MỤC LỤC

Chương I. MỞ ĐẦU

Algorithms + Data Structures = Programs

Niklaus Wirth.

Lưu trữ và xử lý thông tin là hạt nhân của tất cả các chương trình máy tính. Việc lập trình giải quyết một bài toán thực tế cần phải qua nhiều bước, nhưng hai bước quan trọng nhất, không thể tách rời, chính là tìm mô hình lưu trữ thông tin (xây dựng cấu trúc dữ liệu) và mô hình xử lý thông tin (tìm giải thuật).

Chương đầu tiên này sẽ nói về quy trình thiết kế một chương trình máy tính, cách xây dựng cấu trúc dữ liệu và cách phân tích đánh giá giải thuật. Những cấu trúc dữ liệu và giải thuật cụ thể sẽ được phân tích rõ hơn trong các chương sau.

Bài 1. Từ bài toán tới chương trình

Có nhiều công đoạn phải thực hiện khi viết chương trình máy tính để giải quyết một bài toán thực tế: Mô hình hóa bài toán, làm rõ các ràng buộc và đặc điểm kỹ thuật, tìm thuật giải, cài đặt, kiểm thử và đánh giá…Trong phần đầu tiên, chúng ta sẽ trình bày sơ lược các bước tiến hành khi phải lập trình giải quyết một bài toán thực tế.

1.1. Mô hình hóa bài toán

Đa số các bài toán thực tế đều không thể mô tả một cách đơn giản và chính xác. Có những bài toán không thể giải quyết được bằng máy tính với trình độ kỹ thuật hiện tại. Ngay cả với những bài toán có thể giải quyết được thì tính khả thi của lời giải luôn phụ thuộc vào một vài tham số (hay ràng buộc) nào đó, chẳng hạn như kích thước dữ liệu, thời gian thực hiện trên máy tính, … Đôi khi ta chỉ có thể biết được những tham số này qua các thí nghiệm.

Nếu mô hình hóa được bài toán, tức là xác định xem phải giải quyết vấn đề gì?, với giả thiết nào đã cho và lời giải cần phải đạt những yêu cầu gì, chúng ta có thể bắt tay vào việc tìm giải pháp. Cách dễ dàng nhất là tìm và sử dụng những chương trình/lời giải sẵn có, nhưng ngay cả khi mô hình của bài toán chưa có lời giải, ít nhất chúng ta cũng có thể khảo sát xem mô hình bài toán có những đặc điểm gì đã biết để từ đó xây dựng một giải pháp tốt.

Hầu hết các công cụ toán học đều được huy động trong việc mô hình hóa và phát biểu bài toán. Xác định đúng mô hình bài toán là rất quan trọng bởi nó ảnh hưởng tới cách thức giải quyết và chất lượng của lời giải. Một vấn đề thực tế thường cho bởi những thông tin khá mơ hồ và hình thức, ta phải xây dựng mô hình bài toán một cách chính xác và chặt chẽ để hiểu và tìm ra giải pháp tốt.

> Ví dụ:
> 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.
> Mô hình hóa: 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. Một vấn đề cần lưu ý nữa là yêu cầu về chất lượng lời giải: Có những bài toán thực tế yêu cầu lời giải tuyệt đối chính xác, trong khi có những bài toán chỉ yêu cầu lời giải tốt tới mức độ nào đó (tối ở mức chấp nhận

được), khi mà lời giải chính xác tuyệt đối đòi hỏi quá nhiều chi phí về thời gian và bộ nhớ.

1.2. Tìm thuật toán

Ngay khi chúng ta có một mô hình toán học cho vấn đề cần giải quyết, chúng ta có thể bắt tay vào tìm giải pháp dưới dạng *thuật toán* hay *giải thuật*. Thuật toán là một hệ thống chặt chẽ và rõ ràng các chỉ thị nhằm xác định một dãy thao tác trên dữ liệu vào sao cho: Bất kể dữ liệu vào như thế nào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

1.2.1. 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, tùy 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 vào giống nhau, 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: Để chỉ ra một số tự nhiên $x: a \le x \le b$, 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
>
> $$x := a + \text{Random}(b - a + 1)$$
>
> thì sẽ có thể thu được các kết quả khác nhau trong mỗi lần thực hiện với dữ liệu vào là a và b tùy 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

Tính khả thi của thuật toán có thể kiểm tra bằng các điều kiện:

- 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 khóa biểu cho một học kỳ thì không thể cho phép máy tính chạy tới học kỳ sau mới ra được.

❑ Mô tả thuật toán

Có ba phương pháp chính mô tả thuật toán:

- Dùng ngôn ngữ tự nhiên (Natural Language): Diễn giải thuật toán bằng các bước thực hiện và nêu rõ nhiệm vụ của các bước. Mô tả thuật toán bằng ngôn ngữ tự nhiên có ưu điểm là dễ đọc, dễ hiểu nhưng đôi khi dễ gây hiểu lầm vì ngôn ngữ tự nhiên thường không đủ chặt chẽ.
- Dùng mã giả (Pseudo code): Mượn những từ khóa, cú pháp của một ngôn ngữ lập trình nào đó để mô tả thuật toán, khác với việc lập trình một chương trình hoàn thiện, khi viết bằng mã giả, ta không cần quá chi tiết trong việc mô tả một số công đoạn, mà chỉ cần đặc tả nhiệm vụ của công đoạn đó. Dùng giả mã có ưu điểm là nếu mã giả dựa trên chính ngôn ngữ lập trình sẽ được sử dụng thì việc lập trình cụ thể trở nên rất dễ dàng, tuy nhiên nếu lập trình viên không biết gì về ngôn ngữ gốc dùng làm mã giả thì có thể gặp nhiều khó khăn trong việc đọc hiểu thuật toán.
- Dùng lưu đồ thuật giải hay sơ đồ khối (Flow chart): Đây là cách cổ điển nhất để mô tả thuật toán. Lưu đồ thuật giải bao gồm các khối và các mũi tên chỉ tiến trình thực hiện, mỗi khối có một chức năng riêng và tùy theo chức năng, có quy định về hình dạng các khối. Khi thiết kế một phần mềm hay thuật toán phức tạp, người ta thường chỉ sử dụng sơ đồ khối để chỉ ra mô hình của tiến trình xử lý, còn việc mô tả chi tiết hoạt động của mỗi khối thì tùy theo tính phức tạp của vấn đề, người ta có thể dùng ngôn ngữ tự nhiên hoặc mã giả.

Ví dụ: Thuật toán Euclide tính ước số chung lớn nhất của hai số nguyên dương.

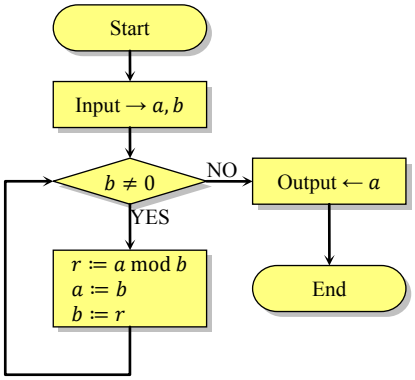
Mô tả bằng ngôn ngữ tự nhiên:

Nhập vào hai số nguyên dương a, b, chừng nào còn thấy $b \neq 0$ thì thay $(a, b) \coloneqq (b, a \bmod b)$. Trả về giá trị của a là ước số chung lớn nhất của hai số ban đầu.

Mô tả bằng mã giả dựa trên PASCAL:

```
Input → a, b;
while b ≠ 0 do
  (a, b) := (b, a mod b);
Output ← a;
```

Mô tả bằng lưu đồ thuật giải:



1.3. Tìm cấu trúc dữ liệu cài đặt thuật toán

Chương trình con (sub routines) là những yếu tố quan trọng của lập trình cấu trúc. Chương trình con thực chất là sự tổng quát hóa của các *phép biến đổi* (hay thao tác – operators) trên dữ liệu. Thay vì bị giới hạn trong các phép toán cộng, trừ, nhân, chia, … của ngôn ngữ và môi trường lập trình, lập trình viên có thể tự do định nghĩa các toán tử dưới dạng chương trình con để áp dụng cho các kiểu dữ liệu phức hợp. Ví dụ như viết các hàm xử lý phép toán cộng, trừ, nhân, chia trên tập các đa thức.

Một ưu điểm khác của chương trình con là chúng có thể sử dụng để giản lược những thao tác của toàn bộ thuật toán. Bằng cách đưa những chương trình con vào một phần riêng chứa những thao tác nhất định để thực hiện một việc nào đó, lập trình viên có thể dễ dàng tìm đến những đoạn mã cần sửa đổi khi cần thiết.

Với những tính chất như vậy của chương trình con, chúng ta có thêm một khái niệm mới về các *kiểu dữ liệu trừu tượng* (Abstract Data Type), có thể coi một kiểu dữ liệu trừu tượng là một mô hình toán học với những thao tác định nghĩa trên mô hình đó. Kiểu dữ liệu trừu tượng có thể không tồn tại trong ngôn ngữ lập trình mà chỉ dùng để tổng quát hóa hoặc tóm lược những thao tác sẽ được thực hiện trên dữ liệu.

Ví dụ: Ta có thể định nghĩa một kiểu dữ liệu trừu tượng tên là TPriorityQueue (hàng đợi có độ ưu tiên) dùng để chứa một tập hợp số. Trên kiểu dữ liệu này có ba thao tác:

- $Insert(v)$: Đưa thêm một giá trị v vào trong tập hợp.
- $Extract$: Trả về phần tử nhỏ nhất trong tập hợp và loại bỏ phần tử đó ra khỏi tập hợp.
- $Update(OldValue, NewValue)$: Thay đổi giá trị của phần tử có giá trị $OldValue$ thành $NewValue$.

Kiểu dữ liệu trừu tượng khi lập trình sẽ được cài đặt bằng các *cấu trúc dữ liệu (data structures)*. Như ví dụ trên ta có thể cài đặt hàng đợi có độ ưu tiên bằng một mảng các số nguyên và viết các hàm/thủ tục $Insert$, $Extract$, $Update$ tương ứng.

Cần phân biệt rõ những khái niệm: *Kiểu dữ liệu* (Data Type), *Cấu trúc dữ liệu* (Data Structure) và *Kiểu dữ liệu trừu tượng* (Abstract Data Type).

Với một ngôn ngữ lập trình và một môi trường lập trình, kiểu dữ liệu của một biến là một tập các giá trị mà biến đó có thể nhận, đồng thời có một số phép toán đã được định nghĩa sẵn trên tập các giá trị đó. Ví dụ trong PASCAL, một biến kiểu Boolean có thể nhận giá trị True hoặc False, nhưng không thể nhận bất cứ một giá trị nào khác. Với kiểu dữ liệu Boolean, có những phép toán được định nghĩa sẵn như AND, OR, NOT, XOR. Tuy các kiểu dữ liệu cơ sở là phụ thuộc vào môi trường lập trình cung cấp (Integer, Boolean, …), nhưng cũng có những quy tắc để định nghĩa những kiểu dữ liệu phức hợp từ những kiểu dữ liệu cơ sở (mảng, bản ghi, …).

Một kiểu dữ liệu trừu tượng là một mô hình toán và các toán tử được định nghĩa trên mô hình. Có thể thiết kế thuật toán dựa trên các kiểu dữ liệu trừu tượng, nhưng khi cài đặt thuật toán đó thành chương trình, l ập trình viên phải tìm cách biểu diễn các kiểu dữ liệu trừu tượng bằng các cấu trúc dữ liệu: bao gồm các kiểu dữ liệu, các thao tác được môi trường lập trình cung cấp và những thao tác tự

Trong kỹ thuật lập trình cấu trúc (Structural Programming), kiểu dữ liệu trừu tượng thường được cài đặt bằng các biến, các thủ tục và hàm xử lý trên các biến đó. Trong kỹ thuật lập trình hướng đối tượng (Object-Oriented Programming), kiểu dữ liệu trừu tượng thường được cài đặt bằng các lớp (class), các thuộc tính và phương thức tác động lên chính đối tượng hay một vài thuộc tính của đối tượng.

Việc lựa chọn cấu trúc dữ liệu chính là tìm cách tổ chức dữ liệu một cách hợp lý. Việc này tùy thuộc vào thuật toán 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, chúng ta nên mô tả thuật toán dựa trên các kiểu dữ liệu trừu tượng, và tùy theo một kiểu dữ liệu trừu tượng cần lưu trữ những thông tin gì và cần có những phép toán gì, chúng ta sẽ xây dựng một cấu trúc dữ liệu phù hợp để cài đặt kiểu dữ liệu trừu tượng đó sao cho hợp lý nhất.

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

1.4. Lập trình

Sau khi đã có thuật toán, ta phải tiến hành lập trình cài đặt 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 làđ ủ, phải biết cách viết chương trình uyển chuyển, khôn khéo 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 hóa ngay toàn bộ chương trình mà nên tiến hành theo *phương pháp tinh chế từng bước* (*Stepwise refinement*):

- Ban đầu, chương trình đư ợc viết để 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 đư ợc 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 đư ợc một định hướng thể hiện trong phong cách viết chương trình. Tránh việc mò mẫm, xóa đi viết lại nhiều lần, biến chương trình thành t ờ giấy nháp.

1.5. Kiểm thử

1.5.1. Chạy thử và tìm lỗi

Chương trình là do con ng ư ời viết ra, mà đ ã là con ng ư ời thì ai c ũng có th ể 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 bị 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.

1.5.2. 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 những 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.

1.6. 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 đã tối ưu hóa thường phức tạp và khó kiểm soát. Việc tối ưu chương trình thường dựa vào các tiêu chí sau đây:

1.6.1. Tính tin cậy

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

1.6.2. 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 trong việc nâng cấp và bảo trì.

1.6.3. 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. Nếu có điều kiện thì chúng ta 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.

1.6.4. 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. Tốc độ phần cứng phát triển rất nhanh, sử dụng tiểu xảo để tăng tốc độ chương trình không phải là một biện pháp nên lạm dụng.

1.7. Tóm tắt

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.

Những bài toán, thuật toán, cấu trúc dữ liệu đề cập tới trong cuốn sách này là những kiến thức rất phổ thông, một người học lập trình không sớm thì muộn cũng phải biết tới. Hy vọng rằng khi học xong chuyên đề này, qua việc cài đặt những cấu trúc dữ liệu và giải thuật hết sức mẫu mực, chúng ta có thể tự rút ra cho mình những kinh nghiệm lập trình mà những kinh nghiệm đó không thể học được qua sách vở.

Bài 2. Vai trò của thuật toán trong tính toán

Chúng ta đã có những hiểu biết cơ bản về khái niệm thuật toán. Trong phần này, chúng ta đi sâu vào việc phân tích những chi tiết sau:

- Thế nào là một thuật toán?
- Tại sao việc học các kiến thức về thuật toán là cần thiết?
- Thuật toán đóng vai trò gì trong các chương trình máy tính?

2.1. Thế nào là một thuật toán

Sau khi mô hình hóa bài toán, thì bước kế tiếp phải làm là thiết kế thuật toán giải quyết bài toán đó. Nhắc lại định nghĩa về thuật toán là một hệ thống chặt chẽ và rõ ràng các chỉ thị nhằm xác định một dãy thao tác trên dữ liệu vào sao cho: Bất kể dữ liệu vào (input) như thế nào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta thu được một kết quả (output) mong muốn.

2.2. Những loại bài toán nào có thuật toán để giải quyết

Những loại bài toán có thể giải quyết bằng thuật toán nằm trong phạm vi hiểu biết của con người, phạm vi đó rất nhỏ bé so với những gì chúng ta chưa biết. Ta có thể kể ra một vài ví dụ điển hình về các bài toán thực tế có thể giải quyết bằng thuật toán:

Dự án bộ gien người (Human Genome Project) có nhiệm vụ lập bản đồ của gần như tất cả 20000-25000 gien trong chuỗi DNA của người, mã hóa chúng bằng các dãy 3 tỉ ký hiệu lưu trữ trong cơ sở dữ liệu và phát triển các công cụ phân tích dữ liệu. Để xử lý một lượng dữ liệu lớn đến như vậy, bất cứ vấn đề nào cũng liên quan đến hàng loạt thuật toán hiệu quả để tiết kiệm thời gian, công sức và tiền bạc. Dự án kết thúc vào năm 2003 (nửa thế kỷ sau khi Watson và Crick đưa ra mô tả về cấu trúc DNA) với phần lớn mục tiêu đã định được hoàn thành.

Mạng Internet cho chúng ta một lượng thông tin khổng lồ. Nhưng để truy cập và sử dụng những thông tin đó, phải có những công cụ xử lý dữ liệu hiệu quả. Ví dụ tìm đường nhanh nhất để truyền tải dữ liệu, xây dựng các máy tìm kiếm để nhanh chóng lọc ra những thông tin cần tìm…, việc xây dựng mỗi công cụ như vậy kéo theo việc phát triển hàng loạt những thuật toán xử lý văn bản, nén, tách và phân loại thông tin, …

Thương mại điện tử cho phép việc trao đổi sản phẩm và dịch vụ trở nên nhanh chóng và đơn giản hơn. Cùng với thương mại điện tử, khả năng bảo mật thông tin cá nhân như mã số thẻ tín dụng, mật mã, … trở nên hết sức quan trọng. Chính vì nhu cầu đó, những thuật toán kiểm tra tính toàn vẹn thông tin trên đường truyền dữ liệu, những thuật toán mã hóa

công khai và chữ ký điện tử đã được phát triển dựa trên các lý thuyết và thuật toán số học.

Mỗi vấn đề kể trên liên quan tới nhiều chuyên ngành của khoa học máy tính: Tin sinh học (BioInformatics), Lý thuyết đồ thị (Graph theory), Trích chọn thông tin (Information extraction), Tìm kiếm thông tin (Information retrieval), Xử lý ngôn ngữ tự nhiên (Natural language processing), An toàn dữ liệu (Data security), Mã hóa công khai (Public-key cryptography), Chữ ký điện tử (Digital signatures). Việc đi sâu vào các lĩnh vực này vượt quá khuôn khổ của cuốn sách, trong giáo trình này, chúng ta chỉ tìm hiểu một số bài toán cơ bản và vài thuật toán phổ biến, có thể coi là nguyên tố cấu thành nên những thuật toán phức tạp hơn. Ngoài việc tìm hiểu những bài toán, thuật toán cụ thể, chúng ta sẽ làm quen với kỹ năng tiếp cận và giải quyết vấn đề, kỹ thuật thiết kế và phân tích thuật toán, và (một chút) kỹ thuật lập trình nữa.

2.3. Những loại bài toán khó

Phần lớn những thuật toán đề cập tới trong cuốn sách này là những thuật toán hiệu suất cao. Hiệu suất của một thuật toán được đo bởi tốc độ, nghĩa là thời gian cần thiết để thuật toán cho kết quả. Tuy vậy, có những bài toán mà lời giải hiệu quả hiện chưa biết, lớp bài toán này được gọi với cái tên NP-Complete.

Lớp bài toán NP-Complete có những tính chất thú vị, vì thế chúng thu hút được rất nhiều sự quan tâm của các nhà khoa học máy tính:

- Mặc dù chưa có lời giải hiệu quả cho bất kỳ bài toán nào trong lớp bài toán NP-complete, cũng chưa ai chứng minh được sự không tồn tại của lời giải hiệu quả cho những bài toán đó.
- Nếu chỉ ra được một thuật toán hiệu quả cho một bài toán NP-Complete thì tất cả các bài toán NP-Complete đều có lời giải hiệu quả.

Việc đưa ra một thuật toán hiệu quả cho một bài toán NP-Complete cũng khó như việc chứng minh sự không tồn tại thuật toán hiệu quả giải quyết những bài toán đó. Chúng ta cũng sẽ khảo sát một vài bài toán NP-Complete để khi gặp một bài toán thực tế có thể quy về bài toán NP-Complete, chúng ta có thể lựa chọn trong những cách giải quyết:

- Thay vì tiêu tốn thời gian tìm một thuật toán đúng và hiệu quả, chúng ta dùng một thuật toán xấp xỉ, có thể cho kết quả không phải tốt nhất, nhưng ở mức chấp nhận được.
- Hoặc, cũng không loại trừ khả năng, chúng ta cố gắng tìm được một lời giải hiệu quả cho một bài toán NP-Complete để trở thành người đầu tiên giải quyết được một trong những thách thức lớn nhất của thế kỷ 20 dành cho thế kỷ 21.

2.4. Tại sao lại phải học về thuật toán

Nếu đến một lúc nào đó, tốc độ máy tính trở nên vô hạn và bộ nhớ máy tính không mất tiền mua thì còn lý do gì để học về thuật toán nữa hay không?. Câu trả lời là có.

Khi gặp một vấn đề, bạn vẫn phải tìm một thuật toán để giải quyết vấn đề đó, và vẫn phải chứng minh được tính đúng đắn và tính dừng của thuật toán. Chỉ có điều, nếu không tính đến chi phí về không gian (bộ nhớ) và thời gian (tốc độ), bạn có thể chọn một thuật toán dễ cài đặt nhất để viết chương trình.

Dĩ nhiên, tốc độ của máy tính có thể ngày càng nhanh hơn, nhưng không thể trở thành vô hạn. Bộ nhớ máy tính có thể trở nên rất rẻ, nhưng không bao giờ là thứ miễn phí. Vì vậy ngoài việc chứng minh tính đúng đắn và tính dừng của thuật toán, việc đánh giá tính hiệu quả của thuật toán luôn luôn được quan tâm trước khi viết chương trình. Chẳng hạn nếu so sánh thuật toán tìm kiếm tuần tự (Cần trung bình n phép so sánh) và thuật toán tìm kiếm nhị phân (cần trung bình $\lg n$ phép so sánh) khi tìm kiếm một giá trị trong dãy có $n = 2^{64}$ phần tử đã được sắp xếp. Nếu máy tính có thể xử lý được 1 tỉ phép so sánh trong một giây, thì thuật toán tìm kiếm tuần tự sẽ mất trung bình khoảng 292 năm để thực hiện một lệnh tìm kiếm, trong khi thuật toán tìm kiếm nhị phân chỉ cần có 64 nano giây mà thôi. Nếu tốc độ của máy tính có thể nhanh hơn 1000 lần thì thuật toán tìm kiếm tuần tự vẫn là không khả thi vì thời gian thực hiện phải tính bằng tháng.

2.5. Chứng minh thuật toán

Một bài toán có thể có nhiều thuật toán để giải quyết. Nhưng cho dù chọn thuật toán nào để lập trình, trước hết ta phải chỉ ra được hai điều kiện: Thuật toán đó phải đúng và thuật toán đó phải dừng.

Xét một bài toán cụ thể:

Cho dãy số $A = (a_1, a_2, \ldots, a_n)$ sắp xếp theo thứ tự không giảm: $a_1 \leq a_2 \leq \cdots \leq a_n$ và một số v, hãy cho biết số v có mặt trong dãy A không, và nếu số v có trong dãy A, hãy cho biết chỉ số phần tử đầu tiên của dãy A có giá trị bằng v.

Xét một thuật toán để giải quyết bài toán này: Thuật toán tìm kiếm nhị phân:

```
Input → a[1..n], v;
L := 1;
H := n;
while L ≤ H do
  begin
    M := (L + H) div 2;
    if a[M] < v then L := M + 1
    else H := M - 1;
  end;
if (L > n) or (a[L] ≠ v) then Output ← "Not Found"
else Output ← L;
```

Để chứng minh tính đúng đắn và tính dừng của thuật toán tìm kiếm nhị phân, ta có nhận xét: Giả sử rằng có thêm hai phần tử $a_0 = -\infty$ và $a_{n+1} = +\infty$, khi đó ta luôn có:

$$a_{L-1} < v \leq a_{H+1}$$

Tính chất này gọi là tính *bất biến vòng lặp* (*Loop invariant*) của thuật toán tìm kiếm nhị phân. Để chứng minh tính bất biến vòng lặp, ta chỉ ra rằng nó thoả mãn 3 tính chất:

- Initialization: Tính chất này đúng trước khi bước vào vòng lặp
  - Trước khi bước vào vòng lặp, $L = 1$ và $H = n$, $a_0 = -\infty$ và $a_{n+1} = +\infty$, rõ ràng tính chất này đúng: $-\infty = a_0 < v < a_{n+1} = +\infty$
- Maintenance: Nếu tính chất này đúng trước một lần lặp, thì sau lần lặp đó tính chất này vẫn đúng
  - Trước mỗi bước lặp, ta có $a_{L-1} < v \leq a_{H+1}$, xét vị trí M nằm giữa L và H: $M \coloneqq (L + H)$ div 2. Nếu $a_M < v$, việc đặt lại $L \coloneqq M + 1$ vẫn đảm bảo $a_{L-1} < v$, còn nếu $v \leq a_M$, việc đặt lại $H \coloneqq M - 1$ vẫn đảm bảo $v \leq a_H$. Vậy tính bất biến vòng lặp được duy trì sau mỗi bước lặp
- Termination: Khi vòng lặp kết thúc, tính bất biến vòng lặp này cho ta một kết quả hữu dụng để có thể chỉ ra tính đúng đắn của thuật toán.
  - Sau mỗi bước lặp, hiệu số $H - L$ sẽ giảm đi và vòng lặp sẽ kết thúc khi chỉ số L đứng sau chỉ số H: $L > H$
  - Mặt khác, do tính bất biến vòng lặp và thứ tự tăng dần của dãy A, từ $a_{L-1} < v \leq a_{H+1}$, suy ra chỉ số $L - 1$ phải đứng trước chỉ số $H + 1$: $L - 1 < H + 1$
  - Hai điều này chỉ ra rằng $H < L < H + 2$, tức là vòng lặp sẽ kết thúc khi $L = H + 1$. Tính hữu dụng của sự bất biến vòng lặp sẽ được sử dụng tại đây để chứng minh tính đúng đắn của thuật toán tìm kiếm nhị phân. $L = H + 1$ và $a_{L-1} < v \leq a_{H+1}$, ta suy ra $a_{L-1} < v \leq a_L$. Kết hợp với thứ tự tăng dần của dãy A, suy ra a_L là phần tử đầu tiên của dãy A có giá trị lớn hơn hay bằng v. Tức là nếu $L \leq n$ và $a_L = v$ thì số v có xuất hiện trong dãy A và a_L chính là phần tử đầu tiên của dãy A có giá trị bằng v.

Có nhiều cách để chứng minh tính đúng đắn và tính dừng của một thuật toán, nhưng cách chứng minh bằng tính bất biến vòng lặp là phổ biến nhất vì tư tưởng của nó gần với quy nạp toán học. Chúng ta không phải lần theo từng bước lặp cụ thể (thông thường chúng ta không biết trước số lần lặp) mà chỉ cần chứng minh tính bất biến vòng lặp và chứng minh tính dừng của vòng lặp là đủ.

Bài tập 2-1.

Hãy chỉ ra những ví dụ thực tế mà ở đó, những bài toán sau đây hiện hữu: Sắp xếp dãy số, nhân ma trận, tìm bao lồi của một tập hợp điểm.

Bài tập 2-2.

Ngoài tốc độ của chương trình, hãy liệt kê thêm một vài tiêu chí cần phải tính đến khi cài đặt thuật toán trên thực tế.

Bài tập 2-3.

Xét bài toán tính giá trị đa thức $p(x) = \sum_{i=0}^{d} a_i x^i$ và hai thuật toán:

Thuật toán tính trực tiếp:

```
Input → a[0..d], x;
S := 0;
for i := 0 to d do
  begin
    t := a[i];
    for j := 1 to d do
      t := t * x;
    S := t;
  end;
Output ← S;
```

Thuật toán tính bằng luật Horner:

```
Input → a[0..d], x;
S := a[d];
for i := 0 to d - 1 do
  S := S * x + a[i];
Output ← S;
```

Chứng minh tính đúng đắn của hai thuật toán bằng tính bất biến vòng lặp.

Bài 3. Đánh giá giải thuật

Đánh giá một giải thuật tức là tìm cách đánh giá, ước lượng nguồn tài nguyên cần phải có khi thực thi chương trình cài đặt giải thuật đó. Nguồn tài nguyên cần huy động có thể là lượng bộ nhớ cần thiết, số lượng bộ vi xử lý, tốc độ đường truyền mạng… Nhưng thông thường, vấn đề được quan tâm nhất là thời gian thực hiện giải thuật.

Cần phân biệt việc đánh giá giải thuật và đánh giá chương trình. Để đánh giá chương trình, ta cần cài đặt chương trình trên một máy cụ thể, thử chạy chương trình với một bộ dữ liệu cụ thể, qua đó kiểm định tính chính xác, đo thời gian thực hiện, lượng bộ nhớ chiếm dụng… của chương trình trong trường hợp cụ thể đó. Còn đánh giá giải thuật là việc làm trước khi viết chương trình, nhằm xác định tính khả thi của giải thuật, chọn ra giải thuật tốt nhất để cài đặt.

Trong một số tài liệu khác, người ta còn phân biệt khái niệm "thuật toán" và "giải thuật" (thực ra đây hoàn toàn là quy ước trong tiếng Việt để tiện trình bày các khái niệm chứ không có thuật ngữ tiếng Anh riêng cho hai khái niệm này). Ta có thể hiểu "thuật toán" đưa ra mô hình giải quyết bài toán còn "giải thuật" bao gồm thuật toán và cách thức cài đặt thuật toán trên một cấu trúc dữ liệu cụ thể. Một thuật toán có thể cài đặt rất hiệu quả trên cấu trúc dữ liệu này nhưng lại thiếu hiệu quả hoặc không thể cài đặt được khi dùng cấu trúc dữ liệu khác (ví dụ như thuật toán tìm kiếm nhị phân có thể cài đặt dễ dàng trên mảng nhưng không thể cài đặt được trên danh sách nối đơn) nên đánh giá thuật toán là phải đánh giá mô hình cài đặt thuật toán đó trên một cấu trúc dữ liệu, hay nói cách khác là đánh giá giải thuật.

Chúng ta sẽ không phân biệt hai khái niệm này, và ngầm định rằng khi nói đánh giá thuật toán (giải thuật) tức là đánh giá mô hình cài đặt thuật toán đó trên một cấu trúc dữ liệu cụ thể.

3.1. Mô hình máy tính để đánh giá giải thuật

3.1.1. Chỉ thị sơ cấp

Trước khi đi vào những kỹ thuật phân tích giải thuật, phải có một mô hình máy sẽ dùng để cài đặt giải thuật. Diễn giải một cách chính xác mô hình máy tính cài đặt giải thuật là rất phức tạp và đụng chạm tới nhiều kiến thức về kiến trúc máy tính. Chúng ta chỉ cần hiểu là mô hình máy tính giống kiến trúc như đa số các máy tính hiện nay: một bộ vi xử lý (thực hiện tuần tự các chỉ thị), một bộ nhớ, và các chỉ thị giao cho máy thực hiện là các chỉ thị sơ cấp . Các chỉ thị sơ cấp *có thời gian thực hiện là hằng số* và *không phụ thuộc dữ liệu*.

> Nếu mô hình máy tính dùng để phân tích giải thuật có chỉ thị SORT là chỉ thị sơ cấp để sắp xếp một dãy số, thì sẽ không còn gì để nói về các thuật toán sắp xếp nữa khi mà tất cả mọi thao tác sắp xếp trên máy tính đó sẽ mất thời gian hằng số bất kể dữ liệu đầu vào.

Với kiến trúc máy tính hiện nay, chúng ta có thể coi các phép toán số học, logic, các phép so sánh trên các giá trị thuộc kiểu dữ liệu đơn giản,… là các chỉ thị sơ cấp. Trong khi đó các lệnh dịch chuyển bộ nhớ không thể coi là các chỉ thị sơ cấp, nó phụ thuộc vào kích thước bộ nhớ cần dịch chuyển. Nói như vậy có nghĩa là các lệnh gán mảng, sao chép một phần xâu ký tự,… không được coi là các chỉ thị sơ cấp, chúng ta phải phân tích các lệnh đó thành các chỉ thị sơ cấp để đánh giá thời gian thực hiện giải thuật.

3.1.2. Các hiệu ứng trên bộ nhớ truy cập ngẫu nhiên

Các bộ vi xử lý máy tính hiện nay đều được trang bị bộ nhớ cache, loại bộ nhớ này được dùng làm trung chuyển dữ liệu giữa RAM và CPU và làm tăng tốc đáng kể những chỉ thị truy cập bộ nhớ tuần tự. Lấy ví dụ trong việc truy cập mảng: Nếu ta có một mảng khá lớn, thì bộ nhớ cache giúp cho việc duyệt các phần tử của mảng từ đầu tới cuối nhanh hơn nhiều so với việc duyệt các phần tử của mảng theo thứ tự ngẫu nhiên. Như vậy, thời gian thực hiện chỉ thị đọc/ghi dữ liệu tại một ô nhớ không phải là hằng số, mà còn phụ thuộc vào những chỉ thị đọc/ghi bộ nhớ thực hiện trước đó.

Các hệ điều hành hiện nay đều có cơ chế quản lý bộ nhớ ảo: Nếu lượng bộ nhớ vật lý không đủ, một phần đĩa cứng sẽ được sử dụng để hoán chuyển dữ liệu với bộ nhớ vật lý. Như vậy nếu chương trình chiếm dụng một lượng bộ nhớ nhiều hơn lượng bộ nhớ vật lý hiện có, máy sẽ phải hoán chuyển dữ liệu với đĩa cứng và làm giảm đáng kể tốc độ thực thi chương trình.

Để đơn giản hóa vấn đề, trong mô hình máy tính cài đặt giải thuật, chúng ta bỏ qua hiệu ứng của bộ nhớ cache và bộ nhớ ảo, coi thời gian thực hiện một chỉ thị đọc/ghi dữ liệu tại mọi ô nhớ là hằng số. Việc chọn một giải thuật để cài đặt cụ thể ngoài những đánh giá lý thuyết về tính "tốt" của giải thuật, còn cần có một sự nhạy cảm của một lập trình viên để chọn ra một giải thuật tận dụng tối đa hiệu ứng của bộ nhớ cache và hạn chế tối đa hiệu ứng của bộ nhớ ảo. Chúng ta sẽ thấy điều nay khi xét các ví dụ về thuật toán QuickSort và HeapSort: mặc dù có cùng mức độ "tốt" trên lý thuyết, bộ nhớ cache sẽ tăng tốc chương trình QuickSort nhanh hơn khoảng 6 lần so với HeapSort.

3.2. Phân tích thời gian thực hiện giải thuật

Một bài toán không chỉ có một giải thuật, để chọn giải thuật tốt nhất, ta phải có một tiêu chuẩn để nói rằng giải thuật này tốt hơn giải thuật kia.

Với một máy tính cụ thể, những chương trình cụ thể, và một bộ dữ liệu cụ thể. Ta có thể so sánh chương trình này nhanh hơn hay chậm hơn chương trình khác đơn giản chỉ bằng phương pháp đo thời gian. Lưu ý rằng việc đánh giá này là dựa trên một bộ dữ liệu cụ thể, với bộ dữ liệu khác, kết quả có thể khác.

Việc đánh giá thời gian thực hiện giải thuật thì không đơn giản như vậy, thời gian thực hiện một giải thuật bằng chương trình máy tính phụ thuộc vào rất nhiều yếu tố. Một yếu tố cần chú ý nhất đó là kích thước của dữ liệu đưa vào. Dữ liệu càng lớn thì thời gian xử lý càng chậm, chẳng hạn như thời gian sắp xếp một dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi n là kích thước dữ liệu đưa vào thì thời gian thực hiện của một giải thuật có thể biểu diễn một cách tương đối như một hàm của n: $T(n)$.

Phần cứng máy tính, ngôn ngữ viết chương trình và chương trình dịch đều ảnh hưởng tới thời gian thực hiện giải thuật, vì vậy chúng ta không thể đánh giá tuyệt đối $T(n)$ bằng đơn vị thời gian (giờ, phút, giây) được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ. Nếu như ta đánh giá được thời gian thực hiện một giải thuật A tỉ lệ thuận với n còn thời gian thực hiện một giải thuật B tỉ lệ thuận với n^2 thì cho dù hai hằng số tỉ lệ có giá trị như thế nào, khi n đủ lớn, thời gian thực hiện của giải thuật A rõ ràng nhanh hơn giải thuật B. Khi đó, nếu nói rằng thời gian thực hiện giải thuật tỉ lệ thuận với n hay tỉ lệ thuận với n^2 cũng cho ta một cách đánh giá tương đối về tốc độ thực hiện của giải thuật đó khi n khá lớn. Cách đánh giá thời gian thực hiện giải thuật dựa trên kích thước dữ liệu vào như vậy gọi là đánh giá *độ phức tạp tính toán của giải thuật* (*algorithm complexity*).

3.3. Tốc độ tăng của hàm

3.3.1. Ký pháp

Cho một giải thuật thực hiện trên dữ liệu với kích thước n. Giả sử $T(n)$ là thời gian thực hiện giải thuật đó, $f(n)$ là một hàm xác định dương với mọi n. Khi đó:

Ta nói $T(n) = \Theta(f(n))$, nếu tồn tại các hằng số dương c_1, c_2 và n_0 sao cho :

$$c_1 f(n) \leq T(n) \leq c_2 f(n), \forall n \geq n_0 \qquad (3.1)$$

Ký pháp này được gọi là *ký pháp Θ lớn* (*big-theta notation*). Trong ký pháp Θ lớn, hàm f được gọi là *giới hạn tiệm cận chặt* (*asymptotically tight bound*) của hàm T.

Ta nói $T(n) = O(f(n))$, nếu tồn tại các hằng số dương c và n_0 sao cho:

$$T(n) \leq cf(n), \ \forall n \geq n_0 \qquad (3.2)$$

Ký pháp này được gọi là *ký pháp chữ O lớn* (*big-O notation*). Trong ký pháp chữ O lớn, hàm f được gọi là *giới hạn tiệm cận trên* (*asymptotic upper bound*) của hàm T.

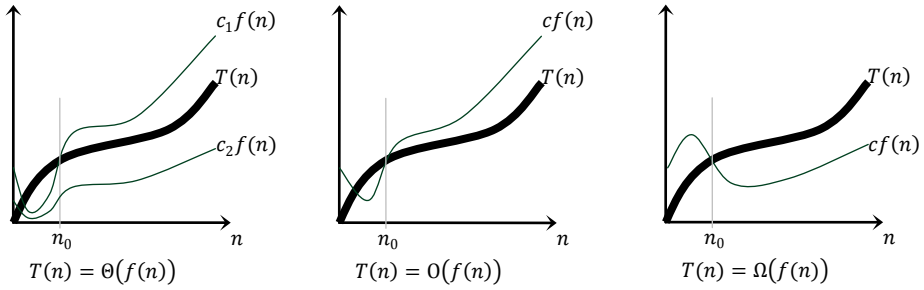
Ta nói $T(n) = \Omega(f(n))$, nếu tồn tại các hằng số dương c và n_0 sao cho:

$$T(n) \geq cf(n), \forall n \geq n_0 \tag{3.3}$$

Ký pháp này gọi là ký pháp Ω lớn (*big-omega notation*). Trong ký pháp Ω lớn, hàm f được gọi là *giới hạn tiệm cận dưới* (*asymptotic lower bound*) của hàm T

Hình 3-1 là biểu diễn đồ thị của ký pháp Θ lớn, O lớn và Ω lớn. Dễ dàng suy ra được kết quả sau:

$$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n)) \text{ và } T(n) = \Omega(f(n)) \tag{3.4}$$



Hình 3-1. Các ký pháp đánh giá thời gian thực hiện.

Ngoài các ký pháp Θ, O, Ω lớn. Người ta còn định nghĩa thêm hai ký pháp "nhỏ":

Ta nói $T(n) = o(f(n))$, nếu với mọi hằng số dương c, tồn tại một hằng số dương n_0 sao cho:

$$T(n) \leq cf(n), \ \forall n \geq n_0 \tag{3.5}$$

Ký pháp này gọi là ký pháp chữ o nhỏ (*little-o notation*). Trong một số tài liệu, ký pháp chữ o nhỏ có thể viết:

$$T(n) = o(f(n)) \Leftrightarrow \lim_{n\to\infty} \frac{T(n)}{f(n)} = 0 \tag{3.6}$$

Ta nói $T(n) = \omega(f(n))$, nếu với mọi hằng số dương c, tồn tại một hằng số dương n_0 sao cho:

$$T(n) \geq cf(n), \ \forall n \geq n_0 \tag{3.7}$$

Ký pháp này gọi là ký pháp chữ o nhỏ (*little-o notation*). Trong một số tài liệu, ký pháp chữ o nhỏ có thể viết:

$$T(n) = o\big(f(n)\big) \Leftrightarrow \lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty \tag{3.8}$$

Ví dụ nếu $T(n) = n^2 + 1$ thì:

$T(n) = \mathrm{O}(n^2)$, thật vậy, chọn $c = 2$ và $n_0 = 1$, rõ ràng $\forall n \geq 1$, ta có:

$$T(n) = n^2 + 1 \leq 2n^2$$

$T(n) = \Omega(n^2)$, thật vậy, chọn $c = 1$ và $n_0 = 1$, rõ ràng với $\forall n \geq 1$, ta có:

$$T(n) = n^2 + 1 \geq n^2$$

$T(n) = \Theta(n^2)$ vì $T(n) = \mathrm{O}(n^2)$ và $T(n) = \Omega(n^2)$

$T(n) \neq o(n^2)$, thật vậy, chọn $c = 1$, bất đẳng thức $n^2 + 1 \leq n^2$ không thể thỏa mãn với mọi giá trị của n

$T(n) = o(n^3)$, thật vậy, với mọi hằng số $c > 0$, chọn $n_0 = 2/c + 1$, khi đó $\forall n \geq n_0$, ta có $cn - 1 > 1$ và $n^2 > 1$, suy ra:

$$n^2 + 1 \leq (cn - 1)n^2 + n^2 = cn^3$$

$T(n) = \omega(n)$, thật vậy, với mọi hằng số $c > 0$, chọn $n_0 = c$, khi đó $\forall n > n_0$, ta có:

$$n^2 + 1 \geq cn + 1 > cn$$

3.3.2. Một số tính chất

Tính bắc cầu (transitivity): Tất cả các ký pháp nêu trên đều có tính bắc cầu:

$$\begin{aligned}
&f(n) = \Theta\big(g(n)\big) \text{ và } g(n) = \Theta\big(h(n)\big) \Rightarrow f(n) = \Theta\big(h(n)\big)\\
&f(n) = \mathrm{O}\big(g(n)\big) \text{ và } g(n) = \mathrm{O}\big(h(n)\big) \Rightarrow f(n) = \mathrm{O}\big(h(n)\big)\\
&f(n) = \Omega\big(g(n)\big) \text{ và } g(n) = \Omega\big(h(n)\big) \Rightarrow f(n) = \Omega\big(h(n)\big)\\
&f(n) = \mathrm{o}\big(g(n)\big) \text{ và } g(n) = \mathrm{o}\big(h(n)\big) \Rightarrow f(n) = \mathrm{o}\big(h(n)\big)\\
&f(n) = \omega\big(g(n)\big) \text{ và } g(n) = \omega\big(h(n)\big) \Rightarrow f(n) = \ \omega\big(h(n)\big)
\end{aligned} \tag{3.9}$$

Tính phản xạ (reflexivity): Chí có các ký pháp "lớn" mới có tính phản xạ:

$$\begin{aligned}
&f(n) = \Theta\big(f(n)\big)\\
&f(n) = \mathrm{O}\big(f(n)\big)\\
&f(n) = \Omega\big(f(n)\big)
\end{aligned} \tag{3.10}$$

Tính đối xứng (symmetry): Chỉ có ký pháp Θ lớn có tính đối xứng:

$$f(n) = \Theta(g(n) \Leftrightarrow g(n) = \Theta\big(f(n)\big) \tag{3.11}$$

Tính chuyển vị đối xứng (transpose symmetry):

$$\begin{aligned}
&f(n) = \mathrm{O}\big(g(n)\big) \Leftrightarrow g(n) = \Omega\big(f(n)\big)\\
&f(n) = o\big(g(n)\big) \Leftrightarrow g(n) = \omega\big(f(n)\big)
\end{aligned} \tag{3.12}$$

Để dễ nhớ ta coi các ký pháp đánh giá thời gian thực hiện tương ứng với các phép so sánh

$$O \leftrightarrow \leq$$
$$\Omega \leftrightarrow \geq$$
$$\Theta \leftrightarrow =$$
$$o \leftrightarrow <$$
$$\omega \leftrightarrow >$$

Từ đó suy ra các tính chất trên.

3.4. Các hàm đánh giá thời gian thực hiện

Để đánh giá thời gian thực hiện của giải thuật qua các ký pháp $O, \Omega, \Theta, o, \omega$ người ta thường dùng những hàm đơn điệu tăng. Một số hàm thông dụng để đánh giá thời gian thực hiện là:

3.4.1. Hàm lấy sàn và lấy trần của một số (Floor/Ceiling)

Sàn của một số x, ký hiệu $\lfloor x \rfloor$, là số nguyên lớn nhất không lớn hơn x. Trần của một số x, ký hiệu $\lceil x \rceil$, là số nguyên nhỏ nhất không nhỏ hơn x.

Với mọi số thực x, ta có:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

Với mọi số nguyên x, ta có

$$\left\lfloor \frac{x}{2} \right\rfloor + \left\lceil \frac{x}{2} \right\rceil = x$$

Với mọi số thực x và hai số nguyên a, b, ta có

$$\lceil \lceil x/a \rceil / b \rceil = \lceil x/(ab) \rceil$$
$$\lfloor \lfloor x/a \rfloor / b \rfloor = \lfloor x/(ab) \rfloor$$
$$\lceil a/b \rceil \leq \big(a + (b-1)\big)/b$$
$$\lfloor a/b \rfloor \geq \big(a - (b-1)\big)/b$$

3.4.2. Hàm đa thức (Polynomials)

Với một số tự nhiên d, một đa thức bậc d với biến n được định nghĩa bởi:

$$p(x) = \sum_{i=0}^{d} a_i n^i = a_d n^d + n x^{d-1} + \cdots + a_1 n + a_0$$

Trong đó các hằng số $a_i (\forall i: 0 \leq i \leq d)$ được gọi là hệ số của đa thức và $a_d \neq 0$. Khi dùng hàm đa thức để đánh giá thời gian thực hiện, hệ số bậc cao nhất d thường là số dương để đảm bảo rằng hàm $p(n)$ sẽ tăng dần tính từ một ngưỡng $n > n_0$ nào đó

3.4.3. Hàm mũ (Exponentials)

Với một cơ số $a > 1$, hàm mũ a^n là hàm đơn điệu tăng với biến n.

Hàm mũ tăng nhanh hơn rất nhiều so với hàm đa thức. Với một đa thức $p(n)$ và hàm mũ a^n với $a > 1$ thì $\lim_{n\to\infty} \frac{a^n}{p(n)} = \infty$ tức là $a^n = \omega\big(p(n)\big)$

3.4.4. Hàm logarit (Logarithms)

Logarit cơ số a của một số n (ký hiệu $\log_a n$) là số b để $a^b = n$

Tùy theo cơ số của logarit, ta có thêm những ký hiệu:

$$\lg n = \log_2 n$$
$$\ln n = \log_e n \ \ (e \approx 2.7182818284590 \dots)$$

Với $n \in \mathbb{R};\ a, b, c \in \mathbb{R}^+$, ta có các tính chất sau:

$$b = a^{\log_a b}$$
$$\log_a(bc) = \log_a b + \log_a c$$
$$\log_a(b^n) = n log_a b$$
$$\log_a c = \log_a b \log_b c$$

3.4.5. Hàm giai thừa (Factorials)

Giai thừa của một số n là tích của các số nguyên dương từ 1 tới n:

$$n! = \prod_{i=1}^{n} i = 1.2.3 \dots n$$

Dùng công thức xấp xỉ của Steeling:

$$n! = \sqrt{\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

ta có thể suy ra được những tính chất sau của hàm giai thừa:

$$n! = o(n^n)$$
$$n! = \omega(a^n)$$

3.4.6. Hàm số Fibonacci

Hàm số Fibonacci được định nghĩa quy nạp như sau:

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

Mỗi số fibonacci là tổng của hai số đứng liền trước:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Hàm Fibonacci có thể tính trực tiếp bằng hệ số tỉ lệ vàng (golden ratio) $\phi = \frac{1+\sqrt{5}}{2}$ và liên hợp của nó $\hat{\phi} = \frac{1-\sqrt{5}}{2} = 1 - \phi$ theo công thức:

$$f(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

Chúng ta có thể chứng minh công thức này một cách đơn giản:

Rõ ràng

$$f(0) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}} = 0$$

$$f(1) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} = 1$$

Ta sẽ chứng minh $f(n) = f(n-1) + f(n-2)\ \forall n \geq 2$. Nhận xét rằng hệ số tỉ lệ vàng $\phi = \frac{1+\sqrt{5}}{2}$ và liên hợp của nó $\hat{\phi} = \frac{1-\sqrt{5}}{2} = 1 - \phi$ là hai nghiệm của phương trình: $x^2 = x + 1$. Tức là

$$\begin{cases} \phi^2 = \phi + 1 \\ (1-\phi)^2 = (1-\phi) + 1 \end{cases}$$

Suy ra

$$\begin{cases} \phi^n = \phi^{n-1} + \phi^{n-2} \\ (1-\phi)^n = (1-\phi)^{n-1} + (1-\phi)^{n-2} \end{cases}$$

Vậy

$$\phi^n - (1-\phi)^n = \phi^{n-1} - (1-\phi)^{n-1} + \phi^{n-2} - (1-\phi)^{n-2}$$

Chia cả hai vế của phương trình cho $\sqrt{5}$

$$\frac{\phi^n - (1-\phi)^n}{\sqrt{5}} = \frac{\phi^{n-1} - (1-\phi)^{n-1}}{\sqrt{5}} + \frac{\phi^{n-2} - (1-\phi)^{n-2}}{\sqrt{5}}$$

$$f(n) = f(n-1) + f(n-2)$$

Suy ra điều phải chứng minh.

Nhìn vào công thức $f(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$, ta có nhận xét rằng $|1-\phi| < 1$, vậy $\left|\frac{(1-\phi)^n}{\sqrt{5}}\right| < \frac{1}{\sqrt{5}} < \frac{1}{2}$, suy ra ta có thể đánh giá độ lớn của $f(n)$ một cách gần đúng bằng giá trị làm tròn $Round(\phi^n / sqrt(5))$. Kết quả này cho ta ước lượng tốc độ tăng của hàm số Fibonacci

$$f(n) = \Theta(\phi^n)$$

3.5. Xác định thời gian thực hiện giải thuật

Việc xác định thời gian thực hiện của một giải thuật bất kỳ có thể rất phức tạp. Tuy nhiên thời gian thực hiện của một số giải thuật trong thực tế có thể tính bằng một số qui tắc đơn giản.

3.5.1. Qui tắc bỏ hằng số

Định lý 3-1

Nếu đoạn chương trình P có thời gian thực hiện $T(n) = O(cf(n))$ với c là một hằng số dương thì có thể coi $T(n) = O(f(n))$.

Thậy vậy, theo định nghĩa $T(n) = O(cf(n))$ nên $\exists c_0, n_0 > 0$ để $T(n) \leq c_0 cf(n), \forall n > n_0$. Chọn $c_1 = c_0 c$ và dùng định nghĩa, ta có $T(n) = O(f(n))$

Qui tắc bỏ hằng số cũng đúng với các ký pháp Ω, Θ, o và ω.

3.5.2. Quy tắc lấy max

Định lý 3-2

Nếu đoạn chương trình P có thời gian thực hiện $T(n) = O(f(n) + g(n))$ thì có thể coi $T(n) = \max(f(n), g(n))$

Thật vậy, $T(n) = O(f(n) + g(n))$ nên $\exists c, n_0 > 0$ để

$$\begin{aligned} T(n) &\leq c(f(n) + g(n)) \\ &\leq 2c.\max(f(n), g(n)), \forall n > n_0 \end{aligned}$$

Vậy theo định nghĩa, ta có $T(n) = O(\max(f(n), g(n))$

Quy tắc lấy max cũng đúng với các ký pháp Ω, Θ, o và ω.

3.5.3. Quy tắc cộng

Định lý 3-3

Nếu đoạn chương trình P có thời gian thực hiện $T_p(n) = O(f(n))$ và đoạn chương trình Q có thời gian thực hiện $T_q(n) = O(g(n))$ thì thời gian thực hiện đoạn chương trình P rồi tiếp đến Q sẽ là $T_p(n) + T_q(n) = O(f(n) + g(n))$

Thật vậy, $T_p(n) = O(f(n))$ nên sẽ $\exists c_p, n_p > 0$ để $T_p(n) \leq c_p f(n), \forall n > n_p$. Lại có $T_q(n) = O(g(n))$ nên sẽ $\exists c_q, n_q > 0$ để $T_q(n) \leq c_q g(n), \forall n > n_q$. Chọn $c = \max(c_p, c_q)$, và $n_0 = \max(n_p, n_q)$, ta có $\forall n > n_0$:

$$\begin{aligned} T_p(n) + T_q(n) &\leq c_p f(n) + c_q g(n) \\ &\leq cf(n) + cg(n) \end{aligned}$$

$$= c\big(f(n) + g(n)\big)$$

Tức là $T_p(n) + T_q(n) = \mathrm{O}\big(f(n) + g(n)\big)$

Quy tắc cộng cũng đúng với các ký pháp Ω, Θ, o và ω.

3.5.4. Quy tắc nhân

Định lý 3-4

Nếu đoạn chương trình P có thời gian thực hiện là $T(n) = \mathrm{O}\big(f(n)\big)$ thì thời gian thực hiện $k(n)$ lần đoạn chương trình P với $k(n) = \mathrm{O}\big(g(n)\big)$ sẽ là $k(n)T(n) = \mathrm{O}\big(g(n)f(n)\big)$

Thật vậy, từ $T(n) = \mathrm{O}\big(f(n)\big)$, nên sẽ $\exists c_T, n_T > 0$ để $T(n) \le c_T f(n), \forall n > n_T$. Lại từ $k(n) = \mathrm{O}\big(g(n)\big)$, nên sẽ $\exists c_k, n_k > 0$ để $k(n) \le c_k g(n), \forall n > n_k$. Chọn $c = c_k c_T$ và $n_0 = \max(n_k, n_T)$, ta có $\forall n > n_0$:

$$k(n)T(n) \le c_k g(n) c_T\big(f(n)\big)$$
$$= cg(n)f(n)$$

Vậy $k(n)T(n) = \mathrm{O}\big(g(n)f(n)\big)$

Quy tắc nhân cũng đúng với các ký pháp Ω, Θ, o và ω.

3.5.5. Phép toán tích cực

Dựa vào những nhận xét đã nêu ở trên về các quy tắc khi đánh giá thời gian thực hiện giải thuật, ta chú ý đặc biệt đến một phép toán mà ta gọi là phép toán tích cực trong một đoạn chương trình. Đó là một *phép toán trong một đoạn chương trình mà thời gian thực hiện phép toán là hằng số và số lần thực hiện không ít hơn các phép toán khác*.

Một cách chính xác hơn, φ là phép toán tính cực trong chương trình nếu với mọi phép toán φ' khác của chương trình thì:

$$T_{\varphi'}(n) = \mathrm{O}\left(T_{\varphi}(n)\right)$$

Trong đó $T_{\varphi'}(n)$ là tổng thời gian thực hiện các phép toán φ' và $T_{\varphi}(n)$ là tổng thời gian thực hiện các phép toán φ trên dữ liệu kích thước n.

Định lý 3-5

Nếu φ_0 là phép toán tích cực trong một đoạn chương trình và $T_0(n)$ là thời gian thực hiện phép toán φ_0 với dữ liệu kích thước n. Gọi $T(n)$ là thời gian thực hiện chương trình trên dữ liệu kích thước n, khi đó $T(n) = \Theta\big(T_0(n)\big)$

Chứng minh

Gọi $\varphi_1, \varphi_2, \dots, \varphi_k$ là các phép toán khác trong chương tình v với thời gian thực hiện của chúng trên dữ liệu kích thước n lần lượt là $T_1(n), T_2(n), \dots, T_k(n)$. Theo định nghĩa phép toán tích cực, ta có:

$$T_i(n) = \mathrm{O}\big(T_0(n)\big), \forall i: 1 \le i \le k$$

Theo quy tắc cộng, ta có:

$$T(n) = \sum_{i=0}^{k} T_i(n) = \mathrm{O}\big((k+1)T_0(n)\big)$$

Theo quy tắc bỏ hằng số, ta có $T(n) = \mathrm{O}\big(T_0(n)\big)$

Mặt khác rõ ràng $T(n) \ge T_0(n), \forall n$ nên $T(n) = \Omega\big(T_0(n)\big)$

Kết hợp lại, ta có $T(n) = \Theta\big(T_0(n)\big)$

Định lý 3-6

Gọi μ là một trong các ký pháp $\Theta, \mathrm{O}, \Omega, o, \omega$. Khi đó nếu $T(n) = \mu\big(f(n)\big)$ và $f(n) = \Theta\big(g(n)\big)$, thì $T(n) = \mu\big(g(n)\big)$. Tức là nếu $f(n)$ và $g(n)$ là *tương đương* Θ thì ta có thể thay $f(n)$ bởi $g(n)$ trong ký pháp đánh giá độ phức tạp tính toán.

Chứng minh

Việc chứng minh hoàn toàn dựa vào định nghĩa.

Hệ quả

Gọi μ là một trong các ký pháp $\Theta, \mathrm{O}, \Omega, o, \omega$. Khi đó:

- Nếu một giải thuật có thời gian thực hiện là $T(n) = \mu\big(p(n)\big)$, trong đó $p(n)$ là một đa thức bậc d thì ta có thể viết $T(n) = \mu(n^d)$
- Nếu một giải thuật có thời gian thực hiện là $T(n) = \mu(\log_a n)$, vì $\log_a n = \Theta(\log_b n)$ với bất kỳ cơ số $b > 0$ nào, vì thế ta có thể viết $T(n) = \mu(\log n)$ mà không cần ghi cơ số của logarit.
- Nếu một giải thuật có độ phức tạp là hằng số: $T(n) = c = \Theta(c)$, tức là thời gian thực hiện không phụ thuộc vào kích thước dữ liệu vào thì ta ký hiệu thời gian thực hiện của giải thuật đó là $T(n) = \Theta(1)$.

Định lý 3-5 cho phép ta đánh giá độ phức tạp tính toán qua số lần thực hiện một phép toán. Định lý 3-6 cho phép ta viết ký pháp đánh giá một cách gọn gàng qua các hàm số thông dụng.

Xét hai đoạn chương trình tính e^x bằng công thức gần đúng:

$$e^x \approx \sum_{i=0}^{n} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!}$$

Thuật toán 1: Tính riêng từng hạng tử rồi cộng lại

```
Input → x, n
s := 0;
for i := 0 to n do
  begin
    t := 1;
    for j := 1 to i do
      t := t * x / j;
    s := s + t;
  end;
Output ← s;
```

Trong thuật toán 1, ta có thể coi phép gán $t := t * x/j$ là phép toán tích cực, số lần thực hiện phép toán này là:

$$0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Vậy thời gian thực hiện thuật toán 1 là $\Theta(n^2)$, vì $\frac{n(n+1)}{2} = \Theta(n^2)$

Thuật toán 2: Tính hạng tử sau qua hạng tử trước:

```
Input → x, n
s := 0;
t := 1;
for i := 1 to n do
  begin
    t := t * x / i;
    s := s + t;
  end;
Output ← s;
```

Trong thuật toán 2, ta có thể coi phép gán $t := t * x/i$ là phép toán tích cực, số lần thực hiện phép toán này là n, vậy thời gian thực hiện giải thuật 2 là $\Theta(n)$.

3.5.6. Phương pháp thế

Trong một số trường hợp, thời gian thực hiện giải thuật với dữ liệu kích thước n được cho bởi công thức truy hồi. Tức là $T(n)$ được tính gián tiếp qua các giá trị $T(n')$ với $n' < n$. Chẳng hạn cho:

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \qquad (3.13)$$

Để ước lượng hàm $f(n)$ sao cho $T(n) = O(f(n))$, trước hết ta phải có một dự đoán đúng, sau đó dùng phép quy nạp để chứng minh dự đoán đó.

Trong trường hợp này, ta đoán $T(n) = O(n \lg n)$ sau đó chứng minh $T(n) \leq cn \lg n$ với một lựa chọn hằng số c và ngưỡng n_0. Bắt đầu với giả thiết quy nạp là giới hạn trên này đúng với $\lfloor n/2 \rfloor$, tức là $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg\lfloor n/2 \rfloor$, thay thế vào công thức truy hồi của $T(n)$:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2c\lfloor n/2 \rfloor \lg\lfloor n/2 \rfloor + n \\ &\leq 2c(n/2) \lg(n/2) + n \\ &\leq cn \lg\lfloor n/2 \rfloor + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

Tại bước cuối cùng, ta suy ra $T(n) \leq cn \lg n$ với cách chọn hằng số c sao cho $c > 1$, vậy $T(n) = O(n \lg n)$.

Đặc điểm của phép chứng minh quy nạp trong ví dụ này là ta phải chỉ ra rằng có thể chọn hằng số c đủ lớn sao cho bất đẳng thức $T(n) \leq cn \lg n$ đúng với mọi giá trị của n. Tuy nhiên trước khi thực hiện phép chứng minh quy nạp, ta phải chứng minh cách chọn hằng số c như vậy luôn cho bất đẳng thức đúng trong các trường hợp cơ sở. Việc chứng minh bất đẳng thức đúng trong các trường hợp cơ sở có thể dẫn đến nhiều rắc rối. Cụ thể trong trường hợp này sẽ không có hằng số dương c nào làm cho bất đẳng thức: $T(n) \leq cn \lg n$ đúng với $n = 1$.

Rất may mắn là do định nghĩa của các ký pháp đánh giá thời gian thực hiện, chúng ta chỉ cần chứng minh bất đẳng thức đúng với mọi giá trị của n bắt đầu từ một ngưỡng n_0 tự chọn, vì vậy trong trường hợp này, ta có thể chọn hằng số $c > 1$ đủ lớn sao cho bất đẳng thức đúng với trường hợp $n = 2$ và $n = 3$. Khi đó phép quy nạp sẽ dẫn về một trong hai trường hợp cơ sở là $n = 2$ hoặc $n = 3$ và bất đẳng thức $T(n) \leq cn \lg n$ sẽ đúng với $\forall n \geq n_0 = 2$.

❑ Thực hiện một phép đoán đúng

Trong phương pháp đánh giá này, việc đầu tiên phải làm là đoán công thức. Vì không có một quy tắc chung nào để đoán cả, nên phải có kinh nghiệm mới đoán được chính xác.

Nếu như công thức truy hồi có dạng gần tương tự như một công thức ta đã biết, hoàn toàn có lý do để ta đoán một công thức tương tự. Chẳng hạn với công thức:

$$T(n) = 2(T(\lfloor n/2 \rfloor) + 17) + n \qquad (3.14)$$

Công thức này nhìn có vẻ phức tạp hơn công thức (3.13), tuy nhiên để ý rằng khi $n \to \infty$ sự khác biệt giữa $\lfloor n/2 \rfloor$ và $\lfloor n/2 \rfloor + 17$ trở nên không đáng kể. Vì thế chúng ta có thể đoán $T(n) = \mathrm{O}(n \lg n)$ và chứng minh điều này bằng phương pháp thế.

❑ Lập giả thuyết đủ mạnh

Đôi khi chúng ta đoán đúng thời gian thực hiện của giải thuật, nhưng không chứng minh được do giả thuyết không đủ mạnh. Xét ví dụ về công thức sau:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \tag{3.15}$$

Nếu đoán $T(n) = \mathrm{O}(n)$ và cố gắng chứng minh $T(n) \le cn$ bằng phương pháp thế, ta có:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned} \tag{3.16}$$

Điều này không suy ra được $T(n) \le cn$ với bất kỳ sự lựa chọn nào của hằng số dương c. Để chứng minh, ta phải đặt một giả thuyết mạnh hơn, chẳng hạn $T(n) \le cn - 1$. Khi đó:

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\le c\lfloor n/2 \rfloor - 1 + c\lceil n/2 \rceil - 1 + 1 \\ &= cn - 1 \end{aligned} \tag{3.17}$$

❑ Tránh những suy diễn sai logic

Cần chú ý tới tính logic trong chứng minh quy nạp, chẳng hạn với công thức (3.13), ta có thể sai lầm khi đoán $T(n) = \mathrm{O}(n)$ bằng cách chứng minh $T(n) \le cn$ như sau:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\le 2c\lfloor n/2 \rfloor + n \\ &\le cn + n \\ &= \mathrm{O}(n) \end{aligned} \tag{3.18}$$

Sai lầm ở đây là ta đã không đi chứng minh giả thuyết đã định: $T(n) \le cn$

❑ Đổi biến

Trong một số trường hợp, sử dụng một vài phép biến đổi đại số có thể làm vấn đề trở nên đơn giản. Chẳng hạn với công thức:

$$T(n) = 2T\left(\sqrt{n}\right) + \lg n \tag{3.19}$$

Công thức này có vẻ phức tạp, tuy nhiên qua một phép đổi biến số, vấn đề trở nên quen thuộc:

Đặt $m = \lg n$, ta có:

$$T(2^m) = 2T\left(2^{m/2}\right) + m \tag{3.20}$$

Nếu đặt $S(m) = T(2^m)$, công thức (3.20) trở thành

$$S(m) = 2S(m/2) + m \tag{3.21}$$

Công thức này có dạng tương tự như công thức (3.13), và ta có thể chứng minh được $S(m) = O(m \lg m)$. Vậy thì:

$$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n) \tag{3.22}$$

3.5.7. Phương pháp áp dụng định lý Master

Định lý 3-7 (Định lý Master)

Định lý Master cho ta một cách đánh giá các hàm $T(n)$ có dạng:

$$T(n) = aT(n/b) + f(n) \tag{3.23}$$

Trong đó $a \geq 1$ và $b > 1$ là hai hằng số, $f(n)$ là một hàm dương, $T(n)$ là một hàm xác định trên tập các số tự nhiên, n/b có thể là $\lfloor n/b \rfloor$ hay $\lceil n/b \rceil$ đều được.

Khi đó,

- Nếu $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ với hằng số $\epsilon > 0$, thì $T(n) = \Theta\left(n^{\log_b a}\right)$
- Nếu $f(n) = \Theta\left(n^{\log_b a}\right)$ thì $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
- Nếu $f(n) = \Omega(n^{\log_b a + \epsilon})$ với hằng số $\epsilon > 0$ và $af(n/b) \leq cf(n)$ với hằng số $c < 1$ và với mọi giá trị n đủ lớn thì $T(n) = \Theta\left(f(n)\right)$

Định lý Master là một định lý quan trọng trong việc phân tích thời gian thực hiện của các giải thuật lặp hay đệ quy có thời gian thực hiện viết được dưới dạng (3.23). Tuy nhiên việc chứng minh định lý khá dài dòng, ta có thể tham khảo trong các tài liệu khác.

3.6. Thời gian thực hiện và tình trạng dữ liệu vào

Có nhiều trường hợp, thời gian thực hiện giải thuật không phải chỉ phụ thuộc vào kích thước dữ liệu mà còn phụ thuộc vào tình trạng của dữ liệu đó nữa. Chẳng hạn thời gian sắp xếp một dãy số theo thứ tự tăng dần mà dãy đưa vào chưa có thứ tự sẽ khác với thời gian sắp xếp một dãy số đã sắp xếp rồi hoặc đã sắp xếp theo thứ tự ngược lại. Lúc này, khi phân tích thời gian thực hiện giải thuật ta sẽ phải xét tới trường hợp tốt nhất, trường hợp trung bình và trường hợp xấu nhất.

Nếu ta xét tất cả các bộ dữ liệu kích thước n và tìm bộ dữ liệu có thời gian thực hiện lâu nhất, gọi thời gian đó là $T_{worst}(n)$. Khi đó việc đánh giá thời gian thực hiện giải thuật

dựa trên hàm $T_{worst}(n)$ được gọi là phân tích thời gian thực hiện giải thuật trong trường hợp xấu nhất (worst-case analysis). Tương tự như vậy, nếu ta gọi $T_{best}(n)$ là thời gian thực hiện nhanh nhất trên các bộ dữ liệu kích thước n thì việc đánh giá thời gian thực hiện giải thuật dựa trên hàm $T_{best}(n)$ được gọi là phân tích thời gian thực hiện giải thuật trong trường hợp tốt nhất (best-case analysis).

Phép phân tích thời gian trung bình thực hiện giải thuật (average-case analysis) được thực hiện như sau: Giả sử rằng dữ liệu vào tuân theo một phân phối xác suất nào đó (chẳng hạn phân phối đều nghĩa là khả năng chọn mỗi bộ dữ liệu vào là như nhau), chúng ta tính giá trị kỳ vọng (trung bình) của thời gian chạy cho mỗi kích thước dữ liệu n: $T_{average}(n)$, sau đó phân tích thời gian thực hiện giải thuật dựa trên hàm $T_{average}(n)$.

Để lấy ví dụ về việc phân tích thời gian thực hiện giải thuật trong các trường hợp, ta xét lại bài toán tìm kiếm:

Cho dãy số $A = (a_1, a_2, \dots, a_n)$ sắp xếp theo thứ tự không giảm:

$$a_1 \le a_2 \le \cdots \le n$$

và một số v, hãy cho biết số v có trong dãy A không, và nếu số v có trong dãy A, hãy cho biết chỉ số phần tử đầu tiên của dãy A có giá trị bằng v.

Thuật toán tìm kiếm tuần tự:

```
Input → a[1..n], v;
p := 1;
while (p ≤ n) and (a[p] < v) do
  p := p + 1;
if (p > n) or (a[p] ≠ v) then Output ← "Not Found"
else Output ← p;
```

Thuật toán tìm kiếm tuần tự sẽ so sánh lần lượt các phần tử của dãy với giá trị v và kết thúc ngay khi gặp phần tử $a_p \ge v$. Cuối cùng là việc kiểm chứng a_p có bằng v hay không. Trong thuật toán tìm kiếm tuần tự có thể coi phép so sánh $p \le n$ là phép toán tích cực.

Trường hợp tốt nhất, giá trị v nhỏ hơn phần tử đầu tiên của dãy A: $p < a_1$, phép toán tích cực thực hiện đúng một lần, khi đó thời gian thực hiện thuật toán tìm kiếm tuần tự là $\Theta(1)$.

Trường hợp xấu nhất, giá trị v lớn hơn phần tử cuối cùng của dãy A: $p > a_n$, phép toán tích cực thực hiện $n+1$ lần, khi đó thời gian thực hiện thuật toán tìm kiếm tuần tự là $\Theta(n)$

Xét trung bình thời gian thực hiện giải thuật, gọi:

- $p_1 = \Pr(p \in (-\infty, a_1))$, $T_1(n)$ là trung bình số lần thực hiện phép toán tích cực khi $p \in (+\infty, a_1)$
- $p_2 = \Pr(p \in [a_1, a_n])$, $T_2(n)$ là trung bình số lần thực hiện phép toán tích cực $p \in [a_1, a_n]$
- $p_3 = \Pr(p \in (a_n, +\infty))$, $T_3(n)$ là trung bình số lần thực hiện phép toán tích cực khi $p \in (a_n, +\infty)$

Khi đó trung bình số lần thực hiện phép toán tích cực trong thuật toán tìm kiếm tuần tự là:

$$T(n) = p_1 T_1(n) + p_2 T_2(n) + p_3 T_3(n) \tag{3.24}$$

Nếu cho các phần tử $a_{1\ldots n}$ và v nhận giá trị trong không gian $\mathbb{R}$, thì do $[a_1, a_n]$ bị chặn cả hai phía còn $(-\infty, a_1]$ và $(a_n, +\infty]$ không bị chặn một phía, ta suy ra được $p_1 = 0.5$, $p_2 = 0$ và $p_3 = 0.5$. Tức là:

$$T(n) = 0.5T_1(n) + 0.5T_3(n) \tag{3.25}$$

Ngoài ra, số lần thực hiện phép toán tích cực khi $v < a_1$ đã được chỉ ra trong trường hợp tốt nhất: $T_1(n) = 1$, số lần thực hiện phép toán tích cực khi $v > a_n$ cũng đã được chỉ ra trong trường hợp xấu nhất: $T_3(n) = n + 1$. Vậy trung bình số lần thực hiện phép toán tích cực là:

$$T(n) = \frac{n+2}{2} \tag{3.26}$$

Vậy thuật toán tìm kiếm tuần tự có thời gian thực hiện trung bình $\Theta(n)$.

Bây giờ ta xét thuật toán tìm kiếm nhị phân:

```
Input → a[1..n], v;
L := 1;
H := n;
while L ≤ H do
  begin
    M := (L + H) div 2;
    if a[M] < v then L := M + 1
    else H := M - 1;
  end;
if (L > n) or (a[L] ≠ v) then Output ← "Not Found"
else Output ← L;
```

Trong thuật toán tìm kiếm nhị phân, có thể coi phép so sánh $L \leq H$ là phép toán tích cực, số lần thực hiện phép toán này chính là số lần thực hiện vòng lặp while cộng thêm 1.

Tại mỗi bước lặp, giả sử đoạn đang xét $a_{L..H}$ có độ dài $m = H - L + 1$, thuật toán sẽ thực hiện một phép toán tích cực và giảm độ dài đoạn đang xét xuống còn nhiều nhất $\lfloor m/2 \rfloor$ phần tử. Vậy số lần thực hiện phép toán tích cực là:

$$T(m) = 1 + T(\lfloor m/2 \rfloor) \tag{3.27}$$

Ta đưa ra dự đoán $T(m) \leq c \lg m$ và tìm cách chứng minh điều này bằng phương pháp thế:

$$\begin{aligned} T(m) &\leq 1 + T(\lfloor m/2 \rfloor) \\ &\leq 1 + c \lg(m/2) \\ &\leq c(1 + \lg(m/2) \\ &= c \lg m \end{aligned} \tag{3.28}$$

Điều dự đoán đúng với mọi cách chọn hằng số $c > 1$, vậy thuật toán tìm kiếm nhị phân có thời gian thực hiện $O(\lg n)$ bất kể tình trạng dữ liệu đầu vào. Điều này chỉ ra rằng mặc dù có trường hợp thuật toán tìm kiếm tuần tự thực hiện nhanh hơn một chút, trong đa số trường hợp, thuật toán tìm kiếm nhị phân ($O(\lg n)$) tốt hơn hẳn thuật toán tìm kiếm tuần tự ($\Omega(n)$).

Cũng không khó khăn để chứng minh được thời gian thực hiện của thuật toán tìm kiếm nhị phân là $\Omega(\lg n)$, tức là $T(n) = \Theta(\lg n)$

3.7. Thời gian thực hiện trong tổng thể chương trình

Việc đánh giá thời gian thực hiện trong trường hợp tốt nhất, xấu nhất và trung bình đã nói ở phần trước là trong trường hợp chúng ta thực hiện giải thuật một lần. Nếu giải thuật được thực hiện nhiều lần trong chương trình, chúng ta cần có một sự đánh giá tổng thể. Thời gian thực hiện giải thuật bây giờ phải tính bằng tổng thời gian thực hiện giải thuật chia cho số lần thực hiện giải thuật trong chương trình. Cách đánh giá này gọi là đánh giá bù trừ (*amortized analysis)*.

Không nên nhầm lẫn đánh giá bù trừ với đánh giá thời gian thực hiện trung bình. Khi đánh giá độ phức tạp trung bình, ta coi như giải thuật được thử với tất cả mọi tình trạng dữ liệu và tính thời gian trung bình thực hiện, *các lần thử được thực hiện độc lập*. Nhưng trong một chương trình, khi mà giải thuật được thực hiện nhiều lần thì lần thực hiện sau có thể tận dụng kết quả của lần thực hiện trước để giảm bớt chi phí về không gian và thời gian. Đánh giá bù trừ phải tính đến cả yếu tố đó.

Ta lại lấy ví dụ về bài toán tìm kiếm: Cho hai dãy số sắp xếp theo thứ tự không giảm:

$$A = (a_1 \leq a_2 \leq \cdots \leq a_n)$$
$$B = (b_1 \leq b_2 \leq \cdots \leq b_n)$$

Hãy cho biết chỉ số những phần tử của dãy B có mặt trong dãy A.

Như những phân tích về thuật toán tìm kiếm tuần tự và thuật toán tìm kiếm nhị phân, nếu ta xét tất cả các phần tử của dãy B và dùng thuật toán tìm kiếm nhị phân để xác định xem phần tử đó có trong dãy A không, thì tổng thời gian để thực hiện thuật toán tìm kiếm sẽ là $\Theta(n \lg n)$, lấy tổng thời gian này chia cho số lần thực hiện thuật toán tìm kiếm nhị phân, ta suy ra thời gian thực hiện của thuật toán tìm kiếm nhị phân trong chương trình là $\Theta(\lg n)$

```
Input → a[1..n], b[1..n];
for i := 1 to n do
  begin
    L := 1; H := n;
    while L ≤ H do //Thuật toán tìm kiếm nhị phân tìm phần tử đầu tiên ≥ b[i] trong đoạn a[L..H]
      begin
        M := (L + H) div 2;
        if a[M] < b[i] then L := M + 1
        else H := M - 1;
      end;
    if (L ≤ n) and (a[L] = b[i]) then
      Output ← i;
  end;
```

Có thể thêm vào những tiểu tiết để giới hạn bớt khoảng tìm kiếm sau mỗi lần thực hiện thuật toán tìm kiếm nhị phân. Ví dụ cho dãy $A = (1, 2,5,6,8,9)$ và $B = (8, 10,11,12,18,20)$, sau khi thực hiện tìm kiếm $b_1 = 8$ trong dãy A, ta có thể rút gọn dãy A xuống còn 2 phần tử $(8,9)$ vì các phần tử $(1,2,5,6)$ chắc chắn nhỏ hơn $b_2 = 10$. Tương tự như vậy, sau khi tìm kiếm giá trị $b_2 = 10$, dãy A bị rút gọn thành dãy rỗng và chúng ta sẽ không mất thời gian thực hiện các lệnh tìm kiếm tiếp theo nữa.

Tuy vậy, tiểu xảo này không giúp ích được gì nếu như tất cả các phần tử của dãy B đều nhỏ hơn a_1. Tức là trong trường hợp xấu nhất, đánh giá bù trừ thuật toán tìm kiếm nhị phân vẫn cho kết quả chi phí thời gian $\Theta(\lg n)$.

Bây giờ ta thử xét thuật toán sau, thuật toán này dựa trên thuật toán tìm kiếm tuần tự

```
Input → a[1..n], b[1..n];
p := 1;
for i := 1 to n do
  begin
    //Tìm phần tử đầu tiên ≥ b[i] trong đoạn a[p..n]
    while (p ≤ n) and (a[p] < b[i]) do
      p := p + 1;
    if (p ≤ n) and (a[p] = b[i]) then
      Output ← i;
  end;
```

Thuật toán duyệt các chỉ số $i = 1 \ldots n$ trong dãy B và tại mỗi bước, tìm phần tử đầu tiên trong dãy A lớn hơn hay bằng b_i, việc tìm kiếm sẽ dừng tại chỉ số p đầu tiên gặp phải thoả mãn $p > n$ hoặc $a_p \geq b_i$. Sau khi xác định b_i có xuất hiện trong dãy A không ($b_i = a_p$?), việc tìm kiếm bước tiếp theo với phần tử b_{i+1} sẽ tiếp tục từ vị trí p đó chứ không cần tìm lại từ đầu dãy A, bởi do tính chất tăng dần của hai dãy A, B, tất cả các phần tử đứng trước a_p chắc chắn nhỏ hơn b_{i+1}.

Trong thuật toán này, ta có thể coi phép toán tích cực là phép so sánh $p \leq n$. Tại mỗi bước của vòng lặp for với biến i, ta có nhận xét rằng nếu như phép toán tích cực được thực hiện T_i lần thì giá trị của p sẽ tăng lên $T_i - 1$ đơn vị (theo nguyên lý của vòng lặp while).

Khi kết thúc thuật toán, giá trị của p không thể vượt quá $n + 1$, có nghĩa là:

$$(T_1 - 1) + (T_2 - 1) + \cdots + (T_n - 1) = \left(\sum_{i=1}^{n} T_i\right) - n \leq n + 1 \qquad (3.29)$$

Tức là nếu xét tổng số lần thực hiện phép toán tích cực:

$$\sum_{i=1}^{n} T_i \leq 2n + 1 < 3n, \forall n > n_0 = 1 \qquad (3.30)$$

Từ đó suy ra số lần thực hiện phép toán tích cực là $O(n)$ tính trên toàn thuật toán lớn. Thuật toán tìm kiếm tuần tự bên trong được thực hiện n lần , suy ra nếu đánh giá bù trừ, thời gian mỗi lần thực hiện thuật toán tìm kiếm tuần tự trong ví dụ này chỉ là $O(1)$, tốt hơn so với thuật toán tìm kiếm nhị phân.

Nguyên lý bù trừ luôn phải được tính đến nếu như thuật toán được thực hiện nhiều lần trong chương trình. Sẽ là sai lầm nếu đánh giá rằng thuật toán tìm kiếm tuần tự trong trường hợp xấu nhất sẽ mất chi phí thời gian $\Omega(n)$ và thuật toán này được thực hiện n lần nên chi phí thời gian tổng thể là $\Omega(n^2)$ trong trường hợp xấu nhất.

3.8. Chi phí thực hiện giải thuật

Khái niệm thời gian thực hiện đặt ra không chỉ dùng để đánh giá chi phí thực hiện một giải thuật về mặt thời gian mà là để đánh giá chi phí thực hiện giải thuật nói chung, bao gồm cả chi phí về không gian (lượng bộ nhớ cần sử dụng). Tuy nhiên ở trên ta chỉ đưa định nghĩa về thời gian thực hiện dựa trên chi phí về thời gian cho dễ trình bày. Việc đánh giá thời gian thực hiện theo các tiêu chí khác cũng tương tự nếu ta biểu diễn được mức chi phí theo một hàm $T(n)$ của kích thước dữ liệu vào n. Nếu phát biểu rằng thời

gian thực hiện của một giải thuật là $\Theta(n^2)$ về thời gian và $\Theta(n)$ về bộ nhớ cũng không có gì sai về mặt ngữ nghĩa cả.

Thông thường, nếu thời gian thực hiện của một giải thuật có thể đánh giá được qua ký pháp Θ thì có thể coi phép đánh giá này là đủ chặt và không cần đánh giá qua những ký pháp khác nữa. Nếu không, để nhấn mạnh đến tính "tốt" của một giải thuật, người ta thường dùng các ký pháp O, o với ý nghĩa: Chi phí th ực hiện giải thuật tối đa là…, ít hơn… Còn để đề cập đến tính "tồi" của một giải thuật, các ký pháp Ω, ω thường được sử dụng với ý nghĩa: Chi phí thực hiện giải thuật tối thiểu là…, cao hơn…

Bài tập 3-1.

Bắt đầu giá trị nào của n thì thuật toán có thời gian thực hiện $100n^2$ sẽ nhanh hơn thuật toán có thời gian thực hiện 2^n.

Bài tập 3-2.

Giả sử thời gian thực hiện của giải thuật là $T(n)$ μs (một micro giây bằng một phần triệu giây), hãy đi ền nốt vào bảng sau giá trị lớn nhất của n mà thuật toán sẽ kết thúc trong giới hạn thời gian t (Để đơn giản, ta coi một tháng có 30 ngày và một năm có 365 ngày).

t / $T(n)$	1 giây	1 phút	1 giờ	1 ngày	1 tháng	1 năm	1 thế kỷ
$\lg n$	2^{10^6}						
$\sqrt{n}$	10^{12}						
n	10^6						
$n \lg n$	62746						
n^2	1000						
n^3	100						
2^n	19						51
$n!$	10						18

Bài tập 3-3.

Hãy chỉ ra tường tận cách thực hiện thuật toán tìm kiếm nhị phân để tìm vị trí xuất hiện đầu tiên của giá trị 8 trong dãy $(1,2,3,3,3,8,8,9)$. Chú ý nêu rõ giá trị các biến sau mỗi bước lặp.

Bài tập 3-4.

Dùng phép quy nạp để chứng minh rằng nếu n là một luỹ thừa của 2, công thức truy hồi:

$$T(n) = \begin{cases} 2, \text{ nếu } n = 2 \\ 2T(n/2) + n, \text{ nếu } n = 2^k, \forall k > 1 \end{cases}$$

có thể viết dưới dạng $T(n) = n \lg n$

Bài tập 3-5.

Có 16 giải thuật có thời gian tính toán trên dữ liệu kích thước n lần lượt là:

$T_1(n) = 2^{100^{1000}}$	$T_2(n) = (\sqrt{2})^{\lg n}$	$T_3(n) = n^2$	$T_4(n) = n!$
$T_5(n) = 3^n$	$T_6(n) = \sum_{k=1}^{n} k$	$T_7(n) = \lg^* n$	$T_8(n) = \lg(n!)$
$T_9(n) = 1$	$T_{10}(n) = \lg^*(\lg n)$	$T_{11}(n) = \ln n$	$T_{12}(n) = n^{\lg(lg\ n)}$
$T_{13}(n) = (\lg n)^{\lg n}$	$T_{14}(n) = 2^n$	$T_{15}(n) = n \lg n$	$T_{16}(n) = \sum_{k=1}^{n} \frac{1}{k}$

Hãy xếp lại các giải thuật theo chiều tăng của thời gian thực hiện, có nghĩa là tìm một thứ tự $i_1, i_2, \ldots, i_{16}$ sao cho $T_{i_j}(n) = O\left(T_{i_{j+1}}(n)\right)$, chỉ rõ những giải thuật nào là "tương đương" về thời gian thực hiện theo nghĩa $T_{i_j}(n) = \Theta\left(T_{i_{j+1}}(n)\right)$

Bài tập 3-6.

"Thuật toán A thực hiện rất chậm vì với dữ liệu kích thước n, thời gian thực hiện không ít hơn $O(2^n)$". Hãy chỉ ra rằng câu nói trên là không đúng logic.

Bài tập 3-7.

Giải thích tại sao không có ký pháp $\theta(f(n))$ để chỉ những hàm vừa là $o(f(n))$, vừa là $\omega(f(n))$.

Bài tập 3-8.

Chứng minh rằng, $\forall a, b \in \mathbb{R}, b > 0$, ta có $(n + a)^b = \theta(n^b)$

Bài tập 3-9.

Trong những mệnh đề sau, mệnh đề nào đúng, mệnh đề nào sai?

- $f(n) = O(g(n)) \Rightarrow g(n) = O(f(n))$

- $f(n) + g(n) = \Theta(\min(f(n), g(n)))$
- $f(n) = O(g(n)) \Rightarrow \lg f(n) = O(\lg g(n))$, trong đó $f(n) \geq 1$ và $g(n) \geq 2$ với mọi giá trị n đủ lớn
- $f(n) = O\left((f(n))^2\right)$
- $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$
- $f(n) = \Theta(f(\lfloor n/2 \rfloor))$
- $g(n) = o(f(n)) \Rightarrow f(n) + g(n) = \Theta(f(n))$

Bài tập 3-10.

Thuật toán tìm kiếm nhị phân có thể viết dưới dạng đệ quy: Đưa việc tìm kiếm trong dãy n phần tử về việc tìm kiếm trong dãy có $\leq \lfloor n/2 \rfloor$ phần tử. Hãy viết thuật toán tìm kiếm nhị phân dạng đệ quy, từ đó viết công thức tính thời gian thực hiện giải thuật dưới dạng công thức truy hồi và dùng phương pháp thế để xác định thời gian thực hiện.

Bài tập 3-11.

Chỉ ra chỗ sai trong chứng minh sau:

Giả sử một giải thuật có thời gian thực hiện $T(n)$ cho bởi

$$T(n) = \begin{cases} 1, \text{nếu } n \leq 1 \\ 2T(\lceil n/2 \rceil) + n, \text{nếu } n > 1 \end{cases}$$

Khi đó $T(n) = \Omega(n \lg n)$ và $T(n) = O(n)$

Chứng minh

$T(n) = \Omega(n \lg n)$, thật vậy có thể chọn một số dương c nhỏ hơn 1 và đủ nhỏ để $T(n) \geq cn \lg n$ với $\forall n < 3$ (chẳng hạn chọn $c = 0.1$). Ta sẽ chứng minh $T(n) \geq cn \lg n$ cũng đúng với $\forall n \geq 3$ bằng phương pháp quy nạp:

$$\begin{aligned} T(n) &= 2T(\lceil n/2 \rceil) + n \\ &\geq 2c\lceil n/2 \rceil \lg(\lceil n/2 \rceil) + n \\ &\geq 2c(n/2) \lg(n/2) + n \\ &\geq cn \lg n - cn + n \\ &= cn \lg n + (1 - c)n \\ &\geq cn \lg n \end{aligned}$$

$T(n) = O(n)$, thật vậy, ta lại có thể chọn một số dương c đủ lớn để $T(n) < cn$ với $\forall n < 3$. Ta sẽ chứng minh quy nạp cho trường hợp $n \geq 3$

$$\begin{aligned} T(n) &= 2T(\lceil n/2 \rceil) + n \\ &\leq 2c\lceil n/2 \rceil + n \end{aligned}$$

$$\leq c(n+1)+n$$
$$\leq (c+1)n+c$$
$$= O(n)$$

Ta được một kết quả "thú vị": Từ $T(n) = \Omega(n \lg n)$ suy ra $n \lg n = O(T(n))$, lại có $T(n) = O(n)$, theo luật bắc cầu ta suy ra: $n \lg n = O(n)$!!!wow!!!

