp xếp. Sau đó tiến hành so sánh với số hiện đang ở vị trí đó nếu như nó khác với số này thì phải tiến hành đổi chỗ. Công việc cứ lặp lại cho đến chỉ dãy số chưa được sắp xếp trở thành rỗng”.

Bước tinh chỉnh đầu tiên được thực hiện nhờ ngôn ngữ tựa C như sau:

*For(*int i =1*,* i ≤ n, i++)

{

+ Xét từ a i đến a n để tìm số nhỏ nhất a j

+ Đổi chỗ giữa ai và aj }

Các bước tiến hành:

+ B1: Xét dãy đã cho. Tìm số nguyên nhỏ nhất a_j trong các số từ a_i đến a_n

+ B2: Đổi chỗ giữa a_j và a_i

Nhiệm vụ đầu có thể được thực hiện bằng cách:

" *Thoạt tiên coi ai là "số nhỏ nhất" tạm thời; lần lượt so sánh ai với ai+1,ai+2, . . . Khi đã so sánh với an rồi thì số nhỏ nhất sẽ được xác định*."

Để xác định ta phải chỉ ra vị trí của nó, hay nói cách khác là nắm được chỉ số của phần tử ấy thông qua một khâu trung gian:

{Bước tinh chỉnh 1}

j =i;

*For(*int k =*j*+1, k ≤ n, k++)

if (a k < a j) j=k;

{Bước tinh chỉnh 2}

B = a i ; a i = a j ; a j = B;

Sau khi đã chỉnh lại cách viết biến chỉ số cho đúng với qui ước ta có chương trình sắp xếp hoàn chỉnh viết dưới dạng thủ tục như sau:

Void Sort(A,n)

{các biến i,j,k kiểu nguyên; biến trung gian B cùng kiểu A}

1. *For(*int i =1, i ≤ n, i++)

{

2- {Chọn số nhỏ nhất}

j =i;

*For(*int k =*j*+1, k ≤ n, k++)

if (A[k] < A[j]) j = k;

1. {Đổi chỗ} B = A[i]; A[i]= A[j]; A[j] = B;

}

Ví dụ 2: Cho ma trận vuông n × n các số nguyên. Hãy in ra các phần tử thuộc đường chéo song song với đường chéo chính theo thứ tự tính từ phải sang trái.

$$A = \begin{pmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{pmatrix}$$

Chọn cách in từ phải sang trái ta có kết quả:

a14

a13 a24

a12 a23 a34

a11 a22 a33 a44

a21 a32 a43

a31 a42

a41

Nửa tam giác trên các cột giảm dần từ n → 1, đưa ra các phần tử thuộc đường chéo ứng với cột j

Nửa tam giác dưới các hàng tăng từ 2 → n. Với mỗi hàng như vậy ta phải đưa ra các phần tử thuộc đường chéo tương đương với hàng i đã cho.

Ta có thể phác hoạ giải thuật như sau:

1. Nhập cấp ma trận n
2. Nhập các phần tử của ma trận A[i,j]
3. In các đường chéo song song với đường chéo chính.

Hai nhiệm vụ (1) và (2) có thể dễ dàng thể hiện bằng Pascal:

1. Cin>>*n;*

1. *for (i = 1, i <= n, i++)*

for (j = 1, j <= n,j++)

Cout <<a[i] [j];

Nhiệm vụ 3 cần phải được phân tích rõ ràng hơn:

Về đường chéo ta có thể phân ra làm 2 loại:

+ Đường chéo ứng với cột từ n đến 1

+ Đường chéo ứng với hàng từ 2 đến n

Cho nên ta tách ra 2 nhiệm vụ con là:

1. *for (j = n, j> = 1, j--)*

in dường chéo ứng với cột j

3.2. For (i = 2, i<=n, i++)

in đường chéo ứng với hàng i

Tới đây phải chi tiết hơn công việc " in đường chéo ứng với cột j"

Với j = n thì in một phần tử bằng cột j

Với j = n - 1 thì in 2 phần tử hàng 1 cột j

hàng 2 cột j+1

Với j = n - 2 thì in 3 phần tử hàng 1 cột j

hàng 2 cột j+1

hàng 3 cột j+2

Ta nhận thấy số lượng các phần tử được in chính là (n - j + 1), còn phần tử được in chính là A[i, j + (i - 1)] với i nhận giá trị từ 1 tới (n - j + 1)

Vậy 3.1 có thể tinh chỉnh tiếp tác vụ in đường chéo ứng với cột j thành:

```
For ( i = 1, i<= (n - j + 1), i++)
    Cout<<a[i],[j + (i - 1)]<<endl
```

Ta tận dụng khả năng của Pascal để in mỗi phần tử trong một khoảng cách 8 kí tự và mỗi đường chéo được in trên một dòng, sau đó để cách một dòng trống.

1. tương tự

For (j = 1,j<= (n - i + 1),j++)

Cout<<a ? i + (j - 1)] [j ? ,<<endl

Toàn bộ giải thuật này có thể được thể hiện bằng ngôn ngữ C như sau:

```
{
do
  Cout<< "Cho biết kích thước ma trận";Cin>>n;
while  (0<n) and (n<=max);
```

//Nhập các pt của mảng

clrscr();

//In đường chéo ma trận

For (j=n,j<=n, j--)

{

```
        For ( i = 1, i<= (n - j + 1), i++)
         Cout<<a[i],[j + (i - 1)]<<endl
        getch();
       }
For ( i = 2, i<=n, i++)
   {
    For (j = 1,j<= (n - i + 1),j++)
      Cout<<a[i + (j - 1)] [j],<<endl
       getch();
   }
}
```