Bài tập 3-12. (Tìm số bị thiếu)

Người ta liệt kê tất cả các số tự nhiên từ 0 tới n theo một thứ tự ngẫu nhiên, sau đó bỏ đi đúng một số. Các số còn lại được cho bởi dãy $a_{1\ldots n}$ và yêu cầu đặt ra là hãy lập trình để tìm số đã bị bỏ đi.

Để tìm ra số bị thiếu, vấn đề sẽ trở thành dễ dàng nếu ta lập một mảng phụ kiểu logic $b_{0\ldots n}$ với giá trị khởi tạo hoàn toàn là $False$, sau đó duyệt các phần tử a_i và đặt $b_{a_i} \coloneqq True$. Cuối cùng ta duyệt mảng $b_{0\ldots n}$ để tìm chỉ số v mà $b_v = False$, khi đó v chính là số đã bị bỏ đi.

Tuy nhiên, giả sử rằng máy tính không thể biết được giá trị của mỗi phần tử a_i bằng một chỉ thị sơ cấp, các phần tử trong dãy $a_{1\ldots n}$ được biểu diễn dưới dạng nhị phân và máy có thể đọc một bit trong $a_{1\ldots n}$ bằng một chỉ thị sơ cấp. Hãy tìm thuật toán để chứng tỏ rằng chỉ với chỉ thị sơ cấp này, ta vẫn có thể tìm được số đã bị bỏ đi trong thời gian $O(n)$

Bài 4. Một số chú ý khi đọc sách

4.1. Ngôn ngữ và môi trường lập trình

Các chương trình minh họa trong cuốn sách này được viết bằng chủ yếu bằng Object Pascal (OBJPAS). Môi trường lập trình tương ứng là Free Pascal for Windows 32 bit - phần mềm miễn phí và phổ biến trong giới học sinh/sinh viên trong việc học lập trình. Tuy nhiên nếu bạn có CodeGear RAD Studio*, bạn nên thực hành trực tiếp trên môi trường đó để có được sự hỗ trợ tối đa từ công cụ phát triển phần mềm chuyên nghiệp này.

Tôi cũng có dự định cung cấp thêm mã nguồn bằng C++ song song với mã nguồn OBJPAS, tuy nhiên việc này còn tùy vào thời gian cho phép. Mã nguồn OBJPAS chắc chắn sẽ giữ. Lý do là vì OBJPAS là ngôn ngữ self-documented (đọc chương trình là hiểu thuật toán) nên để diễn giải thuật toán thì dùng OBJPAS làm mã giả và chương trình sẽ thuật tiện hơn nhiều†. Ngoài hai ngôn ngữ này, tôi chưa có ý định chuyển sang ngôn ngữ nào khác, vì hiện tại chỉ có các trình dịch OBJPAS và C++ cho ra mã khả thi có tốc độ tốt.

Không nên quan trọng hóa việc sử dụng ngôn ngữ lập trình nào. Ngôn ngữ nào bạn thành thạo nhất sẽ là ngôn ngữ mạnh nhất. Để học thuật toán thì điều quan trọng là tư duy thiết kế và đánh giá giải thuật, còn để phát triển phần mềm điều quan trọng lại là hiểu và nắm vững công cụ phát triển. Bạn có thể học OBJPAS/C++ trong 3 tuần nếu bạn đã học lập trình trên một ngôn ngữ nào đó, nhưng để học Delphi/Visual C++, theo tôi cần ít nhất 3 năm mới đủ độ thành thạo chưa kể việc phải liên tục cập nhật những kiến thức cho phiên bản mới.

Trong phần lớn các chương trình, tôi cố gắng không sử dụng những đặc tả hướng đối tượng, bởi các chương trình minh hoạ đều ngắn. Ngoài ra việc viết theo đặc tả hướng đối tượng yêu cầu người đọc phải đọc từ đầu tới cuối để hiểu được kiến trúc kế thừa (inheritance) và đa hình (polymorphism) của thư viện lớp, điều này có thể gây mất thời gian nếu như người đọc chỉ cần tra cứu một bài toán, hay một thuật toán trong cuốn sách này.

---

* CodeGear RAD Studio bao gồm Delphi®, Delphi® for .NET, and C++Builder®

† Dĩ nhiên là có thể viết mã C++ theo kiểu self-documented, nhưng đó không phải phong cách viết C++ chuyên nghiệp. Viết như thế chẳng tận dụng được chút ưu điểm nào của C++ cả, chỉ đơn thuần là "dịch thuật" từ mã Object Pascal sang C++ mà việc này có rất nhiều phần mềm chuyển mã bán tự động làm được rồi. Phong cách lập trình C++ chú trọng đến tính ngắn gọn và hiệu quả (cần hiểu chương trình phải có tài liệu diễn giải kèm theo) trong khi lập trình Object Pascal đề cao tính chặt chẽ và cấu trúc.

4.2. Nhập/xuất dữ liệu

Nếu không có ghi chú thêm, tất cả các chương trình minh họa trong cuốn sách này đều sử dụng thiết bị nhập/xuất chuẩn (standard input/output devices): Mặc định là chương trình sẽ lấy dữ liệu vào từ bàn phím và in kết quả ra màn hình. Nếu như lượng dữ liệu nhập/xuất lớn, bạn có thể sửa đổi một chút mã nguồn cho nhập/xuất dữ liệu qua các files văn bản, hoặc làm theo cách sau:

Soạn thảo một file văn bản chứa dữ liệu vào (ví dụ INPUT.TXT) theo đúng khuôn dạng do chương trình quy định.

Chọn một tên file chứa kết quả ra (ví dụ OUTPUT.TXT), tốt nhất là file chưa có trên đĩa để tránh bị ghi đè lên những thông tin quan trọng.

Trong FPC, bạn vào menu Run/Parameters…, gõ vào tham số dòng lệnh:

<INPUT.TXT >OUTPUT.TXT

(Dấu < dùng để chỉ ra file văn bản được dùng như thiết bị nhập chuẩn, dấu > dùng để chỉ ra file văn bản được dùng như thiết bị xuất chuẩn, nếu bạn viết >PRN, chương trình sẽ in kết quả ra máy in)

Sau khi chạy chương trình, bạn sẽ có file OUTPUT.TXT chứa kết quả tương ứng với dữ liệu vào trong file INPUT.TXT.

Chương II. MỘT SỐ CẤU TRÚC DỮ LIỆU CƠ BẢN

Kiểu dữ liệu trừu tượng là một mô hình toán học với những thao tác định nghĩa trên mô hình đó. Kiểu dữ liệu trừu tượng có thể không tồn tại trong ngôn ngữ lập trình mà chỉ dùng để tổng quát hóa hoặc tóm lược những thao tác sẽ được thực hiện trên dữ liệu. Kiểu dữ liệu trừu tượng được cài đặt trên máy tính bằng các cấu trúc dữ liệu: Trong kỹ thuật lập trình cấu trúc (Structural Programming), cấu trúc dữ liệu là các biến cùng với các thủ tục và hàm thao tác trên các biến đó. Trong kỹ thuật lập trình hướng đối tượng (Object-Oriented Programming), cấu trúc dữ liệu là kiến trúc thứ bậc của các lớp, các thuộc tính và phương thức tác động lên chính đối tượng hay một vài thuộc tính của đối tượng.

Trong chương này, chúng ta sẽ khảo sát một vài kiểu dữ liệu trừu tượng cơ bản. Những kiểu dữ liệu trừu tượng phức tạp hơn sẽ được nói đến trong các chương sau hoặc mô tả chi tiết trong từng thuật toán mỗi khi thấy cần thiết.

Bài 5. Danh sách

5.1. Khái niệm danh sách

Danh sách là một tập sắp thứ tự các phần tử cùng một kiểu. Đối với danh sách, người ta có một số thao tác: Tìm một phần tử trong danh sách, chèn một phần tử vào danh sách, xóa một phần tử khỏi danh sách, sắp xếp lại các phần tử trong danh sách theo một trật tự nào đó v.v…

Việc cài đặt một danh sách trong máy tính tức là tìm một cấu trúc dữ liệu cụ thể mà máy tính hiểu được để lưu các phần tử của danh sách đồng thời viết các đoạn chương trình con mô tả các thao tác cần thiết đối với danh sách.

Vì danh sách là một tập sắp thứ tự các phần tử cùng kiểu, ta ký hiệu $TElement$ là kiểu dữ liệu của các phần tử trong danh sách, khi cài đặt cụ thể, $TElement$ có thể là bất cứ kiểu dữ liệu nào được chương trình dịch chấp nhận (Số nguyên, số thực, ký tự, …).

5.2. Biểu diễn danh sách bằng mảng

Khi cài đặt danh sách bằng mảng một chiều , ta cần có một biến nguyên n lưu số phần tử hiện có trong danh sách. Nếu mảng được đánh số bắt đầu từ 1 thì các phần tử trong danh sách được cất giữ trong mảng bằng các phần tử được đánh số từ 1 tới n: $A = a[1 \dots n]$

5.2.1. Truy cập phần tử trong mảng

Việc truy cập một phần tử ở vị trí p trong mảng có thể thực hiện rất dễ dàng qua phần tử a_p. Vì các phần tử của mảng có kích thước bằng nhau và được lưu trữ liên tục trong bộ nhớ, việc truy cập một phần tử được thực hiện bằng một phép toán tính địa chỉ phần tử có thời gian tính toán là hằng số. Vì vậy nếu cài đặt bằng mảng, việc truy cập một phần tử trong danh sách ở vị trí bất kỳ có độ phức tạp là $\Theta(1)$.

5.2.2. Chèn phần tử vào mảng:

Để chèn một phần tử v vào mảng tại vị trí p, trước hết ta dồn tất cả các phần tử từ vị trí p tới tới vị trí n về sau một vị trí (tạo ra "chỗ trống" tại vị trí p), đặt giá trị v vào vị trí p, và tăng số phần tử của mảng lên 1.

```
procedure Insert(p: Integer; const v: TElement); //Thủ tục chèn phần tử v vào vị trí p
var
  i: Integer;
begin
  for i := n downto p do
    a[i + 1] := a[i];
  a[p] := v;
  n := n + 1;
end;
```

Trường hợp tốt nhất, vị trí chèn nằm sau phần tử cuối cùng của danh sách ($p = n + 1$), khi đó thời gian thực hiện của phép chèn là $\Theta(1)$. Trường hợp xấu nhất, ta cần chèn tại vị trí 1, khi đó thời gian thực hiện của phép chèn là $\Theta(n)$. Cũng dễ dàng chứng minh được rằng thời gian thực hiện trung bình của phép chèn là $\Theta(n)$.

5.2.3. Xóa phần tử khỏi mảng

Để xóa một phần tử tại vị trí p của mảng mà vẫn giữ nguyên thứ tự các phần tử còn lại: Trước hết ta phải dồn tất cả các phần tử từ vị trí $p + 1$ tới n lên trước một vị trí (thông tin của phần tử thứ p bị ghi đè), sau đó giảm số phần tử của mảng (n) đi .

```
procedure Delete(p: Integer); //Thủ tục xóa phần tử tại vị trí p
var
  i: Integer;
begin
  for i := p to n - 1 do
    a[i] := a[i + 1];
  n := n - 1;
end;
```

Trường hợp tốt nhất, vị trí xóa nằm cuối danh sách ($p = n$), khi đó thời gian thực hiện của phép xóa là $\Theta(1)$. Trường hợp xấu nhất, ta cần xóa tại vị trí 1, khi đó thời gian thực hiện của phép xóa là $\Theta(n)$. Cũng dễ dàng chứng minh được rằng thời gian thực hiện trung bình của phép xóa là $\Theta(n)$.

Trong trường hợp cần xóa một phần tử mà không cần duy trì thứ tự của các phần tử khác , ta chỉ cần đưa giá trị phần tử cuối cùng vào vị trí cần xóa rồi giảm số phần tử của mảng (n) đi 1. Khi đó thời gian thực hiện của phép xóa chỉ là $\Theta(1)$.

5.3. Biểu diễn danh sách bằng danh sách nối đơn

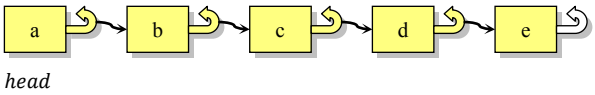
Danh sách nối đơn (Singly-linked list) gồm các nút được nối với nhau theo một chiều. Mỗi nút là một bản ghi (record) gồm hai trường:

- Trường $Info$ chứa giá trị lưu trong nút đó

- Trường *Link* chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó trong danh sách là nút nào, trong trường hợp là nút cuối cùng (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt, chẳng hạn con trỏ *nil*.


```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record; //Kiểu biến động chứa thông tin trong một nút
    Info: TElement;
    Link: PNode;
  end;
```

Nút đầu tiên trong danh sách (*head*) đóng vai trò quan trọng trong danh sách nối đơn. Để duyệt danh sách nối đơn, ta bắt đầu từ nút đầu tiên, dựa vào trường liên kết để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại



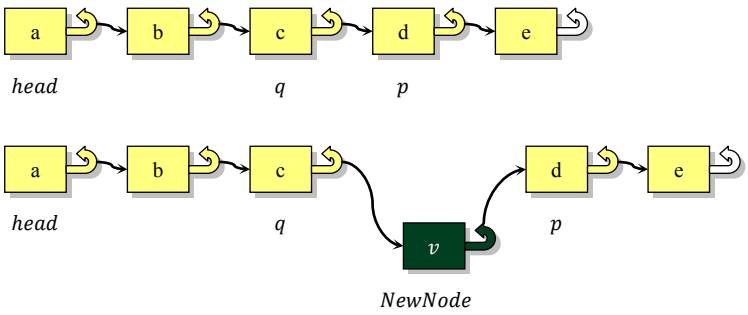
Hình 5-1. Danh sách nối đơn

5.3.1. Truy cập phần tử trong danh sách nối đơn

Bản thân danh sách nối đơn đã là một kiểu dữ liệu trừu tượng. Để cài đặt kiểu dữ liệu trừu tượng này, chúng ta có thể dùng mảng các nút (trường *Link* chứa chỉ số của nút kế tiếp) hoặc biến cấp phát động (trường *Link* chứa con trỏ tới nút kế tiếp). Tuy nhiên vì cấu trúc nối đơn, việc xác định phần tử đứng thứ p trong danh sách bắt buộc phải duyệt từ đầu danh sách qua p nút, việc này mất thời gian trung bình $\Theta(n)$, và tỏ ra không hiệu quả như thao tác trên mảng. Nói cách khác, danh sách nối đơn tiện lợi cho việc truy cập tuần tự nhưng không hiệu quả nếu chúng ta thực hiện nhiều phép truy cập ngẫu nhiên danh sách.

5.3.2. Chèn phần tử vào danh sách nối đơn

Để chèn thêm một nút chứa giá trị v vào vị trí của nút p trong danh sách nối đơn, trước hết ta tạo ra một nút mới *NewNode* chứa giá trị v và cho nút này liên kết tới p. Nếu p đang là nút đầu tiên của danh sách (*head*) thì cập nhật lại *head* bằng *NewNode*, còn nếu p không phải nút đầu tiên của danh sách, ta tìm nút q là nút đứng liền trước nút p và chỉnh lại liên kết: q liên kết tới *NewNode* thay vì liên kết tới thẳng p (Hình 5-2).



Hình 5-2. Chèn phần tử vào danh sách nối đơn

```
procedure Insert(p: PNode; const v: TElement); //Thủ tục chèn phần tử v vào vị trí nút p
var
  NewNode, q: PNode;
begin
  New(NewNode);
  NewNode^.Info := v;
  NewNode^.Link := p;
  if head = p then
    head := NewNode
  else
    begin
      q := head;
      while q^.Link ≠ p do q := q^.Link;
      q^.Link := NewNode;
    end;
end;
```

Việc chỉnh lại liên kết trong phép chèn phần tử vào danh sách nối đơn mất thời gian $\Theta(1)$, tuy nhiên việc tìm nút đứng liền trước nút p yêu cầu phải duyệt từ đầu danh sách, việc này mất thời gian trung bình $\Theta(n)$. Vậy phép chèn một phần tử vào danh sách nối đơn mất thời gian trung bình $\Theta(n)$ để thực hiện.

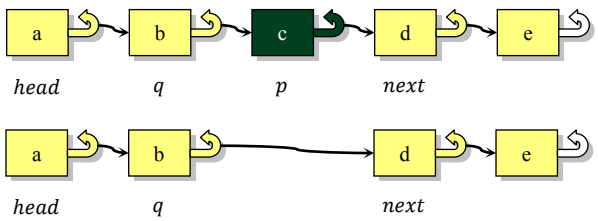
5.3.3. Xóa phần tử khỏi danh sách nối đơn:

Để xóa nút p khỏi danh sách nối đơn, gọi $next$ là nút đứng liền sau p trong danh sách. Xét hai trường hợp:

- Nếu p là nút đầu tiên trong danh sách $head = p$ thì ta đặt lại $head$ bằng $next$.
- Nếu p không phải nút đầu tiên trong danh sách, tìm nút q là nút đứng liền trước nút p và chỉnh lại liên kết: q liên kết tới $next$ thay vì liên kết tới p (Hình 5-3)

Việc cuối cùng là huỷ nút p.

```
procedure Delete(p: PNode); //Thủ tục xóa nút p của danh sách nối đơn
var
  Next, q: PNode;
begin
  Next := p^.Link;
  if p = head then head := Next
  else
    begin
      q := head;
      while q^.Link <> p do q := q^.Link;
      q^.Link := Next;
    end;
  Dispose(p);
end;
```



Hình 5-3. Xóa phần tử khỏi danh sách nối đơn

Cũng giống như phép chèn, phép xóa một phần tử khỏi danh sách nối đơn cũng mất thời gian trung bình $\Theta(n)$ để thực hiện.

Trên đây mô tả các thao tác với danh sách biểu diễn dưới dạng danh sách nối đơn các biến động. Chúng ta có thể cài đặt danh sách nối đơn bằng một mảng, mỗi nút chứa trong một phần tử của mảng và trường liên kết *link* chính là chỉ số của nút kế tiếp. Khi đó mọi thao tác chèn/xóa phần tử cũng được thực hiện tương tự như trên:

```
const
  max = 10000;
type
  TNode = record
    Info: TElement;
    Link: Integer;
  end;
  TList = array[1..max] of TNode;
var
  Nodes: TList;
  FirstNode: Integer;
```

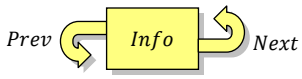
5.4. Biểu diễn danh sách bằng danh sách nối kép

Việc xác định nút đứng liền trước một nút p trong danh sách nối đơn bắt buộc phải duyệt từ đầu danh sách, thao tác này mất thời gian trung bình $\Theta(n)$ để thực hiện và ảnh hưởng

trực tiếp tới thời gian thực hiện thao tác chèn/xóa phần tử. Để khắc phục nhược điểm này, người ta sử dụng danh sách nối kép.

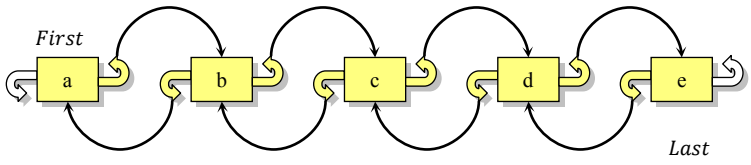
Danh sách nối kép gồm các nút được nối với nhau theo hai chiều. Mỗi nút là một bản ghi (record) gồm ba trường:

- Trường *Info* chứa giá trị lưu trong nút đó.
- Trường $Next$ chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó là nút nào, trong trường hợp nút đứng cuối cùng trong danh sách (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt (chẳng hạn con trỏ nil)
- Trường $Prev$ chứa liên kết (con trỏ) tới nút liền trước, tức là chứa một thông tin đủ để biết nút liền trước nút đó là nút nào, trong trường hợp nút đứng đầu tiên trong danh sách (không có nút liền trước), trường liên kết này được gán một giá trị đặc biệt (chẳng hạn con trỏ nil)




```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record; //Kiểu biến động chứa thông tin trong một nút
    Info: TElement;
    Next, Prev: PNode;
  end;
```

Khác với danh sách nối đơn, trong danh sách nối kép ta quan tâm tới hai nút: Nút đầu tiên ($First$) và phần tử cuối cùng ($Last$). Có hai cách duyệt danh sách nối kép: Hoặc bắt đầu từ $First$, dựa vào liên kết $Next$ để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua $Last$) thì dừng lại. Hoặc bắt đầu từ $Last$, dựa vào liên kết $Prev$ để đi sang nút liền trước, đến khi gặp giá trị đặc biệt (duyệt qua $First$) thì dừng lại



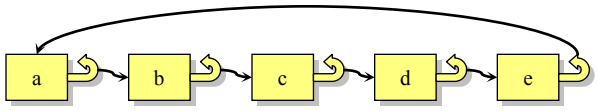
Hình 5-4. Danh sách nối kép

Giống như danh sách nối đơn, việc chèn/xóa nút trong danh sách nối kép cũng đơn giản chỉ là kỹ thuật chỉnh lại các mối liên kết giữa các nút cho hợp lý. Tuy nhiên ta có thể xác định được dễ dàng nút đứng liền trước/liền sau của một nút trong thời gian $\Theta(1)$, nên các

thao tác chèn/xóa trên danh sách nối kép chỉ mất thời gian $\Theta(1)$, tốt hơn hẳn so với cài đặt bằng mảng hay danh sách nối đơn.

5.5. Biểu diễn danh sách bằng danh sách nối vòng đơn

Trong danh sách nối đơn, phần tử cuối cùng trong danh sách có trường liên kết được gán một giá trị đặc biệt (thường sử dụng nhất là giá trị nil). Nếu ta cho trường liên kết của phần tử cuối cùng trỏ thẳng về phần tử đầu tiên của danh sách thì ta sẽ được một kiểu danh sách mới gọi là danh sách nối vòng đơn.



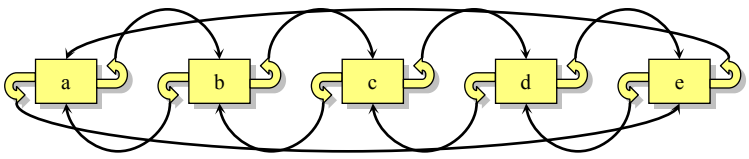
Hình 5-5. Danh sách nối vòng đơn

Đối với danh sách nối vòng đơn, ta chỉ cần biết một nút bất kỳ của danh sách là ta có thể duyệt được hết các nút trong danh sách bằng cách đi theo hướng liên kết. Chính vì lý do này, khi chèn/xóa vào danh sách nối vòng đơn, ta không phải xử lý các trường hợp riêng khi nút đứng đầu danh sách. Mặc dù vậy, danh sách nối vòng đơn vẫn cần thời gian trung bình $\Theta(n)$ để thực hiện thao tác chèn/xóa vì việc xác định nút đứng liền trước một nút cho trước cũng gặp trở ngại như với danh sách nối đơn.

5.6. Biểu diễn danh sách bằng danh sách nối vòng kép

Danh sách nối vòng đơn chỉ cho ta duyệt các nút của danh sách theo một chiều, nếu cài đặt bằng danh sách nối vòng kép thì ta có thể duyệt các nút của danh sách cả theo chiều ngược lại nữa. Danh sách nối vòng kép có thể tạo thành từ danh sách nối kép nếu ta cho trường $Prev$ của nút $First$ liên kết tới nút $Last$ còn trường $Next$ của nút $Last$ thì liên kết tới nút $First$.

Tương tự như danh sách nối kép, danh sách nối vòng kép cho phép thao tác chèn/xóa phần tử có thể thực hiện trong thời gian $\Theta(1)$.



Hình 5-6. Danh sách nối vòng kép

5.7. Biểu diễn danh sách bằng cây

Có nhiều thao tác trên danh sách, nhưng những thao tác phổ biến nhất là truy cập phần tử, chèn và xóa phần tử. Ta đã khảo sát cách cài đặt danh sách bằng mảng hoặc danh sách liên kết, nếu như mảng cho phép thao tác truy cập ngẫu nhiên tốt hơn danh sách liên kết, thì thao tác chèn/xóa phần tử trên mảng lại mất khá nhiều thời gian.

Dưới đây là bảng so sánh thời gian thực hiện các thao tác trên danh sách.

Phương pháp	Truy cập ngẫu nhiên	Chèn	Xóa
Mảng	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Danh sách nối đơn	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Danh sách nối kép	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Danh sách nối vòng đơn	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Danh sách nối vòng kép	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Cây là một kiểu dữ liệu trừu tượng mà trong một số trường hợp có thể gián tiếp dùng để biểu diễn danh sách. Với một cách đánh số thứ tự cho các nút của cây (duyệt theo thứ tự giữa), mỗi phép truy cập ngẫu nhiên, chèn, xóa phần tử trên danh sách có thể thực hiện trong thời gian $O(\log n)$. Chúng ta sẽ tiếp tục chủ đề này trong một bài riêng.

Bài tập 5-1.

Viết chương trình thực hiện các phép chèn, xóa, và tìm kiếm một phần tử trong danh sách các số nguyên đã sắp xếp theo thứ tự tăng dần biểu diễn bởi:

- Mảng
- Danh sách nối đơn
- Danh sách nối kép

Bài tập 5-2.

Viết chương trình nối hai danh sách số nguyên đã sắp xếp, tổng quát hơn, viết chương trình nối k danh sách số nguyên đã sắp xếp để được một danh sách gồm tất cả các phần tử

Bài tập 5-3.

Giả sử chúng ta biểu diễn một đa thức $p(x) = a_1x^{b_1} + a_2x^{b_2} + \cdots + a_nx^{b_n}$, trong đó $b_1 > b_2 > \cdots > b_n$ dưới dạng một danh sách nối đơn mà nút thứ i của danh sách chứa hệ số a_i, số mũ b_i và con trỏ tới nút kế tiếp (nút $i + 1$). Hãy tìm thuật toán cộng và nhân hai đa thức theo các biểu diễn này.

Bài tập 5-4.

Một số nhị phân $a_n a_{n-1} \dots a_0$, trong đó $a_i \in \{0,1\}$ có giá trị bằng $\sum_{i=0}^{n} a_i 2^i$. Người ta biểu diễn số nhị phân này bằng một danh sách nối đơn gồm n nút, có nút đầu danh sách chứa giá trị a_n, mỗi nút trong danh sách chứa một chữ số nhị phân a_i và con trỏ tới nút kế tiếp là nút chứa chữ số nhị phân a_{i-1}. Hãy lập chương trình thực hiện phép toán "cộng 1" trên số nhị phân đã cho và đưa ra biểu diễn nhị phân của kết quả.

Gợi ý: Sử dụng phép đệ quy

Bài 6. Ngăn xếp và Hàng đợi

Ngăn xếp và hàng đợi là hai kiểu dữ liệu trừu tượng rất quan trọng và được sử dụng nhiều trong thiết kế thuật toán. Về bản chất, ngăn xếp và hàng đợi là danh sách tức là một tập hợp các phần tử cùng kiểu có tính thứ tự.

Trong phần này chúng ta sẽ tìm hiểu hoạt động của ngăn xếp và hàng đợi và cách cài đặt chúng bằng các cấu trúc dữ liệu. Tương tự như danh sách, ta gọi kiểu dữ liệu của các phần tử sẽ chứa trong ngăn xếp và hàng đợi là $TElement$. Khi cài đặt chương trình cụ thể, kiểu $TElement$ có thể là kiểu số nguyên, số thực, ký tự, hay bất kỳ kiểu dữ liệu nào được chương trình dịch chấp nhận.

6.1. Ngăn xếp

Ngăn xếp (*Stack*) là một kiểu danh sách mà việc bổ sung một phần tử và loại bỏ một phần tử được thực hiện ở cuối danh sách.

Có thể hình dung ngăn xếp như một chồng đĩa, đĩa nào được đặt vào chồng sau cùng sẽ nằm trên tất cả các đĩa khác và sẽ được lấy ra đầu tiên. Vì nguyên tắc "vào sau ra trước", ngăn xếp còn có tên gọi là *danh sách kiểu LIFO* (*Last In First Out*). Vị trí cuối danh sách được gọi là *đỉnh* (*Top*) của ngăn xếp.

Đối với ngăn xếp có sáu thao tác cơ bản:

- $Init$: Khởi tạo một ngăn xếp rỗng
- $IsEmpty$: Cho biến ngăn xếp có rỗng không?
- $IsFull$: Cho biết ngăn xếp có đầy không?
- Get: Đọc giá trị phần tử ở đỉnh ngăn xếp
- $Push$: Đẩy một phần tử vào ngăn xếp
- Pop: Lấy ra một phần tử từ ngăn xếp

6.1.1. Biểu diễn ngăn xếp bằng mảng

Cách biểu diễn ngăn xếp bằng mảng cần có một mảng $Items$ để lưu các phần tử trong ngăn xếp và một biến nguyên *Top* để lưu chỉ số của phần tử tại đỉnh ngăn xếp. Ví dụ:

```
const
  max = ...; //Dung lượng cực đại của ngăn xếp
type
  TStack = record
    Items: array[1..max] of TElement;
    Top: Integer;
  end;
var
  Stack: TStack;
```

Sáu thao tác cơ bản của ngăn xếp có thể viết như sau:

```
//Khởi tạo ngăn xếp rỗng
procedure Init;
begin
  Stack.Top := 0;
end;

//Hàm kiểm tra ngăn xếp có rỗng không?
function IsEmpty: Boolean;
begin
  Result := Stack.Top = 0;
end;

//Hàm kiểm tra ngăn xếp có đầy không?
function IsFull: Boolean;
begin
  Result := Stack.Top = max;
end;

//Đọc giá trị phần tử ở đỉnh ngăn xếp
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    with Stack do Result := Items[Top]; //Trả về phần tử ở đỉnh ngăn xếp
end;

//Đẩy một phần tử x vào ngăn xếp
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Stack is Full" //Báo lỗi ngăn xếp đầy
  else
    with Stack do
      begin
        Top := Top + 1; //Tăng chỉ số đỉnh Stack
        Items[Top] := x; //Đặt x vào vị trí đỉnh Stack
      end;
end;
```

```
//Lấy một phần tử ra khỏi ngăn xếp
function Pop: TElement;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    with Stack do
      begin
        Result := Items[Top]; //Trả về phần tử ở đỉnh ngăn xếp
        Top := Top - 1; //Giảm chỉ số đỉnh ngăn xếp
      end;
end;
```

6.1.2. Biểu diễn ngăn xếp bằng danh sách nối đơn kiểu LIFO

Ta sẽ trình bày cách cài đặt ngăn xếp bằng danh sách nối đơn các biến động và con trỏ. Trong cách cài đặt này, ngăn xếp sẽ bị đầy nếu như vùng không gian nhớ dùng cho các biến động không còn đủ để thêm một phần tử mới. Tuy nhiên, việc kiểm tra điều này phụ thuộc vào máy tính, chương trình dịch và ngôn ngữ lập trình. Mặt khác, không gian bộ nhớ dùng cho các biến động thường rất lớn nên ta sẽ không viết mã cho hàm $IsFull$: Kiểm tra ngăn xếp tràn.

Các khai báo dữ liệu:

```
type
  PNode = ^TNode; //Kiểu con trỏ liên kết giữa các nút
  TNode = record //Kiểu dữ liệu cho một nút
    Info: TElement;
    Link: PNode;
  end;
var
  Top: PNode; //Con trỏ tới phần tử đỉnh ngăn xếp
```

Các thao tác trên ngăn xếp:

```
//Khởi tạo ngăn xếp rỗng
procedure Init;
begin
  Top := nil;
end;

//Kiểm tra ngăn xếp có rỗng không
function IsEmpty: Boolean;
begin
  Result := Top = nil;
end;
```

```
//Đọc giá trị phần tử ở đỉnh ngăn xếp
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    Result := Top^.Info;
end;

//Đẩy một phần tử x vào ngăn xếp
procedure Push(const x: TElement);
var
  p: PNode;
begin
  New(p); //Tạo nút mới
  p^.Info := x;
  p^.Link := Top; //Nối vào danh sách liên kết
  Top := p; //Dịch con trỏ đỉnh ngăn xếp
end;

//Lấy một phần tử khỏi ngăn xếp
function Pop: TElement;
var
  p: PNode;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    begin
      Result := Top^.Info; //Lấy phần tử tại con trỏ Top
      p := Top^.Link;
      Dispose(Top); //Giải phóng bộ nhớ
      Top := P; //Dịch con trỏ đỉnh ngăn xếp
    end;
end;
```

6.2. Hàng đợi

Hàng đợi (*Queue*) là một kiểu danh sách mà việc bổ sung một phần tử được thực hiện ở cuối danh sách và việc loại bỏ một phần tử được thực hiện ở đầu danh sách.

Khi cài đặt hàng đợi, có hai vị trí quan trọng là vị trí đầu danh sách ($Front$), nơi các phần tử được lấy ra, và vị trí cuối danh sách ($Rear$), nơi các phần tử được đưa vào.

Có thể hình dung hàng đợi như một đoàn người xếp hàng mua vé: Người nào xếp hàng trước sẽ được mua vé trước. Vì nguyên tắc "vào trước ra trước", hàng đợi còn có tên gọi là *danh sách kiểu FIFO* (*First In First Out*).

Tương tự như ngăn xếp, có sáu thao tác cơ bản trên hàng đợi:

- $Init$: Khởi tạo một hàng đợi rỗng
- $sEmpty$: Cho biến hàng đợi có rỗng không?

- $IsFull$: Cho biết hàng đợi có đầy không?
- Get: Đọc giá trị phần tử ở đầu hàng đợi
- $Push$: Đẩy một phần tử vào hàng đợi
- Pop: Lấy ra một phần tử từ hàng đợi

6.2.1. Biểu diễn hàng đợi bằng mảng

Ta có thể biểu diễn hàng đợi bằng một mảng $Items$ để lưu các phần tử trong hàng đợi, một biến nguyên $Front$ để lưu chỉ số phần tử đầu hàng đợi và một biến nguyên $Rear$ để lưu chỉ số phần tử cuối hàng đợi. Chỉ một phần của mảng $Items$ từ vị trí $Front$ tới $Rear$ được sử dụng lưu trữ các phần tử trong hàng đợi. Ví dụ:

```
const
  max = ...; //Dung lượng cực đại
type
  TQueue = record
    Items: array[1..max] of TElement;
    Front, Rear: Integer;
  end;
var
  Queue: TQueue;
```

Sáu thao tác cơ bản trên hàng đợi có thể viết như sau:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  Queue.Front := 1;
  Queue.Rear := 0;
end;

//Kiểm tra hàng đợi có rỗng không
function IsEmpty: Boolean;
begin
  Result := Queue.Front > Queue.Rear;
end;

//Kiểm tra hàng đợi có đầy không
function IsFull: Boolean;
begin
  Result := Queue.Rear = max;
end;
```

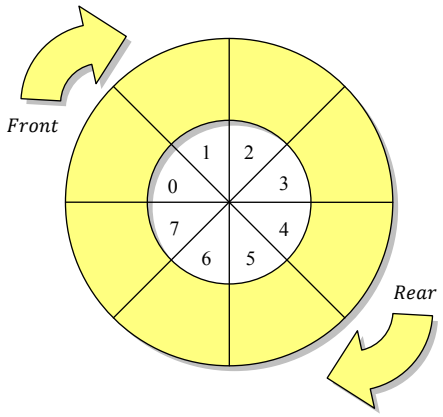
```
//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do Result := Items[Front];
end;

//Đẩy một phần tử x vào hàng đợi
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Queue is Full" //Báo lỗi hàng đợi đầy
  else
    with Queue do
      begin
        Rear := Rear + 1;
        Items[Rear] := x;
      end;
end;

//Lấy một phần tử khỏi hàng đợi
function Pop: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do
      begin
        Result := Items[Front];
        Front := Front + 1;
      end;
end;
```

6.2.2. Biểu diễn hàng đợi bằng danh sách vòng

Xét việc biểu diễn ngăn xếp và hàng đợi bằng mảng, giả sử mảng có tối đa 10 phần tử, ta thấy rằng nếu như làm 6 lần thao tác $Push$, rồi 4 lần thao tác Pop, rồi tiếp tục 8 lần thao tác $Push$ nữa thì không có vấn đề gì xảy ra cả. Lý do là vì chỉ số Top lưu đỉnh của ngăn xếp sẽ được tăng lên 6000, rồi giảm về 2000, sau đó lại tăng trở lại lên 10000 (chưa vượt quá chỉ số mảng). Nhưng nếu ta thực hiện các thao tác đó đối với cách cài đặt hàng đợi như trên thì sẽ gặp thông báo lỗi tràn mảng, bởi mỗi lần đẩy phần tử vào ngăn xếp, chỉ số cuối hàng đợi $Rear$ luôn tăng lên và không bao giờ bị giảm đi cả. Đó chính là nhược điểm mà ta nói tới khi cài đặt: Chỉ có các phần tử từ vị trí $Front$ tới $Rear$ là thuộc hàng đợi, các phần tử từ vị trí 1 tới $Front - 1$ là vô nghĩa.

Để khắc phục điều này, ta có thể biểu diễn hàng đợi bằng một danh sách vòng (dùng mảng hoặc danh sách nối vòng đơn): coi như các phần tử của hàng đợi được xếp quanh vòng tròn theo một chiều nào đó (chẳng hạn chiều kim đồng hồ). Các phần tử nằm trên

phần cung tròn từ vị trí $Front$ tới vị trí $Rear$ là các phần tử của hàng đợi. Có thêm một biến n lưu số phần tử trong hàng đợi. Việc đẩy thêm một phần tử vào hàng đợi tương đương với việc ta dịch chỉ số $Rear$ theo chiều vòng một vị trí rồi đặt giá trị mới vào đó. Việc lấy ra một phần tử trong hàng đợi tương đương với việc lấy ra phần tử tại vị trí $Front$ rồi dịch chỉ số $Front$ theo chiều vòng. (Hình 6-1)



Hình 6-1. Dùng danh sách vòng mô tả hàng đợi

Để tiện cho việc dịch chỉ số theo vòng, khi cài đặt danh sách vòng bằng mảng, người ta thường dùng cách đánh chỉ số từ 0 để tiện sử dụng phép chia lấy dư (modulus - mod).

```
const
  max = ...; //Dung lượng cực đại
type
  TQueue = record
    Items: array[0..max - 1] of TElement;
    n, Front, Rear: Integer;
  end;
var
  Queue: TQueue;
```

Sáu thao tác cơ bản trên hàng đợi cài đặt trên danh sách vòng được viết dưới dạng giả mã như sau:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  with Queue do
    begin
      Front := 0;
      Rear := max - 1;
      n := 0;
    end;
end;
```

```
//Kiểm tra hàng đợi có rỗng không
function IsEmpty: Boolean;
begin
  Result := Queue.n = 0;
end;

//Kiểm tra hàng đợi có đầy không
function IsFull: Boolean;
begin
  Result := Queue.n = max;
end;

//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do Result := Items[Front];
end;

//Đẩy một phần tử vào hàng đợi
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Queue is Full" //Báo lỗi hàng đợi đầy
  else
    with Queue do
      begin
        Rear := (Rear + 1) mod max;
        Items[Rear] := x;
        Inc(n);
      end;
end;

//Lấy một phần tử ra khỏi hàng đợi
function Pop: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do
      begin
        Result := Items[Front];
        Front := (Front + 1) mod max;
        Dec(n);
      end;
end;
```

6.2.3. Biểu diễn hàng đợi bằng danh sách nối đơn kiểu FIFO

Tương tự như cài đặt ngăn xếp bằng biến động và con trỏ trong một danh sách nối đơn, ta cũng không viết hàm *IsFull* để kiểm tra hàng đợi đầy.

Các khai báo dữ liệu:

```
type
  PNode = ^TNode; //Kiểu con trỏ liên kết giữa các nút
  TNode = record //Kiểu dữ liệu cho một nút
    Info: TElement;
    Link: PNode;
  end;
var
  Front, Rear: PNode; //Con trỏ tới phần tử đầu và cuối hàng đợi
```

Các thao tác trên hàng đợi:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  Front := nil;
end;

//Kiểm tra hàng đợi có rỗng không
function IsEmpty: Boolean;
begin
  Result := Front = nil;
end;

//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    Result := Front^.Info;
end;

//Đẩy một phần tử x vào hàng đợi
procedure Push(const x: TElement);
var
  p: PNode;
begin
  New(p); //Tạo một nút mới
  p^.Info := x;
  p^.Link := nil;
  //Nối nút đó vào danh sách
  if Front = nil then
    Front := p
  else
    Rear^.Link := p;
  Rear := p; //Dịch con trỏ Rear
end;
```

```
//Lấy một phần tử ra khỏi hàng đợi
function Pop: TElement;
var
  P: PNode;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    begin
      Result := Front^.Info; //Lấy phần tử tại con trỏ Front
      P := Front^.Link;
      Dispose(Front); //Giải phóng bộ nhớ
      Front := p; //Dịch con trỏ Front
    end;
end;
```

6.3. Một số chú ý về kỹ thuật cài đặt

Ngăn xếp và hàng đợi là hai kiểu dữ liệu trừu tượng tương đối dễ cài đặt, các thủ tục và hàm mô phỏng các thao tác có thể viết rất ngắn. Tuy vậy trong các chương trình dài, các thao tác vẫn nên được tách biệt ra thành chương trình con để dễ dàng gỡ rối hoặc thay đổi cách cài đặt (ví dụ đổi từ cài đặt bằng mảng sang cài đặt bằng danh sách nối đơn). Điều này còn giúp ích cho lập trình viên trong trường hợp muốn biểu diễn các kiểu dữ liệu trừu tượng bằng các lớp và đối tượng. Nếu có băn khoăn rằng việc gọi thực hiện chương trình con sẽ làm chương trình chạy chậm hơn việc viết trực tiếp, bạn có thể đặt các thao tác đó dưới dạng inline functions.

Bài tập 6-1.

Hàng đợi hai đầu (*doubled-ended queue*) là một danh sách được trang bị bốn thao tác:

$PushF(v)$: Đẩy phần tử v vào đầu danh sách

$PushR(v)$: Đẩy phần tử v vào cuối danh sách

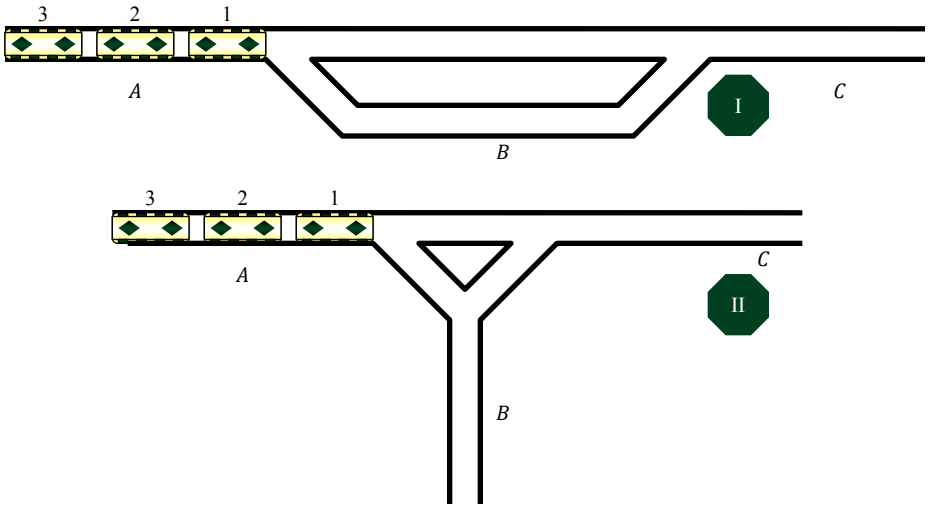
$PopF$: Loại bỏ phần tử đầu danh sách

$PopR$: Loại bỏ phần tử cuối danh sách

Hãy tìm cấu trúc dữ liệu thích hợp để cài đặt kiểu dữ liệu trừu tượng hàng đợi hai đầu.

Bài tập 6-2.

Có hai sơ đồ đường ray xe lửa bố trí như hình sau:



Ban đầu có n toa tàu xếp theo thứ tự từ 1 tới n từ phải qua trái trên đường ray A. Người ta muốn xếp lại các toa tàu theo thứ tự mới từ phải qua trái $(p_1, p_2, \ldots, p_n)$ lên đường ray C theo nguyên tắc: Các toa tàu không được "vượt nhau" trên ray, mỗi lần chỉ được chuyển một toa tàu từ $A \to B$, $A \to B$ hoặc $A \to C$. Hãy cho biết điều đó có thể thực hiện được trên sơ đồ đường ray nào trong hai sơ đồ trên.

Bài 7. Cây

7.1. Định nghĩa

Cây là một kiểu dữ liệu trừu tượng gồm một tập hữu hạn các nút, giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha-con". Có một nút đặc biệt gọi là gốc (*root*).

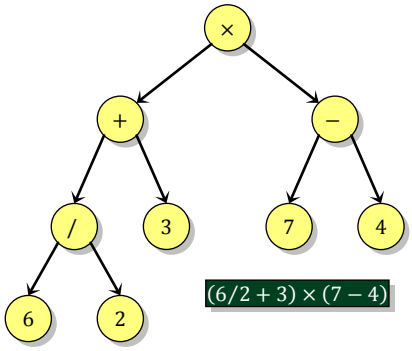
Có thể định nghĩa cây bằng cách đệ quy như sau:

- Một nút là một cây, nút đó cũng là gốc của cây ấy
- Nếu r là một nút và $r_1, r_2, \ldots, r_k$ lần lượt là gốc của các cây $T_1, T_2, \ldots, T_k$, thì ta có thể xây dựng một cây mới T bằng cách cho nút r trở thành cha của các nút $r_1, r_2, \ldots, r_k$. Cây T này nút gốc là r còn các cây $T_1, T_2, \ldots, T_k$ trở thành các cây con hay nhánh con (subtree) của nút gốc.

Để tiện, người ta còn cho phép tồn tại một cây không có nút nào mà ta gọi là cây rỗng (null tree), ký hiệu Λ.

Một vài hình ảnh của cấu trúc cây:

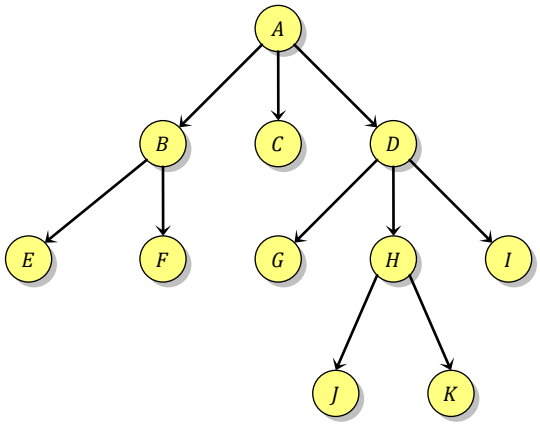
- Mục lục của một cuốn sách với phần, chương, bài, mục v.v… có cấu trúc của cây
- Cấu trúc thư mục trên đĩa cũng có cấu trúc cây, thư mục gốc có thể coi là gốc của cây đó với các cây con là các thư mục con (sub-directories) và tệp (files) nằm trên thư mục gốc.
- Gia phả của một họ tộc cũng có cấu trúc cây.
- Một biểu thức số học gồm các phép toán cộng, trừ, nhân, chia cũng có thể lưu trữ trong một cây mà các toán hạng được lưu trữ ở các nút lá, các toán tử được lưu trữ ở các nút nhánh, mỗi nhánh là một biểu thức con (Hình 7-1).



Hình 7-1. Cây biểu diễn biểu thức

7.2. Các khái niệm cơ bản

Nếu $(r_1, r_2, \dots, r_k)$ là dãy các nút trên cây sao cho r_i là nút cha của nút r_{i+1} với $\forall i: 1 \leq i < k$, thì dãy này được gọi là một *đường đi* (*path*) từ r_1 tới r_k. Chiều dài của đường đi bằng số nút trên đường đi trừ đi 1. Quy ước rằng có đường đi độ dài 0 từ một nút đến chính nó. Như cây ở Hình 7-2, (A, B, F) là đường đi độ dài 2, (A, D, H, K) là đường đi độ dài 3.



Hình 7-2. Cây

Nếu có một đường đi độ dài khác 0 từ nút a tới nút d thì nút a gọi là *tiền bối* (ancestor) của nút d và nút d được gọi là *hậu duệ* (descendant) của nút a. Như cây ở Hình 7-2, nút A là tiền bối của tất cả các nút trên cây, nút H là hậu duệ của nút A và nút D. Một số quy ước còn cho phép một nút là tiền bối cũng như hậu duệ của chính nút đó, trong trường hợp này, người ta có thêm khái niệm *tiền bối thực sự* (*proper ancestor*) và *hậu duệ đích thực* (*proper descendant*) trùng với khái niệm tiền bối và hậu duệ mà ta đã định nghĩa.

Trong cây, chỉ duy nhất một nút không có tiền bối là nút gốc. Một nút không có hậu duệ gọi là *nút lá* (*leaf*) của cây, các nút không phải lá được gọi là *nút nhánh* (branch). Như cây ở Hình 7-2, các nút C, E, F, G, I, J, K là các nút lá.

Độ cao của một nút là độ dài đường đi dài nhất từ nút đó tới một nút lá hậu duệ của nó, độ cao của nút gốc gọi là *chiều cao* (*height*) của cây. Như cây ở Hình 7-2, cây có chiều cao là 3 (đường đi (A, D, H, J)).

Độ sâu (*depth*) của một nút là độ dài đường đi duy nhất từ nút gốc tới nút đó. Như cây ở Hình 7-2, nút A có độ sâu là 0, nút B, C, D có độ sâu là 1, nút E, F, G, H, I có độ sâu là 2, và nút J, K có độ sâu là 3. Có thể định nghĩa chiều cao của cây là độ sâu lớn nhất của các nút trong cây.

Một tập hợp các cây đôi một không có nút chung được gọi là *rừng* (*forest*), có thể coi tập các cây con của một nút là một rừng.

Những nút con của cùng một nút được gọi là *anh em* (*sibling*). Với một cây, nếu chúng ta có tính đến thứ tự anh em thì cây đó gọi là *cây có thứ tự* (*ordered tree*), còn nếu chúng ta không quan tâm tới thứ tự anh em thì cây đó gọi là *cây không có thứ tự* (*unordered tree*).

7.3. Biểu diễn cây tổng quát

Trong thực tế, có một số ứng dụng đòi hỏi một cấu trúc dữ liệu dạng cây nhưng không có ràng buộc gì về số con của một nút trên cây, ví dụ như cấu trúc thư mục trên đĩa hay hệ thống đề mục của một cuốn sách. Khi đó, ta phải tìm cách mô tả một cách khoa học cấu trúc dữ liệu dạng cây tổng quát. Giả sử $TElement$ là kiểu dữ liệu của các phần tử chứa trong mỗi nút của cây, khi đó ta có thể biểu diễn cây bằng một trong các cấu trúc dữ liệu sau:

7.3.1. Biểu diễn bằng liên kết tới nút cha

Với T là một cây, trong đó các nút được đánh số từ 1 tới n, khi đó ta có thể gán cho mỗi nút i một nhãn $Parent[i]$ là số hiệu nút cha của nút i. Nếu nút i là nút gốc, thì $Parent[i]$ được gán giá trị 0. Cách biểu diễn này có thể cài đặt bằng một mảng các nút, mỗi nút là một bản ghi bên trong chứa giá trị lưu tại nút ($Info$) và nhãn $Parent$.

```
const
  max = ...; //Dung lượng cực đại
type
  TNode = record
    Info: TElement;
    Parent: Integer;
  end;
  TTree = array[1..max] of TNode;
var
  Tree: Tree;
```

Trong cách biểu diễn này, nếu chúng ta cần biết nút cha của một nút thì chỉ cần truy xuất trường *Parent* của nút đó. Tuy nhiên nếu ta cần liệt kê tất cả các nút con của một nút thì không có cách nào khác là phải duyệt toàn bộ danh sách nút và kiểm tra trường *Parent*. Thực hiện việc này mất thời gian $\Theta(n)$ với mỗi nút.

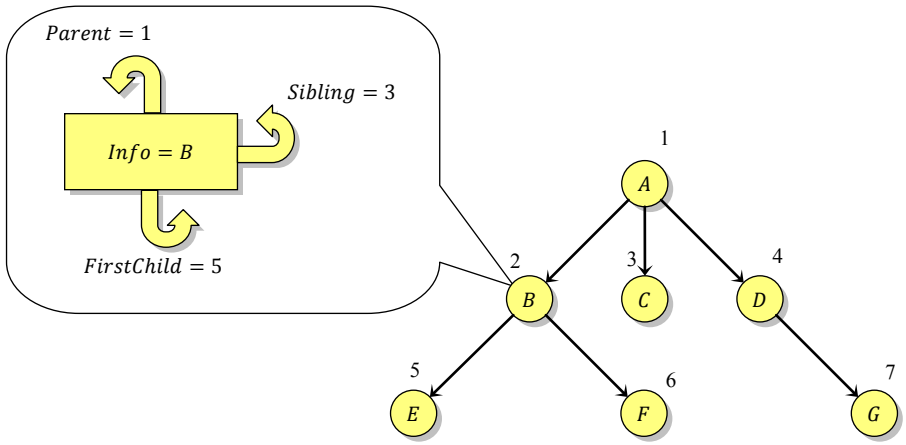
7.3.2. Biểu diễn bằng cấu trúc liên kết

Cách biểu diễn này coi mỗi nút của cây là một bản ghi gồm 4 trường:

- Trường $Info$: Chứa giá trị lưu trong nút

- Trường $Parent$: Chứa con trỏ liên kết tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đang xét là nút nào. Trong trường hợp nút đang xét là gốc (không có nút cha), trường $Parent$ được gán một giá trị đặc biệt (nil).
- Trường $FirstChild$: Chứa liên kết (con trỏ) tới nút con đầu tiên (con cả) của nút đang xét, trong trường hợp nút đang xét là nút lá (không có nút con), trường này được gán một giá trị đặc biệt (nil).
- Trường $Sibling$: Chứa liên kết (con trỏ) tới nút em kế cận (nút cùng cha với nút đang xét, khi sắp thứ tự các nút con thì nút $Sibling$ đứng liền sau nút đang xét). Trong trường hợp nút đang xét không có nút em, trường này được gán một giá trị đặc biệt (nil).

```
type
  PNode = ^TNode;
  TNode = record
    Info: TElement;
    Parent, FirstChild, Sibling: PNode;
  end;
```



Hình 7-3. Cấu trúc nút của cây tổng quát

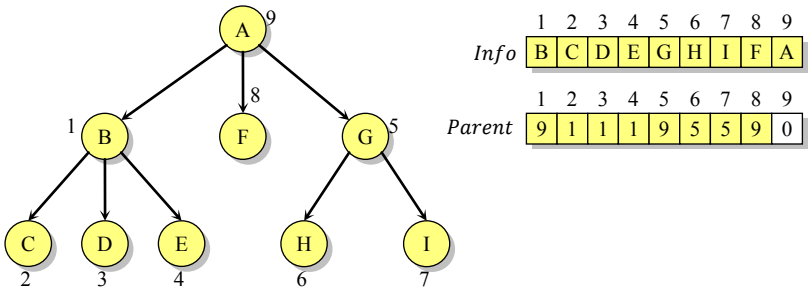
Trong các biểu diễn này, từ một nút r bất kỳ, ta có thể đi theo liên kết *FirstChild* để đến nút con đầu tiên, nút này chính là nút đầu tiên trong một danh sách nối đơn các nút con: Từ nút *FirstChild*, đi theo liên kết *Sibling*, ta có thể duyệt tất cả các nút con của nút r.

Trong trường hợp phải thực hiện nhiều lần phép chèn/xóa một cây con, người ta có thể biểu diễn danh sách các nút con của một nút dưới dạng danh sách móc nối kép để việc chèn/xóa được thực hiện hiệu quả hơn, khi đó thay vì trường liên kết đơn *Sibling*, mỗi nút sẽ có hai trường *Prev* và N*ext* chứa liên kết tới nút anh liền trước và em liền sau của một nút.

7.3.3. Biểu diễn bằng mảng

Để lưu trữ cây tổng quát có n nút bằng mảng, ta đánh số các nút trên cây bằng các số tự nhiên từ 1 tới n theo một thứ tự tùy ý và xây dựng cấu trúc dữ liệu gồm có:

- Một biến $Root$ lưu chỉ số của nút gốc.
- Một mảng $Info[1 \dots n]$, trong đó $Info[i]$ là giá trị lưu trong nút thứ i.
- Một mảng $Parent[1 \dots n]$, trong đó $Parent[i]$ là chỉ số nút cha của nút thứ i, quy ước $Parent[root] = 0$
- Một mảng $Children[1 \dots n-1]$ được chia làm n đoạn, đoạn thứ i gồm một dãy liên tiếp các phần tử là chỉ số các nút con của nút i. Nói cách khác, dùng mảng $Children$ để liệt kê các nút con của nút 1, tiếp theo đến các nút con của nút 2, … theo đúng thứ tự đánh chỉ số. Mảng $Children$ sẽ chứa tất cả chỉ số của mọi nút trên cây ngoại trừ nút gốc nên nó sẽ gồm $n-1$ phần tử. Lưu ý r ằng khi chia mảng Children làm n đoạn thì sẽ có những đoạn rỗng tương ứng với danh sách các nút con của một nút lá.
- Một mảng $Head[1 \dots n+1]$, để đánh dấu vị trí cắt đoạn trong mảng$Children$: $Head[i]$ là vị trí đứng liền trước đoạn thứ i, hay nói cách khác: Đoạn trong mảng $Children$ tính từ chỉ số $Head[i]+1$ tới $Head[i+1]$ sẽ được dùng để chứa chỉ số các nút con của nút thứ i. Khi $Head[i] = Head[i+1]$ có nghĩa là đoạn thứ i rỗng. Quy ước rằng $Head[n+1] = n-1$.

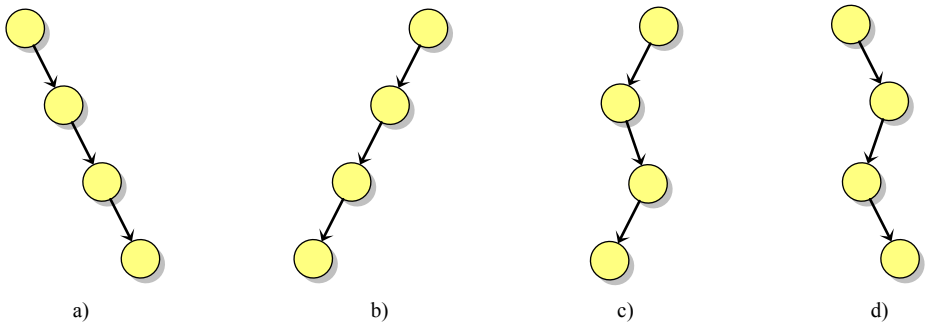


Hình 7-4. Biểu diễn cây tổng quát bằng mảng

7.4. Cây nhị phân

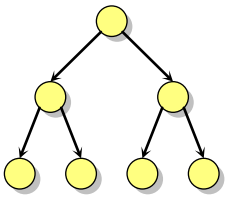
Cây nhị phân (*binary tree*) là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con. Với một nút thì người ta cũng phân biệt cây con trái và cây con phải của nút đó, tức là cây nhị phân là cây có thứ tự.

7.4.1. Một số dạng đặc biệt của cây nhị phân



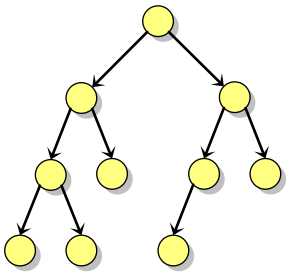
Hình 7-5. Cây nhị phân suy biến

Các cây nhị phân trong Hình 7-5 được gọi là *cây nhị phân suy biến* (degenerate binary tree), trong cây nhị phân suy biến, các nút không phải lá chỉ có đúng một cây con. Cây a) được gọi là cây lệch phải, cây b) được gọi là cây lệch trái, cây c) và d) được gọi là cây zíc-zắc.



Hình 7-6. Cây nhị phân hoàn chỉnh

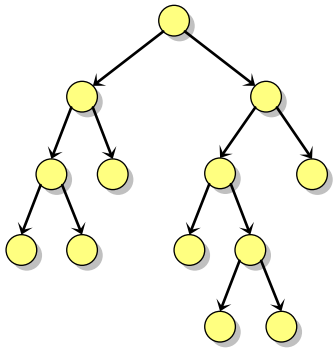
Cây trong Hình 7-6 được gọi là *cây nhị phân hoàn chỉnh* (*complete binary tree*). Cây nhị phân hoàn chỉnh có mọi nút lá nằm ở cùng một độ sâu và mọi nút nhánh đều có hai nhánh con. Số nút ở độ sâu h của cây nhị phân hoàn chỉnh là 2^h. Tổng số nút của cây nhị phân hoàn chỉnh độ cao h là $2^{h+1} - 1$



Hình 7-7. Cây nhị phân gần hoàn chỉnh

Cây trong hình Hình 7-7 được gọi là *cây nhị phân gần hoàn chỉnh* (*nearly complete binary tree*). Một cây nhị phân độ cao h được gọi là cây nhị phân gần hoàn chỉnh nếu ta

bỏ đi mọi nút ở độ sâu h thì được một cây nhị phân hoàn chỉnh. Cây nhị phân hoàn chỉnh hiển nhiên là cây nhị phân gần hoàn chỉnh.



Hình 7-8. Cây nhị phân đầy đủ

Cây trong Hình 7-8 được gọi là *cây nhị phân đầy đủ* (*full binary tree*). Cây nhị phân đầy đủ là cây nhị phân mà mọi nút nhánh của nó đều có hai nút con.

Dễ dàng chứng minh được những tính chất sau:

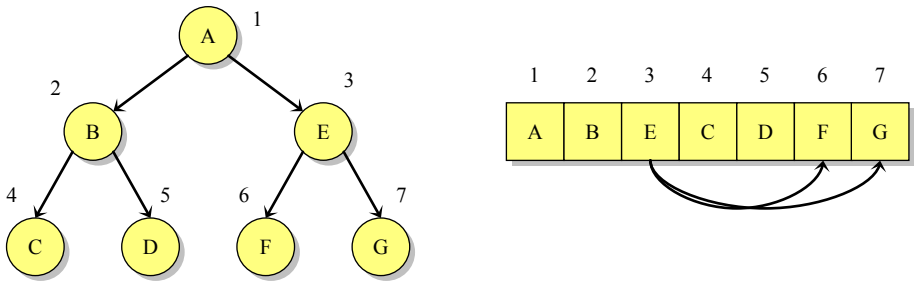
- Trong các cây nhị phân có cùng số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân gần hoàn chỉnh thì có chiều cao nhỏ nhất.
- Số lượng tối đa các nút ở độ sâu d của cây nhị phân là 2^d
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là $2^{h+1} - 1$
- Cây nhị phân gần hoàn chỉnh có n nút thì chiều cao của nó là $\lfloor \lg n \rfloor$.

7.4.2. Biểu diễn cây nhị phân

Rõ ràng có thể sử dụng các cách biểu diễn cây tổng quát để biểu diễn cây nhị phân, nhưng dựa vào những đặc điểm riêng của cây nhị phân, chúng ta có thể có những cách biểu diễn hiệu quả hơn. Trong phần này chúng ta xét một số cách biểu diễn đặc thù cho cây nhị phân, tương tự như với đối với cây tổng quát, ta gọi $TElement$ là kiểu dữ liệu của các phần tử chứa trong các nút của cây nhị phân.

❑ Biểu diễn bằng mảng

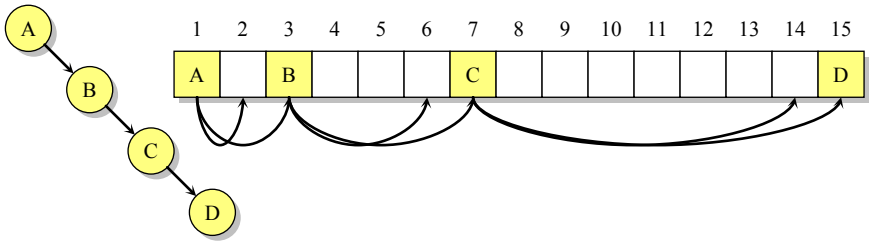
Một cây nhị phân hoàn chỉnh có n nút thì có chiều cao là $\lfloor \lg n \rfloor$, tức là các nút sẽ nằm ở các độ sâu từ 0 tới $\lfloor \lg n \rfloor$, Khi đó ta có thể liệt kê tất cả các nút từ độ sâu 0 (nút gốc) tới độ sâu $\lfloor \lg n \rfloor$, sao cho với các nút cùng độ sâu thì thứ tự liệt kê là từ trái qua phải. Thứ tự liệt kê cho phép ta đánh số các nút từ 1 tới n (Hình 7-9).



Hình 7-9. Đánh số các nút của cây nhị phân hoàn chỉnh để biểu diễn bằng mảng

Với cách đánh số này, hai con của nút thứ i sẽ là các nút thứ $2i$ và $2i + 1$. Cha của nút thứ i là nút thứ $\lfloor i/2 \rfloor$. Ta có thể lưu trữ cây bằng một mảng $Info$ trong đó phần tử chứa trong nút thứ i của cây được lưu trữ trong mảng bởi $Info[i]$. Với cây nhị phân ở Hình 7-9, ta có thể dùng mảng $Info = (A, B, E, C, D, E, G)$ để chứa các giá trị trên cây.

Trong trường hợp cây nhị phân không hoàn chỉnh, ta có thể thêm vào một số nút giả để được cây nhị phân hoàn chỉnh. Khi biểu diễn bằng mảng thì những phần tử tương ứng với các nút giả sẽ được gán một giá trị đặc biệt. Chính vì lý do này nên việc biểu diễn cây nhị phân không hoàn chỉnh bằng mảng sẽ rất lãng phí bộ nhớ trong trường hợp phải thêm vào nhiều nút giả. Ví dụ ta cần tới một mảng 15 phần tử để lưu trữ cây nhị phân lệch phải chỉ gồm 4 nút (Hình 7-10).



Hình 7-10. Nhược điểm của phương pháp biểu diễn cây nhị phân bằng mảng

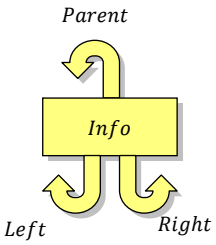
❑ Biểu diễn bằng cấu trúc liên kết.

Khi biểu diễn cây nhị phân bằng cấu trúc liên kết, mỗi nút của cây là một bản ghi (record) gồm 4 trường:

- Trường $Info$: Chứa giá trị lưu tại nút đó
- Trường $Parent$: Chứa liên kết (con trỏ) tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đó là nút nào, đối với nút gốc, trường này được gán một giá trị đặc biệt (nil).
- Trường $Left$: Chứa liên kết (con trỏ) tới nút con trái, tức là chứa một thông tin đủ để biết nút con trái của nút đó là nút nào, trong trường hợp không có nút con trái, trường này được gán một giá trị đặc biệt (nil).

- Trường $Right$: Chứa liên kết (con trỏ) tới nút con phải, tức là chứa một thông tin đủ để biết nút con phải của nút đó là nút nào, trong trường hợp không có nút con phải, trường này được gán một giá trị đặc biệt (nil).

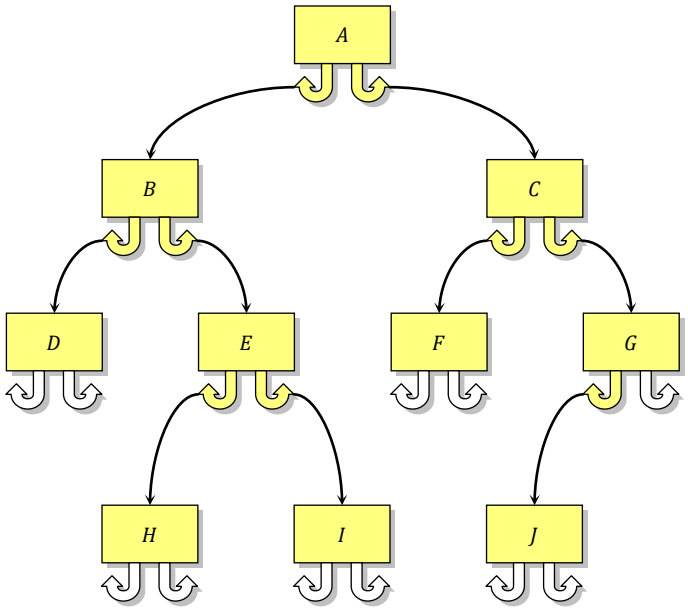
Đối với cây ta chỉ cần phải quan tâm giữ lại nút gốc ($Root$), bởi từ nút gốc, đi theo các hướng liên kết $Left$, $Right$ ta có thể duyệt mọi nút khác.




```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record //Cấu trúc biến động chứa thông tin trong một nút
    Info: TElement;
    Left, Right: PNode
  end;
var
  Root: PNode; //Con trỏ tới nút gốc
```

Trong trường hợp biết rõ giới hạn về số nút của cây, ta có thể lưu trữ các nút trong một mảng, và dùng chỉ số mảng như liên kết tới một nút:

```
const
  max = ...; //Dung lượng cực đại
type
  TNode = record //Cấu trúc biến động chứa thông tin trong một nút
    Info: TElement;
    Left, Right: Integer; //Chỉ số của nút con trái và nút con phải
  end;
  TTree = array[1..max] of TNode;
var
  Tree: TTree;
  Root: Integer; //Chỉ số của nút gốc
```



Hình 7-11. Biểu diễn cây nhị phân bằng cấu trúc liên kết

7.4.3. Phép duyệt cây nhị phân

Phép xử lý các nút trên cây mà ta gọi chung là phép thăm (Visit) các nút một cách hệ thống sao cho mỗi nút chỉ được thăm một lần gọi là phép duyệt cây.

Giả sử rằng cấu trúc một nút của cây được đặc tả như sau:

```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record //Cấu trúc biến động chứa thông tin trong một nút
    Info: TElement;
    Left, Right: PNode
  end;
var
  Root: PNode; //Con trỏ tới nút gốc
```

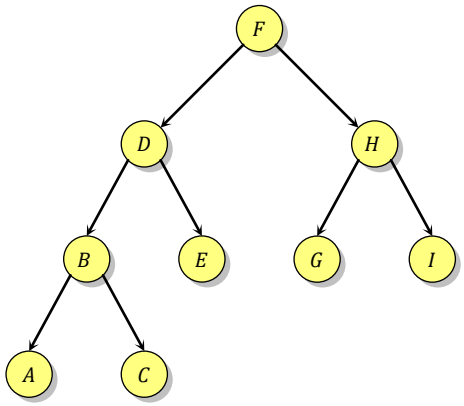
Quy ước rằng nếu như một nút không có nút con trái (hoặc nút con phải) thì liên kết $Left$ ($Right$) của nút đó được liên kết thẳng tới một nút đặc biệt mà ta gọi là nil, nếu cây rỗng thì nút gốc của cây đó cũng được gán bằng nil. Khi đó có ba cách duyệt cây hay được sử dụng:

❑ Duyệt theo thứ tự trước

Trong phép duyệt *theo thứ tự trước* (*preorder traversal*) thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê trước tất cả các giá trị lưu trong hai nhánh con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(Node: PNode); //Duyệt nhánh cây gốc node^
begin
  if Node ≠ nil then
    begin
      Output ← Node^.Info;
      Visit(Node^.Left);
      Visit(Node^.Right);
    end;
end;
```

Quá trình duyệt theo thứ tự trước bắt đầu bằng lời gọi $Visit(Root)$.



Hình 7-12

Hình 7-12 là một cây nhị phân có 9 nút. Nếu ta duyệt cây này theo thứ tự trước thì quá trình duyệt theo thứ tự trước sẽ lần lượt liệt kê các giá trị:

$$F, D, B, A, C, E, H, G, I$$

❑ Duyệt theo thứ tự giữa

Trong phép duyệt theo thứ tự giữa (*inorder traversal*) thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau tất cả các giá trị lưu ở nút con trái và được liệt kê trước tất cả các giá trị lưu ở nút con phải của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(Node: PNode); //Duyệt nhánh cây gốc node^
begin
  if Node ≠ nil then
    begin
      Visit(Node^.Left);
      Output ← Node^.Info;
      Visit(Node^.Right);
    end;
end;
```

Quá trình duyệt theo thứ tự giữa cũng bắt đầu bằng lời gọi $Visit(Root)$.

Nếu ta duyệt cây ở Hình 7-12 theo thứ tự giữa thì quá trình duyệt sẽ liệt kê lần lượt các giá trị:

$$A, B, C, D, E, F, G, H, I$$

❑ **Duyệt theo thứ tự sau**

Trong phép duyệt theo thứ tự sau thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau tất cả các giá trị lưu trong hai nhánh con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(Node: PNode); //Duyệt nhánh cây gốc node^
begin
  if Node ≠ nil then
    begin
      Visit(Node^.Left);
      Visit(Node^.Right);
      Output ← Node^.Info;
    end;
end;
```

Quá trình duyệt theo thứ tự sau cũng bắt đầu bằng lời gọi $Visit(Root)$.

Cũng với cây ở Hình 7-12, nếu ta duyệt theo thứ tự sau thì các giá trị sẽ lần lượt được liệt kê theo thứ tự:

$$A, C, B, E, D, G, I, H, F$$

7.5. Cây k-phân

Cây k-phân là một dạng cấu trúc cây mà mỗi nút trên cây có tối đa k nút con (có tính đến thứ tự của các nút con).

Cũng tương tự như việc biểu diễn cây nhị phân, người ta có thể thêm vào cây k-phân một số nút giả để cho mỗi nút nhánh của cây k-phân đều có đúng k nút con, các nút con được xếp thứ tự từ nút con thứ nhất tới nút con thứ k, sau đó đánh số các nút trên cây k-phân bắt đầu từ 0 trở đi, bắt đầu từ mức 1, hết mức này đến mức khác và từ "trái qua phải" ở mỗi mức.

Theo cách đánh số này, nút con thứ j của nút i sẽ là: $ki + j$. Nếu i không phải là nút gốc ($i > 0$) thì nút cha của nút i là nút $\lfloor (i-1)/k \rfloor$. Ta có thể dùng một mảng $Info$ đánh số từ 0 để lưu các giá trị trên các nút: Giá trị tại nút thứ i được lưu trữ ở phần tử $Info[i]$. Đây là cơ chế biểu diễn cây k-phân bằng mảng.

Cây k-phân cũng có thể biểu diễn bằng cấu trúc liên kết với cấu trúc dữ liệu cho mỗi nút của cây là một bản ghi (record) gồm 3 trường:

- Trường $Info$: Chứa giá trị lưu trong nút đó.

- Trường $Parent$: Chứa liên kết (con trỏ) tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đó là nút nào, đối với nút gốc, trường này được gán một giá trị đặc biệt (nil).
- Trường $Links$: Là một mảng gồm k phần tử, phần tử thứ i chứa liên kết (con trỏ) tới nút con thứ i, trong trường hợp không có nút con thứ i thì $Links[i]$ được gán một giá trị đặc biệt.

Đối với cây k-phân, ta cũng chỉ cần giữ lại nút gốc, bởi từ nút gốc, đi theo các hướng liên kết có thể đi tới mọi nút khác.

Bài tập 7-1.

Xét hai nút x, y trên một cây nhị phân, ta nói nút x nằm bên trái nút y (nút y nằm bên phải nút x) nếu:

- Hoặc nút x nằm trong nhánh con trái của nút y
- Hoặc nút y nằm trong nhánh con phải của nút x
- Hoặc tồn tại một nút z sao cho x nằm trong nhánh con trái và y nằm trong nhánh con phải của nút z

Chỉ ra rằng với hai nút x, y bất kỳ trên một cây nhị phân ($x \neq y$) chỉ có đúng một trong bốn mệnh đề sau là đúng:

- x nằm bên trái y
- x nằm bên phải y
- x là tiền bối thực sự của y
- y là tiền bối thực sự của x

Bài tập 7-2.

Với mỗi nút x trên cây nhị phân T, giả sử rằng ta biết được các giá trị $Preorder[x]$, $Inorder[x]$ và $Postorder[x]$ lần lượt là thứ tự duyệt trước, giữa, sau của x. Tìm cách chỉ dựa vào các giá trị này để kiểm tra hai nút có quan hệ tiền bối-hậu duệ hay không.

Bài tập 7-3.

Bậc (degree) của một nút là số nút con của nó. Chứng minh rằng trên cây nhị phân, số lá nhiều hơn số nút bậc 2 đúng một nút.

Bài tập 7-4.

Chỉ ra rằng cấu trúc của một cây nhị phân có thể khôi phục một cách đơn định nếu ta biết được thứ tự duyệt trước và giữa của các nút. Tương tự như vậy, cấu trúc cây có thể khôi phục nếu ta biết được thứ tự duyệt sau và giữa của các nút.

Bài tập 7-5.

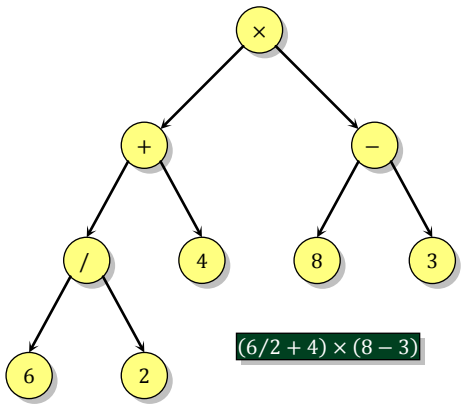
Tìm ví dụ về hai cây nhị phân khác nhau nhưng có thứ tự trước của các nút giống nhau và thứ tự sau của các nút cũng giống nhau trên hai cây.

Bài 8. Ký pháp tiền tố, trung tố và hậu tố

Để kết thúc chương này, chúng ta nói tới một ứng dụng của ngăn xếp và cây nhị phân: Bài toán phân tích và tính giá trị biểu thức.

8.1. Biểu thức dưới dạng cây nhị phân

Chúng ta có thể biểu diễn các biểu thức số học gồm các phép toán cộng, trừ, nhân, chia bằng một cây nhị phân đầy đủ, trong đó các nút lá biểu thị các toán hạng (hằng, biến), các nút không phải là lá biểu thị các toán tử (phép toán số học chẳng hạn). Mỗi phép toán trong một nút sẽ tác động lên hai biểu thức con nằm ở cây con bên trái và cây con bên phải của nút đó. Ví dụ: Biểu thức $(6/2+4)\times(8-3)$ được biểu diễn trong cây ở Hình 8-1.



Hình 8-1. Cây biểu diễn biểu thức

8.2. Các ký pháp cho cùng một biểu thức

Với cây nhị phân biểu diễn biểu thức trong Hình 8-1,

- Nếu duyệt theo thứ tự trước, ta sẽ được × + / 6 2 4 − 8 3, đây là *dạng tiền tố* (*prefix*) của biểu thức. Trong ký pháp này, toán tử được viết trước hai toán hạng tương ứng, người ta còn gọi ký pháp này là ký pháp Ba lan.
- Nếu duyệt theo thứ tự giữa, ta sẽ được 6 / 2 + 4 × 8 − 3 . Ký pháp này bị nhập nhằng vì thiếu dấu ngoặc. Nếu thêm vào thủ tục duyệt một cơ chế bổ sung các cặp dấu ngoặc vào mỗi biểu thức con, ta sẽ thu được biểu thức $\big(((6/2)+4)\times(8-3)\big)$. Ký pháp này gọi là *dạng trung tố* (*infix*) của một biểu thức (Thực ra chỉ cần thêm các dấu ngoặc đủ để tránh sự mập mờ mà thôi, không nhất thiết phải thêm vào đầy đủ các cặp dấu ngoặc).

- Nếu duyệt theo thứ tự sau, ta sẽ được $6\ 2\ /\ 4\ +\ 8\ 3\ -\ \times$, đây là *dạng hậu tố* (*postfix*) của biểu thức. Trong ký pháp này toán tử được viết sau hai toán hạng, người ta còn gọi ký pháp này là *ký pháp nghịch đảo Balan* (*Reverse Polish Notation - RPN*)

Chỉ có dạng trung tố mới cần có dấu ngoặc, dạng tiền tố và hậu tố không cần phải có dấu ngoặc. Chúng ta sẽ thảo luận về tính đơn định của dạng tiền tố và hậu tố trong phần sau.

8.3. Cách tính giá trị biểu thức

Có một vấn đề cần lưu ý là khi máy tính giá trị một biểu thức số học gồm các toán tử hai ngôi (toán tử gồm hai toán hạng như $+, -, \times, /$) thì máy chỉ thực hiện được phép toán đó với hai toán hạng. Nếu biểu thức phức tạp thì máy phải chia nhỏ và tính riêng từng biểu thức trung gian, sau đó mới lấy giá trị tìm được để tính tiếp*. Ví dụ như biểu thức $1 + 2 + 4$ máy sẽ phải tính $1 + 2$ trước được kết quả là 3 sau đó mới đem 3 cộng với 4 chứ không thể thực hiện phép cộng một lúc ba số được.

Khi lưu trữ biểu thức dưới dạng cây nhị phân thì ta có thể coi mỗi nhánh con của cây đó biểu diễn một biểu thức trung gian mà máy cần tính trước khi tính biểu thức lớn. Như ví dụ trên, máy sẽ phải tính hai biểu thức $6/2 + 4$ và $8 - 3$ trước khi làm phép tính nhân cuối cùng. Để tính biểu thức $6/2 + 4$ thì máy lại phải tính biểu thức $6/2$ trước khi đem cộng với 4.

Vậy để tính một biểu thức lưu trữ trong một nhánh cây nhị phân gốc r, máy sẽ làm giống như hàm đệ quy sau:

```
function Calculate(r: Nút): Giá trị; //Tính biểu thức con trong nhánh cây gốc r
begin
  if «nút r chứa một toán hạng» then
    Result := «Giá trị chứa trong nút r»
  else //Nút r chứa một toán tử ♦
    begin
      x := Calculate(nút con trái của r);
      y := Calculate(nút con phải của r);
      Result := x ♦ y;
    end;
end;
```

(Trong trường hợp lập trình trên các hệ thống song song, việc tính giá trị biểu thức ở cây con trái và cây con phải có thể tiến hành đồng thời làm giảm đáng kể thời gian tính toán biểu thức).

---

* Thực ra đây là việc của trình dịch ngôn ngữ bậc cao, còn máy chỉ tính các phép toán với hai toán hạng theo trình tự của các lệnh được phân tích ra.

8.4. Tính giá trị biểu thức hậu tố

Để ý rằng khi tính toán biểu thức, máy sẽ phải quan tâm tới việc tính biểu thức ở hai nhánh con trước, rồi mới xét đến toán tử ở nút gốc. Điều đó làm ta nghĩ t ới phép duyệt cây theo thứ tự sau và ký pháp hậu tố. Năm 1920, nhà lô-gic học người Balan Jan Łukasiewicz đã chứng minh rằng biểu thức hậu tố không cần phải có dấu ngoặc vẫn có thể tính được một cách đúng đắn bằng cách đọc lần lượt biểu thức từ trái qua phải và dùng một Stack để lưu các kết quả trung gian:

- Bước 1: Khởi tạo một ngăn xếp rỗng
- Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:
  - Nếu phần tử này là một toán hạng thì đẩy giá trị của nó vào ngăn xếp.
  - Nếu phần tử này là một toán tử ♦, ta lấy từ ngăn xếp ra hai giá trị (y và x) sau đó áp dụng toán tử ♦ đó vào hai giá trị vừa lấy ra, đẩy kết quả tìm được (x♦y) vào ngăn xếp (ra hai vào một).
- Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong ngăn xếp chỉ còn duy nhất một phần tử, phần tử đó chính là giá trị của biểu thức.

Ví dụ: Tính biểu thức $6\ 2\ /\ 4\ +\ 8\ 3\ -\ \times$ tương ứng với biểu thức trung tố $(6/2+4)\times(8-3)$

Đọc	Xử lý	Ngăn xếp
6	Đẩy vào 6	6
2	Đẩy vào 2	6, 2
/	Lấy ra 2 và 6, đẩy vào $6/2=3$	3
4	Đẩy vào 4	3, 4
+	Lấy ra 4 và 3, đẩy vào $3+4=7$	7
8	Đẩy vào 8	7, 8
3	Đẩy vào 3	7, 8, 3
-	Lấy ra 3 và 8, đẩy vào $8-3=5$	7, 5
×	Lấy ra 5 và 7, đẩy vào $7\times5=35$	35

Ta được kết quả là 35

Dưới đây ta sẽ viết một chương trình đơn giản tính giá trị biểu thức RPN.

Input

Biểu thức số học RPN, hai toán hạng liền nhau được phân tách bởi dấu cách. Các toán hạng là số thực, các toán tử là +, -, * hoặc /.

Output

Kết quả biểu thức

Sample Input	Sample Output
6 2 / 4 + 8 3 - *	35.0000

Để đơn giản, chương trình không kiểm tra lỗi viết sai biểu thức RPN, việc đó chỉ là thao tác tỉ mỉ chứ không phức tạp lắm, chỉ cần xem lại thuật toán và cài thêm các lệnh bắt lỗi tại mỗi bước.

Tính giá trị biểu thức RPN

```
{$MODE OBJFPC}
{$INLINE ON}
program CalculatingRPNExpression;
type
  TStackNode = record // Ngăn xếp được cài đặt bằng danh sách móc nối kiểu LIFO
    Value: Real;
    Link: Pointer;
  end;
  PStackNode = ^TStackNode;
var
  RPN: AnsiString;
  Top: PStackNode;

procedure Push(const v: Real); inline; //Đẩy một toán hạng là số thực v vào ngăn xếp
var
  p: PStackNode;
begin
  New(p);
  p^.Value := v;
  p^.Link := Top;
  Top := p;
end;

function Pop: Real; inline; //Lấy một toán hạng ra khỏi ngăn xếp
var
  p: PStackNode;
begin
  Result := Top^.Value;
  p := Top^.Link;
  Dispose(Top);
  Top := p;
end;

procedure ProcessToken(const Token: AnsiString); //Xử lý một phần tử trong biểu thức RPN
var
  x, y: Real;
  Err: LongInt;
begin
  if Token[1] in ['+', '-', '*', '/'] then //Nếu phần tử Token là toán tử
    begin //Lấy ra hai phần tử khỏi ngăn xếp, thực hiện toán tử và đẩy giá trị vào ngăn xếp
      y := Pop; x := Pop;
      case Token[1] of
        '+': Push(x + y);
        '-': Push(x - y);
        '*': Push(x * y);
```

```
        '/': Push(x / y);
      end;
    end
  else //Nếu phần tử Token là toán hạng thì đẩy giá trị của nó vào ngăn xếp
    begin
      Val(Token, x, Err);
      Push(x);
    end;
end;

procedure Parsing; //Xử lý biểu thức RPN
var
  i, j: LongInt;
begin
  j := 0; //j là vị trí đã xử lý xong
  for i := 1 to Length(RPN) do //Quét biểu thức từ trái sang phải
    if RPN [i] in [' ', '+', '-', '*', '/'] then //Nếu gặp toán tử hoặc dấu phân cách toán hạng
      begin
        if i > j + 1 then //Trước vị trí i có một toán hạng chưa xử lý
          ProcessToken(Copy(RPN, j + 1, i - j - 1)); //Xử lý toán hạng đó
        if RPN [i] in ['+', '-', '*', '/'] then //Nếu vị trí i chứa toán tử
          ProcessToken(RPN [i]); //Xử lý toán tử đó
        j := i; //Đã xử lý xong đến vị trí i
      end;
  if j < Length(RPN) then //Trường hợp có một toán hạng còn sót lại (biểu thức chỉ có 1 toán hạng)
    ProcessToken(Copy(RPN, j + 1, Length(RPN) - j)); //Xử lý nốt
end;

begin
  ReadLn(RPN); //Đọc biểu thức
  Top := nil; //Khởi tạo ngăn xếp rỗng,
  Parsing; //Xử lý
  Write(Pop:0:4); //Lấy ra phần tử duy nhất còn lại trong ngăn xếp và in ra kết quả.
end.
```

8.5. Chuyển từ dạng trung tố sang hậu tố

Có thể nói rằng việc tính toán biểu thức viết bằng ký pháp nghịch đảo Balan là khoa học hơn, máy móc và đơn giản hơn việc tính toán biểu thức viết bằng ký pháp trung tố. Chỉ riêng việc không phải xử lý dấu ngoặc đã cho ta thấy ưu điểm của ký pháp RPN. Chính vì lý do này, các chương trình d ịch vẫn cho phép lập trình viên viết biểu thức trên ký pháp trung tố theo thói quen, nhưng trước khi dịch ra các lệnh máy thì tất cả các biểu thức đều được chuyển về dạng RPN. Vấn đề đặt ra là phải có một thuật toán chuyển biểu thức dưới dạng trung tố về dạng RPN một cách hiệu quả, dưới đây ta trình bày thuật toán đó:

Thuật toán sử dụng một ngăn xếp $Stack$ để chứa các toán tử và dấu ngoặc mở. Thủ tục $Push(v)$ để đẩy một phần tử vào $Stack$, hàm Pop để lấy ra một phần tử từ $Stack$, hàm Get để đọc giá trị phần tử nằm ở đỉnh $Stack$ mà không lấy phần tử đó ra. Ngoài ra mức độ ưu tiên của các toán tử được quy định bằng hàm $Priority$: Ưu tiên cao nhất là dấu

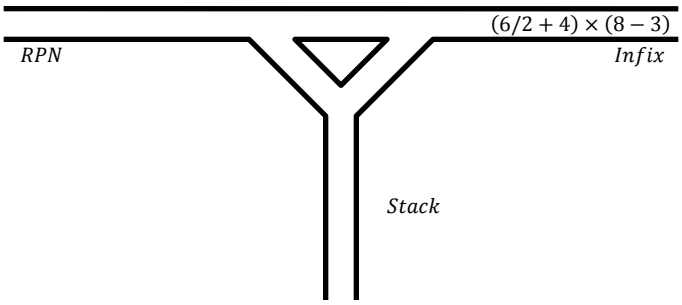
nhân (*) và dấu chia (/) với mức ưu tiên là 2, tiếp theo là dấu cộng (+) dấu (-) với mức ưu tiên là 1, ưu tiên thấp nhất là dấu ngoặc mở với mức ưu tiên là 0.

```
Stack := Ø;
for «phần tử token đọc được từ biểu thức trung tố» do
  case token of //token có thể là toán hạng, toán tử, hoặc dấu ngoặc được đọc lần lượt theo thứ tự từ trái qua phải
    '(': Push(token); //Gặp dấu ngoặc mở thì đẩy vào ngăn xếp
    ')': //Gặp dấu ngoặc đóng thì lấy ra và hiển thị các phần tử trong ngăn xếp cho tới khi lấy tới dấu ngoặc mở
       repeat
         x := Pop;
         if x ≠ '(' then Output← x;
       until x = '(';
    '+', '-', '*', '/': //Gặp toán tử
      begin
        //Chừng nào đỉnh ngăn xếp có phần tử với mức ưu tiên lớn hơn hay bằng token, lấy phần tử đó ra và hiển thị
        while (Stack ≠ Ø) and (Priority(token) ≤ Priority(Get)) do
          Output ← Pop;
        Push(token); //Đẩy toán tử token vào ngăn xếp
      end;
    else //Gặp toán hạng thì hiển thị luôn
      Output ← token;
  end;
//Khi đọc xong biểu thức, lấy ra và hiển thị tất cả các phần tử còn lại trong ngăn xếp
while Stack ≠ Ø do
  Output ← Pop
```

Ví dụ với biểu thức trung tố $(6/2 + 4) \times (8 - 3)$

Đọc	Xử lý	Ngăn xếp	Output	Chú thích
(	Đẩy "(" vào ngăn xếp	(		
6	Hiển thị	(	6	
/	Đẩy "/" vào ngăn xếp	(/	6	"/" > "("
2	Hiển thị	(/	6 2	
+	Lấy "/" khỏi ngăn xếp và hiển thị, đẩy "+" vào ngăn xếp	(+	6 2 /	"(" < "+" < "/"
4	Hiển thị	(+	6 2 / 4	
)	Lấy "+" và "(" khỏi ngăn xếp, hiển thị "+"	Ø	6 2 / 4 +	
×	Đẩy "×" vào ngăn xếp	×	6 2 / 4 +	
(	Đẩy "(" vào ngăn xếp	×(	6 2 / 4 +	
8	Hiển thị	×(	6 2 / 4 + 8	
-	Đẩy "-" vào ngăn xếp	×(-	6 2 / 4 + 8	"-" > ")"
3	Hiển thị	×(-	6 2 / 4 + 8 3	
)	Lấy "-" và "(" khỏi ngăn xếp, hiển thị "-"	×	6 2 / 4 + 8 3 -	
Hết	Lấy nốt "×" ra và hiển thị	Ø	6 2 / 4 + 8 3 - ×	

Thuật toán này có tên là thuật toán "xếp toa tàu" (*shunting yards*) do Edsger Dijkstra đề xuất năm 1960. Tên gọi này xuất phát từ mô hình đường ray tàu hỏa:



Hình 8-2. Mô hình "xếp toa tàu" của thuật toán chuyển từ dạng trung tố sang hậu tố

Trong Hình 8-2, mỗi toa tàu tương ứng với một phần tử trong biểu thức trung tố nằm ở "đường ray" $Infix$. Có ba phép chuyển toa tàu: Từ đường ray $Infix$ sang thẳng đường ray RPN, từ đường ray $Infix$ xuống đường ray $Stack$, hoặc từ đường ray $Stack$ lên đường ray RPN. Thuật toán chỉ đơn thuần dựa trên các luật chuyển mà theo các luật đó ta sẽ chuyển được tất cả các toa tàu sang đường ray RPN để được một thứ tự tương ứng với biểu thức hậu tố* (ví dụ toán hạng ở $Infix$ sẽ được chuyển thẳng sang RPN hay dấu "(" ở $Infix$ sẽ được chuyển thẳng xuống $Stack$).

Dưới đây là chương trình chuyển biểu thức viết ở dạng trung tố sang dạng RPN:

Input

Biểu thức trung tố

Output

Biểu thức hậu tố

Sample Input	Sample Output
(6 / 2 + 4) * (8 - 3)	6 2 / 4 + 8 3 - *

Chuyển từ dạng trung tố sang hậu tố

```
{$MODE OBJFPC}
{$INLINE ON}
program ConvertInfixToRPN;
type
  TStackNode = record // Ngăn xếp được cài đặt bằng danh sách móc nối kiểu LIFO
    Value: AnsiChar;
    Link: Pointer;
  end;
```

* Thực ra thì còn thao tác loại bỏ các dấu ngoặc trong biểu thức RPN nữa, nhưng điều này không quan trọng, dấu ngoặc là thừa trong biểu thức RPN vì như đã nói về phương pháp tính: biểu thức RPN có thể tính đơn định mà không cần các dấu ngoặc.

```
  PStackNode = ^TStackNode;
var
  Infix: AnsiString;
  Top: PStackNode;

procedure Push(c: AnsiChar); inline; //Đẩy một phần tử v vào ngăn xếp
var
  p: PStackNode;
begin
  New(p);
  p^.Value := c;
  p^.Link := Top;
  Top := p;
end;

function Pop: AnsiChar; inline; //Lấy một phần tử ra khỏi ngăn xếp
var
  p: PStackNode;
begin
  Result := Top^.Value;
  p := Top^.Link;
  Dispose(Top);
  Top := p;
end;

function Get: AnsiChar; inline; //Đọc phần tử ở đỉnh ngăn xếp
begin
  Result := Top^.Value;
end;

function Priority(c: Char): LongInt; inline; //Mức ưu tiên của các toán tử
begin
  case c of
    '*', '/': Result := 2;
    '+', '-': Result := 1;
    '(': Result := 0;
  end;
end;

//Xử lý một phần tử đọc được từ biểu thức trung tố
procedure ProcessToken(const Token: AnsiString);
var
  x: AnsiChar;
  Opt: AnsiChar;
begin
  Opt := Token[1];
  case Opt of
    '(': Push(Opt); //Token là dấu ngoặc mở
    ')': //Token là dấu ngoặc đóng
      repeat
        x := Pop;
        if x <> '(' then Write(x, ' ')
        else Break;
      until False;
    '+', '-', '*', '/': //Token là toán tử
      begin
        while (Top <> nil) and (Priority(Opt) <= Priority(Get)) do
```

```
          Write(Pop, ' ');
        Push(Opt);
      end;
    else //Token là toán hạng
      Write(Token, ' ');
  end;
end;

procedure Parsing;
const
  Operators = ['(', ')', '+', '-', '*', '/'];
var
  i, j: LongInt;
begin
  j := 0; //j là vị trí đã xử lý xong
  for i := 1 to Length(Infix) do
    if Infix[i] in Operators + [' '] then //Nếu gặp dấu ngoặc, toán tử hoặc dấu cách
      begin
        if i > j + 1 then //Trước vị trí i có một toán hạng chưa xử lý
          ProcessToken(Copy(Infix, j + 1, i - j - 1)); //xử lý toán hạng đó
        if Infix[i] in Operators then //Nếu vị trí i chứa toán tử hoặc dấu ngoặc
          ProcessToken(Infix[i]); //Xử lý ký tự đó
        j := i; //Cập nhật, đã xử lý xong đến vị trí i
      end;
  if j < Length(Infix) then //Xử lý nốt toán hạng còn sót lại
    ProcessToken(Copy(Infix, j + 1, Length(Infix) - j));
  //Đọc hết biểu thức trung tố, lấy nốt các phần tử trong ngăn xếp ra và hiển thị
  while Top <> nil do
    Write(Pop, ' ');
  WriteLn;
end;

begin
  ReadLn(Infix); //Nhập dữ liệu
  Top := nil; //Khởi tạo ngăn xếp rỗng
  Parsing; //Đọc biểu thức trung tố và chuyển thành dạng RPN
end.
```

8.6. Xây dựng cây nhị phân biểu diễn biểu thức

Ngay trong phần đầu tiên, chúng ta đã biết rằng các dạng biểu thức trung tố, tiền tố và hậu tố đều có thể được hình thành bằng cách duyệt cây nhị phân biểu diễn biểu thức đó theo các trật tự khác nhau. Vậy tại sao không xây dựng ngay cây nhị phân biểu diễn biểu thức đó rồi thực hiện các công việc tính toán ngay trên cây?. Khó khăn gặp phải chính là thuật toán xây dựng cây nhị phân trực tiếp từ dạng trung tố có thể kém hiệu quả, trong khi đó từ dạng hậu tố lại có thể khôi phục lại cây nhị phân biểu diễn biểu thức một cách rất đơn giản, gần giống như quá trình tính toán biểu thức hậu tố:

- Bước 1: Khởi tạo một ngăn xếp rỗng dùng để chứa các nút trên cây
- Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:

- Tạo ra một nút mới z chứa phần tử mới đọc được
- Nếu phần tử này là một toán tử, lấy từ ngăn xếp ra hai nút (theo thứ tự là y và x), cho x trở thành con trái và y trở thành con phải của nút z
- Đẩy nút z vào ngăn xếp

- Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong ngăn xếp chỉ còn duy nhất một phần tử, phần tử đó chính là gốc của cây nhị phân biểu diễn biểu thức.

Bài tập 8-1.

Biểu thức có thể có dạng phức tạp hơn, chẳng hạn biểu thức bao gồm cả phép lấy số đối ($-x$), phép tính lũy thừa (x^y), hàm số với một hay nhiều biến số. Chúng ta có thể biểu diễn những biểu thức dạng này bằng một cây tổng quát và từ đó có thể chuyển biểu thức về dạng RPN để thực hiện tính toán. Hãy xây dựng thuật toán để chuyển biểu thức số học (dạng phức tạp) về dạng RPN và thuật toán tính giá trị biểu thức đó.

Bài tập 8-2.

Viết chương trình chuyển biểu thức logic dạng trung tố sang dạng RPN. Ví dụ chuyển: a and b or c and d thành: $a\ b$ and $c\ d$ and or

Bài tập 8-3.

Chuyển các biểu thức sau đây ra dạng RPN

a) $A \times (B + C)$
b) $A + (B/C) + D$
c) $A \times (B + -C)$
d) $A - (B + C)^{D/E}$
e) $(A \text{ or } B) \text{ and } \big(C \text{ or } (D \text{ or not } E)\big)$
f) $(A = B) \text{ or } (C = D)$

Bài tập 8-4.

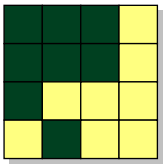
Với một ảnh đen trắng hình vuông kích thước $2^n \times 2^n$, người ta dùng phương pháp sau để mã hóa ảnh:

- Nếu ảnh chỉ gồm toàn điểm đen thì ảnh đó có thể được mã hóa bằng xâu chỉ gồm một ký tự 'B'
- Nếu ảnh chỉ gồm toàn điểm trắng thì ảnh đó có thể được mã hóa bằng xâu chỉ gồm một ký tự 'W'

- Nếu P, Q, R, S lần lượt là xâu mã hóa của bốn ảnh vuông kích thước bằng nhau thì $\&PQRS$ là xâu mã hóa của ảnh vuông tạo thành bằng cách đặt 4 ảnh vuông ban đầu theo sơ đồ:

$$\begin{matrix} P & Q \\ S & R \end{matrix}$$

Ví dụ "&B&BWWBW&BWBW" và "&&BBBB&BWWBW&BWBW" là hai xâu mã hóa của cùng một ảnh dưới đây:



Bài toán đặt ra là cho số nguyên dương n và hai xâu mã hóa của hai ảnh kích thước $2^n \times 2^n$. Hãy cho biết hai ảnh đó có khác nhau không và nếu chúng khác nhau hãy chỉ ra một vị trí có màu khác nhau trên hai ảnh.

Chương III. SẮP XẾP VÀ THỨ TỰ THỐNG KÊ

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng Tin học nhằm nhiều mục đích khác nhau, nhưng thường là hai mục đích chính: xây dựng một danh sách có thứ tự các đối tượng (*ordering)* hoặc phân loại các đối tượng (*categorizing*) dựa trên một số thuộc tính nào đó của các đối tượng.

Bài toán sắp xếp là một trong những bài toán cơ bản nhất của khoa học máy tính. Hiếm có bài toán nào lại có nhiều thuật toán để giải quyết như bài toán sắp xếp. Học một thuật toán sắp xếp không chỉ là hiểu và cài đặt được thuật toán, mà quan trọng hơn là học cách tiếp cận vấn đề, cách tổ chức dữ liệu, cách đánh giá giải thuật và cách cài đặt chương trình.

Bài 9. Bài toán sắp xếp

Hãy tưởng tượng nếu bạn cần tra từ điển nhưng các từ mục trong cuốn từ điển của bạn bị đặt tùy tiện không theo một trật tự nào cả thì bạn sẽ mất bao nhiêu thời gian để tra một từ. Nếu bạn chỉ có một cuốn từ điển Việt-Anh và bạn cố dùng nó để tra nghĩa một từ tiếng Anh mà bạn chưa biết, bạn sẽ thấy ngay rằng việc đi ra hiệu sách mua một cuốn từ điển Anh-Việt sẽ là một giải pháp kinh tế hơn. Ví dụ này có vẻ hiếm gặp trên thực tế, nhưng giả sử rằng bạn là người hoàn toàn không biết tiếng Trung và được hỏi từ "排序" nghĩa là gì thì dù bạn có từ điển tiếng Trung-Việt trong tay đi nữa cũng sẽ chẳng có ý nghĩa gì cả, bởi vì bạn không biết thứ tự các từ mục tiếng Trung được sắp xếp trong từ điển như thế nào.

Hãy thử nghĩ xem nếu không có những nhân viên thư viện hàng ngày sắp đặt các cuốn sách vào đúng kệ sách của nó thì bạn sẽ mất bao nhiêu thời gian để tìm một cuốn sách trong thư viện khi chỉ có tên sách hay mã số sách.

Yêu cầu về sắp xếp xuất hiện hàng ngày trong đời sống. Có thể coi sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Chẳng hạn như thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với các từ v.v… Có hai mục đích chính của việc sắp xếp các đối tượng: xây dựng một danh sách có thứ tự các đối tượng (*ordering)* hoặc phân loại các đối tượng (*categorizing*) dựa trên một hoặc vài tiêu chí nào đó.

Trong các ứng dụng tin học, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước: Một tập các đối tượng cần sắp xếp là tập *các bản ghi* (*records*), mỗi bản ghi bao gồm một số *trường* (*fields*) khác nhau. Nhưng không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ là một trường nào đó (hay một vài trường nào đó) được chú ý tới thôi. Trường như vậy ta gọi là *khóa* (*key*). Sắp xếp sẽ được tiến hành dựa vào giá trị của khóa này.

Ví dụ hồ sơ tuyển sinh của một trường Đại học là một danh sách thí sinh, mỗi thí sinh có số báo danh, họ tên, điểm thi:

STT	SBD	Họ và tên	Điểm thi
1	A100	Nguyễn Văn A	20
2	B200	Trần Thị B	25
3	X150	Phạm Văn C	18
4	G180	Đỗ Thị D	21

Khi muốn liệt kê danh sách những thí sinh trúng tuyển tức là phải sắp xếp các thí sinh theo thứ tự từ điểm cao nhất tới điểm thấp nhất. Ở đây khóa sắp xếp chính là điểm thi. Khi sắp xếp, các bản ghi trong bảng sẽ được đặt lại vào các vị trí sao cho giá trị khóa tương ứng với chúng có đúng thứ tự đã ấn định. Nói cách khác: *Việc so sánh hai bản ghi thì chỉ dựa trên giá trị khóa nhưng việc hoán chuyển vị trí thì phải thực hiện trên toàn bản ghi*.

Vì kích thước của toàn bản ghi có thể rất lớn, nên nếu việc sắp xếp thực hiện trực tiếp trên các bản ghi sẽ đòi hỏi sự chuyển đổi vị trí của các bản ghi, kéo theo việc thường xuyên phải di chuyển, copy những vùng nhớ lớn, gây ra những tổn phí thời gian khá nhiều. Thường người ta khắc phục tình trạng này bằng cách xây dựng một *bảng khóa* gồm các con trỏ tới các bản ghi trong bảng chính. Mỗi con trỏ trong bảng khóa sẽ tương ứng với một bản ghi trong bảng chính, tức là mỗi con trỏ chỉ chứa một lượng thông tin đủ để biết bản ghi tương ứng với nó là bản ghi nào. Sau đó, *việc sắp xếp được thực hiện trực tiếp trên các con trỏ của bảng khóa, bảng chính không hề bị ảnh hưởng gì*, việc truy cập vào một bản ghi nào đó của bảng chính được thực hiện được gián tiếp qua con trỏ của bảng khóa.

Như ở ví dụ trên, ta có thể coi số thứ tự của mỗi bản ghi trong bảng chính là con trỏ. Sau khi sắp xếp bảng khóa theo thứ tự giảm dần của điểm thi trong bản ghi tương ứng, ta sẽ được thứ tự 2,4,1,3. Từ đó ta có thể biết được rằng người có điểm cao nhất là người mang số thứ tự 2, tiếp theo là người mang số thứ tự 4, tiếp nữa là người mang số thứ tự 1, và cuối cùng là người mang số thứ tự 3, còn muốn liệt kê danh sách đầy đủ thì ta chỉ việc đối chiếu với bảng ban đầu và liệt kê theo đúng thứ tự 2,4,1,3.

Có nhiều cách dựng bảng khóa, chẳng hạn như sử dụng kiểu dữ liệu con trỏ trong ngôn ngữ lập trình và gán mỗi con trỏ tương ứng với địa chỉ của một bản ghi cần sắp xếp. Trong trường hợp số thứ tự của bản ghi trong bảng chính được sử dụng làm con trỏ như ở ví dụ trên, thì người ta gọi đây là kỹ thuật *sắp xếp bằng chỉ số*: Bảng ban đầu không hề bị ảnh hưởng gì cả, việc sắp xếp chỉ đơn thuần là đánh lại chỉ số cho các bản ghi theo thứ tự sắp xếp. Cụ thể là nếu $r_1, r_2, \dots, r_n$ là các bản ghi cần sắp xếp theo một thứ tự nhất định thì việc sắp xếp bằng chỉ số tức là xây dựng một dãy $i_1, i_2, \dots, i_n$ sao cho $r_{i_1}, r_{i_2}, \dots, r_{i_n}$ chính là thứ tự cần sắp xếp (Bản ghi r_{i_j} sẽ phải đứng sau $j-1$ bản ghi khác khi sắp thứ tự)

Do khóa có vai trò đặc biệt như vậy nên khi trình bày các thuật toán sắp xếp, ta sẽ coi *khóa như đại diện cho các bản ghi* và để cho đơn giản, ta chỉ nói tới giá trị của khóa mà thôi. Các thao tác trong kỹ thuật sắp xếp lẽ ra là tác động lên toàn bản ghi giờ đây chỉ làm trên khóa. Còn việc cài đặt các phương pháp sắp xếp trên danh sách các bản ghi và kỹ thuật sắp xếp bằng chỉ số, ta coi như bài tập.

Bài toán sắp xếp giờ đây có thể phát biểu như sau:

Xét quan hệ thứ tự toàn phần "nhỏ hơn hoặc bằng" ký hiệu "≤" trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

- Tính phổ biến: Với $\forall a, b \in S$, hoặc là $a \leq b$, hoặc $b \leq a$, nghĩa là hai phần tử bất kỳ trong tập hợp là so sánh được với nhau.
- Tính phản xạ: Với $\forall a \in S$: $a \leq a$
- Tính phản đối xứng: Với $\forall a, b \in S$, nếu $a \leq b$ và $b \leq a$ thì bắt buộc $a = b$.
- Tính bắc cầu: Với $\forall a, b, c \in S$, nếu có $a \leq b$ và $b \leq c$ thì $a \leq c$

Trong trường hợp $a \leq b$ và $a \neq b$, ta dùng ký hiệu "<" cho gọn

Cho một dãy $K = (k_1, k_2, \ldots, k_n)$ gồm n phần tử của S gọi là *khóa*. Giữa hai khóa bất kỳ có quan hệ thứ tự toàn phần "≤". Bài toán sắp xếp yêu cầu hoán vị các phần tử của dãy K để:

$$k_1 \leq k_2 \leq \cdots \leq k_n$$

Giả sử cấu trúc dữ liệu cho dãy khóa được mô tả như sau:

```
const
  MaxLength = ...; //Tùy chọn số phần tử thực sự của mảng
type
  TKey = ...; //Kiểu dữ liệu một khóa
  TKeyArray = array[1..MaxLength] of TKey;
var
  k: TKeyArray; //Dãy khóa
  n: Integer; //Số phần tử thực sự của mảng cần sắp xếp
```

Thì những thuật toán sắp xếp dưới đây được viết dưới dạng thủ tục sắp xếp dãy khóa $k_{1\ldots n}$ với kiểu chỉ số đánh cho từng khóa trong dãy là số nguyên Integer.

Trong nhiều thuật toán, phép *đảo giá trị hai khóa* được sử dụng khá thường xuyên, vì vậy chúng ta sẽ viết một thủ tục $Swap(var\ x, y: TKey)$ để đảo giá trị hai tham biến x, y qua một biến trung gian $temp$. Trong cài đặt cụ thể, thủ tục $Swap$ có thể được viết tối ưu hơn bằng hợp ngữ để tăng tốc độ của chương trình.

```
procedure Swap(var x, y: TKey);
var
  temp: TKey;
begin
  temp := x; x := y; y := temp;
end;
```

Bài 10. Các thuật toán sắp xếp tổng quát

Bài này sẽ giới thiệu một số thuật toán *sắp xếp tổng quát* (*general-purpose sorting*). Việc cài đặt các thuật toán này chỉ cần có thông tin về quan hệ thứ tự toàn phần ($\leq$) trên dãy khóa là đủ. Tất cả các mô hình sắp xếp trong bài đều là sắp xếp tăng dần ($k_i \leq k_{i+1}, \forall i$) nhưng bạn có thể sử dụng các thuật toán này để sắp xếp dãy khóa theo bất kỳ quan hệ thứ tự toàn phần nào bằng cách định nghĩa lại quan hệ "nhỏ hơn hay bằng" giữa các khóa.

Chính vì các thuật toán chỉ quan tâm tới quan hệ $\leq$ nên có thể gọi chúng là các thuật toán sắp xếp chỉ dựa trên so sánh (*comparison sorts*).

10.1. Thuật toán sắp xếp kiểu chọn (Selection Sort)

Một trong những thuật toán sắp xếp đơn giản nhất là phương pháp sắp xếp kiểu chọn. Ý tưởng cơ bản của cách sắp xếp này là:

Ở lượt thứ nhất, ta chọn trong dãy khóa $k_{1\ldots n}$ ra khóa nhỏ nhất (khóa $\leq$ mọi khóa khác) và đảo giá trị của nó với k_1, khi đó giá trị khóa k_1 trở thành giá trị khóa nhỏ nhất.

Ở lượt thứ hai, ta chọn trong dãy khóa $k_{2\ldots n}$ ra khóa nhỏ nhất và đảo giá trị của nó với k_2.

…

Ở lượt thứ i, ta chọn trong dãy khóa $k_{i\ldots n}$ ra khóa nhỏ nhất và đảo giá trị của nó với k_i.

…

Làm tới lượt thứ $n-1$, chọn trong dãy hai khóa (k_{n-1}, k_n) ra khóa nhỏ nhất và đảo giá trị của nó với k_{n-1}.

```
procedure SelectionSort;
var
  i, j, jmin: Integer;
begin
  for i := 1 to n - 1 do //Làm n - 1 lượt
    begin
      //Chọn trong số các khóa trong đoạn k[i…n] ra khóa k[jmin] nhỏ nhất
      jmin := i;
      for j := i + 1 to n do
        if k[j] < k[jmin] then jmin := j;
      if jmin ≠ i then //Đưa khóa nhỏ nhất đó về vị trí i
        Swap(k[jmin], k[i]);
    end;
end;
```

Đối với phương pháp kiểu lựa chọn, có thể coi phép so sánh $k_j < k_{jmin}$ là phép toán tích cực để đánh giá hiệu suất thuật toán về mặt thời gian (tương đương Θ với số lần lặp của vòng lặp for j…). Ở lượt thứ i, để chọn ra khóa nhỏ nhất bao giờ cũng cần $n-i$ phép

toán tích cực, số lượng này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khóa cả. Từ đó suy ra tổng số phép so sánh sẽ phải thực hiện là:

$$(n-1)+(n-2)+\cdots+1=\frac{n*(n-1)}{2}$$

Vậy thời gian thực hiện thuật toán sắp xếp kiểu chọn là $\Theta(n^2)$ không phụ thuộc dữ liệu đầu vào.

Thuật toán sắp xếp kiểu chọn có thể cài đặt được trên danh sách nối đơn.

10.2. Thuật toán sắp xếp nổi bọt (Bubble Sort)

Trong thuật toán sắp xếp nổi bọt, dãy các khóa sẽ được duyệt từ cuối dãy lên đầu dãy (từ k_n về k_1), nếu gặp hai khóa kế cận bị ngược thứ tự thì đổi chỗ của chúng cho nhau. Sau lần duyệt như vậy, khóa nhỏ nhất trong dãy khóa sẽ được chuyển về vị trí đầu tiên và vấn đề lặp lại với việc sắp xếp dãy khóa $k_{2\ldots n}$:

```
procedure BubbleSort;
var
  i, j: Integer;
begin
  for i := 2 to n do
    for j := n downto i do //Duyệt từ cuối dãy lên, làm nổi khóa nhỏ nhất trong đoạn k[i-1...n] về vị trí i-1
      if k[j] < k[j - 1] then
        Swap(k[j - 1], k[j]);
end;
```

Đối với thuật toán sắp xếp nổi bọt, có thể coi phép toán tích cực là phép so sánh $k_j < k_{j-1}$ (Tương đương Θ với số lần lặp của vòng lặp for j…). Số lần thực hiện phép so sánh này là:

$$(n-1)+(n-2)+\cdots+1=\frac{n*(n-1)}{2}$$

Vậy thuật toán sắp xếp nổi bọt cũng có độ phức tạp là $\Theta(n^2)$. Bất kể tình trạng dữ liệu vào như thế nào.

Thuật toán sắp xếp nổi bọt cũng có thể cài đặt được trên danh sách nối đơn.

Một vài cải tiến:

Gọi mỗi vị trí j mà $k_j < k_{j-1}$ là một *nghịch thế* thì nếu vòng lặp

```
for j := n downto i do…
```

không tìm thấy nghịch thế nào cả, chúng ta có thể cho thuật toán dừng ngay vì một dãy khóa không còn nghịch thế chính là dãy khóa đã sắp. Ý tưởng này có thể mở rộng thêm: Trong vòng lặp for đối với j, ta ghi nhận lại t là vị trí nghịch thế cuối cùng tìm thấy (vị trí thực hiện phép *wap* cuối cùng), khi đó có thể nhận thấy rằng các phần tử từ k_1 tới k_{t-1}

đã nằm ở vị trí đúng nên thay vì tăng cận dưới i lên 1, ta sẽ tăng i lên bằng t và tiếp tục công việc sắp xếp dãy khóa $k_{i\ldots n}$…

Với cải tiến này, trường hợp tốt nhất là dãy khóa ban đầu đã được sắp xếp sẵn, khi đó thời gian thực hiện giải thuật trong trường hợp tốt nhất chỉ là $\Theta(n)$. Tuy vậy cần phải nhấn mạnh rằng ưu điểm của Bubbe Sort không nằm ở khía cạnh tốc độ mà nằm ở tính đơn giản. Phép cải tiến này không làm tăng tốc độ thực thi trung bình của Bubble Sort lên bao nhiêu nhưng lại làm Bubble Sort chậm hơn trong trường hợp xấu nhất (dãy khóa được sắp theo thứ tự ngược lại) và hy sinh ưu điểm chính của Bubble Sort.

Có một thuật toán khác sửa đổi thuật toán sắp xếp nổi bọt và có tốc độ cao hơn một chút trên thực tế là thuật toán Cocktail Sort. Thuật toán này còn có một số tên gọi khác như Bidirectional BubbleSort, Shuttle Sort… thuật toán gồm hai pha:

- Nhẹ nổi lên: Với dãy khóa $k_{1\ldots n}$, đầu tiên ta duyệt từ cuối dãy khóa lên và đổi chỗ từng cặp khóa liên tiếp gặp phải nếu thấy khóa đứng trước lớn hơn khóa đứng sau, kết thúc quá trình này k_1 là khóa nhỏ nhất và vấn đề còn lại là sắp xếp dãy khóa $k_{2\ldots n}$.
- Nặng chìm xuống: Tiếp theo với dãy khóa $k_{2\ldots n}$, ta duyệt từ đầu dãy khóa xuống và đổi chỗ từng cặp khóa liên tiếp gặp phải nếu thấy khóa đứng trước lớn hơn khóa đứng sau, kết thúc quá trình này k_n là khóa lớn nhất và vấn đề còn lại là sắp xếp dãy khóa $k_{2\ldots n-1}$.

Hai pha này được thực hiện luân phiên cho tới khi đoạn các phần tử cần sắp xếp còn ít hơn 2 phần tử.

Có thể cài đặt thuật toán Cocktail Sort sao cho sau mỗi pha thì đoạn cần sắp xếp có thể co lại nhiều hơn một phần tử bằng cách ghi nhận vị trí thực hiện phép $Swap$ cuối cùng tại mỗi pha và sử dụng vị trí đó để làm vị trí đầu (hay vị trí cuối) cho đoạn cần sắp xếp ở pha tiếp theo tùy theo pha tiếp theo là "nặng chìm xuống" hay "nhẹ nổi lên".

```
procedure CocktailSort;
var
  L, H, Bound, i: Integer;
begin
  L := 1; H := n;
  repeat
    Bound := H;
    for i := H downto L + 1 do //Nhẹ nổi lên
      if k[i] < k[i - 1] then
        begin
          Swap(k[i - 1], k[i]);
          Bound := i;
        end;
    L := Bound; //Cập nhật vị trí đầu đoạn mới
    for i := L to H - 1 do //Nặng chìm xuống
      if k[i] > k[i + 1] then
        begin
          Swap(k[i], k[i + 1]);
          Bound := i;
        end;
    H := Bound; //Cập nhật vị trí cuối đoạn mới
  until L ≥ H;
end;
```

Khi cài đặt thực tế, Cocktail Sort có thời gian thực hiện tốt hơn một chút so với Bubble Sort. Cocktail Sort không cài đặt được trên danh sách nối đơn nhưng có thể cài đặt được trên dánh sách nối kép.

10.3. Thuật toán sắp xếp kiểu chèn (Insertion Sort)

Xét dãy khóa $k_{1\ldots n}$. Ta thấy dãy con chỉ gồm mỗi một khóa là k_1 có thể coi là đã sắp xếp rồi. Với $2 \leq i \leq n$, giả sử $i-1$ phần tử đầu của dãy khóa đã được sắp xếp:

$$k_1 \leq k_2 \leq \cdots \leq k_{i-1}$$

khi đó, ta có chèn phần tử k_i vào một vị trí hợp lý trong đoạn đầu gồm $i-1$ phần tử đã được sắp xếp của dãy để được một dãy mới mà i phần tử đầu của dãy khóa đã đư ợc sắp xếp. Thuật toán sắp xếp kiểu chèn thực hiện quy trình này lần lượt với i từ 2 tới n và cho một dãy khóa đã sắp xếp sau khi kết thúc.

```
procedure InsertionSort;
var
  i, j: Integer;
  temp: TKey;
begin
  for i := 2 to n do
    begin //Chèn giá trị k[i] vào dãy k[1...i-1] để toàn đoạn k[1...i] trở thành đã sắp xếp
      temp := k[i]; //Lưu giữ lại giá trị k[i]
      j := i - 1;
      while (j > 0) and (temp < k[j]) do //So sánh giá trị cần chèn với lần lượt các khóa k[j] đứng trước
        begin //Nếu khóa k[j] lớn hơn
          k[j + 1] := k[j]; //Đẩy lùi giá trị k[j] về phía sau một vị trí, tạo ra chỗ trống tại vị trí j
          j := j - 1; //Xét tiếp phần tử liền trước
        end;
      k[j + 1] := temp; //Đưa giá trị chèn vào "chỗ trống" mới tạo ra
    end;
end;
```

Đối với thuật toán sắp xếp kiểu chèn, thì chi phí thời gian thực hiện thuật toán phụ thuộc vào tình trạng dãy khóa ban đầu. Nếu coi phép toán tích cực ở đây là phép kiểm tra điều kiện:

$$(j > 0)\ and\ (temp < k_j)$$

Khi đó:

Trường hợp tốt nhất ứng với dãy khóa đã sắp xếp rồi, mỗi lượt chỉ cần 1 phép toán tích cực, và như vậy tổng số lần thực hiện phép toán tích cực là $n-1$. Phân tích trong trường hợp tốt nhất, thời gian thực hiện thuật toán sắp xếp kiểu chèn là $\Theta(n)$.

Trường hợp xấu nhất ứng với dãy khóa đã có thứ tự ngược với thứ tự cần sắp thì ở lượt thứ i, cần có i phép toán tích cực và tổng số lần thực hiện phép toán tích cực là:

$$n + (n-1) + \cdots + 2 = \frac{(n-1)(n+2)}{2}$$

Vậy trong trường hợp xấu nhất, thời gian thực hiện của thuật toán sắp xếp kiểu chèn là $\Theta(n^2)$

Trường hợp các giá trị khóa xuất hiện một cách ngẫu nhiên, giả sử tại lượt thứ i, khóa k_i phải chèn vào vị trí $i-\Delta\ (0 \le \Delta < i)$, khi đó số lần thực hiện phép toán tích cực là $\Delta + 1$. Xét với tất cả các khả năng của dữ liệu, ta có thể coi Δ là biến ngẫu nhiên phân bố đều trong tập $\{0,1,2,\dots,i-1\}$ với xác suất $1/i$. Vậy thì tại mỗi bước lặp for với một giá trị của i, số lần trung bình thực hiện phép toán tích cực là giá trị kỳ vọng:

$$\frac{1}{i}\sum_{\Delta=0}^{i-1}(\Delta+1) = \frac{i-1}{2}$$

Vậy xét trong toàn bộ thuật toán số lần trung bình thực hiện phép toán tích cực là:

$$\sum_{i=2}^{n} \frac{i-1}{2} = \frac{n(n-1)}{4}$$

Vậy phân tích trong trường hợp trung bình, thời gian thực hiện thuật toán sắp xếp kiểu chèn là $\Theta(n^2)$.

Nhìn về kết quả đánh giá, ta có thể thấy rằng thuật toán sắp xếp kiểu chèn tỏ ra tốt hơn so với thuật toán sắp xếp chọn và sắp xếp nổi bọt. Tuy nhiên, chi phí thời gian thực hiện của thuật toán sắp xếp kiểu chèn vẫn còn khá lớn.

Có thể cải tiến thuật toán sắp xếp chèn nhờ nhận xét: Khi dãy khóa $k[1 \dots i-1]$ đã được sắp xếp thì việc tìm vị trí chèn có thể làm bằng thuật toán tìm kiếm nhị phân và kỹ thuật chèn có thể làm bằng các lệnh dịch chuyển vùng nhớ cho nhanh. Tuy nhiên điều đó cũng không làm giảm đi độ phức tạp của thuật toán bởi trong trường hợp xấu nhất, ta phải mất $n-1$ lần chèn và tại mỗi bước lặp for với một giá trị của i ta phải dịch lùi i khóa để tạo ra khoảng trống trước khi đẩy giá trị khóa chèn vào chỗ trống đó.

Thuật toán sắp xếp kiểu chèn có thể cài đặt trên danh sách nối đơn, (dùng kết hợp với một thủ tục đệ quy)

10.4. Shell Sort

Nhược điểm của thuật toán sắp xếp kiểu chèn thể hiện khi mà ta luôn phải chèn một khóa vào vị trí gần đầu dãy. Để khắc phục nhược điểm này, người ta thường sử dụng thuật toán sắp xếp chèn với độ dài bước giảm dần, ý tưởng ban đầu cho thuật toán được đưa ra bởi [**16**] nên thuật toán còn có một tên gọi khác: Shell Sort

Xét dãy khóa: $k_{1\dots n}$. Với một số nguyên dương $step$: $1 \le step < n$, ta có thể chia dãy đó thành $step$ dãy con:

Dãy con 1: $(k_1, k_{1+step}, k_{1+2step}, \dots)$

Dãy con 2: $(k_2, k_{2+step}, k_{2+2step}, \dots)$

…

Dãy con $step$: $(k_{step}, k_{2step}, k_{3step}, \dots)$

Ví dụ như dãy $k = (4,6,7,2,3,5,1,9,8)$, $n = 9$; $step = 3$. Có 3 dãy con:

Chỉ số	1	2	3	4	5	6	7	8	9
Dãy khóa chính	4	6	7	2	3	5	1	9	8
Dãy con 1	4			2			1		
Dãy con 2		6			3			9	
Dãy con 3			7			5			8

Những dãy con như vậy được gọi là dãy con theo độ dài bước $step$. Tư tưởng của thuật toán Shell Sort là: Với một bước $step$, áp dụng thuật toán sắp xếp kiểu chèn từng dãy con độc lập để làm mịn dần dãy khóa chính. Rồi giảm $step$ đi và lặp lại quá trình tương tự … cho tới khi dãy khóa đã được sắp với bước $step = 1$ thì ta được dãy khóa sắp xếp.

Như ở ví dụ trên, nếu dùng thuật toán sắp xếp kiểu chèn thì khi gặp khóa $k_7 = 1$, là khóa nhỏ nhất trong dãy khóa, nó phải chèn vào vị trí 1, tức là phải đẩy lùi 6 khóa đứng trước nó. Nhưng nếu coi 1 là khóa của dãy con 1 thì nó chỉ cần chèn vào trước 2 khóa trong dãy con đó mà thôi. Đây chính là nguyên nhân Shell Sort hiệu quả hơn sắp xếp chèn: Khóa nhỏ được nhanh chóng đưa về *gần* vị trí đúng của nó. Bước cuối cùng của ShellSort chính là thuật toán sắp xếp kiểu chèn, nhưng với tình trạng của dãy khóa gần như đã được sắp xếp, và như chúng ta đã biết, thuật toán sắp xếp chèn hoạt động tốt khi mà dãy khóa gần như đã được sắp xếp rồi:

```
procedure ShellSort;
var
  i, j, step: Integer;
  temp: TKey;
begin
  step := n div 2;
  while step ≠ 0 do //Làm mịn dãy với độ dài bước step
    begin
      for i := step + 1 to n do
        begin //Sắp xếp chèn trên dãy con k[i-step], k[i], k[i+step], k[i+2step],...
          tmp := k[i]; j := i - step;
          while (j > 0) and (k[j] > temp) do
            begin
              k[j + step] := k[j];
              j := j - step;
            end;
          k[j + step] := temp;
        end;
      if step = 2 then step := 1
      else step := step * 10 div 22;
    end;
end;
```

Dễ thấy rằng để Shell Sort hoạt động đúng thì chỉ cần dãy bước $step$ sau một số bước lặp sẽ kết thúc tại $step = 1$ là được, đã có một số nghiên cứu về việc chọn các độ dài bước giảm dần cho Shell Sort nhằm tăng hiệu quả của thuật toán.

Shell Sort hoạt động nhanh và dễ cài đặt, tuy vậy việc đánh giá độ phức tạp tính toán của Shell Sort là tương đối khó, ta chỉ thừa nhận các kết quả sau đây:

Nếu các bước $Step$ được chọn theo thứ tự ngược từ dãy: $1,3,7,15, \dots, 2^{i-1}, \dots$ thì thời gian thực hiện của Shell Sort là $O(n^{3/2})$.

Nếu các bước $Step$ được chọn theo thứ tự ngược từ dãy: $1,2,3,4,6,8,9,12,16,\ldots,2^i3^j,\ldots$ (Dãy tăng dần của các phần tử dạng 2^i3^j) thì thời gian thực hiện của Shell Sort là $O(n\log^2 n)$.

Mô hình cài đặt của Shell Sort trong bài sử dụng dãy các bước giảm dần tốt nhất được biết cho tới nay: Đầu tiên sắp xếp theo độ dài bước $step := \lfloor n/2 \rfloor$ và sau mỗi lần lặp, độ dài bước sẽ giảm dần theo công thức:

$$step := \begin{cases} 1, \text{ nếu } step = 2 \\ \left\lfloor \frac{step}{2.2} \right\rfloor, \text{ nếu } step \neq 2 \end{cases}$$

Trong cài đặt chương trình cụ thể, với cấu hình máy tính phổ biến hiện nay, Shell Sort tỏ ra là thuật toán nhanh nhất trong các thuật toán sắp xếp tổng quát nếu kích thước dữ liệu vào không quá lớn. Có lẽ không cần so sánh với các thuật toán sắp xếp nói trên vì Shell Sort là sự khác biệt về độ phức tạp tính toán. Ngay cả với những thuật toán sắp xếp tốc độ cao mà chúng ta sẽ trình bày dưới đây, Shell Sort hầu như luôn nhanh hơn HeapSort trong cài đặt thực tế và chỉ chịu thua kém QuickSort khi dữ liệu vào đủ lớn ($\gtrsim 50000$ khóa)

10.5. QuickSort

Thuật toán sắp xếp kiểu phân đoạn – QuickSort [**9**] là phương pháp sắp xếp tốt nhất, theo nghĩa: dù dãy khóa thuộc kiểu dữ liệu có thứ tự nào, QuickSort cũng có thể sắp xếp được và những thuật toán sắp xếp tổng quát nhanh nhất hiện nay đều dựa trên một vài cải tiến từ QuickSort. Người đề xuất ra phương pháp đã mạnh dạn lấy chữ "Quick" để đặt tên cho thuật toán.

10.5.1. Mô hình cài đặt của QuickSort

Ý tưởng chủ đạo của phương pháp là chiến lược "chia để trị": Sắp xếp dãy khóa $k_{1\ldots n}$ thì có thể coi là sắp xếp đoạn từ chỉ số 1 tới chỉ số n trong dãy khóa đó. Để sắp xếp một đoạn trong dãy khóa, nếu đoạn đó có ít hơn 2 khóa thì không cần phải làm gì cả, còn nếu đoạn đó có ít nhất 2 khóa, ta chọn một khóa ngẫu nhiên nào đó của đoạn làm "chốt" ($Pivot$). Mọi khóa nhỏ hơn khóa chốt được xếp vào vị trí đứng trước chốt, mọi khóa lớn hơn khóa chốt được xếp vào vị trí đứng sau chốt. Sau phép hoán chuyển như vậy thì chốt sẽ được đặt vào vị trí đúng của nó còn các phần tử khác bị phân ra trên hai đoạn mà mọi khóa nằm bên trái chốt đều $\leq$ chốt và mọi khóa nằm bên phải chốt đều $\geq$ chốt. Vấn đề trở thành sắp xếp hai đoạn mới tạo ra (có độ dài ngắn hơn đoạn ban đầu) bằng phương pháp tương tự.

```
procedure QuickSort;

  procedure Partition(L, H: Integer); //Sắp xếp đoạn k[L...H]
  var
    i, j: Integer;
    Pivot: TKey;
  begin
    if L ≥ H then Exit; //Nếu đoạn có ít hơn 2 phần tử thì không cần sắp xếp
    Pivot := k[L]; //Chép giá trị khóa đầu đoạn vào chốt, tạo "chỗ trống" tại vị trí đầu đoạn
    i := L; j := H; //i: chỉ số đầu đoạn, j: chỉ số cuối đoạn
    repeat
      while (k[j] > Pivot) and (i < j) do j := j - 1; //Tìm khóa k[j] ≤ chốt từ cuối đoạn
      if i < j then //Chuyển khóa k[j] tìm được vào "chỗ trống", tạo chỗ trống mới tại vị trí j
        begin
          k[i] := k[j]; i := i + 1;
        end;
      while (k[i] < Pivot) and (i < j) do i := i + 1; //Tìm khóa k[i] ≥ chốt từ đầu đoạn
      if i < j then //Chuyển khóa k[i] tìm được vào "khoảng trống", tạo chỗ trống mới tại vị trí i
        begin
          k[j] := k[i]; j := j - 1;
        end;
    until i = j;
    k[i] := Pivot; //Đưa chốt vào vị trí đúng của nó
    Partition(L, i - 1); //Sắp xếp đoạn bên trái chốt
    Partition(i + 1, H); //Sắp xếp đoạn bên phải chốt
  end;

begin
  Partition(1, n);
end;
```

10.5.2. Tính đúng đắn của thuật toán

Tính đúng đắn của thuật toán có thể chứng minh được qua tính đúng đắn của thủ tục $Partition$. Ta sẽ chỉ ra rằng thủ tục $Partition(L,H)$ sẽ phân đoạn $k_{L...H}$ theo cách: đưa giá trị chốt $Pivot$ vào vị trí đúng của nó, các giá trị đứng bên trái chốt đều $\leq Pivot$ và các giá trị đứng bên phải chốt đều $\geq Pivot$.

Trước tiên, k_L được chọn làm khóa chốt. Phép gán $Pivot \coloneqq k_L$ chép giá trị khóa chốt ra biến $Pivot$. Có thể hình dung giá trị k_L được "chuyển" ra biến $Pivot$ để lại một "chỗ trống" tại vị trí L. Trước khi bước vào vòng lặp repeat…until, i là vị trí "chỗ trống" nằm ở đầu đoạn ($i = L$) và j là vị trí cuối đoạn ($j = H$).

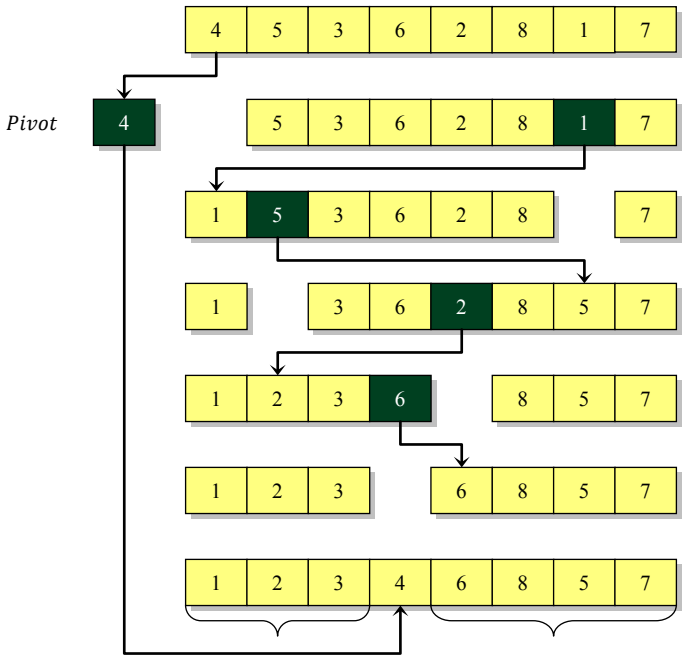
Không khó khăn để chứng minh các tính chất sau của vòng lặp repeat…until

- Trước và sau mỗi lần lặp, ta đảm bảo được $L \leq i \leq j \leq H$ và "chỗ trống" được duy trì ở vị trí i.
- Các lệnh gán $k_i \coloneqq k_j$ hay $k_j \coloneqq k_i$ chỉ đơn thuần là đưa một khóa vào "chỗ trống" và vị trí cũ của khóa trở thành chỗ trống mới. Điều này có nghĩa là chúng ta không bị mất dữ liệu (do ghi đè) bởi các lệnh gán giá trị khóa.

- Trước và sau mỗi lần lặp tất cả các khóa đứng trước k_i đều $\leq Pivot$ và tất cả các khóa đứng sau k_j đều $\geq Pivot$
- Hiệu số $j - i$ giảm dần sau mỗi lần lặp.

Vậy vòng lặp repeat…until chắc chắn sẽ kết thúc khi $i = j$. Theo các nhận xét trên, vị trí $i = j$ chính là "chỗ trống". Mọi khóa đứng trước "chỗ trống" đều $\leq Pivot$ và mọi khóa đứng sau "chỗ trống" đều $\geq Pivot$. Công việc cuối cùng là đưa $Pivot$ vào chỗ trống và thực hiện đệ quy quá trình phân đoạn với hai đoạn con. Tính đúng đắn và tính dừng của thuật toán được chứng minh.

Hình 10-1 là ví dụ về quá trình phân đoạn dãy khóa (4,5,3,6,2,8,1,7)



Hình 10-1. Ví dụ về quá trình phân đoạn

10.5.3. Thuật toán ngẫu nhiên

Có rất nhiều mô hình cài đặt QuickSort đã được đề xuất nhưng những mô hình cài đặt ở trên là một trong những mô hình hiệu quả nhất. Các cách cài đặt khác có thể ngắn gọn hơn nhưng lại hàm chứa những thao tác thừa trong việc hoán chuyển khóa.

Cách cài đặt trên sử dụng chốt là giá trị khóa đứng đầu đoạn. Có thể sửa đổi một chút mô hình cài đặt mỗi khi phân đoạn thì một khóa ngẫu nhiên trong đoạn được chọn làm chốt (*Randomized QuickSort*):

```
procedure QuickSort;
```

```
procedure Partition(L, H: Integer); //Sắp xếp đoạn k[L...H]
var
  i, j: Integer;
  Pivot: TKey;
begin
  if L ≥ H then Exit; //Nếu đoạn có ít hơn 2 phần tử thì không cần sắp xếp
  i := L + Random(H - L + 1); //Chọn i là chỉ số ngẫu nhiên trong đoạn L...H
  Pivot := k[i]; //Chép giá trị khóa k[i] vào chốt, tạo "chỗ trống" tại vị trí i
  k[i] := k[L]; //Chuyển khóa đầu đoạn vào "chỗ trống", vị trí đầu đoạn trở thành "chỗ trống"
  i := L; j := H; //Thực hiện tương tự trên
  repeat
    while (i < j) and (k[j] > Pivot) do j := j - 1; //Tìm khóa k[j] ≤ chốt từ cuối đoạn
    if i < j then //Chuyển khóa k[j] tìm được vào "chỗ trống", tạo chỗ trống mới tại vị trí j
      begin
        k[i] := k[j]; i := i + 1;
      end;
    while (i < j) and (k[i] < Pivot) do i := i + 1; //Tìm khóa k[i] ≥ chốt từ đầu đoạn
    if i < j then //Chuyển khóa k[i] tìm được vào "khoảng trống", tạo chỗ trống mới tại vị trí i
      begin
        k[j] := k[i]; j := j - 1;
      end;
  until i = j;
  k[i] := Pivot; //Đưa chốt vào vị trí đúng của nó
  Partition(L, i - 1); //Sắp xếp đoạn bên trái chốt
  Partition(i + 1, H); //Sắp xếp đoạn bên phải chốt
end;

begin
  Partition(1, n);
end;
```

Chúng ta sẽ phân tích ưu điểm của thuật toán ngẫu nhiên sau khi đánh giá độ phức tạp tính toán QuickSort.

10.5.4. Phân tích thời gian thực hiện giải thuật QuickSort

Việc đánh giá thời gian thực hiện giải thuật QuickSort là một trong những ví dụ kinh điển về kỹ thuật phân tích đánh giá thuật toán.

Với một lời gọi thủ tục $Partition(L, H)$, chúng ta xét quá trình phân đoạn (vòng lặp repeat…until) trước, trong quá trình này có thể coi phép toán so sánh hai giá trị khóa là phép toán tích cực. Để ý rằng mỗi khi một khóa k_i được so sánh với khóa chốt thì i sẽ tăng lên 1 và mỗi lần một khóa k_j được so sánh với khóa chốt thì j sẽ giảm đi 1. Trong quá trình phân đoạn, i tăng từ L và không vượt quá H, j giảm từ H và không nhỏ hơn L, tức là số phép toán tích cực phải thực hiện không lớn hơn $2(H - L + 1)$. Mặt khác khi quá trình phân đoạn kết thúc, i phải lớn hơn j tức là số phép toán tích cực phải thực hiện không nhỏ hơn $H - L + 1$. Điều này chỉ ra rằng thời gian phân đoạn sẽ là một hàm

$f(L,H) = \Theta(H - L + 1)$. Để thuận tiện cho các phép chứng minh tiếp theo, ta ký thiệu thời gian phân đoạn với một đoạn độ dài n trong dãy khóa là một đại lượng $\Theta(n)$

Ngay sau quá trình phân đoạn là hai lời gọi đệ quy $Partition$ với hai đoạn con. Ta có thể giả thiết rằng giá trị các khóa là hoàn toàn phân biệt khi xét với tình trạng dữ liệu tổng quát. Với giả thiết như vậy thì sau quá trình phân đoạn, chốt đã nằm ở vị trí đúng của nó và không tham gia vào quá trình phân đoạn đệ quy tiếp theo. Hai lời gọi đệ quy $Partition(L,j)$ và $Partition(i,H)$ sẽ thực hiện trên một đoạn gồm q phần tử và một đoạn gồm $n-1-q$ phần tử, $(0 \le q < n)$. Tức là nếu gọi $T(n)$ là thời gian thực hiện QuickSort trên dãy khóa n phần tử thì:

$$T(n) = T(q) + T(n-1-q) + \Theta(n) \qquad (10.1)$$

Trước hết ta chứng minh rằng thời gian thực hiện QuickSort là $\Omega(n \lg n)$ trong trường hợp tốt nhất. Tức là:

$$T(n) = \min_{0 \le q < n} \{T(q) + T(n-1-q)\} + \Theta(n) = \Omega(n \lg n)$$

Ta đưa ra dự đoán $T(n) \ge cn \lg n$ với một cách chọn hằng số c nào đó. thế dự đoán này vào công thức của $T(n)$, ta có:

$$T(n) \ge c \min_{0 \le q < n} \left\{ \underbrace{q \lg q + (n-1-q)\lg(n-1-q)}_{f(q)} \right\} + \Theta(n)$$

Hàm số $f(q) = q \lg q + (n-1-q)\lg(n-1-q)$ đạt cực tiểu tại $q = \frac{n-1}{2}$*, thế vào công thức, ta có:

---

* Đạo hàm cấp 1 của $f(q)$ là: $f'(q) = \frac{\ln q - \ln(n-1-q)}{\ln 2}$, đạo hàm cấp 2 của $f(q)$ là : $f''(q) = \frac{1}{\ln 2}\left(\frac{1}{q} + \frac{1}{n-1-q}\right) > 0$.

Vậy nên

$$\begin{cases} f'(q) < 0, & \text{nếu } q < \dfrac{n-1}{2} \\ f'(q) = 0, & \text{nếu } q = \dfrac{n-1}{2} \\ f'(q) > 0, & \text{nếu } q > \dfrac{n-1}{2} \end{cases}$$

Tức là hàm $f(q)$ nghịch biến trong $\left[0, \frac{n-1}{2}\right)$, đồng biến trong $\left(\frac{n-1}{2}, n-1\right]$ và đạt cực tiểu tại $q = \frac{n-1}{2}$.

$$\begin{aligned} T(n) &\geq c\left((n-1)\lg\frac{n-1}{2}\right)+\Theta(n) \\ &= c\left(n\lg\frac{n-1}{2}-\lg\frac{n-1}{2}\right)+\Theta(n) \\ &\geq c\left(n\lg\frac{n}{4}-\lg\frac{n-1}{2}\right)+\Theta(n), (\text{chọn } n\geq 2) \\ &= cn\lg n-2cn-c\lg\frac{n-1}{2}+\Theta(n) \\ &\geq cn\lg n \end{aligned} \tag{10.2}$$

Bất đẳng thức cuối cùng đúng nếu ta chọn hằng số c đủ nhỏ để đại lượng $\Theta(n)$ lớn át đi đại lượng $2cn + c\lg\frac{n-1}{2}$. Chúng ta chứng minh được thời gian thực hiện của QuickSort là $\Omega(n\lg n)$ trong trường hợp tốt nhất.

Tiếp theo, ta chứng minh thời gian thực hiện trung bình của QuickSort là $O(n\lg n)$. Xét các *giá trị khóa* trong dãy khóa, phép so sánh hai giá trị khóa được dùng làm phép toán tích cực. Ký hiệu các giá trị khóa theo thứ tự tăng dần là $z_1 < z_2 < \cdots < z_n$ và ký hiệu $z_{i\ldots j}$ là tập các giá trị khóa $\{z_i, z_{i+1}, \ldots, z_j\}$

Dựa trên mô hình cài đặt QuickSort, ta có các nhận xét sau:

- Mỗi cặp giá trị khóa (z_i, z_j) sẽ được so sánh với nhau không quá một lần vì mọi lệnh so sánh giá trị khóa đều là lệnh so sánh với khóa chốt và khóa chốt không tham gia trong các lệnh gọi $Partition$ đệ quy tiếp theo
- Nếu một giá trị khóa nằm giữa z_i và z_j được chọn làm chốt trước cả z_i và z_j thì hai giá trị z_i và z_j sẽ bị phân ra nằm trong hai đoạn rời nhau và chúng sẽ không bao giờ được so sánh với nhau nữa.
- Nếu z_i hoặc z_j được chọn làm chốt trước tất cả các giá trị khác trong $z_{i\ldots j}$ thì z_i và z_j sẽ được so sánh với nhau đúng một lần.

Ta định nghĩa một đại lượng X_{ij} như sau:

$$X_{ij} = \begin{cases} 1, & \text{nếu trong quá trình thực hiện có phép so sánh } z_i \text{ với } z_j \\ 0, & \text{nếu trong quá trình thực hiện không có phép so sánh } z_i \text{ với } z_j \end{cases}$$

Tổng số phép toán tích cực được thực hiện trong quá trình thực hiện thuật toán sẽ là:

$$X = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}$$

Xét tất cả các khả năng của dãy khóa thì trung bình số lần thực hiện phép toán tích cực trên một bộ dữ liệu kích thước n sẽ là giá trị kỳ vọng:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(X_{ij} = 1) \tag{10.3}$$

Xác suất để $X_{ij} = 1$ là xác suất mà z_i được so sánh với z_j, tức là bằng xác suất mà z_i hoặc z_j được chọn làm chốt trước tất cả các giá trị khác trong $z_{i\ldots j}$. Nếu coi quá trình chọn chốt là hoàn toàn ngẫu nhiên, thì xác suất để z_i (z_j) được chọn làm chốt trước tất cả các giá trị khác trong $z_{i\ldots j}$ là $\frac{1}{|z_{i\ldots j}|} = \frac{1}{j-i+1}$. Tức là:

$$\Pr(X_{ij} = 1) = \frac{2}{j-i+1}$$

Thế vào công thức (10.3), ta có:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr(X_{ij} = 1) \\ = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{\mathrm{j-i+1}} \\ = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad (\text{đổi biến } k := j-i) \\ < \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \\ = \sum_{i=1}^{n} \mathrm{O}(\ln(n)) = \mathrm{O}(n \ln n) = \mathrm{O}(n \lg n) \tag{10.4}$$

(do $\lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} = \ln n$, và $\ln n = \Theta(\lg n)$)

Ta chứng minh được thời gian thực hiện trung bình giải thuật QuickSort là $\mathrm{O}(n \lg n)$

Nếu ký hiệu $T_{best}(n)$ là thời gian thực hiện QuickSort trong trường hợp tốt nhất, $T_{average}(n)$ là thời gian trung bình thực hiện QuickSort thì từ các kết quả trên:

$$\Omega(n \lg n) = T_{best}(n) < T_{average}(n) = \mathrm{O}(n \lg n)$$

Vậy thì phân tích trong trường hợp tốt nhất và trung bình, QuickSort có thời gian thực hiện giải thuật là $\Theta(n \lg n)$.

Từ những kết quả đã chứng minh được, ta có thể đánh giá thời gian thực hiện của QuickSort trong trường hợp xấu nhất không mấy khó khăn.

Trong việc đánh giá thời gian trung bình, chúng ta đã biết số lần thực hiện phép toán tích cực là $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, trong đó $X_{ij} \in \{0,1\}$, vậy thì ngay cả trong trường hợp xấu nhất, số lượng này luôn nhỏ hơn n^2, tức là:

$$T_{worst}(n) = O(n^2)$$

Ở đây ta ký hiệu $T_{worst}(n)$ là thời gian thực hiện QuickSort trong trường hợp xấu nhất.

Để đánh giá $T_{worst}(n)$ qua ký pháp Ω, ta xem lại công thức (10.1):

$$T(n) = T(q) + T(n-1-q) + \Theta(n)$$

Nếu ta thay $q = 0$:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned} \qquad (10.5)$$

Ta sẽ chứng minh trong trường hợp này $T(n) = \Omega(n^2)$. Dự đoán rằng sẽ $\exists c > 0$ để $T(n) > cn^2$, thế vào công thức (10.5), ta có:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &\geq c(n-1)^2 + \Theta(n) \\ &= cn^2 - 2cn - c + \Theta(n) \\ &\geq cn^2 \end{aligned} \qquad (10.6)$$

Bất đẳng thức cuối cùng đúng với các chọn hằng số c đủ nhỏ để đại lượng $\Theta(n)$ lớn át đi đại lượng $2cn + c$.

Thời gian thực hiện QuickSort trong trường hợp xấu nhất ít ra là không nhỏ hơn $T(n)$ trong trường hợp này. Tức là:

$$T_{worst}(n) = \Omega(n^2)$$

Kết hợp lại ta có $T_{worst}(n) = \Theta(n^2)$

Kết luận:

- Trường hợp tốt nhất với QuickSort là trong mỗi lần chọn chốt, thuật toán chọn đúng trung vị của đoạn (phần tử sẽ nằm giữa đoạn khi sắp thứ tự) và tiếp tục với hai đoạn con có độ dài cân bằng, trong trường hợp này thời gian thực hiện của thuật toán là $\Theta(n \lg n)$. Cũng chứng minh được rằng chi phí về bộ nhớ phụ trong trường hợp tốt nhất là $\Theta(\lg n)$.

- Trường hợp xấu nhất với QuickSort là trong mỗi lần chọn chốt, thuật toán chọn đúng phần tử nhỏ nhất hay lớn nhất trong đoạn, sau khi đưa chốt về vị trí đúng, thuật toán sẽ phải tiếp tục với đoạn gồm tất cả các phần tử còn lại, trong trường hợp này thời gian thực hiện của thuật toán là $\Theta(n^2)$. Cũng chứng minh được rằng chi phí về bộ nhớ phụ trong trường hợp xấu nhất là $\Theta(n)$*.
- Trung bình thời gian thực hiện QuickSort là $\Theta(n \lg n)$ và trung bình chi phí về bộ nhớ phụ là $\Theta(\lg n)$.

10.6. Trung vị và thứ tự thống kê

Việc chọn chốt cho phép phân đoạn quyết định hiệu quả của QuickSort, nếu chọn chốt không tốt, rất có thể việc phân đoạn bị suy biến thành trường hợp xấu khiến QuickSort hoạt động chậm. Những ví dụ sau đây cho thấy có thể dễ dàng tìm ra những bộ dữ liệu khiến QuickSort hoạt động chậm với phép chọn chốt tồi. Bạn có thể cài đặt cụ thể và thử sử dụng các chiến lược chọn chốt đơn định với những bộ dữ liệu sau:

- Nếu như chọn chốt là khóa đầu đoạn hay chọn chốt là khóa cuối đoạn thì QuickSort sẽ trở thành "Slow" Sort với dãy $(1,2,\dots,n)$ với n đủ lớn.
- Nếu như chọn chốt là khóa giữa đoạn thì QuickSort cũng trở thành "Slow" Sort với dãy $(1,2,\dots,n,n,\dots,2,1)$ với n đủ lớn.
- Trong trường hợp chọn chốt là khóa nằm ở vị trí ngẫu nhiên trong đoạn, thật khó có thể tìm ra một bộ dữ liệu khiến cho QuickSort hoạt động chậm. Nhưng ta cũng cần hiểu rằng với mọi thuật toán tạo số ngẫu nhiên, trong $n!$ dãy hoán vị của dãy $(1,2,\dots,n)$ thế nào cũng có một dãy làm QuickSort bị suy biến (rơi vào trường hợp xấu), tuy nhiên xác suất xảy ra dãy này quá nhỏ† và cũng rất khó để chỉ ra được nên việc sử dụng cách chọn chốt là khóa nằm ở vị trí ngẫu nhiên có thể coi là an toàn với các trường hợp suy biến của QuickSort.

Phần trung vị và thứ tự thống kê (median and order statistics) này được trình bày sau nội dung thảo luận về QuickSort bởi nó cung cấp một chiến lược chọn chốt "đẹp" trên lý thuyết, nghĩa là chúng ta luôn đưa được QuickSort về trường hợp tốt nhất. Ta xét bài toán

---

* Có thể cải tiến cài đặt của QuickSort để chi phí bộ nhớ phụ là $O(\lg n)$ trong trường hợp xấu nhất bằng cách: sau khi phân đoạn, chúng ta sẽ chỉ gọi đệ quy để sắp xếp đoạn ngắn hơn và lặp lại quá trình phân đoạn với đoạn dài hơn.

† Mọi bộ tạo số ngẫu nhiên trong máy tính đều có luật sinh, nếu nắm bắt được luật sinh có thể tạo ra bộ dữ liệu xấu cho QuickSort, nhưng nếu luật sinh luôn thay đổi dựa trên mạch đồng hồ máy tính thì không thể nắm bắt được do thời điểm tạo dữ liệu và thời điểm chạy QuickSort là khác nhau. Vì vậy khi cài đặt QuickSort trên Free Pascal, nên có lệnh Randomize ở đầu chương trình.

tìm trung vị của dãy khóa và bài toán tổng quát hơn: Bài toán *thứ tự thống kê* (*Order statistics*).

10.6.1. Bài toán

Cho dãy khóa $K = (k_1, k_2, \dots, k_n)$ hãy chỉ ra khóa sẽ đứng thứ p trong dãy khi sắp thứ tự. Khi $p = \lceil n/2 \rceil$ thì bài toán thứ tự thống kê trở thành bài toán tìm trung vị của dãy khóa. Sau đây ta sẽ nói về một số cách giải quyết bài toán thứ tự thống kê với mục tiêu cuối cùng là tìm ra một thuật toán để giải bài toán này với thời gian thực hiện trong trường hợp xấu nhất là $O(n)$.

10.6.2. Thuật toán

Cách tệ nhất là sắp xếp lại toàn bộ dãy khóa và đưa ra khóa đứng thứ p của dãy đã sắp. Trong các thuật toán sắp xếp tổng quát mà ta thảo luận trong bài, không thuật toán nào cho phép thực hiện việc này trong thời gian $O(n)$.

Cách thứ hai là một sửa đổi QuickSort: Trước hết ta viết một hàm $Partition(L, H, s)$ chọn khóa chốt $Pivot := k_s$ $(L \le s \le H)$ và phân đoạn $k_{L\dots H}$ làm hai đoạn con (thực ra là ba): Các khóa của đoạn đầu ≤ chốt, tiếp theo là chốt (vị trí tách đoạn), rồi đến các khóa của đoạn sau ≥ chốt. Hàm này trả về vị trí tách đoạn:

```
function Partition(L, H, s: Integer): Integer; //Phân đoạn k[L...H] dựa vào chốt k[s]
var
  i, j: Integer;
  Pivot: TKey;
begin
  Pivot := k[s]; k[s] := k[L];
  i := L; j := H;
  repeat
    while (k[j] > Pivot) and (i < j) do j := j - 1;
    if i < j then
      begin
        k[i] := k[j]; i := i + 1;
      end;
    while (k[i] < Pivot) and (i < j) do i := i + 1;
    if i < j then
      begin
        k[j] := k[i]; j := j - 1;
      end;
  until i = j;
  k[i] := Pivot;
  Result := i; //Trả về vị trí tách đoạn
end;
```

Để tìm giá trị đứng thứ p khi sắp xếp dãy khóa, ta phân đoạn dãy khóa theo một chốt ngẫu nhiên nằm trong dãy, khi đó ta hoàn toàn có thể xác định được khóa cần tìm nằm ở đoạn nào. Nếu khóa đó nằm ở vị trí tách đoạn thì xong, nếu không ta lặp lại quá trình tìm

kiếm ở một trong hai phân đoạn chứ không cần gọi đệ quy để sắp xếp cả hai phân đoạn như QuickSort.

```
L := 1; H := n;
repeat //Tìm trong đoạn k[L...H]
  i := Partition(L, H, L + Random(H - L + 1));
  if p < i then H := i - 1 //Tìm tiếp trong đoạn k[L...i – 1]
  else L := i + 1; //Tìm tiếp trong đoạn k[i + 1...H] nếu p > i
until p = i;
Output ← k[p];
```

Thời gian thực hiện của giải thuật trong trường hợp tốt nhất và trung bình là $O(n)$. Trường hợp xấu nhất xảy ra khi mà $p = n$ và trong quá trình thực hiện, chốt Pivot được chọn luôn là khóa nhỏ nhất của đoạn $k_{L\ldots H}$, khi đó thời gian thực hiện giải thuật là $\Omega(n^2)$. Mặc dù thuật toán *hoạt động tốt trên thực tế*, trên lý thuyết ta vẫn phải hướng tới một thuật toán tốt hơn nữa.

Cách thứ ba: Sự bí hiểm của số 5.

Ta sẽ viết một thủ tục $Select(L, H, p)$ để hoán chuyển các khóa trong đoạn $k_{L\ldots H}$ sao cho khóa ở vị trí p: k_p $(L \le p \le H)$ mang giá trị đúng của nó khi sắp xếp dãy khóa $k_{L\ldots H}$. Cụ thể là ta sẽ hoán chuyển các khóa để khóa k_p không nhỏ hơn các khóa đứng trước nó và không lớn hơn các khóa đứng sau nó: $k_{L\ldots p-1} \le k_p \le k_{p+1\ldots H}$.

Nếu dãy này có không quá 60 khóa, thuật toán sắp xếp kiểu chọn sẽ được áp dụng trên dãy khóa này để dồn các khóa nhỏ nhất trong đoạn về các vị trí $k_{L\ldots p}$:

```
procedure SelectionSort(L, H, p: Integer); //Dồn p phần tử nhỏ nhất trong đoạn k[L...H] về đầu đoạn
var
  i, j, jmin: Integer;
begin
  for i := L to p do
    begin
      jmin := i;
      for j := i + 1 to H do
         if k[j] < k[jmin] then jmin := j;
      if jmin ≠ i then Swap(k[i], k[jmin]);
    end;
end;
```

Nếu dãy này có nhiều hơn 60 khóa, ta chia các khóa $k_{L\ldots H}$ thành các nhóm 5 khóa:

$k_{L\ldots L+4}, k_{L+5\ldots L+9}, k_{L+10\ldots L+14}, \cdots$

Nếu cuối cùng quá trình chia nhóm còn lại ít hơn 5 khóa (do độ dài đoạn $k_{L\ldots H}$ không chia hết cho 5), ta bỏ qua không xét những khóa dư thừa này.

Với mỗi nhóm 5 khóa kể trên, ta tìm trung vị của nhóm (gọi tắt là trung vị nhóm - khóa đứng thứ 3 khi sắp thứ tự 5 khóa) và đẩy trung vị nhóm ra đầu đoạn theo thứ tự:

Trung vị của $k_{L\ldots L+4}$ sẽ được đảo giá trị cho k_L

Trung vị của $k_{L+5\ldots L+9}$ sẽ được đảo giá trị cho k_{L+1}

…

Giả sử trung vị của nhóm cuối cùng sẽ được đảo giá trị cho k_q. Phép dồn trung vị nhóm ra phần đầu đoạn $k_{L\ldots q}$ được thực hiện bằng lệnh $q := Gathering(L, H)$

```
function Gathering(L, H: Integer): Integer;
var
  i, j: Integer;
begin
  i := L; j := L;
  repeat
    SelectionSort(i, i + 4, i + 2); //Đưa trung vị của nhóm k[i...i+4] về vị trí i + 2
    Swap(k[i + 2], k[j]); //Dồn trung vị nhóm ra đầu đoạn
    i := i + 5;
    j := j + 1;
  until i + 4 > H;
  Result := j - 1; //Trả về vị trí cuối của đoạn các trung vị nhóm vừa dồn ra
end;
```

Sau khi các trung vị nhóm đã tập trung về các vị trí $k_{L\ldots q}$, ta đặt chốt $Pivot$ bằng trung vị của các trung vị nhóm thông qua một lệnh gọi đệ quy thủ tục Select:

```
Select(L, q, (L + q) div 2); //Đưa trung vị của các trung vị nhóm về vị trí đúng
Pivot := k[(L + q) div 2];
```

Tiếp tục các lệnh của thủ tục $Select$ như thế nào sẽ bàn sau, bây giờ ta giả sử thủ tục này hoạt động đúng để xét một tính chất quan trọng của chốt $Pivot$:

Nếu độ dài đoạn $k_{L\ldots H}$ là $n = H - L + 1$ thì có $\lfloor n/5 \rfloor$ nhóm, nên cũng có $\lfloor n/5 \rfloor$ trung vị nhóm. $Pivot$ là trung vị của các trung vị nhóm nên $Pivot$ phải lớn hơn hay bằng $\lfloor \lfloor n/5 \rfloor /2 \rfloor$ trung vị nhóm, mỗi trung vị nhóm lại lớn hơn hay bằng 3 khóa của nhóm (tính cả chính nó). Vậy có thể suy ra rằng $Pivot$ lớn hơn hay bằng $3\lfloor \lfloor n/5 \rfloor /2 \rfloor$ khóa của đoạn $k_{L\ldots H}$. Lập luận tương tự, ta có $Pivot$ nhỏ hơn hay bằng $3\lfloor \lfloor n/5 \rfloor /2 \rfloor$ khóa khác của đoạn $k_{L\ldots H}$. Với $n \geq 60$, ta có $3\lfloor \lfloor n/5 \rfloor /2 \rfloor \geq n/4$. Suy ra:

- Có ít nhất $n/4$ khóa nhỏ hơn hay bằng $Pivot \Rightarrow$ có nhiều nhất $3n/4$ khóa lớn hơn $Pivot$
- Có ít nhất $n/4$ khóa lớn hơn hay bằng $Pivot \Rightarrow$ có nhiều nhất $3n/4$ khóa nhỏ hơn $Pivot$

Ta quay lại xây dựng tiếp thủ tục $Select$, khi đã xác định được chốt $Pivot$, chúng ta dùng chốt này để chia đoạn $k_{L...H}$ thành hai đoạn con: Đoạn đầu $\leq Pivot$, tiếp theo là khóa ở vị trí tách đoạn $= Pivot$, và đoạn sau $\geq Pivot$. Thuật toán sẽ kết thúc ngay nếu vị trí p chính là vị trí tách đoạn, nếu không quá trình sẽ tiếp tục với đoạn đầu hay đoạn sau, có độ dài tối đa bằng $3/4$ đoạn ban đầu.

```
procedure Select(L, H, p: Integer); //Đưa phần tử đứng thứ p khi sắp thứ tự về vị trí k[p]
var
  i, s, q: Integer;
begin
  if H - L < 60 then //Độ dài đoạn không quá 60
    begin
      SelectionSort(L, H, p); //Thực hiện sắp xếp chọn
      Exit; //Thoát luôn
    end;
  q := Gathering(L, H); //Dồn các trung vị nhóm về đầu đoạn k[L...q]
  s := (L + q) div 2; //s vị trí giữa đoạn các trung vị nhóm
  Select(L, q, s); //Đưa trung vị các các trung vị nhóm về vị trí đúng s
  i := Partition(L, H, s); //Thực hiện phân đoạn k[L...H] theo chốt = trung vị của các trung vị nhóm
  if p < i then Select(L, i - 1, p) //Vị trí p nằm ở đoạn đầu, làm tương tự với đoạn con k[L...i - 1]
  else
    if p > i then Select(i + 1, H, p); //Vị trí p nằm ở đoạn sau, làm tương tự với đoạn con k[i + 1...H]
end;
```

Nếu gọi $T(n)$ là thời gian thực hiện thủ tục $Select$ trong trường hợp xấu nhất với độ dài dãy khóa $k_{L...H}$ bằng n. Khi đó với $n \leq 60$ thì thuật toán sắp xếp chọn sẽ được thực hiện, có thể coi đoạn chương trình này kết thúc trong thời gian ít hơn c_1, với c_1 là một hằng số đủ lớn. Với $n \geq 60$, nhìn vào các đoạn mã trong hàm $Select$, lệnh $Select(L, q, s)$ có thời gian thực hiện không quá $T(n/5)$. Lệnh $Select(L, i-1, p)$ cũng như lệnh $Select(i+1, H, p)$ có thời gian thực hiện không quá $T(3n/4)$ do tính chất của trung vị các trung vị nhóm. Thời gian thực hiện các lệnh khác trong thủ tục $Select$ tổng lại có thể coi là không quá $c_2 n$ với c_2 là một hằng số đủ lớn.

Vậy thì ngay cả trong trường hợp xấu nhất:

$$T(n) \leq \begin{cases} c_1, & \text{nếu } n \leq 60 \\ c_2 n + T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right), & \text{nếu } n > 60 \end{cases} \qquad (10.7)$$

Ta sẽ chứng minh rằng $T(n) = O(n)$.

Đặt $c := \max(c_1, 20c_2)$, khi đó với $1 \leq n \leq 60$, rõ ràng $T(n) \leq c_1 \leq c \leq cn$. Giả thiết quy nạp rằng $T(m) \leq cm, \forall m < n$, ta sẽ chứng minh $T(n) \leq cn$, thật vậy:

$$\begin{aligned} T(n) &\le c_2 n + T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) \\ &\le \frac{1}{20}cn + \frac{1}{5}cn + \frac{3}{4}cn \\ &= cn \end{aligned} \tag{10.8}$$

Vậy $T(n) = O(n)$. Sự bí ẩn của việc chọn số 5 cho kích thước nhóm đã được giải thích $(1/5 + 3/4 < 1)$

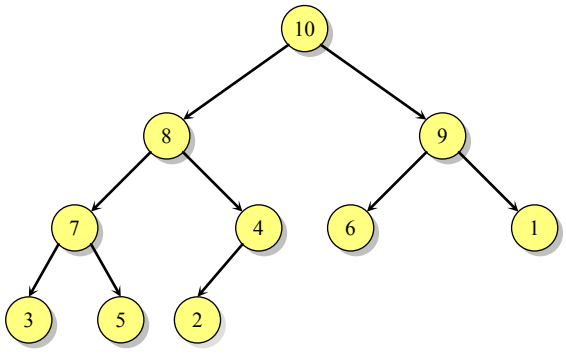
Chúng ta đã có thuật toán để giải bài toán thứ tự thống kê trong thời gian $O(n)$. Từ kết quả này, chúng ta có thể cài đặt thuật toán QuickSort để chỉ mất thời gian $O(n \lg n)$ ngay cả trong trường hợp xấu nhất. Bởi tại mỗi lần phân đoạn của QuickSort, ta có thể tìm được trung vị của đoạn trong thời gian $O(n)$ bằng việc giải quyết bài toán thứ tự thống kê. Điều đó có nghĩa là cho tới thời điểm này, khi giải mọi bài toán có chứa thủ tục sắp xếp, ta có thể coi thời gian thực hiện thủ tục sắp xếp đó là $O(n \lg n)$ với mọi tình trạng dữ liệu vào.

10.7. HeapSort

Thuật toán sắp xếp kiểu vun đống - HeapSort [**20**] - không những đóng góp một phương pháp sắp xếp hiệu quả mà còn xây dựng một cấu trúc dữ liệu quan trọng để biểu diễn hàng đợi có độ ưu tiên.

10.7.1. Đống

Đống (*Binary heap*) là một dạng cây nhị phân gần hoàn chỉnh đặc biệt mà giá trị lưu tại mọi nút có độ ưu tiên cao hơn hay bằng giá trị lưu trong hai nút con của nó. Trong thuật toán sắp xếp kiểu vun đống, ta coi quan hệ "ưu tiên hơn hay bằng" là quan hệ "lớn hơn hoặc bằng": ≥



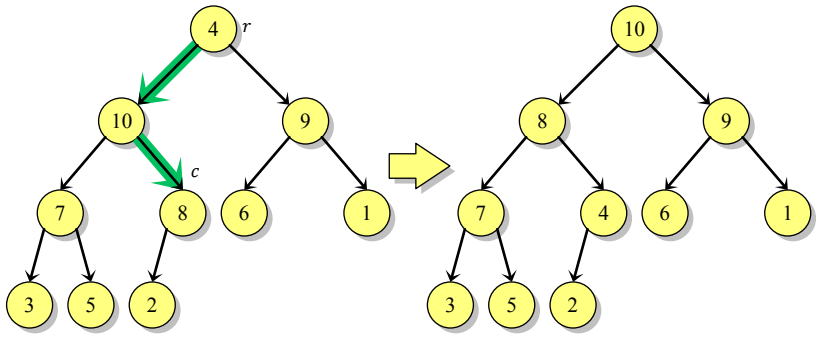
Hình 10-2. Binary Heap

10.7.2. Vun đống

Trong mục 7.4.2, ta đã biết một dãy khóa $k[1 \ldots n]$ là biểu diễn của một cây nhị phân gần hoàn chỉnh mà k_i là giá trị lưu trong nút thứ i, nút con của nút thứ i là nút $2i$ và nút $2i + 1$, nút cha của nút thứ j là nút $\lfloor j/2 \rfloor$. Vấn đề đặt ra là sắp lại dãy khóa đã cho để nó biểu diễn một đống.

Vì cây nhị phân chỉ gồm có một nút hiển nhiên là đống, nên ta sẽ thực hiện thuật toán vun đống từ dưới lên (từ lá lên gốc) để đảm bảo rằng khi bắt đầu thuật toán vun một nhánh cây gốc r thành đống thì hai nhánh con của nó (nhánh gốc $2r$ và $2r + 1$) đã được vun thành đống trước đó rồi.

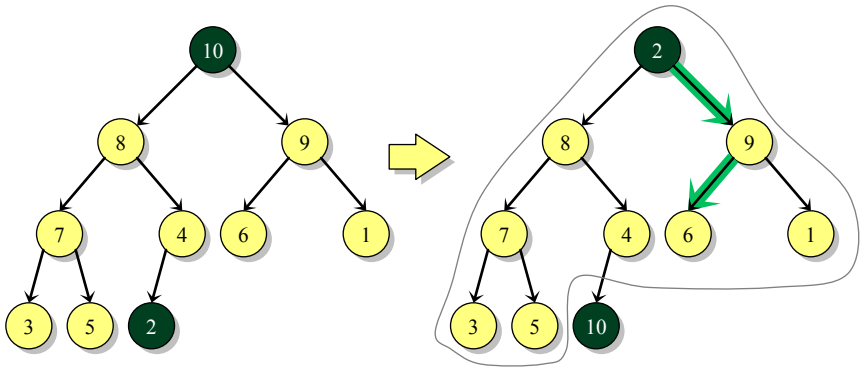
Giả sử ở nút r chứa giá trị $temp$. Từ r, ta cứ đi tới nút con chứa giá trị lớn nhất trong 2 nút con, cho tới khi gặp phải một nút c mà không nút con nào của c chứa giá trị lớn hơn $temp$ (nút lá cũng là trường hợp riêng của điều kiện này). Dọc trên đường đi từ r tới c, ta đẩy giá trị chứa ở nút con lên nút cha và đặt giá trị $temp$ vào nút c.



Hình 10-3. Vun đống

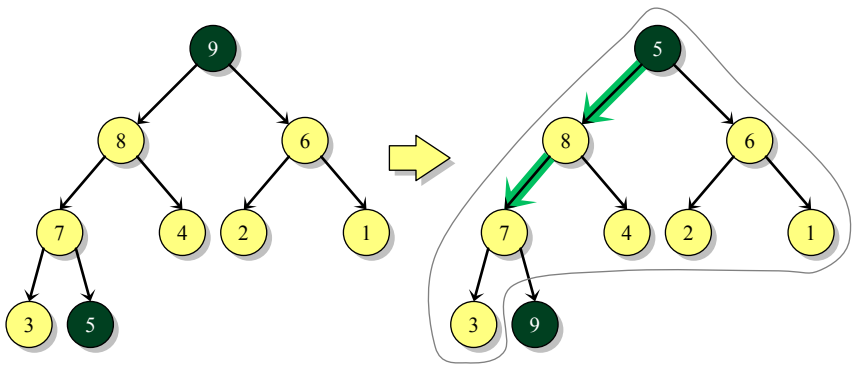
10.7.3. Tư tưởng của HeapSort

Đầu tiên, dãy khóa $k_{1\ldots n}$ được vun từ dưới lên để nó biểu diễn một đống, khi đó khóa k_1 tương ứng với nút gốc của đống là khóa lớn nhất, ta đảo giá trị khóa đó cho k_n và không tính tới k_n nữa (Hình 10-4).



Hình 10-4. Đảo giá trị k_1 cho k_n và xét phần còn lại

Còn lại dãy khóa $k_{1\ldots n-1}$ tuy không còn là biểu diễn của một đống nữa nhưng nó lại biểu diễn cây nhị phân gần hoàn chỉnh mà hai nhánh con của nút 1 (nhánh gốc 2 và nhánh gốc 3) đã là đống rồi. Vậy chỉ cần vun một lần, ta được một đống, lại đảo giá trị k_1 cho k_{n-1} và tiếp tục quá trình này cho tới khi đống chỉ còn lại 1 nút (Hình 10-5).



Hình 10-5. Vun phần còn lại thành đống rồi lại đảo giá trị k_1 cho k_{n-1}…

10.7.4. Mô hình cài đặt HeapSort

Thuật toán HeapSort có hai thủ tục chính:

- Thủ tục $Heapify(r,p)$ vun cây gốc r thành đống trong điều kiện hai cây gốc $2r$ và $2r+1$ đã là đống rồi, các khóa đứng sau k_p: $k_{p+1\ldots n}$ đã nằm ở vị trí đúng và không được tính tới nữa.
- Thủ tục $HeapSort$ thực hiện quá trình sắp xếp theo ý tưởng trên

```
procedure HeapSort;
var
  i: Integer;
```

```
  procedure Heapify(r, p: Integer); //Vun cây gốc r thành đống
  var
    temp: TKey;
    c: Integer;
  begin
    temp := k[r];
    repeat
      //Từ r tìm  nút con c mang giá trị lớn hơn trong hai nút con
      c := r * 2;
      if (c < p) and (k[c] < k[c + 1]) then c := c + 1;
      if (c > p) or (k[c] ≤ temp) then Break; //Dừng ngay nếu r không có nút con mang giá trị > temp
      k[r] := k[c]; //Đẩy giá trị từ nút con lên nút cha
      r := c; //Đi xuống nút c
    until False;
    k[r] := temp; //Đặt giá trị vào nút cuối đường đi
  end;

begin
  for i := n div 2 downto 1 do Heapify(i, n); //Vun dãy từ dưới lên tạo thành đống
  for i := n downto 2 do
    begin
      Swap(k[1], k[i]); //Chuyển khóa lớn nhất ra cuối dãy
      Heapify(1, i - 1); //Vun phần còn lại thành đống
    end;
end;
```

Không khó để chứng minh tính đúng đắn của HeapSort. Xét về thời gian thực hiện giải thuật, ta đã biết rằng cây nhị phân gần hoàn chỉnh có n nút thì chiều cao của nó là $\lfloor \lg n \rfloor$. Cứ cho là trong trường hợp xấu nhất thủ tục $Heapify$ phải thực hiện tìm đường đi từ nút gốc tới nút lá ở xa nhất thì đường đi tìm được cũng chỉ dài bằng chiều cao của cây nên thời gian thực hiện một lần gọi $Heapify$ là $O(\lg n)$. Từ đó có thể suy ra, phép khởi tạo đống - vun dãy từ dưới lên tạo thành đống - mất thời gian $O(n \lg n)$*, thao tác sắp xếp thực hiện $n-1$ lần việc đảo giá trị hai khóa và vun đống nên cũng mất thời gian $O(n \lg n)$. Vậy thời gian thực hiện giải thuật HeapSort là $O(n \lg n)$ bất kể tình trạng dữ liệu đầu vào. Cũng chứng minh được thời gian thực hiện giải thuật HeapSort cũng là $\Omega(n \lg n)$ nếu như các giá trị khóa trong dãy là hoàn toàn phân biệt [**6**].

Chi phí về bộ nhớ phụ sử dụng trong HeapSort là $O(1)$.

10.7.5. Hàng đợi ưu tiên

Hàng đợi có độ ưu tiên, gọi tắt là *hàng đợi ưu tiên* (*Priority Queue*) là một cấu trúc dữ liệu quan trọng trong việc cài đặt nhiều thuật toán. Hàng đợi ưu tiên là một kiểu danh sách chứa các phần tử của một tập hữu hạn S, mỗi phần tử của S được gán một mức độ

---

* Thực ra thao tác khởi tạo đống cài đặt bằng phép vun từ dưới lên chỉ mất thời gian $O(n)$. Tham khảo [**BO**] hoặc [**BO**]

ưu tiên nhất định. Chúng ta đánh số các phần tử từ 1 tới n và đồng nhất mỗi phần tử với chỉ số của nó. Gọi $p[i]$ là một số phản ánh mức độ ưu tiên của phần tử $i \in \{1,2,\dots,n\}$.

Với một hàng đợi ưu tiên, có các thao tác chính:

- Phép $Insert(i)$: Nếu phần tử i chưa có trong hàng đợi ưu tiên, thao tác này đẩy i vào hàng đợi ưu tiên.
- Phép Get: Trả về phần tử có mức độ ưu tiên lớn nhất trong hàng đợi ưu tiên.
- Phép $Extract$: Trả về phần tử có mức độ ưu tiên lớn nhất trong hàng đợi ưu tiên và loại bỏ phần tử đó khỏi hàng đợi ưu tiên.
- Phép $Update(i, newp)$: Thay đổi mức độ ưu tiên của i thành $newp$.

❑ Cài đặt hàng đợi ưu tiên bằng mảng biểu diễn binary heap

```
const
  n = ...; //Số phần tử cực đại
type
  TBinaryHeap = record
    Items: array[1..n] of Integer;
    nItems: Integer;
    Pos: array[1..n] of Integer;
  end;
var
  Heap: TBinaryHeap;
```

Cách cài đặt này sử dụng:

- Mảng $Items[1 \dots n]$ để lưu các phần tử trong hàng đợi ưu tiên $Heap$. Mảng này biểu diễn một cấu trúc binary heap mà nút r là cha của hai nút $2r$ và $2r+1$. Độ ưu tiên của nút cha lớn hơn hay bằng độ ưu tiên của cả hai nút con. Tức là nếu nút c là hậu duệ của nút r thì $p[Items[r]] \geq p[Items[c]]$
- $nItems$ là số phần tử thực sự đang nằm trong $Heap$
- $Pos[1 \dots n]$ là mảng quản lý vị trí, $Pos[i]$ là vị trí của phần tử i trong mảng $Items$, quy ước rằng $Pos[i] = 0 \Leftrightarrow i \notin Heap$. Cụ thể là nếu $Pos[i] \neq 0$ thì $Pos[i] = j \Leftrightarrow Items[j] = i$. Tức là nếu coi Pos và $Items$ biểu diễn hai ánh xạ $\{1 \dots n\} \to \{1 \dots n\}$ thì ánh xạ Pos là ánh xạ ngược của ánh xạ $Items$ và ngược lại.

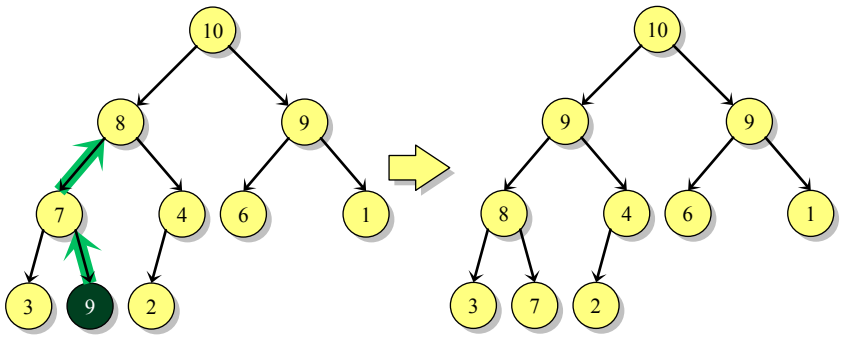
❑ Khởi tạo hàng đợi ưu tiên rỗng

Trước hết chúng ta viết một thủ tục $Init$ tương ứng với thao tác khởi tạo hàng đợi ưu tiên $Heap := \emptyset$:

```
procedure Init;
var
  i: Integer;
begin
  with Heap do
    begin
      nItems := 0;
      for i := 1 to n do Pos[i] := 0;
    end;
end;
```

❑ Phép Up-Heap

Giả sử chúng ta có một hàng đợi ưu tiên $Heap$ và i là một phần tử đang nằm trong $Heap$. Nếu ta tăng độ ưu tiên $p[i]$ lên thì có thể phá vỡ cấu trúc binary heap. Vì vậy mỗi khi độ ưu tiên của một phần tử i tăng lên, chúng ta phải dùng một phép $UpHeap(i)$ để khôi phục cấu trúc binary heap. Phép $UpHeap$ thực hiện như sau: Tìm nút chứa phần tử i trong $Heap$, bắt đầu từ nút này ta cứ đi lên nút cha cho tới khi gặp nút gốc hoặc một nút có nút cha chứa phần tử có độ ưu tiên $\geq p[i]$. Dọc trên đường đi ta kéo phần tử từ nút cha xuống nút con và đặt phần tử i vào nút cuối đường đi (Hình 10-6). Chú ý rằng mỗi khi thay đổi vị trí phần tử của Heap, chúng ta phải cập nhật mảng Pos cho tương thích với cấu trúc hiện tại của mảng $Items$



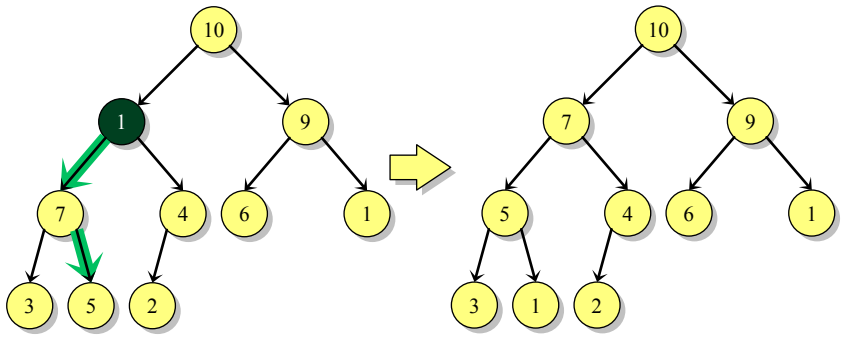
Hình 10-6. Up-Heap

```
procedure UpHeap(i: Integer);
var
  r, c: Integer;
begin
  with Heap do
    begin
      c := Pos[i]; //Tìm nút chứa phần từ i trong Heap
      repeat
        r := c div 2; //Xét nút cha của c
        if (r = 0) or (p[Items[r]] ≥ p[i]) then //Nếu c là gốc hoặc nút cha của c có độ ưu tiên ≥ p[i]
          Break; //Thoát ngay
        Items[c] := Items[r]; //Kéo phần tử từ nút cha xuống nút con
        Pos[Items[c]] := c; //Cập nhật lại mảng quản lý vị trí
        c := r; //Đi lên phía gốc
      until False;
      Items[c] := i; //Đặt phần tử i vào nút cuối đường đi
      Pos[i] := c; //Cập nhật lại mảng quản lý vị trí
    end;
end;
```

Vì độ sâu của một nút trong cây nhị phân gần hoàn chỉnh không vượt quá $\lfloor \lg nItems \rfloor$ nên phép $UpHeap$ có thời gian thực hiện là $O(\lg nItems) = O(\lg n)$.

❑ Phép Down-Heap

Đối ngẫu với phép $UpHeap$, phép $DownHeap(i)$ dùng để khôi phục lại cấu trúc binary heap mỗi khi ta giảm độ ưu tiên của một phần tử i. Cách làm giống như thủ tục Heapify của HeapSort: Tìm nút chứa phần tử i trong Heap. Bắt đầu từ nút này ta đi theo nút con mang độ ưu tiên lớn hơn trong hai nút con cho tới khi gặp một nút không có nút con nào mang độ ưu tiên lớn hơn $p[i]$. Dọc trên đường đi này ta đẩy phần tử chứa ở nút con lên nút cha và đặt phần tử i vào nút cuối đường đi (Hình 10-7). Tương tự như thao tác $UpHeap$, ta sẽ cập nhật mảng Pos cho tương thích với cấu trúc hiện tại của mảng $Items$ mỗi khi thay đổi vị trí phần tử.



Hình 10-7. Down-Heap

```
procedure DownHeap(i: Integer);
var
  r, c: Integer;
begin
  with Heap do
    begin
      r := Pos[i]; //Tìm nút chứa phần tử i
      repeat
         c := r * 2; //Xét nút con trái
         if (c < nItems) and (p[Items[c]] < p[Items[c + 1]]) then
           c := c + 1; //Chuyển sang nút con phải nếu nó chứa phần tử ưu tiên hơn
         if (c > nItems) or (p[Items[c]] ≤ p[i]) then
           Break; //Dừng ngay nếu r không có nút con chứa phần tử ưu tiên hơn p[i]
         Items[r] := Items[c]; //Đẩy phần tử từ nút con lên nút cha
         Pos[Items[r]] := r; //Cập nhật mảng quản lý vị trí
         r := c; //Đi xuống dưới lá
      until False;
      Items[r] := i; //Đặt phần tử i vào nút cuối đường đi
      Pos[i] := r; //Cập nhật mảng quản lý vị trí
    end;
end;
```

Tương tự như phép $UpHeap$, dễ thấy rằng thời gian thực hiện phép $DownHeap$ là $O(\lg nItems) = O(\lg n)$.

❑ Cài đặt các phép toán của hàng đợi ưu tiên

Khi đã cài đặt xong thao tác Up/Down Heap, thì các phép toán khác của hàng đợi ưu tiên có thể cài đặt khá dễ dàng:

Phép chèn $Insert(i)$ sẽ kiểm tra i nằm trong $Heap$ chưa, nếu chưa, chúng ta đưa i vào cuối mảng $Items$ tương ứng với một nút lá và thực hiện phép $UpHeap(i)$

```
procedure Insert(i: Integer);
begin
  with Heap do
    if Pos[i] = 0 then //i chưa có trong Heap
      begin
        nItems := nItems + 1; //Tăng số phần tử lên 1
        Items[nItems] := i; //Đưa i vào cuối mảng, tương ứng với một nút lá của binary heap
        Pos[i] := nItems; //Cập nhật mảng quản lý vị trí
        UpHeap(i);
      end;
end;
```

Phép Get chỉ đơn thuần trả về giá trị phần tử chứa trong gốc của $Heap$

```
function Get: Integer;
begin
  Result := Heap.Items[1];
end;
```

Phép Extract trả về phần tử chứa trong gốc của $Heap$, đưa phần tử cuối mảng $Items$ về vị trí 1 (đè lên phần tử đang chứa ở gốc), giảm số phần tử đi 1 và thực hiện phép $DownHeap(Items[1])$ nếu $Heap \neq \emptyset$

```
function Extract: Integer;
begin
  with Heap do
    begin
      Result := Items[1]; //Trả về phần tử ở gốc Heap
      Items[1] := Items[nItems]; //Đưa phần tử cuối Heap về thế chỗ
      Pos[Items[1]] := 1; //Cập nhật mảng quản lý vị trí
      Pos[Result] := 0;
      nItems := nItems - 1; //Giảm số phần tử đi 1
      if nItems > 1 then DownHeap(Items[1]); //Nếu Heap còn phần tử thì thực hiện DownHeap
    end;
end;
```

Phép cập nhật $Update(i, newp)$ tùy theo $newp$ ưu tiên hơn hay kém mức độ ưu tiên của i sẽ thực hiện phép $UpHeap$ hay $DownHeap$ nếu i là phần tử đang nằm trong $Heap$

```
procedure Update(i: Integer; newp: Integer);
begin
  if newp > p[i] then //nếu tăng độ ưu tiên của i
    begin
      p[i] := newp;
      if Heap.Pos[i] <> 0 then UpHeap(i)
    end
  else //Nếu không
    begin
      p[i] := newp;
      if Heap.Pos[i] <> 0 then DownHeap(i);
    end;
end;
```

Có thể thấy rằng các phép $Insert$, $Extract$ và $Update$ đều có thời gian thực hiện $O(\lg nItems) = O(\lg n)$ vì chúng đều dựa trên phép Up/Down Heap.

10.8. Thuật toán sắp xếp kiểu trộn (Merge Sort)

Thuật toán sắp xếp kiểu trộn (Merge Sort hay Collation Sort) là một trong những thuật toán sắp xếp cổ điển nhất, được đề xuất bởi John von Neumann năm 1945. Cho tới nay, người ta vẫn coi Merge Sort là một thuật toán sắp xếp ngoài mẫu mực, được đưa vào giảng dạy rộng rãi và được tích hợp trong nhiều phần mềm thương mại.

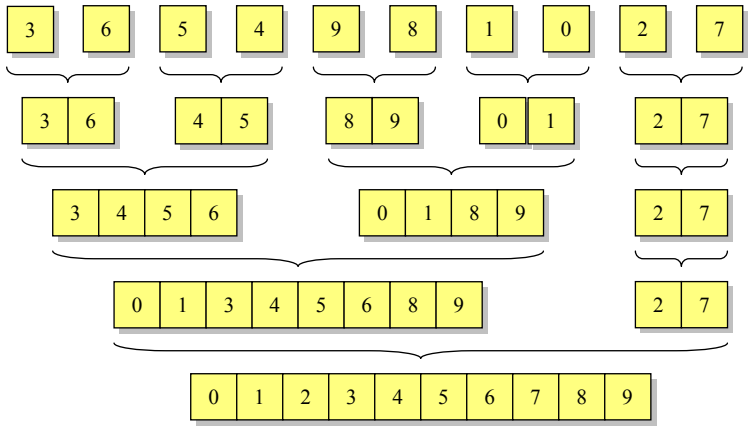
10.8.1. Phép trộn 2 đường

Phép trộn 2 đường là phép hợp nhất hai dãy khóa đã sắp xếp để ghép lại thành một dãy khóa có kích thước bằng tổng kích thước của hai dãy khóa ban đầu và dãy khóa tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện của nó khá đơn giản: Dùng một dãy khóa

phụ có kích thước bằng tổng kích thước hai dãy khóa banđ ầu gọi là miền sắp xếp, miền sắp xếp được tổ chức dưới dạng hàng đợi. Thuật toán so sánh hai khóa đứng đầu hai dãy, chọn ra khóa nhỏ nhất và đưa nó vào miền sắp xếp. Sau đó, khóa này bị loại ra khỏi dãy khóa chứa nó. Quá trình tiếp tục cho tới khi một trong hai dãy khóa đã cạn, khi đó chỉ cần chuyển toàn bộ dãy khóa còn lại ra miền sắp xếp. Khi thuật toán kết thúc, miền sắp xếp sẽ chứa tất cả các phần tử của hai dãy khóa theo thứ tự tăng dần.

10.8.2. Sắp xếp bằng trộn hai đường trực tiếp

Ta có thể coi mỗi khóa trong dãy khóa $k[1 \ldots n]$ là một mạch với độ dài 1, đ ĩ nhiên các mạch độ dài 1 có thể coi là đã đư ợc sắp. Nếu trộn hai mạch liên tiếp lại thành một mạch có độ dài 2, ta lại được dãy gồm các mạch đã đư ợc sắp. Cứ tiếp tục như vậy, số mạch trong dãy sẽ giảm dần sau mỗi lần trộn (Hình 10-8)



Hình 10-8. Thuật toán sắp xếp kiểu trộn

Để tiến hành thuật toán sắp xếp trộn hai đường trực tiếp, ta viết các thủ tục:

Thủ tục $Merge(var\ Source, Dest: TKeyArray;\ a, b, c: Integer)$: Trộn mạch $Source[a \ldots b]$ với mạch $Source[b+1 \ldots c]$ để được mạch $Dest[a \ldots c]$.

Thủ tục $OnePassMergeSort(var\ Source, Dest: TKeyArray; len: Integer)$: Trộn lần lượt các cặp mạch độ dài len liên tiếp trong $Source$ để tạo thành mạch độ dài $2len$ trong $Dest$. Nếu cuối cùng trong $Source$ còn lại hai mạch mà mạch thứ hai có độ dài $< len$, ta sẽ trộn nốt hai mạch này sang $Dest$. Còn nếu cuối cùng trong $Source$ chỉ còn lại một mạch thì ta đưa thẳng mạch duy nhất còn lại đó sang $Dest$.

Cuối cùng là thủ tục MergeSort, thủ tục này cần một dãy khóa phụ $t_{1\ldots n}$. Trước hết ta gọi $OnePassMergeSort(k, t, 1)$ để trộn hai khóa liên tiếp của k thành một mạch trong t, sau đó lại gọi $OnePassMergeSort(k, t, 2)$ để trộn hai mạch liên tiếp trong t thành một mạch trong k, rồi lại gọi $OnePassMergeSort(k, t, 4)$ để trộn hai mạch liên tiếp trong k

thành một mạch trong t …Như vậy k và t được sử dụng với vai trò luân phiên: một dãy chứa các mạch và một dãy dùng để trộn các cặp mạch liên tiếp để được mạch lớn hơn.

```
procedure MergeSort;
var
  Flag: Boolean;
  len: Integer;

  //Trộn Source[a...b] và Source[b + 1...c] thành Dest[a...c]
  procedure Merge(var Source, Dest: TKeyArray; a, b, c: Integer);
  var
    i, j, p: Integer;
  begin
    p := a; i := a; j := b + 1;
    while (i ≤ b) and (j ≤ c) do
      begin
        if Source[i] ≤ Source[j] then
          begin
            Dest[p] := Source[i]; i := i + 1;
          end
        else
          begin
            Dest[p] := Source[j]; j := j + 1;
          end;
        p := p + 1;
      end;
    if i ≤ b then //Mạch Source[a...b] hết trước, đưa phần cuối mạch Source[b + 1...c] vào Dest
      Dest[p...c] := Source[i...b]
    else //Mạch Source[b + 1...c] đã hết, đưa phần cuối mạch Source[a...b] vào Dest
      Dest[p...c] := Source[j...c];
  end;

  procedure OnePassMergeSort(var Source, Dest: TKeyArray; len: Integer);
  var
    a, b, c: Integer;
  begin
    a := 1; b := len; c := len * 2;
    while c ≤ n do
      begin
        Merge(Source, Dest, a, b, c);
        a := a + len * 2;
        b := b + len * 2;
        c := c + len * 2;
      end;
    if b < n then //Còn lại 2 mạch mà mạch thứ hai có độ dài < len
      Merge(Source, Dest, a, b, n)
    else //Chỉ còn lại tối đa 1 mạch
      Dest[a...n] := Source[a...n];
  end;
```

```
begin //Thuật toán sắp xếp kiểu trộn
  len := 1; Flag := True;
  while len < n do
    begin
      if Flag then OnePassMergeSort(k, t, len);
      else OnePassMergeSort(t, k, len);
      len := len * 2;
      Flag := not Flag; //Đảo cờ luân phiên vai trò k và t
    end;
  if not Flag then //Nếu kết quả cuối cùng nằm ở dãy khóa phụ
    k := t; //Chuyển sang dãy khóa chính
end;
```

Về thời gian thực hiện giải thuật, ta thấy rằng trong thủ tục $OnePassMergeSort$ được gọi không quá $\lceil \lg n \rceil$ lần. Trong thủ tục này có thể coi phép toán tích cực là thao tác đưa một khóa vào miền sắp xếp (trong thủ tục Merge). Mỗi lần gọi thủ tục $OnePassMergeSort$, tất cả các khóa trong dãy khóađư ợc chuyển hoàn toàn sang miền sắp xếp, nên thời gian thực hiện thủ tục $OnePassMergeSort$ là $\Theta(n)$. Từ đó suy ra thời gian thực hiện giải thuật Merge Sort là $\Theta(n \lg n)$ bất chấp trạng thái dữ liệu vào.

Về chi phí bộ nhớ, Merge Sort cần một lượng bộ nhớ phụ $\Theta(n)$ để làm miền sắp xếp.

Cùng là những thuật toán sắp xếp tổng quát với độ phức tạp trung bình như nhau, nhưng không giống như QuickSort hay HeapSort, Merge Sort có thể dễ dàng cài đặt để sắp xếp các phần tử trong danh sách móc nối.

Người ta còn có thể lợi dụng được trạng thái dữ liệu vào để khiến MergeSort chạy nhanh hơn: ngay từ đầu, ta không coi mỗi khóa của dãy khóa là một mạch mà coi những đoạn đã được sắp trong dãy khóa là một mạch. Bởi một dãy khóa bất kỳ có thể coi là gồm các mạch đã sắp xếp nằm liên tiếp nhau. Khi đó người ta gọi phương pháp này là phương pháp *trộn hai đường tự nhiên (Natural Merge Sort)***.**

Tổng quát hơn nữa, thay vì phép trộn hai mạch, người ta có thể sử dụng phép trộn k mạch, khi đó ta được thuật toán sắp xếp trộn k đường.

10.9. Một vài đặc trưng của thuật toán sắp xếp

Chúng ta tạm dừng bàn về các thuật toán sắp xếp để xét vài đặc trưng của chúng. Có hai tính chất đối với một thuật toán sắp xếp:

- *Stability*: Tính ổn định: Một thuật toán sắp xếp được gọi là ổn định nếu nó bảo toàn thứ tự ban đầu của các bản ghi mang khóa sắp xếp bằng nhau trong danh sách. Ví dụ như ban đầu danh sách sinh viên được xếp theo thứ tự tên alphabet, thì khi sắp xếp danh sách sinh viên theo thứ tự giảm dần của điểm thi, những sinh viên bằng điểm nhau sẽ được dồn về một đoạn trong danh sách và vẫn được giữ nguyên thứ tự tên alphabet. Hãy xem lại nhưng thuật toán sắp xếp ở trước, trong những thuật toán đó,

thuật toán sắp xếp nổi bọt, thuật toán sắp xếp kiểu chèn và thuật toán sắp xếp kiểu trộn là những thuật toán sắp xếp ổn định, còn những thuật toán sắp xếp khác (và nói chung những thuật toán sắp xếp đòi hỏi phải đảo giá trị 2 bản ghi ở vị trí bất kỳ) là không ổn định. Nói chung dù một thuật toán sắp xếp không ổn định thì ta có thể chuyển nó thành ổn định bằng cách thêm cho mỗi bản ghi một khóa thứ cấp là số thứ tự ban đầu của bản ghi đó, khi hai bản ghi có khóa (sơ cấp) bằng nhau thì khóa thứ cấp sẽ được sử dụng như một chỉ số phụ để quyết định khóa nào đứng trước. Tuy nhiên nếu làm như vậy, chúng ta sẽ mất đi tính chất thứ hai:

- *In-place*: Không gian nhớ phụ mà thuật toán yêu cầu thêm không phụ thuộc vào n. Trong các thuật toán mà chúng ta đã khảo sát thì QuickSort và Merge Sort không có tính chất này còn các thuật toán khác đều là thuật toán sắp xếp In-place.

Sau khi khảo sát một số thuật toán sắp xếp, chúng ta có thể học thêm một cách nghĩ để tiếp cận bài toán: Khi một thuật toán đã có không đạt yêu cầu về một mặt nào đó, ta có thể cải tiến nó bằng cách hy sinh một đặc tính "tốt" mà ta không cần đến để đạt được mục đích. Ví dụ như ShellSort cải tiến từ Insertion Sort nhưng hy sinh tính ổn định để đạt được tốc độ cao hơn.

Chính xuất phát từ cách nghĩ này mà có tác giả [**13**] đã hy sinh tính ổn định của Bubble Sort và dùng cách làm mịn dãy tương tự như ShellSort để được thuật toán Comb Sort có thời gian thực hiện giải thuật trong trường hợp xấu nhất chỉ là $O(n \lg n)$. Dưới đây là mô hình cài đặt của Comb Sort. Việc chứng minh tính đúng đắn và đánh giá thời gian thực hiện giải thuật xin dành cho bạn đọc.

```
procedure CombSort;
var
  i, step:Integer;
  NoSwap: Boolean;
begin
  step := n;
  repeat
    if Step > 1 then
      begin
        Step := Step * 10 div 13;
        if (Step = 10) or (Step = 9) then
          Step := 11;
      end;
    NoSwap := True;
    for i := Step + 1 to n do
      if k[i] < k[i - Step] then
        begin
          Swap(k[i], k[i - Step]);
          NoSwap := False;
        end;
  until (step = 1) and NoSwap;
end;
```

Bài tập 10-1.

Viết thuật toán QuickSort không đệ quy

Bài tập 10-2.

Trong mô hình cài đặt QuickSort, phép so sánh $k_j > Pivot$ có thể thay thế bởi $k_j \geq Pivot$, phép so sánh $k_i < Pivot$ có thể thay thế bởi $k_i \leq Pivot$. Việc thay thế này có thể làm giảm số lần dịch chuyển khóa đi chút ít (các khóa bằng $Pivot$ sẽ không bị dịch chuyển). Hãy giải thích tại sao người ta không sử dụng cách này trong thuật toán Randomized QuickSort.

Gợi ý: Tính hiệu quả của QuickSort không phụ thuộc nhiều vào số lần dịch chuyển khóa mà phụ thuộc chủ yếu vào trạng thái phân đoạn tại mỗi bước. Mục đích của Randomized QuickSort là để khó chỉ ra được bộ dữ liệu rơi vào trường hợp xấu. Hãy thử áp dụng sửa đổi trên và khảo sát hoạt động của QuickSort khi sắp xếp một dãy toàn các khóa giống nhau.

Bài tập 10-3.

Cho một danh sách thí sinh gồm n người, mỗi người cho biết tên (xâu ký tự) và điểm thi (số thực), hãy chọn ra m người điểm cao nhất. Giải quyết bằng thuật toán $O(n)$ trong trường hợp xấu nhất.

Bài tập 10-4.

Cho dãy $K = (k_1, k_2, \dots, k_n)$, Tìm giải thuật $O(n)$ để xác định giá trị xuất hiện nhiều hơn $n/2$ lần trong dãy hoặc thông báo rằng không tồn tại giá trị như vậy.

Gợi ý: Nếu một giá trị xuất hiện nhiều hơn $n/2$ lần thì giá trị đó phải là trung vị của dãy.

Bài tập 10-5.

Cho dãy $K = (k_1, k_2, \dots, k_{m\times n})$, Tìm giải thuật $O(n^2 m)$ để liệt kê các giá trị xuất hiện nhiều hơn m lần trong dãy

Gợi ý: Nếu sắp xếp dãy K và chia dãy làm n đoạn độ dài bằng nhau thì mỗi đoạn sẽ có m phần tử. Những giá trị xuất hiện nhiều hơn m lần trong dãy chắc chắn sẽ chỉ nằm trong các vị trí cuối của mỗi đoạn (ở thứ tự thống kê $i \times m$). Việc xác định giá trị cũng như tần suất của các phần tử này mất thời gian $O(n^2 m)$

Bài tập 10-6.

Một hệ thống quản lý hai dãy số $A = (a_1, a_2, \dots, a_n)$ và $B = (b_1, b_2, \dots, b_n)$. Hai dãy số này đều là hoán vị của dãy số $(1,2, \dots, n)$. Bạn không biết về giá trị cụ thể của các phần tử nhưng có thể đưa vào hệ thống một lệnh $Ask(i, j)$ và nhận được câu trả lời a_i lớn hơn, nhỏ hơn, hay bằng b_j. Nhiệm vụ của bạn là lập chương trình tương tác v ới hệ thống để xác định giá trị cụ thể của từng phần tử trong hai dãy số.

Gợi ý: Dùng thuật toán tương tự như QuickSort nhưng chọn chốt trong B để phân đoạn A rồi lại chọn chốt trong A phân đoạn B. Trung bình $\Theta(n \lg n)$.

Bài tập 10-7.

Giả sử chúng ta có một tập S các khóa và một phép toán so sánh $\prec$ giữa các cặp khóa thỏa mãn các điều kiện sau:

- Với $\forall a, b \in S$, phép toán $a \prec b$ cho một giá trị logic ($True$ hoặc $False$)
- Với $\forall a \in S$: $a \prec a = False$
- Với $\forall a, b \in S$: Nếu $a \prec b = True$ thì $b \prec a = False$
- Với $\forall a, b, c \in S$: Nếu $a \prec b = True$ và $b \prec c = True$ thì $a \prec c = True$

Ta có thể **tạm** gọi tính chất thứ nhất là tính phổ biến, tính chất thứ hai là tính bất phản xạ, tính chất thứ ba là tính phản đối xứng và tính chất thứ tư là tính chất bắc cầu.

Chỉ ra rằng tất cả các thuật toán trong bài vẫn hoạt động tốt nếu ta chỉ có quan hệ $\prec$ thay vì quan hệ thứ tự toàn phần.

Bài 11. Sắp xếp trên các dãy khóa số

Chúng ta đã khảo sát một vài thuật toán sắp xếp tổng quát, những thuật toán này chỉ dựa vào phép so sánh hai khóa để xác định thứ tự các bản ghi. Chúng không cần biết đến giá trị khóa mà chỉ cần biết đến quan hệ thứ tự toàn phần trên tập các khóa là có thể thực hiện việc sắp xếp. Chính vì vậy, về mặt kỹ thuật, có thể gọi các thuật toán này là các *thuật toán sắp xếp chỉ dựa trên phép so sánh*, gọi tắt là *thuật toán sắp xếp so sánh.*

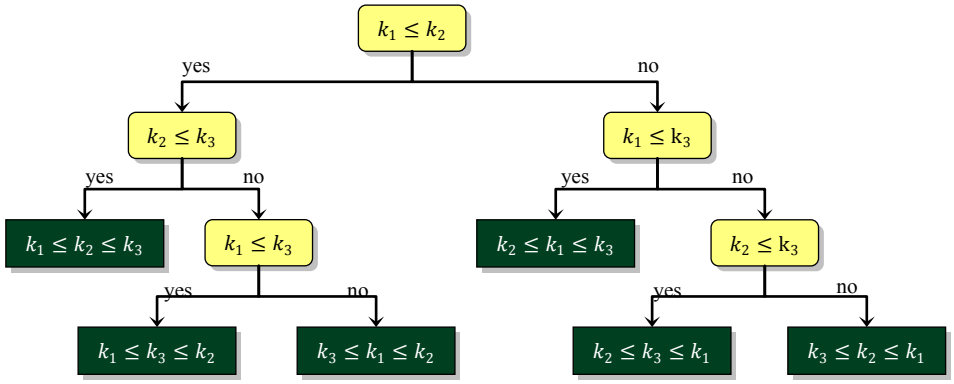
Có những thuật toán sắp xếp thực hiện trong thời gian $\Theta(n \lg n)$ như QuickSort*, HeapSort, Merge Sort; trong khi đó có những thuật toán khác thực hiện trong thời gian $\Theta(n^2)$ như Selection Sort, Bubble Sort. Có thể hiểu con số đánh giá lý thuyết này một cách ước lệ: Cho dù khi cài đặt thực tế QuickSort có nhanh hơn HeapSort thì cũng chỉ nhanh hơn không quá một hằng số C lần, nhưng QuickSort sẽ nhanh hơn Selection Sort "vô hạn lần", vì với một hằng số C lớn tùy ý, ta có thể chỉ ra một bộ dữ liệu (đủ lớn) để QuickSort nhanh hơn Selection Sort nhiều hơn C lần.

Một câu hỏi rất tự nhiên được đặt ra là liệu có tồn tại một thuật toán nhanh hơn QuickSort "vô hạn lần" hay không?

Câu trả lời là Không mà Có…

11.1. Rào cản của các thuật toán sắp xếp so sánh

Cơ chế sắp xếp dựa trên so sánh có thể diễn tả một cách trừu tượng bằng mô hình cây quyết định (*decision tree*) [**8**]. Cây quyết định là một cây nhị phân có các nút nhánh tương ứng với một phép so sánh và các nút lá tương ứng với một quyết định cuối cùng về thứ tự các khóa.



Hình 11-1. Cây quyết định với cơ chế sắp xếp dựa trên so sánh.

* QuickSort với phép chọn trung vị $O(n)$

Hình 11-1 là ví dụ về một cây quyết định tương ứng với bài toán sắp xếp 3 khóa (k_1, k_2, k_3): Đầu tiên ta kiểm tra $k_1 \le k_2$, nếu đúng thì ta kiểm tra tiếp $k_2 \le k_3$, nếu kết quả cũng là đúng thì thứ tự sắp xếp đúng là (k_1, k_2, k_3), còn nếu $k_2 > k_3$ thì tùy theo kết quả so sánh $k_1 \le k_3$ đúng hay sai, ta sẽ có kết luận thứ tự sắp xếp đúng là (k_1, k_3, k_2) hay (k_3, k_1, k_2)…

Nếu các giá trị khóa trong dãy $K = k_{1\ldots n}$ là hoàn toàn phân biệt thì có tất cả $n!$ hoán vị của dãy khóa. Thuật toán sắp xếp phải thực hiện đúng với mọi tình trạng ban đầu của dãy khóa K nên mỗi hoán vị đều có khả năng là một thứ tự sắp xếp đúng. Để đưa ra một quyết định cuối cùng về thứ tự của các khóa tương ứng với một nút lá của cây quyết định, thuật toán luôn phải thực hiện một loạt phép so sánh dọc trên đường đi từ nút gốc tới nút lá đó.

Thứ tự so sánh trên cây quyết định tùy thuộc vào thuật toán sắp xếp, chẳng hạn có thuật toán thực hiện phép so sánh $k_1 \le k_2$ trước, nhưng một thuật toán khác lại thực hiện phép so sánh $k_1 \le k_3$ trước. Nhưng theo lập luận trên, với mọi thuật toán sắp xếp và mọi thứ tự so sánh trên cây quyết định, cây này chắc chắn có $n!$ nút lá.

Ta đã biết rằng một cây nhị phân có độ cao h thì ở độ sâu 0 có 1 nút là nút gốc, ở độ sâu 1 có tối đa 2 nút, ở độ sâu 2 có tối đa 4 nút, tổng quát, ở độ sâu h, có tối đa 2^h nút. Suy ra cây nhị phân có độ cao h thì không có quá 2^h nút lá. Nhìn theo cách ngược lại, cây có m nút lá thì phải có độ cao không ít hơn $\lfloor \lg m \rfloor$. Cây quyết định có tổng cộng $n!$ nút lá, và như vậy chiều cao của cây không ít hơn $\lfloor \lg(n!) \rfloor$.

Trong tất cả các khả năng của dữ liệu vào, sẽ có trường hợp xấu nhất theo nghĩa: để đưa ra quyết định cuối cùng, thuật toán phải đi từ nút gốc tới nút lá theo một đường đi dài bằng chiều cao của cây. Điều này chỉ ra rằng mọi thuật toán sắp xếp so sánh đều phải thực hiện không ít hơn $\lfloor \lg(n!) \rfloor$ phép so sánh trong trường hợp xấu nhất. Nói cách khác, mọi thuật toán sắp xếp so sánh đều có thời gian thực hiện giải thuật là $\Omega(\lg(n!))$ trong trường hợp xấu nhất.

Tốc độ tăng của hàm $\lg(n!)$ có thể ước lượng qua công thức xấp xỉ Sterling:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

khi n đủ lớn. Tức là $\Omega(\lg(n!)) = \Omega(n \lg n)$.*

---

* Thực ra chứng minh trực tiếp $\lg(n!) = \Omega(n \lg n)$ cũng không khó. Có thể thấy rằng $n!$ là tích của n phần tử mà ít nhất một nửa trong số đó có giá trị $\ge n/2$. Vậy $n! \ge (n/2)^{n/2}$, tức là $\lg(n!) \ge n/2 \lg(n/2) = \frac{1}{2} n \lg n - \frac{1}{2} n = \Omega(n \lg n)$

Định lý 11-1

Mọi thuật toán sắp xếp so sánh đều có thời gian thực hiện $\Omega(n \lg n)$ trong trường hợp xấu nhất.

Có thể có suy nghĩ rằng tuy mọi thuật toán sắp xếp so sánh đều có thời gian thực hiện $\Omega(n \lg n)$ trong trường hợp xấu nhất nhưng vẫn có thể tồn tại một thuật toán so sánh có thời gian thực hiện trung bình là $O(n)$ hoặc gì đó "nhỏ hơn hẳn" $n \lg n$ (tức là $o(n \lg n)$). Câu trả lời là không.

Số phép so sánh trung bình có thể tính bằng độ sâu trung bình của các nút lá trên cây quyết định. Ta chứng minh rằng độ sâu trung bình của các nút lá trên cây gồm m lá thì không thể nhỏ hơn $\lg m$. Rõ ràng điều này đúng với $m = 1$ (cây nhị phân suy biến). Giả thiết quy nạp rằng điều này đúng với $\forall m' < m$. Xét cây gồm m lá. Nếu nút gốc của nó chỉ có một nhánh con mọi lá sẽ nằm trong nhánh con đó và độ sâu trung bình của các nút lá sẽ là $\lg m + 1 > \lg m$. Nếu nút gốc của nó có hai nhánh con với số lá lần lượt là x và $m - x$ thì xét trên nhánh con thứ nhất, độ sâu trung bình của các lá $\geq \lg x$ và trên nhánh con thứ hai, độ sâu trung bình của các lá $\geq \lg(m - x)$ theo giả thiết quy nạp. Tức là độ sâu trung bình của các lá trên cây chính sẽ không ít hơn:

$$h(x) = \frac{x \lg x + (m - x) \lg(\mathrm{m} - \mathrm{x})}{m} + 1$$

Dùng đạo hàm và khảo sát hàm số $h(x)$, hàm này đạt cực tiểu tại $x = m/2$ và tại điểm cực tiểu, $h(x) = \lg m$. Suy ra trung bình số phép so sánh của bất kỳ thuật toán sắp xếp so sánh nào cũng không ít hơn $\lg(n!) = \Omega(n \lg n)$.

Định lý 11-2

Mọi thuật toán sắp xếp so sánh đều có thời gian thực hiện trung bình $\Omega(n \lg n)$.

Đến đây chúng ta nhận thấy rằng nếu không tận dụng được thông tin gì khác ngoài quan hệ thứ tự toàn phần của các khóa thì không thể bẻ gãy rào cản $\Omega(n \lg n)$. Để vượt qua giới hạn tốc độ này cần phải tận dụng những thông tin khác nữa, và một trong những cách tiếp cận là sử dụng chính giá trị khóa.

11.2. Sắp xếp bằng phép đếm phân phối (Counting Sort)

Có một thuật toán sắp xếp đơn giản cho trường hợp đặc biệt: Dãy khóa $K = k_{1\ldots n}$ chứa các khóa là các số nguyên nằm trong khoảng từ 0 tới m. (TKey = 0..m)

Ta dựng dãy $c_{0\ldots m}$ gồm các biến đếm, ở đây c_v là số lần xuất hiện giá trị v trong dãy khóa K:

```
for v := 0 to m do c[v] := 0; //Khởi tạo dãy biến đếm
for i := 1 to n do c[k[i]] := c[k[i]] + 1;
```

Ví dụ với dãy khóa: (1,2,2,3,0,0,1,1,3,3) $(n = 10, m = 3)$ sau bước đếm ta có:

$$c_0 = 2; c_1 = 3; c_2 = 2; c_3 = 3$$

Dựa vào dãy biến đếm, ta hoàn toàn có thể biết được: sau khi sắp xếp thì giá trị v phải nằm từ vị trí nào tới vị trí nào. Như ví dụ trên thì giá trị 0 phải nằm từ vị trí 1 tới vị trí 2; giá trị 1 phải đứng liên tiếp từ vị trí 3 tới vị trí 5; giá trị 2 đứng ở vị trí 6 và 7 còn giá trị 3 nằm ở ba vị trí cuối 8, 9, 10:

$$(0,0,1,1,1,2,2,3,3,3)$$

Tức là sau khi sắp xếp:

Giá trị 0 đứng trong đoạn từ vị trí 1 tới vị trí c_0.

Giá trị 1 đứng trong đoạn từ vị trí $c_0 + 1$ tới vị trí $c_0 + c_1$.

Giá trị 2 đứng trong đoạn từ vị trí $c_0 + c_1 + 1$ tới vị trí $c_0 + c_1 + c_2$.

…

Giá trị v trong đoạn đứng từ vị trí $\sum_{x=0}^{v-1} c_x + 1$ tới vị trí $\sum_{x=0}^{v} c_x$.

…

Để ý vị trí cuối của mỗi đoạn, nếu ta tính lại dãy c như sau:

```
for v := 1 to m do c[v] := c[v - 1] + c[v];
```

Thì c_v sẽ là vị trí cuối của đoạn chứa giá trị v trong dãy khóa đã sắp xếp.

Muốn dựng lại dãy khóa sắp xếp, ta thêm một dãy khóa phụ $t_{1\ldots n}$. Sau đó duyệt lại dãy khóa $k_{1\ldots n}$, mỗi khi gặp khóa mang giá trị v ta đưa giá trị đó vào $t[c_v]$ và giảm c_v đi 1. Sau quá trình duyệt, $t_{1\ldots n}$ là dãy khóa đã sắp.

```
for i := n downto 1 do
  begin
    v := k[i];
    t[c[v]] := v;
    c[v] := c[v] - 1;
  end;
```

Đó là toàn bộ tư tưởng của thuật toán sắp xếp bằng phép đếm phân phối (Counting Sort):

```
procedure CountingSort;
var
  c: array[0..m] of Integer; //Dãy biến đếm
  t: TKeyArray; //Dãy khóa phụ
  i: Integer;
  v: TKey;
begin
  for v := 0 to m do c[v] := 0; //Khởi tạo dãy biến đếm
  for i := 1 to n do c[k[i]] := c[k[i]] + 1; //Đếm
  for v := 1 to m do c[v] := c[v - 1] + c[v]; //Tính vị trí cuối mỗi đoạn
  for i := n downto 1 do //Phân phối
    begin
      v := k[i];
      t[c[v]] := v;
      c[v] := c[v] - 1;
    end;
  k := t; //Chép kết quả từ dãy khóa phụ sang dãy khóa chính
end;
```

Rõ ràng thời gian thực hiện giải thuật là $\Theta(n+m)$. Trên thực tế, người ta thường áp dụng Counting Sort khi mà $m = O(n)$*, khi đó thời gian thực hiện có thể viết là $O(n)$.

Counting Sort phá được rào cản $\Omega(n \lg n)$ vì nó không phải thuật toán sắp xếp so sánh, không có một phép so sánh nào cần thực hiện trong thuật toán. Một điều đáng lưu ý của Counting Sort là nó có tính ổn định, hãy nhìn vào mô hình cài đặt, nếu hai bản ghi có khóa sắp xếp bằng nhau thì khi đưa giá trị vào dãy bản ghi phụ, bản ghi nào vào trước sẽ nằm phía sau. Vậy nên ta sẽ đẩy giá trị các bản ghi vào dãy phụ theo thứ tự ngược lại để giữ được thứ tự tương đối ban đầu (nếu khóa sắp xếp bằng nhau, bản ghi nào đang đứng trước vẫn đứng trước).

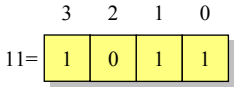
Trong một số bài toán, tính ổn định của thuật toán sắp xếp quyết định tới cả tính đúng đắn của toàn thuật toán lớn. Chính tính "nhanh" của QuickSort và tính ổn định của phép đếm phân phối là cơ sở nền tảng cho hai thuật toán sắp xếp cực nhanh trên các dãy khóa số mà ta sẽ trình bày dưới đây.

11.3. Thuật toán sắp xếp cơ số (Radix Sort)

11.3.1. MSD Radix Sort

Khi dãy khóa là các số tự nhiên, ta có thể coi mỗi khóa là một dãy z bit đánh số từ bit 0 (bit đơn vị) tới bit $z-1$ (bit cao nhất) (Hình 11-2).

* Thực ra thì Couting Sort chỉ hay được sử dụng khi mà $m \ll n$

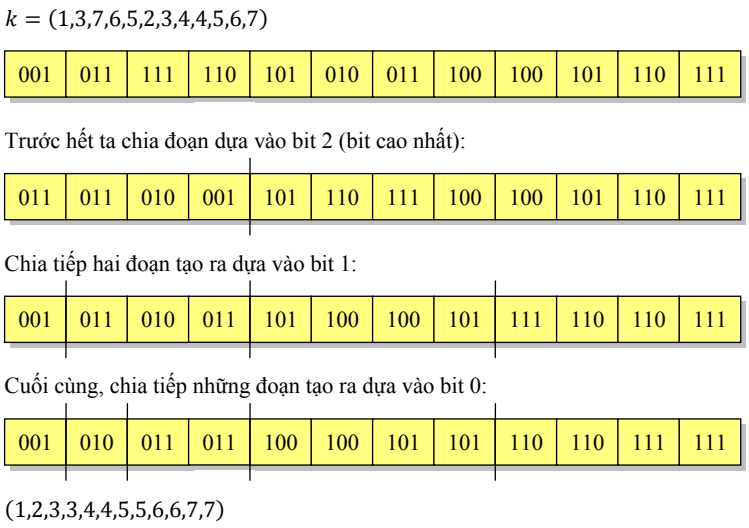


Hình 11-2. Đánh số các bit

Thuật toán sắp xếp cơ số từ bit cao nhất (*most significant digit (MSD) radix sort*) thực hiện như sau: Vì mỗi khóa đều là dãy z bit nên những khóa bắt đầu bằng bit 0 chắc chắn phải nhỏ hơn những khóa bắt đầu bằng bit 1. Vì vậy trước hết thuật toán sẽ đưa tất cả những khóa có bit $z-1$ là 0 về đầu dãy, đưa những khóa có bit $z-1$ là 1 về cuối dãy và phân dãy khóa làm hai đoạn. Tiếp tục quá trình phân đoạn với từng đoạn, ta thấy với những khóa thuộc cùng một đoạn thì có bit cao nhất giống nhau, nên ta có thể áp dụng quá trình phân đoạn tương tự trên theo bit thứ $z-2$ và cứ tiếp tục như vậy …

Quá trình phân đoạn kết thúc nếu như đoạn đang xét là rỗng hay ta đã tiến hành phân đoạn đến tận bit đơn vị, tức là tất cả các khóa thuộc một trong hai đoạn mới tạo ra đều có bit đơn vị bằng nhau (điều này đồng nghĩa với sự bằng nhau ở tất cả những bit khác, tức là bằng nhau về giá trị khóa).

MSD Radix Sort làm việc khá giống QuickSort, chỉ khác là việc phân đoạn dựa vào một bit trong khóa chứ không dựa vào việc so sánh với giá trị khóa chốt.



Hình 11-3. MSD Radix Sort

```
procedure MSDRadixSort;
var
  z: Integer;
```

```
procedure Partition(L, H: Integer; b: Integer); //Phân đoạn k[L...H] dựa vào bit b
var
  i, j: Integer;
  temp: TKey;
begin
  if L >= H then Exit;
  i := L; j := H;
  temp := k[L]; //sao lưu khóa đầu đoạn, tạo chỗ trống tại vị trí đầu đoạn
  repeat
    while («bit b của k[j] là 1») and (i < j) do j := j - 1; //Lùi j để tìm khóa có bít b = 0
    if i < j then //Đưa khóa tìm được vào chỗ trống
      begin
        k[i] := k[j]; Inc(i);
      end;
    while («bit b của k[i] là 0») and (i < j) do i := i - 1; //Tiến i để tìm khóa có bít b = 1
    if i < j then //Đưa khóa tìm được vào chỗ trống
      begin
        k[j] := k[i];
        Dec(j);
      end;
  until i = j;
  k^[i] := temp; //đưa khóa sao lưu vào khoảng trống
  if b > 0 then
    begin
      if «bit b của temp là 1» then i := i - 1; //Tính vị trí phân đoạn
      Partition(L, i, b - 1); Partition(i + 1, H, b - 1);
    end;
end;

begin
  «Dựa vào giá trị lớn nhất, xác định z là số bit cần dùng để biểu diễn một khóa»;
  Partition(1, n, z - 1);
end;
```

Với MSD Radix Sort, ta hoàn toàn có thể làm trên hệ cơ số R khác chứ không nhất thiết phải làm trên hệ nhị phân (ý tưởng cũng tương tự như trên), tuy nhiên quá trình phân đoạn sẽ không phải chia làm 2 mà chia thành R đoạn. Về độ phức tạp của thuật toán, ta thấy để phân đoạn bằng một bit thì thời gian sẽ là cn để chia tất cả các đoạn cần chia bằng bit đó (c là hằng số). Vậy tổng thời gian phân đoạn bằng z bit sẽ là cnz. Trong trường hợp xấu nhất, thời gian thực hiện giải thuật là $\Theta(nz)$. Và trung bình thời gian thực hiện của MSD Radix Sort là $O(n \min(z, \lg n))$.

Nói chung, MSD Radix Sort cài đặt như trên chỉ thể hiện tốc độ tối đa trên các hệ thống cho phép xử lý trực tiếp trên các bit: Hệ thống phải cho phép lấy một bit ra dễ dàng và thực hiện với thời gian nhanh hơn hẳn so với các thao tác trên BYTE, WORD, DWORD, QWORD... Khi đó MSD Radix Sort sẽ tốt hơn nhiều QuickSort. (Ta thử lập trình sắp xếp các dãy nhị phân độ dài z theo thứ tự từ điển để khảo sát). Trên các máy tính hiện nay chỉ cho phép xử lý trực tiếp trên BYTE (hay WORD, DWORD v.v…), việc tách một bit ra

khỏi Byte đó để xử lý lại rất chậm và làm ảnh hưởng không nhỏ tới tốc độ của thuật toán. Chính vì vậy, tuy đây là một phương pháp hay, nhưng khi cài đặt cụ thể thì tốc độ cũng chỉ ngang ngửa chứ không thể qua mặt QuickSort được.

11.3.2. LSD Radix Sort

Ta sẽ trình bày ý tưởng chính của thuật toán sắp xếp cơ số bắt đầu từ bit thấp nhất (*Least Significant Digit (LSD) Radix Sort*) qua một ví dụ:

Sắp xếp dãy khóa:

925	817	821	638	639	744	742	563	570	166

Trước hết, ta sắp xếp dãy khóa này theo thứ tự tăng dần của chữ số hàng đơn vị bằng một thuật toán sắp xếp khác, được dãy khóa:

570	821	742	563	744	925	166	817	638	639

Sau đó, ta sắp xếp dãy khóa mới tạo thành theo thứ tự tăng dần của chữ số hàng chục bằng một thuật toán sắp xếp *ổn định*, được dãy khóa:

817	821	925	638	639	742	744	563	166	570

Vì thuật toán sắp xếp ta sử dụng là ổn định, nên nếu hai khóa có chữ số hàng chục giống nhau thì khóa nào có chữ số hàng đơn vị nhỏ hơn sẽ đứng trước. Nói như vậy có nghĩa là dãy khóa thu được sẽ có thứ tự tăng dần về giá trị tạo thành từ hai chữ số cuối.

Cuối cùng, ta sắp xếp lại dãy khóa theo thứ tự tăng dần của chữ số hàng trăm cũng bằng một thuật toán sắp xếp ổn định, thu được dãy khóa:

166	563	570	638	639	742	744	817	821	925

Lập luận tương tự như trên dựa vào tính ổn định của phép sắp xếp, dãy khóa thu được sẽ có thứ tự tăng dần về giá trị tạo thành bởi cả ba chữ số, đó là dãy khóa đã sắp.

(Có thể coi số chữ số của mỗi khóa là bằng nhau, như ví dụ trên nếu có số 15 trong dãy khóa thì ta có thể coi nó là 015)

Cũng từ ví dụ, ta có thể thấy rằng số lượt sắp xếp phải thực hiện đúng bằng số chữ số tạo thành một khóa. Với một hệ cơ số lớn, biểu diễn một giá trị khóa sẽ phải dùng ít chữ số hơn. Chẳng hạn số 12345 trong hệ thập phân phải dùng tới 5 chữ số, còn trong hệ cơ số 1000 chỉ cần dùng 2 chữ số AB mà thôi, ở đây A là chữ số mang giá trị 12 còn B là chữ số mang giá trị 345.

Tốc độ của sắp xếp cơ số trực tiếp phụ thuộc rất nhiều vào thuật toán sắp xếp ổn định tại mỗi bước. Không có một lựa chọn nào khác tốt hơn phép đếm phân phối. Tuy nhiên, phép đếm phân phối có thể không cài đặt được hoặc kém hiệu quả nếu như tập giá trị

khóa quá rộng, không cho phép dựng ra dãy các biến đếm hoặc phải sử dụng dãy biến đếm quá dài (Điều này xảy ra nếu chọn hệ cơ số quá lớn).

Một lựa chọn khôn ngoan là nên chọn hệ cơ số thích hợp cho từng trường hợp cụ thể để dung hòa tới mức tối ưu nhất ba mục tiêu:

- Việc lấy ra một chữ số của một số được thực hiện dễ dàng
- Sử dụng ít lần gọi phép đếm phân phối.
- Phép đếm phân phối thực hiện nhanh

```
procedure LSDRadixSort;
const
  Radix = ...; //Tự chọn hệ cơ số
var
  p, z: Integer; //z là số chữ số cho 1 khóa, các chữ số được đánh chỉ số từ 0 tới z-1 từ hàng đơn vị lên
  Flag: Boolean;
  t: TKeyArray; //Dãy khóa phụ

  function GetDigit(key: TKey; p: Integer): TKey; //Lấy chữ số thứ p của key
  begin
    Result := key div Radix^p mod Radix; //Trường hợp cụ thể cần chọn Radix để có mẹo viết tốt hơn
  end;

  //Thuật toán đếm phân phối: sắp xếp ổn định dãy x theo chữ số p, ghi kết quả sang y
  procedure DCount(var x, y: TKeyArray; p: Integer);
  var
    i: Integer;
    d: TKey;
    c: array[0..Radix - 1] of Integer;
  begin
    for d := 0 to Radix - 1 do c[d] := 0; //Khởi tạo dãy biến đếm
    for i := 1 to n do //Đếm
      begin
        d := GetDigit(x[i], p);
        c[d] := c[d] + 1;
      end;
    for d := 1 to Radix - 1 do c[d] := c[d - 1] + c[d]; //Tính vị trí cuối mỗi đoạn
    for i := n downto 1 do //Phân phối, điền giá trị sang dãy y
      begin
        d := GetDigit(x[i], p);
        y[c[d]] := x[i];
        c[d] := c[d] - 1;
      end;
  end;
```

```
begin //LSD Radix Sort
  «Dựa vào giá trị lớn nhất trong dãy khóa, xác định z là số chữ số của một khóa»
  Flag := True;
  for p := 0 to z - 1 do
    begin //Sắp xếp từ hàng đơn vị lên
      if Flag then DCount(k, t, p);
      else DCount(t, k, p);
      Flag := not Flag; //Đảo cờ, luân phiên vai trò k và t
    end;
  if not Flag then k := t; //Kết quả cuối cùng đang nằm ở t thì sao chép sang k
end;
```

Ta đã biết thời gian thực hiện của phép đếm phân phối là $\Theta(n + Radix)$. Mà $Radix$ là một hằng số tự ta chọn từ trước, nên khi n lớn, thời gian thực hiện một phép đếm phân phối là $\Theta(n)$. Thuật toán sử dụng z lần phép đếm phân phối nên có thể thấy thời gian thực hiện giải thuật LSD Radix Sort là $\Theta(nz)$ bất kể tình trạng dữ liệu đầu vào.

Ta có thể coi sắp xếp cơ số trực tiếp là một mở rộng của phép đếm phân phối, khi dãy số chỉ toàn các số có 1 chữ số (trong hệ $Radix$) thì đó chính là phép đếm phân phối. Sự khác biệt ở đây là: Sắp xếp cơ số trực tiếp có thể thực hiện với các khóa mang giá trị lớn; còn phép đếm phân phối chỉ có thể làm trong trường hợp các khóa mang giá trị nhỏ, bởi nó cần một lượng bộ nhớ đủ rộng để giăng ra dãy biến đếm số lần xuất hiện cho từng giá trị.

Bài tập 11-1.

Cho biết độ sâu nhỏ nhất của một lá trong cây quyết định của một thuật toán sắp xếp so sánh (tương ứng với số phép so sánh cần thực hiện trong trường hợp tốt nhất)

Bài tập 11-2.

Một hệ thống lưu trữ tập S gồm n số nguyên $\in [0 \dots k]$. Hệ thống nhận được rất nhiều câu hỏi dạng: Có bao nhiêu phần tử của S nằm trong đoạn $[a, b]$. Hãy làm một phép tiền xử lý trên tập S để có thể trả lời mọi câu hỏi nhận được, mỗi câu hỏi trong thời gian $\Theta(1)$.

Bài tập 11-3.

Tự tìm hiểu về thuật toán Bucket Sort, cài đặt và so sánh với các thuật toán khác (có thể sử dụng mã nguồn của chương trình SortDemo ở bài sau).

Bài tập 11-4.

Giả sử bạn có dãy n số nguyên dương, các số nguyên dương được viết dưới dạng thập phân và số lượng các chữ số (của tất cả các phần tử trong dãy) là m. Tìm thuật toán sắp xếp dãy trong thời gian $O(m)$. Lập trình thuật toán đó.

Cách giải: Counting Sort không thể áp dụng được vì nếu dãy chỉ có một số có m chữ số thì thời gian thực hiện sẽ là $\Omega(10^m)$. Radix Sort cũng không áp dụng được vì nếu dãy có

một số có $m/2$ *chữ số và* $m/2$ *số khác có 1 chữ số thì Radix Sort sẽ mất thời gian* $\Omega\left(\left(\frac{m}{2}+1\right)\left(\frac{m}{2}\right)\right) = \Omega(m^2)$

Cách đúng là dùng số chữ số của mỗi phần tử làm khóa để sắp xếp dãy ($\mathrm{O}(m)$*). Với một nhóm phần tử có cùng số chữ số thì tiếp tục dùng Radix Sort sắp xếp nhóm đó (*$\mathrm{O}(i.m_i)$*) .*

Bài tập 11-5.

Bạn được cho n xâu ký tự, tổng số ký tự của các xâu là m. Tìm thuật toán sắp xếp n xâu đã cho theo thứ tự từ điển trong thời gian $\mathrm{O}(m)$. Lập trình thuật toán đó.

Gợi ý: Sửa đổi MSD Radix Sort.

Bài 12. So sánh các thuật toán sắp xếp

12.1. Cài đặt chương trình

Ta sẽ cài đặt tất cả các thuật toán sắp xếp đã khảo sát bằng một chương trình có giao diện dưới dạng menu, mỗi chức năng tương ứng với một thuật toán sắp xếp

Ở thuật toán MSD Radix Sort, ta chọn $z = 2$ (hệ nhị phân). Ở thuật toán LSD Radix Sort, ta chọn $Radix = 256$ (sử dụng hệ cơ số 256), khi đó nếu một giá trị số tự nhiên x biểu diễn bằng d chữ số trong hệ 256: $x = \overline{x_{d-1}\dots x_1 x_0}_{(256)}$ thì

$$\begin{aligned} x_p &= x \text{ div } 256^p \text{ mod } 256 \\ &= x \text{ div } 2^{8p} \text{ mod } 2^8 \\ &= x \text{ shr } (p \text{ shl } 3) \text{ and } \$FF \end{aligned}$$

Mỗi thuật toán sắp xếp sẽ nhận dữ liệu vào là 3 số $n, minV, maxV$ và một giá trị $Seed$. Thuật toán sẽ khởi tạo nhân của bộ tạo số ngẫu nhiên $RandSeed := Seed$ và sinh ra dãy n khóa là số nguyên lấy ngẫu nhiên trong $[minV, maxV]$. Điều này đảm bảo rằng nếu hai thuật toán nhận vào cùng một bộ dữ liệu $(n, minV, maxV, Seed)$, chúng sẽ sinh ra hai dãy khóa đầu vào giống nhau trước khi thực hiện sắp xếp.

Để dễ dàng thêm vào những sửa đổi sau này, mặc dù chúng ta lập trình sắp xếp dãy khóa là các số nguyên, nhưng chúng ta sẽ định nghĩa ba kiểu dữ liệu khác nhau, kiểu $TKey$ là kiểu dữ liệu của một khóa, kiểu $TIndex$ là kiểu chỉ số trong dãy khóa và kiểu $TBitIndex$ là kiểu chỉ số của các bit. Các biến cũng được khai báo theo quy tắc này để nếu muốn sửa các khóa thành kiểu dữ liệu khác thì chỉ cần định nghĩa lại kiểu TKey là đủ.

Bộ sinh dữ liệu có 4 tham số: tham số thứ nhất n là số khóa, tham số thứ hai $MinV$ là giá trị nhỏ nhất, tham số thứ ba $SupV$ là giá trị lớn nhất cộng thêm 1, tham số cuối cùng $Seed$ là nhân của bộ tạo số ngẫu nhiên. Các tham số này có thể nhập vào thông qua thủ tục $Setup$. Dãy khóa sẽ được sinh ra bằng lời gọi thủ tục $CreateKeyArray$: Cấp phát bộ nhớ cho dãy khóa $k^\wedge$ và sinh ngẫu nhiên các phần tử của dãy khóa dựa vào các thiết lập của bộ sinh.

Kiểu thủ tục sắp xếp $TSortProc$ được định nghĩa là một thủ tục không có tham số. Việc thực hiện một thuật toán sắp xếp được cài đặt trong thủ tục $Perform$. Thủ tục này nhận vào một tham số $SortProc$ kiểu $TSortProc$, sinh ra dãy khóa và gọi thủ tục $SortProc$ để thực hiện sắp xếp, ngoài ra có các thao tác đo thời gian, tự kiểm tra và in kết quả…

💻 Sorting Algorithm Demo

```
{$MODE OBJFPC}
program SortDemo;
```

```
uses Windows, SysUtils, Math;

const
  Limit = 100000000;
  nMenu = 15;
  MenuS: array [1..nMenu] of AnsiString = (
  '1: Setup Key Generator',
  '2: View Key Sequence',
  'A: Selection Sort',
  'B: Bubble Sort',
  'C: Cocktail Sort',
  'D: Insertion Sort',
  'E: Shell Sort',
  'F: Comb Sort',
  'G: QuickSort',
  'H: HeapSort',
  'I: Merge Sort',
  'J: Counting Sort',
  'K: MSD Radix Sort',
  'L: LSD Radix Sort',
  '0: Exit'
  );

type
  TKey = type Integer; //Kiểu khóa
  TIndex = type Integer; //Kiểu chỉ số của khóa
  TBitIndex = type Integer; //Kiểu chỉ số các bit
  TKeyArray = array[1..Limit] of TKey;
  PKeyArray = ^TKeyArray;
  TIndexArray = array[0..Limit] of TIndex;
  PIndexArray = ^TIndexArray;

  TSortProc = procedure; //Kiểu thủ tục sắp xếp
var
  n: TIndex;
  MinV, SupV: TKey;
  Seed: Cardinal;
  k: PKeyArray;
  StartTime, FinishTime: Cardinal;

procedure Setup; //Thiết lập các tham số cho bộ sinh dữ liệu
begin
  Write('Number of keys: (1..', Limit, '): ');
  ReadLn(n);
  Write('Min: (0..', Limit, '): ');
  ReadLn(MinV);
  Write('Max: (Min..', Limit, '): ');
  ReadLn(SupV);
  Inc(SupV);
  Randomize;
  Seed := RandSeed;
end;

procedure Pause; //Tạm dừng màn hình, đợi người dùng bấm Enter để tiếp tục
begin
  WriteLn;
```

```
  Write('Press Enter to continue...');
  ReadLn;
end;

procedure DisplayInput; //Hiện dãy khóa do bộ sinh dữ liệu sinh ra
var
  i: TIndex;
begin
  RandSeed := Seed;
  for i := 1 to n do Write(MinV + Random(SupV - MinV): 10);
  Pause;
end;

procedure CreateKeyArray;
var
  i: TIndex;
begin
  k := GetMemory(n * SizeOf(TKey)); //Cấp phát bộ nhớ
  RandSeed := Seed;
  for i := 1 to n do //Sinh dữ liệu
    k^[i] := Random(SupV - MinV) + MinV;
end;

procedure SelfCheck; //Thủ tục dùng để gỡ rối, kiểm tra dãy kết quả có đúng không
var
  i: TIndex;
begin
  Write('Verifying result...');
  for i := 1 to n - 1 do
    if k^[i] > k^[i + 1] then
      begin
        WriteLn('Wrong!');
        Exit;
      end;
  WriteLn('Correct!');
end;

procedure PrintResult; //In kết quả
var
  ans: AnsiChar;
  i: TIndex;
begin
  WriteLn('Running Time: ', FinishTime - StartTime, ' ms');
  {$IFDEF DEBUG} //Nếu chạy ở Debug mode thì kiểm tra kết quả
  SelfCheck;
  {$ENDIF}
  Write('View sorted sequence Y/N?'); ReadLn(ans);
  if UpCase(ans) = 'Y' then //In kết quả nếu được yêu cầu
    begin
      for i := 1 to n do Write(k^[i]: 10);
      Pause;
    end;
end;

//Nhận vào một lớp tương ứng với một thuật toán và chạy thuật toán đó
procedure Perform(SortProc: TSortProc);
begin
```

```
  try
    CreateKeyArray; //Sinh dãy khóa
    try
      //Trước hết in ra vài thông tin về dữ liệu
      WriteLn(Format('Input: %.0n keys in [%.0n...%.0n]',
        [n + 0.0, MinV + 0.0, SupV - 1.0]));
      Write('Sorting...');
      StartTime := GetTickCount; //Bắt đầu tính giờ
      SortProc; //Thực hiện sắp xếp
      FinishTime := GetTickCount; //Đo thời gian
      PrintResult; //In kết quả
    finally
      FreeMemory(k); //Hủy đối tượng
    end;
  except
    on E:exception do //Chặn lỗi nếu có
      WriteLn('An exception has occurred: ', E.message);
  end;
end;

procedure Swap(var x, y: TKey); inline; //Đảo giá trị hai tham biến x, y
var
  temp: TKey;
begin
  temp := x; x := y; y := temp;
end;

//Thuật toán sắp xếp kiểu chọn
procedure SelectionSort;
var
  i, j, jmin: TIndex;
begin
  for i := 1 to n - 1 do //Làm n - 1 lượt
    begin
      //Chọn trong số các khóa trong đoạn k^[i...n] ra khóa k^[jmin] nhỏ nhất
      jmin := i;
      for j := i + 1 to n do
        if k^[j] < k^[jmin] then jmin := j;
      if jmin <> i then //Đưa khóa nhỏ nhất đó về vị trí i
        Swap(k^[jmin], k^[i]);
    end;
end;

//Thuật toán sắp xếp nổi bọt
procedure BubbleSort;
var
  i, j: TIndex;
begin
  for i := 2 to n do
    for j := n downto i do //Duyệt từ cuối dãy lên, làm nổi khóa nhỏ nhất trong đoạn k^[i-1...n] về vị trí i-1
      if k^[j] < k^[j - 1] then
        Swap(k^[j - 1], k^[j]);
end;

//Thuật toán Cocktail Sort
procedure CocktailSort;
var
```

```
  L, H, Bound, i: TIndex;
begin
  L := 1; H := n;
  repeat
    Bound := H;
    for i := H downto L + 1 do //Nhẹ nổi lên
      if k^[i] < k^[i - 1] then
        begin
          Swap(k^[i - 1], k^[i]);
          Bound := i;
        end;
    L := Bound; //Cập nhật vị trí đầu đoạn mới
    for i := L to H - 1 do //Nặng chìm xuống
      if k^[i] > k^[i + 1] then
        begin
          Swap(k^[i], k^[i + 1]);
          Bound := i;
        end;
    H := Bound; //Cập nhật vị trí cuối đoạn mới
  until L >= H;
end;

//Thuật toán sắp xếp kiểu chèn
procedure InsertionSort;
var
  i, j: TIndex;
  temp: TKey;
begin
  for i := 2 to n do
    begin //Chèn giá trị k^[i] vào dãy k^[1...i-1] để toàn đoạn k^[1...i] trở thành đã sắp xếp
      temp := k^[i]; //Lưu giữ lại giá trị k^[i]
      j := i - 1;
      while (j > 0) and (temp < k^[j]) do //So sánh giá trị cần chèn với lần lượt các khóa k^[j] đứng trước
        begin //Nếu khóa k^[j] lớn hơn
          k^[j + 1] := k^[j]; //Đẩy lùi giá trị k^[j] về phía sau một vị trí, tạo ra khoảng trống tại vị trí j
          Dec(j); //Xét tiếp phần tử liền trước
        end;
      k^[j + 1] := temp; //Đưa giá trị chèn vào "khoảng trống" mới tạo ra
    end;
end;

//Thuật toán Shell Sort
procedure ShellSort;
var
  i, j, step: TIndex;
  temp: TKey;
begin
  step := n div 2;
  while step <> 0 do //Làm mịn dãy với độ dài bước step
    begin
      for i := step + 1 to n do
        begin //Sắp xếp chèn trên dãy con k^[i-step], k^[i], k^[i+step], k^[i+2step],...
          temp := k^[i]; j := i - step;
          while (j > 0) and (k^[j] > temp) do
            begin
              k^[j + step] := k^[j];
              j := j - step;
```

```
            end;
          k^[j + step] := temp;
        end;
      if step = 2 then step := 1
      else step := step * 10 div 22;
    end;
end;
```

//Thuật toán Comb Sort

```
procedure CombSort;
var
  i, step:TIndex;
  NoSwap: Boolean;
begin
  step := n;
  repeat
    if step > 1 then
      begin
        step := step * 10 div 13;
        if (step = 10) or (step = 9) then
          step := 11;
      end;
    NoSwap := True;
    for i := step + 1 to n do
      if k^[i] < k^[i - step] then
        begin
          Swap(k^[i], k^[i - step]);
          NoSwap := False;
        end;
  until (step = 1) and NoSwap;
end;
```

//Thuật toán sắp xếp kiểu phân đoạn

```
procedure QuickSort;

  procedure Partition(L, H: TIndex); //Sắp xếp đoạn dãy khóa k[L...H]
  var
    i, j: TIndex;
    Pivot: TKey;
  begin
    if L >= H then Exit;
    i := L + Random(H - L + 1); //Chọn một vị trí bất kỳ trong đoạn
    Pivot := k^[i]; //Lấy khóa chốt
    k^[i] := k^[L]; //Tạo chỗ trống tại vị trí đầu đoạn
    i := L; j := H;
    repeat
      while (k^[j] > Pivot) and (i < j) do Dec(j); //Lùi j để tìm khóa < chốt
      if i < j then
        begin //Đưa khóa tìm được vào chỗ trống  tại vị trí i
          k^[i] := k^[j]; Inc(i);
        end
      else Break;
      while (k^[i] < Pivot) and (i < j) do Inc(i); //Tiến i để tìm khóa > chốt
      if i < j then
        begin //Đưa khóa tìm được vào chỗ trống tại vị trí j
          k^[j] := k^[i]; Dec(j);
        end
```

```
      else Break;
    until i = j;
    k^[i] := Pivot;
    Partition(L, Pred(i));
    Partition(Succ(i), H);
  end;

begin
  Partition(1, n);
end;
```

//Thuật toán sắp xếp kiểu vun đống

```
procedure HeapSort;
var
  i: TIndex;

  procedure Heapify(r, p: TIndex); //Vun cây gốc r thành đống
  var
    temp: TKey;
    c: TIndex;
  begin
    temp := k^[r];
    repeat
      //Từ r tìm  nút con c mang giá trị lớn hơn trong hai nút con
      c := r shl 1;
      if (c < p) and (k^[c] < k^[c + 1]) then Inc(c);
      if (c > p) or (k^[c] <= temp) then Break; //Dừng ngay nếu r không có nút con mang giá trị > temp
      k^[r] := k^[c]; //Đẩy giá trị từ nút con lên nút cha
      r := c; //Đi xuống nút c
    until False;
    k^[r] := temp; //Đặt giá trị vào nút cuối đường đi
  end;

begin
  for i := n shr 1 downto 1 do Heapify(i, n); //Vun dãy từ dưới lên tạo thành đống
  for i := n downto 2 do
    begin
      Swap(k^[1], k^[i]); //Chuyển khóa lớn nhất ra cuối dãy
      Heapify(1, i - 1); //Vun phần còn lại thành đống
    end;
end;
```

//Thuật toán sắp xếp kiểu trộn

```
procedure MergeSort;
var
  t, tmp: PKeyArray;
  len: TIndex;

  //Trộn Source[a...b] và Source[b + 1...c] thành Dest[a...c]
  procedure Merge(var Source, Dest: TKeyArray; a, b, c: TIndex);
  var
    i, j, p: TIndex;
  begin
    p := a; i := a; j := b + 1;
    while (i <= b) and (j <= c) do
      begin
        if Source[i] <= Source[j] then
```

```
          begin
            Dest[p] := Source[i]; Inc(i);
          end
        else
          begin
            Dest[p] := Source[j]; Inc(j);
          end;
        Inc(p);
      end;
    if i <= b then //Mạch Source[a...b] hết trước, đưa phần cuối mạch Source[b + 1...c] vào Dest
      Move(Source[i], Dest[p], (c - p + 1) * SizeOf(TKey))
    else //Mạch Source[b + 1...c] đã hết, đưa phần cuối mạch Source[a...b] vào Dest
      Move(Source[j], Dest[p], (c - p + 1) * SizeOf(TKey));
  end;

  procedure OnePassMergeSort(var Source, Dest: TKeyArray; len: TIndex);
  var
    a, b, c: TIndex;
  begin
    a := 1; b := len; c := len shl 1;
    while c < n do
      begin
        Merge(Source, Dest, a, b, c);
        Inc(a, len shl 1); Inc(b, len shl 1); Inc(c, len shl 1);
      end;
    if b < n then //Còn lại 2 mạch mà mạch thứ hai có độ dài < len
      Merge(Source, Dest, a, b, n)
    else //Chỉ còn lại tối đa 1 mạch
      Move(Source[a], Dest[a], (n - a + 1) * SizeOf(TKey));
  end;

begin //Thuật toán sắp xếp kiểu trộn
  t := GetMemory(n * SizeOf(TKey));
  try
    len := 1;
    while len < n do
      begin
        OnePassMergeSort(k^, t^, len);
        len := len shl 1;
        tmp := k; k := t; t := tmp; //Đảo vai trò k và t
      end;
  finally
    FreeMemory(t);
  end;
end;

//Thuật toán đếm phân phối
procedure CountingSort;
var
  c: PIndexArray; //Dãy biến đếm
  t, tmp: PKeyArray;
  i: TIndex;
  v: TKey;
begin
  c := GetMemory((SupV - MinV) * SizeOf(TIndex));
  FillChar(c^, (SupV - MinV) * SizeOf(TIndex), 0);
  t := GetMemory(n * SizeOf(TKey));
```

```
  try
    for v := 0 to SupV - MinV - 1 do
      c^[v] := 0; //Khởi tạo dãy biến đếm
    for i := 1 to n do
      Inc(c^[k^[i] - MinV]);
    for v := 1 to SupV - MinV - 1 do
      Inc(c^[v], c^[v - 1]); //Tính vị trí cuối mỗi đoạn
    for i := n downto 1 do //Phân phối
      begin
        v := k^[i] - MinV;
        t^[c^[v]] := k^[i];
        Dec(c^[v]);
      end;
    tmp := k; k := t; t := tmp; //Đảo dãy khóa phụ và dãy khóa chính
  finally
    FreeMemory(c);
    FreeMemory(t); //Giải phóng bộ nhớ cho dãy khóa phụ
  end;
end;

//Thuật toán MSD Radix Sort
procedure MSDRadixSort;
const
  maxBit = 31;
var
  b, z: TBitIndex;
  mask: array[0..maxBit] of TKey;

  procedure Partition(L, H: TIndex; b: TBitIndex); //Phân đoạn k[L...H] dựa vào bit b
  var
    i, j: TIndex;
    MaskValue: TKey;
    temp: TKey;
  begin
    if L >= H then Exit;
    i := L; j := H;
    MaskValue := Mask[b];
    temp := k^[L];
    repeat
      while (k^[j] and MaskValue <> 0) and (i < j) do Dec(j); //Lùi j, tìm khóa có bit b = 0
      if i < j then
        begin //Đưa khóa tìm được vào chỗ trống ở vị trí i
          k^[i] := k^[j]; Inc(i);
        end
      else Break;
      while (k^[i] and MaskValue = 0) and (i < j) do Inc(i); //Tiến i, tìm khóa có bit b = 1
      if i < j then //Đưa khóa tìm được vào chỗ trống ở vị trí j
        begin
          k^[j] := k^[i]; Dec(j);
        end
      else Break;
    until i = j;
    k^[i] := temp;
    if b > 0 then
      begin
        if temp and MaskValue <> 0 then Dec(i); //Tính vị trí phân đoạn
        Partition(L, i, Pred(b)); Partition(Succ(i), H, Pred(b));
```

```
      end;
  end;

begin
  z := Ceil(Log2(SupV));
  for b := 0 to z - 1 do //Tính các giá trị mặt nạ
    mask[b] := 1 shl b;
  Partition(1, n, z - 1);
end;

//Thuật toán LSD Radix Sort
procedure LSDRadixSort;
const
  Radix = 256;
var
  t: PKeyArray;
  p, z: TBitIndex;
  tmp: PKeyArray;

  //This function has to be changed whenever the value of Radix changes
  function GetDigit(key: TKey; p: TBitIndex): TKey;
  begin
    GetDigit := (key shr (p shl 3)) and $FF;
  end;

  //Sắp xếp bằng đếm phân phối theo chữ số p từ Source sang Dest
  procedure DCount(var x, y: TKeyArray; p: TBitIndex);
  var
    i: TIndex;
    d: TKey;
    c: array[0..Radix - 1] of TIndex;
  begin
    FillChar(c, SizeOf(c), 0);
    for i := 1 to n do
      begin
        d := GetDigit(x[i], p);  Inc(c[d]);
      end;
    for d := 1 to Radix - 1 do Inc(c[d], c[d - 1]);
    for i := n downto 1 do
      begin
        d := GetDigit(x[i], p);
        y[c[d]] := x[i];
        Dec(c[d]);
      end;
  end;

begin
  t := GetMemory(n * SizeOf(TKey));
  try
    z := Ceil(LogN (Radix, SupV));
    for p := 0 to z - 1 do
      begin
        DCount(k^, t^, p);
        tmp := t; t := k; k := tmp;
      end;
  finally
    FreeMemory(t);
```

```
  end;
end;

//In ra menu và ký tự tương ứng với menu được chọn
function MenuInteraction: AnsiChar;
var
  i: LongInt;
begin
  WriteLn('Sorting Algorithm Demo');
  WriteLn('======================');
  for i := 1 to nMenu do
    WriteLn(MenuS[i]);
  WriteLn;
  Write('Enter your choice: ');
  ReadLn(Result);
  Result := UpCase(Result);
  for i := 1 to nMenu do
    if MenuS[i][1] = Result then
      WriteLn(Copy(MenuS[i], 4, Length(MenuS[i]) - 3));
end;

begin
  //Khởi tạo một cấu hình mặc định
  n := 12345;
  MinV := 0;
  SupV := 6789;
  Randomize;
  Seed := RandSeed;
  repeat
    case MenuInteraction of
      '1': Setup;
      '2': DisplayInput;
      'A': Perform(@SelectionSort);
      'B': Perform(@BubbleSort);
      'C': Perform(@CocktailSort);
      'D': Perform(@InsertionSort);
      'E': Perform(@ShellSort);
      'F': Perform(@CombSort);
      'G': Perform(@QuickSort);
      'H': Perform(@HeapSort);
      'I': Perform(@MergeSort);
      'J': Perform(@CountingSort);
      'K': Perform(@MSDRadixSort);
      'L': Perform(@LSDRadixSort);
      '0': Halt;
    end;
  until False;
end.
```

Những con số về thời gian và tốc độ chương trình đo được là qua thử nghiệm trên một bộ dữ liệu cụ thể, với một máy tính cụ thể và một công cụ lập trình cụ thể. Với bộ dữ liệu khác, máy tính và môi trường lập trình khác, kết quả có thể khác.

Chú ý rằng thời gian đo bằng hàm $GetTickCount$ chỉ chính xác tới khoảng 50 milli giây, có nghĩa là sự chênh lệch ít hơn 50 milli giây hầu như không phản ánh gì về sự khác biệt tốc độ. Muốn đo thời gian chính xác hơn cần phải sử dụng bộ đếm khác, chẳng hạn như bộ đếm nhịp CPU hoặc bộ đếm Multimedia.

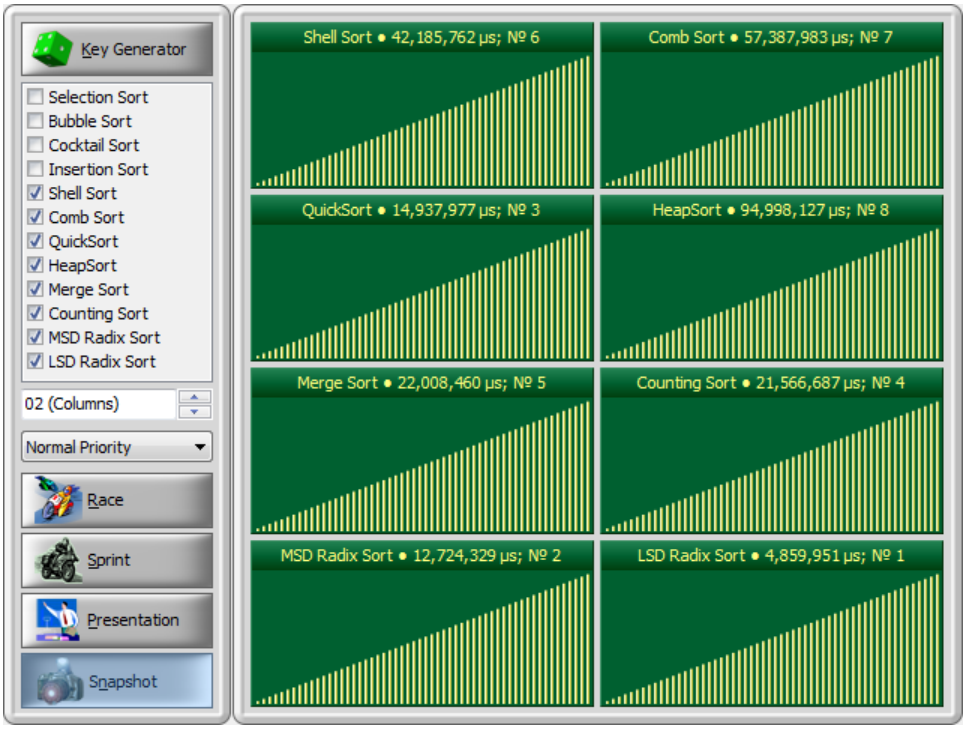
12.2. Đánh giá, nhận xét

Tôi đã viết lại chương trình này trên CodeGear™ RAD Studio-Delphi 2009 để đưa vào một số cải tiến:

- Thiết kế dựa trên kiến trúc đa luồng (multi-threads) cho phép chạy đồng thời hai hay nhiều thuật toán sắp xếp để so sánh tốc độ, bên cạnh đó vẫn có thể chạy tuần tự các thuật toán sắp xếp để đo thời gian thực hiện chính xác của chúng.
- Quá trình sắp xếp được hiển thị trực quan trên màn hình.
- Thời gian thực hiện mỗi thuật toán được đo bằng μs thay vì ms ($1\ \mu s = 10^{-6}$ giây), sử dụng bộ đếm nhịp CPU.
- Bộ sinh dữ liệu ngoài việc cho phép sinh dãy khóa ngẫu nhiên còn cho phép sinh dãy khóa theo một tình trạng nào đó.

Thử nghiệm cả hai chương trình, một trên Free Pascal 2.2 và một trên Delphi 2007, nhìn chung tốc độ sắp xếp của chương trình viết trên Delphi nhanh hơn nhiều so với chương trình trên FP, tuy nhiên khi so sánh tốc độ tương đối giữa các thuật toán vẫn có nhiều khác biệt giữa hai chương trình. Có một số thuật toán thực hiện nhanh bất ngờ so với dự đoán và cũng có một số thuật toán thực hiện chậm hơn hẳn so với đánh giá lý thuyết. Điều này có thể giải thích do hiệu ứng của bộ nhớ cache và cách thức tối ưu mã lệnh nhưng hơi phức tạp và cũng không thực sự cần thiết. Ta chỉ rút ra kinh nghiệm rằng tốc độ thực thi của một thuật toán phụ thuộc rất nhiều vào phần cứng máy tính và chương trình dịch.

Hình 12-1 là giao diện chính, bạn có thể tham khảo chương trình kèm theo.



Hình 12-1. Máy AMD 64X2 6 GHz, 2GB RAM tỏ ra chậm chạp khi sắp xếp 10^8 khóa $\in [0, 10^7]$ cho dù những thuật toán sắp xếp tốt nhất đã được áp dụng

Cùng một mục đích sắp xếp như nhau, nhưng có nhiều phương pháp giải quyết khác nhau. Nếu chỉ dựa vào thời gian đo được trong một ví dụ cụ thể mà đánh giá thuật toán này tốt hơn thuật toán kia về mọi mặt là điều không nên. Việc chọn một thuật toán sắp xếp thích hợp cho phù hợp với từng yêu cầu, từng điều kiện cụ thể là kỹ năng của người lập trình.

Những thuật toán có độ phức tạp $\Omega(n^2)$ (Selection Sort, Bubble Sort, Cocktail Sort, Insertion Sort) thì chỉ nên áp dụng với dãy khóa kích thư ớc nhỏ. Tuy nhiên mã lệnh của chúng lại đơn giản, dễ nhớ mà người mới học lập trình nào cũng có thể cài đặt được. Các thuật toán tốc độ cao hơn (ShellSort, CombSort, QuickSort, HeapSort, Merge Sort) nên sử dụng trong trường hợp dãy khóa có kích thước lớn.

Thuật toán đếm phân phối và thuật toán sắp xếp bằng cơ số nên được tận dụng trong trường hợp các khoá sắp xếp là số tự nhiên (hay là một kiểu dữ liệu có thể quy ra thành các số tự nhiên) bởi những thuật toán này có tốc độ rất cao. Thuật toán sắp xếp cơ số cũng có thể sắp xếp dãy khoá có số thực hay số âm nhưng cần đưa vào một số sửa đổi nhỏ.

Những thuật toán trên đều là kết quả của những cách tiếp cận khoa học đối với bài toán sắp xếp. Chúng không chỉ đơn thuần cho ta hiểu thêm về một cách sắp xếp mới, mà việc

cài đặt chúng cũng cho ta thêm nhiều kinh nghiệm: Kỹ thuật sử dụng số ngẫu nhiên, kỹ thuật "chia để trị", kỹ thuật dùng các biến với vai trò luân phiên v.v…

Phương pháp sắp xếp nhanh nhất là phương pháp biết dùng đúng thuật toán tốt nhất cho từng trường hợp cụ thể. Để đánh giá thuật toán nào thích hợp với trường hợp nào, ngoài những tính toán lý thuyết, còn phải cài đặt và chạy thử với nhiều bộ dữ liệu khác nhau (thậm chí phải thử trên nhiều máy khác nhau). Trong các thư viện của các ngôn ngữ lập trình như STL library của C++ hay Delphi-Jedi của Object Pascal đều có trang bị sẵn IntroSort [**14**] – một thuật toán sắp xếp hỗn hợp sử dụng QuickSort, Insertion Sort và HeapSort, hiện nay đang được coi là thuật toán sắp xếp nhanh nhất trên thực tế.

Học thuộc lòng việc cài đặt các thuật toán sắp xếp không phải là việc dễ, chúng ta cần hiểu chúng và có lẽ chỉ cần thuộc vài thuật toán để cài đặt được nhanh và tránh nhầm lẫn. Đối với cá nhân tôi thì nếu không sử dụng thư viện sắp xếp có sẵn, tôi sẽ:

- Thuộc Insertion Sort để dùng trong trường hợp cần cài đặt nhanh và dữ liệu nhỏ.
- Thuộc QuickSort để dùng trong trường hợp dữ liệu vừa và lớn.
- Thuộc Counting Sort và Radix Sort để dùng trong trường hợp tốc độ là ưu tiên số một, hoặc trường hợp các khóa có cấu trúc xâu (string)
- Còn các thuật toán khác, khi cần vẫn lập trình được nhưng sẽ phải tốn thêm "năng lượng tư duy".

Chương IV. KỸ THUẬT THIẾT KẾ THUẬT TOÁN

"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change"

Charles Darwin

Chương này giới thiệu một số kỹ thuật quan trọng trong việc tiếp cận bài toán và tìm thuật toán. Các lớp thuật toán sẽ được thảo luận trong chương này là: Vét cạn (exhaustive search), Chia để trị (divide and conquer), Quy hoạch động (dynamic programming) và Tham lam (greedy).

Các bài toán trên thực thế có muôn hình muôn vẻ, không thể đưa ra một cách thức chung để tìm giải thuật cho mọi bài toán. Các phương pháp này cũng chỉ là những "chiến lược" kinh điển.

Khác với những thuật toán cụ thể mà chúng ta đã biết như QuickSort, tìm kiếm nhị phân,…, các vấn đề trong chương này không thể học theo kiểu "thuộc và cài đặt", cũng như không thể tìm thấy các thuật toán này trong bất cứ thư viện lập trình nào. Chúng ta chỉ có thể khảo sát một vài bài toán cụ thể và học cách nghĩ, cách tiếp cận vấn đề, cách thiết kế giải thuật. Từ đó rèn luyện kỹ năng linh hoạt khi giải các bài toán thực tế.

Bài 13. Liệt kê

Có một số bài toán trên thực tế yêu cầu chỉ rõ: trong một tập các đối tượng cho trước có bao nhiêu đối tượng thoả mãn những điều kiện nhất định và đó là những đối tượng nào. Bài toán này gọi là *bài toán liệt kê* hay *bài toán duyệt*.

Nếu ta biểu diễn các đối tượng cần tìm dưới dạng một cấu hình các biến số thì để giải bài toán liệt kê, cần phải xác định được một *thuật toán* để có thể theo đó lần lượt xây dựng được tất cả các cấu hình đang quan tâm. Có nhiều phương pháp liệt kê, nhưng chúng cần phải đáp ứng được hai yêu cầu dưới đây:

- Không được lặp lại một cấu hình
- Không được bỏ sót một cấu hình

Trước khi nói về các thuật toán liệt kê, chúng ta giới thiệu một số khái niệm cơ bản:

13.1. Vài khái niệm cơ bản

13.1.1. Thứ tự từ điển

Nhắc lại rằng quan hệ thứ tự toàn phần "nhỏ hơn hoặc bằng" ký hiệu "≤" trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

Với $\forall\, a, b, c \in S$

- Tính phổ biến (Universality): Hoặc là $a \leq b$, hoặc $b \leq a$;
- Tính phản xạ (Reflexivity): $a \leq a$
- Tính phản đối xứng (Antisymmetry) : Nếu $a \leq b$ và $b \leq a$ thì bắt buộc $a = b$
- Tính bắc cầu (Transitivity): Nếu có $a \leq b$ và $b \leq c$ thì $a \leq c$.

Các quan hệ $\geq, >, <$ có thể tự suy ra từ quan hệ $\leq$ này.

Trên các dãy hữu hạn, người ta cũng xác định một quan hệ thứ tự:

Xét $a_{1\ldots n}$ và $b_{1\ldots n}$ là hai dãy độ dài n, trên các phần tử của a và b đã có quan hệ thứ tự toàn phần "≤". Khi đó $a_{1\ldots n} \leq b_{1\ldots n}$ nếu như :

- Hoặc hai dãy giống nhau: $a_i = b_i, \forall i: 1 \leq i \leq n$
- Hoặc tồn tại một số nguyên dương $k \leq n$ để $a_k < b_k$ và $a_i = b_i, \forall i: 1 \leq i < k$

Thứ tự đó gọi là *thứ tự từ điển (lexicographic order)* trên các dãy độ dài n.

Khi hai dãy a và b có số phần tử khác nhau, người ta cũng xác định được thứ tự từ điển. Bằng cách thêm vào cuối dãy a hoặc dãy b những phần tử đặc biệt gọi là ϵ để độ dài của a và b bằng nhau, và coi những phần tử ϵ này nhỏ hơn tất cả các phần tử khác, ta lại đưa về xác định thứ tự từ điển của hai dãy cùng độ dài.

Ví dụ:

$$(1,2,3,4) < (5,6)$$
$$(a,b,c) < (a,b,c,d)$$
$$\text{"calculator"} < \text{"computer"}$$

Thứ tự từ điển cũng là một quan hệ thứ tự toàn phần trên các dãy.

13.1.2. Chỉnh hợp, tổ hợp, hoán vị.

Cho S là một tập hữu hạn gồm n phần tử và k là một số tự nhiên. Gọi X là tập các số nguyên dương từ 1 tới k: $X = \{1,2,\dots,k\}$

❑ Chỉnh hợp lặp

Một ánh xạ $f: X \to S$ cho tương ứng mỗi phần tử $i \in X$ một và chỉ một phần tử $f(i) \in S$, được gọi là một chỉnh hợp lặp chập k của S

Do X là tập hữu hạn (k phần tử) nên ánh xạ f có thể xác định qua bảng các giá trị $\big(f(1), f(2), \dots, f(k)\big)$, vì vậy ta có thể đồng nhất f với dãy giá trị $\big(f(1), f(2), \dots, f(k)\big)$ và coi dãy giá trị này cũng là một chỉnh hợp lặp chập k của S.

Ví dụ $S = \{A, B, C, D, E, F\}$. Một ánh xạ f cho bởi:

i	1	2	3
$f(i)$	E	C	E

tương ứng với tập ảnh (E, C, E) là một chỉnh hợp lặp của S

Số chỉnh hợp lặp chập k của tập n phần tử là n^k

❑ Chỉnh hợp không lặp

Mỗi đơn ánh $f: X \to S$ được gọi là một chỉnh hợp không lặp chập k của S. Nói cách khác, một chỉnh hợp không lặp là một chỉnh hợp lặp có các phần tử khác nhau đôi một.

Ví dụ một chỉnh hợp không lặp chập 3 (A, C, E) của tập $S = \{A, B, C, D, E, F\}$

i	1	2	3
$f(i)$	A	C	E

Số chỉnh hợp không lặp chập k của tập n phần tử là ${}_nP_k = \frac{n!}{(n-k)!}$

❑ Hoán vị

Khi $k = n$ mỗi song ánh $f: X \to S$ được gọi là một hoán vị của S. Nói cách khác một hoán vị của S là một chỉnh hợp không lặp chập n của S.

Ví dụ: (A, D, C, E, B, F) là một hoán vị của $S = \{A, B, C, D, E, F\}$

i	1	2	3	4	5	6
$f(i)$	A	D	C	E	B	F

Số hoán vị của tập n phần tử là $P_n = {}_nP_n = n!$

❑ Tổ hợp

Mỗi tập con gồm k phần tử của S được gọi là một tổ hợp chập k của S.

Lấy một tổ hợp chập k của S, xét tất cả $k!$ hoán vị của nó, mỗi hoán vị sẽ là một chỉnh hợp không lặp chập k của S. Điều đó tức là khi liệt kê tất cả các chỉnh hợp không lặp chập k thì mỗi tổ hợp chập k sẽ được tính $k!$ lần. Như vậy nếu xét về mặt số lượng:

Số tổ hợp chập k của tập n phần tử là $\binom{n}{k} = \frac{{}_nP_k}{k!} = \frac{n!}{k!(n-k)!}$

Ta có công thức khai triển nhị thức:

$$(x + a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Vì vậy số $\binom{n}{k}$ còn được gọi là *hệ số nhị thức* (*binomial coefficient*) thứ k, bậc n

13.2. Phương pháp sinh

Phương pháp sinh có thể áp dụng để giải bài toán liệt kê nếu như hai điều kiện sau thoả mãn:

- Có thể xác định được một thứ tự trên tập các cấu hình tổ hợp cần liệt kê. Từ đó có thể biết được cấu hình đầu tiên và cấu hình cuối cùng theo thứ tự đó.
- Xây dựng được thuật toán từ một cấu hình chưa phải cấu hình cuối, sinh ra được cấu hình kế tiếp nó.

13.2.1. Mô hình sinh

Phương pháp sinh có thể viết bằng mô hình chung:

```
«Xây dựng cấu hình đầu tiên»;
repeat
  «Đưa ra cấu hình đang có»;
  «Từ cấu hình đang có sinh ra cấu hình kế tiếp nếu còn»;
until «hết cấu hình»;
```

13.2.2. Liệt kê các dãy nhị phân độ dài n

Một dãy nhị phân độ dài n là một dãy $x_{1..n}$ trong đó $x_i \in \{0,1\}, \forall i: 1 \le i \le n$.

Có thể nhận thấy rằng một dãy nhị phân $x_{1\ldots n}$ là biểu diễn nhị phân của một giá trị nguyên $v(x)$ nào đó $(0 \le v(x) < 2^n)$. Số các dãy nhị phân độ dài n bằng 2^n, thứ tự từ điển trên các dãy nhị phân độ dài n tương đương với quan hệ thứ tự trên các giá trị số mà chúng biểu diễn. Vì vậy, liệt kê các dãy nhị phân theo thứ tự từ điển nghĩa là phải chỉ ra lần lượt các dãy nhị phân biểu diễn các số nguyên theo thứ tự $0,1,\ldots,2^n-1$.

Ví dụ với $n = 3$, có 8 dãy nhị phân độ dài 3 được liệt kê:

x	000	001	010	011	100	101	110	111
$v(x)$	0	1	2	3	4	5	6	7

Theo thứ tự liệt kê, dãy đầu tiên là $\underbrace{00\ldots0}_{n}$ và dãy cuối cùng là $\underbrace{11\ldots1}_{n}$. Nếu ta có một dãy nhị phân độ dài n, ta có thể sinh ra dãy nhị phân kế tiếp bằng cách cộng thêm 1 (theo cơ số 2 có nhớ) vào dãy hiện tại.

```
10101111
    + 1
--------
10110000
```

Dựa vào tính chất của phép cộng hai số nhị phân, cấu hình kế tiếp có thể sinh từ cấu hình hiện tại bằng cách: xét từ cuối dãy lên đầu dãy (xét từ hàng đơn vị lên), tìm số 0 gặp đầu tiên…

- Nếu thấy thì thay số 0 đó bằng số 1 và đặt tất cả các phần tử phía sau vị trí đó bằng 0.
- Nếu không thấy thì thì toàn dãy là số 1, đây là cấu hình cuối cùng.

Input

Số nguyên dương n.

Output

Các dãy nhị phân độ dài n.

Sample Input	Sample Output
3	000 001 010 011 100 101 110 111

Thuật toán sinh liệt kê các dãy nhị phân độ dài n

```
program BinaryStringEnumeration;
var
  x: AnsiString;
  n, i: LongInt;
begin
  ReadLn(n);
  SetLength(x, n);
  FillChar(x[1], n * SizeOf(AnsiChar), '0'); //Cấu hình ban đầu x=00..0
  repeat
    WriteLn(x);
    //Tìm số 0 đầu tiên từ cuối dãy
    i := n;
    while (i > 0) and (x[i] = '1') do Dec(i);
    if i > 0 then //Nếu tìm thấy
      begin
        x[i] := '1'; //Thay x[i] bằng số 1
        if i < n then //Đặt x[i+1..n] := 0
          FillChar(x[i + 1], (n - i) * SizeOf(AnsiChar), '0');
      end
    else
      Break; //Không tìm thấy số 0 nào trong dãy thì dừng
  until False;
end.
```

13.2.3. Liệt kê các tập con có k phần tử

Ta sẽ lập chương trình liệt kê các tập con k phần tử của tập $S = \{1,2,\dots,n\}$ theo thứ tự từ điển.

Ví dụ: $n = 5, k = 3$, có 10 tập con:

{1, 2, 3}	{1, 2, 4}	{1, 2, 5}	{1, 3, 4}	{1, 3, 5}
{1, 4, 5}	{2, 3, 4}	{2, 3, 5}	{2, 4, 5}	{3, 4, 5}

Bài toán liệt kê các tập con k phần tử của tập $S = \{1,2,\dots,n\}$ có thể quy về bài toán liệt kê các dãy k phần tử $x_{1\dots k}$, trong đó $1 \le x_1 < x_2 < \dots < x_k \le n$. Nếu sắp xếp các dãy này theo thứ tự từ điển, ta nhận thấy:

Tập con đầu tiên (cấu hình khởi tạo) là $\{1,2,\dots,k\}$.
Tập con cuối cùng (cấu hình kết thúc) là $\{n-k+1, n-k+2, \dots, n\}$.

Xét một tập con $\{x_{1\dots k}\}$ trong đó $1 \le x_1 < x_2 < \dots < x_k \le n$, ta có nhận xét rằng giới hạn trên (giá trị lớn nhất có thể nhận) của x_k là n, của x_{k-1} là $n-1$, của x_{k-2} là $n-2$…
Tổng quát: giới hạn trên của x_i là $n-k+i$.

Còn tất nhiên, giới hạn dưới (giá trị nhỏ nhất có thể nhận) của x_i là $x_{i-1}+1$.

Từ một dãy $x_{1\dots k}$ đại diện cho một tập con của S, nếu tất cả các phần tử trong x đều đã đạt tới giới hạn trên thì x là cấu hình cuối cùng , nếu không thì ta phải sinh ra một dãy mới

tăng dần thoả mãn: dãy mới vừa đủ lớn hơn dãy cũ theo nghĩa không có một dãy k phần tử nào chen giữa chúng khi sắp thứ tự từ điển.

> Ví dụ: $n = 9, k = 6$. Cấu hình đang có $x = (1,2,\underline{6,7,8,9})$. Các phần tử $x_{3\ldots6}$ đã đạt tới giới hạn trên, nên để sinh cấu hình mới ta không thể sinh bằng cách tăng một phần tử trong số các phần tử $x_{3\ldots6}$ lên được, ta phải tăng $x_2 = 2$ lên 1 đơn vị thành $x_2 = 3$. Được cấu hình mới $x = (1,3,6,7,8,9)$. Cấu hình này lớn hơn cấu hình trước nhưng chưa thoả mãn tính chất *vừa đủ lớn*. Muốn tìm cấu hình vừa đủ lớn hơn cấu hình cũ, cần có thêm thao tác: Thay các giá trị $x_{3\ldots6}$ bằng các giới hạn dưới của chúng. Tức là:
>
> $$x_3 := x_2 + 1 = 4$$
> $$x_4 := x_3 + 1 = 5$$
> $$x_5 := x_4 + 1 = 6$$
> $$x_6 := x_5 + 1 = 7$$
>
> Ta được cấu hình mới $x = (1,3,4,5,6,7)$ là cấu hình kế tiếp. Tiếp tục với cấu hình này, ta lại nhận thấy rằng $x_6 = 7$ chưa đạt giới hạn trên, như vậy chỉ cần tăng x_6 lên 1 là được cấu hình mới $x = (1,3,4,5,6,8)$.

Thuật toán sinh dãy con kế tiếp từ dãy đang có $x_{1\ldots k}$ có thể xây dựng như sau:

Tìm từ cuối dãy lên đầu cho tới khi gặp một phần tử x_i chưa đạt giới hạn trên $n - k + i\ldots$

- Nếu tìm thấy:
  - Tăng x_i lên 1
  - Đặt tất cả các phần tử $x_{i+1\ldots k}$ bằng giới hạn dưới của chúng
- Nếu không tìm thấy tức là mọi phần tử đã đạt giới hạn trên, đây là cấu hình cuối cùng

Input

Hai số nguyên dương $n, k, (1 \le k \le n \le 100)$

Output

Các tập con k phần tử của tập $\{1,2,\ldots,n\}$

Sample Input	Sample Output
5 3	{1, 2, 3} {1, 2, 4} {1, 2, 5} {1, 3, 4} {1, 3, 5} {1, 4, 5} {2, 3, 4} {2, 3, 5} {2, 4, 5} {3, 4, 5}

💻 Thuật toán sinh liệt kê các tập con k phần tử

```
program SubSetEnumeration;
const
  max = 100;
var
  x: array[1..max] of LongInt;
  n, k, i, j: LongInt;
begin
  ReadLn(n, k);
  for i := 1 to k do x[i] := i; //Khởi tạo x := (1, 2, ..., k)}
  repeat
    //In ra cấu hình hiện tại
    Write('{');
    for i := 1 to k do
      begin
        Write(x[i]);
        if i < k then Write(', ');
      end;
    WriteLn('}');
    //Duyệt từ cuối dãy lên tìm x[i] chưa đạt giới hạn trên n – k + i
    i := k;
    while (i > 0) and (x[i] = n - k + i) do Dec(i);
    if i > 0 then //Nếu tìm thấy
      begin
        Inc(x[i]); //Tăng x[i] lên 1
        //Đặt x[i + 1...k] bằng giới hạn dưới của chúng
        for j := i + 1 to k do x[j] := x[j - 1] + 1;
      end
    else Break;
  until False;
end.
```

13.2.4. Liệt kê các hoán vị

Ta sẽ lập chương trình liệt kê các hoán vị của tập $S = \{1,2,\dots,n\}$ theo thứ tự từ điển.

Ví dụ với n = 3, có 6 hoán vị:

(1,2,3); (1,3,2); (2,1,3); (2,3,1); (3,1,2); (3,2,1)

Mỗi hoán vị của tập $S = \{1,2,\dots,n\}$ có thể biểu diễn dưới dạng một một dãy số $x_{1\dots n}$. Theo thứ tự từ điển, ta nhận thấy:

Hoán vị đầu tiên cần liệt kê: $(1,2,\dots,n)$

Hoán vị cuối cùng cần liệt kê: $(n,n-1,\dots,1)$

Bắt đầu từ hoán vị $(1,2,\dots,n)$, ta sẽ sinh ra các hoán vị còn lại theo quy tắc: Hoán vị sẽ sinh ra phải là hoán vị vừa đủ lớn hơn hoán vị hiện tại theo nghĩa không thể có một hoán vị nào khác chen giữa chúng khi sắp thứ tự.

Giả sử hoán vị hiện tại là $x = (3,2,6,5,4,1)$, xét 4 phần tử cuối cùng, ta thấy chúng được xếp giảm dần, điều đó có nghĩa là cho dù ta có hoán vị 4 phần tử này thế nào, ta cũng được một hoán vị bé hơn hoán vị hiện tại. Như vậy ta phải xét đến $x_2 = 2$ và thay nó bằng một giá trị khác. Ta sẽ thay bằng giá trị nào?, không thể là 1 bởi nếu vậy sẽ được hoán vị nhỏ hơn, không thể là 3 vì đã có $x_1 = 3$ rồi (phần tử sau không được chọn vào những giá trị mà phần tử trước đã chọn). Còn lại các giá trị: 4, 5 và 6. Vì cần một hoán vị vừa đủ lớn hơn hiện tại nên ta chọn $x_2 := 4$. Còn các giá trị $x_{3\ldots6}$ sẽ lấy trong tập $\{2,6,5,1\}$. Cũng vì tính vừa đủ lớn nên ta sẽ tìm biểu diễn nhỏ nhất của 4 số này gán cho $x_{3\ldots6}$ tức là $x_{3\ldots6} := (1,2,5,6)$. Vậy hoán vị mới sẽ là $(3,4,1,2,5,6)$.

Ta có nhận xét gì qua ví dụ này: Đoạn cuối của hoán vị hiện tại $x_{3\ldots6}$ được xếp giảm dần, số $x_5 = 4$ là số nhỏ nhất trong đoạn cuối giảm dần thoả mãn điều kiện lớn hơn $x_2 = 2$. Nếu đảo giá trị x_5 và x_2 thì ta sẽ được hoán vị $(3,4,6,5,2,1)$, trong đó đoạn cuối $x_{3\ldots6}$ vẫn được sắp xếp giảm dần. Khi đó muốn biểu diễn nhỏ nhất cho các giá trị trong đoạn cuối thì ta chỉ cần đảo ngược đoạn cuối.

Trong trường hợp hoán vị hiện tại là $(2,1,3,4)$ thì hoán vị kế tiếp sẽ là $(2,1,4,3)$. Ta cũng có thể coi hoán vị $(2,1,3,4)$ có đoạn cuối giảm dần, đoạn cuối này chỉ gồm 1 phần tử (4)

Thuật toán sinh hoán vị kế tiếp từ hoán vị hiện tại có thể xây dựng như sau:

Xác định đoạn cuối giảm dần dài nhất, tìm chỉ số i của phần tử x_i đứng liền trước đoạn cuối đó. Điều này đồng nghĩa với việc tìm từ vị trí sát cuối dãy lên đầu, gặp chỉ số i đầu tiên thỏa mãn $x_i < x_{i+1}$.

- Nếu tìm thấy chỉ số i như trên
  - Trong đoạn cuối giảm dần, tìm phần tử x_k nhỏ nhất vừa đủ lớn hơn x_i. Do đoạn cuối giảm dần, điều này thực hiện bằng cách tìm từ cuối dãy lên đầu gặp chỉ số k đầu tiên thoả mãn $x_k > x_i$ (có thể dùng tìm kiếm nhị phân).
  - Đảo giá trị x_k và x_i
  - Lật ngược thứ tự đoạn cuối giảm dần ($x_{i+1\ldots k}$), đoạn cuối trở thành tăng dần.
- Nếu không tìm thấy tức là toàn dãy đã sắp giảm dần, đây là cấu hình cuối cùng

Input

Số nguyên dương $n \leq 100$

Output

Các hoán vị của dãy $(1,2,\ldots,n)$

Sample Input	Sample Output
3	(1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)

Thuật toán sinh liệt kê hoán vị

```
{$INLINE ON}
program PermutationEnumeration;
const
  max = 100;
var
  x: array[1..max] of LongInt;
  n, i, k, l, h: LongInt;

//Thủ tục đảo giá trị hai tham biến x, y
procedure Swap(var x, y: LongInt); inline;
var
  temp: LongInt;
begin
  temp := x; x := y; y := temp;
end;

begin
  ReadLn(n);
  for i := 1 to n do x[i] := i;
  repeat
    //In cấu hình hiện tại
    Write('(');
    for i := 1 to n do
      begin
        Write(x[i]);
        if i < n then Write(', ');
      end;
    WriteLn(')');
    //Sinh cấu hình kế tiếp
    //Tìm i là chỉ số đứng trước đoạn cuối giảm dần
    i := n - 1;
    while (i > 0) and (x[i] > x[i + 1]) do Dec(i);
    if i > 0 then //Nếu tìm thấy
      begin
        //Tìm từ cuối dãy phần tử đầu tiên (x[k]) lớn hơn x[i]
        k := n;
        while x[k] < x[i] do Dec(k);
        //Đảo giá trị x[k] và x[i]
        Swap(x[k], x[i]);
        //Lật ngược thứ tự đoạn cuối giảm dần, đoạn cuối trở thành tăng dần
        l := i + 1; h := n;
        while l < h do
          begin
            Swap(x[l], x[h]);
            Inc(l);
```

```
            Dec(h);
          end;
      end
    else Break; //Cả dãy là giảm dần, hết cấu hình
  until False;
end.
```

Nhược điểm của phương pháp sinh là không thể sinh ra được cấu hình thứ p nếu như chưa có cấu hình thứ $p-1$, điều đó làm phương pháp sinh ít tính phổ dụng trong những thuật toán duyệt hạn chế. Hơn thế nữa, không phải cấu hình ban đầu lúc nào cũng dễ tìm được, không phải kỹ thuật sinh cấu hình kế tiếp cho mọi bài toán đều đơn giản (Sinh các chỉnh hợp không lặp chập k theo thứ tự từ điển chẳng hạn). Ta sang một chuyên mục sau nói đến một phương pháp liệt kê có tính phổ dụng cao hơn, để giải các bài toán liệt kê phức tạp hơn đó là: Thuật toán quay lui (Back tracking).

13.3. Thuật toán quay lui

Thuật toán quay lui dùng để giải bài toán liệt kê các cấu hình. Thuật toán này làm việc theo cách:

- Mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử
- Mỗi phần tử được chọn bằng cách thử tất cả các khả năng.

Giả sử cấu hình cần liệt kê có dạng $x_{1\ldots n}$, khi đó thuật toán quay lui sẽ xét tất cả các giá trị x_1 có thể nhận, thử cho x_1 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_1, thuật toán sẽ xét tất cả các giá trị x_2 có thể nhận, lại thử cho x_2 nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x_2 lại xét tiếp các khả năng chọn x_3, cứ tiếp tục như vậy… Mỗi khi ta tìm được đầy đủ một cấu hình thì liệt kê ngay cấu hình đó.

Có thể mô tả thuật toán quay lui theo cách quy nạp: Thuật toán sẽ liệt kê các cấu hình n phần tử dạng $x_{1\ldots n}$ bằng cách thử cho x_1 nhận lần lượt các giá trị có thể. Với mỗi giá trị thử gán cho x_1, thuật toán tiếp tục liệt kê toàn bộ các cấu hình $n-1$ phần tử $x_{2\ldots n}$…

13.3.1. Mô hình quay lui

```
//Thủ tục này thử cho x[i] nhận lần lượt các giá trị mà nó có thể nhận
procedure Attempt(i);
begin
  for «mọi giá trị v có thể gán cho x[i]» do
    begin
      «Thử cho x[i] := v»;
      if «x[i] là phần tử cuối cùng trong cấu hình» then
        «Thông báo cấu hình tìm được»
      else
        begin
          «Ghi nhận việc cho x[i] nhận giá trị V (nếu cần)»;
          Attempt(i + 1); //Gọi đệ quy để chọn tiếp x[i+1]
          «Nếu cần, bỏ ghi nhận việc thử x[i] := V để thử giá trị khác»;
        end;
    end;
end;
```

Thuật toán quay lui sẽ bắt đầu bằng lời gọi $Attempt(1)$.

Tên gọi thuật toán quay lui là dựa trên cơ chế duyệt các cấu hình: Mỗi khi thử chọn một giá trị cho x_i, thuật toán sẽ gọi đệ quy để tìm tiếp x_{i+1}, … và cứ như vậy cho tới khi tiến trình duyệt xét tìm tới phần tử cuối cùng của cấu hình. Còn sau khiđã xét h ết tất cả khả năng chọn x_i, tiến trình sẽ lùi lại thử áp đặt một giá trị khác cho x_{i-1}.

13.3.2. Liệt kê các dãy nhị phân

Biểu diễn dãy nhị phân độ dài n dưới dạng dãy $x_{1...n}$. Ta sẽ liệt kê các dãy này bằng cách thử dùng các giá trị $\{0,1\}$ gán cho x_i. Với mỗi giá trị thử gán cho x_i lại thử các giá trị có thể gán cho x_{i+1}…

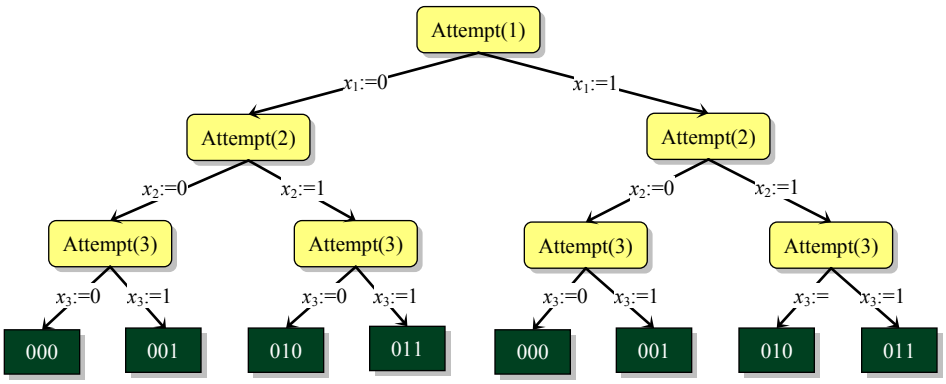
Sau đây là chương trình li ệt kê các dãy nhị phân với quy định khuôn dạng Input/Output như trong mục 13.2.2.

💻 Thuật toán quay lui liệt kê các dãy nhị phân độ dài n

```
program BinaryStringEnumeration;
var
  x: AnsiString;
  n: LongInt;

procedure Attempt(i: LongInt); //Thử các cách chọn x[i]
var
  j: AnsiChar;
begin
  for j := '0' to '1' do //Xét các giá trị j có thể gán cho x[i]
    begin //Với mỗi giá trị đó
      x[i] := j; //Thử đặt x[i]
      if i = n then WriteLn(x) //Nếu i = n thì in kết quả
      else Attempt(i + 1); //Nếu x[i] chưa phải phần tử cuối thì tìm tiếp x[i + 1]
    end;
end;

begin
```

```
  ReadLn(n);
  SetLength(x, n);
  Attempt(1); //Khởi động thuật toán quay lui
end.
```

Ví dụ: Khi $n = 3$, các lời gọi đệ quy thực hiện thuật toán quay lui có thể vẽ như cây trong Hình 13-1.



Hình 13-1. Cây tìm kiếm quay lui trong bài toán liệt kê dãy nhị phân

13.3.3. Liệt kê các tập con có k phần tử

Để liệt kê các tập con k phần tử của tập $S = \{1,2,\dots,n\}$ ta có thể đưa về liệt kê các cấu hình $x_{1\dots k}$, ở đây $1 \le x_1 < x_2 < \cdots < x_k \le n$.

Theo các nhận xét ở mục 13.2.3, giá trị cận dưới và cận trên của x_i là:

$$x_{i-1} + 1 \le x_i \le n - k + i \qquad (13.1)$$

(Giả thiết rằng có thêm một số $x_0 = 0$ khi xét công thức (13.1) với $i = 1$)

Thuật toán quay lui sẽ xét tất cả các cách chọn x_1 từ 1 $(= x_0 + 1)$ đến $n - k + 1$, với mỗi giá trị đó, xét tiếp tất cả các cách chọn x_2 từ $x_1 + 1$ đến $n - k + 2$, … cứ như vậy khi chọn được đến x_k thì ta có một cấu hình cần liệt kê.

Dưới đây là chương trình liệt kê các tập con k phần tử bằng thuật toán quay lui với khuôn dạng Input/Output như quy định trong mục 13.2.3.

💻 Thuật toán quay lui liệt kê các tập con k phần tử

```
program SubSetEnumeration;
const
  max = 100;
var
  x: array[0..max] of LongInt;
  n, k: LongInt;

procedure PrintResult; //In ra tập con {x[1..k]}
var
```

```
  i: LongInt;
begin
  Write('{');
  for i := 1 to k do
    begin
      Write(x[i]);
      if i < k then Write(', ');
    end;
  WriteLn('}');
end;

procedure Attempt(i: LongInt); //Thử các cách chọn giá trị cho x[i]
var
  j: LongInt;
begin
  for j := x[i - 1] + 1 to n - k + i do
    begin
      x[i] := j;
      if i = k then PrintResult
      else Attempt(i + 1);
    end;
end;

begin
  ReadLn(n, k);
  x[0] := 0;
  Attempt(1); //Khởi động thuật toán quay lui
end.
```

Về cơ bản, các chương trình cài đặt thuật toán quay lui chỉ khác nhau ở thủ tục $Attempt$. Ví dụ ở chương trình liệt kê dãy nhị phân, thủ tục này sẽ thử chọn các giá trị 0 hoặc 1 cho x_i; còn ở chương trình liệt kê các tập con k phần tử, thủ tục này sẽ thử chọn x_i là một trong các giá trị nguyên từ cận dưới $x_{i-1}+1$ tới cận trên $n-k+i$. Qua đó ta có thể thấy tính phổ dụng của thuật toán quay lui: mô hình cài đặt có thể thích hợp cho nhiều bài toán. Ở phương pháp sinh tuần tự, với mỗi bài toán lại phải có một thuật toán sinh cấu hình kế tiếp, điều đó làm cho việc cài đặt mỗi bài một khác, bên cạnh đó, không phải thuật toán sinh kế tiếp nào cũng dễ tìm ra và cài đặt được.

13.3.4. Liệt kê các chỉnh hợp không lặp chập k

Để liệt kê các chỉnh hợp không lặp chập k của tập $S = \{1,2,\dots,n\}$ ta có thể đưa về liệt kê các cấu hình $x_{1\dots k}$, các $x_i \in S$ và khác nhau đôi một.

Thủ tục $Attempt(i)$ – xét tất cả các khả năng chọn x_i – sẽ thử hết các giá trị từ 1 đến n chưa bị các phần tử đứng trước $x_{1\dots i-1}$ chọn. Muốn xem các giá trị nào chưa được chọn ta sử dụng kỹ thuật dùng mảng đánh dấu:

- Khởi tạo một mảng $Free[1 \dots n]$ mang kiểu logic boolean. Ở đây $Free[j]$ cho biết giá trị j có còn tự do hay đã bị chọn rồi. Ban đầu khởi tạo tất cả các phần tử mảng $Free[1 \dots n]$ là True có nghĩa là các giá trị từ 1 đến n đều tự do.
- Tại bước chọn các giá trị có thể của x_i ta chỉ xét những giá trị j còn tự do ($Free[j] \coloneqq True$).
- Trước khi gọi đệ quy $Attempt(i+1)$ để thử chọn tiếp x_{i+1}: ta đặt giá trị j vừa gán cho x_i là "đã bị chọn" ($Free[j] \coloneqq False$) để các thủ tục $Attempt(i+1)$, $Attempt(i+2)$… gọi sau này không chọn phải giá trị j đó nữa.
- Sau khi gọi đệ quy $Attempt(i+1)$: có nghĩa là sắp tới ta sẽ thử gán một giá trị khác cho x_i thì ta sẽ đặt giá trị j vừa thử cho x_i thành "tự do" ($Free[j] \coloneqq True$), bởi khi x_i đã nhận một giá trị khác rồi thì các phần tử đứng sau ($x_{i+1 \dots k}$) hoàn toàn có thể nhận lại giá trị j đó.
- Tất nhiên ta chỉ cần làm thao tác đáng dấu/bỏ đánh dấu trong thủ tục $Attempt(i)$ có $i \neq k$, bởi khi $i = k$ thì tiếp theo chỉ có in kết quả chứ không cần phải chọn thêm phần tử nào nữa.

Input

Hai số nguyên dương n, k $(1 \leq k \leq n \leq 100)$.

Output

Các chỉnh hợp không lặp chập k của tập $\{1,2, \dots, n\}$

Sample Input	Sample Output
3 2	(1, 2) (1, 3) (2, 1) (2, 3) (3, 1) (3, 2)

💻 Thuật toán quay lui liệt kê các chỉnh hợp không lặp chập k

```
program ArrangementEnumeration;
const
  max = 100;
var
  x: array[1..max] of LongInt;
  Free: array[1..max] of Boolean;
  n, k: LongInt;

procedure PrintResult; //Thủ tục in cấu hình tìm được
var
  i: LongInt;
begin
  Write('(');
```

```
  for i := 1 to k do
    begin
      Write(x[i]);
      if i < k then Write(', ');
    end;
  WriteLn(')');
end;

procedure Attempt(i: LongInt); //Thử các cách chọn x[i]
var
  j: LongInt;
begin
  for j := 1 to n do
    if Free[j] then //Chỉ xét những giá trị j còn tự do
      begin
        x[i] := j;
        if i = k then PrintResult //Nếu đã chọn được đến x[k] thì in kết quả
        else
          begin
            Free[j] := False; //Đánh dấu: j đã bị chọn
            Attempt(i + 1); //Attempt(i + 1) sẽ chỉ xét những giá trị còn tự do gán cho x[i+1]
            Free[j] := True; //Bỏ đánh dấu, sắp tới sẽ thử một cách chọn khác của x[i]
          end;
      end;
end;

begin
  ReadLn(n, k);
  FillChar(Free[1], n * SizeOf(Boolean), True);
  Attempt(1); //Khởi động thuật toán quay lui
end.
```

Khi $k = n$ thì đây là chương trình liệt kê hoán vị.

13.3.5. Liệt kê các cách phân tích số

Cho một số nguyên dương n, hãy tìm tất cả các cách phân tích số n thành tổng của các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là 1 cách và chỉ được liệt kê một lần.

Ta sẽ dùng thuật toán quay lui để liệt kê các nghiệm, mỗi nghiệm tương ứng với một dãy x, để tránh sự trùng lặp khi liệt kê các cách phân tích, ta đưa thêm ràng buộc: dãy x phải có thứ tự không giảm: $x_1 \le x_2 \le \cdots$

Thuật toán quay lui được cài đặt bằng thủ tục đệ quy $Attempt(i)$: thử các giá trị có thể nhận của x_i, mỗi khi thử xong một giá trị cho x_i, thủ tục sẽ gọi đệ quy $Attempt(i+1)$ để thử các giá trị có thể cho x_{i+1}. Trước mỗi bước thử các giá trị cho x_i, ta lưu trữ $m = \sum_{j=1}^{i-1} x_j$ là tổng của tất cả các phần tử đứng trước x_i: $x_{1\ldots i-1}$ và thử đánh giá miền giá trị mà x_i có thể nhận.

Rõ ràng giá trị nhỏ nhất mà x_i có thể nhận chính là x_{i-1} vì dãy x có thứ tự không giảm (Giả sử rằng có thêm một phần tử $x_0 = 1$, phần tử này không tham gia vào việc liệt kê cấu hình mà chỉ dùng để hợp thức hoá giá trị cận dưới của x_1)

Nếu x_i chưa phải là phần tử cuối cùng, tức là sẽ phải chọn tiếp ít nhất một phần tử $x_{i+1} \geq x_i$ nữa mà việc chọn thêm x_{i+1} không làm cho tổng vượt quá n. Ta có:

$$\begin{aligned} n &\geq \sum_{j=1}^{i-1} x_j + x_i + x_{i+1} \\ &= m + x_i + x_{i+1} \\ &\geq m + 2x_i \end{aligned} \tag{13.2}$$

Tức là nếu x_i chưa phải phần tử cuối cùng (cần gọi đệ quy chọn tiếp x_i) thì giá trị lớn nhất x_i có thể nhận là $\left\lfloor \frac{n-m}{2} \right\rfloor$, còn dĩ nhiên nếu x_i là phần tử cuối cùng thì bắt buộc x_i phải bằng $n - m$.

Vậy thì thủ tục $Attempt(i)$ sẽ gọi đệ quy $Attempt(i+1)$ để tìm tiếp khi mà giá trị x_i được chọn còn cho phép chọn thêm một phần tử khác lớn hơn hoặc bằng nó mà không làm tổng vượt quá n: $x_i \leq \left\lfloor \frac{n-m}{2} \right\rfloor$. Ngược lại, thủ tục này sẽ in kết quả ngay nếu x_i mang giá trị đúng bằng số thiếu hụt của tổng $i-1$ phần tử đầu so với n. Ví dụ đơn giản khi $n = 10$ thì thử $x_1 \in \{6,7,8,9\}$ là việc làm vô nghĩa vì như vậy cũng không ra nghiệm mà cũng không chọn tiếp x_2 được nữa.

Với giá trị khởi tạo $m := 0$ và $x_0 := 1$, thuật toán quay lui sẽ được khởi động bằng lời gọi $Attempt(1)$ và hoạt động theo cách sau:

- Với mỗi giá trị j: $x_{i-1} \leq j \leq \left\lfloor \frac{n-m}{2} \right\rfloor$, thử gán $x_i := j$, cập nhật $m := m + j$, sau đó gọi đệ quy tìm tiếp, sau khi đã thử xong các giá trị có thể cho x_{i+1}, biến m được phục hồi lại như cũ $m := m - j$ trước khi thử gán một giá trị khác cho x_i.
- Cuối cùng gán $x_i := n - m$ và in kết quả ra dãy $x_{1..i}$.

Input

Số nguyên dương $n \leq 100$

Output

Các cách phân tích số n.

Sample Input	Sample Output
6	6 = 1+1+1+1+1+1 6 = 1+1+1+1+2 6 = 1+1+1+3 6 = 1+1+2+2 6 = 1+1+4 6 = 1+2+3 6 = 1+5 6 = 2+2+2 6 = 2+4 6 = 3+3 6 = 6

Thuật toán quay lui liệt kê các cách phân tích số

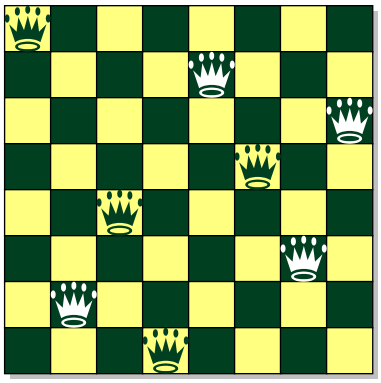
```
program NumberAnalysis;
const
  max = 100;
var
  x: array[0..max] of LongInt;
  n, m: LongInt;

procedure Init; //Khởi tạo
begin
  m := 0;
  x[0] := 1;
end;

procedure PrintResult(k: LongInt); //In kết quả ra dãy x[1..k]
var
  i: LongInt;
begin
  Write(n, ' = ');
  for i := 1 to k - 1 do Write(x[i], '+');
  WriteLn(x[k]);
end;

procedure Attempt(i: LongInt); //Thuật toán quay lui
var
  j: LongInt;
begin
  for j := x[i - 1] to (n - m) div 2 do //Trường hợp còn chọn tiếp x[i+1]
    begin
      x[i] := j; //Thử đặt x[i]
      m := m + j; //Cập nhật tổng m
      Attempt(i + 1); //Chọn tiếp
      m := m - j; //Phục hồi tổng m
    end;
  x[i] := n - m; //Nếu x[i] là phần tử cuối thì nó bắt buộc phải là n-m
  PrintResult(i); //In kết quả
end;

begin
  ReadLn(n);
  Init;
  Attempt(1); //Khởi động thuật toán quay lui
```

```
end.
```

Bây giờ ta xét tiếp một ví dụ kinh điển của thuật toán quay lui…

13.3.6. Bài toán xếp hậu

Xét bàn cờ tổng quát kích thước $n \times n$. Một quân hậu trên bàn cờ có thể ăn được các quân khác nằm tại các ô cùng hàng, cùng cột hoặc cùng đường chéo. Hãy tìm các xếp n quân hậu trên bàn cờ sao cho không quân nào ăn quân nào. Ví dụ một cách xếp với $n = 8$ được chỉ ra trong Hình 13-2.



Hình 13-2. Một cách xếp 8 quân hậu lên bàn cờ 8×8

Nếu đánh số các hàng từ trên xuống dưới theo thứ tự từ 1 tới n, các cột từ trái qua phải theo thứ tự từ 1 tới n. Thì khi đặt n quân hậu lên bàn cờ, mỗi hàng phải có đúng 1 quân hậu (hậu ăn được ngang), ta gọi quân hậu sẽ đặt ở hàng 1 là quân hậu 1, quân hậu ở hàng 2 là quân hậu 2… quân hậu ở hàng n là quân hậu n. Vậy một nghiệm của bài toán sẽ được biết khi ta tìm ra được vị trí cột của những quân hậu.

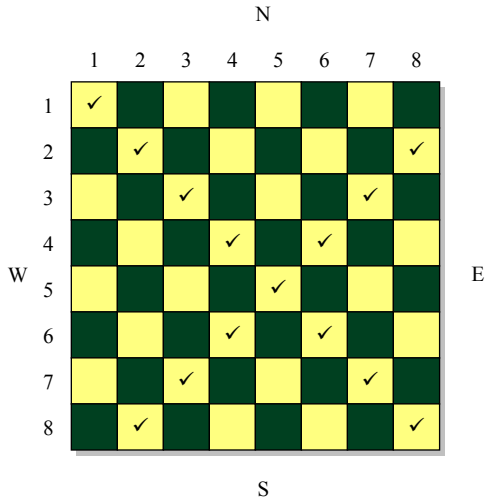
Định hướng bàn cờ theo 4 hướng: Đông (Phải), Tây (Trái), Nam (Dưới), Bắc (Trên). Một quân hậu ở ô (x, y) (hàng x, cột y) sẽ khống chế.

- Toàn bộ hàng x
- Toàn bộ cột y
- Toàn bộ các ô (i, j) thỏa mãn đẳng thức $i + j = x + y$. Những ô này nằm trên một đường chéo theo hướng Đông Bắc-Tây Nam (ĐB-TN).
- Toàn bộ các ô (i, j) thỏa mãn đẳng thức $i - j = x - y$. Những ô này nằm trên một đường chéo Đông Nam-Tây Bắc (ĐN-TB)

Từ những nhận xét đó, ta có ý tưởng đánh số các đường chéo trên bàn cờ.

- Với mỗi hằng số α: $2 \le \alpha \le 2n$. Tất cả các ô (i, j) trên bàn cờ thỏa mãn $i + j = \alpha$ nằm trên một đường chéo ĐB-TN, gọi đường chéo này là đường chéo ĐB-TN mang chỉ số α

- Với mỗi hằng số $\beta: 1-n \le \beta \le n-1$. Tất cả các ô (i,j) trên bàn cờ thỏa mãn $i-j=\beta$ nằm trên một đường chéo ĐN-TB, gọi đường chéo này là đường chéo ĐN-TB mang chỉ số β



Hình 13-3. Đường chéo ĐB–TN mang chỉ số 10 và đường chéo ĐN–TB mang chỉ số 0

Chúng ta sẽ sử dụng ba mảng logic để đánh dấu:

- Mảng $a_{1\ldots n}$. $a_j = True$ nếu như cột i còn tự do, $a_j = False$ nếu như cột j đã bị một quân hậu khống chế.
- Mảng $b_{2\ldots 2n}$. $b_\alpha = True$ nếu như đường chéo ĐB-TN thứ α còn tự do, $b_\alpha = False$ nếu như đường chéo đó đã bị một quân hậu khống chế.
- Mảng $c_{1-n\ldots n-1}$. $c_\beta = True$ nếu như đường chéo ĐN-TB thứ β còn tự do, $c_\beta = False$ nếu như đường chéo đó đã bị một quân hậu khống chế.

Ban đầu cả 3 mảng đánh dấu đều mang giá trị $True$. (Chưa có quân hậu nào trên bàn cờ, các cột và đường chéo đều tự do)

Thuật toán quay lui:

Xét tất cả các cột, thử đặt quân hậu 1 vào một cột, với mỗi cách đặt như vậy, xét tất cả các cách đặt quân hậu 2 không bị quân hậu 1 ăn, lại thử 1 cách đặt và xét tiếp các cách đặt quân hậu 3…Mỗi cách đặt được đến quân hậu n cho ta 1 nghiệm.

- Khi chọn vị trí cột j cho quân hậu thứ i, ta phải chọn ô (i,j) không bị các quân hậu đặt trước đó ăn, tức là phải chọn j thỏa mãn: cột j còn tự do: $a_j = True$, đường chéo ĐB-TN chỉ số $i+j$ còn tự do: $b_{i+j} = True$, đường chéo ĐN-TB chỉ số $i-j$ còn tự do; $c_{i-j} = True$.

- Khi thử đặt được quân hậu vào ô (i, j), nếu đó là quân hậu cuối cùng $(i = n)$ thì ta có một nghiệm. Nếu không:
  - Trước khi gọi đệ quy tìm cáchđ ặt quân hậu thứ $i + 1$, ta đánh dấu cột và 2 đường chéo bị quân hậu vừa đặt khống chế: $a_j = b_{i+j} = c_{i-j} := False$ để các lần gọi đệ quy tiếp sau chọn cách đặt các quân hậu kế tiếp sẽ không chọn vào những ô bị quân hậu vừa đặt khống chế.
  - Sau khi gọi đệ quy tìm cách đặt quân hậu thứ $i + 1$, có nghĩa là sắp tới ta lại thử một cách đặt khác cho quân hậu i, ta bỏ đánh dấu cột và 2 đường chéo vừa bị quân hậu vừa thử đặt khống chế $a_j = b_{i+j} = c_{i-j} := True$ tức là cột và 2 đường chéo đó lại thành tự do, bởi khi đã đặt quân hậu i sang vị trí khác rồi thì trên cột j và 2 đường chéo đó hoàn toàn có thể đặt một quân hậu khác

Hãy xem lại trong các chương trình liệt kê chỉnh hợp không lặp và hoán vị về kỹ thuật đánh dấu. Ở đây chỉ khác với liệt kê hoán vị là: liệt kê hoán vị chỉ cần một mảng đánh dấu xem giá trị có tự do không, còn bài toán xếp hậu thì cần phải đánh dấu cả 3 thành phần: Cột, đường chéo ĐB–TN, đường chéo ĐN–TB. Trường hợp đơn giản hơn: Yêu cầu liệt kê các cách đặt n quân xe lên bàn cờ $n \times n$ sao cho không quân nào ăn quân nào chính là bài toán liệt kê hoán vị.

Input

Số nguyên dương $n \leq 100$

Output

Một cách đặt các quân hậu lên bàn cờ $n \times n$

Sample Input	Sample Output
8	(1, 1) (2, 5) (3, 8) (4, 6) (5, 3) (6, 7) (7, 2) (8, 4)

💻 Thuật toán quay lui giải bài toán xếp hậu

```
{$MODE OBJFPC}
{$INLINE ON}
program NQueens;
const
  max = 100;
var
  n: LongInt;
```

```
  x: array[1..max] of LongInt;
  a: array[1..max] of Boolean;
  b: array[2..2 * max] of Boolean;
  c: array[1 - max..max - 1] of Boolean;
  Found: Boolean;

procedure PrintResult; //In kết quả mỗi khi tìm ra nghiệm
var
  i: LongInt;
begin
  for i := 1 to n do WriteLn('(', i, ', ', x[i], ') ');
  Found := True;
end;

//Kiểm tra ô (i, j) còn tự do hay đã bị một quân hậu khống chế?
function IsFree(i, j: LongInt): Boolean; inline;
begin
  Result := a[j] and b[i + j] and c[i - j];
end;

//Đánh dấu / bỏ đánh dấu một ô (i, j)
procedure SetFree(i, j: LongInt; Enabled: Boolean); inline;
begin
  a[j] := Enabled;
  b[i + j] := Enabled;
  c[i - j] := Enabled;
end;

procedure Attempt(i: LongInt); //Thử các cách đặt quân hậu i vào hàng i
var
  j: LongInt;
begin
  for j := 1 to n do //Xét tất cả các cột
    if IsFree(i, j) then //Tìm vị trí đặt chưa bị khống chế
      begin
        x[i] := j; //Thử đặt vào ô (i, j)
        if i = n then
          begin
            PrintResult; //Đặt đến con hậu n thì in ra 1 nghiệm
            Exit;
          end
        else
          begin
            SetFree(i, j, False); //Đánh dấu
            Attempt(i + 1); //Thử các cách đặt quân hậu thứ i + 1
            if Found then Exit;
            SetFree(i, j, True); //Bỏ đánh dấu
          end;
      end;
end;

begin
  ReadLn(n);
  //Đánh dấu tất cả các cột và đường chéo là tự do
  FillChar(a[1], n * SizeOf(Boolean), True);
  FillChar(b[2], (n + n - 1) * SizeOf(Boolean), True);
  FillChar(c[1 - n], (n + n - 1) * SizeOf(Boolean), True);
```

```
  Found := False;
  Attempt(1); //Khởi động thuật toán quay lui
end.
```

Thuật toán dùng một biến $Found$ làm cờ báo xem đã tìm ra nghiệm hay chưa, nếu $Found = True$, thuật toán quay lui sẽ ngưng ngay quá trình tìm kiếm.

Một sai lầm dễ mắc phải là chỉ đặt lệnh dừng thuật toán quay lui trong phép thử

```
if i = n then
  begin
    PrintResult;
    Exit;
  end;
```

Nếu làm như vậy lệnh Exit chỉ có tác dụng trong thủ tục $Attempt(n)$, muốn ngưng cả một dây chuyền đệ quy, cần phải thoát liền một loạt các thủ tục đệ quy: $Attempt(n), Attempt(n-1), \ldots, Attempt(1)$. Đặt lệnh Exit vào sau lời gọi đệ quy chính là đặt lệnh Exit cho cả một dây chuyền đệ quy mỗi khi tìm ra nghiệm.

Một số môi trường lập trình có lệnh dừng cả chương trình (như ở chương trình trên chúng ta thay lệnh Exit bằng lệnh Halt cũng đúng). Nhưng nếu thuật toán quay lui chỉ là một phần trong chương trình, sau khi thực hiện thuật toán sẽ còn phải làm nhiều việc khác nữa, khi đó lệnh ngưng vô điều kiện cả chương trình ngay khi tìm ra nghiệm là không được phép. Cài đặt dãy Exit là một cách làm chính thống để ngưng dây chuyền đệ quy.

13.4. Kỹ thuật nhánh cận

Có một lớp bài toán đặt ra trong thực tế yêu cầu tìm ra *một* nghiệm thoả mãn một số điều kiện nào đó, và nghiệm đó là *tốt nhất* theo một chỉ tiêu cụ thể, đó là lớp bài toán *tối ưu* (*optimization*). Nghiên cứu lời giải các lớp bài toán tối ưu thuộc về lĩnh vực quy hoạch toán học. Tuy nhiên cũng cần phải nói rằng trong nhiều trường hợp chúng ta chưa thể xây dựng một thuật toán nào thực sự hữu hiệu để giải bài toán, mà cho tới nay việc tìm nghiệm của chúng vẫn phải dựa trên mô hình *liệt kê* toàn bộ các cấu hình có thể và đánh giá, tìm ra cấu hình tốt nhất. Việc tìm phương án tối ưu theo cách này còn có tên gọi là *vét cạn (exhaustive search)*. Chính nhờ kỹ thuật này cùng với sự phát triển của máy tính điện tử mà nhiều bài toán khó đã tìm thấy lời giải.

Việc liệt kê cấu hình có thể cài đặt bằng các phương pháp liệt kê: Sinh tuần tự và tìm kiếm quay lui. Dưới đây ta sẽ tìm hiểu kỹ hơn cơ chế của thuật toán quay lui để giới thiệu một phương pháp hạn chế không gian duyệt.

Mô hình thuật toán quay lui là tìm kiếm trên một cây phân cấp. Nếu giả thiết rằng mỗi nút nhánh của cây chỉ có 2 nút con thì cây có độ cao n sẽ có tới 2^n nút lá, con số này lớn

hơn rất nhiều lần so với kích thước dữ liệu đầu vào n. Chính vì vậy mà nếu như ta có thao tác thừa trong việc chọn x_i thì sẽ phải trả giá rất lớn về chi phí thực thi thuật toán bởi quá trình tìm kiếm lòng vòng vô nghĩa trong các bước chọn kế tiếp $x_{i+1}, x_{i+2}, \ldots$ Khi đó, một vấn đề đặt ra là trong quá trình liệt kê lời giải ta cần tận dụng những thông tin đã tìm được để *loại bỏ sớm những phương án chắc chắn không phải tối ưu*. Kỹ thuật đó gọi là kỹ thuật đánh giá nhánh cận (Branch-and-bound) trong tiến trình quay lui.

13.4.1. Mô hình kỹ thuật nhánh cận

Dựa trên mô hình thuật toán quay lui, ta xây dựng mô hình sau:

```
procedure Init;
begin
   «Khởi tạo một cấu hình bất kỳ BestSolution»;
end;

//Thủ tục này thử chọn cho x[i] tất cả các giá trị nó có thể nhận
procedure Attempt(i: LongInt);
begin
  for ⟨Mọi giá trị v có thể gán cho x[i]⟩ do
    begin
      «Thử đặt x[i] := v»;
      if «Còn hi vọng tìm ra cấu hình tốt hơn BestSolution» then
        if «x[i] là phần tử cuối cùng trong cấu hình» then
          «Cập nhật BestSolution»
        else
          begin
            «Ghi nhận việc thử x[i] := v nếu cần»;
            Attempt(i + 1); //Gọi đệ quy, chọn tiếp x[i + 1]
            «Bỏ ghi nhận việc đã thử cho x[i] := v (nếu cần)»;
          end;
    end;
end;

begin
  Init;
  Attempt(1);
  «Thông báo cấu hình tối ưu BestSolution»;
end.
```

Kỹ thuật nhánh cận thêm vào cho thuật toán quay lui khả năng đánh giá theo từng bước, nếu tại bước thứ i, giá trị thử gán cho x_i không có hi vọng tìm thấy cấu hình tốt hơn cấu hình $BestSolution$ thì thử giá trị khác ngay mà không cần phải gọi đệ quy tìm tiếp hay ghi nhận kết quả nữa. Nghiệm của bài toán sẽ được làm tốt dần, bởi khi tìm ra một cấu hình mới tốt hơn $estSolution$, ta không in kết quả ngay mà sẽ cập nhật $BestSolution$ bằng cấu hình mới vừa tìm được.

Dưới đây ta sẽ khảo sát một vài kỹ thuật đánh giá nhánh cận qua các bài toán cụ thể.

13.4.2. Bài toán xếp ba lô

Cho n đồ vật, đồ vật thứ i có trọng lượng là w_i và giá trị là v_i $(w_i, v_i \in \mathbb{R}^+)$. Cho một balô có giới hạn trọng lượng là m, hãy chọn ra một số đồ vật cho vào balô sao cho tổng trọng lượng của chúng không vượt quá m và tổng giá trị của chúng là lớn nhất có thể.

Bài toán này (Knapsack) là một bài toán nổi tiếng về độ khó: Hiện chưa có một lời giải hiệu quả cho nghiệm tối ưu trong trường hợp tổng quát. Có rất nhiều cố gắng để giải quyết bài toán Knapsack như các thuật toán gần đúng dựa trên chiến lược tham lam (greedy), hay những thuật toán tốt cho những trường hợp đặt biệt dựa trên quy hoạch động (khi $w_{1\ldots n}$ và m là những số nguyên tương đối nhỏ). Dưới đây ta sẽ xây dựng thuật toán quay lui và kỹ thuật nhánh cận để giải bài toán Knapsack.

❑ Mô hình duyệt

Với một đồ vật bất kỳ thì có hai khả năng: Hoặc đồ vật đó được chọn hoặc đồ vật đó không được chọn. Bằng cách mô hình hoá bài toán như vậy, một cách chọn các đồ vật có thể biểu diễn dưới dạng một dãy nhị phân độ dài n. Tuy nhiên nếu duyệt qua toàn bộ các dãy nhị phân độ dài n và đánh giá tìm nghiệm tối ưu thì số cấu hình cần duyệt có thể là một con số khổng lồ. Chỉ với $n = 50$ sẽ có 2^{50} khả năng cần thử, giả sử trong một giây máy có thể thử 1 triệu khả năng thì cũng phải mất ≈ 35 năm mới có thể tìm ra nghiệm. Ta phải tìm cách loại bỏ sớm những khả năng chắc chắn không phải tối ưu để có thể thực thi với dữ liệu lớn.

❑ Lập hàm cận bằng cách "chơi sai luật"

Với mỗi đồ vật i, gọi mật độ (giá trị riêng) của đồ vật đó là v_i/w_i. Có một thuật toán tham lam cho kết quả tương đối tốt, tuy không phải lúc nào cũng cho phương án tối ưu: Xét lần lượt các đồ vật theo thứ tự giảm dần của mật độ, với mỗi đồ vật xét theo thứ tự đó, ta sẽ đưa ngay vào balô nếu không bị vượt quá giới hạn trọng lượng.

Ta sẽ sửa đổi thuật toán tham lam này một chút để lập một hàm tính giá cận trên cho mỗi phép thử. Giả sử các đồ vật đã được sắp xếp theo thứ tự giảm dần của mật độ. Xét bài toán cần chọn các đồ vật trong số $k, k+1, \ldots, n$ với giới hạn trọng lượng $Limit$. Vì mật độ của các đồ vật đã được sắp giảm dần, ta có:

$$\frac{v_k}{w_k} \geq \frac{w_{k-1}}{v_{k-1}} \geq \cdots \geq \frac{v_n}{w_n}$$

Bây giờ ta sẽ xét lần lượt các đồ vật $k, k+1, \ldots, n$ đưa vào balô …

- Nếu việc lấy đồ vật đang xét i không làm vượt quá giới hạn trọng lượng tối đa $Limit$ thì đưa ngay đồ vật i vào balô, tổng giá trị các đồ vật trong balô được tăng lên v_i.

- Nếu việc đưa đồ vật đang xét i vào làm vượt quá giới hạn trọng lượng thì đưa một phần có trọng lượng $Quantity$ của vật đó vào ba lô sao cho vừa đúng với giới hạn trọng lượng $Limit$, tổng giá trị các đồ vật trong ba lô được tăng lên $Quantity \times \frac{v_i}{w_i}$. Thuật toán dừng.

Kết thúc thuật toán, gọi tổng giá trị lấy được là $Estimate(k, Limit)$

Ví dụ với 5 đồ vật đã được sắp xếp theo chiều giảm dần của mật độ:

i	1	2	3	4	5
v_i	2	2	2	2	2
w_i	1	2	3	4	5

Với $k = 1$, giới hạn trọng lượng là 8, ta sẽ lấy nguyên đồ vật 1, nguyên đồ vật 2, nguyên đồ vật 3 và một nửa đồ vật 4. Được đúng trọng lượng 8 và giá trị lấy được là $Estimate(1,8) = 7$.

Rõ ràng phương án chọn các đồ vật như vậy là sai luật (phải lấy nguyên đồ vật chứ không được lấy một phần), nhưng hãy thử xét lại thuật toán và giá trị lấy được, ta có nhận xét: Cho dù phương án chọn đúng luật có tốt như thế nào chăng nữa, tổng giá trị các đồ vật được chọn không thể vượt qua con số $Estimate(k, Limit)$ của cách chọn sai luật.

Hàm $Estimate(k, Limit)$ ước lượng giá trị cận trên có thể lấy được trong bài toán chọn các đồ vật $k, k+1, \ldots, n$ với giới hạn trọng lượng $Limit$ có thể viết như sau:

```
function Estimate(k: Integer; Limit: Real): Real;
var
  i: Integer;
  Quantity: Real;
begin
  Result := 0;
  for i := k to n do
    begin
      if w[i] ≤ Limit then Quantity := w[i]
      else Quantity := Limit;
      Result := Result + Quantity / w[i] * v[i];
      Limit := Limit - Quantity;
      if Limit = 0 then Break;
    end;
end;
```

❑ Đánh giá tương quan giữa các phần tử của cấu hình

Sau khi đặt trạng thái cho một đồ vật (chọn hay không chọn), có thể dùng hàm cận trên $Estimate$ để ước lượng xem có nên thử với những đồ vật tiếp theo nữa hay không. Tuy nhiên, không phải lúc nào chúng ta cũng phải thử hai trạng thái của một đồ vật mà có thể

dựa vào trạng thái của những đồ vật đã xét trước đó để xác định sớm những đồ vật chắc chắn không được chọn.

Sắp xếp các đồ vật theo thứ tự giảm dần của mật độ và biểu diễn một cách chọn các đồ vật dưới dạng dãy $x_{1\ldots n}$, trong đó $x_i = True$ nếu đồ vật i được chọn vào ba lô, $x_i = False$ nếu đồ vật i không được chọn vào ba lô.

Với hai đồ vật a và b, ta nói đồ vật a tốt hơn đồ vật b (hay đồ vật b tồi hơn đồ vật a) nếu $a < b$, đồng thời đồ vật a không nặng hơn và không rẻ hơn đồ vật b:

$$(a < b) \text{ and } (w_a \leq w_b) \text{ and } (v_a \geq v_b)$$

Điều này có nghĩa là nếu một phương án nào đó có chọn đồ vật b mà không chọn đồ vật a thì ta có thể thay đồ vật b bằng đồ vật a để được một phương án khác ít ra là không tệ hơn phương án đang có. Với điều kiện $a < b$, quan hệ "tốt hơn" là quan hệ phản đối xứng và không phản xạ, tức là không thể có trường hợp đồ vật a tốt hơn đồ vật b đồng thời đồ vật b tốt hơn đồ vật a.

Đánh giá này cho ta thêm một tiêu chuẩn để hạn chế không gian duyệt: Trong quá trình duyệt quay lui, ta sẽ không bao giờ thử chọn một đồ vật đưa vào ba lô nếu trước đó có đồ vật tốt hơn nó mà không được chọn. Tiêu chuẩn này có thể viết bằng hàm $Selectable(p, q)$, $(p < q)$: Cho biết có thể nào chọn đồ vật q trong điều kiện ta đã biết trạng thái của các đồ vật từ 1 tới p (chọn hay không chọn):

```
function Selectable(p, q: Integer): Boolean;
var
  i: Integer;
begin
  for i := 1 to p do
    if not x[i] and //Vật i không được chọn và i tốt hơn q
      (obj[i].w <= obj[q].w) and (obj[i].v >= obj[q].v) then
        Exit(False); //Kết luận q chắc chắn không được chọn
  Result := True;
end;
```

Hàm $Selectable$ sẽ được dùng trong thủ tục quay lui, đồng thời tích hợp vào hàm $Estimate$ để có một đánh giá cận chặt hơn.

❑ Cài đặt

Ta sẽ thử lần lượt với các đồ vật từ 1 tới n. Khởi tạo một phương án chọn các đồ vật bất kỳ có tổng giá trị thu được là $MaxV$. Thủ tục $Attempt(i)$ (thử các giá trị có thể nhận của x_i) trước hết sẽ đánh giá xem có hy vọng tìm ra phương án thu được nhiều hơn $MaxV$ hay không, nếu không có hy vọng thì sẽ thủ tục sẽ thoát ra ngay lùi về

$Attempt(i-1)$ để thử cách chọn khác cho x_{i-1}. Khi cài đặt ta có thể khởi tạo $MaxV := 0$ và cập nhật dần phương án tối ưu.

Mỗi khi thử đặt một giá trị cho x_i, thủ tục $Attempt(i)$ sẽ tính lại $SumW$ là tổng trọng lượng và $SumV$ là tổng giá trị các đồ vật đã được chọn cho tới bước này. Việc cập nhật phương án tối ưu sẽ được thực hiện khi xét tới đồ vật cuối cùng ($i = n$) và tại bước cuối cùng này ta có $SumV > MaxV$.

Input

Dòng 1 chứa số nguyên dương $m \leq 1000$ và số thực dương m cách nhau một dấu cách

n dòng tiếp theo, dòng thứ i ghi hai số thực dương w_i, v_i cách nhau một dấu cách.

Output

Phương án chọn các đồ vật có tổng trọng lượng $\leq m$ và tổng giá trị lớn nhất có thể.

Sample Input	Sample Output
5 11.0 2.9 3.3 4.0 4.4 5.0 4.4 9.0 9.9 4.1 4.4	Selected objects: 1. Object 1: Weight = 2.90; Value = 3.30 2. Object 2: Weight = 4.00; Value = 4.40 3. Object 5: Weight = 4.10; Value = 4.40 Total weight: 11.00 Total value : 12.10

Bài toán xếp balô

```
{$MODE OBJFPC}
program Knapsack;

const
  max = 1000;

type
  TObj = record //Kiểu dữ liệu cho một đồ vật
    w, v: Real; //Trọng lượng và giá trị
    id: Integer; //Chỉ số
  end;

var
  obj: array[1..max] of TObj;
  x, Best: array[1..max] of boolean;
  SumW, SumV: Real;
  MaxV: Real;
  n: Integer;
  m: Real;

procedure Enter; //Nhập dữ liệu
var
  i: Integer;
begin
  ReadLn(n, m);
```

```
  for i := 1 to n do
    with obj[i] do
      begin
        ReadLn(w, v);
        id := i;
      end;
end;

//Định nghĩa toán tử: đồ vật x < đồ vật y nếu mật độ x > mật độ y, toán tử này dùng để sắp xếp
operator < (const x, y: TObj): Boolean;
begin
  Result := x.v / x.w > y.v / y.w
end;

procedure ShellSort; //Sắp xếp các đồ vật theo mật độ giảm dần
var
  temp: Tobj;
  i, j, step: Integer;
begin
  step := n div 2;
  while step > 0 do
    begin
      for i := step + 1 to n do
        begin
          temp := obj[i]; j := i - step;
          while (j > 0) and (temp < obj[j]) do
            begin
              obj[j + step] := obj[j];
              Dec(j, step);
            end;
          obj[j + step] := temp;
        end;
      if step = 2 then step := 1
      else step := step * 10 div 22;
    end;
end;

procedure Init; //Khởi tạo
begin
  SumW := 0; //Một ba lô rỗng
  SumV := 0;
  MaxV := -1; //Một giá trị tồi hơn mọi phương án có thể
end;

//Đánh giá xem có thể chọn đồ vật q hay không khi đã quyết định xong với các đồ vật 1..q
function Selectable(p, q: Integer): Boolean;
var
  i: Integer;
begin
  for i := 1 to p do
    if not x[i] and //Đồ vật i không được chọn và i tốt hơn q
      (obj[i].w <= obj[q].w) and (obj[i].v >= obj[q].v) then
        Exit(False);
  Result := True;
end;

//Ước lượng giá trị cận trên của phép chọn
```

```
function Estimate(k: Integer; Limit: Real): Real;
var
  i: Integer;
  Quantity: Real;
begin
  Result := 0;
  for i := k to n do
    if Selectable(k - 1, i) then
      begin
        if obj[i].w <= Limit then Quantity := obj[i].w
        else Quantity := Limit;
        Result := Result + Quantity / obj[i].w * obj[i].v;
        Limit := Limit - Quantity;
        if Limit = 0 then Break;
      end;
end;

procedure UpdateSolution; //Cập nhật phương án tối ưu mỗi khi tìm ra nghiệm tốt hơn
begin
  MaxV := SumV;
  Move(x, Best, n * SizeOf(Boolean));
end;

procedure Attempt(i: Integer); //Thuật toán quay lui
begin
  //Đánh giá xem có nên thử tiếp không, nếu không có hy vọng tìm ra nghiệm tốt hơn thì thoát ngay
  if SumV + Estimate(i, m - SumW) <= MaxV then
    Exit;
  //Điều kiện để chọn đồ vật i vào ba lô
  if (SumW + obj[i].w <= m) and Selectable(i - 1, i) then
    begin
      x[i] := True; //Thử chọn đồ vật i
      SumW := SumW + obj[i].w; //Cập nhật tổng trọng lượng đang có trong ba lô
      SumV := SumV + obj[i].v; //Cập nhật tổng giá trị đang có trong ba lô
      if i = n then //Nếu đã thử tới đồ vật n
        begin
          if SumV > MaxV then //và tìm được phương án tốt hơn
            UpdateSolution; //thì cập nhật kết quả
        end
      else //Nếu chưa tới đồ vật n
        Attempt(i + 1); //Thử đồ vật kế tiếp
      SumW := SumW - obj[i].w; //Sau khi thử chọn xong, bỏ đồ vật i khỏi ba lô
      SumV := SumV - obj[i].v; //phục hồi SumW và SumV như khi chưa chọn đồ vật i
    end;
  x[i] := False; //Thử không chọn đồ vật i
  if i = n then //Nếu đã thử tới đồ vật n
    begin
      if SumV > MaxV then //và tìm được phương án tốt hơn
        UpdateSolution //thì cập nhật kết quả
    end
  else //Nếu chưa tới đồ vật n
    Attempt(i + 1); //thử đồ vật kế tiếp
end;

procedure PrintResult; //In kết quả
var
  i, Count: Integer;
```

```
  TotalWeight: Real;
begin
  WriteLn('Selected objects: ');
  Count := 0;
  TotalWeight := 0;
  for i := 1 to n do
    if Best[i] then
      with obj[i] do
        begin
          Inc(Count);
          Write(Count, '. Object ', id, ': ');
          WriteLn('Weight = ', w:1:2, '; Value = ', v:1:2);
          TotalWeight := TotalWeight + w;
        end;
  WriteLn('Total weight: ', TotalWeight:1:2);
  WriteLn('Total value : ', MaxV:1:2);
end;

begin
  Enter; //Nhập dữ liệu
  ShellSort; //Sắp xếp theo chiều giảm dần của mật độ
  Init; //Khởi tạo
  Attempt(1); //Khởi động thuật toán quay lui
  PrintResult; //In kết quả
end.
```

13.4.3. Bài toán dò mìn

Cho một bãi mìn dạng lưới ô vuông kích thước $m \times n$, trong đó các hàng được đánh số từ 1 tới m và các cột được đánh số từ 1 tới n. Trên mỗi ô có thể có chứa một quả mìn hoặc không, để biểu diễn bản đồ mìn đó, người ta có hai cách:

Cách 1: dùng bản đồ đánh dấu: sử dụng một lưới ô vuông kích thước $m \times n$, trên đó mỗi ô (i, j) được đánh dấu là có hay không có mìn (chẳng hạn dấu "+" tương ứng với ô có mìn và dấu "-" tương ứng với ô không có mìn)

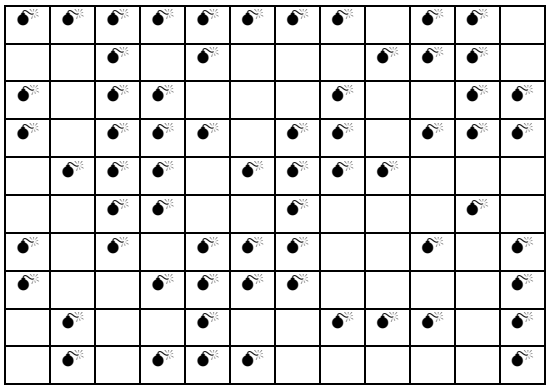
Cách 2: dùng bản đồ mật độ: sử dụng một lưới ô vuông kích thước $m \times n$, trên đó tại ô (i, j) ghi một số trong khoảng từ 0 đến 8 cho biết tổng số mìn trong các ô lân cận với ô (i, j) (ô lân cận với ô (i, j) là ô có chung với ô (i, j) ít nhất 1 đỉnh).

Giả thiết rằng hai bản đồ được ghi chính xác theo tình trạng mìn trên hiện trường.

Lúc cài bãi mìn người ta đã vẽ cả bản đồ đánh dấu và bản đồ mật độ, tuy nhiên sau một thời gian dài, khi người ta muốn gỡ mìn ra khỏi bãi thì vấn đề hết sức khó khăn bởi bản đồ đánh dấu đã bị thất lạc. Nhiệm vụ của bạn là hãy tái tạo lại bản đồ đánh dấu của bãi mìn từ bản đồ mật độ tương ứng.

1	3	3	4	3	3	2	2	4	4	3	2
3	6	5	7	4	4	4	4	4	5	5	4
1	5	4	6	4	3	3	3	5	6	6	4
2	6	6	6	4	4	5	5	5	3	4	3
2	4	6	6	5	4	5	5	3	4	4	3
2	5	5	5	5	6	5	5	3	3	2	2
1	4	3	6	5	6	4	3	1	1	4	2
2	4	3	4	5	6	4	4	4	3	5	2
3	2	4	5	6	6	4	2	2	1	4	2
2	1	3	2	3	2	2	2	3	2	3	1

→



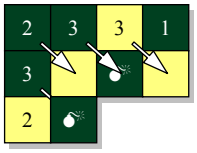
Bài toán dò mìn (Mine sweeper) cũng là một bài toán khó mà cho tới nay không có lời giải hiệu quả ngoài mô hình duyệt. Chúng ta sẽ từng bước khảo sát bài toán để đưa ra một thuật toán có thể thực hiện được với bãi mìn với kích thước hàng triệu ô.

❑ Mô hình duyệt

Ta mã hoá bản đồ mật độ của bãi mìn bằng ma trận $B = \{b_{ij}\}_{m\times n}$ trong đó $0 \le b_{ij} \le 8$. Bản đồ đánh dấu của bãi mìn có thể mã hoá dưới dạng một ma trận $A = \{a_{ij}\}_{m\times n}$, trong đó $a_{ij} = 1$ tương ứng với ô (i,j) có mìn và $a_{ij} = 0$ tương ứng với ô (i,j) không có mìn. Việc xác định bản đồ đánh dấu A có thể đưa về bài toán liệt kê tất cả các ma trận nhị phân kích thước $m \times n$ và tìm ma trận tương ứng với bản đồ mật độ B. Tuy nhiên nếu phải xét tất cả các ô và với mỗi ô phải thử hai giá trị có thể $\in \{0,1\}$ thì thuật toán này rất chậm và bất khả thi với dữ liệu lớn.

Ta có nhận xét đầu tiên giúp hạn chế đáng kể không gian duyệt: Bản đồ đánh dấu A hoàn toàn có thể khôi phục nếu ta xác định đúng các phần tử thuộc hàng 1 và cột 1 của nó.

Thật vậy, nếu ta xác định đúng được các phần tử thuộc hàng 1 và cột 1 của bản đồ mật độ thì ô $(1,1)$ chỉ còn đúng một ô lân cận với nó chưa xác định được tình trạng mìn là ô $(2,2)$, vậy từ số ghi trên ô $(1,1)$ của bản đồ mật độ ta có thể xác định chính xác tình trạng mìn của ô $(2,2)$. Tiếp tục, ta lại có ô $(1,2)$ chỉ còn đúng một ô lân cận với nó chưa xác định được tình trạng mìn là ô $(2,3)$ nên ta có thể xác định được tình trạng mìn của ô $(2,3)$ dựa vào tổng số mìn trong các ô lân cận ô $(1,2)$… cứ như vậy ta sẽ xác định được toàn bộ hàng 2 của bản đồ đánh dấu và bằng cách làm tương tự với các hàng khác, ta sẽ xác định được toàn bộ bản đồ đánh dấu của bãi mìn.



Hình 13-4. Phép nội suy tình trạng mìn

Nhận xét này làm giảm đáng kể chi phí cho quá trình duyệt: Thay vì phải xét và thử tất cả $m \times n$ ô, ta chỉ phải thử $m + n - 1$ ô nằm trên hàng 1 và cột 1 mà thôi, tình trạng các ô khác sẽ được suy ra từ bản đồ mật độ mà không cần dùng phép thử.

❑ Loại bỏ sớm các phương án chắc chắn không phải nghiệm

Bài toán dò mìn bây giờ trở thành bài toán xác định hàng 1 và cột 1 của bản đồ đánh dấu A. Với mỗi ô (i, j) nằm trên hàng 1 hoặc cột 1, thuật toán quay lui sẽ thử xác định $a_{ij} = 0$ (không có mìn) hay $a_{ij} = 1$ (có mìn), để tiện cài đặt, ta có thêm một trạng thái $a_{ij} = 2$ tương ứng với ô chưa biết tình trạng mìn (chưa thử đến).

Cấu hình cần liệt kê có dạng một dãy nhị phân độ dài $m + n - 1$ trong đó mỗi phần tử tương ứng với một ô thuộc hàng 1 hoặc cột 1. Sau khi thử xác định tình trạng mìn của một ô (0 hay 1), thuật toán sẽ kiểm tra cách đặt đó có xung đột với tình trạng mìn của những ô đã xác định tình trạng mìn rồi hay không. Nếu không có sự xung đột, thuật toán mới thử xác định tình trạng mìn của ô kế tiếp. Cách kiểm tra xung đột được thực hiện bằng phương pháp sau:

Lập một hàm $CountNeighbor(p)$ nhận vào một ô p và trả về ba biến đếm: $Count_0$ là số ô không có mìn, $Count_1$ là số ô có mìn và $Count_2$ là số ô chưa xác định tìm trạng mìn lân cận với ô p. Sau khi thử tình trạng mìn của một ô $q = (i, j)$ nào đó, ta sẽ xét tất cả các ô $p = (x, y)$ lân cận ô q và gọi hàm $CountNeighbor(p)$ xác định ba biến đếm tương ứng với ô p. Xung đột sẽ xảy ra nếu:

$$(b_{xy} > Count_1) \text{ hoặc } (b_{xy} < Count_1 + Count_2) \qquad (13.3)$$

Tức là xung đột xảy ra nếu số mìn quanh ô $p = (x, y)$ đã vượt quá số lượng được cho, hoặc cho dù những ô chưa xác định tình trạng còn lại đều có mìn cả thì số mìn cũng chưa đủ số lượng cần thiết. Phép kiểm tra xung đột bằng điều kiện này gọi là *phép kiểm tra lỗi tức thời.*

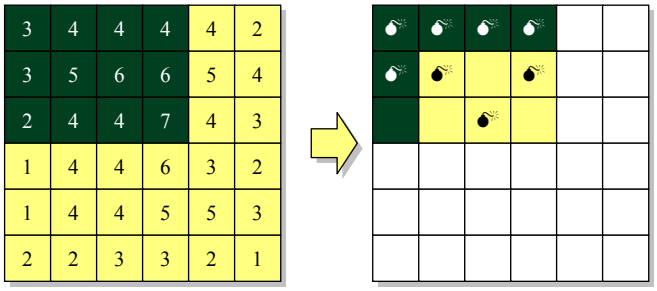
❑ Thứ tự duyệt là quan trọng

Bản đồ đánh dấu hoàn toàn có thể xác định nếu biết được chính xác hàng 1 và cột 1. Bằng lập luận tương tự, ta có nhận xét sau: Nếu biết được chính xác tình trạng mìn trên các ô thuộc hàng 1 từ ô $(1,1)$ đến ô $(1, y)$ đồng thời biết được chính xác tình trạng mìn

trên các ô thuộc cột 1 từ ô $(1,1)$ đến ô $(x,1)$ thì ta có thể suy ra được tình trạng mìn trên tất cả các ô thuộc hình chữ nhật $(1,1)-(x,y)$.

Phép suy diễn này được viết bằng hàm *Fill*, hàm này sẽ được gọi sau mỗi bước thử, khi mà thuật toán quay lui tìm ra một phần cấu hình. Hàm *Fill* trả về giá trị kiểu logic: True nếu phép suy diễn thành công và False nếu phép suy diễn thất bại.

Đến đây ta có thêm một tiêu chuẩn để loại bỏ sớm những phương án chắc chắn không tìm ra nghiệm: Thuật toán quay lui sẽ thử xác định tình trạng mìn của các ô trên hàng 1 và cột 1 theo một danh sách định trước, để sau mỗi phép thử ta xác định được tình trạng mìn của phần đầu hàng 1 và cột 1. Lúc này, hàm *Fill* sẽ được sử dụng để điền nốt trạng thái các ô nằm trong hình chữ nhật tương ứng. Nếu hàm *Fill* thành công, ta sẽ thử ô kế tiếp trong danh sách, còn nếu hàm *Fill* thất bại ta sẽ thử phương án khác ngay. Phép kiểm tra lỗi này gọi là *phép kiểm tra lỗi suy diễn*



Hình 13-5. Nếu biết được tình trạng mìn trên các ô thuộc phần đầu hàng 1 và phần đầu cột 1, ta có thể biết được tình trạng mìn trên toàn bộ các ô thuộc hình chữ nhật ở góc trên trái bản đồ đánh dấu

Có nhiều cách tạo danh sách các ô thuộc hàng 1 và cột 1, miễn sao sau mỗi phép thử thì tình trạng mìn của phần đầu hàng 1 và cột 1 được xác định. Chẳng hạn danh sách bắt đầu bằng các ô thuộc hàng 1 từ ô $(1,1)$ tới ô $(1,n)$ theo đúng thứ tự từ trái qua phải, tiếp theo là các ô thuộc cột 1 từ ô $(2,1)$ tới ô $(m,1)$ theo đúng thứ tự từ trên xuống dưới.

$$\begin{matrix} 1 & 2 & \dots & n \\ n+1 & & & \\ n+2 & & & \\ \vdots & & & \\ n+m-1 & & & \end{matrix}$$

Tuy nhiên nếu theo danh sách này, thì hàm *Fill* sẽ chỉ phát huy tác dụng từ bước duyệt ô thứ $n+1$ trở đi. Vì n phần tử đầu tiên của danh sách đều là các ô trên hàng 1 nên hàm *Fill* luôn thành công vì nó không phải điền thêm ô nào cả. Có nghĩa là nếu có một phép thử sai về tình trạng mìn của một ô nằm gần đầu hàng 1 thì có thể thuật toán sẽ phải đợi qua rất nhiều bước thử vô nghĩa mới quay về đặt lại tình trạng ô đó. Một cách làm khôn

ngoan hơn là định lại thứ tự duyệt các ô để tận dụng hàm $Fill$ loại bỏ sớm những phép thử sai. Thứ tự tốt để duyệt các ô là:

1 2 4 6
3
5
7

Tức là đầu tiên ta thử tình trạng mìn trên ô $(1,1)$. Các ô tiếp theo sẽ được thử theo thứ tự cứ một ô ở hàng 1 rồi lại đến một ô ở cột 1… Như vậy trong trường hợp m, n khá lớn thì bắt đầu từ ô thứ ba trở đi trong danh sách thử, hàm $Fill$ luôn luôn phải điền thêm một số ô khác và nếu nó không điền được (thất bại) ta có thể xác định sớm hơn những lỗi trong các phép thử vừa thực hiện.

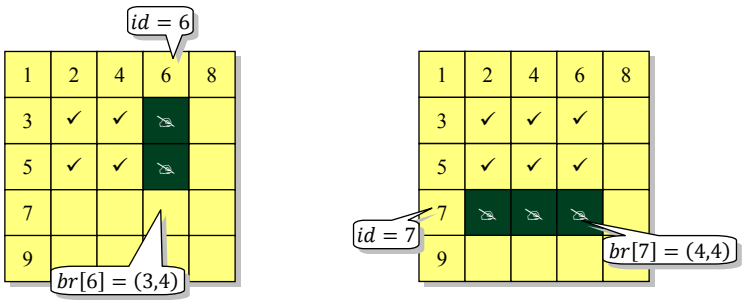
❑ Cài đặt

Để tiện cho các phép xử lý, bản đồ đánh dấu A sẽ được cho thêm hai hàng và hai cột tạo thành một đường viền bao quanh bản đồ. Như vậy các hàng của bản đồ A sẽ được đánh số từ 0 tới $m + 1$ và các cột của bản đồ A sẽ được đánh số từ 0 tới $n + 1$.

Vị trí mỗi ô được biểu diễn bằng một biến kiểu bản ghi $TPoint$ gồm hai trường số nguyên x, y tương ứng với chỉ số hàng và chỉ số cột của một ô. Danh sách các ô theo thứ tự duyệt được lưu trong mảng $List[1 \ldots m + n - 1]$. Tương ứng với mỗi ô $List[id]$ sẽ là một ô $br[id]$ chỉ ra vị trí góc phải dưới của hình chữ nhật sẽ được điền bằng hàm $Fill$ sau mỗi phép thử xác định tình trạng mìn trên ô $List[id]$. Ngoài ra có hàm sau:

Hàm $CanBe(p: TPoint; s: Integer): Boolean$, cho biết ô p có thể nhận trạng thái $s \in \{0,1\}$ hay không bằng phép kiểm tra lỗi tức thời.

Hàm $Fill(id: Integer): Boolean$, thực hiện phép suy diễn mỗi khi thử dò trạng thái mìn tới ô $List[id]$. Hàm trả về giá trị True nếu phép suy diễn thành công và trả về giá trị False nếu phép suy diễn thất bại. Để hàm hoạt động nhanh, ta sẽ chỉ điền trạng thái mìn vào những ô chưa được điền ở những bước trước, điều này có thể thực hiện thông qua các giá trị $br[.]$: Nếu $List[id]$ là ô nằm trên hàng 1 thì hàm $Fill$ sẽ điền thêm các ô trên cột $br[id].y$ từ hàng 2 tới hàng $br[id].x$, nếu không thì hàm $Fill$ sẽ điền thêm các ô trên hàng $br[id].x$ từ cột 2 tới cột $br[id].y$ (xem Hình 13-6). Chú ý rằng hàm $Fill$ sẽ được gọi mỗi khi thử đặt một tình trạng mìn cho một ô trong mảng $List$, vì vậy trước khi thử đặt một tình trạng khác hoặc lùi về thử những ô trước, ta phải phục hồi lại những ô bị hàm $Fill$ điền trở thành trạng thái "chưa biết", điều này được thực hiện bằng thủ tục $UndoFill$ với cách làm tương tự như hàm $Fill$.



Hình 13-6. Hai trường hợp của phép suy diễn: hàm *Fill* chỉ điền vào những ô chưa được điền ở bước trước

Input

- Dòng 1 chứa hai số nguyên dương $m, n \leq 2000$
- m dòng tiếp theo, dòng thứ i ghi n số trên hàng i của bản đồ mật độ theo đúng thứ tự từ trái qua phải.

Output

Cho biết tổng số lượng mìn trong bãi và bản đồ đánh dấu của bãi mìn

Sample Input	Sample Output
10 12 1 3 3 4 3 3 2 2 4 4 3 2 3 6 5 7 4 4 4 4 4 5 5 4 1 5 4 6 4 3 3 3 5 6 6 4 2 6 6 6 4 4 5 5 5 3 4 3 2 4 6 6 5 4 5 5 3 4 4 3 2 5 5 5 5 6 5 5 3 3 2 2 1 4 3 6 5 6 4 3 1 1 4 2 2 4 3 4 5 6 4 4 4 3 5 2 3 2 4 5 6 6 4 2 2 1 4 2 2 1 3 2 3 2 2 2 3 2 3 1	Number of mines: 65 + + + + + + + + - + + - - - + - + - - - + + + - + - + + - - - + - - + + + - + + + - + + - + + + - + + + - + + + - - - - - + + - - + - - - + - + - + - + + + - - + - + + - - + + + + - - - - + - + - - + - - + + + - + - + - + + + - - - - - +

Dò mìn

```
{$MODE OBJFPC}
program MineSweeper;
const
  max = 2000;
  dx: array[0..7] of Integer = (-1, -1, -1, 0, 0, 1, 1, 1);
  dy: array[0..7] of Integer = (-1, 0, 1, -1, 1, -1, 0, 1);
  csSafe = 0;
  csMine = 1;
  csUnknown = 2;
type
  TCellCount = array[0..2] of Integer;
  TPoint = record //Một ô gồm 2 toạ độ: hàng x cột y
    x, y: Integer;
  end;
var
  a: array[0..max + 1, 0..max + 1] of Integer; //Bản đồ đánh dấu
```

```
  b: array[1..max, 1..max] of Integer; //Bản đồ mật độ
  list, br: array[1..2 * max - 1] of TPoint;
  m, n: Integer;
  Found: Boolean;

procedure Enter; //Nhập dữ liệu
var
  i, j: Integer;
begin
  ReadLn(m, n);
  for i := 1 to m do
    begin
      for j := 1 to n do Read(b[i, j]);
      ReadLn;
    end;
end;

function Point(x, y: Integer): TPoint; inline;
begin
  Result.x := x;
  Result.y := y;
end;

function IsValid(const p: TPoint): Boolean; inline; //p có nằm trong bảng không?
begin
  with p do
    Result := (1 <= x) and (x <= m) and (1 <= y) and (y <= n);
end;

procedure Init; //Khởi tạo
var
  i, j, k, t, p: Integer;
  maxX, maxY: Integer;
begin
  //Bản đồ đánh dấu toàn giá trị csUnknown = 2, ngoài ra có thêm một đường viền =  csSafe = 0
  for i := 0 to m + 1 do
    for j := 0 to n + 1 do
      if IsValid(Point(i, j)) then a[i, j] := csUnknown
      else a[i, j] := csSafe;
  //Lập danh sách các ô để duyệt
  if m > n then t := m else t := n;
  k := 1;
  list[1] := Point(1, 1); //ô (1, 1) đầu tiên
  br[1] := Point(1, 1); // góc phải dưới hình chữ nhật tương ứng cũng là (1, 1)
  maxX := 1; maxY := 1;
  for p := 2 to t do
    begin
      if p <= m then //Xét ô (p, 1)
        begin
          Inc(k);
          list[k] := Point(p, 1); //đưa vào danh sách
          maxX := p;
          br[k] := Point(maxX, maxY); //tính góc phải dưới của hình chữ nhật tương ứng
        end;
      if p <= n then //Xét ô (1, p)
        begin
          Inc(k);
```

```
          list[k] := Point(1, p); //đưa vào danh sách
          maxY := p;
          br[k] := Point(maxX, maxY); //tính góc phải dưới của hình chữ nhật tương ứng
        end;
    end;
  Found := False;
end;

//Hàm CountNeighbor nhận vào ô p và đếm Count[s] là số ô mang trạng thái s quanh ô p
procedure CountNeighbor(const p: TPoint; var Count: TCellCount);
var
  d: Integer;
begin
  FillChar(Count, SizeOf(Count), 0);
  for d := 0 to 7 do //Xét 8 ô lân cận theo 8 hướng
    Inc(Count[a[p.x + dx[d], p.y + dy[d]]]);
end;

//Hàm CanBe kiểm tra xem ô p có thể nhận trạng thái s được không bằng phép kiểm tra lỗi tức thời
function CanBe(const p: TPoint; s: Integer): Boolean;
var
  q: TPoint;
  Save: Integer;
  Count: TCellCount;
  k: Integer;
  d: Integer;
begin
  if not IsValid(p) then //p nằm ngoài bảng thì p chỉ có thể nhận trạng thái csSafe = 0
    Exit(s = csSafe);
  Save := a[p.x, p.y]; //Lưu trữ trạng thái hiện tại
  a[p.x, p.y] := s; //Thử đặt trạng thái s
  try
    for d := 0 to 7 do //Xét 8 ô lân cận
      begin
        q.x := p.x + dx[d]; q.y := p.y + dy[d];
        if IsValid(q) then //không quan tâm đến các ô nằm trên viền giả
          begin
            CountNeighbor(q, Count); //Đếm số ô mang trạng thái 0, 1, 2 quanh ô q
            //nếu số mìn vượt quá hoặc chắc chắn không đủ thì không thể chấp nhận, thoát ngay
            if (Count[csMine] > b[q.x, q.y]) or
               (Count[csMine] + Count[csUnknown] < b[q.x, q.y]) then
              begin
                Result := False;
                Exit;
              end;
          end;
      end;
    Result := True; //Không có xung đột tức thời
  finally
    a[p.x, p.y] := Save; //Trước khi thoát thủ tục, phục hồi lại trạng thái cũ của ô p
  end;
end;

//Điền a[x,y]  dựa vào b[x-1, y-1]
function FillCell(x, y: Integer): Boolean;
var
  Count: TCellCount;
```

```
begin
  a[x, y] := csUnknown;
  CountNeighbor(Point(x - 1, y - 1), Count);
  if Count[csMine] = b[x - 1, y - 1] then //số mìn quanh ô (x - 1, y - 1) đã đủ
    a[x, y] := csSafe //thì ô (x, y) không thể có mìn nữa
  else
    if Count[csMine] + 1 = b[x - 1, y - 1] then //số mìn còn thiếu 1
      a[x, y] := csMine; //thì ô (x, y) chắc chắn có mìn
  Result := a[x, y] <> csUnknown;
end;

//Sau khi thử đến ô id trong list, hàm Fill sẽ được gọi để nới rộng hình chữ nhật nội suy
function Fill(id: Integer): Boolean;
var
  i, j: Integer;
begin
  if list[id].x = 1 then //ô id nằm ở hàng 1 thì nới rộng thêm 1 cột
    begin
      for i := 2 to br[id].x do
        if not FillCell(i, br[id].y) then //Điền thất bại
          Exit(False);
    end
  else //ô id nằm ở cột 1, nới rộng thêm 1 hàng
    for j := 2 to br[id].y do
      if not FillCell(br[id].x, j) then //điền thất bại
        Exit(False);
  Result := True; //điền thành công
end;

//Sau mỗi phép thử, hàm UndoFill được gọi để phục hồi co nhỏ hình chữ nhật nội suy
procedure UndoFill(id: Integer);
var
  i, j: Integer;
begin
  if list[id].x = 1 then //ô id nằm ở hàng 1, co lại 1 cột
    for i := 2 to br[id].x do
      a[i, br[id].y] := csUnknown
  else //ô id nằm ở cột 1, co lại 1 hàng
    for j := 2 to br[id].y do
      a[br[id].x, j] := csUnknown;
end;

procedure Attempt(k: Integer); //Thuật toán quay lui, thử điền trạng thái cho ô list[k]
begin
  if k = m + n then //có m + n – 1 ô trong list, thử đến ô m + n tức là tìm ra kết quả
    begin
      Found := True;
      Exit;
    end;
  if CanBe(list[k], csSafe) then //Nếu có thể đặt trạng thái csSafe = 0 cho ô list[k]
    begin
      a[list[k].x, list[k].y] := csSafe; //Thử đặt
      if Fill(k) then //Dùng hàm Fill kiểm tra lỗi suy diễn và nới rộng hình chữ nhật nội suy
        begin
          Attempt(k + 1); //Không có lỗi suy diễn thì thử tiếp ô list[k+1]
          if Found then Exit; //Tìm ra nghiệm là thoát ngay
        end;
```

```
      UndoFill(k); //Co hình chữ nhật nội suy lại
    end;
  if CanBe(list[k], csMine) then //Nếu có thể đặt trạng thái csMine = 1 cho ô list[k]
    begin
      a[list[k].x, list[k].y] := csMine; //Thử đặt
      if Fill(k) then //Dùng hàm Fill kiểm tra lỗi suy diễn và nới rộng hình chữ nhật nội suy
        begin
          Attempt(k + 1); //Không có lỗi suy diễn thì thử tiếp ô list[k+1]
          if Found then Exit; //Tìm ra nghiệm là thoát ngay
        end;
      UndoFill(k); //Co hình chữ nhật nội suy lại
    end;
  //Trước khi quay lui để thử ô list[k-1], đặt trạng thái ô list[k] trở về csUnknown = 2
  a[list[k].x, list[k].y] := csUnknown;
end;

procedure PrintResult; //In kết quả
var
  i, j: Integer;
  MineCount: Integer;
begin
  MineCount := 0;
  for i := 1 to m do
    for j := 1 to n do
      if a[i, j] = csMine then Inc(MineCount);
  WriteLn('Number of mines: ', MineCount);
  for i := 1 to m do
    begin
      for j := 1 to n do
        if a[i, j] = csMine then Write('+ ')
        else Write('- ');
      WriteLn;
    end;
end;

begin
  Enter; //Nhập liệu
  Init; //Khởi tạo
  Attempt(1); //Khởi động thuật toán quay lui
  PrintResult; //In kết quả
end.
```

13.4.4. Dãy ABC

Cho trước một số nguyên dương $n \leq 1000$, hãy tìm một xâu chỉ gồm các ký tự 'A', 'B', 'C' thoả mãn 3 điều kiện:

- Có độ dài n.
- Hai đoạn con bất kỳ liền nhau đều khác nhau (đoạn con là một dãy ký tự liên tiếp của xâu).
- Có ít ký tự 'C' nhất.

❑ Thuật toán 1: Ước lượng hàm cận

Ta sẽ dùng thuật toán quay lui để liệt kê các dãy n ký tự mà mỗi phần tử của dãy được chọn trong tập $\{A, B, C\}$. Giả sử cấu hình cần liệt kê có dạng $x_{1\dots n}$ thì:

Nếu dãy $x_{1\dots n}$ thoả mãn 2 đoạn con bất kỳ liền nhau đều khác nhau, thì trong 4 ký tự liên tiếp bất kỳ bao giờ cũng phải có ít nhất một ký tự 'C'. Như vậy với một đoạn gồm k ký tự liên tiếp của dãy $x_{1\dots n}$ thì số ký tự 'C' trong đoạn đó luôn $\geq \lfloor k/4 \rfloor$

Sau khi thử chọn $x_i \in \{A, B, C\}$, nếu ta đã có t_i ký tự 'C' trong đoạn $x_{1\dots i}$, thì cho dù các bước chọn tiếp sau làm tốt như thế nào chăng nữa, số ký tự 'C' phải chọn thêm không bao giờ ít hơn $\left\lfloor \frac{n-i}{4} \right\rfloor$. Tức là nếu theo phương án chọn x_i như thế này thì số ký tự 'C' trong dãy kết quả (khi chọn đến x_n) không thể ít hơn $t_i + \left\lfloor \frac{n-i}{4} \right\rfloor$. Ta dùng con số này để đánh giá nhánh cận, nếu nó nhiều hơn số ký tự 'C' trong cấu hình tốt nhất đang cho tới thời điểm hiện tại thì chắc chắn có làm tiếp cũng chỉ được một cấu hình tồi tệ hơn, ta bỏ qua ngay cách chọn này và thử phương án khác.

Tôi đã thử và thấy thuật toán này hoạt động khá nhanh với $n \leq 100$, tuy nhiên với những giá trị $n \geq 200$ thì vẫn không đủ kiên nhẫn để đợi ra kết quả. Dưới đây là một thuật toán khác tốt hơn, đi đôi với nó là một chiến lược chọn hàm cận khá hiệu quả, khi mà ta khó xác định hàm cận thật chặt bằng công thức tường minh.

❑ Thuật toán 2: Lấy ngắn nuôi dài

Với mỗi độ dài m, ta gọi $f[m]$ là số ký tự 'C' trong xâu có độ dài m thoả mãn hai đoạn con bất kỳ liền nhau phải khác nhau và có ít ký tự 'C' nhất. Rõ ràng ta có $f[0] = 0$, ta sẽ lập trình tính các $f[m]$ trong điều kiện các $f[0 \dots m-1]$ đã biết.

Tương tự như thuật toán 1, giả sử cấu hình cần tìm có dạng $x_{1\dots m}$ thì sau khi thử chọn x_i, nếu ta đã có t_i ký tự 'C' trong đoạn $x_{1\dots i}$, thì cho dù các bước chọn tiếp sau làm tốt như thế nào chăng nữa, số ký tự 'C' phải chọn thêm không bao giờ ít hơn $f[m-i]$, tức là nếu chọn tiếp thì số ký tự 'C' không thể ít hơn $t_i + f[m-i]$. Ta dùng cận này kết hợp với thuật toán quay lui để tìm xâu tối ưu độ dài m cũng như để tính giá trị $f[m]$

Như vậy ta phải thực hiện thuật toán n lần với các độ dài xâu $m \in \{1, 2, \dots, n\}$, tuy nhiên lần thực hiện sau sẽ sử dụng những thông tin đã có của lần thực hiện trước để làm một hàm cận chặt hơn và thực hiện trong thời gian chấp nhận được.

❑ Cài đặt

Input

Số nguyên dương n

Output

Xâu ABC cần tìm

Sample Input	Sample Output
10	Analyzing... f[1] = 0 f[2] = 0 f[3] = 0 f[4] = 1 f[5] = 1 f[6] = 1 f[7] = 1 f[8] = 2 f[9] = 2 f[10] = 2 The best string of 10 letters is: ABACABCBAB Number of 'C' letters: 2

Dãy ABC

```
{$MODE OBJFPC}
program ABC_STRING;
const
  max = 1000;
  modulus = 12345;
var
  n, m, MinC, CountC: Integer;
  Powers: array[0..max] of Integer;
  f: array[0..max] of Integer;
  x, Best: AnsiString;

procedure Init; //Với một độ dài m <= n, khởi tạo thuật toán quay lui
var
  i: Integer;
begin
  SetLength(x, m);
  MinC := m;
  CountC := 0;
  Powers[0] := 1;
  for i := 1 to m do
    Powers[i] := Powers[i - 1] * 3 mod modulus;
  f[0] := 0;
end;

//Đổi ký tự c ra một chữ số trong hệ cơ số 3
function Code(c: Char): Integer; inline;
begin
  Result := Ord(c) - Ord('A');
end;

//Hàm Same(i, l) cho biết xâu gồm l ký tự kết thúc tại x[i] có trùng với xâu l ký tự liền trước nó không ?
function Same(i, j, k: Integer): Boolean; inline;
begin
```

```
  while k <= i do
    begin
      if x[k] <> x[j] then
        Exit(False);
      Inc(k); Inc(j);
    end;
  Result := True;
end;

//Hàm Check(i) cho biết x[i] có làm hỏng tính không lặp của dãy x[1..i] hay không. Thuật toán Rabin-Karp
function Check(i: Integer): Boolean;
var
  j, k: Integer;
  h: array[1..max] of Integer;
begin
  h[i] := Code(x[i]);
  for j := i - 1 downto 1 do
    begin
      h[j] := (Powers[i - j] * Code(x[j]) + h[j + 1]) mod modulus;
      if odd(i - j) then
        begin
          k := i - (i - j) div 2;
          if (h[k] * (Powers[k - j] + 1) mod modulus = h[j])
             and Same(i, j, k) then
            Exit(False);
        end;
    end;
  Result := True;
end;

//Giữ lại kết quả tốt hơn vừa tìm được
procedure UpdateSolution;
begin
  MinC := CountC;
  if m = n then
    Best := x;
end;

//Thuật toán quay lui
procedure Attempt(i: Integer); //Thử các giá trị có thể nhận của X[i]
var
  j: AnsiChar;
begin
  for j := 'A' to 'C' do //Xét tất cả các khả năng
    begin
      x[i] := j; //Thử đặt x[i]
      if Check(i) then //nếu giá trị đó vào không làm hỏng tính không lặp
        begin
          if j = 'C' then Inc(CountC); //Cập nhật số ký tự C cho tới bước này
          if CountC + f[m - i] < MinC then //Đánh giá nhánh cận
            if i = m then UpdateSolution //Cập nhật kết quả nếu đã đến lượt thử cuối
            else Attempt(i + 1); //Chưa đến lượt thử cuối thì thử tiếp
          if j = 'C' then Dec(CountC); //Phục hồi số ký tự C như cũ
        end;
    end;
end;
```

```
procedure PrintResult;
begin
  WriteLn('The best string of ', n, ' letters is:');
  WriteLn(Best);
  WriteLn('Number of ''C'' letters: ', MinC);
end;

begin
  ReadLn(n);
  WriteLn('Analyzing...');
  for m := 1 to n do
    begin
      Init;
      Attempt(1);
      f[m] := MinC;
      WriteLn('f[', m, '] = ', f[m]);
    end;
  WriteLn;
  PrintResult;
end.
```

13.4.5. Tóm lược

Chúng ta đã khảo sát kỹ thuật nhánh cận áp dụng trong thuật toán quay lui để giải quyết một số bài toán tối ưu. Kỹ thuật này còn có thể áp dụng cho lớp các bài toán duyệt nói chung để hạn chế bớt không gian tìm kiếm.

Khi cài đặt thuật toán quay lui có đánh giá nhánh cận, cần có:

- Một hàm cận tốt để loại bỏ sớm những phương án chắc chắn không phải nghiệm
- Một thứ tự duyệt tốt để nhanh chóng đi tới nghiệm tối ưu

Có một số trường hợp mà khó có thể tìm ra một thứ tự duyệt tốt thì ta có thể áp dụng một thứ tự ngẫu nhiên của các giá trị cho mỗi bước thử và dùng một hàm chặn thời gian để chấp nhận ngay phương án tốt nhất đang có sau một khoảng thời gian nhất định và ngưng quá trình thử (ví dụ 1 giây). Một cách làm khác là ta sẽ chỉ duyệt tới một độ sâu nhất định, sau đó một thuật toán tham lam sẽ được áp dụng để tìm ra một nghiệm có thể không phải tối ưu nhưng tốt ở mức chấp nhận được. Chiến lược này có tên gọi "bé→duyệt, to→tham".

Bài tập 13-1.

Hãy lập chương trình nhập vào hai số n và k, liệt kê các chỉnh hợp lặp chập k của tập $\{1,2,\ldots,n\}$.

Bài tập 13-2.

Hãy liệt kê các dãy nhị phân độ dài n mà trong đó cụm chữ số "01" xuất hiện đúng 2 lần.

Bài tập 13-3.

Nhập vào một danh sách n tên người. Liệt kê tất cả các cách chọn ra đúng k người trong số n người đó.

Bài tập 13-4.

Để liệt kê tất cả các tập con của tập $\{1,2,\dots,n\}$ ta có thể dùng phương pháp liệt kê tập con như trên hoặc dùng phương pháp liệt kê tất cả các dãy nhị phân. Mỗi số 1 trong dãy nhị phân tương ứng với một phần tử được chọn trong tập. Ví dụ với tập $\{1,2,3,4\}$ thì dãy nhị phân 1010 sẽ tương ứng với tập con $\{1,3\}$. Hãy lập chương trình in ra tất cả các tập con của tập $\{1,2,\dots,n\}$ theo hai phương pháp.

Bài tập 13-5.

Cần xếp n người một bàn tròn, hai cách xếp được gọi là khác nhau nếu tồn tại hai người ngồi cạnh nhau ở cách xếp này mà không ngồi cạnh nhau trong cách xếp kia. Hãy đếm và liệt kê tất cả các cách xếp.

Bài tập 13-6.

Người ta có thể dùng phương pháp sinh để liệt kê các chỉnh hợp không lặp chập k. Tuy nhiên có một cách khác là liệt kê tất cả các tập con k phần tử của tập hợp, sau đó in ra đủ $k!$ hoán vị của các phần tử trong mỗi tập hợp. Hãy viết chương trình liệt kê các chỉnh hợp không lặp chập k của tập $\{1,2,\dots,n\}$ theo cả hai cách.

Bài tập 13-7.

Liệt kê tất cả các hoán vị chữ cái trong từ MISSISSIPPI theo thứ tự từ điển.

Bài tập 13-8.

Cho hai số nguyên dương l,n. Hãy liệt kê các xâu nhị phân độ dài n có tính chất, bất kỳ hai xâu con nào độ dài l liền nhau đều khác nhau.

Bài tập 13-9.

Với $n=5,k=3$ vẽ cây tìm kiếm quay lui của chương trình liệt kê tổ hợp chập k của tập $\{1,2,\dots,n\}$

Bài tập 13-10.

Cho tập S gồm n số nguyên, hãy liệt kê tất cả các tập con k phần tử của tập S thỏa mãn: độ chênh lệch về giá trị giữa hai phần tử bất kỳ trong tập con đó không vượt quá t (t cho trước)

Bài tập 13-11.

Một dãy $x_{1\ldots n}$ gọi là một hoán vị hoàn toàn của tập $\{1,2,\ldots,n\}$ nếu nó là một hoán vị thoả mãn: $x_i \neq i, \forall i: 1 \leq i \leq n$. Hãy viết chương trình liệt kê tất cả các hoán vị hoàn toàn của tập $\{1,2,\ldots,n\}$

Bài tập 13-12.

Lập trình đếm số cách xếp n quân hậu lên bàn cờ $n \times n$ sao cho không quân nào ăn quân nào.

Bài tập 13-13.

Mã đi tuần: Cho bàn cờ tổng quát kích thước $n \times n$ và một quân Mã, hãy chỉ ra một hành trình của quân Mã xuất phát từ ô đang đứng đi qua tất cả các ô còn lại của bàn cờ, mỗi ô đúng 1 lần.

Bài tập 13-14.

Xét sơ đồ giao thông gồm n nút giao thông đánh số từ 1 tới n và m đoạn đường nối chúng, mỗi đoạn đường nối 2 nút giao thông. Hãy nhập dữ liệu về mạng lưới giao thông đó, nhập số hiệu hai nút giao thông s và t. Hãy in ra tất cả các cách đi từ s tới t mà mỗi cách đi không được qua nút giao thông nào quá một lần.

Bài tập 13-15.

Cho một số nguyên dương $n \leq 10000$, hãy tìm một hoán vị của dãy $(1,2,\ldots,2n)$ sao cho tổng hai phần tử liên tiếp của dãy là số nguyên tố. Ví dụ với $n = 5$, ta có dãy $(1,6,5,2,3,8,9,4,7,10)$

Bài tập 13-16.

Một dãy dấu ngoặc hợp lệ là một dãy các ký tự "(" và ")" được định nghĩa như sau:

- Dãy rỗng là một dãy dấu ngoặc hợp lệ độ sâu 0
- Nếu A là dãy dấu ngoặc hợp lệ độ sâu k thì (A) là dãy dấu ngoặc hợp lệ độ sâu $k+1$
- Nếu A và B là hai dãy dấu ngoặc hợp lệ với độ sâu lần lượt là p và q thì AB là dãy dấu ngoặc hợp lệ độ sâu là $\max(p,q)$

Độ dài của một dãy ngoặc là tổng số ký tự "(" và ")"

Ví dụ: Có 5 dãy dấu ngoặc hợp lệ độ dài 8 và độ sâu 3:

(((()())

((()()())

((()))()

(()(()))

()((()))

Bài toán đặt ra là khi cho biết trước hai số nguyên dương n, k và $1 \leq k \leq n \leq 10000$. Hãy liệt kê các dãy ngoặc hợp lệ có độ dài là $2n$ và độ sâu là k. Trong trường hợp có nhiều hơn 100 dãy thì chỉ cần đưa ra 100 dãy nhỏ nhất theo thứ tự từ điển

Bài tập 13-17.

Có m người thợ và n công việc $(1 \leq m, n \leq 100)$, mỗi thợ có khả năng làm một số công việc nào đó. Hãy chọn ra một tập ít nhất những người thợ sao cho bất kỳ công việc nào trong số công việc đã cho đều có người làm được trong số những người đã chọn.

Bài 14. Chia để trị và giải thuật đệ quy

14.1. Chia để trị

Ta nói một đối tượng là đệ quy nếu nó được định nghĩa qua chính nó hoặc một đối tượng khác cùng dạng với chính nó bằng quy nạp.

> Ví dụ: Đặt hai chiếc gương cầu đối diện nhau. Trong chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ nhất… Ở một góc nhìn hợp lý, ta có thể thấy một dãy ảnh vô hạn của cả hai chiếc gương.
> Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngồi bên máy vô tuyến truyền hình và cứ như thế…
> Trong toán học, ta cũng hay gặp các định nghĩa đệ quy:
>
> - Giai thừa của n ($n!$): Nếu $n = 0$ thì $n! = 1$; nếu $n > 0$ thì $n! = n(n-1)!$
> - Ký hiệu số phần tử của một tập hợp hữu hạn S là $|S|$: Nếu $S = \emptyset$ thì $|S| = 0$; Nếu $S \neq \emptyset$ thì tất có một phần tử $x \in S$, khi đó $|S| = |S - \{x\}| + 1$. Đây là phương pháp định nghĩa tập các số tự nhiên.

Ta nói một bài toán P mang bản chất đệ quy nếu lời giải của một bài toán P có thể được thực hiện bằng lời giải của các bài toán $P_1, P_2, \dots, P_n$ có dạng giống như P. Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: $P_1, P_2, \dots, P_n$ tuy có dạng giống như P, nhưng theo một nghĩa nào đó, chúng phải "nhỏ" hơn P, dễ giải hơn P và việc giải chúng không cần dùng đến P.

Chia để trị (*divide and conquer*) là một phương pháp thiết kế giải thuật cho các bài toán mang bản chất đệ quy: Để giải một bài toán lớn, ta phân rã nó thành những bài toán con đồng dạng, và cứ tiến hành phân rã cho tới khi những bài toán con đủ nhỏ để có thể giải trực tiếp. Sau đó những nghiệm của các bài toán con này sẽ được phối hợp lại để được nghiệm của bài toán lớn hơn cho tới khi có được nghiệm bài toán ban đầu.

Khi nào một bài toán có thể tìm được thuật giải bằng phương pháp chia để trị?. Có thể tìm thấy câu trả lời qua việc giải đáp các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng phối hợp của những bài toán cùng loại nhưng nhỏ hơn hay không ? Khái niệm "nhỏ hơn" là thế nào ? (Xác định quy tắc phân rã bài toán)
- Trường hợp đặc biệt nào của bài toán có thể coi là đủ nhỏ để có thể giải trực tiếp được? (Xác định các bài toán cơ sở)

14.2. Giải thuật đệ quy

Các giải thuật đệ quy là hình ảnh trực quan nhất của phương pháp chia để trị. Trong các ngôn ngữ lập trình cấu trúc, các giải thuật đệ quy thường được cài đặt bằng các chương trình con đệ quy. Một chương trình con đệ quy gồm hai phần:

- Phần *neo* (*anchor*): Phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.
- Phần *đệ quy* (*recursion*): Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ quy giải những bài toán con đó. Khi đã có lời giải (đáp số) của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

Phần đệ quy mô phỏng quá trình phân rã bài toán theo nguyên lý chia để trị. Phần neo tương ứng với những bài toán con đủ nhỏ có thể giải trực tiếp được, nó quyết định tính hữu hạn dừng của lời giải.

Sau đây là một vài ví dụ về giải thuật đệ quy.

14.2.1. Tính giai thừa

Định nghĩa giai thừa của một số tự nhiên n, ký hiệu $n!$, là tích của các số nguyên dương từ 1 tới n:

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n$$

Hoàn toàn có thể sử dụng một thuật toán lặp để tính $n!$, tuy nhiên nếu chúng ta thử nghĩ một theo một cách khác: Vì $n! = n \times (n-1)!$ nên để tính $n!$ (bài toán lớn) ta đi tính $(n-1)!$ (bài toán con) rồi lấy kết quả nhân với n.

Cách nghĩ này cho ta một định nghĩa quy nạp của hàm giai thừa, bài toán tính giai thừa của một số được đưa về bài toán tính giai thừa của một số khác nhỏ hơn.

```
function Factorial(n: Integer): Integer; //Nhận vào số tự nhiên n và trả về n!
begin
  if n = 0 then Result := 1 //Phần neo
  else Result := n * Factorial(n - 1); //Phần đệ quy
end;
```

Ở đây, phần neo định nghĩa kết quả hàm tại $n = 0$, còn phần đệ quy (ứng với $n > 0$) sẽ định nghĩa kết quả hàm qua giá trị của n và giai thừa của $n - 1$.

Ví dụ: Dùng hàm này để tính 3!

$$3! = 3 \times 2!$$
$$\downarrow$$
$$2! = 2 \times 1!$$
$$\downarrow$$
$$1! = 1 \times 0!$$
$$\downarrow$$
$$0! = 1$$

14.2.2. Đổi cơ số

Để biểu diễn một giá trị số $x \in \mathbb{N}$ trong hệ nhị phân: $x = \overline{x_d x_{d-1} \dots x_1 x_0}_{(2)}$, ta cần tìm dãy các chữ số nhị phân $x_0, x_1, \dots, x_d \in \{0,1\}$ để:

$$x = \sum_{i=0}^{d} x_i 2^i = x_d \times 2^d + x_{d-1} \times 2^{d-1} + \dots + x_1 \times 2 + x_0$$

Có nhiều thuật toán lặp để tìm biểu diễn nhị phân của x, tuy nhiên chúng ta có thể sử dụng phương pháp chia để trị để thiết kế một giải thuật đệ quy khá ngắn gọn.

Ta có nhận xét là x_0 (chữ số hàng đơn vị) chính bằng số dư của phép chia x cho 2 ($x \bmod 2$) và nếu loại bỏ x_0 khỏi biểu diễn nhị phân của x ta sẽ được số:

$$\overline{x_d x_{d-1} \dots x_1}_{(2)} = x \operatorname{div} 2$$

Vậy để tìm biểu diễn nhị phân của x (bài toán lớn), ta có thể tìm biểu diễn nhị phân của số $x \operatorname{div} 2$ (bài toán nhỏ) rồi viết thêm giá trị $x \bmod 2$ vào sau biểu diễn đó. Ngoài ra khi $x = 0$ hay $x = 1$, biểu diễn nhị phân của x chính là x nên có thể coi đây là những bài toán đủ nhỏ có thể giải trực tiếp được. Toàn bộ thuật toán có thể viết bằng một thủ tục đệ quy $Convert(x)$ như sau:

```
procedure Convert(x: Integer);
begin
  if x ≥ 2 then Convert(x div 2);
  Output ← x mod 2;
end;
```

14.2.3. Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán đặt ra như sau:

- Các con thỏ không bao giờ chết
- Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái)
- Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới

Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, $n = 5$, ta thấy:
Giữa tháng thứ 1: 1 cặp (ab) (cặp ban đầu)
Giữa tháng thứ 2: 1 cặp (ab) (cặp ban đầu vẫn chưa đẻ)
Giữa tháng thứ 3: 2 cặp (AB)(cd) (cặp ban đầu đẻ ra thêm 1 cặp con)
Giữa tháng thứ 4: 3 cặp (AB)(cd)(ef) (cặp ban đầu tiếp tục đẻ)
Giữa tháng thứ 5: 5 cặp (AB)(CD)(ef)(gh)(ik) (cả cặp (AB) và (CD) cùng đẻ)

Bây giờ, ta xét tới việc tính số cặp thỏ ở tháng thứ n: $f(n)$. Nếu ta đang ở tháng $n-1$ và tính số thỏ ở tháng n thì:

Số tháng tới = Số hiện có + Số được sinh ra trong tháng tới

Mặt khác, với tất cả các cặp thỏ ≥ 1 tháng tuổi thì sang tháng sau, chúngđ ều ≥ 2 tháng tuổi và đều sẽ sinh. Tức là:

Số được sinh ra trong tháng tới = Số tháng trước

Vậy:

Số tháng tới = Số hiện có + Số tháng trước

$$f(n) = f(n-1) + f(n-2)$$

Vậy có thể tính được $f(n)$ theo công thức sau:

$$f(n) = \begin{cases} 1, \text{ nếu } n \leq 2 \\ f(n-1) + f(n-2), \text{ nếu } n > 2 \end{cases}$$

```
function f(n: Integer): Integer; //Tính số cặp thỏ ở tháng thứ n
begin
  if n ≤ 2 then Result := 1 //Phần neo
  else Result := f(n - 1) + f(n - 2); //Phần đệ quy
end;
```

14.3. Hiệu lực của chia để trị và đệ quy

Chia để trị là một cách thức tiếp cận bài toán. Có rất nhiều bài toán quen thuộc có thể giải bằng đệ quy thay vì lời giải lặp nhờ cách tiếp cận này. Ngoài những bài toán kể trên, có thể đưa ra một số ví dụ khác:

- Bài toán tìm giá trị lớn nhất trong một dãy số: Để tìm giá trị lớn nhất của dãy $(a_1, a_2, \ldots, a_n)$, nếu dãy chỉ có một phần tử thì đó chính là giá trị lớn nhất, nếu không ta tìm giá trị lớn nhất của dãy gồm $\lfloor n/2 \rfloor$ phần tử đầu và giá trị lớn nhất của dãy gồm $\lceil n/2 \rceil$ phần tử cuối, sau đó chọn ra giá trị lớn nhất trong hai giá trị này. Phương pháp này tỏ ra thích hợp khi cài đặt thuật toán trên các máy song song: Việc tìm giá trị lớn nhất trong hai nửa dãy sẽ được thực hiện độc lập và đồng thời trên hai bộ xử lý khác

nhau, làm giảm thời gian thực hiện trên máy. Tương tự như vậy, thuật toán tìm kiếm tuần tự cũng có thể viết bằng đệ quy.

- Thuật toán tìm kiếm nhị phân cũng có thể viết bằng đệ quy: Đưa việc tìm kiếm một giá trị trên dãy đã sắp xếp về việc tìm kiếm giá trị đó trên một dãy con có độ dài bằng một nửa.
- Thuật toán QuickSort đã có mô hình chu ẩn viết bằng đệ quy, nhưng không chỉ có QuickSort, một loạt các thuật toán sắp xếp khác cũng có thể cài đặt bằng đệ quy, ví dụ thuật toán Merge Sort: Để sắp xếp một dãy khóa, ta sẽ sắp xếp riêng dãy các phần tử mang chỉ số lẻ và dãy các phần tử mang chỉ số chẵn rồi trộn chúng lại. Hay thuật toán Insertion Sort: Để sắp xếp một dãy khóa, ta lấy ra một phần tử bất kỳ, sắp xếp các phần tử còn lại và chèn phần tử đã lấy ra vào vị trí đúng của nó…
- Để liệt kê các hoán vị của dãy số $(1,2,\dots,n)$ ta lấy lần lượt từng phần tử trong dãy ra ghép với $(n-1)!$ hoán vị của các phần tử còn lại.

Cần phải nhấn mạnh rằng phương pháp chia để trị là một cách tiếp cận bài toán và tìm lời giải đệ quy, nhưng nguyên lý này còn đóng vai trò chủ đạo trong việc thiết kế nhiều thuật toán khác nữa.

Ngoài ra, không được lạm dụng chia để trị và đệ quy. Có những trường hợp lời giải đệ quy tỏ ra ngắn gọn và hiệu quả (ví dụ như mô hình cài đặt thuật toán QuickSort) nhưng cũng có những trường hợp giải thuật đệ quy không nhanh hơn hay đơn giản hơn giải thuật lặp (tính giai thừa); thậm chí còn có những trường hợp mà lạm dụng đệ quy sẽ làm phức tạp hóa vấn đề, kéo theo một quá trình tính toán cồng kềnh như giải thuật tính số Fibonacci ở trên (có thể chứng minh bằng quy nạp là số phép cộng trong giải thuật đệ quy để tính số Fibonacci thứ n đúng bằng $f(n)-1$ trong khi giải thuật lặp chỉ cần không quá n phép cộng).

Vì bài toán giải bằng phương pháp chia để trị mang bản chất đệ quy nên để chứng minh tính đúng đắn và phân tích thời gian thực hiện giải thuật, người ta thường sử dụng các công cụ quy nạp toán học. Nói riêng về việc phân tích thời gian thực hiện giải thuật, còn có một công cụ rất mạnh là Định lý 3-7 (Định lý Master).

Giả sử rằng ta có một bài toán kích thước n và một thuật toán chia để trị. Gọi $T(n)$ là thời gian thực hiện giải thuật đó. Nếu thuật toán phân rã bài toán lớn ra thành a bài toán con, mỗi bài toán con có kích thước n/b, sau đó giải độc lập a bài toán con và phối hợp nghiệm lại thì thời gian thực hiện giải thuật sẽ là

$$T(n) = aT(n/b) + f(n)$$

Ở đây ta ký hiệu $f(n)$ là thời gian phân rã bài toán lớn, phối hợp nghiệm của các bài toán con…nói chung là các chi phí thời gian khác ngoài việc giải các bài toán con.

Định lý Master nói rằng nếu $a \geq 1$ và $b > 1$, (n/b có thể là $\lfloor n/b \rfloor$ hay $\lceil n/b \rceil$ không quan trọng), khi đó:

- Nếu $f(n) = \mathrm{O}\left(n^{\log_b a - \epsilon}\right)$ với hằng số $\epsilon > 0$, thì $T(n) = \Theta\left(n^{\log_b a}\right)$
- Nếu $f(n) = \Theta\left(n^{\log_b a}\right)$ thì $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
- Nếu $f(n) = \Omega(n^{\log_b a + \epsilon})$ với hằng số $\epsilon > 0$ và $af(n/b) \leq cf(n)$ với hằng số $c < 1$ và với mọi giá trị n đủ lớn thì $T(n) = \Theta\left(f(n)\right)$

Tuy việc chứng minh định lý này khá phức tạp nhưng chúng ta cần phải nhớ để áp dụng được nhanh. Ta sẽ xét tiếp một vài bài toán kinh điển áp dụng phương pháp chia để trị và đệ quy

14.3.1. Tính lũy thừa

Bài toán đặt ra là cho hai số nguyên dương x, n. Hãy tính giá trị x^n.

❑ Thuật toán 1

Ta có thể tính trực tiếp bằng thuật toán lặp[*]: Viết một hàm $Power(x, n)$ để tính x^n

```
function Power(x, n: Integer): Integer;
var
  i: Integer;
begin
  Result := 1;
  for i := 1 to n do Result := Resul * x;
end;
```

Nếu coi phép nhân (*) là phép toán tích cực thì có thể thấy rằng thuật toán tính trực tiếp này cần n phép nhân. Vậy thời gian thực hiện giải thuật là $\Theta(n)$.

❑ Thuật toán 2

Xét một thuật toán chia để trị dựa vào tính chất sau của x^n

$$x^n = \begin{cases} 1, \text{ nếu } n = 0 \\ x \times x^{n-1}, \text{ nếu } n > 0 \end{cases}$$

[*] Một số công cụ lập trình có cung cấp sẵn hàm mũ. Như trong Free Pascal, ta có thể tính $x^n = Exp(Ln(x) * n)$ hay sử dụng thư viện Math để dùng các hàm $Power(x, n)$ hoặc $IntPower(x, n)$. Tuy nhiên cách làm này có hạn chế là không thể tùy biến được nếu chúng ta thực hiện tính toán trên số lớn, khi mà kết quả x^n buộc phải biểu diễn bằng mảng hoặc xâu ký tự…

```
function Power(x, n: Integer): Integer;
begin
  if n = 0 then Result := 1
  else Result := x * Power(x, n - 1);
end;
```

Bài toán tính x^n được đưa về bài toán tính x^{n-1} nếu $n > 0$. Xét về thời gian thực hiện giải thuật, thuật toán cần thực hiện tất cả n phép nhân (phép toán tích cực) nên thời gian thực hiện cũng là $\Theta(n)$.

Thuật toán này nếu có tốt hơn thuật toán trước thì chỉ ở chỗ viết gọn hơn một chút, không có cải thiện nào về tốc độ, lại tốn bộ nhớ hơn (bộ nhớ chứa tham số truyền cho chương trình con khi gọi đệ quy).

❑ Thuật toán 3

Ta xét một thuật toán chia để trị khác:

$$x^n = \begin{cases} 1, \text{ nếu } n = 0 \\ \left(x^{n/2}\right)^2, \text{ nếu } n > 0 \text{ và } n \text{ chẵn} \\ \left(x^{\lfloor n/2 \rfloor}\right)^2 \times x, \text{ nếu } n > 0 \text{ và } n \text{ lẻ} \end{cases}$$

```
function Power(x, n: Integer): Integer;
begin
  if n = 0 then Result := 1
  else
     begin
       Result := Power(x, n div 2);
       Result := Result * Result;
       if n mod 2 = 1 then Result := Result * x;
     end;
end;
```

Việc tính x^n được quy về việc tính $x^{\lfloor n/2 \rfloor}$ rồi đem kết quả bình phương lên (thêm 1 phép nhân), sau đó giữ nguyên kết quả hoặc nhân thêm kết quả với x tùy theo n chẵn hay n lẻ.

Nếu gọi $T(n)$ là thời gian thực hiện hàm $Power(x, n)$ thì ngoài lệnh gọi đệ quy, các lệnh khác trong hàm $Power(x, n)$ tuy có tổng thời gian thực hiện không phải là hằng số nhưng bị chặn (trên và dưới) bởi hằng số. Vậy:

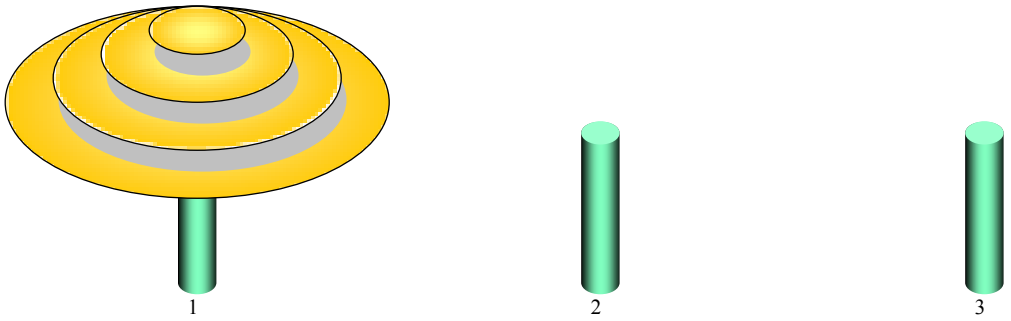
$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(1)$$

Áp dụng Định lý 3-7 (Định lý Master) (trường hợp 2), ta có $T(n) = \Theta(\lg n)$. Thuật toán thứ ba tỏ ra nhanh hơn hai thuật toán trước. Ví dụ cụ thể, để tính 2^{12}, chúng ta chỉ mất 6 phép nhân so với 12 phép nhân của hai cách tính trước.

14.3.2. Tháp Hà Nội

❑ Bài toán

Đây là một bài toán mang tính chất một trò chơi, tương truyền rằng tại ngôi đền Benares có ba cái cọc kim cương. Khi khai sinh ra thế giới, thượng đế đặt 64 cái đĩa bằng vàng chồng lên nhau theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt trên một chiếc cọc.



Hình 14-1. Tháp Hà Nội

Các nhà sư lần lượt chuyển các đĩa sang cọc khác theo luật:

- Khi di chuyển một đĩa, phải đặt nó vào vị trí ở một trong ba cọc đã cho
- Mỗi lần chỉ có thể chuyển một đĩa và phải là đĩa ở trên cùng của chồng đĩa
- Tại một vị trí, đĩa nào mới chuyển đến sẽ phải đặt lên trên cùng
- Đĩa lớn hơn không bao giờ được phép đặt lên trên đĩa nhỏ hơn (hay nói cách khác: một đĩa chỉ được đặt trên cọc hoặc đặt trên một đĩa lớn hơn)

Ngày tận thế sẽ đến khi toàn bộ chồng đĩa được chuyển sang một cọc khác.

Trong trường hợp có 2 đĩa, cách làm có thể mô tả như sau: Chuyển đĩa nhỏ sang cọc 3, đĩa lớn sang cọc 2 rồi chuyển đĩa nhỏ từ cọc 3 sang cọc 2.

Những người mới bắt đầu có thể giải quyết bài toán một cách dễ dàng khi số đĩa là ít, nhưng họ sẽ gặp rất nhiều khó khăn khi số các đĩa nhiều hơn. Tuy nhiên, với tư duy quy nạp toán học và một máy tính thì công việc trở nên khá dễ dàng:

❑ Thuật toán chia để trị

Giả sử chúng ta có n đĩa.

Nếu $n = 1$ thì ta chuyển đĩa duy nhất đó từ cọc 1 sang cọc 2 là xong.

Nếu ta có phương pháp chuyển được $n - 1$ đĩa từ cọc 1 sang cọc 2, thì tổng quát, cách chuyển $n - 1$ đĩa từ cọc x sang cọc y $(1 \leq x, y \leq 3)$ cũng tương tự.

Giả sử rằng ta có phương pháp chuyển được $n-1$ đĩa giữa hai cọc bất kỳ. Để chuyển n đĩa từ cọc x sang cọc y, ta gọi cọc còn lại là $z(=6-x-y)$. Coi đĩa to nhất là … cọc, chuyển $n-1$ đĩa còn lại từ cọc x sang cọc z, sau đó chuyển đĩa to nhất đó sang cọc y và cuối cùng lại coi đĩa to nhất đó là cọc, chuyển $n-1$ đĩa còn lại đang ở cọc z sang cọc y chồng lên đĩa to nhất.

Như vậy để chuyển n đĩa (bài toán lớn), ta quy về hai phép chuyển $n-1$ đĩa (bài toán nhỏ) và một phép chuyển 1 đĩa (bài toán cơ sở). Cách làm đó được thể hiện trong thủ tục đệ quy dưới đây:

```
procedure Move(n, x, y: Integer); //Thủ tục chuyển n đĩa từ cọc x sang cọc y
begin
  if n = 1 then
    Output ← Chuyển 1 đĩa từ x sang y
  else //Để chuyển n > 1 đĩa từ cọc x sang cọc y, ta chia làm 3 công đoạn
    begin
      Move(n - 1, x, 6 - x - y); //Chuyển n - 1 đĩa từ cọc x sang cọc trung gian
      Move(1, x, y); //Chuyển đĩa to nhất từ x sang y
      Move(n - 1, 6 - x - y, y); //Chuyển n - 1 đĩa từ cọc trung gian sang cọc y
    end;
end;
```

Việc chuyển n đĩa bây giờ trở nên rất đơn giản qua một lệnh gọi $Move(n,1,2)$

Có thể chứng minh số phép chuyển đĩa để giải bài toán Tháp Hà Nội với n đĩa là 2^n-1 bằng quy nạp:

Rõ ràng là tính chất này đúng với $n=1$, bởi ta cần $2^1-1=1$ phép chuyển để thực hiện yêu cầu.

Với $n>1$; giả sử (quy nạp) rằng để chuyển $n-1$ đĩa giữa hai cọc ta cần $2^{n-1}-1$ phép chuyển, khi đó để chuyển n đĩa từ cọc x sang cọc y, nhìn vào giải thuật đệ quy ta có thể thấy rằng trong trường hợp này nó cần $(2^{n-1}-1)+1+(2^{n-1}-1)=2^n-1$ phép chuyển. Tính chất được chứng minh đúng với n.

Vậy thì công thức này sẽ đúng với mọi n.

14.3.3. Nhân đa thức

❑ Bài toán

Cho hai đa thức $A(x)=\sum_{i=0}^{m} a_i x^i$ và $B(x)=\sum_{j=0}^{n} b_j x^j$. Bài toán đặt ra là tìmđa th ức $C(x)=A(x)B(x)=\sum_{k=0}^{m+n} c_k x^k$.

Một đa thức hoàn toàn xác định nếu ta biết được giá trị các hệ số của nó. Như trong bài toán này, công việc chính là đi tìm các giá trị c_k:

$$c_k = \sum_{i+j=k} a_i b_j \,, \forall k: 0 \le k \le m+n$$

❑ Phương pháp tính trực tiếp

Để tìm đa thức $C(x)$ một cách trực tiếp, có thể sử dụng thuật toán sau:

```
for k := 0 to m + n do c[k] := 0; //Khởi tạo các hệ số đa thức C bằng 0
//Xét mọi cặp hệ số a[i], b[j], cộng dồn tích a[i] * b[j] vào c[i + j]
for i := 0 to m do
  for j := 0 to n do
    c[i + j] := c[i + j] + a[i] * b[j];
```

Dễ thấy rằng thời gian thực hiện giải thuật nhân đa thức trực tiếp là $\Theta(mn)$. Dưới đây chúng ta sẽ tiếp cận bài toán theo phương pháp chia để trị và giới thiệu một thuật toán mới.

❑ Thuật toán chia để trị

Đa thức $A(x)$ có bậc m và đa thức $B(x)$ có bậc n. Nếu một trong hai đa thức này có bậc 0 thì bài toán trở thành nhân một đa thức với một số, có thể coi đây là trường hợp đơn giản và có thể tính trực tiếp được. Nếu hai đa thức này đều có bậc lớn hơn 0, không giảm tính tổng quát, ta có thể coi bậc của hai đa thức này bằng nhau và là số lẻ, bởi vì việc tăng bậc của một đa thức và gán hệ số bậc cao nhất của nó bằng 0 không làm ảnh hưởng tới kết quả tính tích hai đa thức.

Bây giờ ta có hai đa thức $A(x)$ và $B(x)$ cùng bậc $n = 2k - 1$:

$$A(x) = \sum_{i=0}^{2k-1} a_i x^i \,; B(x) = \sum_{i=0}^{2k-1} b_i x^i$$

Xét đa thức $A(x)$:

$$\begin{aligned} A(x) &= a_{(2k-1)}x^{2k-1} + \cdots + a_k x^k + a_{k-1}x^{k-1} + \cdots + a_0 \\ &= x^k \underbrace{(a_{2k-1}x^{k-1} + \cdots + a_k)}_{A_h(x)} + \underbrace{(a_{k-1}x^{k-1} + \cdots + a_0)}_{A_l(x)} \end{aligned} \qquad (14.1)$$

Vậy nếu chia dãy hệ số của $A(x)$ làm hai nửa bằng nhau, nửa những hệ số cao tương ứng với một đa thức $A_h(x)$, và nửa những hệ số thấp tương ứng với một đa thức $A_l(x)$. Thì $A_h(x)$ và $A_l(x)$ lần lượt là thương và dư của phép chia đa thức $A(x)$ cho x^k:

$$A(x) = x^k A_h(x) + A_l(x) \qquad (14.2)$$

Tương tự như vậy ta có thể phân tích $B(x)$ thành:

$$B(x) = x^k B_h(x) + B_l(x) \tag{14.3}$$

Các đa thức $A_h(x)$, $A_l(x)$, $B_h(x)$ và $B_l(x)$ đều là đa thức bậc $k-1$. Xét tích $A(x)B(x)$:

$$\begin{aligned} A(x)B(x) &= \Big(x^k A_h(x) + A_l(x)\Big)\Big(x^k B_h(x) + B_l(x)\Big) \\ &= x^{2k} \underbrace{\big(A_h(x)B_h(x)\big)}_{P(x)} + x^k \underbrace{\big(A_h(x)B_l(x) + A_l(x)B_h(x)\big)}_{R(x)} \\ &\quad + \underbrace{A_l(x)B_l(x)}_{Q(x)} \end{aligned} \tag{14.4}$$

Để tính $A(x)B(x)$, ta quy về việc tìm ba đa thức $P(x), Q(x), R(x)$. Việc tính $P(x)$ cũng như $Q(x)$, đều cần một phép nhân đa thức bậc $k-1$. Sau khi tính được $P(x)$ và $Q(x)$ thì $R(x)$ cũng có thể tính mà chỉ dùng một phép nhân đa thức bậc $k-1$ theo cách:

Dùng một phép nhân đa thức bậc $k-1$ để tính:

$$S(x) = \big(A_h(x) + A_l(x)\big)\big(B_h(x) + B_l(x)\big) \tag{14.5}$$

Sau đó tính

$$R(x) = S(x) - P(x) - Q(x) \tag{14.6}$$

Bạn có thể kiểm chứng đẳng thức $S(x) = P(x) + Q(x) + R(x)$ để suy ra tính đúng đắn của công thức tính.

Xét công thức (14.4), nếu gọi $T(n)$ là thời gian thực hiện phép nhân hai đa thức bậc n thì có thể nhận thấy rằng ngoài thời gian thực hiện ba phép nhân đa thức kể trên, các phép toán khác (tính tổng đa thức, nhân đa thức với đơn thức) có thời gian thực hiện $\Theta(n)$. Vậy

$$T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n) \tag{14.7}$$

Áp dụng Định lý 3-7 (Định lý Master), trường hợp 1, ta có $T(n) = \Theta(n^{\lg_2 3}) \approx \Theta(n^{1,585})$

Điều này chỉ ra rằng thuật toán chia để trị tỏ ra tốt hơn thuật toán tính trực tiếp.

❑ Cài đặt

Chúng ta sẽ cài đặt thuật toán nhân hai đa thức $A(x) = a_m x^m + a_{m-1}x^{m-1} + \cdots + a_0$ và $B(x) = b^n x^n + b_{(n-1)}x^{n-1} \ \ldots + b_0$ với khuôn dạng nhập xuất dữ liệu như sau:

Input

- Dòng 1 chứa hai số tự nhiên m, n lần lượt là bậc của đa thức $A(x)$ và đa thức $B(x)$
- Dòng 2 chứa $m+1$ số thực $a_m, a_{m-1}, \ldots., a_0$ là các hệ số của $A(x)$
- Dòng 3 chứa $n+1$ số thực $b_n, b_{n-1}, \ldots, b_0$ là các hệ số của $B(x)$

Output

Các hệ số của đa thức $C(x)$ từ hệ số bậc cao nhất đến hệ số bậc thấp nhất

Sample Input	Sample Output
1 2 2.0 1.0 1.0 2.0 3.0	2.0 5.0 8.0 3.0

$A(x) = 2x + 1$
$B(x) = x^2 + 2x + 3$
$C(x) = 2x^3 + 5x^2 + 8x + 3$

Nhân đa thức

```
{$MODE OBJFPC}
program PolynomialMultipication;
uses Math;
type
  TPolynomial = array of Real; //Đa thức được biểu diễn bằng mảng động các hệ số
var
  a, b, c: TPolynomial;
  m, n: LongInt;

//Nếu hai đa thức p, q khác bậc, thêm vài hạng tử bậc cao nhất có hệ số 0 để bậc của chúng bằng nhau
procedure Equalize(var p, q: TPolynomial);
var
  lenP, lenQ, len: LongInt;
begin
  lenP := Length(p);
  lenQ := Length(q);
  len := Max(lenP, lenQ);
  SetLength(p, len);
  SetLength(q, len);
  if lenP < len then //p bị tăng bậc
    FillChar(p[lenP], (len - lenP) * SizeOf(Real), 0); //Đặt p[lenP...len - 1] := 0
  if lenQ < Length(q) then //q bị tăng bậc
    FillChar(q[lenQ], (len - lenQ) * SizeOf(Real), 0); //Đặt q[lenQ...len – 1] := 0;
end;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(m, n);
  SetLength(a, m + 1);
  SetLength(b, n + 1);
  for i := m downto 0 do Read(a[i]);
  ReadLn;
  for i := n downto 0 do Read(b[i]);
  Equalize(a, b); //Làm bậc của hai đa thức bằng nhau
end;

//Đa thức p có bậc len - 1 được phân ra làm hai đa thức pL, pH
procedure DivMod(const p: TPolynomial; out pL, pH: TPolynomial);
var
  len, sublen: LongInt;
begin
  len := Length(p); //len là số hệ số của p
  sublen := (len + 1) div 2; //Tính sublen = Ceil(len / 2)
```

```
  pL := Copy(p, 0, sublen); //Đưa phần hệ số thấp sang pL
  pH := Copy(p, sublen, len - sublen); //Đưa phần hệ số cao sang pH
  Equalize(pL, pH); //Nếu len lẻ, pH có thể có bậc nhỏ hơn pL → làm cho chúng cùng bậc
end;

//Định nghĩa toán tử gán: gán đa thức bằng một số thực v
operator := (v: Real): TPolynomial;
begin
  SetLength(Result, 1);
  Result[0] := v;
end;

//Tính tổng hai đa thức
operator +(const a, b: TPolynomial): TPolynomial;
var
  i: Integer;
begin
  SetLength(Result, Length(a));
  for i := 0 to High(Result) do
    Result[i] := a[i] + b[i];
end;

//Tính hiệu hai đa thức
operator -(const a, b: TPolynomial): TPolynomial;
var
  i: Integer;
begin
  SetLength(Result, Length(a));
  for i := 0 to High(Result) do
    Result[i] := a[i] - b[i];
end;

//Tính tích hai đa thức
operator *(const a, b: TPolynomial): TPolynomial;
var
  aH, aL, bH, bL:  TPolynomial;
  P, Q, R: TPolynomial;
  i, k: LongInt;
begin
  if High(a) = 0 then //Trường hợp cơ sở: Hai đa thức bậc 0
    begin
      Result := a[0] * b[0];
      Exit;
    end;
  //Tách mỗi đa thức thành 2 đa thức
  DivMod(a, aL, aH);
  DivMod(b, bL, bH);
  k := Length(aH);
  P := aH * bH;
  Q := aL * bL;
  R := (aH + aL) * (bH + bL) - (aH * bH + aL * bL);
  //P, Q, R đều có bậc 2k - 2, Kết qua có bậc = 4k - 2
  SetLength(Result, 4 * k - 1);
  //Trước hết cộng P * x^(2k) vào Result ↔ Điền các hệ số của P vào Result[2k...4k - 2]
  for i := 0 to High(P) do
    Result[i + 2 * k] := P[i];
  //Tiếp theo cộng Q vào Result ↔ Điền các hệ số của Q vào Result[0...2k - 2]
```

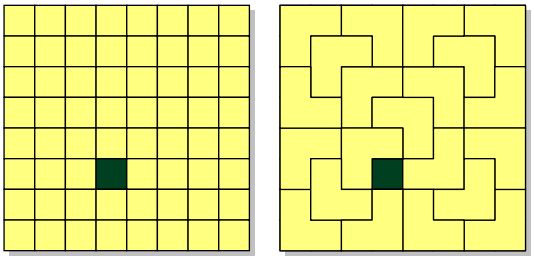
```
  for i := 0 to High(Q) do
    Result[i] := Q[i];
  Result[2 * k - 1] := 0; //Còn duy nhất một hệ số của Result chưa khởi tạo, đặt = 0
  //Cuối cùng cộng R * x^k vào Result
  for i := 0 to High(R) do
    Result[i + k] := Result[i + k] + R[i];
end;

procedure PrintResult;
var
  i, d: Integer;
begin
  //Loại bỏ những hạng tử cao nhất bằng 0 từ đa thức kết quả
  d := High(c);
  while (d > 0) and (c[d] = 0) do Dec(d);
  SetLength(c, d + 1);
  //In kết quả
  for i := High(c) downto 0 do Write(c[i]:1:1, ' ');
  WriteLn;
end;

begin
  Enter;
  c := a * b;
  PrintResult;
end.
```

Bài tập 14-1.

Khi phải làm việc với các số lớn vượt quá phạm vi biểu diễn của các kiểu dữ liệu chuẩn, người ta có thể biểu diễn các số lớn bằng mảng các chữ số: $n = \overline{n_d n_{d-1} \dots n_0} = \sum_{i=0}^{d} n_i 10^i \ (\forall i: 0 \le n_i \le 9)$, và cài đặt các phép toán số học trên số lớn để thực hiện các phép toán này như với các kiểu dữ liệu chuẩn. Thuật toán nhân hai số lớn [**12**] có cách làm tương tự như thuật toán nhân hai đa thức. Hãy cài đặt thuật toán này để nhân hai số lớn.

Bài tập 14-2.

Cho một bảng ô vuông kích thước $2^k \times 2^k$ ô vuông đơn vị, trên đó ta bỏ đi một ô. Hãy tìm cách lát kín các ô còn lại của bảng bằng các mảnh ghép dạng  sao cho không có hai mảnh nào chồng nhau. Ví dụ với $k = 3$



Bài tập 14-3.

Xét phép nhân hai ma trận $A_{m\times n}$ và $B_{n\times p}$ để được ma trận $C_{m\times p}$:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} \times b_{jk}\ , (1 \le i \le m, 1 \le k \le p)$$

Ma trận C có thể tính trực tiếp bằng thuật toán:

```
for i := 1 to m do
  for k := 1 to p do
    begin
      c[i, k] := 0;
      for j := 1 to n do c[i, k] := c[i, k] + a[i, j] * b[j, k];
    end;
```

Thuật toán tính trực tiếp có thời gian thực hiện $\Theta(mnp)$.

Xét thuật toán chia để trị: Thêm những hàng 0 và cột 0 vào ma trận để ba ma trận A, B, C đều có kích thước $2^s \times 2^s$. Chia mỗi ma trận làm 4 phần bằng nhau, mỗi phần là một ma trận con kích thước $2^{s-1} \times 2^{s-1}$:

$$A = \begin{bmatrix} A^{(11)} & A^{(12)} \\ A^{(21)} & A^{(22)} \end{bmatrix}; B = \begin{bmatrix} B^{(11)} & B^{(12)} \\ B^{(21)} & B^{(22)} \end{bmatrix}; C = \begin{bmatrix} C^{(11)} & C^{(12)} \\ C^{(21)} & C^{(22)} \end{bmatrix}$$

khi đó nếu $C = A \times B$ thì:

$$C^{(11)} = A^{(11)}B^{(11)} + A^{(12)}B^{(21)}$$
$$C^{(12)} = A^{(11)}B^{(12)} + A^{(12)}B^{(22)}$$
$$C^{(21)} = A^{(21)}B^{(11)} + A^{(22)}B^{(21)}$$
$$C^{(22)} = A^{(21)}B^{(12)} + A^{(22)}B^{(22)}$$

Để tính ma trận C theo cách này, chúng ta cần 8 phép nhân ma trận con. Nếu đặt kích thước của mỗi ma trận là $n = 2^s$ thì thời gian thực hiện giải thuật có thể biểu diễn bằng hàm:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Áp dụng Định lý 3-7 (Định lý Master), trường hợp 1, ta có $T(n) = \Theta(n^3)$, không có gì tốt hơn thuật toán tính trực tiếp.

Tuy nhiên các ma trận con $C^{(11)}, C^{(12)}, C^{(21)}, C^{(22)}$ có thể tính chỉ cần 7 phép nhân ma trận con bằng thuật toán Strassen [**18**]. Các phép nhân ma trận con cần thực hiện là:

$$M_1 = \left(A^{(11)} + A^{(22)}\right)\left(B^{(11)} + B^{(22)}\right)$$
$$M_2 = \left(A^{(21)} + A^{(22)}\right)B^{(11)}$$
$$M_3 = A^{(11)}\left(B^{(12)} - B^{(22)}\right)$$

$$M_4 = A^{(22)}\left(B^{(21)} - B^{(11)}\right)$$
$$M_5 = \left(A^{(11)} + A^{(12)}\right)B^{(22)}$$
$$M_6 = \left(A^{(21)} - A^{(11)}\right)\left(B^{(11)} + B^{(12)}\right)$$
$$M_7 = \left(A^{(12)} - A^{(22)}\right)\left(B^{(21)} + B^{(22)}\right)$$

Khi đó:

$$C^{(11)} = M_1 + M_4 - M_5 + M_7$$
$$C^{(12)} = M_3 + M_5$$
$$C^{(21)} = M_2 + M_4$$
$$C^{(22)} = M_1 - M_2 + M_3 + M_6$$

Xét về thời gian thực hiện giải thuật Strassen, ta có:

$$(n) = 7T(n/2) + \Theta(n^2)$$

Áp dụng Định lý 3-7 (Định lý Master), trường hợp 1, ta có $T(n) = \Theta\left(n^{\lg 7}\right) \approx \Theta(n^{2,807})$.

Hãy cài đặt thuật toán Strassen tính tích hai ma trận.

Tự đọc thêm về thuật toán Coppersmith-Winograd [**4**], hiện đang là thuật toán nhanh nhất hiện nay để nhân hai ma trận trong thời gian $O(n^{2,376})$

Bài tập 14-4.

Trên mặt phẳng cho ***n*** hình tròn, hãy tính diện tích miền mặt phẳng bị ***n*** hình tròn này chiếm chỗ.

Gợi ý: Ta có thể coi các hình tròn đã cho không trùng nhau. Xét một hình chữ nhật ***R*** chứa tất cả các hình tròn, ta cần tính phần diện tích của ***R*** bị các hình tròn chiếm chỗ. Nếu ***R*** có giao với chỉ một hình tròn, ta chỉ cần đo diện tích phần giao. Nếu ***R*** có giao với nhiều hơn một hình tròn, ta chia ***R*** làm bốn hình chữ nhật con ở bốn góc và tính tổng diện tích phần giao của bốn hình chữ nhật con này với các hình tròn bằng đệ quy.

Bài tập 14-5. (Cặp điểm gần nhất)

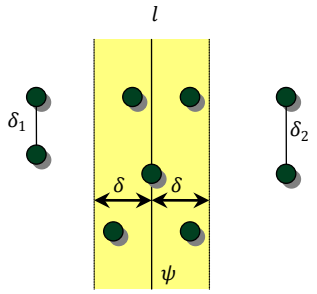
Trên mặt phẳng với hệ tọa độ trực chuẩn Oxy cho n điểm phân biệt: $P_1(x_1, y_1), P_2(x_2, y_2), \ldots, P_n(x_n, y_n)$. Hãy tìm hai điểm gần nhau nhất.

Khoảng cách giữa hai điểm $P_i(x_i, y_i)$ và $P_j\left(x_j, y_j\right)$ là khoảng cách Euclid:

$$\left|P_iP_j\right| = \sqrt{\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2}$$

Để tìm cặp điểm gần nhất, chúng ta có thuật toán $\Theta(n^2)$: thử tất cả các bộ đôi và đo khoảng cách. Tuy nhiên có một thuật toán tốt hơn dựa trên chiến lược chia để trị:

- Phân hoạch tập các điểm làm hai tập rời nhau S_L và S_R, một tập gồm $\lfloor n/2 \rfloor$ điểm và một tập gồm $\lceil n/2 \rceil$ điểm thỏa mãn: Mọi điểm thuộc S_L đều có hoành độ $\leq$ mọi điểm thuộc S_R. Điều này có thể thực hiện bằng cách tìm một đường thẳng l song song với trục tung, đặt tại hoành độ là trung vị của các hoành độ, các điểm nằm bên trái l đưa vào S_L, các điểm nằm bên phải l vào S_R, còn các điểm nằm trên l sẽ được chia vào S_L và S_R để giữ cho lực lượng của hai tập hơn kém nhau không quá 1.
- Lần lượt giải bài toán tìm cặp điểm gần nhất trên S_L và S_R, gọi δ_L và δ_R lần lượt là khoảng cách ngắn nhất tìm được trên hai tập. Đặt $\delta = \min(\delta_L, \delta_R)$.
- Nhận xét rằng nếu trong n điểm đã cho có một cặp điểm có khoảng cách ngắn hơn δ thì hai điểm đó phải có một điểm thuộc S_L và một điểm thuộc S_R. Tức là cả hai điểm phải nằm trong một dải ψ giới hạn bởi hai đường thẳng song song cách l một khoảng δ (Hình 14-2). Việc còn lại là tìm cặp điểm gần nhất trong dải này và cực tiểu hóa δ nếu cặp điểm tìm được có khoảng cách ngắn hơn cặp điểm đang có.



Hình 14-2. Thuật toán chia để trị tìm cặp điểm gần nhất

Dễ thấy rằng để hai điểm có khoảng cách nhỏ hơn δ thì điều kiện cần là độ chênh lệch tung độ giữa hai điểm phải nhỏ hơn δ.

Hãy chứng minh rằng nếu $P \in \psi$ thì có không quá 8 điểm khác thuộc ψ có độ chênh lệch tung độ nhỏ hơn δ so với P.

Hãy chỉ ra rằng nếu ban đầu các điểm được sắp xếp theo thứ tự tăng dần của tung độ thì có thể tìm thuật toán $O(n)$ để tìm cặp điểm gần nhất trong ψ nếu cặp điểm đó có khoảng cách nhỏ hơn δ. (*gợi ý: chiếu các điểm của ψ lên trục tung, với mỗi điểm, từ ảnh chiếu của nó xét 8 ảnh lân cận và đo khoảng cách từ điểm đang xét tới 8 điểm tương ứng*).

Hãy tìm thuật toán $O(n \lg n)$ và lập trình thuật toán đó để giải bài toán tìm cặp điểm gần nhất trong không gian hai chiều.

Hãy tìm thuật toán $O(n(\lg n)^{d-1})$ để giải bài toán tìm cặp điểm gần nhất trong không gian Euclid d chiều.

Bài 15. Quy hoạch động

Trong khoa học tính toán, quy hoạch động là một phương pháp hiệu quả để giải các bài toán tối ưu mang bản chất đệ quy.

Tên gọi "*Quy hoạch động*" – (*Dynamic Programming*) được Richard Bellman đề xuất vào những năm 1940 để mô tả một phương pháp giải các bài toán mà người giải cần đưa ra lần lượt một loạt các quyết định tối ưu. Tới năm 1953, Bellman chỉnh lại tên gọi này theo nghĩa mới và đưa ra nguyên lý giải quyết các bài toán bằng phương pháp quy hoạch động.

Không có một thuật toán tổng quát để giải tất cả các bài toán quy hoạch động. Mục đích của bài này là cung cấp một cách tiếp cận mới trong việc giải quyết các bài toán tối ưu mang bản chất đệ quy, đồng thời đưa ra các ví dụ để người đọc hình thành các kỹ năng trong việc tiếp cận và giải quyết các bài toán quy hoạch động.

15.1. Công thức truy hồi

15.1.1. Bài toán ví dụ

Chúng ta sẽ tìm hiểu về công thức truy hồi và phương pháp giải công thức truy hồi qua một ví dụ:

Cho số tự nhiên $n \leq 400$. Hãy cho biết có bao nhiêu cách phân tích số n thành tổng của dãy các số nguyên dương, các cách phân tích là hoán vị của nhau chỉ tính là một cách.

Ví dụ: $n = 5$ có 7 cách phân tích:

1. 5 = 1 + 1 + 1 + 1 + 1
2. 5 = 1 + 1 + 1 + 2
3. 5 = 1 + 1 + 3
4. 5 = 1 + 2 + 2
5. 5 = 1 + 4
6. 5 = 2 + 3
7. 5 = 5

Ta coi trường hợp $n = 0$ cũng có 1 cách phân tích thành tổng các số nguyên dương (0 là tổng của dãy rỗng)

Để giải bài toán này, mục 13.3.5 đã giới thiệu phương pháp liệt kê tất cả các cách phân tích và đếm số cấu hình. Bây giờ hãy thử nghĩ xem, có cách nào *tính ngay ra số lượng các cách phân tích mà không cần phải liệt kê* hay không. Bởi vì khi số cách phân tích

tương đối lớn, phương pháp liệt kê tỏ ra khá chậm. (ví dụ chỉ với $n = 100$ đã có 190.569.292 cách phân tích).

15.1.2. Đặt công thức truy hồi

Nếu gọi $f[m, v]$ là số cách phân tích số v thành tổng các số nguyên dương $\leq m$. Khi đó các cách phân tích số v thành tổng một dãy các số nguyên dương $\leq m$ có thể chia làm hai loại:

- Loại 1: Không chứa số m trong phép phân tích, khi đó số cách phân tích loại này chính là số cách phân tích số v thành tổng các số nguyên dương $\leq m - 1$, tức là bằng $f[m - 1, v]$.
- Loại 2: Có chứa ít nhất một số m trong phép phân tích. Khi đó nếu trong các cách phân tích loại này ta bỏ đi số m đó thì ta sẽ được các cách phân tích số $v - m$ thành tổng các số nguyên dương $\leq m$. Có nghĩa là về mặt số lượng, số các cách phân tích loại này bằng $f[m, v - m]$

Trong trường hợp $m > v$ thì rõ ràng chỉ có các cách phân tích loại 1, còn trong tư ờng hợp $m \leq v$ thì sẽ có cả các cách phân tích loại 1 và loại 2. Vì thế:

$$f[m, v] = \begin{cases} f[m-1, v], & \text{nếu } m > v \\ f[m-1, v] + f[m, v-m], & \text{nếu } m \leq v \end{cases} \qquad (15.1)$$

Ta có công thức xây dựng $f[m, v]$ từ $f[m - 1, v]$ và $f[m, v - m]$. Công thức này có tên gọi là *công thức truy hồi* (*recurrence*) đưa việc tính $f[m, v]$ về việc tính các $f[m', v']$ với dữ liệu nhỏ hơn. Tất nhiên cuối cùng ta sẽ quan tâm đến $f[n, n]$: Số các cách phân tích n thành tổng các số nguyên dương $\leq n$.

Ví dụ với $n = 5$, các giá trị $f[m, v]$ có thể cho bởi bảng:

f	0	1	2	3	4	5	v
0	1	0	0	0	0	0	
1	1	1	1	1	1	1	
2	1	1	2	2	3	3	
3	1	1	2	3	4	5	
4	1	1	2	3	5	6	
5	1	1	2	3	5	7	
m							

Từ công thức tính, ta thấy rằng $f[m, v]$ được tính qua giá trị của một phần tử ở hàng trên ($f[m - 1, v]$) và một phần tử ở cùng hàng, bên trái: (f$[m, v - m]$) . Ví dụ $f[5,5]$ sẽ được tính bằng $f[4,5] + f[5,0]$, hay $f[3,5]$ sẽ được tính bằng $f[2,5] + f[3,2]$. Chính vì vậy để tính $f[m, v]$ thì $f[m - 1, v]$ và $f[m, v - m]$ phải được tính trước. Suy ra thứ tự

hợp lý để tính các phần tử trong bảng thì ta sẽ phải tính lần lượt các hàng từ trên xuống và trên mỗi hàng thì tính theo thứ tự từ trái qua phải.

Điều đó có nghĩa là ban đầu ta phải tính hàng 0 của bảng. Theo định nghĩa $f[0,v]$ là số dãy có các phần tử là số nguyên dương ≤ 0 mà tổng bằng v, theo quy ước của đầu bài thì $f[0,0] = 1$ và dễ thấy rằng $f[0,v] = 0, \forall v > 0$.

Bây giờ giải thuật đếm trở nên rất đơn giản:

- Khởi tạo hàng 0 của bảng f: $f[0,0] \coloneqq 1$; $f[0,v] \coloneqq 0, \forall v > 0$
- Dùng công thức truy hồi tính ra tất cả các phần tử của bảng f
- Cuối cùng cho biết $f[n,n]$ là số cách phần tích cần tìm

Chúng ta sẽ cài đặt chương trình đếm số cách phân tích số với khuôn dạng nhập/xuất dữ liệu như sau:

Input

Số tự nhiên $n \leq 400$

Output

Số cách phân tích số n

Sample Input	Sample Output
400	6727090051741041926 Analyses

Đếm số cách phân tích số

```
program Counting1;
const
  max = 400;
var
  f: array[0..max, 0..max] of Int64;
  n, m, v: LongInt;
begin
  Readln(n);
  //Khởi tạo hàng 0 của bảng
  FillChar(f[0, 1], n * SizeOf(Int64), 0);
  f[0, 0] := 1;
  //Tính bảng f
  for m := 1 to n do
    for v := 0 to n do
      if v < m then f[m, v] := f[m - 1, v]
      else f[m, v] := f[m - 1, v] + f[m, v - m];
  Writeln(f[n, n], ' Analyses');
end.
```

15.1.3. Cải tiến thứ nhất

Cách làm trên có thể tóm tắt lại như sau: Khởi tạo hàng 0 của bảng, sau đó dùng hàng 0 tính hàng 1, dùng hàng 1 tính hàng 2, v.v… tới khi tính được hết hàng n. Có thể nhận

thấy rằng khi đã tính xong hàng thứ m thì việc lưu trữ các hàng từ 0 tới $m-1$ là không cần thiết bởi vì việc tính hàng $+1$ chỉ phụ thuộc các giá trị lưu trữ trên hàng m. Do đó quá trình tính toán chỉ cần lưu trữ hai hàng của mảng f: hàng x tương ứng với hàng vừa được tính của bảng f và hàng y tương ứng với hàng sắp tính của bảng f: Đầu tiên hàng x được gán các giá trị tương ứng trên hàng 0 của bảng f; sau đó, công thức truy hồi sẽ được áp dụng để tính hàng y từ hàng x. Tiếp theo, hai hàng x và y sẽ đổi vai trò cho nhau và công thức truy hồi tiếp tục dùng hàng x tính hàng y, hàng y sau khi tính sẽ gồm các giá trị tương ứng trên hàng 2 của bảng f, v.v… Vậy ta có một cách cài đặt cải tiến sau:

💻 Đếm số cách phân tích số (Tính xoay hai hàng)

```
program Counting2;
const
  max = 400;
var
  f: array[0..1, 0..max] of Int64;
  x, y: LongInt;
  n, m, v: LongInt;
begin
  Readln(n);
  FillChar(f[0, 1], n * SizeOf(Int64), 0);
  f[0, 0] := 1;
  x := 0; y := 1; //x: hàng đã có, y: hàng đang tính
  for m := 1 to n do
    begin //Dùng hàng x tính hàng y
      for v := 0 to n do
        if v < m then f[y, v] := f[x, v]
        else f[y, v] := f[x, v] + f[y, v - m];
      x := 1 - x; y := 1 - y; //Đảo vai trò x và y, tính xoay lại
    end;
  Writeln(f[x, n], ' Analyses');
end.
```

15.1.4. Cải tiến thứ hai

Ta vẫn còn cách tốt hơn nữa, tại mỗi bước, ta chỉ cần lưu lại một hàng của bảng f như mảng một chiều, sau đó áp dụng công thức truy hồi lên mảng đó tính lại chính nó, để sau khi tính, mảng một chiều sẽ chứa các giá trị trên hàng kế tiếp của bảng f.

💻 Đếm số cách phân tích số (Tính trực tiếp trên một hàng)

```
program Counting3;
const
  max = 400;
var
  f: array[0..max] of Int64;
  n, m, v: LongInt;
begin
  Readln(n);
  FillChar(f[1], n * SizeOf(Int64), 0);
  f[0] := 1;
  for m := 1 to n do
```

```
    for v := m to n do //Chỉ cần tính lại các f[v] với v ≥ m
      Inc(f[v], f[v - m]);
  Writeln(f[n], ' Analyses');
end.
```

15.1.5. Cài đặt đệ quy

Xem lại công thức truy hồi tính $f[m,v] = f[m-1,v] + f[m,v-m]$, ta nhận thấy rằng để tính $f[m,v]$ ta phải biết được chính xác $f[m-1,v]$ và $f[m,v-m]$. Như vậy việc xác định thứ tự tính các phần tử trong bảng f (phần tử nào tính trước, phần tử nào tính sau) là quan trọng. Trong trường hợp thứ tự này khó xác định, ta có thể tính dựa trên một hàm đệ quy mà không cần phải quan tâm tới thứ tự tính toán. Trước khi trình bày phương pháp này, ta sẽ thử viết một hàm đệ quy theo cách quen thuộc để giải công thức truy hồi:

💻 Đếm số cách phân tích số (Đệ quy)

```
{$MODE OBJFPC}
program Counting4;
var
  n: LongInt;

function Getf(m, v: LongInt): Int64;
begin
  if m = 0 then //phần neo
    if v = 0 then Result := 1
    else Result := 0
  else //phần đệ quy
    if v < m then Result := Getf(m - 1, v)
    else Result := Getf(m - 1, v) + Getf(m, v - m);
end;

begin
  Readln(n);
  Writeln(Getf(n, n), ' Analyses');
end.
```

Phương pháp cài đặt này tỏ ra khá chậm vì với mỗi giá trị của m và v, hàm $Getf(m,v)$ có thể bị tính nhiều lần (bài sau sẽ giải thích rõ hơn điều này). Ta có thể cải tiến bằng cách kết hợp hàm đệ quy $Getf(m,v)$ với một mảng hai chiều f. Ban đầu các phần tử của f được coi là "chưa biết" (bằng cách gán một giá trị đặc biệt). Hàm $Getf(m,v)$ trước hết sẽ tra cứu tới $f[m,v]$, nếu $f[m,v]$ chưa biết thì hàm $Getf(m,v)$ sẽ gọi đệ quy để tính giá trị của $f[m,v]$ rồi dùng giá trị này gán cho kết quả hàm, còn nếu $f[m,v]$ đã biết thì hàm này chỉ việc gán kết quả hàm là $f[m,v]$ mà không cần gọi đệ quy để tính toán nữa.

💻 Đếm số cách phân tích số (Đệ quy kết hợp với bảng giá trị)

```
{$MODE OBJFPC}
program Counting5;
const
  max = 400;
var
```

```
  n: LongInt;
  f: array[0..max, 0..max] of Int64;

function Getf(m, v: LongInt): Int64; //Tính f[m, v]
begin
  if f[m, v] = -1 then //Nếu f[m, v] chưa được tính thì đi tính f[m, v]
    if m = 0 then //phần neo
      if v = 0 then f[m, v] := 1
      else f[m, v] := 0
    else //phần đệ quy
      if v < m then f[m, v] := Getf(m - 1, v)
      else f[m, v] := Getf(m - 1, v) + Getf(m, v - m);
  Result := f[m, v]; //Gán kết quả hàm bằng f[m, v]
end;

begin
  Readln(n);
  FillChar(f, SizeOf(f), $FF); //Các phần tử của f được gán giá trị đặc biệt (-1)
  Writeln(Getf(n, n), ' Analyses');
end.
```

Việc sử dụng phương pháp đệ quy để giải công thức truy hồi là một kỹ thuật đáng lưu ý, vì khi gặp một công thức truy hồi phức tạp, khó xác định thứ tự tính toán thì phương pháp này tỏ ra rất hiệu quả, hơn thế nữa nó làm rõ hơn bản chất đệ quy của công thức truy hồi.

Ngoài ra, nếu nhập vào $n = 5$ và in ra bảng f sau khi tính, ta sẽ được bảng sau:

f	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	-1	3
3	1	1	2	-1	-1	5
4	1	1	-1	-1	-1	6
5	1	-1	-1	-1	-1	7

Nhìn vào bảng kết quả f với $n = 5$, ta thấy còn rất nhiều phần tử mang giá trị -1 (chưa được tính), điều này chỉ ra rằng cài đặt đệ quy còn cho phép chúng ta bỏ qua không cần tính những phần tử không tham gia vào việc tính $f[n, n]$ và làm tăng tốc đáng kể quá trình tính toán.

15.1.6. Tóm tắt kỹ thuật giải công thức truy hồi

Công thức truy hồi có vai trò quan trọng trong bài toán đếm, chẳng hạn đếm số cấu hình thoả mãn điều kiện nào đó, hoặc tìm tương ứng giữa một số thứ tự và một cấu hình.

Ví dụ trong bài này là thuộc dạng đếm số cấu hình thoả mãn điều kiện nào đó. Tuy nhiên nếu đặt lại bài toán: Nếu đem tất cả các dãy số nguyên dương không giảm có tổng bằng n sắp xếp theo thứ tự từ điển thì dãy thứ p là dãy nào?, hoặc cho một dãy số nguyên dương

không giảm có tổng bằng n, hỏi dãy đó đứng thứ mấy?. Lúc này ta sẽ có bài toán tìm tương ứng giữa một số thứ tự và một cấu hình - Một dạng bài toán có thể giải quyết hiệu quả bằng công thức truy hồi.

Ta cũng đã khảo sát một vài kỹ thuật phổ biến để giải công thức truy hồi: Phương pháp tính trực tiếp bảng giá trị, phương pháp tính luân phiên (chỉ phải lưu trữ các phần tử tích cực), phương pháp tự cập nhật giá trị, và phương pháp đệ quy kết hợp với bảng lưu trữ. Những phương pháp này sẽ đóng vai trò quan trọng trong việc cài đặt chương trình giải công thức truy hồi - Hạt nhân của bài toán quy hoạch động.

15.1.7. ★ Nói thêm về bài toán phân tích số

Bài toán *phân tích số* (*integer partitioning*) là một bài toán quan trọng trong lĩnh vực số học và vật lý. Chúng ta không đi sâu vào chi tiết hai lĩnh vực này mà chỉ nói tới một kết quả đã được chứng minh bằng lý thuyết số học về các *số ngũ giác* (*pentagonal numbers*):

Những số ngũ giác là những số nguyên có dạng $p(k) = \frac{k(3k-1)}{2}$ với $k \in \mathbb{Z}$. Bằng cách cho k nhận lần lượt các giá trị trong dãy $0, +1, -1, +2, -2, +3, -3, \ldots$ ta có thể liệt kê các số ngũ giác theo thứ tự tăng dần là:

$$0,1,2,5,7,12,15, \ldots$$

Gọi $f(n)$ là số cách phân tích số n thành tổng của các số nguyên dương (không tính hoán vị). Quy ước $f(0) = 1$ và $f(n) = 0$ với $\forall n < 0$, khi đó:

$$\begin{aligned} f(n) &= \sum_{k=1}^{\infty} (-1)^{k+1} \left(f\big(n - p(k)\big) + f\big(n - p(-k)\big) \right) \\ &= f(n-1) + f(n-2) - f(n-5) - f(n-7) + \cdots \end{aligned} \tag{15.2}$$

Chúng ta có thể sử dụng công thức truy hồi (15.2) để đếm số cách phân tích số trong thời gian $O\big(n\sqrt{n}\big)$ thay vì $\Theta(n^2)$ nếu dùng công thức (15.1) như các chương trình trước đó. Trong chương trình dưới đây, để tránh lỗi tràn số khi phép cộng có thể cho kết quả vượt quá phạm vi biểu diễn số nguyên 64 bit, chúng ta tính $f(n)$ theo cách:

$$\begin{aligned} SumA &:= \sum_{k=1}^{\infty} (-1)^{k+1} f\big(n - p(k)\big) \\ SumB &:= \sum_{k=1}^{\infty} (-1)^{k+1} f\big(n - p(-k)\big) \\ f(n) &:= SumA + SumB \end{aligned} \tag{15.3}$$

Đếm số cách phân tích số (Dùng số ngũ giác)

```
{$MODE OBJFPC}
{$INLINE ON}
program Counting6;
const
  max = 400;
var
  n, i, k, s: LongInt;
  f: array[0..max] of Int64;
  SumA, SumB: Int64;

function p(k: LongInt): LongInt; inline; //Số ngũ giác p(k)
begin
  Result := k * (3 * k - 1) shr 1;
end;

begin
  Readln(n);
  f[0] := 1;
  for i := 1 to n do
    begin
      SumA := 0;
      SumB := 0;
      s := 1;
      k := 1;
      while p(k) <= i do
        begin
          Inc(SumA, s * f[i - p(k)]);
          if p(-k) <= i then
            Inc(SumB, s * f[i - p(-k)]);
          Inc(k);
          s := -s; //Đảo dấu hạng tử
        end;
      f[i] := SumA + SumB;
    end;
  Writeln(f[n], ' Analyses');
end.
```

15.2. Phương pháp quy hoạch động

15.2.1. Bài toán quy hoạch động

Trong toán học và khoa học máy tính, *quy hoạch động* (*dynamic programming*) là một phương pháp hiệu quả giải những bài toán tối ưu có ba tính chất sau đây:

- Bài toán lớn có thể phân rã thành những bài toán con đồng dạng, những bài toán con đó có thể phân rã thành những bài toán nhỏ hơn nữa …(*recursive form*).
- Lời giải tối ưu của các bài toán con có thể sử dụng để tìm ra lời giải tối ưu của bài toán lớn (*optimal substructure*)
- Hai bài toán con trong quá trình phân rã có thể có chung một số bài toán con khác (*overlapping subproblems*).

Tính chất thứ nhất và thứ hai là điều kiện cần của một bài toán quy hoạch động. Tính chất thứ ba nêu lên đặc điểm của một bài toán mà cách giải bằng phương pháp quy hoạch động hiệu quả hơn hẳn so với phương pháp giải đệ quy thông thường.

Với những bài toán có hai tính chất đầu tiên, chúng ta thường nghĩ đến các thuật toán chia để trị và đệ quy: Để giải quyết một bài toán lớn, ta chia nó ra thành nhiều bài toán con đồng dạng và giải quyết độc lập các bài toán con đó.

Khác với thuật toán đệ quy, phương pháp quy hoạch động thêm vào cơ chế lưu trữ nghiệm hay một phần nghiệm của mỗi bài toán khi giải xong nhằm mục đích *sử dụng lại*, hạn chế những thao tác thừa trong quá trình tính toán.

Chúng ta xét một ví dụ đơn giản: Dãy Fibonacci là dãy vô hạn các số nguyên dương $f_1, f_2, \ldots$ được định nghĩa bằng công thức truy hồi sau:

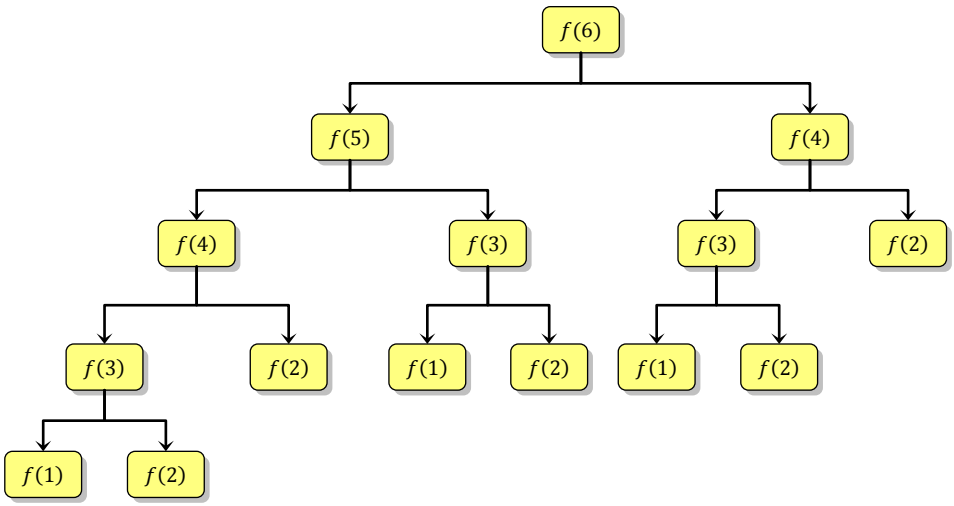
$$f_i = \begin{cases} 1, & \text{nếu } i \leq 2 \\ f_{i-1} + f_{i-2}, & \text{nếu } i > 2 \end{cases}$$

Hãy tính f_6.

Xét đoạn chương trình sau:

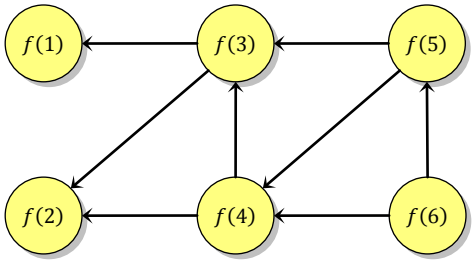
```
program TopDown;

function f(i: Integer): Integer;
begin
  if i ≤ 2 then Result := 1
  else Result := f(i - 1) + f(i - 2);
end;

begin
  Output ← f(6);
end.
```

Trong cách giải này, hàm đệ quy $f(i)$ được dùng để tính số Fibonacci thứ i. Chương trình chính gọi $f(6)$, nó sẽ gọi tiếp $f(5)$ và $f(4)$ để tính … Quá trình tính toán có thể vẽ như cây trong Hình 15-1**Error! Reference source not found.**.



Hình 15-1. Hàm đệ quy tính số Fibonacci

Ta thấy rằng để tính $f(6)$, máy phải tính một lần $f(5)$, hai lần $f(4)$, ba lần $f(3)$, năm lần $f(2)$ và ba lần $f(1)$. Sự thiếu hiệu quả có thể giải thích bởi tính chất: Hai bài toán trong quá trình phân rã có thể có chung một số bài toán con. Ví dụ nếu coi lời gọi hàm $f(i)$ tương ứng với bài toán tính số Fibonacci thứ i, thì bài toán $f(4)$ là bài toán con chung của cả $f(5)$ và $f(6)$. Điều đó chỉ ra rằng nếu tính $f(5)$ và $f(6)$ một cách độc lập, bài toán $f(4)$ sẽ phải giải hai lần.



Hình 15-2. Tính chất tập các bài toán con gối nhau (overlapping subproblems) khi tính số Fibonacci.

Để khắc phục những nhược điểm trong việc giải độc lập các bài toán con, mỗi khi giải xong một bài toán trong quá trình phân rã, chúng ta sẽ *lưu trữ lời giải* của nó nhằm mục đích *sử dụng lại*. Chẳng hạn với bài toán tính số Fibonacci thứ 6, ta có thể dùng một mảng $f_{1...6}$ để lưu trữ các giá trị số Fibonacci, khởi tạo sẵn giá trị cho f_1 và f_2 bằng 1, từ đó tính tiếp lần lượt f_3, f_4, f_5, f_6, đảm bảo mỗi giá trị Fibonacci chỉ phải tính một lần:

```
program BottomUp;
var
  f: array[1..6] of Integer;
  i: Integer;
begin
  f[1] := 1; f[2] := 1;
  for i := 3 to 6 do
    f[i] := f[i - 1] + f[i - 2];
  Output ← f[6];
end.
```

Kỹ thuật lưu trữ lời giải của các bài toán con nhằm mục đích sử dụng lại được gọi là *memoization**, nếu tới một bước nào đó mà lời giải một bài toán con không còn cần thiết nữa, chúng ta có thể bỏ lời giải đó đi khỏi không gian lưu trữ để tiết kiệm bộ nhớ. Ví dụ khi ta đang cần tính f_6 thì chỉ cần biết giá trị f_5 và f_4 là đủ nên việc lưu trữ các giá trị $f_{1...3}$ là vô nghĩa. Những kỹ thuật này chúng ta đã bàn đến trong mục 15.1.3 và 15.1.4. Cụ thể bài toán tính số Fibonacci, ta có thể cài đặt dùng hai biến luân phiên:

```
program BottomUp;
var
  a, b: Integer;
  i: Integer;
begin
  a := 1; b := 1;
  for i := 3 to 6 do
    begin
      b := a + b; //b := f[i]
      a := b - a; //a := f[i - 1]
    end;
  Output ← b;
end.
```

Cách giải này bắt đầu từ việc giải bài toán nhỏ nhất, lưu trữ nghiệm và từ đó giải quyết những bài toán lớn hơn…cách tiếp cận này gọi là cách tiếp cận từ dưới lên (bottom-up). Cách tiếp cận từ dưới lên không cần có chương trình con đệ quy, vì vậy bớt đi một số đáng kể lời gọi chương trình con và tiết kiệm được bộ nhớ chứa tham số và biến địa phương của chương trình con đệ quy. Tuy nhiên với cách tiếp cận từ dưới lên, chúng ta phải xác định rõ thứ tự giải các bài toán để khi bắt đầu giải một bài toán nào đó thì tất cả các bài toán con của nó phải được giải sẵn từ trước, điều này đôi khi tương đối phức tạp (xác định thứ tự tô-pô trên tập các bài toán phân rã). Trong trường hợp khó (hoặc không thể) xác định thứ tự hợp lý giải quyết các bài toán con, người ta sử dụng cách tiếp cận từ

---

* Từ này không có trong từ điển tiếng Anh, có thể coi như từ đồng nghĩa với *memorization*, gốc từ: *memo* (Bản ghi nhớ)

trên xuống (top-down) kết hợp với kỹ thuật lưu trữ nghiệm (memoization), kỹ thuật này chúng ta cũng đã bàn đến trong mục 15.1.5. Với bài toán tính số Fibonacci, ta có thể cài đặt bằng hàm đệ quy như sau:

```
program TopDownWithMemoization;
var
  f: array[1..6] of Integer;
  i: Integer;

function Getf(i: Integer): Integer;
begin
  if f[i] = 0 then //f[i] chưa được tính thì mới đi tính f[i]
    if i ≤ 2 then f[i] := 1
    else f[i] := Getf(i - 1) + Getf(i - 2);
  Result := f[i]; //Gán kết quả hàm bằng f[i]
end;

begin
  for i := 1 to 6 do f[i] := 0;
  Output ← Getf(6);
end.
```

Trước khi đi vào phân tích cách tiếp cận một bài toán quy hoạch động, ta làm quen với một số thuật ngữ sau đây:

- Bài toán giải theo phương pháp quy hoạch động gọi là *bài toán quy hoạch động*.
- Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn gọi là *công thức truy hồi* của quy hoạch động.
- Tập các bài toán nhỏ nhất có ngay lời giải để từ đó giải quyết các bài toán lớn hơn gọi là *cơ sở quy hoạch động*.
- Không gian lưu trữ lời giải các bài toán con để tìm cách phối hợp chúng gọi là *bảng phương án của quy hoạch động*.

15.2.2. Cách giải một bài toán quy hoạch động

Việc giải một bài toán quy hoạch động phải qua khá nhiều bước, từ những phân tích ở trên, ta có thể hình dung:

Quy hoạch động = Chia để trị + Cơ chế lưu trữ nghiệm để sử dụng lại

(Dynamic Programming = Divide and Conquer + Memoization)

Không có một "thuật toán" nào cho chúng ta biết một bài toán có thể giải bằng phương pháp quy hoạch động hay không? Khi gặp một bài toán cụ thể, chúng ta phải phân tích bài toán, sau đó kiểm chứng ba tính chất của một bài toán quy hoạch động trước khi tìm lời giải bằng phương pháp quy hoạch động.

Nếu như bài toán đặt ra đúng là một bài toán quy hoạch động thì việc đầu tiên phải làm là phân tích xem một bài toán lớn có thể phân rã thành những bài toán con đồng dạng như thế nào, sau đó xây dựng cách tìm nghiệm của bài toán lớn trong điều kiện chúng ta đã biết nghiệm của những bài toán con - *tìm công thức truy hồi*. Đây là công đoạn khó nhất vì chẳng có phương pháp tổng quát nào xây dựng công thức truy hồi cả, tất cả đều dựa vào kinh nghiệm và độ nhạy cảm của người lập trình khi đọc và phân tích bài toán.

Sau khi xây dựng công thức truy hồi, ta phải phân tích xem tại mỗi bước tính nghiệm của một bài toán, chúng ta *cần lưu trữ bao nhiêu bài toán con* và với mỗi bài toán con thì cần *lưu trữ toàn bộ nghiệm hay một phần nghiệm*. Từ đó xác định lượng bộ nhớ cần thiết để lưu trữ, nếu lượng bộ nhớ cần huy động vượt quá dung lượng cho phép, chúng ta cần tìm một công thức khác hoặc thậm chí, một cách giải khác không phải bằng quy hoạch động.

Một điều không kém phần quan trọng là phải chỉ ra được *bài toán nào cần phải giải trước bài toán nào*, hoặc chí ít là chỉ ra được khái niệm bài toán này nhỏ hơn bài toán kia nghĩa là thế nào. Nếu không, ta có thể rơi vào quá trình lòng vòng: Giải bài toán P_1 cần phải giải bài toán P_2, giải bài toán P_2 lại cần phải giải bài toán P_1 (công thức truy hồi trong trường hợp này là không thể giải được). Cũng nhờ chỉ rõ quan hệ "nhỏ hơn" giữa các bài toán mà ta có thể xác định được một tập các bài toán nhỏ nhất có thể giải trực tiếp, nghiệm của chúng sau đó được dùng làm cơ sở giải những bài toán lớn hơn.

Trong đa số các bài toán quy hoạch động, nếu bạn tìm được đúng công thức truy hồi và xác định đúng tập các bài toán cơ sở thì coi như phần lớn công việc đã xong. Việc tiếp theo là cài đặt chương trình giải công thức truy hồi:

- Nếu giải công thức truy hồi theo cách tiếp cận từ dưới lên, trước hết cần giải tất cả các bài toán cơ sở, lưu trữ nghiệm vào bảng phương án, sau đó dùng công thức truy hồi tính nghiệm của những bài toán lớn hơn và lưu trữ vào bảng phương án cho tới khi bài toán ban đầu được giải. Cách này yêu cầu phải xác định được chính xác thứ tự giải các bài toán (từ nhỏ đến lớn). Ưu điểm của nó là cho phép dễ dàng loại bỏ nghiệm những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.
- Nếu giải công thức truy hồi theo cách tiếp cận từ trên xuống, bạn cần phải có cơ chế đánh dấu bài toán nào đã được giải, bài toán nào chưa. Nếu bài toán chưa được giải, nó sẽ được phân rã thành các bài toán con và giải bằng đệ quy. Cách này không yêu cầu phải xác định thứ tự giải các bài toán nhưng cũng chính vì vậy, khó khăn sẽ gặp phải nếu muốn loại bỏ bớt nghiệm của những bài toán con không dùng đến nữa để tiết kiệm bộ nhớ lưu trữ cho bảng phương án.

Công đoạn cuối cùng là chỉ ra nghiệm của bài toán ban đầu. Nếu bảng phương án lưu trữ toàn bộ nghiệm của các bài toán thì chỉ việc đưa ra nghiệm của bài toán ban đầu, nếu bảng phương án chỉ lưu trữ một phần nghiệm, thì trong quá trình tính công thức truy hồi, ta để lại một "manh mối" nào đó (gọi là vết) để khi kết thúc quá trình giải, ta có thể truy vết tìm ra toàn bộ nghiệm.

Chúng ta sẽ khảo sát một số bài toán quy hoạch động kinh điển để hình dung được những kỹ thuật cơ bản trong việc phân tích bài toán quy hoạch động, tìm công thức truy hồi cũng như lập trình giải các bài toán quy hoạch động.

15.3. Một số bài toán quy hoạch động

15.3.1. Dãy con đơn điệu tăng dài nhất

Cho dãy số nguyên $A = (a_1, a_2, \ldots, a_n)$. Một dãy con của A là một cách chọn ra trong A một số phần tử giữ nguyên thứ tự. Như vậy A có 2^n dãy con.

Yêu cầu: Tìm dãy conđơn đi ệu tăng của A có độ dài lớn nhất. Tức là tìm một số k lớn nhất và dãy chỉ số $i_1 < i_2 < \ldots < i_k$ sao cho $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

Input

- Dòng 1 chứa số n ($n \le 10^6$)
- Dòng 2 chứa n số nguyên $a_1, a_2, \ldots, a_n$ ($\forall i: |a_i| \le 10^6$)

Output

Dãy con đơn điệu tăng dài nhất của dãy $A = (a_1, a_2, \ldots, a_n)$

Sample Input	Sample Output
12 1 2 3 8 9 4 5 6 2 3 9 10	Length = 8 a[1] = 1 a[2] = 2 a[3] = 3 a[6] = 4 a[7] = 5 a[8] = 6 a[11] = 9 a[12] = 10

❑ **Thuật toán**

Bổ sung vào A hai phần tử: $a_0 = -\infty$ và $a_{n+1} = +\infty$. Khi đó dãy con đơn đi ệu tăng dài nhất của dãy A chắc chắn sẽ bắt đầu từ a_0 và kết thúc ở a_{n+1}. Với mỗi giá trị i ($0 \le i \le n+1$), gọi $f[i]$ là độ dài dãy con đơn điệu tăng dài nhất bắt đầu tại a_i.

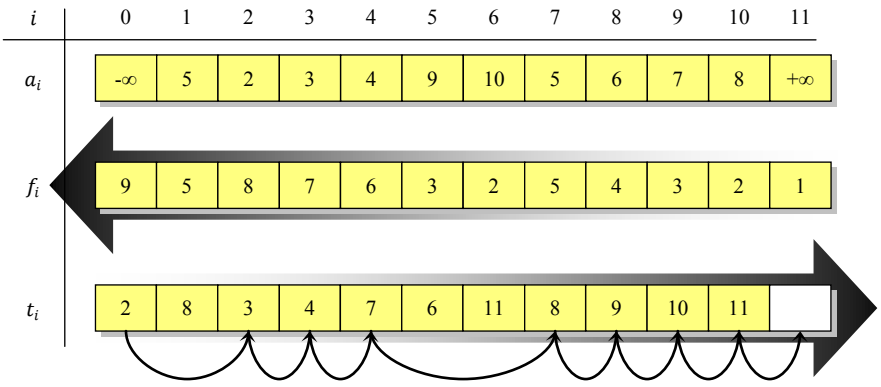
Ta sẽ xây dựng cách tính các giá trị $f[i]$ bằng công thức truy hồi. Theo định nghĩa, $f[i]$ là độ dài dãy con đơn đi ệu tăng dài nhất bắt đầu tại a_i, dãy con này đư ợc thành lập bằng

cách lấy a_i ghép vào đầu một trong số những dãy con đơn điệu tăng dài nhất bắt đầu tại vị trí a_j nào đó đứng sau a_i. Ta sẽ chọn dãy nào để ghép thêm a_i vào đầu?, tất nhiên ta chỉ được ghép a_i vào đầu những dãy con bắt đầu tại a_j nào đó lớn hơn a_i (để đảm bảo tính tăng) và đ nhiên ta s ẽ chọn dãy dài nhất để ghép a_i vào đầu (để đảm bảo tính dài nhất). Vậy $f[i]$ được tính như sau: Xét tất cả các chỉ số j từ $i+1$ đến $n+1$ mà $a_j > a_i$, chọn ra chỉ số $jmax$ có $f[jmax]$ lớn nhất và đặt $f[i] := f[jmax] + 1$.

Kỹ thuật lưu vết sẽ được thực hiện song song với quá trình tính toán. Mỗi khi tính $f[i] := f[jmax] + 1$, ta đặt $t[i] := jmax$ để ghi nhận rằng dãy con dài nhất bắt đầu tại a_i sẽ có phần tử kế tiếp là a_{jmax} :

$$\begin{aligned} f[i] &:= \max\{f[j]: i < j \le n+1 \text{ và } a_j > a_i\} + 1 \\ t[i] &:= \arg\max\{f[j]: i < j \le n+1 \text{ và } a_j > a_i\} \end{aligned} \qquad (15.4)$$

Từ công thức truy hồi tính các giá trị $f[.]$, ta nhận thấy rằng để tính $f[i]$ thì các giá trị $f[j]$ với $j > i$ phải được tính trước. Tức là thứ tự tính toán đúng phải là tính từ cuối mảng f về đầu mảng f. Với thứ tự tính toán như vậy, cơ sở quy hoạch động (bài toán nhỏ nhất) sẽ là phép tính $f[n+1]$, theo định nghĩa thì dãy con đơn điệu tăng dài nhất bắt đầu tại $a_{n+1} = +\infty$ chỉ có một phần tử là chính nó nên $f[n+1] = 1$.



Hình 15-3. Giải công thức truy hồi và truy vết

Sau khi tính xong các giá trị $f[.]$ và $t[.]$ việc chỉ ra dãy con đơn điệu tăng dài nhất được thực hiện bằng cách truy vết theo chiều ngược lại, từ đầu mảng t về cuối mảng t, cụ thể là ta bắt đầu từ phần tử $t[0]$:

$i_1 = t[0]$ chính là chỉ số phần tử đầu tiên được chọn (a_{i_1}).

$i_2 = t[i_1]$ là chỉ số phần tử thứ hai được chọn (a_{i_2}).

$i_3 = t[i_2]$ là chỉ số phần tử thứ ba được chọn (a_{i_3}).

…

❑ Cài đặt

Dưới đây ta sẽ cài đặt chương trình giải bài toán tìm dãy con đơn điệu tăng dài nhất theo những phân tích trên.

💻 Tìm dãy con đơn điệu tăng dài nhất

```
program LongestIncreasingSubsequence;
const
  max = 1000000;
  maxV = 1000000;
var
  a, f, t: array[0..max + 1] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do Read(a[i]);
end;

procedure Optimize; //Quy hoạch động
var
  i, j, jmax: LongInt;
begin
  a[0] := -maxV - 1; a[n + 1] := maxV + 1;
  f[n + 1] := 1; //Đặt cơ sở quy hoạch động
  for i := n downto 0 do //Giải công thức truy hồi
    begin //Tính f[i]
      //Tìm jmax > j thoả mãn a[jmax] > a[j] và f[jmax] lớn nhất
      jmax := n + 1;
      for j := i + 1 to n do
        if (a[j] > a[i]) and (f[j] > f[jmax]) then jmax := j;
      f[i] := f[jmax] + 1; //Lưu trữ nghiệm - memoization
      t[i] := jmax; //Lưu vết
    end;
end;

procedure PrintResult; //In kết quả
var
  i: LongInt;
begin
  WriteLn('Length = ', f[0] - 2); //Dãy kết quả phải bỏ đi hai phần tử a[0] và a[n + 1]
  i := t[0]; //Truy vết bắt đầu từ t[0]
  while i <> n + 1 do
    begin
      WriteLn('a[', i, '] = ', a[i]);
      i := t[i]; //Xét phần tử kế tiếp
    end;
end;

begin
  Enter;
  Optimize;
```

```
  PrintResult;
end.
```

❑ Cải tiến

Nhắc lại công thức truy hồi tính các $f[i]$ là:

$$f[i] = \begin{cases} 1, \text{ nếu } i = n+1 \\ \max\{f[j]: (j > i) \text{ và } (a_j > a_i)\} + 1, \text{ nếu } 0 \le i \le n \end{cases} \quad (15.5)$$

Để giải công thức truy hồi bằng cách cài đặt trên, ta sẽ cần thời gian $\Omega(n^2)$ để tính mảng f. Với kích thước dữ liệu lớn ($n \approx 10^6$) thì chương trình này chạy rất chậm. Ta cần một giải pháp tốt hơn cho bài toán này.

Quan sát sau đây sẽ cho ta một thuật toán với độ phức tạp tính toán là $O(n \lg m)$ với n là độ dài dãy A và m là độ dài dãy con đơn điệu tăng dài nhất tìm được.

Xét đoạn cuối dãy A bắt đầu từ chỉ số i: $(a_i, a_{i+1}, \dots, a_{n+1} = +\infty)$. Gọi:

- m là độ dài dãy con tăng dài nhất trong đoạn này.
- Với mỗi độ dài λ $(1 \le \lambda \le m)$, gọi s_λ là chỉ số của phần tử a_{s_λ} thoả mãn: Tồn tại dãy đơn điệu tăng độ dài λ bắt đầu từ a_{s_λ}. Nếu có nhiều phần tử trong dãy A cùng thoả mãn điều kiện này thì ta chọn a_{s_λ} là phần tử lớn nhất trong số những phần tử đó.
- Với mỗi chỉ số j $(i \le j \le n)$, gọi t_j là chỉ số phần tử liền sau phần tử a_j trong dãy con đơn điệu tăng dài nhất bắt đầu tại a_j.

Chúng ta sẽ tính m, các giá trị $s[\lambda]$ và các giá trị t_j bằng phương pháp quy hoạch động theo i.

Trường hợp cơ sở: Rõ ràng khi $i = n+1$, đoạn cuối dãy A bắt đầu từ chỉ số i chỉ có một phần tử là $a_{n+1} = +\infty$ nên trong trường hợp này, $m = 1$, $s_1 = n+1$.

Nhận xét rằng nếu đã biết được các giá trị m, $s_{(.)}$ và $t_{(.)}$ tương ứng với một chỉ số i nào đó thì các giá trị $s_{(.)}$ có tính chất:

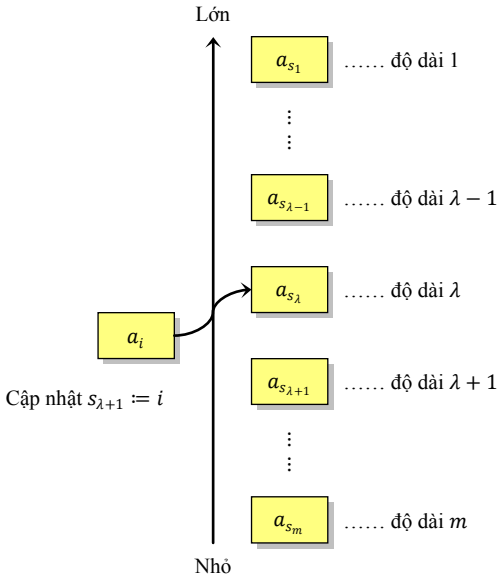
$$a_{s_m} < a_{s_{m-1}} < \dots < a_{s_1} \quad (15.6)$$

Thật vậy, vì dãy con tăng dài nhất bắt đầu tại a_{s_λ} có độ dài λ, dãy con này nếu xét từ phần tử thứ hai trở đi sẽ tạo thành một dãy con tăng độ dài $\lambda - 1$. Phần tử thứ hai này chắc chắn nhỏ hơn $a_{s_{\lambda-1}}$ (theo cách xây dựng $s_{\lambda-1}$). Vậy ta có $a_{s_\lambda} < a_{s_{\lambda-1}}$ $(\forall \lambda)$.

Để tính các giá trị m, $s_{(.)}$ và $t_{(.)}$ tương ứng với chỉ số $i = C$, trong trường hợp đã biết các giá trị m, $s_{(.)}$ và $t_{(.)}$ tương ứng với chỉ số $i = C+1$, ta có thể làm như sau:

- Dùng thuật toán tìm kiếm nhị phân trên dãy $a_{s_m} < a_{s_{m-1}} < \cdots < a_{s_1}$ để tìm giá trị λ lớn nhất thỏa mãn $a_i < a_{s_\lambda}$. Đến đây ta xác định được độ dài dãy con tăng dài nhất bắt đầu tại a_i là $\lambda + 1$, lưu lại vết $t_i = s_\lambda$. Có thể hiểu thao tác này là đem a_i nối vào đầu một dãy con tăng độ dài λ để được một dãy con tăng độ dài $\lambda + 1$.
- Cập nhật lại $m_{\text{mới}} := \max(m_{\text{cũ}}, \lambda + 1)$ và $s_{\lambda+1} := i$.

Bản chất của thuật toán là tại mỗi bước ta lưu thông tin về các dãy con tăng độ dài $1, 2, \ldots, m$. Nếu có nhiều dãy con tăng cùng một độ dài, ta chỉ quan tâm tới dãy có phần tử bắt đầu lớn nhất (để dễ nối thêm một phần tử khác vào đầu dãy). Hiện tại $a_{s_{\lambda+1}}$ đang là phần tử bắt đầu của một dãy con tăng độ dài $\lambda + 1$. Sau khi nối a_i vào đầu một dãy con tăng độ dài λ thì a_i cũng là phần tử bắt đầu của một dãy con tăng độ dài $\lambda + 1$. Xét về mặt giá trị, do tính chất của thuật toán tìm kiếm nhị phân ở trên, ta có $a_i \geq a_{s_{\lambda+1}}$, tức là dãy bắt đầu tại a_i "dễ" nối thêm phần tử khác vào đầu hơn nên ta cập nhật lại $s_{\lambda+1} := i$. (Hình 15-4)



Hình 15-4

Tìm dãy con đơn điệu tăng dài nhất (cải tiến)

```
{$MODE OBJFPC}
program LongestIncreasingSubsequence;
const
  max = 1000000;
  maxV = 1000000;
var
  a, t: array[0..max + 1] of LongInt;
```

```
  s: array[1..max + 2] of LongInt;
  n, m: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do Read(a[i]);
end;

procedure Init; //Khởi tạo
begin
  a[0] := -maxV - 1; a[n + 1] := maxV + 1;//Thêm vào 2 phần tử
  m := 1; //Độ dài dãy con tăng dài nhất bắt đầu từ a[n + 1]
  s[1] := n + 1; //Tính s[λ] với λ = 1
end;

//Thuật toán tìm kiếm nhị phân nhận vào một phần tử a[i] = v,
//và tìm trong dãy a[s[1]] > a[s[2]] > ... > a[s[m]], trả về chỉ số λ lớn nhất thoả mãn a[s[λ]] > v
function BinarySearch(v: LongInt): LongInt;
var
  Low, Middle, High: LongInt;
begin
  Low := 2; High := m;
  while Low <= High do
    begin
      Middle := (Low + High) div 2;
      if a[s[Middle]] > v then Low := Middle + 1
      else High := Middle - 1;
    end;
  Result := High;
end;

procedure Optimize; //Quy hoạch động
var
  i, lambda: LongInt;
begin
  for i := n downto 0 do //Giải công thức truy hồi
    begin
      lambda := BinarySearch(a[i]);
      t[i] := s[lambda]; //Lưu vết
      if lambda + 1 > m then   //Cập nhật m là độ dài dãy con tăng dài nhất cho tới thời điểm này
        m := lambda + 1;
      s[lambda + 1] := i;
    end;
end;

procedure PrintResult; //In kết quả
var
  i: LongInt;
begin
  WriteLn('Length = ', m - 2); //Dãy kết quả phải bỏ đi hai phần tử a[0] và a[n + 1]
  i := t[0]; //Truy vết bắt đầu từ t[0]
  while i <> n + 1 do
    begin
      WriteLn('a[', i, '] = ', a[i]);
```

```
      i := t[i]; //Xét phần tử kế tiếp
    end;
end;

begin
  Enter;
  Init;
  Optimize;
  PrintResult;
end.
```

15.3.2. Bài toán xếp ba lô

Cho n đồ vật, đồ vật thứ i có trọng lượng là w_i và giá trị là v_i $(w_i, v_i \in \mathbb{Z}^+)$. Cho một balô có giới hạn trọng lượng là m, hãy chọn ra một số đồ vật cho vào balô sao cho tổng trọng lượng của chúng không vượt quá m và tổng giá trị của chúng là lớn nhất có thể.

Input

- Dòng 1 chứa hai số nguyên dương n, m $(n \leq 1000; m \leq 10000)$
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên dương w_i, v_i cách nhau ít nhất một dấu cách $(w_i, v_i \leq 10000)$

Output

Phương án chọn các đồ vật có tổng trọng lượng $\leq m$ và tổng giá trị lớn nhất có thể.

Sample Input	Sample Output
5 11 3 30 4 40 5 40 9 100 4 40	Selected objects: 1. Object 5: Weight = 4; Value = 40 2. Object 2: Weight = 4; Value = 40 3. Object 1: Weight = 3; Value = 30 Total weight: 11 Total value : 110

❑ Thuật toán

Bài toán Knapsack tuy chưa có thuật toán giải hiệu quả trong trường hợp tổng quát nhưng với ràng buộc dữ liệu cụ thể như ở bài này thì có thuật toán quy hoạch động khá hiệu quả để giải, dưới đây chúng ta sẽ trình bày cách làm đó.

Gọi $f[i,j]$ là giá trị lớn nhất có thể có bằng cách chọn trong các đồ vật $\{1,2,\ldots,i\}$ với giới hạn trọng lượng j. Mục đích của chúng ta là đi tìm $f[n,m]$: Giá trị lớn nhất có thể có bằng cách chọn trong n đồ vật đã cho với giới hạn trọng lượng m.

Với giới hạn trọng lượng j, việc chọn trong số các đồ vật $\{1,2,\ldots,i\}$ để có giá trị lớn nhất sẽ là một trong hai khả năng:

- Nếu không chọn đồ vật thứ i, thì $f[i,j]$ là giá trị lớn nhất có thể có bằng cách chọn trong số các đồ vật $\{1,2,\ldots,i-1\}$ với giới hạn trọng lượng bằng j. Tức là $f[i,j] = f[i-1,j]$
- Nếu có chọn đồ vật thứ i (tất nhiên chỉ xét tới trường hợp này khi mà $w_i \leq j$), thì $f[i,j]$ bằng giá trị đồ vật thứ i $(= v_i)$ cộng với giá trị lớn nhất có thể có được bằng cách chọn trong số các đồ vật $\{1,2,\ldots,i-1\}$ với giới hạn trọng lượng $j - w_i$ $(= f[i-1, j-w_i])$. Tức là về mặt giá trị thu được, $f[i,j] = f[i-1,j-w_i] + v_i$.

Theo cách xây dựng $f[i,j]$, ta suy ra công thức truy hồi:

$$f[i,j] = \begin{cases} f[i-1,j], \text{nếu } j < w_i \\ \max\{f[i-1,j], f[i-1,j-w_i] + v_i\}, \text{nếu } j \geq w_i \end{cases} \quad (15.7)$$

Từ công thức truy hồi, ta thấy rằng để tính các phần tử trên hàng i của bảng f thì ta cần phải biết các phần tử trên hàng liền trước đó (hàng $i-1$). Vậy nếu ta biết được hàng 0 của bảng f thì có thể tính ra mọi phần tử trong bảng. Từ đó suy ra cơ sở quy hoạch động: $f[0,j]$ theo định nghĩa là giá trị lớn nhất có thể bằng cách chọn trong số 0 đồ vật, nên hiển nhiên $f[0,j] = 0$.

Tính xong bảng phương án thì ta quan tâmđ ến $f[n,m]$, đó chính là giá trị lớn nhất thu được khi chọn trong cả n đồ vật với giới hạn trọng lượng m. Việc còn lại là chỉ ra phương án chọn các đồ vật.

- Nếu $f[n,m] = f[n-1,m]$ thì tức là phương án tối ưu không chọn đồ vật n, ta truy tiếp $f[n-1,m]$.
- Còn nếu $f[n,m] \neq f[n-1,m]$ thì ta thông báo rằng phép chọn tối ưu có chọn đồ vật n và truy tiếp $f[n-1,m-w_n]$.

Cứ tiếp tục cho tới khi truy lên tới hàng 0 của bảng phương án.

❑ Cài đặt

Rõ ràng ta có thể giải công thức truy hồi theo kiểu từ dưới lên: Khởi tạo hàng 0 của bảng phương án f toàn số 0, sau đó dùng công thức truy hồi tính lần lượt từ hàng 1 tới hàng n. Tuy nhiên trong chương tình này chúng ta s ẽ giải công thức truy hồi theo kiểu từ trên xuống (sử dụng hàm đệ quy kết hợp với kỹ thuật lưu trữ nghiệm) vì cách giải này không phải đi tính toàn bộ bảng phương án.

💻 Bài toán xếp ba lô

```
{$MODE OBJFPC}
program Knapsack;
const
  maxN = 1000;
```

```
  maxM = 10000;
var
  w, v: array[1..maxN] of LongInt;
  f: array[0..maxN, 0..maxM] of LongInt;
  n, m: LongInt;
  SumW, SumV: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n, m);
  for i := 1 to n do ReadLn(w[i], v[i]);
end;

function Getf(i, j: LongInt): LongInt; //Tính f[i, j]
begin
  if f[i, j] = -1 then //Nếu f[i, j] chưa được tính thì mới đi tính
    if i = 0 then f[i, j] := 0 //Cơ sở quy hoạch động
    else
      if (j < w[i]) or
         (Getf(i - 1, j ) > Getf(i - 1, j - w[i]) + v[i]) then
        f[i, j] := Getf(i - 1, j) //Không chọn đồ vật i thì tốt hơn
      else
        f[i, j] := Getf(i - 1, j - w[i]) + v[i]; //Chọn đồ vật i thì tốt hơn
  Result := f[i, j]; //Trả về f[i, j] trong kết quả hàm
end;

procedure PrintResult; //In kết quả
var
  Count: LongInt;
begin
  WriteLn('Selected objects:');
  Count := 0;
  SumW := 0;
  while n <> 0 do //Truy vết từ f[n, m]
    begin
      if Getf(n, m) <> Getf(n - 1, m) then //Phương án tối ưu có chọn đồ vật n
        begin
          Inc(Count);
          Write(Count, '. Object ', n, ': ');
          WriteLn('Weight = ', w[n], '; Value = ', v[n]);
          Inc(SumW, w[n]);
          Dec(m, w[n]); //Chọn đồ vật n rồi thì giới hạn trọng lượng giảm đi w[n]
        end;
      Dec(n);
    end;
  WriteLn('Total weight: ', SumW);
  WriteLn('Total value : ', SumV);
end;

begin
  Enter;
  FillChar(f, SizeOf(f), $FF); //Khởi tạo các phần tử của f là -1 (chưa được tính)
  SumV := Getf(n, m);
  PrintResult;
end.
```

Nếu thêm vào một đoạn chương trình để in ra bảng f sau khi giải công thức truy hồi thì với dữ liệu như trong ví dụ, bảng f sẽ là:

f	0	1	2	3	4	5	6	7	8	9	10	11
0	0	-1	0	0	0	-1	0	0	0	-1	-1	0
1	-1	-1	0	30	-1	-1	30	30	-1	-1	-1	30
2	-1	-1	0	-1	-1	-1	40	70	-1	-1	-1	70
3	-1	-1	0	-1	-1	-1	-1	70	-1	-1	-1	80
4	-1	-1	-1	-1	-1	-1	-1	70	-1	-1	-1	100
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	110

Ta thấy còn rất nhiều giá trị -1 trong bảng, nghĩa là có rất nhiều phần tử trong bảng f không tham gia vào quá trình tính $f[5,11]$. Điều đó chỉ ra rằng cách giải công thức truy hồi từ trên xuống trong trường hợp này sẽ hiệu quả hơn cách giải từ dưới lên vì nó không cần tính toàn bộ bảng phương án của quy hoạch động.

15.3.3. Biến đổi xâu

Với xâu ký tự $X = x_1x_2\dots x_m$, xét 3 phép biến đổi:

- $Insert(i, c)$: Chèn ký tự c vào sau vị trí i của xâu X.
- $Delete(i)$: Xoá ký tự tại vị trí i của xâu X.
- $Replace(i, c)$: Thay ký tự tại vị trí i của xâu X bởi ký tự c.

Yêu cầu: Cho hai xâu ký tự $X = x_1x_2\dots x_m$ và $Y = y_1y_2\dots y_n$ $(m, n \leq 1000)$ chỉ gồm các chữ cái in hoa, hãy tìm một số ít nhất các phép biến đổi kể trên để biến xâu X thành xâu Y.

Input

- Dòng 1 chứa xâu X
- Dòng 2 chứa xâu Y

Output

Cách biến đổi X thành Y

Sample Input	Sample Output
ACGGTAG CCTAAG	Number of alignments: 4 X = ACGGTAG Insert(5, A) X = ACGGTAAG Delete(4) X = ACGTAAG Delete(3) X = ACTAAG Replace(1, C) X = CCTAAG

Bài toán này (*String alignment*[*]) có ứng dụng khá nhiều trong Tin-sinh học để phân tích và so sánh các yếu tố di truyền (DNA, RNA, proteins, chromosomes, genes, …). Dữ liệu di truyền thường có dạng chuỗi, chẳng hạn một phân tử DNA có thể biểu diễn dưới dạng chuỗi các ký tự {A, C, G, T} hay một phân tử RNA có thể biểu diễn bằng chuỗi các ký tự {A, C, G, U}. Để trả lời câu hỏi hai mẫu dữ liệu di truyền (mã hoá dưới dạng xâu ký tự) giống nhau đến mức nào, người ta có thể dùng số lượng các phép biến đổi kể trên để đo mức độ khác biệt giữa hai mẫu. Tuỳ vào từng yêu cầu thực tế, người ta còn có thể thêm vào nhiều phép biến đổi nữa, hoặc gán trọng số cho từng vị trí biến đổi (vì các vị trí trong xâu có thể có tầm quan trọng khác nhau).

Có thể kể ra vài ứng dụng cụ thể của bài toán gióng hàng. Một ứng dụng là xác định chủng loại của một virus chưa biết. Chẳng hạn khi phát hiện một virus mới gây bệnh mà con người chưa kịp hình thành khả năng miễn dịch, người ta sẽ đối sánh mẫu RNA của virus đó với tất cả các mẫu RNA của các virus đã có trong cơ sở dữ liệu, từ đó tìm ra các loại virus tương tự, xác định chủng loại của virus và điều chế vaccine phòng bệnh. Một ứng dụng khác là phân tích các chuỗi DNA để xác định mối liên quan giữa các loài, từ đó xây dựng cây tiến hoá từ một loài tổ tiên nào đó. Ngoài ứng dụng trong sinh học phân tử, bài toán gióng hàng còn có nhiều ứng dụng khác trong khoa học máy tính, lý thuyết mã, nhận dạng tiếng nói, v.v…

❑ Thuật toán

Ta trở lại bài toán với hai xâu ký tự $X = x_1x_2\ldots x_m$ và $Y = y_1y_2\ldots y_n$. Để biến đổi xâu X thành xâu Y, chúng ta sẽ phân tích một bài toán con: Tìm một số ít nhất các phép biến đổi để biến i ký tự đầu tiên của xâu X thành j ký tự đầu tiên của xâu Y. Cụ thể, ta gọi $f[i,j]$ là số phép biến đổi tối thiểu để biến xâu $x_1x_2\ldots x_i$ thành xâu $y_1y_2\ldots y_j$. Khi đó:

- Nếu $x_i = y_j$ thì vấn đề trở thành biến xâu $x_1x_2\ldots x_{i-1}$ thành xâu $y_1y_2\ldots y_{j-1}$. Tức là nếu xét về số lượng các phép biến đổi, trong trường hợp này $f[i,j] = f[i-1,j-1]$.
- Nếu $x_i \neq y_j$. Ta có ba lựa chọn:
  - Hoặc chèn ký tự y_j vào cuối xâu $x_1x_2\ldots x_i$ rồi biến xâu $x_1x_2\ldots x_i$ thành xâu $y_1y_2\ldots y_{j-1}$. Tức là nếu theo sự lựa chọn này, $f[i,j] = f[i,j-1] + 1$
  - Hoặc xóa ký tự cuối của xâu $x_1x_2\ldots x_i$ rồi biến xâu $x_1x_2\ldots x_{i-1}$ thành xâu $y_1y_2\ldots y_j$. Tức là nếu theo sự lựa chọn này, $f[i,j] = f[i-1,j] + 1$

[*] Tạm dịch: Gióng hàng

- Hoặc thay ký tự cuối của xâu $x_1x_2 \dots x_i$ bởi ký tự y_j rồi biến xâu $x_1x_2 \dots x_{i-1}$ thành xâu $y_1y_2 \dots y_{j-1}$. Tức là nếu theo sự lựa chọn này, $f[i,j] = f[i-1, j-1] + 1$

Vì chúng ta cần $f[i,j] \rightarrow \min$ nên suy ra công thức truy hồi:

$$f[i,j] = \begin{cases} f[i-1,j-1], \text{nếu } x_i = y_j \\ 1 + \min \begin{cases} f[\mathrm{i,j-1}] \\ f[\mathrm{i-1,j}] \\ f[i-1,j-1] \end{cases}, \text{nếu } x_i \neq y_j \end{cases} \quad (15.8)$$

Từ công thức truy hồi, ta thấy rằng nếu biểu diễn f dưới dạng ma trận thì phần tử ở ô (i,j) $(f[i,j])$ được tính dựa vào phần tử ở ô nằm bên trái: $(i,j-1)$, ô nằm phía trên: $(i-1,j)$ và ô nằm ở góc trái trên $(i-1,j-1)$. Từ đó suy ra tập các bài toán cơ sở:

- f[0, j] là số phép biến đổi biễn xâu rỗng thành j ký tự đầu của xâu Y, nó cần tối thiểu j phép chèn: $f[0,j] = j$
- f[i, 0] là số phép biến đổi biến xâu gồm i ký tự đầu của X thành xâu rỗng, nó cần tối thiểu i phép xoá: $f[i,0] = i$

Vậy đầu tiên bảng phương án $f[0 \dots m, 0 \dots n]$ được khởi tạo hàng 0 và cột 0 là cơ sở quy hoạch động. Từ đó dùng công thức truy hồi tính ra tất cả các phần tử bảng f, sau khi tính xong thì $f[m,n]$ cho ta biết số phép biến đổi tối thiểu để biến xâu $X = x_1x_2 \dots x_m$ thành $Y = y_1y_2 \dots y_n$.

Việc cuối cùng là chỉ ra cụ thể cách biến đổi tối ưu, ta bắt đầu truy vết từ $f[m,n]$, việc truy vết chính là dò ngược lại xem $f[m,n]$ được tìm như thế nào.

- Nếu $x_m = y_n$ thì ta tìm cách biến $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_{n-1}$, tức là truy tiếp $f[m-1, n-1]$
- Nếu $x_m \neq y_n$ thì xét ba trường hợp
  - Nếu $f[m,n] = f[m,n-1] + 1$ thì ta thực hiện phép $Insert(m, y_n)$ để chèn y_n vào cuối xâu X rồi tìm cách biến đổi $x_1x_2 \dots x_m$ thành $y_1y_2 \dots y_{n-1}$ (truy tiếp $f[m, n-1]$)
  - Nếu $f[m,n] = f[m-1,n] + 1$ thì ta thực hiện phép $Delete(m)$ để xóa ký tự cuối xâu X rồi tìm cách biến đổi $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_n$ (truy tiếp $f[m-1, n]$)
  - Nếu $f[m,n] = f[m-1,n-1] + 1$ thì ta thực hiện phép $Replace(m, y_n)$ để thay ký tự cuối xâu X bởi y_n rồi tìm cách biến đổi $x_1x_2 \dots x_{m-1}$ thành $y_1y_2 \dots y_{n-1}$ (truy tiếp $f[m-1, n-1]$)

Ta đưa về việc truy vết với m, n nhỏ hơn và cứ lặp lại cho tới khi quy về bài toán cơ sở: $m = 0$ hoặc $n = 0$.

Ví dụ với $X = 'ACGGTAG'$ và $Y = 'CCTAAG'$, việc truy vết trên bảng phương án f được chỉ ra trong Hình 15-5.

			C	C	T	A	A	G
	f	0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
A	1	1	1	2	3	3	4	5
C	2	2	1	1	2	3	4	5
G	3	3	2	2	2	3	4	4
G	4	4	3	3	3	3	4	4
T	5	5	4	4	3	4	4	5
A	6	6	5	5	4	3	4	5
G	7	7	6	6	5	4	4	4

Hình 15-5. Truy vết tìm cách biến đổi xâu

❑ Cài đặt

Biến đổi xâu

```
{$MODE OBJFPC}
{$INLINE ON}
program StringAlignment;
const
  max = 1000;
type
  TStrOperator = (soDoNothing, soInsert, soDelete, soReplace);
var
  x, y: AnsiString;
  f: array[0..max, 0..max] of LongInt;
  m, n: LongInt;

procedure Enter; //Nhập dữ liệu
begin
  ReadLn(x);
  ReadLn(y);
  m := Length(x); n := Length(y);
end;

function Min3(x, y, z: LongInt): LongInt; inline; //Tính min của 3 số
begin
  if x < y then Result := x else Result := y;
  if z < Result then Result := z;
end;

procedure Optimize; //Giải công thức truy hồi
var
  i, j: LongInt;
begin
  //Lưu cơ sở quy hoạch động
  for j := 0 to n do f[0, j] := j;
  for i := 1 to m do f[i, 0] := i;
```

```
  //Dùng công thức truy hồi tính toàn bộ bảng phương án
  for i := 1 to m do
    for j := 1 to n do
      if x[i] = y[j] then
        f[i, j] := f[i - 1, j - 1]
      else
        f[i, j] := Min3(f[i, j - 1], f[i - 1, j - 1], f[i - 1, j]) + 1;
end;

//Thực hiện thao tác op với tham số vị trí p và ký tự c trên xâu X
procedure Perform(op: TStrOperator; p: LongInt = 0; c: Char = #0);
begin
  case op of
    soInsert: //Phép chèn
      begin
        WriteLn('Insert(', p, ', ', c, ')');
        Insert(c, x, p + 1);
      end;
    soDelete: //Phép xoá
      begin
        WriteLn('Delete(', p, ')');
        Delete(x, p, 1);
      end;
    soReplace: //Phép thay
      begin
        WriteLn('Replace(', p, ', ', c, ')');
        x[p] := c;
      end;
  end;
  WriteLn('X = ', x); //In ra xâu X sau khi biến đổi
end;

procedure PrintResult; //In kết quả
begin
  WriteLn('Number of alignments: ', f[m, n]);
  Perform(soDoNothing); //In ra xâu X ban đầu
  while (m <> 0) and (n <> 0) do //Làm tới khi m = n = 0
    if x[m] = y[n] then //Hai ký tự cuối giống nhau, không làm gì cả, truy tiếp f[m – 1, n – 1]
      begin
        Dec(m); Dec(n);
      end
    else //Tại vị trí m của xâu X cần có một phép biến đổi
      if f[m, n] = f[m, n - 1] + 1 then //Nếu là phép chèn
        begin
          Perform(soInsert, m, y[n]);
          Dec(n); //Truy sang phải: f[m, n – 1]
        end
      else
        if f[m, n] = f[m - 1, n] + 1 then //Nếu là phép xoá
          begin
            Perform(soDelete, m);
            Dec(m); //Truy lên trên: f[m – 1, n]
          end
        else //Không phải chèn, không phải xoá thì phải là phép thay thế
          begin
            Perform(soReplace, m, y[n]);
            Dec(m); Dec(n); //Truy chéo lên trên: f[m – 1, n – 1]
```

```
          end;
  while m > 0 do
    begin
      Perform(soDelete, m);
      Dec(m);
    end;
  while n > 0 do
    begin
      Perform(soInsert, 0, y[n]);
      Dec(n);
    end;
end;

begin
  Enter;
  Optimize;
  PrintResult;
end.
```

15.3.4. Phân hoạch tam giác

Trên mặt phẳng với hệ tọa độ trực chuẩn Oxy, cho một đa giác lồi $P_1P_2\dots P_n$, trong đó toạ độ đỉnh P_i là (x_i, y_i). Một bộ $n-3$ đường chéo đôi một không cắt nhau sẽ chia đa giác đã cho thành $n-2$ tam giác, ta gọi bộ $n-3$ đường chéo đó là một phép tam giác phân của đa giác lồi ban đầu.

Bài toán đặt ra là hãy tìm một phép tam giác phân có trọng số (tổng độ dài các đường chéo) nhỏ nhất.

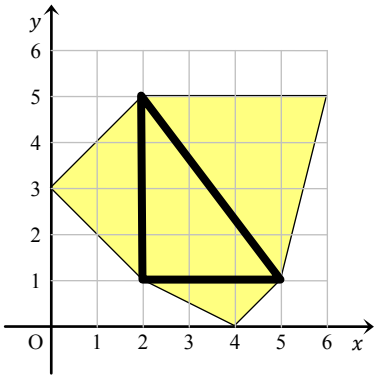
Input

- Dòng 1 chứa số n là số đỉnh của đa giác ($3 \le n \le 200$)
- n dòng tiếp theo, dòng thứ i chứa toạ độ của đỉnh P_i: gồm hai số thực x_i, y_i

Output

Phép tam giác phân nhỏ nhất.

Sample Input	Sample Output
6 4.0 0.0 5.0 1.0 6.0 5.0 2.0 5.0 0.0 3.0 2.0 1.0	Minimum triangulation: 12.00 P[2] - P[6] = 3.00 P[2] - P[4] = 5.00 P[4] - P[6] = 4.00



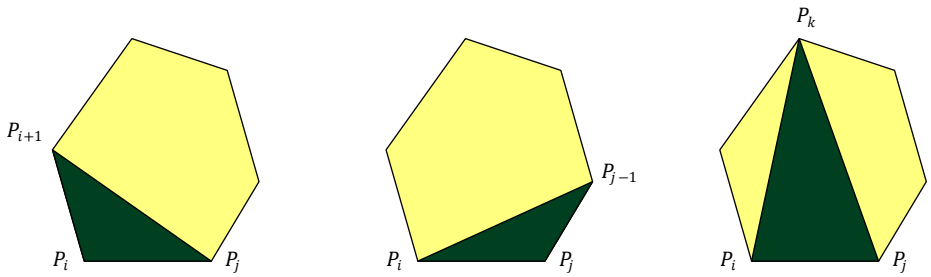
❑ Thuật toán

Ký hiệu $d(i,j)$ là khoảng cách giữa hai điểm P_i và P_j:

$$d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{15.9}$$

Gọi $f[i,j]$ là trọng số phép tam giác phân nhỏ nhất để chia đa giác $P_iP_{i+1}\dots P_j$, khi đó $f[1,n]$ chính là trọng số phép tam giác phân nhỏ nhất để chia đa giác ban đầu.

Bằng quan sát hình học của một phép tam giác phân trên đa giác $P_iP_{i+1}\dots P_j$, ta nhận thấy rằng trong bất kỳ phép tam giác phân nào, luôn tồn tại một và chỉ một tam giác chứa cạnh P_iP_j. Tam giác chứa cạnh P_iP_j sẽ có một đỉnh là P_i, một đỉnh là P_j và một đỉnh P_k nào đó $(i < k < j)$. Xét một phép tam giác phân nhỏ nhất để chia đa giác $P_iP_{i+1}\dots P_j$, ta quan tâm tới tam giác chứa cạnh P_iP_j: $\Delta\, P_iP_jP_k$, có ba khả năng xảy ra (Hình 15-6):



Hình 15-6. Ba trường hợp của tam giác chứa cạnh P_iP_j

Khả năng thứ nhất, đỉnh $P_k \equiv P_{i+1}$, khi đó đường chéo $P_{i+1}P_j$ sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $P_{i+1}P_{i+2}\dots P_j$. Tức là trong trường hợp này:

$$f[i,j] = f^{(k=i+1)}[i,j] = d(i+1,j) + f[i+1,j] \tag{15.10}$$

Khả năng thứ hai, đỉnh $P_k \equiv P_{j-1}$, khi đó đường chéo P_iP_{j-1} sẽ thuộc phép tam giác phân đang xét, những đường chéo còn lại sẽ phải tạo thành một phép tam giác phân nhỏ nhất để chia đa giác lồi $P_iP_{i+1}\dots P_{j-1}$. Tức là trong trường hợp này:

$$f[i,j] = f^{(k=j-1)}[i,j] = d(i,j-1) + f[i,j-1] \tag{15.11}$$

Khả năng thứ ba, đỉnh P_k không trùng P_{i+1} và cũng không trùng P_{j-1}, khi đó cả hai đường chéo P_iP_k và P_jP_k đều thuộc phép tam giác phân, những đường chéo còn lại sẽ tạo thành hai phép tam giác phân nhỏ nhất tương ứng với đa giác $P_iP_{i+1}\dots P_k$ và đa giác $P_kP_{k+1}\dots P_j$. Tức là trong trường hợp này:

$$f[i,j] = f^{(k)}[i,j] = d(i,k) + d(k,j) + f[i,k] + f[k,j] \tag{15.12}$$

Vì $f[i,j]$ là trọng số của phép tam giác phân nhỏ nhất trên đa giác $P_iP_{i+1}\dots P_j$ nên nó sẽ phải là trọng số nhỏ nhất của phép tam giác phân trên tất cả các khả năng chọn đỉnh P_k tức là:

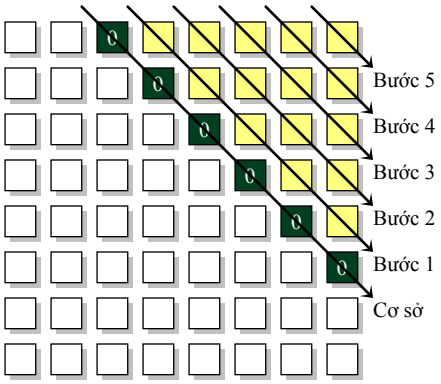
$$f[i,j] = \min_{k:i<k<j}\{f^{(k)}[i,j]\} \tag{15.13}$$

Trong đó

$$f^{(k)}[i,j] = \begin{cases} d(i+1,j) + f[i+1,j]; \text{ nếu } k = i+1 \\ d(i+1,j) + f[i+1,j]; \text{ nếu } k = j-1 \\ d(i,k) + d(k,j) + f[i,k] + f[k,j]; \text{ nếu } i+1 < k < j-1 \end{cases} \tag{15.14}$$

Ta xây dựng xong công thức truy hồi tính $f[i,j]$. Từ công thức truy hồi, ta nhận thấy rằng $f[i,j]$ được tính qua các giá trị $f[i,k]$ và $f[k,j]$ với $i < k < j$, điều đó chỉ ra rằng hiệu số $j-i$ chính là thứ tự lớn-nhỏ của các bài toán: Các giá trị $f[i,j]$ với $j-i$ lớn sẽ được tính sau các giá trị $f[i',j']$ có hiệu số $j'-i'$ nhỏ hơn. Thứ tự lớn-nhỏ này cho phép tìm ra cơ sở quy hoạch động: Các giá trị $f[i,i+2]$ là trọng số của phép tam giác phân nhỏ nhất để chia đa giác lồi gồm 3 đỉnh (tức là tam giác), trong trường hợp này chúng ta không cần chia gì cả và trọng số phép tam giác phân là $f[i,i+2] = 0$.

Việc giải công thức truy hồi sẽ được thực hiện theo trình tự: Đầu tiên bảng phương án f được điền cơ sở quy hoạch động: Các phần tử trên đường chéo $f[i,i+2]$ được đặt bằng 0, dùng công thức truy hồi tính tiếp các phần tử trên đường chéo $f[i,i+3]$, rồi tính tiếp đường chéo $f[i,i+4]$… Cứ như vậy cho tới khi tính được phần tử $f[1,n]$.



Hình 15-7. Quy trình tính bảng phương án

Song song với quá trình gi ải công thức truy hồi, tại mỗi bước tính $f[i,j] \coloneqq \min_{k:i<k<j}\{f^{(k)}[i,j]\}$, ta lưu lại $Trace[i,j]$ là chỉ số k mà tại đó $f[i,j]$ đạt cực tiểu, tức là:

$$Trace[i,j] \coloneqq \arg\min_{k:i<k<j}\{f^{(k)}[i,j]\} \tag{15.15}$$

Sau khi quá trình tính toán kết thúc, để in ra phép tam giác phân nhỏ nhất đối với đa giác $P_iP_{i+1}\dots P_j$, ta sẽ sẽ dựa vào phần tử $k = Trace[i,j]$ để đưa vấn đề về việc in ra phép tam giác phân đối với hai đa giác $P_iP_{i+1}\dots P_k$ và $P_kP_{k+1}\dots P_j$. Điều này có thể được thực hiện đơn giản bằng một thủ tục đệ quy.

❑ Cài đặt

💻 Phân hoạch tam giác

```
{$MODE OBJFPC}
{$INLINE ON}
program Triangulation;
const
  max = 200;
var
  x, y: array[1..max] of Real;
  f: array[1..max, 1..max] of Real;
  Trace: array[1..max, 1..max] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do ReadLn(x[i], y[i]);
end;

function d(i, j: LongInt): Real; //Hàm đo khoảng cách P[i] – P[j]
begin
```

```
  Result := Sqrt(Sqr(x[i] - x[j]) + Sqr(y[i] - y[j]));
end;

//Hàm cực tiểu hoá Target := Min(Target, Value)
function Minimize(var Target: Real; Value: Real): Boolean; inline;
begin
  Result := Value < Target; //Trả về True nếu Target được cập nhật
  if Result then Target := Value;
end;

procedure Optimize; //Giải công thức truy hồi
var
  m, i, j, k: LongInt;
begin
for i := 1 to n - 2 do
    f[i, i + 2] := 0;
  for m := 3 to n - 1 do
    for i := 1 to n - m do
      begin //Tính f[i, i + m]
        j := i + m;
        //Trường hợp k = i + 1
        f[i, j] := d(i + 1, j) + f[i + 1, j];
        Trace[i, j] := i + 1;
        //Trường hợp i + 1 < k < j - 1
        for k := i + 2 to j - 2 do
          if Minimize(f[i, j],
             d(i, k) + d(k, j) + f[i, k] + f[k, j]) then
            Trace[i, j] := k;
        //Trường hợp k = j - 1
        if Minimize(f[i, j], d(i, j - 1) + f[i, j - 1]) then
          Trace[i, j] := j - 1;
      end;
end;

//Với i + 1 < j, thủ tục này in ra đường chéo P[i] – P[j]
procedure WriteDiagonal(i, j: LongInt); inline;
begin
  if i + 1 < j then
    WriteLn('P[', i, '] - P[', j, '] = ', d(i, j):0:2);
end;

//Truy vết bằng đệ quy, in ra cách tam giác phân đa giác P[i..j]
procedure TraceRecursively(i, j: LongInt);
var
  k: LongInt;
begin
  if j <= i + 2 then Exit; //Đa giác có ≤ 3 đỉnh thì không cần phân hoạch
  k := Trace[i, j]; //Lấy vết
  WriteDiagonal(i, k);
  WriteDiagonal(k, j);
  TraceRecursively(i, k); //In ra cách phân hoạch đa giác P[i..k]
  TraceRecursively(k, j); //In ra cách phân hoạch đa giác P[k..j]
end;

procedure PrintResult; //In kết quả
begin
  WriteLn('Minimum triangulation: ', f[1, n]:0:2);
```

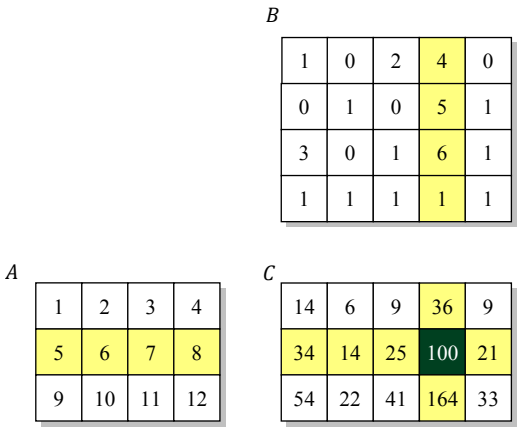
```
  TraceRecursively(1, n);
end;

begin
  Enter;
  Optimize;
  PrintResult;
end.
```

15.3.5. Phép nhân dãy ma trận

Với ma trận $A = \{a_{ij}\}$ kích thước $p \times q$ và ma trận $B = \{b_{jk}\}$ kích thước $q \times r$. Người ta có phép nhân hai ma trận đó để được ma trận $C = \{c_{ik}\}$ kích thước $p \times r$. Mỗi phần tử của ma trận C được tính theo công thức:

$$c_{ik} = \sum_{j=1}^{q} a_{ij} \times b_{jk} \; ; (1 \le i \le p; 1 \le k \le r) \qquad (15.16)$$

Ví dụ với A là ma trận kích thước 3×4, B là ma trận kích thước 4×5 thì C sẽ là ma trận kích thước 3×5 (Hình 15-8).



Hình 15-8. Phép nhân hai ma trận

Ta xét thuật toán để nhân hai ma trận $A(p \times q)$ và $B(q \times r)$:

```
for i := 1 to p do
  for j := 1 to r do
    begin
      c[i, j] := 0;
      for k := 1 to q do
        c[i, j] := c[i, j] + a[i, k] * b[k, j];
    end;
```

Phí tổn để thực hiện phép nhân ma trận có thể đánh giá qua số lần thực hiện phép nhân số học, để nhân hai ma trận $A(p \times q)$ và $B(q \times r)$ chúng ta cần $p \times q \times r$ phép nhân*.

Phép nhân ma trận không có tính chất giao hoán nhưng có tính chất kết hợp:

$$(AB)C = A(BC) \qquad (15.17)$$

Vậy nếu A là ma trận cấp 3×4, B là ma trận cấp 4×6 và C là ma trận cấp 6×8 thì:

- Để tính $(AB)C$, phép tính (AB) cho ma trận kích thước 3×6 sau $3 \times 4 \times 6 = 72$ phép nhân số học, sau đó nhân tiếp với C được ma trận kết quả kích thước 3×8 sau $3 \times 6 \times 8 = 144$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 216.
- Để tính $A(BC)$, phép tính (BC) cho ma trận kích thước 4×8 sau $4 \times 6 \times 8 = 192$ phép nhân số học, lấy A nhân với ma trận này được ma trận kết quả kích thước 3×8 sau $3 \times 4 \times 8 = 96$ phép nhân số học. Vậy tổng số phép nhân số học phải thực hiện sẽ là 288.

Ví dụ này cho chúng ta thấy rằng trình tự thực hiện có ảnh hưởng lớn tới chi phí. Vấn đề đặt ra là tính số phí tổn ít nhất khi thực hiện phép nhân một dãy các ma trận:

$$\prod_{i=1}^{n} M_i = M_1 M_2 \dots M_n$$

Trong đó:

M_1 là ma trận kích thước $a_0 \times a_1$

M_2 là ma trận kích thước $a_1 \times a_2$

…

M_n là ma trận kích thước $a_{n-1} \times a_n$

Input

---

* Để nhân hai ma trận, có một số thuật toán tốt hơn, chẳng hạn thuật toán Strassen $O(n^{\lg 7})$ hay thuật toán Coppersmith-Winograd $O(n^{2,376})$. Nhưng để đơn giản cho bài toán này, chúng ta dùng thuật toán nhân ma trận đơn giản nhất.

- Dòng 1 chứa số nguyên dương $n \leq 100$
- Dòng 2 chứa $n+1$ số nguyên dương $a_0, a_1, \ldots, a_n$ cách nhau ít nhất một dấu cách $(\forall i: a_i \leq 1000)$

Output

Cho biết phương án nhân ma trận tối ưu và số phép nhân phải thực hiện

Sample Input	Sample Output
6 9 5 3 2 4 7 8	The best solution: ((M[1].(M[2].M[3])).((M[4].M[5]).M[6])) Number of numerical multiplications: 432

❑ Thuật toán

Cách giải của bài toán này gần giống như bài toán tam giác phân: Trước hết, nếu dãy chỉ có một ma trận thì chi phí bằng 0, tiếp theo, chi phí để nhân một cặp ma trận có thể tính được ngay. Bằng việc ghi nhận chi phí của phép nhân hai ma trận liên tiếp ta có thể sử dụng những thông tin đó để tối ưu hoá chi phí nhân những bộ ba ma trận liên tiếp … Cứ tiếp tục như vậy cho tới khi ta tính được phí tổn nhân n ma trận liên tiếp.

Gọi $f[i,j]$ là số phép nhân số học tối thiểu cần thực hiện để nhân đoạn ma trận liên tiếp:

$$\prod_{z=i}^{j} M_z = M_i M_{i+1} \ldots M_j \tag{15.18}$$

Với một cách nhân tối ưu dãy ma trận $\prod_{z=i}^{j} M_z$, ta quan tâm tới phép nhân ma trận cuối cùng, chẳng hạn $\prod_{z=i}^{j} M_z$ cuối cùng được tính bằng tích của ma trận X và ma trận Y

$$\prod_{z=i}^{j} M_z = X \times Y \tag{15.19}$$

Trong đó X là ma trận tạo thành qua phép nhân dãy ma trận từ M_i đến M_k và Y là ma trận tạo thành qua phép nhân dãy ma trận từ M_{k+1} tới M_j với một chỉ số k nào đó.

Vì phép kết hợp chúng ta đang xét là tối ưu để tính $\prod_{z=i}^{j} M_z$, nên chi phí của phép kết hợp này sẽ bằng tổng của ba đại lượng:

- Chi phí nhân tối ưu dãy ma trận từ M_i đến M_k để được ma trận X $(= f[i,k])$
- Chi phí nhân tối ưu dãy ma trận từ M_{k+1} tới M_j để được ma trận Y $(= f[k+1,j])$
- Chi phí tính tích $X \times Y$ $\left(= a_{i-1} \times a_k \times a_j\right)$

Ta có công thức truy hồi và công thức tính vết:

$$f[i,j] = \min_{k:i\le k<j}\{f[i,k] + f[k+1,j] + a_{i-1}\,a_k a_j\}$$
$$t[i,j] = \arg\min_{k:i\le k<j}\{f[i,k] + f[k+1,j] + a_{i-1}\,a_k a_j\} \quad (15.20)$$

Cơ sở quy hoạch động:

$$f[i,i] = 0, \forall i: 1 \le i \le n \quad (15.21)$$

Cách truy vết để in ra phép nhân tối ưu $\prod_{z=i}^{j} M_z$ sẽ quy về việc in ra phép nhân tối ưu $\prod_{z=i}^{k} M_z$ và $\prod_{z=k+1}^{j} M_z$, với $k = t[i,j]$. Điều này có thể thực hiện bằng một thủ tục đệ quy.

❑ Cài đặt

Nhân tối ưu dãy ma trận

```
program MatrixChainMultiplication;
const
  maxN = 100;
  maxSize = 1000;
  maxValue = maxN * maxSize * maxSize * maxSize;
var
  a: array[0..maxN] of LongInt;
  f: array[1..maxN, 1..maxN] of Int64;
  t: array[1..maxN, 1..maxN] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 0 to n do Read(a[i]);
end;

procedure Optimize; //Giải công thức truy hồi
var
  m, i, j, k: LongInt;
  Trial: Int64;
begin
  for i := 1 to n do
    f[i, i] := 0;
  for m := 1 to n - 1 do
    for i := 1 to n - m do
      begin //Tính f[i, i + m]
        j := i + m;
        f[i, j] := maxValue;
        for k := i to j - 1 do //Thử các vị trí phân hoạch k
          begin
            Trial := f[i, k] + f[k + 1, j] + Int64(a[i - 1]) * a[k] * a[j];
            if Trial < f[i, j] then //Cực tiểu hoá f[i, j] và lưu vết
              begin
                f[i, j] := Trial;
```

```
                t[i, j] := k;
              end;
          end;
      end;
end;

procedure Trace(i, j: LongInt); //In ra cách nhân tối ưu dãy M[i..j]
begin
  if i = j then Write('M[', i, ']')
  else
    begin
      Write('(');
      Trace(i, t[i, j]);
      Write('.');
      Trace(t[i, j] + 1, j);
      Write(')');
    end;
end;

procedure PrintResult; //In kết quả
begin
  WriteLn('The best solution: ');
  Trace(1, n);
  WriteLn;
  WriteLn('Number of numerical multiplications: ', f[1, n]);
end;

begin
  Enter;
  Optimize;
  PrintResult;
end.
```

15.3.6. Du lịch Đông↔Tây

Bạn là người thắng cuộc trong một cuộc thi do một hãng hàng không tài trợ và phần thưởng là một chuyến du lịch do bạn tuỳ chọn. Có n thành phố và chúng được đánh số từ 1 tới n theo vị trí từ Tây sang Đông (không có hai thành phố nào ở cùng kinh độ), giữa hai thành phố có thể có một tuyến bay hai chiều do hãng quản lý. Chuyến du lịch của bạn phải xuất phát từ thành phố 1, bay theo các tuyến bay của hãng tới thành phố n và chỉ được bay từ Tây sang Đông, sau đó lại bay theo các tuyến bay của hãng về thành phố 1 và chỉ được bay từ Đông sang Tây. Hành trình không được thăm bất kỳ thành phố nào quá một lần, ngoại trừ thành phố 1 là nơi bắt đầu và kết thúc hành trình.

Yêu cầu đặt ra là tìm hành trình du lịch qua nhiều thành phố nhất.

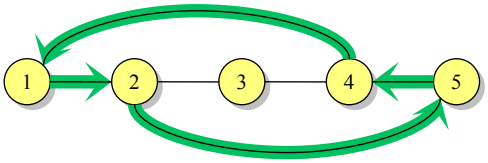
Input

- Dòng 1 chứa số thành phố (n) và số tuyến bay (m), $3 \le n \le 100; m \le \binom{n}{2}$
- m dòng tiếp, mỗi dòng chứa thông tin về một tuyến bay: gồm chỉ số hai thành phố tương ứng với tuyến bay đó.

Output

Hành trình tối ưu tìm được hoặc thông báo rằng không thể thực hiện được hành trình theo yêu cầu đặt ra.

Sample Input	Sample Output
5 6 1 2 2 3 3 4 4 5 1 4 2 5	The best tour: Number of cities: 4 1 2 5 4 1



❑ Thuật toán

Ta có thể đặt một mô hình đồ thị $G = (V, E)$ với tập đỉnh V là tập các thành phố và tập cạnh E là tập các tuyến bay. Bài toán này nếu không có ràng buộc về hành trình Tây–Đông-Tây thì có thể quy dẫn về bài toán tìm chu trình Hamilton (Hamilton Circuit) trên đồ thị G. Bài toán tìm chu trình Hamilton thuộc về lớp bài toán cho tới nay chưa có thuật toán hiệu quả với độ phức tạp đa thức để giải quyết, nhưng bài toán này nhờ có ràng buộc về hướng đi nên chúng ta có thể xây dựng một thuật toán cho kết quả tối ưu với độ phức tạp $O(n^3)$. Tại IOI'93*, bài toán du lịch Đông–Tây này được coi là bài toán khó nhất trong số bốn bài toán của đề thi.

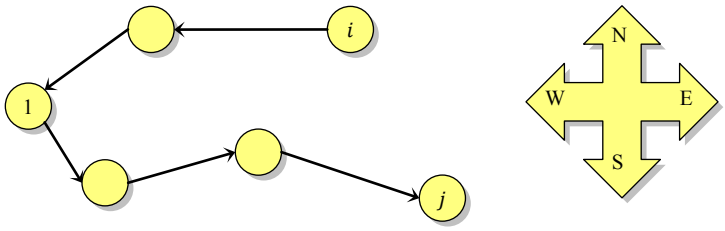
Dễ thấy rằng hành trình đi qua nhiều thành phố nhất cũng là hành trình đi bằng nhiều tuyến bay nhất mà không có thành phố nào thăm qua hai lần ngoại trừ thành phố 1 là nơi bắt đầu và kết thúc hành trình. Với hai thành phố i và j trong đó $i < j$, ta xét các đường bay xuất phát từ thành phố i, bay theo hướng Tây tới thành phố 1 rồi bay theo hướng Đông tới thành phố j sao cho không có thành phố nào bị thăm hai lần.

- Nếu tồn tại đường bay như vậy, ta xét đường bay qua nhiều tuyến bay nhất, ký hiệu $(i \rightsquigarrow 1 \rightsquigarrow j)$ và gọi $f[i,j]$ là số tuyến bay dọc theo đường bay này.
- Nếu không tồn tại đường bay thoả mãn điều kiện đặt ra, ta coi $f[i,j] = -1$

Chú ý rằng chúng ta chỉ quan tâm tới các $f[i,j]$ với $i < j$, tức là thành phố i ở phía Tây thành phố j mà thôi.

---

*The 5th International Olympiad in Informatics (Mendoza – Argentina)

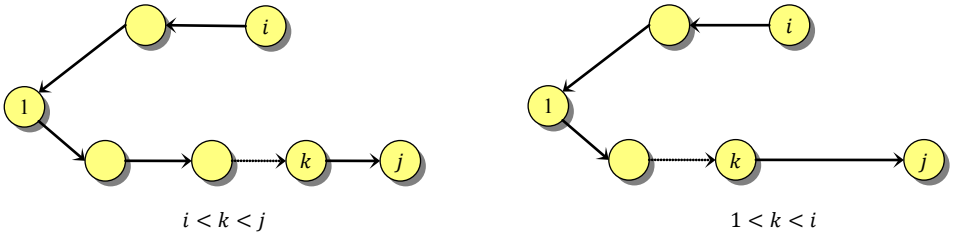


Hình 15-9. Đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$)

Nếu tính được các $f[i,n]$ thì tua du lịch cần tìm sẽ gồm các tuyến bay trên một đường bay ($i \rightsquigarrow 1 \rightsquigarrow n$) nào đó ghép thêm tuyến bay (n,i), tức là số tuyến bay (nhiều nhất) trên tua du lịch tối ưu cần tìm sẽ là $\max_i\{f[i,n]\}+1$. Vấn đề bây giờ là phải xây dựng công thức tính các $f[i,j]$.

Với $i<j$, xét đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$), đường bay này sẽ bay qua một thành phố k nào đó trước khi kết thúc tại thành phố j. Có hai khả năng xảy ra:

- Thành phố k nằm phía đông (bên phải) thành phố i $(k>i)$, khi đó đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) có thể coi là đường bay ($i \rightsquigarrow 1 \rightsquigarrow k$) ghép thêm tuyến bay (k,j). Vậy trong trường hợp này $f[i,j]=f^{(k)}[i,j]=f[i,k]+1$
- Thành phố k nằm phía tây (bên trái) thành phố i $(k<i)$, khi đó đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) có thể coi là đường bay ($k \rightsquigarrow 1 \rightsquigarrow i$) theo chiều ngược lại rồi ghép thêm tuyến bay (k,j). Vậy trong trường hợp này $f[i,j]=f^{(k)}[i,j]=f[k,i]+1$



Hình 15-10. Đường bay ($i \rightsquigarrow 1 \rightsquigarrow j$) và hai trường hợp về vị trí thành phố k

Vì tính cực đại của $f[i,j]$, ta suy ra công thức truy hồi:

$$f[i,j]=\max_k\{f^{(k)}[i,j]\} \tag{15.22}$$

Trong đó các chỉ số k thoả mãn $1\le k<j$, $k\ne i$, $f^{(k)}[i,j]\ne -1$ và (k,j) là một tuyến bay. Các giá trị $f^{(k)}[i,j]$ trong công thức truy hồi (15.22) được tính bởi:

$$f^{(k)}[i,j]=\begin{cases} f[i,k]+1; & \text{nếu } k>i \\ f[k,i]+1; & \text{nếu } k<i \end{cases} \tag{15.23}$$

Các phần tử của bảng f sẽ được tính theo từng hàng từ trên xuống và trên mỗi hàng thì các phần tử sẽ được tính từ trái qua phải. Các phần tử hàng 1 của bảng f ($f[1,j]$) sẽ được tính trước để làm cơ sở quy hoạch động.

Bài toán tính các giá trị $f[1,j]$ lại là một bài toán quy hoạch động: $f[1,j]$ theo định nghĩa là số tuyến bay trên đường bay Tây→Đông qua nhiều tuyến bay nhất từ 1 tới j. Đường bay này sẽ bay qua thành phố k nào đó trước khi kết thúc tại thành phố j. Theo nguyên lý bài toán con tối ưu, ta có công thức truy hồi:

$$f[1,j] = \max_k\{f[1,k]\} + 1 \qquad (15.24)$$

với các thành phố k thoả mãn: $k < j$, $f[1,k] \neq -1$ và (k,j) là một tuyến bay.

Cơ sở quy hoạch động để tính các $f[1,j]$ chính là $f[1,1] = 0$ (số tuyến bay trên đường bay Tây→Đông qua nhiều tuyến bay nhất từ 1 tới 1)

Quy trình giải có thể tóm tắt qua hai bước chính:

- Đặt $f[1,1] := 0$ và tính các $f[1,j]$ $(1 < j \leq n)$ bằng công thức (15.24)
- Dùng các $f[1,j]$ vừa tính được làm cơ sở, tính các $f[i,j]$ $(2 \leq i \leq n)$ bằng công thức (15.22)

Song song với việc tính mỗi $f[i,j]$, chúng ta lưu trữ lại $Trace[i,j]$ là thành phố k đứng liền trước thành phố j trên đường bay $(i \rightsquigarrow 1 \rightsquigarrow j)$. Việc chỉ ra tua du lịch tối ưu sẽ quy về việc chỉ ra một đường bay $(i \rightsquigarrow 1 \rightsquigarrow n)$ nào đó, việc chỉ ra đường bay $(i \rightsquigarrow 1 \rightsquigarrow j)$ có thể quy về việc chỉ ra đường bay $(i \rightsquigarrow 1 \rightsquigarrow Trace[i,j])$ hoặc đường bay $(Trace[i,j] \rightsquigarrow 1 \rightsquigarrow i)$ tuỳ theo giá trị $Trace[i,j]$ lớn hơn hay nhỏ hơn i.

❑ Cài đặt

💻 Du lịch Đông↔Tây

```
{$MODE OBJFPC}
{$INLINE ON}
program TheLongestBitonicTour;
const
  max = 1000;
type
  TPath = record
    nCities: LongInt;
    c: array[1..max] of LongInt;
  end;
var
  a: array[1..max, 1..max] of Boolean;
  f: array[1..max, 1..max] of LongInt;
  Trace: array[1..max, 1..max] of LongInt;
  p, q: TPath;
  n: LongInt;
  LongestTour: LongInt;
```

```
procedure Enter; //Nhập dữ liệu
var
  i, m, u, v: LongInt;
begin
  FillChar(a, SizeOf(a), False); //a[u, v] = True ↔ (u,v) là một tuyến bay
  ReadLn(n, m);
  for i := 1 to m do
    begin
      ReadLn(u, v); //Tuyến bay hai chiều: a[u, v] = a[v, u]
      a[u, v] := True;
      a[v, u] := True;
    end;
end;

//Target := Max(Target, Value), trả về True nếu Target được cực đại hoá.
function Maximize(var Target: LongInt; Value: LongInt): Boolean; inline;
begin
  Result  := Value > Target;
  if Result then Target := Value;
end;

procedure Optimize; //Giải công thức truy hồi
var
  i, j, k: LongInt;
begin
  //Tính các f[1, j]
  f[1, 1] := 0; //Cơ sở QHĐ để tính các f[1, j]
  for j := 2 to n do
    begin //Tính f[1, j]
      f[1, j] := -1;
      for k := 1 to j - 1 do
        if a[k, j] and (f[1, k] <> -1) and
           Maximize(f[1, j], f[1, k] + 1) then
          Trace[1, j] := k; //Lưu vết mỗi khi f[1, j] được cực đại hoá
    end;
  //Dùng các f[1, j] làm cơ sở QHĐ, tính các f[i, j]
  for i := 2 to n - 1 do
    for j := i + 1 to n do
      begin
        f[i, j] := -1;
        for k := i + 1 to j - 1 do //k nằm phía Đông i
          if a[k, j] and (f[i, k] <> -1) and
             Maximize(f[i, j], f[i, k] + 1) then
            Trace[i, j] := k;
        for k := 1 to i - 1 do //k nằm phía Tây i
          if a[k, j] and (f[k, i] <> -1) and
             Maximize(f[i, j], f[k, i] + 1) then
            Trace[i, j] := k;
      end;
end;

procedure FindPaths; //Truy vết tìm đường
var
  i, j, k, t: LongInt;
begin
  //Tính LongestTour là số tuyến bay nhiều nhất trên đường bay k→1→n với mọi k
```

```
  LongestTour := -1;
  for k := 1 to n - 1 do
    if a[k, n] and Maximize(LongestTour, f[k, n]) then
      i := k;
  if LongestTour = -1 then Exit;
  j := n;
  //Tua du lịch cần tìm sẽ là gồm các tuyến bay trên đường bay i→1→j=n  ghép thêm tuyến (i, n)
  //Chúng ta tìm 2 đường bay Đông→Tây: p:i→1 và q:j→1 gồm các tuyến bay trên đường bay i→1→j
  p.nCities := 1; p.c[1] := i;  //Lưu trữ thành phố cuối cùng của đường p
  q.nCities := 1; q.c[1] := j;  //Lưu trữ thành phố cuối cùng của đường q
  repeat
    if i < j then //Truy vết đường bay i→1→j
      with q do
        begin
          k := Trace[i, j]; //Xét thành phố k đứng liền trước j
          Inc(nCities); c[nCities] := k; //Đưa k vào đường q
          j := k;
        end
    else //Truy vết đường bay j→1→i
      with p do
        begin
          k := Trace[j, i]; //Xét thành phố k đứng liền trước i
          Inc(nCities); c[nCities] := k; //Đưa k vào đường p
          i := k;
        end;
  until i = j; //Khi i=j thì chắc chắn  i = j = 1
end;

procedure PrintResult; //In kết quả
var
  i: LongInt;
begin
  if LongestTour = -1 then
    WriteLn('NO SOLUTION!')
  else
    begin
      WriteLn('The best tour: ');
      WriteLn('Number of cities: ', LongestTour + 1);
      //Để in ra tua du lịch tối ưu, ta sẽ in ghép 2 đường:
      //In ngược đường p để có đường đi Tây→Đông 1→i
      //In xuôi đường q để có đường đi Đông→Tây n→1
      for i := p.nCities downto 1 do Write(p.c[i], ' ');
      for i := 1 to q.nCities do Write(q.c[i], ' ');
    end;
end;

begin
  Enter;
  Optimize;
  FindPaths;
  PrintResult;
end.
```

15.3.7. Thuật toán Viterbi

Thuật toán Viterbi được đề xuất bởi Andrew Viterbi như một phương pháp hiệu chỉnh lỗi trên đường truyền tín hiệu số. Nó được sử dụng để giải mã chập sử dụng trong điện thoại

di động kỹ thuật số (CDMA/GSM), đường truyền vệ tinh, mạng không dây, v.v… Trong ngành khoa học máy tính, thuật toán Viterbi được sử dụng rộng rãi trong Tin-Sinh học, xử lý ngôn ngữ tự nhiên, nhận dạng tiếng nói. Khi nói tới thuật toán Viterbi, không thể không kể đến một ứng dụng quan trọng của nó trong mô hình Markov ẩn (Hidden Markov Models - HMMs) để tìm dãy trạng thái tối ưu đối với một dãy tín hiệu quan sát được.

Việc trình bày cụ thể một mô hình thực tế có ứng dụng của thuật toán Viterbi là rất dài dòng, chúng ta sẽ tìm hiểu thuật toán này qua một bài toán đơn giản hơn để qua đó hình dung được cách thức thuật toán Viterbi tìm đường đi tối ưu trên lưới như thế nào.

❑ Bài toán

Một dây chuyền lắp ráp ô tô có một robot và n dụng cụ đánh số từ 1 tới n. Có tất cả m loại bộ phận trong một chiếc ô tô đánh số từ 1 tới m. Mỗi chiếc ô tô phải được lắp ráp từ t bộ phận $O = (o_1, o_2, \ldots, o_t)$ theo đúng thứ tự này $(\forall i: 1 \le o_i \le m)$. Biết được những thông tin sau:

- Tại mỗi thời điểm, robot chỉ có thể cầm được 1 dụng cụ.
- Tại thời điểm bắt đầu, robot không cầm dụng cụ gì cả và phải chọn một trong số n dụng cụ đã cho, thời gian chọn không đáng kể.
- Khi đã có dụng cụ, robot sẽ sử dụng nó để lắp một bộ phận trong dãy O, biết thời gian để Robot lắp bộ phận loại v bằng dụng cụ thứ i là b_{iv} $(1 \le i \le n; 1 \le v \le m)$
- Sau khi lắp xong mỗi bộ phận, robot được phép đổi dụng cụ khác để lắp bộ phận tiếp theo, biết thời gian đổi từ dụng cụ i sang dụng cụ j là a_{ij}. (a_{ij} có thể khác a_{ji} và a_{ii} luôn bằng 0).

Hãy tìm ra quy trình lắp ráp ô tô một cách nhanh nhất.

Input

- Dòng 1: Chứa ba số nguyên dương n, m, t $(n, m, t \le 500)$
- Dòng 2: Chứa t số nguyên dương $o_1, o_2, \ldots, o_t$ $(\forall i: 1 \le o_i \le m)$
- n dòng tiếp theo, dòng thứ i chứa n số nguyên, số thứ j là a_{ij} $(0 \le a_{ij} \le 500)$
- n dòng tiếp theo, dòng thứ j chứa m số nguyên, số thứ v là b_{iv} $(0 \le b_{iv} \le 500)$

Output

Quy trình lắp ráp ô tô nhanh nhất

Sample Input	Sample Output
3 4 6 1 2 3 2 3 4 0 1 2 3 0 4 5 6 0 8 8 1 5 8 1 8 8 1 8 8 5	Component Tool ---------- ---------- 1 3 2 2 3 1 2 2 3 1 4 1 Time for assembling: 23

❑ Thuật toán

Gọi $f[k,i]$ là thời gian ít nhất để lắp ráp lần lượt các bộ phận $o_k, o_{k+1}, \ldots, o_t$ mà dụng cụ dùng để lắp bộ phận đầu tiên trong dãy (o_k) là dụng cụ i. Nếu dụng cụ dùng để lắp bộ phận tiếp theo (o_{k+1}) là dụng cụ j thì thời gian ít nhất để lắp lần lượt các bộ phận $o_k, o_{k+1}, \ldots, o_t$: $f[k,i]$ sẽ được tính bằng:

- Thời gian lắp bộ phận o_k: $b[i, o_k]$
- Cộng với thời gian đổi từ dụng cụ i sang dụng cụ j: $a[i,j]$
- Cộng với thời gian ít nhất để lắp ráp lần lượt các bộ phận $o_{k+1}, o_{k+2}, \ldots, o_t$, trong đó bộ phận o_{k+1} được lắp bằng dụng cụ j: $f[k+1,j]$

Vì chúng ta muốn $f[k,i]$ là nhỏ nhất có thể, chúng ta sẽ thử mọi khả năng chọn dụng cụ j để lắp o_{k+1} và chọn phương án tốt nhất để cực tiểu hoá $f[k,i]$. Từ đó suy ra công thức truy hồi:

$$f[k,i] = \min_{j:1\le j\le n} \{b[i,o_k] + a[i,j] + f[k+1,j]\} \qquad (15.25)$$

Bảng phương án f là bảng hai chiều t hàng, n cột. Trong đó các phần tử ở hàng k sẽ được tính qua các phần tử ở hàng $k+1$, như vậy ta sẽ phải tính những phần tử ở hàng t của bảng làm cơ sở quy hoạch động. Theo định nghĩa, $f[t,i]$ là thời gian (ít nhất) để lắp duy nhất một bộ phận o_t bằng dụng cụ i nên suy ra $f[t,i] = b[i,o_t]$.

Song song với việc tính các phần tử bảng f bằng công thức (15.25), mỗi khi tính được $f[k,i]$ đạt giá trị nhỏ nhất tại $j = j^*$ nào đó, chúng ta đặt:

$$Trace[k,i] := j^* = \arg\min_{j:1\le j\le n} \{b[i,o_k] + a[i,j] + f[k+1,j]\} \qquad (15.26)$$

để chỉ ra rằng phương án tối ưu tương ứng với $f[k,j]$ sẽ phải chọn dụng cụ tiếp theo là dụng cụ $Trace[k,i]$ để lắp bộ phận o_{k+1}.

Sau khi tính được tất cả các phần tử của bảng f, ta quan tâm tới các phần tử trên hàng 1. Theo định nghĩa $f[1,i]$ là thời gian ít nhất để lắp hoàn chỉnh một chiếc ô tô với dụng cụ

đầu tiên được sử dụng là i. Cũng tương tự trên, chúng ta muốn lắp hoàn chỉnh một chiếc ô tô trong thời gian ngắn nhất nên công việc còn lại là thử mọi khả năng chọn dụng cụ đầu tiên và chọn ra dụng cụ s_1 có $f[1, s_1]$ nhỏ nhất, $f[1, s_1]$ chính là thời gian ngắn nhất để lắp hoàn chỉnh một chiếc ô tô. Từ cơ chế lưu vết, ta có dụng cụ tiếp theo cần sử dụng để lắp o_2 là $s_2 = Trace[1, s_1]$, dụng cụ cần sử dụng để lắp o_3 là $s_3 = Trace[2, s_2]$,… Những công đoạn chính trong việc thiết kế một thuật toán quy hoạch động đã hoàn tất, bao gồm: Dựng công thức truy hồi, tìm cơ sở quy hoạch động, tìm cơ chế lưu vết và truy vết.

❑ Cài đặt

Nhìn vào công thức truy hồi (15.25) để tính các $f[k, i]$, có thể nhận thấy rằng việc tính phần tử trên hàng k chỉ cần dựa vào các phần tử trên hàng $k + 1$. Vậy thì tại mỗi bước tính một hàng của bảng phương án, ta chỉ cần lưu trữ hai hàng liên tiếp dưới dạng hai mảng một chiều x và y: mảng $x_{1\ldots n}$ tương ứng với hàng vừa được tính (hàng $k + 1$) và mảng $y_{1\ldots n}$ tương ứng với hàng liền trước sắp được tính (hàng k), công thức truy hồi dùng mảng x tính mảng y sẽ được viết lại là:

$$y[i] = \min_{j:1 \le j \le n} \{b[i, o_k] + a[i, j] + x[j]\} \qquad (15.27)$$

Sau khi tính xong một hàng, hai mảng x và y được đổi vai trò cho nhau và quá trình tính được lặp lại. Cuối cùng chúng ta vẫn lưu trữ được hàng 1 của bảng phương án và từ đó truy vết tìm ra phương án tối ưu.

Mẹo nhỏ này không những tiết kiệm bộ nhớ cho bảng phương án mà còn làm tăng tốc đáng kể quá trình tính toán so với phương pháp tính trực tiếp trên mảng hai chiều f. Tốc độ được cải thiện nhờ hai nguyên nhân chính:

- Việc truy cập phần tử của mảng một chiều nhanh hơn việc truy cập phần tử của mảng hai chiều vì phép tính địa chỉ được thực hiện đơn giản hơn, hơn nữa nếu bạn bật chế độ kiểm tra tràn phạm vi (range checking) khi dịch chương trình, truy cập phần tử mảng một chiều chỉ cần kiểm tra phạm vi một chỉ số trong khi truy cập phần tử mảng hai chiều phải kiểm tra phạm vi cả hai chỉ số.
- Nếu dùng hai mảng một chiều tính xoay lẫn nhau, vùng bộ nhớ của cả hai mảng bị đọc/ghi nhiều lần và bộ vi xử lý sẽ có cơ chế nạp cả hai mảng này vào vùng nhớ cache ngay trong bộ vi xử lý, việc đọc/ghi dữ liệu sẽ không qua RAM nữa nên đạt tốc độ nhanh hơn nhiều. (Sự chậm chạp của RAM so với cache cũng có thể so sánh như tốc độ của đĩa cứng và RAM).

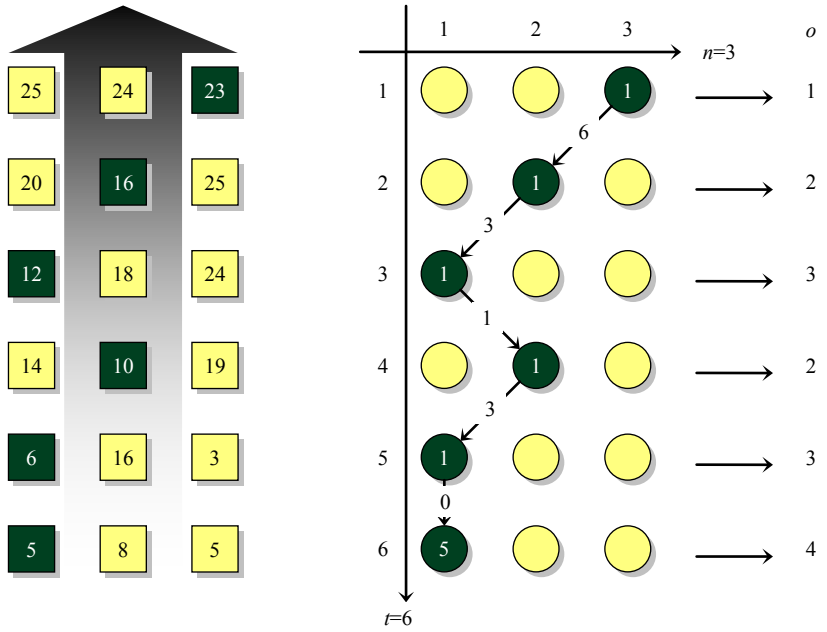
Bạn có thể cài đặt thử phương pháp tính trực tiếp mảng hai chiều và phương pháp tính xoay vòng hai mảng một chiều để so sánh tốc độ trên các bộ dữ liệu lớn. Để thấy rõ hơn sự khác biệt về tốc độ, có thể đổi yêu cầu của bài toán chỉ là đưa ra thời gian ngắn nhất để lắp ráp mà không cần đưa ra phương án thực thi, khi đó mảng hai chiều *Trace* cũng không cần dùng đến nữa.

Lắp ráp ô tô

```
{$MODE OBJFPC}
{$INLINE ON}
program ViterbiAlgorithm;
const
  maxN = 500;
  maxM = 500;
  maxT = 500;
  maxTime = 500;
  Infinity = maxT * maxTime + 1;
type
  TLine = array[1..maxN] of LongInt;
  PLine = ^TLine;
var
  a: array[1..maxN, 1..maxN] of LongInt;
  b: array[1..maxN, 1..maxM] of LongInt;
  o: array[1..maxT] of LongInt;
  x, y: PLine;
  Trace: array[1..maxT, 1..maxN] of LongInt;
  n, m, t: LongInt;
  MinTime: LongInt;
  Tool: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i, j, k: LongInt;
begin
  ReadLn(n, m, t);
  for i := 1 to t do Read(o[i]);
  ReadLn;
  for i := 1 to n do
    begin
      for j := 1 to n do Read(a[i, j]);
      ReadLn;
    end;
  for i := 1 to n do
    begin
      for k := 1 to m do Read(b[i, k]);
      ReadLn;
    end;
end;

//Target := Min(Target, Value); Trả về True nếu Target được cực tiểu hoá
function Minimize(var Target: LongInt; Value: LongInt): Boolean; inline;
begin
  Result := Value < Target;
  if Result then Target := Value;
```

```
end;

procedure Optimize; //Quy hoạch động
var
  i, j, k: LongInt;
  temp: PLine;
begin
  New(x); New(y);
  try
    for i := 1 to n do //Cơ sở quy hoạch động: x^ := hàng t của bảng phương án
      x^[i] := b[i, o[t]];
    for k := t - 1 downto 1 do
      begin //Dùng x^ tính y^ ↔ dùng f[k + 1, .] tính f[k, .]
        for i := 1 to n do
          begin
            y^[i] := Infinity;
            for j := 1 to n do
              if Minimize(y^[i], a[i, j] + x^[j]) then
                Trace[k, i] := j; //Cực tiểu hoá y^[i] kết hợp lưu vết
            Inc(y^[i], b[i, o[k]]);
          end;
        temp := x; x := y; y := temp; //Đảo hai con trỏ ↔ đảo vai trò x và y
      end;
    //x^ giờ đây là hàng 1 của bảng phương án, tìm phần tử nhỏ nhất của x^ và dụng cụ đầu tiên được dùng
    MinTime := Infinity;
    for i := 1 to n do
      if x^[i] < MinTime then
        begin
          MinTime := x^[i];
          Tool := i;
        end;
  finally
    Dispose(x); Dispose(y);
  end;
end;

procedure PrintResult; //In kết quả
var
  k: LongInt;
begin
  WriteLn('Component  Tool');
  WriteLn('---------------');
  for k := 1 to t do
    begin
      WriteLn(o[k]:5, Tool:9); //In ra dụng cụ Tool dùng để lắp o[k]
      Tool := Trace[k, Tool]; //Chuyển sang dụng cụ kế tiếp
    end;
  WriteLn('Time for assembling: ', MinTime);
end;

begin
  Enter;
  Optimize;
  PrintResult;
end.
```

Người ta thường phát biểu thuật toán Viterbi trên mô hình đồ thị: Xét đồ thị có hướng gồm $t \times n$ đỉnh, tập các đỉnh này được chia làm t lớp, mỗi lớp có đúng n đỉnh. Các cung của đồ thị chỉ nối từ một đỉnh đến một đỉnh khác thuộc lớp kế tiếp. Mỗi đỉnh và mỗi cung của đồ thị đều có gán một trọng số (chi phí). Trọng số (chi phí) của một đường đi trên đồ thị bằng tổng trọng số các đỉnh và các cung đi qua. Thuật toán Viterbi chính là để tìm đường đi ngắn nhất (có chi phí ít nhất) trên lưới từ lớp 1 tới lớp t. Trong ví dụ cụ thể này, trọng số cung nối đỉnh i của lớp k với đỉnh j của lớp $k+1$ chính là $a[i,j]$ và trọng số đỉnh i của lớp k chính là $b[i,o_k]$. Thuật toán Viterbi cần thời gian $\Theta(n^2t)$ để tính bảng phương án và cần thời gian $\Theta(t)$ để truy vết tìm ra nghiệm. Hình 15-11 mô tả cách thức tính bảng phương án và truy vết trên ví dụ cụ thể của đề bài.



Hình 15-11. Thuật toán Viterbi tính toán và truy vết

Một bài toán quy hoạch động có thể có nhiều cách tiếp cận khác nhau, chọn cách nào là tuỳ theo yêu cầu bài toán sao cho thuận tiện. Điều quan trọng nhất để giải một bài toán quy hoạch động chính là phải nhìn ra được bản chất đệ quy của bài toán và tìm ra công thức truy hồi để giải. Việc này không có phương pháp chung nào cả mà hoàn toàn dựa vào sự khéo léo và kinh nghiệm của bạn - những kỹ năng chỉ có được nhờ luyện tập.

Bài tập 15-1.

Cho ba số nguyên dương n, k và p $(n, k \leq 1000)$, hãy cho biết có bao nhiêu số tự nhiên có không quá n chữ số mà tổng các chữ số đúng bằng k, đồng thời cho biết nếu mang tất cả các số đó sắp xếp theo thứ tự tăng dần thì số đứng thứ p là số nào.

Gợi ý: Tìm công thức truy hồi tính $f[i,j]$ là số các số có $\leq i$ chữ số mà tổng các chữ số đúng bằng j.

Bài tập 15-2.

Cho dãy số nguyên dương $A = (a_1, a_2, \ldots, a_n)$ $(n \leq 1000; \forall i: a_i \leq 1000)$ và một số m. Hãy chọn ra một số phần tử trong dãy A mà các phần tử được chọn có tổng đúng bằng m.

Gợi ý: Tìm công thức truy hồi tính $f[t]$ là chỉ số nhỏ nhất thoả mãn: tồn tại cách chọn trong dãy $a_1, a_2, \ldots, a_{f[t]}$ ra một số phần tử có tổng đúng bằng t.

Bài tập 15-3.

Cho dãy số tự nhiên $A = (a_1, a_2, \ldots, a_n)$. Ban đầu các phần tử của A được đặt liên tiếp theo đúng thứ tự cách nhau bởi dấu "?":

$$a_1 ? a_2 ? \ldots ? a_n$$

Yêu cầu: Cho trước số nguyên m, hãy tìm cách thay các dấu "?" bằng dấu cộng hay dấu trừ để được một biểu thức số học cho giá trị là m. Biết rằng $1 \leq n \leq 1000$ và $0 \leq a_i \leq 1000, \forall i$.

Ví dụ: Ban đầu dãy A là $1?2?3?4$, với $m = 0$ sẽ có phương án $1 - 2 - 3 + 4$.

Bài tập 15-4. Cho một lưới ô vuông kích thước $m \times n$ $(m, n \leq 1000)$, các hàng của lưới được đánh số từ 1 tới m và các cột của lưới được đánh số từ 1 tới n, trên mỗi ô của lưới ghi một số nguyên có giá trị tuyệt đối không quá 1000. Người ta muốn tìm một cách đi từ cột 1 tới cột n của lưới theo quy tắc: Từ một ô (i, j) chỉ được phép đi sang một trong các ô ở cột bên phải có đỉnh chung với ô (i, j). Hãy chỉ ra cách đi mà tổng các số ghi trên các ô đi qua là lớn nhất.

7	2	1	2	6
1	2	5	4	5
1	5	3	5	2
5	2	3	1	1

Bài tập 15-5.

Lập trình giải bài toán cái túi với kích thước dữ liệu: $n \leq 10000$, $m \leq 10000$ và giới hạn bộ nhớ 10MB.

Gợi ý: Vẫn sử dụng công thức truy hồi như ví dụ trong bài, nhưng thay đổi cơ chế lưu trữ. Giải công thức truy hồi lần 1, không lưu trữ toàn bộ bảng phương án mà cứ cách 100 hàng mới lưu lại 1 hàng. Sau đó với hai hàng được lưu trữ liên tiếp thì lấy hàng trên làm cơ sở, giải công thức truy hồi lần 2 tính qua 100 hàng đến hàng dưới, nhưng lần này lưu trữ toàn bộ những hàng tính được để truy vết.

Bài tập 15-6. (Dãy con chung dài nhất Longest Common Subsequence-LCS)

Xâu ký tự S gọi là xâu con của xâu ký tự T nếu có thể xoá bớt một số ký tự trong xâu T để được xâu S. Cho hai xâu ký tự $X = x_1x_2\dots x_m$ và $Y = y_1y_2\dots y_n$ $(m,n \le 1000)$. Tìm xâu Z có độ dài lớn nhất là xâu con của cả X và Y. Ví dụ: $X = 'abcdefghi123'$, $Y = 'abc1def2ghi3'$ thì xâu Z cần tìm là $Z = 'abcdefghi3'$.

Gợi ý: Tìm công thức truy hồi tính $f[i,j]$ là độ dài xâu con chung dài nhất của hai xâu $x_1x_2\dots x_i$ và $y_1y_2\dots y_j$.

Bài tập 15-7.

Một số nguyên dương x gọi là con của số nguyên dương y nếu ta có thể xoá bớt một số chữ số của y để được x. Cho hai số nguyên dương a và b $(a,b \le 10^{1000})$ hãy tìm số c là con của cả a và b sao cho giá trị của c là lớn nhất có thể. Ví dụ với $a = 123456123$, $b = 456123456$ thì $c = 456123$

Bài tập 15-8.

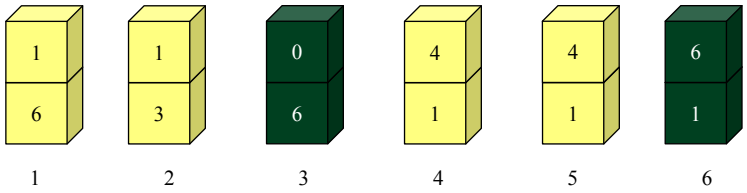
Một xâu ký tự X gọi là chứa xâu ký tự Y nếu như có thể xoá bớt một số ký tự trong xâu X để được xâu Y. Một xâu ký tự gọi là đối xứng (palindrome) nếu nó không thay đổi khi ta viết các ký tự trong xâu theo thứ tự ngược lại. Cho một xâu ký tự S có độ dài không quá 1000, hãy tìm xâu đối xứng T chứa xâu S và có độ dài ngắn nhất có thể.

Bài tập 15-9.

Có n loại tiền giấy đánh số từ 1 tới n, tờ giấy bạc loại i có mệnh giá là là một số nguyên dương v_i $(n \le 100, \forall i: v_i \le 10000)$. Hỏi muốn mua một món hàng giá là m $(m \le 10000)$ thì có bao nhiêu cách trả số tiền đó bằng những loại giấy bạc đã cho, nếu tồn tại cách trả, cho biết cách trả phải dùng ít tờ tiền nhất.

Bài tập 15-10.

Cho n quân cờ dominos xếp dựng đứng theo hàng ngang và được đánh số từ 1 đến n. Quân cờ dominos thứ i có số ghi ở ô trên là a_i và số ghi ở ô dưới là b_i. Xem hình vẽ:



Biết rằng $1 \le n \le 1000$ và $0 \le a_i, b_i \le 6, \forall i$. Cho phép lật ngược các quân dominos. Khi quân dominos thứ i bị lật, nó sẽ có số ghi ở ô trên là b_i và số ghi ở ô dưới là a_i. Vấn đề đặt ra là hãy tìm cách lật các quân dominos sao cho chênh lệch giữa tổng các số ghi ở hàng trên và tổng các số ghi ở hàng dưới là tối thiểu. Nếu có nhiều phương án lật tốt như nhau, thì chỉ ra phương án phải lật ít quân nhất.

Như ví dụ trên thì sẽ lật hai quân dominos thứ 3 và thứ 6. Khi đó:

Tổng các số ở hàng trên: $1 + 1 + 6 + 4 + 4 + 1 = 17$

Tổng các số ở hàng dưới: $6 + 3 + 0 + 1 + 1 + 6 = 17$

Bài tập 15-11.

Xét bảng $H = \{h_{ij}\}$ kích thước 4×4, các hàng và các cột được đánh chỉ số A, B, C, D. Trên 16 ô của bảng, mỗi ô ghi 1 ký tự A hoặc B hoặc C hoặc D.

H	A	B	C	D
A	A	A	B	B
B	C	D	A	B
C	B	C	B	A
D	B	D	D	D

Cho xâu $S = s_1 s_2 \dots s_n$ chỉ gồm các chữ cái $\{\text{A}, \text{B}, \text{C}, \text{D}\}$. $(1 \le n \le 10^6)$. Xét phép co $R(i)$: thay ký tự s_i và s_{i+1} bởi ký tự $h[s_i, s_{i+1}]$. Ví dụ: $S = 'ABCD'$, áp dụng liên tiếp 3 lần $R(1)$ sẽ được: $\text{ABCD} \to \text{ACD} \to \text{BD} \to \text{B}$

Yêu cầu: Cho trước một ký tự $x \in \{\text{A}, \text{B}, \text{C}, \text{D}\}$, hãy chỉ ra thứ tự thực hiện $n - 1$ phép co để ký tự còn lại cuối cùng trong S là x.

Bài tập 15-12.

Bạn có n việc cần làm đánh số từ 1 tới n, việc thứ i cần làm liên tục trong t_i đơn vị thời gian và nếu bạn hoàn thành việc thứ i không muộn hơn thời điểm d_i, bạn sẽ thu được số tiền là p_i. Bạn bắt đầu từ thời điểm 0 và không được phép làm một lúc hai việc mà phải thực hiện các công việc một cách tuần tự. Hãy chọn ra một số việc và lên kế hoạch hoàn thành các việc đã chọn sao cho tổng số tiền thu được là nhiều nhất. Biết rằng $n \le 10^6$ và các giá trị t_i, d_i, p_i là số nguyên dương không quá 10^6

Bài tập 15-13.

Cho dãy số nguyên $A = (a_1, a_2, \ldots, a_n)$ $(1 \le n \le 10000; |a_i| \le 10^9)$ và một số nguyên dương $k \le 1000$, hãy chọn ra một dãy con gồm nhiều phần tử nhất của A có tổng chia hết cho k.

Bài tập 15-14.

Công ty trách nhiệm hữu hạn "Vui vẻ" có n cán bộ đánh số từ 1 tới n ($n \le 10^5$). Cán bộ thứ i có đánh giá độ vui tính là h_i và có một thủ trưởng trực tiếp b_i, giả thiết rằng nếu i là giám đốc công ty thì $b_i = 0$. Bạn cần giúp công ty mời một nhóm cán bộ đến dự dạ tiệc "Những người thích đùa" sao cho tổng đánh giá độ vui tính của những người dự tiệc là lớn nhất, sao cho trong số những người được mời không đồng thời có mặt nhân viên cùng thủ trưởng trực tiếp của người đó.

Bài 16. Tham lam

Tham lam (*greedy*) là một phương pháp giải các bài toán tối ưu. Các thuật toán tham lam dựa vào sự đánh giá *tối ưu cục bộ địa phương* (*local optimum*) để đưa ra *quyết định tức thì* tại mỗi bước lựa chọn, với hy vọng cuối cùng sẽ tìm ra được phương án *tối ưu tổng thể* (*global optimum*).

Thuật toán tham lam có trường hợp luôn tìm rađúng phương án t ối ưu, có trường hợp không. Nhưng trong trường hợp thuật toán tham lam không tìm rađúng phương án t ối ưu, chúng ta thường thu được một phương án khả dĩ chấp nhận được.

Với một bài toán có nhiều thuật toán để giải quyết, thông thường thuật toán tham lam có tốc độ tốt hơn hẳn so với các thuật toán tối ưu tổng thể.

Khác với các kỹ thuật thiết kế thuật toán như chia để trị, liệt kê, quy hoạch động mà chúng ta đã bi ết, rất khó để đưa ra một quy trình chungđ ể tiếp cận bài toán, tìm thuật toán cũng như cài đặt thuật toán tham lam. Nếu nói về cách nghĩ, tôi chỉ có thể đưa ra hai kinh nghiệm:

- Thử tìm một thuật toán tối ưu tổng thể (ví dụ như quy hoạch động), sau đó đánh giá lại thuật toán, nhìn lại mỗi bước tối ưu và đặt câu hỏi "Liệu tại bước này có cần phải làm đến thế không?".
- Hoặc thử nghĩ xem nếu như không có máy tính, không có khái niệm gì về các chiến lược tối ưu tổng thể thì ở góc độ con người, chúng ta sẽ đưa ra giải pháp như thế nào?

16.1. Giải thuật tham lam

Giả sử có n loại tiền giấy, loại tiền thứ i có mệnh giá là v_i (đồng). Hãy chỉ ra cách trả dùng ít tờ tiền nhất để mua một mặt hàng có giá là m đồng.

Bài toán này là một bài toán có thể giải theo nhiều cách…

Nếu có các loại tiền như hệ thống tiền tệ của Việt Nam: 1 đồng, 2 đồng, 5 đồng, 10 đồng, 20 đồng, 50 đồng, 100 đồng, 200 đồng, 500 đồng…thì để trả món hàng giá 9 đồng:

- Một người máy được cài đặt chiến lược tìm kiếm vét cạn sẽ duyệt tổ hợp của tất cả các tờ tiền và tìm tổ hợp gồm ít tờ tiền nhất có tổng mệnh giá là 9 đồng. Vấn đề sẽ xảy ra nếu món hàng không phải giá 9 đồng mà 9 tỉ đồng thì người máy sẽ nghĩ đến khi … hết điện thì thôi.
- Một người máy khác được cài đặt chiến lược chia để trị (đệ quy hoặc quy hoạch động) sẽ nghĩ theo cách khác, để trả 9 đồng dĩ nhiên không cần dùng đến những tờ tiền mệnh giá lớn hơn 9. Bài toán trở thành đổi 9 đồng ra những tờ tiền 1 đồng, 2

đồng và 5 đồng. Nếu có một tờ k đồng trong phương án tối ưu thì vấn đề còn lại là đổi $9-k$ đồng, quy về ba bài toán con (với $k=1,2,5$), giải chúng và chọn phương án tốt nhất trong ba lời giải. Khi mà số tiền cần quy đổi rất lớn, giới hạn về thời gian và bộ nhớ sẽ làm cho giải pháp của người máy này bất khả thi.

- Bây giờ hãy thử nghĩ theo một cách rất con người, khi các bà nội trợ đi mua sắm, họ chẳng có máy tính, không có khái niệm gì về thuật toán chia để trị, quy hoạch động, vét cạn,… (mà có biết họ cũng chẳng dùng). Có điều để trả 9 đồng, chắc chắn họ sẽ trả bằng 1 tờ 5 đồng và 2 tờ 2 đồng. Cách làm của họ rất đơn giản, mỗi khi rút một tờ tiền ra trả, họ *quyết định tức thời* bằng cách lấy ngay tờ tiền mệnh giá cao nhất không vượt quá giá trị cần trả, mà không cần để ý đến "hậu quả" của sự quyết định đó.

Xét về giải pháp của con người trong bài toán đổi tiền, trên thực tế tôi chưa biết hệ thống tiền tệ nào khiến cho cách làm này sai. Tuy nhiên trên lý thuyết, có thể chỉ ra những hệ thống tiền tệ mà cách làm này cho ra giải pháp không tối ưu.

Giả sử chúng ta có 3 loại tiền: 1 đồng, 5 đồng và 8 đồng. Nếu cần đổi 10 đồng thì phương án tối ưu phải là dùng 2 tờ 5 đồng, nhưng cách làm này sẽ cho phương án 1 tờ 8 đồng và 2 tờ 1 đồng. Hậu quả của phép chọn tờ tiền 8 đồng đã làm cho giải pháp cuối cùng không tối ưu, nhưng dù sao cũng có thể tạm chấp nhận được trên thực tế.

Hậu quả của phép chọn tham lam tức thời sẽ tệ hại hơn nếu chúng ta chỉ có 2 loại tiền: 5 đồng và 8 đồng. Khi đó thuật toán này sẽ thất bại nếu cần đổi 10 đồng vì khi rút tờ 8 đồng ra rồi, không có cách nào đổi 2 đồng nữa.

16.2. Thiết kế một thuật toán tham lam

Thực ra chỉ có một kinh nghiệm duy nhất khi tiếp cận bài toán và tìm thuật toán tham lam là phải *khảo sát kỹ bài toán* để tìm ra các tính chất đặc biệt mà ở đó ta có thể đưa ra quyết định tức thời tại từng bước dựa vào sự đánh giá tối ưu cục bộ địa phương.

Những ví dụ dưới đây nêu lên một vài phương pháp tiếp cận bài toán và thiết kế giải thuật tham lam phổ biến.

16.3. Một số ví dụ về giải thuật tham lam

Với các bài toán mang bản chất đệ quy chúng ta có thể áp dụng quy trình sau để tìm thuật toán tham lam (nếu có):

- Phân rã bài toán lớn ra thành các bài toán con đồng dạng mà nghiệm của các bài toán con có thể dùng để chỉ ra nghiệm của bài toán lớn. Bước này giống như thuật toán chia để trị và chúng ta có thể thiết kế sơ bộ một mô hình chia để trị.
- Chỉ ra rằng *không cần giải toàn bộ các bài toán con* mà chỉ cần giải *một* bài toán con thôi là có thể chỉ ra nghiệm của bài toán lớn. Điều này cực kỳ quan trọng, nó chỉ ra

rằng sự lựa chọn tham lam tức thời (tối ưu cục bộ) tại mỗi bước sẽ dẫn đến phương án tối ưu tổng thể.

- Phân tích dãy quyết định tham lam để sửa mô hình chiad ể trị thành một giải thuật lặp.

16.3.1. Xếp lịch thực hiện các nhiệm vụ

Bài toán đầu tiên chúng ta khảo sát là bài toán xếp lịch thực hiện các nhiệm vụ (*activity selection*), phát biểu như sau:

Chúng ta có rất nhiều nhiệm vụ trong ngày. Giả sử có n nhiệm vụ và nhiệm vụ thứ i phải bắt đầu ngay sau thời điểm s_i, thực hiện liên tục và kết thúc tại thời điểm f_i. Có thể coi mỗi nhiệm vụ tương ứng với một khoảng thời gian thực hiện $(s_i, f_i]$. Hãy chọn ra nhiều nhất các nhiệm vụ để làm, sao cho không có thời điểm nào chúng ta phải làm hai nhiệm vụ cùng lúc, hay nói cách khác, khoảng thời gian thực hiện hai nhiệm vụ bất kỳ là không giao nhau.

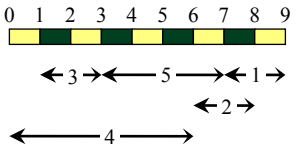
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên s_i, f_i $(0 \leq s_i < f_i \leq 10^9)$

Output

Phương án chọn ra nhiều nhiệm vụ nhất để thực hiện

Sample Input	Sample Output
5 7 9 6 8 1 3 0 6 3 7	Task 3: (1, 3] Task 5: (3, 7] Task 1: (7, 9] Number of selected tasks: 3



❑ Chia để trị

Vì các nhiệm vụ được chọn ra phải thực hiện tuần tự, chúng ta sẽ quan tâm đến nhiệm vụ đầu tiên. Nhiệm vụ đầu tiên có thể là bất kỳ nhiệm vụ nào trong số n nhiệm vụ đã cho.

Nhận xét rằng đã chọn nhiệm vụ thứ i làm nhiệm vụ đầu tiên thì tất cả những nhiệm vụ khác muốn làm sẽ phải làm sau thời điểm f_i. Bài toán trở thành chọn nhiều nhiệm vụ nhất trong số những nhiệm vụ được bắt đầu sau thời điểm f_i (bài toán con).

Vậy thì chúng ta có thuật toán chia để trị: Với tập các nhiệm vụ $M = \{1,2, \dots, n\}$, thử tất cả n khả năng chọn nhiệm vụ đầu tiên. Với mỗi phép thử lấy nhiệm vụ i làm nhiệm vụ đầu tiên, xác định M_i là tập các nhiệm vụ bắt đầu sau thời điểm f_i và giải bài toán con với

tập các nhiệm vụ $M_i \subsetneq M$. Sau n phép thử chọn nhiệm vụ đầu tiên, ta *đánh giá n kết quả* tìm được và lấy phương án thực hiện được nhiều nhiệm vụ nhất.

❑ Phép chọn tham lam

Thuật toán chia để trị phân rã bài toán lớn thành các bài toán con dựa trên phép chọn nhiệm vụ đầu tiên để làm, ta có một quan sát làm cơ sở cho phép chọn tham lam:

Định lý 16-1

Trong số các phương án tối ưu, chắc chắn có một phương án mà nhiệm vụ đầu tiên được chọn là nhiệm vụ kết thúc sớm nhất.

Chứng minh

Gọi i_1 là nhiệm vụ kết thúc sớm nhất. Với một phương án tối ưu bất kỳ, giả sử thứ tự các nhiệm vụ cần thực hiện trong phương án tối ưu đó là $(j_1, j_2, \dots, j_k)$. Do i_1 là nhiệm vụ kết thúc sớm nhất nên chắc chắn nó không thể kết thúc muộn hơn j_1, vì vậy việc thay j_1 bởi i_1 trong phương án này sẽ không gây ra sự xung đột nào về thời gian thực hiện các nhiệm vụ. Sự thay thế này cũng không làm giảm bớt số lượng nhiệm vụ thực hiện được trong phương án tối ưu, nên $(\boxed{i_1}, j_2, \dots, j_k)$ cũng sẽ là một phương án tối ưu.

Yêu cầu của bài toán là chỉ cần đưa ra một phương án tối ưu, vì thế ta sẽ chỉ ra phương án tối ưu có nhiệm vụ đầu tiên là nhiệm vụ kết thúc sớm nhất trong số n nhiệm vụ. Điều này có nghĩa là chúng ta không cần thử n khả năng chọn nhiệm vụ đầu tiên, đi giải các bài toán con, rồi mới đánh giá chúng để đưa ra quyết định cuối cùng. Chúng ta sẽ đưa ngay ra *quyết định tức thời*: chọn ngay nhiệm vụ kết thúc sớm nhất i_1 làm nhiệm vụ đầu tiên.

Sau khi chọn nhiệm vụ i_1, bài toán lớn quy về bài toán con: Chọn nhiều nhiệm vụ nhất trong số các nhiệm vụ được bắt đầu sau khi i_1 kết thúc. Phép chọn tham lam lại cho ta một quyết định tức thời: nhiệm vụ tiếp theo trong phương án tối ưu sẽ là nhiệm vụ bắt đầu sau thời điểm $f[i_1]$ và có thời điểm kết thúc sớm nhất, gọi đó là nhiệm vụ i_2. Và cứ như vậy chúng ta chọn tiếp các nhiệm vụ $i_3, i_4, \dots$

❑ Cài đặt giải thuật tham lam

Tư tưởng của giải thuật tham lam có thể tóm tắt lại:

Chọn i_1 là nhiệm vụ kết thúc sớm nhất

Chọn i_2 là nhiệm vụ kết thúc sớm nhất bắt đầu sau khi i_1 kết thúc: $s[i_2] \geq f[i_1]$

Chọn i_3 là nhiệm vụ kết thúc sớm nhất bắt đầu sau khi i_2 kết thúc: $s[i_3] \geq f[i_2]$

…

Cứ như vậy cho tới khi không còn nhiệm vụ nào chọn được nữa.

Đến đây ta có thể thiết kế một giải thuật lặp:

- Sắp xếp các nhiệm vụ theo thứ tự không giảm của thời điểm kết thúc $f[.]$
- Khởi tạo thời điểm $FinishTime := 0$
- Duyệt các nhiệm vụ theo danh sách đã sắp xếp (nhiệm vụ kết thúc sớm sẽ được xét trước nhiệm vụ kết thúc muộn), nếu xét đến nhiệm vụ i có $s[i] \geq FinishTime$ thì chọn ngay nhiệm vụ i vào phương án tối ưu và cập nhật $FinishTime$ thành thời điểm kết thúc nhiệm vụ i: $FinishTime := f[i]$

Xếp lịch thực hiện các nhiệm vụ

```
program ActivitySelection;
const
  max = 100000;
var
  s, f: array[1..max] of LongInt;
  id: array[1..max] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do
    begin
      ReadLn(s[i], f[i]);
      id[i] := i;
    end;
end;

//QuickSort: Sắp xếp bằng chỉ số các công việc id[L]...id[H] tăng theo thời điểm kết thúc
procedure QuickSort(L, H: LongInt); //f[id[L]] ≤ f[id[L + 1]] ≤..≤ f[id[H]]
var
  i, j: LongInt;
  pivot: LongInt;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (f[id[j]] > f[pivot]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (f[id[i]] > f[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end
    else Break;
```

```
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure GreedySelection; //Thuật toán tham lam
var
  i, nTasks: LongInt;
  FinishTime: LongInt;
begin
  FinishTime := 0;
  nTasks := 0;
  for i := 1 to n do //Xét lần lượt các nhiệm vụ id[i] theo thời điểm kết thúc tăng dần
    if s[id[i]] >= FinishTime then //Nếu gặp nhiệm vụ bắt đầu sau FinishTime
      begin
        WriteLn('Task ', id[i], ': (', s[id[i]], ', ', f[id[i]], ']'); //Chọn tức thì
        Inc(nTasks);
        FinishTime := f[id[i]]; //Cập nhật lại thời điểm kết thúc mới FinishTime
      end;
  WriteLn('Number of selected tasks: ', nTasks);
end;

begin
  Enter;
  QuickSort(1, n);
  GreedySelection;
end.
```

Dễ thấy rằng thuật toán chọn tham lam ở thủ tục $GreedySelection$ được thực hiện trong thời gian $\Theta(n)$. Thời gian mất chủ yếu nằm ở thuật toán sắp xếp các công việc theo thời điểm kết thúc. Như ở ví dụ này chúng ta dùng QuickSort: trung bình $\Theta(n \lg n)$.

16.3.2. Phủ

Trên trục số cho n đoạn: $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$. Hãy chọn một số ít nhất trong số n đoạn đã cho để phủ hết đoạn $[p, q]$.

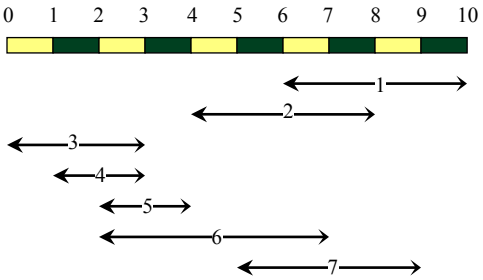
Input

- Dòng 1 chứa ba số nguyên n, p, q $(1 \le n \le 10^5, 0 \le p \le q \le 10^9)$
- n dòng tiếp theo, dòng thứ i chứa hai số nguyên a_i, b_i $(0 \le a_i \le b_i \le 10^9)$

Output

Cách chọn ra ít nhất các đoạn để phủ đoạn $[p, q]$

Sample Input	Sample Output
7 1 9 6 10 4 8 0 3 1 3 2 4 2 7 5 9	Selected Intervals: Interval 3: [0, 3] Interval 6: [2, 7] Interval 1: [6, 10]



❑ Chia để trị

Trước hết ta sẽ tìm thuật toán trong trường hợp dữ liệu vào đảm bảo tồn tại phương án phủ đoạn $[p, q]$. Sau đó, việc kiểm tra sự tồn tại của lời giải sẽ được tích hợp vào khi cài đặt thuật toán.

Phương án tối ưu để phủ hết đoạn $[p, q]$ chắc chắn phải chọn một đoạn $[a_i, b_i]$ nào đó để phủ điểm p. Giả sử đoạn $[a_i, b_i]$ chứa điểm p được chọn vào phương án tối ưu, khi đó nếu đoạn này phủ tới cả điểm q, ta có lời giải, còn nếu không, bài toán trở thành phủ đoạn $[b_i, q]$ bằng ít đoạn nhất trong số các đoạn còn lại.

Ta phân rã một bài toán thành nhiều bài toán con tương ứng với mỗi cách chọn đoạn $[a_i, b_i]$ phủ điểm p, giải các bài toán con này và chọn phương án tốt nhất trong tất cả các phương án.

❑ Phép chọn tham lam

Để thuận tiện trong việc trình bày thuật toán, với một đoạn $[a_i, b_i]$, ta gọi a_i là cận dưới (lower bound) và b_i là cận trên (upper bound) của đoạn đó.

Thuật toán chia để trị dựa vào sự lựa chọn đoạn phủ điểm p, ta có một nhận xét làm cơ sở cho phép chọn tham lam:

Định lý 16-2

Trong phương án tối ưu sẽ chỉ có một đoạn phủ điểm p. Hơn nữa, nếu có nhiều phương án cùng tối ưu, đoạn có cận trên lớn nhất phủ điểm p chắc chắn được chọn vào một phương án tối ưu nào đó.

Chứng minh

Thật vậy, nếu có phương án chứa hai đoạn phủ điểm p thì nếu ta bỏ đi đoạn có cận trên nhỏ hơn, đoạn $[p, q]$ vẫn được phủ, nên phương án này không tối ưu.

Mặt khác, gọi $[a_{i_1}, b_{i_1}]$ là đoạn có cận trên b_{i_1} lớn nhất phủ điểm p. Với một phương án tối ưu bất kỳ, giả sử rằng phương án đó chọn đoạn $[a_{j_1}, b_{j_1}]$ để phủ điểm p, theo giả thiết, cả hai đoạn $[a_{i_1}, b_{i_1}]$ và $[a_{j_1}, b_{j_1}]$ đều phủ điểm p, nhưng đoạn $[a_{i_1}, b_{i_1}]$ sẽ phủ về bên phải điểm p nhiều hơn đoạn $[a_{j_1}, b_{j_1}]$. Điều này chỉ ra rằng nếu thay thế đoạn $[a_{j_1}, b_{j_1}]$ bởi đoạn $[a_{i_1}, b_{i_1}]$ ta vẫn phủ được cả đoạn $[p, q]$ và không làm ảnh hưởng tới số đoạn được chọn trong phương án tối ưu. Suy ra điều phải chứng minh.

Vậy thì thay vì thử tất cả các khả năng chọn đoạn phủ điểm p, ta có thể đưa ra quyết định tức thì: Chọn ngay đoạn $[a_{i_1}, b_{i_1}]$ có cận trên lớn nhất phủ được điểm p vào phương án tối ưu.

❑ Cài đặt giải thuật tham lam

Giải thuật cho bài toán này có thể tóm tắt như sau:

Chọn đoạn $[a_{i_1}, b_{i_1}]$ có b_{i_1} lớn nhất thỏa mãn $p \in [a_{i_1}, b_{i_1}]$. Nếu đoạn này phủ hết đoạn $[p, q]$: $q \leq b_{i_1}$ thì xong. Nếu không, ta lại chọn tiếp đoạn $[a_{i_2}, b_{i_2}]$ có b_{i_2} lớn nhất và phủ được điểm $p_{\text{mới}} = b_{i_1}$, và cứ tiếp tục như vậy, ta có phương án tối ưu:

$$[a_{i_1}, b_{i_1}]; [a_{i_2}, b_{i_2}]; [a_{i_3}, b_{i_3}]; \dots; [a_{i_k}, b_{i_k}]$$

Nhận xét:

- Các đoạn được chọn vào phương án tối ưu có cận trên tăng dần: $b_{i_1} < b_{i_2} < \cdots < b_{i_k}$. Điều này có thể dễ hình dung được qua cách chọn: Mỗi khi chọn một đoạn mới, đoạn này sẽ phải phủ qua cận trên của đoạn trước đó.
- Các đoạn được chọn vào phương án tối ưu có cận dưới tăng dần: $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$. Thật vậy, nếu cận dưới của các đoạn được chọn không tăng dần thì trong số các đoạn được chọn sẽ có hai đoạn chứa nhau, suy ra phương án này không phải phương án tối ưu.

Đến đây ta có thể thiết kế một giải thuật lặp:

Sắp xếp danh sách các đoạn đã cho theo thứ tự không giảm của cận dưới. Đặt chỉ số đầu danh sách $i := 1$

- Bước 1: Xét phần danh sách bắt đầu từ vị trí i gồm các đoạn có cận dưới $\leq p$. Tìm trong phần này để chọn ra đoạn có cận trên $RightMost$ lớn nhất vào phương án tối ưu.
- Bước 2: Nếu $RightMost \geq q$, thuật toán kết thúc. Nếu không, đặt $p := RightMost$, cập nhật i thành chỉ số đứng sau đoạn vừa xét và lặp lại từ bước 1.

(Sau bước 2, chúng ta có thể thoải mái bỏ qua phần đầu danh sách gồm những đoạn đứng trước vị trí i, bởi tất cả những đoạn nằm trong phần này đều có cận trên $\leq RightMost$ mà như đã nhận xét, đoạn tiếp theo cần chọn chắc chắn phải có cận trên $> RightMost$)

Phủ

```
program IntervalCover;
const
  maxN = 100000;
var
  a, b: array[1..maxN] of LongInt;
  id: array[1..maxN] of LongInt;
  p, q: LongInt;
  n: LongInt;
  NoSolution: Boolean;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n, p, q);
  for i := 1 to n do
    begin
      ReadLn(a[i], b[i]);
      id[i] := i;
    end;
end;

//Sắp xếp các bằng chỉ số, thứ tự không giảm theo giá trị cận dưới: a[id[L]] ≤ a[id[L + 1]] ≤...≤ a[id[H]]
procedure QuickSort(L, H: LongInt);
var
  i, j: LongInt;
  pivot: LongInt;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (a[id[j]] > a[pivot]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (a[id[i]] < a[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;
```

```
//Phép chọn tham lam
procedure GreedySelection;
var
  i, j: LongInt;
  RightMost, k: LongInt;
begin
  i := 1;
  RightMost := -1;
  //Danh sách các đoạn theo thứ tự không giảm của cận dưới: id[1], id[2], ..., id[n]
  repeat
    //Duyệt bắt đầu từ id[i], xét các đoạn có cận dưới <=p, chọn ra đoạn id[j] có cận trên RightMost lớn nhất
    j := 0;
    while (i <= n) and (a[id[i]] <= p) do
      begin
        if RightMost < b[id[i]] then
          begin
            RightMost := b[id[i]];
            j := i;
          end;
        Inc(i);
      end;
    if j = 0 then //nếu không chọn được, bài toán vô nghiệm
      begin
        NoSolution := True;
        Exit;
      end;
    id[j] := -id[j]; //Đánh dấu, đoạn id[j] được chọn vào phương án tối ưu
    p := RightMost; //Cập nhật lại p, nếu chưa phủ đến q, tìm tiếp trong danh sách (không tìm lại từ đầu)
  until p >= q;
  NoSolution := False;
end;

procedure PrintResult; //In kết quả
var
  i, j: LongInt;
begin
  if NoSolution then
    WriteLn('No Solution!')
  else
    begin
      WriteLn('Selected Intervals: ');
      for i := 1 to n do
        begin
          j := -id[i];
          if j > 0 then //In ra các đoạn được đánh dấu
            WriteLn('Interval ', j, ': [', a[j], ', ', b[j], ']');
        end;
    end;
end;

begin
  Enter;
  QuickSort(1, n);
  GreedySelection;
  PrintResult;
end.
```

Dễ thấy rằng thời gian thực hiện giải thuật chọn tham lam trong thủ tục $GreedySelection$ là $\Theta(n)$, chi phí thời gian của thuật toán chủ yếu nằm ở giai đoạn sắp xếp. Trong chương trình này chúng ta dùng QuickSort: Trung bình $\Theta(n \lg n)$.

16.3.3. Mã hóa Huffman

Mã hóa Huffman [**11**] được sử dụng rộng rãi trong các kỹ thuật nén dữ liệu. Trong đa số các thử nghiệm, thuật toán mã Huffman có thể nén dữ liệu xuống còn từ 10% tới 80% tùy thuộc vào tình trạng dữ liệu gốc.

❑ Cơ chế nén dữ liệu

Xét bài toán lưu trữ một dãy dữ liệu gồm n ký tự. Thuật toán Huffman dựa vào tần suất xuất hiện của mỗi phần tử để xây dựng cơ chế biểu diễn mỗi ký tự bằng một dãy bit.

Giả sử dữ liệu là một xâu 100 ký tự $\in \{A, B, C, D, E, F\}$. Tần suất (số lần) xuất hiện của mỗi ký tự cho bởi:

Ký tự: c	A	B	C	D	E	F
Tần suất: $f(c)$	45	13	12	16	9	5

Một cách thông thường là biểu diễn mỗi ký tự bởi một dãy bit chiều dài cố định (như bảng mã ASCII sử dụng 8 bit cho một ký tự AnsiChar hay bảng mã Unicode sử dụng 16 bit cho một ký tự WideChar). Chúng ta có 6 ký tự nên có thể biểu diễn mỗi ký tự bằng một dãy 3 bit:

Ký tự	A	B	C	D	E	F
Mã bit	000	001	010	011	100	101

Vì mỗi ký tự chiếm 3 bit nên để biểu diễn xâu ký tự đã cho sẽ cần $3 \times 100 = 300$ bit. Cách biểu diễn này gọi là biểu diễn bằng từ mã chiều dài cố định (fixed-length codewords).

Một cách khác là biểu diễn mỗi ký tự bởi một dãy bit sao cho ký tự xuất hiện nhiều lần sẽ được biểu diễn bằng dãy bít ngắn trong khi ký tự xuất hiện ít lần hơn sẽ được biểu diễn bằng dãy bit dài:

Ký tự	A	B	C	D	E	F
Mã bit	0	101	100	111	1101	1100

Với cách làm này, xâu ký tự đã cho có thể biểu diễn bằng:

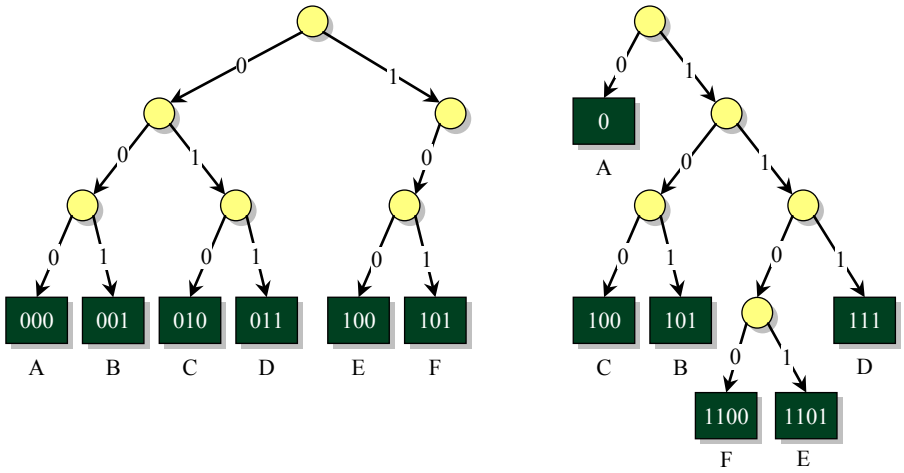
$$1 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5 = 224 \text{ bit}$$

Dữ liệu được nén xuống còn xấp xỉ 75%. Cách biểu diễn này được gọi là biểu diễn bằng từ mã chiều dài thay đổi (variable-length codewords).

❑ Mã phi tiền tố

Dãy bít biểu diễn một ký tự gọi là từ mã (codeword) của ký tự đó. Xét tập các từ mã của các ký tự, nếu không tồn tại một từ mã là tiền tố* của một từ mã khác thì tập từ mã này được gọi là *mã phi tiền tố* (*prefix-free codes* hay còn gọi là *prefix codes*). Hiển nhiên cách biểu diễn bằng từ mã chiều dài cố định là mã phi tiền tố.

Một mã phi tiền tố có thể biểu diễn bằng một cây nhị phân. Trong đó ta gọi nhánh con trái và nhánh con phải của một nút lần lượt là là nhánh 0 và nhánh 1. Các nút lá tương ứng với các ký tự và đường đi từ nút gốc tới nút lá sẽ tương ứng với một dãy nhị phân biểu diễn ký tự đó. Hình 16-1 là hai cây nhị phân tương ứng với hai mã tiền tố chiều dài cố định và chiều dài thay đổi như ở ví dụ trên.



Hình 16-1. Cây biểu diễn mã tiền tố

Một xâu ký tự S (bản gốc) sẽ được mã hóa (nén) theo cách thức: Viết các từ mã của từng chữ cái nối tiếp nhau tạo thành một dãy bit. Dãy bit này gọi là bản nén của bản gốc S. Trong ví dụ này xâu "AABE" sẽ được nén thành dãy bit "001011101" nếu sử dụng mã tiền tố chiều dài thay đổi như cây bên phải Hình 16-1.

Tính chất phi tiền tố đảm bảo rằng với một từ mã, nếu xuất phát từ gốc cây, đọc từ mã từ trái qua phải và mỗi khi đọc một bit ta rẽ sang nhánh con tương ứng thì khi đọc xong từ mã, ta sẽ đứng ở một nút lá. Chính vì vậy việc giải mã (giải nén) sẽ được thực hiện theo thuật toán: Bắt đầu từ nút gốc và đọc bản nén, đọc được bit nào rẽ sang nhánh con tương ứng. Mỗi khi đến được nút lá, ta xuất ra ký tự tương ứng và quay trở về nút gốc để đọc tiếp cho tới hết.

* Một xâu ký tự X được gọi là tiền tố của xâu ký tự Y nếu $\exists s: Y = X + s$

Nếu ký hiệu C là tập các ký tự, mỗi ký tự $c \in C$ được biểu diễn bằng từ mã gồm $d_T(c)$ bit thì số bit trong bản nén là:

$$B(T) = \sum_{c \in C} d_T(c) f(c) \qquad (16.1)$$

Trong đó $f(c)$ là số lần xuất hiện ký tự c trong xâu ký tự gốc và $d_T(c)$ cũng là độ sâu của nút lá tương ứng với ký tự c. $B(T)$ gọi là chi phí của cây T.

Kích thước bản nén phụ thuộc vào cấu trúc cây nhị phân tương ứng với mã phi tiền tố được sử dụng. Bài toán đặt ra là tìm mã phi tiền tố để nén bản gốc thành bản nén gồm ít bit nhất. Điều này tương đương với việc tìm cây T có $B(T)$ nhỏ nhất tương ứng với một bản gốc là xâu ký tự đầu vào S.

❑ Thuật toán Huffman

Giả sử các nút trên cây nhị phân chứa bên trong các thông tin:

- Ký tự tương ứng với nút đó nếu là nút lá (c)
- Tần suất của nút (f). Tần suất của một nút là số lần duyệt qua nút đó khi giải nén. Dễ thấy rằng tần suất nút lá chính là tần suất của ký tự tương ứng trong bản gốc và tần suất của một nút nhánh bằng tổng tần suất của hai nút con.
- Hai liên kết trái, phải (l và r)

Xét danh sách ban đầu gồm tất cả các nút lá của cây. Thuật toán Huffman làm việc theo cách: Lấy từ danh sách ra hai nút x, y có tần suất thấp nhất. Tạo ra một nút z làm nút cha của cả hai nút x và y, tần suất của nút z được gán bằng tổng tần suất hai nút x, y, sau đó z được đẩy vào danh sách (ra hai vào một). Thuật toán sẽ kết thúc khi danh sách chỉ còn một nút (tương ứng với nút gốc của cây được xây dựng). Có thể hình dung tại mỗi bước, thuật toán quản lý một rừng các cây và tìm cách nhập hai cây lại cho tới khi rừng chỉ còn một cây.

Danh sách các nút trong thuật toán Huffman được tổ chức dưới dạng hàng đợi ưu tiên. Trong đó nút x được gọi là ưu tiên hơn nút y nếu tần suất nút x nhỏ hơn tần suất nút y: $x.f < y.f$

Hai thao tác trên hàng đợi ưu tiên được sử dụng là:

- Hàm $Extract$: Lấy nút có tần suất nhỏ nhất ra khỏi hàng đợi ưu tiên, trả về trong kết quả hàm
- Thủ tục $Insert(z)$: Thêm một nút mới (z) vào hàng đợi ưu tiên

Khi đó có thể viết cụ thể hơn thuật toán Huffman:

```
H := Ø; //Khởi tạo hàng đợi ưu tiên H
for ∀c∈C do
  begin
    «Tạo ra một nút mới Node có tần suất f(c) và chứa ký tự c»;
    Insert(Node);
  end;
while |H| > 1 do
  begin
    x := Extract; y := Extract; //Lấy từ hàng đợi ưu tiên ra hai nút có tần suất nhỏ nhất
    «Tạo ra một nút mới z»;
    z.f := x.f + y.f; //z có tần suất bằng tổng tần suất hai nút x, y
    z.l := x; z.r := y; //cho x và y làm con trái và con phải của z
    Insert(z); //Đẩy z vào hàng đợi ưu tiên
  end;
```

Phép lựa chọn tham lam trong thuật toán Huffman thể hiện ở sự quyết định tức thì tại mỗi bước: Chọn hai cây có tần suất nhỏ nhất để nhập vào thành một cây. Tính đúng đắn của thuật toán Huffman được chứng minh như sau:

Bổ đề 16-3

Cây tương ứng với mã phi tiền tố tối ưu phải là cây nhị phân đầy đủ. Tức là mọi nút nhánh của nó phải có đúng hai nút con.

Chứng minh

Thật vậy, nếu một mã tiền tố tương ứng với cây nhị phân T không đầy đủ thì sẽ tồn tại một nút nhánh p chỉ có một nút con q. Xóa bỏ nút nhánh p và đưa nhánh con gốc q của nó vào thế chỗ, ta được một cây mới T' mà các ký tự vẫn chỉ nằm ở lá, độ sâu của mọi lá trong nhánh cây gốc q giảm đi 1 còn đ ộ sâu của các lá khác được giữ nguyên. Tức là T' sẽ biểu diễn một mã phi tiền tố nhưng với chi phí thấp hơn T. Vậy T không tối ưu.

Bổ đề 16-4

Gọi x và y là hai ký tự có tần suất nhỏ nhất. Khi đó tồn tại một cây tối ưu sao cho hai nút lá chứa x và y là con của cùng một nút. Hay nói cách khác, hai nút lá x, y là anh-em (siblings).

Chứng minh

Tần suất của nút lá bằng tần suất của ký tự chứa trong nên sẽ bất biến trên mọi cây. Để tiện trình bày ta đồng nhất nút lá với ký tự chứa trong nó.

Với T là một cây tối ưu bất kỳ, lấy một nút nhánh sâu nhất và gọi a và b là hai nút con của nó. Dễ thấy rằng a và b phải là nút lá. Không giảm tính tổng quát, giả sử rằng $f(a) \leq f(b)$ và $f(x) \leq f(y)$.

Đảo nút a và nút x cho nhau, ta được một cây mới T' mà sự khác biệt về chi phí của T và T' chỉ khác nhau ở hai nút a và x. Tức là nếu xét về mức chênh lệch chi phí:

$$\begin{aligned}(T) - B(T') &= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_{T'}(x) + f(a)d_{T'}(a)] \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) - f(a)d_T(x) \\ &= \big(f(a) - f(x)\big)\big(d_T(a) - d_T(x)\big) \\ &\geq 0\end{aligned} \tag{16.2}$$

(Bởi x là nút có tần suất thấp nhất nên $f(a) - f(x) \geq 0$, đồng thời a là nút lá sâu nhất nên $d_T(a) - d_T(x) \geq 0$)

Điều này chỉ ra rằng nếu đảo hai nút a và x, ta sẽ được cây T' ít ra là không tệ hơn cây T. Tương tự như vậy, nếu ta đảo tiếp hai nút b và y trên cây T', ta sẽ được cây T'' ít ra là không tệ hơn cây T'. Từ cây T tối ưu, suy ra T'' tối ưu, và trên T'' thì x và y là hai nút con của cùng một nút (ĐPCM).

Định lý 16-5

Gọi C là tập các ký tự trong xâu ký tự đầu vào S và x, y là hai ký tự có tần suất thấp nhất. Gọi C' là tập các ký tự có được từ C bằng cách thay hai ký tự x, y bởi một ký tự z: $C' = C - \{x, y\} \cup \{z\}$ với tần suất $f(z) := f(x) + f(y)$. Gọi T' là cây tối ưu trên C', khi đó nếu cho nút lá z trở thành nút nhánh với hai nút con x, y, ta sẽ được cây T là cây tối ưu trên C.

Chứng minh

Ta có

$$\begin{aligned}f(x)d_T(x) + f(y)d_T(y) &= \big(f(x) + f(y)\big)(d_{T'}(z) + 1) \\ &= f(z)d_{T'}(z) + \big(f(x) + f(y)\big)\end{aligned} \tag{16.3}$$

Từ đó suy ra $B(T) = B(T') + f(x) + f(y)$ hay $B(T') = B(T) - f(x) - f(y)$.

Giả sử phản chứng rằng T không tối ưu, gọi $\hat{T}$ là cây tối ưu trên tập ký tự C. Xét cây $\hat{T}$, không giảm tính tổng quát, có thể coi x, y là anh-em (Bổ đề 16-4). Đưa ký tự z vào nút cha chung của x và y với tần suất $f(z) := f(x) + f(y)$ rồi cắt bỏ hai nút lá x, y trên $\hat{T}$. Ta sẽ được cây mới $\hat{T}'$ tương ứng với một mã phi tiền tố trên C'. Trong đó:

$$\begin{aligned}B(\hat{T}') &= B(\hat{T}) - f(x) - f(y) \\ &< B(T) - f(x) - f(y) \text{ (do T không tối ưu)} \\ &= B(T')\end{aligned} \tag{16.4}$$

Vậy $B(\hat{T}') < B(T')$, mâu thuẫn với giả thiết T' là tối ưu trên tập ký tự C'. Ta có điều phải chứng minh.

Tính đúng đắn của thuật toán Huffman được suy ra trực tiếp từ Định lý 16-5: Để tìm cây tối ưu trên tập ký tự C, ta tìm hai ký tự có tần suất thấp nhất x, y và nhập chúng lại thành

một ký tự z. Sau khi thiết lập quan hệ cha-con giữa z và x, y, ta quy về bài toán con: Tìm cây tối ưu trên tập $C' := C - \{x, y\} \cup \{z\}$ có lực lượng ít hơn C một phần tử.

❑ Cài đặt

Chúng ta sẽ cài đặt chương trình tìm mã phi tiền tố tối ưu để mã hóa các ký tự trong một xâu ký tự đầu vào S.

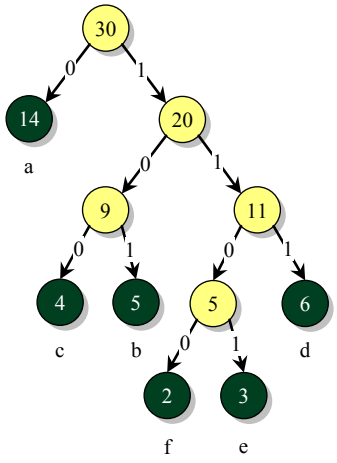
Input

Xâu ký tự S

Output

Các từ mã tương ứng với các ký tự.

Sample Input	Sample Output
abcdefabcdefabcdeabcdabdadaaaaaaaa	0 = a 100 = c 101 = b 1100 = f 1101 = e 111 = d



Hàng đợi ưu tiên của các nút trong chương trình được tổ chức dưới dạng Binary Heap (Mục 10.7.5). Trên đó ta cài đặt một thao tác $Heapify(r)$ để vun nhánh Heap gốc r thành đống trong điều kiện hai nhánh con của nó (gốc $2r$ và $2r + 1$) đã là đống rồi. Tại mỗi bước của thuật toán Huffman, thay vì cài đặt cơ chế "ra hai vào một", ta sẽ sử dụng một mẹo nhỏ: Lấy khỏi Heap (Pop) một nút x, đọc nút y ở gốc Heap (Get), tạo nút z là cha chung của x và y, đưa nút z vào gốc Heap đè lên nút y rồi thực hiện vun đống từ gốc: $Heapify(1)$.

💻 Mã hóa Huffman

```
{$MODE OBJFPC}
{$INLINE ON}
program HuffmanCode;
const
  MaxLeaves = 256; //Các ký tự là kiểu AnsiChar, cần đổi kích thước nếu dùng kiểu dữ liệu khác
```

```
type
  TNode = record //Cấu trúc nút trên cây Huffman
    f: LongInt;
    c: AnsiChar;
    l, r: Pointer;
  end;
  PNode = ^TNode; //Kiểu con trỏ tới nút
  THeap = record //Cấu trúc dữ liệu Binary Heap
    items: array[1..MaxLeaves] of PNode;
    nItems: LongInt;
  end;
var
  s: AnsiString; //Xâu ký tự đầu vào
  Heap: THeap; //Hàng đợi ưu tiên
  TreeRoot: PNode; //Gốc cây Huffman
  bits: array[1..MaxLeaves] of LongInt;

//Tạo ra một nút mới chứa tần suất fq, ký tự ch, và hai nhánh con left, right
function NewNode(fq: LongInt; ch: AnsiChar; left, right: PNode): PNode;
begin
  New(Result);
  with Result^ do
    begin
      f:= fq;
      c := ch;
      l := left;
      r := right;
    end;
end;

//Định nghĩa quan hệ ưu tiên hơn là quan hệ <: nút x ưu tiên hơn nút y nếu tần suất của x nhỏ hơn
operator < (const x, y: TNode): Boolean; inline;
begin
  Result := x.f < y.f;
end;

//Vun nhánh Heap gốc r thành đống
procedure Heapify(r: LongInt);
var
  c: LongInt;
  temp: PNode;
begin
   with Heap do
     begin
       temp := items[r];
       repeat
         c := r * 2;
         if (c < nItems) and (items[c + 1]^ < items[c]^) then
           Inc(c);
         if (c > nItems) or not (items[c]^ < temp^) then Break;
         items[r] := items[c];
         r := c;
       until False;
       items[r] := temp;
     end;
end;
```

```
//Đọc nút ở gốc Heap
function Get: PNode; inline;
begin
  with Heap do Result := items[1];
end;

//Lấy  nút có tần suất nhỏ nhất ra khỏi Heap
function Pop: PNode; inline;
begin
  with Heap do
    begin
      Result := items[1];
      items[1] := items[nItems];
      Dec(nItems);
      Heapify(1);
    end;
end;

//Thay nút ở gốc Heap bởi Node
procedure UpdateRootHeap(Node: PNode); inline;
begin
  with Heap do
    begin
      items[1] := Node;
      Heapify(1);
    end;
end;

procedure InitHeap; //Khởi tạo Heap ban đầu chứa các nút lá
var
  c: AnsiChar;
  i: LongInt;
  freq: array[AnsiChar] of LongInt;
begin
  FillChar(freq, SizeOf(freq), 0);
  for i := 1 to Length(s) do Inc(freq[s[i]]); //Đếm tần suất
  with Heap do
    begin
      nItems := 0;
      for c := Low(AnsiChar) to High(AnsiChar) do
        if freq[c] > 0 then //Xét các ký tự trong S
          begin
            Inc(nItems);
            items[nItems] := NewNode(freq[c], c, nil, nil); //Tạo nút chứa ký tự
          end;
      for i := nItems div 2 downto 1 do //Vun đống từ dưới lên
        Heapify(i);
    end;
end;

procedure BuildTree; //Thuật toán Huffman
var
  x, y, z: PNode;
begin
  while Heap.nItems > 1 do //Chừng nào Heap còn nhiều hơn 1 phần tử
    begin
      x := Pop; //Lấy nút tần suất nhỏ nhất khỏi Heap
```

```
      y := Get; //Đọc nút tần suất nhỏ nhất tiếp theo ở gốc Heap
      z := NewNode(x^.f + y^.f, #0, x, y); //Tạo nút z là cha của x và y
      UpdateRootHeap(z); //Đưa z vào gốc Heap và thực hiện vun đống: ↔ Ra x và y, vào z
    end;
  TreeRoot := Heap.Items[1]; //Giữ lại gốc cây Huffman
end;

//Thủ tục này in ra các từ mã của các lá trong cây Huffman gốc node, depth là độ sâu của node
procedure Traversal(node: PNode; depth: LongInt); //Duyệt cây Huffman gốc node
var
  i: LongInt;
begin
  if node^.l = nil then //Nếu node là nút lá
    begin //In ra dãy bit biểu diễn ký tự
      for i := 1 to depth do Write(bits[i]);
      WriteLn(' = ', node^.c);
      Dispose(node);
    end
  else //Nếu node là nút nhánh
    begin
       bits[depth + 1] := 0; //Các từ mã của các lá trong nhánh con trái sẽ có bit tiếp theo là 0
       Traversal(node^.l, depth + 1); //Duyệt nhánh trái
       bits[depth + 1] := 1; //Các từ mã của các lá trong nhánh con phải sẽ có bit tiếp theo là 1
       Traversal(node^.r, depth + 1); //Duyệt nhánh phải
    end;
end;

begin
  ReadLn(s);
  InitHeap;
  BuildTree;
  Traversal(TreeRoot, 0);
end.
```

Xét riêng thuật toán Huffman ở thủ tục BuildTree. Ta thấy rằng nếu n là số ký tự (số nút lá trong cây Huffman) thì vòng lặp while sẽ lặp $n-1$ lần. Các lời gọi thủ tục bên trong vòng lặp đó có thời gian thực hiện $O(\lg n)$. Vậy thuật toán Huffman có thời gian thực hiện $O(n \lg n)$.

Chúng ta đã khảo sát vài bài toán mà ở đó, thuật toán tham lam được thiết kế dựa trên một mô hình chia để trị.

Thuật toán tham lam cần đưa ra quyết định tức thì tại mỗi bước lựa chọn. Quyết định này sẽ ảnh hưởng ngay tới sự lựa chọn ở bước kế tiếp. Chính vì vậy, đôi khi phân tích kỹ hai quyết định liên tiếp có thể cho ta những tính chất của nghiệm tối ưu và từ đó có thể xây dựng một thuật toán hiệu quả.

16.3.4. Lịch xử lý lỗi

Bạn là một người quản trị một hệ thống thông tin, và trong hệ thống cùng lúc đang xảy ra n lỗi đánh số từ 1 tới n. Lỗi i sẽ gây ra thiệt hại d_i sau mỗi ngày và để khắc phục lỗi đó

cần t_i ngày. Tại một thời điểm, bạn chỉ có thể xử lý một lỗi. Hãy lên lịch trình xử lý lỗi sao cho tổng thiệt hại của n lỗi là nhỏ nhất có thể.

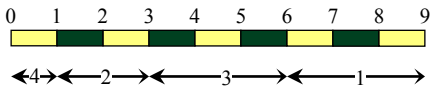
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- n dòng tiếp theo, mỗi dòng chứa hai số nguyên dương $d_i, t_i \leq 100$

Output

Lịch xử lý lỗi

Sample Input	Sample Output
4 1 3 3 2 4 3 2 1	Bug 4: fixed time = 1; damage = 2 Bug 2: fixed time = 3; damage = 9 Bug 3: fixed time = 6; damage = 24 Bug 1: fixed time = 9; damage = 9 Minimum Damage = 44



Do mỗi lỗi chỉ ngưng gây thiệt hại khi nó được khắc phục, nên dễ thấy rằng lịch trình cần tìm sẽ phải có tính chất: Khi bắt tay vào xử lý một lỗi, ta sẽ phải làm liên tục cho tới khi lỗi đó được khắc phục. Tức là ta cần tìm một hoán vị của dãy số $(1,2,\dots,n)$ tương ứng với thứ tự trong lịch trình khắc phục các lỗi.

Giả sử rằng lịch trình $A = (1,2,\dots i, i+1,\dots,n)$ xử lý lần lượt các lỗi từ 1 tới n là lịch trình tối ưu. Ta lấy một lịch trình khác $B = (1,2,\dots,i+1,i,\dots,n)$ có được bằng cách đảo thứ tự xử lý hai lỗi liên tiếp: i và $i+1$ trên lịch trình A

Ta nhận xét rằng ngoài hai lỗi i và $i+1$, thời điểm các lỗi khác được khắc phục là giống nhau trên hai lịch trình, có nghĩa là sự khác biệt về tổng thiệt hại của lịch trình A và lịch trình B chỉ nằm ở thiệt hại do hai lỗi i và $i+1$ gây ra.

Gọi T là thời điểm lỗi $i-1$ được khắc phục (trên cả hai lịch trình A và B). Khi đó:

Nếu theo lịch trình $A = (1,2,\dots i, i+1,\dots,n)$, thiệt hại của hai lỗi i và $i+1$ gây ra là:

$$\alpha = (T + t_i)d_i + (T + t_i + t_{i+1})d_{i+1} \qquad (16.5)$$

Nếu theo lịch trình $B = (1,2,\dots,i+1,i,\dots,n)$, thiệt hại của hai lỗi i và $i+1$ gây ra là:

$$\beta = (T + t_{i+1} + t_i)d_i + (T + t_{i+1})d_{i+1} \qquad (16.6)$$

Thiệt hại theo lịch trình A không thể lớn hơn thiệt hại theo lịch trình B (do A tối ưu), tức là thiệt hại α sẽ không lớn hơn thiệt hại β. Loại bỏ những hạng tử giống nhau ở công thức tính α và β, ta có:

$$\alpha \le \beta \Leftrightarrow t_i d_{i+1} \le t_{i+1} d_i$$
$$\alpha \le \beta \Leftrightarrow \frac{t_i}{d_i} \le \frac{t_{i+1}}{d_{i+1}} \quad (16.7)$$

Vậy thì nếu lịch trình $A = (1,2,\dots,n)$ là tối ưu, nó sẽ phải thỏa mãn:

$$\frac{t_1}{d_1} \le \frac{t_2}{d_2} \le \cdots \le \frac{t_n}{d_n}$$

Ngoài ra có thể thấy rằng nếu $\frac{t_i}{d_i} = \frac{t_{i+1}}{d_{i+1}}$ thì $\alpha = \beta$, hai lịch trình sửa chữa A, B có cùng mức độ thiệt hại. Trong trường hợp này, việc sửa lỗi i trước hay $i+1$ trước đều cho các phương án cùng tối ưu. Từ đó suy ra phương án tối ưu cần tìm là phương án khắc phục các lỗi theo thứ tự tăng dần của tỉ số $\frac{t_{(.)}}{d_{(.)}}$. Lời giải lặp đơn thuần chỉ là một thuật toán sắp xếp.

Lịch xử lý lỗi

```
{$MODE OBJFPC}
{$INLINE ON}
program BugFixes;
const
  max = 100000;
type
  TBug = record
    d, t: LongInt;
  end;
var
  bugs: array[1..max] of TBug;
  id: array[1..max] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do
    with bugs[i] do
      begin
        ReadLn(d, t);
        id[i] := i;
      end;
end;

//Định nghĩa lại toán tử < trên các lỗi. Lỗi x gọi là "nhỏ hơn" lỗi y nếu x.t/x.d < y.t/y.d
operator < (const x, y: TBug): Boolean; inline;
begin
  Result := x.t * y.d < y.t * x.d;
end;

//Sắp xếp các lỗi bugs[L...H] theo quan hệ thứ tự "nhỏ hơn" định nghĩa ở trên
```

```
procedure QuickSort(L, H: LongInt);
var
  i, j: LongInt;
  pivot: LongInt;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (bugs[pivot] < bugs[id[j]]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (bugs[id[i]] < bugs[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure PrintResult; //In kết quả
var
  i: LongInt;
  Time, Damage: LongInt;
  TotalDamage: Int64;
begin
  Time := 0;
  TotalDamage := 0;
  for i := 1 to n do
    with bugs[id[i]] do
      begin
        Time := Time + t;
        Damage := Time * d;
        WriteLn('Bug ', id[i], ': fixed time = ', Time, '; damage = ', Damage);
        Inc(TotalDamage, Damage);
      end;
  WriteLn('Minimum Damage = ', TotalDamage);
end;

begin
  Enter;
  QuickSort(1, n);
  PrintResult;
end.
```

16.3.5. Thuật toán Johnson

❑ Bài toán

Bài toán lập lịch gia công trên hai máy (*Two-machine flow shop model*): Có n chi tiết đánh số từ 1 tới n và hai máy A, B. Một chi tiết cần gia công trên máy A trước rồi chuyển sang gia công trên máy B. Thời gian gia công chi tiết i trên máy A và B lần lượt là a_i và b_i. Tại một thời điểm, mỗi máy chỉ có thể gia công một chi tiết. Hãy lập lịch gia công các chi tiết sao cho việc gia công toàn bộ n chi tiết được hoàn thành trong thời gian sớm nhất.

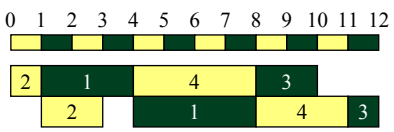
Input

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- Dòng 2 chứa n số nguyên dương $a_1, a_2, \dots, a_n$ $(\forall i: a_i \leq 100)$
- Dòng 3 chứa n số nguyên dương $b_1, b_2, \dots, b_n$ $(\forall i: b_i \leq 100)$

Output

Lịch gia công tối ưu

Sample Input	Sample Output
4 3 1 2 4 4 2 1 3	Job 2: A[0, 1]; B[1, 3] Job 1: A[1, 4]; B[4, 8] Job 4: A[4, 8]; B[8, 11] Job 3: A[8, 10]; B[11, 12] Time = 12



❑ Thuật toán

Nhận xét rằng luôn tồn tại một lịch gia công tối ưu sao mà các chi tiết gia công trên máy A theo thứ tự như thế nào thì sẽ được gia công trên máy B theo thứ tự đúng như vậy. Máy A sẽ hoạt động liên tục không nghỉ còn máy B có thể có những khoảng thời gian chết khi chờ chi tiết từ máy A. Như vậy ta cần tìm một lịch gia công tối ưu dưới dạng một hoán vị của dãy số $(1,2, \dots, n)$.

Định lý 16-6 (Định lý Johnson)

Một lịch gia công tối ưu cần thỏa mãn tính chất: Nếu chi tiết i được gia công trước chi tiết j thì $\min(a_i, b_j) \leq \min(a_j, b_i)$. Hơn nữa nếu $\min(a_i, b_j) = \min(a_j, b_i)$ thì việc đảo thứ tự gia công chi tiết i và chi tiết j trong phương án tối ưu vẫn sẽ duy trì được tính tối ưu của phương án.

Định lý 16-7

Xét quan hệ hai ngôi "nhỏ hơn hoặc bằng" (ký hiệu $\preccurlyeq$) xác định trên tập các chi tiết. Quan hệ này định nghĩa như sau: $i \preccurlyeq j$ nếu:

- Hoặc $\min(a_i, b_j) < \min(a_j, b_i)$
- Hoặc $\min(a_i, b_j) = \min(a_j, b_i)$ và $i \leq j$

Khi đó quan hệ $\preccurlyeq$ là một quan hệ thứ tự toàn phần (có đầy đủ các tính chất: phổ biến, phản xạ, phản đối xứng, và bắc cầu)

Việc chứng minh cụ thể hai định lý trên khá dài dòng, các bạn có thể tham khảo trong các tài liệu khác.

❑ Cài đặt

Định lý 16-6 (Định lý Johnson) và Định lý 16-7 chỉ ra rằng thuật toán Johnson có thể cài đặt bằng một thuật toán sắp xếp so sánh. Tuy nhiên khi cài đặt thuật toán sắp xếp so sánh, chúng ta chỉ cần cài đặt phép toán $i \prec j$: cho biết chi tiết i bắt buộc phải gia công trước chi tiết j bằng việt kiểm tra bất đẳng thức $\min(a_i, b_j) < \min(a_j, b_i)$. Bởi khi $\min(a_i, b_j) = \min(a_j, b_i)$, gia công chi tiết i trước hay j trước đều được

💻 Thuật toán Johnson

```
{$MODE OBJFPC}
{$INLINE ON}
program JohnsonAlgorithm;
uses Math;
const
  maxN = 100000;
type
  TJob = record
    a, b: LongInt;
  end;
var
  jobs: array[1..maxN] of TJob;
  id: array[1..maxN] of LongInt;
  n: LongInt;

procedure Enter; //Nhập dữ liệu
var
  i: LongInt;
begin
  ReadLn(n);
  for i := 1 to n do Read(jobs[i].a);
  ReadLn;
  for i := 1 to n do Read(jobs[i].b);
  for i := 1 to n do id[i] := i;
end;

//Định nghĩa toán tử "phải làm trước": <. Chi tiết x phải làm trước chi tiết y nếu Min(x.a, y.b) < Min(y.a, x.b)
```

```
operator < (const x, y: TJob): Boolean; inline; //Toán tử này dùng cho sắp xếp
begin
  Result := Min(x.a, y.b) < Min(y.a, x.b);
end;

//Thuật toán QuickSort sắp xếp bằng chỉ số
procedure QuickSort(L, H: LongInt);
var
  i, j: LongInt;
  pivot: LongInt;
begin
  if L >= H then Exit;
  i := L + Random(H - L + 1);
  pivot := id[i]; id[i] := id[L];
  i := L; j := H;
  repeat
    while (jobs[pivot] < jobs[id[j]]) and (i < j) do Dec(j);
    if i < j then
      begin
        id[i] := id[j]; Inc(i);
      end
    else Break;
    while (jobs[id[i]] < jobs[pivot]) and (i < j) do Inc(i);
    if i < j then
      begin
        id[j] := id[i]; Dec(j);
      end;
    else Break;
  until i = j;
  id[i] := pivot;
  QuickSort(L, i - 1); QuickSort(i + 1, H);
end;

procedure PrintResult; //In kết quả
var
  StartA, StartB, FinishA, FinishB: LongInt;
  i, j: LongInt;
begin
  StartA := 0;
  StartB := 0;
  for i := 1 to n do //Duyệt danh sách đã sắp xếp
    begin
      j := id[i];
      FinishA := StartA + jobs[j].a;
      StartB := Max(StartB, FinishA);
      FinishB := StartB + jobs[j].b;
      WriteLn('Job ', j, ': ',
        'A[', StartA, ', ', FinishA, ']; B[', StartB, ', ', FinishB, ']');
      StartA := FinishA;
      StartB := FinishB;
    end;
  WriteLn('Time = ', FinishB);
end;

begin
  Enter;
  QuickSort(1, n);
```

```
  PrintResult;
end.
```

Thời gian thực hiện thuật toán Johnson cài đặt theo cách này có thể đánh giá bằng thời gian thực hiện giải thuật sắp xếp. Ở đây ta dùng QuickSort (Trung bình $\Theta(n \lg n)$). Có thể cài đặt thuật toán Johnson theo một cách khác để tận dụng các thuật toán sắp xếp dãy khóa số, cách làm như sau:

Chia các chi tiết làm hai nhóm: Nhóm SA gồm các chi tiết i có $a_i < b_i$ và nhóm SB gồm các chi tiết j có $a_j \geq b_j$. Sắp xếp các chi tiết trong nhóm SA theo thứ tự tăng dần của các a_i, sắp xếp các chi tiết trong nhóm SB theo thứ tự giảm dần của các b_i rồi nối hai danh sách đã sắp xếp lại.

Bài tập 16-1.

Bạn là người lập lịch giảng dạy cho n chuyên đề, chuyên đề thứ i cần bắt đầu ngay sau thời điểm s_i và kết thúc tại thời điểm f_i: $(s_i, f_i]$. Mỗi chuyên đề trong thời gian diễn gia hoạt động giảng dạy sẽ cần một phòng học riêng. Hãy xếp lịch giảng dạy sao cho số phòng học phải sử dụng là ít nhất. Tìm thuật toán $O(n \lg n)$.

Bài tập 16-2.

Trên trục số cho n điểm đen và n điểm trắng hoàn toàn phân biệt. Hãy tìm n đoạn, mỗi đoạn nối một điểm đen với một điểm trắng, sao cho không có hai đoạn thẳng nào có chung đầu mút và tổng độ dài n đoạn là nhỏ nhất có thể. Tìm thuật toán $O(n \lg n)$.

Bài tập 16-3.

Xét mã phi tiền tố của một tập n ký tự. Nếu các ký tự được sắp xếp sẵn theo thứ tự không giảm của tuần suất thì chúng ta có thể xây dựng cây Huffman trong thời gian $O(n)$ bằng cách sử dụng hai hàng đợi: Tạo ra các nút lá chứa ký tự và đẩy chúng vào hàng đợi 1 theo thứ tự từ nút tần suất thấp nhất tới nút tần suất cao nhất. Khởi tạo hàng đợi 2 rỗng, lặp lại các thao tác sau $n - 1$ lần:

- Lấy hai nút x, y có tần suất thấp nhất ra khỏi các hàng đợi bằng cách đánh giá các phần tử ở đầu của cả hai hàng đợi
- Tạo ra một nút z làm nút cha của hai nút x, y với tần suất bằng tổng tần suất của x và y
- Đẩy z vào cuối hàng đợi 2

Chứng minh tính đúng đắn và cài đặt thuật toán trên.

Bài tập 16-4.

Bài toán xếp ba lô (Knapsack) chúng ta đã kh ảo sát trong chuyên đề kỹ thuật nhánh cận và quy hoạch động là bài toán 0/1 Knapsack: Với một đồ vật chỉ có hai trạng thái: chọn hay không chọn. Chúng ta cũng đã khảo sát bài toán Fractional Knapsack: Cho phép chọn một phần đồ vật: Với một đồ vật trọng lượng w và giá trị v, nếu ta lấy một phần trọng lượng $w' \leq w$ của đồ vật đó thì sẽ được giá trị là $v * \frac{w'}{v}$. Chỉ ra rằng hàm $Estimate$ trong mục 13.4.2 cho kết quả tối ưu đối với bài toán Fractional Knapsack.

Bài tập 16-5.

Giáo sư X lái xe trong một chuyến hành trình "Xuyên Việt" từ Cao Bằng tới Cà Mau dài l km. Dọc trên đường đi có n trạm xăng và trạm xăng thứ i cách Cao Bằng d_i km. Bình xăng của xe khi đổ đầy có thể đi được m km. Hãy xác định các điểm dừng để đổ xăng tại các trạm xăng sao cho số lần phải dừng đổ xăng là ít nhất trong cả hành trình.

Bài tập 16-6.

Cho n điểm thực trên trục số: $x_1, x_2, \ldots, x_n \in \mathbb{R}$. Hãy chỉ ra ít nhất các đoạn dạng $[i, i+1]$ với i là số nguyên để phủ hết n điểm đã cho.

Bài tập 16-7.

Bạn có n nhiệm vụ, mỗi nhiệm vụ cần làm trong một đơn vị thời gian. Nhiệm vụ thứ i có thời điểm phải hoàn thành (deadline) là d_i và nếu bạn hoàn thành nhiệm vụ đó sau thời hạn d_i thì sẽ phải mất một khoản phạt p_i. Bạn bắt đầu từ thời điểm 0 và tại mỗi thời điểm chỉ có thể thực hiện một nhiệm vụ. Hãy lên lịch thực hiện các nhiệm vụ sao cho tổng số tiền bị phạt là ít nhất.

Bài tập 16-8. ★

Một lần Tôn Tẫn đua ngựa với vua Tề. Tôn Tẫn và vua Tề mỗi người có n con ngựa đánh số từ 1 tới n, con ngựa thứ i của Tôn Tẫn có tốc độ là a_i, con ngựa thứ i của vua Tề có tốc độ là b_i. Luật chơi như sau:

- Có tất cả n cặp đua, mỗi cặp đua có một ngựa của Tôn Tẫn và một ngựa của vua Tề.
- Con ngựa nào cũng phải tham gia đúng một cặp đua
- Trong một cặp đua, con ngựa nào tốc độ cao hơn sẽ thắng, nếu hai con ngựa có cùng tốc độ thì kết quả của cặp đua đó sẽ hoà.
- Trong một cặp đua, con ngựa của bên nào thắng thì bên đó s ẽ được 1 điểm, hoà và thua không có điểm.

Hãy giúp Tôn Tẫn chọn ngựa ra đấu n cặp đua với vua Tề sao cho hiệu số: Điểm của Tôn Tẫn - Điểm của vua Tề là lớn nhất có thể.

Tài liệu tham khảo

1 Adelson-Velsky, Georgy Maximovich and Landis, Yevgeniy Mikhailovich. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences* (1962), 263-266.

2 Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms*. Addison Wesley, 1983.

3 Bayer, Rudolf. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1 (1972), 290–306.

4 Coppersmith, Don and Winograd, Shmuel. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing* (New York, United States 1987), ACM, 1-6.

5 Cormen, Thomas H., E., Leiserson Charles, L., Rivest Ronald, and Clifford, Stein. *Introduction to Algorithms*. MIT Press, 2001.

6 Ding, Yuzheng and Weiss, Mark A. Best case lower bounds for heapsort. *Computing*, 49, 1 (1993), 1-9.

7 Floyd, Robert W. Algorithm 245 - Treesort 3. *Communications of the ACM*, 7, 12 (1964), 701.

8 Ford, Lester R. Jr. and Johnson, Selmer M. A Tournament Problem. *The American Mathematical Monthly*, 66, 5 (1959), 387-389.

9 Hoare, Charles Antony Richard. QuickSort. *Computer Journal*, 5, 1 (1962), 10-15.

10 Hopcroft, John Edward and Tarjan, Robert Endre. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16, 6 (1973), 372-378.

11 Huffman, David Albert. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40, 9 (1952), 1098-1101.

12 Karatsuba, Anatolii Alexeevich and Ofman, Yu. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145 (1962), 293-294. Translation in Physics-Doklady 7, 595-596, 1963.

13 Lacey, Stephen and Box, Richard. A fast, easy sort. *BYTE*, 16, 4 (1991), 315-ff.

14 Musser, David R. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience*, 27, 8 (1997), 983 - 993.

15 Seidel, Raimund G. and Aragon, Cecilia R. Randomized search trees. *Algorithmica*, 16 (1996), 464-497.

16 Shell, Donald L. A high-speed sorting procedure. *Communications of the ACM*, 2, 7 (1959), 30-32.

17 Sleator, Daniel Dominic and Tarjan, Robert Endre. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32, 3 (1985), 652-686.

18 Strassen, Volker. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13, 4 (1969), 354-356.

19 Tarjan, Robert Endre. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1, 2 (1972), 146-160.

20 Williams, J.W.J. Algorithm 232: Heapsort. *Communications of the ACM*, 7, 6 (1964), 347-348.